Convolutional Neural Network 2

CSE 849 Deep Learning

Spring 2025

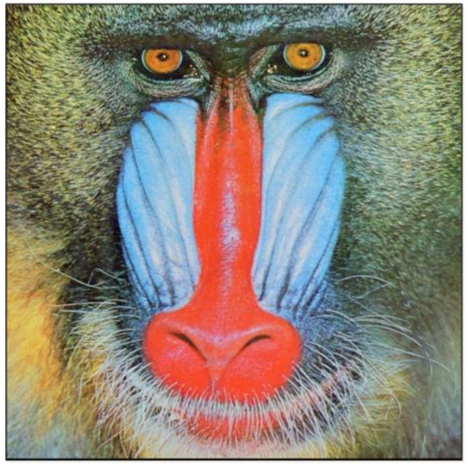
Zijun Cui

Partial slides are adapted from CMU 11-785 Introduction to Deep Learning by Bhiksha Raj

Notation

- “Channel”: the number of filters
- “Depth”: for each filter, the depth equals to the number of feature maps from the previous layer

Convolutional Filters



Original: Mandrill

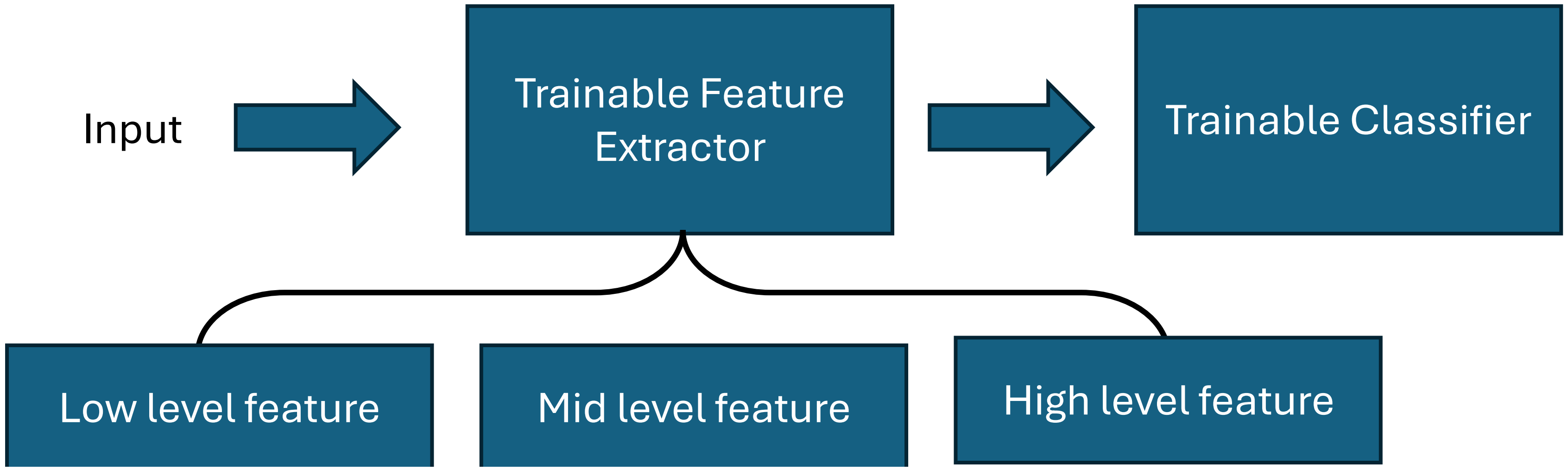
Smoothed with
Gaussian kernel

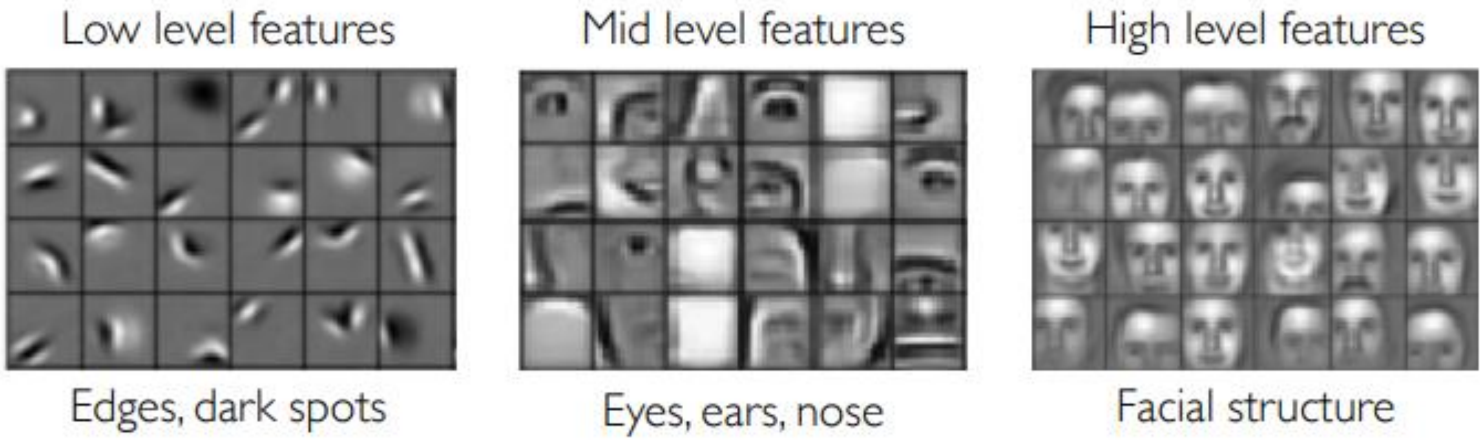
Original Image

Edges

More can be found in computer vision

Recall: Deep Learning = Learning Representations





Outline

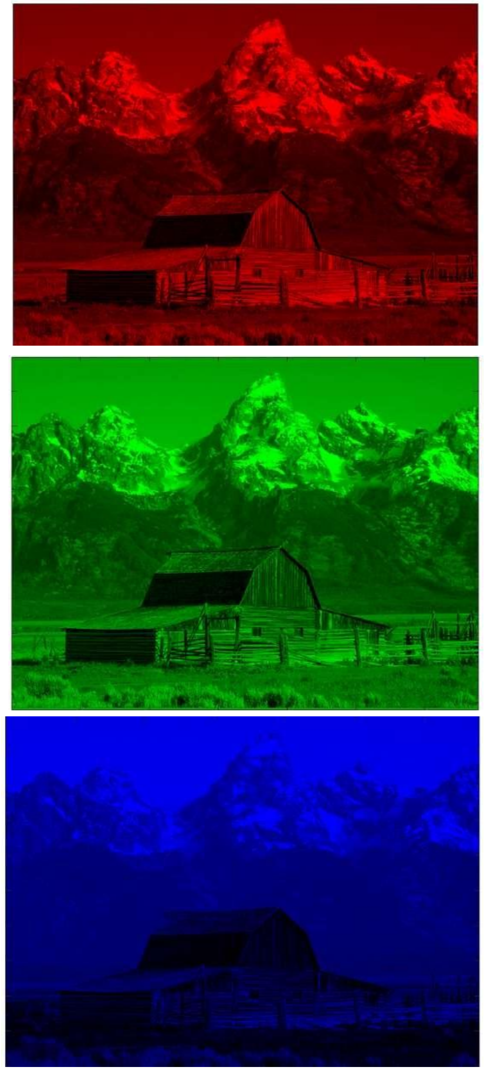
- An CNN model for image classification

- More recent CNN architecture designs
  - Group Convolution and ResNeXt
  - Neural Architecture Search and EfficientNets
  - RegNets and NFNets

Setting everything together

- Typical image classification task
  - Assuming maxpooling..

Convolutional Neural Networks



- Input: 1 or 3 images
  - Grey scale or color
  - Will assume color to be generic

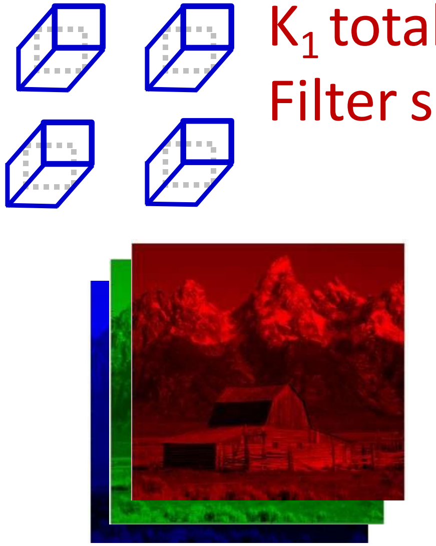
Convolutional Neural Networks

- Input: 3 pictures

Preprocessing

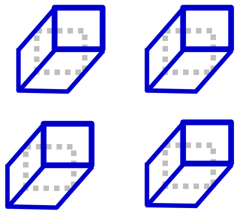
- Large images are a problem
  - Too large
  - Compute and memory intensive to process
- Sometimes scaled to smaller sizes, e.g. 128x128 or even 32x32
  - Based on how much will fit on your GPU
  - Typically cropped to *square* images
  - Filters are also typically square

Convolutional Neural Networks



- Input is convolved with a set of K_1 filters
  - Typically K_1 is a power of 2, e.g. 2, 4, 8, 16, 32,..
  - Filters are typically 5x5, 3x3, or even 1x1

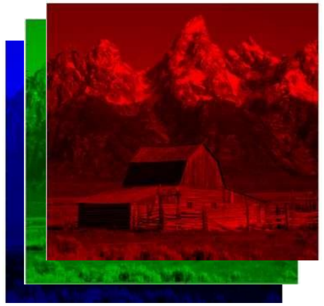
Convolutional Neural Networks



K_1 total filters

Filter size: $L \times L \times 3$

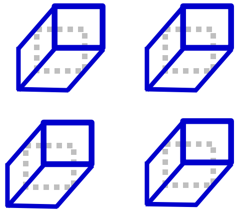
Small enough to capture fine features
(particularly important for scaled-down images)



$I \times I$ *image*

- Input is convolved with a set of K_1 filters
  - Typically K_1 is a power of 2, e.g. 2, 4, 8, 16, 32,..
  - Filters are typically 5x5, 3x3, or even 1x1

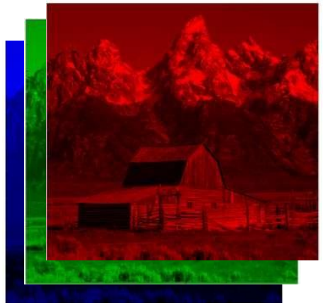
Convolutional Neural Networks



K_1 total filters

Filter size: $L \times L \times 3$

Small enough to capture fine features
(particularly important for scaled-down images)

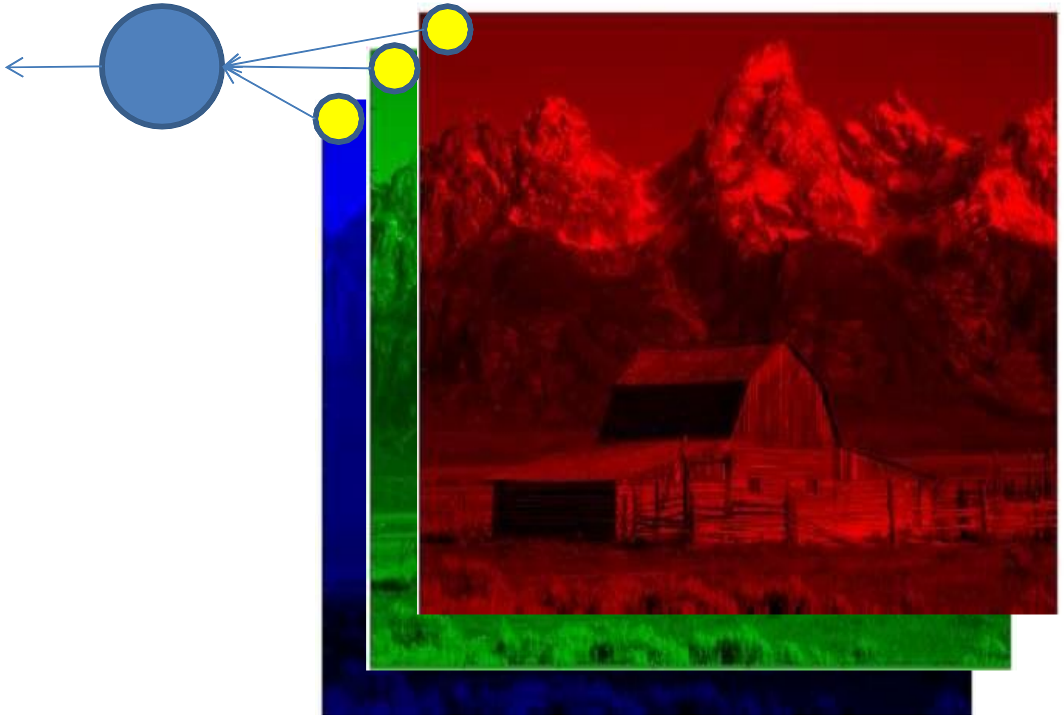


$I \times I$ $image$

What on earth is this?

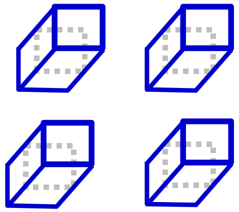
- Input is convolved with a set of K_1 filters
  - Typically K_1 is a power of 2, e.g. 2, 4, 8, 16, 32,..
  - Filters are typically 5x5, 3x3, or even 1x1

The 1x1 filter



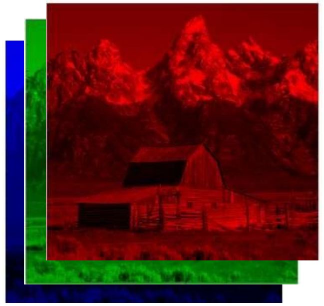
- A 1x1 filter is simply a perceptron that operates over the *depth* of the stack of maps, but has no spatial extent
  - Takes one pixel from each of the maps (at a given location) as input

Convolutional Neural Networks



K_1 total filters

Filter size: $L \times L \times 3$



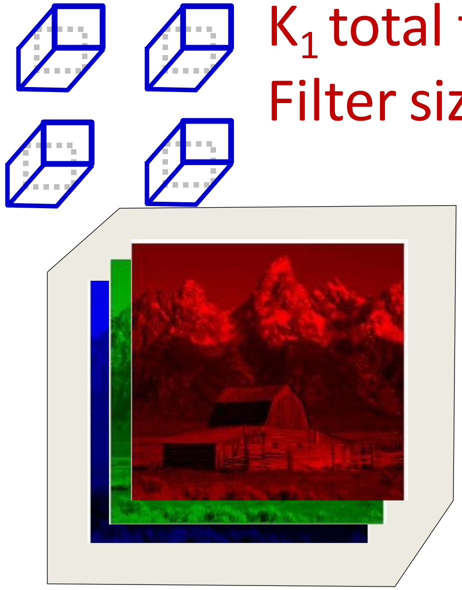
$I \times I\ image$

Parameters to choose: K_1, L and S

1. Number of filters K_1
2. Size of filters $L \times L \times 3 + bias$
3. *Stride* of convolution S

- Input is convolved with a set of K_1 filters
  - Typically K_1 is a power of 2, e.g. 2, 4, 8, 16, 32,..
  - Filters are typically 5x5(x3), 3x3(x3), or even 1x1(x3)
  - ***Typical stride***: 1 or 2

Convolutional Neural Networks



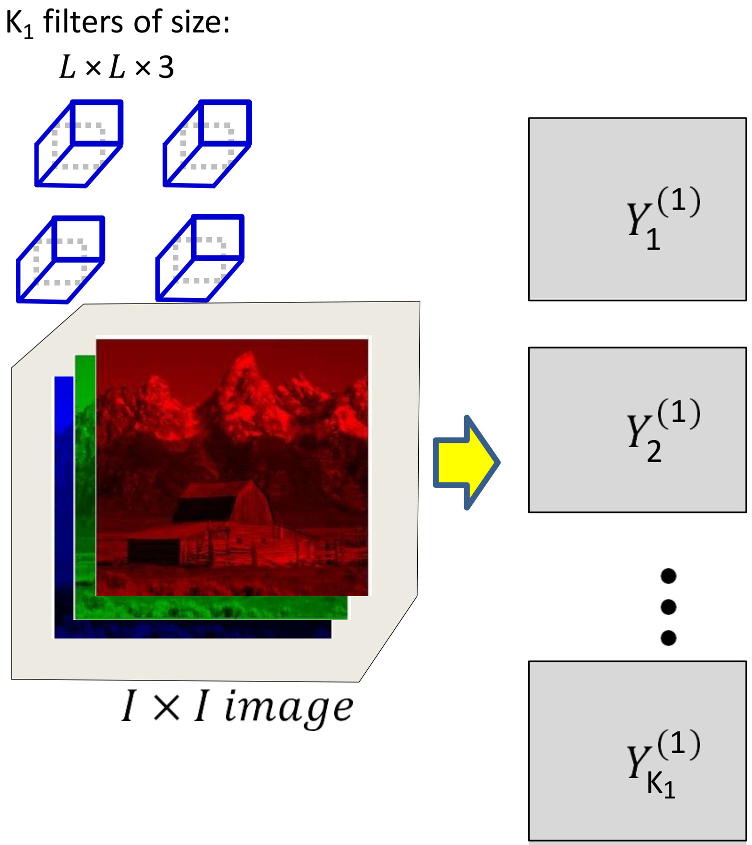
K_1 total filters

Filter size: $L \times L \times 3$

$I \times I$ *image*

- The input may be zero-padded according to the size of the chosen filters

Convolutional Neural Networks



The layer includes a convolution operation followed by an activation (typically RELU)

$$z_m^{(1)}(i,j) = \sum_{c \in \{R,G,B\}} \sum_{k=1}^{L} \sum_{l=1}^{L} w_m^{(1)}(c,k,l) I_c(i+k, j+l) + b_m^{(1)}$$

$$Y_m^{(1)}(i,j) = f\left(z_m^{(1)}(i,j)\right)$$

- **First convolutional layer:** Several convolutional filters
  - Filters are “3-D” (third dimension is color)
  - Convolution followed typically by a RELU activation
- Each filter creates a single 2-D output map

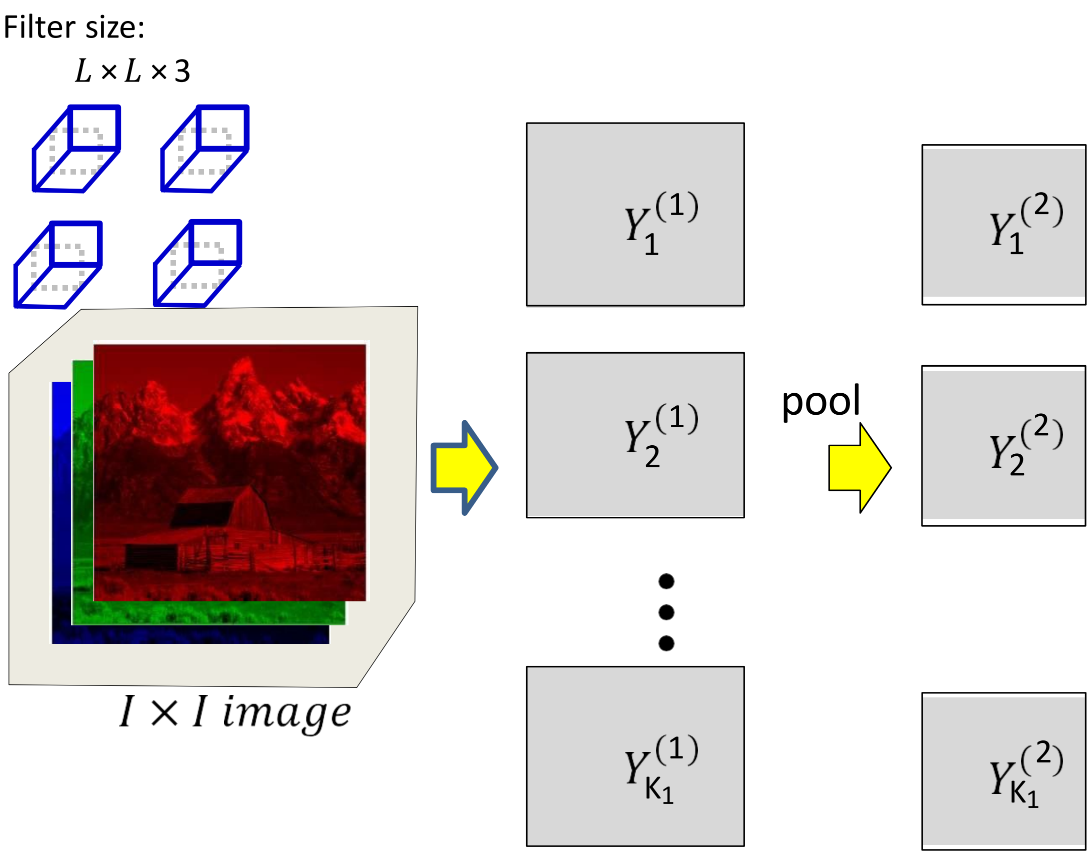
Learnable parameters in the first convolutional layer

- The first convolutional layer comprises K_1 filters, each of size $L \times L \times 3$
  - Spatial span: $L \times L$
  - Depth : 3 (3 colors)

- This represents a total of $K_1(3L^2 + 1)$ parameters
  - "+ 1" because each filter also has a bias

- All of these parameters must be learned

Convolutional Neural Networks

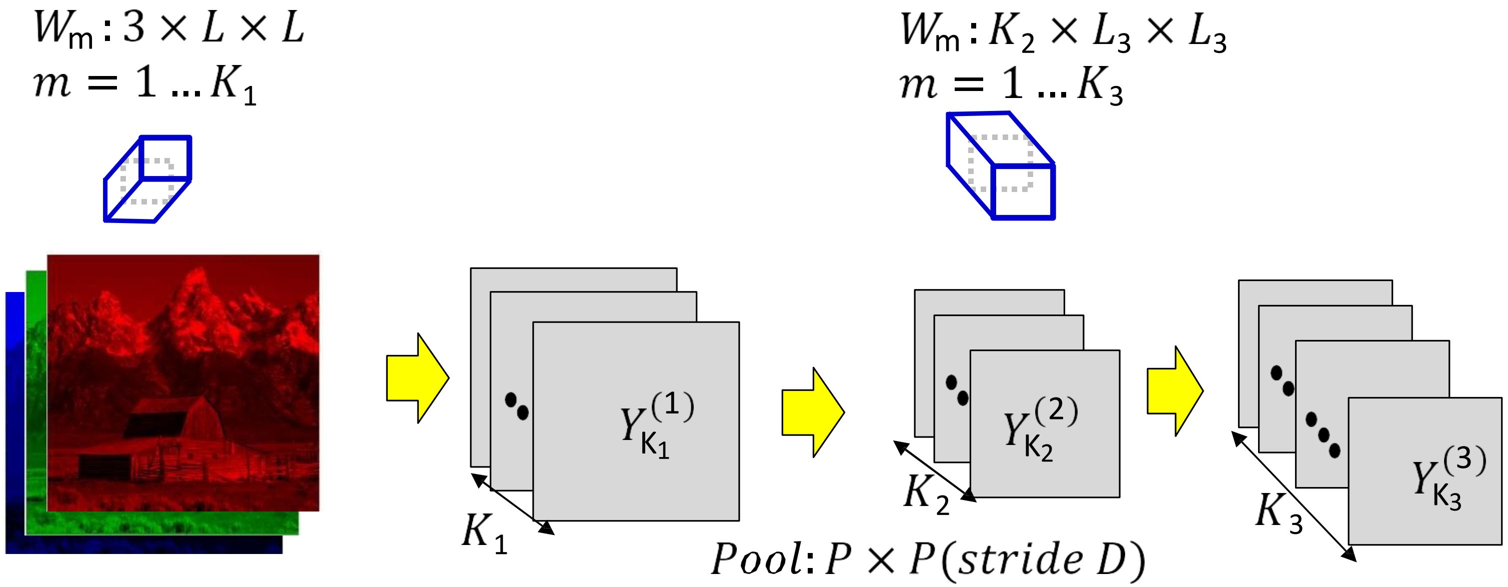


The layer pools PxP blocks of $Y_m^{(1)}$ into a single value
It employs a stride D between adjacent blocks

First pooling/downsampling layer: From each $P \times P$ block of each map, *pool* down to a single value

– For max pooling, during training keep track of which position had the highest value

Convolutional Neural Networks

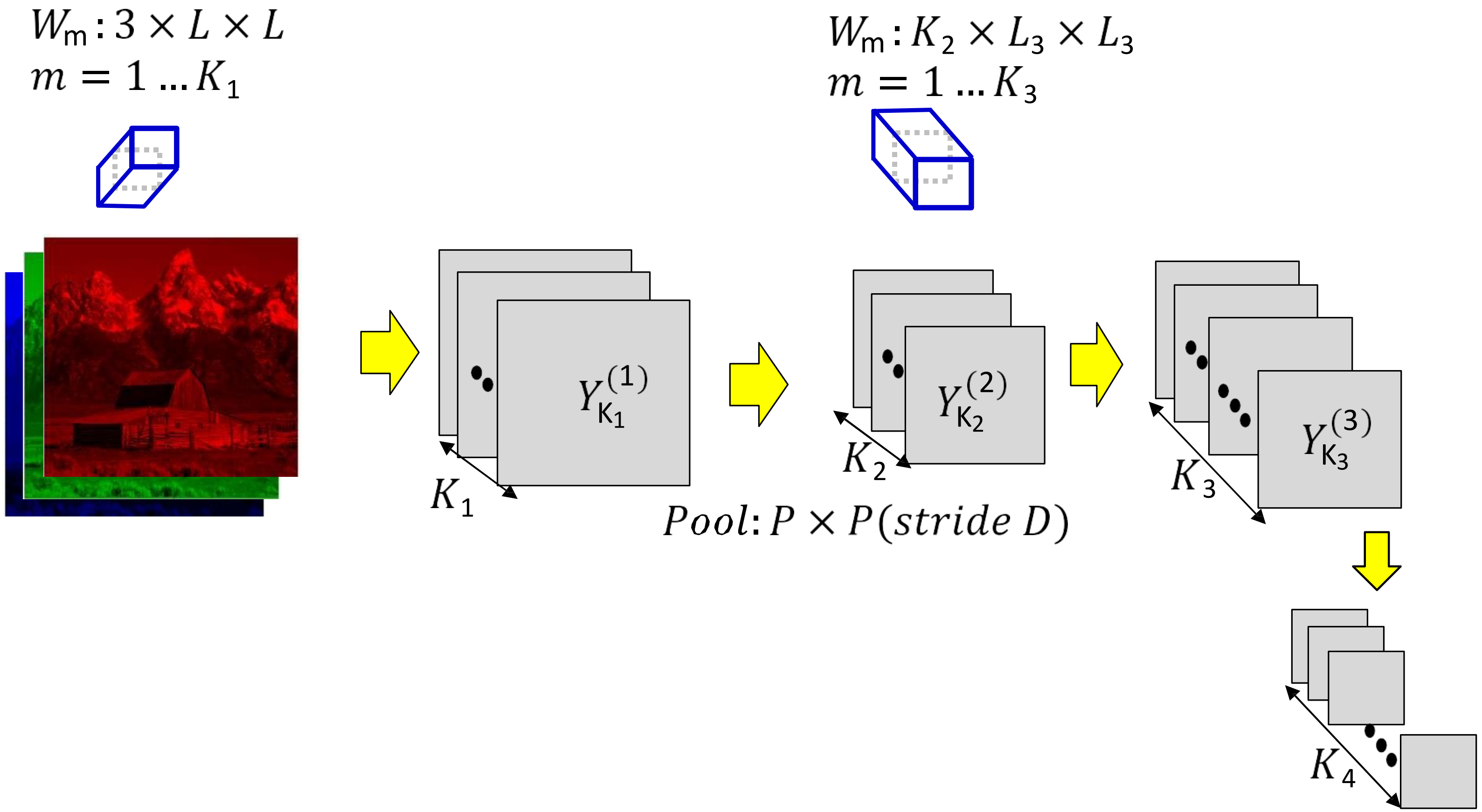


$$z_m^{(n)}(i,j) = \sum_{r=1}^{K_{n-1}} \sum_{k=1}^{L_n} \sum_{l=1}^{L_n} w_m^{(n)}(r,k,l)\, Y_r^{(n-1)}(i+k, j+l) + b_m^{(n)}$$

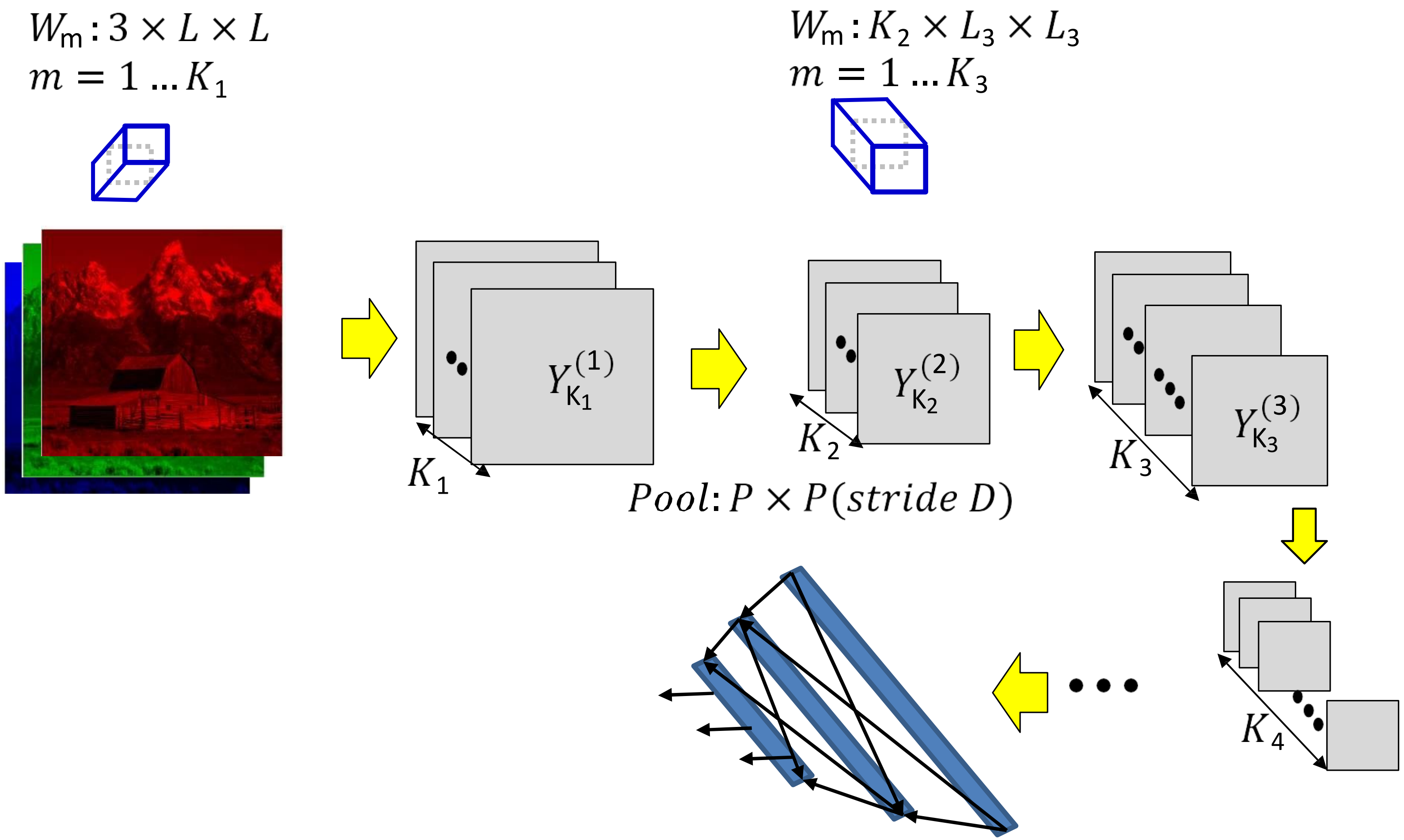
$$Y_m^{(n)}(i,j) = f\left(z_m^{(n)}(i,j)\right)$$

- **Second convolutional layer:** K_3 3-D filters resulting in K_3 2-D maps

Convolutional Neural Networks



Convolutional Neural Networks



- **This continues for several layers until the final convolved output is fed to a softmax**
  - **Or a full MLP**

The Size of the Layers

- Each convolution layer with stride 1 typically maintains the size of the image
  - With appropriate zero padding
  - If performed *without* zero padding it will decrease the size of the input

- Each convolution layer will generally *increase* the *number* of maps from the previous layer
  - In general, the number of convolutional filters increases with layers

- Each pooling layer with stride D *decreases* the *size* of the maps by a factor of D

- Filters within a layer must all be the same size, but sizes may vary with layer
  - Similarly for pooling

Parameters to choose (design choices)

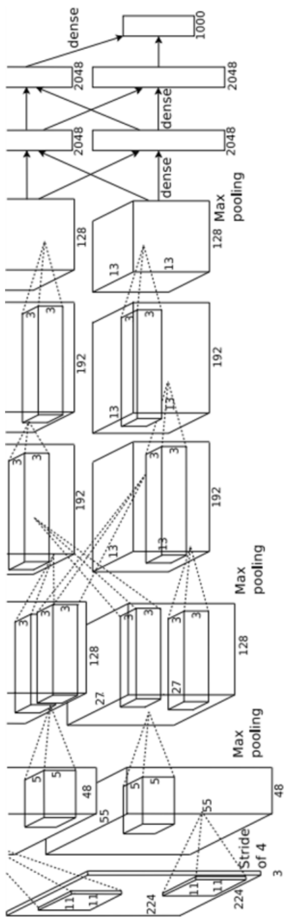
- Number of convolutional and downsampling layers
  - And arrangement (order in which they follow one another)

- For each convolution layer:
  - Number of filters K_i
  - Spatial extent of filter $L_i \times L_i$
    - The "depth" of the filter is fixed by the number of filters in the previous layer K_{i-1}
  - The stride S_i

- For each downsampling/pooling layer:
  - Spatial extent of filter $P_i \times P_i$
  - The stride D_i

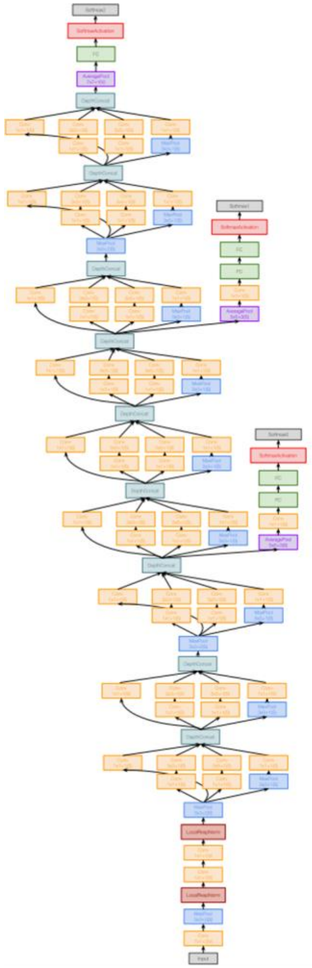
- For the final MLP:
  - Number of layers, and number of neurons in each layer

CNN Architectures

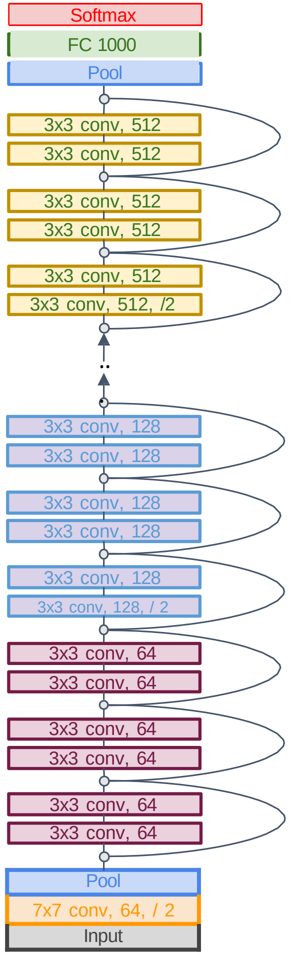


AlexNet

VGG



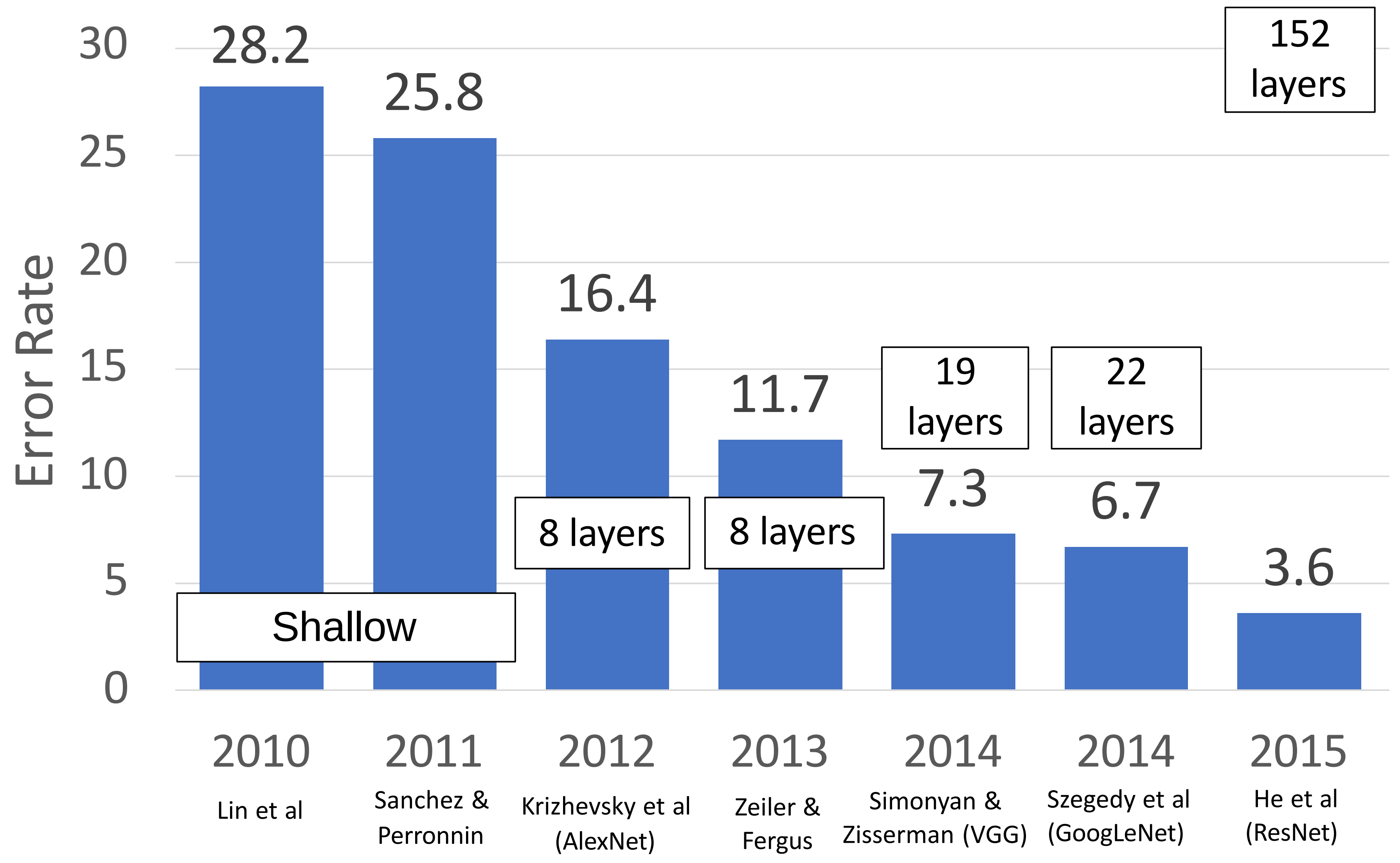
GoogLeNet



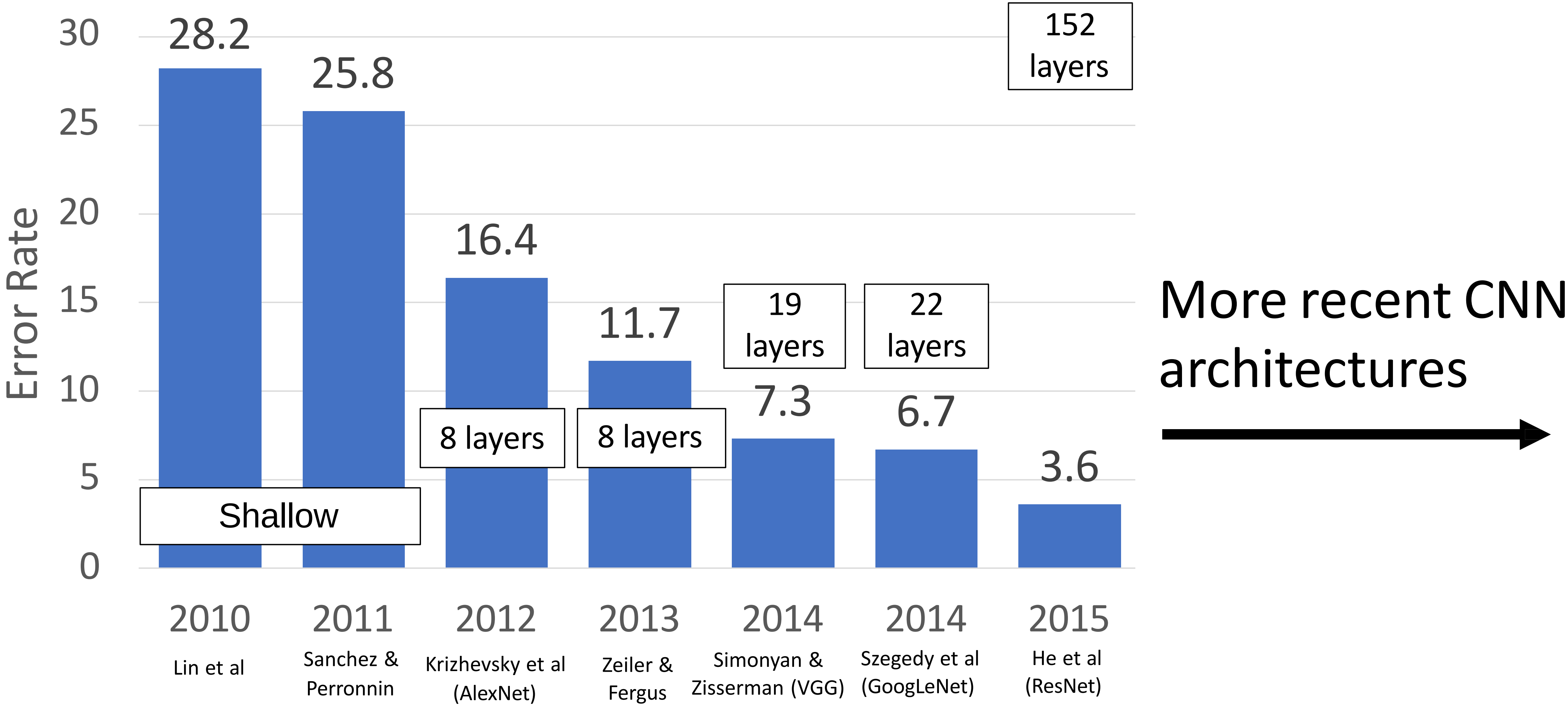
ResNet

This and the following slides are revised based on U Mich Class 598 by Justin Johnson

ImageNet Classification Challenge



ImageNet Classification Challenge

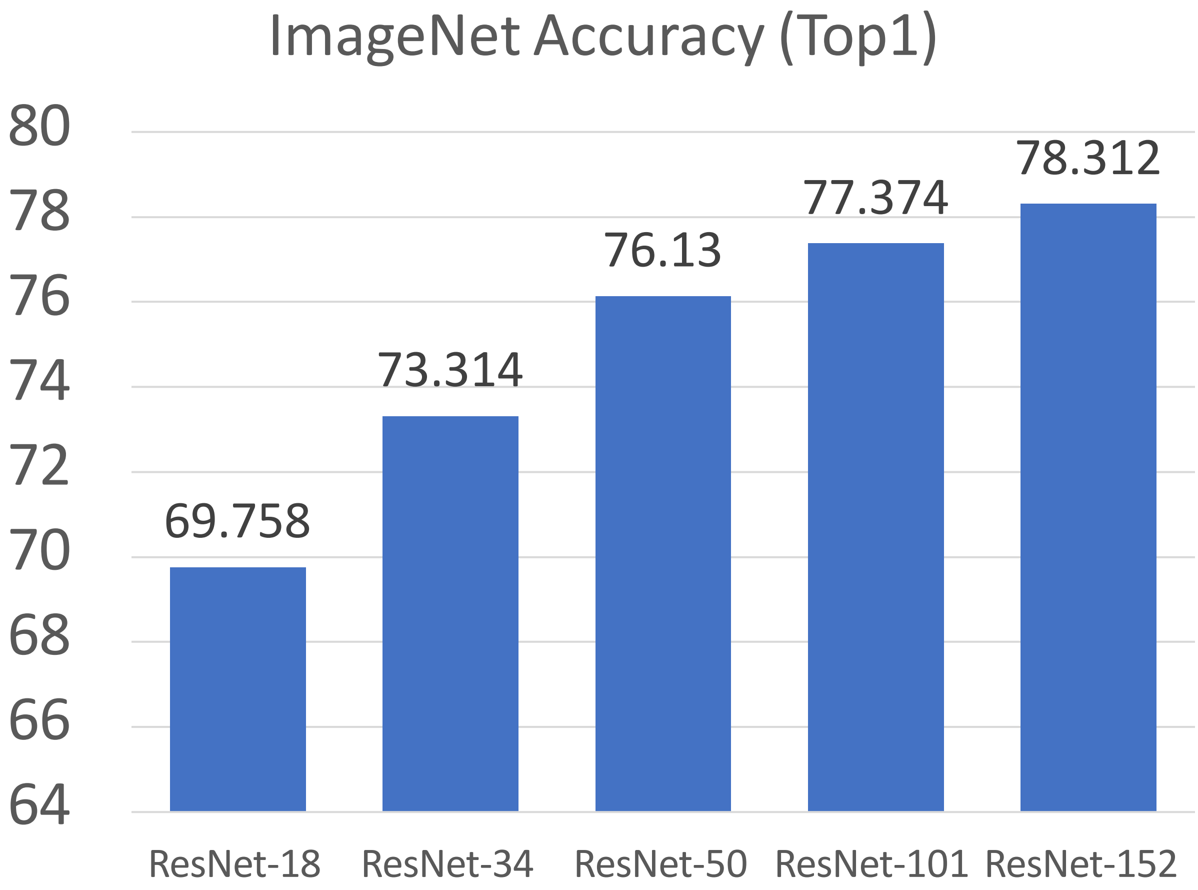


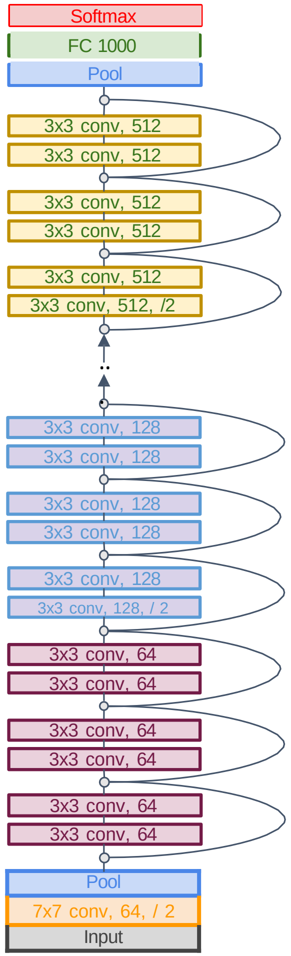
More recent CNN architectures

Post-ResNet Architectures

ResNet made it possible to increase accuracy with larger, deeper models

Many followup architectures emphasize **efficiency**: can we improve accuracy while controlling for model "complexity"?





Measures of Model Complexity

Parameters: How many learnable parameters does the model have?

Floating Point Operations (FLOPs): How many arithmetic operations does it take to compute the forward pass of the model?
Watch out, lots of subtlety here:

- Many papers only count operations in conv layers (ignore ReLU, pooling, BatchNorm)
- Most papers use “1 FLOP” = ”1 multiply and 1 addition” so dot product of two N-dim vectors takes N FLOPs
- Other sources (e.g. NVIDIA marketing material) count “1 multiply and one addition” = 2 FLOPs, so dot product of two N-dim vectors takes 2N FLOPs

Network Runtime: How long does a forward pass of the model take on real hardware?

Standard Convolution

- <u>Standard Convolution (groups=1)</u>

- All convolutional kernels touch all C_{in} channels of the input

- Input: C_{in} x H x W

- Weight: C_{out} x C_{in} x K x K

- Output: C_{out} x H’ x W’

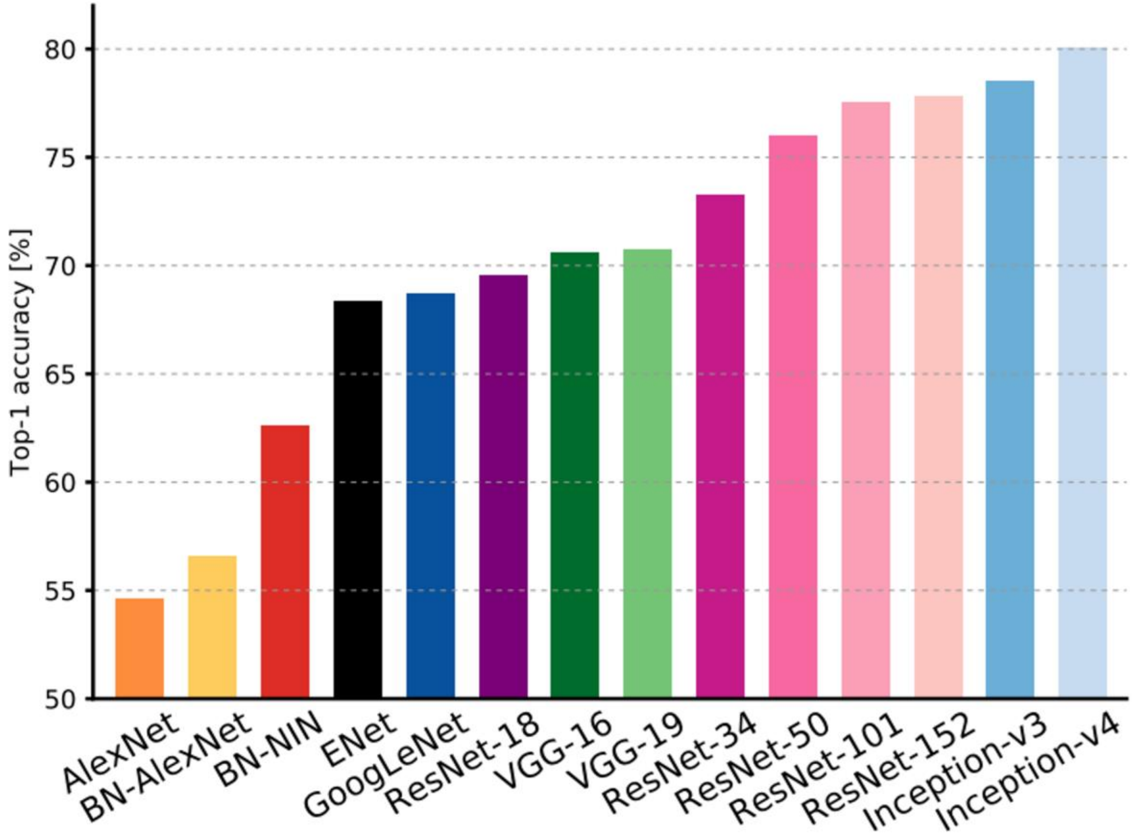
- Define: 1 FLOP” = ”1 multiply and 1 addition

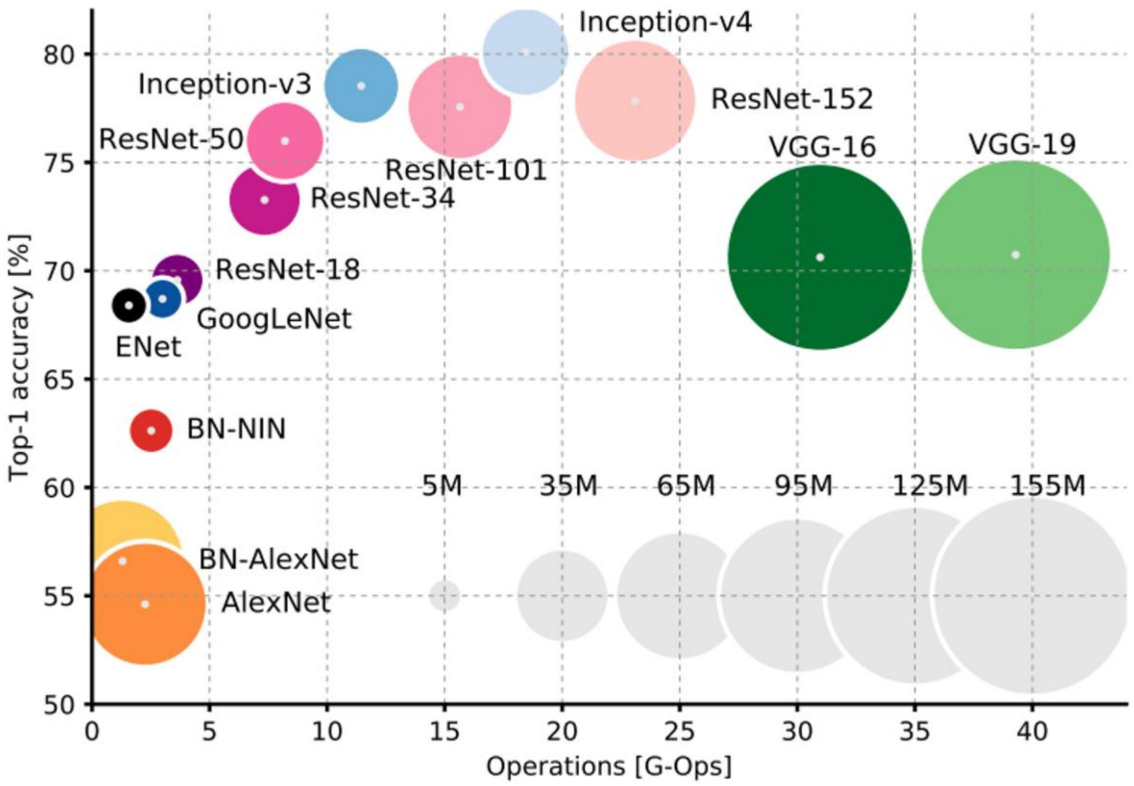
- For each output element, FLOP = C_{in} x K x K

- In total, the number of output elements is C_{out} x H’ x W’

- Hence, the total FLOPS is $C_{out}C_{in}K^2H'W'$

Comparing Complexity



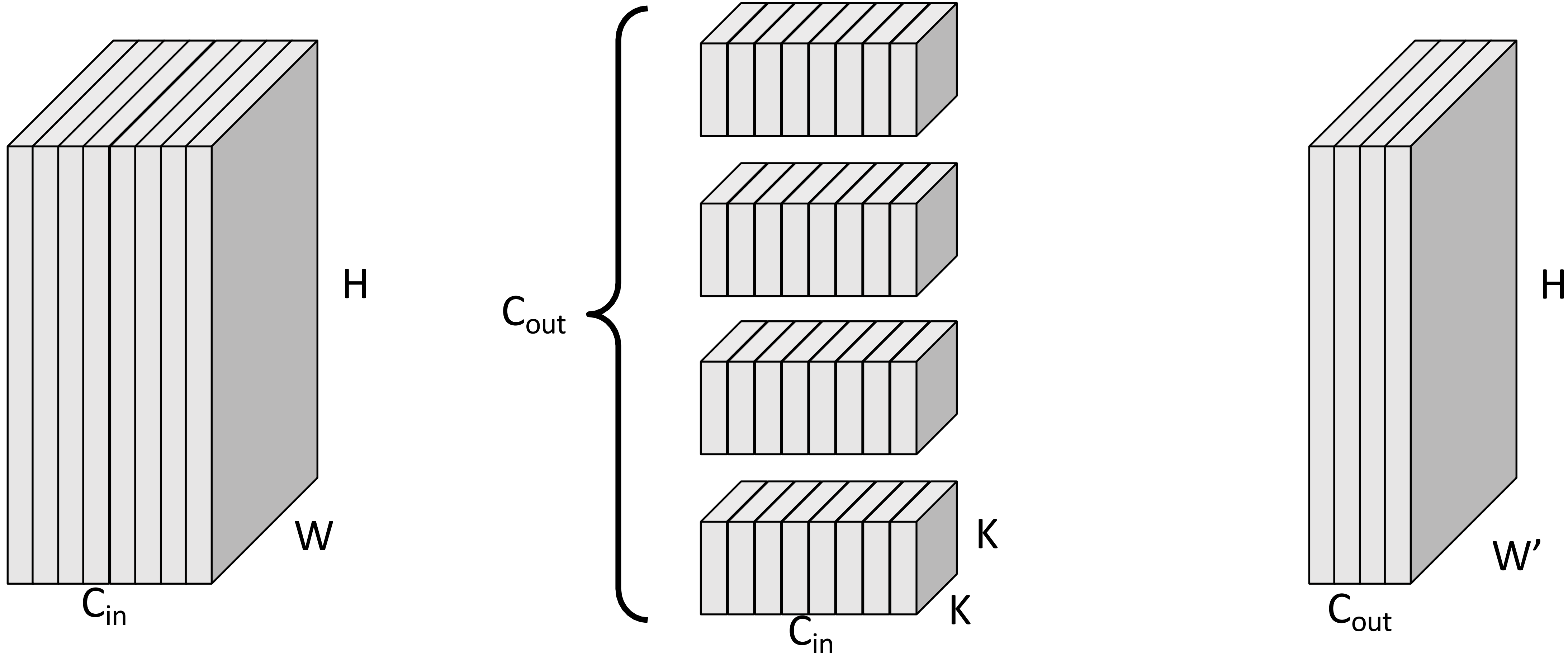


Canziani et al, "An analysis of deep neural network models for practical applications", 2017

Key ingredient: Grouped / Separable convolution

Convolution Layer

Each filter has the same number of channels as the input



Input: C_{in} x H x W

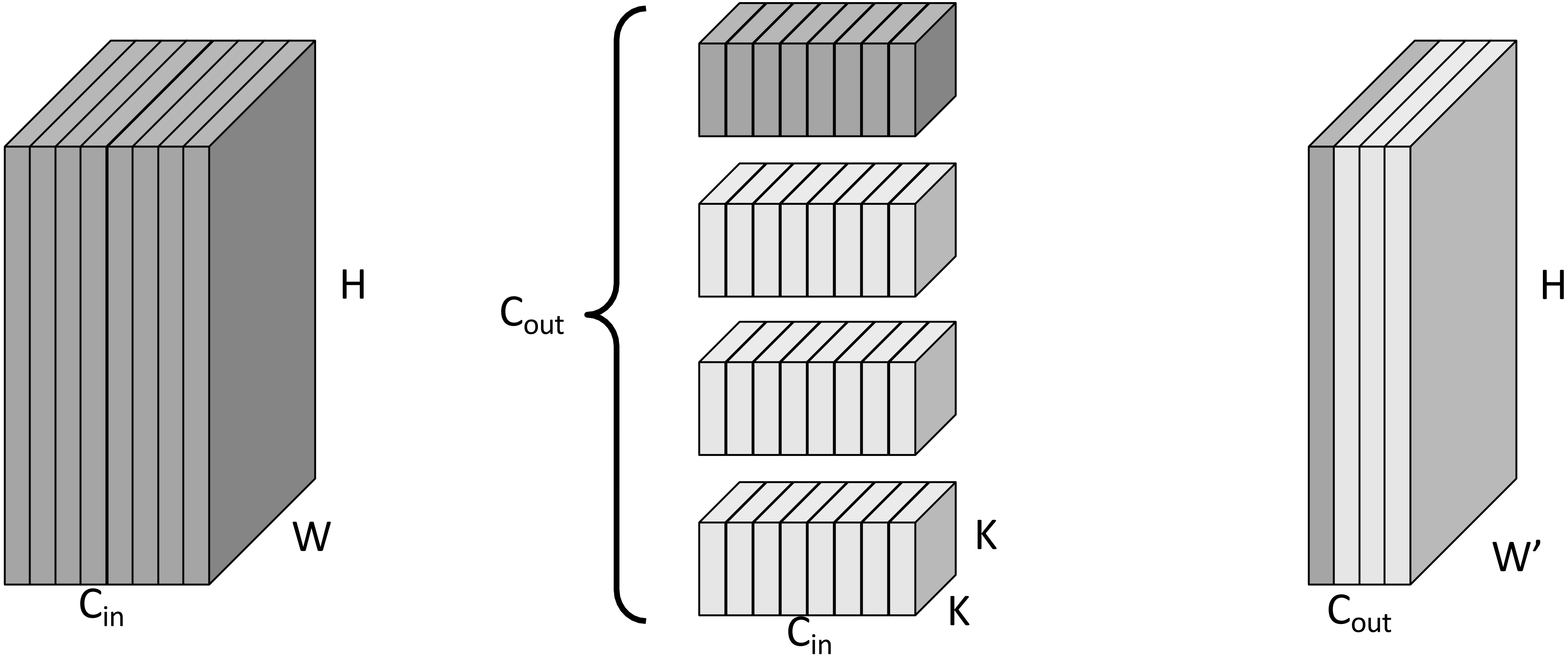
Weights: C_{out} x C_{in} x K x K

Output: C_{out} x H’ x W’

Recall: Convolution Layer

Each filter has the same number of channels as the input

Each plane of the output depends on the full input and one filter



Input: C_{in} x H x W

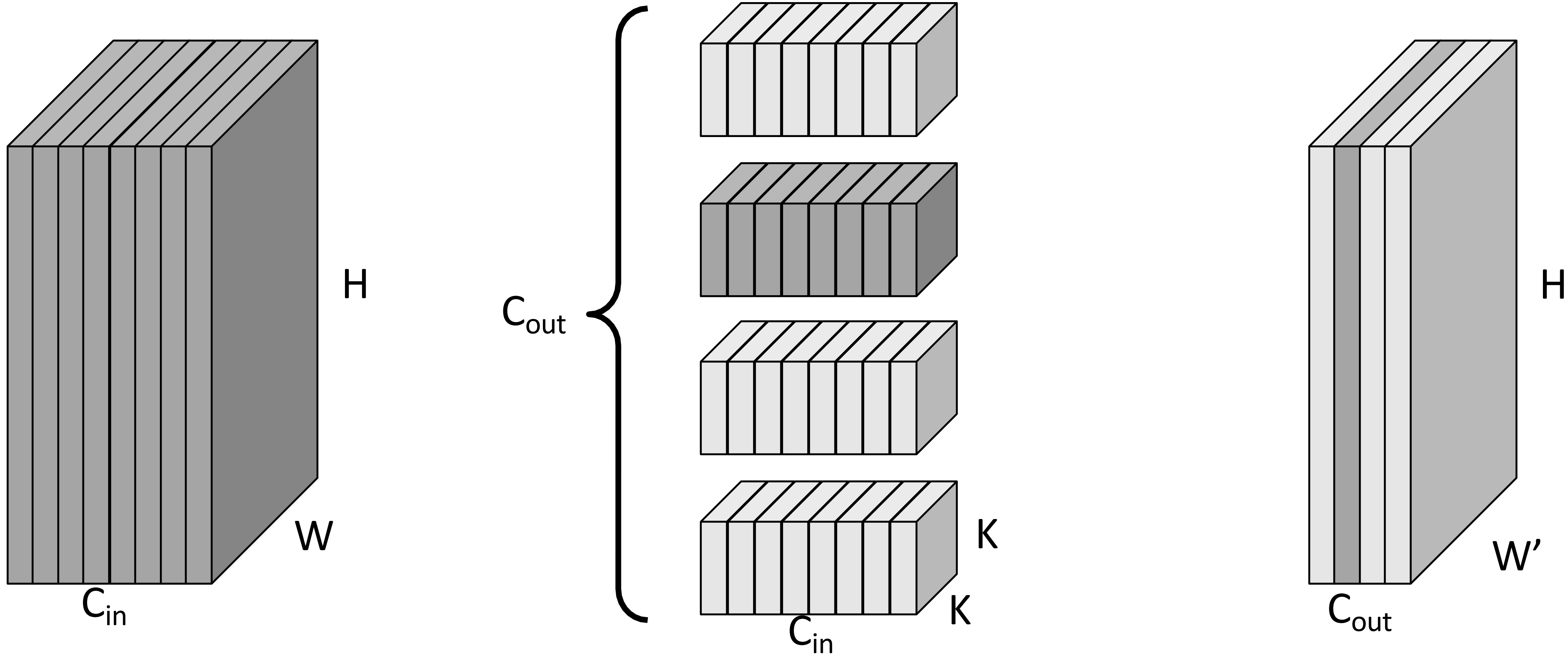
Weights: C_{out} x C_{in} x K x K

Output: C_{out} x H’ x W’

Recall: Convolution Layer

Each filter has the same number of channels as the input

Each plane of the output depends on the full input and one filter



Input: C_{in} x H x W

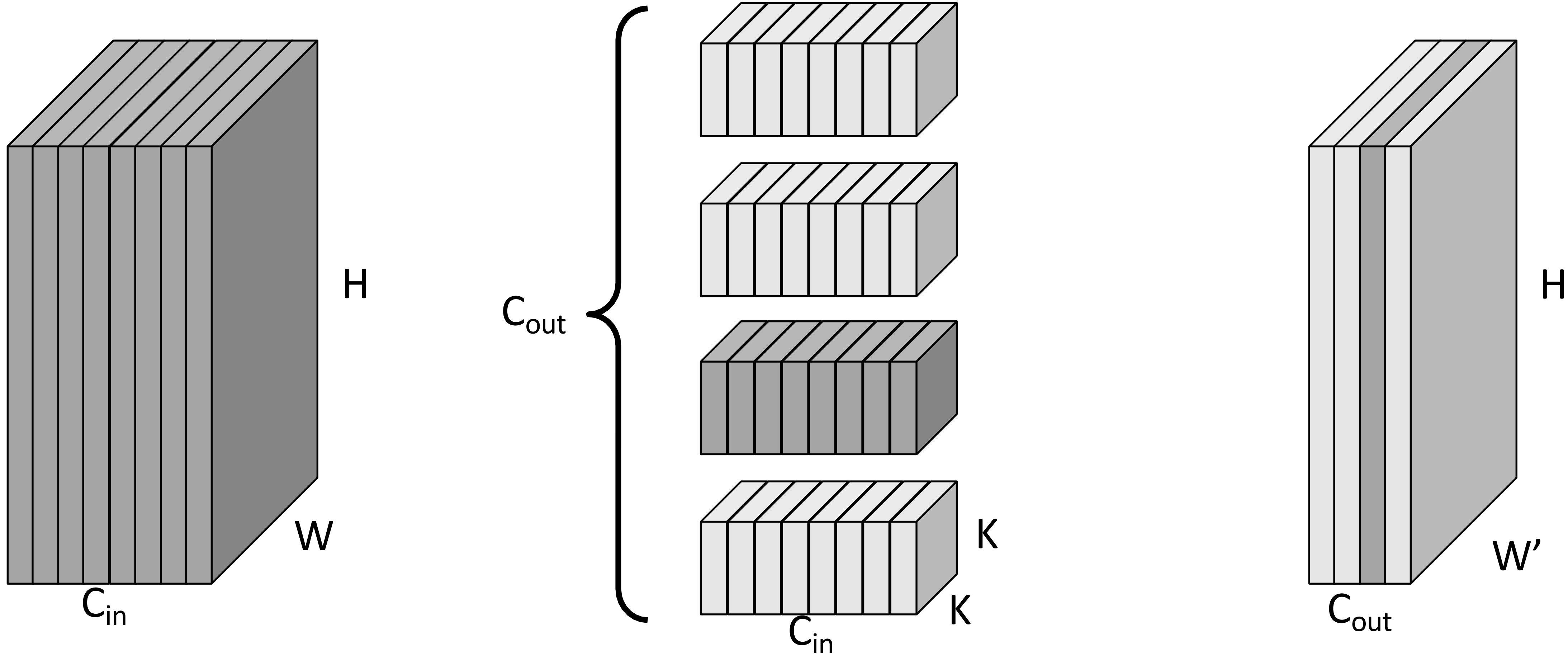
Weights: C_{out} x C_{in} x K x K

Output: C_{out} x H' x W'

Recall: Convolution Layer

Each filter has the same number of channels as the input

Each plane of the output depends on the full input and one filter



Input: C_{in} x H x W

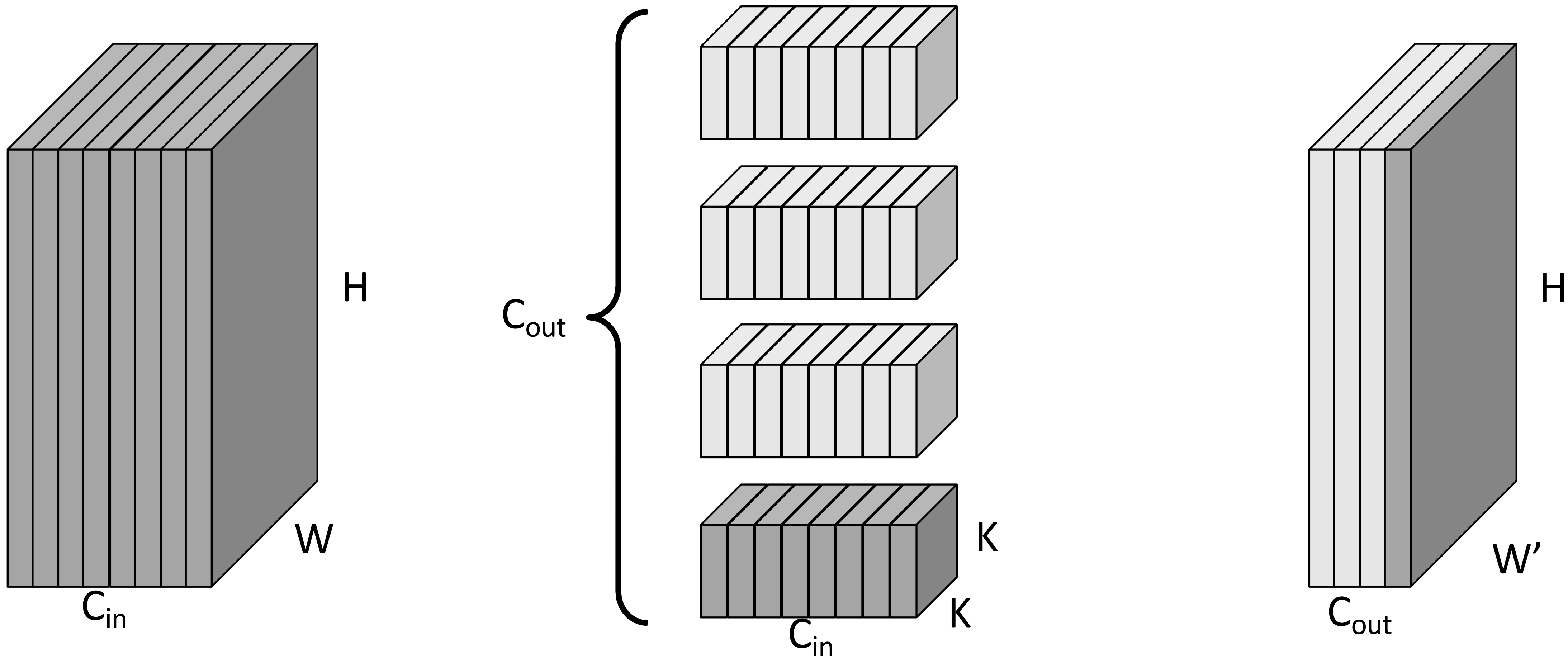
Weights: C_{out} x C_{in} x K x K

Output: C_{out} x H' x W'

Recall: Convolution Layer

Each filter has the same number of channels as the input

Each plane of the output depends on the full input and one filter



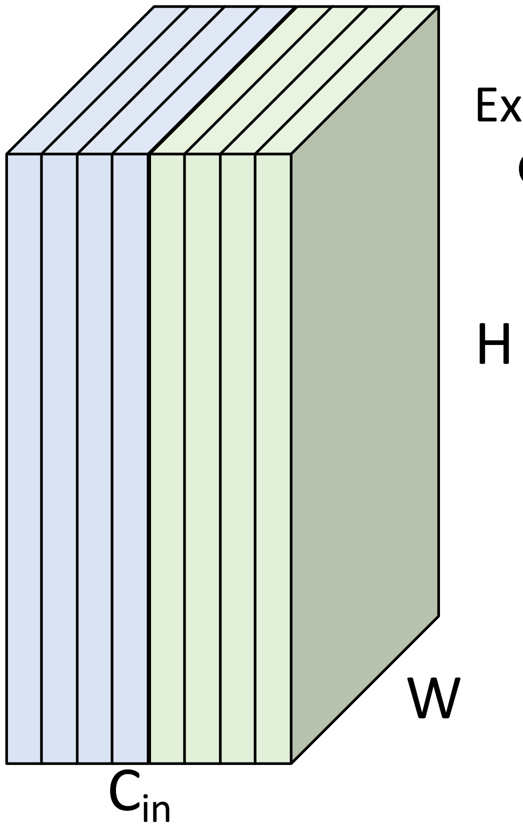
Input: C_{in} x H x W

Weights: C_{out} x C_{in} x K x K

Output: C_{out} x H' x W'

Grouped Convolution

Divide channels of input into G **groups** with (C_{in}/G) channels each



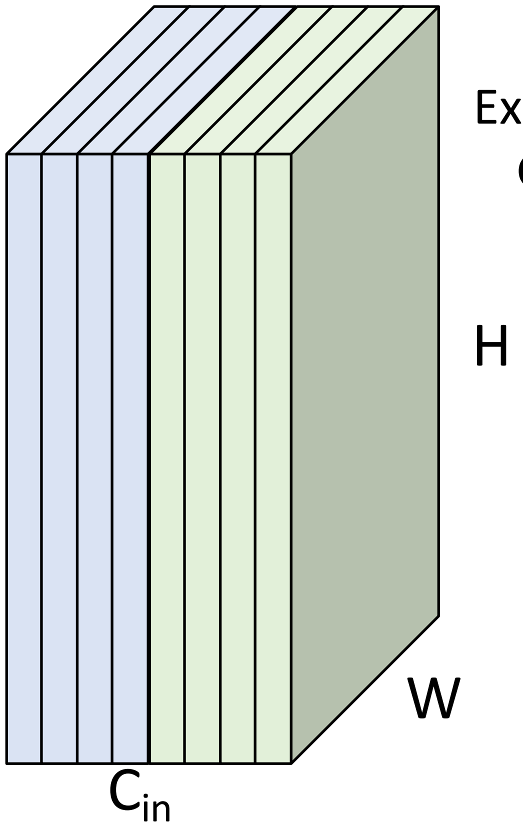
Example:
G = 2

Input: C_{in} x H x W

Grouped Convolution

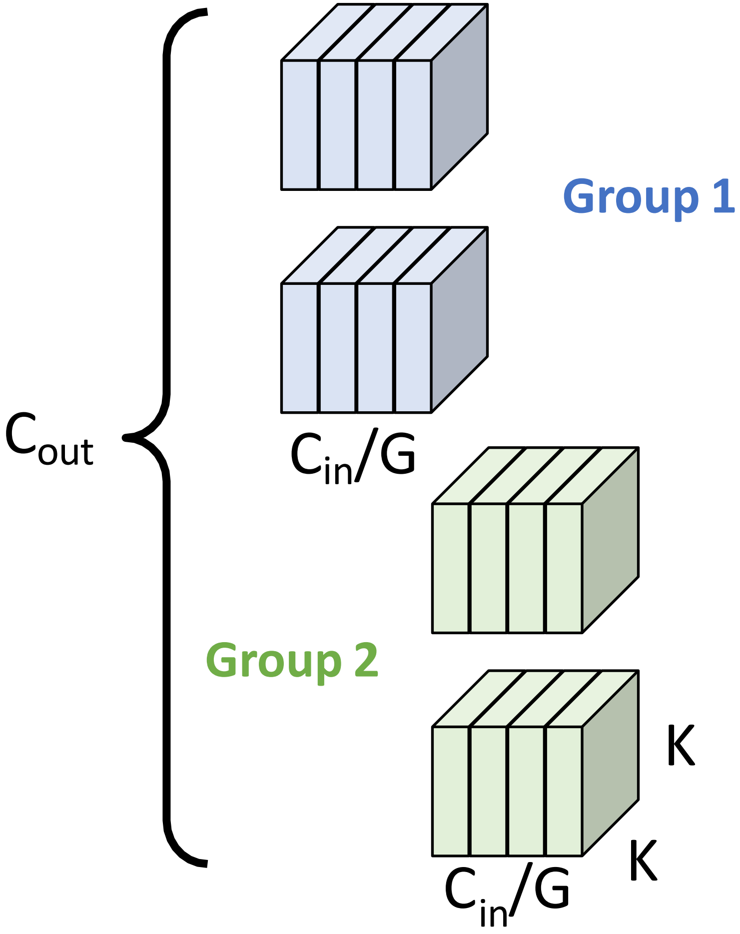
Divide channels of input into G **groups** with (C_{in}/G) channels each

Example: G = 2



Input: C_{in} x H x W

Divide filters into G groups; each group looks at a **subset** of input channels



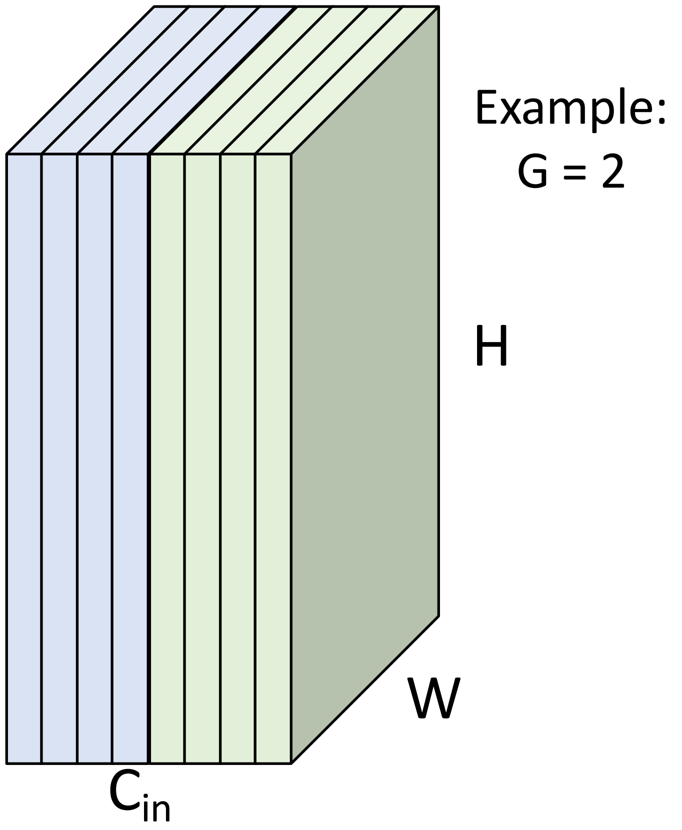
Weights: C_{out} x (C_{in}/G) x K x K

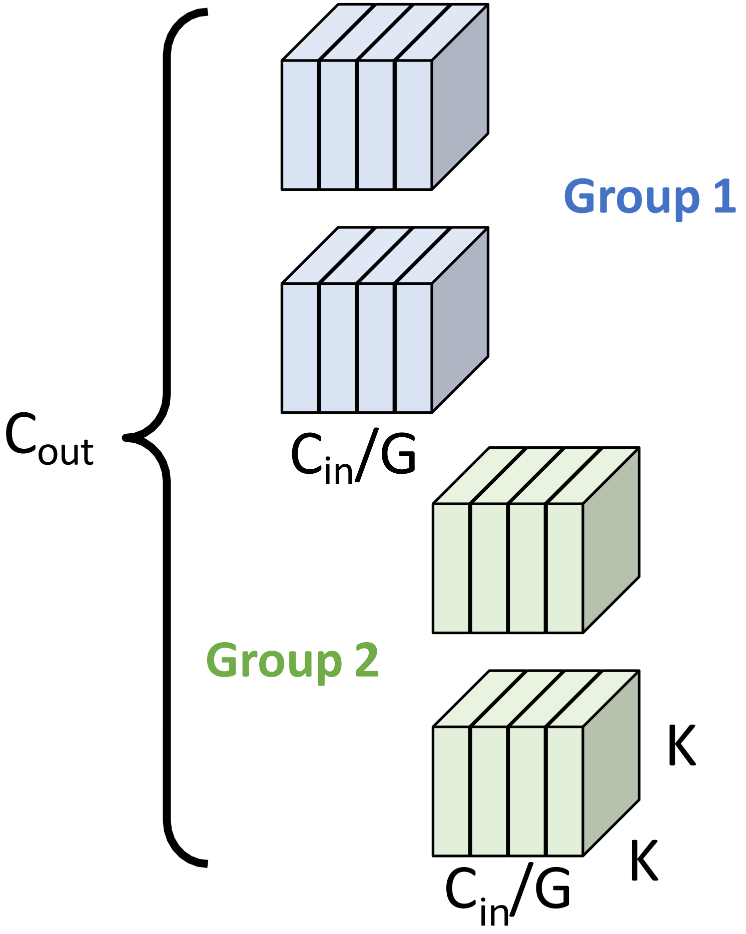
Grouped Convolution

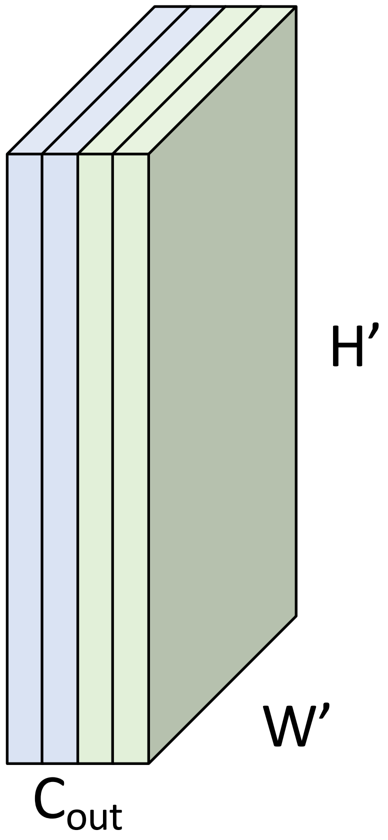
Divide channels of input into G **groups** with (C_{in}/G) channels each

Divide filters into G groups; each group looks at a **subset** of input channels

Each plane of the output depends on one filter and a **subset** of the input channels







Input: C_{in} x H x W

Weights: C_{out} x (C_{in}/G) x K x K

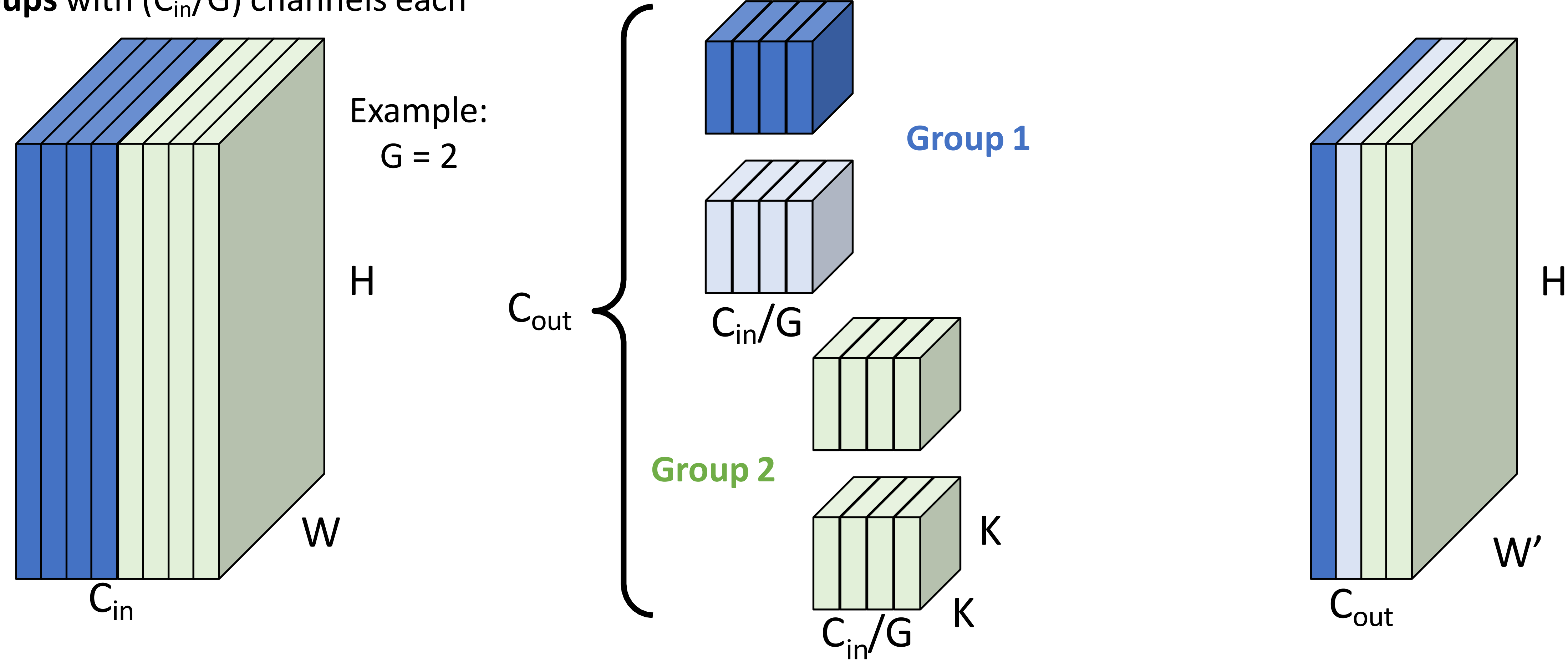
Output: C_{out} x H’ x W’

Grouped Convolution

Divide channels of input into G **groups** with (C_{in}/G) channels each

Divide filters into G groups; each group looks at a **subset** of input channels

Each plane of the output depends on one filter and a **subset** of the input channels



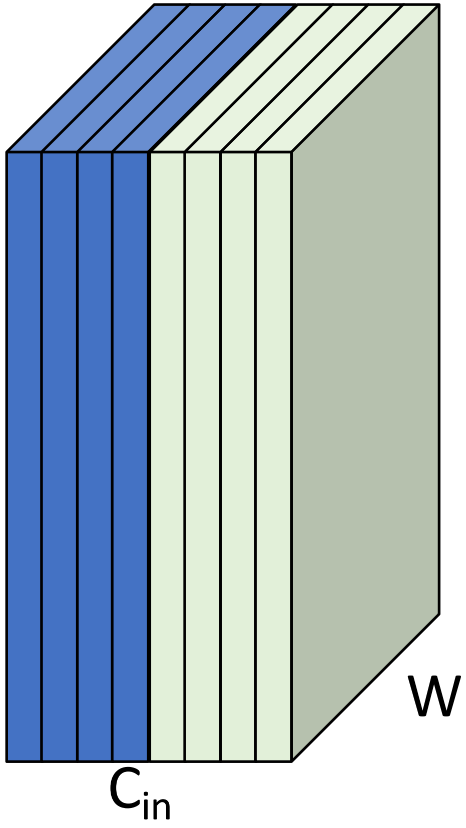
Input: $C_{in} \times H \times W$

Weights: $C_{out} \times (C_{in}/G) \times K \times K$

Output: $C_{out} \times H' \times W'$

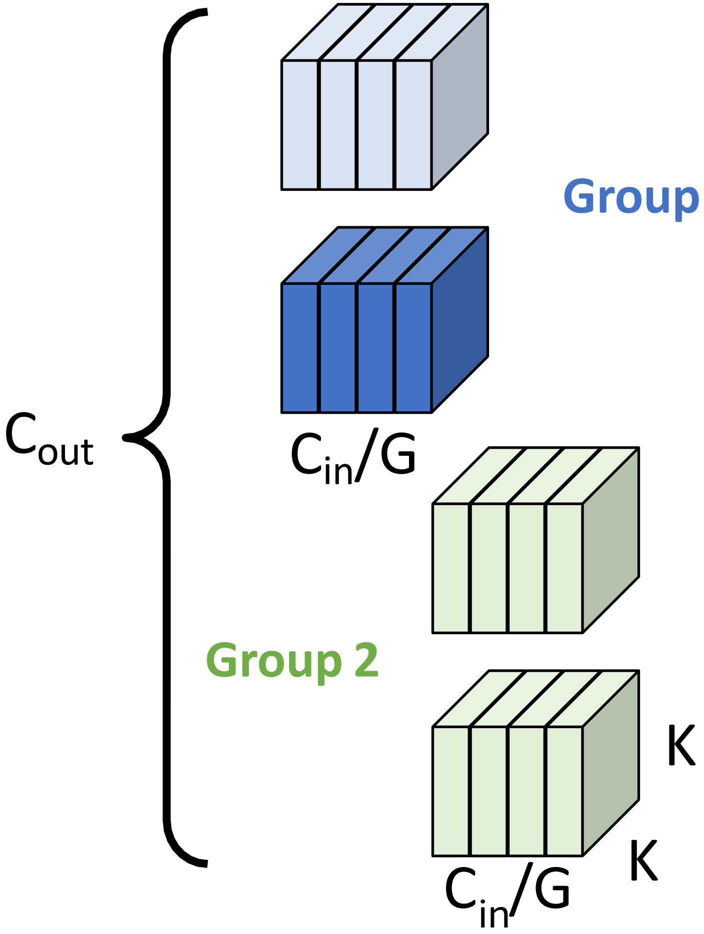
Grouped Convolution

Divide channels of input into G **groups** with (C_{in}/G) channels each

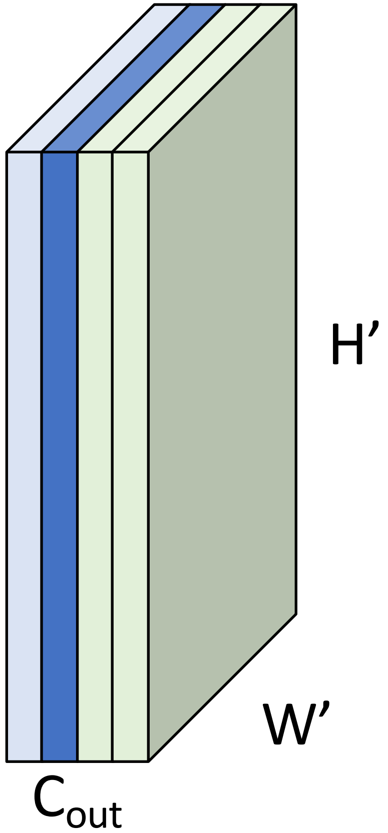


Example:
G = 2

Divide filters into G groups; each group looks at a **subset** of input channels



Each plane of the output depends on one filter and a **subset** of the input channels



Input: C_{in} x H x W

Weights: C_{out} x (C_{in}/G) x K x K

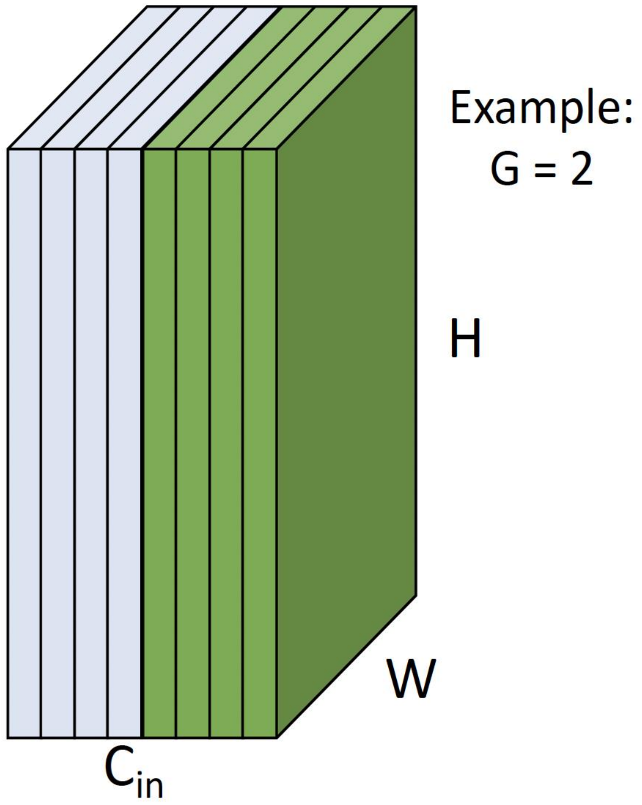
Output: C_{out} x H' x W'

Grouped Convolution

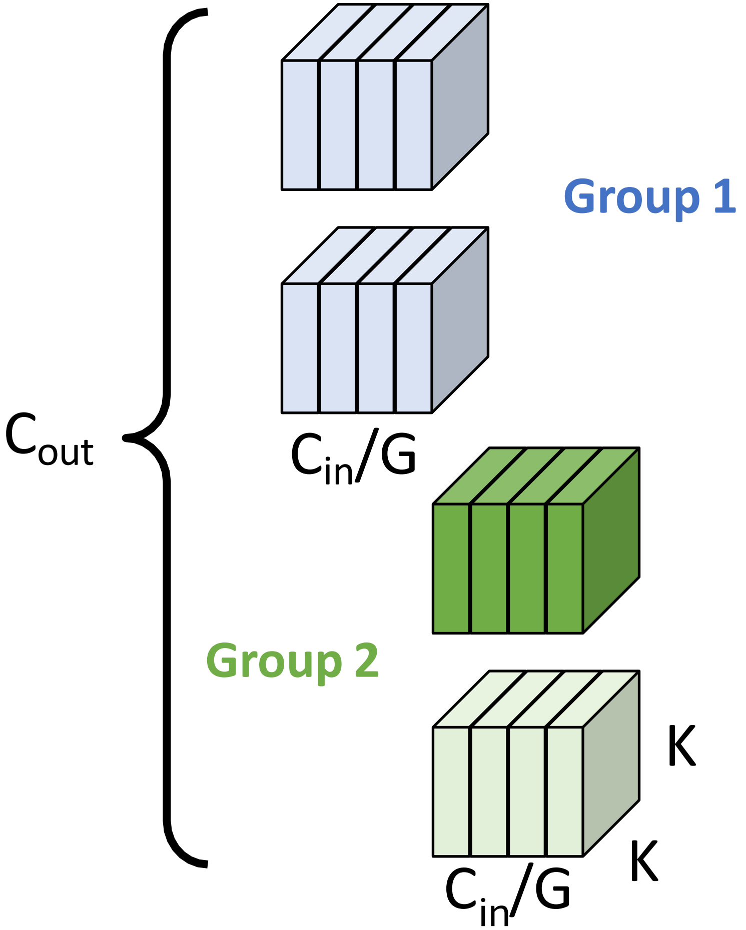
Divide channels of input into G **groups** with (C_{in}/G) channels each

Divide filters into G groups; each group looks at a **subset** of input channels

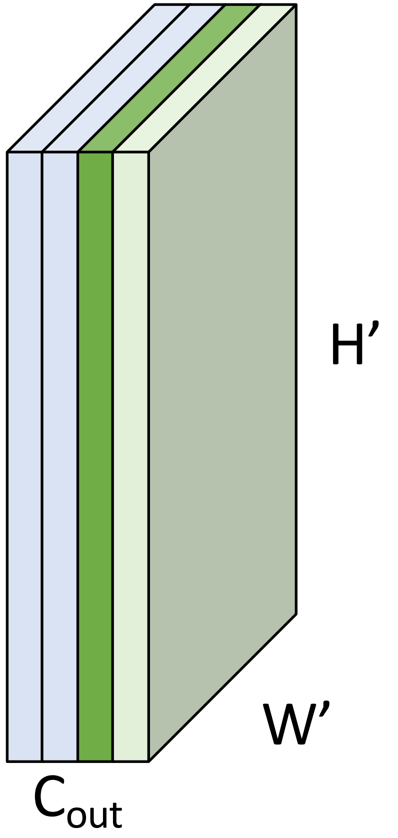
Each plane of the output depends on one filter and a **subset** of the input channels



Input: C_{in} x H x W



Weights: C_{out} x (C_{in}/G) x K x K



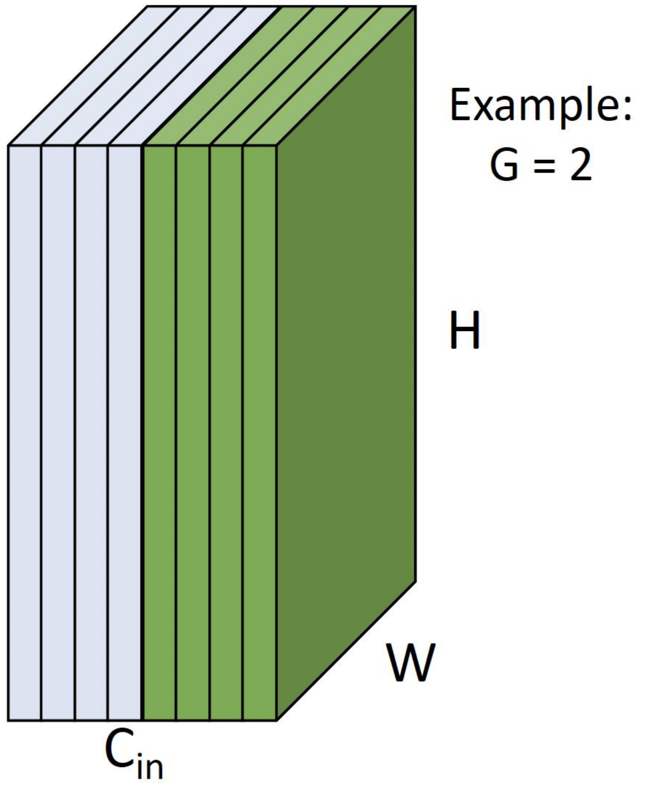
Output: C_{out} x H’ x W’

Grouped Convolution

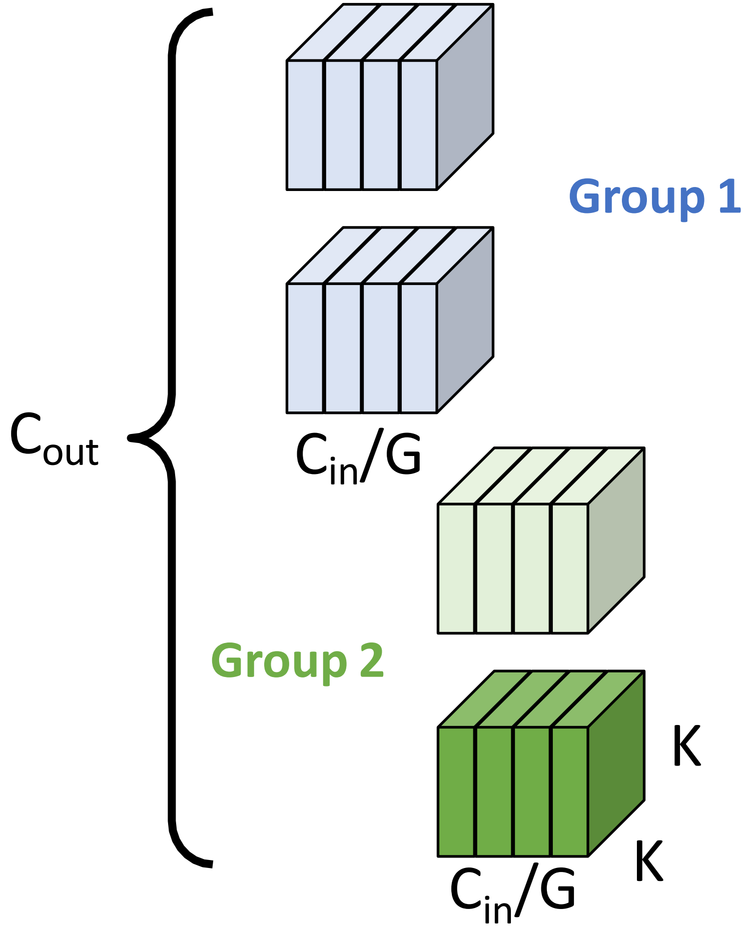
Divide channels of input into G **groups** with (C_{in}/G) channels each

Divide filters into G groups; each group looks at a **subset** of input channels

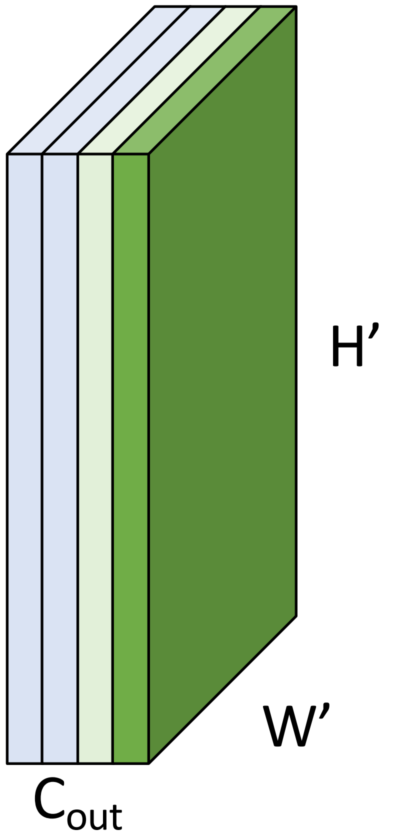
Each plane of the output depends on one filter and a **subset** of the input channels



Input: C_{in} x H x W



Weights: C_{out} x (C_{in}/G) x K x K



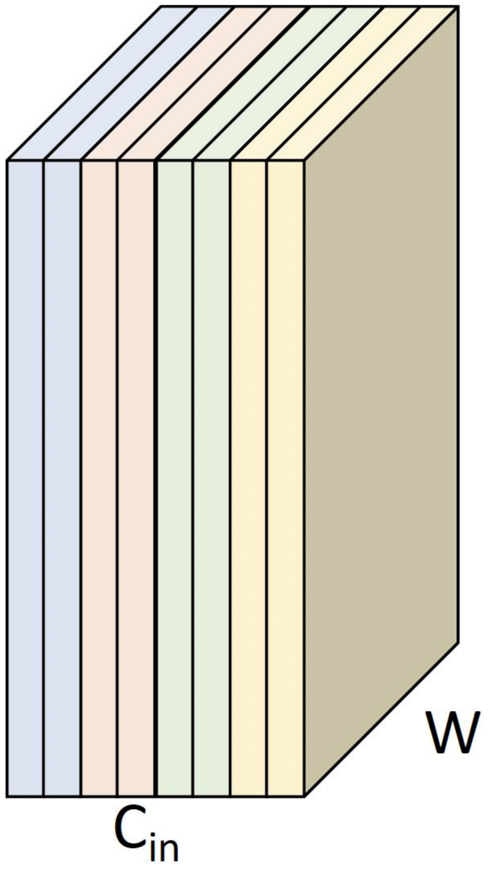
Output: C_{out} x H’ x W’

Grouped Convolution

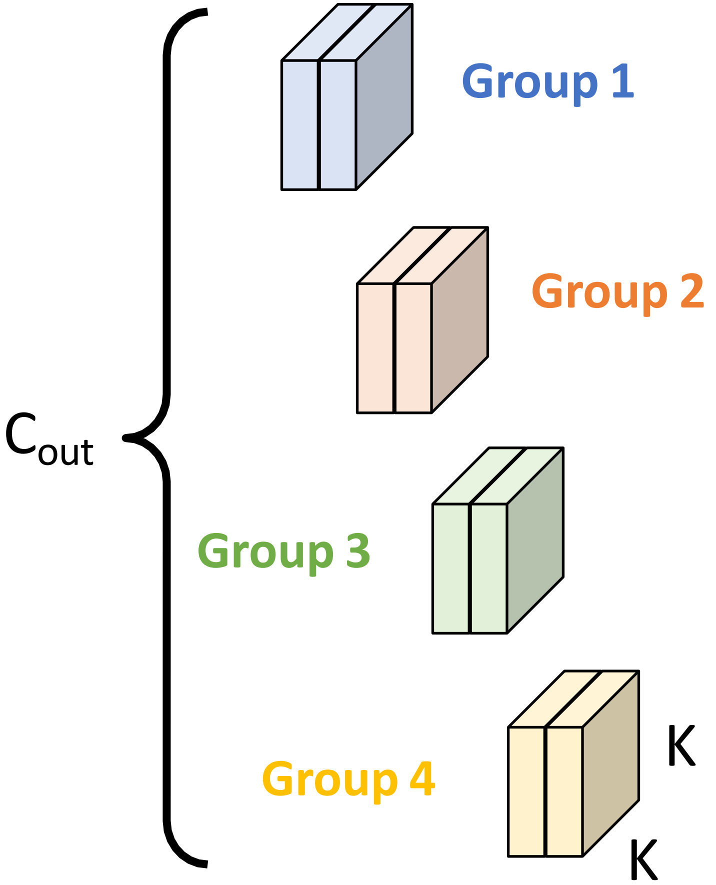
Divide channels of input into G **groups** with (C_{in}/G) channels each

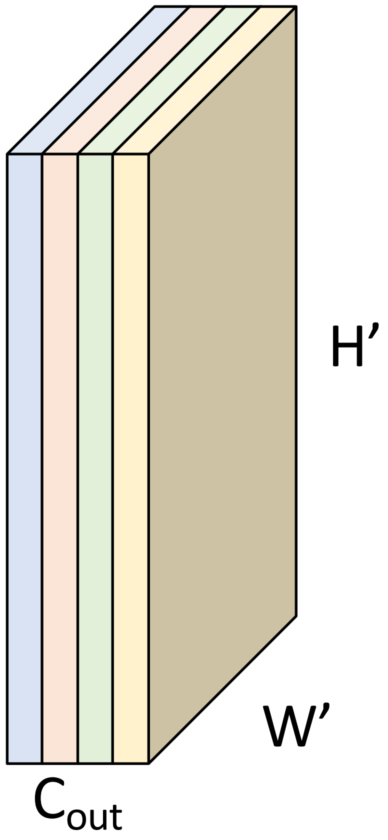
Divide filters into G groups; each group looks at a **subset** of input channels

Each plane of the output depends on one filter and a **subset** of the input channels



Example:
G = 4





Input: C_{in} x H x W

Weights: C_{out} x (C_{in}/G) x K x K

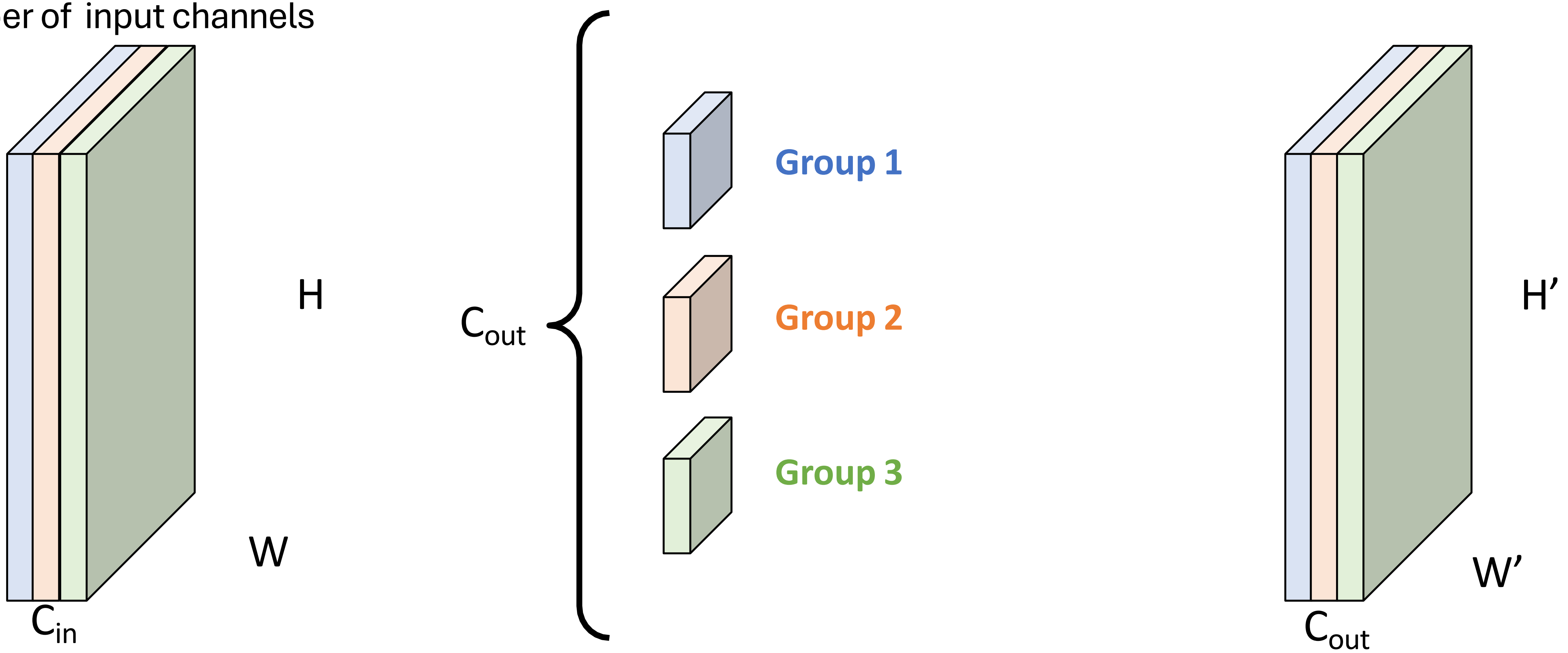
Output: C_{out} x H’ x W’

Special Case: Depthwise Convolution

Output only mixes *spatial* information from input; *channel* information not mixed

Number of groups equals number of input channels

Common to also set $C_{out} = G$



Input: $C_{in} \times H \times W$

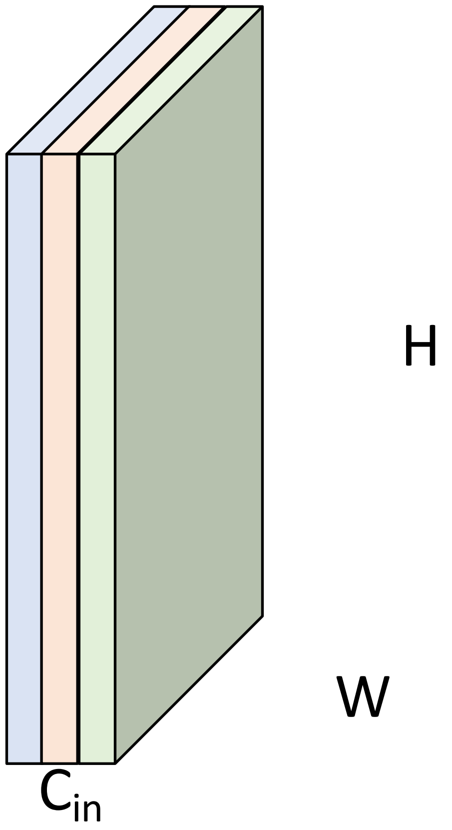
Weights: $C_{out} \times 1 \times K \times K$

Output: $C_{out} \times H' \times W'$

Special Case: Depthwise Convolution

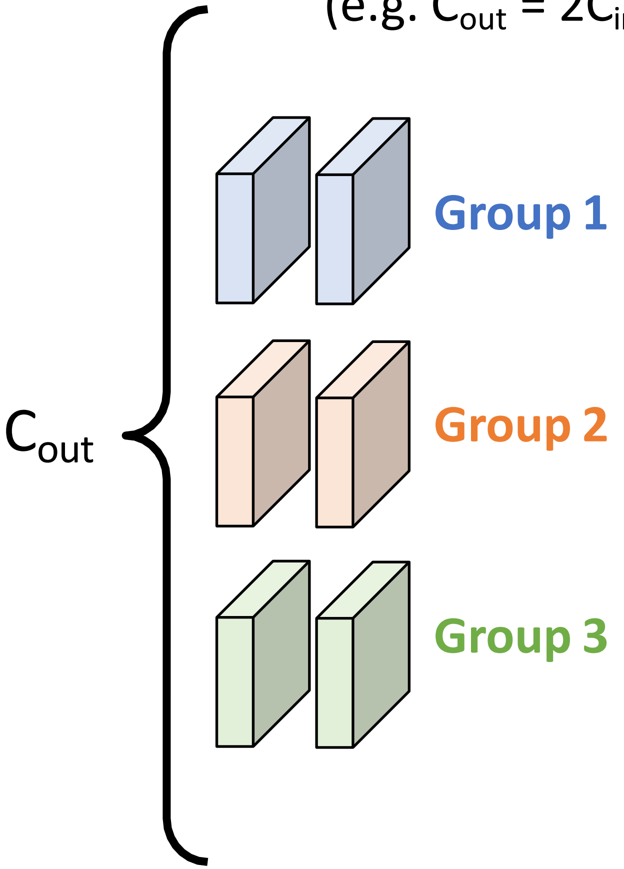
Output only mixes *spatial* information from input; *channel* information not mixed

Number of groups equals number of input channels

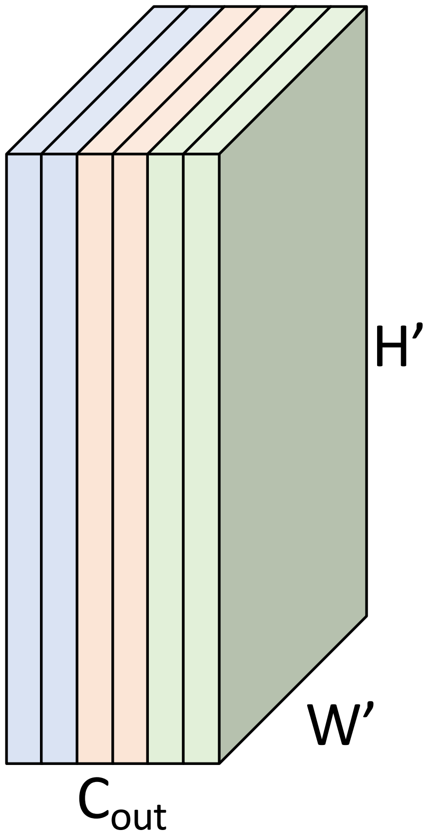


Input: C_{in} x H x W

Can still have multiple filters per group (e.g. $C_{out} = 2C_{in}$)



Weights: C_{out} x 1 x K x K



Output: C_{out} x H' x W'

Grouped Convolution vs Standard Convolution

<u>Grouped Convolution (G groups)</u>:
G parallel conv layers; each “sees” C_{in}/G input channels and produces C_{out}/G output channels

Input: C_{in} x H x W
Split to G x [(C_{in} / G) x H x W]
Weight: G x (C_{out} / G) x (C_{in} x G) x K x K
G parallel convolutions
Output: G x [(C_{out} / G) x H’ x W’]
Concat to C_{out} x H’ x W’
FLOPs: $C_{out}C_{in}K^2H'W'/G$

- <u>Standard Convolution (groups=1)</u>
- All convolutional kernels touch all C_{in} channels of the input
- Input: C_{in} x H x W
- Weight: C_{out} x C_{in} x K x K
- Output: C_{out} x H’ x W’
- FLOPs: $C_{out}C_{in}K^2H'W'$

Using G groups reduces FLOPs by a factor of G!

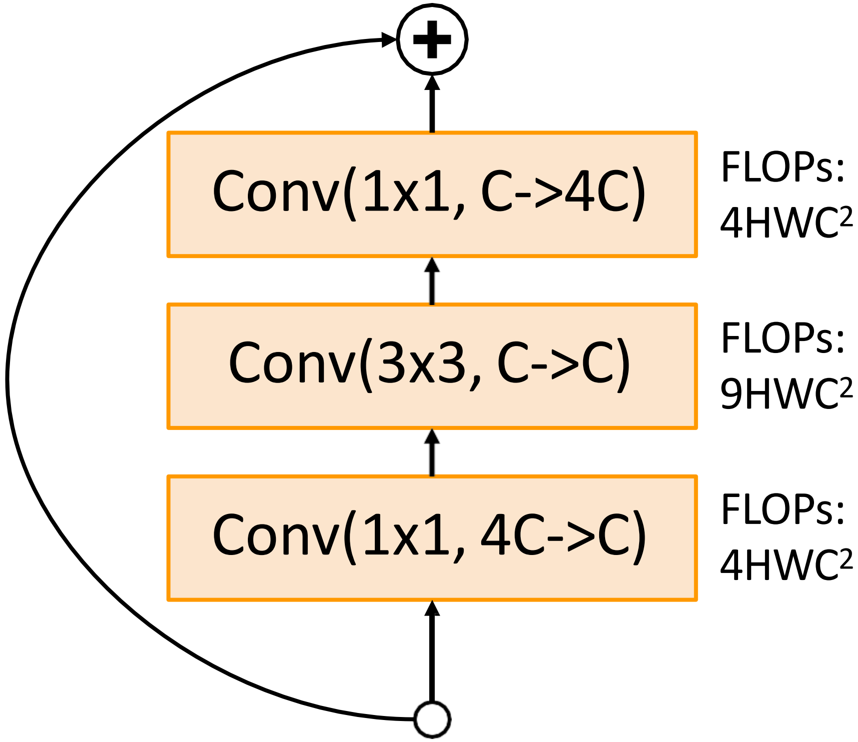
Grouped Convolution in PyTorch

PyTorch convolution gives an option for groups!

Conv2d

CLASS `torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros')` [SOURCE]

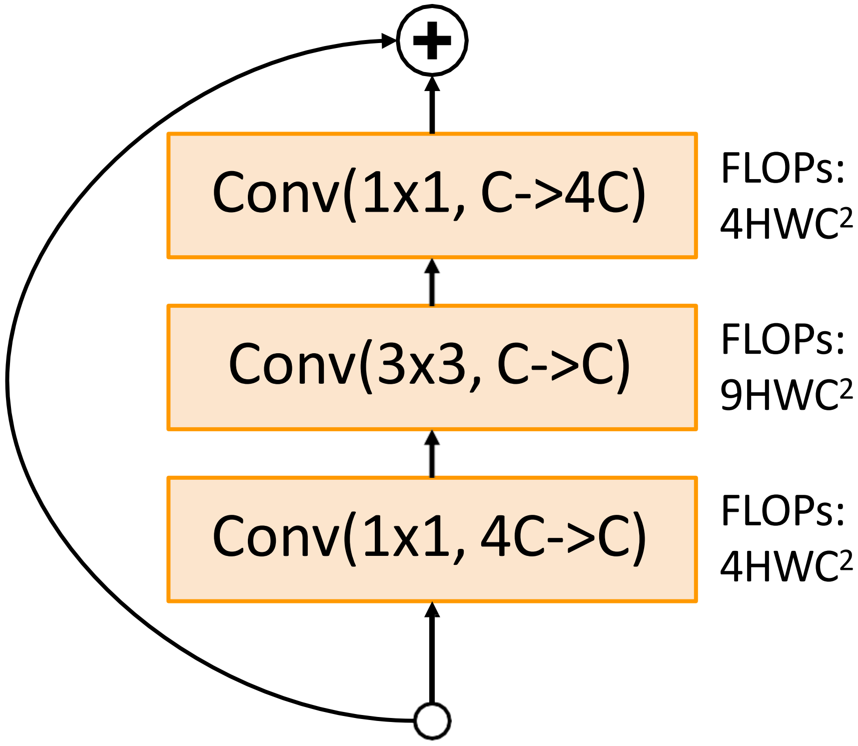
Improving ResNets



"Bottleneck" Residual block

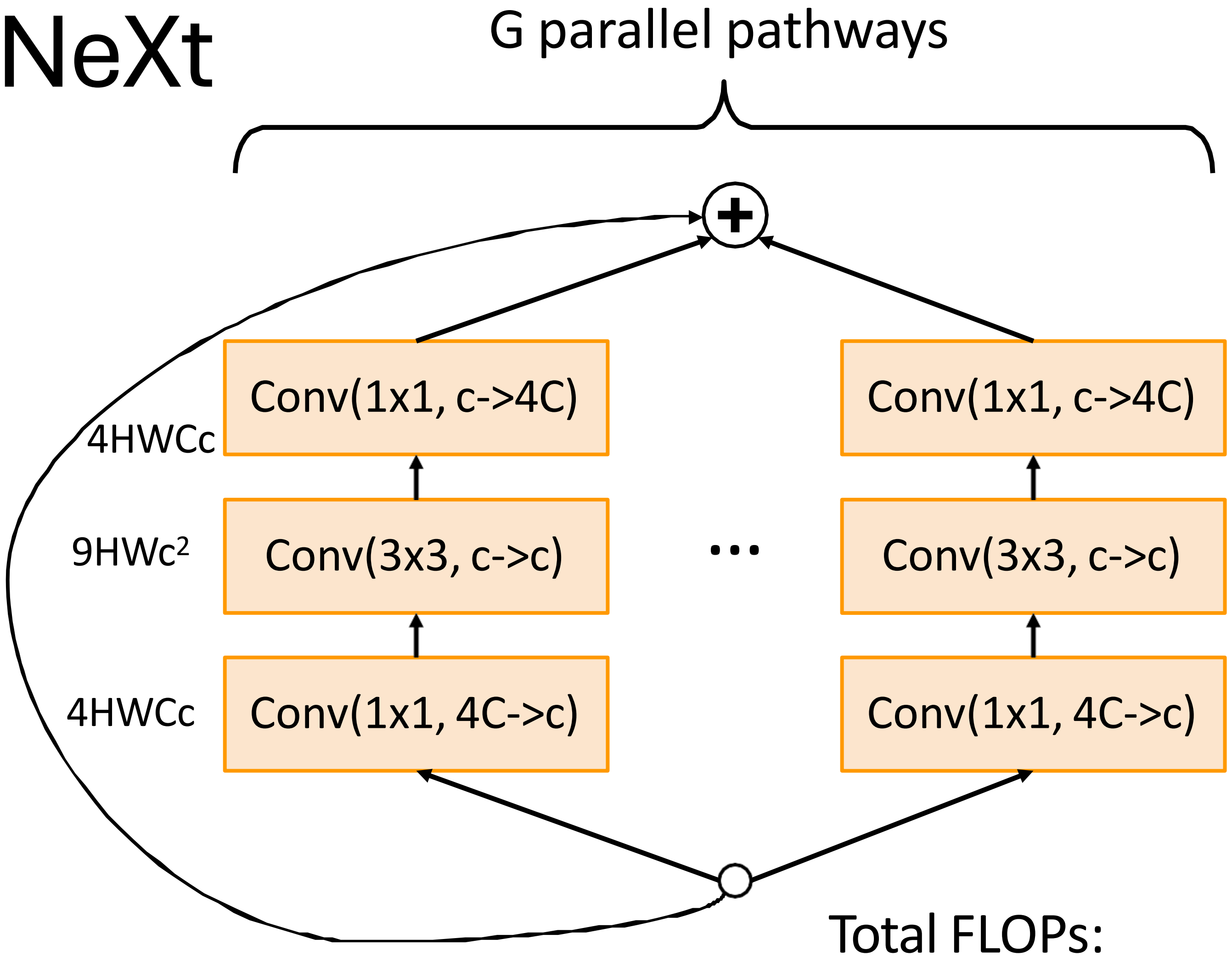
Total FLOPs: $17HWC^2$

Improving ResNets: ResNeXt



"Bottleneck"
Residual block

Total FLOPs:
$17HWC^2$



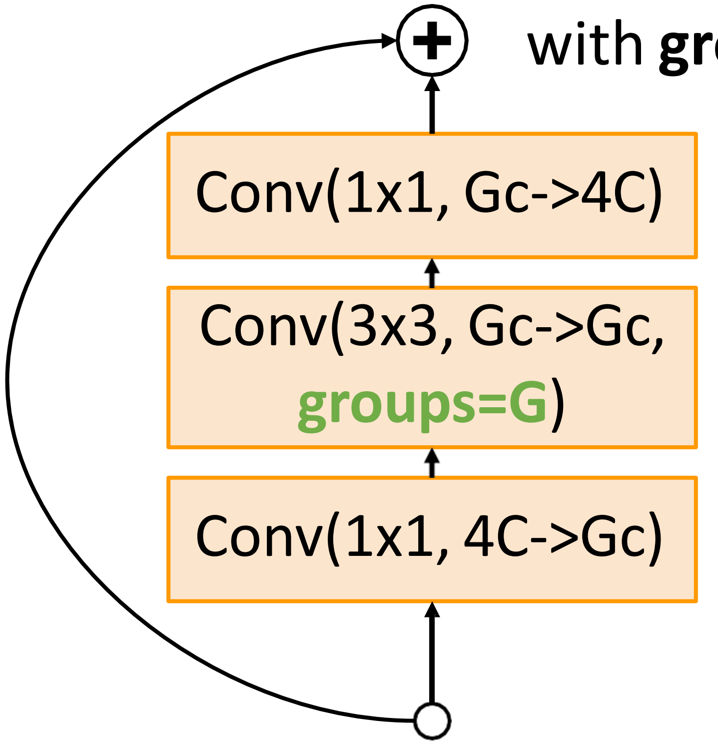
Total FLOPs:
$(8Cc + 9c^2)*HWG$

Same FLOPs when
$9Gc^2 + 8GCc - 17C^2 = 0$

Example: C=64, G=4, c=24; C=64, G=32, c=4

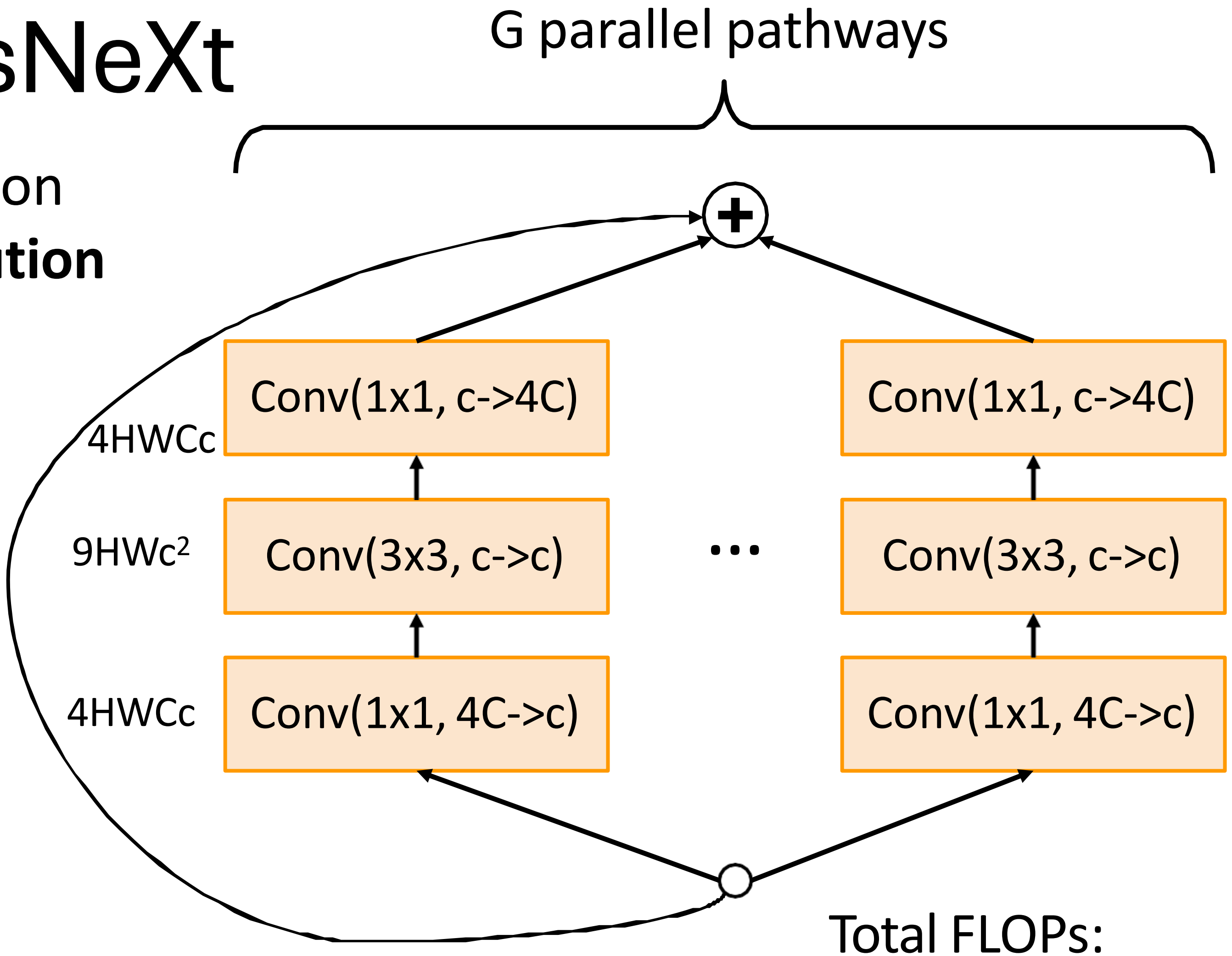
Xie et al, "Aggregated residual transformations for deep neural networks", CVPR 2017

Improving ResNets: ResNeXt



ResNeXt block:
Grouped convolution

Equivalent formulation
with **grouped convolution**



Total FLOPs:
$(8Cc + 9c^2)*HWG$

Same FLOPs when
$9Gc^2 + 8GCc - 17C^2 = 0$

Example: C=64, G=4, c=24; C=64, G=32, c=4

Xie et al, "Aggregated residual transformations for deep neural networks", CVPR 2017

ResNeXt: Maintain computation by adding groups!

Model	Groups	Group width	Top-1 Error
ResNet-50	1	64	23.9
ResNeXt-50	2	40	23
ResNeXt-50	4	24	22.6
ResNeXt-50	8	14	22.3
ResNeXt-50	32	4	22.2

Model	Groups	Group width	Top-1 Error
ResNet-101	1	64	22.0
ResNeXt-101	2	40	21.7
ResNeXt-101	4	24	21.4
ResNeXt-101	8	14	21.3
ResNeXt-101	32	4	21.2

Adding groups improves performance **with same FLOPs!**

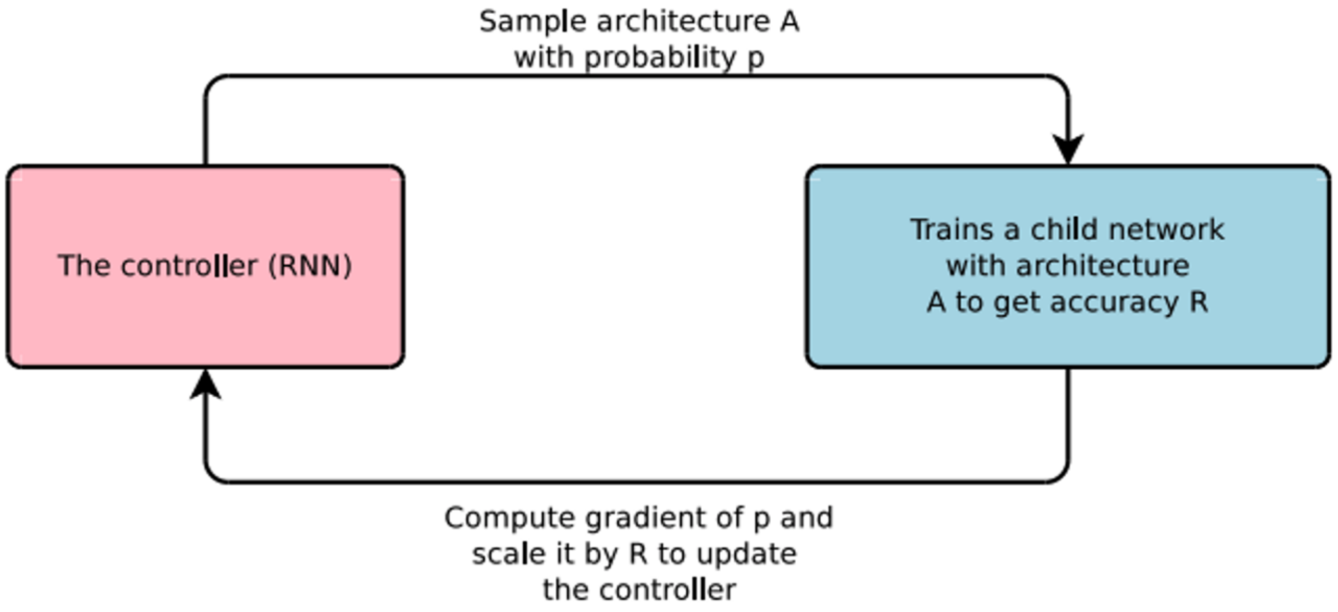
Often denoted e.g. ResNeXt-50-32x4d: 32 groups,
Blocks in first stage have 4 channels per group (#channels still doubles at each stage)

Xie et al, “Aggregated residual transformations for deep neural networks”, CVPR 2017

Neural Architecture Search (NAS)

Designing neural network architectures is hard – let's automate it!

- One network (**controller**) outputs network architectures
- Sample **child networks** from controller and train them
- After training a batch of child networks, make a gradient step on controller network (Using **policy gradient**)
- Over time, controller learns to output good architectures!



Zoph and Le, "Neural Architecture Search with Reinforcement Learning", ICLR 2017
Zoph et al, "Learning transferable architectures for scalable image recognition", CVPR 2018

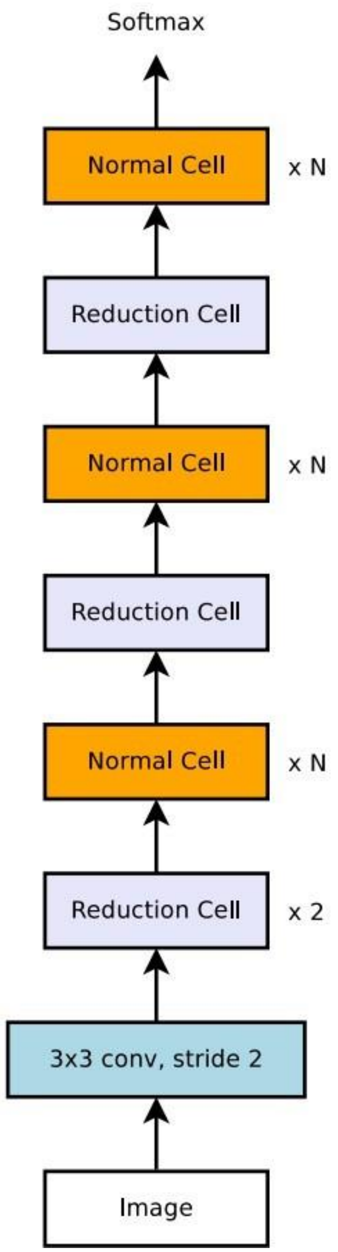
Neural Architecture Search (NAS)

- Search for reusable "block" designs which can use the following operators:
  - Identity
  - 1x1 conv
  - 3x3 conv
  - 3x3 dilated conv
  - 1x7 then 7x1 conv
  - 1x3 then 3x1 conv
  - 3x3, 5x5, or 7x7 depthwise-separable conv
  - 3x3 avg pool
  - 3x3, 5x5, or 7x7 max pool

The "Normal cell" maintains the same image resolution

The "Reduction cell" reduces image resolution by 2x

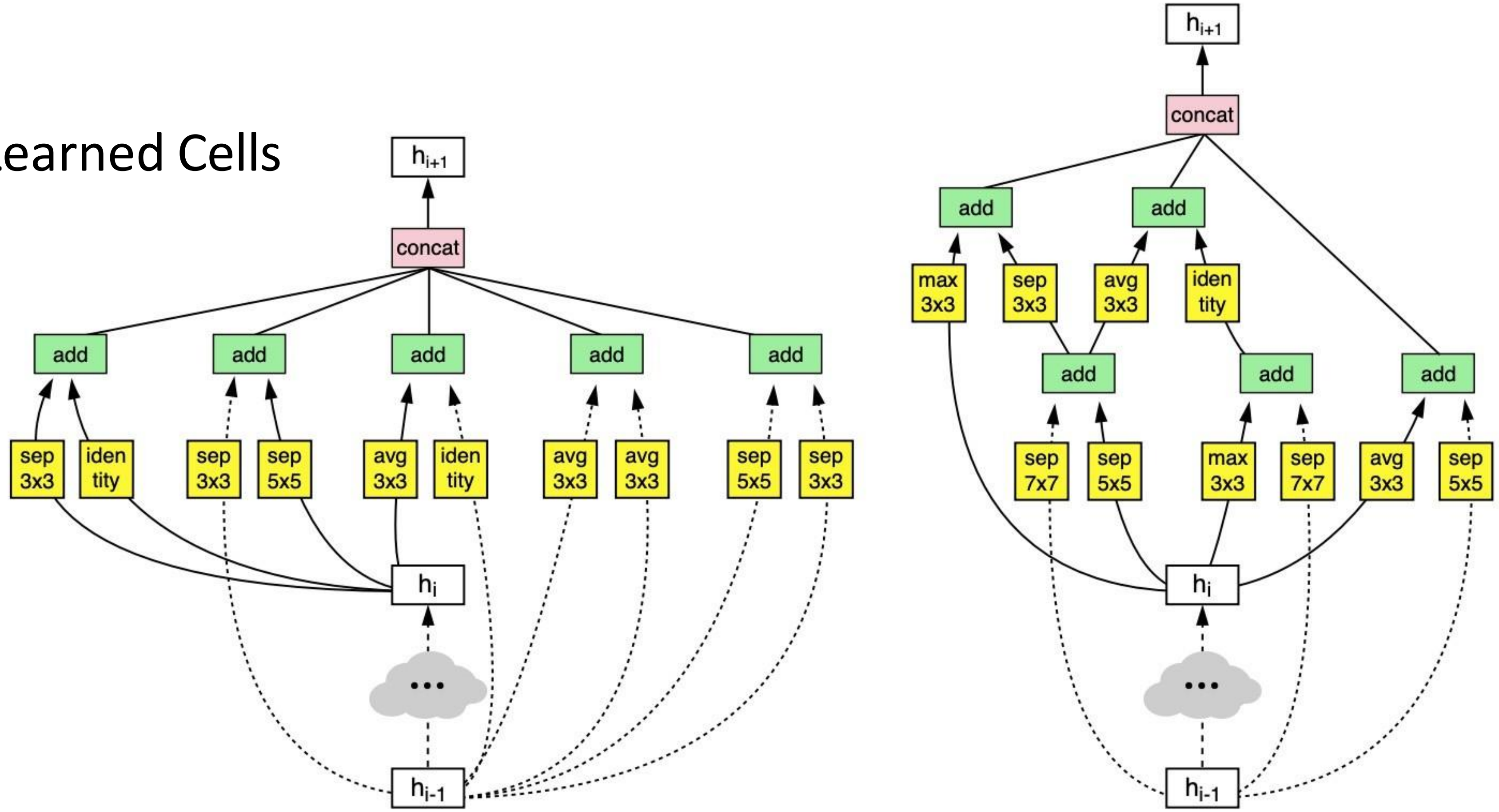
Combine two learned cells in a regular pattern to create overall architecture



Zoph and Le, "Neural Architecture Search with Reinforcement Learning", ICLR 2017
Zoph et al, "Learning transferable architectures for scalable image recognition", CVPR 2018

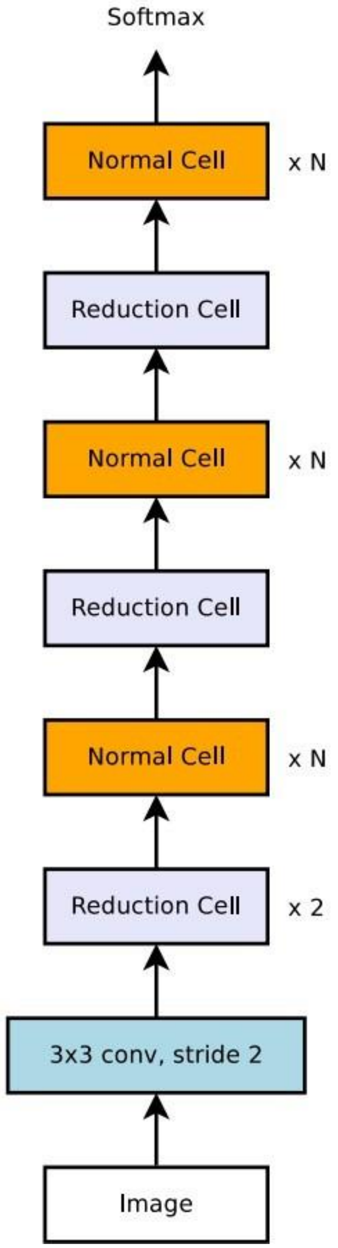
Neural Architecture Search (NAS)

Learned Cells



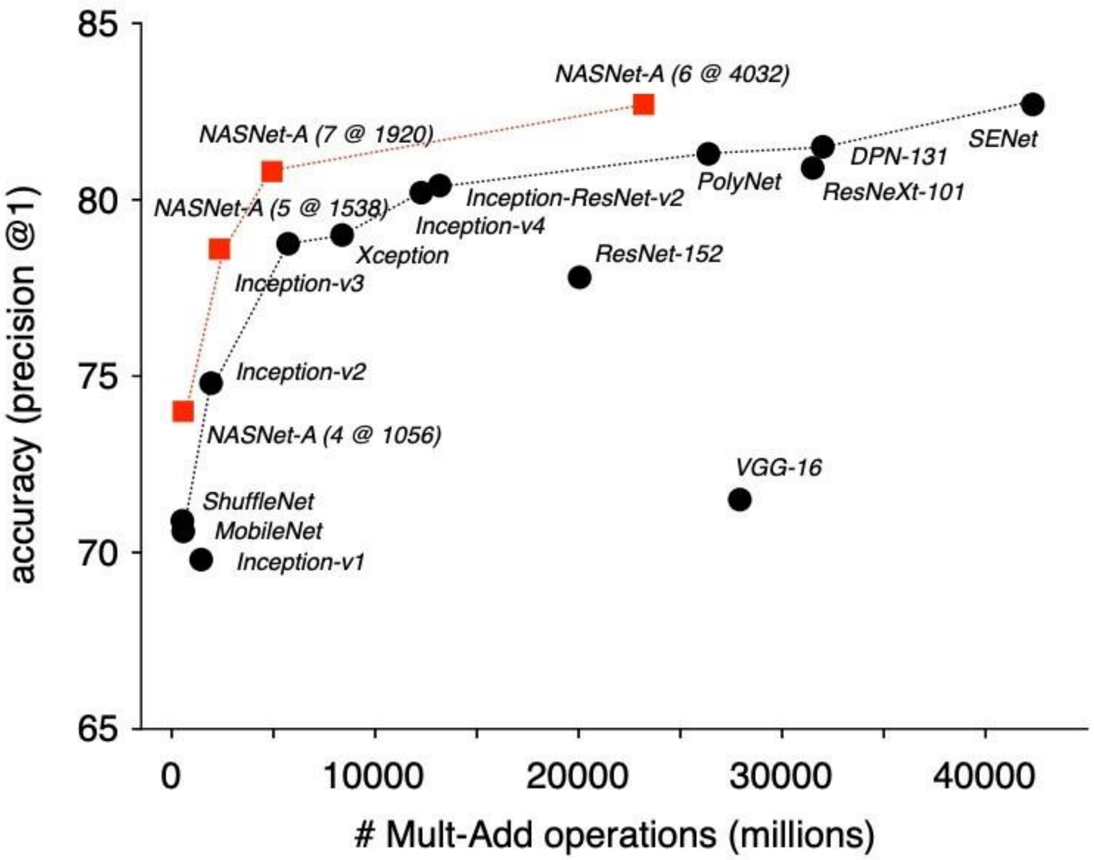
Normal Cell

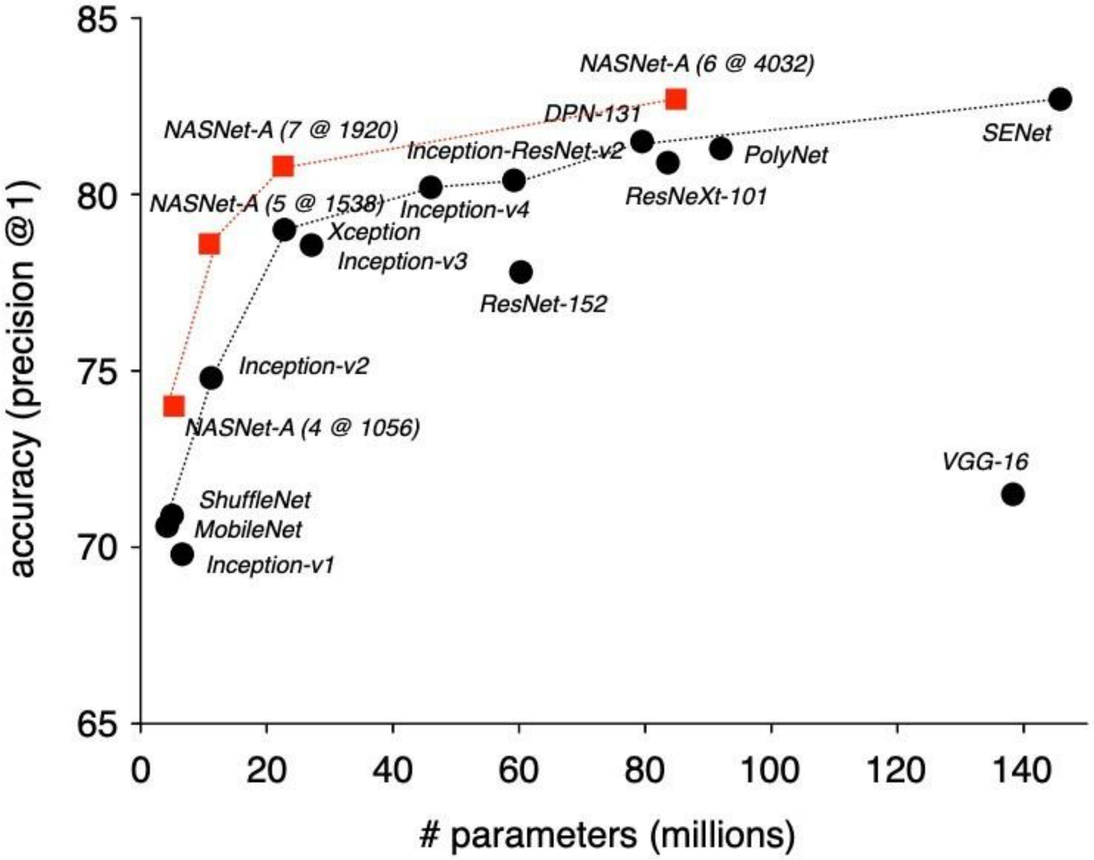
Reduction Cell



Zoph and Le, "Neural Architecture Search with Reinforcement Learning", ICLR 2017
Zoph et al, "Learning transferable architectures for scalable image recognition", CVPR 2018

Neural Architecture Search (NAS)



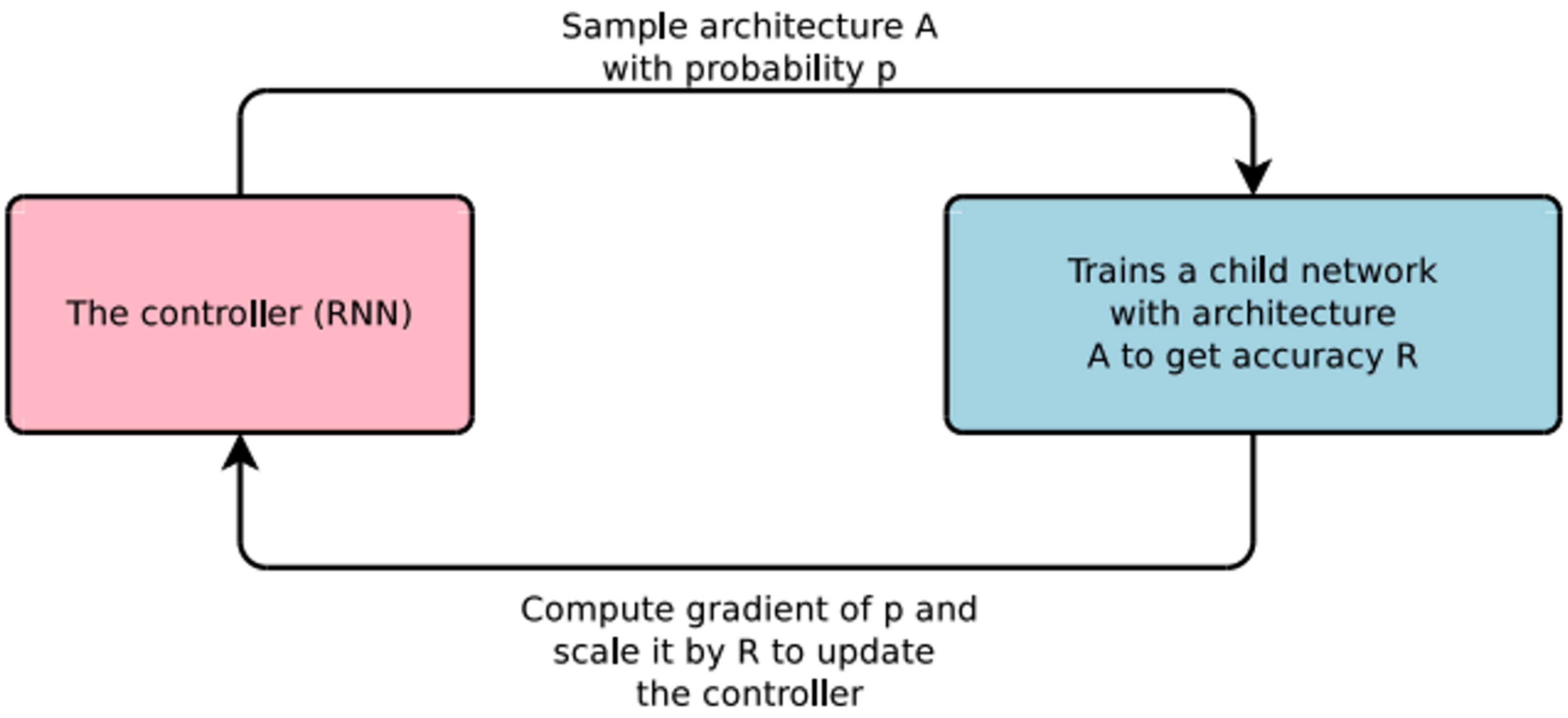


Zoph and Le, “Neural Architecture Search with Reinforcement Learning”, ICLR 2017
Zoph et al, “Learning transferable architectures for scalable image recognition”, CVPR 2018

Big Problem: NAS is Very Expensive!

Original NAS paper: Each update to the controller requires training **800 child models** for 50 epochs on CIFAR10; Total of **12,800** child models are trained

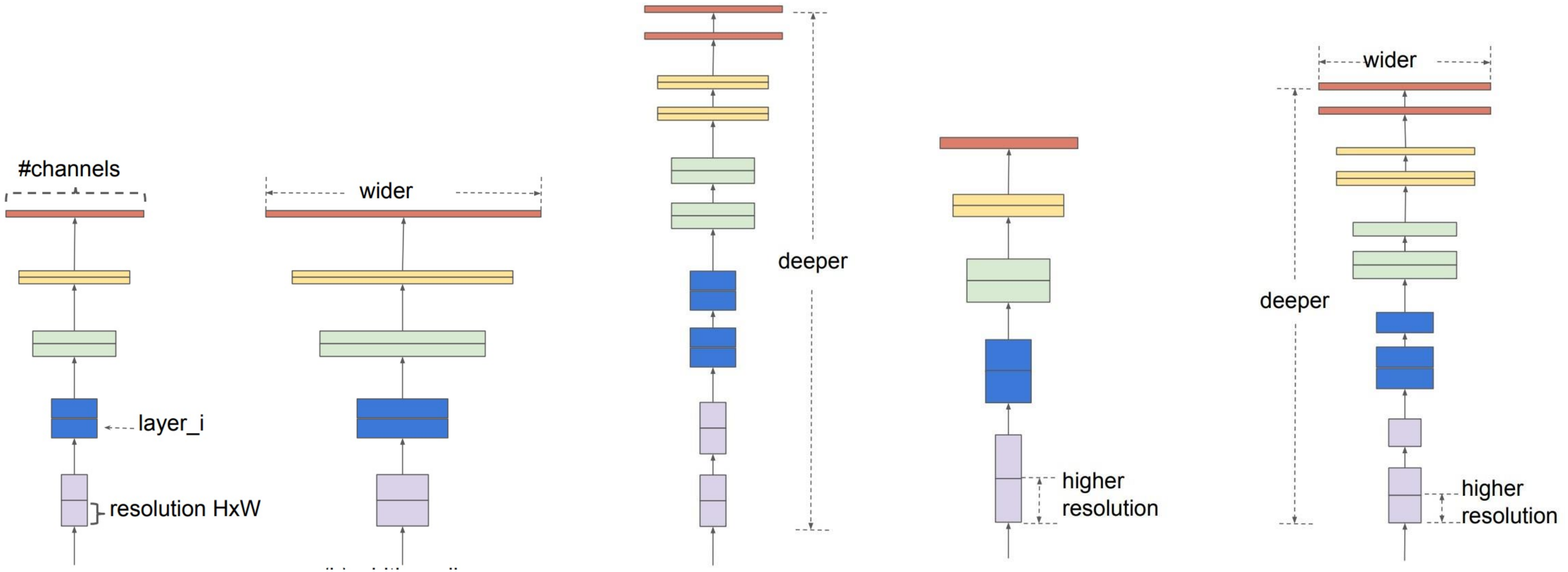
Later work improved efficiency, but still expensive



Zoph and Le, “Neural Architecture Search with Reinforcement Learning”, ICLR 2017

Model Scaling

Starting from a given architecture, how should you **scale it up** to improve performance?



Baseline Architecture

Width : Increase channels in all layers

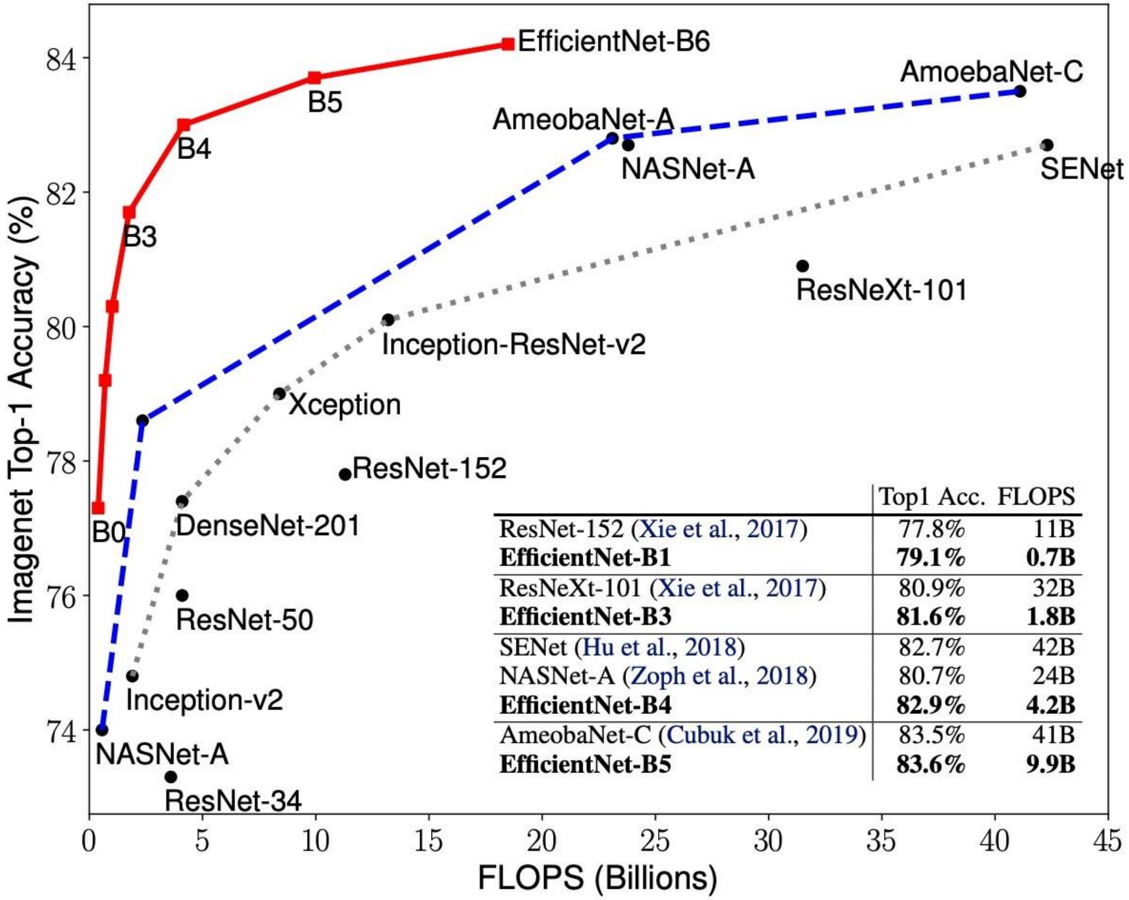
Depth: Use more layers

Resolution: Higher resolution input image

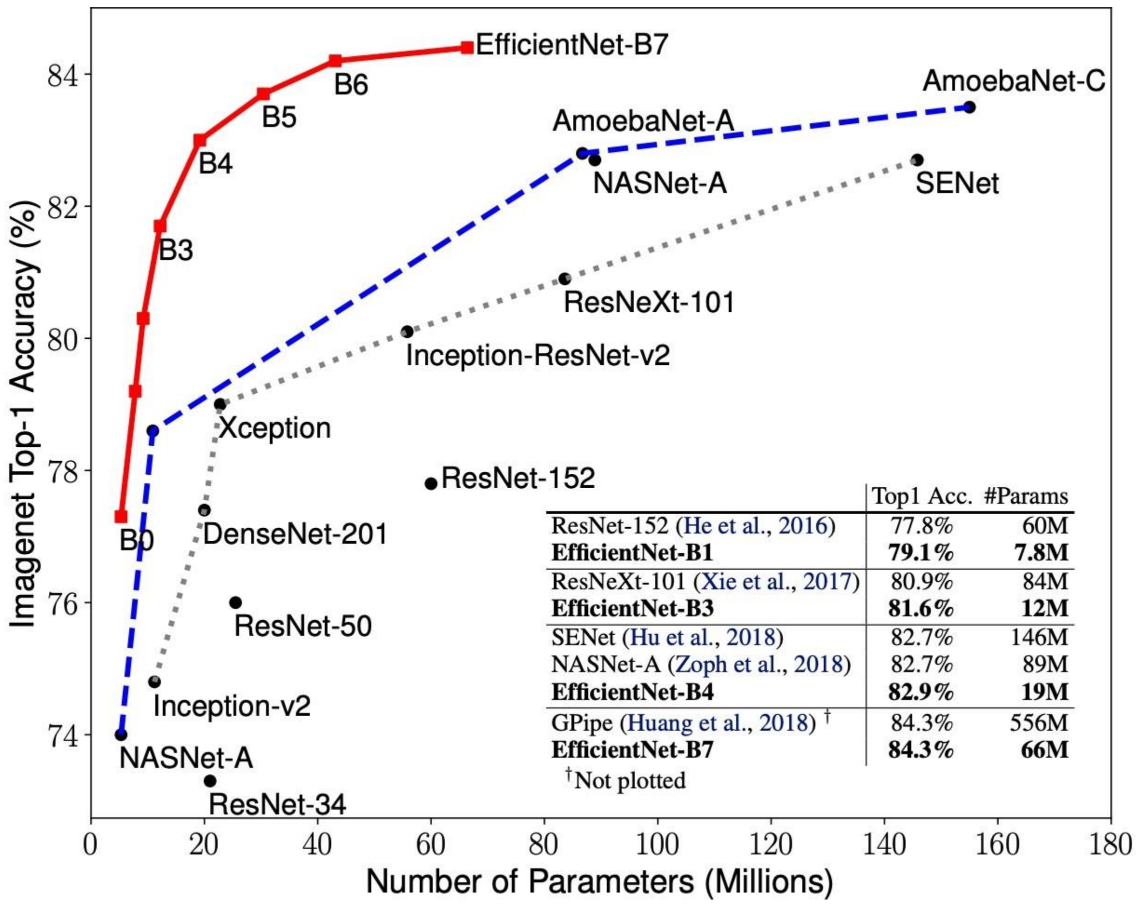
Compound: Scale all jointly

Tan and Le, "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks", ICML 2019

Model Scaling: EfficientNets



	Top1 Acc.	FLOPS
ResNet-152 (Xie et al., 2017)	77.8%	11B
EfficientNet-B1	**79.1%**	**0.7B**
ResNeXt-101 (Xie et al., 2017)	80.9%	32B
EfficientNet-B3	**81.6%**	**1.8B**
SENet (Hu et al., 2018)	82.7%	42B
NASNet-A (Zoph et al., 2018)	80.7%	24B
EfficientNet-B4	**82.9%**	**4.2B**
AmeobaNet-C (Cubuk et al., 2019)	83.5%	41B
EfficientNet-B5	**83.6%**	**9.9B**



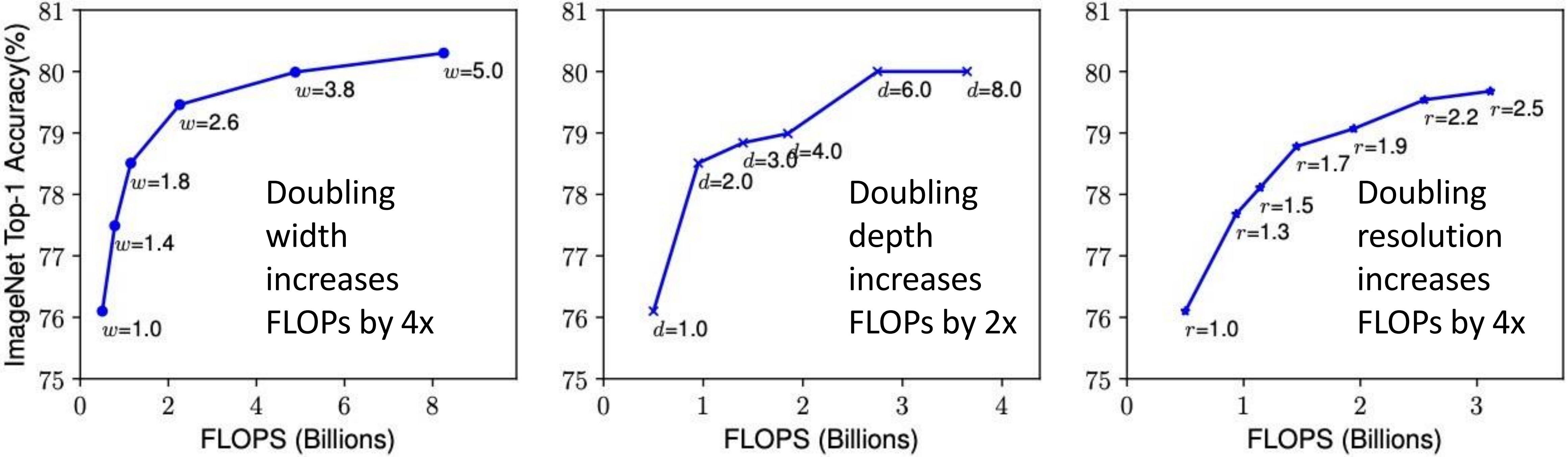
	Top1 Acc.	#Params
ResNet-152 (He et al., 2016)	77.8%	60M
EfficientNet-B1	**79.1%**	**7.8M**
ResNeXt-101 (Xie et al., 2017)	80.9%	84M
EfficientNet-B3	**81.6%**	**12M**
SENet (Hu et al., 2018)	82.7%	146M
NASNet-A (Zoph et al., 2018)	82.7%	89M
EfficientNet-B4	**82.9%**	**19M**
GPipe (Huang et al., 2018) †	84.3%	556M
EfficientNet-B7	**84.3%**	**66M**

†Not plotted

Tan and Le, “EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks”, ICML 2019

Model Scaling: EfficientNets

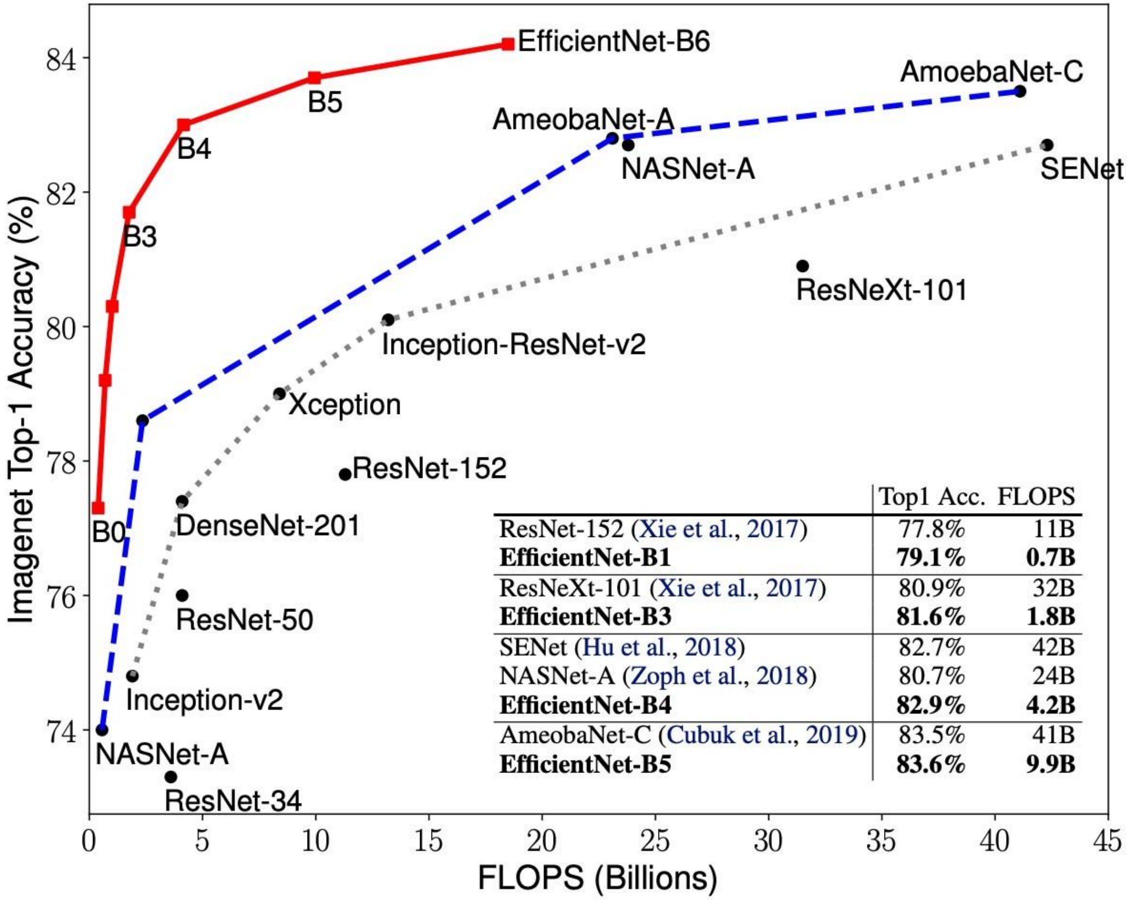
Scale width only | Scale depth only | Scale resolution only



Scaling any of width, depth, or resolution has diminishing returns.
For optimal results, need to scale them all jointly!

Tan and Le, “EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks”, ICML 2019

Model Scaling: EfficientNets



	Top1 Acc.	FLOPS
ResNet-152 (Xie et al., 2017)	77.8%	11B
EfficientNet-B1	**79.1%**	**0.7B**
ResNeXt-101 (Xie et al., 2017)	80.9%	32B
EfficientNet-B3	**81.6%**	**1.8B**
SENet (Hu et al., 2018)	82.7%	42B
NASNet-A (Zoph et al., 2018)	80.7%	24B
EfficientNet-B4	**82.9%**	**4.2B**
AmeobaNet-C (Cubuk et al., 2019)	83.5%	41B
EfficientNet-B5	**83.6%**	**9.9B**

Big problem: *Real-world runtime does not correlate well with FLOPs!*

- Runtime depends on the device (mobile CPU, server CPU, GPU, TPU); A model which is fast on one device may be slow on another
- Depthwise convolutions are efficient on mobile, but not on GPU / TPU – they become memory-bound
- The “naïve” FLOP counting we have done for convolutions can be incorrect – alternate conv algorithms can reduce FLOPs in some settings (FFT for large kernels, Winograd for 3x3 conv)
- EfficientNet was designed to minimize FLOPs, not actual runtime – so it is surprisingly slow!

Tan and Le, “EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks”, ICML 2019

Vasilache et al, “Fast Convolutional Nets With fbfft: A GPU Performance Evaluation”, ICLR 2015
Lavin and Gray, “Fast Algorithms for Convolutional Neural Networks”, CVPR 2016

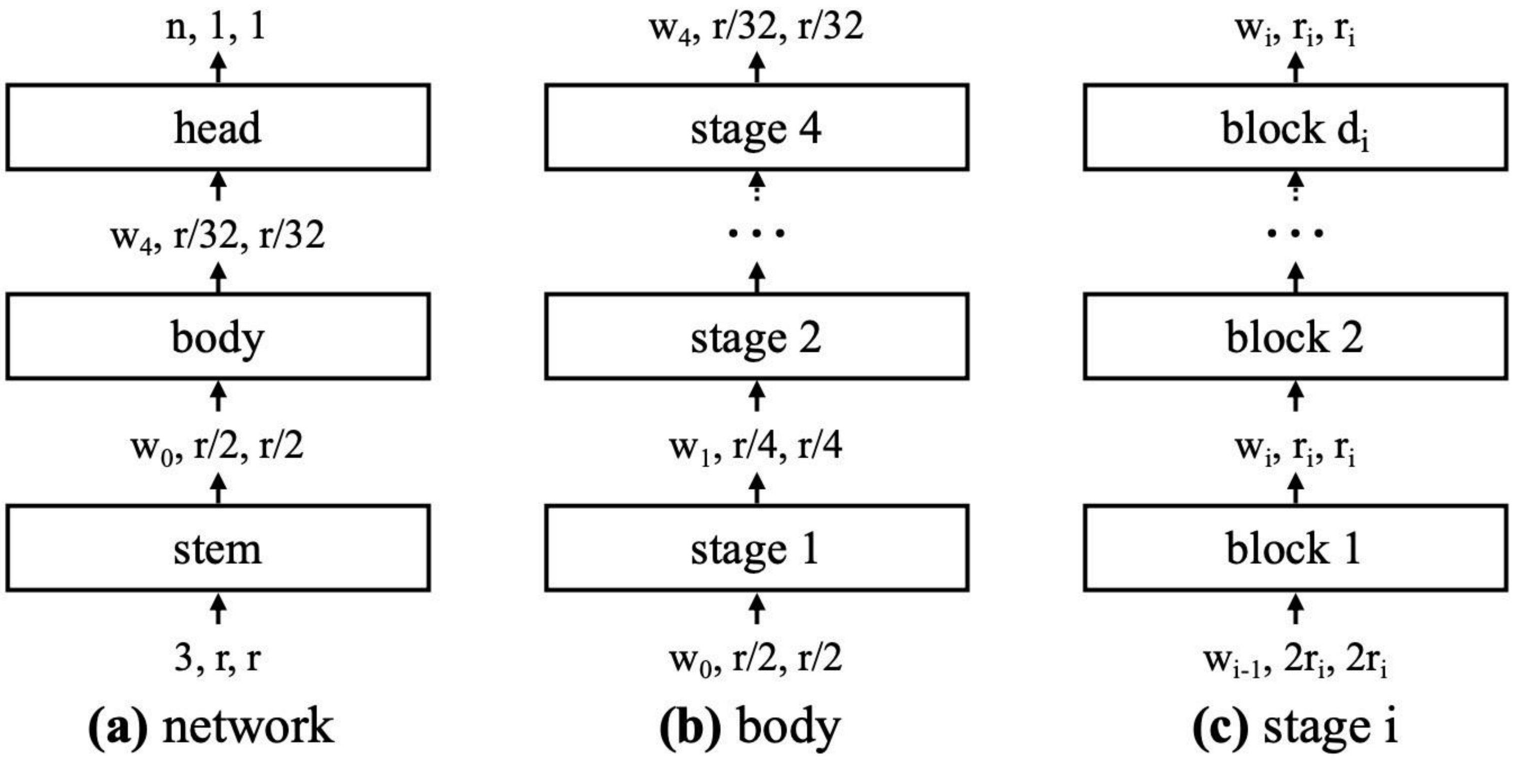
Beyond NAS – back to hand-designed models!

Instead of using NAD
smartly tweak ResNet-style models to improve performance, scaling, runtime on GPU / TPU

RegNets: Simple block design, optimize macro architecture and scaling
NFNets: Remove Batch Normalization

RegNets: Network Design Spaces

Network design is simple: **Stem** of 3x3 convs, a **body** of 4 *stages*, and a **head;** Each stage has multiple **blocks**: First block downsamples by 2x, others keep resolution the same



Radosavovic et al, "Designing Network Design Spaces", CVPR 2020
Dollar et al, "Fast and Accurate Model Scaling", CVPR 2021

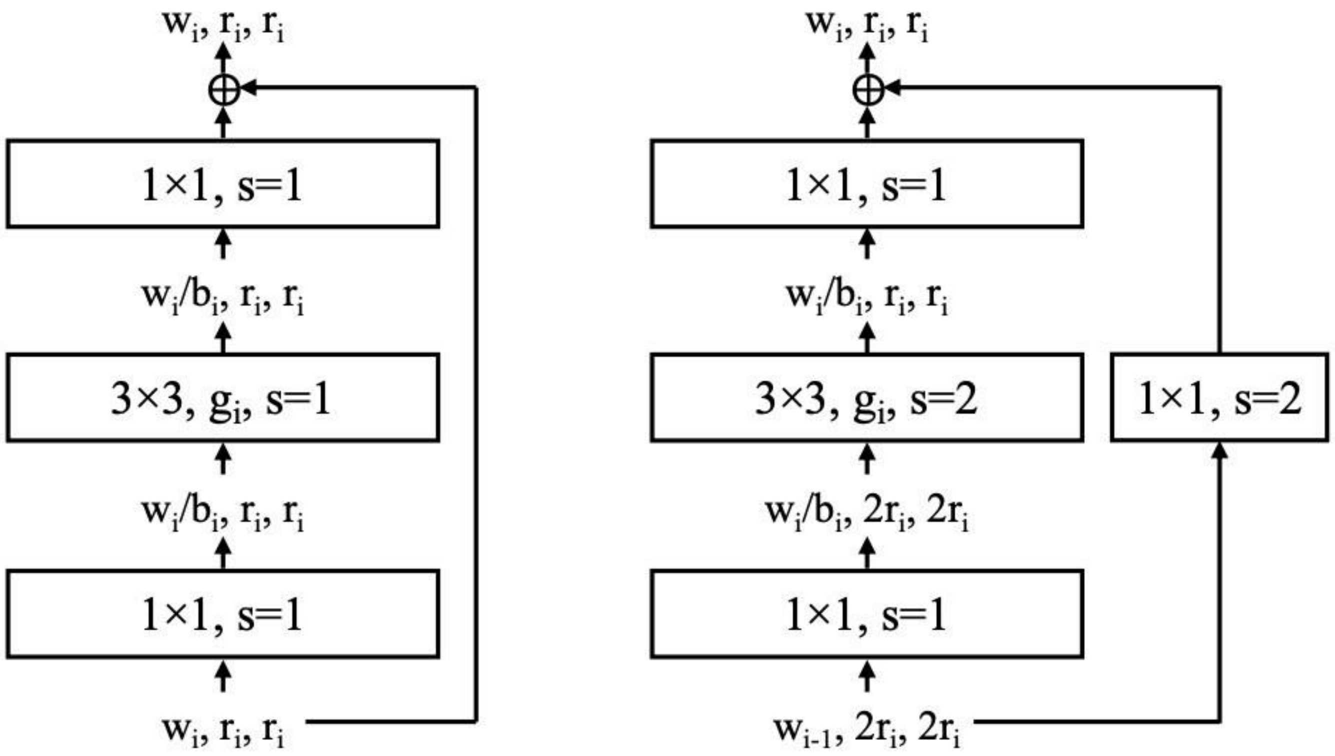
RegNets: Network Design Spaces

Block design is simple, generalizes ResNext

Each stage has 4 parameters:

- Number of blocks
- Number of input channels w
- Bottleneck ratio b
- Group width g

The *design space* for the network has just 16 parameters

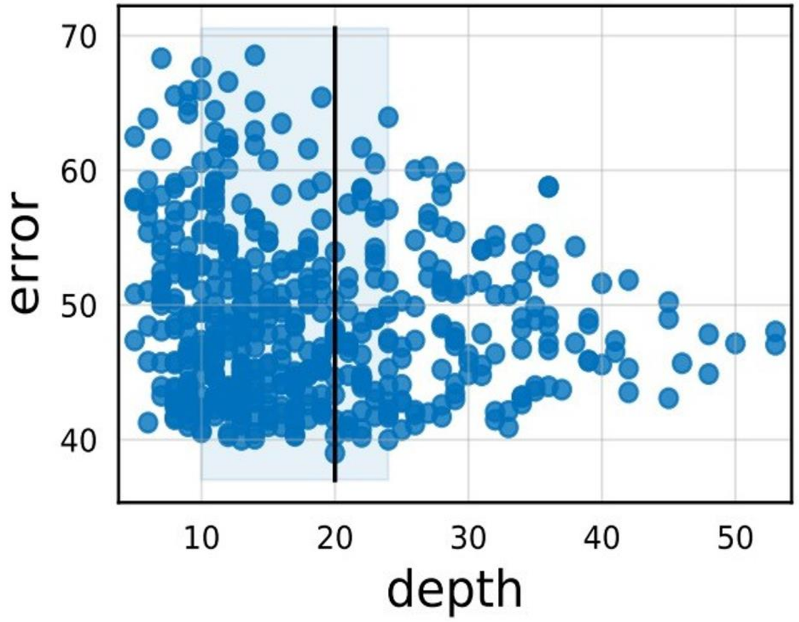


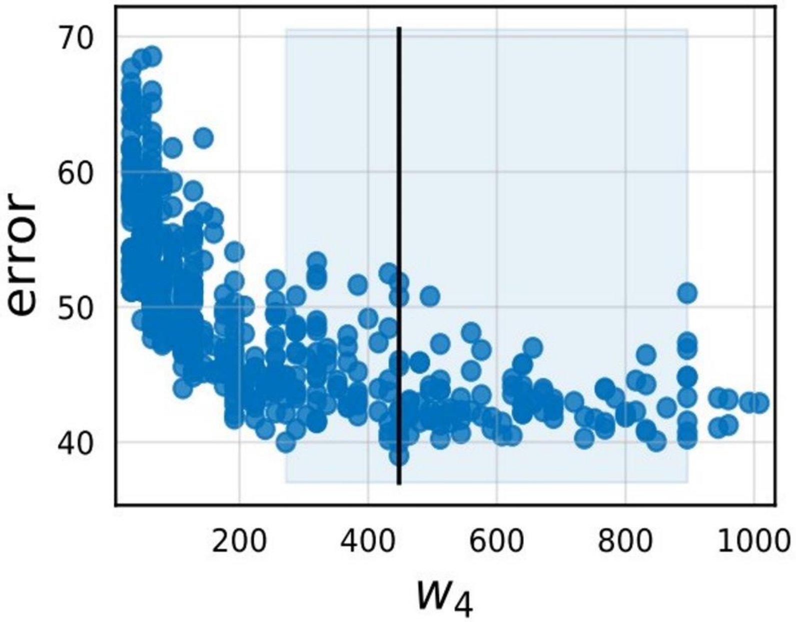
(a) X block, s=1 **(b)** X block, s=2

Radosavovic et al, “Designing Network Design Spaces”, CVPR 2020
Dollar et al, “Fast and Accurate Model Scaling”, CVPR 2021

RegNets: Network Design Spaces

Randomly sample architectures from the design space, examine trends:





Radosavovic et al, “Designing Network Design Spaces”, CVPR 2020
Dollar et al, “Fast and Accurate Model Scaling”, CVPR 2021

RegNets: Network Design Spaces

Use results to *refine* the design space: Reduce degrees of freedom from 16 to bias toward better-performing architectures:

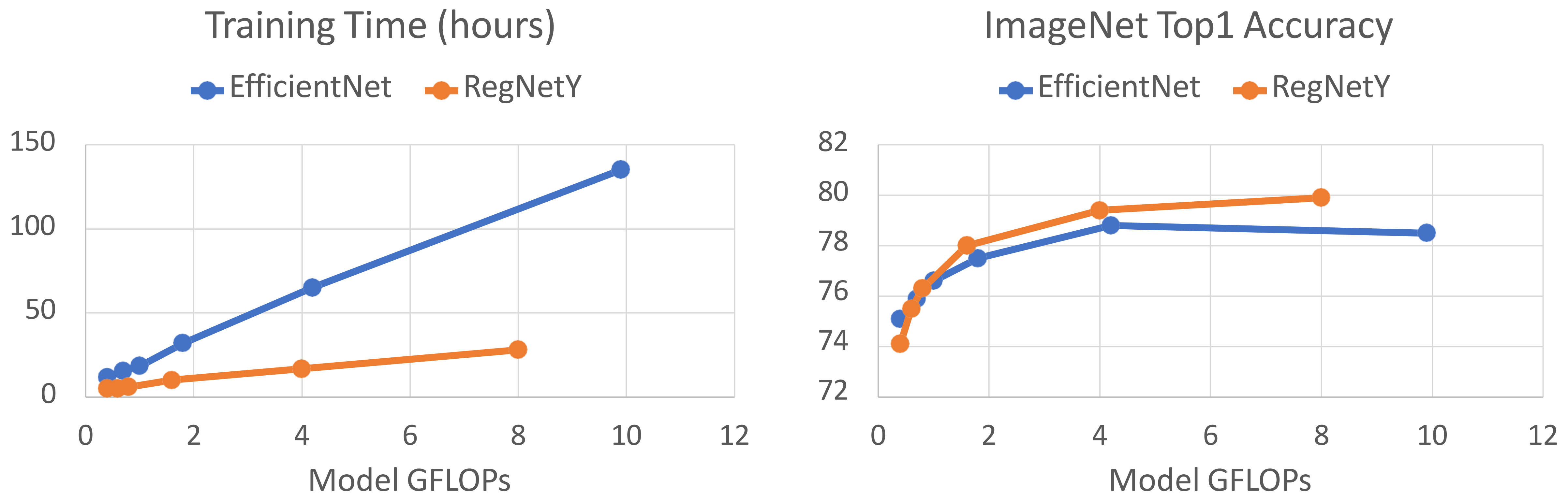
- Share bottleneck ratio across all stages (16 -> 13 params)
- Share group width across all stages (13 -> 10 params)
- Force width, blocks per stage to increase *linearly* across stages

Final design space has 6 parameters:

- Overall depth d, bottleneck ratio b, group width g
- Initial width w_0, width growth rate w_a, blocks per stage w_m

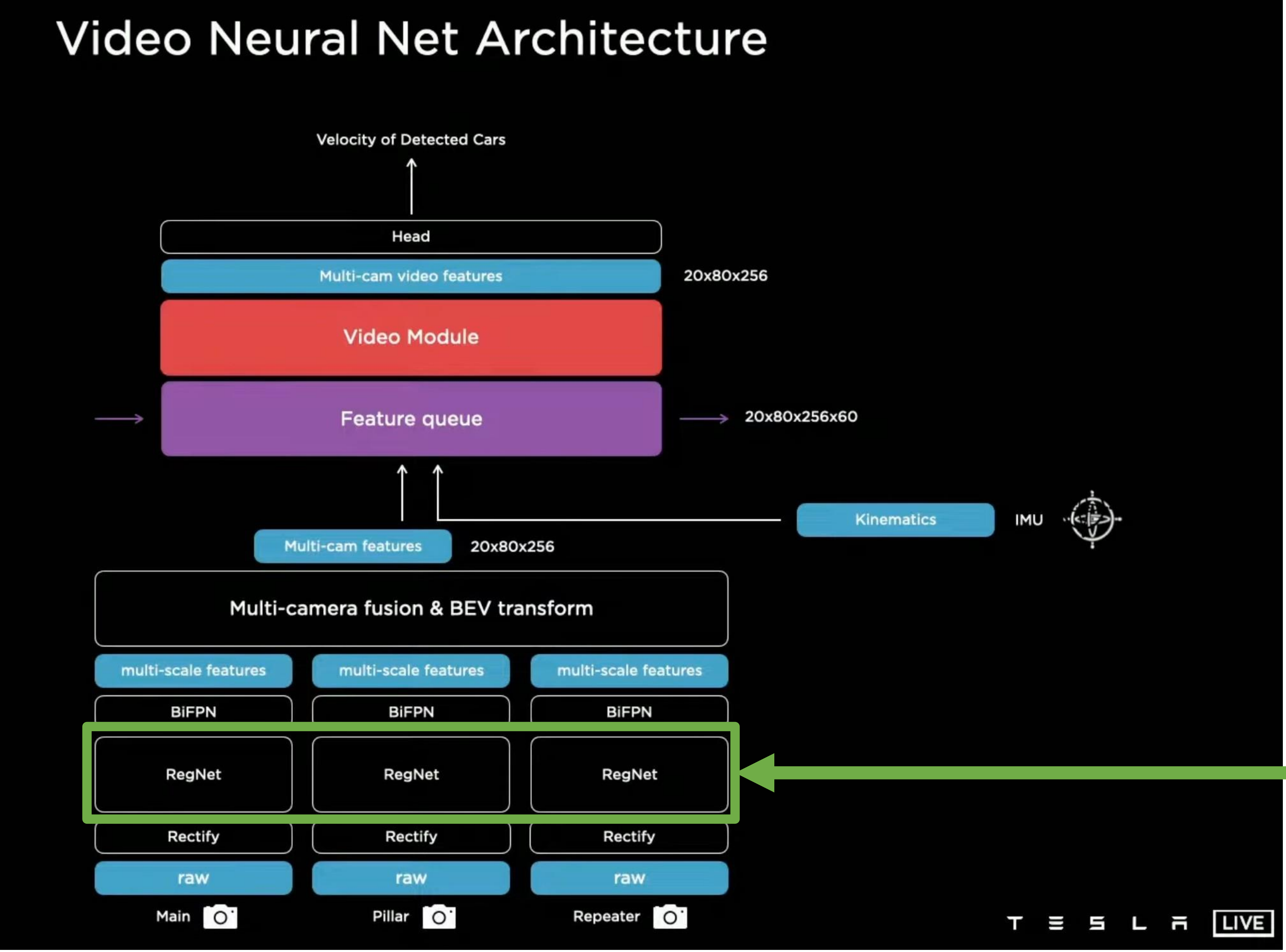
Radosavovic et al, "Designing Network Design Spaces", CVPR 2020
Dollar et al, "Fast and Accurate Model Scaling", CVPR 2021

RegNets: Network Design Spaces



At same FLOPs, RegNet models get similar accuracy as EfficientNets but are up to 5x faster in training (each iteration is faster)

Radosavovic et al, "Designing Network Design Spaces", CVPR 2020
Dollar et al, "Fast and Accurate Model Scaling", CVPR 2021



Tesla Vision system uses RegNets to process inputs from each camera

Tesla AI Day 2021, https://www.youtube.com/watch?v=j0z4FweCy4M

Training ResNets without Batch Normalization

- Batch Normalization has good properties:
  - Makes it easy to train deep networks >= 10 layers
  - Makes learning rates, initialization less critical
  - Adds regularization
  - ”Free” at inference: can be merged into linear layers
- But also has bad properties:
  - Doesn’t work with small minibatches
  - Different behavior at train and test
  - Slow at training time

NFNets are ResNets without Batch Normalization!

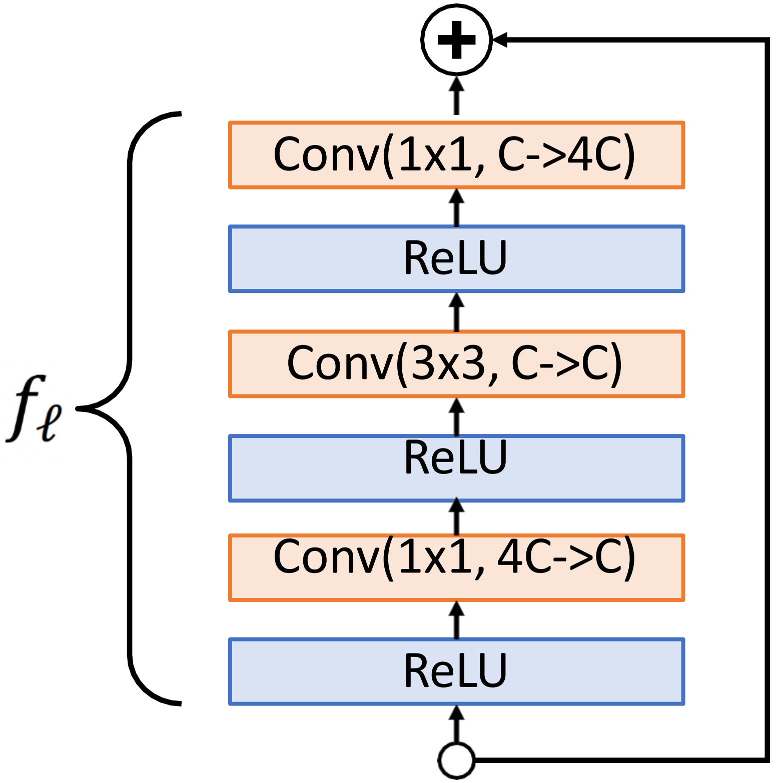
Brock et al, “Characterizing Signal Propagation to Close the Performance Gap in Unnormalized ResNets”, ICLR 2021
Brock et al, “High-Performance Large-Scale Image Recognition without Normalization”, ICML 2021

NFNets

Consider a pre-activation ResNet block $x_{\ell+1} = f_\ell(x_\ell) + x_\ell$

Problem: Variance grows with each block:

$$Var(x_{\ell+1}) = Var(x_\ell) + Var\big(f_\ell(x_\ell)\big)$$



Brock et al, “Characterizing Signal Propagation to Close the Performance Gap in Unnormalized ResNets”, ICLR 2021
Brock et al, “High-Performance Large-Scale Image Recognition without Normalization”, ICML 2021
He et al, “Identity Mappings in Deep Residual Networks”, ECCV 2016

NFNets: Scaled Residual Blocks

Consider a pre-activation ResNet block $x_{\ell+1} = f_\ell(x_\ell) + x_\ell$

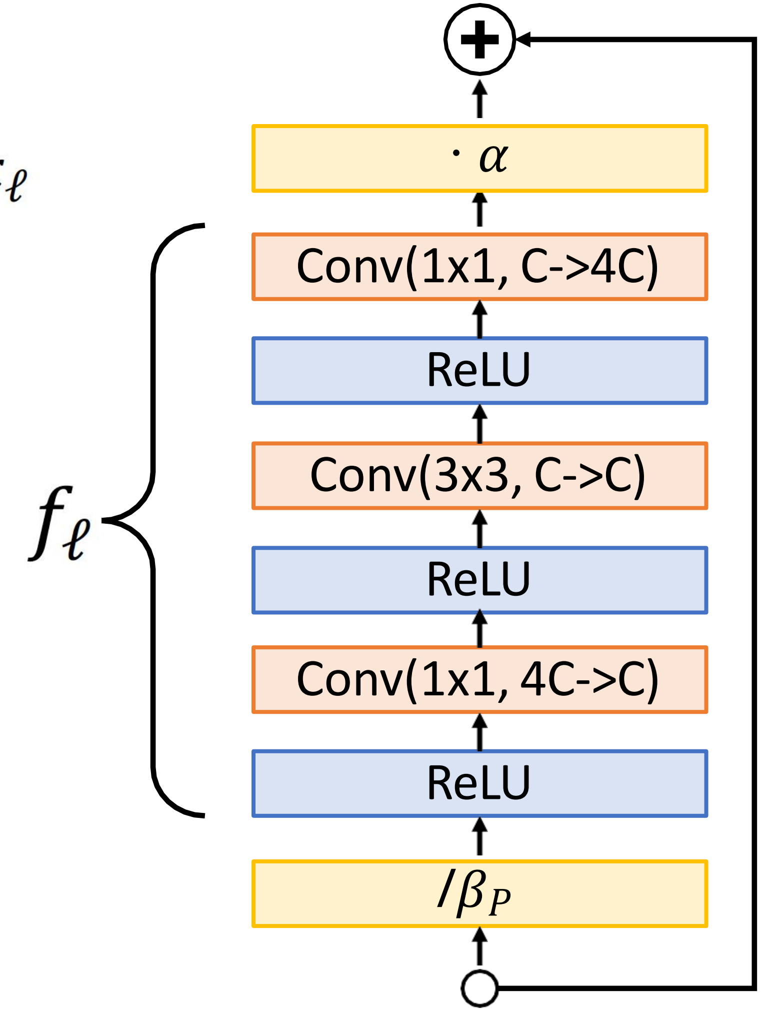
Problem: Variance grows with each block:

$$Var(x_{\ell+1}) = Var(x_\ell) + Var\big(f_\ell(x_\ell)\big)$$

Solution: Re-parameterize block:

$$x_{\ell+1} = x_\ell + \alpha f_\ell(x_\ell/\beta_\ell)$$

α is a hyperparameter, $\beta_\ell = \sqrt{Var(x_\ell)}$ at initialization; both are constants during training



Brock et al, "Characterizing Signal Propagation to Close the Performance Gap in Unnormalized ResNets", ICLR 2021

Brock et al, "High-Performance Large-Scale Image Recognition without Normalization", ICML 2021

NFNets: Scaled Residual Blocks

Consider a pre-activation ResNet block $x_{\ell+1} = f_\ell(x_\ell) + x_\ell$

Problem: Variance grows with each block:

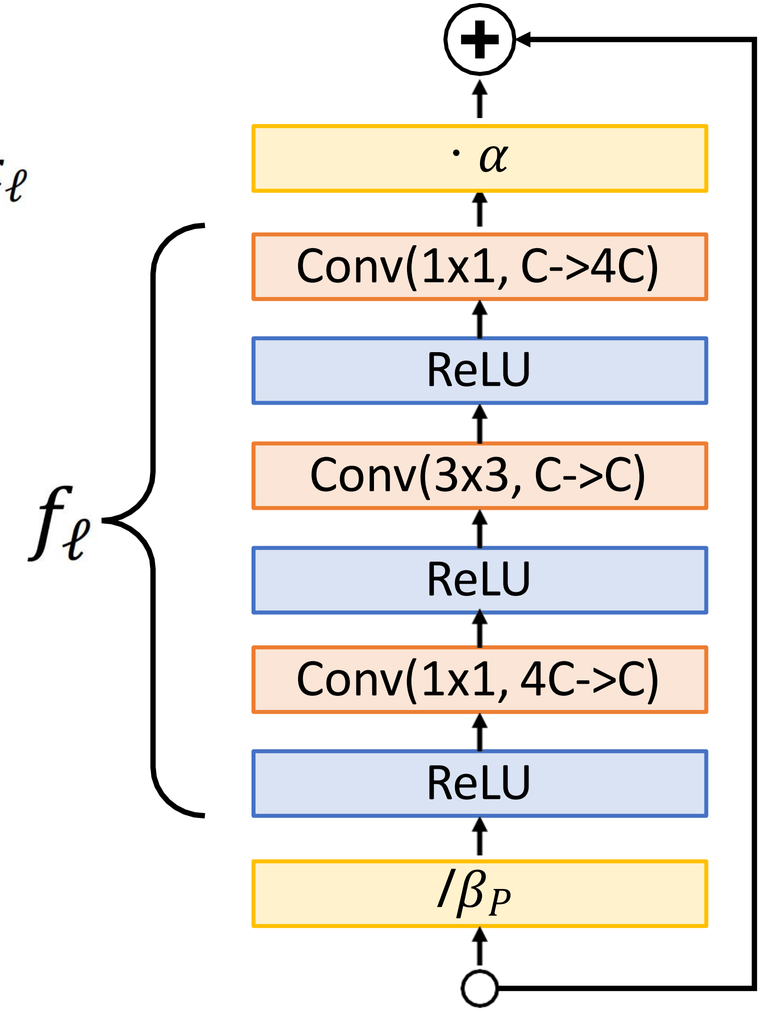
$$Var(x_{\ell+1}) = Var(x_\ell) + Var\big(f_\ell(x_\ell)\big)$$

Solution: Re-parameterize block:

$$x_{\ell+1} = x_\ell + \alpha f_\ell(x_\ell/\beta_\ell)$$

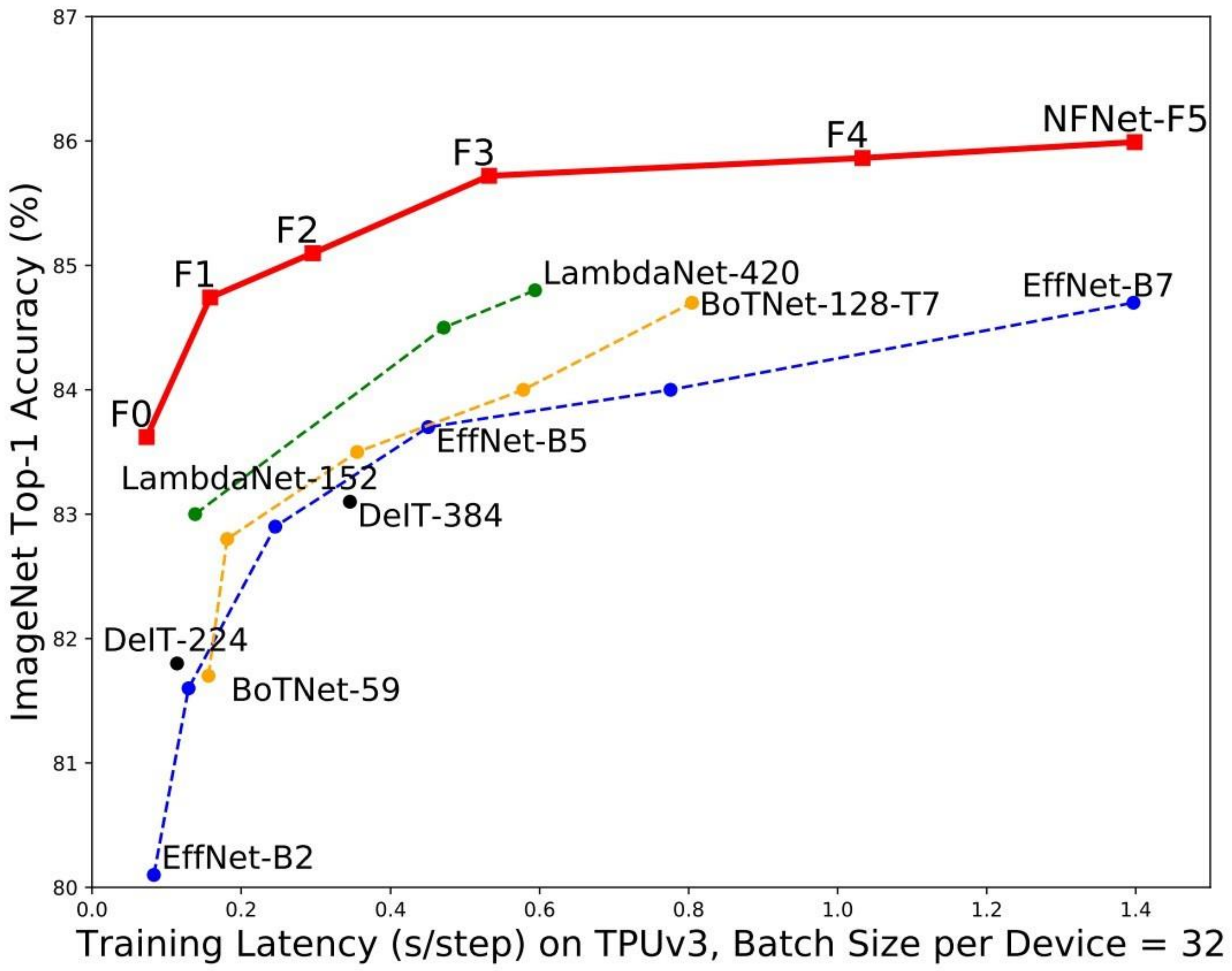
α is a hyperparameter, $\beta_\ell = \sqrt{Var(x_\ell)}$ at initialization; both are constants during training

Now $Var\big(x_{\ell+1)}\big) = Var(x_\ell) + \alpha^2$; resets to $1 + \alpha^2$ after each downsampling block



Brock et al, "Characterizing Signal Propagation to Close the Performance Gap in Unnormalized ResNets", ICLR 2021
Brock et al, "High-Performance Large-Scale Image Recognition without Normalization", ICML 2021

NFNets



Always be careful with plots like this – different papers use different metric for x-axis:

- FLOPs
- Params
- Test-time runtime
- Training-time runtime
- Runtime on CPU / GPU / TPU /…

Brock et al, "Characterizing Signal Propagation to Close the Performance Gap in Unnormalized ResNets", ICLR 2021
Brock et al, "High-Performance Large-Scale Image Recognition without Normalization", ICML 2021