ETRI

MobileViT: Light-weight, General-purpose, and Mobile-friendly Vision Transformer

Published in ICLR'22 (Apple) Cited by 61 (arxiv Oct. 21)

Speaker: Jemin Lee

https://leejaymin.github.io/index.html

Up-to-date List

Backbone

- **MobileViT: "MobileViT: Light-weight, General-purpose, and Mobile-friendly Vision Transformer", ICLR, 2022 (Apple).**
  - https://iclr.cc/virtual/2022/poster/6230
- MobileViTv2: "Separable Self-attention for Mobile Vision Transformers", arXiv22.6
- EfficientFormer: "EfficientFormer: Vision Transformers at MobileNet Speed", arXiv22.6
- TinyViT, ECCV22
- EfficientViT, Arxiv22.6
- EdgeViTs, Arxiv22.7
- EdgeNeXt Arxiv22.7
- LightViT Arxiv22.7

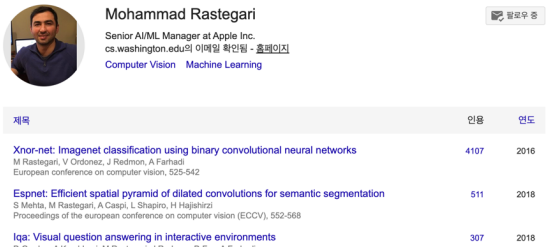
Machine Learning Research Overview Research Updates and Events Work with us

Computer Vision | ICLR

Paper | March 2022

MobileViT: Light-weight, General-purpose, and Mobile-friendly Vision Transformer

Sachin Mehta, Mohammad Rastegari



View publication ↗

View source code (GitHub) ↗

Copy Bibtex

Light-weight convolutional neural networks (CNNs) are the de-facto for mobile vision tasks. Their spatial inductive biases allow them to learn representations with fewer parameters across different vision tasks. However, these networks are spatially local. To learn global representations, self-attention-based vision trans-formers (ViTs) have been adopted. Unlike CNNs, ViTs are heavy-weight. In this paper, we ask the following question: is it possible to combine the strengths of CNNs and ViTs to build a light-weight and low latency network for mobile vision tasks? Towards this end, we introduce MobileViT, a light-weight and general-purpose vision transformer for mobile devices. MobileViT presents a different perspective for the global processing of information with transformers, i.e., transformers as convolutions. Our results show that MobileViT significantly outperforms CNN- and ViT-based networks across different tasks and datasets. On the ImageNet-1k dataset, MobileViT achieves top-1 accuracy of 78.4% with about 6 million parameters, which is 3.2% and 6.2% more accurate than MobileNetv3 (CNN-based) and DeIT (ViT-based) for a similar number of parameters. On the MS-COCO object detection task, MobileViT is 5.7% more accurate than MobileNetv3 for a similar number of parameters.

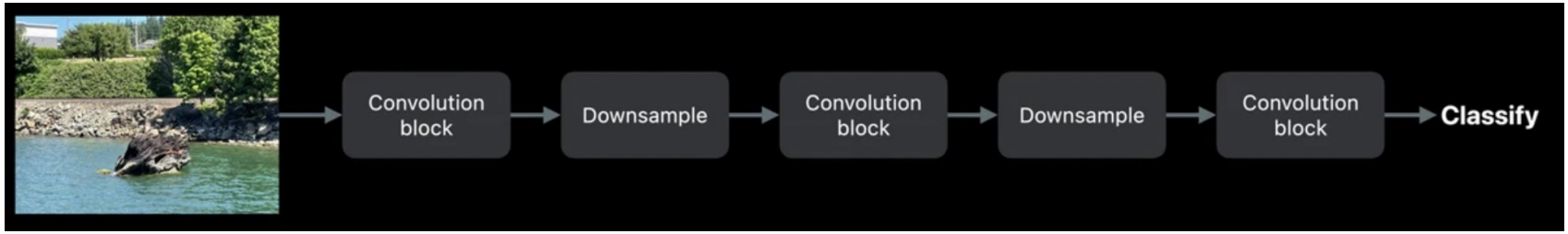
Learning Visual Representations in CNN

Pros

- Inductive bias
- Variably-sized datasets
- Fast
- Accurate
- General

Cons

- Local representations
- Weak data scalability



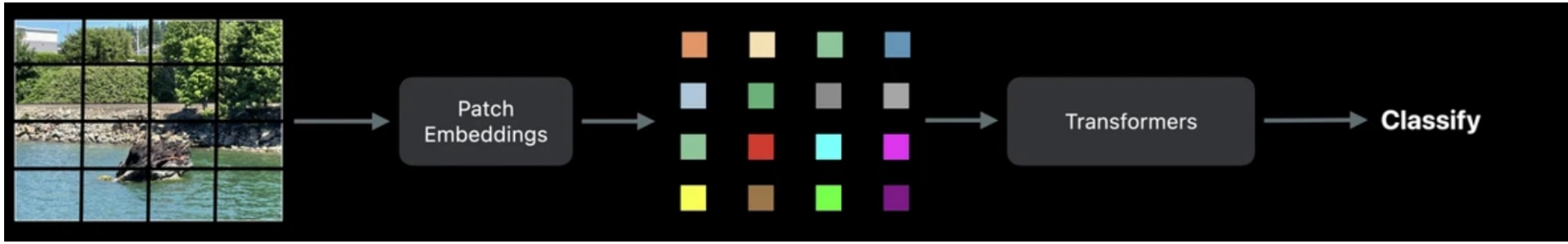
Learning Visual Representations in ViT

Pros

- Non-local representations
- Competitive to heavy-weight CNNs (Resnet50)

Cons

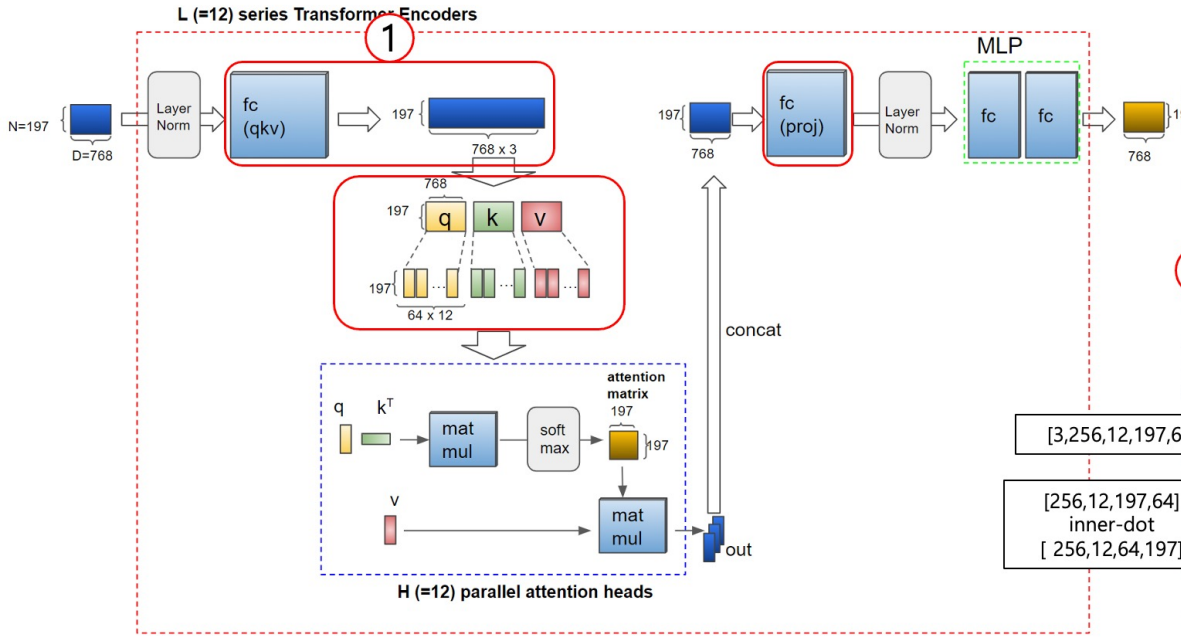
- No inductive bias
- Large datasets
- Slow
- Poor performance

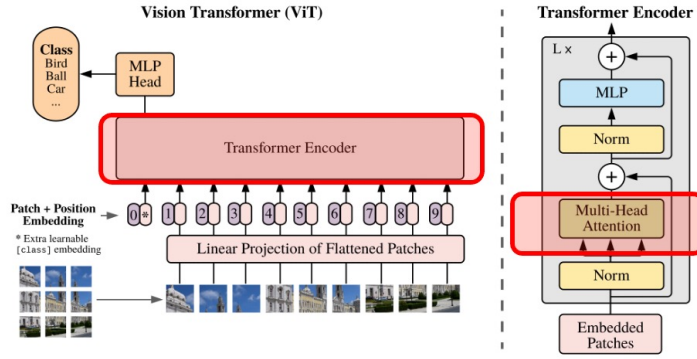


Vision Transformer (ViT) ICLR 2021 (ArXiv on Oct. 2020)
Cited by 6k within 2 years

Recap ViT

Transformer in ViT






```python
class Attention(nn.Module):
    def __init__(self, dim, num_heads=8, qkv_bias=False, attn_drop=0., proj_drop=0.):
        super().__init__()
        assert dim % num_heads == 0, 'dim should be divisible by num_heads'
        self.num_heads = num_heads
        head_dim = dim // num_heads
        self.scale = head_dim ** -0.5

1       self.qkv = nn.Linear(dim, dim * 3, bias=qkv_bias)
        self.attn_drop = nn.Dropout(attn_drop)
        self.proj = nn.Linear(dim, dim)
        self.proj_drop = nn.Dropout(proj_drop)

    def forward(self, x):
        B, N, C = x.shape
        qkv = self.qkv(x).reshape(B, N, 3, self.num_heads, C // self.num_heads).permute(2, 0, 3, 1, 4)
        q, k, v = qkv.unbind(0)   # make torchscript happy

        attn = (q @ k.transpose(-2, -1)) * self.scale
        attn = attn.softmax(dim=-1)
        attn = self.attn_drop(attn)

        x = (attn @ v).transpose(1, 2).reshape(B, N, C)
        x = self.proj(x)
        x = self.proj_drop(x)
        return x
```

B: 256, C:768, N:197

[3,256,12,197,64]

[256,12,197,64]
Batch,
Head,
P^2 + 1
768(16x16x3) /12

[256,12,197,64]
inner-dot
[256,12,64,197]

[256,12,197,197]
inner-dot
[256,12,197,64]

256,197,768

URL colab

https://youtu.be/L7_nNfVeIw8

What MobileViT focused on designing for

Transformer

- Non-local representations
- Competitive to CNNs (light (mobilenet) + heavy (resnet))

CNN

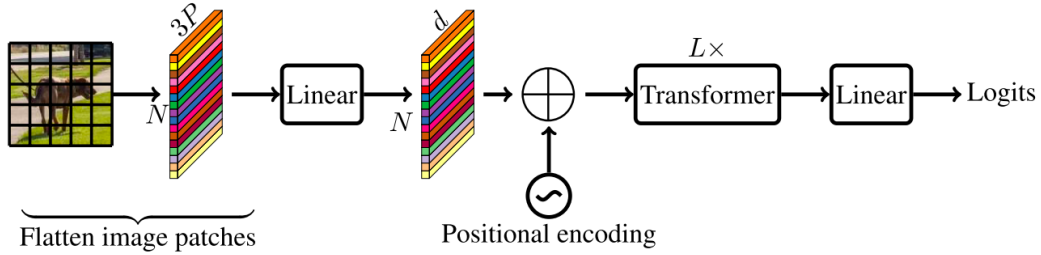
- Inductive bias
- Variably-sized datasets

Fast

Accurate

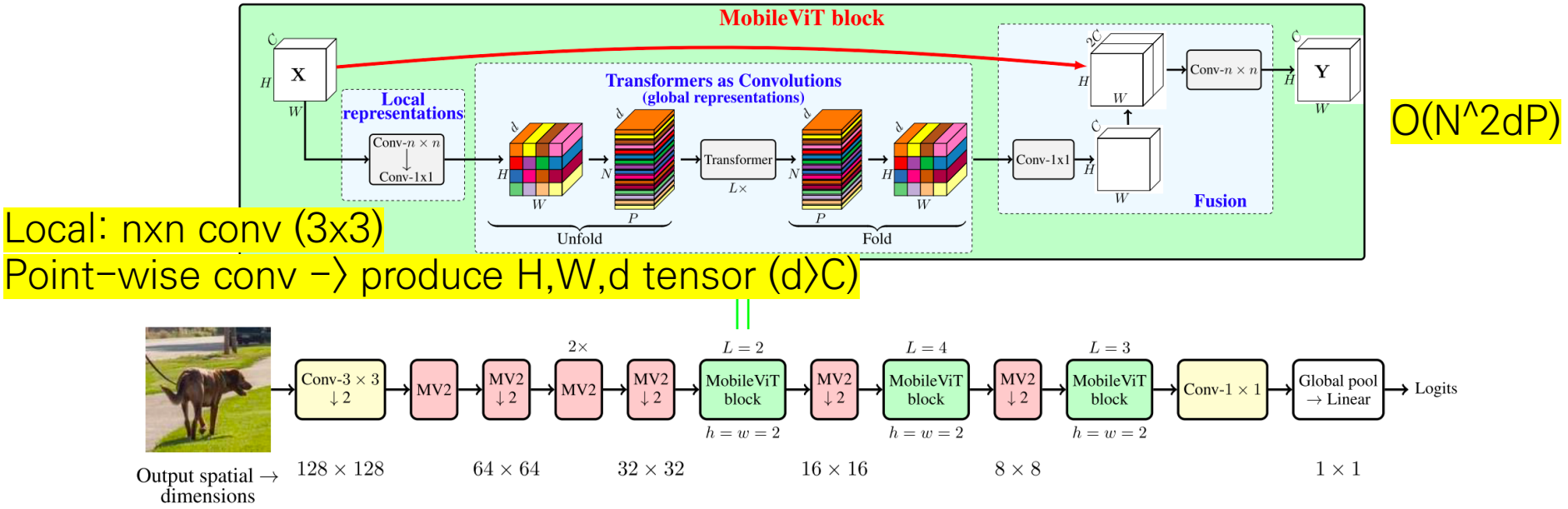
General

Model the local & global information



O(N^2 d)

(a) **Standard visual transformer (ViT)**



O(N^2dP)

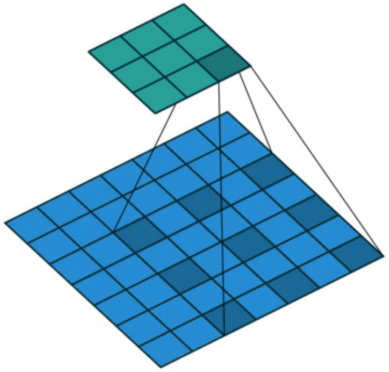
(b) **MobileViT**. Here, Conv-$n \times n$ in the MobileViT block represents a standard $n \times n$ convolution and MV2 refers to MobileNetv2 block. Blocks that perform down-sampling are marked with $\downarrow 2$.

Figure 1: **Visual transformers vs. MobileViT**

Model Long range dependencies with Dilated Conv.

Dilated conv could learn long range dependences in input tensors

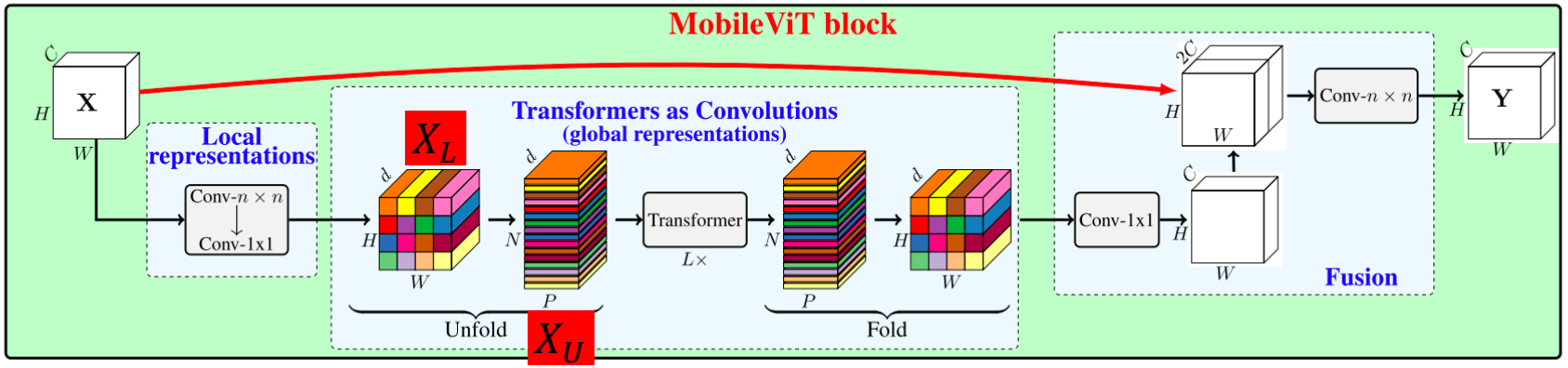
However, it requires careful selection of dilation rates



Alternative: self-attention (ViT) -> heavy weights

MobileViT -> learn global representation with spatial inductive bais

MobileViT block



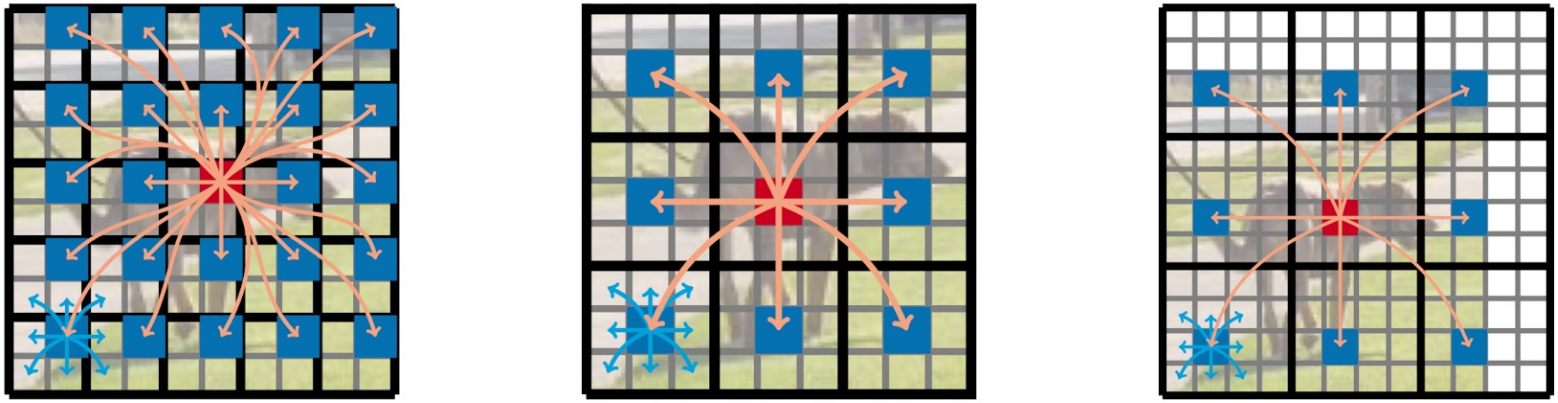
To enable MobileViT to learn global representations with spatial inductive bias, we unfold $\mathbf{X}_L$ into N non-overlapping flattened patches $\mathbf{X}_U \in \mathbb{R}^{P \times N \times d}$. Here, $P = wh$, $N = \frac{HW}{P}$ is the number of patches, and $h \leq n$ and $w \leq n$ are height and width of a patch respectively. For each $p \in \{1, \cdots, P\}$, inter-patch relationships are encoded by applying transformers to obtain $\mathbf{X}_G \in \mathbb{R}^{P \times N \times d}$ as:

$$\mathbf{X}_G(p) = \text{Transformer}(\mathbf{X}_U(p)), 1 \leq p \leq P \quad (1)$$

Unlike ViTs that lose the spatial order of pixels, MobileViT neither loses the patch order nor the spatial order of pixels within each patch (Figure 1b). Therefore, we can fold $\mathbf{X}_G \in \mathbb{R}^{P \times N \times d}$ to obtain $\mathbf{X}_F \in \mathbb{R}^{H \times W \times d}$. $\mathbf{X}_F$ is then projected to low C-dimensional space using a point-wise convolution and combined with $\mathbf{X}$ via concatenation operation. Another $n \times n$ convolutional layer is then used to fuse these concatenated features. Note that because $\mathbf{X}_U(p)$ encodes local information from $n \times n$ region using convolutions and $\mathbf{X}_G(p)$ encodes global information across P patches for the p-th location, each pixel in $\mathbf{X}_G$ can encode information from all pixels in $\mathbf{X}$, as shown in Figure 4. Thus, the overall effective receptive field of MobileViT is $H \times W$.

Patch sizes	# Params.	Time ⇓	Top-1 ⇑
2,2,2	5.7 M	9.85 ms	78.4
3,3,3†	5.7 M	14.69 ms	**78.5**
4,4,4	5.7 M	8.23 ms	77.6
8,4,2	5.7 M	8.20 ms	77.3

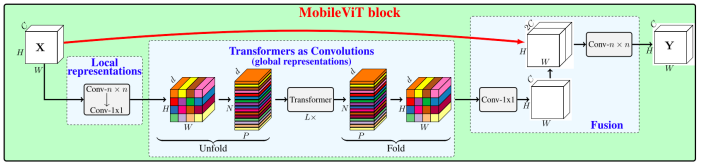
Table 6: **Impact of patch sizes**. Here, the patch sizes are for spatial levels at 32×32, 16×16, and 8×8, respectively. Also, results are shown for MobileViT-S model on the ImageNet-1k dataset. Results are with exponential moving average. † Spatial dimensions of feature map are not multiple of patch dimensions. Therefore, we use bilinear interpolation in folding and unfolding operations to resize the feature map.

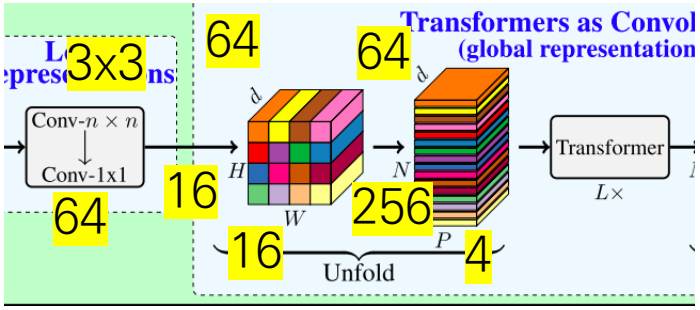


(a) $h = w = 2 < n = 3$ (b) $h = w = n = 3$ (c) $h = w = 4 > n = 3$

Figure 13: **Relationship between kernel size** ($n \times n$) **for convolutions and patch size** ($h \times w$) **for folding and unfolding in MobileViT.** In **a** and **b**, the **red** pixel is able to aggregate information from all pixels using local (**cyan** colored arrows) and global (**orange** colored arrows) information while in **(c)**, every pixel is not able to aggregate local information using convolutions with kernel size of 3×3 from 4×4 patch region. Here, each cell in **black** and **gray** grids represents a patch and pixel, respectively.

Details of MobileViT Block





[512, 64, 32, 32] −〉 [512x4 (BP), 16x16(N), 64 (C)]

X −〉 [2048, 4 (head), 256, 16 (c//head)]

```
def forward(self, x):   self: Attention(\n  (qkv): Linear(in_features=64, out_features=192, bias=True
    B, N, C = x.shape   B: 2048   C: 64   N: 256
    qkv = self.qkv(x).reshape(B, N, 3, self.num_heads, C // self.num_heads).permute(2, 0, 3, 1, 4)
    q, k, v = qkv.unbind(0)   # make torchscript happy (cannot use tensor as tuple)

    attn = (q @ k.transpose(-2, -1)) * self.scale
    attn = attn.softmax(dim=-1)
    attn = self.attn_drop(attn)

    x = (attn @ v).transpose(1, 2).reshape(B, N, C)
    x = self.proj(x)
    x = self.proj_drop(x)
    return x
```

```
def forward(self, x: torch.Tensor) -> torch.Tensor:
    shortcut = x

    # Local representation
    x = self.conv_kxk(x)
    x = self.conv_1x1(x)

    # Unfold (feature map -> patches)
    patch_h, patch_w = self.patch_size
    B, C, H, W = x.shape
    new_h, new_w = math.ceil(H / patch_h) * patch_h, math.ceil(W / patch_w) * patch_w
    num_patch_h, num_patch_w = new_h // patch_h, new_w // patch_w  # n_h, n_w
    num_patches = num_patch_h * num_patch_w  # N
    interpolate = False
    if new_h != H or new_w != W:
        # Note: Padding can be done, but then it needs to be handled in attention function.
        x = F.interpolate(x, size=(new_h, new_w), mode="bilinear", align_corners=False)
        interpolate = True

    # [B, C, H, W] --> [B * C * n_h, n_w, p_h, p_w]
    x = x.reshape(B * C * num_patch_h, patch_h, num_patch_w, patch_w).transpose(1, 2)
    # [B * C * n_h, n_w, p_h, p_w] --> [BP, N, C] where P = p_h * p_w and N = n_h * n_w
    x = x.reshape(B, C, num_patches, self.patch_area).transpose(1, 3).reshape(B * self.patch_area, num_patches, -1)

    # Global representations
    x = self.transformer(x)
    x = self.norm(x)

    # Fold (patch -> feature map)
    # [B, P, N, C] --> [B*C*n_h, n_w, p_h, p_w]
    x = x.contiguous().view(B, self.patch_area, num_patches, -1)
    x = x.transpose(1, 3).reshape(B * C * num_patch_h, num_patch_w, patch_h, patch_w)
    # [B*C*n_h, n_w, p_h, p_w] --> [B*C*n_h, p_h, n_w, p_w] --> [B, C, H, W]
    x = x.transpose(1, 2).reshape(B, C, num_patch_h * patch_h, num_patch_w * patch_w)
    if interpolate:
        x = F.interpolate(x, size=(H, W), mode="bilinear", align_corners=False)

    x = self.conv_proj(x)
    if self.conv_fusion is not None:
        x = self.conv_fusion(torch.cat((shortcut, x), dim=1))
    return x
```

MobileViT block

Relationship to convolution

- MobileViT block replaces the local processing (GEMM) in convolutions with deeper global processing
- Transformer as a convolution

Light-weight

- ViT, Deit -> L=12, d=192
- MobileViT -> L = {2,4,3}, d={96,120,144} at special level 32x32, 16x16, and 8x8

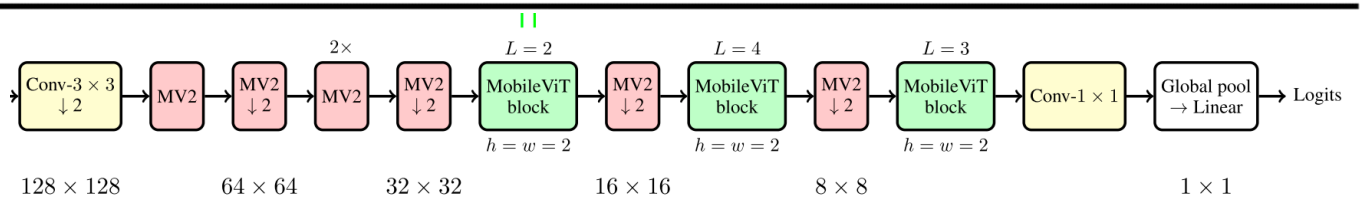
Computational cost. The computational cost of multi-headed self-attention in MobileViT and ViTs (Figure 1a) is $O(N^2Pd)$ and $O(N^2d)$, respectively. In theory, MobileViT is inefficient as compared to ViTs. However, in practice, MobileViT is more efficient than ViTs. MobileViT has $2\times$ fewer FLOPs and delivers 1.8% better accuracy than DeIT on the ImageNet-1K dataset (Table 3; §4.3). We believe that this is because of similar reasons as for the light-weight design (discussed above).

MobileViT architecture. Our networks are inspired by the philosophy of light-weight CNNs. We train MobileViT models at three different network sizes (S: small, XS: extra small, and XXS: extra extra small) that are typically used for mobile vision tasks (Figure 3c). The initial layer in MobileViT is a strided 3×3 standard convolution, followed by MobileNetv2 (or MV2) blocks and MobileViT blocks (Figure 1b and §A). We use Swish (Elfwing et al., 2018) as an activation function. Following CNN models, we use $n = 3$ in the MobileViT block. The spatial dimensions of feature maps are usually multiples of 2 and $h, w \leq n$. Therefore, we set $h = w = 2$ at all spatial levels (see §C for more results). The MV2 blocks in MobileViT network are mainly responsible for down-sampling. Therefore, these blocks are shallow and narrow in MobileViT network. Spatial-level-wise parameter distribution of MobileViT in Figure 3d further shows that the contribution of MV2 blocks towards total network parameters is very small across different network configurations.

Three types of MobileViT

Layer	Output size	Output stride	Repeat	Output channels		
				XXS	XS	S
Image	256×256	1				
Conv-3×3, $\downarrow 2$	128×128	2	1	16	16	16
MV2			1	16	32	32
MV2, $\downarrow 2$	64×64	4	1	24	48	64
MV2			2	24	48	64
MV2, $\downarrow 2$	32×32	8	1	48	64	96
MobileViT block ($L = 2$)			1	48 ($d = 64$)	64 ($d = 96$)	96 ($d = 144$)
MV2, $\downarrow 2$	16×16	16	1	64	80	128
MobileViT block ($L = 4$)			1	64 ($d = 80$)	80 ($d = 120$)	128 ($d = 192$)
MV2, $\downarrow 2$	8×8	32	1	80	96	160
MobileViT block ($L = 3$)			1	80 ($d = 96$)	96 ($d = 144$)	160 ($d = 240$)
Conv-1×1			1	320	384	640
Global pool	1×1	256	1			
Linear				1000	1000	1000
Network Parameters				1.3 M	2.3 M	5.6 M

Input = (100,3,256,256)
X = (400, 256, 64)
(2048, 256, 64)



MobileNetv2 Block

Linear bottlenecks

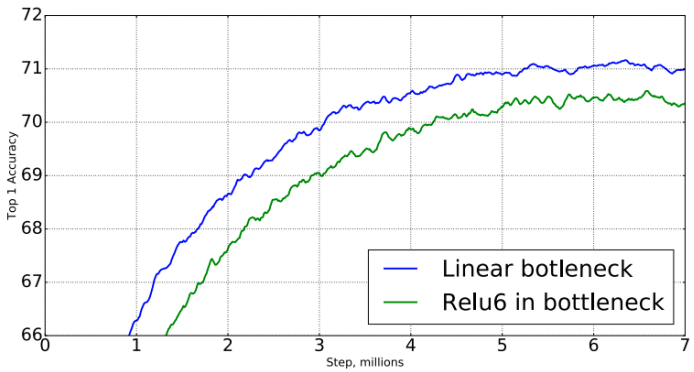
Inverted residuals

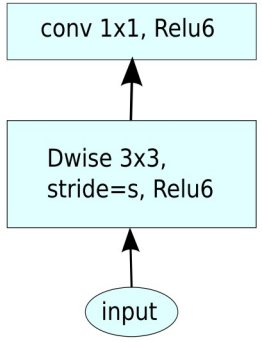
Two blocks

- Stride1, stride 2

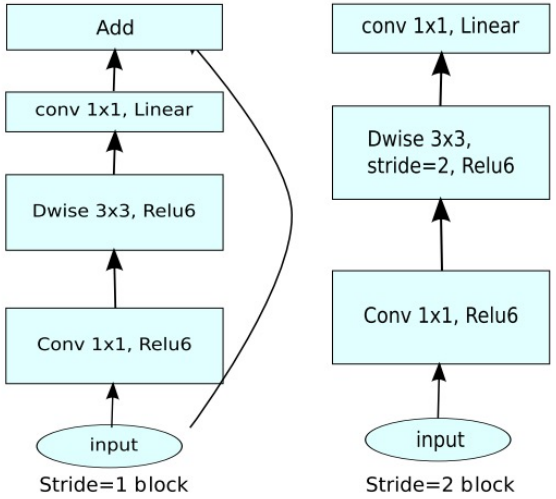
Input	Operator	Output
$h \times w \times k$	1x1 conv2d , ReLU6	$h \times w \times (tk)$
$h \times w \times tk$	3x3 dwise s=s, ReLU6	$\frac{h}{s} \times \frac{w}{s} \times (tk)$
$\frac{h}{s} \times \frac{w}{s} \times tk$	linear 1x1 conv2d	$\frac{h}{s} \times \frac{w}{s} \times k'$

T=6





(b) MobileNet[27]



(d) Mobilenet V2

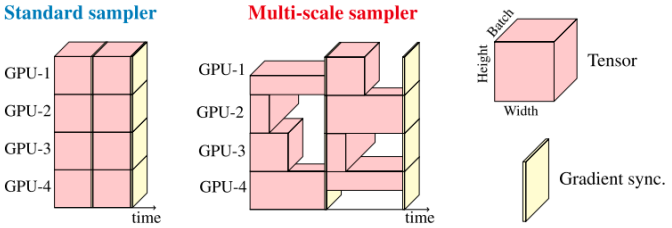
MobileNetV2: Inverted Residuals and Linear Bottlenecks, cited by 10K
https://gaussian37.github.io/dl-concept-mobilenet_v2/

Multi-Scale Sampler for Training Efficiency

However, most of these works sample a new spatial resolution after a fixed number of iterations.

The multi-scale sampler

- Reduces the training time as it requires fewer optimizer updates with variably-sized batches
- Improves performance by about 0.5%
- Forces the network to learn better multi-scale representations



(a) Standard vs. multi-scale sampler illustration

Sampler	# Updates	Epoch time
Standard	375 k	380 sec
Multi-scale (ours)	**232 k**	**270 sec**

(b) Training efficiency. Here, **standard** sampler refers to PyTorch's Distributed-DataParallel sampler.

Figure 5: **Multi-scale vs. standard sampler.**

Good point!
Not leaving other models for comparison

Model	# Params	# Epochs	# Updates ⇓	Top-1 accuracy ⇑		Training time ⇓	
ResNet-50 w/ standard sampler (PyTorch)	25 M	–	–	76.2	(0.0%)	–	–
ResNet-50 w/ standard sampler (our repro.)†	25 M	150	187 k	77.1	(+0.9%)	54 k sec.	(1.35×)
ResNet-50 w/ multi-scale sampler (Ours)†	25 M	150	116 k	**78.6**	(+2.4%)	40 k sec.	(1×)
MobileNetv2-1.0 w/ standard sampler (PyTorch)	3.5 M	–	–	71.9	(0.0%)	–	(–)
MobileNetv2-1.0 w/ standard sampler (our repro.)†	3.5 M	300	375 k	72.1	(+0.2%)	78 k sec.	(1.16×)
MobileNetv2-1.0 w/ multi-scale sampler (Ours)†	3.5 M	300	232 k	**73.3**	(+1.4%)	67 k sec.	(1×)

Table 5: **Multi-scale sampler is generic.** All models are trained with basic augmentation on the ImageNet-1k. †Results are with exponential moving average.

Implementation details

Imagenet-1k

- The dataset provides 1.28 million and 50 thousand images for training and validation, respectively.

The MobileViT networks are trained using PyTorch for 300 epochs on

- 8 NVIDIA GPUs with an effective
- batch size of 1,024 images using AdamW optimizer (Loshchilov & Hutter, 2019),
- label smoothing cross-entropy loss (smoothing=0.1), and
- multi-scale sampler (S = {(160, 160), (192, 192), (256, 256), (288, 288), (320, 320)}).

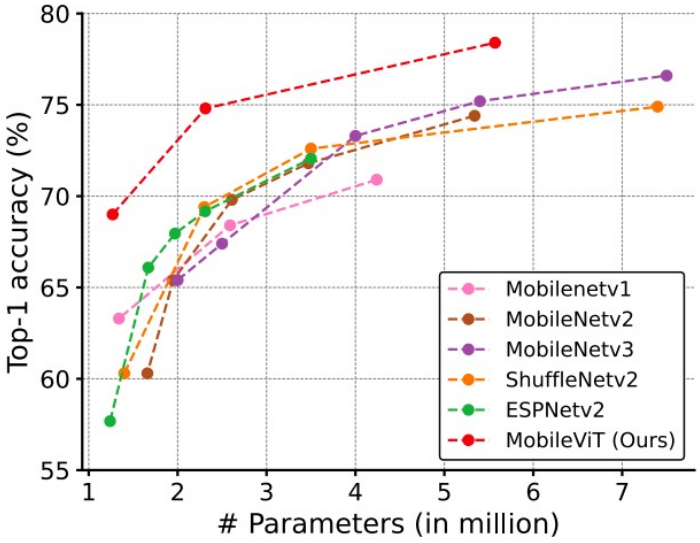
The learning rate is

- increased from 0.0002 to 0.002 for the first 3k iterations and then annealed to 0.0002 using a cosine schedule (Loshchilov & Hutter, 2017).
- We use L2 weight decay of 0.01.

We use basic data augmentation (i.e., random resized cropping and horizontal flipping) and evaluate the performance using a single crop top-1 accuracy.

For inference, an exponential moving average of model weights is used.

Comparison with CNNs



(a) Comparison with light-weight CNNs

Model	# Params. ⇓	Top-1 ⇑
MobileNetv1	2.6 M	68.4
MobileNetv2	2.6 M	69.8
MobileNetv3	2.5 M	67.4
ShuffleNetv2	2.3 M	69.4
ESPNetv2	2.3 M	69.2
MobileViT-XS (Ours)	2.3 M	**74.8**

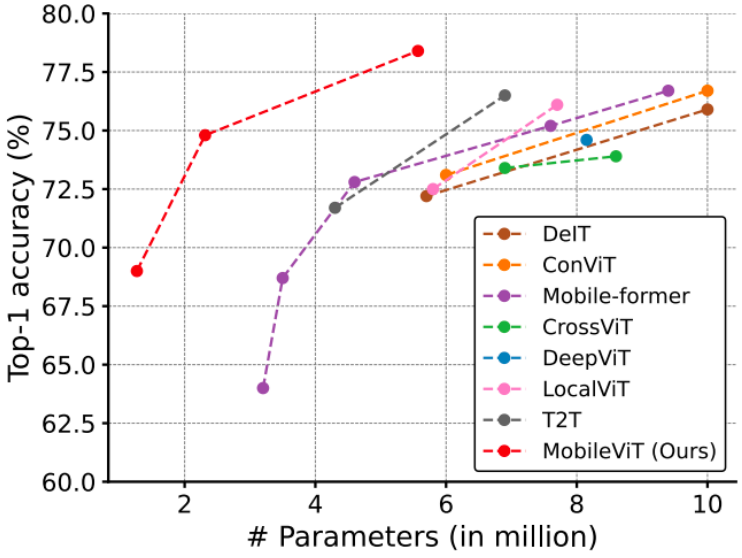
(b) Comparison with light-weight CNNs (similar parameters)

Model	# Params. ⇓	Top-1 ⇑
DenseNet-169	14 M	76.2
EfficientNet-B0	5.3 M	76.3
ResNet-101	44.5 M	77.4
ResNet-101-SE	49.3 M	77.6
MobileViT-S (Ours)	5.6 M	**78.4**

(c) Comparison with heavy-weight CNNs

Figure 6: **MobileViT vs. CNNs** on ImageNet-1k validation set. All models use basic augmentation.

Comparison with ViTs



(a)

Row #	Model	Augmentation	# Params. ⇓	Top-1 ⇑
R1	DeIT	Basic	5.7 M	68.7
R2	T2T	Advanced	4.3 M	71.7
R3	DeIT	Advanced	5.7 M	72.2
R4	PiT	Basic	10.6 M	72.4
R5	Mobile-former	Advanced	4.6 M	72.8
R6	PiT	Advanced	4.9 M	73.0
R7	CrossViT	Advanced	6.9 M	73.4
R8	MobileViT-XS (Ours)	Basic	2.3 M	**74.8**
R9	CeiT	Advanced	6.4 M	76.4
R10	DeIT	Advanced	10 M	75.9
R11	T2T	Advanced	6.9 M	76.5
R12	ViL	Advanced	6.7 M	76.7
R13	LocalVit	Advanced	7.7 M	76.1
R14	Mobile-former	Advanced	9.4 M	76.7
R15	PVT	Advanced	13.2 M	75.1
R16	ConViT	Advanced	10 M	76.7
R17	PiT	Advanced	10.6 M	78.1
R18	BoTNet	Basic	20.8 M	77.0
R19	BoTNet	Advanced	20.8 M	78.3
R20	MobileViT-S (Ours)	Basic	5.6 M	**78.4**

(b)

Figure 7: **MobileViT vs. ViTs** on ImageNet-1k validation set. Here, **basic** means ResNet-style augmentation while **advanced** means a combination of augmentation methods with basic (e.g., MixUp (Zhang et al., 2018), RandAugmentation (Cubuk et al., 2019), and CutMix (Zhong et al., 2020)).

Evaluation for the general-purpose nature

MobileViT is well performed on object detection and semantic segmentation.

MobileViT on Down-stream Tasks

Object Detection on MS-COCO using Single Shot Detector (SSD)

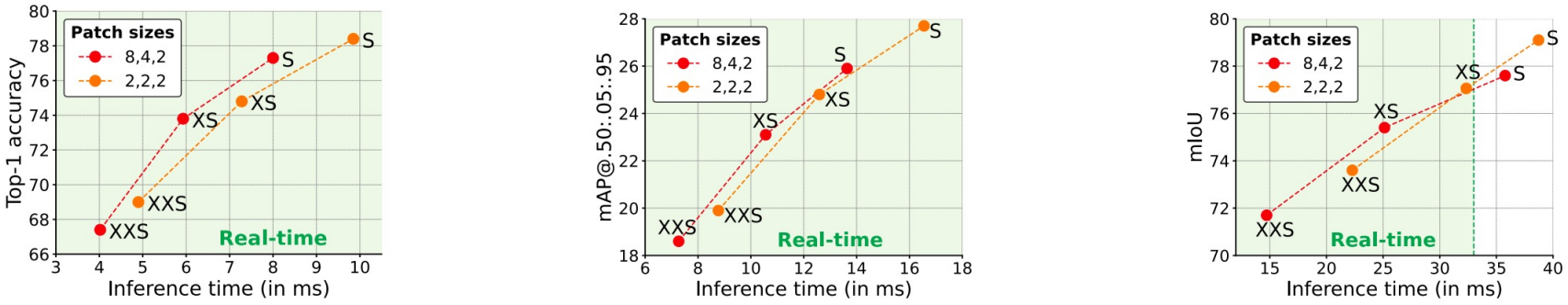
	Parameters	mAP
MNASNet	4.9 M	23.0
MobileViT	2.7 M	24.8

Semantic Segmentation on PASCAL VOC using DeepLabv3

	Parameters	mIOU
MobileNetv2	4.5 M	75.7
MobileViT	2.9 M	77.1

Performance on Mobile Devices

CoreML on iPhone 12



(a) Classification @ 256 × 256 (b) Detection @ 320 × 320 (c) Segmentation @ 512 × 512

Figure 8: **Inference time of MobileViT models on different tasks.** Here, dots in green color region represents that these models runs in real-time (inference time < 33 ms).

Mobile-friendly. Figure 8 shows the inference time of MobileViT networks with two patch size settings (Config-A: $2, 2, 2$ and Config-B: $8, 4, 2$) on three different tasks. Here p_1, p_2, p_3 in Config-X denotes the height h (width $w = h$) of a patch at an output stride[2] of 8, 16, and 32, respectively. The models with smaller patch sizes (Config-A) are more accurate as compared to larger patches (Config-B). This is because, unlike Config-A models, Config-B models are not able to encode the information from all pixels (Figure 13 and §C). On the other hand, for a given parameter budget, Config-B models are faster than Config-A even though the theoretical complexity of self-attention in both configurations is the same, i.e., $\mathcal{O}(N^2Pd)$. With larger patch sizes (e.g., P=8^2=64), we have fewer number of patches N as compared to smaller patch sizes (e.g., P=2^2=4). As a result, the computation cost of self-attention is relatively less. Also, Config-B models offer a higher degree of parallelism as compared to Config-A because self-attention can be computed simultaneously for more pixels in a larger patch (P=64) as compared to a smaller patch (P=4). Hence, Config-B models are faster than Config-A. To further improve MobileViT's latency, linear self-attention (Wang et al., 2020) can be used. Regardless, all models in both configurations run in real-time (inference speed $\geq$ 30 FPS) on a mobile device except for MobileViT-S models for the segmentation task. This is expected as these models process larger inputs (512×512) as compared to classification (256×256) and detection (320×320) networks.

Discussion

Dedicated CUDA kernels exist for transformers on GPU

CNNs benefit from several device-level optimizations, including batch normalization fusion with convolutional layers

- Not available for mobile devices
- The resultant inference graph of MobileViT and ViT-based networks for mobile devices is sub-optimal.

Model	# Params. ⇓	FLOPs ⇓	Time ⇓	Top-1 ⇑
MobileNetv2†	3.5 M	**0.3 G**	**0.92 ms**	73.3
DeIT	5.7 M	1.3 G	10.99 ms	72.2
PiT	4.9 M	0.7 G	10.56 ms	73.0
MobileViT (Ours)	**2.3 M**	0.7 G	7.28 ms	**74.8**

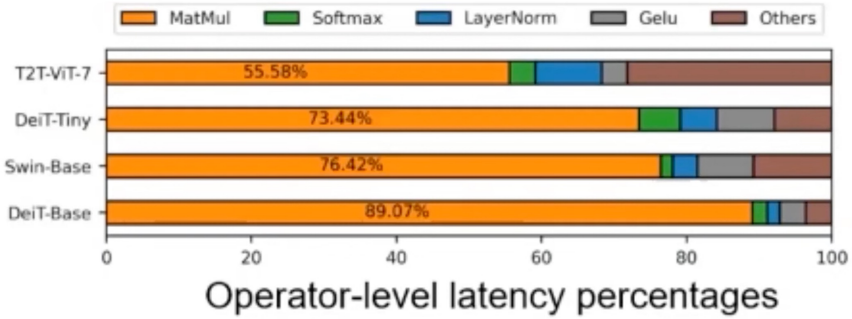
Table 3: **ViTs are slower than CNNs.** †Results with multi-scale sampler (§B).

A systematic measurement

- Latency bottlenecks analysis:
  - Layer-level and operator-layer

Model	Pre/post - processing	Attention	FFN
DeiT-Base	0.92%	37.92%	61.16%
Swin-Base	6.41%	34.58%	59.01%
DeiT-Tiny	3.6%	44.76%	51.64%
T2T-ViT-7	44.36%	33.59%	22.06%

Table 3: Layer-level latency percentages on Pixel 4.



- Findings:
  - Compared to Attention layers, FFN layers are more time-consuming
  - The large number of computation-intensive operators (the MatMul) are the latency bottlenecks
  - Non-computation-intensive operators (SoftMax, LayerNorm, Gelu) takes up a significant percentage (19%) of latency in lite transformers.

Towards Efficient Vision Transformer Inference: A First Study of Transformers on Mobile Devices, HotMobile22

Source code

Conclusions

- Proposed a hybrid model for learning visual representations
- Runs in real-time on mobile devices (iPhone12)
- State-of-the-art performance across different tasks and datasets

Source Code:

CVNet
(PyTorch, Ours)

Keras/Tensorflow
(Third-party)

Kornia
(Third-party)

& many more...

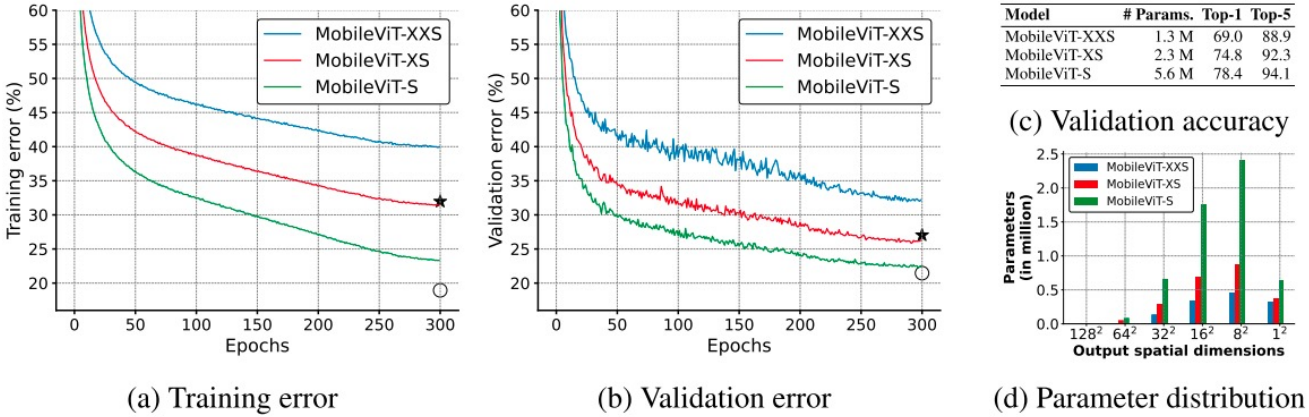
CVNet v0.1

CUDA_VISIBLE_DEVICES=0 cvnets-eval --common.config-file $CFG_FILE/mobilevit_xxs.yaml --common.results-loc classification_results --model.classification.pretrained $MODEL_WEIGHTS/mobilevit_xxs.pt

```
2022-07-27 10:06:14 - LOGS    - Model statistics for an input of size torch.Size([1, 3, 256, 256])
================================================================
                        MobileViT Summary
================================================================
ConvLayer               Params:    0.000 M      MACs :    7.078 M
----------------------------------------------------------------
InvertedResidual        Params:    0.001 M      MACs :   21.496 M
----------------------------------------------------------------
InvertedResidual
+InvertedResidual
+InvertedResidual       Params:    0.008 M      MACs :   35.127 M
----------------------------------------------------------------
InvertedResidual
+MobileViTBlock         Params:    0.140 M      MACs :  162.316 M
----------------------------------------------------------------
InvertedResidual
+MobileViTBlock         Params:    0.342 M      MACs :  103.555 M
----------------------------------------------------------------
InvertedResidual
+MobileViTBlock         Params:    0.433 M      MACs :   32.692 M
----------------------------------------------------------------
ConvLayer               Params:    0.026 M      MACs :    1.638 M
----------------------------------------------------------------
Classifier              Params:    0.321 M      MACs :    0.321 M
================================================================
Overall parameters   =    1.272 M
Overall MACs         =  364.224 M
Overall parameters (sanity check) =    1.272 M
================================================================
2022-07-27 10:06:16 - LOGS    - Epoch:  -1 [     100/   50000], top1: 87.0000, top5: 94.0000, LR: 0.000000, Avg. batch load time: 0.000, Elapsed time:  1.95
ookmarks Alt+2  10:07:29 - LOGS    - *** Evaluation summary for epoch -1
         top1=69.0000 || top5=88.8940
2022-07-27 10:07:29 - LOGS    - Evaluation took 74.9268114566803 seconds
```

Reproduce accuracy with Timm

Paper



Model	# Params.	Top-1	Top-5
MobileViT-XXS	1.3 M	69.0	88.9
MobileViT-XS	2.3 M	74.8	92.3
MobileViT-S	5.6 M	78.4	94.1

(c) Validation accuracy

(a) Training error (b) Validation error (d) Parameter distribution

Figure 3: **MobileViT shows similar generalization capabilities as CNNs**. Final training and validation errors of MobileNetv2 and ResNet-50 are marked with ⋆ and ○, respectively (§B).

Github CVNet: Readme.md

MobileViTv1 (Legacy)

Note: These resutls are from CVNets v0.1. We discontinued the support of OpenCV a
v0.1.

Model	Parameters	Top-1	Pretrained weights	Config file
MobileViT-XXS	1.3 M	69.0	Link	Link
MobileViT-XS	2.3 M	74.7	Link	Link
MobileViT-S	5.6 M	78.3	Link	Link

Timm

- XXS: "top1": 68.906, 0.107s
- XS: "top1": 74.634, 0.178s
- S: "top1": 78.316, 0.219s

Comparison on latency with timm libary

```
Test: [ 190/196]  Time: 0.458s (0.467s,  548.06/s)  Loss:  0.8204 (0.5575)  Acc@1:  74.219 ( 84.436)  Acc@5:  97.656 ( 97.272)
 * Acc@1 84.532 (15.468) Acc@5 97.296 (2.704)
--result
{
    "model": "vit_base_patch16_224",
    "top1": 84.532,
    "top1_err": 15.468,
    "top5": 97.296,
    "top5_err": 2.704,
    "param_count": 86.57,
    "img_size": 224,
    "crop_pct": 0.9,
    "interpolation": "bicubic"
}
```

/ssd/dataset/imagenet –model vit_base_patch16_224 --pretrained

```
Test: [ 190/196]  Time: 0.108s (0.409s,  625.43/s)  Loss:  1.4420 (1.3398)  Acc@1:  58.594 ( 68.707)  Acc@5:  91.016 ( 88.878)
 * Acc@1 68.906 (31.094) Acc@5 88.946 (11.054)
--result
{
    "model": "mobilevit_xxs",
    "top1": 68.906,
    "top1_err": 31.094,
    "top5": 88.946,
    "top5_err": 11.054,
    "param_count": 1.27,
    "img_size": 256,
    "crop_pct": 0.9,
    "interpolation": "bicubic"
}
```

/ssd/dataset/imagenet –model mobilevit_xxs --pretrained

```
Test: [ 190/196]  Time: 0.221s (0.416s,  615.52/s)  Loss:  1.2057 (0.9181)  Acc@1:  69.141 ( 78.215)  Acc@5:  95.312 ( 94.118)
 * Acc@1 78.316 (21.684) Acc@5 94.154 (5.846)
--result
{
    "model": "mobilevit_s",
    "top1": 78.316,
    "top1_err": 21.684,
    "top5": 94.154,
    "top5_err": 5.846,
    "param_count": 5.58,
    "img_size": 256,
    "crop_pct": 0.9,
    "interpolation": "bicubic"
}
```

/ssd/dataset/imagenet –model mobilevit_s --pretrained