SplitMixer: Fat Trimmed From MLP-like Models

Anonymous authors
Paper under double-blind review

Abstract

We present SplitMixer, a simple and lightweight isotropic MLP-like architecture, for visual recognition. It contains two types of interleaving convolutional operations to mix information across spatial locations (spatial mixing) and channels (channel mixing). The first one includes sequentially applying two depthwise 1D kernels, instead of a 2D kernel, to mix spatial information. The second one is splitting the channels into overlapping or non-overlapping segments, with or without shared parameters, and applying our proposed channel mixing approaches or 3D convolution to mix channel information. Depending on design choices, a number of SplitMixer variants can be constructed to balance accuracy, the number of parameters, and speed. We show, both theoretically and experimentally, that SplitMixer performs on par with the state-of-the-art MLP-like models while having a significantly lower number of parameters and FLOPS. For example, without strong data augmentation and optimization, SplitMixer achieves around 94% accuracy on CIFAR-10 with only 0.28M parameters, while ConvMixer achieves the same accuracy with about 0.6M parameters. The well-known MLP-Mixer achieves 85.45% with 17.1M parameters. On CIFAR-100 dataset, SplitMixer achieves around 73% accuracy, on par with ConvMixer, but with $\sim$52% fewer parameters and FLOPS. Our model also fares well over Flowers102, Food101, and ImageNet-1K datasets. We hope that our results spark further research towards finding more efficient vision architectures and facilitate the development of MLP-like models.

1 Introduction

Architectures based exclusively on multi-layer perceptrons (MLPs) (Tolstikhin et al., 2021) have emerged as strong competitors to Vision Transformers (ViT) (Dosovitskiy et al., 2020) and Convolutional Neural Networks (CNNs) (Krizhevsky et al., 2012; He et al., 2016). They achieve compelling performance on several computer vision problems, in particular large-scale object classification. Further, they are very simple and efficient, and perform on par with more complicated architectures. MLP-like models contain two types of layers to mix information across spatial locations (spatial mixing) and channels (channel mixing). These operations can be implemented via self-attention as in ViT, MLPs as in MLP-Mixer, or convolutions as in ConvMixer (Trockman & Kolter, 2021). There is in fact a high degree of similarity among these models (Appendix 6).

We propose the SplitMixer, a conceptually and technically simple, yet very efficient architecture in terms of accuracy, the number of required parameters, and computation. Our model is similar in spirit to the ConvMixer and MLP-Mixer models in that it accepts image patches as input, dissociates spatial mixing from channel mixing, and maintains equal size and resolution throughout the network, hence an isotropic architecture. Similar to ConvMixer, it uses standard convolutions to achieve the mixing steps. Unlike ConvMixer, however, it uses **1D** convolutions to mix spatial information. This modification maintains the accuracy but does not lower the number of parameters significantly. The biggest reduction in the number of parameters is achieved by how we modify channel mixing. Instead of applying 1×1 convolutions across all channels, we apply them to channel segments that may or may not overlap each other. We implement this part with our ad-hoc solutions or with 3D convolution. This way, we find some architectures that are very frugal in terms of model size and computational needs, and at the same time exhibit high accuracy (See Appendix H).

Despite its simplicity, SplitMixer achieves excellent performance. For example, without strong data augmentations, it attains around 94% Top-1 accuracy on CIFAR10 with only 0.27M parameters and 71M FLOPS. ConvMixer achieves the same accuracy but with 0.59M parameters and 152M FLOPS (almost twice more expensive). MLP-Mixer can only achieve 85.45% with 17.1M parameters and

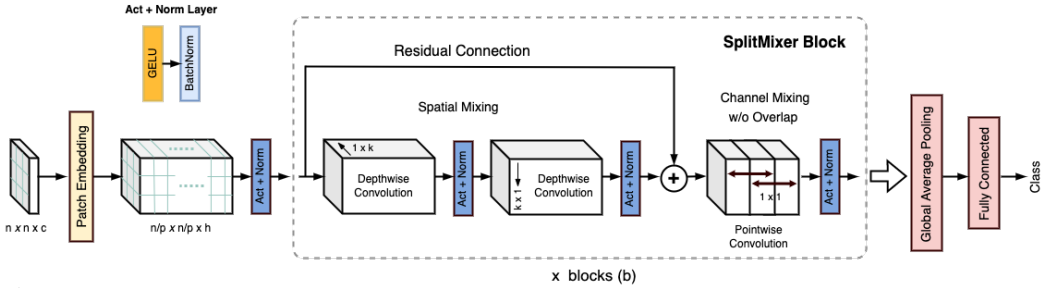

Figure 1: Basic architecture of SplitMixer. The input image is evenly divided into several image patches which are tokenized with linear projections. A number of 1D depthwise convolutions (spatial mixing) and pointwise convolutions (channel mixing) are repeatedly applied to the projections. For channel mixing, we split the channels into segments (hence the name SplitMixer) and perform convolution on them. We implement this part with our ad-hoc solutions or 3D convolution. Finally, a global average pooling layer followed by a fully-connected layer is used for class prediction.

1.21G FLOPS. ResNet50 (He et al., 2016) achieves 80.76% using 23.84M parameters, and MobileNet attains 89.81% accuracy using 0.24M parameters.

In summary, our main contributions are as follows: a) Applying 1D depthwise convolution sequentially across width and height for spatial mixing. We are inspired by the extensive use of spatial separable convolutions and depthwise separable convolutions in the literature (*e.g.* MobileNet), b) Splitting the channels into overlapping or non-overlapping segments and applying 1×1 pointwise convolution to segments for channel mixing, c) Theoretical analyses and empirical support for computational efficiency of the proposed solution, as well as measuring model throughput, and d) Ablation analyses to determine the contribution of different model components.

2 SplitMixer

The overall architecture of SplitMixer is depicted in Figure 1. It consists of a patch embedding layer followed by repeated applications of fully convolutional SplitMixer blocks. Patch embeddings with patch size p and embedding dimension h are implemented as 2D convolution with c input channels (3 for RGB images), h output channels, kernel size p, and stride p:

$$z_0 = \mathsf{N}\left(\sigma\{\texttt{Conv}_{c\to h}(I, \texttt{stride=}p, \texttt{kernel\_size=}p)\}\right) \tag{1}$$

where N is a normalization technique (*e.g.* BatchNorm by Ioffe & Szegedy (2015)), σ is an element-wise nonlinearity (*e.g.* GELU by Hendrycks & Gimpel (2016)), and $\mathbf{I} \in \mathbb{R}^{n\times n\times c}$ is the input image. The SplitMixer block itself consists of two 1D depthwise convolutions (*i.e.* grouped convolution with groups equal to the number of channels h) followed by several pointwise convolutions with kernel size 1×1. Each convolution is followed by nonlinearity and normalization [1]. Therefore, each block can be written as:

$$z'_l = \mathsf{N}\left(\sigma\{\texttt{ConvDepthwise}(z_{l-1})\}\right) \quad // \;\; 1 \times \text{k Conv across width} \tag{2}$$

$$z'_l = \mathsf{N}\left(\sigma\{\texttt{ConvDepthwise}(z'_l)\}\right) + z_{l-1} \quad // \;\; \text{k} \times 1 \text{ Conv across height} \tag{3}$$

$$z_l = \mathsf{N}\left(\sigma\{\texttt{ConvPointwise}(z'_l)\}\right) \quad // \;\; 1 \times 1 \text{ Conv across channels} \tag{4}$$

The SplitMixer block is applied b times (indexed by l), after which global pooling is applied to obtain a feature vector of size h. Finally, a softmax classifier maps this vector to the class label. Please see Appendix B for PyTorch implementations. In what follows, we describe the spatial and channel mixing layers of the architecture.

2.1 Spatial mixing

We replace the $k \times k$ kernels[2] in ConvMixer by two 1D kernels: 1) a $1 \times k$ kernel across width, and 2) a $k \times 1$ kernel across height. This reduces $k^2 \times h$ parameters to $2k \times h$ in each SplitMixer block. Similarly the $W \times H \times k^2 \times h$ FLOPS is reduced to $W \times H \times 2k \times h$, where W and H are width and height of the input tensor $\mathbf{X} \in \mathbb{R}^{W\times H\times h}$, respectively. Therefore, separating the 2D kernel into two 1D kernels results in $\frac{k}{2}$ times savings in parameters and FLOPS. The two 1D convolutions are applied sequentially and each one is followed by a GELU activation and BatchNorm (denoted as "Act + Norm" in Figure 1).

[1] We use GELU and BatchNorm throughout the paper, except in ablation experiments.

[2] Throughout the paper, a tensor or a kernel is represented as `width` $\times$ `height` $\times$ `channels`.

2.2 CHANNEL MIXING

We notice that most of parameters in ConvMixer reside in the channel mixing layer. For h channels and kernel size k ($h >> k$), in each block there are $h \times k^2$ parameters in the spatial mixing part and h^2 parameters in the channel mixing part. Thus, the fraction of parameters in the two parts is $\frac{h \times k^2}{h^2} = \frac{k^2}{h}$ which is much smaller than 1 (*e.g.* $5^2/256$). Therefore, most of the parameters are used for channel mixing.

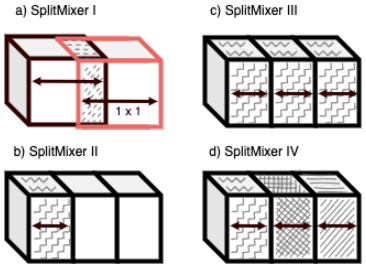

Figure 2: Channel mixing approaches: a) channels are split into two overlapping segments, and only one segment is convolved in each block (no parameter sharing across segments), b) channels are equally split into a number of segments, and only one segment is convolved in each block (no overlap or parameter sharing), c) all segments are convolved in each block and parameters are shared across segments, and d) all segments are convolved in each block (no parameter sharing).

Implementation using 3D convolution. The basic idea here is to utilize 3D convolutions with certain strides. The output will be a set of interleaved maps coming from different segments. The same 3D kernel (with shared parameters) is applied to all segments. A certain number of 3D kernels will be needed to obtain an output tensor with the same number of channels as the input. Applying m 3D kernels of size $1 \times 1 \times \frac{h}{m}$ and stride $\frac{h}{m}$ (assume h is divisible by m), will require $\frac{h^2}{m}$ parameters. Hence, more parameters and computation will be saved by increasing the number of segments (*i.e.* smaller 3D kernels). While being easy to implement, using 3D convolution has some restrictions. For example, kernel parameters have to be shared across segments, and all segments have to be convolved. Further, we find that channel mixing using 3D convolution is much slower than our other approaches (mentioned next). Please see Appendix C for the implementation of the channel mixing layer using 3D convolution.

Other channel mixing approaches. A number of approaches are proposed that differ depending on whether they allow overlap or parameter sharing among segments. They offer different degrees of trade-off in accuracy, number of parameters, and FLOPS. Notice that both of our spatial and channel mixing modifications can be used in tandem or separately. In other words, they are independent of each other. The channel mixing approaches are shown in Figure 2 and are explained below.

2.2.1 **SplitMixer-I:** OVERLAPPING SEGS, NO PARAM SHARING, UPDATE ONE SEG PER BLOCK

The input tensor is split into two overlapping segments along the channel dimension. The intuition here is that the overlapped channels allow efficient propagation of information from the first segment to the second. Let m be the size of each segment and a fraction of h, *i.e.* $m = \alpha \times h, \alpha > 0.5$. The two segments can be represented as $X[:m]$ and $X[h-m:]$ in PyTorch. For instance, for $\alpha = 2/3$, one-third of the middle channels are shared between the two segments. We choose to apply convolution to only one segment in each block, *e.g.* the left segment in odd blocks and the right segment in even blocks. m number of 1×1 convolutions are applied to the segment that should be updated. Therefore, the output has the same number of channels as the original segment, which is then concatenated to the other (unaltered) segment. The final output is a tensor with h channels to be processed in the next block. In the experiments, we choose $\alpha = \frac{i}{2i-1}, i \in \{2 \cdots 6\}$. The reduction in parameters per block can be approximated as[3]:

$$h^2 - (\alpha \times h)^2 = (1 - \alpha^2) \times h^2 \tag{5}$$

which means $1 - \alpha^2$ fraction of parameters are reduced (*e.g.* 56% parameter reduction for $\alpha = 2/3$). Notice that the bigger the α, the less saving in parameters. Similarly, the reduction in FLOPS can be approximated as:

$$W \times H \times h \times h - W \times H \times (\alpha \times h) \times (\alpha \times h) = (1 - \alpha^2) \times W \times H \times h^2 \tag{6}$$

These equations show that the saving in FLOPS is the same as the saving in parameters. We also tried a variation of this design which is updating both segments in the same block. This new variation has fewer parameters and FLOPS than ConvMixer. It saves less parameters compared to SplitMixers (ratio equal to $1 - 2\alpha^2$) but achieves slightly higher accuracy (*i.e.* trade-off in favor of accuracy). Results are shown in Appendix G.

[3]For simplicity, here we discard bias, BatchNorm, and optimizer parameters.

2.2.2 **SplitMixer-II:** NON-OVERLAPPING SEGS, NO PARAM SHARING, UPDATE ONE SEG PER BLOCK

We first split the h channels into s non-overlapping segments, each with size $\frac{h}{s}$, along the channel dimension[4]. In each block, only one segment is convolved and updated. Parameters are not shared across the segments. Following the above calculation, saving in parameters and FLOPS is $1 - \frac{1}{s^2}$. For example, for $s = 2$, $\sim$75% of the parameters are reduced. The same argument holds for FLOPS.

2.2.3 **SplitMixer-III:** NON-OVERLAPPING SEGS, PARAM SHARING, UPDATE ALL SEGS PER BLOCK

Here, h channels are split into s non-overlapping segments with shared parameters. Notice that under this setting, h must be divisible by s in order to get all the channels convolved. All segments are convolved and updated simultaneously in each block. Due to parameter sharing, the reduction in parameters is the same as SplitMixer-II, *i.e.* the two SplitMixers have the same number of parameters for the same number of segments. The number of FLOPS, however, is higher now since computation is done over all s segments. The number of FLOPS is the same as SplitMixer-IV, which will be calculated in the following subsection.

2.2.4 **SplitMixer-IV:** NON-OVERLAPPING SEGS, NO PARAM SHARING, UPDATE ALL SEGS PER BLOCK

This approach is similar to SplitMixer-III with the difference that here parameters are not shared across the segments. All segments are convolved and updated, and the results are concatenated. The reduction in parameters per block is:

$$h^2 - s \times (h/s)^2 = (1 - \frac{1}{s}) \times h^2 \tag{7}$$

which results in $1 - \frac{1}{s}$ parameter saving. For example, 66.6% of the parameters are reduced for $s = 3$. More savings can be achieved with more segments. The reduction in FLOPS is:

$$W \times H \times h \times h - s \times (W \times H \times \frac{h}{s} \times \frac{h}{s}) = (1 - \frac{1}{s}) \times W \times H \times h^2 \tag{8}$$

which means the same saving in FLOPS as in parameters.

Comparison of channel mixing approaches. Among the mixing approaches, the SplitMixer-II saves the most parameters and computation but achieves lower accuracy. SplitMixer-I strikes a good balance between accuracy and model size (and FLOPS) thanks to its partial channel sharing. We assumed the same number of blocks in all mixing approaches. In practice, a smaller number of blocks might be required when all segments are updated simultaneously in each block. Notice that apart from these approaches, there may be some other ways to perform channel mixing. For example, in SplitMixer-I, parameters can be shared across the overlapped segments, or multiple segments can overlap. We leave these explorations to future research (See Appendix G). We have also empirically measured the amount of potential saving in parameters and FLOPS over CIFAR-10 and ImageNet datasets, for model specifications mentioned in the next section. Results are shown in Appendix A.

Naming convention. We name SplitMixers after their hidden dimension h and number of blocks b like SplitMixer-A-h/b, where A is a specific model type (I, II, ...).

3 EXPERIMENTS AND RESULTS

We conducted several experiments to evaluate the performance of SplitMixer in terms of accuracy, the number of parameters, and FLOPS. Our goal was not to obtain the best possible accuracy. Rather, we were interested in knowing whether and how much parameters and computation can be reduced relative to ConvMixer. To this end, we used their code and parameter settings. A thorough comparison of ConvMixer with other models is made in Trockman & Kolter (2021). We implemented our model in PyTorch and used a Tesla V100 GPU with 32GB RAM to run it.

[4]Notice that when h is not divisible by the number of segments, the last segment will be longer (*e.g.* dividing $h = 256$ into 3 segments means the segments would have dimension [85, 85, 86], in order).

3.1 Experimental setup

We used RandAugment (Cubuk et al., 2020), random horizontal flip, and gradient clipping. Due to limited computational resources, *we did no perform extensive hyperparameter tuning*, so better results than those reported here may be possible. All models were trained for 100 epochs with batch size 512 over CIFAR-{10,100} and 64 over Flowers102 and Food101 datasets. Unless stated otherwise, h and b were set to 256 and 8 across all datasets.

We used AdamW (Loshchilov & Hutter, 2018) as the optimizer, with weight decay set to 0.005 (0.1 for Flowers102). The learning rate (lr) was adjusted with the OneCycleLR scheduler (max-lr was set to 0.05 for CIFAR-{10,100}, 0.03 for Flowers102, and 0.01 for Food101). We utilized the ithop library[5] for measuring the number of parameters and FLOPS.

3.2 Results on CIFAR-{10,100} datasets

Both datasets contain 50,000 training images and 10,000 test images (resolution is 32×32); each class has the same number of samples. We set $p = 2$ and $k = 5$ over both datasets.

Shown in the top panel of Figure 3, ConvMixer scores slightly above 94% on CIFAR-10[6]. SplitMixer-I has about the same accuracy as ConvMixer but with less than 0.3M parameters which are almost half of the ConvMixer parameters. The same statement holds for FLOPS as shown in Appendix E. SplitMixer with 1D spatial mixing and regular 1×1 channel mixing as in ConvMixer (denoted as "SplitMixer 1D S + ConvMixer C" in the Figure) attains about the same accuracy as ConvMixer with slightly lower parameters and FLOPS. SplitMixer-I with 2D spatial kernels and segmented channel mixing (denoted as "SplitMixer 2D S + C") performs on par with SplitMixer-I. Performance of the SplitMixer-II quickly drops with more segments (and subsequently fewer parameters). SplitMixers III and IV also perform well (above 90%). Interestingly, with only about 76K parameters, SplitMixer-III reaches about 91% accuracy. SplitMixer-V (Appendix G), performs close to ConvMixer, but it does not save many parameters or FLOPS.

Qualitatively similar results are obtained over the CIFAR-100 dataset. Here, ConvMixer scores 73.9% accuracy, above the 72.5% by SplitMixer-I, but with twice more parameters and FLOPS.

On both datasets, increasing the number of segments saves more parameters and FLOPS but at the expense of accuracy. Interestingly, SplitMixer-I with only channel mixing does very well. Our channel mixing approach is much more effective than 1D spatial mixing in terms of lowering the number of parameters and FLOPS.

Results using 3D convolution. We experimented with a model that uses 128 kernels of size $1 \times 1 \times 128$ and stride 128 along the channel dimension (*i.e.* channels are partitioned into two non-overlapping segments each of size 128) for channel mixing. This model scores 93.09% and 71.99% on CIFAR-10 and CIFAR-100, respectively. While having similar accuracy, the number of parameters (about 0.2 M), and FLOPS (about 0.08 G) as SplitMixer models, this model is much slower to train (each epoch takes twice more time). Notice that, among the channel mixing approaches, only SplitMixer-III can be considered as 3D convolution. Thus, performing channel mixing through strided 3D convolution is a subset of our proposed solutions.

3.3 Results on Flowers102 and Food101 datasets

Flowers102 contains 1020 training images (10 per class) and 6149 test images. Food101 contains 750 training images and 250 test images for each of its 101 classes. We used larger patch ($p = 7$) and kernel sizes ($k = 7$) since image size is bigger in these datasets (both resized to 224×224).

Results are shown in Figures 3. The patterns are consistent with what we observed over CIFAR datasets. SplitMixer variants, with small number of segments, perform close to the ConvMixer. Over the Flowers102 dataset, SplitMixer-I scores 62.03%, higher than the 60.47% by ConvMixer. Similarly, over Food101, SplitMixer-I scores 1% lower than ConvMixer, but with less than half of ConvMixer's parameters and FLOPS. In general, increasing the overlap between segments (by raising

[5] `https://github.com/Lyken17/pytorch-OpCounter`

[6] The original ConvMixer paper has reported 96% accuracy on CIFAR-10 with Mixup and Cutmix data augmentation with 0.7M parameters. We expect even better results for SplitMixer with stronger data augmentation.

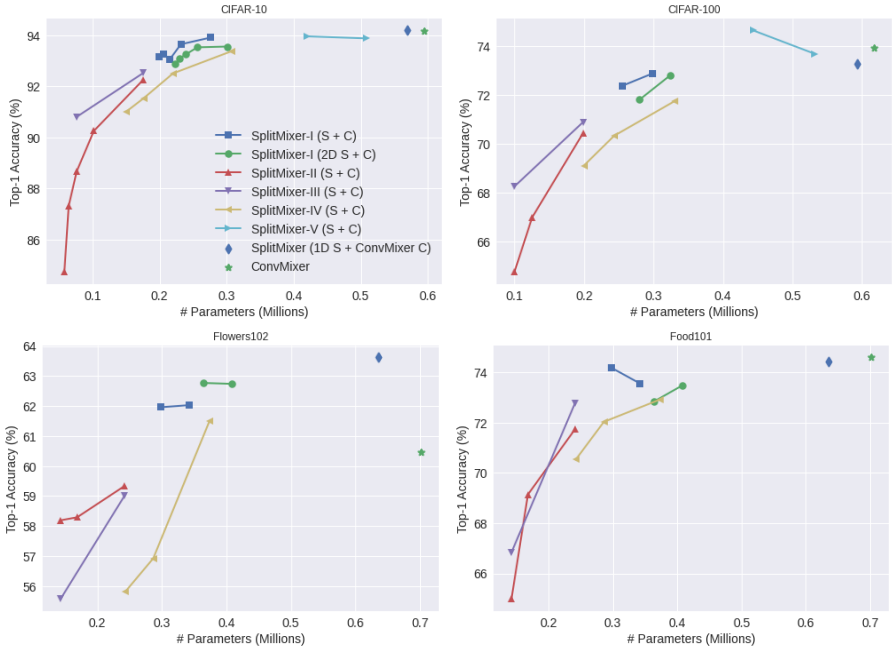

Figure 3: Performance of SplitMixer variant, accuracy *vs.* parameters. S stands for 1D spatial convolution and C stands for 1×1 pointwise convolution over channel segments. We plug in our components into ConvMixer, denoted here as "2D + C" (2D convolution kernels plus our channel mixing approach) and "1D S + ConvMixer C" (our 1D kernels plus channel mixing as is done in ConvMixer, *i.e.* 1×1 convolution across all channels without splitting). Data points are for different values of split ratio or number of segments depending on the model type. We have collected more data points on CIFAR-10 than other datasets. Please see Appendix G for a description of SplitMixer-V. See also Appendix E for accuracy vs. FLOPS plots.

α in SplitMixer-I) or reducing the number of segments enhances the accuracy, but also increases the number of parameters across datasets (not conclusive on Food101 dataset). Further, SplitMixer is effective over both small and large datasets.

Comparison with state of the art. Table 1 shows a comparison of SplitMixer with models from MLP, Transformer, and CNN families. Some results are borrowed from Lv et al. (2022) where they trained the models for 200 epochs, whereas here we trained our models for 100 epochs. While the experimental conditions in Lv et al. (2022) might not be exactly the same as ours, cross-examination still provides insights into how our models fare compared to others, in particular the MLP-based models. Our models outperform other models while having significantly smaller sizes and computational needs. For example, SplitMixer-I has about the same number of parameters as ResNet20, but is about 2% better on CIFAR-10 and 5% better on CIFAR-100. Over Flowers102, SplitMixer drastically outperforms other models in all three aspects, including accuracy, number of parameters, and FLOPS. Both SplitMixer and ConvMixer are on par with other models on the Food101 dataset, with ConvMixer performing slightly better. Please see also Appendix H.

3.4 Results on ImageNet-1K dataset

Results are shown in Table 2. We use SplitMixer-I with a 2/3 overlap ratio for the experiments. Two parameter settings are considered: a) hidden dimension equal to 1536, 20 blocks, kernel size 9, patch size 7, and GELU activation, and b) hidden dimension equal to 768, 32 blocks, kernel size 7, patch size 7, and ReLU activation. The two settings are trained for 150 and 300 epochs, respectively. The training settings (including image augmentations, optimizers, schedulers, etc.) and hyperparameters are the same as ConvMixers without any tuning. In agreement with the above-mentioned results, here SplitMixer performs close to ConvMixer (79.35% vs. 81.37%) in setting one. It, however, requires less than half of ConvMiexer parameters. A similar observation is made in the second setting. Our model performs $\sim$4% lower than ConvMixer but has much fewer parameters. Notice that here we only tested two settings, and the results seem promising compared to ConvMixer and other models.

We observe a slower convergence of our models. We suspect this might be due to using OneCycle scheduler. To test this, we continued the training in the second setting for an additional 50 epochs.

Model Family	Model	Params/FLOPS (M) / (G)	CIFAR 10	CIFAR 100	Params/FLOPS (M) / (G)	Flowers 102	Food 101
CNN	ResNet20 He et al. (2016)	**0.27** / **0.04**	91.99	67.39	**0.28** / 2.03	57.94	74.91
Transformer	ViT Dosovitskiy et al. (2020)	2.69 / 0.19	86.57	60.43	2.85 / 0.94	50.69	66.41
MLP	AS-MLP Lian et al. (2021)	26.20 / 0.33	87.30	65.16	26.30 / 1.33	48.92	74.92
”	gMLP Liu et al. (2021a)	4.61 / 0.34	86.79	61.60	6.54 / 1.93	47.35	73.56
”	ResMLP Touvron et al. (2021a)	14.30 / 0.93	86.52	61.40	14.99 / 1.23	45.00	68.40
”	ViP Hou et al. (2021)	29.30 / 1.17	88.97	70.51	30.22 / 1.76	42.16	69.91
”	MLP-Mixer Tolstikhin et al. (2021)	17.10 / 1.21	85.45	55.06	18.20 / 4.92	49.41	61.86
”	S-FC (β-LASSO) Neyshabur (2020)	- / -	85.19	59.56	- / -	-	-
”	MDMLP Lv et al. (2022)	0.30 / 0.28	90.90	64.22	0.41 / 1.59	60.39	**77.85**
”	ConvMixer	0.60 / 0.15	**94.17**	**73.92**	0.70 / 0.70	60.47	74.59
”	SplitMixer-I (ours)	0.28 / 0.07	93.91	72.44	0.34 / **0.33**	**62.03**	73.56

Table 1: Comparison with other models. The best numbers in each column are highlighted in bold. The number of parameters and FLOPS are averaged over CIFAR-10 and CIFAR-100 for our models. Notice that some variants of SplitMixer perform better than the numbers reported here over Flowers102 and Food101 datasets. Results, except ConvMixer and our model, are reproduced from Lv et al. (2022) where they have trained models for 200 epochs. We have trained ConvMixer and SplitMixer for 100 epochs.

Network	Patch Size	Kernel Size	# Params ($\times 10^6$)	# FLOPS ($\times 10^9$)	Act. Fn.	# Epochs	ImNet top-1 (%)
ConvMixer-1536/20	7	9	51.6	51.3	G	150	81.37
ConvMixer-768/32	7	7	21.1	0.33	R	300	80.16
SplitMixer-I-1536/20	7	9	23.5	22.6	G	150	79.35
SplitMixer-I-768/32	7	7	9.8	0.15	R	300 (350)	(75.05) 75.38
ResNet-152	–	3	60.2	-	R	150	79.64
DeiT-B	16	–	86	-	G	300	81.8
ResMLP-B24/8	8	–	129	-	G	400	81.0

Table 2: Results over ImageNet-1k. Models trained and evaluated on 224×224 images.

The performance improved from 75.05% to 75.38%. Thus, more training epochs, or a higher max learning rate during the initial training setup, might help in case of future usage of our models.

3.5 Ablation experiments

We conducted a series of ablation experiments to study the role of different design choices and model components. Results are shown in Table 3 over CIFAR-{10,100} datasets. We took SplitMixer-I as the baseline and discarded or added pieces to it. Our findings are summarized below:

- Completely removing the residual connections does not hurt the performance much. These connections, however, might be important for very deep SplitMixers.
- Moving the residual connection to after channel mixing seems to hurt the performance. We find that the best place for the residual connections is right after spatial mixing.
- Switching to LayerNorm, instead of BatchNorm, leads to a drastic performance drop.
- The choice of activation function, GELU *vs.* ReLU, is not very important. In fact, we found that using ReLU sometimes helps.
- Gradient norm clipping hinders the performance slightly, thus it is not very important.
- Data augmentation, here as RandAug, is critical to gaining high performance. Notice that, unlike ConvMixer, we do not have Mixup and CutMix.
- SplitMixer-I with only 1D spatial mixing, and no channel mixing, performs very poorly. The same is true for ablating the spatial mixing *i.e.* having only channel mixing. We find that spatial mixing is more important than channel mixing in our models.
- Keeping only one of the segments in channel mixing, hence 1D spatial mixing plus channel mixing using m channels ($m < h$), lowers the accuracy by a large margin. This indicates that there is a substantial benefit in having a larger h and splitting it into segments (and having overlaps between

Ablation of SplitMixer-I-256/8 on CIFAR-$\{10, 100\}$		
Ablation	CIFAR-10	CIFAR-10
SplitMixer-I (baseline)	93.91	72.88
– Residual in Eq. 3	92.24	71.34
+ Residual in Eq. 4	92.35	70.44
BatchNorm $\rightarrow$ LayerNorm	88.28	66.60
GELU $\rightarrow$ ReLU	93.39	72.56
– RandAug	90.87	66.54
– Gradient Norm Clipping	93.38	71.95
SplitMixer-I (Spatial only)	76.24	53.25
SplitMixer-I (Channel only)	64.21	40.46
One segment with size $\alpha \times h; \alpha = \frac{2}{3}$	76.28	51.28

Table 3: Ablation study of SplitMixer-I-256/8 with split ratio of 2/3 (Top-1 Acc).

them). Notice that in each block, only one segment is updated. In other words, simply lowering the number of channels does not lead to the gains that we achieve with our models. Any ConvMixer, small or large, can be optimized using our techniques.

3.6 The role of the number of blocks

We wondered about the utility of the proposed modifications over deeper networks. To this end, we varied the number of blocks b of ConvMixer and SplitMixer-I in the range 2 to 10 in steps of 2, and trained the models. Other parameters were kept the same as above. As the results in Figure 4 show, increasing the number of blocks improves the accuracy of both models on CIFAR{10,100}. SplitMixer-I performs slightly below the ConvMixer, but it has a huge advantage in terms of the number of parameters and FLOPS, in particular over deeper networks. The model size and computation grow slower for SplitMixer compared to ConvMixer.

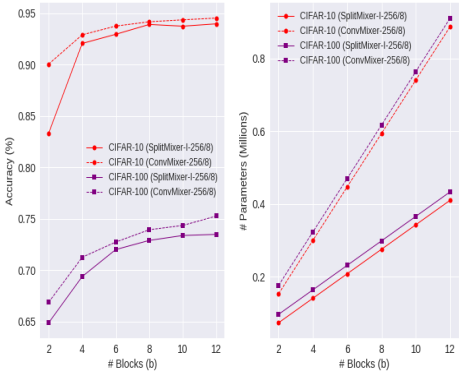

Figure 4: The role of the number of blocks b on model performance.

3.7 Model throughput

We measured throughput using batches of 64 images on a single Tesla v100 GPU with 32GB RAM (NVIDIA, 2017), averaging over 100 such batches. Similar to ConvMixer, we considered CUDA execution time rather than "wall-clock" time. Here, we used the network built for the FLOWER102 classification ($h = 256$, $d = 8$, $p = 7$, $k = 7$, and image size 224×224). We measured throughput when our model was the only process running on the GPU. Results are shown in Table 4. The throughput of our model is almost three times higher than ConvMixer. As expected, the throughput is higher with more channel segments since the number of FLOPS is lower. We also measured throughput using a Tesla v100 GPU with 16GB RAM. Results are presented in Appendix F.

Network	Throughput (img/sec)						
ConvMixer	815.84						
	Overlap ratio						
	2/3	3/5	4/7	5/9	6/11	-	-
SplitMixer-I	2097.55	2208.40	2210.06	2220.09	2231.42	-	-
	Number of segments						
	2	3	4	5	6	7	8
SplitMixer-II	2322.02	2291.44	2440.16	2464.33	2474.318	-	-
SplitMixer-III	2112.290	-	2171.70	-	-	-	2185.61
SplitMixer-IV	2110.92	2084.55	2170.57	2146.76	-	-	-

Table 4: Model throughput for SplitMixer-A-256/8 on a Tesla v100 GPU with 32GB RAM over a batch of 64 images of size 224×224, averaged over 100 such batches.

4 Related Work

For about a decade, CNNs have been the de-facto standard in computer vision (He et al., 2016). Recently, the Vision Transformers (ViT) by Dosovitskiy et al. (2020) and its variants (Touvron et al., 2021c; d'Ascoli et al., 2021; Liu et al., 2021b; Touvron et al., 2021b; Bao et al., 2021; Wang et al., 2021), and the multi-layer perceptron mixer (MLP-Mixer) by Tolstikhin et al. (2021) and its variants (Touvron et al., 2021a; Li et al., 2021) have challenged CNNs. These models have shown impressive results, even better than CNNs, in large-scale image classification. Unlike CNNs that exploit local convolutions to encode spatial information, vision transformers take advantage of the self-attention mechanism to capture global information. MLP-based models, on the other hand, capture global information through a series of spatial and channel mixing operations.

MLP-Mixer borrows some design choices from recent transformer-based architectures (Vaswani et al., 2017). Following ViT, it converts an image to a set of patches and linearly embeds them to a set of tokens. These tokens are processed by a number of "isotropic" blocks, which are in essence similar to the repeated transformer-encoder blocks (Vaswani et al., 2017). For example, MLP-Mixer replaces self-attention with MLPs applied across different dimensions (*i.e.* spatial and channel location mixing). ResMLP (Touvron et al., 2021a) is a data-efficient variation on this scheme. CycleMLP (Chen et al., 2021), gMLP (Liu et al., 2021a), and vision permutator (Hou et al., 2021), conduct different approaches to perform spatial and channel mixing. For example, the vision permutator permutes a tensor along the height, width, and channel to apply MLPs. Some works attempt to bridge convolutional networks and vision transformers and use one to improve the other (Cordonnier et al., 2019; d'Ascoli et al., 2021; Dai et al., 2021; Guo et al., 2021; Wang et al., 2021; Bello et al., 2019; Ramachandran et al., 2019; Bello, 2021).

We are primarily inspired by the ConvMixer (Trockman & Kolter (2022)). This model introduces a simpler version of MLP-Mixer but is essentially the same. It replaces the MLPs in MLP-Mixer with convolutions. In general, Convolution-based MLP models are smaller than their heavy Transformer-, CNN-, and MLP-based counterparts. Here, we show that it is possible to trim these models even more. Perhaps the biggest advantage of the MLP-based models is that they are easy to understand and implement, which in turn helps replicate results and compare models. Please see Appendix 6.

5 Discussion and Conclusion

We proposed SplitMixer, a very simple yet efficient model, that is similar in spirit to ConvMixer, ViT, and MLP-Mixer models. SplitMixer uses 1D convolutions for spatial mixing and splits the channels into several segments and performs 1×1 convolution on them for channel mixing. Our experiments, even without extensive hyperparameter tuning, demonstrate that these modifications result in models that are very efficient in terms of the number of parameters and computation. In terms of accuracy, they outperform several MLP-based models and some other model types with similar size constraints. **Our main point is that SplitMixer allows sacrificing a small amount of accuracy to achieve big gains in reducing parameters and FLOPS.**

The proposed solution based on separable filters, depthwise convolution, and channel splitting is quite efficient in terms of parameters and computation. However, if a network is already small, reducing the parameters too much may cause the network not to learn properly during training. Thus, a balance is required to enhance efficiency without significantly reducing effectiveness.

We propose the following directions for future research in this area: a) Trying a wider range of hyperparameters and design choices for SplitMixer, such as strong data augmentation (*e.g.* Mixup, Cutmix), deeper models, larger patch sizes, overlapped image patches, label smoothing (Müller et al., 2019), and stochastic depth (Huang et al., 2016). Previous research has shown that some classic models can achieve state-of-the-art performance through carefully-designed training regimes (Wightman et al., 2021), b) We tried several ways to split and mix the channels and learned that some perform better than others. There might be even better approaches to do this, c) Incorporating techniques similar to the ones proposed here to optimize other MLP-like models is also a promising direction, d) MLP-like models, including SplitMixer, lack effective means of explanation and visualization, which need to be addressed in the future (See Appendix I), and e) Our results entertain the idea that it may be possible to find model classes that have fewer parameters than the number of data points. This may challenge the current belief that deep networks must be overparameterized to perform well.

REFERENCES

Hangbo Bao, Li Dong, and Furu Wei. Beit: Bert pre-training of image transformers. *arXiv preprint arXiv:2106.08254*, 2021.

Irwan Bello. Lambdanetworks: Modeling long-range interactions without attention. *arXiv preprint arXiv:2102.08602*, 2021.

Irwan Bello, Barret Zoph, Ashish Vaswani, Jonathon Shlens, and Quoc V Le. Attention augmented convolutional networks. In *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 3286–3295, 2019.

Shoufa Chen, Enze Xie, Chongjian Ge, Ding Liang, and Ping Luo. Cyclemlp: A mlp-like architecture for dense prediction. *arXiv preprint arXiv:2107.10224*, 2021.

Jean-Baptiste Cordonnier, Andreas Loukas, and Martin Jaggi. On the relationship between self-attention and convolutional layers. *arXiv preprint arXiv:1911.03584*, 2019.

Ekin D Cubuk, Barret Zoph, Jonathon Shlens, and Quoc V Le. Randaugment: Practical automated data augmentation with a reduced search space. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pp. 702–703, 2020.

Zihang Dai, Hanxiao Liu, Quoc V Le, and Mingxing Tan. Coatnet: Marrying convolution and attention for all data sizes. *arXiv preprint arXiv:2106.04803*, 2021.

Stéphane d'Ascoli, Hugo Touvron, Matthew Leavitt, Ari Morcos, Giulio Biroli, and Levent Sagun. Convit: Improving vision transformers with soft convolutional inductive biases. *arXiv preprint arXiv:2103.10697*, 2021.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

Jianyuan Guo, Kai Han, Han Wu, Chang Xu, Yehui Tang, Chunjing Xu, and Yunhe Wang. Cmt: Convolutional neural networks meet vision transformers. *arXiv preprint arXiv:2107.06263*, 2021.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.

Qibin Hou, Zihang Jiang, Li Yuan, Ming-Ming Cheng, Shuicheng Yan, and Jiashi Feng. Vision permutator: A permutable mlp-like architecture for visual recognition, 2021.

Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Q Weinberger. Deep networks with stochastic depth. In *European conference on computer vision*, pp. 646–661. Springer, 2016.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pp. 448–456. PMLR, 2015.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

Jiachen Li, Ali Hassani, Steven Walton, and Humphrey Shi. Convmlp: Hierarchical convolutional mlps for vision. *arXiv preprint arXiv:2109.04454*, 2021.

Dongze Lian, Zehao Yu, Xing Sun, and Shenghua Gao. As-mlp: An axial shifted mlp architecture for vision. *arXiv preprint arXiv:2107.08391*, 2021.

Hanxiao Liu, Zihang Dai, David R So, and Quoc V Le. Pay attention to mlps. *arXiv preprint arXiv:2105.08050*, 2021a.

Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows, 2021b.

Ilya Loshchilov and Frank Hutter. Fixing weight decay regularization in adam. 2018.

Tian Lv, Chongyang Bai, and Chaojie Wang. Mdmlp: Image classification from scratch on small datasets with mlp. *arXiv preprint arXiv:2205.14477*, 2022.

Rafael Müller, Simon Kornblith, and Geoffrey E Hinton. When does label smoothing help? *Advances in neural information processing systems*, 32, 2019.

Behnam Neyshabur. Towards learning convolutions from scratch. *Advances in Neural Information Processing Systems*, 33:8078–8088, 2020.

Tesla NVIDIA. Nvidia tesla v100 gpu architecture, 2017.

Prajit Ramachandran, Niki Parmar, Ashish Vaswani, Irwan Bello, Anselm Levskaya, and Jonathon Shlens. Stand-alone self-attention in vision models. *arXiv preprint arXiv:1906.05909*, 2019.

Ilya Tolstikhin, Neil Houlsby, Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Thomas Unterthiner, Jessica Yung, Daniel Keysers, Jakob Uszkoreit, Mario Lucic, et al. Mlp-mixer: An all-mlp architecture for vision. *arXiv preprint arXiv:2105.01601*, 2021.

Hugo Touvron, Piotr Bojanowski, Mathilde Caron, Matthieu Cord, Alaaeldin El-Nouby, Edouard Grave, Armand Joulin, Gabriel Synnaeve, Jakob Verbeek, and Hervé Jégou. Resmlp: Feedforward networks for image classification with data-efficient training. *arXiv preprint arXiv:2105.03404*, 2021a.

Hugo Touvron, Matthieu Cord, Matthijs Douze, Francisco Massa, Alexandre Sablayrolles, and Hervé Jégou. Training data-efficient image transformers & distillation through attention. In *International Conference on Machine Learning*, pp. 10347–10357. PMLR, 2021b.

Hugo Touvron, Matthieu Cord, Alexandre Sablayrolles, Gabriel Synnaeve, and Hervé Jégou. Going deeper with image transformers. *arXiv preprint arXiv:2103.17239*, 2021c.

Asher Trockman and J Zico Kolter. Orthogonalizing convolutional layers with the cayley transform. *arXiv preprint arXiv:2104.07167*, 2021.

Asher Trockman and J Zico Kolter. Patches are all you need? *arXiv preprint arXiv:2201.09792*, 2022.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pp. 5998–6008, 2017.

Wenhai Wang, Enze Xie, Xiang Li, Deng-Ping Fan, Kaitao Song, Ding Liang, Tong Lu, Ping Luo, and Ling Shao. Pyramid vision transformer: A versatile backbone for dense prediction without convolutions. *arXiv preprint arXiv:2102.12122*, 2021.

Ross Wightman, Hugo Touvron, and Hervé Jégou. Resnet strikes back: An improved training procedure in timm, 2021.

6 A UNIFIED VIEW OF VISION TRANSFORMER AND MLP-MIXER

MLP-Mixer borrows some design ideas from Vision Transformers. The most obvious one is splitting the input image into patches and mapping each patch to an embedding vector using a linear layer. Both ViT and MLP-Mixer do not use convolutions, or at least claim not to. However, one can argue that the linear embedding is in fact convolution with stride equal to the patch size and parameter sharing across patches. Here, we cross examine both architectures and show that their similarities go beyond the embedding layer:

1. The embedding layer in the two models is the same and is implemented using an MLP with a single layer.
2. Channel mixing is done in the exact same way in both models via a two-layer MLP. Please see the figures.
3. Both models use skip connections the same way in both channel and token mixing parts.
4. Both models use LayerNorm for normalization.
5. The major difference between the models is the way they implement token mixing. Token Mixing in the ViT happens in the Multi-head self attention (MHSA) layer, whereas in the ViT it is done via a two-layer MLP. MHSA can have multiple heads. In extreme cases, it can have one head of size d (embedding dimension), or d heads of size 1. In either case (or other cases), the information after the self attention is passed through an MLP. Effectively, the MSHA layer does both token mixing and channel mixing.
6. After multiple layers of token and channel mixing the models map information to class labels. In ViT, an extra token called [cls] token (with dimension d) is mapped to the class labels using a two layer MLP. In MLP-Mixer this is done the same way using an MLP, but first information is pooled across different patches (the Average Pooling layer).

The major difference in the models is how token mixing is done. ViT uses self attention, while MLP-Mixer uses MLP to do so. There are two other differences which do not seem to be crucial:

1. The [CLS] token in ViT already contains the summary information from other patches. Pooling information across patches as it is done in MLP-Mixer (the average pooling layer) does not seem to matter much. However, it needs to be studied.
2. MLP-Mixer does not utilize positional encoding[7]. ViT authors showed that including positional information indeed improves accuracy (See Table 8 in their appendix). Positional encoding helps maintain positional information, which will otherwise be lost after several layers of token and channel mixing throughout the network. Interestingly, without explicitly accounting for spatial information, the MLP-Mixer still performs very well and on-par with ViT. It would be interesting to see if adding spatial information to the MLP-Mixer can improve its accuracy.

[7]Unlike NLP where order or the words can alter the meaning of a sentence, reordering the image patches does not seem to result in a viable scene and does not happen naturally. Thus, it might not be important in vision tasks!

A Parameter and FLOPS Saving

Here, we empirically show how much parameters and FLOPS can be reduced by SplitMixer. Experiments are conducted on CIFAR-10 and ImageNet datasets using model specifications mentioned in the main text.

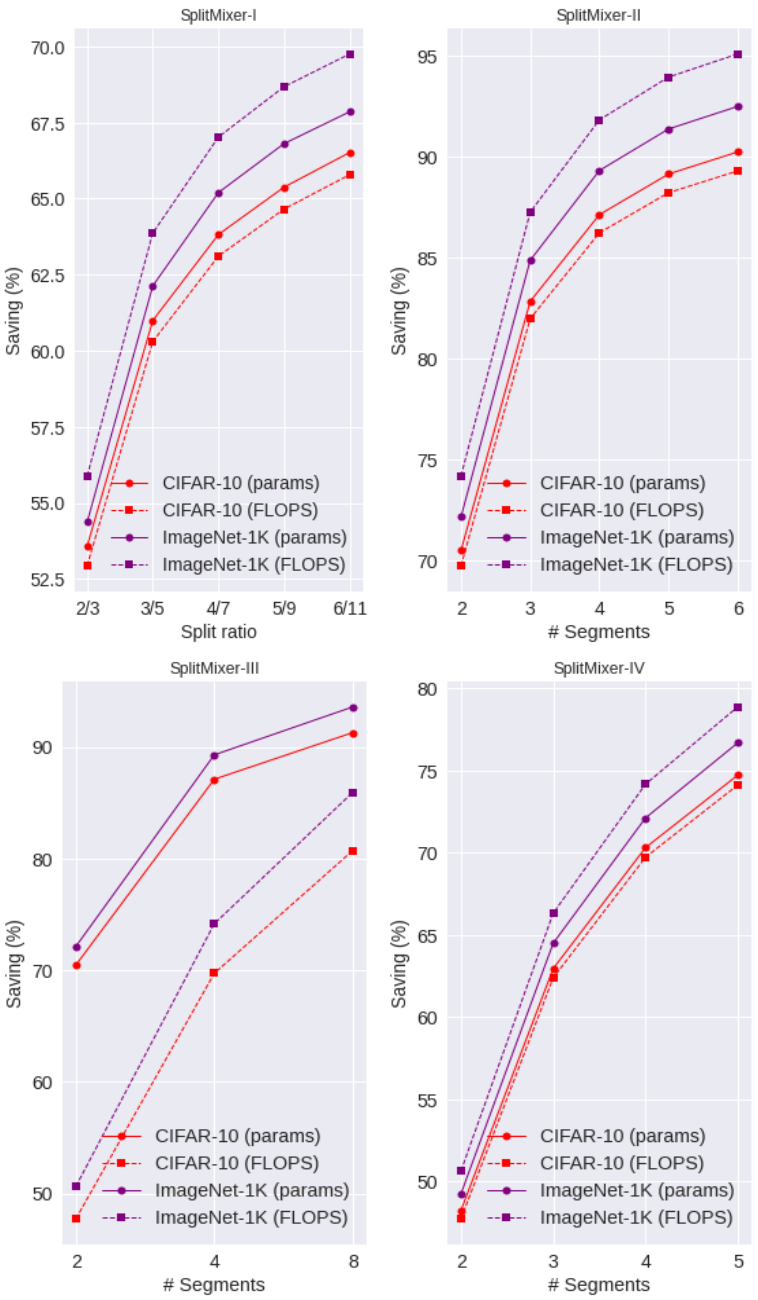

Figure 5: Potential savings in parameters and FLOPS for different SplitMixer variants.

B PyTorch Implementations

```python
class ChannelMixerI(nn.Module):
    ''' Partial overlap; In each block only one segment is convolved. '''
    def __init__(self, hdim, is_odd=0, ratio=2/3, **kwargs):
        super().__init__()
        self.hdim = hdim
        self.partial_c = int(hdim *ratio)
        self.mixer = nn.Conv2d(self.partial_c, self.partial_c, kernel_size=1)
        self.is_odd = is_odd

    def forward(self, x):
        if self.is_odd == 0:
            idx = self.partial_c
            return torch.cat((self.mixer(x[:, :idx]), x[:, idx:]), dim=1)
        else:
            idx = self.hdim - self.partial_c
            return torch.cat((x[:, :idx], self.mixer(x[:, idx:])), dim=1)


def SplitMixerI(dim, blocks, kernel_size=5, patch_size=2, n_classes=10, ratio=2/3):
    return nn.Sequential(
        nn.Conv2d(3, dim, kernel_size=patch_size, stride=patch_size),
        nn.GELU(),
        nn.BatchNorm2d(dim),
        *[nn.Sequential(
                Residual(nn.Sequential(
                    nn.Conv2d(dim, dim, (1,kernel_size), groups=dim, padding="same"),
                    nn.GELU(),
                    nn.BatchNorm2d(dim),
                    nn.Conv2d(dim, dim, (kernel_size,1), groups=dim, padding="same"),
                    nn.GELU(),
                    nn.BatchNorm2d(dim)
                )),
                ChannelMixerI(dim, i % 2,ratio),
                nn.GELU(),
                nn.BatchNorm2d(dim),
        ) for i in range(blocks)],
        nn.AdaptiveAvgPool2d((1,1)),
        nn.Flatten(),
        nn.Linear(dim, n_classes)
    )


def SplitMixerI_channel_only(dim, blocks, kernel_size=5, patch_size=2, n_classes=10, ratio=2/3):
    return nn.Sequential(
        nn.Conv2d(3, dim, kernel_size=patch_size, stride=patch_size),
        nn.GELU(),
        nn.BatchNorm2d(dim),
        *[nn.Sequential(
                Residual(nn.Sequential(
                    nn.Conv2d(dim, dim, (kernel_size,kernel_size), groups=dim, padding="same"),
                    nn.GELU(),
                    nn.BatchNorm2d(dim),
                )),
                ChannelMixerI(dim, i % 2,ratio),
                nn.GELU(),
                nn.BatchNorm2d(dim),
        ) for i in range(blocks)],
        nn.AdaptiveAvgPool2d((1,1)),
        nn.Flatten(),
        nn.Linear(dim, n_classes)
    )
```

Figure 6: SplitMixer-I.

```python
class ChannelMixerII(nn.Module):
    ''' No overlap; In each block only one segment is convolved. '''
    def __init__(self, hdim, remainder=0, num_segments=3, **kwargs):
        super().__init__()
        self.hdim = hdim
        self.remainder = remainder
        self.num_segments = num_segments
        self.bin_dim = int(hdim / num_segments)
        self.c = hdim - self.bin_dim * (num_segments - 1) if (
            remainder == num_segments - 1) else self.bin_dim
        self.mixer = nn.Conv2d(self.c, self.c, kernel_size=1)

    def forward(self, x):
        start = self.remainder * self.bin_dim
        end = self.hdim if (self.remainder == self.num_segments - 1) else (
            (self.remainder + 1) * self.bin_dim)
        return torch.cat((x[:, :start], self.mixer(x[:, start : end]),
                          x[:, end:]), dim=1)


def SplitMixerII(dim, blocks, kernel_size=5, patch_size=2, n_classes=10, n_part=2):
    return nn.Sequential(
        nn.Conv2d(3, dim, kernel_size=patch_size, stride=patch_size),
        nn.GELU(),
        nn.BatchNorm2d(dim),
        *[nn.Sequential(
                Residual(nn.Sequential(
                    nn.Conv2d(dim, dim, (1,kernel_size), groups=dim, padding="same"),
                    nn.GELU(),
                    nn.BatchNorm2d(dim),
                    nn.Conv2d(dim, dim, (kernel_size,1), groups=dim, padding="same"),
                    nn.GELU(),
                    nn.BatchNorm2d(dim)
                )),
                ChannelMixerII(dim, i % n_part, n_part),
                nn.GELU(),
                nn.BatchNorm2d(dim),
        ) for i in range(blocks)],
        nn.AdaptiveAvgPool2d((1,1)),
        nn.Flatten(),
        nn.Linear(dim, n_classes)
    )
```

Figure 7: SplitMixer-II.

```python
class ChannelMixerIII(nn.Module):
    ''' No overlap; In each block all segments are convolved;
        Parameters are shared across segments. '''
    def __init__(self, hdim, num_segments=3, **kwargs):
        super().__init__()
        assert hdim % num_segments == 0, (
            f'hdim {hdim} need to be divisible by num_segments {num_segments}')
        self.hdim = hdim
        self.num_segments = num_segments
        self.c = hdim // num_segments
        self.mixer = nn.Conv2d(self.c, self.c, kernel_size=1)

    def forward(self, x):
        c = self.c
        x = [self.mixer(x[:, c * i : c * (i + 1)]) for i in range(self.num_segments)]
        return torch.cat(x, dim=1)


def SplitMixerIII(dim, blocks, kernel_size=5, patch_size=2, n_classes=10, n_part=2):
    return nn.Sequential(
        nn.Conv2d(3, dim, kernel_size=patch_size, stride=patch_size),
        nn.GELU(),
        nn.BatchNorm2d(dim),
        *[nn.Sequential(
                Residual(nn.Sequential(
                    nn.Conv2d(dim, dim, (1,kernel_size), groups=dim, padding="same"),
                    nn.GELU(),
                    nn.BatchNorm2d(dim),
                    nn.Conv2d(dim, dim, (kernel_size,1), groups=dim, padding="same"),
                    nn.GELU(),
                    nn.BatchNorm2d(dim)
                )),
                ChannelMixerIII(dim, n_part),
                nn.GELU(),
                nn.BatchNorm2d(dim),
        ) for i in range(blocks)],
        nn.AdaptiveAvgPool2d((1,1)),
        nn.Flatten(),
        nn.Linear(dim, n_classes)
    )
```

Figure 8: SplitMixer-III.

```python
class ChannelMixerIV(nn.Module):
    ''' No overlap; In each block all segments are convolved;
      No parameter sharing across segments. '''
    def __init__(self, hdim, num_segments=3, **kwargs):
        super().__init__()
        self.hdim = hdim
        self.num_segments = num_segments
        c = hdim // num_segments
        last_c = hdim - c * (num_segments - 1)
        self.mixer = nn.ModuleList(
            [nn.Conv2d(c, c, kernel_size=1) for _ in range(num_segments - 1)
            ] + ([nn.Conv2d(last_c, last_c, kernel_size=1)]))
        self.c, self.last_c = c, last_c

    def forward(self, x):
        c, last_c = self.c, self.last_c
        x = [self.mixer[i](x[:, c * i : c * (i + 1)]) for i in (
            range(self.num_segments - 1))] + [self.mixer[-1](x[:, -last_c:])]
        return torch.cat(x, dim=1)


def SplitMixerIV(dim, blocks, kernel_size=5, patch_size=2, n_classes=10, n_part=2):
    return nn.Sequential(
        nn.Conv2d(3, dim, kernel_size=patch_size, stride=patch_size),
        nn.GELU(),
        nn.BatchNorm2d(dim),
        *[nn.Sequential(
                Residual(nn.Sequential(
                    nn.Conv2d(dim, dim, (1,kernel_size), groups=dim, padding="same"),
                    nn.GELU(),
                    nn.BatchNorm2d(dim),
                    nn.Conv2d(dim, dim, (kernel_size,1), groups=dim, padding="same"),
                    nn.GELU(),
                    nn.BatchNorm2d(dim)
                )),
                ChannelMixerIV(dim, n_part),
                nn.GELU(),
                nn.BatchNorm2d(dim),
        ) for i in range(blocks)],
        nn.AdaptiveAvgPool2d((1,1)),
        nn.Flatten(),
        nn.Linear(dim, n_classes)
    )
```

Figure 9: SplitMixer-IV.

C CHANNEL MIXING USING 3D CONVOLUTION

```python
class StrideMixer(nn.Module):
    def __init__(self, dim, blocks, kernel_size=5, patch_size=2, num_classes=10,
                 channel_kernel_size=128, channel_stride=128): # setting III
        super().__init__()
        self.patch_emb = nn.Sequential(
            nn.Conv2d(3, dim, kernel_size=patch_size, stride=patch_size),
            nn.GELU(),
            nn.BatchNorm2d(dim),)

        # calculate channel conv h_out
        c_out = (dim - channel_kernel_size) / channel_stride + 1
        assert dim % c_out == 0, 'setting is not valid, double check channel_kernel_size and channel_stride
        h_out = int(dim / c_out)

        self.spatial_mixer, self.channel_mixer = ModuleList(), ModuleList()
        for _ in range(blocks):
            spatial_mixer = Residual(nn.Sequential(
                nn.Conv2d(dim, dim, (1, kernel_size), groups=dim, padding="same"),
                nn.GELU(),
                nn.BatchNorm2d(dim),
                nn.Conv2d(dim, dim, (kernel_size, 1), groups=dim, padding="same"),
                nn.GELU(),
                nn.BatchNorm2d(dim)))
            self.spatial_mixer.append(spatial_mixer)

            channel_mixer = nn.Sequential(
                nn.Conv3d(1, h_out, (channel_kernel_size, 1, 1), stride=(channel_stride, 1, 1)),
                nn.GELU(),
                nn.Flatten(1, 2),
                nn.BatchNorm2d(dim))
            self.channel_mixer.append(channel_mixer)

        self.head = nn.Sequential(
            nn.AdaptiveAvgPool2d((1,1)),
            nn.Flatten(),
            nn.Linear(dim, num_classes)
            )

        self.blocks = blocks

    def forward(self, x):
        x = self.patch_emb(x)
        for i in range(self.blocks):
            x = self.spatial_mixer[i](x).unsqueeze(1)
            x = self.channel_mixer[i](x)
        return self.head(x)
```

Figure 10: Channel mixing using 3D convolution.

D PARAMETER SAVING AS A FUNCTION OF SEGMENT OVERLAP

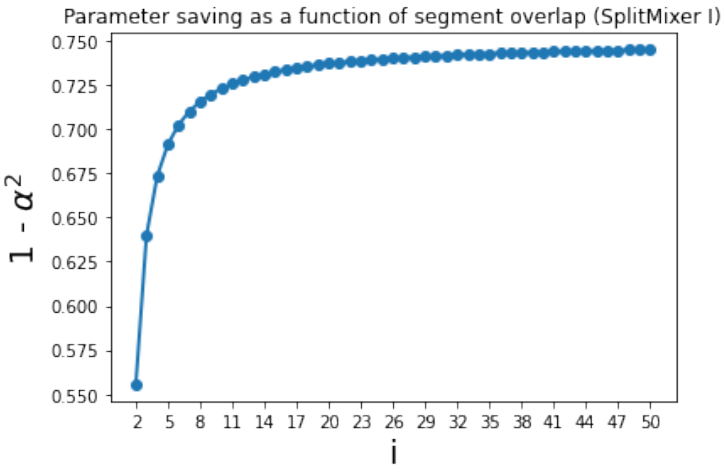

Figure 11: Parameter saving as a function of segment overlap (SplitMixer I). See Eq. 5. About 75% of parameters can be saved in the limit (*i.e.* as i approaches infinity).

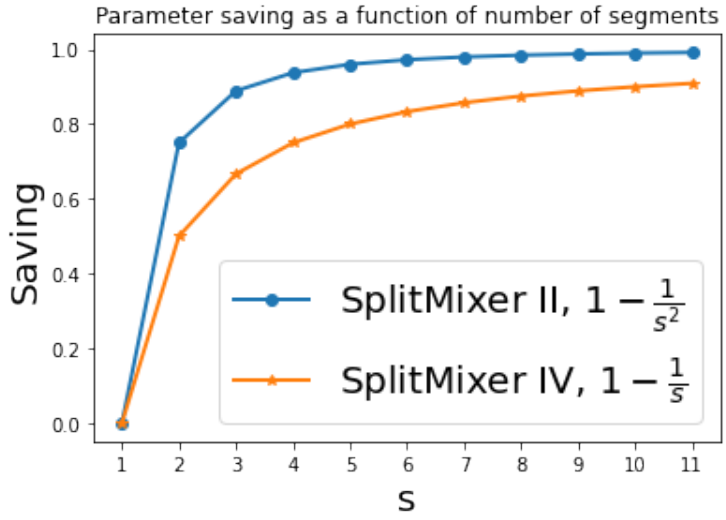

Figure 12: Parameter saving as a function of segment overlap (SplitMixer II and SplitMixer IV). Please see the main text.

E ACCURACY VS. FLOPS

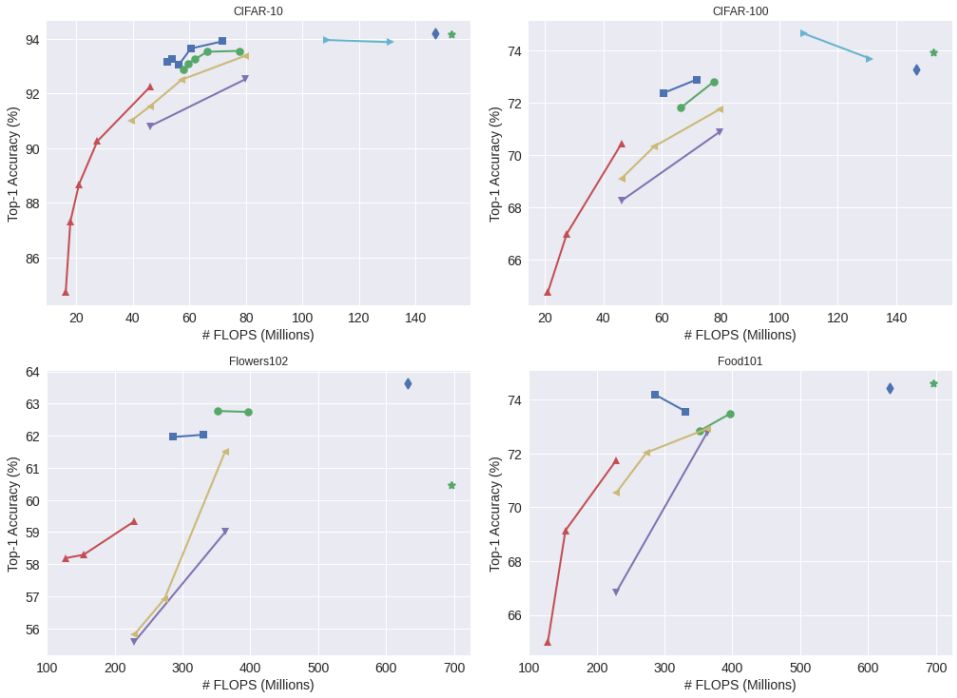

Figure 13: Accuracy *vs.* FLOPS for different variants of SplitMixer. S stands for 1D spatial convolution and C stands for 1×1 pointwise convolution over channel segments. We plug in our components into ConvMixer, denoted here as "2D + C" (2D convolution kernels plus our channel mixing approach) and "1D S + ConvMixer C" (our 1D kernels plus channel mixing as is done in ConvMixer, *i.e.* 1×1 convolution across all channels without splitting). Data points are for different values of split ratio or number of segments depending on the model type. We have collected more data points on CIFAR-10 than other datasets. Notice that the ratio of FLOPS over the number of parameters is almost the same for all models except SplitMixer-III, where this ratio is higher since all segments are updated in each block and parameters are shared across segments (see Fig. 2). That is why the plots for parameters and FLOPS are almost the same for each model, except SplitMixer-III. Please see Appendix G for a description of SplitMixer-V.

F Model Throughput

Network	Throughput (img/sec)						
ConvMixer	483.37						
	Overlap ratio						
	2/3	3/5	4/7	5/9	6/11	-	-
SplitMixer-I	733.57	742.80	746.88	734.04	734.07	-	-
	Number of segments						
	2	3	4	5	6	7	8
SplitMixer-II	770.18	790.13	815.27	821.58	825.91	-	-
SplitMixer-III	699.80	-	718.14	-	-	-	718.25
SplitMixer-IV	696.39	681.12	715.64	714.95	-	-	-

Table 5: Model throughput for SplitMixer-A-256/8 on a Tesla v100 GPU with 16GB RAM over a batch of 64 images of size 224×224, averaged over 100 such batches.

G Other Variants of SplitMixer

Here, we introduce another variant of SplitMixer, called SplitMixer-V, which is similar to SplitMixer-I with the difference that in each block both overlapped segmented are updated. The approximate reduction in parameters per block is:

$$h^2 - 2 \times (\alpha \times h)^2 = (1 - 2 \times \alpha^2) \times h^2 \tag{9}$$

which means $1 - 2\alpha^2$ fraction of parameters are reduced (*e.g.* 11% parameter reduction for $\alpha = 2/3$, much lower than 56% obtained using SplitMixer-I with the same α). The saving in FLOPS is the same.

We tested SplitMixer-V over CIFAR-{10,100} datasets with the same parameters used for other SplitMixers on these datasets. Results are shown in Table below. This model performs close to ConvMixer and is sometimes even better than it. It, however, does not save much of the parameters and FLOPS, compared to the other SplitMixers. As was mentioned in the main text, SplitMixer variants offer different degrees of trade-off between accuracy and the number of parameters (and FLOPS).

	CIFAR-10			CIFAR-100		
	Accuracy (%)	Params (M)	FLOPS (M)	Accuracy (%)	Params (M)	FLOPS (M)
ConvMixer	94.17	0.59	152	73.92	0.62	152
SplitMixer-V-256/8						
$\alpha = 2/3$	93.88	0.51	131	73.68	0.53	131
$\alpha = 3/5$	93.96	0.42	108	74.63	0.44	108

Table 6: Performance of the SplitMixer-V-256/8 model *vs.* ConvMixer.

The code for this design is given below.

```python
class ChannelMixerV(nn.Module):
    ''' Partial overlap; In each block all segments are convolved;
      No parameter sharing across segments. '''
    def __init__(self, hdim, ratio=2/3, **kwargs):
        super().__init__()
        self.hdim = hdim
        self.c = int(hdim *ratio)
        self.mixer1 = nn.Conv2d(self.c, self.c, kernel_size=1)
        self.mixer2 = nn.Conv2d(self.c, self.c, kernel_size=1)

    def forward(self, x):
        c, hdim = self.c, self.hdim
        x = torch.cat((self.mixer1(x[:, :c]), x[:, c:]), dim=1)
        return torch.cat((x[:, :(hdim - c)], self.mixer2(x[:, (hdim - c):])), dim=1)


def SplitMixer-V(dim, blocks, kernel_size=5, patch_size=2, n_classes=10, ratio=2/3):
    return nn.Sequential(
        nn.Conv2d(3, dim, kernel_size=patch_size, stride=patch_size),
        nn.GELU(),
        nn.BatchNorm2d(dim),
        *[nn.Sequential(
                Residual(nn.Sequential(
                    nn.Conv2d(dim, dim, (1,kernel_size), groups=dim, padding="same"),
                    nn.GELU(),
                    nn.BatchNorm2d(dim),
                    nn.Conv2d(dim, dim, (kernel_size,1), groups=dim, padding="same"),
                    nn.GELU(),
                    nn.BatchNorm2d(dim)
                )),
                ChannelMixerV(dim, ratio),
                nn.GELU(),
                nn.BatchNorm2d(dim),
        ) for i in range(blocks)],
        nn.AdaptiveAvgPool2d((1,1)),
        nn.Flatten(),
        nn.Linear(dim, n_classes)
    )
```

Figure 14: SplitMixer-V.

H Further Comparison with State of the Art

To illustrate the efficiency of the proposed modifications, in Figure 15 we plot accuracy *vs.* number of parameters for our models and state-of-the-art models that do not use external data for training. Over CIFAR-{10,100} datasets, in the low-parameter regime, our models push the envelope towards the top-left corner, which means a better trade-off between accuracy and model size (also speed). Our models even outperform some very well-known architectures such as MobileNet (Howard et al., 2017)

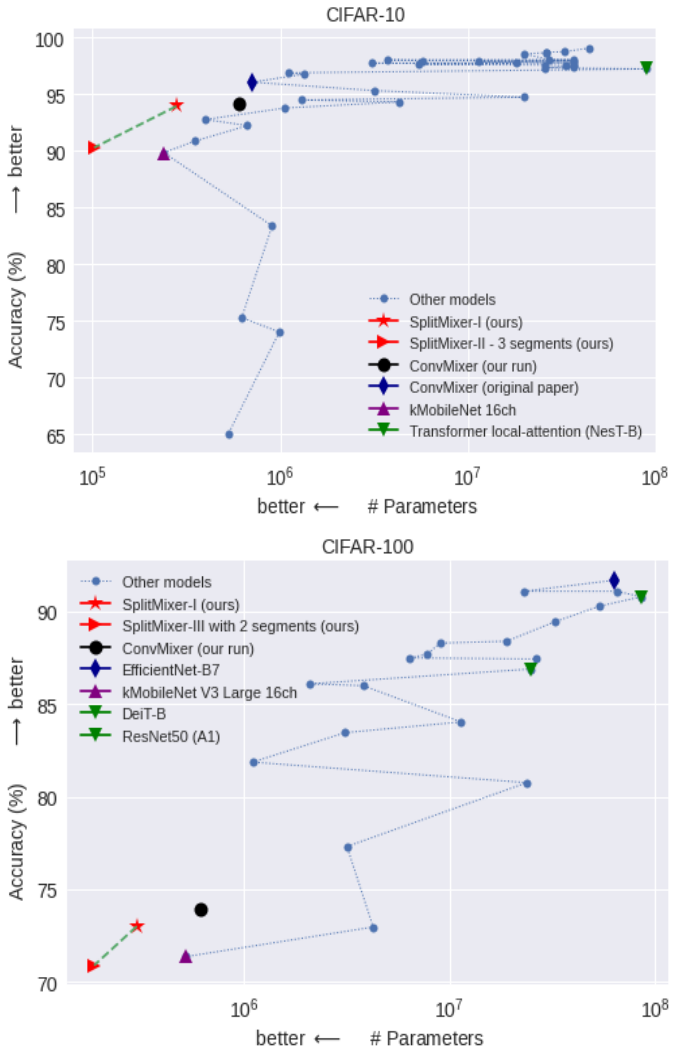

Figure 15: Comparison of our proposed SplitMixer architectures with state-of-the-art models that do not use external data for training. Results are shown over CIFAR-{10,100} datasets. Notice that **we have not optimized our models for the best performance**. Rather, we ran the ConvMixer and our models using the exact same code, parameters, and machines to measure how much we can save parameters and computation relative to ConvMixer. Please consult Trockman & Kolter (2021) for a more detailed comparison of ConvMixer with other models. We have borrowed some data from `https://paperswithcode.com/` to generate these plots.

I VISUALIZATION OF FEATURE MAPS

Here, we visualize the output maps of the SplitMixer-I-256/8 model trained over CIFAR-10 with $\alpha = 2/3$, $p = 2$, and $k = 5$. Notice that in each block 170 channels are convolved (first 170 channels in even blocks and the second 170 channels in odd blocks; with overlap). Notice that the learned 2×2 kernels for patch embedding are too small to plot here.

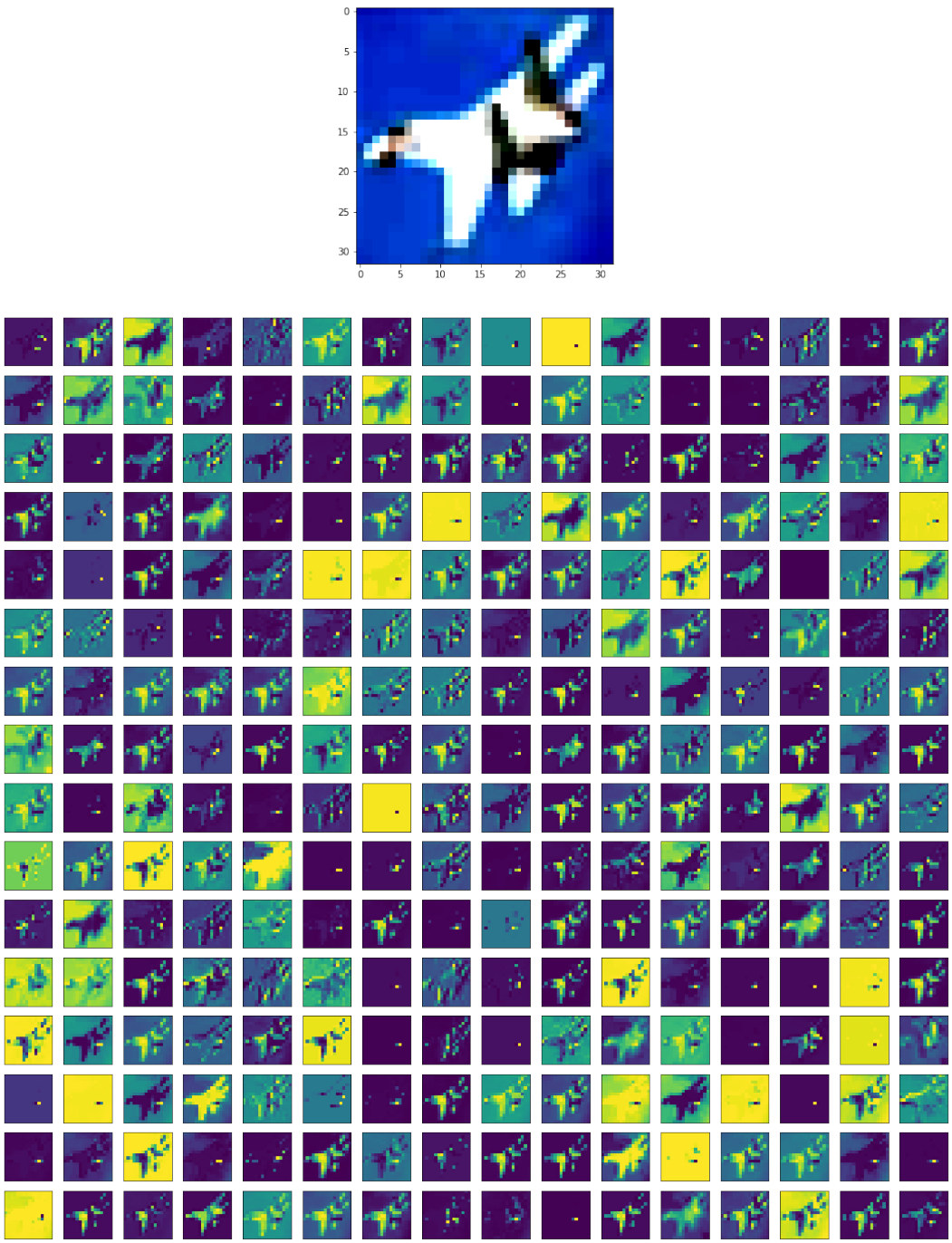

Figure 16: Top) A sample image from CIFAR-10 dataset, Bottom) The 256 feature maps after patch embedding (*i.e.* after the first conv2d).

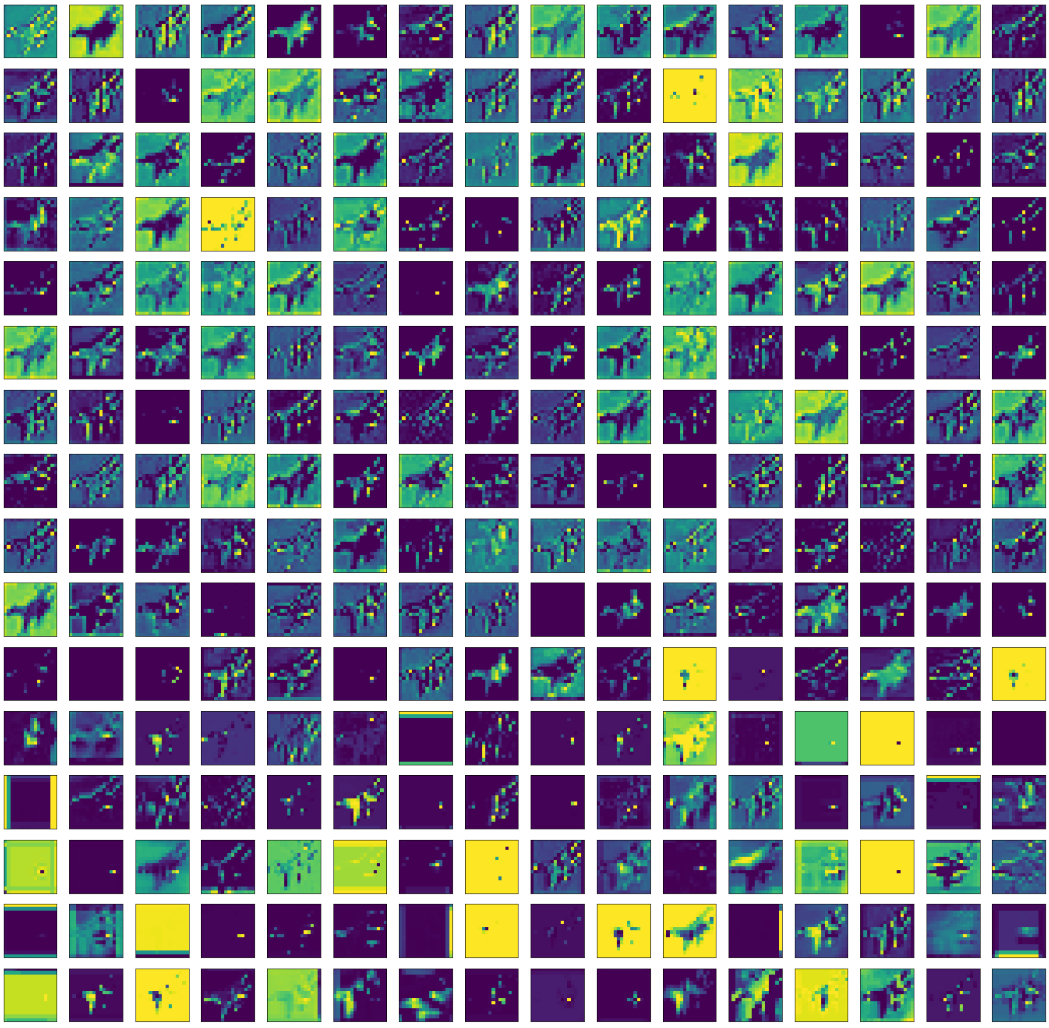

Figure 17: The output maps after the first SplitMixer block.

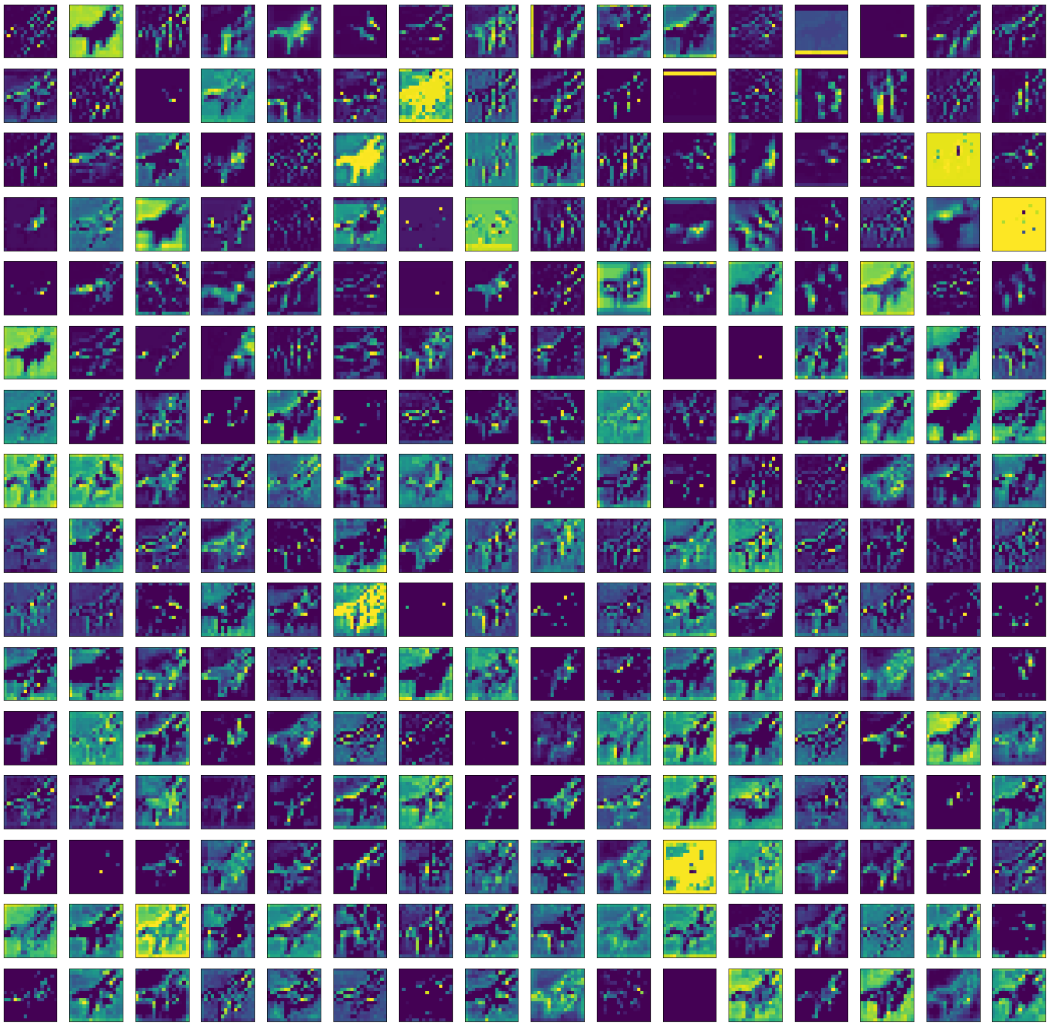

Figure 18: The output maps after the second SplitMixer block.

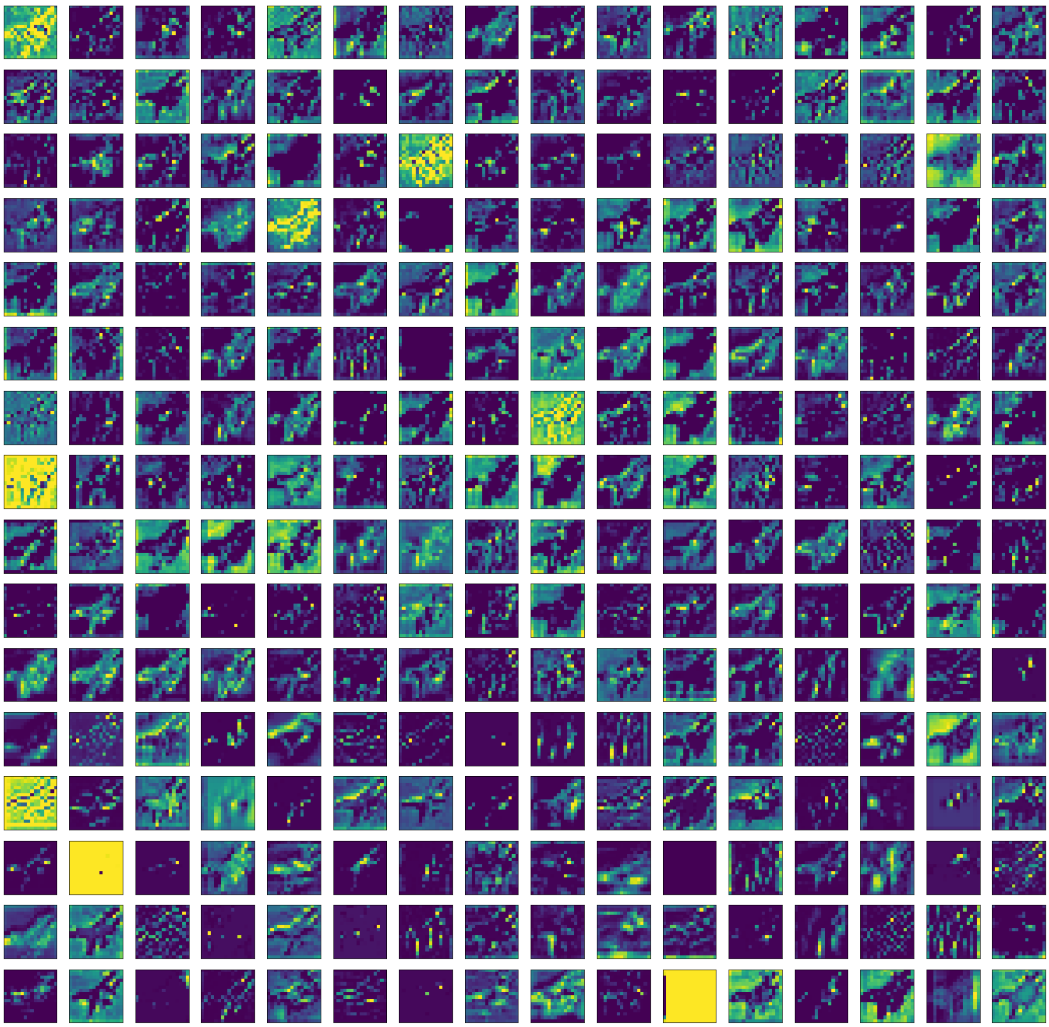

Figure 19: The output maps after the third SplitMixer block.

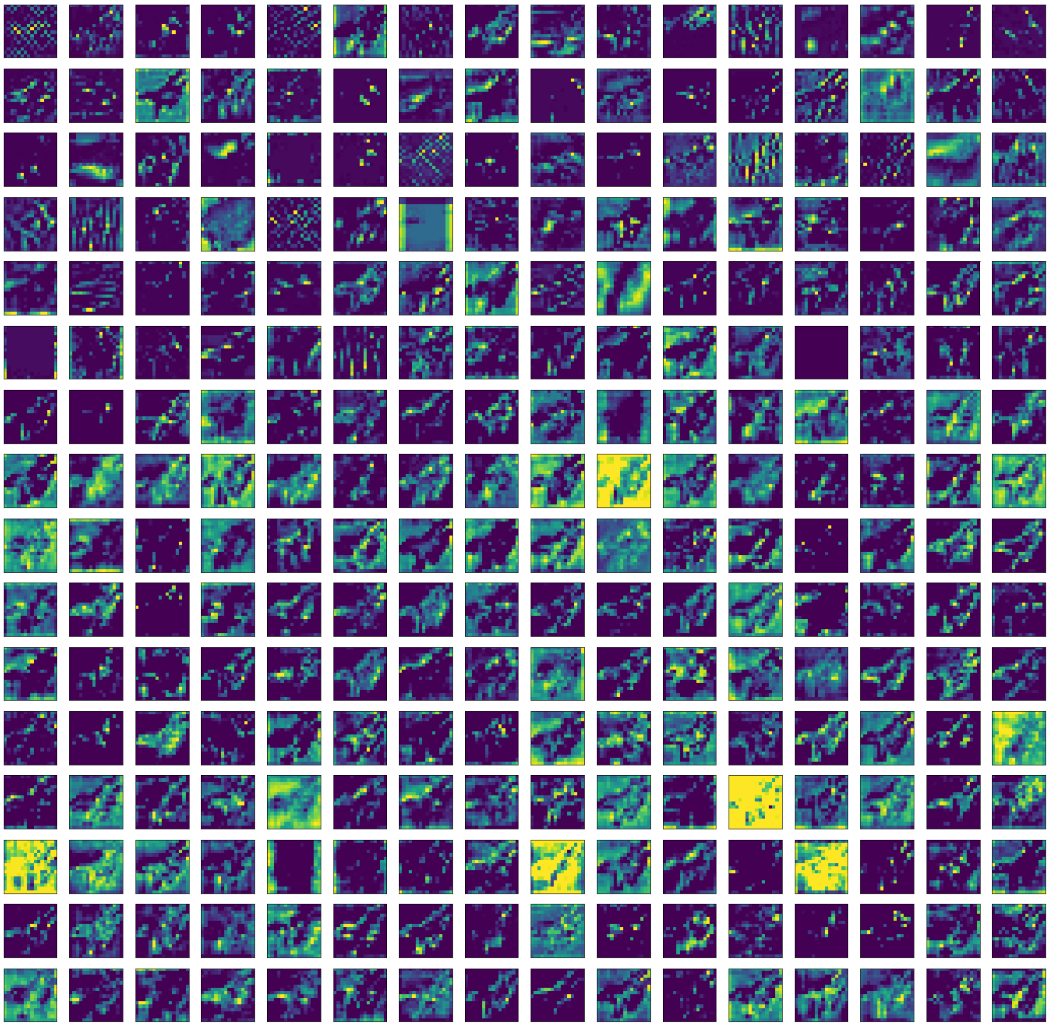

Figure 20: The output maps after the fourth SplitMixer block.