SplitMixer: Fat Trimmed From MLP-like Models

Ali Borji

Paper: https://arxiv.org/pdf/2207.10255.pdf (Borji & Lin)
Code: https://github.com/aliborji/splitmixer
Slides: https://github.com/aliborji/splitmixer/blob/main/slides.pdf

MLP-based Architectures

- MLP-Mixer
- ResMLP
- Vision Permutator
- MDMLP
- …
- **ConvMixer [We build on this]**

SplitMixer: Fat Trimmed From MLP-like Models

Ali Borji[1], Sikun Lin[2*]
[1]Quintic AI, [2]University of California, Santa Barbara

July 26, 2022

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 about 52% fewer parameters and FLOPS. We hope that our results spark further research towards finding more efficient vision architectures and facilitate the development of MLP-like models. Code is available at https://github.com/aliborji/splitmixer.

MLP-Mixer

MLP-Mixer: An all-MLP Architecture for Vision

Ilya Tolstikhin*, Neil Houlsby*, Alexander Kolesnikov*, Lucas Beyer*, Xiaohua Zhai, Thomas Unterthiner, Jessica Yung, Andreas Steiner, Daniel Keysers, Jakob Uszkoreit, Mario Lucic, Alexey Dosovitskiy

*equal contribution

Google Research, Brain Team

{tolstikhin, neilhoulsby, akolesnikov, lbeyer, xzhai, unterthiner, jessicayung†, andstein, keysers, usz, lucic, adosovitskiy}@google.com

†work done during Google AI Residency

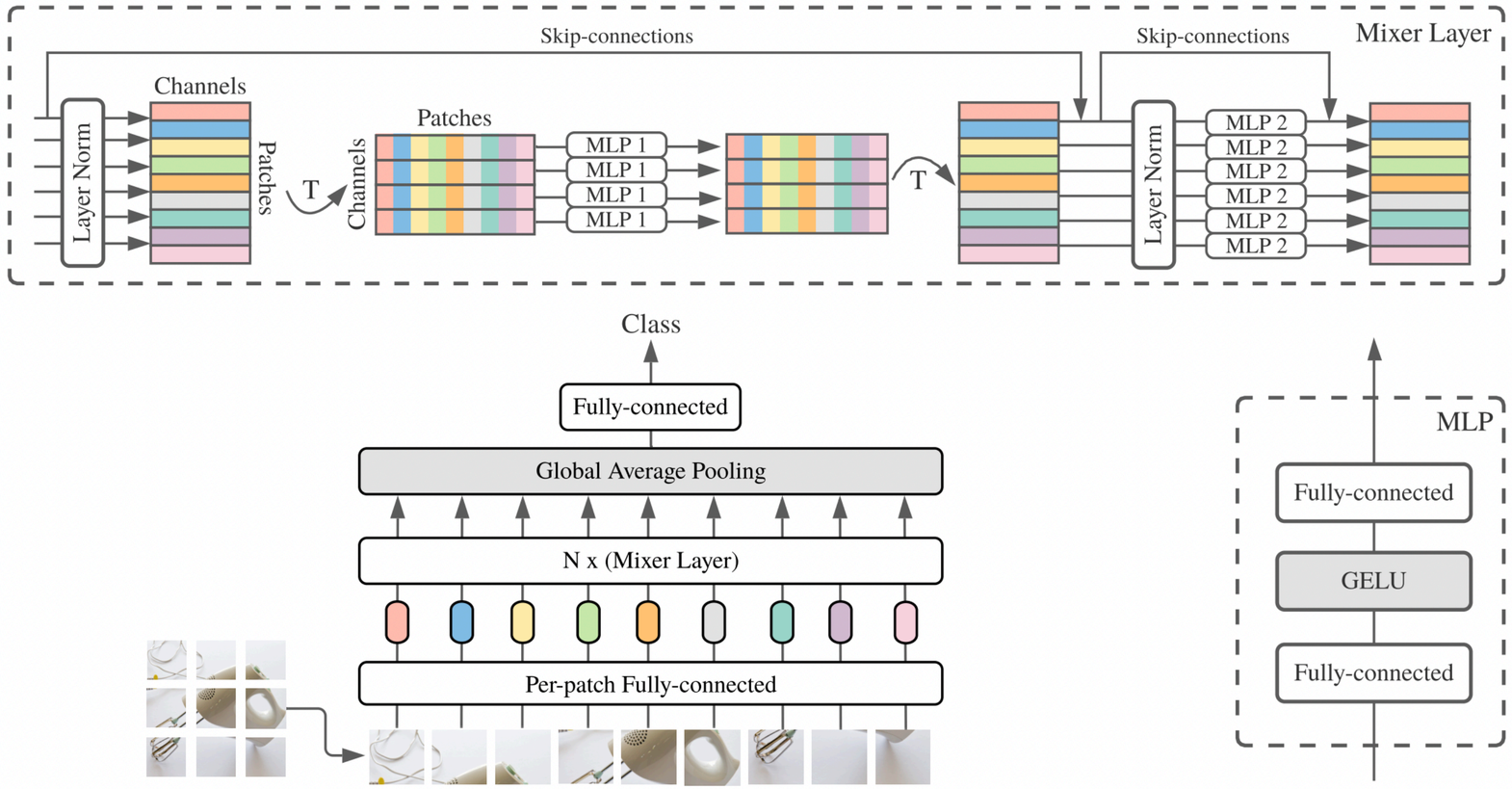


Figure 1: MLP-Mixer consists of per-patch linear embeddings, Mixer layers, and a classifier head. Mixer layers contain one token-mixing MLP and one channel-mixing MLP, each consisting of two fully-connected layers and a GELU nonlinearity. Other components include: skip-connections, dropout, and layer norm on the channels.

ResMLP

ResMLP: Feedforward networks for image classification with data-efficient training

Hugo Touvron[1,2] Piotr Bojanowski[1] Mathilde Caron[1,3]
Matthieu Cord[2,4] Alaaeldin El-Nouby[1,3] Edouard Grave[1] Gautier Izacard[1]
Armand Joulin[1] Gabriel Synnaeve[1] Jakob Verbeek[1] Hervé Jégou[1]

[1]Facebook AI [2]Sorbonne University [3]Inria [4]valeo.ai

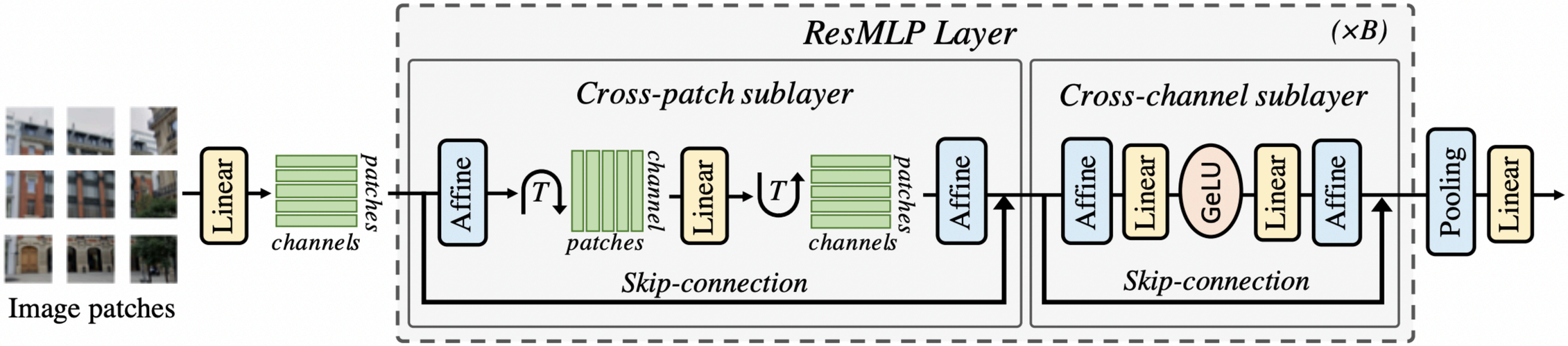


Figure 1: **The ResMLP architecture.** After linearly projecting the image patches into high dimensional embeddings, ResMLP sequentially processes them with (1) a cross-patch linear sublayer; (2) a cross-channel two-layer MLP. The MLP is the same as the FCN sublayer of a Transformer. Each sublayer has a residual connection and two Affine element-wise transformations.

ConvMixer

~~Convolutions~~ ~~Attention~~ ~~MLPs~~
Patches Are All You Need? 🤷

Asher Trockman, J. Zico Kolter[1]
Carnegie Mellon University and [1]Bosch Center for AI

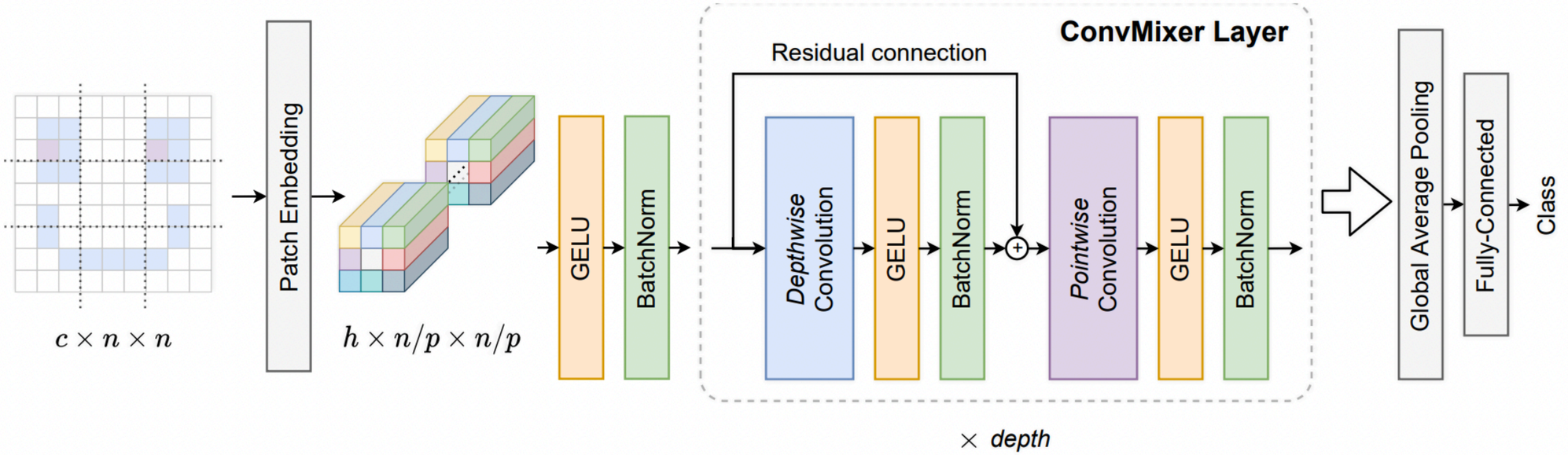
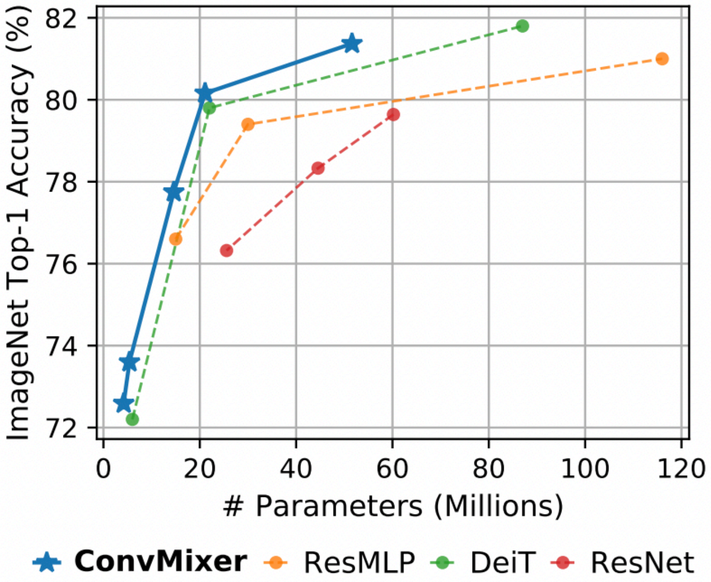


Figure 2: ConvMixer uses "tensor layout" patch embeddings to preserve locality, and then applies d copies of a simple fully-convolutional block consisting of *large-kernel* depthwise convolution followed by pointwise convolution, before finishing with global pooling and a simple linear classifier.

ConvMixer



Current "Most Interesting" **ConvMixer** Configurations *vs.* Other Simple Models							
Network	Patch Size	Kernel Size	# Params ($\times 10^6$)	Throughput (img/sec)	Act. Fn.	# Epochs	ImNet top-1 (%)
ConvMixer-1536/20	7	9	51.6	134	G	150	81.37
ConvMixer-768/32	7	7	21.1	206	R	300	80.16
ResNet-152	–	3	60.2	828	R	150	79.64
DeiT-B	16	–	86	792	G	300	81.8
ResMLP-B24/8	8	–	129	181	G	400	81.0

Table 1: Models trained and evaluated on 224×224 ImageNet-1k only. See more in Appendix A.

SplitMixer

- A simple and lightweight isotropic MLP-like architecture, for visual recognition
- Two types of interleaving convolutional operations:
  - Spatial mixing
  - Channel mixing
- **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, . . .)**

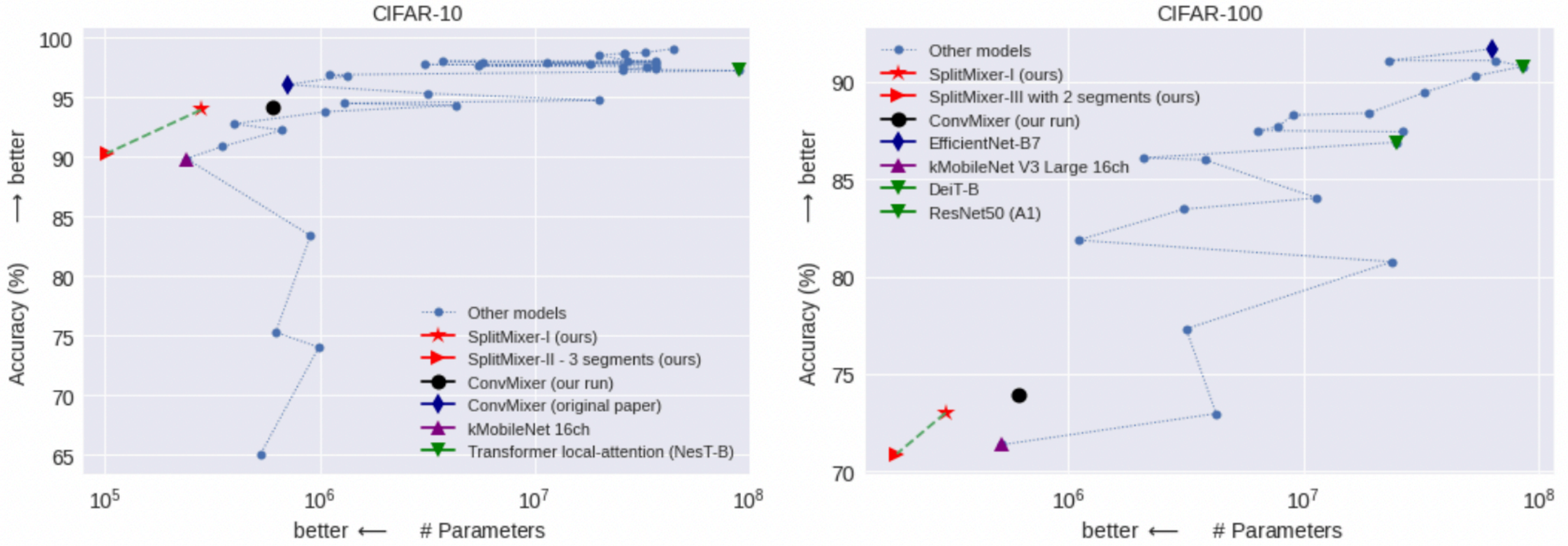


Figure 1: Comparison of our proposed SplitMixer architectures with state-of-the-art models that do not use external data for training. Results are shown over CIFAR-{10,100} datasets. Notice that **we have not optimized our models for the best performance**. Rather, we ran the ConvMixer and our models using the exact same code, parameters, and machines to measure how much we can save parameters and computation relative to ConvMixer. Please consult [33] for a more detailed comparison of ConvMixer with other models. We have borrowed some data from `https://paperswithcode.com/` to generate these plots.

SplitMixer

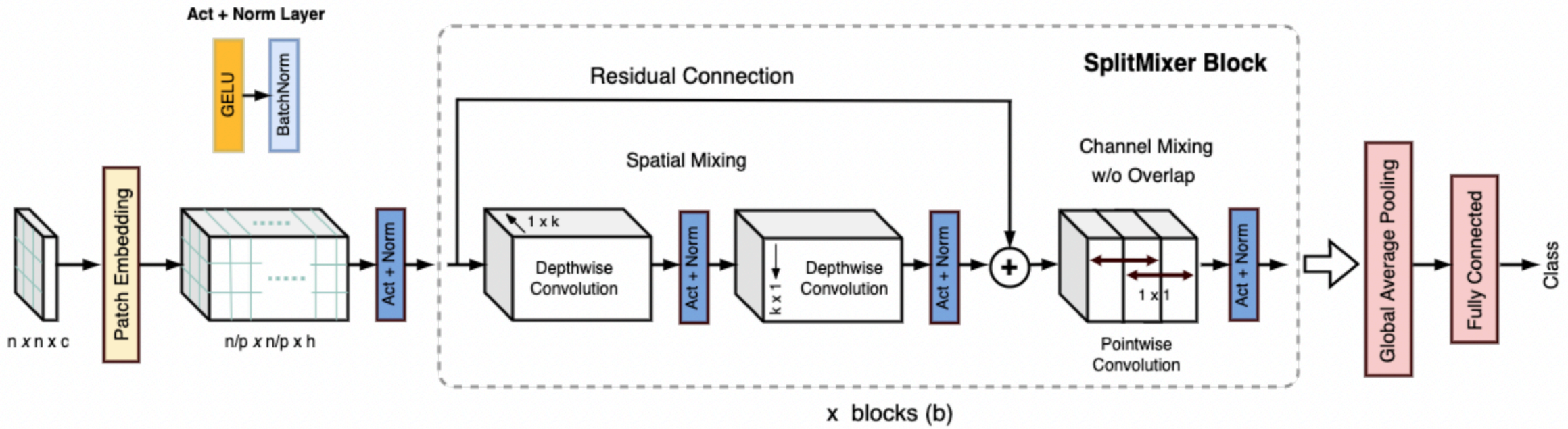


Figure 2: Basic architecture of SplitMixer. The input image is evenly divided into several image patches which are tokenized with linear projections. A number of 1D depthwise convolutions (spatial mixing) and pointwise convolutions (channel mixing) are repeatedly applied to the projections. For channel 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.

Spatial Mixing

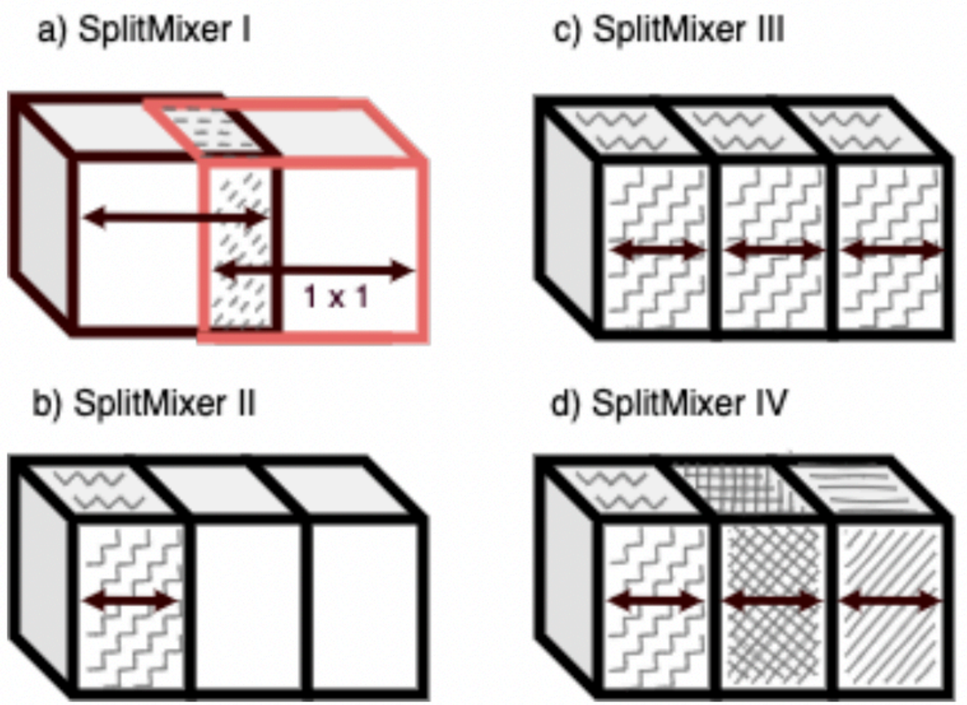
- Replace the k × k kernels in ConvMixer by two 1D kernels:
  - a 1 × k kernel across width
  - a k × 1 kernel across height
- **Saving in parameters:** $k^2 \times h$ parameters is reduced to $2k \times h$ in each SplitMixer block
- **Saving in FLOPS:** $W \times H \times k^2$ FLOPS is reduced to $W \times H \times 2k$, where W and H are width and height of the input tensor $\mathbf{X} \in \mathbb{R}^{W \times H \times h}$
- **Separating the 2D kernel into two 1D kernels results in k/2 times savings in parameters and FLOPS**

Channel Mixing

- **Most of the 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. Saving in FLOPS: $W \times H \times k^2$ FLOPS is reduced to $W \times H \times 2k$, where W and H are width and height of the input tensor
- **Thus, the fraction of parameters in the two parts is k^2/h (e.g. $5^2/256$)**

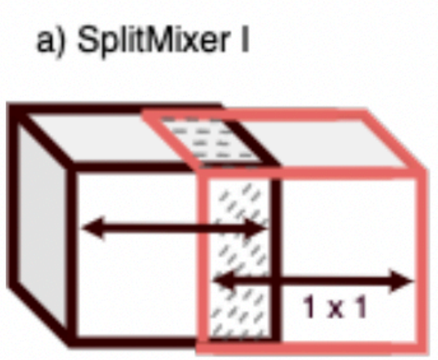
Channel Mixing

Figure 3: 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).



SplitMixer-I:
Overlapping segments, no parameter sharing, update one segment per block

- The input tensor is split into two overlapping segments along the channel dimension
- 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. (e.g., $\alpha = 2/3$)
- Apply convolution to only one segment in each block, e.g. the left segment in odd blocks and the right segment in even blocks
- The output has the same number of channels as the original segment, which is then concatenated to the other (unaltered) segment



SplitMixer-I: (cnt'd)
Overlapping segments, no parameter sharing, update one segment per block

- We choose $\alpha = \frac{i}{2i-1}, i \in \{2 \cdots 6\}$
- **Saving in parameters:** per block can be approximated as:

$$h^2 - (\alpha \times h)^2 = (1 - \alpha^2) \times h^2$$

- Which means 1 – α^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
- **Saving in FLOPS:** can be approximated as:

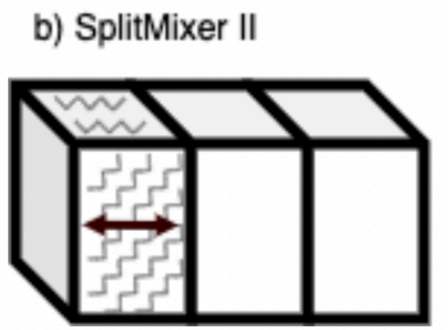
$$W \times H \times h \times h - W \times H \times (\alpha \times h) \times (\alpha \times h) = (1 - \alpha^2) \times W \times H \times h^2$$

SplitMixer-II:
Non-overlapping segments, no parameter sharing, update one segment per block

- We first split the **h** channels into **s** non-overlapping segments, each with size **h/s**, along the channel dimension

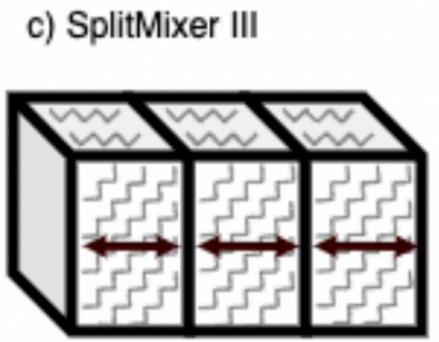
- In each block, only one segment is convolved and updated. Parameters are not shared across the segments

- **Saving in parameters and FLOPS:** $1 - \frac{1}{s^2}$
  For example, for s = 2, roughly 75% of the parameters are reduced. The same argument holds for FLOPS



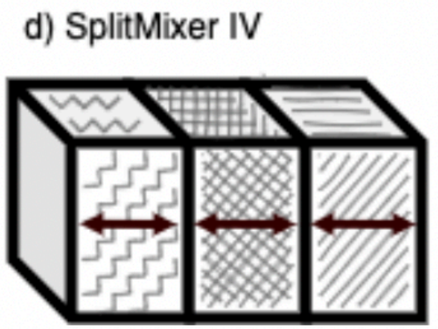
SplitMixer-III:
Non-overlapping segments, parameter sharing, update all segments per block

- Here, **h** channels are split into **s** non-overlapping segments with shared parameters (**h** must be divisible by **s**)
- All segments are convolved and updated simultaneously in each block
- **Saving in parameters:** Due to parameter sharing, the reduction in parameters is the same as SplitMixer-II
- **The number of FLOPS, however, is higher now since computation is done over all s segments**



SplitMixer-IV:
Non-overlapping segments, no parameter sharing, update all segments per block

- 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. **Saving in parameters** per block is:

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

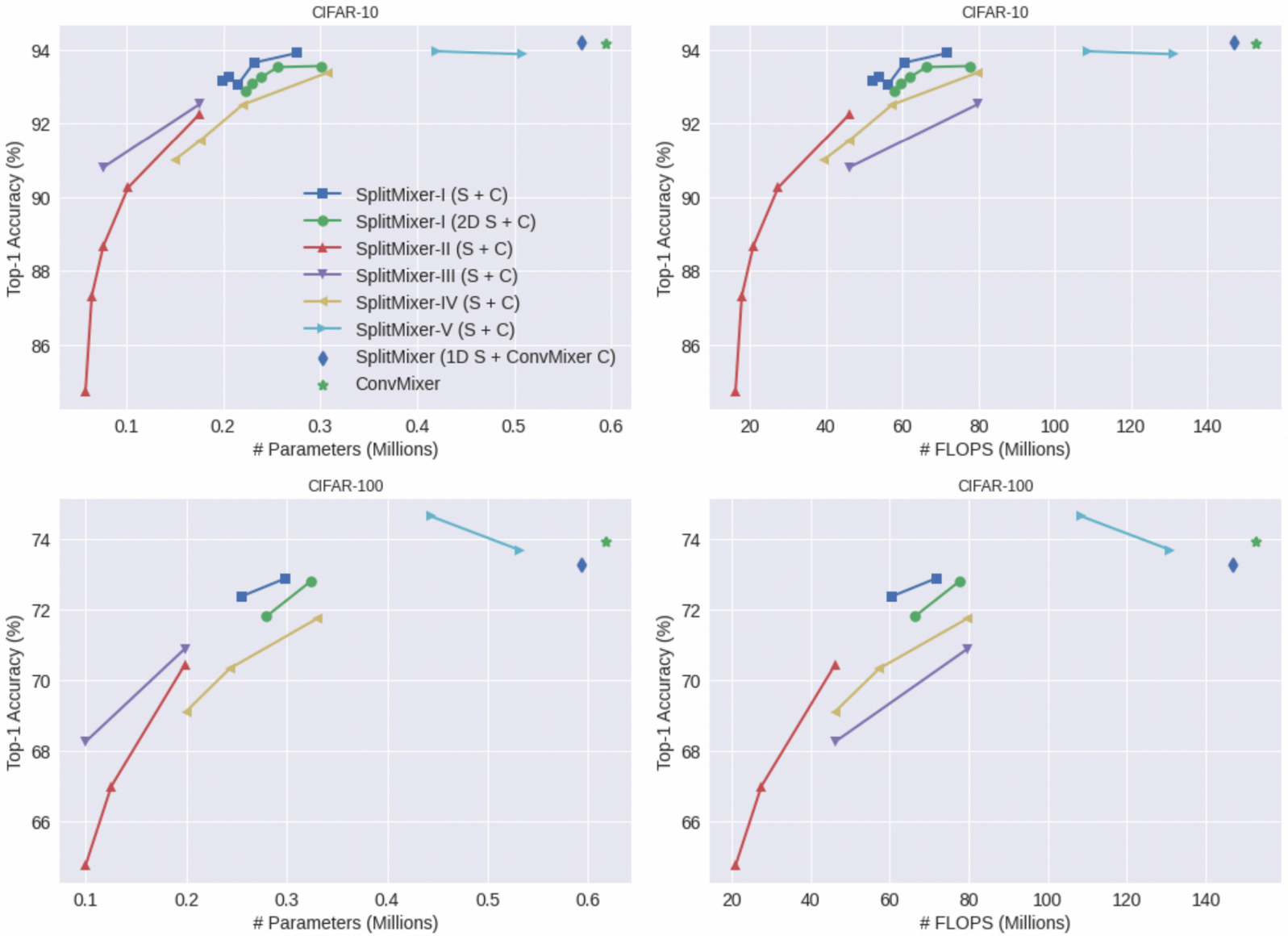
- Which results in **1 – 1/s** parameter saving. For example, 66.6% of the parameters are reduced for s = 3
- **Saving in FLOPS**: $W \times H \times h \times h - s \times (W \times H \times \frac{h}{s} \times \frac{h}{s}) = (1 - \frac{1}{s}) \times W \times H \times h^2$
- Which means the same saving in FLOPS as in parameters

Experiments Setup

- We used RandAugment, 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}**
  - 64 over **Flowers102 and Food101** datasets
- Across all datasets, **h and b** were set to **256 and 8.**
- We used AdamW 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).

Results on CIFAR-{10,100}

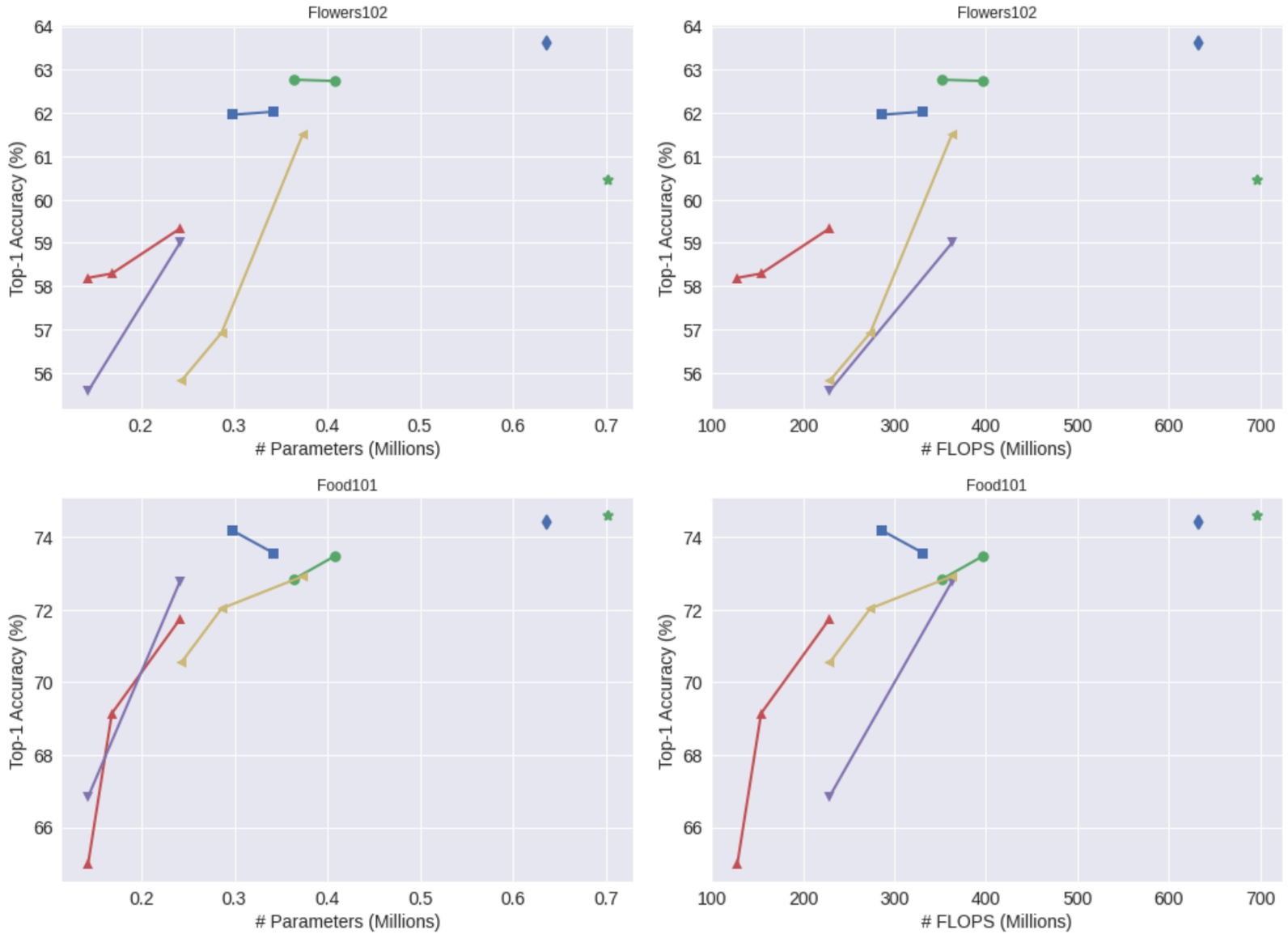
- patch size p = 2
- Kernel size k = 5



Results on Flowers102 and Food101

- patch size p = 7
  kernel size k = 7

- Image size
  is bigger in these
  datasets
  (both resized to
  224 × 224)



Comparison with other models

- 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 borrowed from Lv et al., 2022 (MDMLP) where they have trained models for 200 epochs.
- We have trained ConvMixer and SplitMixer for 100 epochs.

Model Family	Model	Params (M)	FLOPS (G)	Top-1 ACC (%)	
				CIFAR10	CIFAR100
CNN	ResNet20 [11]	**0.27**	**0.04**	91.99	67.39
Transformer	ViT [9]	2.69	0.19	86.57	60.43
MLP	AS-MLP [19]	26.20	0.33	87.30	65.16
”	gMLP [20]	4.61	0.34	86.79	61.60
”	ResMLP [30]	14.30	0.93	86.52	61.40
”	ViP [13]	29.30	1.17	88.97	70.51
”	MLP-Mixer [29]	17.10	1.21	85.45	55.06
”	S-FC (β-LASSO) [25]	-	-	85.19	59.56
”	MDMLP [23]	0.30	0.28	90.90	64.22
”	ConvMixer [34]	0.60	0.15	**94.17**	**73.92**
”	SplitMixer-I (ours)	0.28	0.07	93.91	72.44

Model Family	Model	Params (M)	FLOPS (G)	ACC (%)	
				Flowers102	Food101
CNN	ResNet20 [11]	**0.28**	2.03	57.94	74.91
Transformer	ViT [9]	2.85	0.94	50.69	66.41
MLP	AS-MLP [19]	26.30	1.33	48.92	74.92
”	gMLP [20]	6.54	1.93	47.35	73.56
”	ResMLP [30]	14.99	1.23	45.00	68.40
”	ViP [13]	30.22	1.76	42.16	69.91
”	MLP-Mixer [29]	18.20	4.92	49.41	61.86
”	MDMLP [25]	0.41	1.59	60.39	**77.85**
”	ConvMixer [34]	0.70	0.70	60.47	74.59
”	SplitMixer-I (ours)	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 borrowed from [23] where they have trained models for 200 epochs. We have trained ConvMixer and SplitMixer for 100 epochs.

Ablation experiments

- Residual connections does not hurt the performance much.
- 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.
- LayerNorm, instead of BatchNorm, leads to 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 is not very important.
- 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. 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 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.

Ablation of SplitMixer-I-256/8 on CIFAR-$\{10, 100\}$

Ablation	CIFAR-10 Acc. (%)	CIFAR-100 Acc. (%)
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 2: Ablation study of SplitMixer-I-256/8 with split ratio of 2/3.

The role of the number of blocks

- 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.

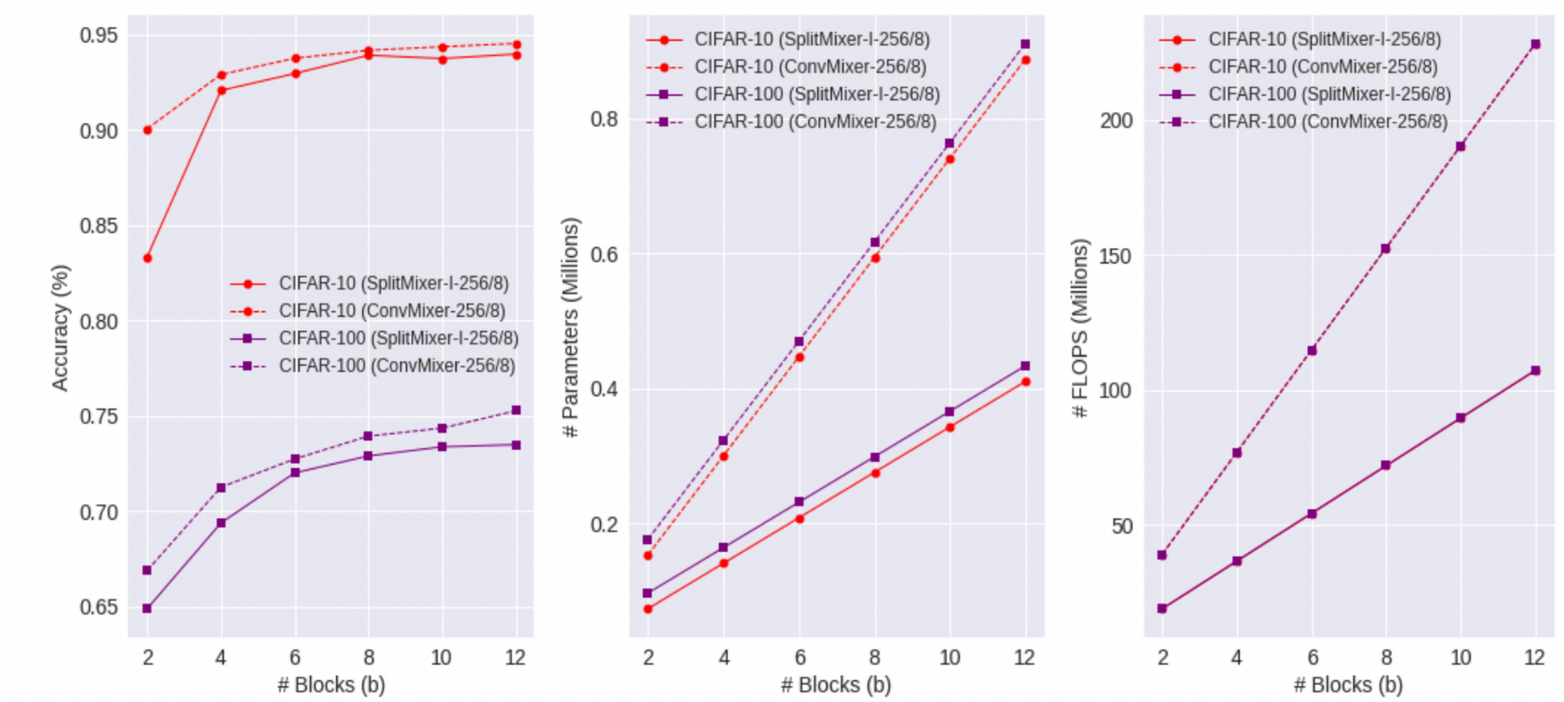


Figure 5: The role of the number of blocks b on model performance. The FLOPS of models over CIFAR-100 are just slightly higher than CIFAR-10, thus not visible in the rightmost panel.

Model throughput

- We measured throughput using batches of 64 images on a single Tesla v100 GPU with 32GB RAM, averaging over 100 such batches.
- We considered CUDA execution time rather than "wall-clock" time
- 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.
- The throughput of our model is almost three times higher than ConvMixer.

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 3: 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.

Discussion and Conclusion

- We proposed SplitMixer, an extremely simple yet very 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 for channel mixing. It performs 1×1 convolution on each segment
- **Our main point is that SplitMixer allows sacrificing a small amount of accuracy to achieve big gains in reducing parameters and FLOPS.**

- **Future research in this area:**
  - 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 [24], and stochastic depth [15].
  - We tried a number of ways to split and mix the channels and learned that some perform better than the others. There might be even better approaches to do this,
  - MLP-like models, including SplitMixer, lack effective means of explanation and visualization, which need to be addressed in the future
  - Large internal resolution and isotropic design make SplitMixer-type models appealing for vision tasks such as semantic segmentation and object detection.

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)

```

```python
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)
    )
```

https://github.com/aliborji/splitmixer


Thank
You