Everything is All You Need: Vision Architectures for the 2020s

Saining Xie

Research Scientist, FAIR

Assistant Professor, Courant Institute of Mathematical Science, NYU (Starting from Jan 2023)

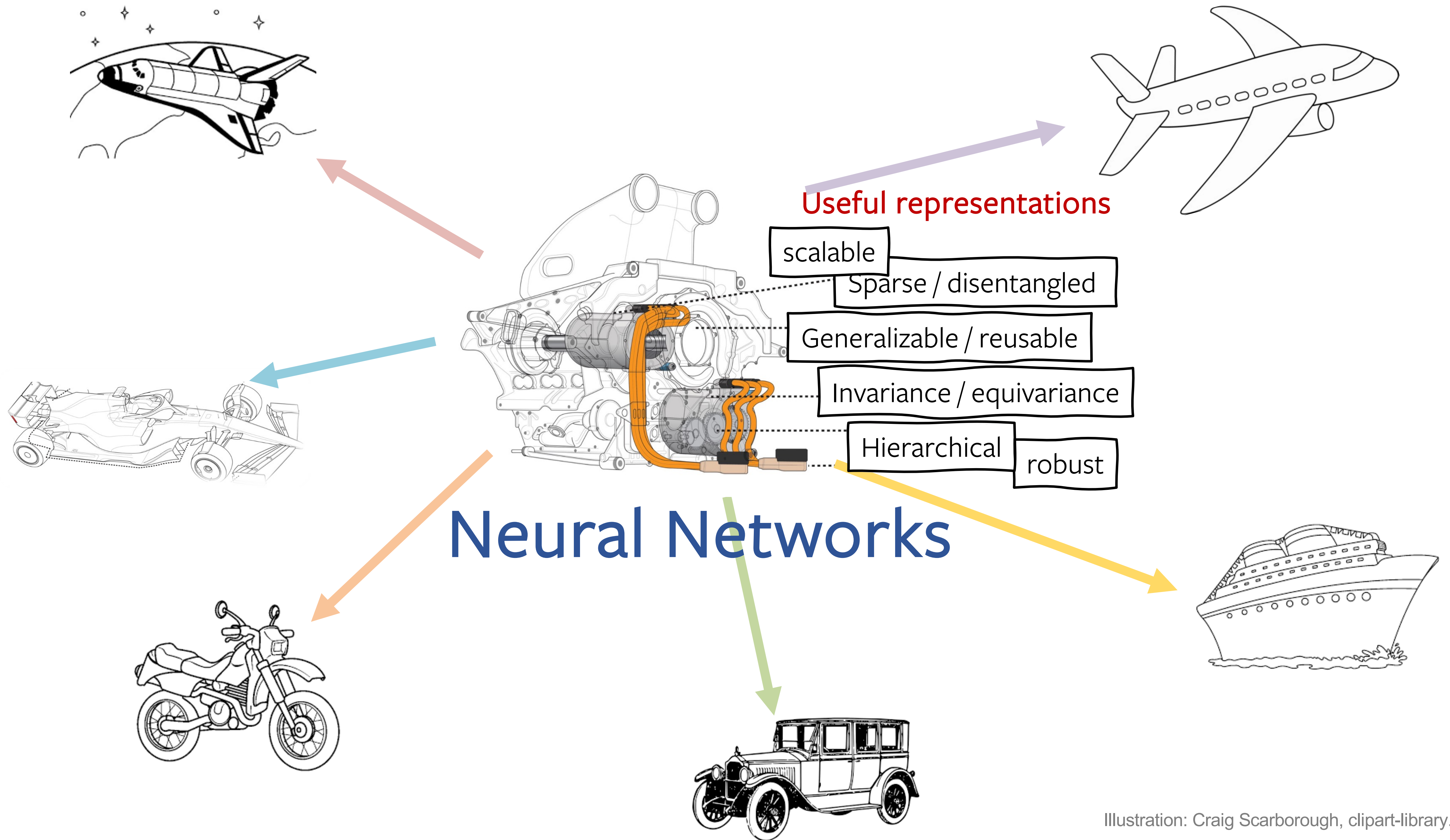


Illustration: Craig Scarborough, clipart-library.com

Convolutional Neural Networks

[Learning Internal Representations by Error Propagation. Rumelhart et al., 1986]

ConvNet using BP

- Receptive field
- Translation equivariance
- Trained by error propagation

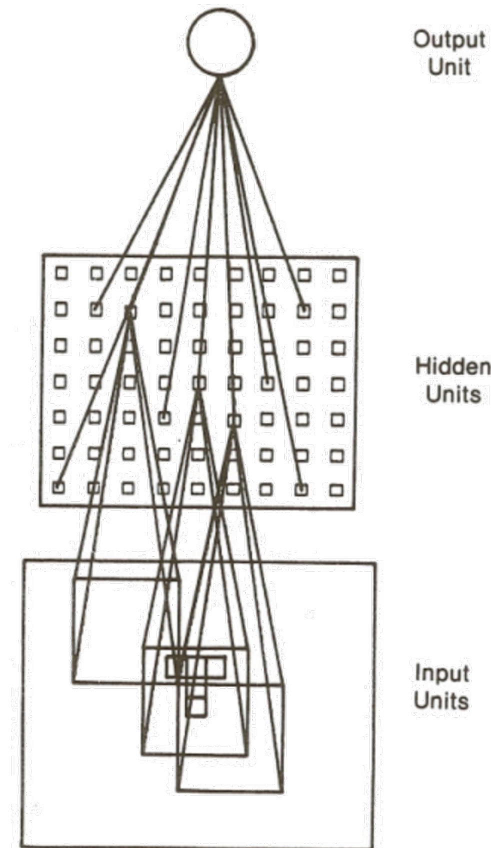
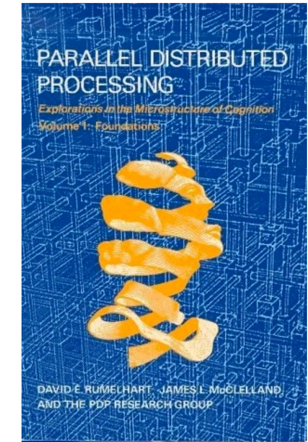


FIGURE 14. The network for solving the T-C problem. See text for explanation.



PARALLEL DISTRIBUTED PROCESSING

Explorations in the Microstructure of Cognition

Volume 1: Foundations

David E. Rumelhart James L. McClelland
and the PDP Research Group

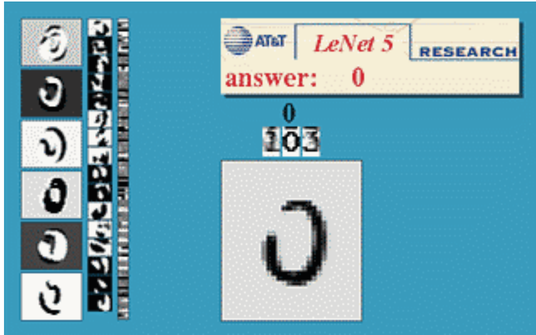
Chisato Asanuma, Francis H. C. Crick, Jeffrey L. Elman, Geoffrey E. Hinton, Michael I. Jordan, Alan H. Kawamoto, Paul W. Munro, Donald A. Norman, Daniel E. Rabin, Terrence J. Sejnowski, Paul Smolensky, Gregory O. Stone, Ronald J. Williams, David Zipser

Institute for Cognitive Science
University of California, San Diego

1986

LeNet

[Backpropagation Applied to Handwritten Zip Code Recognition, LeCun et al., 1989]



INPUT
32x32

C1: feature maps
6@28x28

S2: f. maps
6@14x14

C3: f. maps 16@10x10

S4: f. maps 16@5x5

C5: layer
120

F6: layer
84

OUTPUT
10

Convolutions

Subsampling

Convolutions

Subsampling

Full connection

Full connection

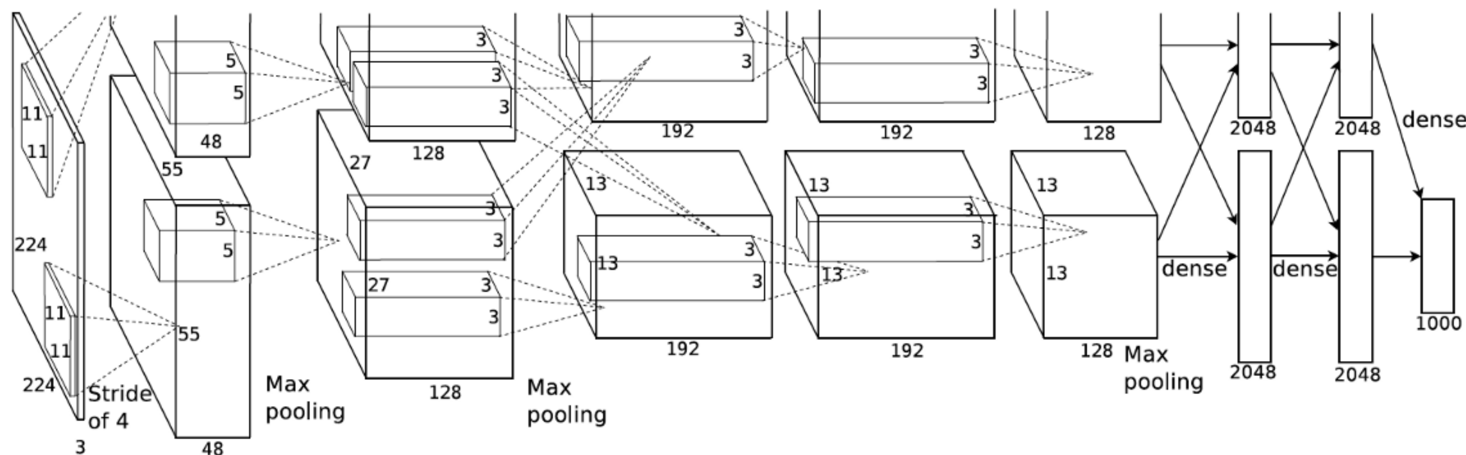
Gaussian connections

LeNet

1989

AlexNet

[Krizhevsky, Sutskever and Hinton, 2012]



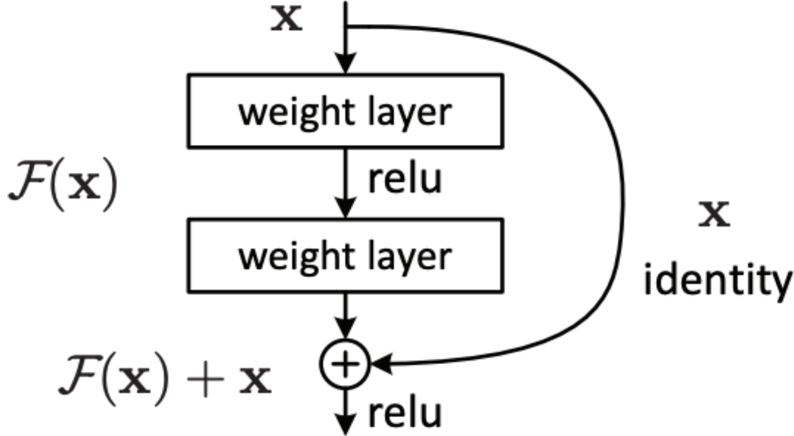
[Deng et al., 2009]
[Russakovsky et al., 2015]

AlexNet

2012

ResNet

[He et al., 2015]



repeating motif: a residual block

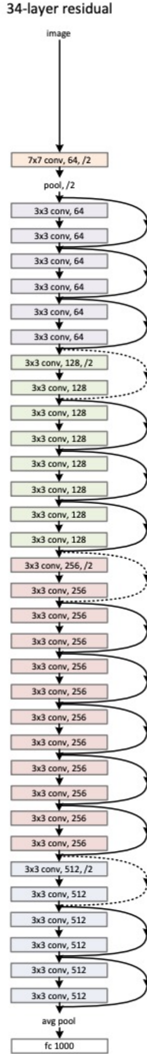


figure: reddit?

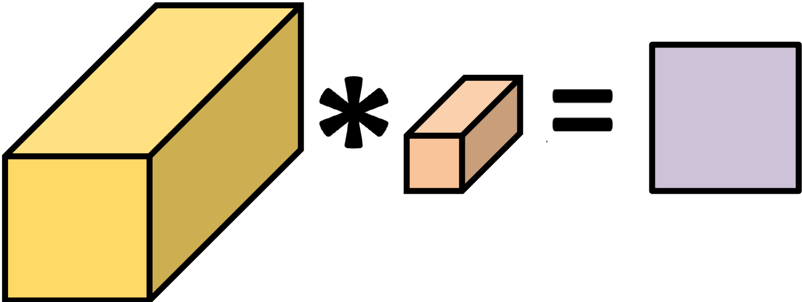
ResNet

2015

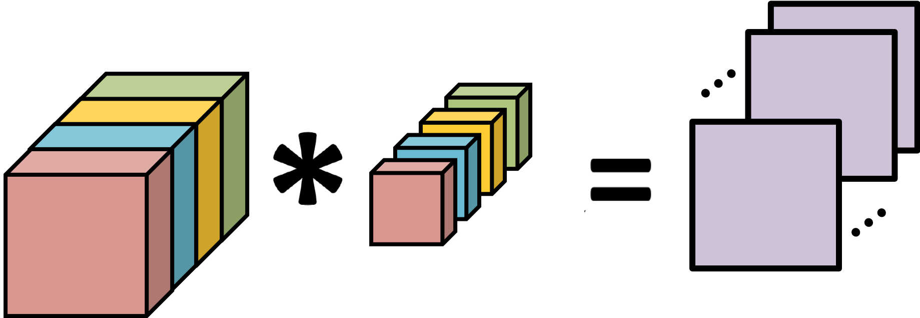
ResNeXt

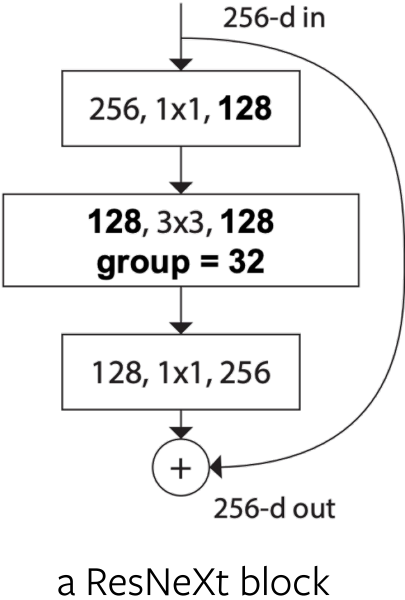
[Xie, Girshick, Dollár, Tu, He, CVPR 2017]

Dense Conv

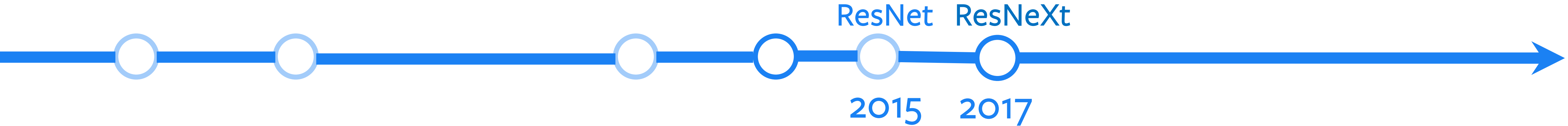


Grouped Conv





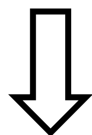
a ResNeXt block



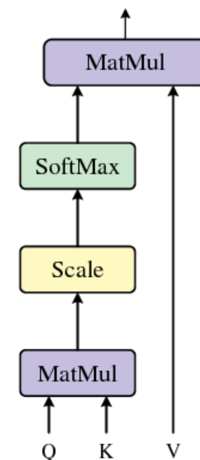
Vision Transformer (ViT)

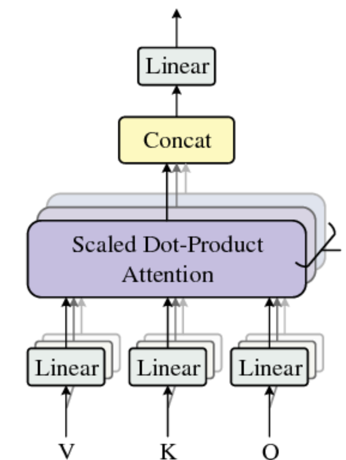
[Dosovitskiy et al., ICLR 2021]

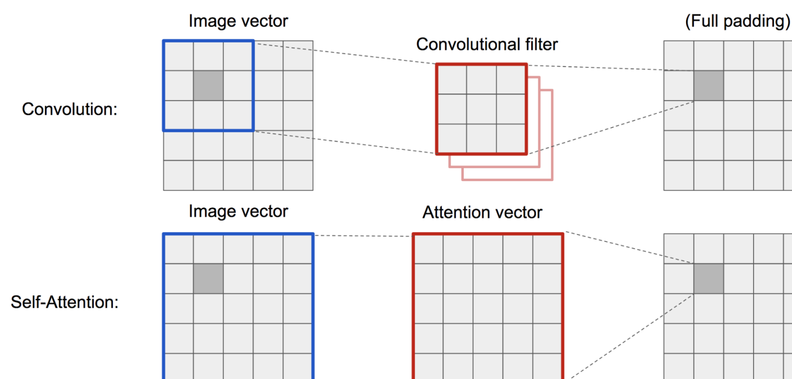
- Self-attention:
    - Less inductive bias



- Scalable
    - w/ bigger model
    - w/ larger data







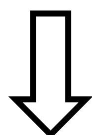
courtesy: Lilian Weng

Vision is NOT just about classification

- Expanding vision capabilities
  - Large resolution
  - Multi-scale
- Vision Transformer
  - Quadratic complexity
  - No hierarchy

Advanced Vision Transformers

- Self-attention:
  - Less inductive bias



- Scalable
  - w/ bigger model
  - w/ larger data

Self-attention (ViT)

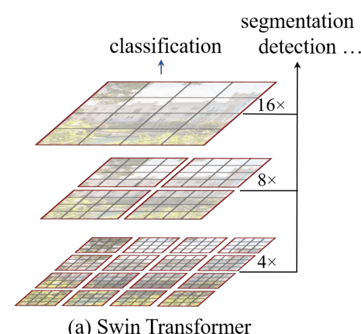
+ Conv Priors

Translation equivariance

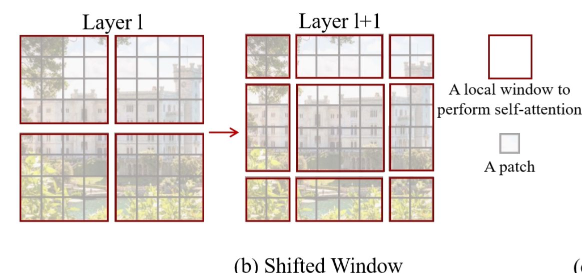
Locality

Hierarchical

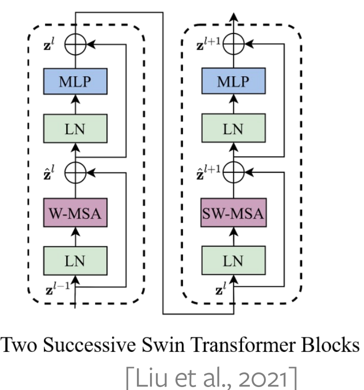
More complicated designs + Specialized modules



(a) Swin Transformer



(b) Shifted Window



(c) Two Successive Swin Transformer Blocks

[Liu et al., 2021]

Swin Transformer

2021

ConvNet losing steam?

Venue	Convolution, CNN, ConvNet	Attention, "-Former"
ECCV 2020	56	54
CVPR 2021	49	78
ICCV 2021	44	176
CVPR 2022	44	263

Swin Transformer

State of the Art | Object Detection on COCO test-dev (using additional training data)

State of the Art | Instance Segmentation on COCO test-dev

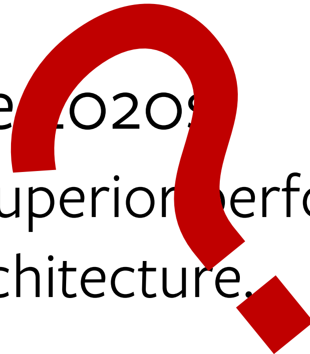
State of the Art | Semantic Segmentation on ADE20K (using additional training data)

Ranked #4 | Action Classification on Kinetics-400 (using additional training data)

Swin Transformers is a *hybrid* architecture

- Similarity:
  - Convolution inductive bias
- Difference:
  - "Core" component (attention vs. convolution)
  - Training procedures
  - Macro and micro architecture design decisions

- Common assumption in the 2020s
  - **Self-attention** is the key for superior performance and scalability.
  - ConvNet is NOT a scalable architecture.

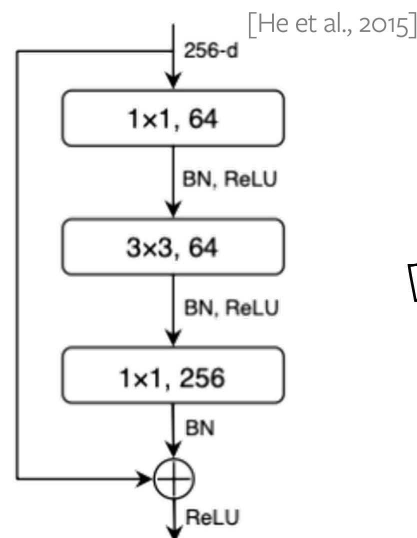


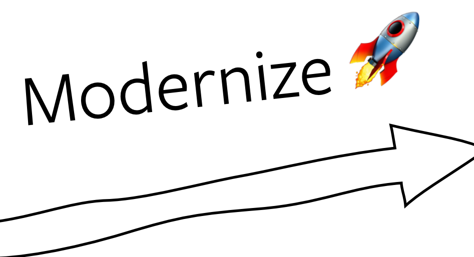
A ConvNet for the 2020s

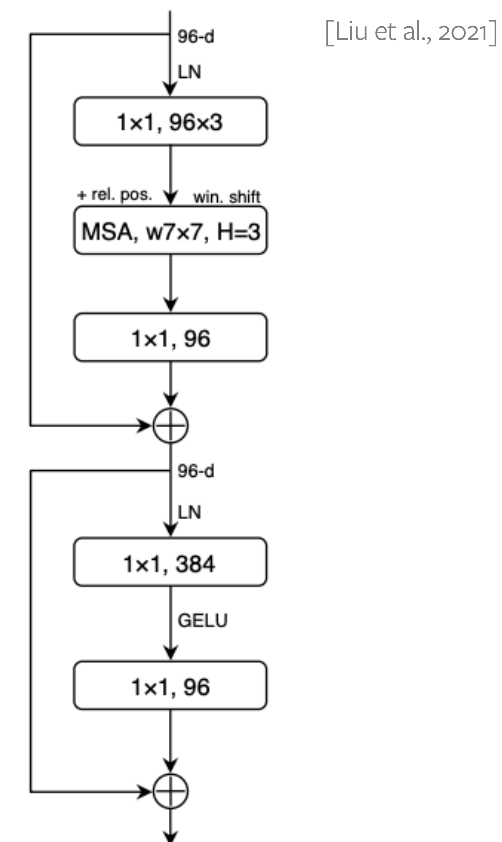
[Liu, Mao, Wu, Feichtenhofer, Darrell, **Xie.** CVPR 2022]

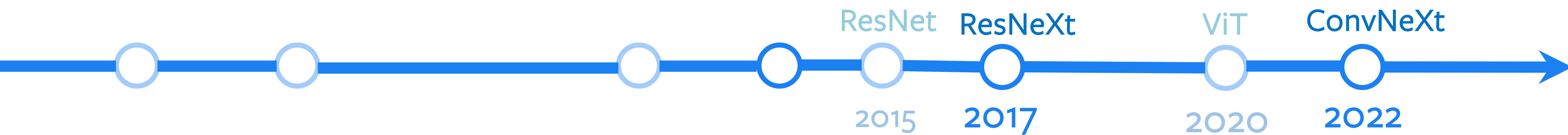
Central question:

- How do the design choices in Transformers impact a ConvNet's performance?









Improved training recipe

("DeiT Recipe") [Touvron et al., 2021]

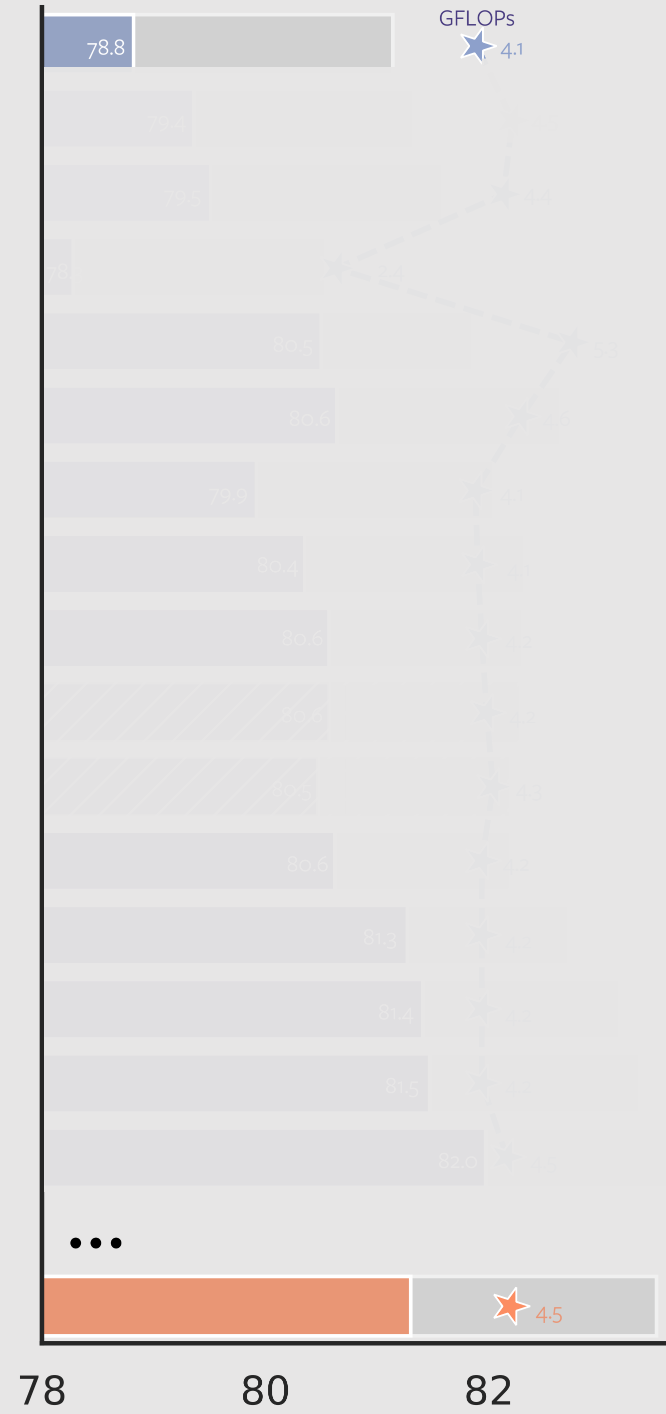
Typical Vision Transformer Training Recipe

Typical ResNet Training Recipe

ResNet-50 ImageNet top-1: 76.7% -> 78.8% 🚀

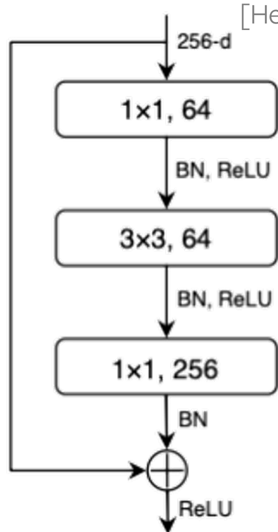
[Revisiting ResNets: Improved Training and Scaling Strategies, Bello et al, 2021]

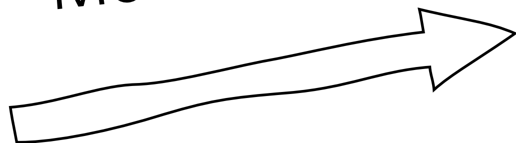
[ResNet strikes back: An improved training procedure in timm Wightman, et al, 2021]



ResNet

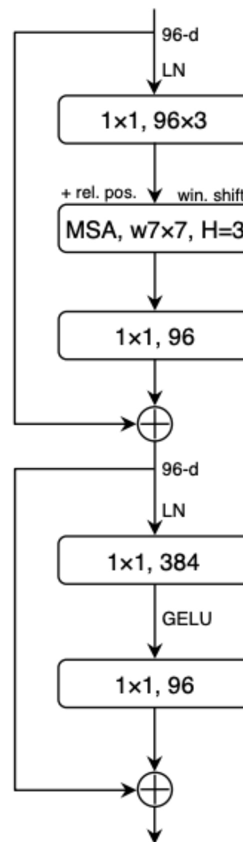
[He et al., 2015]





SwinTransformer

[Liu et al., 2021]



ResNet-50/200

Macro Design

stage ratio

"patchify" stem

Swin-T/B

ImageNet Top1 Acc (%)

78 80 82

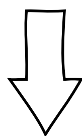
Input Stem (ResNet)

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
		3×3 max pool, stride 2				
conv2_x	56×56	[3×3, 64 3×3, 64] ×2	[3×3, 64 3×3, 64] ×3	[1×1, 64 3×3, 64 1×1, 256] ×3	[1×1, 64 3×3, 64 1×1, 256] ×3	[1×1, 64 3×3, 64 1×1, 256] ×3
conv3_x	28×28	[3×3, 128 3×3, 128] ×2	[3×3, 128 3×3, 128] ×4	[1×1, 128 3×3, 128 1×1, 512] ×4	[1×1, 128 3×3, 128 1×1, 512] ×4	[1×1, 128 3×3, 128 1×1, 512] ×8
conv4_x	14×14	[3×3, 256 3×3, 256] ×2	[3×3, 256 3×3, 256] ×6	[1×1, 256 3×3, 256 1×1, 1024] ×6	[1×1, 256 3×3, 256 1×1, 1024] ×23	[1×1, 256 3×3, 256 1×1, 1024] ×36
conv5_x	7×7	[3×3, 512 3×3, 512] ×2	[3×3, 512 3×3, 512] ×3	[1×1, 512 3×3, 512 1×1, 2048] ×3	[1×1, 512 3×3, 512 1×1, 2048] ×3	[1×1, 512 3×3, 512 1×1, 2048] ×3
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

ResNet-50/200
GFLOPs
78.8
4.1
Macro Design
stage ratio
79.4
4.5
"patchify" stem
79.5
4.4
...
Swin-T/B
81.3
4.5
ImageNet Top1 Acc (%)
78
80
82

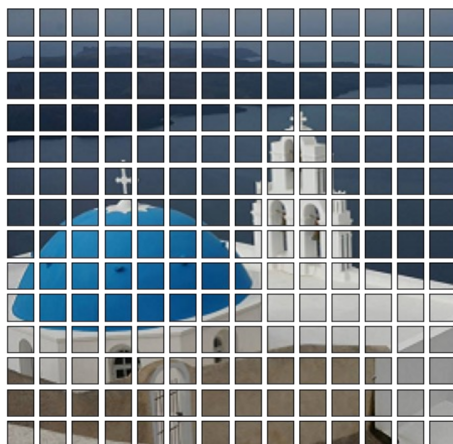
Input Stem

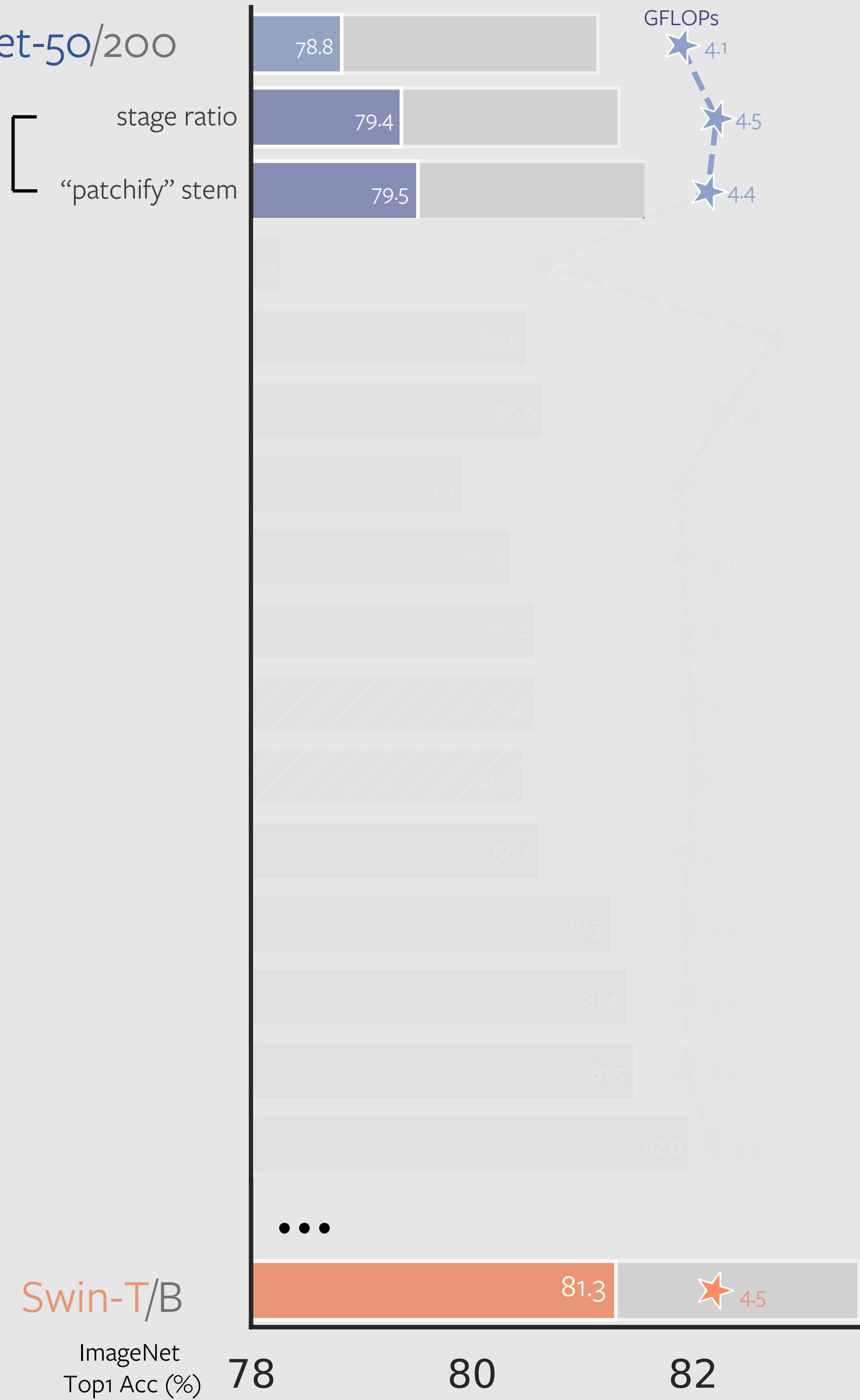
Overlapping Conv + Max Pooling



Non-overlapping Conv (4x4, stride 4)

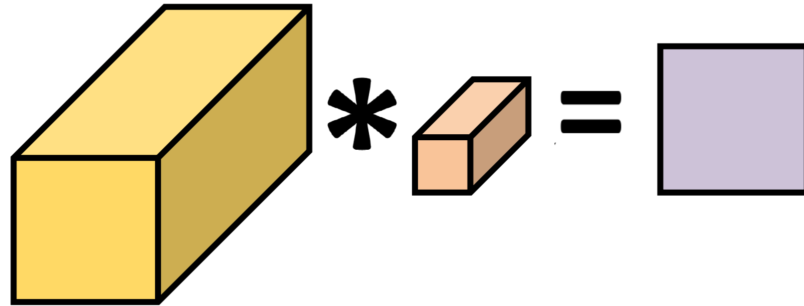
(a.k.a Patchify)



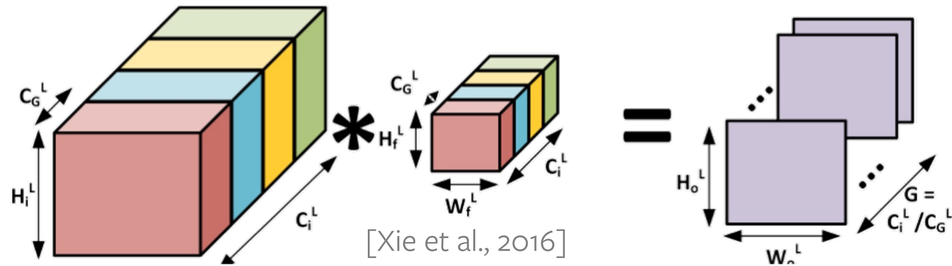


ResNeXt-ify

Dense Conv

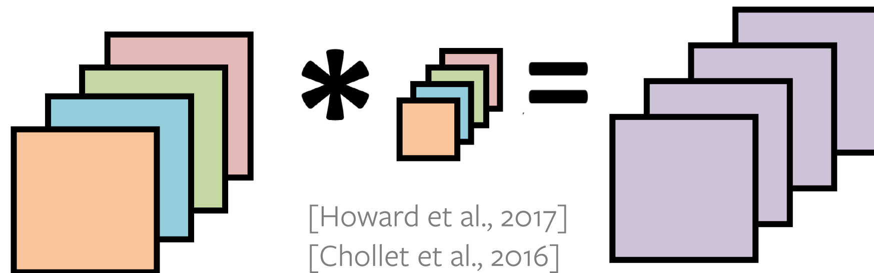


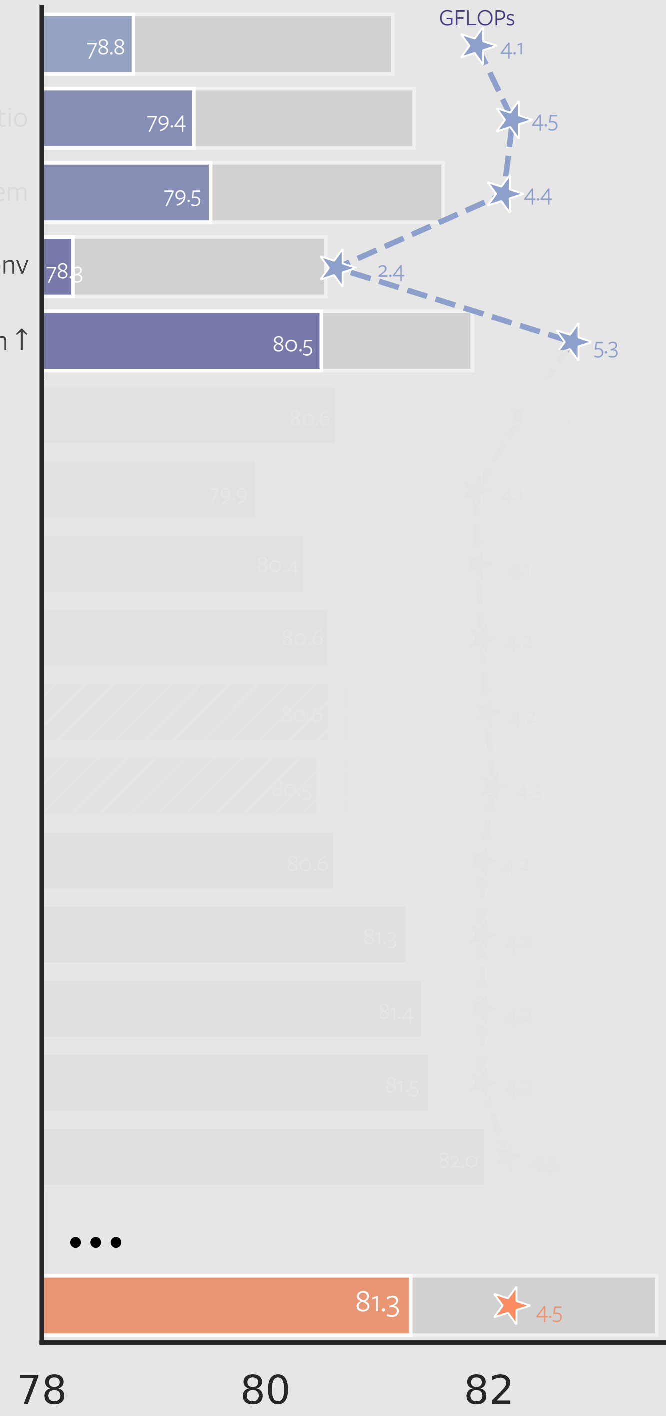
Grouped Conv



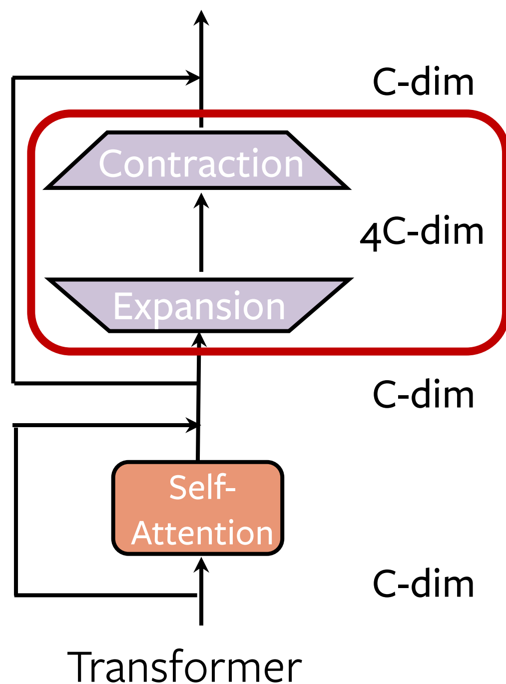
Depthwise Conv

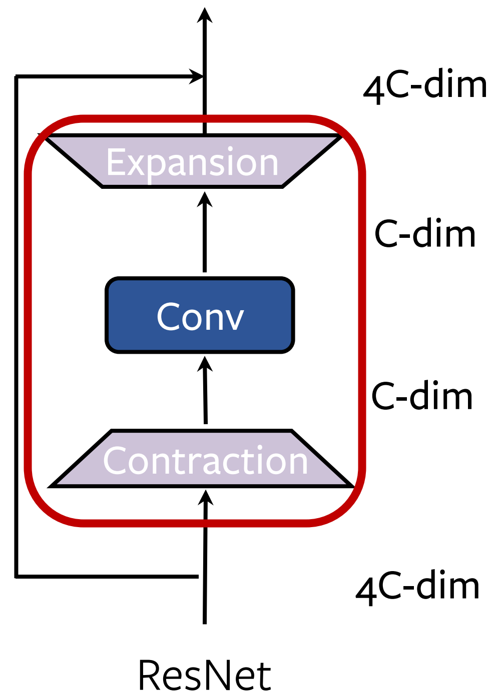
groups = # channels





Inverted Bottleneck





ResNet-50/200

GFLOPs

Macro Design

stage ratio 79.4 4.5

"patchify" stem 79.5 4.4

ResNeXt

depth conv 78.3 2.4

width ↑ 80.5 5.3

Inverted Bottleneck

inverting dims 80.6 4.6

78.8 4.1

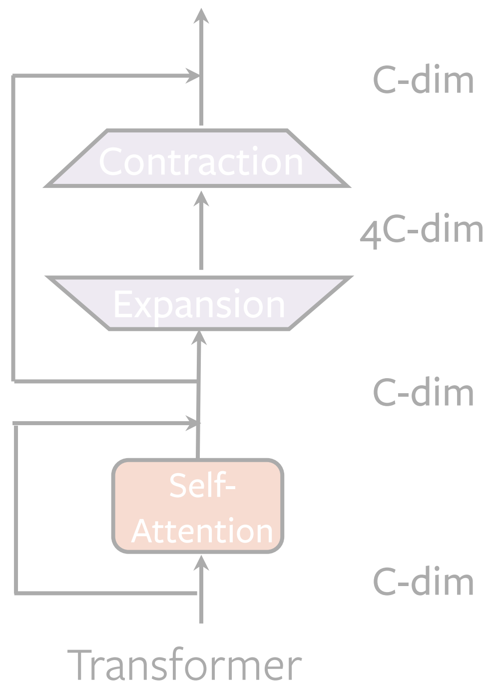
...

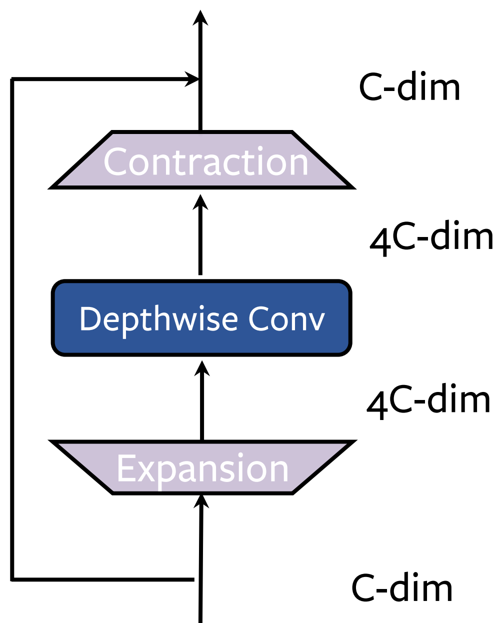
Swin-T/B 81.3 4.5

ImageNet Top1 Acc (%)

78 80 82

Inverted Bottleneck





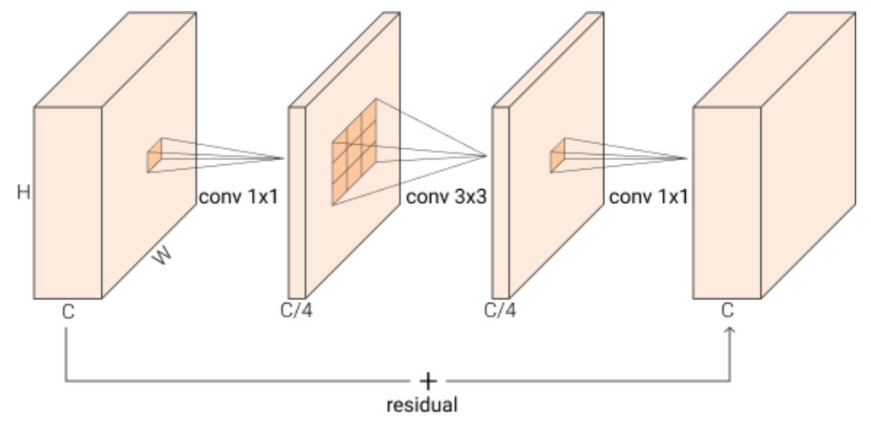
ResNet-50/200

GFLOPs

		Top1 Acc	GFLOPs
ResNet-50/200		78.8	4.1
Macro Design	stage ratio	79.4	4.5
	"patchify" stem	79.5	4.4
ResNeXt	depth conv	78.3	2.4
	width ↑	80.5	5.3
Inverted Bottleneck	inverting dims	80.6	4.6
...			
Swin-T/B		81.3	4.5

ImageNet Top1 Acc (%) 78 80 82

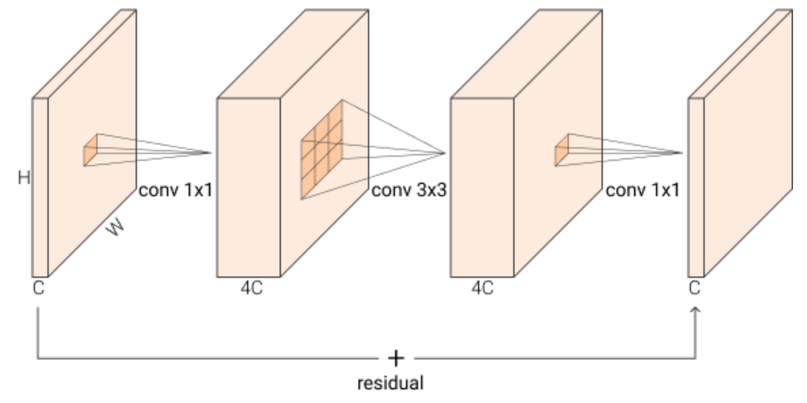
2016 ResNet

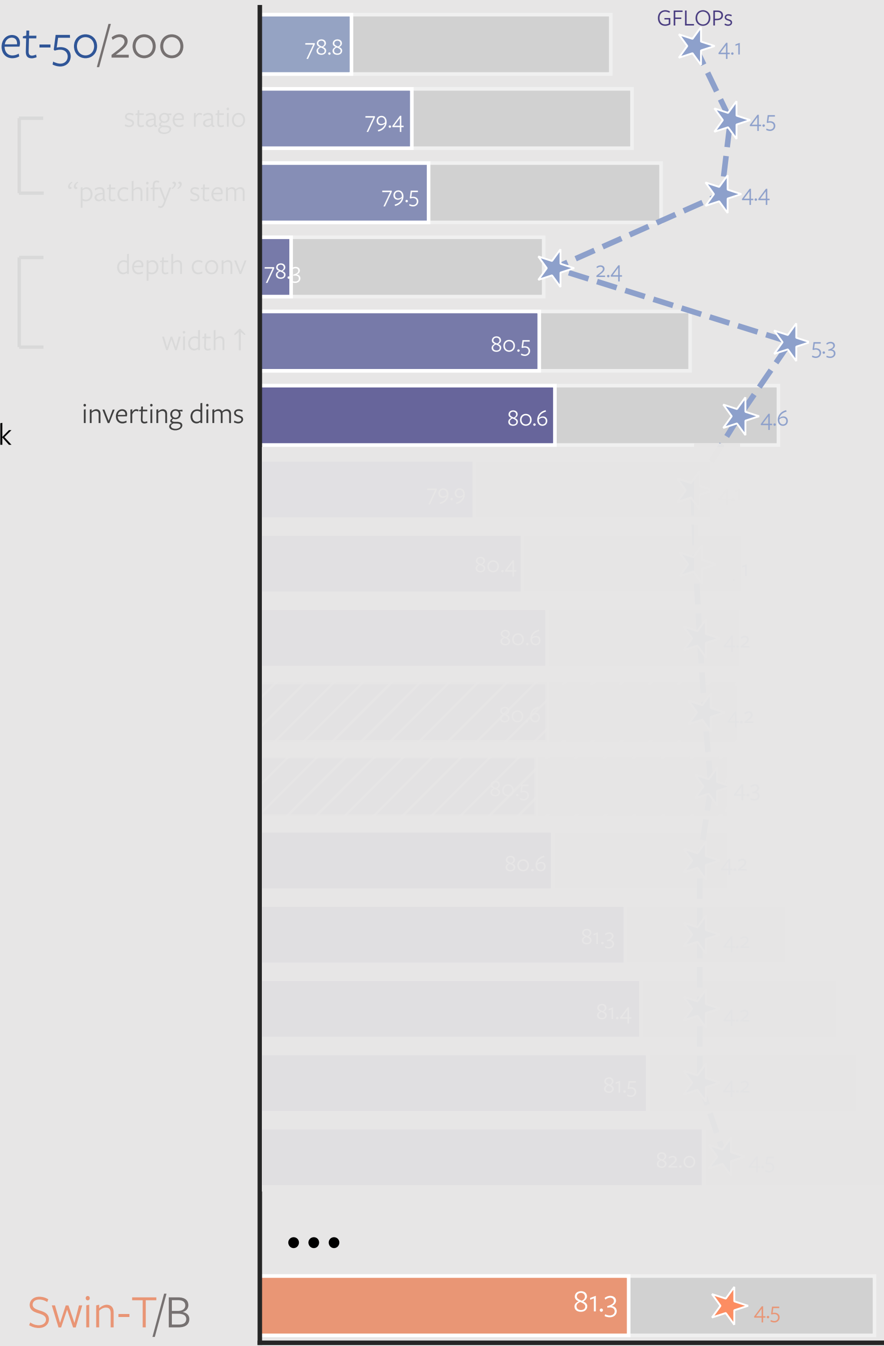


2018 Inverted Bottleneck

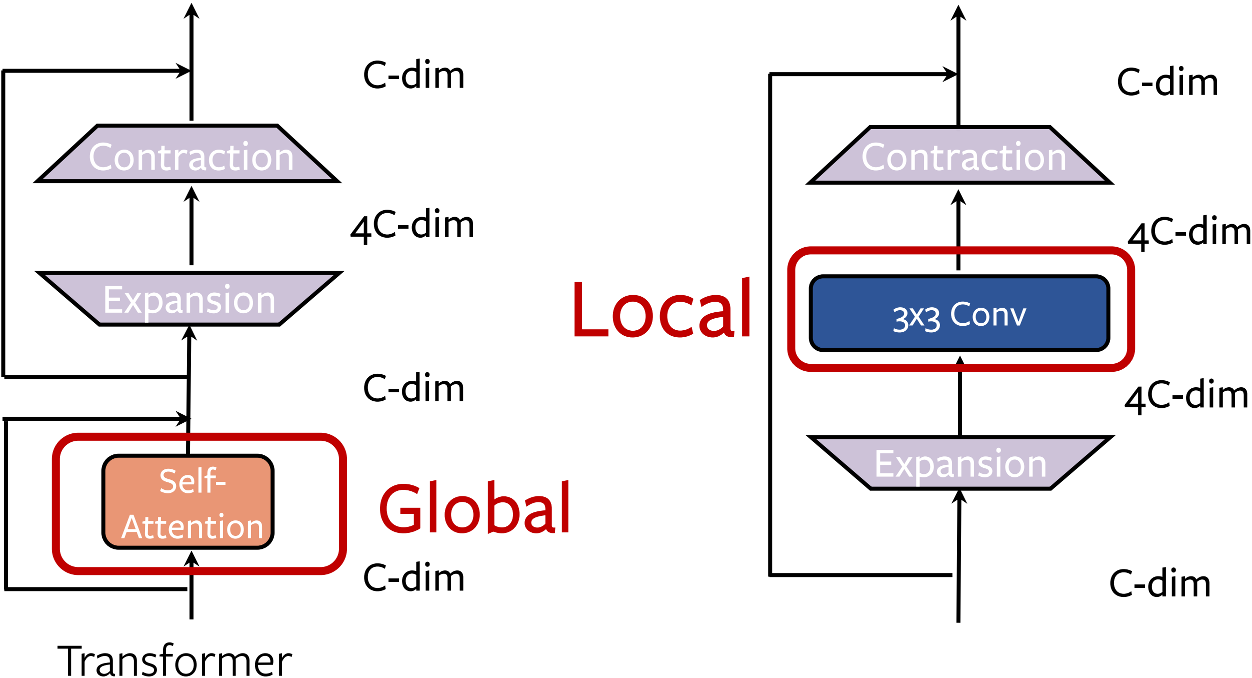
MobileNet v2

[Sandler et al, 2018]



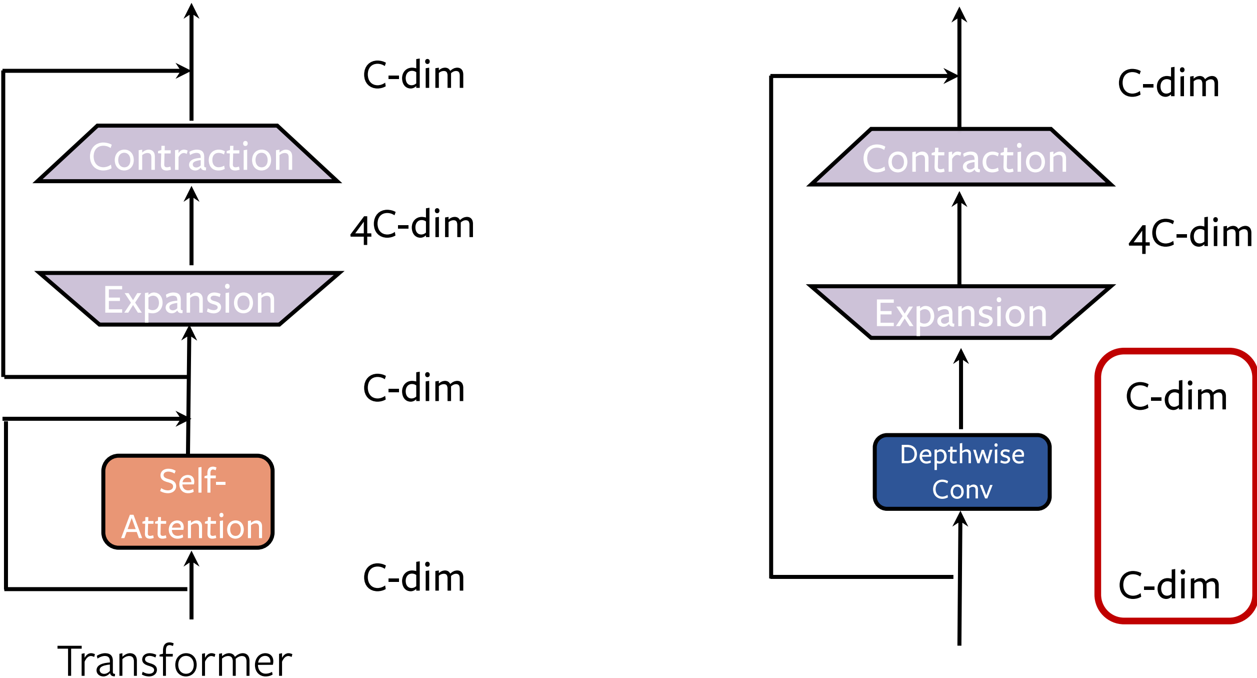


Large kernel size



ResNet-50/200
GFLOPs
Macro Design
stage ratio
"patchify" stem
ResNeXt
depth conv
width ↑
Inverted Bottleneck
inverting dims
78.8 — 4.1
79.4 — 4.5
79.5 — 4.4
78.3 — 2.4
80.5 — 5.3
80.6 — 4.6
...
Swin-T/B
81.3 — 4.5
ImageNet Top1 Acc (%)
78
80
82

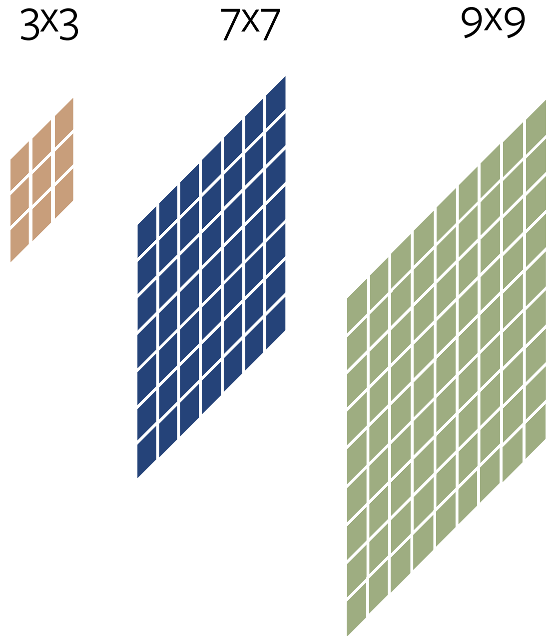
Large kernel size

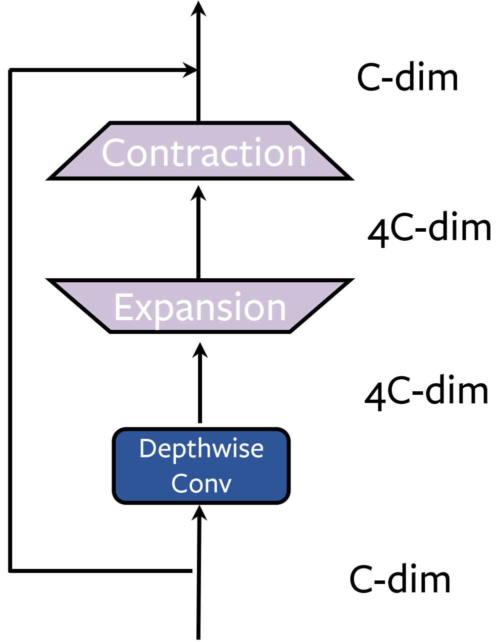


Prerequisite: move depth-wise before "expansion".
Reduce flops w/ larger kernel size.

ResNet-50/200
GFLOPs
Macro Design
stage ratio
"patchify" stem
ResNeXt
depth conv
width ↑
Inverted Bottleneck
inverting dims
move ↑ d. conv
78.8
79.4
79.5
78.3
80.5
80.6
79.9
4.1
4.5
4.4
2.4
5.3
4.6
4.1
...
Swin-T/B
81.3
4.5
ImageNet Top1 Acc (%)
78
80
82

Large kernel size





Larger kernel size helps; performance saturates at 7x7
Swin's choice of local window size is also 7 🤔

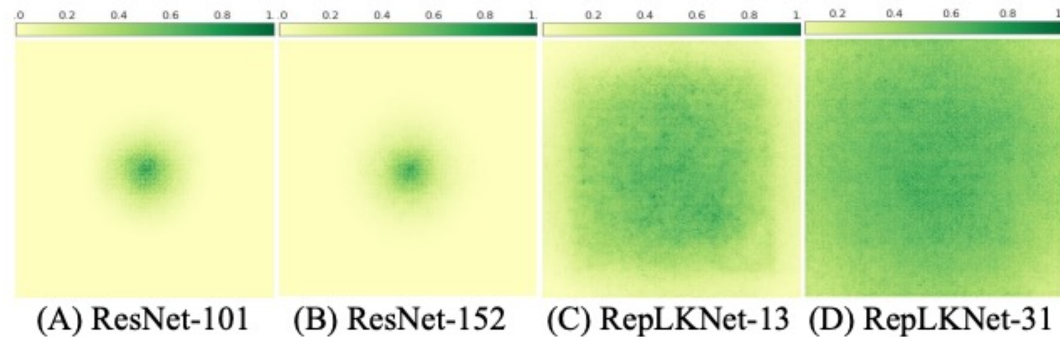
ResNet-50/200

Group	Step	ImageNet Top1 Acc (%)	GFLOPs
	ResNet-50/200	78.8	4.1
Macro Design	stage ratio	79.4	4.5
	"patchify" stem	79.5	4.4
ResNeXt	depth conv	78.3	2.4
	width ↑	80.5	5.3
Inverted Bottleneck	inverting dims	80.6	4.6
Large Kernel	move ↑ d. conv	79.9	4.1
	kernel sz. → 5	80.4	4.1
	kernel sz. → 7	80.6	4.2
	kernel sz. → 9	80.6	4.2
	kernel sz. → 11	80.5	4.3
	...		
	Swin-T/B	81.3	4.5

ImageNet Top1 Acc (%): 78, 80, 82

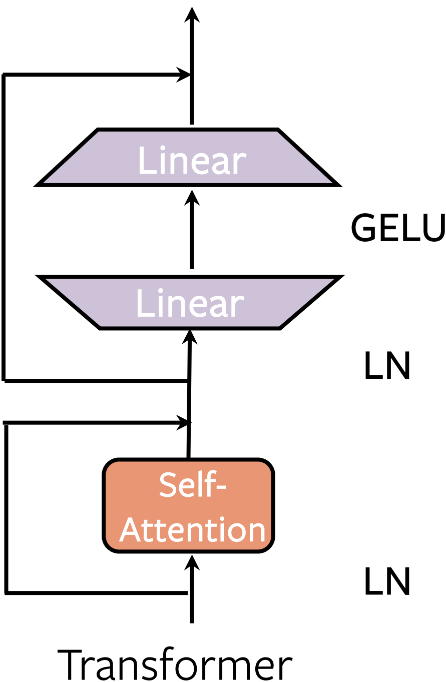
Even larger kernel sizes?

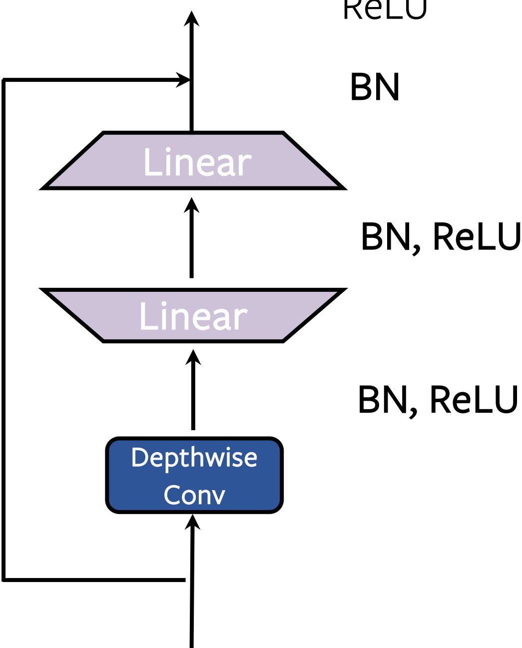
Scaling Up Your Kernels to 31x31: Revisiting Large Kernel Design in CNNs, Ding, Zhang, Han and Ding., CVPR 2022



Kernel size	Architecture	ImageNet Top-1	ImageNet Params	ImageNet FLOPs	ADE20K mIoU	ADE20K Params	ADE20K FLOPs
7-7-7-7	ConvNeXt-Tiny	81.0	29M	4.5G	44.6	60M	939G
7-7-7-7	ConvNeXt-Small	82.1	50M	8.7G	45.9	82M	1027G
7-7-7-7	ConvNeXt-Base	82.8	89M	15.4G	47.2	122M	1170G
31-29-27-13	ConvNeXt-Tiny	81.6	32M	6.1G	**46.2**	64M	973G
31-29-27-13	ConvNeXt-Small	82.5	58M	11.3G	**48.2**	90M	1081G

Micro Design





ResNet-50/200

GFLOPs

Macro Design: stage ratio 79.4 (4.5); "patchify" stem 79.5 (4.4)

ResNeXt: depth conv 78.3 (2.4); width ↑ 80.5 (5.3)

Inverted Bottleneck: inverting dims 80.6 (4.6)

Large Kernel: move ↑ d. conv 79.9 (4.1); kernel sz. → 5 80.4 (4.1); kernel sz. → 7 80.6 (4.2); kernel sz. → 9 80.6 (4.2); kernel sz. → 11 80.5 (4.3)

Micro Design: ReLU→GELU; fewer activations; fewer norms; BN → LN; sep. d.s. conv

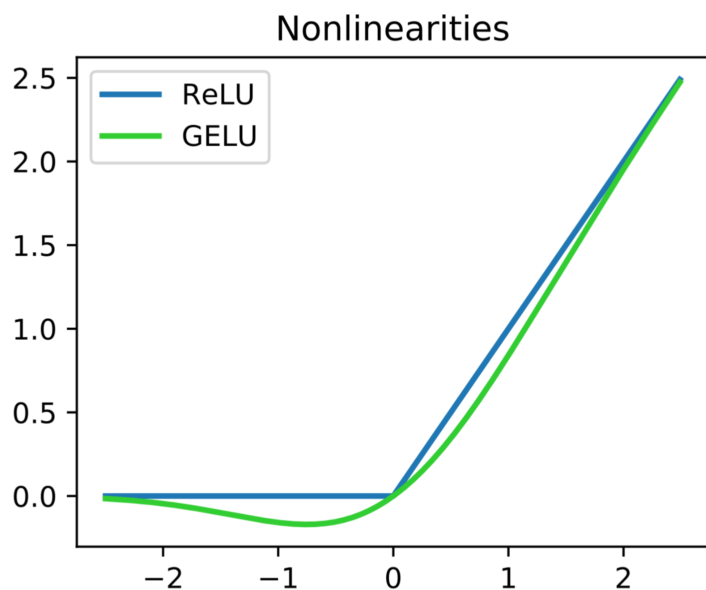
ResNet-50: 78.8 (4.1)

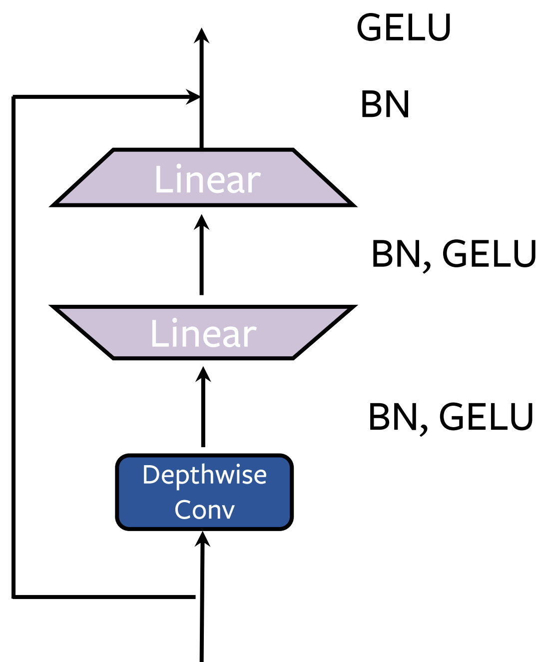
...

Swin-T/B: 81.3 (4.5)

ImageNet Top1 Acc (%): 78, 80, 82

Activations





ReLU → GELU

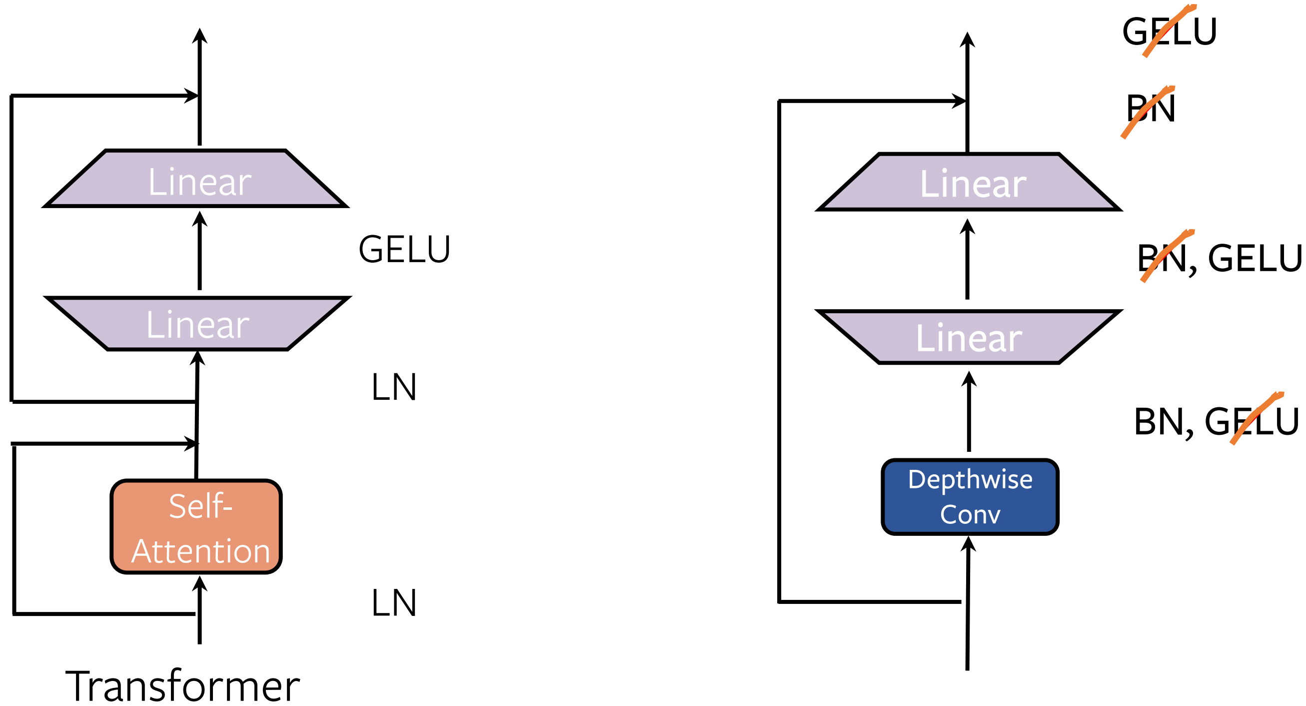
ResNet-50/200

GFLOPs

Group	Step	ImageNet Top1 Acc (%)	GFLOPs
ResNet-50/200		78.8	4.1
Macro Design	stage ratio	79.4	4.5
	"patchify" stem	79.5	4.4
ResNeXt	depth conv	78.3	2.4
	width ↑	80.5	5.3
Inverted Bottleneck	inverting dims	80.6	4.6
Large Kernel	move ↑ d. conv	79.9	4.1
	kernel sz. → 5	80.4	4.1
	kernel sz. → 7	80.6	4.2
	kernel sz. → 9	80.6	4.2
	kernel sz. → 11	80.5	4.3
Micro Design	ReLU→GELU	80.6	4.2
	fewer activations		
	fewer norms		
	BN → LN		
	sep. d.s. conv		
...			
Swin-T/B		81.3	4.5

ImageNet Top1 Acc (%): 78, 80, 82

Fewer Activations / Norms



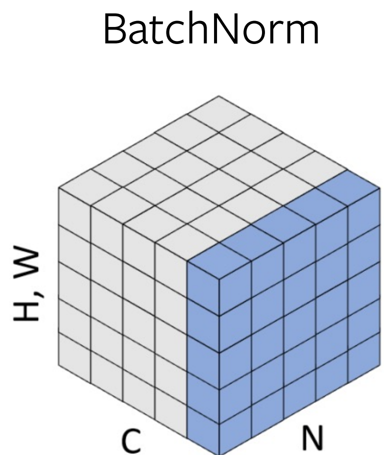
One norm, one activation is good enough per block

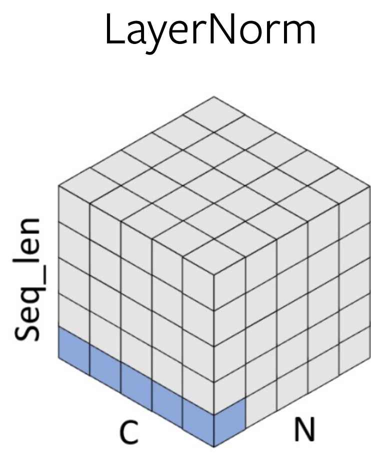
+0.8% without changing FLOPs

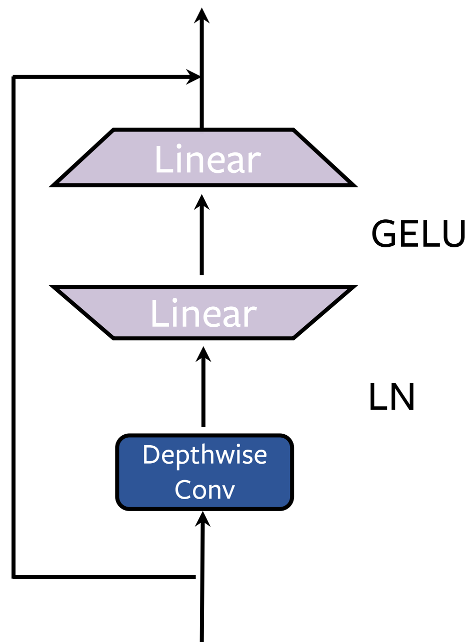
ResNet-50/200
GFLOPs
Macro Design
stage ratio
"patchify" stem
ResNeXt
depth conv
width ↑
Inverted Bottleneck
inverting dims
Large Kernel
move ↑ d. conv
kernel sz. → 5
kernel sz. → 7
kernel sz. → 9
kernel sz. → 11
Micro Design
ReLU→GELU
fewer activations
fewer norms
BN → LN
sep. d.s. conv
...
Swin-T/B
ImageNet Top1 Acc (%)
78
80
82

78.8 4.1
79.4 4.5
79.5 4.4
78.3 2.4
80.5 5.3
80.6 4.6
79.9 4.1
80.4 4.1
80.6 4.2
80.6 4.2
80.5 4.3
80.6 4.2
81.3 4.2
81.3 4.5

Normalization





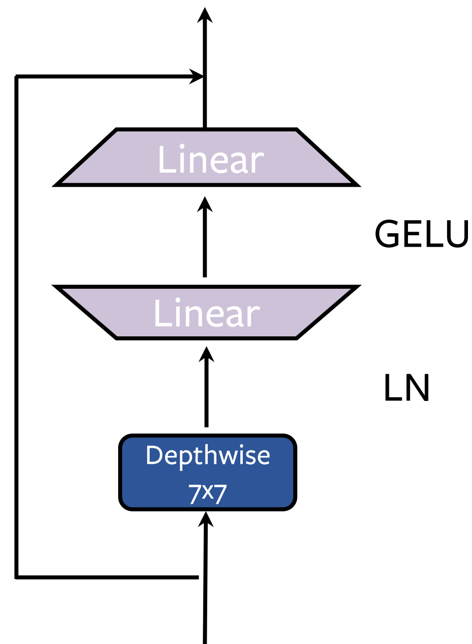


One **LayerNorm** is good enough.
Say 👋 to **BatchNorm** pitfalls!

Group	Step	ImageNet Top1 Acc (%)	GFLOPs
ResNet-50/200		78.8	4.1
Macro Design	stage ratio	79.4	4.5
	"patchify" stem	79.5	4.4
ResNeXt	depth conv	78.3	2.4
	width ↑	80.5	5.3
Inverted Bottleneck	inverting dims	80.6	4.6
Large Kernel	move ↑ d. conv	79.9	4.1
	kernel sz. → 5	80.4	4.1
	kernel sz. → 7	80.6	4.2
	kernel sz. → 9	80.6	4.2
	kernel sz. → 11	80.5	4.3
Micro Design	ReLU→GELU	80.6	4.2
	fewer activations	81.3	4.2
	fewer norms	81.4	4.2
	BN → LN	81.5	4.2
	sep. d.s. conv		
...			
Swin-T/B		81.3	4.5

ImageNet Top1 Acc (%): 78, 80, 82

ConvNeXt Block



Minimal design:

⭐ as simple as possible (but not simpler).

⭐ only 100 lines of code
(vs. 500+ for advanced vision transformers)

Group	Step	ImageNet Top1 Acc (%)	GFLOPs
ResNet-50/200		78.8	4.1
Macro Design	stage ratio	79.4	4.5
	“patchify” stem	79.5	4.4
ResNeXt	depth conv	78.3	2.4
	width ↑	80.5	5.3
Inverted Bottleneck	inverting dims	80.6	4.6
Large Kernel	move ↑ d. conv	79.9	4.1
	kernel sz. → 5	80.4	4.1
	kernel sz. → 7	80.6	4.2
	kernel sz. → 9	80.6	4.2
	kernel sz. → 11	80.5	4.3
Micro Design	ReLU→GELU	80.6	4.2
	fewer activations	81.3	4.2
	fewer norms	81.4	4.2
	BN → LN	81.5	4.2
	sep. d.s. conv	82.0	4.5
ConvNeXt-T/B			
...			
Swin-T/B		81.3	4.5

ImageNet Top1 Acc (%): 78 80 82

```python
class Block(nn.Module):
    r""" ConvNeXt Block. There are two equivalent implementations:
    (1) DwConv -> LayerNorm (channels_first) -> 1x1 Conv -> GELU -> 1x1 Conv; all in (N, C, H, W)
    (2) DwConv -> Permute to (N, H, W, C); LayerNorm (channels_last) -> Linear -> GELU -> Linear; Permute back
    We use (2) as we find it slightly faster in PyTorch

    Args:
        dim (int): Number of input channels.
        drop_path (float): Stochastic depth rate. Default: 0.0
        layer_scale_init_value (float): Init value for Layer Scale. Default: 1e-6.
    """
    def __init__(self, dim, drop_path=0., layer_scale_init_value=1e-6):
        super().__init__()
        self.dwconv = nn.Conv2d(dim, dim, kernel_size=7, padding=3, groups=dim) # depthwise conv
        self.norm = LayerNorm(dim, eps=1e-6)
        self.pwconv1 = nn.Linear(dim, 4 * dim) # pointwise/1x1 convs, implemented with linear layers
        self.act = nn.GELU()
        self.pwconv2 = nn.Linear(4 * dim, dim)
        self.gamma = nn.Parameter(layer_scale_init_value * torch.ones((dim)),
                                    requires_grad=True) if layer_scale_init_value > 0 else None
        self.drop_path = DropPath(drop_path) if drop_path > 0. else nn.Identity()

    def forward(self, x):
        input = x
        x = self.dwconv(x)
        x = x.permute(0, 2, 3, 1) # (N, C, H, W) -> (N, H, W, C)
        x = self.norm(x)
        x = self.pwconv1(x)
        x = self.act(x)
        x = self.pwconv2(x)
        if self.gamma is not None:
            x = self.gamma * x
        x = x.permute(0, 3, 1, 2) # (N, H, W, C) -> (N, C, H, W)

        x = input + self.drop_path(x)
        return x
```

ConvNeXt block

```python
def window_partition(x, window_size):
    """
    Args:
        x: (B, H, W, C)
        window_size (int): window size

    Returns:
        windows: (num_windows*B, window_size, window_size, C)
    """
    B, H, W, C = x.shape
    x = x.view(B, H // window_size, window_size, W // window_size, window_size, C)
    windows = x.permute(0, 1, 3, 2, 4, 5).contiguous().view(-1, window_size, window_size, windo
    return windows


def window_reverse(windows, window_size, H, W):
    """
    Args:
        windows: (num_windows*B, window_size, window_size, C)
        window_size (int): Window size
        H (int): Height of image
        W (int): Width of image

    Returns:
        x: (B, H, W, C)
    """
    B = int(windows.shape[0] / (H * W / window_size / window_size))
    x = windows.view(B, H // window_size, W // window_size, window_size, window_si
    x = x.permute(0, 1, 3, 2, 4, 5).contiguous().view(B, H, W, -1)
    return x


class WindowAttention(nn.Module):
    r""" Window based multi-head self attention (W-MSA) module with relative posit
    It supports both of shifted and non-shifted window.

    Args:
        dim (int): Number of input channels.
        window_size (tuple[int]): The height and width of the window.
        num_heads (int): Number of attention heads.
        qkv_bias (bool, optional):  If True, add a learnable bias to query, key, v
        qk_scale (float | None, optional): Override default qk scale of head_dim *
        attn_drop (float, optional): Dropout ratio of attention weight. Default: 0
        proj_drop (float, optional): Dropout ratio of output. Default: 0.0
    """

    def __init__(self, dim, window_size, num_heads, qkv_bias=True, qk_scale=None,

        super().__init__()
        self.dim = dim
        self.window_size = window_size  # Wh, Ww
        self.num_heads = num_heads
        head_dim = dim // num_heads
        self.scale = qk_scale or head_dim ** -0.5

        # define a parameter table of relative position bias
        self.relative_position_bias_table = nn.Parameter(
            torch.zeros((2 * window_size[0] - 1) * (2 * window_size[1] - 1), num_h

        # get pair-wise relative position index for each token inside the window
        coords_h = torch.arange(self.window_size[0])
        coords_w = torch.arange(self.window_size[1])
        coords = torch.stack(torch.meshgrid([coords_h, coords_w]))  # 2, Wh, Ww
        coords_flatten = torch.flatten(coords, 1)  # 2, Wh*Ww
        relative_coords = coords_flatten[:, :, None] - coords_flatten[:, None, :]
        relative_coords = relative_coords.permute(1, 2, 0).contiguous()  # Wh*Ww,
        relative_coords[:, :, 0] += self.window_size[0] - 1  # shift to start from
        relative_coords[:, :, 1] += self.window_size[1] - 1
        relative_coords[:, :, 0] *= 2 * self.window_size[1] - 1
        relative_position_index = relative_coords.sum(-1)  # Wh*Ww, Wh*Ww
        self.register_buffer("relative_position_index", relative_position_index)

        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)

        trunc_normal_(self.relative_position_bias_table, std=.02)
        self.softmax = nn.Softmax(dim=-1)

    def forward(self, x, mask=None):
        """
        Args:
            x: input features with shape of (num_windows*B, N, C)
            mask: (0/-inf) mask with shape of (num_windows, Wh*Ww, Wh*Ww) or None
        """
        B_, N, C = x.shape
        qkv = self.qkv(x).reshape(B_, N, 3, self.num_heads, C // self.num_heads).p
        q, k, v = qkv[0], qkv[1], qkv[2]  # make torchscript happy (cannot use te
```

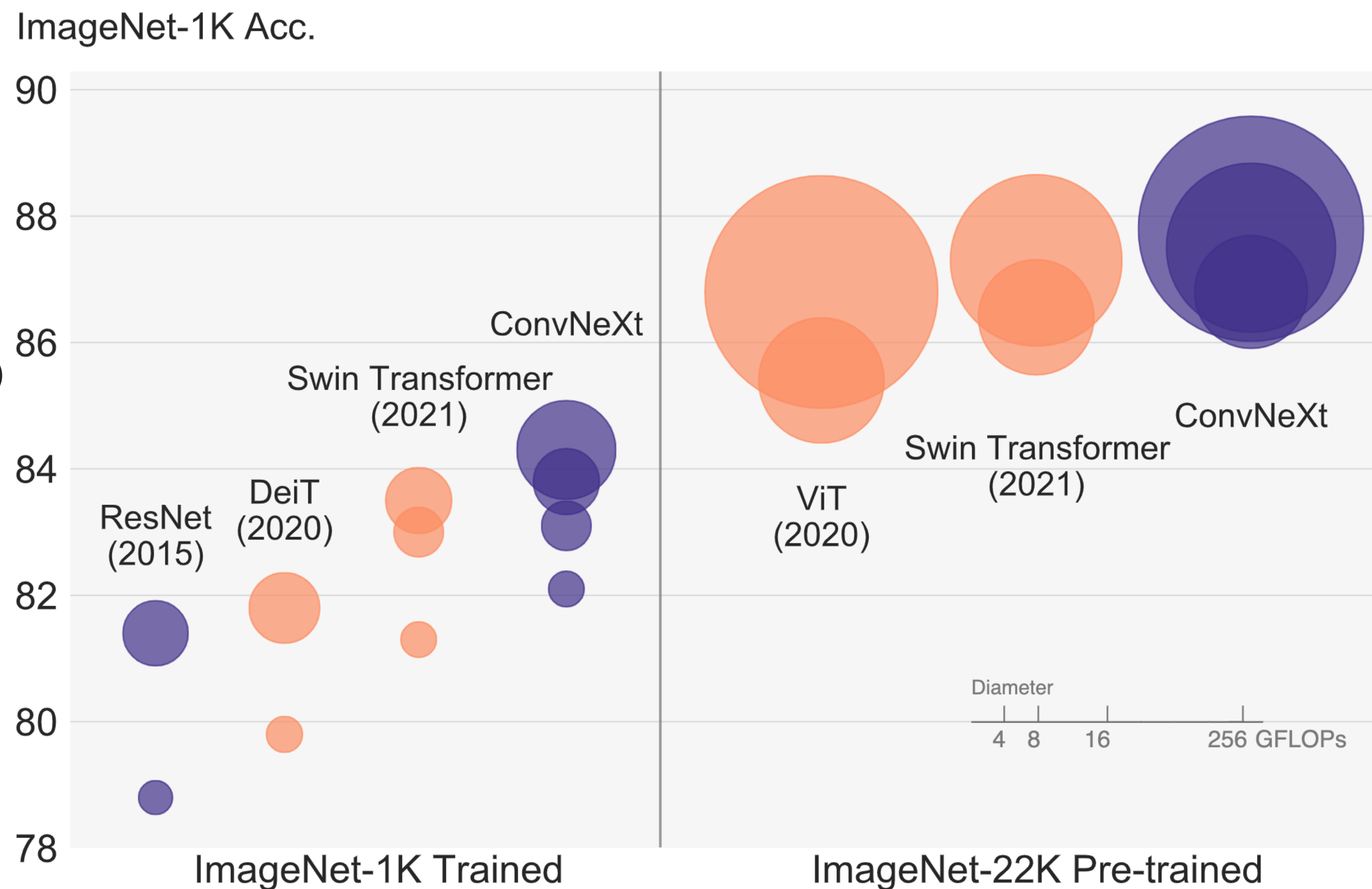
```python
        q = q * self.scale
        attn = (q @ k.transpose(-2, -1))

        relative_position_bias = self.relative_position_bias_table[self.relative_position_index.view(-1)].view(
            self.window_size[0] * self.window_size[1], self.window_size[0] * self.window_size[1], -1)  # Wh*Ww,Wh*Ww,nH
        relative_position_bias = relative_position_bias.permute(2, 0, 1).contiguous()  # nH, Wh*Ww, Wh*Ww
        attn = attn + relative_position_bias.unsqueeze(0)

        if mask is not None:
            nW = mask.shape[0]
            attn = attn.view(B_ // nW, nW, self.num_heads, N, N) + mask.unsqueeze(1).unsqueeze(0)
            attn = attn.view(-1, self.num_heads, N, N)
            attn = self.softmax(attn)
        else:
            attn = self.softmax(attn)

        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 extra_repr(self) -> str:
        return f'dim={self.dim}, window_size={self.window_size}, num_heads={self.num_heads}'

    def flops(self, N):
        # calculate flops for 1 window with token length of N
        flops = 0
        # qkv = self.qkv(x)
        flops += N * self.dim * 3 * self.dim
        # attn = (q @ k.transpose(-2, -1))
        flops += self.num_heads * N * (self.dim // self.num_heads) * N
        #  x = (attn @ v)
        flops += self.num_heads * N * N * (self.dim // self.num_heads)
        # x = self.proj(x)
        flops += N * self.dim * self.dim
        return flops


class SwinTransformerBlock(nn.Module):
    r""" Swin Transformer Block.

    Args:
        dim (int): Number of input channels.
        input_resolution (tuple[int]): Input resulotion.
        num_heads (int): Number of attention heads.
        window_size (int): Window size.
        shift_size (int): Shift size for SW-MSA.
        mlp_ratio (float): Ratio of mlp hidden dim to embedding dim.
        qkv_bias (bool, optional): If True, add a learnable bias to query, key, value. Default: True
        qk_scale (float | None, optional): Override default qk scale of head_dim ** -0.5 if set.
        drop (float, optional): Dropout rate. Default: 0.0
        attn_drop (float, optional): Attention dropout rate. Default: 0.0
        drop_path (float, optional): Stochastic depth rate. Default: 0.0
        act_layer (nn.Module, optional): Activation layer. Default: nn.GELU
        norm_layer (nn.Module, optional): Normalization layer.  Default: nn.LayerNorm
    """

    def __init__(self, dim, input_resolution, num_heads, window_size=7, shift_size=
                 mlp_ratio=4., qkv_bias=True, qk_scale=None, drop=0., attn_drop=0.,
                 act_layer=nn.GELU, norm_layer=nn.LayerNorm):
        super().__init__()
        self.dim = dim
        self.input_resolution = input_resolution
        self.num_heads = num_heads
        self.window_size = window_size
        self.shift_size = shift_size
        self.mlp_ratio = mlp_ratio
        if min(self.input_resolution) <= self.window_size:
            # if window size is larger than input resolution, we don't partition wi
            self.shift_size = 0
            self.window_size = min(self.input_resolution)
        assert 0 <= self.shift_size < self.window_size, "shift_size must in 0-windo

        self.norm1 = norm_layer(dim)
        self.attn = WindowAttention(
            dim, window_size=to_2tuple(self.window_size), num_heads=num_heads,
            qkv_bias=qkv_bias, qk_scale=qk_scale, attn_drop=attn_drop, proj_drop=dr

        self.drop_path = DropPath(drop_path) if drop_path > 0. else nn.Identity()
        self.norm2 = norm_layer(dim)
        mlp_hidden_dim = int(dim * mlp_ratio)
        self.mlp = Mlp(in_features=dim, hidden_features=mlp_hidden_dim, act_layer=a

        if self.shift_size > 0:
            # calculate attention mask for SW-MSA
            ...
                        slice(-self.window_size, -self.shift_size),
                        slice(-self.shift_size, None))
            w_slices = (slice(0, -self.window_size),
                        slice(-self.window_size, -self.shift_size),
                        slice(-self.shift_size, None))
            cnt = 0
            for h in h_slices:
                for w in w_slices:
                    img_mask[:, h, w, :] = cnt
                    cnt += 1
```

```python
            mask_windows = window_partition(img_mask, self.window_size)  # nW, window_size, window_size, 1
            mask_windows = mask_windows.view(-1, self.window_size * self.window_size)
            attn_mask = mask_windows.unsqueeze(1) - mask_windows.unsqueeze(2)
            attn_mask = attn_mask.masked_fill(attn_mask != 0, float(-100.0)).masked_fill(attn_mask == 0, float(0.0))
        else:
            attn_mask = None

        self.register_buffer("attn_mask", attn_mask)

    def forward(self, x):
        H, W = self.input_resolution
        B, L, C = x.shape
        assert L == H * W, "input feature has wrong size"

        shortcut = x
        x = self.norm1(x)
        x = x.view(B, H, W, C)

        # cyclic shift
        if self.shift_size > 0:
            shifted_x = torch.roll(x, shifts=(-self.shift_size, -self.shift_size), dims=(1, 2))
        else:
            shifted_x = x

        # partition windows
        x_windows = window_partition(shifted_x, self.window_size)  # nW*B, window_size, window_size, C
        x_windows = x_windows.view(-1, self.window_size * self.window_size, C)  # nW*B, window_size*window_size, C

        # W-MSA/SW-MSA
        attn_windows = self.attn(x_windows, mask=self.attn_mask)  # nW*B, window_size*window_size, C

        # merge windows
        ...
        # reverse cyclic shift
        if self.shift_size > 0:
            x = torch.roll(shifted_x, shifts=(self.shift_size, self.shift_size), dims=(1, 2))
        else:
            x = shifted_x
        x = x.view(B, H * W, C)
        x = shortcut + self.drop_path(x)

        # FFN
        x = x + self.drop_path(self.mlp(self.norm2(x)))

        return x
```

Swin Transformer block

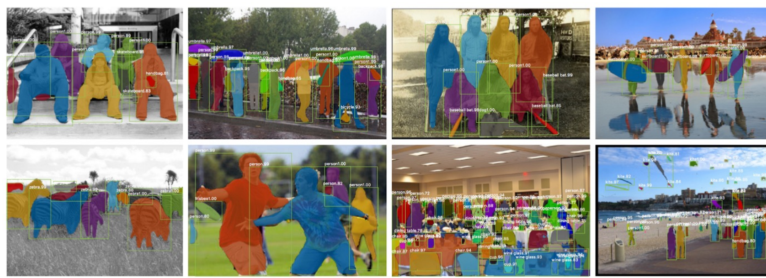
ConvNeXt: Results

- Attention is *NOT* essential
- ConvNets can be scalable
  (while being much **simpler** in design)



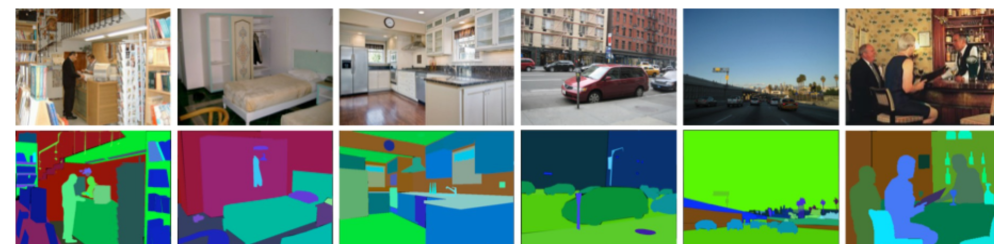
ConvNeXt: Downstream Transfer (important!)

Consistently ***outperforms*** SOTA vision transformers



backbone	FLOPs	FPS	AP^{box}	AP^{box}_{50}	AP^{box}_{75}	AP^{mask}	AP^{mask}_{50}	AP^{mask}_{75}
Mask-RCNN 3× schedule								
○ Swin-T	267G	23.1	46.0	68.1	50.3	41.6	65.1	44.9
• ConvNeXt-T	262G	25.6	**46.2**	67.9	50.8	**41.7**	65.0	44.9
Cascade Mask-RCNN 3× schedule								
• ResNet-50	739G	11.4	46.3	64.3	50.5	40.1	61.7	43.4
• X101-32	819G	9.2	48.1	66.5	52.4	41.6	63.9	45.2
• X101-64	972G	7.1	48.3	66.4	52.3	41.7	64.0	45.1
○ Swin-T	745G	12.2	50.4	69.2	54.7	43.7	66.6	47.3
• ConvNeXt-T	741G	13.5	**50.4**	69.1	54.8	**43.7**	66.5	47.3
○ Swin-S	838G	11.4	51.9	70.7	56.3	45.0	68.2	48.8
• ConvNeXt-S	827G	12.0	**51.9**	70.8	56.5	**45.0**	68.4	49.1
○ Swin-B	982G	10.7	51.9	70.5	56.4	45.0	68.1	48.9
• ConvNeXt-B	964G	11.4	**52.7**	71.3	57.2	**45.6**	68.9	49.5
○ Swin-B‡	982G	10.7	53.0	71.8	57.5	45.8	69.4	49.7
• ConvNeXt-B‡	964G	11.5	**54.0**	73.1	58.8	**46.9**	70.6	51.3
○ Swin-L‡	1382G	9.2	53.9	72.4	58.8	46.7	70.1	50.8
• ConvNeXt-L‡	1354G	10.0	**54.8**	73.8	59.8	**47.6**	71.3	51.7
• ConvNeXt-XL‡	1898G	8.6	**55.2**	74.2	59.9	**47.7**	71.6	52.2

COCO Detection and Instance Segmentation



backbone	input crop.	mIoU	#param.	FLOPs
ImageNet-1K pre-trained				
○ Swin-T	512^2	45.8	60M	945G
• ConvNeXt-T	512^2	**46.7**	60M	939G
○ Swin-S	512^2	49.5	81M	1038G
• ConvNeXt-S	512^2	**49.6**	82M	1027G
○ Swin-B	512^2	49.7	121M	1188G
• ConvNeXt-B	512^2	**49.9**	122M	1170G
ImageNet-22K pre-trained				
○ Swin-B‡	640^2	51.7	121M	1841G
• ConvNeXt-B‡	640^2	**53.1**	122M	1828G
○ Swin-L‡	640^2	53.5	234M	2468G
• ConvNeXt-L‡	640^2	**53.7**	235M	2458G
• ConvNeXt-XL‡	640^2	**54.0**	391M	3335G

ADE20K Semantic Segmentation

"The ConvNeXt model release saved us during the ECCV deadline. We don't have an internal Swin Transformer implementation at Google. But we were able to easily implement ConvNeXt and obtained SOTA results with the pre-trained model."

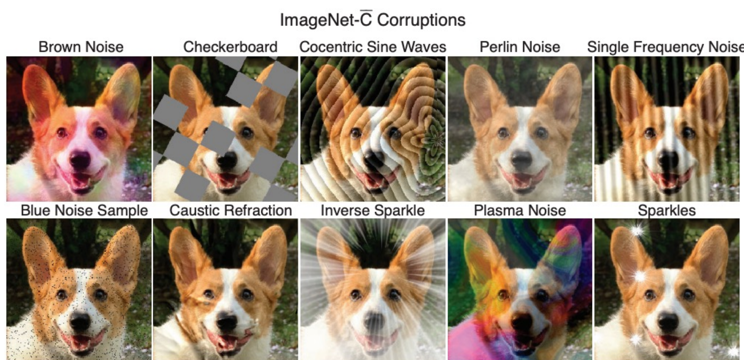
– A vision researcher at Google

Preliminary observation: inference speed on A100s

model	image size	FLOPs	throughput (image / s)	IN-1K / 22K trained, 1K acc.
○ Swin-T	224^2	4.5G	1325.6	81.3 / –
● ConvNeXt-T	224^2	4.5G	**1943.5** (+47%)	**82.1** / –
○ Swin-S	224^2	8.7G	857.3	83.0 / –
● ConvNeXt-S	224^2	8.7G	**1275.3** (+49%)	**83.1** / –
○ Swin-B	224^2	15.4G	662.8	83.5 / 85.2
● ConvNeXt-B	224^2	15.4G	**969.0** (+46%)	**83.8** / **85.8**
○ Swin-B	384^2	47.1G	242.5	84.5 / 86.4
● ConvNeXt-B	384^2	45.0G	**336.6** (+39%)	**85.1** / **86.8**
○ Swin-L	224^2	34.5G	435.9	– / 86.3
● ConvNeXt-L	224^2	34.4G	**611.5** (+40%)	**84.3** / **86.6**
○ Swin-L	384^2	103.9G	157.9	– / 87.3
● ConvNeXt-L	384^2	101.0G	**211.4** (+34%)	85.5 / **87.5**
● ConvNeXt-XL	224^2	60.9G	**424.4**	– / **87.0**
● ConvNeXt-XL	384^2	179.0G	**147.4**	– / **87.8**

Table 12. **Inference throughput comparisons on an A100 GPU.** ConvNeXt enjoys up to ∼49% higher throughput compared with a Swin Transformer with similar FLOPs.

Robust models? Scale matters!



Model	Data/Size	FLOPs / Params	Clean	C (↓)	C̄ (↓)	A	R	SK
ResNet-50	1K/224^2	4.1 / 25.6	76.1	76.7	57.7	0.0	36.1	24.1
Swin-T [42]	1K/224^2	4.5 / 28.3	81.2	62.0	-	21.6	41.3	29.1
RVT-S* [44]	1K/224^2	4.7 / 23.3	81.9	49.4	37.5	25.7	47.7	34.7
ConvNeXt-T	1K/224^2	4.5 / 28.6	82.1	53.2	40.0	24.2	47.2	33.8
Swin-B [42]	1K/224^2	15.4 / 87.8	83.4	54.4	-	35.8	46.6	32.4
RVT-B* [44]	1K/224^2	17.7 / 91.8	82.6	46.8	**30.8**	28.5	48.7	36.0
ConvNeXt-B	1K/224^2	15.4 / 88.6	**83.8**	**46.8**	34.4	**36.7**	**51.3**	**38.2**
ConvNeXt-B	22K/384^2	45.1 / 88.6	86.8	43.1	30.7	62.3	64.9	51.6
ConvNeXt-L	22K/384^2	101.0 / 197.8	87.5	40.2	29.9	65.5	66.7	52.8
ConvNeXt-XL	22K/384^2	179.0 / 350.2	**87.8**	**38.8**	**27.1**	**69.3**	**68.2**	**55.0**

Table 8. **Robustness evaluation of ConvNeXt.** We do not make use of any specialized modules or additional fine-tuning procedures.

ConvNeXt

[Liu, Mao, Wu, Feichtenhofer, Darrell, **Xie.** CVPR 2022]

ResNet-50/200 78.8 • 4.1

Swin-T/B 81.3 • 4.5

ImageNet Top1 Acc (%) 78 80 82

- Not to push for SOTA
  - Focus on enabling fair comparisons

⭐ ConvNeXt represents the **community effort!**

Many design choices have been examined **separately** over the last decade, but *not* **collectively**.

ResNet 2015 · ResNeXt 2017 · ViT 2020 · ConvNeXt 2022

kaggle.com

Which image models are best?

Explore and run machine learning code with Kaggle Notebooks | Using data from No attached data sources

Inference speed/accuracy

(training is similar)

Vision architectures for the 2020s:

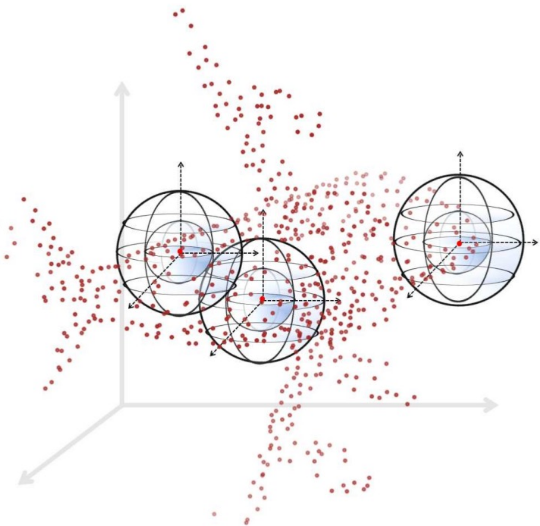
Transformers or ConvNets?

Credit: Godzilla vs. Kong (2021)

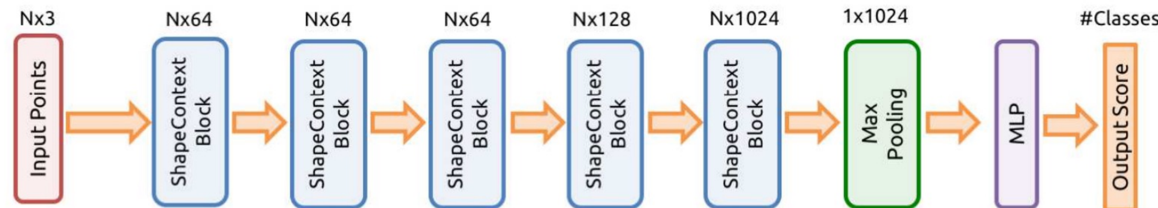
Disclaimer: not a Transformer hater! (who would be?)

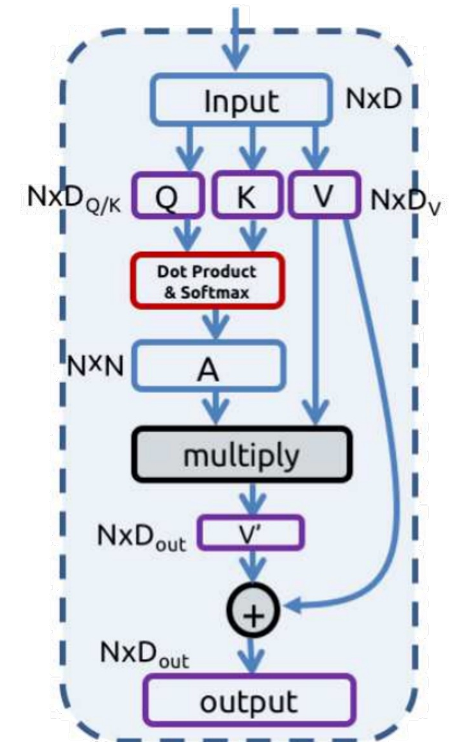
- Self-attention is awesome for sequences/sets.
- I use it quite often!

E.g., first pure Transformer model used for point cloud processing.



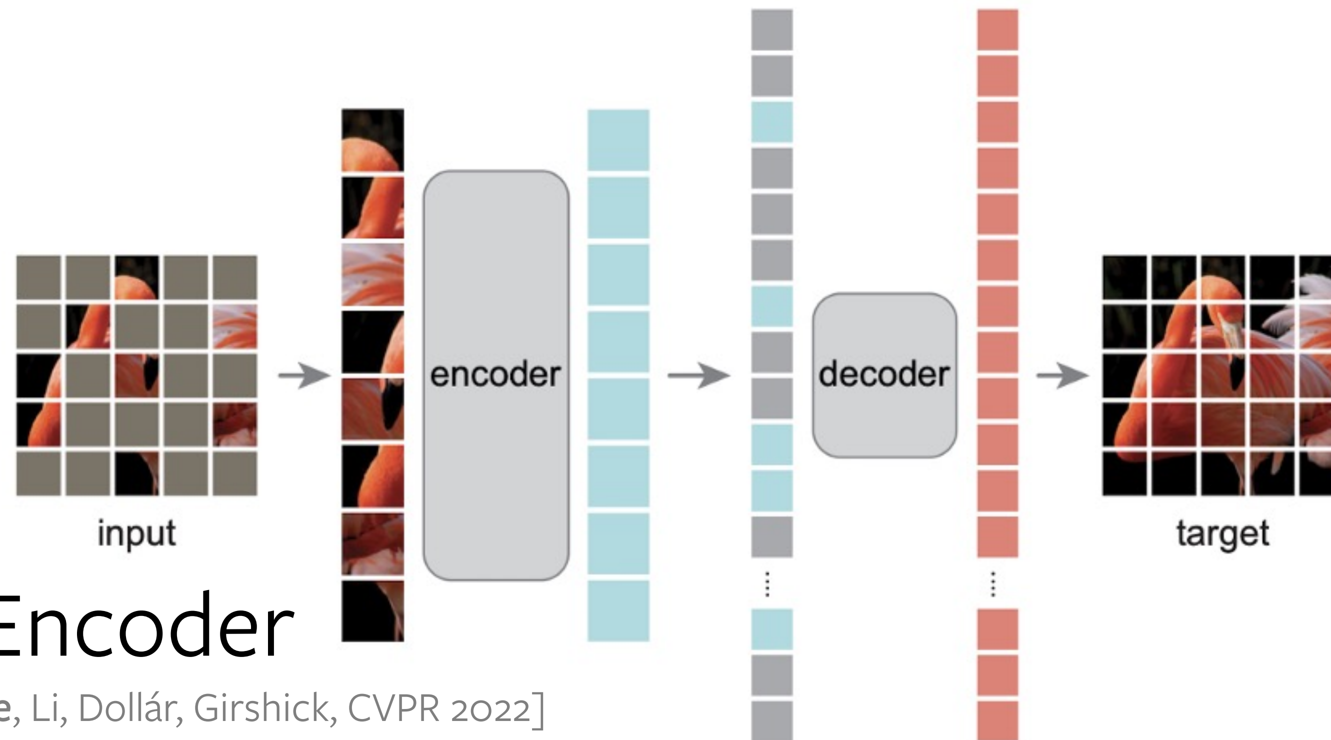
[Xie, et al., CVPR 2018]





Attentional ShapeContextNet block

Transformer can lead to flexible designs



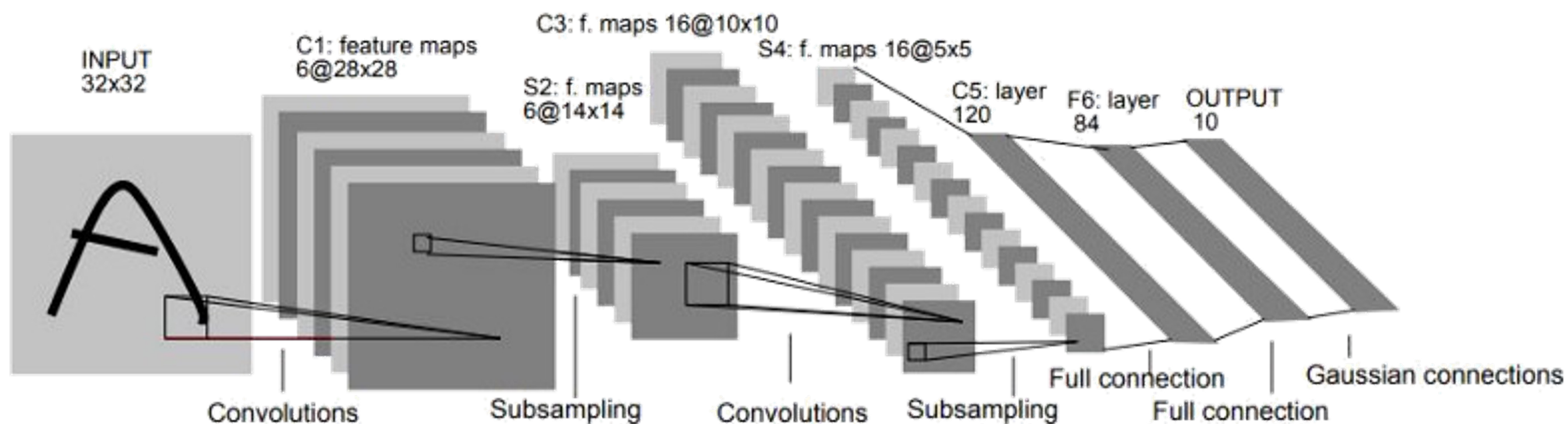
Masked Auto-Encoder

[He, Chen, **Xie**, Li, Dollár, Girshick, CVPR 2022]

Transformer enables asymmetric encoder-decoder structure.
Significant speed up!

ConvNet or Transformer? An ill-defined question

- Really, what is a ConvNet?

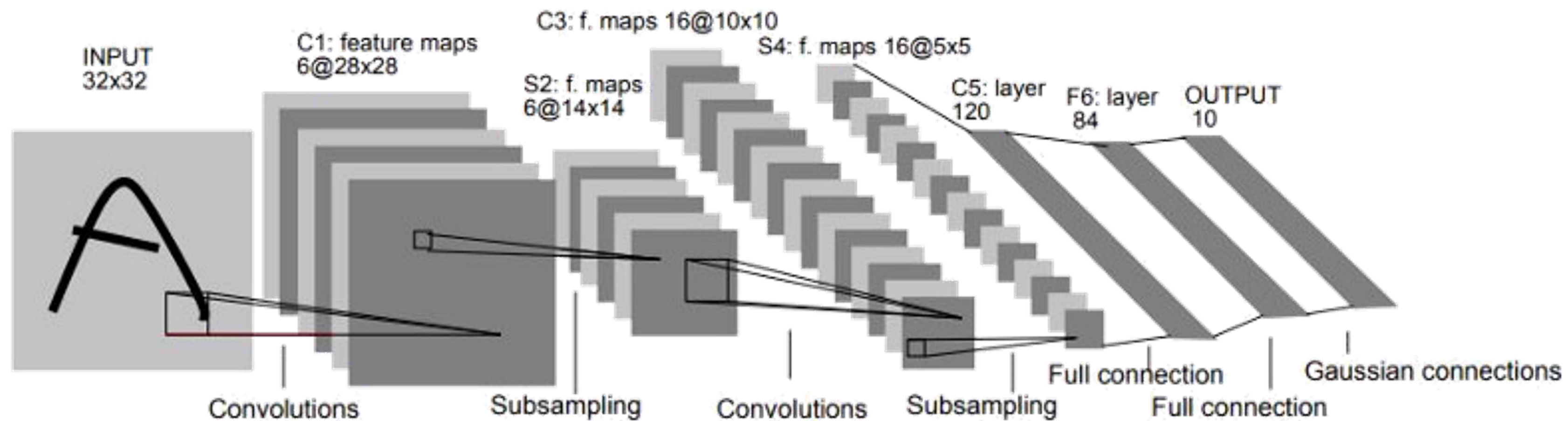


An ill-defined question

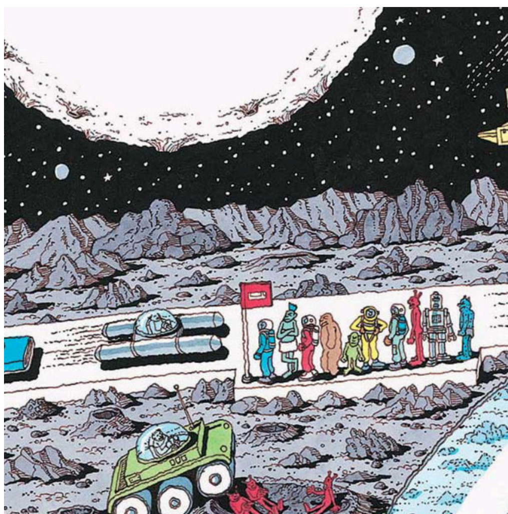
- Two ways to think about a ConvNet
  - A network using ConvNet *inductive bias*
  - A network using *convolution operations*

What is a ConvNet?

- A network using ConvNet *inductive bias*
    - Sliding Window
    - Weight Sharing
    - Hierarchical structure
    - Locality



ConvNet inductive bias has been & will be essential



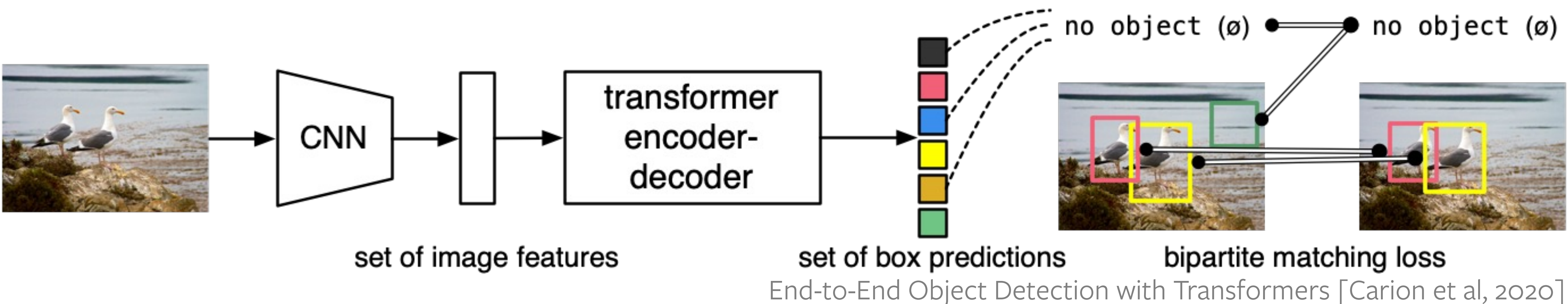
Given a small image

Vanilla Transformer: no problem!

For practitioners, a hybrid system is all you need

- A Conv + self-attention network can work better
  e.g. LeViT [Graham et al., 2021], CoAtNet [Dai et al., 2021], CoaT [Xu et al., 2021], ...

- For images, you might not need a full Transformer;
  You need self-attention blocks at the end!

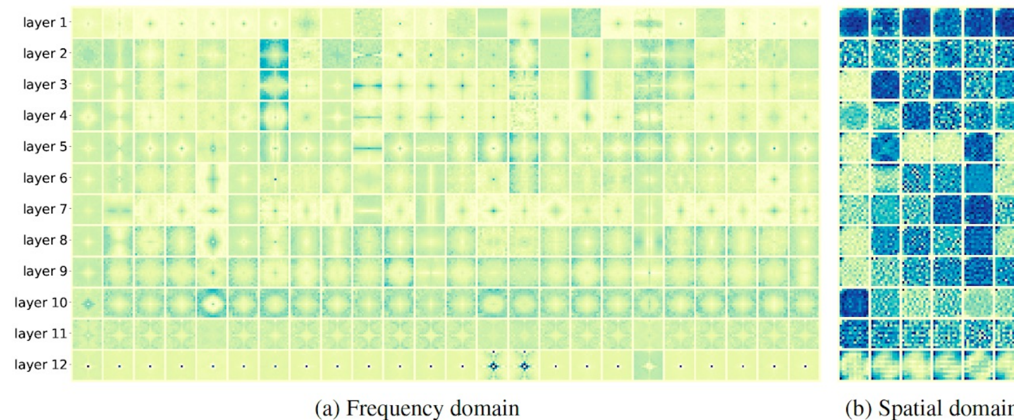


End-to-End Object Detection with Transformers [Carion et al, 2020]

What is a ConvNet?

- A network using ***Convolution operations*** (but not self-attention)

  - May not need overlapping convolutions
  - May not need locality inductive bias (global circular convolution, FFT)
  - May not need feature hierarchy (isotropic architecture)



(a) Frequency domain (b) Spatial domain

[Rao, Zhao, et al., 2021]

ML Empiricist Utopia

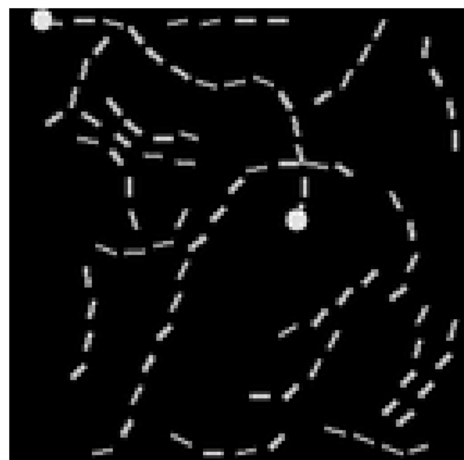
- “Inductive biases must be removed!”

no image inductive bias at all: pixels as sequence

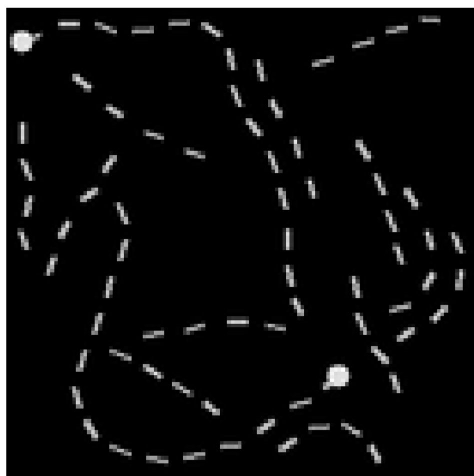
Model	ListOps	Text	Retrieval	Image	Pathfinder	Path-X	Avg
Transformer	36.37	64.27	57.46	42.44	71.40	FAIL	54.39
Local Attention	15.82	52.98	53.39	41.46	66.63	FAIL	46.06
Sparse Trans.	17.07	63.58	**59.59**	**44.24**	71.71	FAIL	51.24
Longformer	35.63	62.85	56.89	42.22	69.71	FAIL	53.46
Linformer	35.70	53.94	52.27	38.56	76.34	FAIL	51.36
Reformer	**37.27**	56.10	53.40	38.07	68.50	FAIL	50.67
Sinkhorn Trans.	33.67	61.20	53.83	41.23	67.45	FAIL	51.39
Synthesizer	36.99	61.68	54.67	41.61	69.45	FAIL	52.88
BigBird	36.05	64.02	59.29	40.83	74.87	FAIL	**55.01**
Linear Trans.	16.13	**65.90**	53.09	42.34	75.30	FAIL	50.55
Performer	18.01	65.40	53.82	42.77	**77.05**	FAIL	51.41

Long Range Arena: A Benchmark for Efficient Transformers

[Y Tay, et al., 2020]



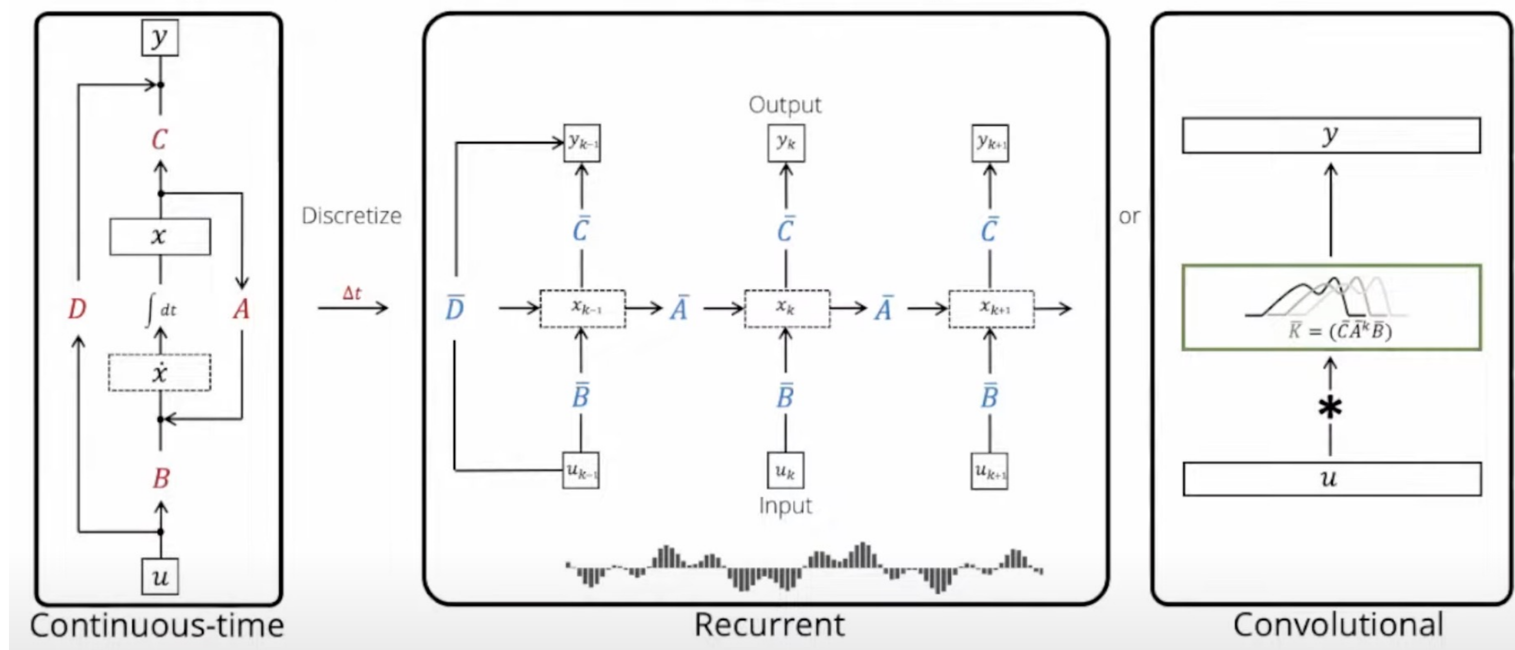
(a) A positive example.



(b) A negative example.

Structured State Spaces (S4)

[Gu, et al., 2021]



Model	ListOps	Text	Retrieval	Image	Pathfinder	Path-X	Avg
Transformer	36.37	64.27	57.46			✗	53.66
						✗	50.56
						✗	54.17
						✗	50.46
						✗	51.18
	35.33	65.11	59.61	38.67	<u>77.80</u>	✗	54.42
Nyströmformer	37.15	65.52	<u>79.56</u>	41.58	70.94	✗	57.46
Luna-256	37.25	64.57	79.29	<u>47.38</u>	77.72	✗	<u>59.37</u>
S4	**58.35**	**76.02**	**87.09**	**87.26**	**86.05**	**88.10**	**80.48**

Efficient training as a CNN!
At inference, view as an RNN

Credit: Godzilla vs. Kong (2021)

“Your assumptions are your windows on the world. Scrub them off every once in a while, or the light won’t come in.”
— Isaac Asimov
Credit: Carol Herring