

# Lecture 10: CNN Architectures Part 2



source from University of Michigan EECS498

# Last Time: Training Deep Networks

## **1. One time setup**

Activation functions, data preprocessing,  
weight initialization, regularization

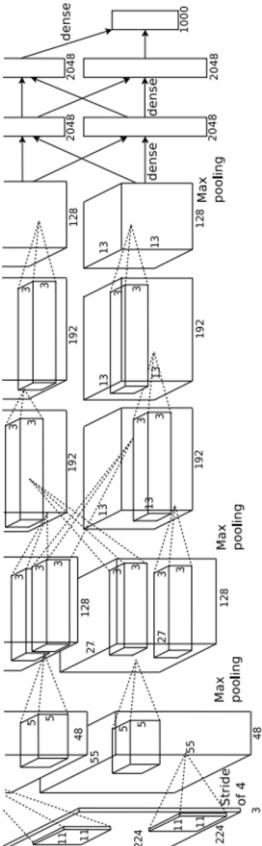
## **2. Training dynamics**

Learning rate schedules;  
hyperparameter optimization

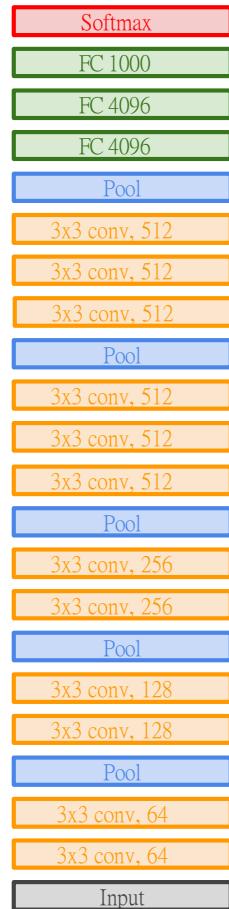
## **3. After training**

Model ensembles

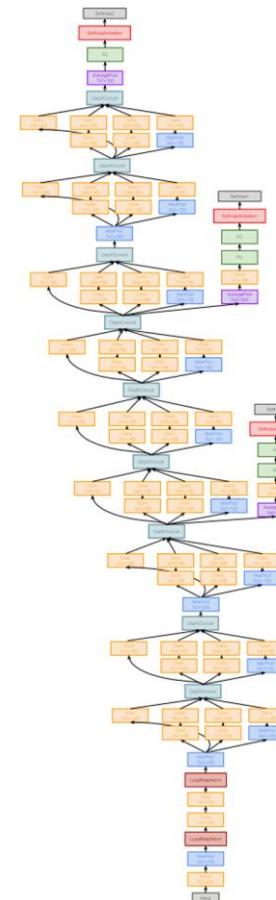
# Previously: CNN Architectures



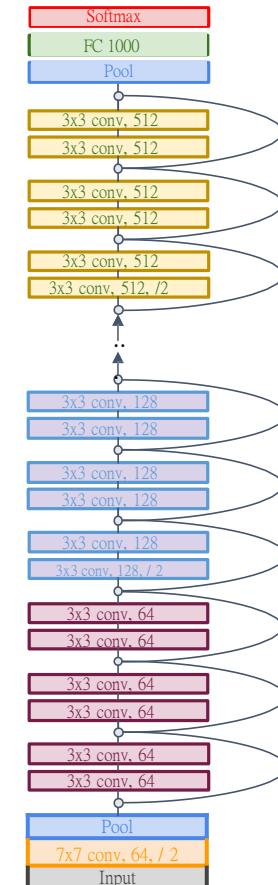
## AlexNet



VGG

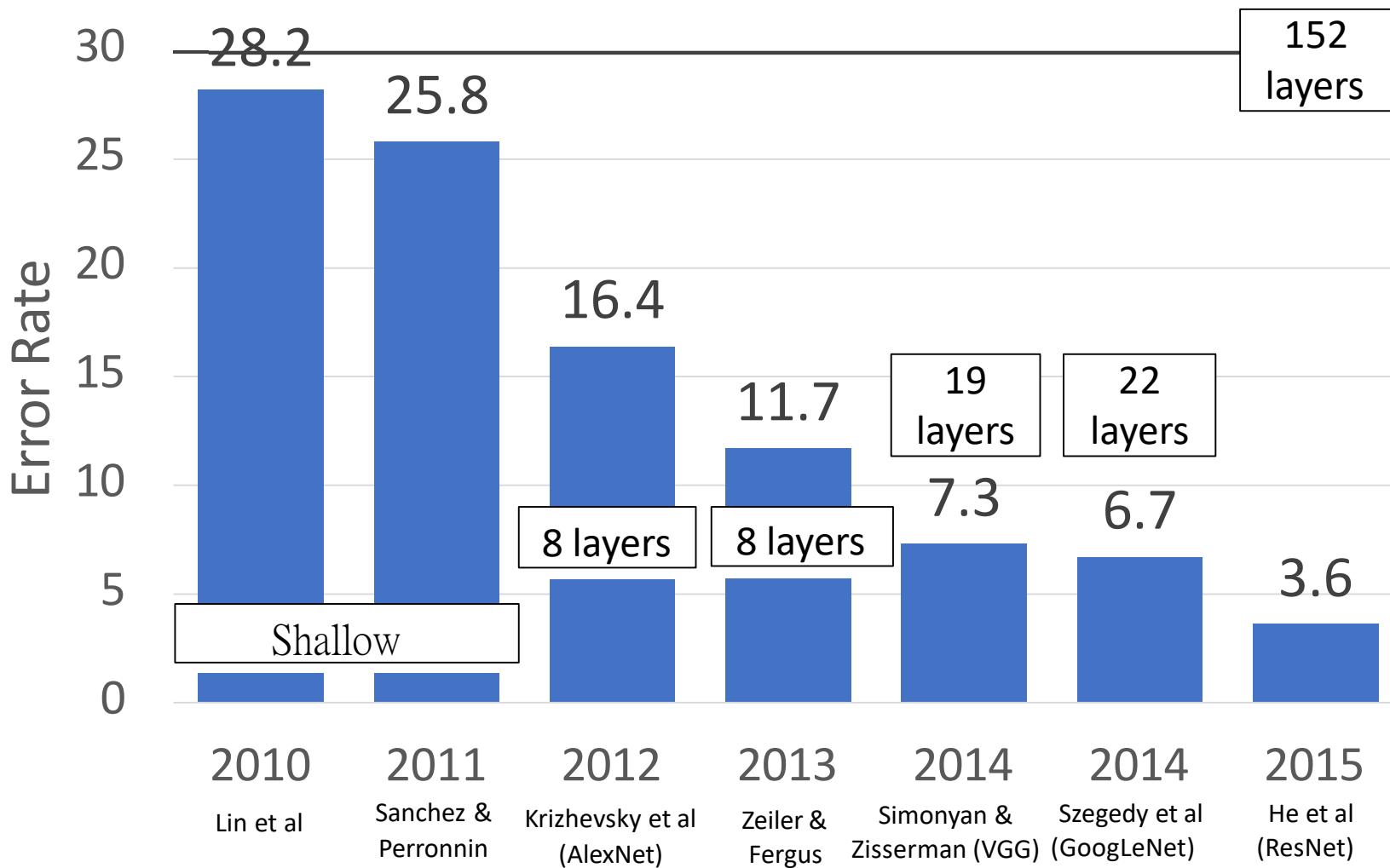


# GoogLeNet

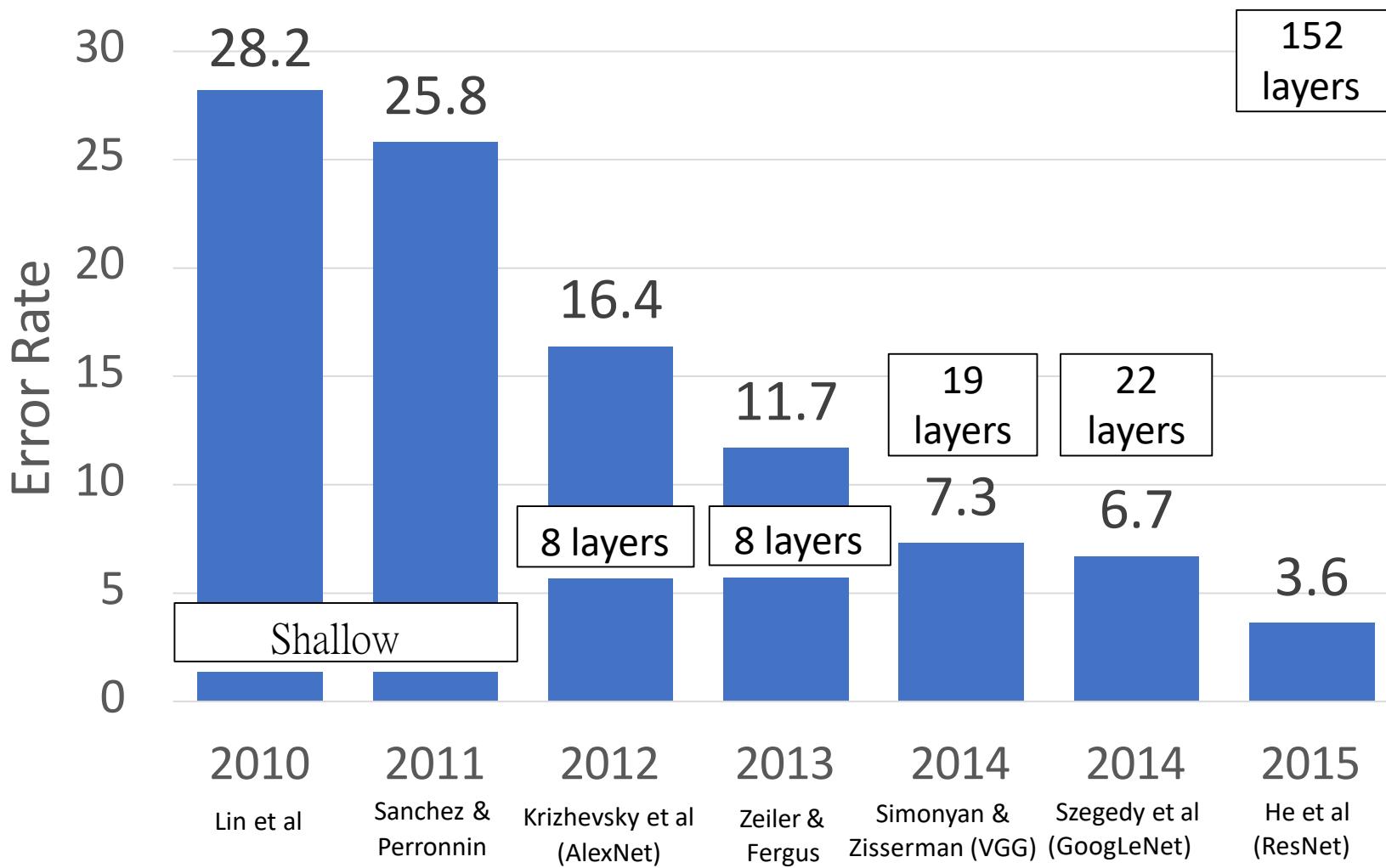


## ResNet

# ImageNet Classification Challenge



# ImageNet Classification Challenge



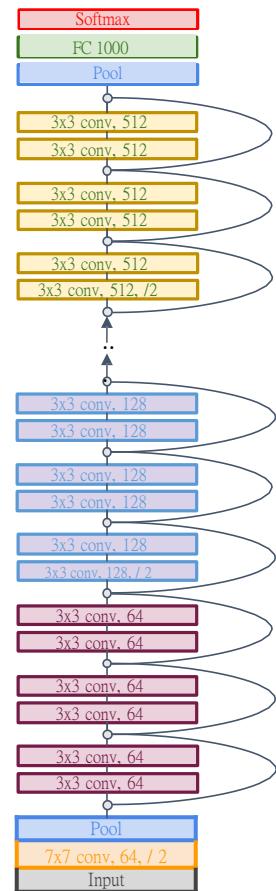
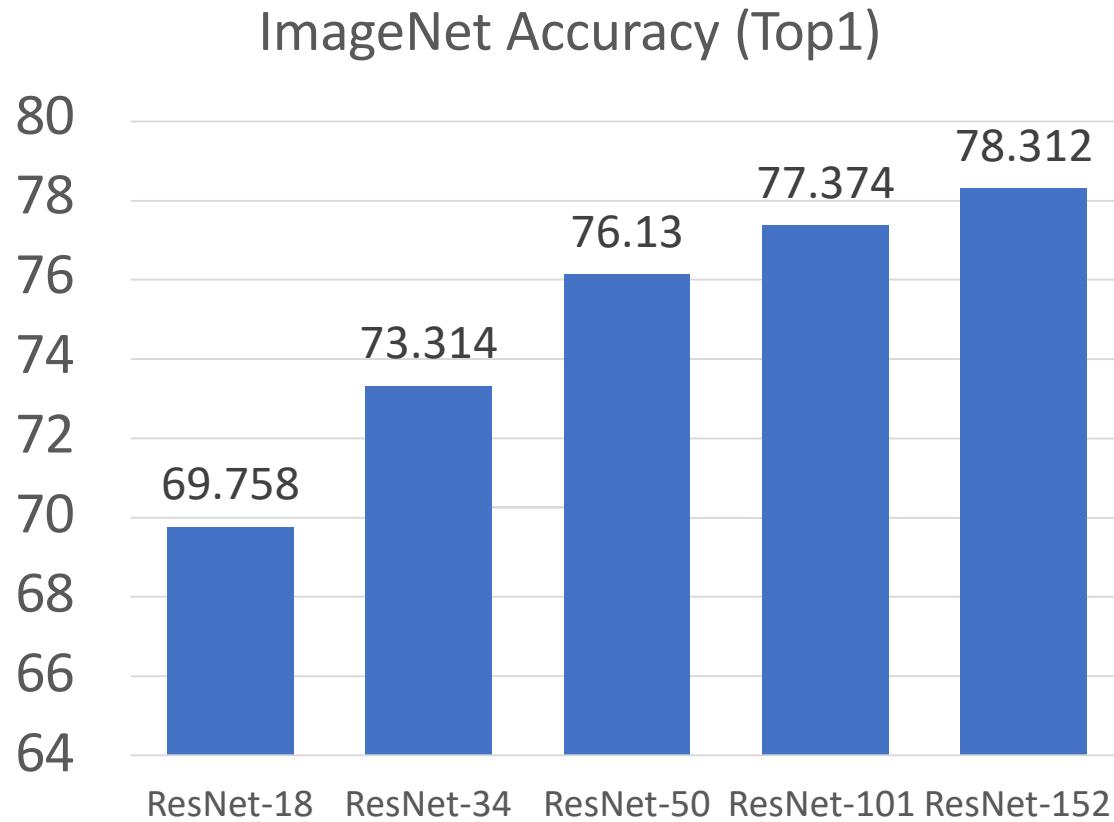
Today:  
More recent CNN  
architectures



# 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?

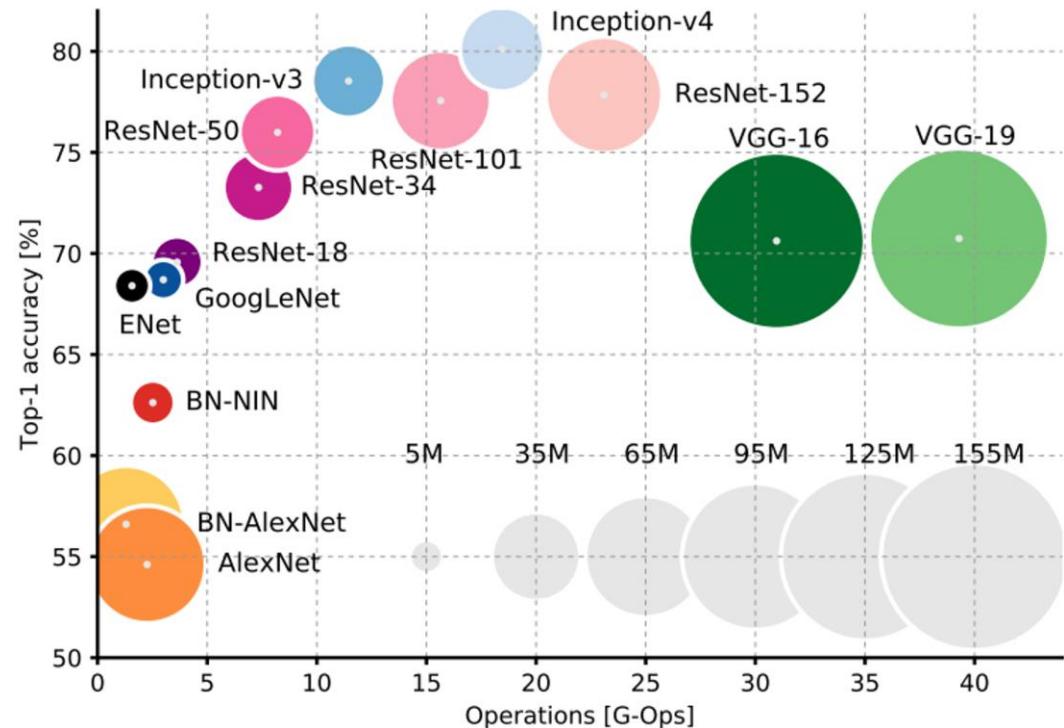
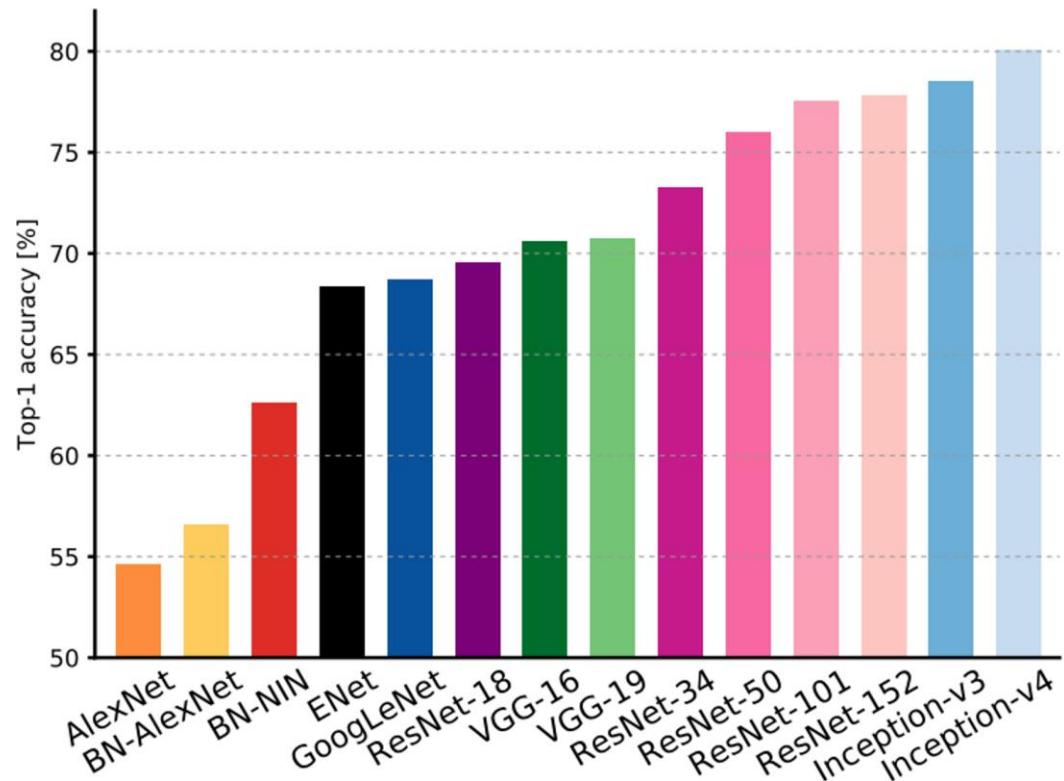
**Floating Point Operations (FLOPs):** How many arithmetic operations does it take to compute the forward pass of the model?

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; some papers say MADD or MACC instead of FLOP
- 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?

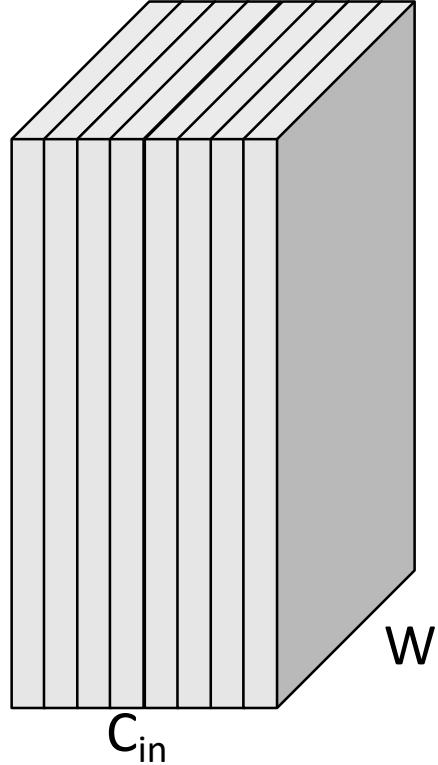
# Comparing Complexity



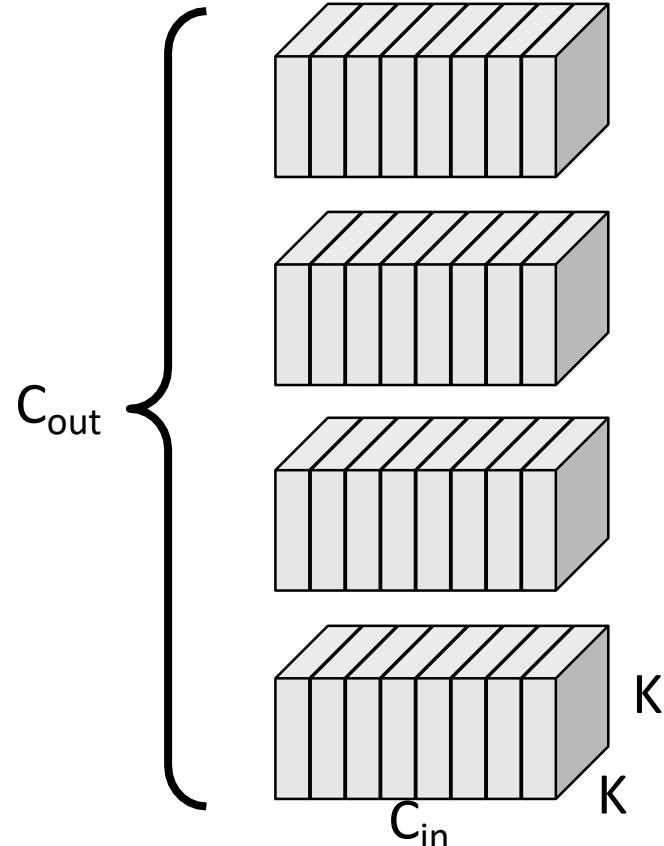
Key ingredient:  
Grouped / Separable convolution

# Recall: Convolution Layer

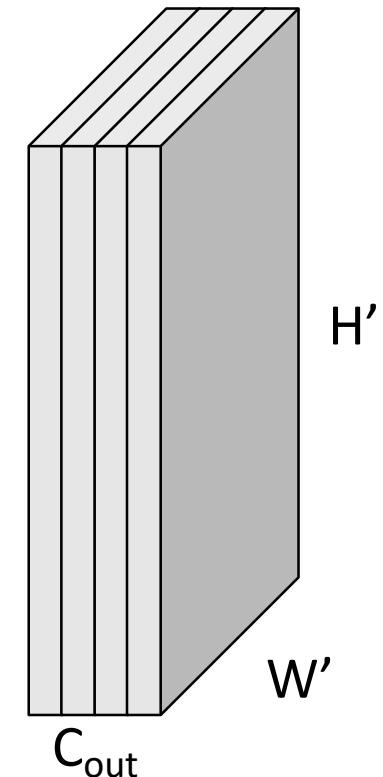
Each filter has the same number of channels as the input



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



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

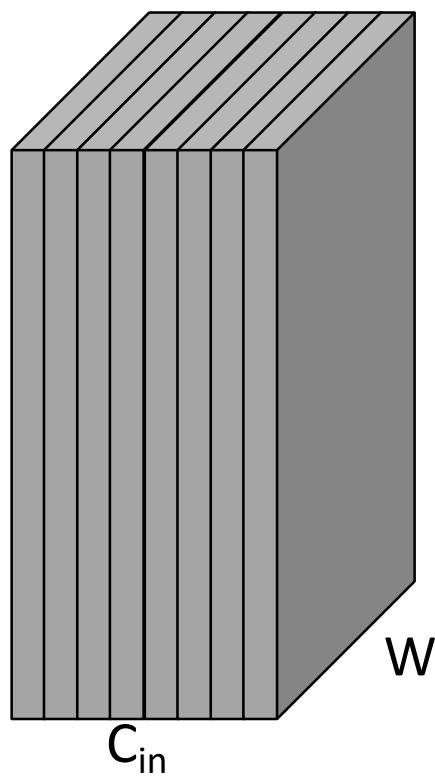


Output:  $C_{out} \times H' \times W'$

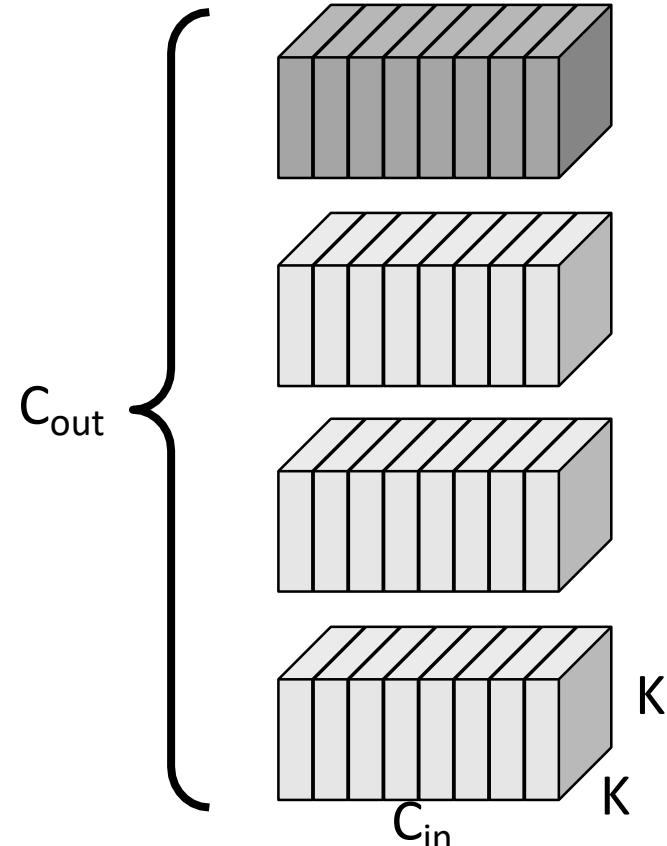
# Recall: Convolution Layer

Each filter has the same number of channels as the input

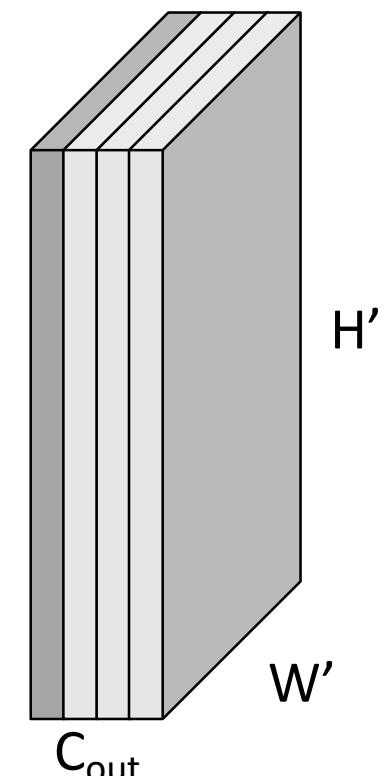
Each plane of the output depends on the full input and one filter



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



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

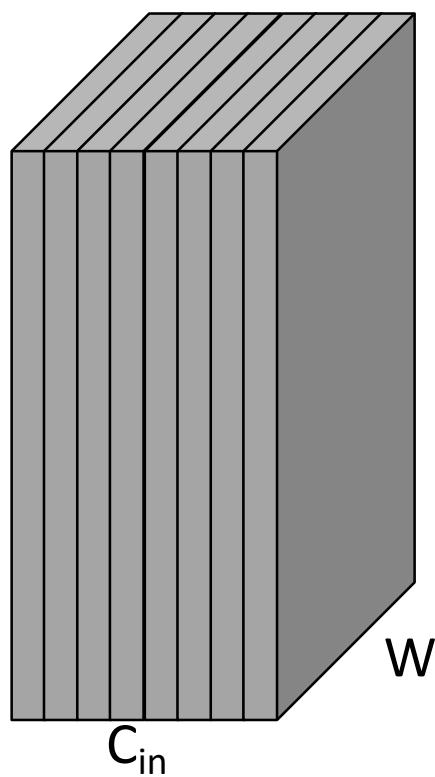


Output:  $C_{out} \times H' \times W'$

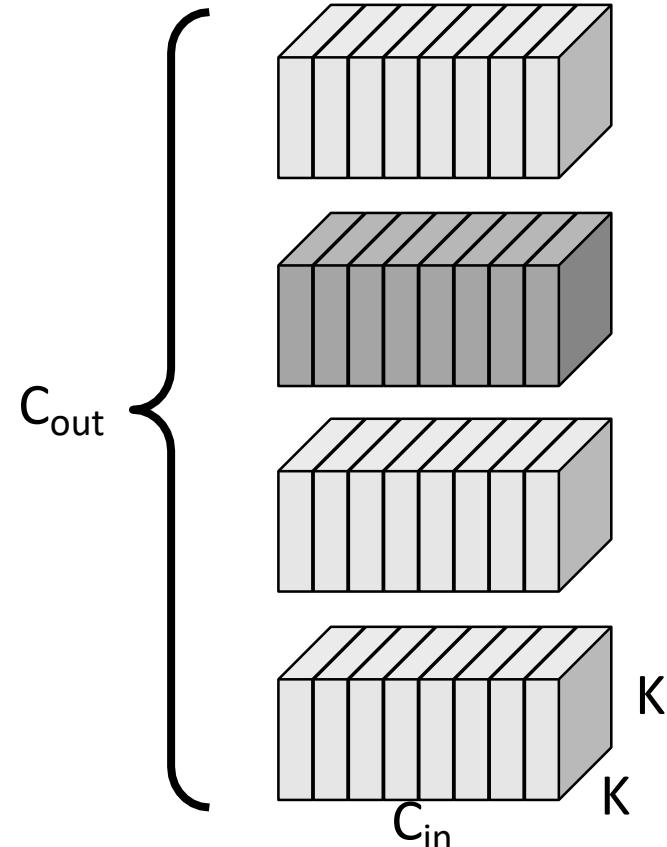
# Recall: Convolution Layer

Each filter has the same number of channels as the input

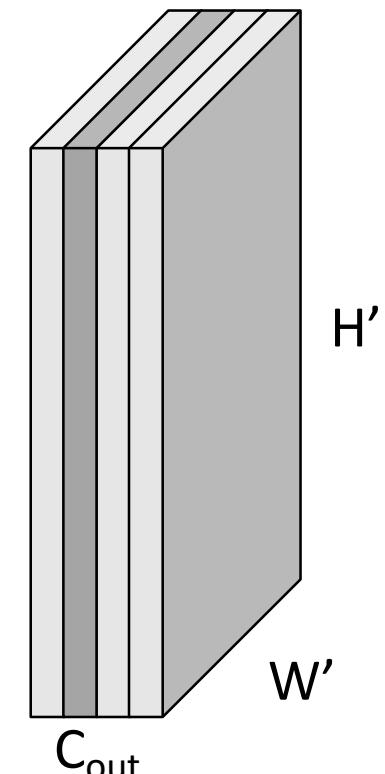
Each plane of the output depends on the full input and one filter



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



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

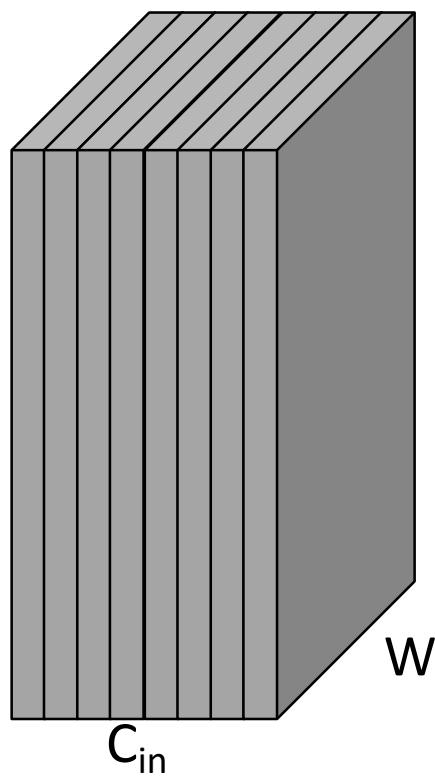


Output:  $C_{out} \times H' \times W'$

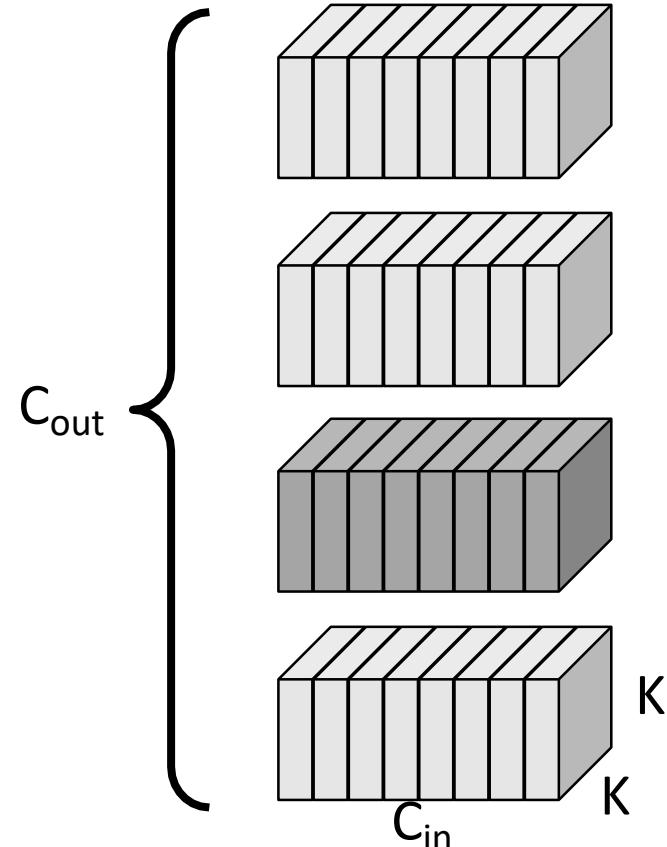
# Recall: Convolution Layer

Each filter has the same number of channels as the input

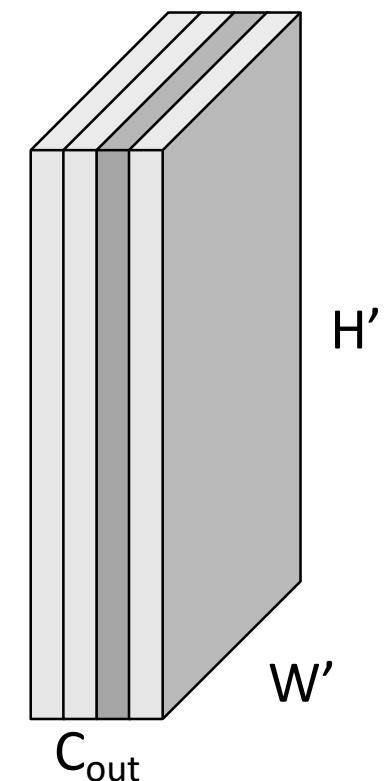
Each plane of the output depends on the full input and one filter



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



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

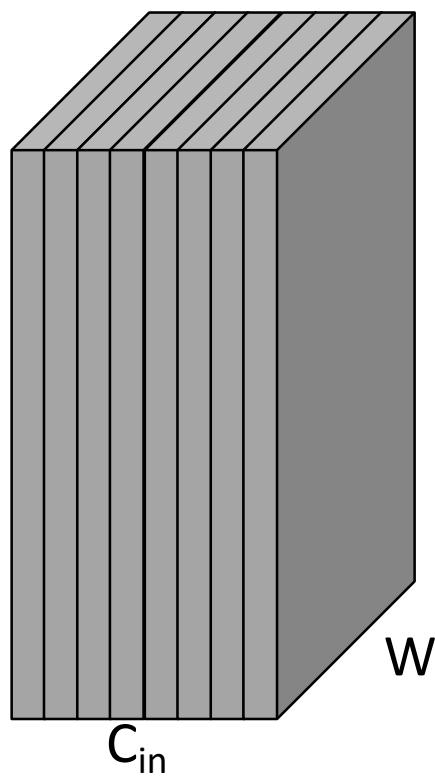


Output:  $C_{out} \times H' \times W'$

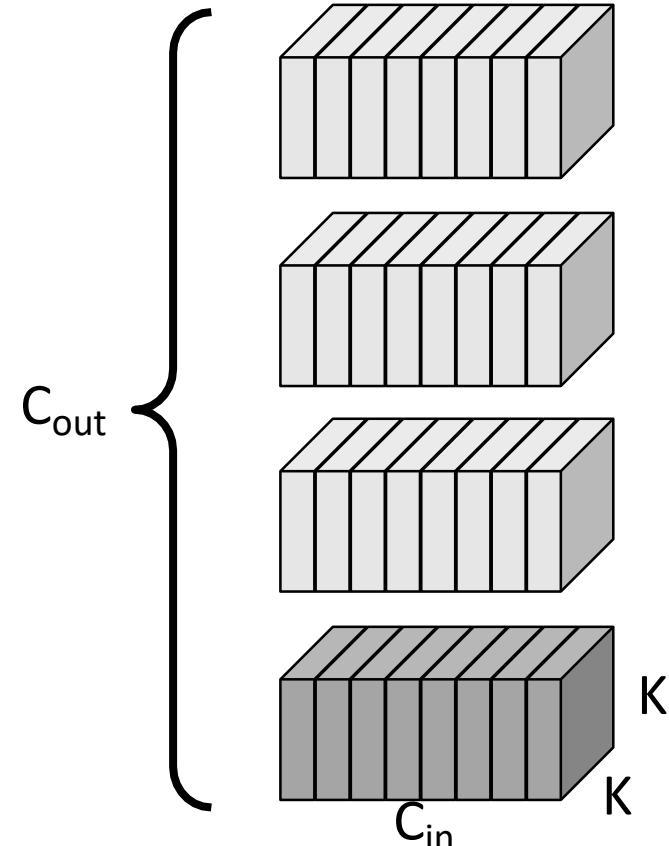
# Recall: Convolution Layer

Each filter has the same number of channels as the input

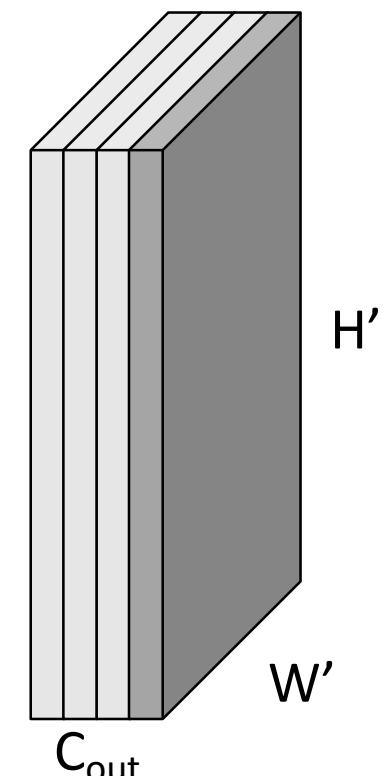
Each plane of the output depends on the full input and one filter



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



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

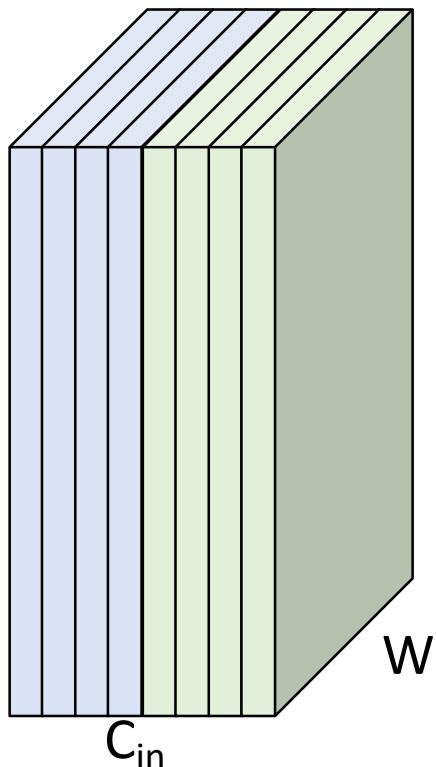


Output:  $C_{out} \times H' \times W'$

# Grouped Convolution

Divide channels of input into G

**groups** with  $(C_{in}/G)$  channels each

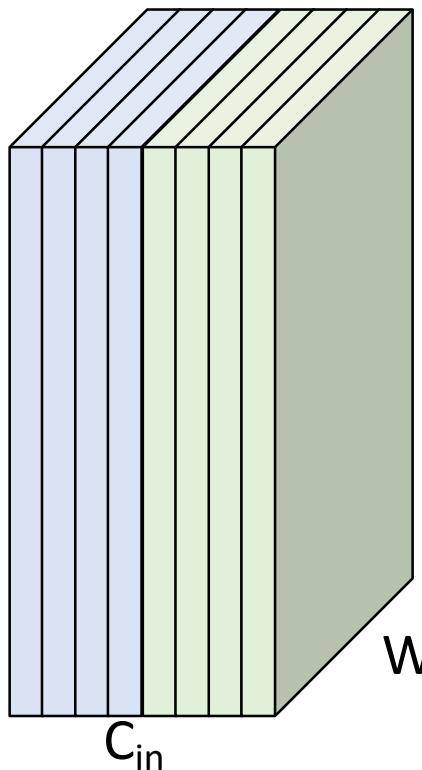


Example:  
 $G = 2$

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

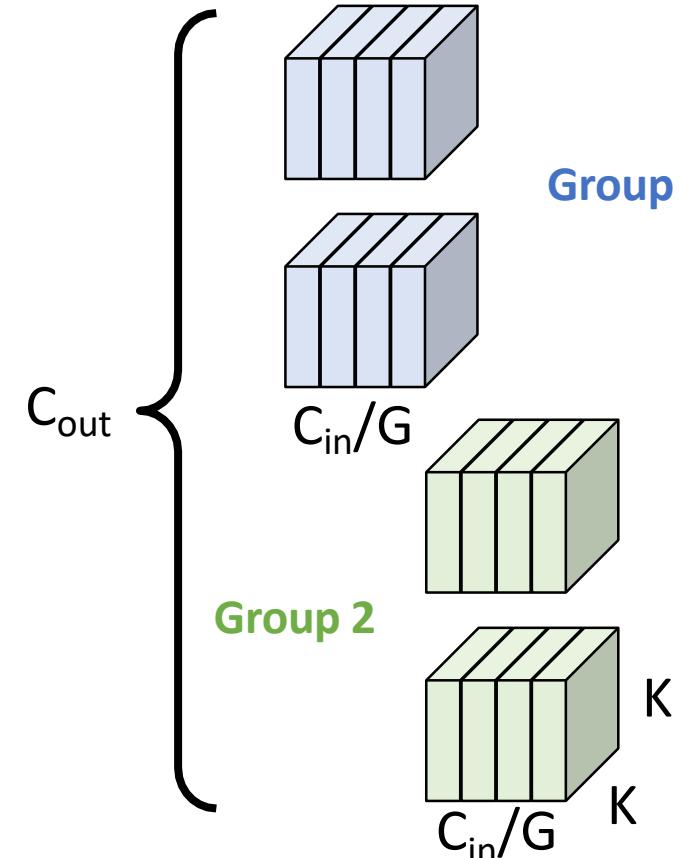
# Grouped Convolution

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

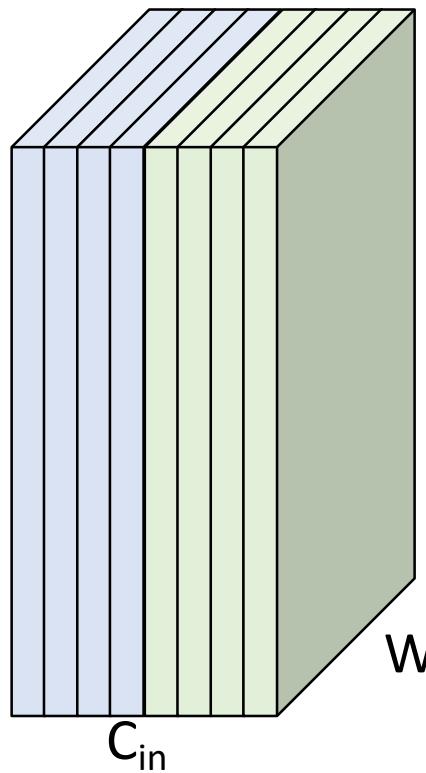


Input:  $C_{in} \times H \times W$

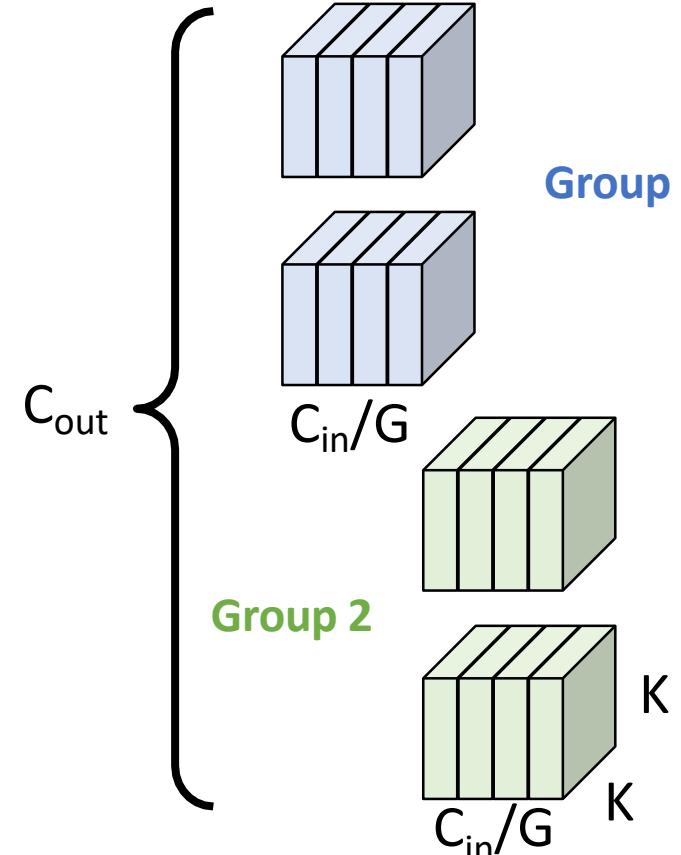
Weights:  $C_{out} \times (C_{in}/G) \times K \times K$

# Grouped Convolution

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



Example:  
 $G = 2$

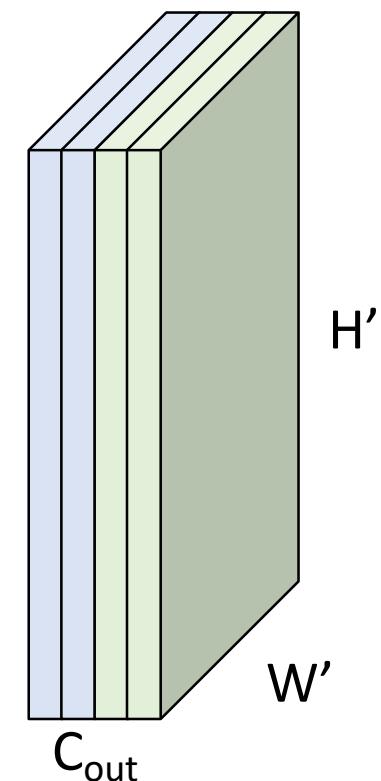


Input:  $C_{in} \times H \times W$

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

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

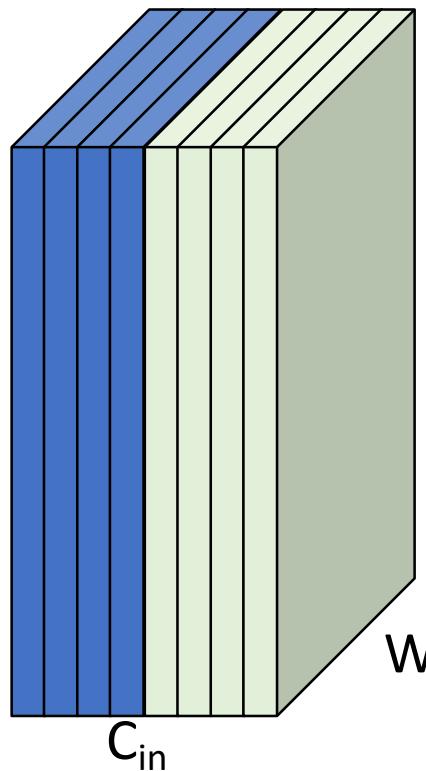
Each plane of the output  
depends on one filter and a  
**subset** of the input channels



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

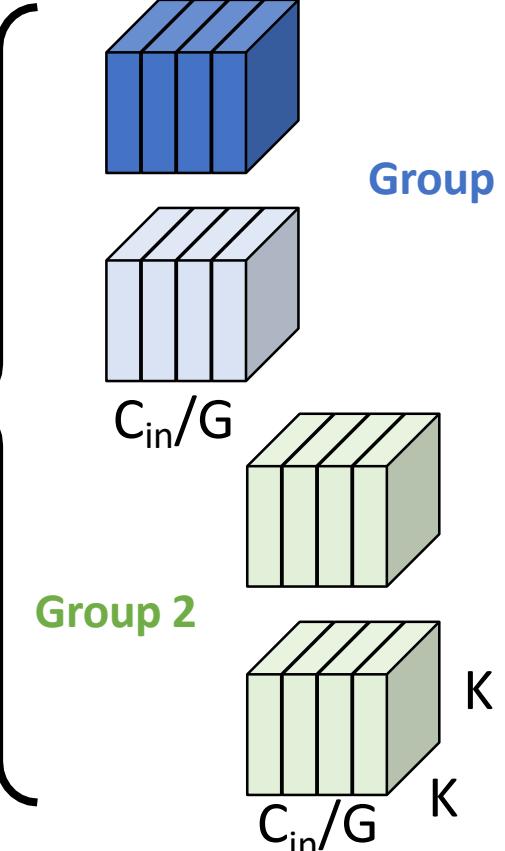
# Grouped Convolution

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



Example:  
 $G = 2$

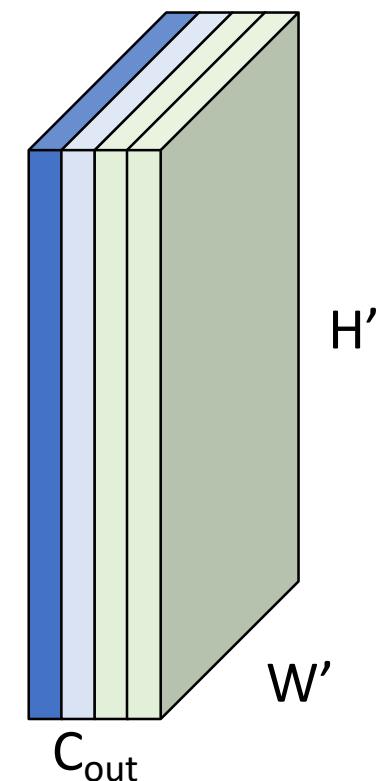
$C_{out}$



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

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

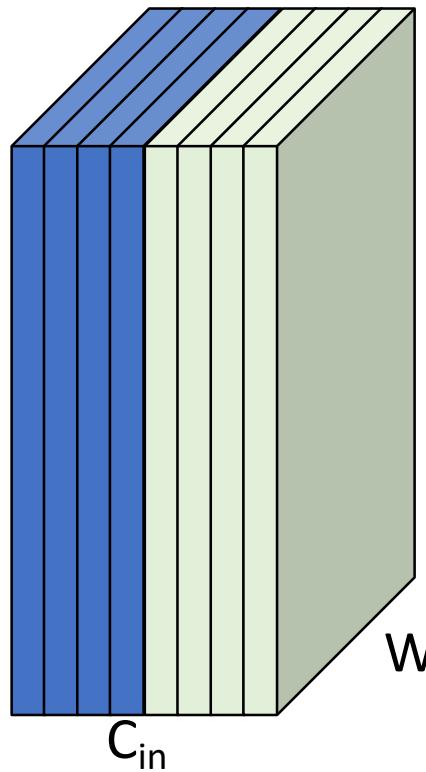
Each plane of the output  
depends on one filter and a  
**subset** of the input channels



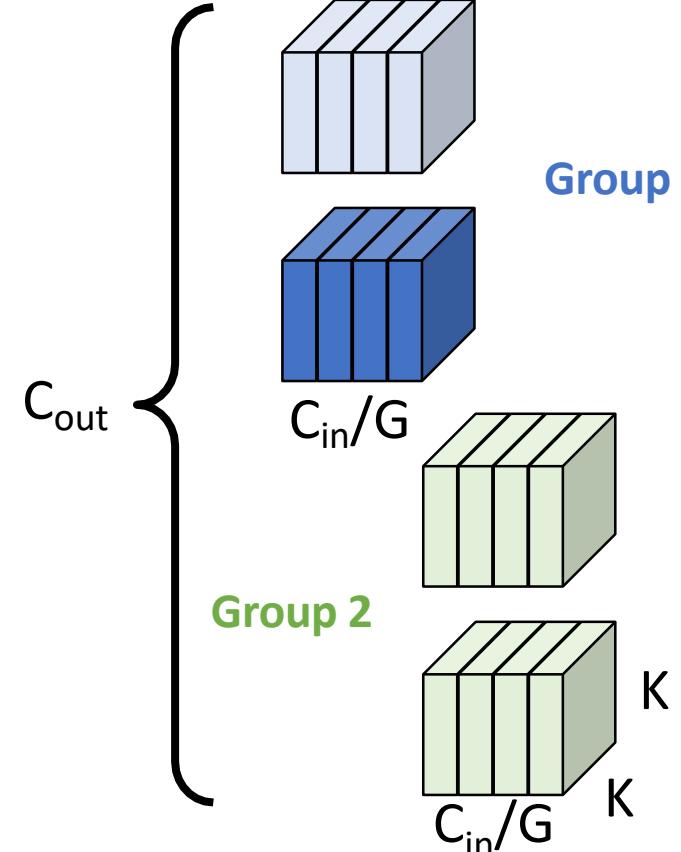
Output:  $C_{out} \times H' \times W'$

# Grouped Convolution

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



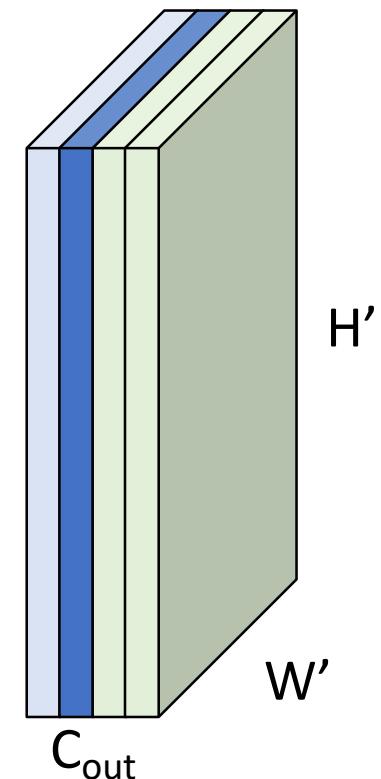
Example:  
 $G = 2$



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

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

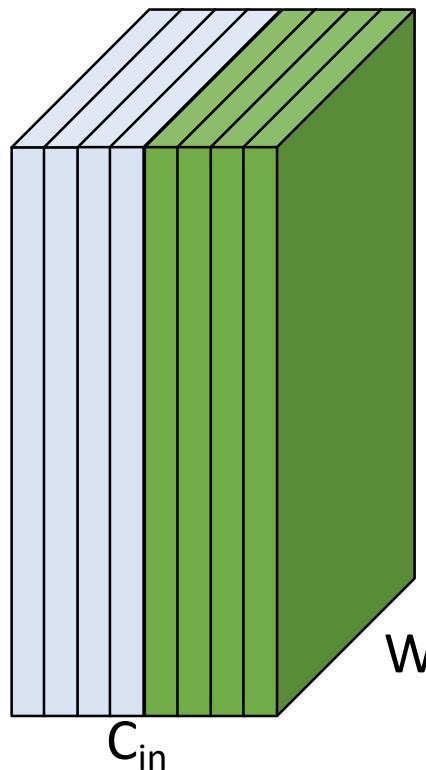
Each plane of the output  
depends on one filter and a  
**subset** of the input channels



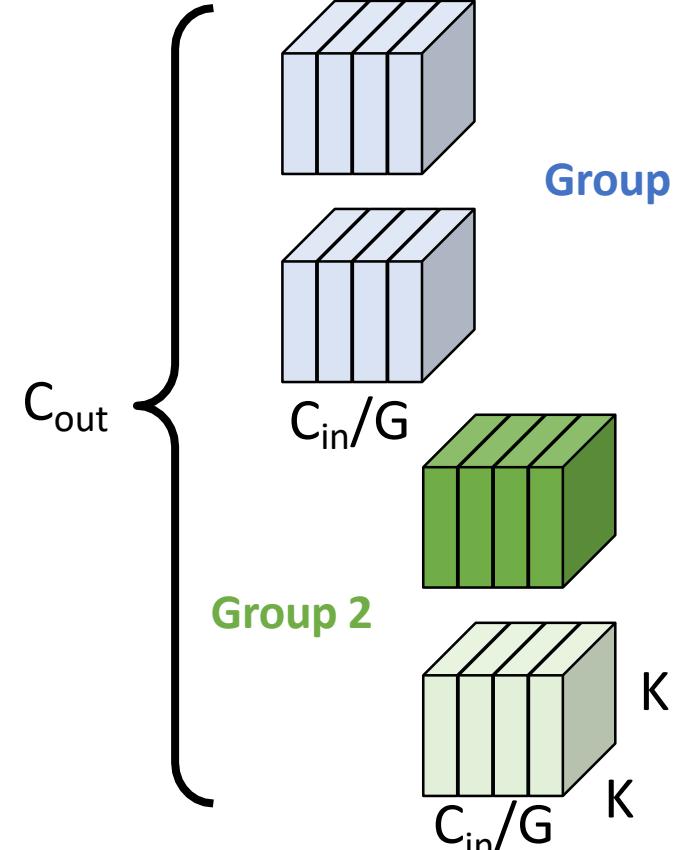
Output:  $C_{out} \times H' \times W'$

# Grouped Convolution

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



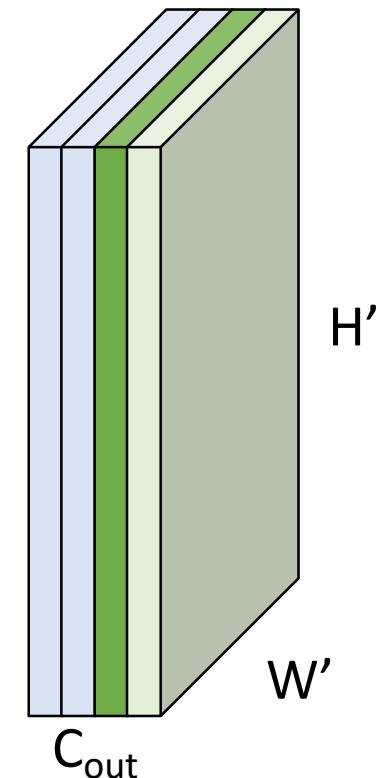
Example:  
 $G = 2$



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

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

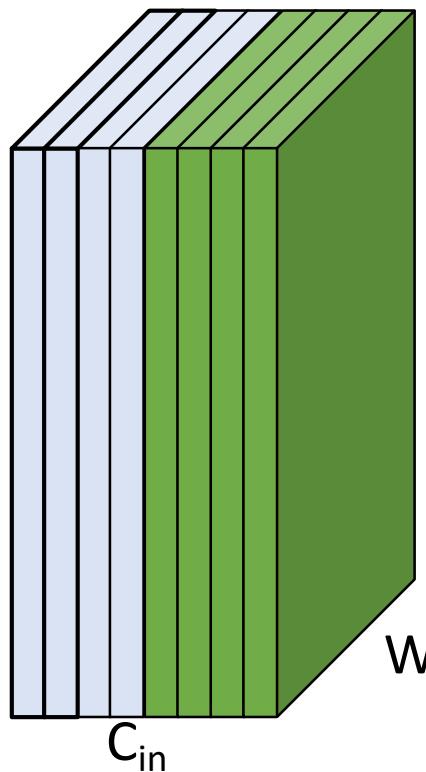
Each plane of the output  
depends on one filter and a  
**subset** of the input channels



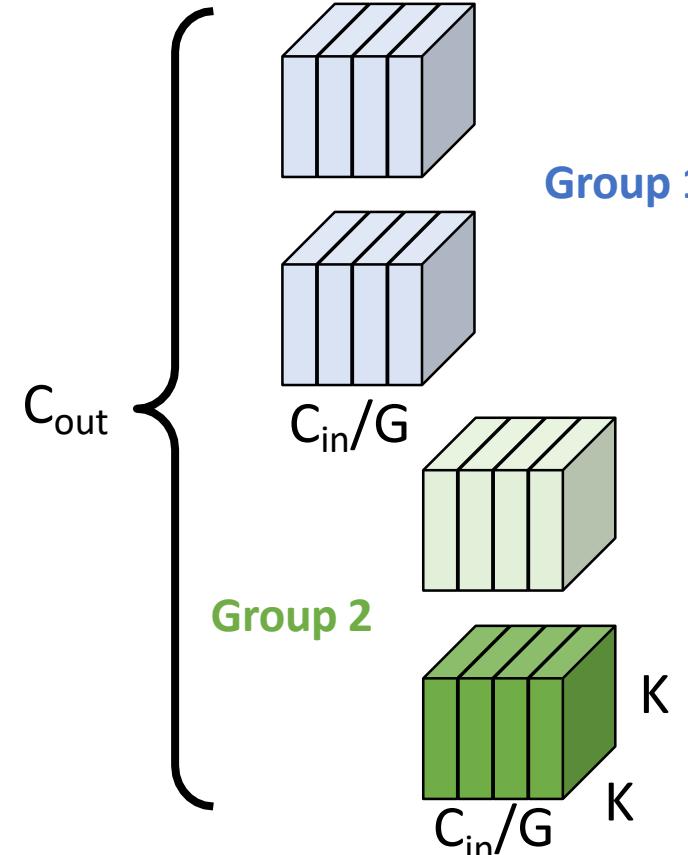
Output:  $C_{out} \times H' \times W'$

# Grouped Convolution

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



Example:  
 $G = 2$

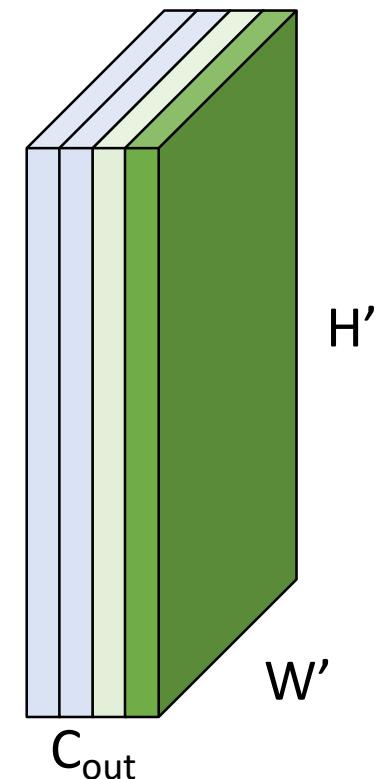


Input:  $C_{in} \times H \times W$

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

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

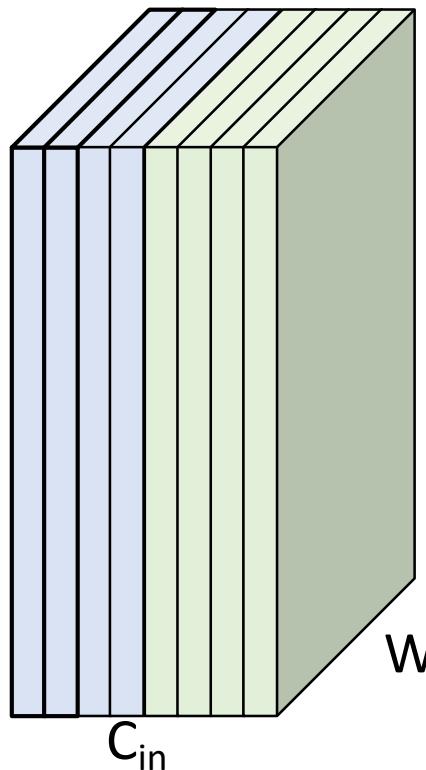
Each plane of the output  
depends on one filter and a  
**subset** of the input channels



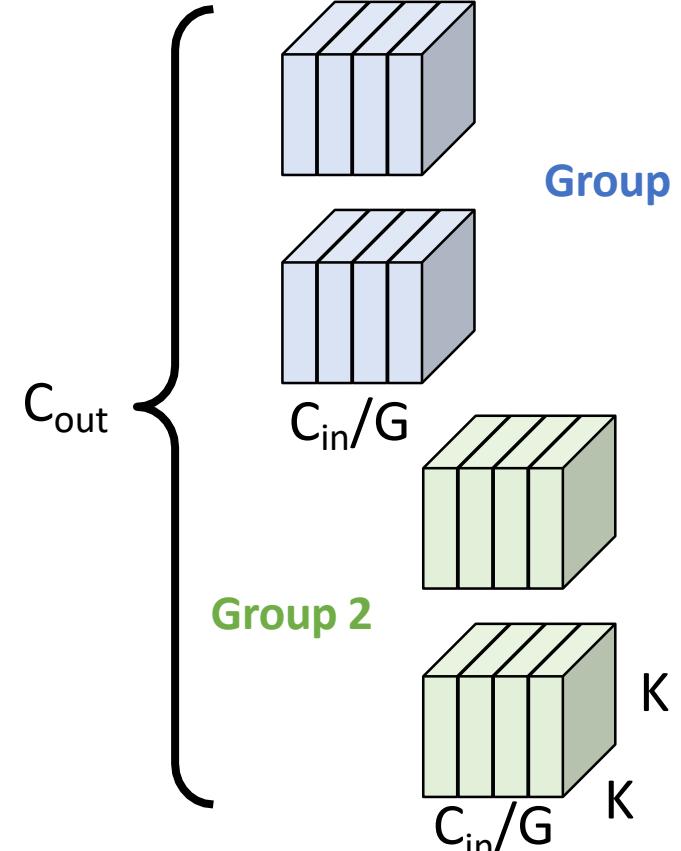
Output:  $C_{out} \times H' \times W'$

# Grouped Convolution

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



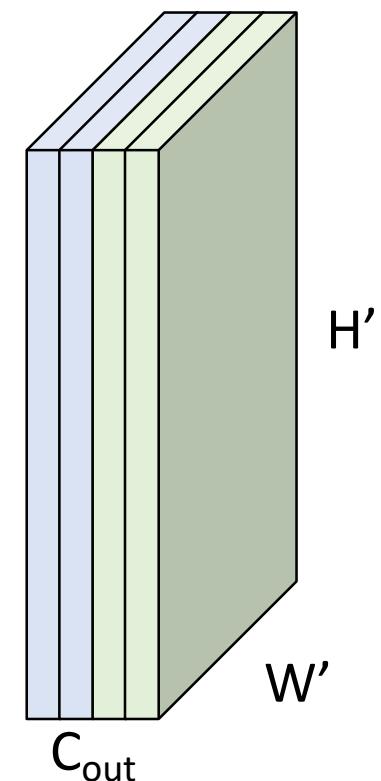
Example:  
 $G = 2$



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

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

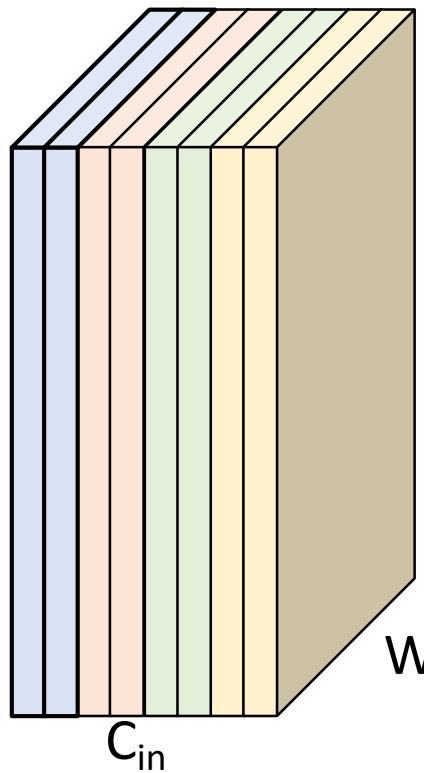
Each plane of the output  
depends on one filter and a  
**subset** of the input channels



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

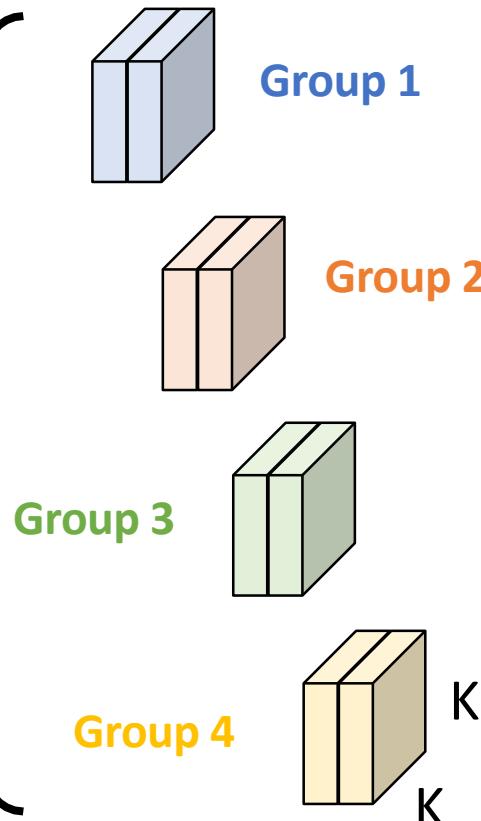
# Grouped Convolution

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



Example:  
 $G = 4$

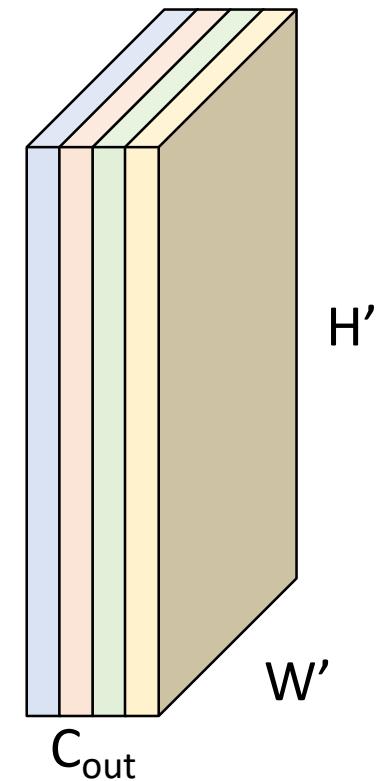
Divide filters into G groups;  
each group looks at a  
**subset** of input channels



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

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

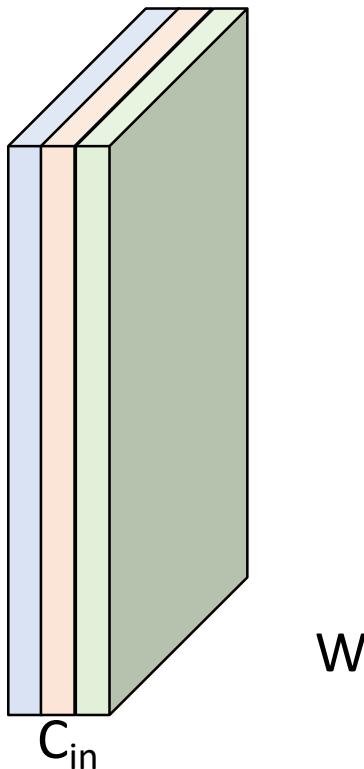
Each plane of the output  
depends on one filter and a  
**subset** of the input channels



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

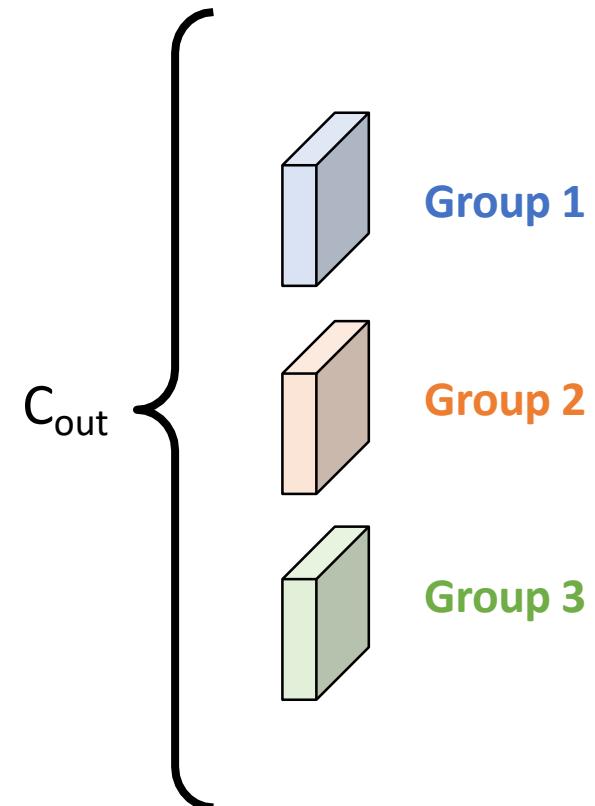
# Special Case: Depthwise Convolution

Number of groups equals number of input channels



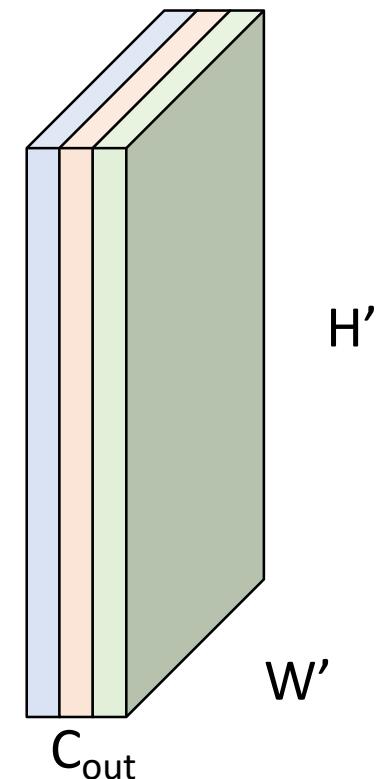
Input:  $C_{in} \times H \times W$

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



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

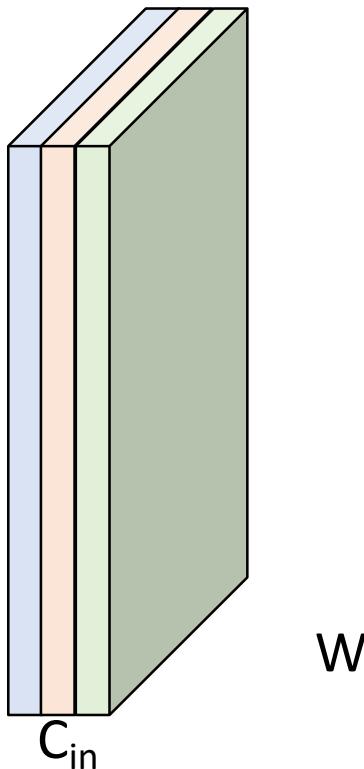
Output only mixes *spatial* information from input;  
*channel* information not mixed



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

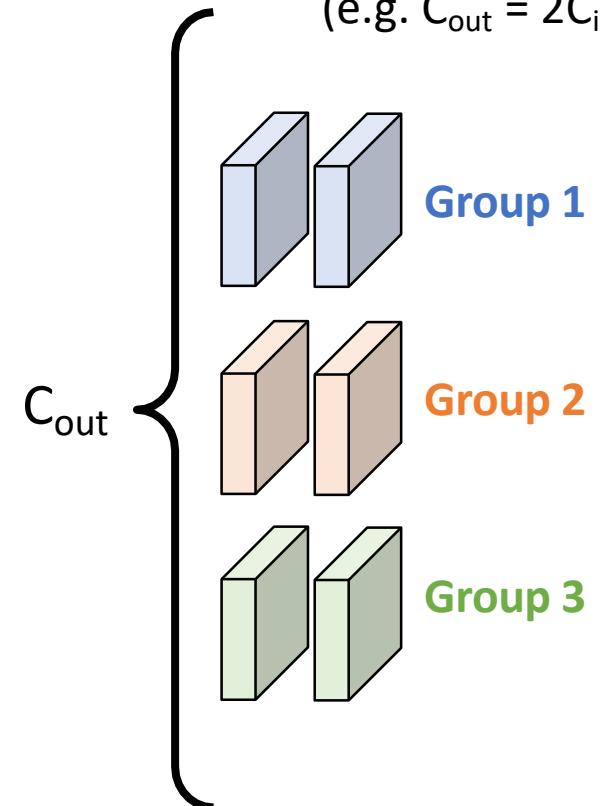
# Special Case: Depthwise Convolution

Number of groups equals number of input channels



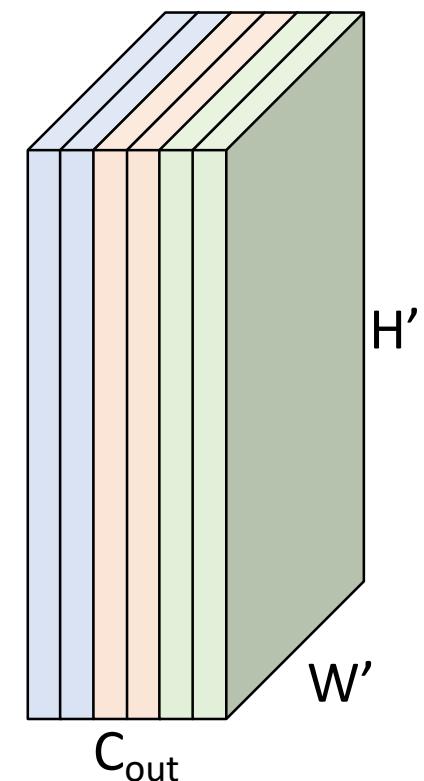
Input:  $C_{in} \times H \times W$

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



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

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



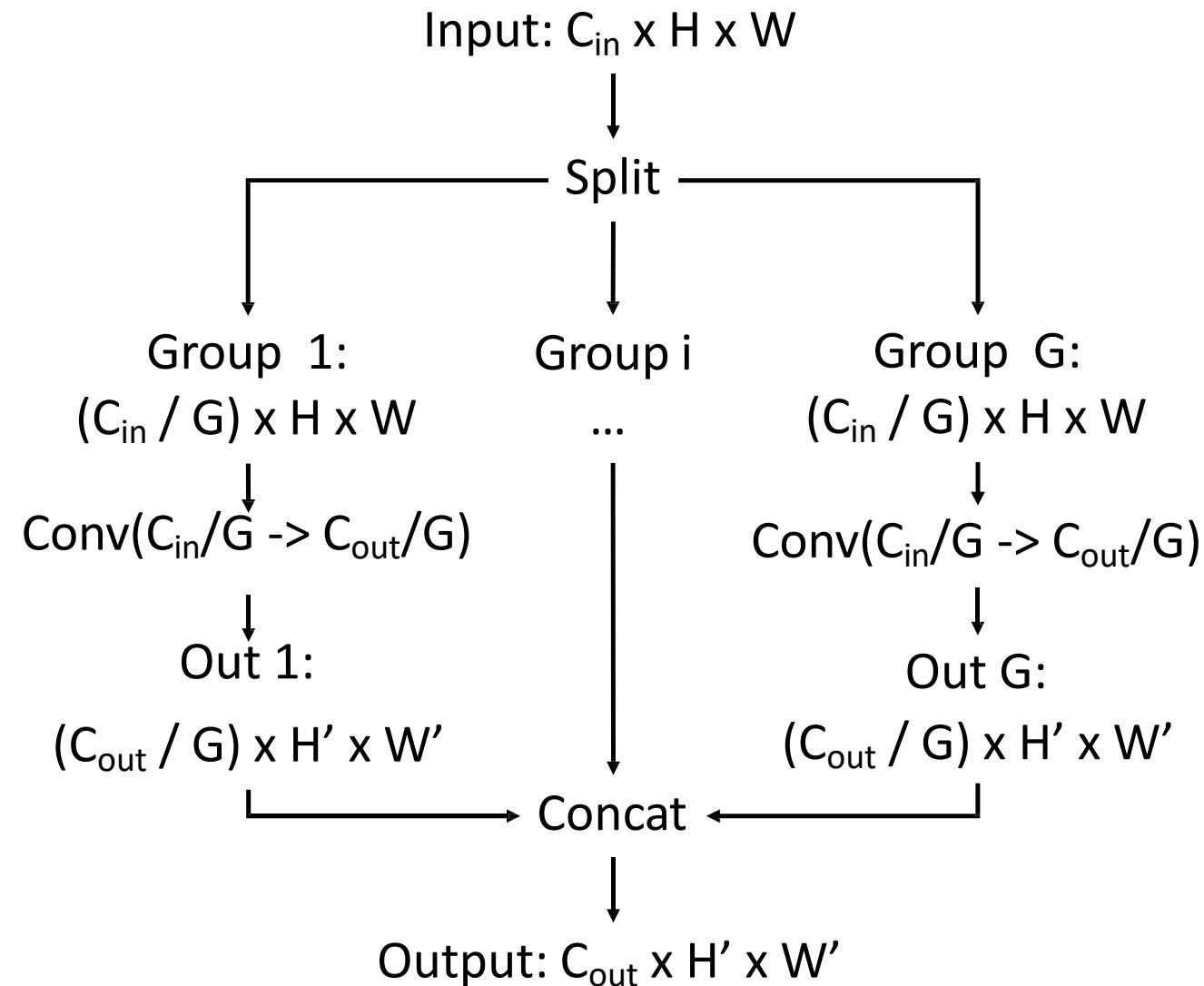
Output:  $C_{out} \times H' \times W'$

# Grouped Convolution

## Grouped Convolution (G groups):

G parallel conv layers; each “sees”  
 $C_{in}/G$  input channels and produces  
 $C_{out}/G$  output channels

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



# Grouped Convolution vs Standard Convolution

## Grouped Convolution (G groups):

G parallel conv layers; each “sees”

$C_{in}/G$  input channels and produces

$C_{out}/G$  output channels

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

Split to  $G \times [(C_{in} / G) \times H \times W]$

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

G parallel convolutions

Output:  $G \times [(C_{out} / G) \times H' \times W']$

Concat to  $C_{out} \times H' \times W'$

FLOPs:  $C_{out} C_{in} K^2 HW / G$

## Standard Convolution (groups=1)

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

Weight:  $C_{out} \times C_{in} \times K \times K$

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

FLOPs:  $C_{out} C_{in} K^2 HW$

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

Using G groups reduces  
FLOPs by a factor of G!

# Grouped Convolution in PyTorch

PyTorch convolution gives an option for groups!

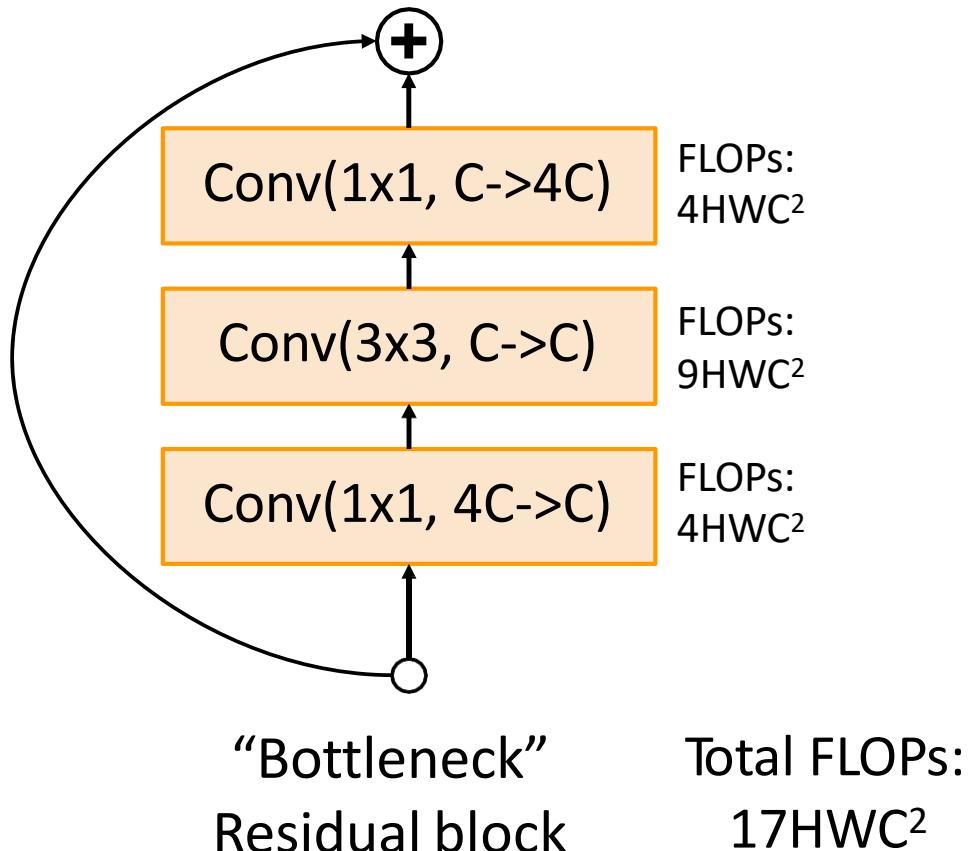
## Conv2d

---

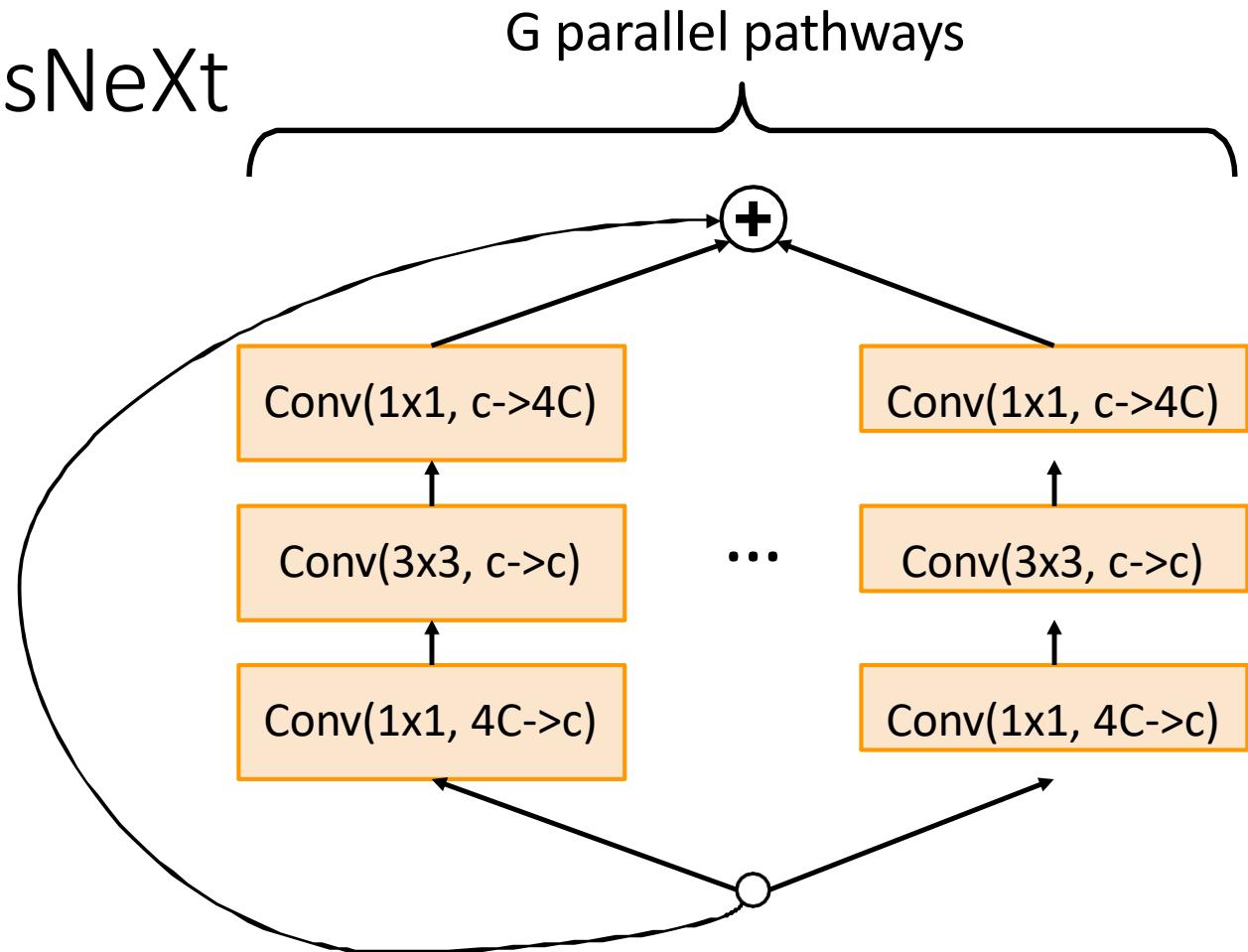
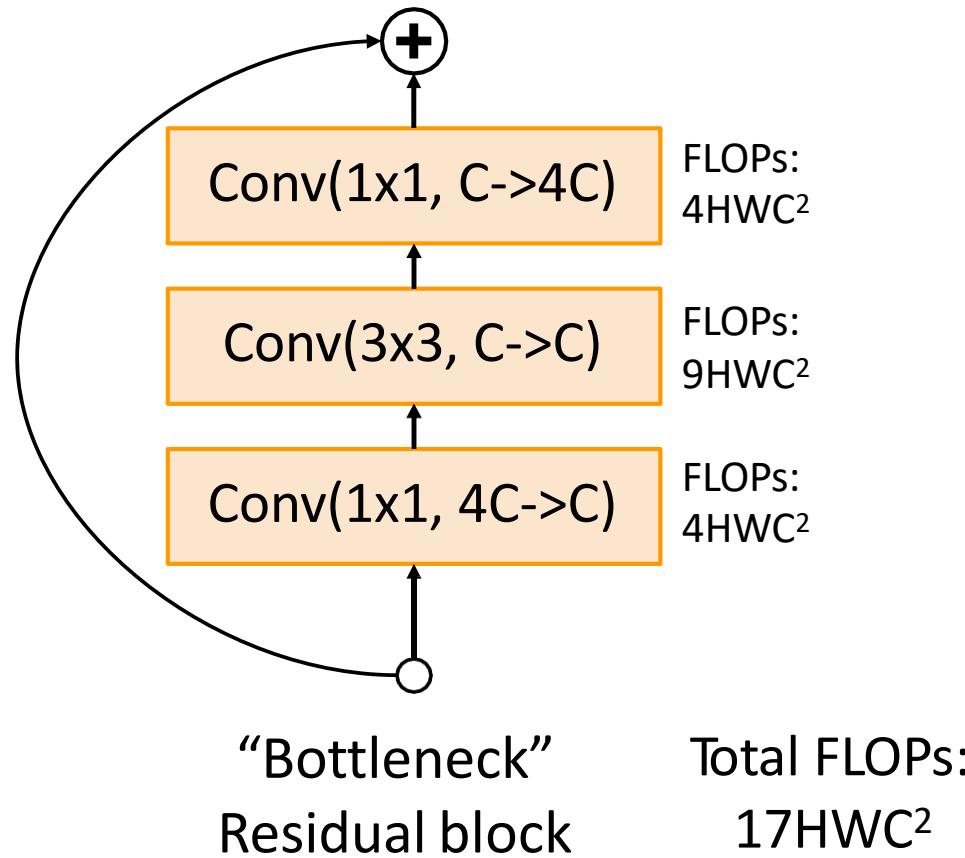
**CLASS** `torch.nn.Conv2d(in_channels, out_channels, kernel_size,  
  stride=1, padding=0, dilation=1, groups=1, bias=True,  
  padding_mode='zeros')`

[SOURCE]

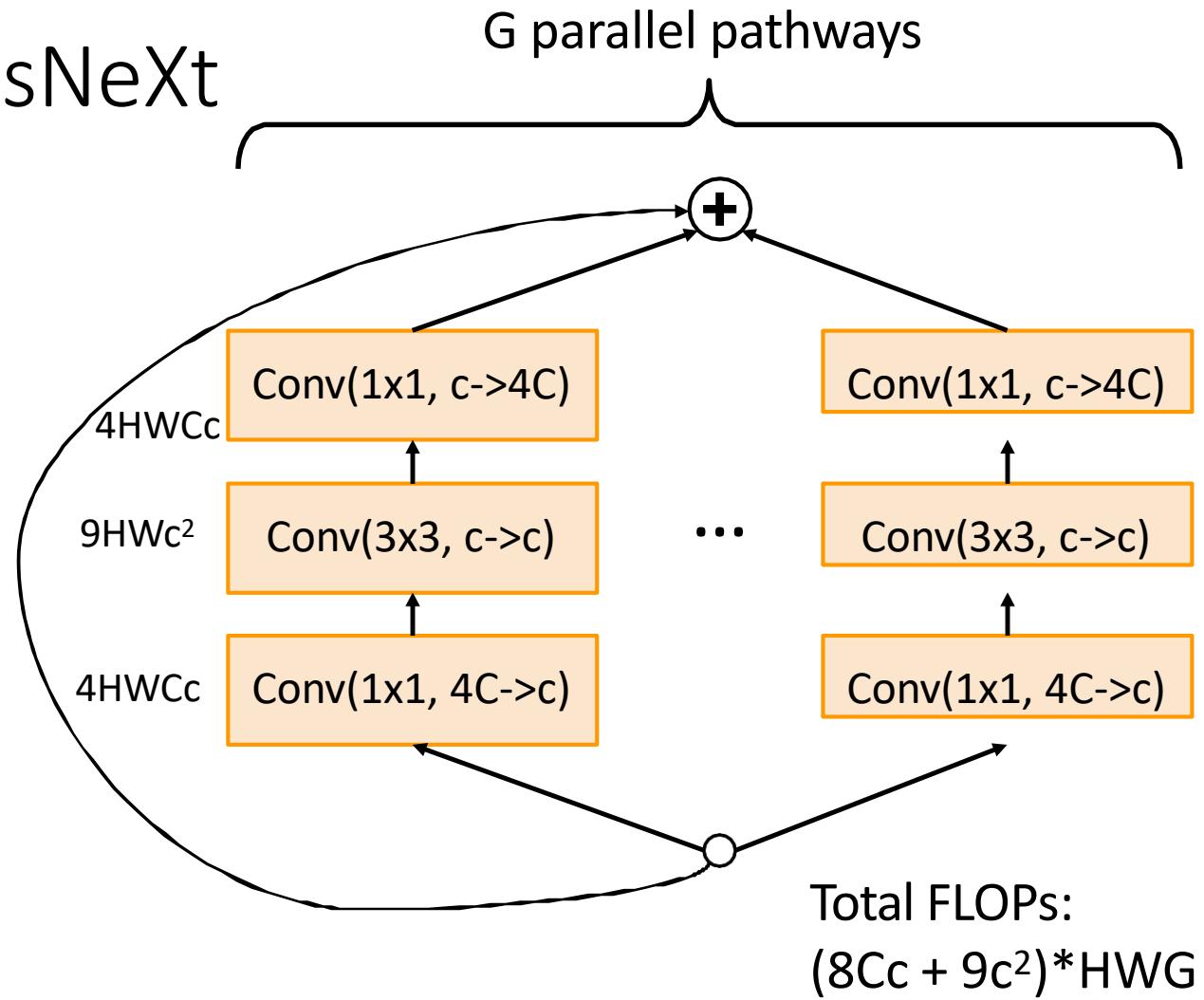
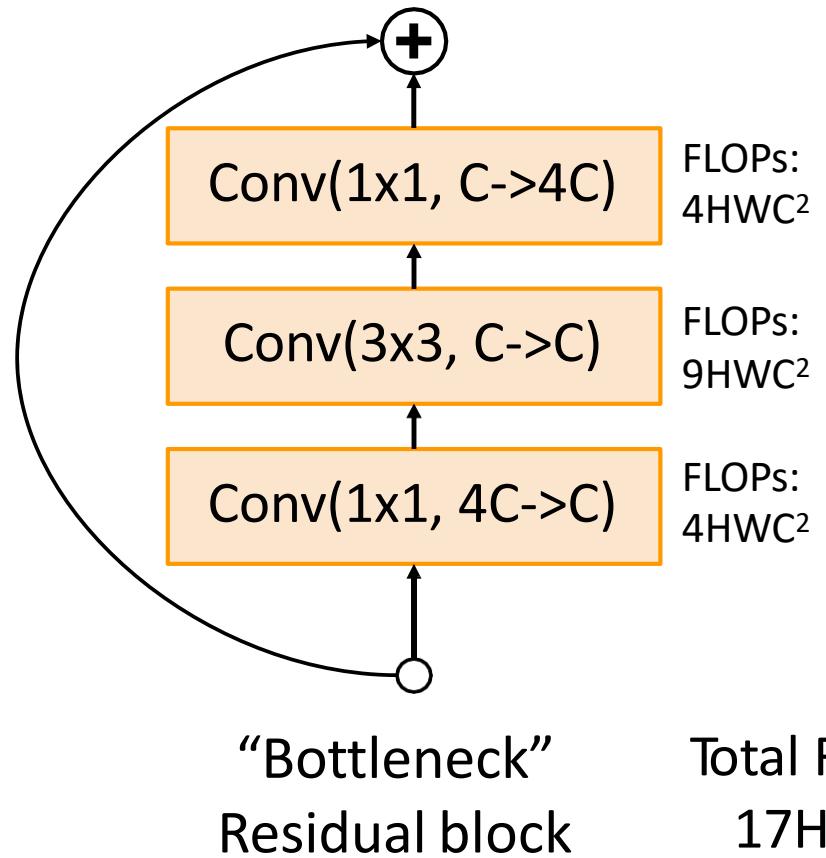
# Improving ResNets



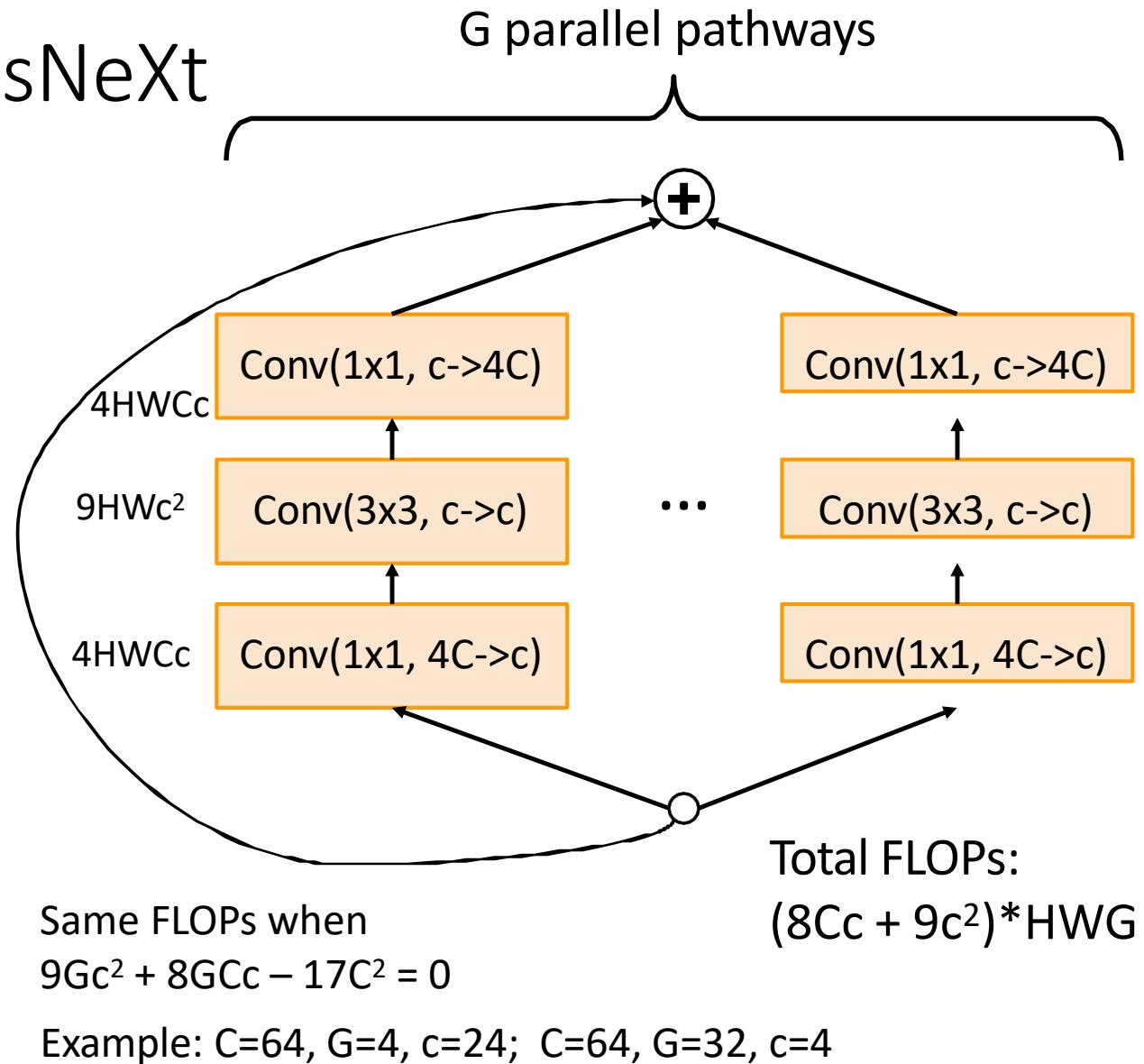
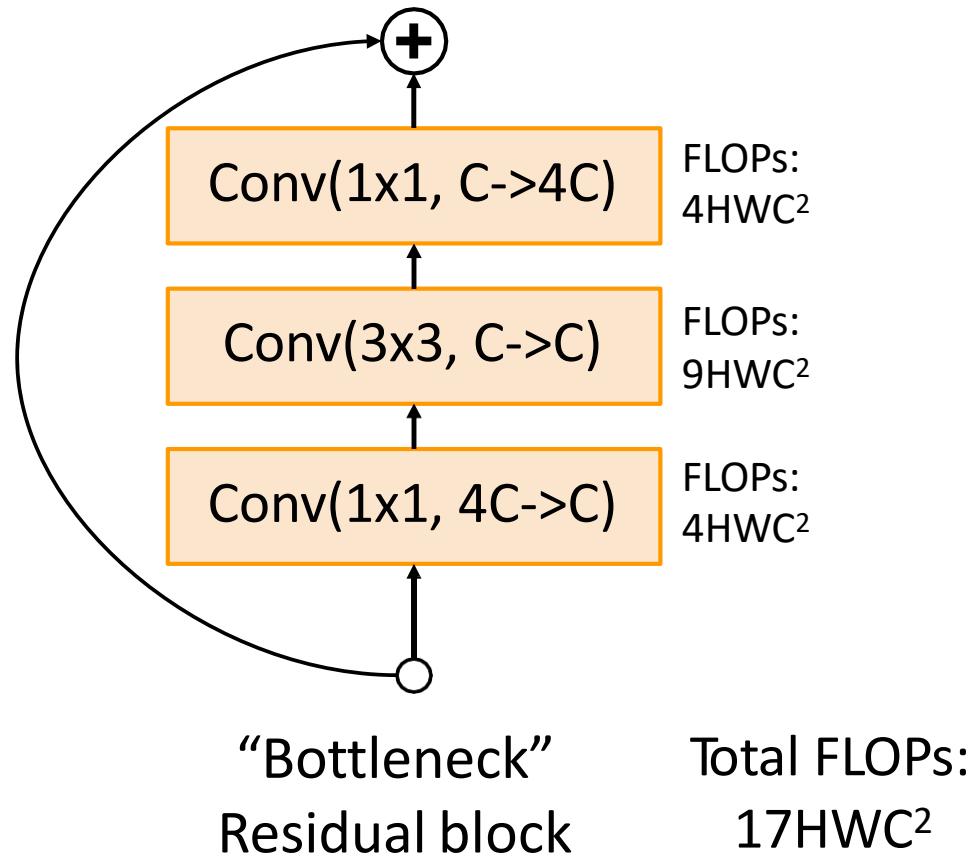
# Improving ResNets: ResNeXt



# Improving ResNets: ResNeXt

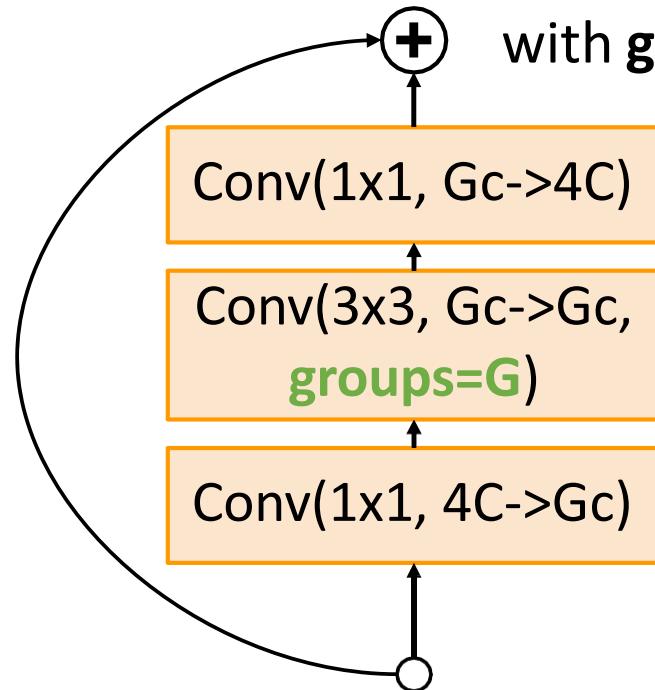


# Improving ResNets: ResNeXt

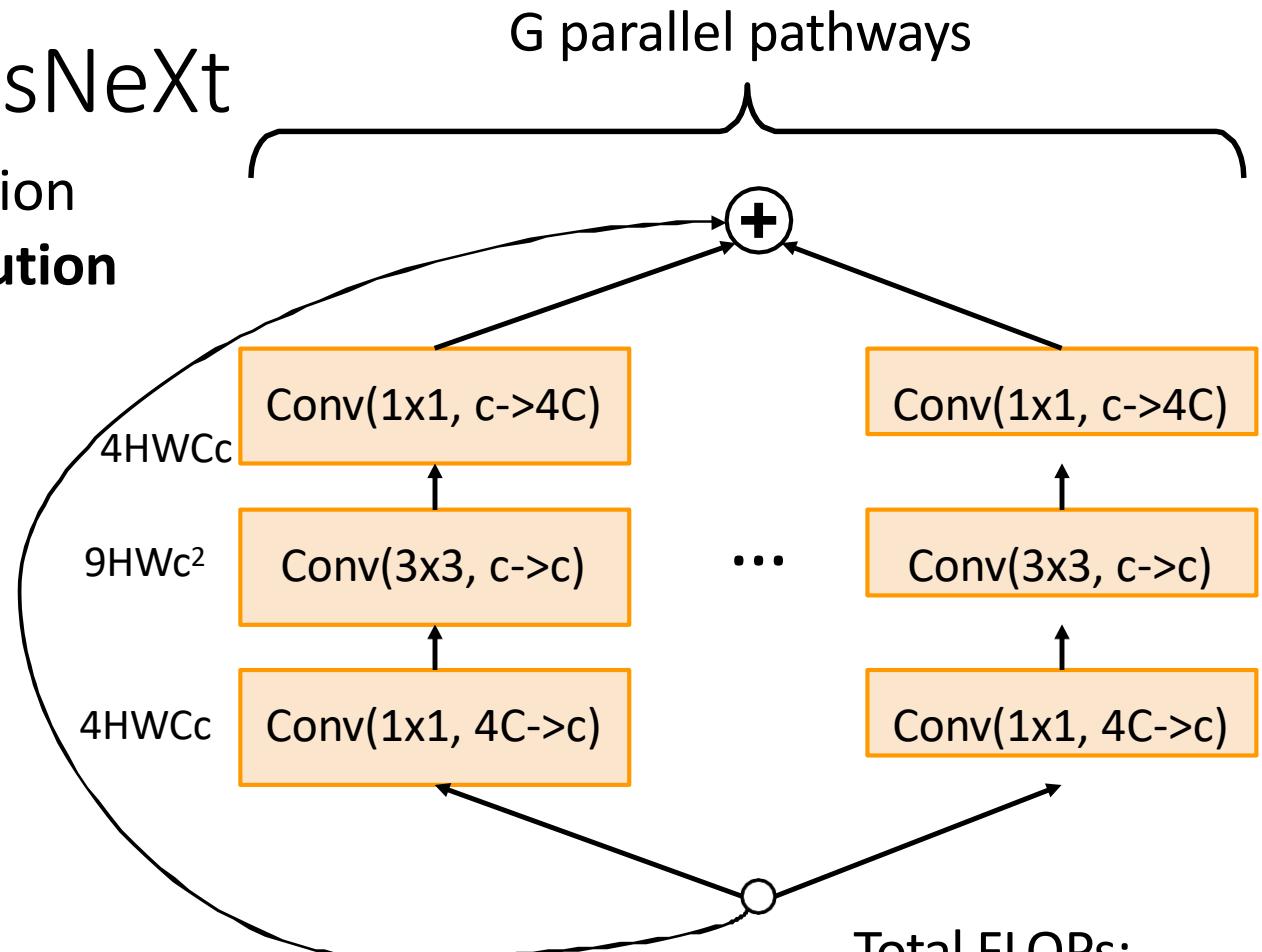


# Improving ResNets: ResNeXt

Equivalent formulation  
with grouped convolution



ResNeXt block:  
Grouped convolution



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

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

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

# ResNeXt: Maintain computation by adding groups!

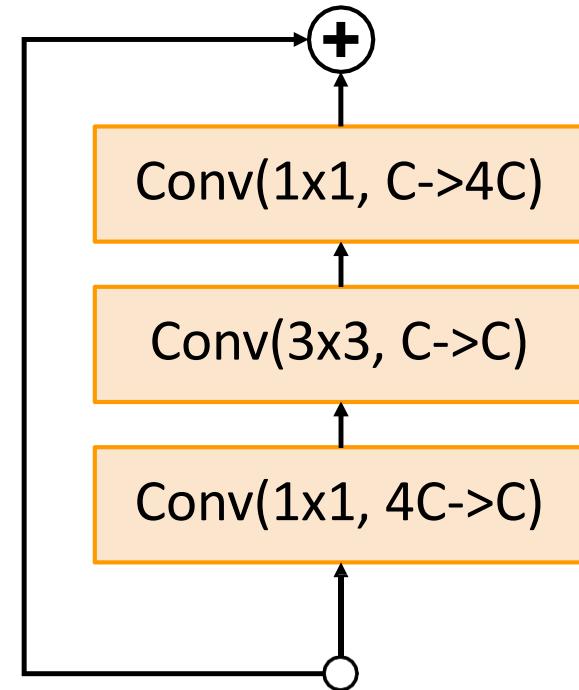
Model	Groups	Group width	Top-1 Error
ResNet-50	1	64	23.9
ResNeXt-50	2	40	23
ResNeXt-50	4	24	22.6
ResNeXt-50	8	14	22.3
ResNeXt-50	32	4	22.2

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)

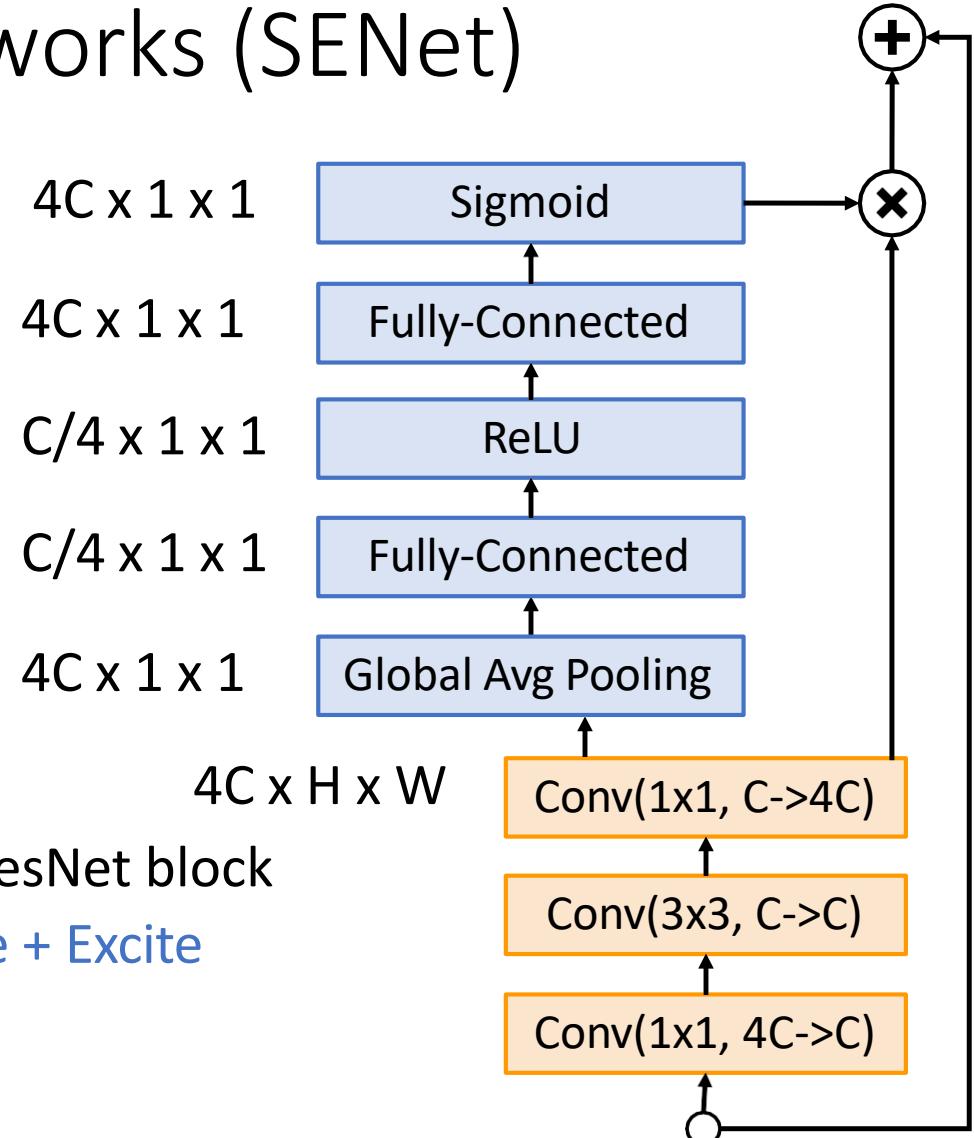
# Squeeze-and-Excitation Networks (SENet)



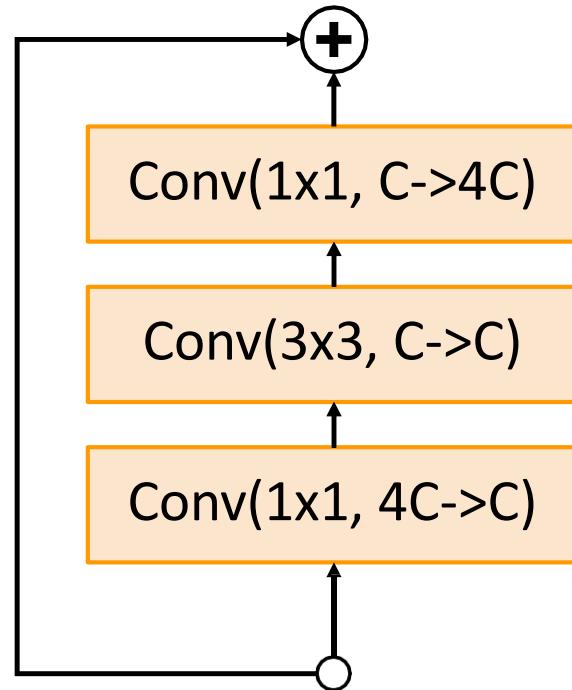
Bottleneck ResNet block

Adds **global context** to each ResNet block

Bottleneck ResNet block  
with Squeeze + Excite

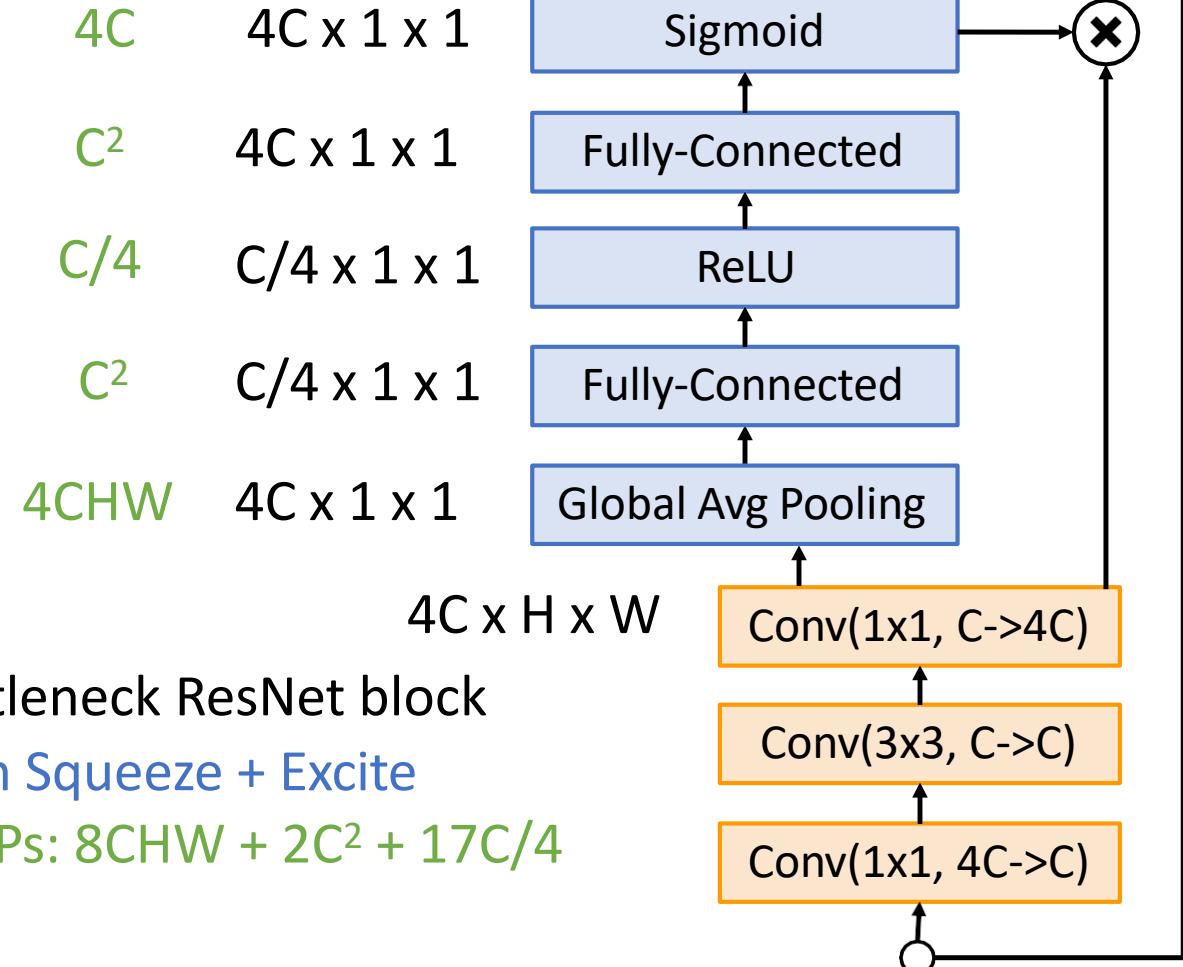


# Squeeze-and-Excitation Networks (SENet)



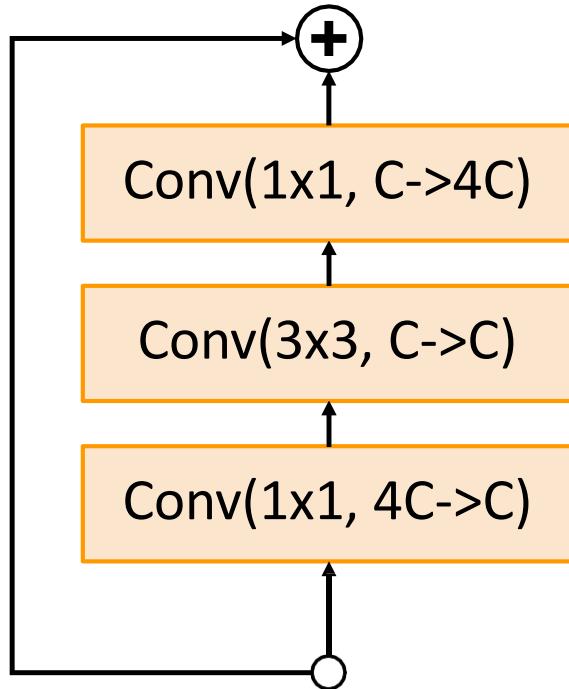
FLOPs:  $17HWC^2$

Adds **global context** to each ResNet block



FLOPs:  $8CHW + 2C^2 + 17C/4$

# Squeeze-and-Excitation Networks (SENet)

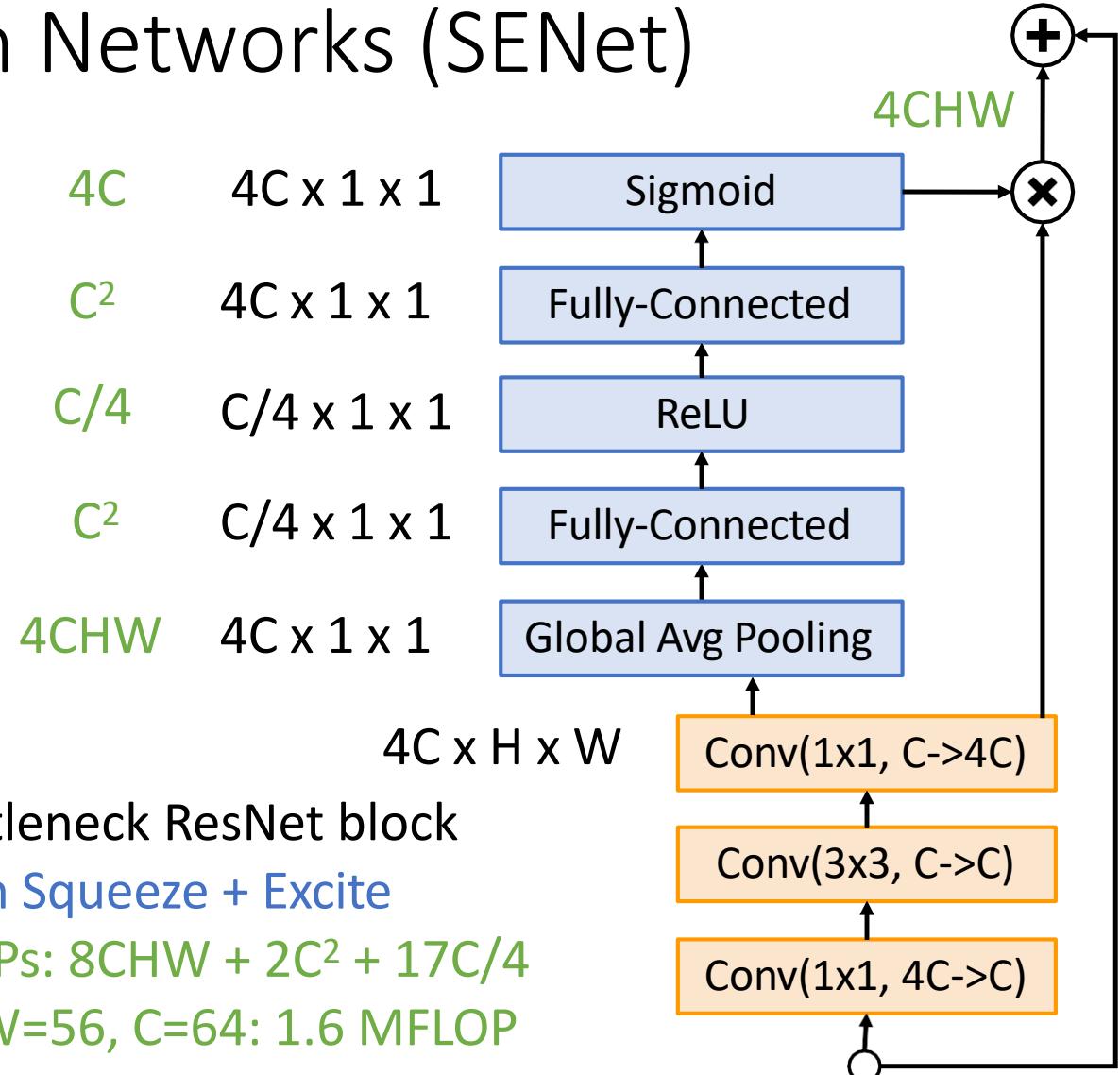


FLOPs:  $17HWC^2$   
 $H=W=56, C=64: 218 \text{ MFLOP}$

Adds **global context** to each ResNet block

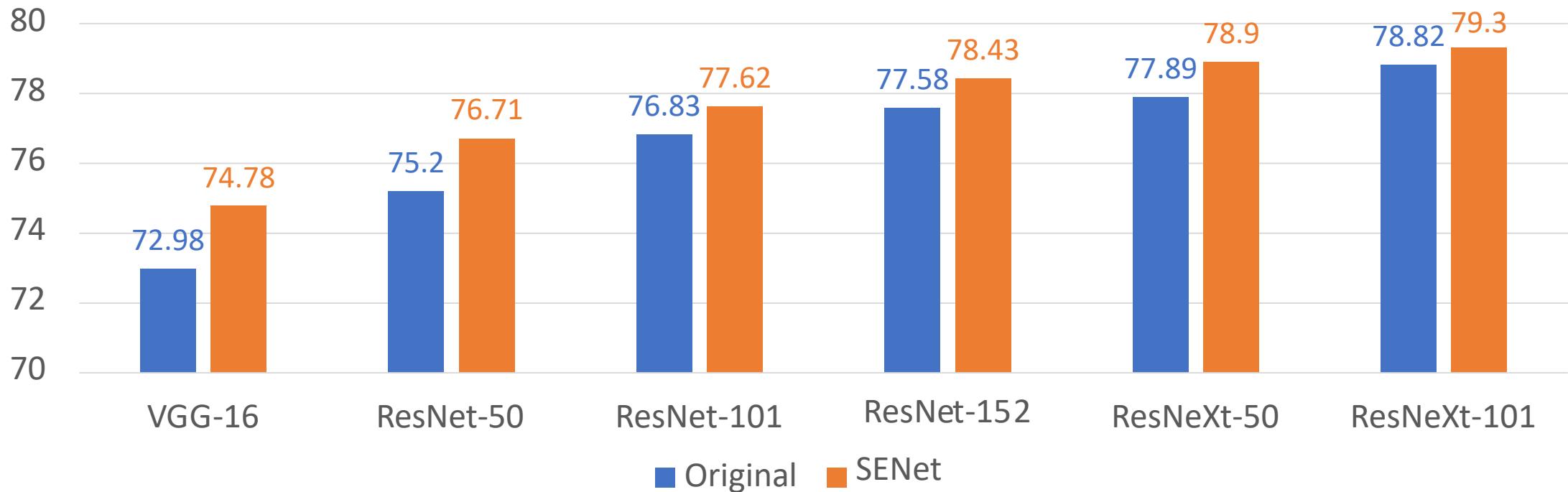
Increases overall FLOPs by < 1%!

Bottleneck ResNet block  
with Squeeze + Excite  
FLOPs:  $8CHW + 2C^2 + 17C/4$   
 $H=W=56, C=64: 1.6 \text{ MFLOP}$



# Squeeze-and-Excitation Networks (SENet)

ImageNet Top-1 Accuracy

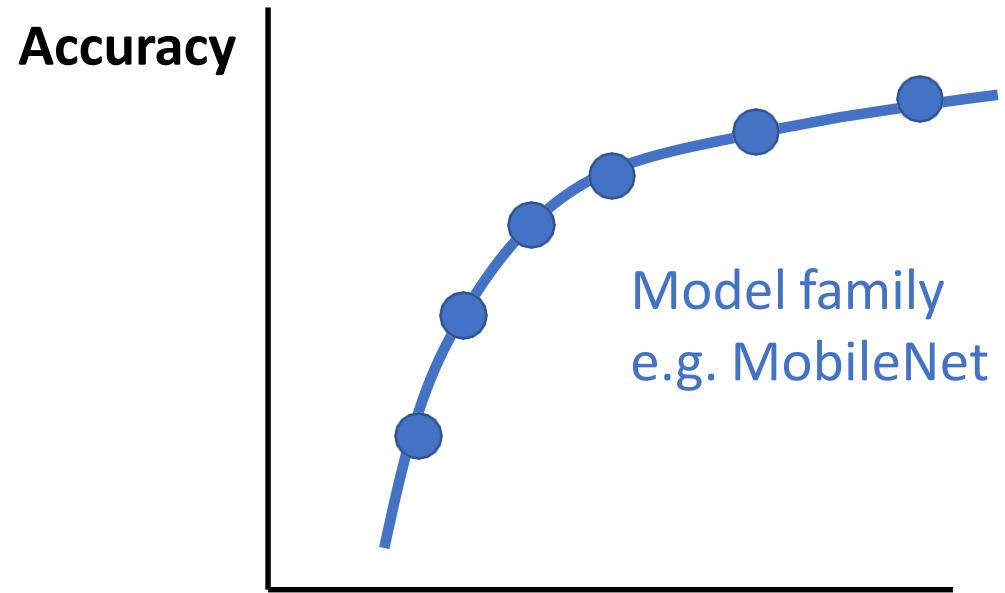


Add SE to any architecture, enjoy 1-2% boost in accuracy

# Tiny Neural Networks for Mobile Devices

Instead of pushing for the largest network with biggest accuracy, consider tiny networks and accuracy / complexity tradeoff

Compare **families of models**:



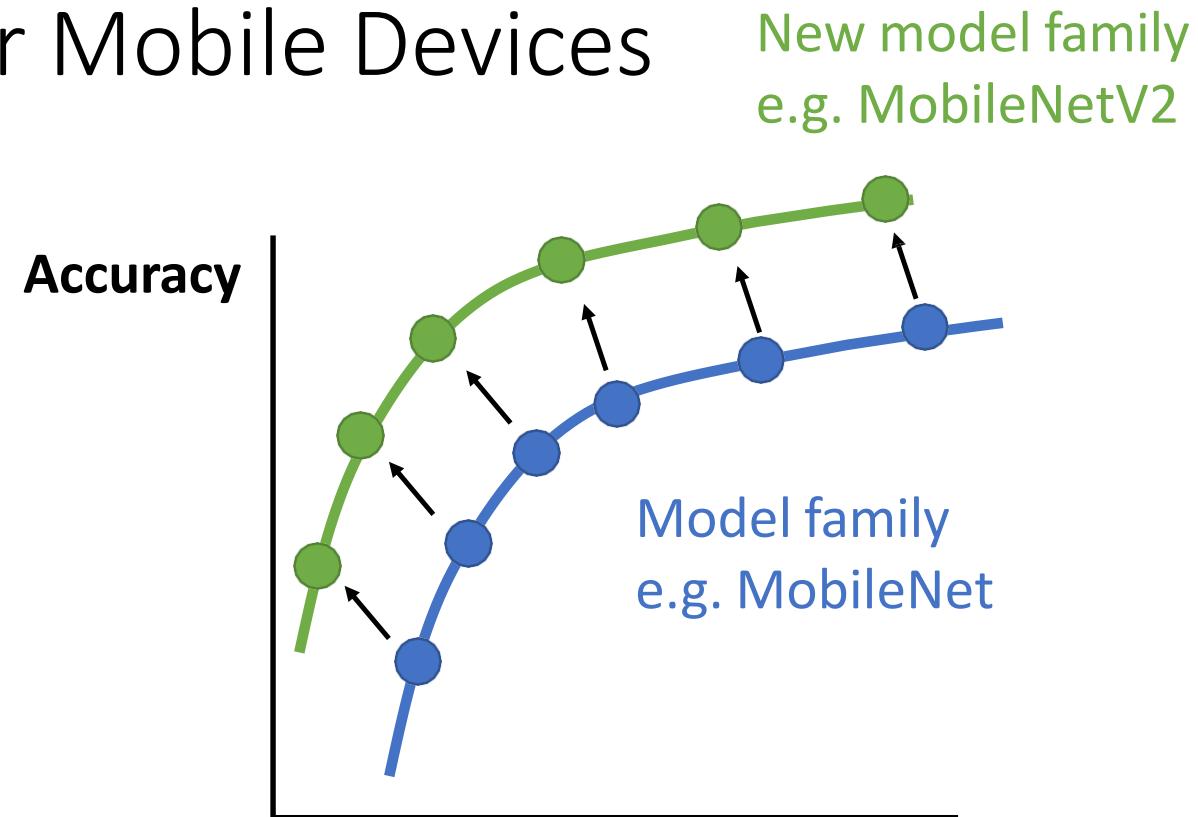
**Model Complexity**  
(FLOPs, #params, runtime speed)

# Tiny Neural Networks for Mobile Devices

Instead of pushing for the largest network with biggest accuracy, consider tiny networks and accuracy / complexity tradeoff

Compare **families of models**:

One family is better than another if it moves the whole curve up and to the left

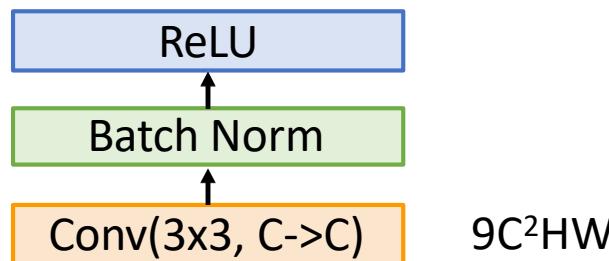


**Model Complexity**  
(FLOPs, #params, runtime speed)

# MobileNets: Tiny Networks (For Mobile Devices)

## Standard Convolution Block

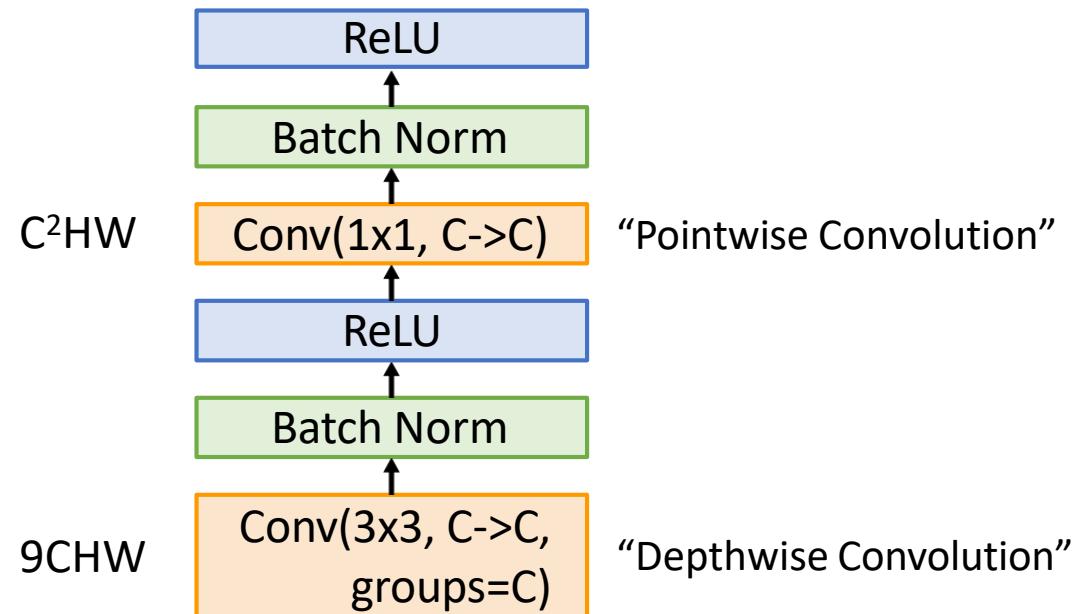
Total cost:  $9C^2HW$



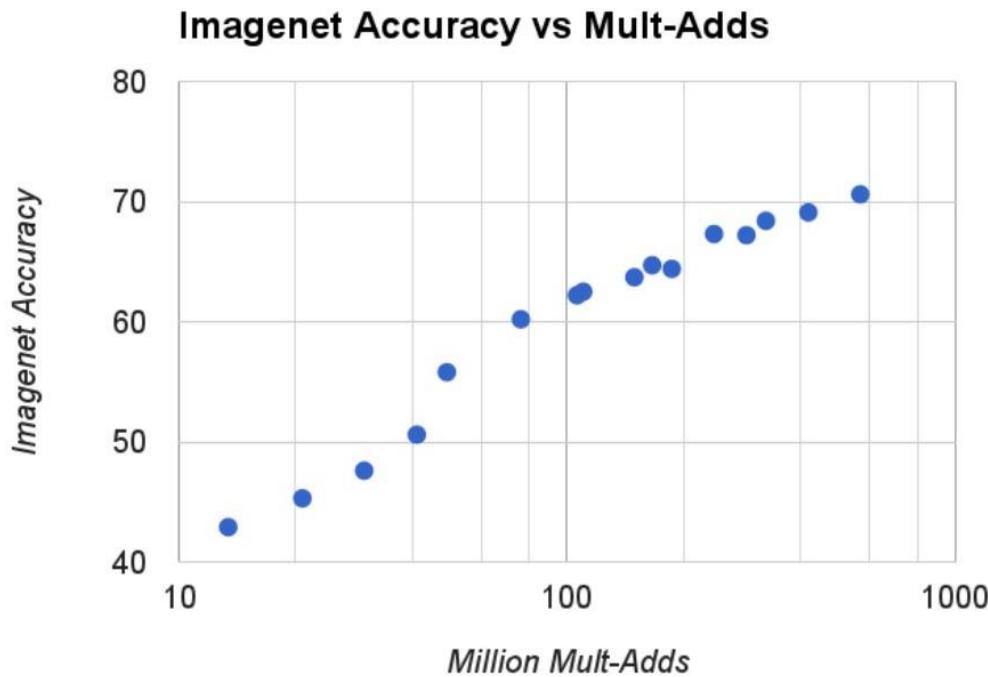
$$\begin{aligned} \text{Speedup} &= 9C^2/(9C+C^2) \\ &= 9C/(9+C) \\ &\Rightarrow 9 \text{ (as } C \rightarrow \infty\text{)} \end{aligned}$$

## Depthwise Separable Convolution

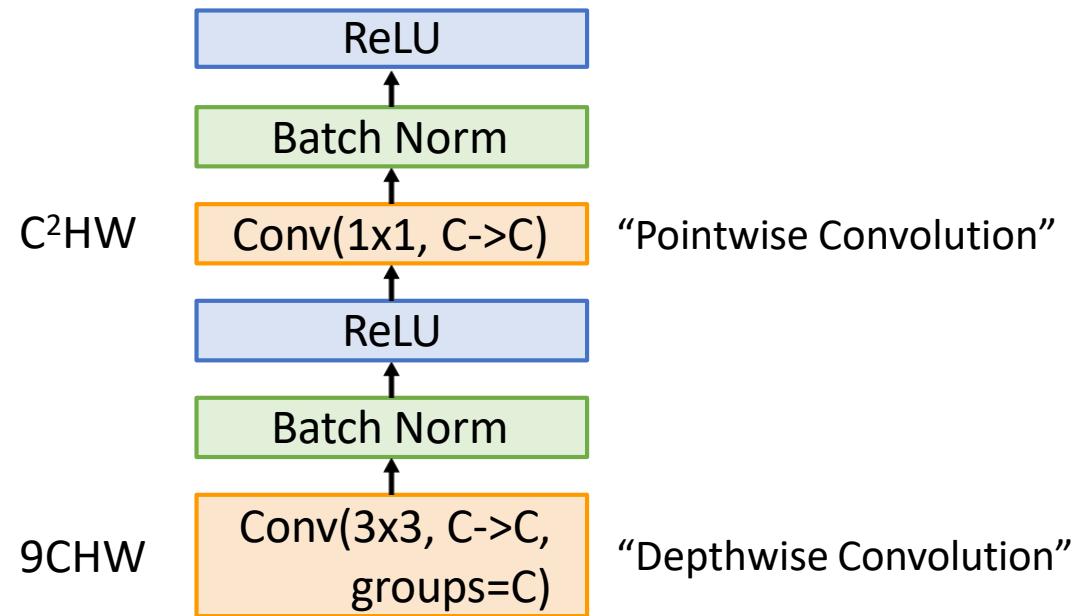
Total cost:  $(9C + C^2)HW$



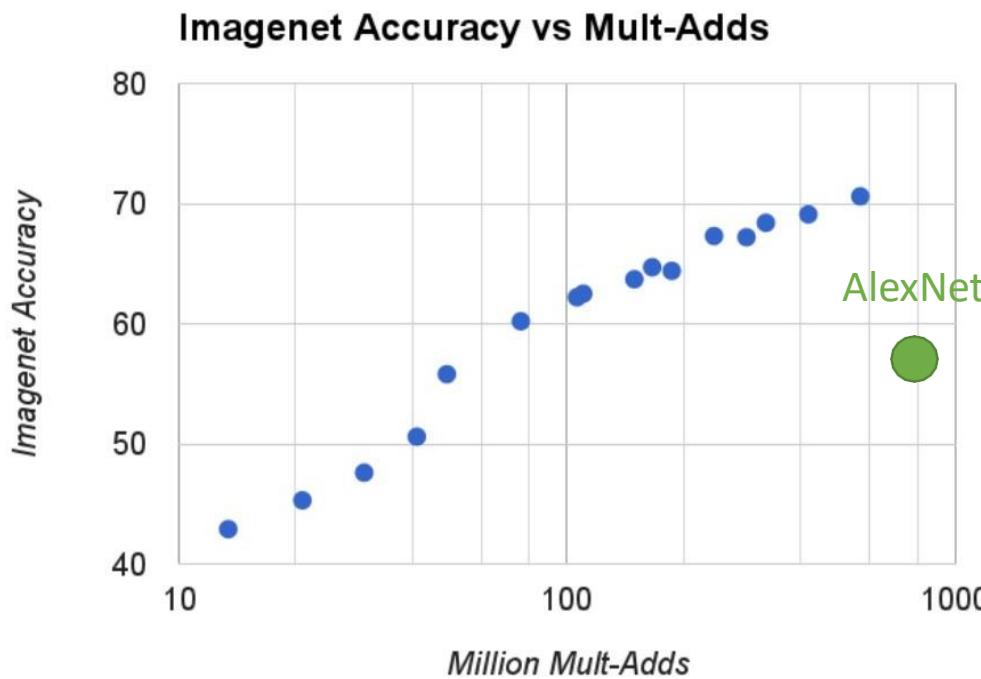
# MobileNets: Tiny Networks (For Mobile Devices)



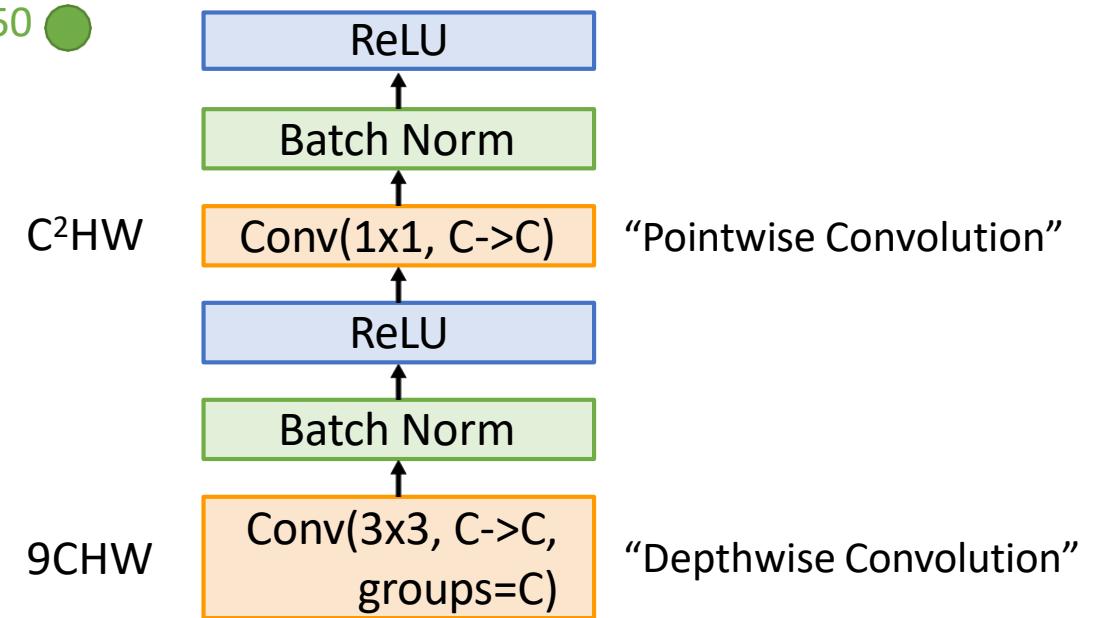
**Depthwise Separable Convolution**  
Total cost:  $(9C + C^2)HW$



# MobileNets: Tiny Networks (For Mobile Devices)

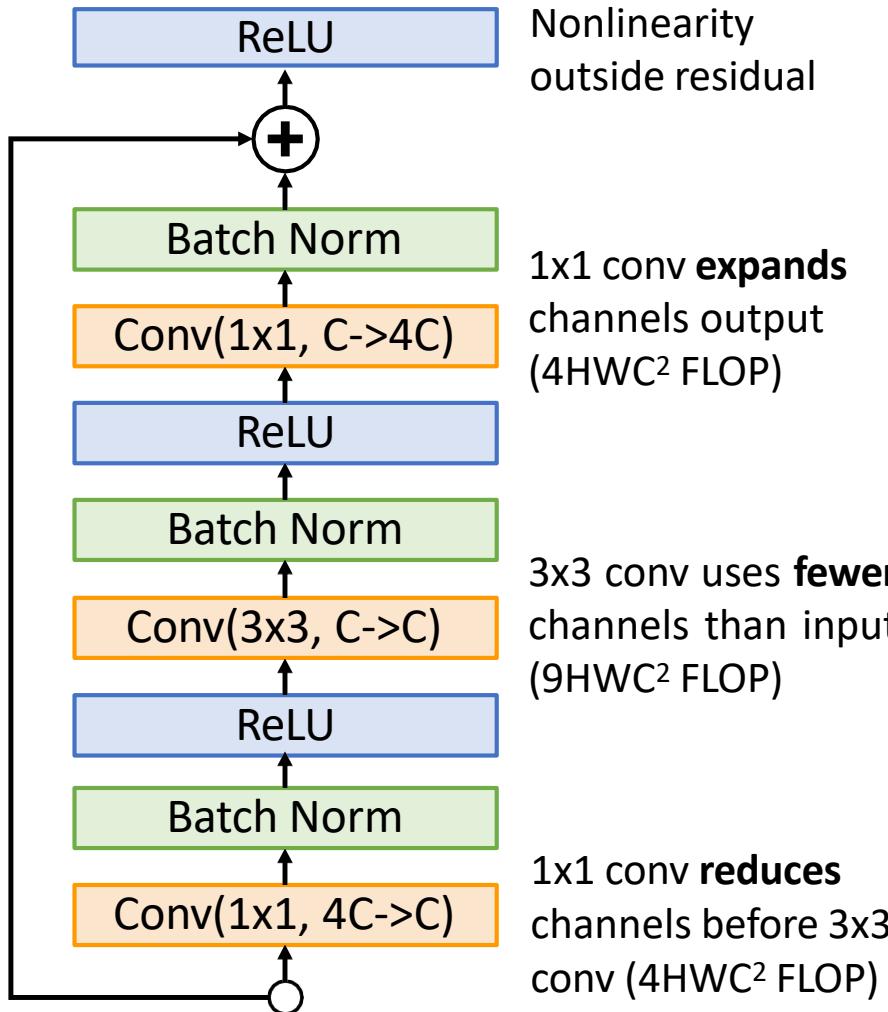


**Depthwise Separable Convolution**  
Total cost:  $(9C + C^2)HW$



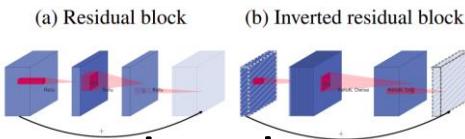
# MobileNetV2: Inverted Bottleneck, Linear Residual

ResNet Bottleneck Block

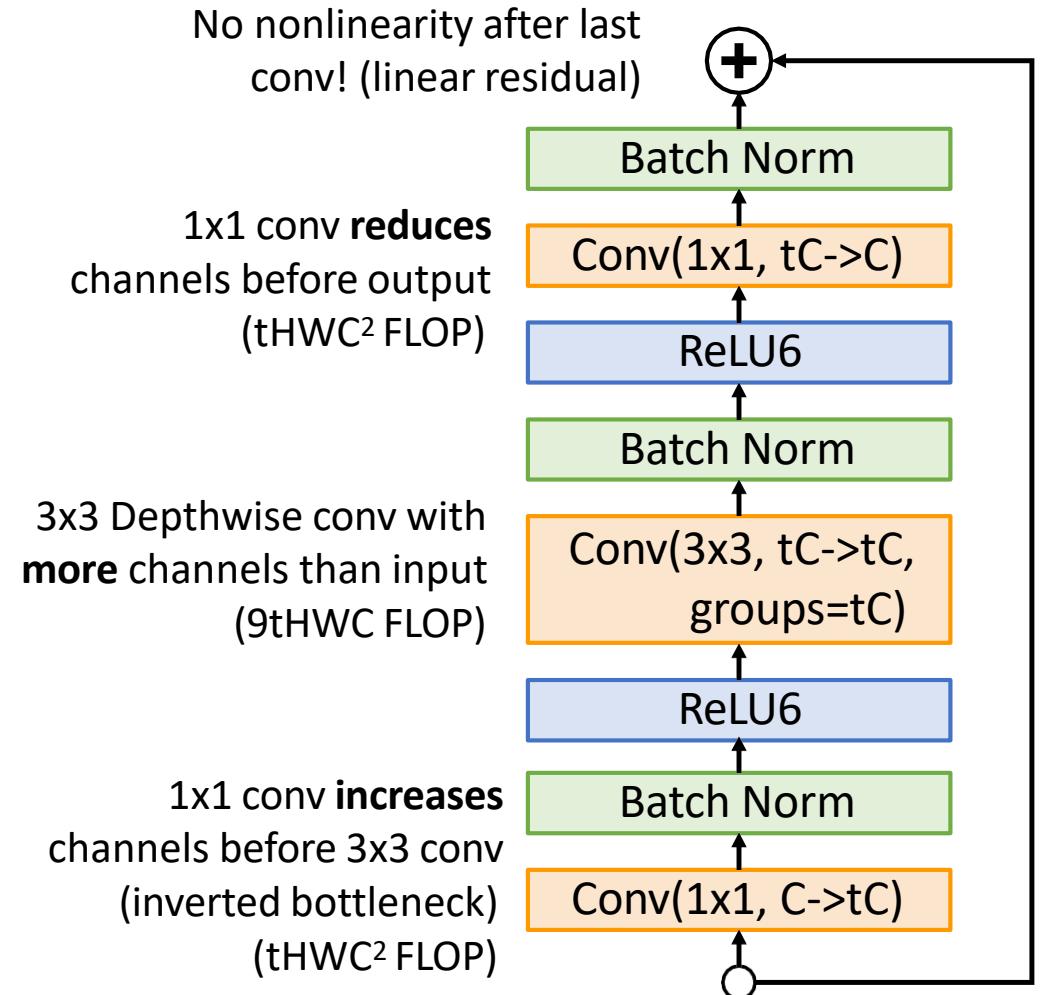
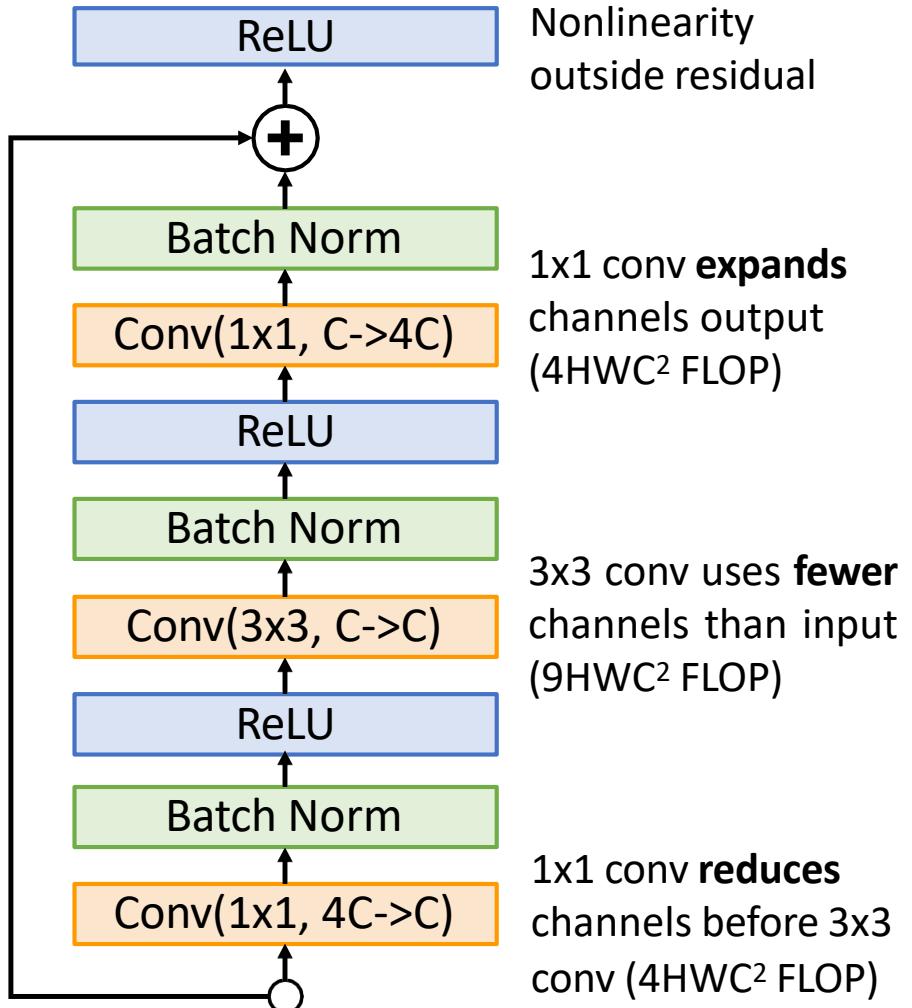


Sandler et al, "MobileNetV2: Inverted Residuals and Linear Bottlenecks", CVPR 2018

# MobileNetV2: Inverted Bottleneck, Linear Residual

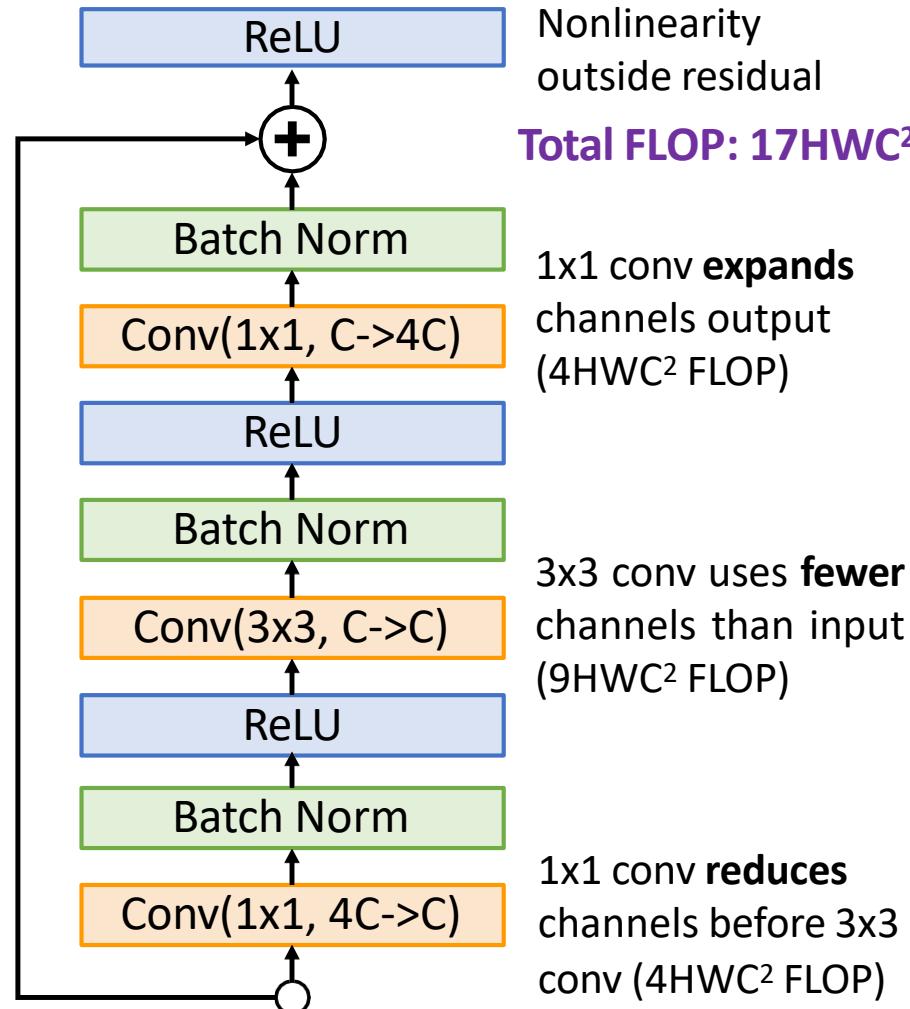


ResNet Bottleneck Block



Sandler et al, "MobileNetV2: Inverted Residuals and Linear Bottlenecks", CVPR 2018

# MobileNetV2: Inverted Bottleneck, Linear Residual



## Total FLOP: $17HWC^2$

## Nonlinearity outside residual

1x1 conv **expands**  
channels output  
( $4HWC^2$  FLOP)

3x3 conv uses **fewer** channels than input  
( $9HWC^2$  FLOP)

**1x1 conv reduces channels before 3x3 conv ( $4HWC^2$  FLOP)**

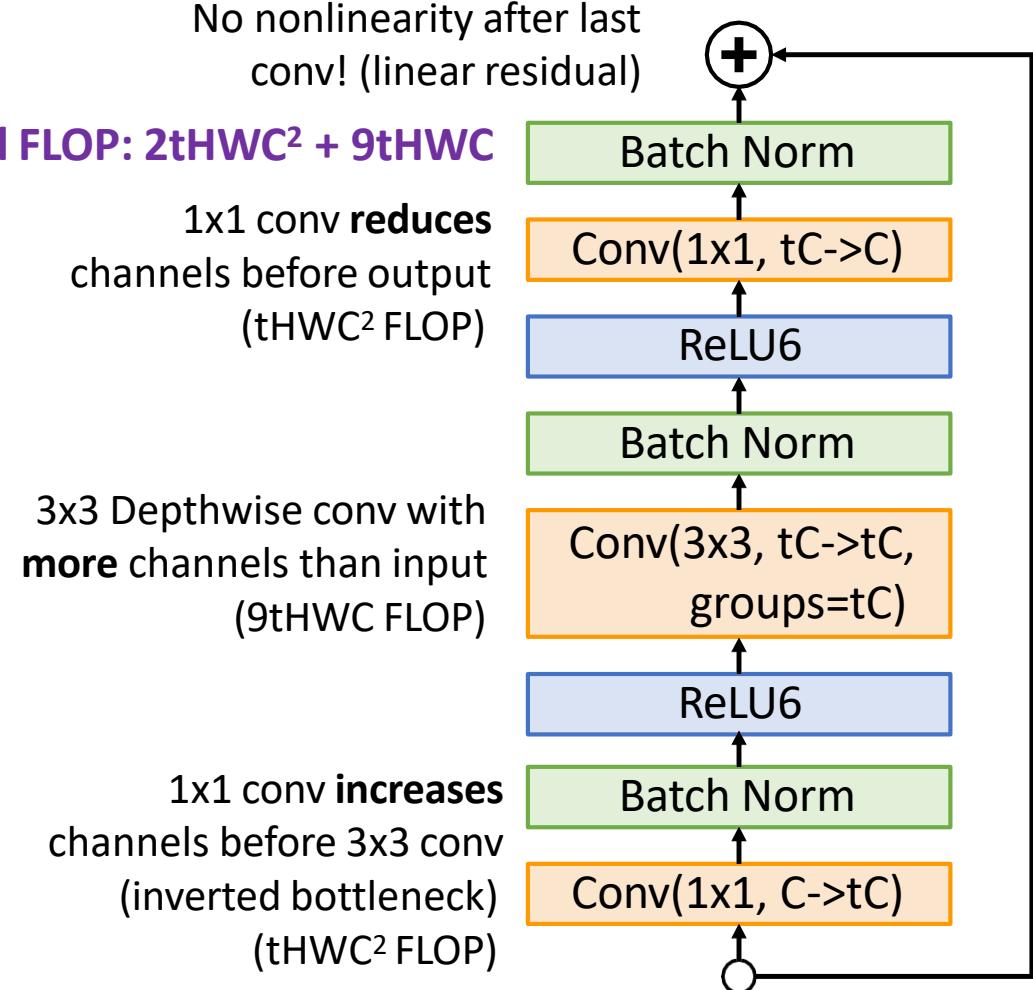
**Total FLOP:  $2tHWC^2 + 9tHWC$**

No nonlinearity after last conv! (linear residual)

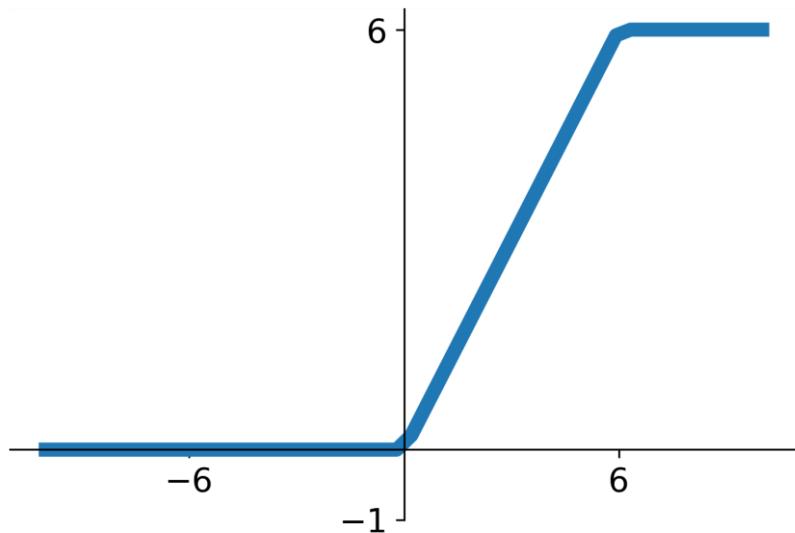
**1x1 conv reduces channels before output  
( $\text{tHWC}^2$  FLOP)**

3x3 Depthwise conv with  
**more** channels than input  
(9tHWC FLOP)

1x1 conv **increases**  
channels before 3x3 conv  
(inverted bottleneck)  
(tHWC<sup>2</sup> FLOP)

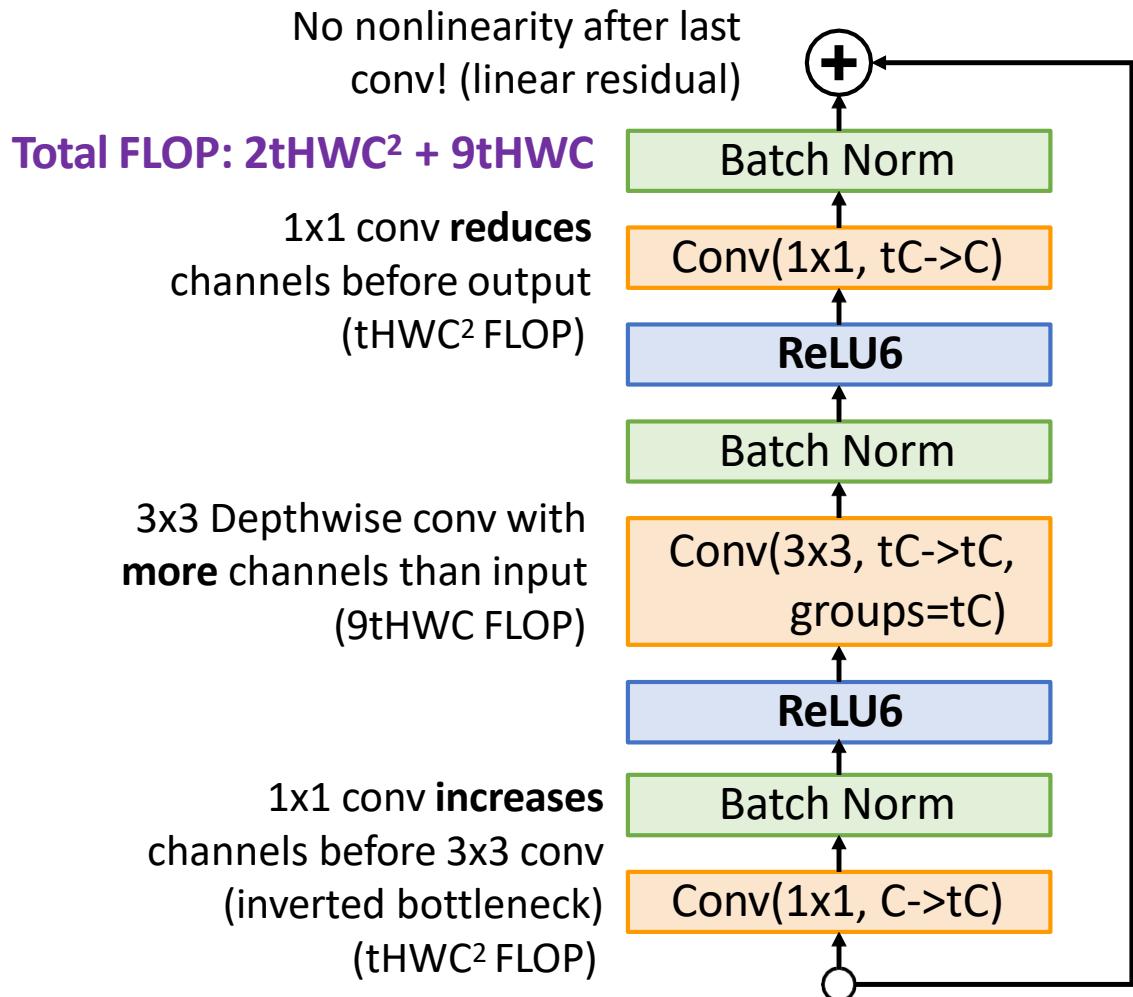


# MobileNetV2: Inverted Bottleneck, Linear Residual



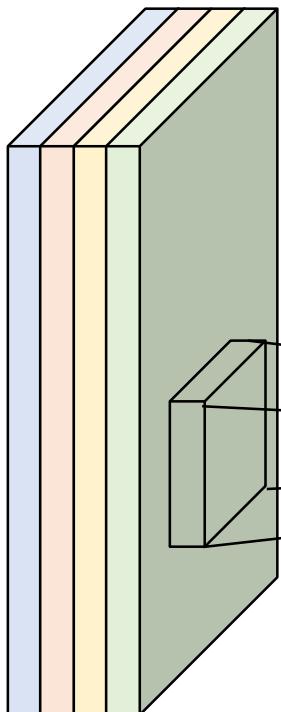
$$ReLU6(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } 0 < x < 6 \\ 6 & \text{if } x \geq 6 \end{cases}$$

Keeps activations in reasonable range  
when running inference in low precision

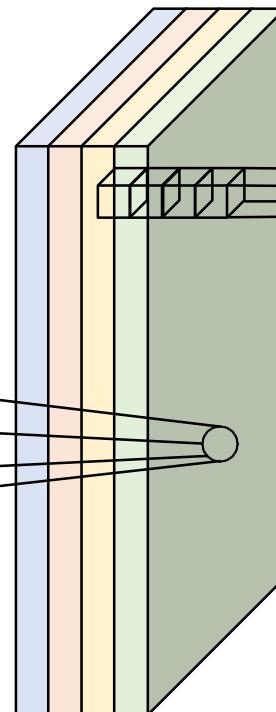


# MobileNetV2: Inverted Bottleneck, Linear Residual

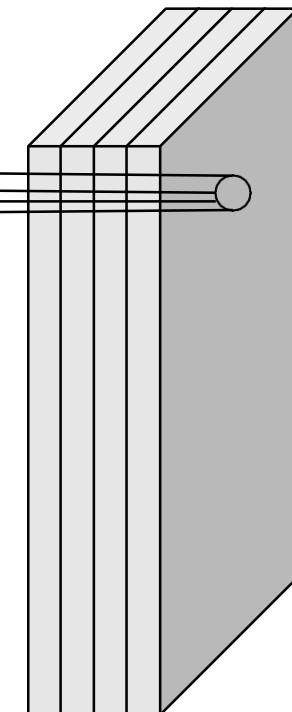
3x3 Depthwise Convolution:  
Mixes data across space,  
Keeps data across channels separate



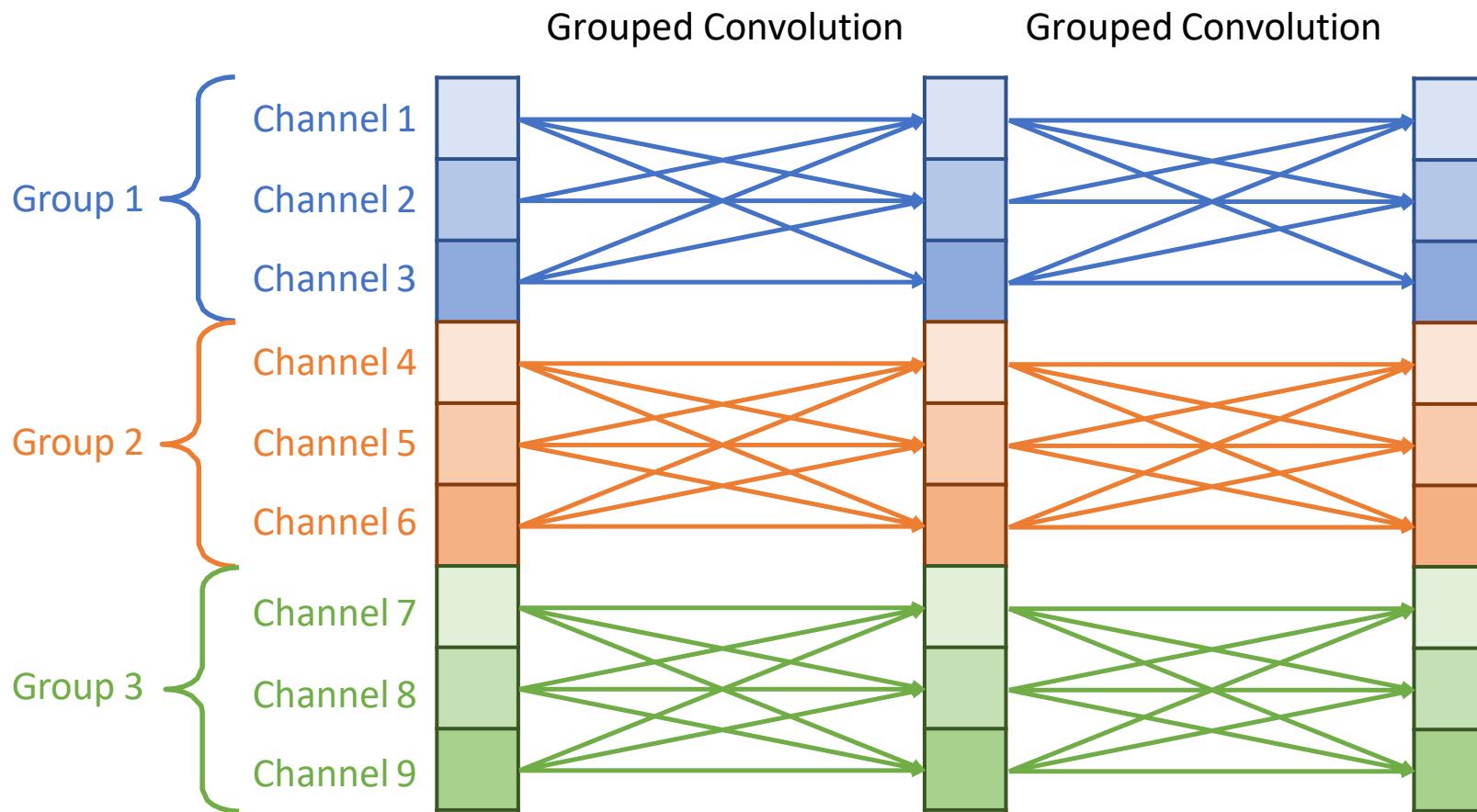
1x1 Convolution:  
Keeps data across space separate,  
Mixes data across channels



**Idea:** Can we mix  
channel info  
more efficiently  
than 1x1 conv?



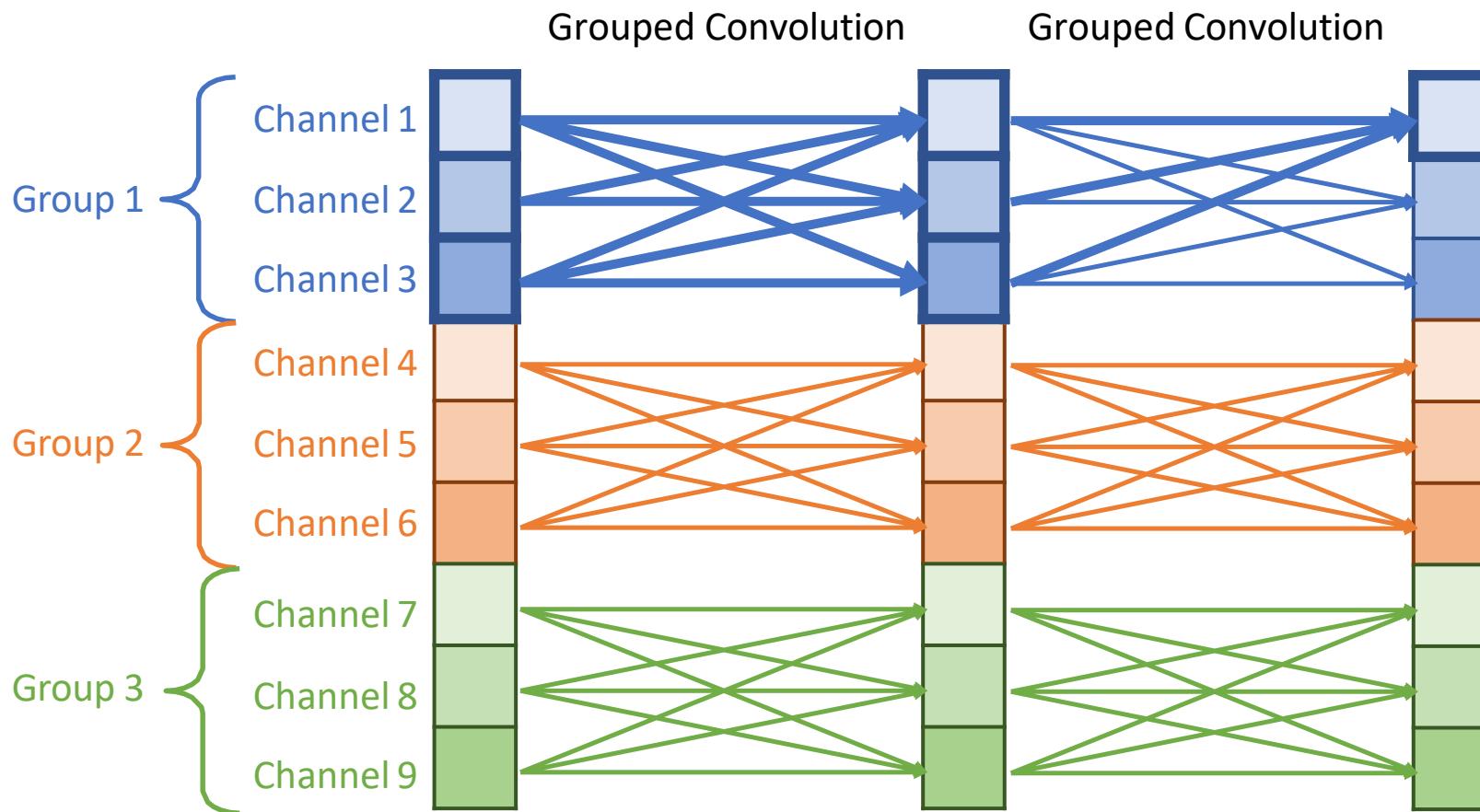
# Stacking Grouped Convolutions



Zhang et al, "ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices", CVPR 2018

**Problem:** Information is never mixed across channels from different groups!

# Stacking Grouped Convolutions

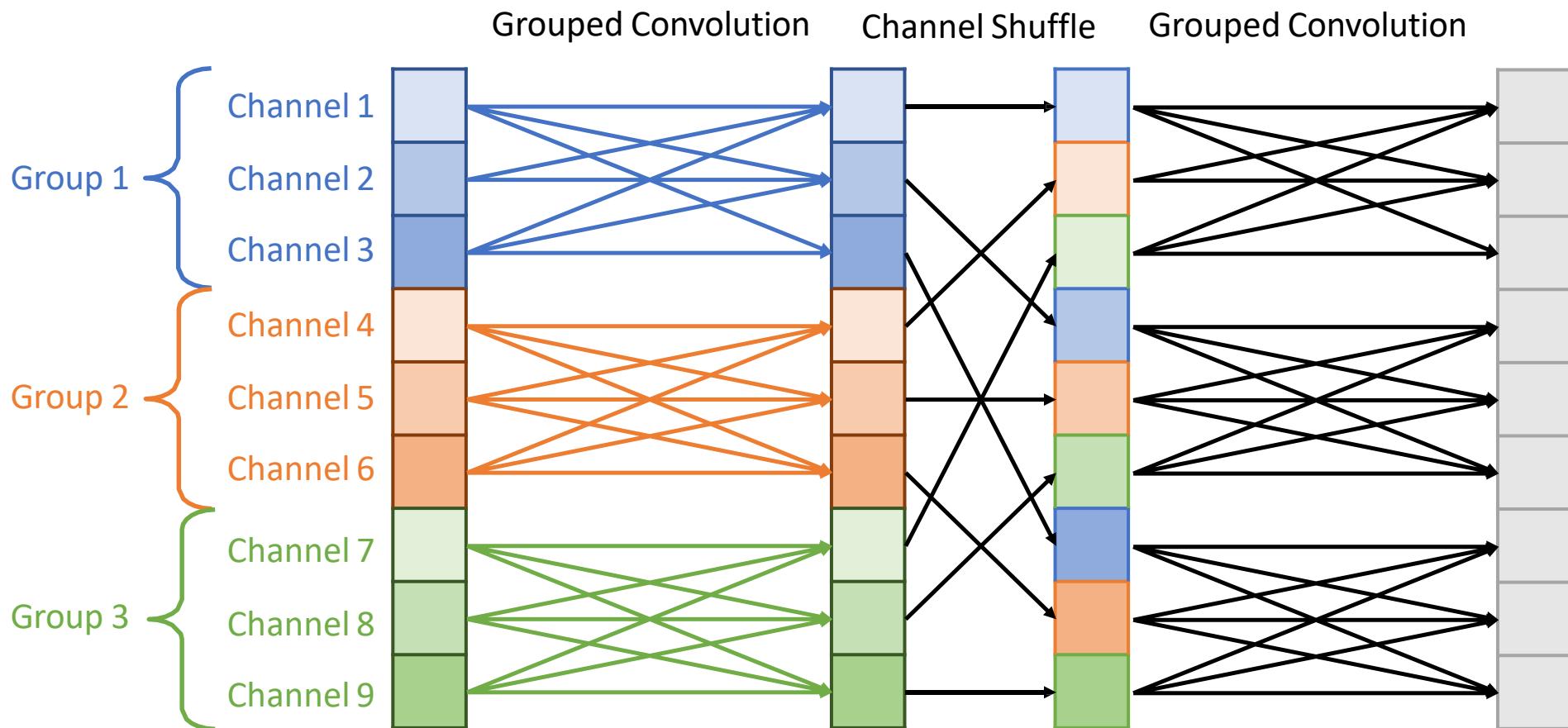


Zhang et al, "ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices", CVPR 2018

**Problem:** Information is never mixed across channels from different groups!

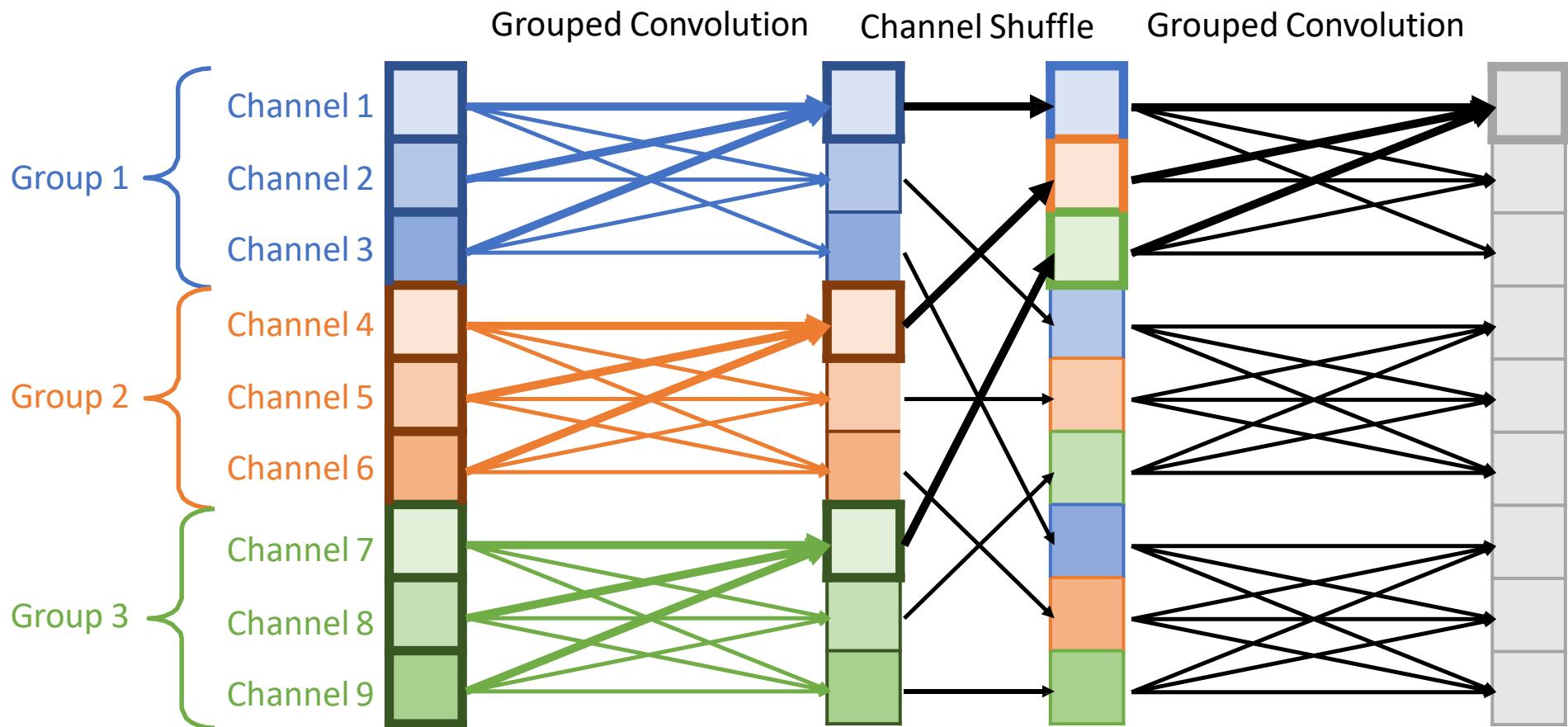
# Stacking Grouped Convolutions

Insert “Channel Shuffle” operators that permute channels between convolutions



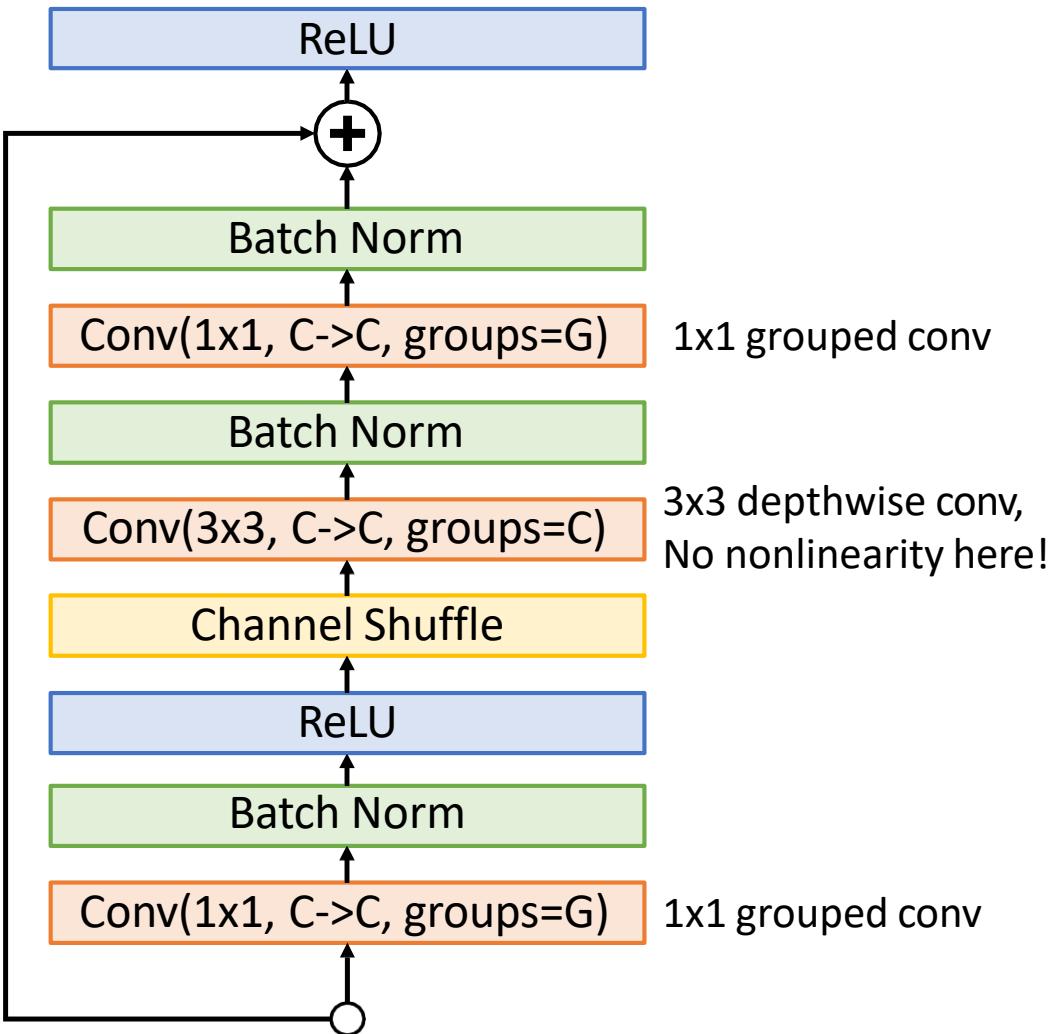
# Stacking Grouped Convolutions

Insert “Channel Shuffle” operators that permute channels between convolutions



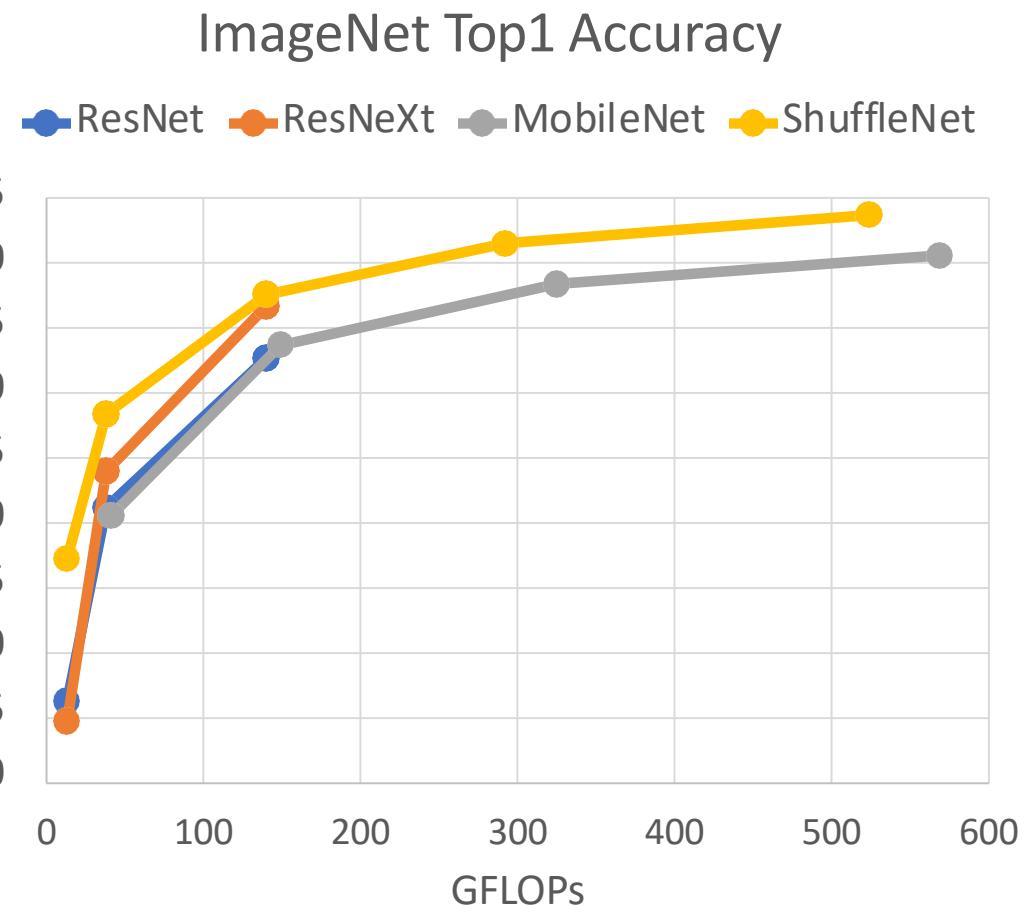
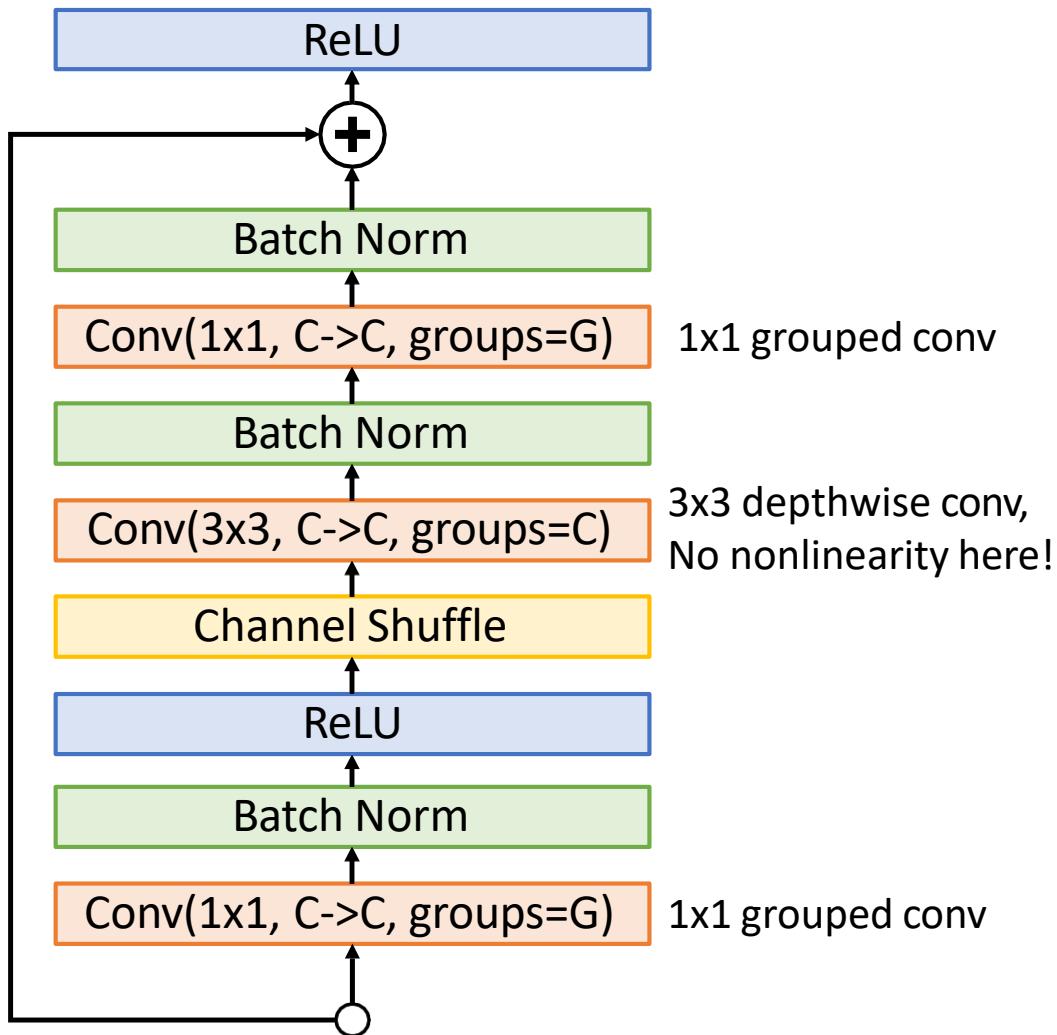
Now channel information is fully “mixed” after two grouped convolutions – no need for any ungrouped convolutions!

# ShuffleNet



Zhang et al, "ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices", CVPR 2018

# ShuffleNet

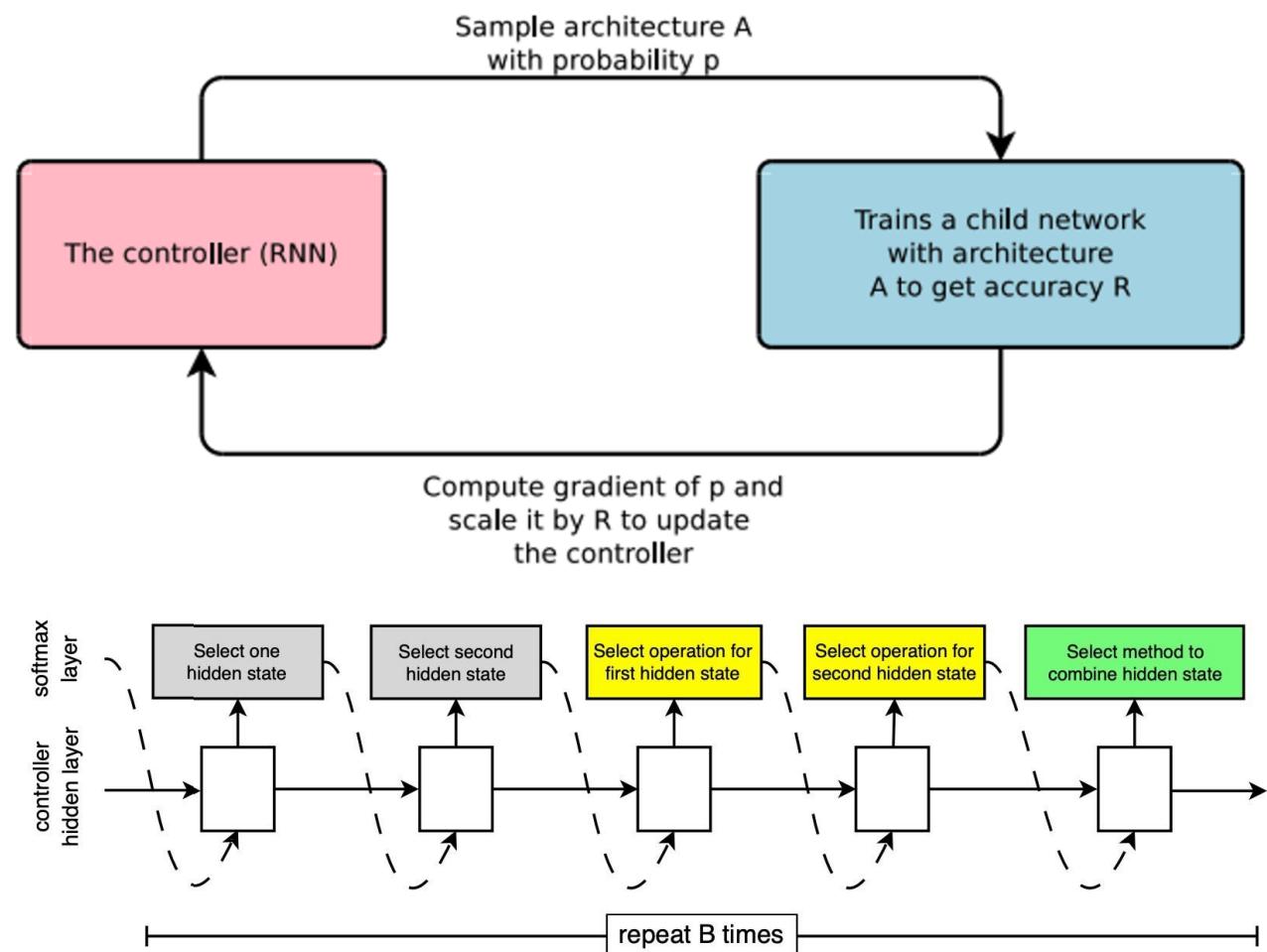


Zhang et al, "ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices", CVPR 2018

# 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

# Neural Architecture Search (NAS)

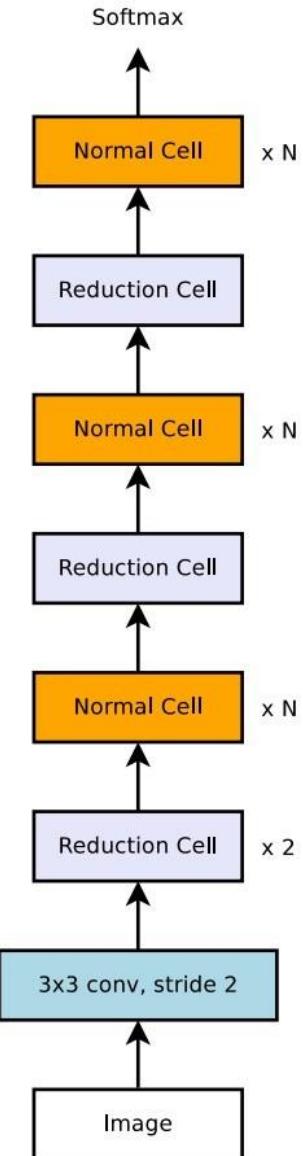
Search for reusable “block” designs which can use the following operators:

- 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

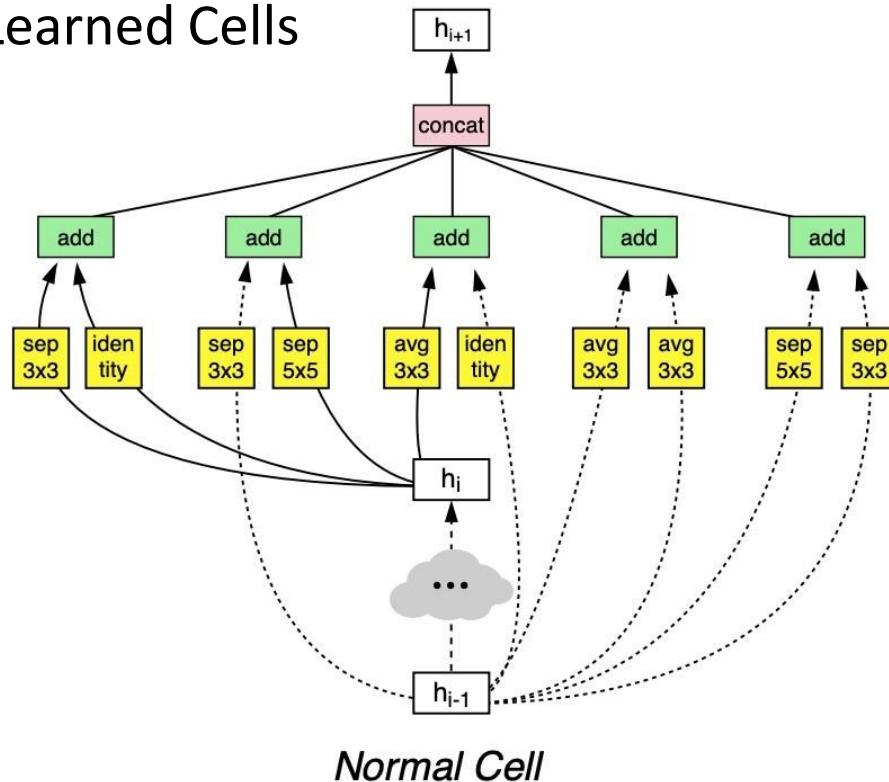
The “Reduction cell” reduces image resolution by 2x

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

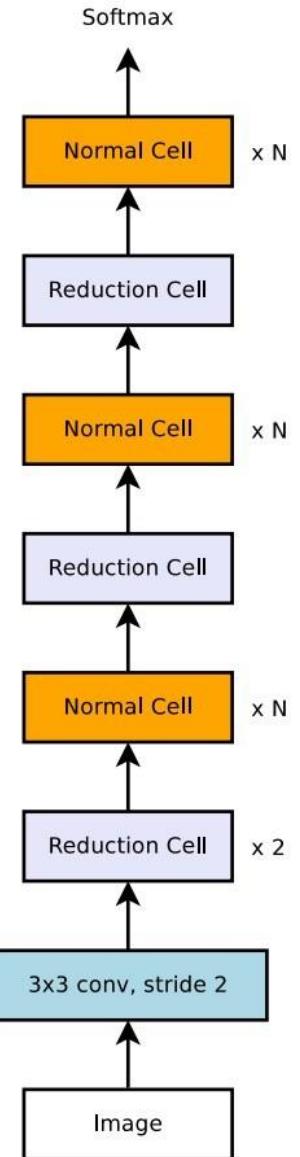


# Neural Architecture Search (NAS)

Learned Cells

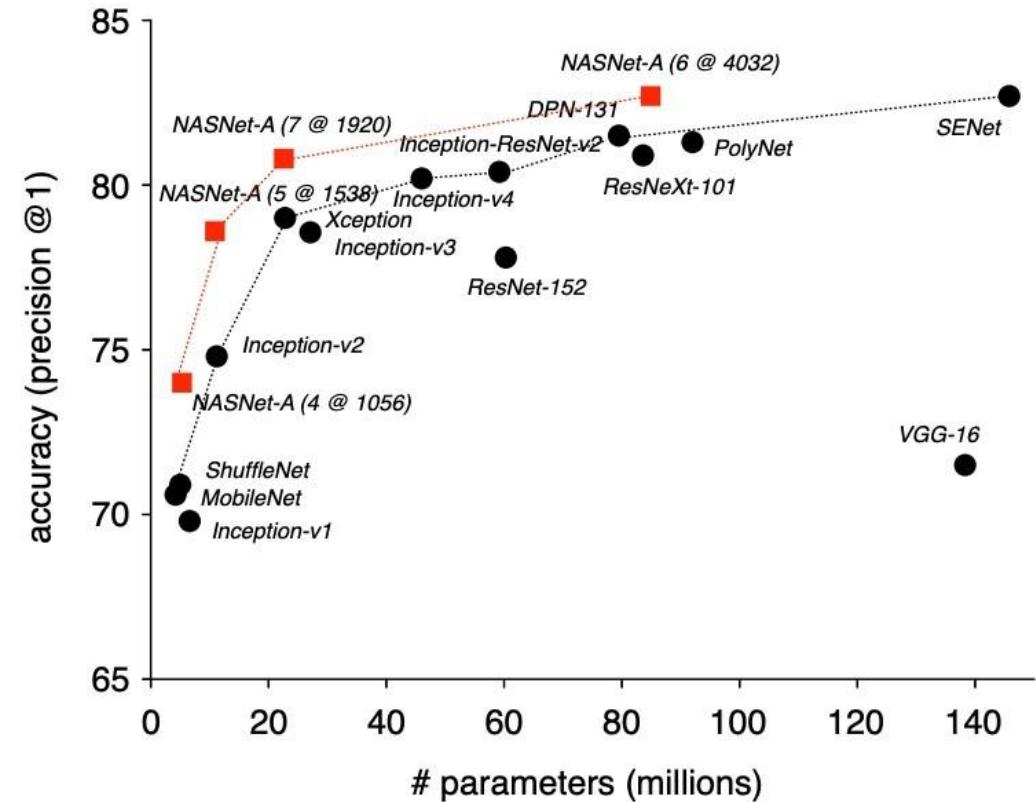
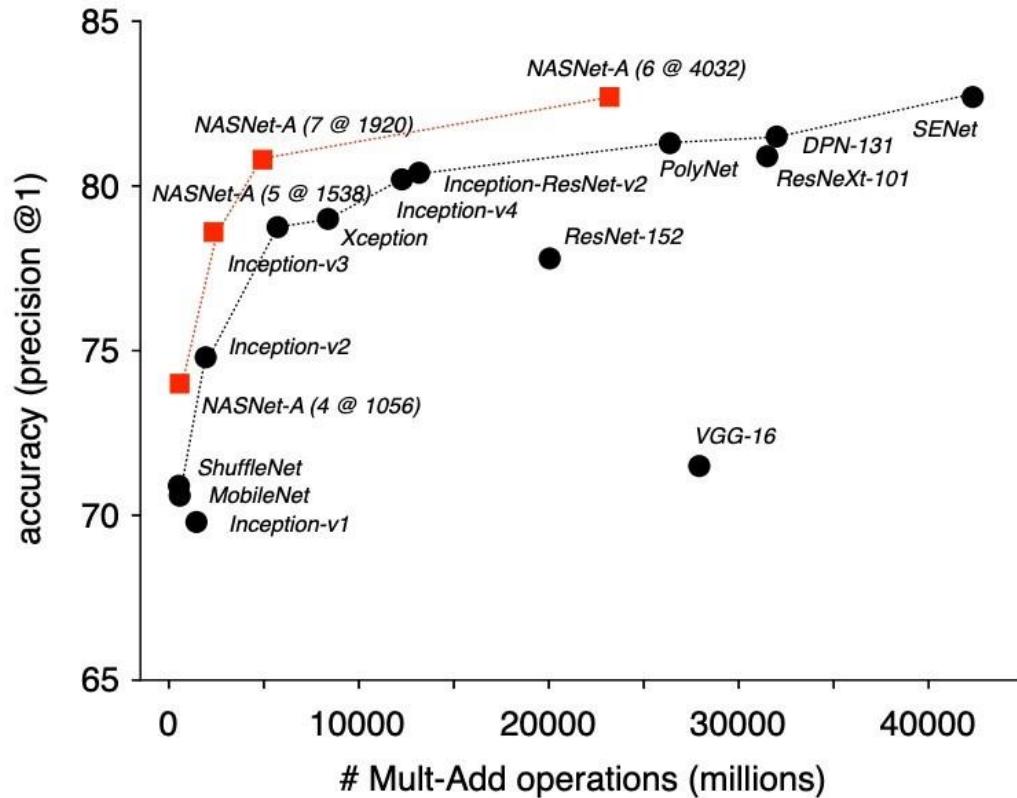


**Reduction Cell**



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

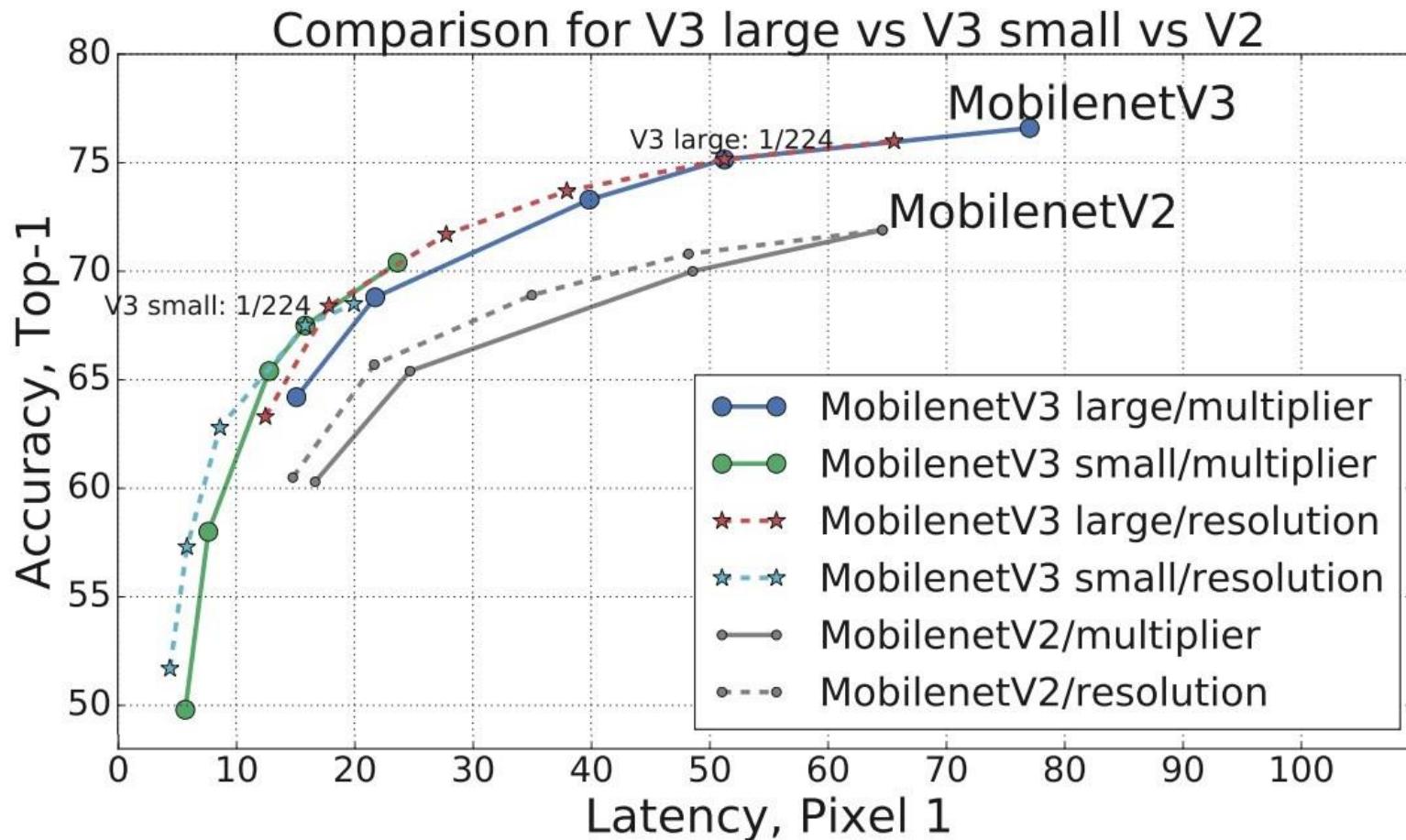
# Neural Architecture Search (NAS)



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

Zoph et al, "Learning transferable architectures for scalable image recognition", CVPR 2018

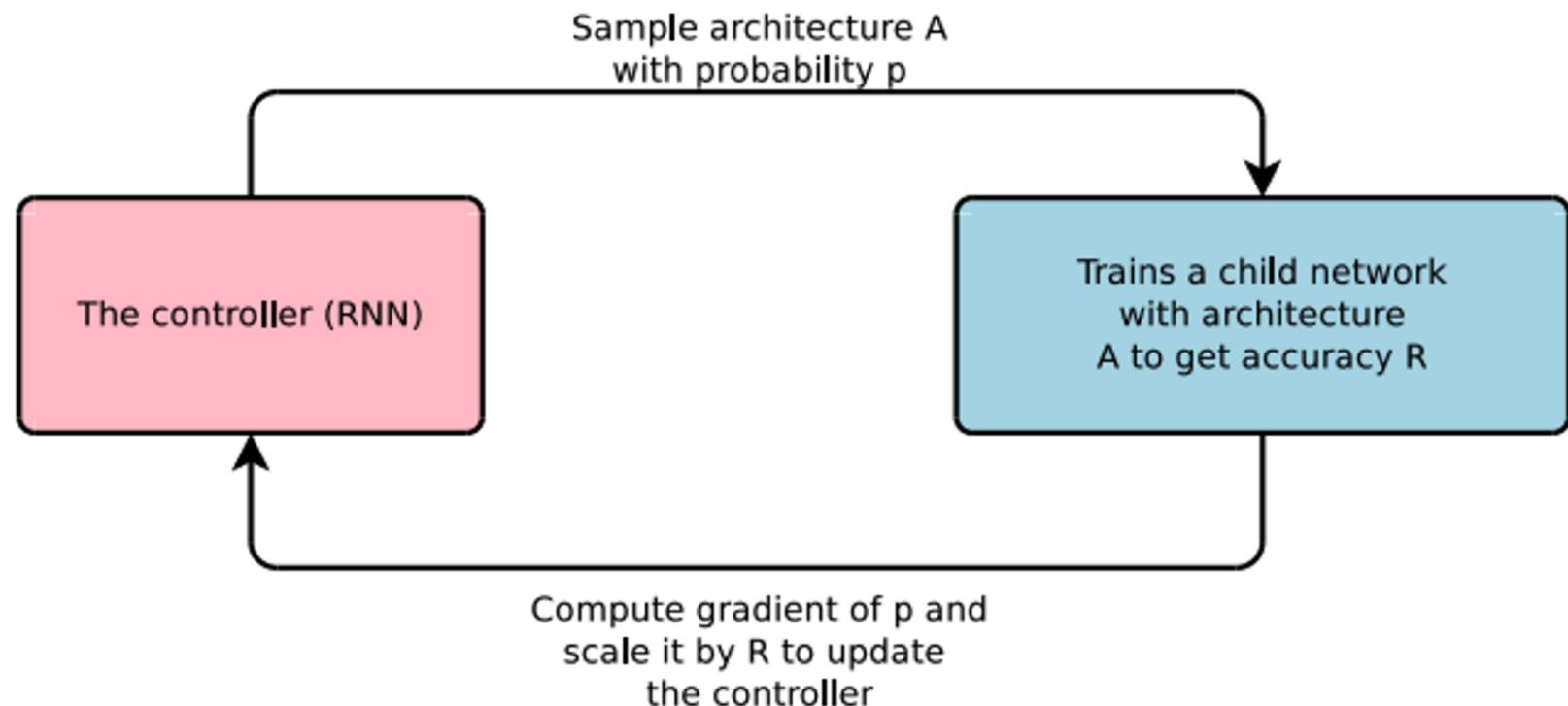
# NAS for MobileNetV3



# 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



# Neural Architecture Search: Many followups

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

Pham et al, "Efficient neural architecture search via parameter sharing", ICML 2018

Brock et al, "SMASH: One-Shot Model Architecture Search through HyperNetworks", ICLR 2018

Ramachandran et al, "Searching for Activation Functions", ICLR 2018 Workshop

Zoph et al, "Learning transferable architectures for scalable image recognition", CVPR 2018

Liu et al, "Progressive Neural Architecture Search", CVPR 2018

Liu et al, "DARTS: differentiable Architecture Search", ICLR 2019

Cai et al, "ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware", ICLR 2019

Xie et al, "SNAS: Stochastic Neural Architecture Search", ICLR 2019

Real et al, "Regularized evolution for image classifier architecture search", AAAI 2019

Tan et al, "MnasNet: Platform-Aware Neural Architecture Search for Mobile", CVPR 2019

Howard et al, "Searching for MobileNetV3", CVPR 2019

Wu et al, "FBNet: Hardware-Aware Efficient ConvNet Design via Differentiable Neural Architecture Search", CVPR 2019

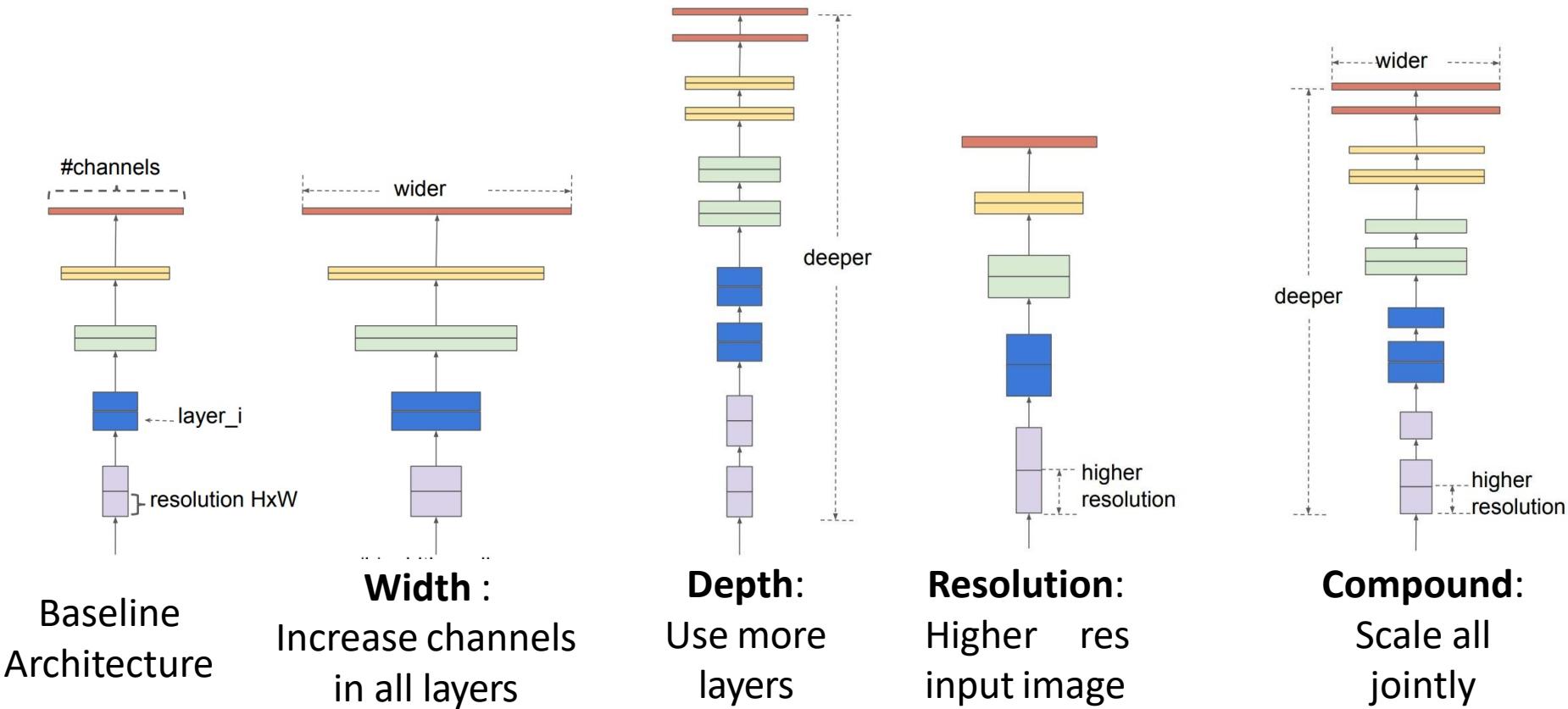
Liu et al, "Auto-DeepLab: Hierarchical Neural Architecture Search for Semantic Image Segmentation", CVPR 2019

Ghiasi et al, "NAS-FPN: Learning Scalable Feature Pyramid Architecture for Object Detection", CVPR 2019

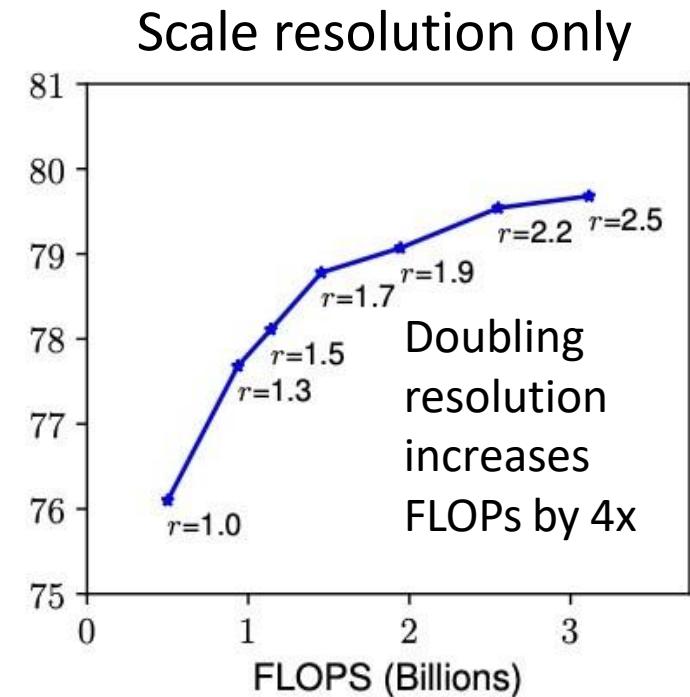
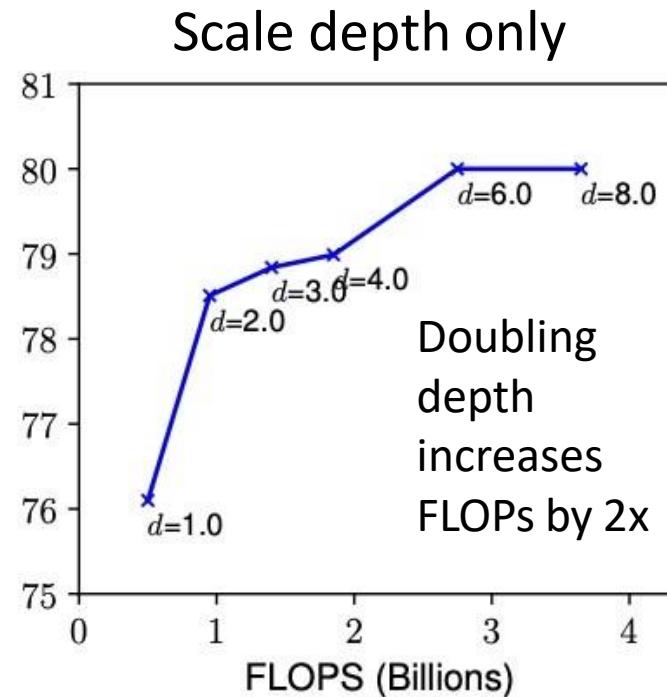
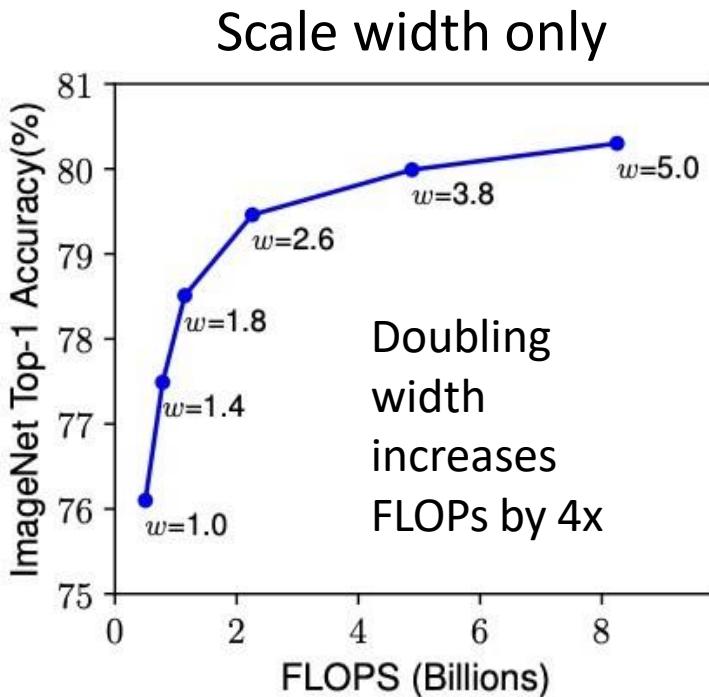
Cubuk et al, "AutoAugment: Learning Augmentation Strategies from Data", CVPR 2019

# Model Scaling

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



# Model Scaling: EfficientNets

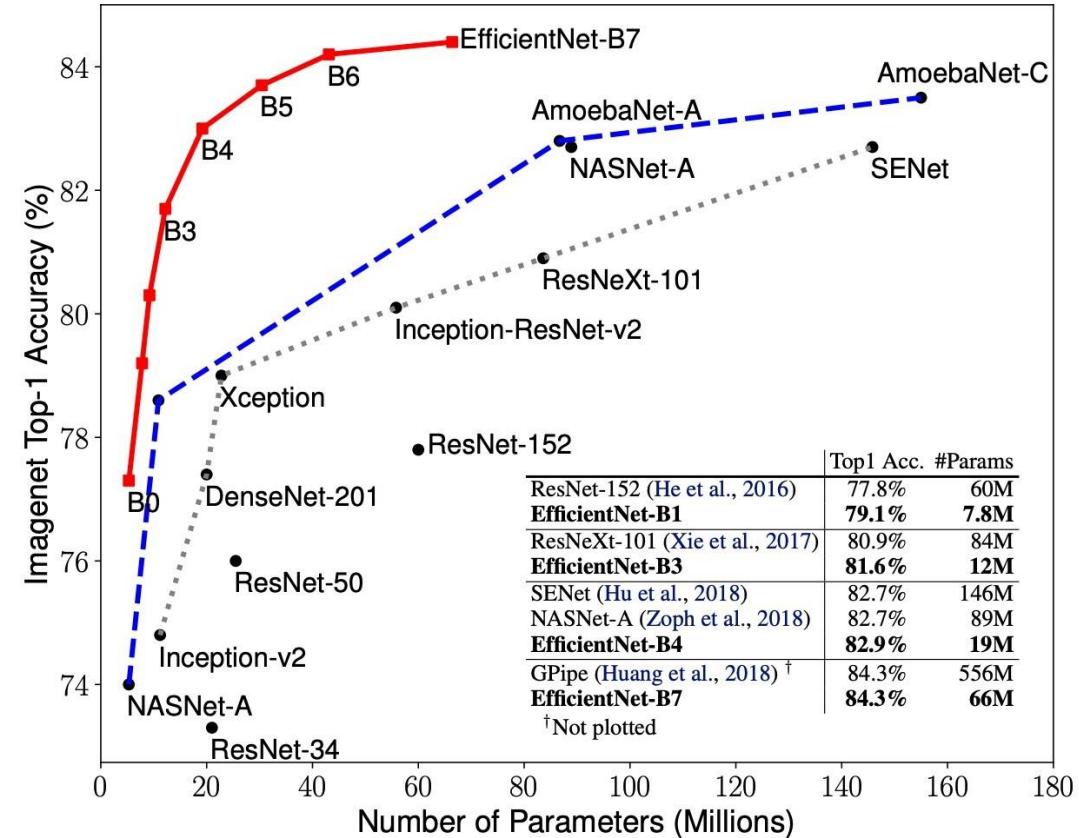
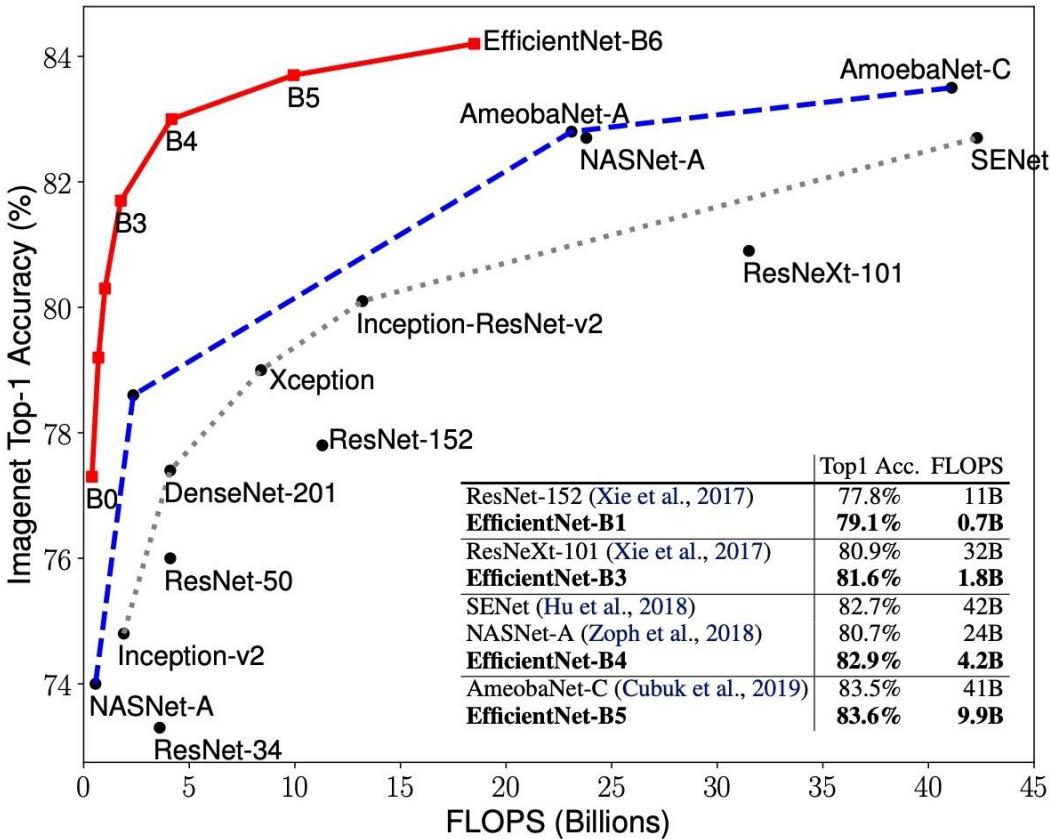


Scaling any of width, depth, or resolution has diminishing returns.  
For optimal results, need to scale them all jointly!

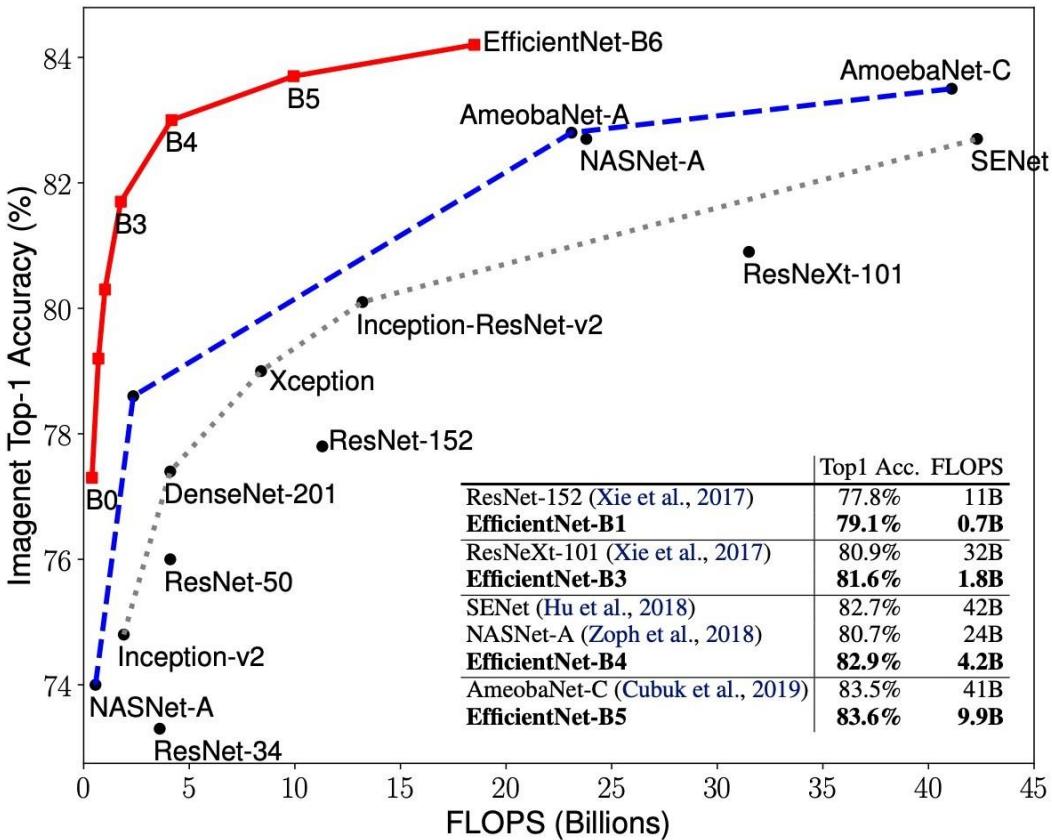
# Model Scaling: EfficientNets

1. Use NAS to get initial EfficientNet-B0 architecture;  
uses depthwise conv, inverted bottlenecks, and SE
2. Find optimal scaling factors  $\alpha$  for depth,  $\beta$  for width,  $\gamma$  for resolution with  $\alpha, \beta, \gamma \geq 1$  and  $\alpha\beta^2\gamma^2 \approx 2$  via grid search on scaling up initial architecture;  
found  $\alpha = 1.2, \beta = 1.1, \gamma = 1.15$
3. Scale initial architecture to arbitrary FLOPs: scaling by  $\alpha'', \beta'', \gamma''$  will increase FLOPs by a factor of  $\approx 2''$

# Model Scaling: EfficientNets



# Model Scaling: EfficientNets



**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!

# Beyond NAS – back to hand-designed models!

Work in the last ~year has started to turn away from NAS;  
instead smartly tweak ResNet-style models to improve performance,  
scaling, runtime on GPU / TPU

**NFNets:** Remove Batch Normalization

**ResNet-RS:** Modern ResNet training recipe, better scaling

**RegNets:** Simple block design, optimize macro architecture and scaling

# Training ResNets without Batch Normalization

- Batch Normalization has good properties:
  - Makes it easy to train deep networks  $\geq 10$  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

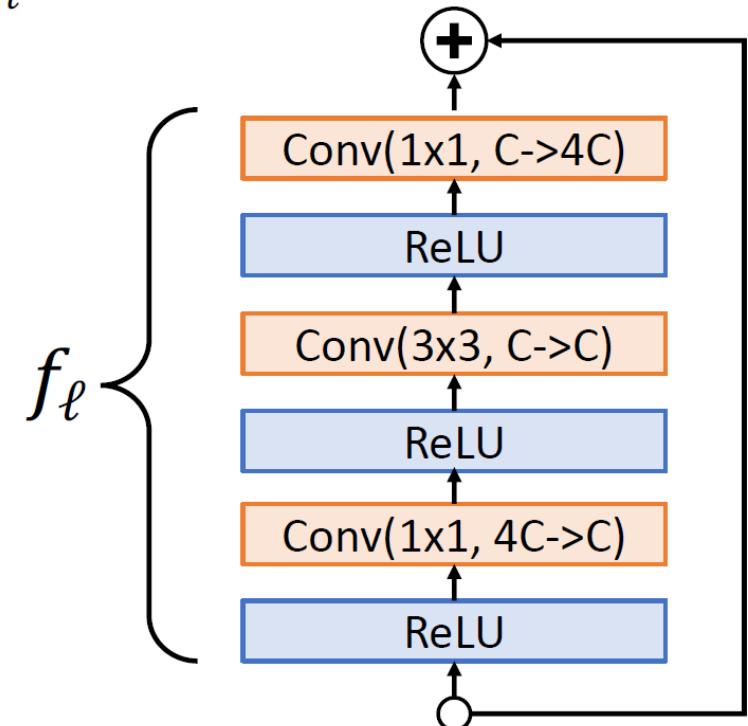
NFNets are ResNets without Batch Normalization!

# NFNets

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

Problem: Variance grows with each block:

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



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

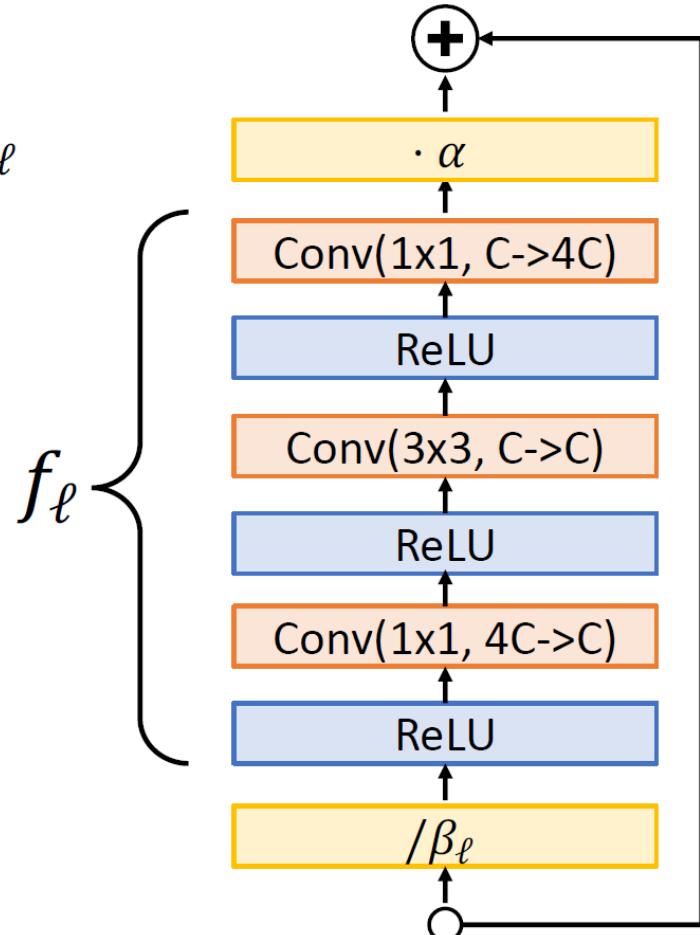
Problem: Variance grows with each block:

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

Solution: Re-parameterize block:

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

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



# 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:

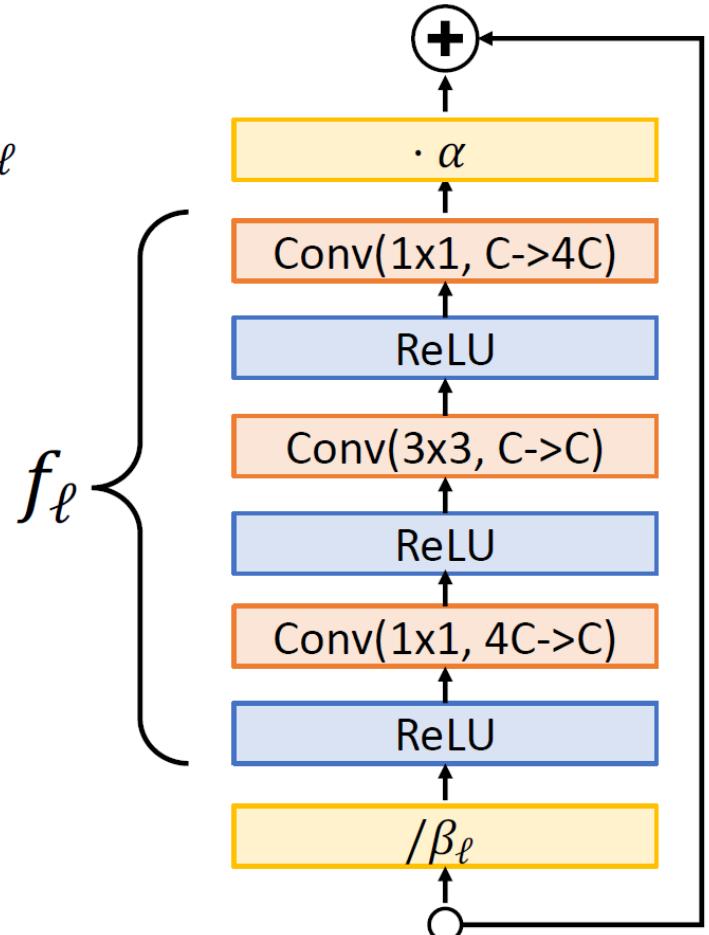
$$\text{Var}(x_{\ell+1}) = \text{Var}(x_\ell) + \text{Var}(f_\ell(x_\ell))$$

Solution: Re-parameterize block:

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

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

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



# NFNets: Weight Standardization

Rather than normalizing *activations* during training,  
instead normalize *weights*!

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: Weight Standardization

Rather than normalizing *activations* during training,  
instead normalize *weights*!

Learn weights  $W$  but convolve with weights  $\hat{W}$  where

$$\hat{W}_{i,j} = \gamma \cdot \frac{W_{i,j} - \text{mean}(W_i)}{\text{std}(W_i)\sqrt{N}}$$

$W_i$  is a single conv filter,  $N = K^2 C_{in}$  is the “fan-in” of the kernel

$\gamma$  is a constant that depends on the nonlinearity

# NFNets: Weight Standardization

Rather than normalizing *activations* during training,  
instead normalize *weights*!

Learn weights  $W$  but convolve with weights  $\hat{W}$  where

$$\hat{W}_{i,j} = \gamma \cdot \frac{W_{i,j} - \text{mean}(W_i)}{\text{std}(W_i)\sqrt{N}}$$

$W_i$  is a single conv filter,  $N = K^2 C_{in}$  is the “fan-in” of the kernel

$\gamma$  is a constant that depends on the nonlinearity

$$\text{For ReLU: } \gamma = \sqrt{2 / (1 - (1/\pi))}$$

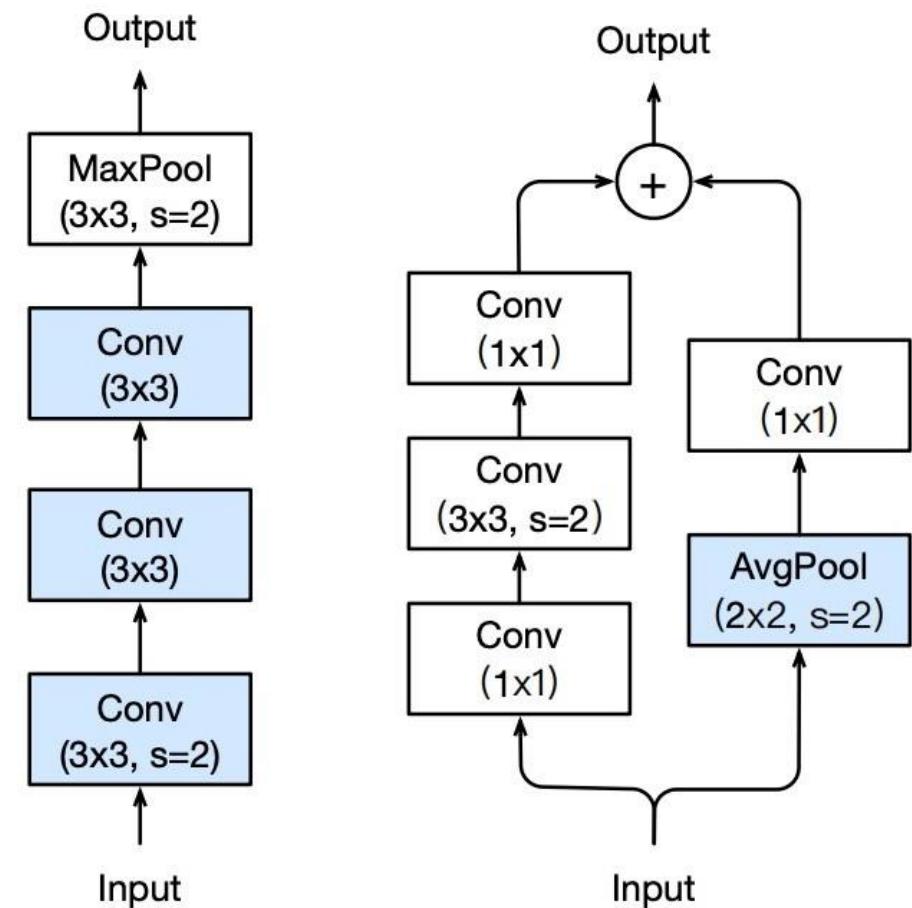
Compute  $\hat{W}$  each iteration during training (and backprop through it);  
at inference use fixed  $\hat{W}$  (zero-overhead like BN)

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: Other Tricks

- **Adaptive Gradient Clipping:** Clip (raw) gradients during training if they get too large
- **Tweak ResNet architecture:**
  - Start from SE-ResNeXt
  - Tweak stem and downsampling blocks (ResNet-D)
  - Change ReLU to GeLU
  - Group width = 128 at all layers
  - Change stage widths:  
 $[256, 512, 1024, 1024] \rightarrow [256, 512, 1536, 1536]$
  - Change stage depths:  $[3, 4, 6, 3] \rightarrow [1, 2, 6, 3]$
- **Stronger regularization:** MixUp, RandAugment, CutMix, DropOut, Stochastic Depth

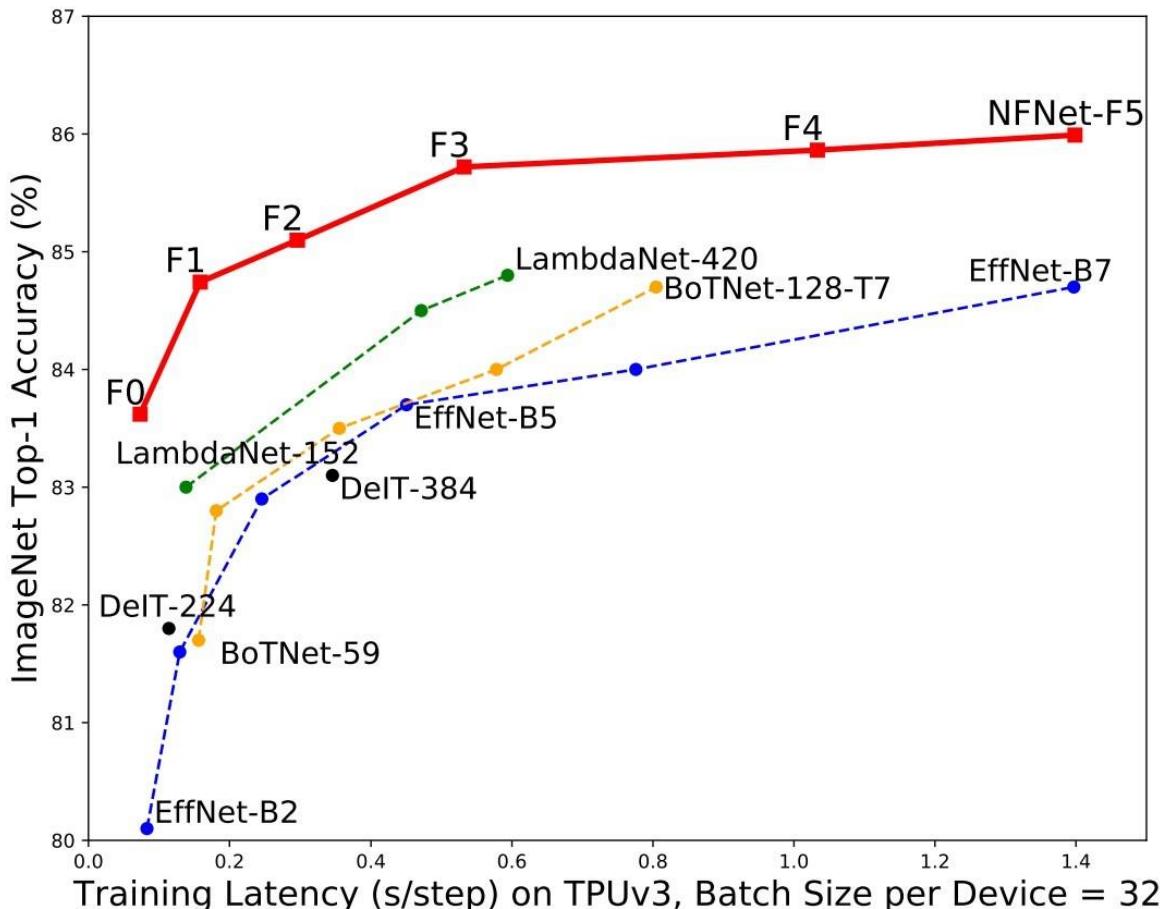


Brock et al, "Characterizing Signal Propagation to Close the Performance Gap in Unnormalized ResNets", ICLR 2021

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

Hu et al, "Bag of Tricks for Image Classification with Convolutional Neural Networks", CVPR 2019

# NFNets: Other Tricks



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 / ?

# Revisiting ResNets

Starting from baseline ResNet-200 model, improve performance with small tweaks:

Model	IN Top1	$\Delta$
Baseline ResNet-200:	79.0	
+Cosine LR decay	79.3	+0.3
+Longer training (90->350 epochs)	78.8	-0.5
+EMA of weights	79.1	+0.3
+Label smoothing	80.4	+1.3
+Stochastic Depth	80.6	+0.2
+RandAugment	81.0	+0.4
+Dropout on FC	80.7	-0.3
+Less weight decay	82.2	+1.5
+Squeeze and Excite	82.9	+0.7
+ResNet-D	83.4	+0.5

# Revisiting ResNets

To get networks of different sizes,  
brute-force search over:

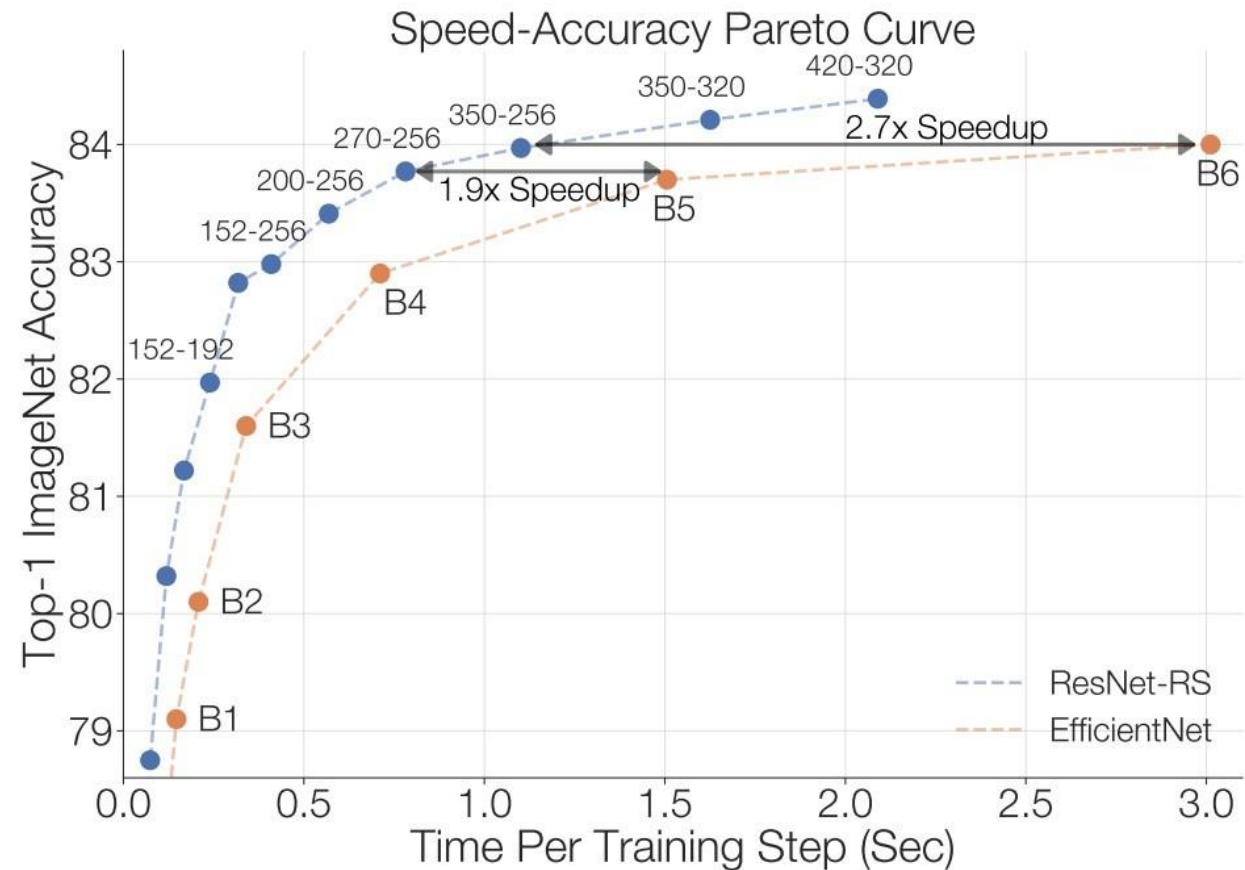
- Initial network width: 0.25x, 0.5x, 1.0x, 1.5x, or 2.0x baseline model
- Overall network depth: 26, 50, 101, 200, 300, 350, or 400 layers
- Input image resolution: 128, 160, 224, 320, or 448 pixels

# Revisiting ResNets

To get networks of different sizes,  
brute-force search over:

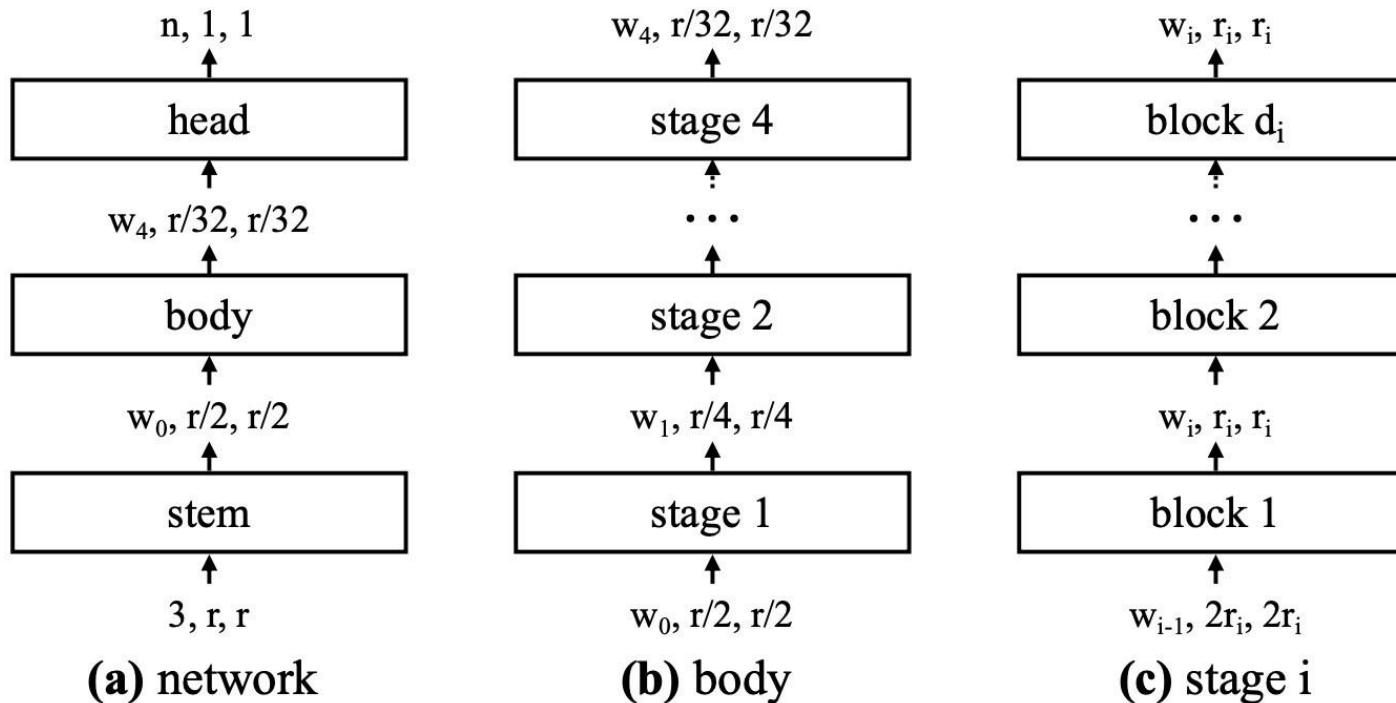
- Initial network width: 0.25x, 0.5x, 1.0x, 1.5x, or 2.0x baseline model
- Overall network depth: 26, 50, 101, 200, 300, 350, or 400 layers
- Input image resolution: 128, 160, 224, 320, or 448 pixels

Significantly faster than EfficientNets  
at same accuracy (times on TPU)



# 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



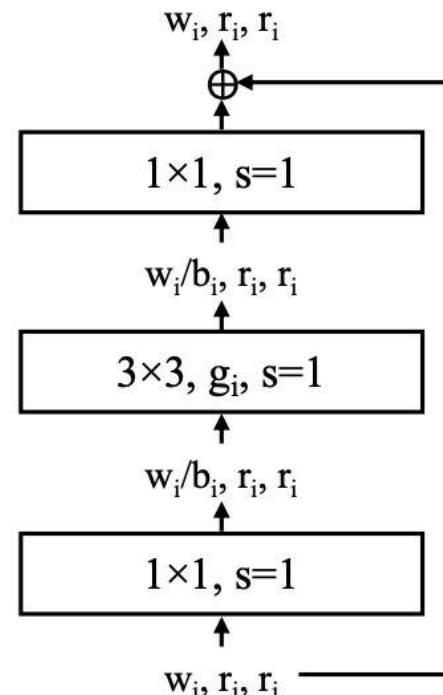
# RegNets: Network Design Spaces

Block design is simple,  
generalizes ResNext

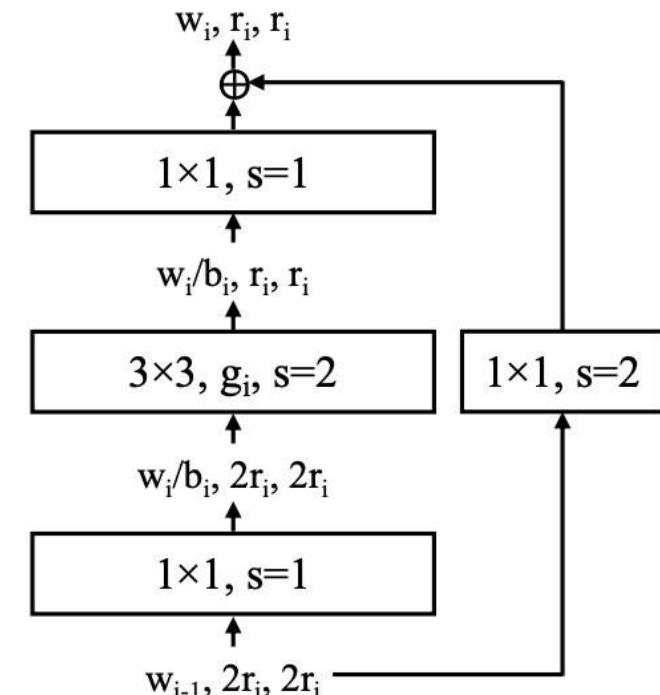
Each stage has 4 parameters:

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

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



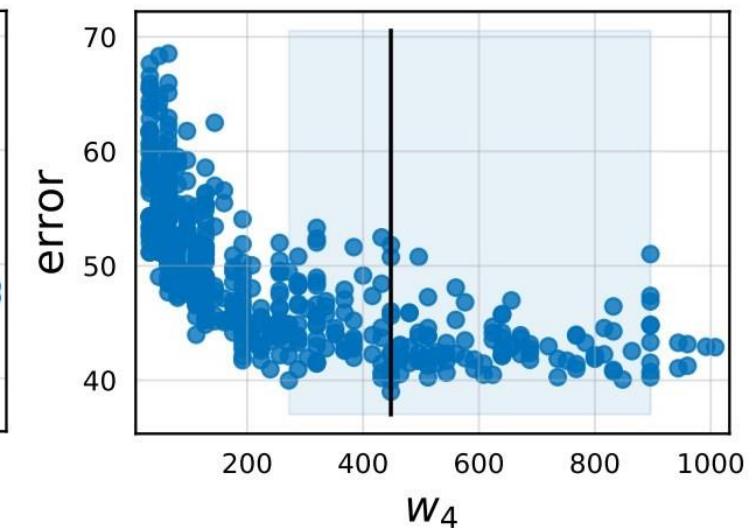
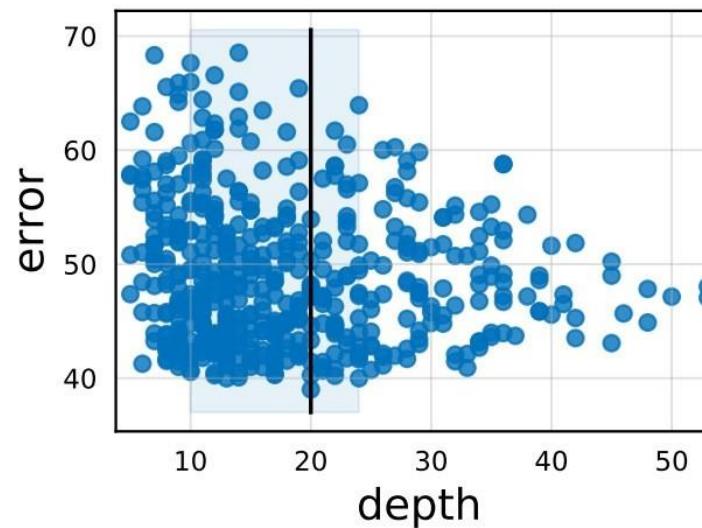
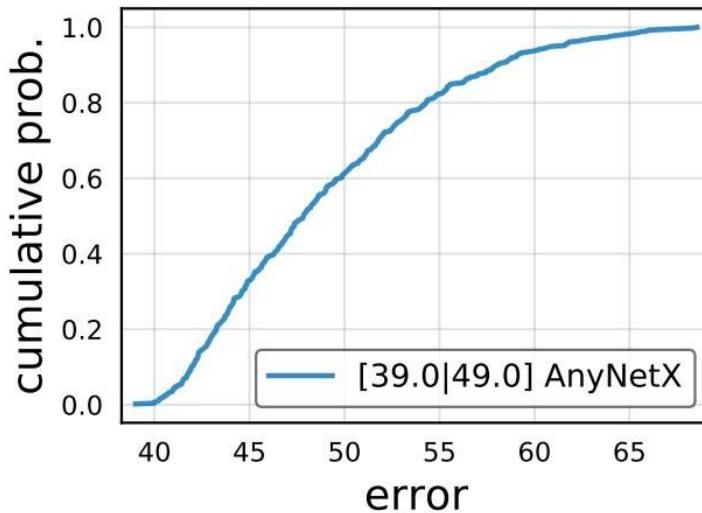
(a) X block,  $s=1$



(b) X block,  $s=2$

# RegNets: Network Design Spaces

Randomly sample architectures from the design space, examine trends:



# RegNets: Network Design Spaces

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

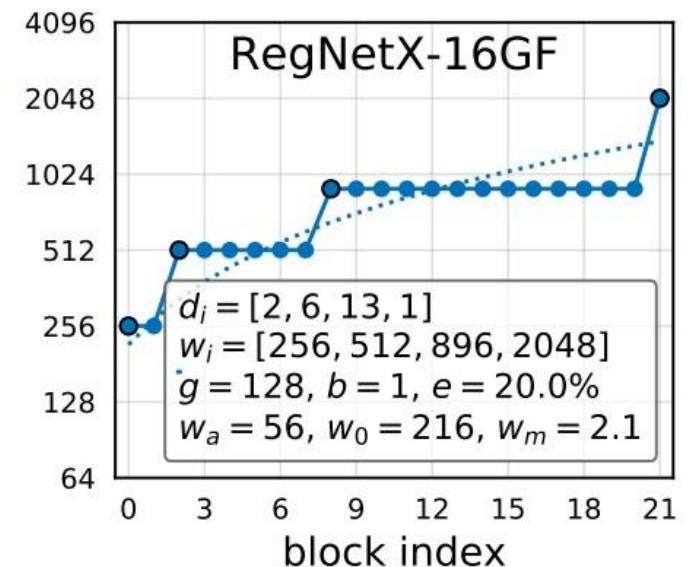
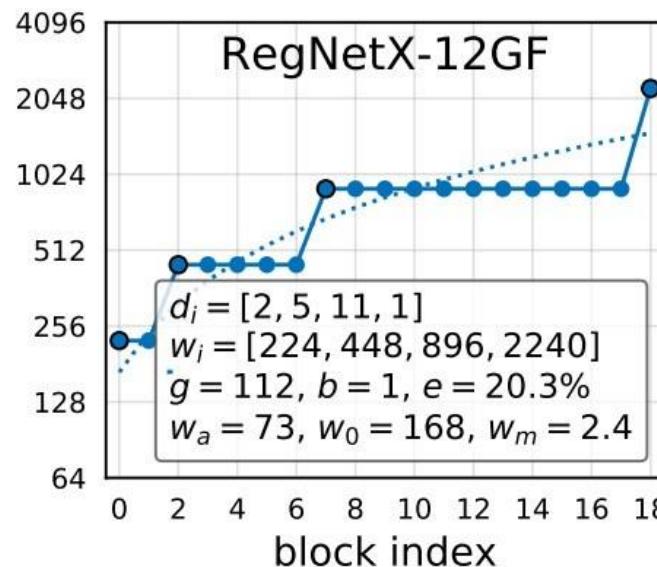
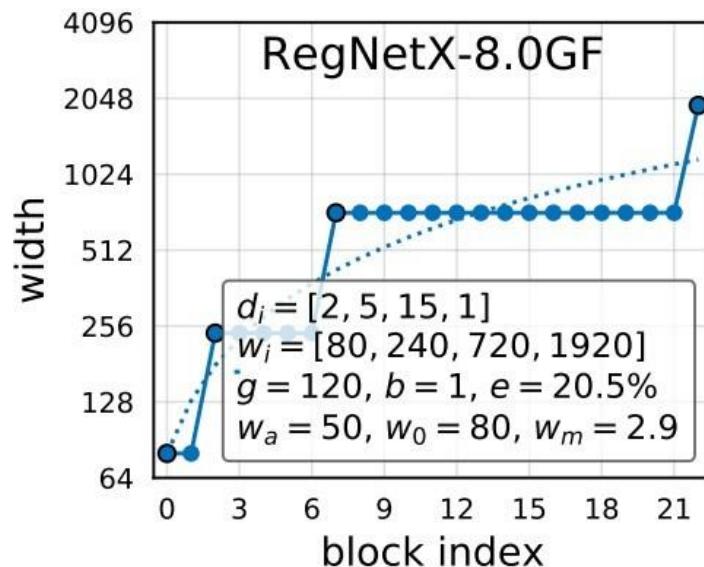
- Share bottleneck ratio across all stages (16  $\rightarrow$  13 params)
- Share group width across all stages (13  $\rightarrow$  10 params)
- Force width, blocks per stage to increase *linearly* across stages

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  $w_m$

# RegNets: Network Design Spaces

Random search finds good-performing models at varying FLOP budgets

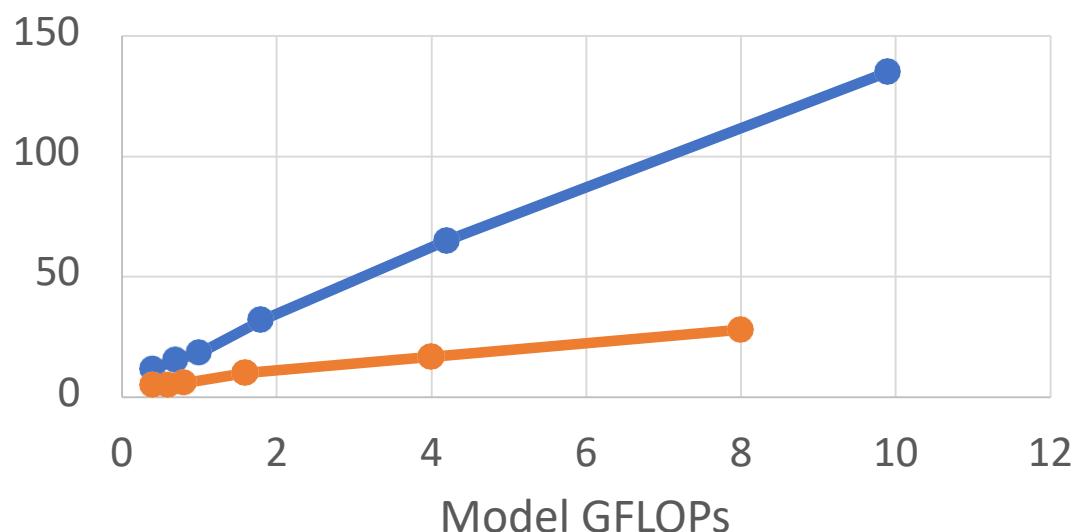


RegNetX is as described above, RegNetY also adds SE

# RegNets: Network Design Spaces

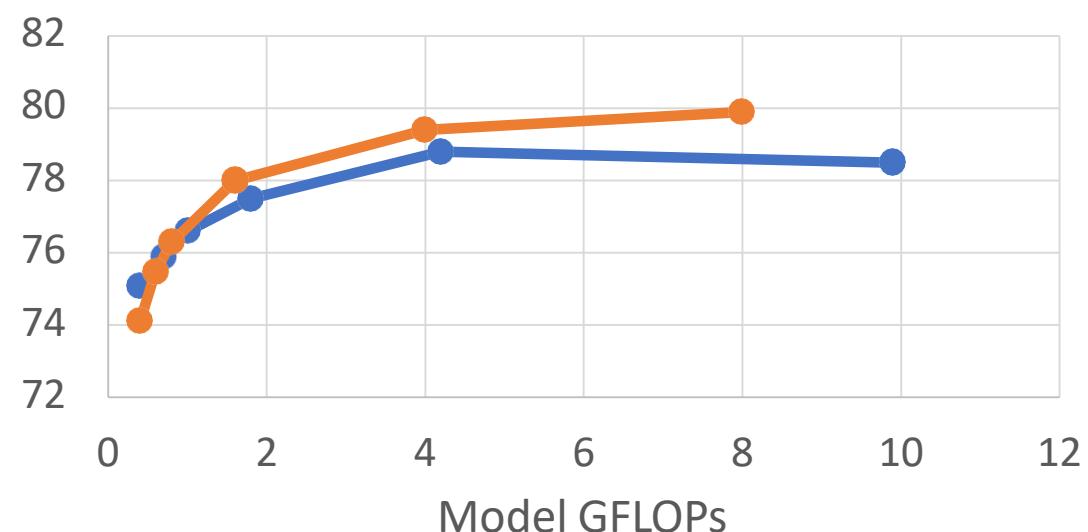
Training Time (hours)

EfficientNet RegNetY



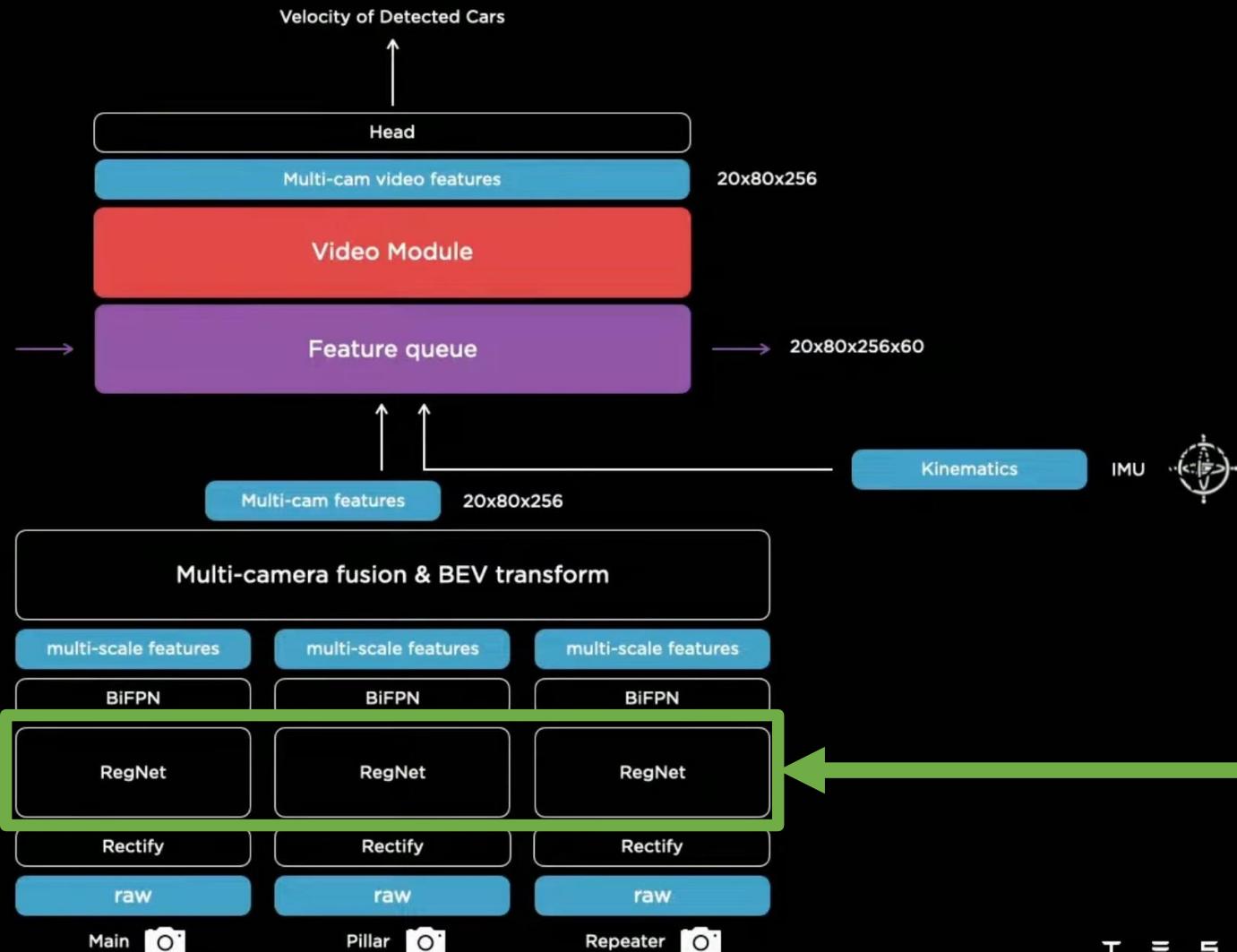
ImageNet Top1 Accuracy

EfficientNet RegNetY



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

# Video Neural Net Architecture



Tesla Vision system uses RegNets to process inputs from each camera

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

# CNN Architectures Summary

- Early work (AlexNet -> VGG -> ResNet): **bigger networks work better**
- New focus on **efficiency**: Improve accuracy, control for network complexity
- **Grouped and Depthwise Convolution** appear in many modern architectures
- **Squeeze-and-Excite** adds accuracy boost to just about any architecture while only adding a tiny amount of FLOPs and runtime
- Tiny networks for **mobile devices** (MobileNet, ShuffleNet)
- **Neural Architecture Search (NAS)** promised to automate architecture design
- More recent work has moved towards **careful improvements to ResNet-like architectures**
- ResNet and ResNeXt are still surprisingly strong and popular architectures!
- RegNet seems like a promising and efficient architecture to use

## A Sneak Peek...

A lot of recent work has started to move away from CNNs entirely!

New classes of models: Vision Transformers, MLP-like models

We will learn more in the following lectures

Next Time: How do we implement all this?

Deep Learning Software