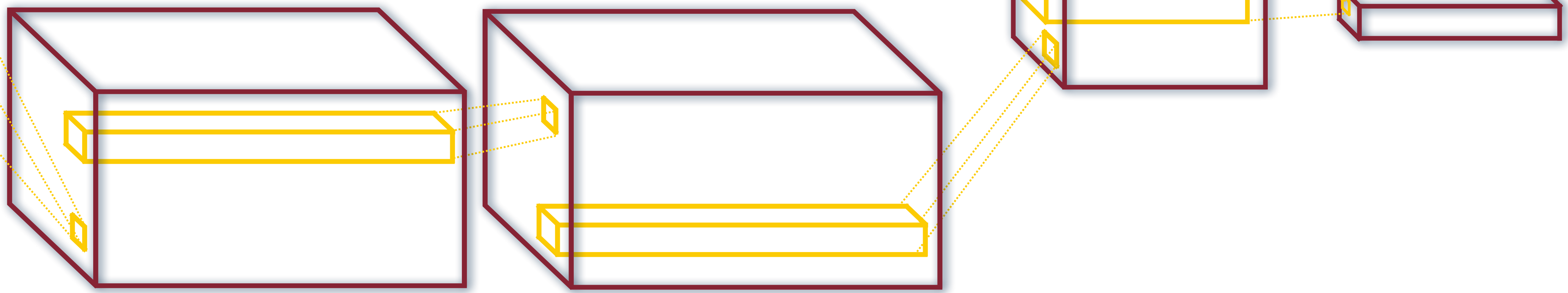
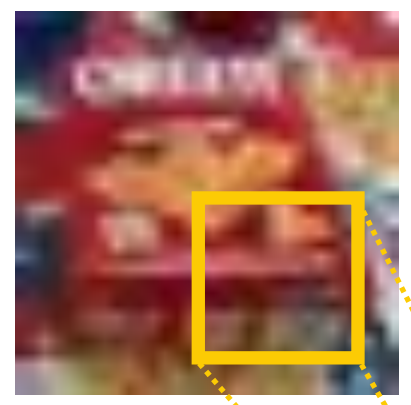


DeepRob

Lecture 8
 CNN Architectures
 University of Michigan and University of Minnesota





Office Hours Schedule

| Monday 01/30 | Tuesday 01/31 | Wednesday 02/01 | Thursday 02/02 | Friday 02/03 |
|--|--|---|---|---|
| 9:00 AM | | | | |
| 9:30 AM | | | | |
| 10:00 AM | | | | |
| 10:30 AM | | | | |
| 11:00 AM | | | | |
| 11:30 AM | | | | |
| 12:00 PM | | | | |
| 12:30 PM | | | | |
| 1:00 PM | | | Jiyue's Office Hours 1:00 PM–3:00 PM 2320 FMCRB Zoom Link | |
| 1:30 PM | | Anthony's Office Hours 1:30 PM–3:00 PM 3320 FMCRB Zoom Link | | |
| 2:00 PM | Huijie's Office Hours 2:00 PM–3:00 PM 2320 FMCRB Zoom Link | | | |
| Huijie's Office Hours 2:30 PM–3:30 PM 2320 FMCRB Zoom Link | Lecture 8 3:00 PM–4:30 PM 1060 FMCRB Zoom Link | | Lecture 9 3:00 PM–4:30 PM 1060 FMCRB Zoom Link | |
| 3:30 PM | | | | |
| 4:00 PM | | | | |
| 4:30 PM | Anthony's Office Hours 4:30 PM–5:30 PM 1060 FMCRB | | Anthony's Office Hours 4:30 PM–5:30 PM 1060 FMCRB | Discussion 5 4:30 PM–5:30 PM 1060 FMCRB Zoom Link |
| 5:00 PM | | | | |
| 5:30 PM | | | | Anthony's Office Hours 5:30 PM–6:30 PM 1060 FMCRB |
| 6:00 PM | | | | |



Project 2—Updates

- Instructions available on the website
- Here: deeprob.org/projects/project2/
- Starter code sent via email
- Implement two-layer neural network and generalize to FCN
- **Autograder will be available in next day or so**
- **Due Thursday, February 9th 11:59 PM EST**



Final Project Overview

- Research-oriented final project
 - Instead of a final exam!
- Objectives
 - Gain experience reading literature
 - Reproduce published results
 - Propose a new idea and test the results!

Final Project Overview

- Research-oriented final project
- Instead of a final exam!

Can be completed in teams of 1-3 people

- Objectives
 - Gain experience reading literature
 - Reproduce published results
 - Propose a new idea and test the results!

Final Project Paper Selection Survey

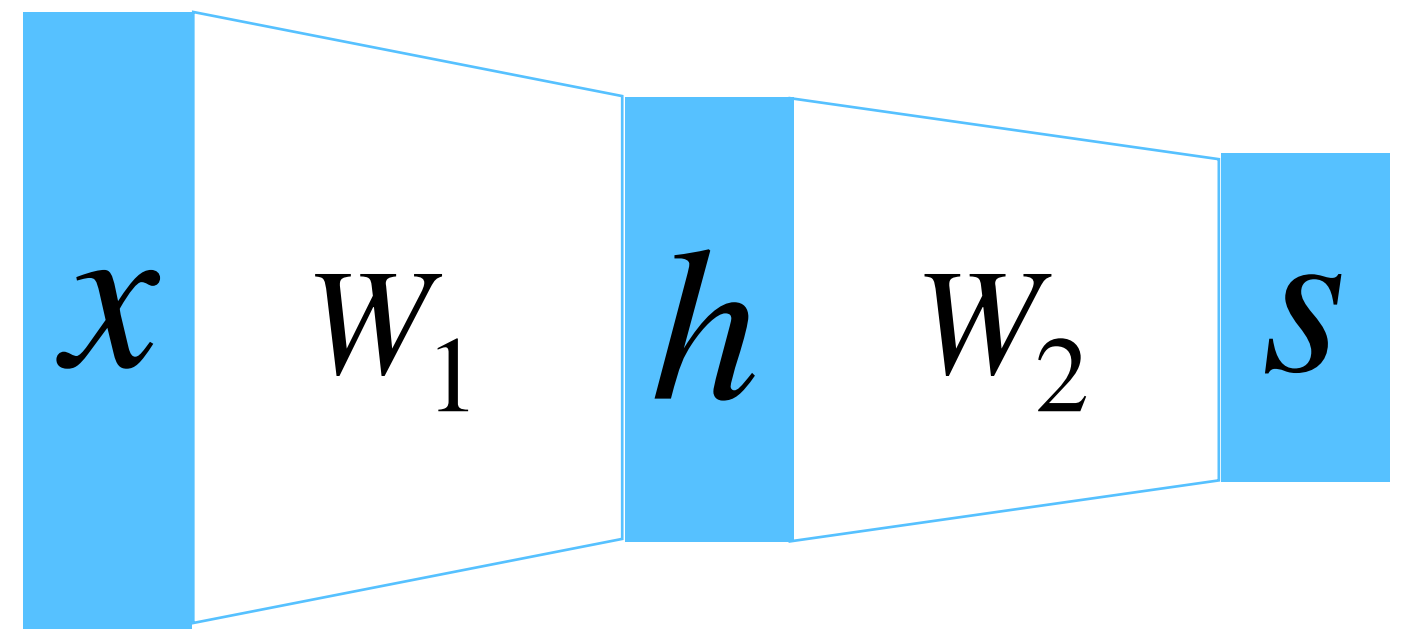
- Published on [gradescope](#)
- To gauge your areas of interest
- Used for forming teams

- **Due Friday, February 3rd 11:59 PM EST**

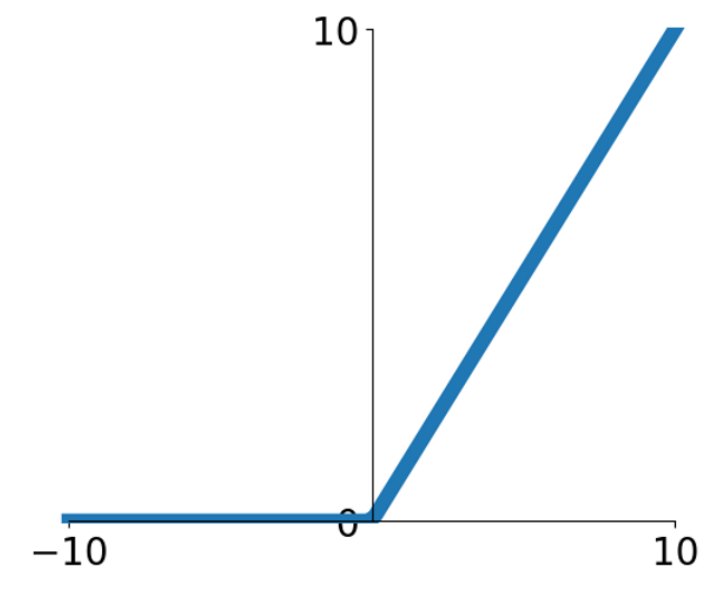


Recap: Components of Convolutional Network

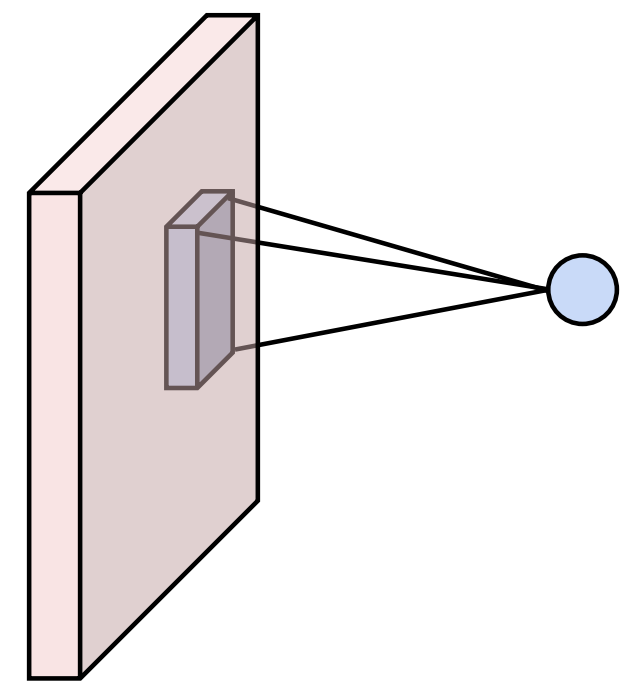
Fully-Connected Layers



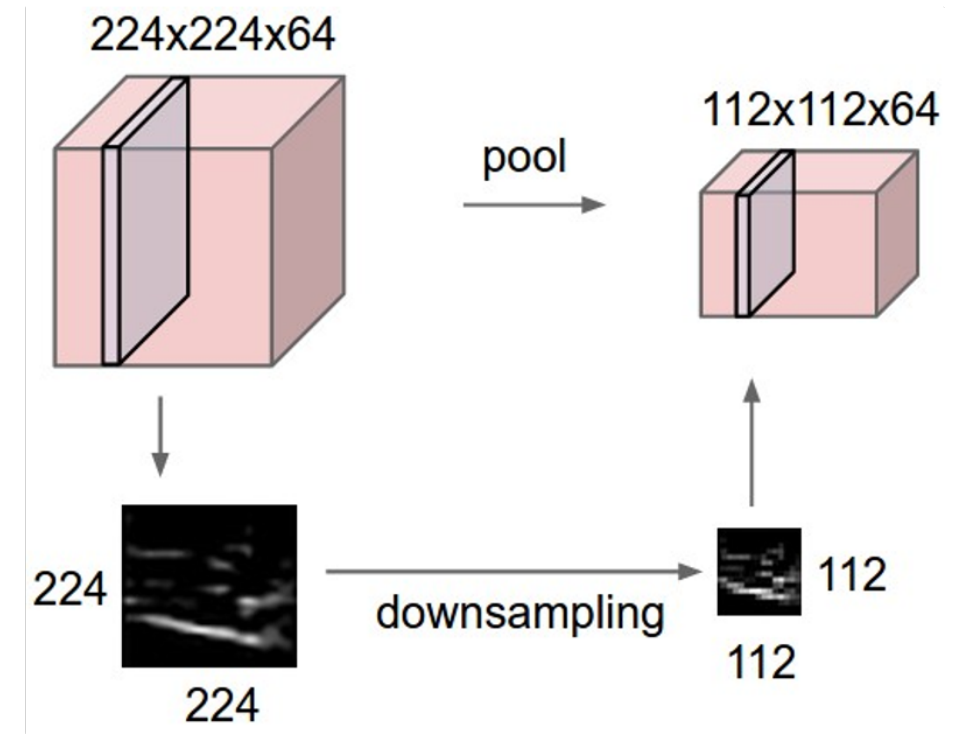
Activation Functions



Convolution Layers



Pooling Layers



Normalization

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$



Batch Normalization

Consider a single layer $y = Wx$

The following could lead to tough optimization:

- Inputs x are not *centered around zero* (need large bias)
- Inputs x have different scaling per-element (entries in W will need to vary a lot)

Idea: force inputs to be “nicely scaled” at each layer!



Batch Normalization

Idea: “Normalize” the inputs of a layer so they have zero mean and unit variance

We can normalize a batch of activations like this:

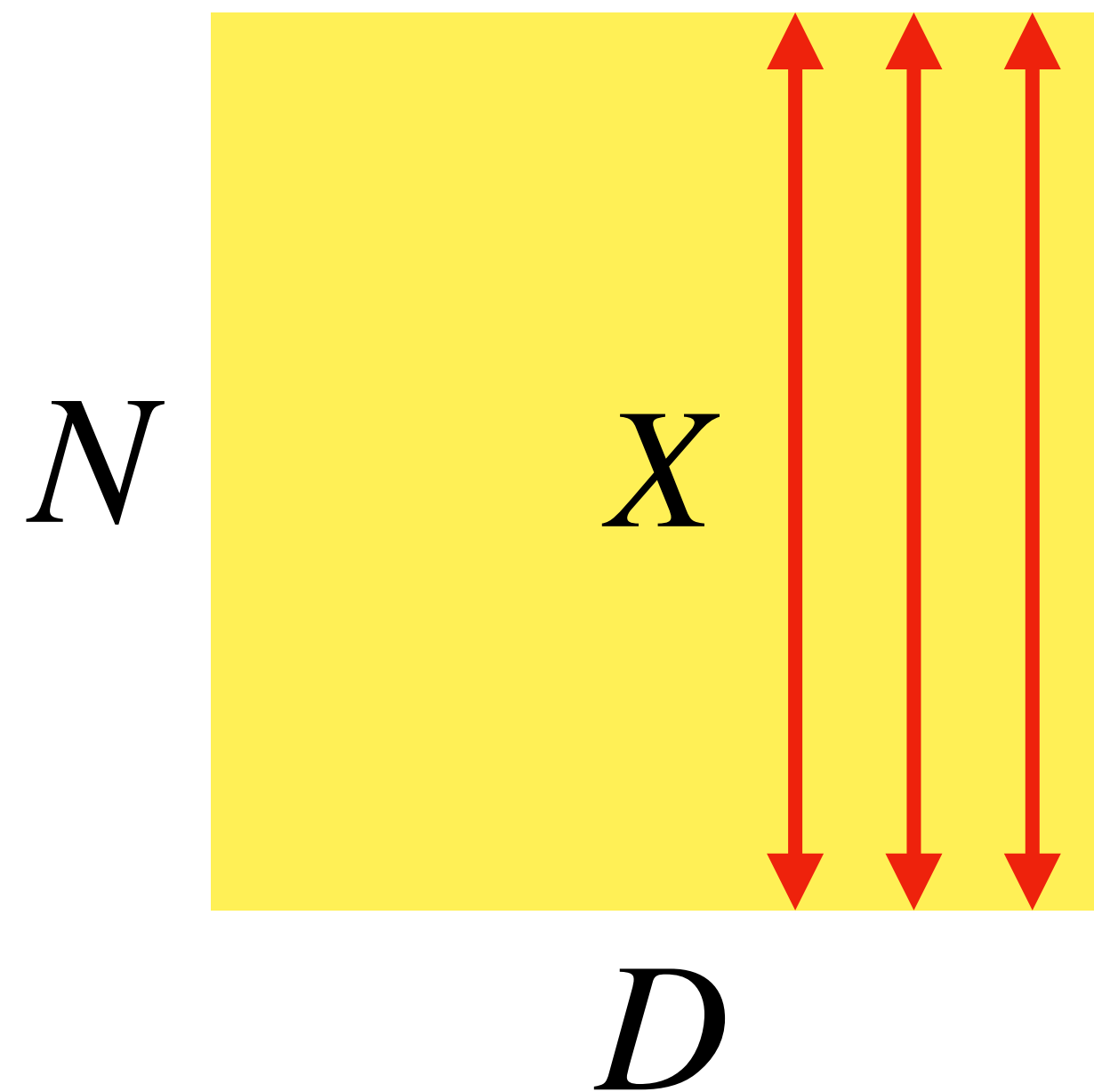
$$\hat{x} = \frac{x - E[x]}{\sqrt{\text{Var}[x]}}$$

This is a **differentiable function**, so we can use it as an operator in our networks and backprop through it!



Batch Normalization

Input: $x \in \mathbb{R}^{N \times D}$



$$\mu_j = \frac{1}{N} \sum_{i=1}^N x_{i,j}$$

Per-channel mean,
shape is D

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (x_{i,j} - \mu_j)^2$$

Per-channel std,
shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

Normalized x ,
shape is $N \times D$

Problem: What if zero-mean, unit variance is too hard of a constraint?

Batch Normalization

Input: $x \in \mathbb{R}^{N \times D}$

Learnable scale and shift parameters: $\gamma, \beta \in \mathbb{R}^D$

Learning $\gamma = \sigma, \beta = \mu$ will recover the identity function (in expectation)

$$\mu_j = \frac{1}{N} \sum_{i=1}^N x_{i,j}$$

Per-channel mean, shape is D

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (x_{i,j} - \mu_j)^2$$

Per-channel std, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

Normalized x , shape is $N \times D$

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Output, shape is $N \times D$



Batch Normalization

Problem: Estimates depend on minibatch; can't do this at test-time

Input: $x \in \mathbb{R}^{N \times D}$

Learnable scale and shift parameters: $\gamma, \beta \in \mathbb{R}^D$

Learning $\gamma = \sigma, \beta = \mu$ will recover the identity function (in expectation)

$$\mu_j = \frac{1}{N} \sum_{i=1}^N x_{i,j}$$

Per-channel mean, shape is D

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (x_{i,j} - \mu_j)^2$$

Per-channel std, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

Normalized x , shape is $N \times D$

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Output, shape is $N \times D$

Batch Normalization: Test-Time

Input: $x \in \mathbb{R}^{N \times D}$

Learnable scale and shift parameters: $\gamma, \beta \in \mathbb{R}^D$

Learning $\gamma = \sigma, \beta = \mu$ will recover the identity function (in expectation)

$\mu_j =$ (Running) average of values seen during training

$\sigma_j^2 =$ (Running) average of values seen during training

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Per-channel mean, shape is D

Per-channel std, shape is D

Normalized x , shape is $N \times D$

Output, shape is $N \times D$



Batch Normalization: Test-Time

Input: $x \in \mathbb{R}^{N \times D}$

$$\mu_j =$$

(Running) average of
values seen during
training

Per-channel mean,
shape is D

**Learnable scale and shift
parameters:** $\gamma, \beta \in \mathbb{R}^D$

Learning $\gamma = \sigma, \beta = \mu$ will
recover the identity
function (in expectation)

$$\mu_j^{test} = 0$$

For each training iteration:

$$\mu_j = \frac{1}{N} \sum_{i=1}^N x_{i,j}$$

$$\mu_j^{test} = 0.99 \mu_j^{test} + 0.01 \mu_j$$

(Similar for σ)

Batch Normalization: Test-Time

Input: $x \in \mathbb{R}^{N \times D}$

Learnable scale and shift parameters: $\gamma, \beta \in \mathbb{R}^D$

Learning $\gamma = \sigma, \beta = \mu$ will recover the identity function (in expectation)

$\mu_j =$ (Running) average of values seen during training

$\sigma_j^2 =$ (Running) average of values seen during training

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Per-channel mean, shape is D

Per-channel std, shape is D

Normalized x , shape is $N \times D$

Output, shape is $N \times D$



Batch Normalization: Test-Time

Input: $x \in \mathbb{R}^{N \times D}$

Learnable scale and shift parameters: $\gamma, \beta \in \mathbb{R}^D$

During testing batchnorm becomes a linear operator!
Can be fused with the previous fully-connected or conv layer

$\mu_j =$ (Running) average of values seen during training

Per-channel mean, shape is D

$\sigma_j^2 =$ (Running) average of values seen during training

Per-channel std, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

Normalized x , shape is $N \times D$

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Output, shape is $N \times D$



Batch Normalization for Convents

Batch Normalization for
fully-connected networks

$$x : N \times D$$

Normalize



$$\mu, \sigma : 1 \times D$$

$$\gamma, \beta : 1 \times D$$

$$y = \frac{(x - \mu)}{\sigma} \gamma + \beta$$

Batch Normalization for
convolutional networks
(Spatial Batchnorm, BatchNorm2D)

$$x : N \times C \times H \times W$$

Normalize

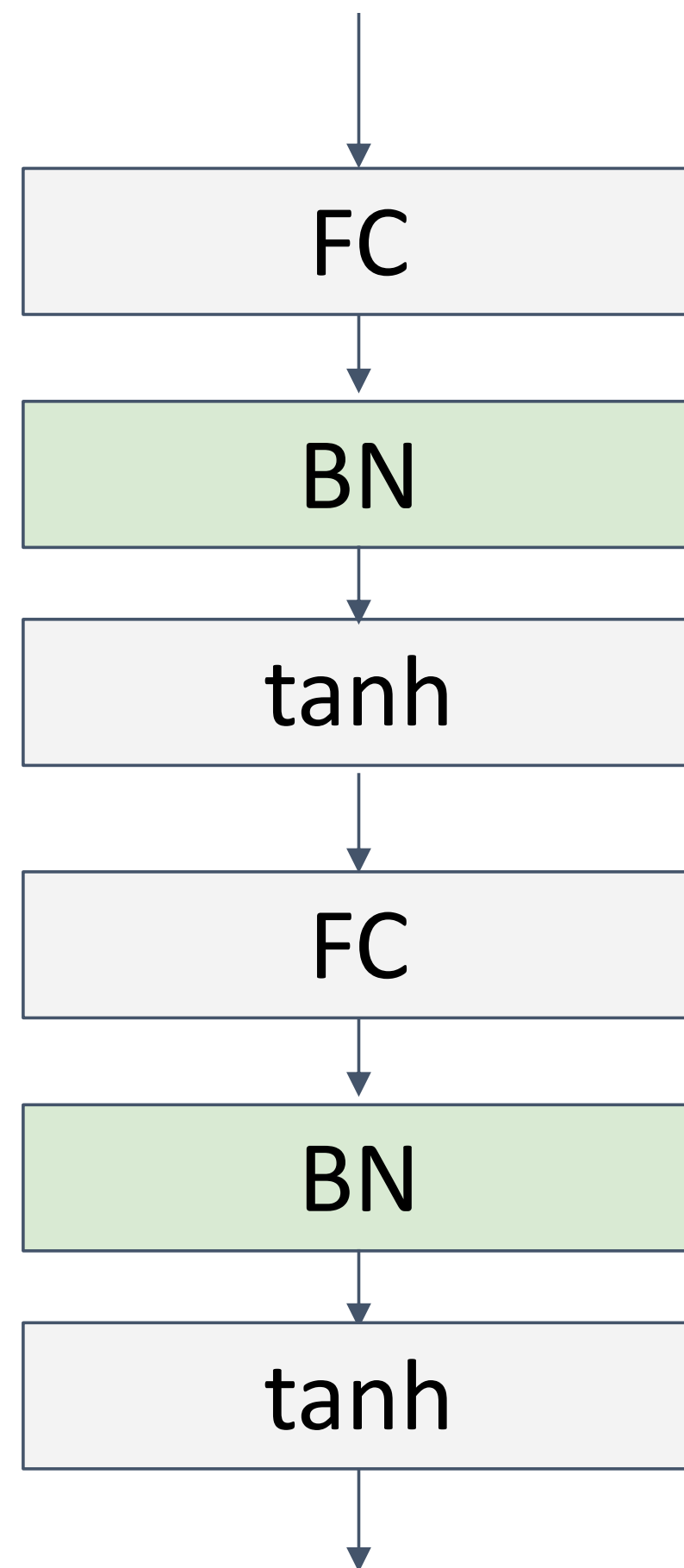


$$\mu, \sigma : 1 \times C \times 1 \times 1$$

$$\gamma, \beta : 1 \times C \times 1 \times 1$$

$$y = \frac{(x - \mu)}{\sigma} \gamma + \beta$$

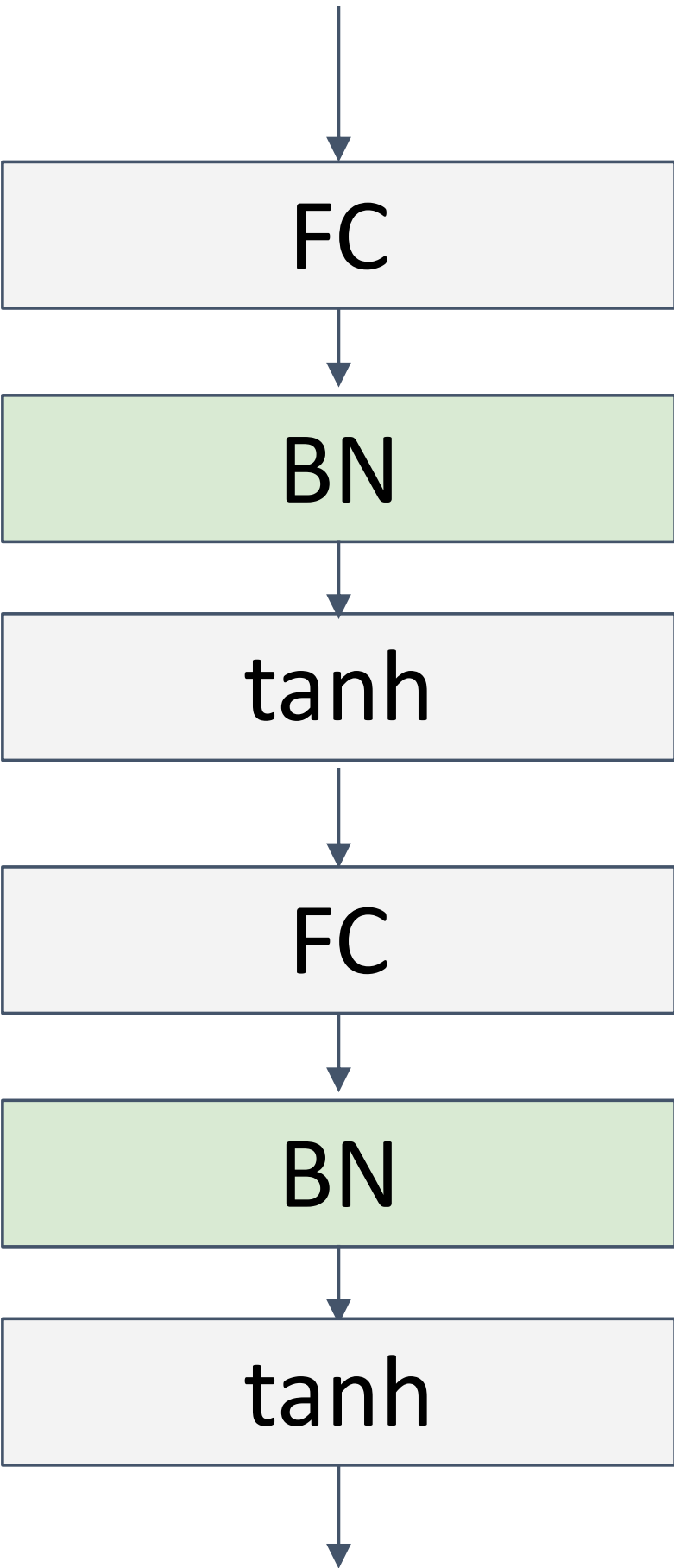
Batch Normalization



Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

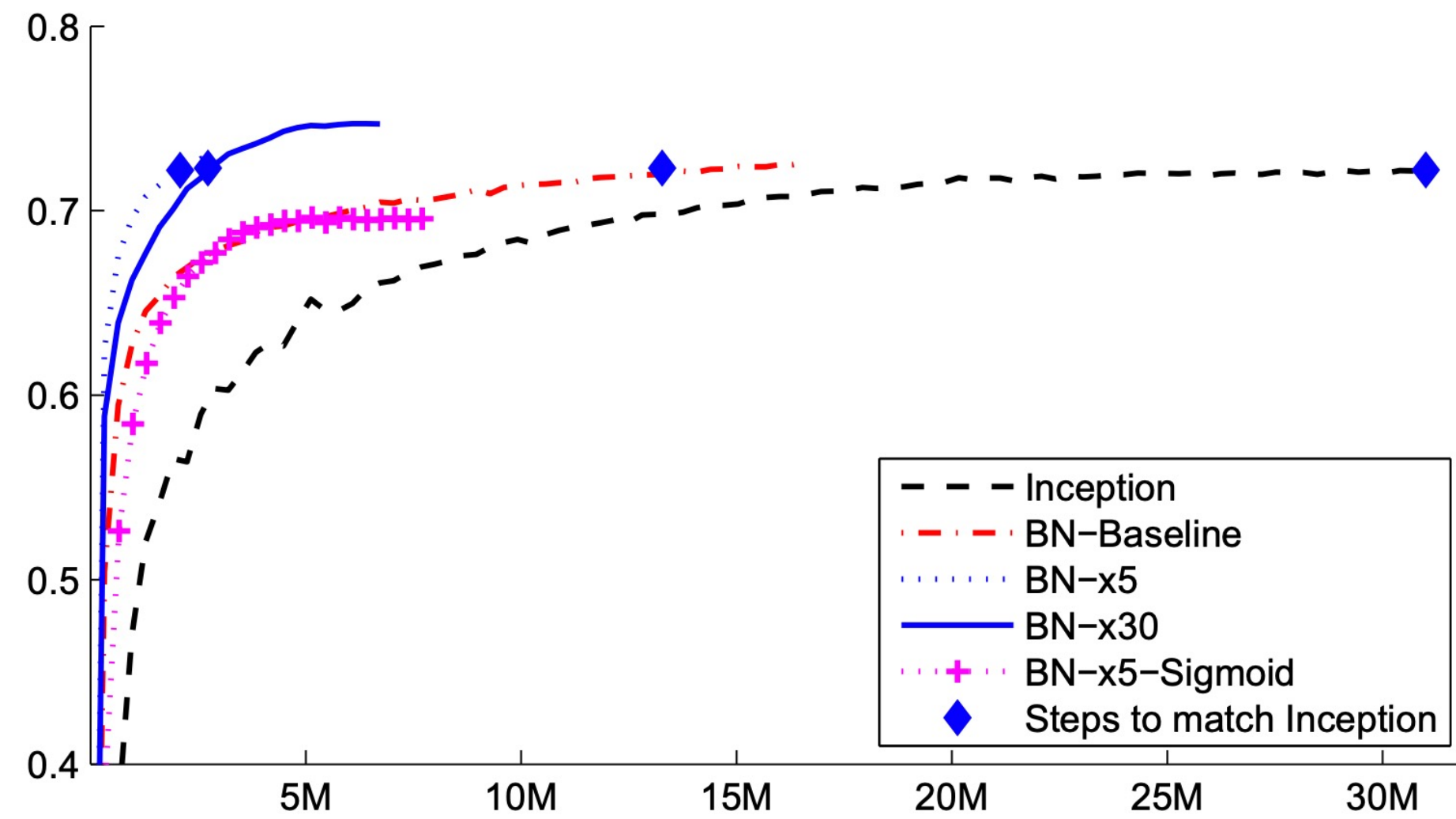
$$\hat{x} = \frac{x - E[x]}{\sqrt{\text{Var}[x]}}$$

Batch Normalization



- Makes deep networks **much** easier to train!
- Allows higher learning rates, faster convergence
- Networks become more robust to initialization
- Acts as regularization during training.
- Zero overhead at test-time: can be fused with conv!

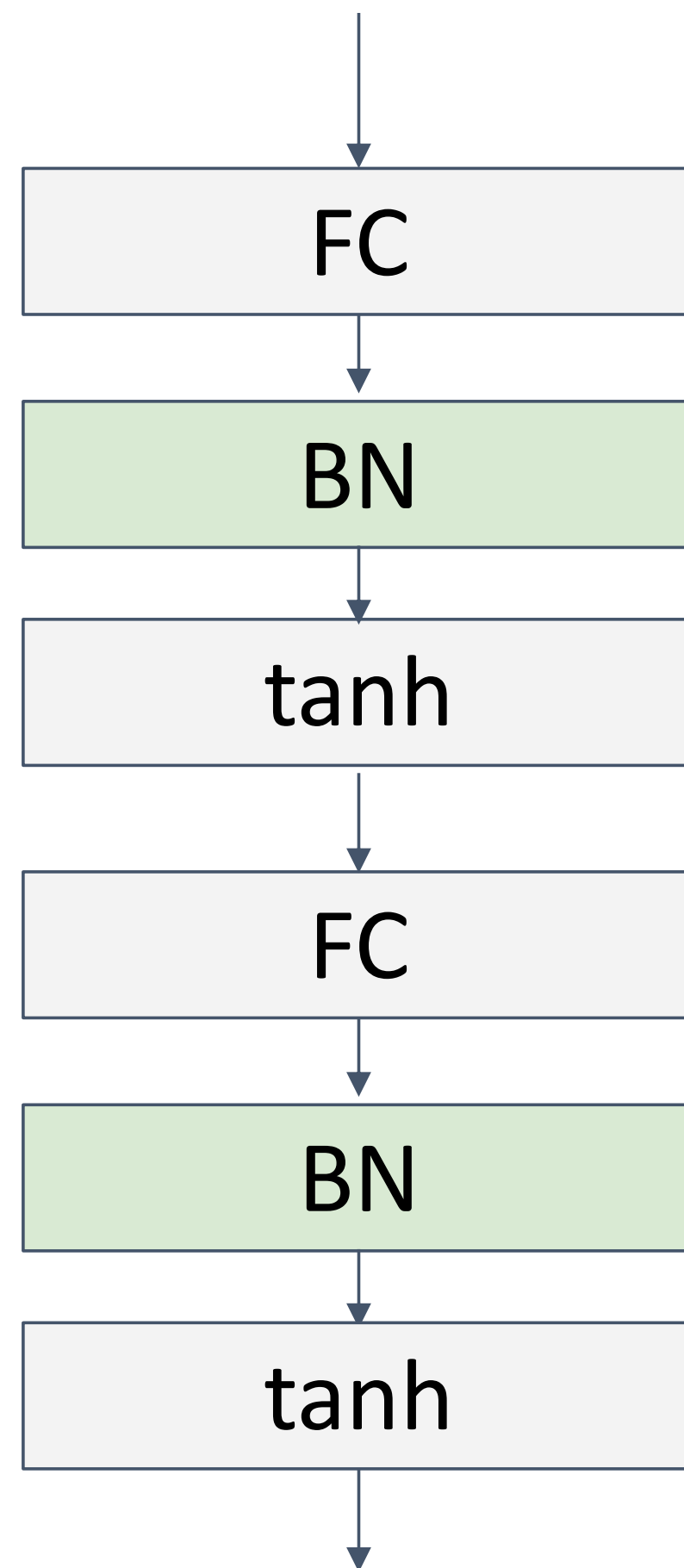
ImageNet accuracy



Training iterations



Batch Normalization



- Makes deep networks **much** easier to train!
- Allows higher learning rates, faster convergence
- Networks become more robust to initialization
- Acts as regularization during training.
- Zero overhead at test-time: can be fused with conv!
- Not well-understood theoretically (yet)
- Behaves differently during training and testing: this is very common source of bugs!

Layer Normalization

Batch Normalization for **fully-connected** networks

$$x : N \times D$$

Normalize

$$\mu, \sigma : 1 \times D$$

$$\gamma, \beta : 1 \times D$$

$$y = \frac{(x - \mu)}{\sigma} \gamma + \beta$$

Layer Normalization for **fully-connected** networks

Same behavior at train and test!

Used in RNNs, Transformers

$$x : N \times D$$

Normalize

$$\mu, \sigma : N \times 1$$

$$\gamma, \beta : 1 \times D$$

$$y = \frac{(x - \mu)}{\sigma} \gamma + \beta$$

Instance Normalization

Batch Normalization for
convolutional networks

$$x : N \times C \times H \times W$$

Normalize

$$\mu, \sigma : 1 \times C \times 1 \times 1$$

$$\gamma, \beta : 1 \times C \times 1 \times 1$$

$$y = \frac{(x - \mu)}{\sigma} \gamma + \beta$$

Instance Normalization for
convolutional networks

$$x : N \times C \times H \times W$$

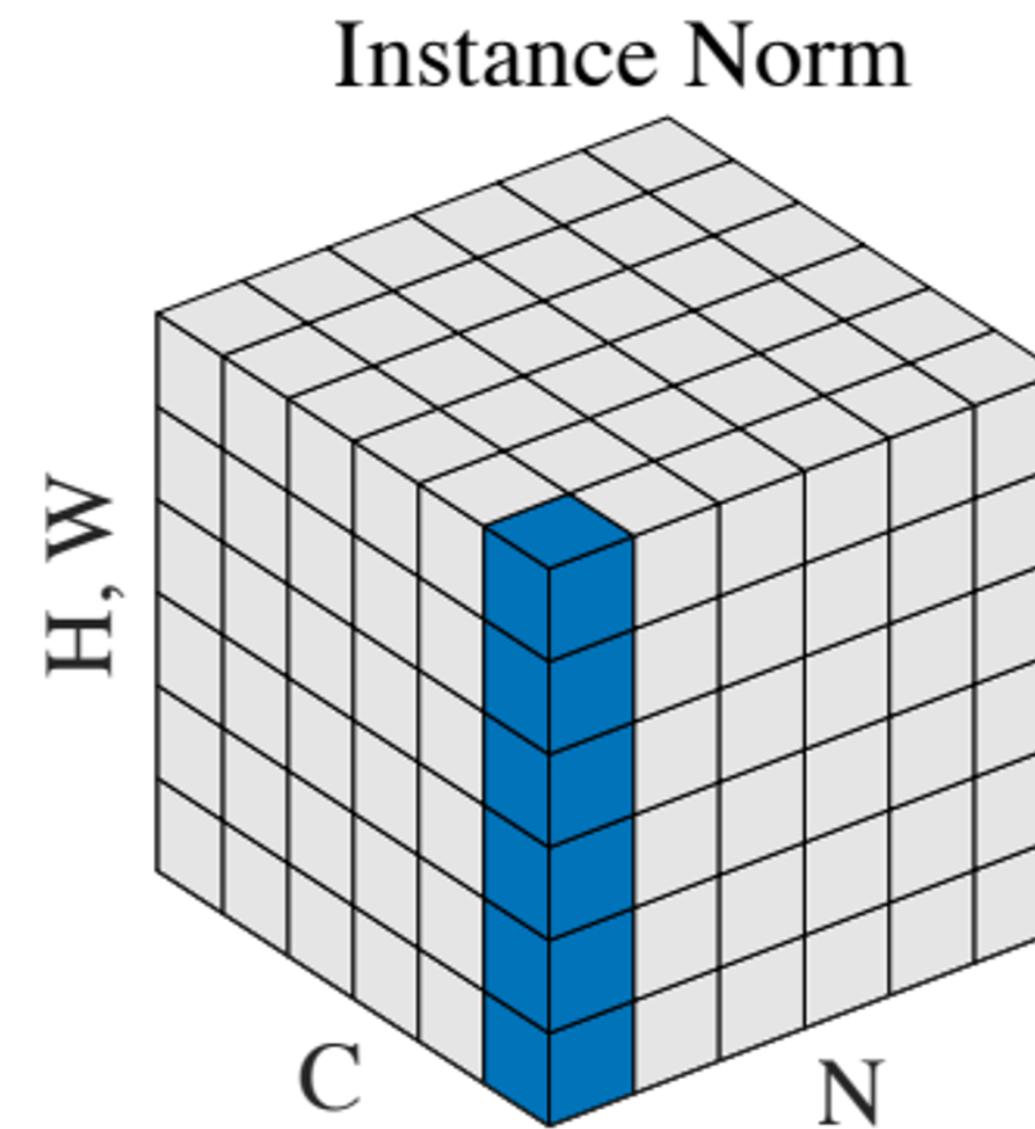
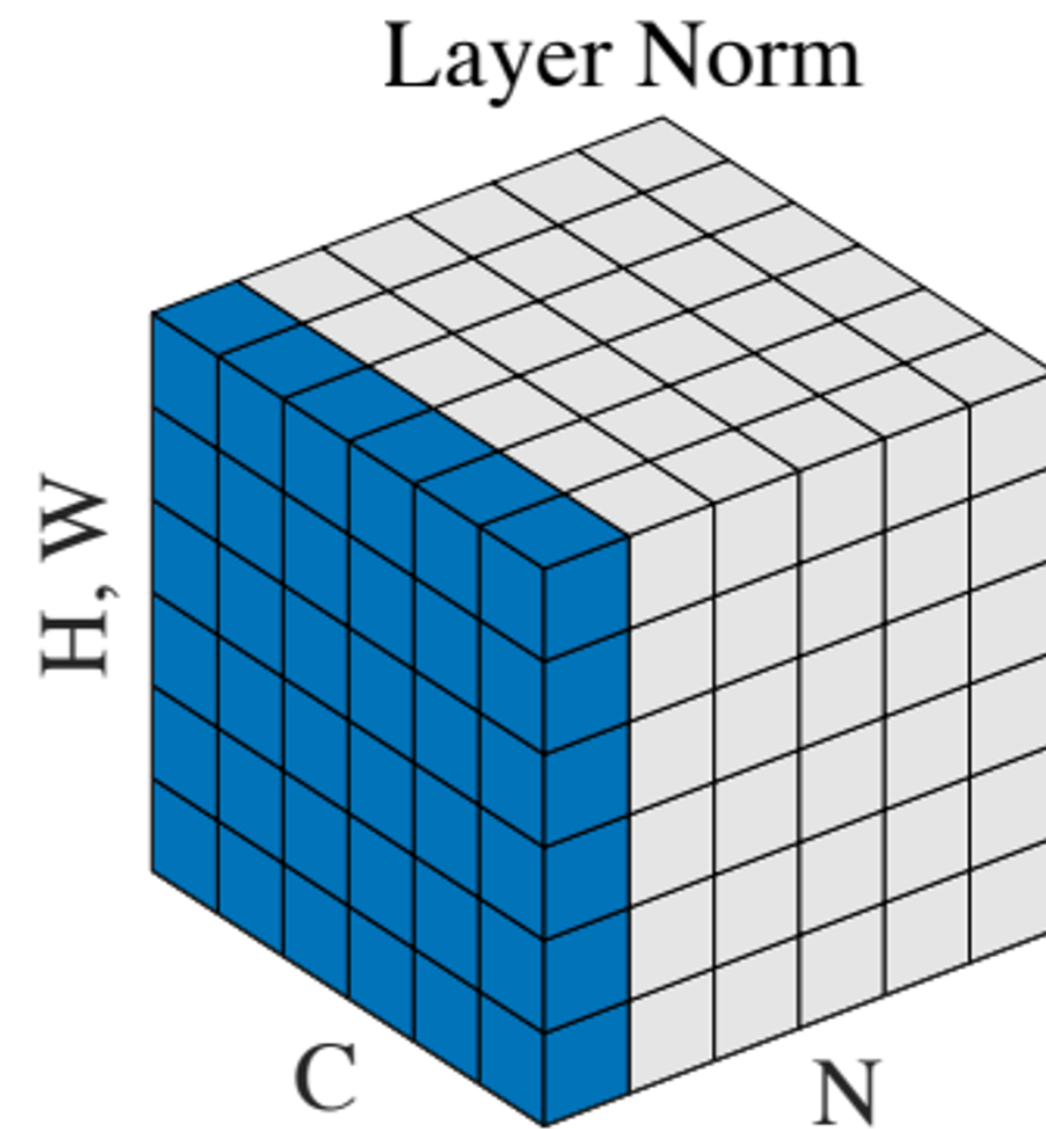
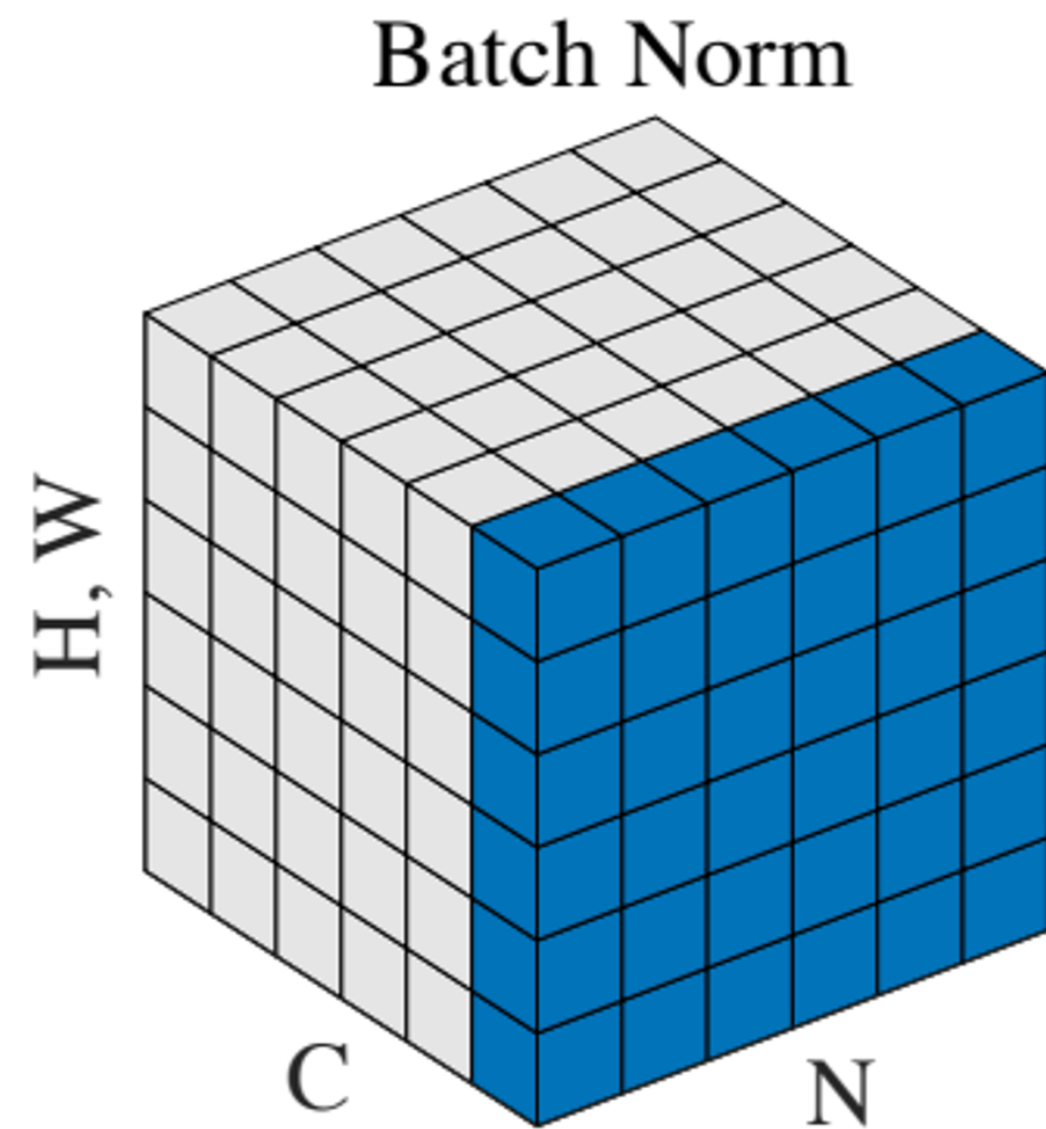
Normalize

$$\mu, \sigma : N \times C \times 1 \times 1$$

$$\gamma, \beta : 1 \times C \times 1 \times 1$$

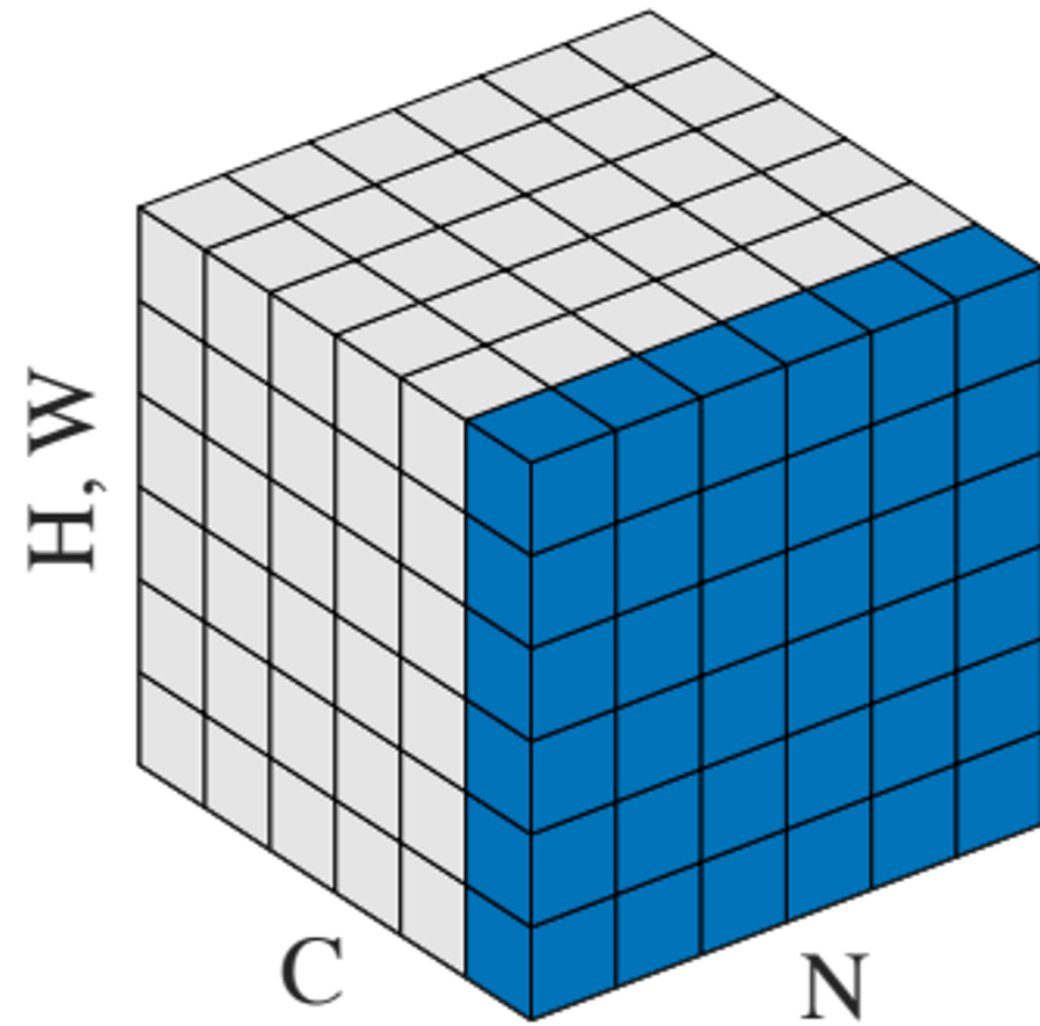
$$y = \frac{(x - \mu)}{\sigma} \gamma + \beta$$

Comparison of Normalization Layers

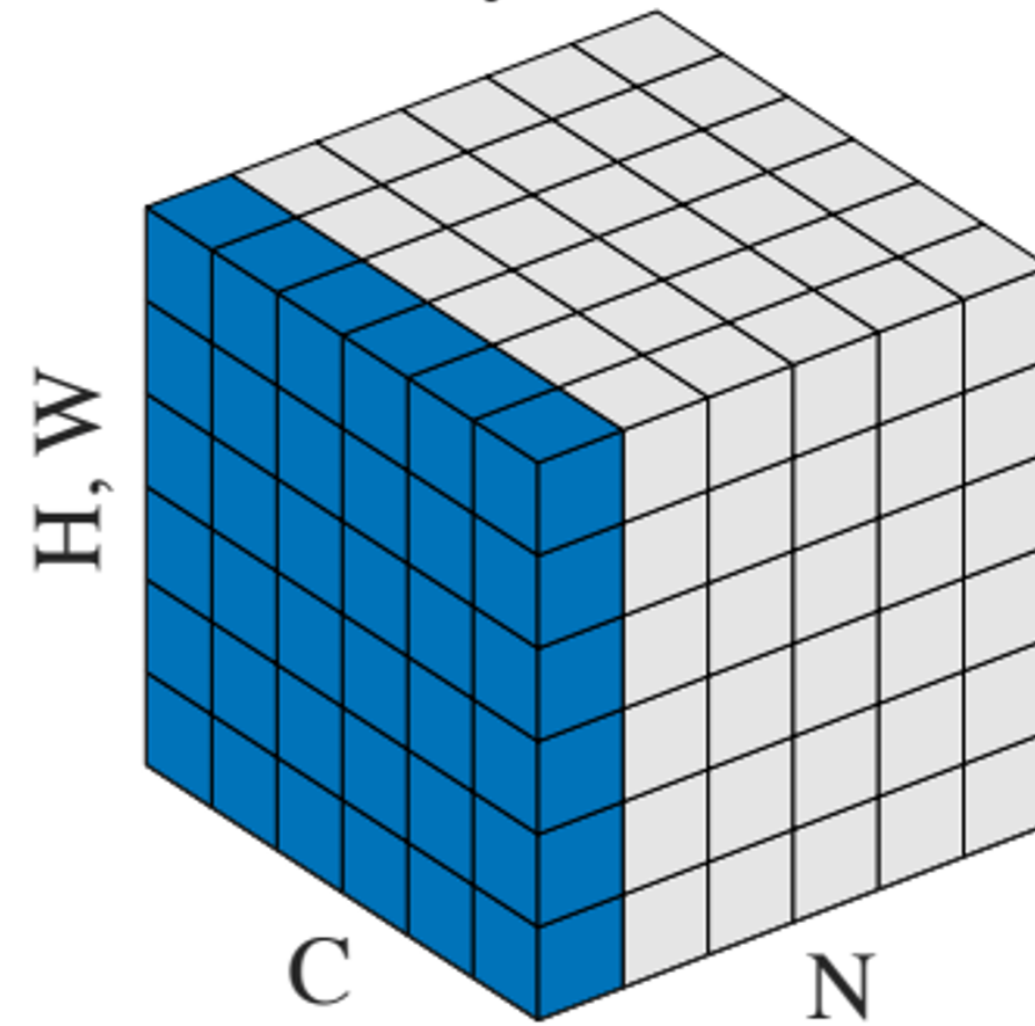


Group Normalization

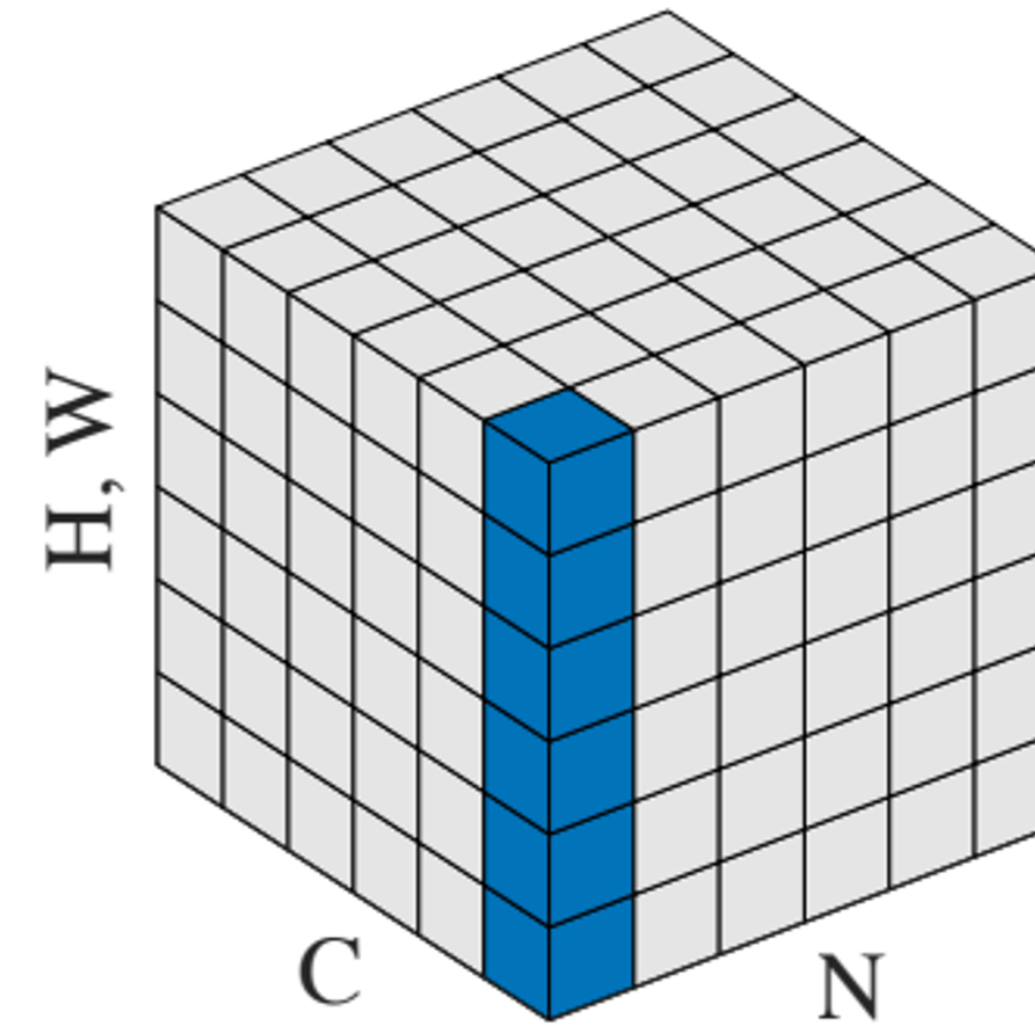
Batch Norm



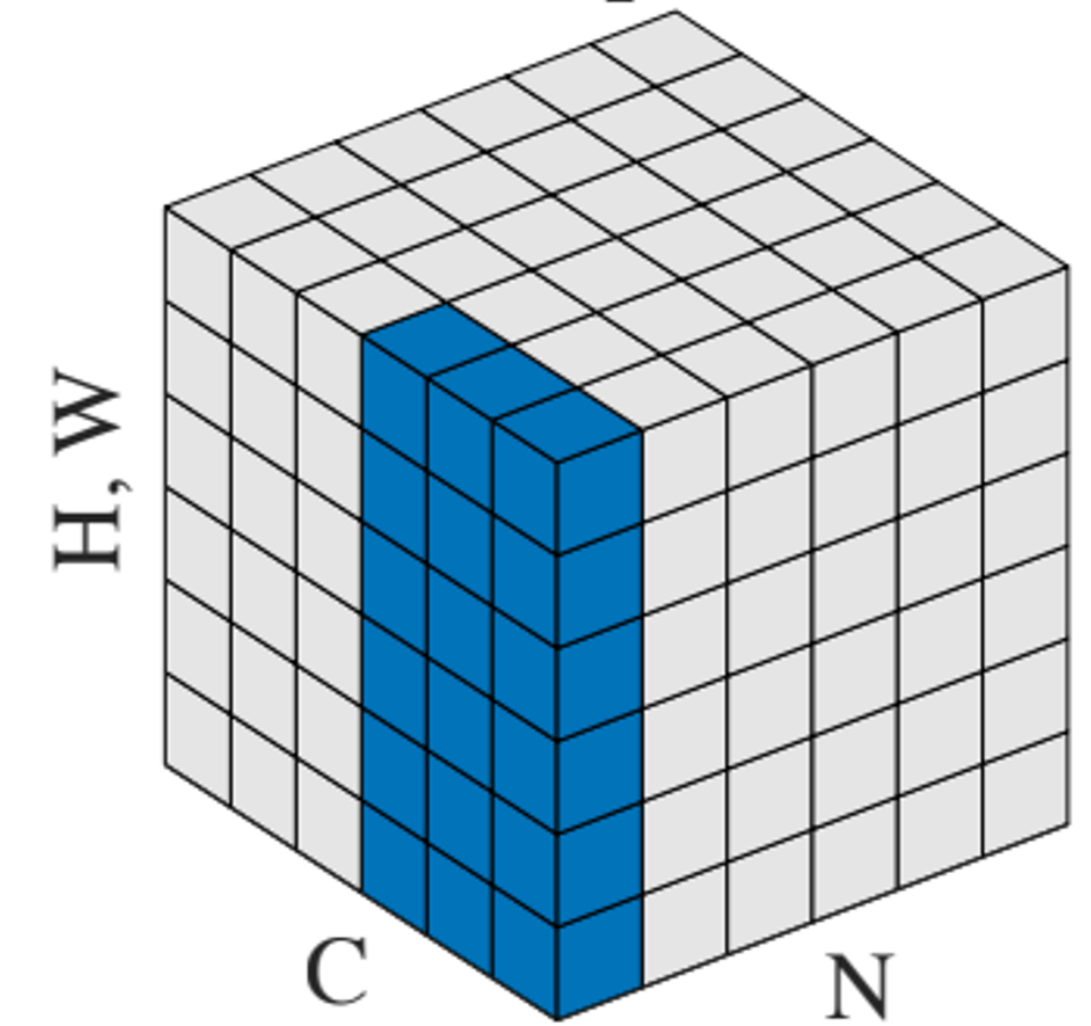
Layer Norm



Instance Norm

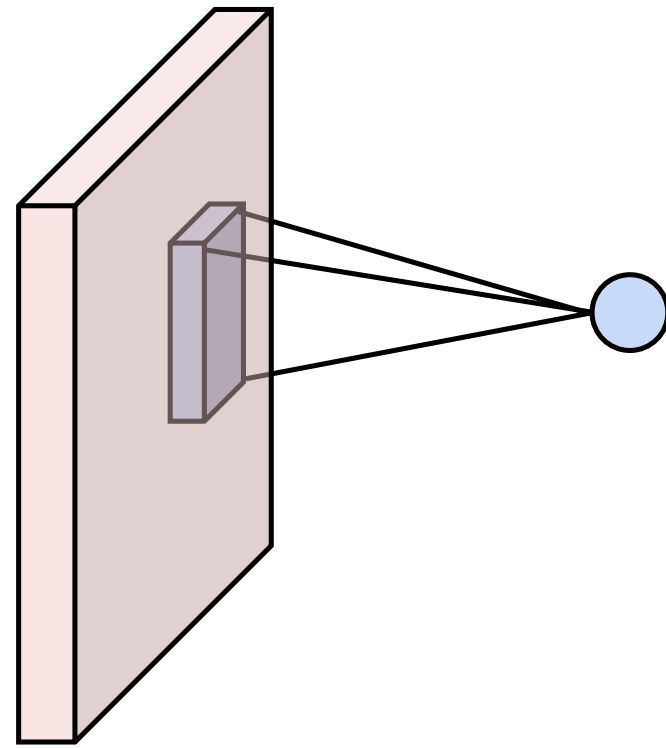


Group Norm

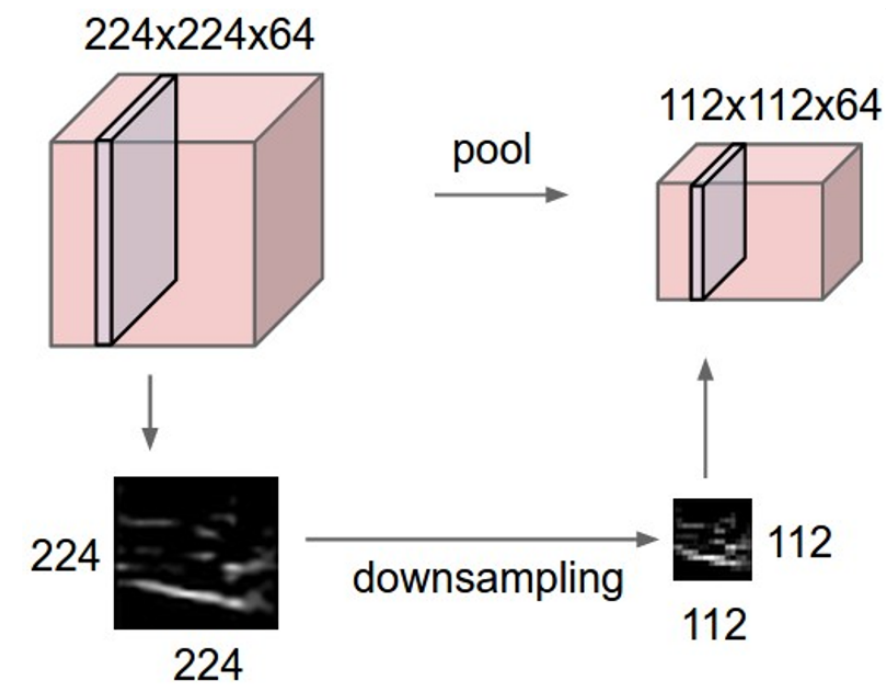


Components of Convolutional Networks

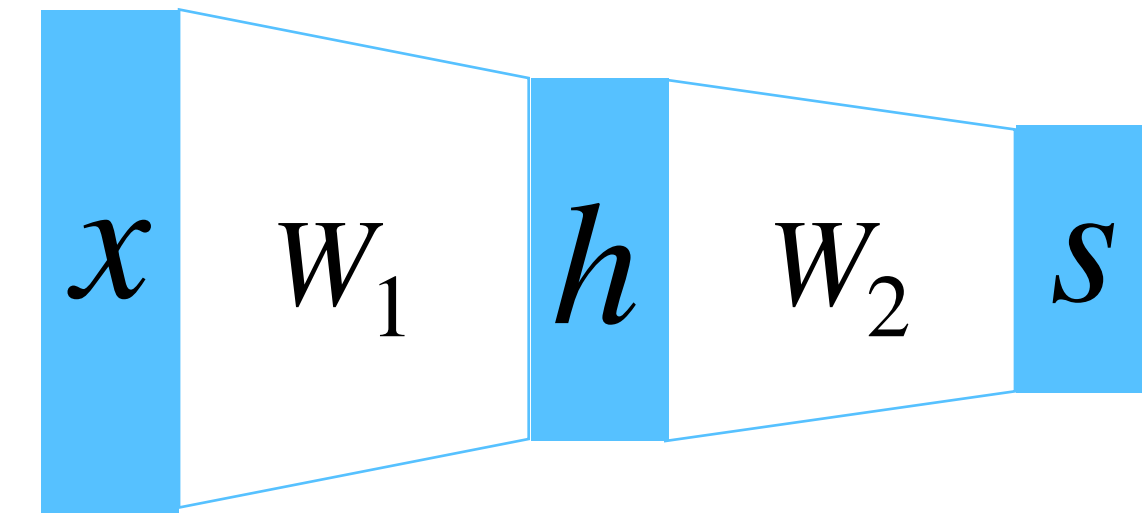
Convolution Layers



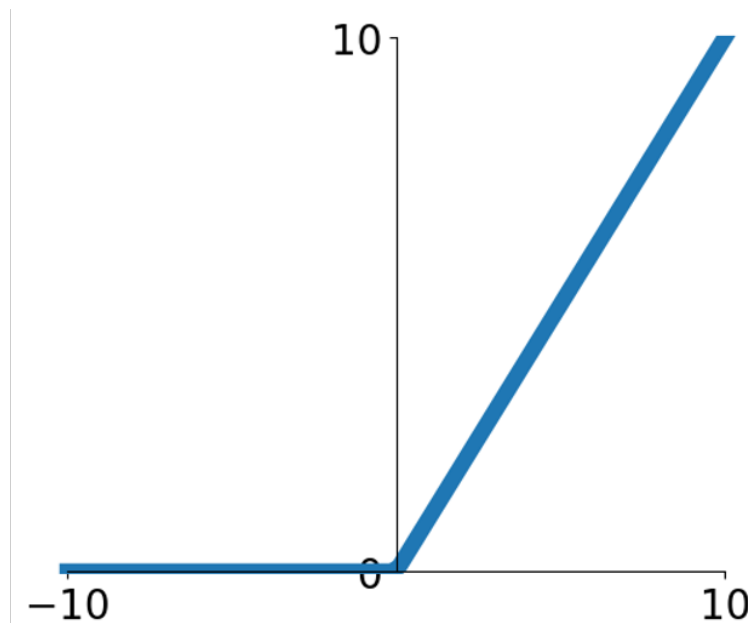
Pooling Layers



Fully-Connected Layers



Activation Function

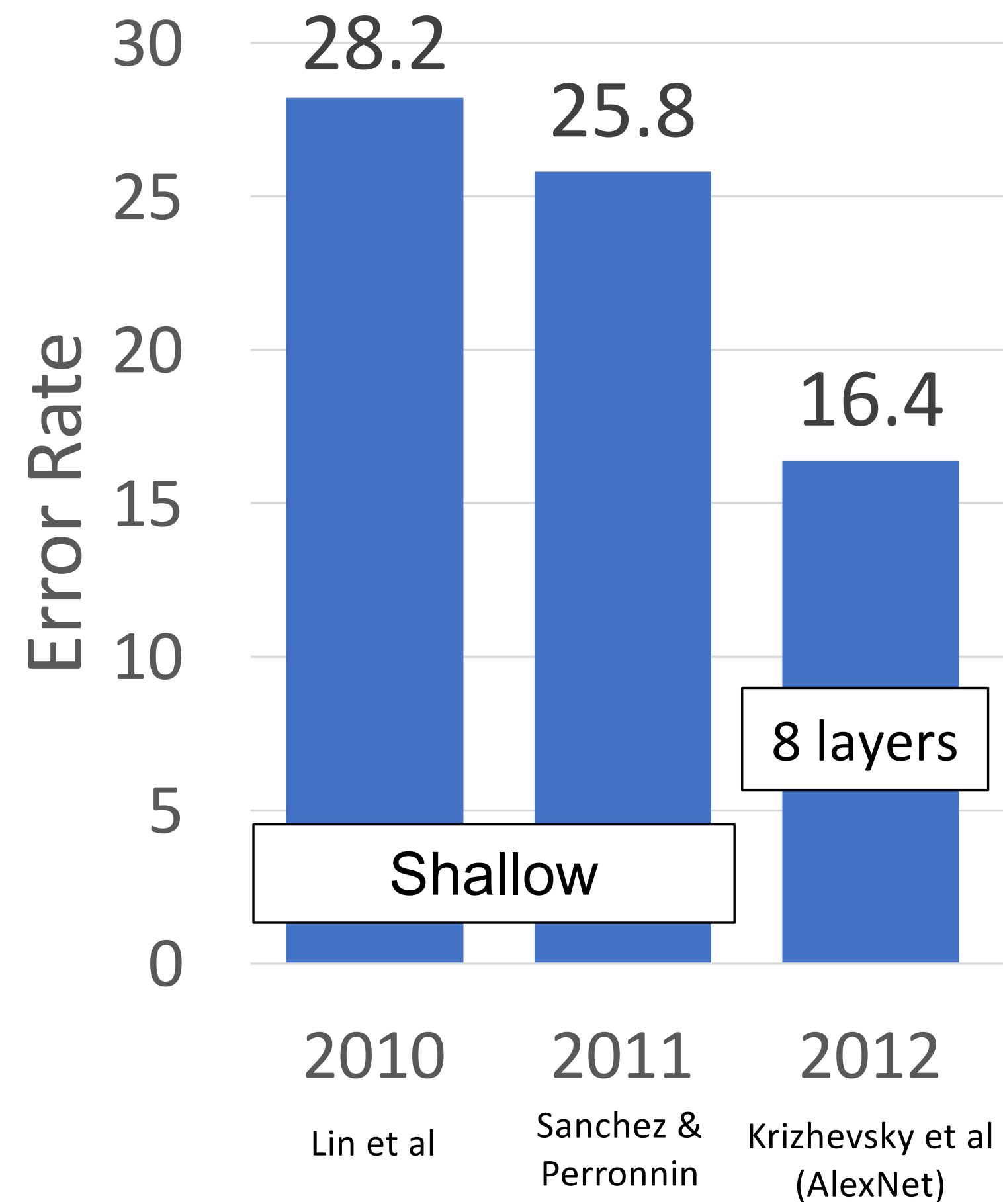


Normalization

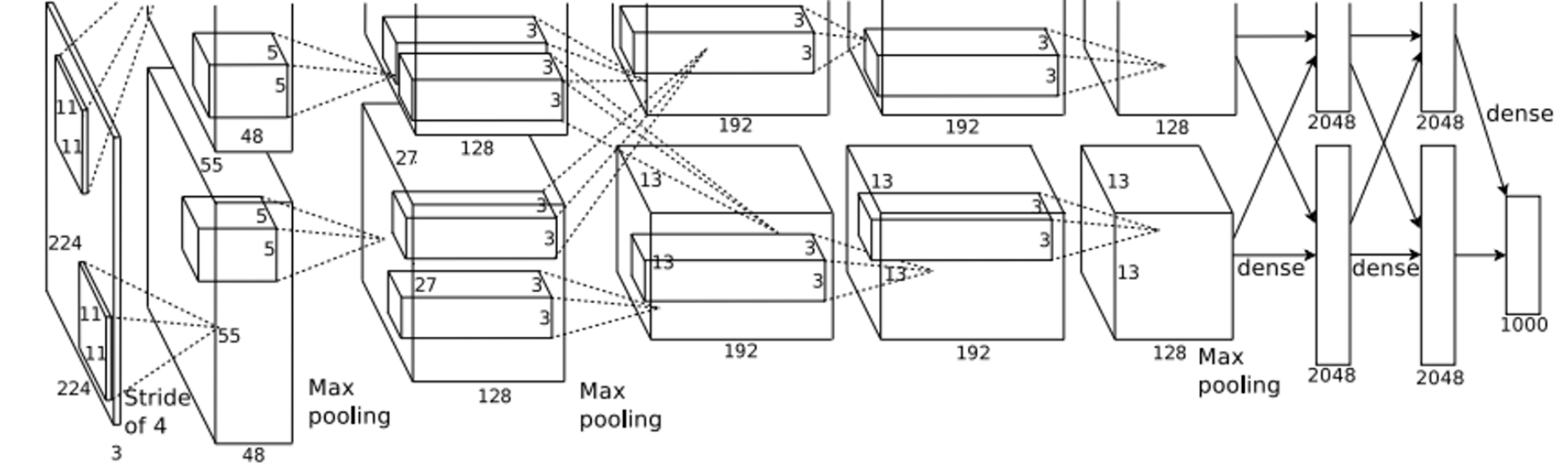
$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

Question: How should we put them together?

ImageNet Classification Challenge



AlexNet



- 227 x 227 inputs
- 5 Convolutional Layers
- Max pooling
- 3 Fully-connected Layers
- ReLU nonlinearities

- Used “Local response normalization”;
Not used anymore
- Trained on two GTX 580 GPUs - only 3GB of memory each! Model split over two GPUs.

AlexNet

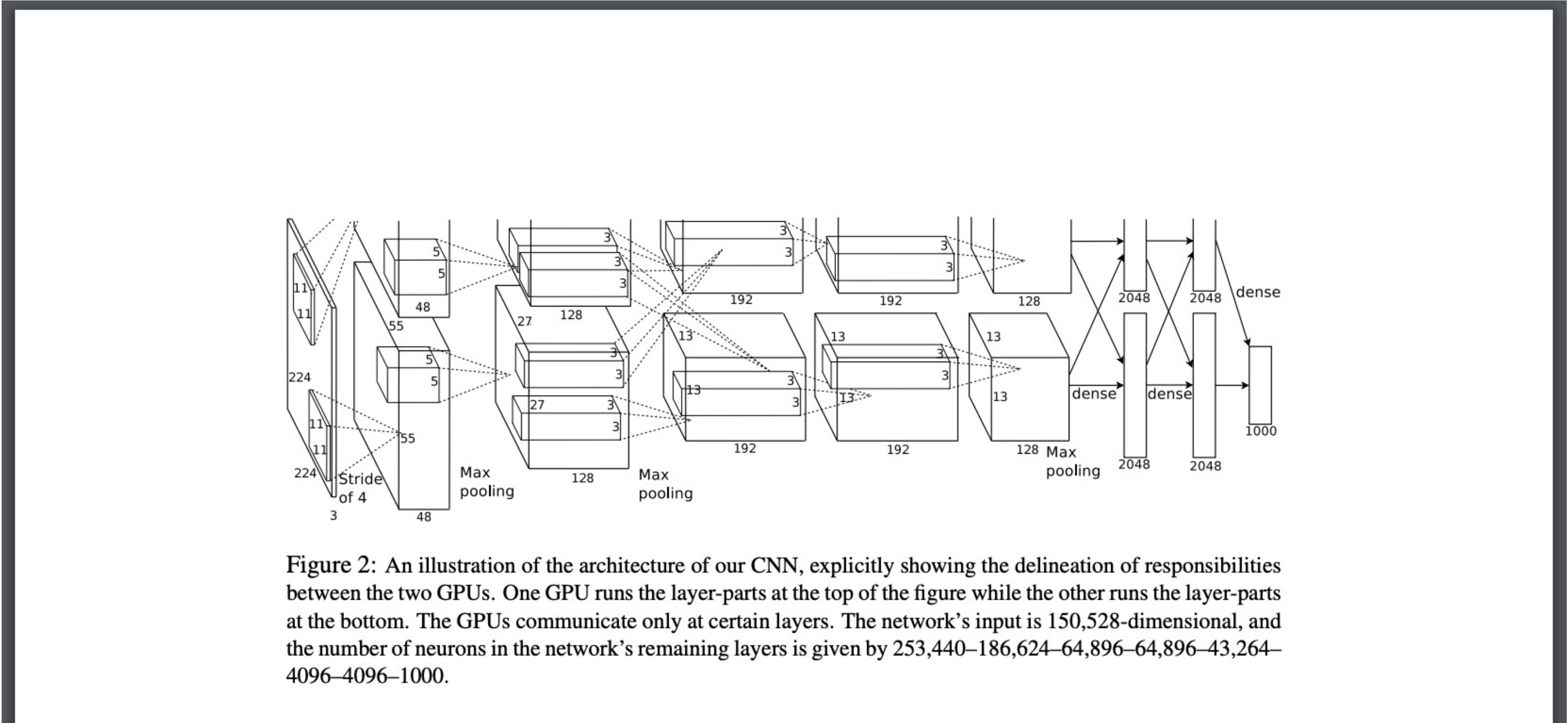
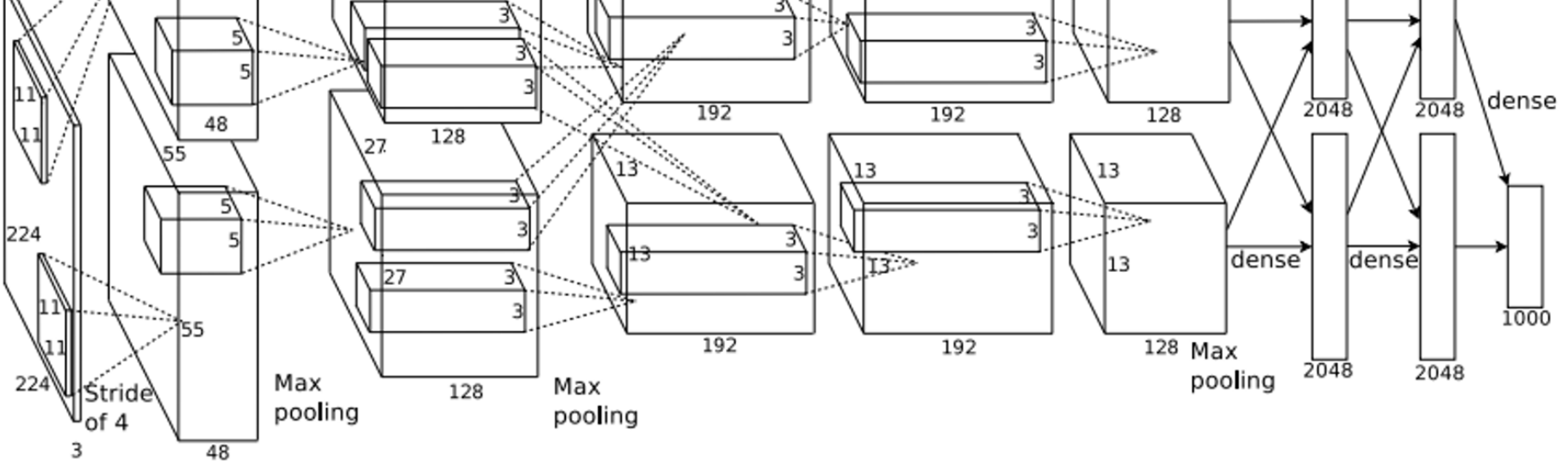
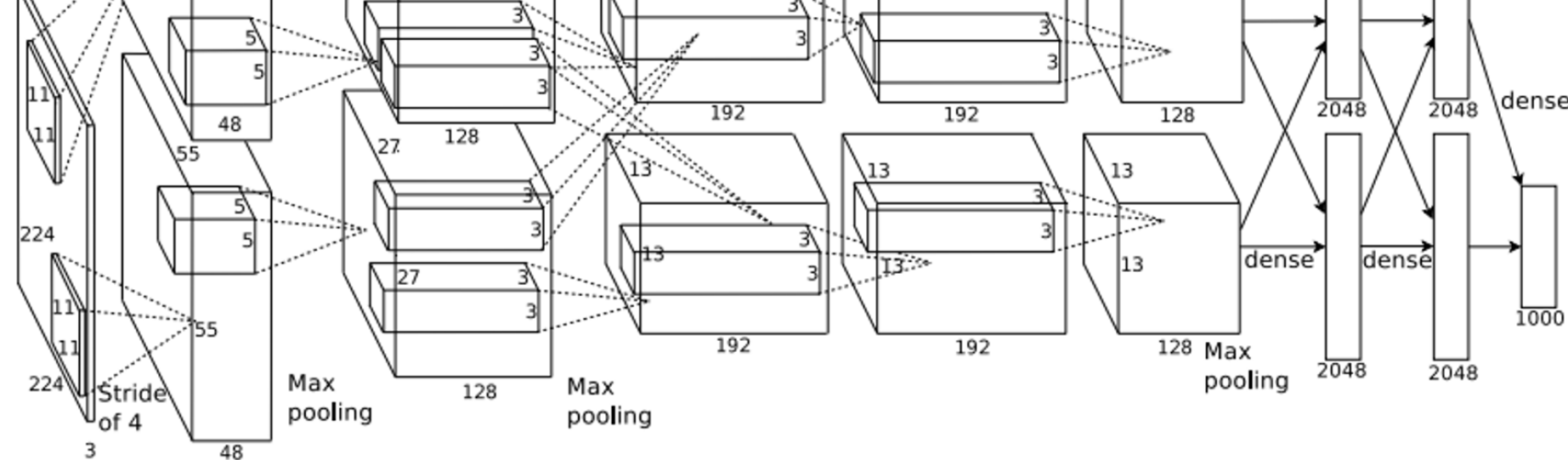


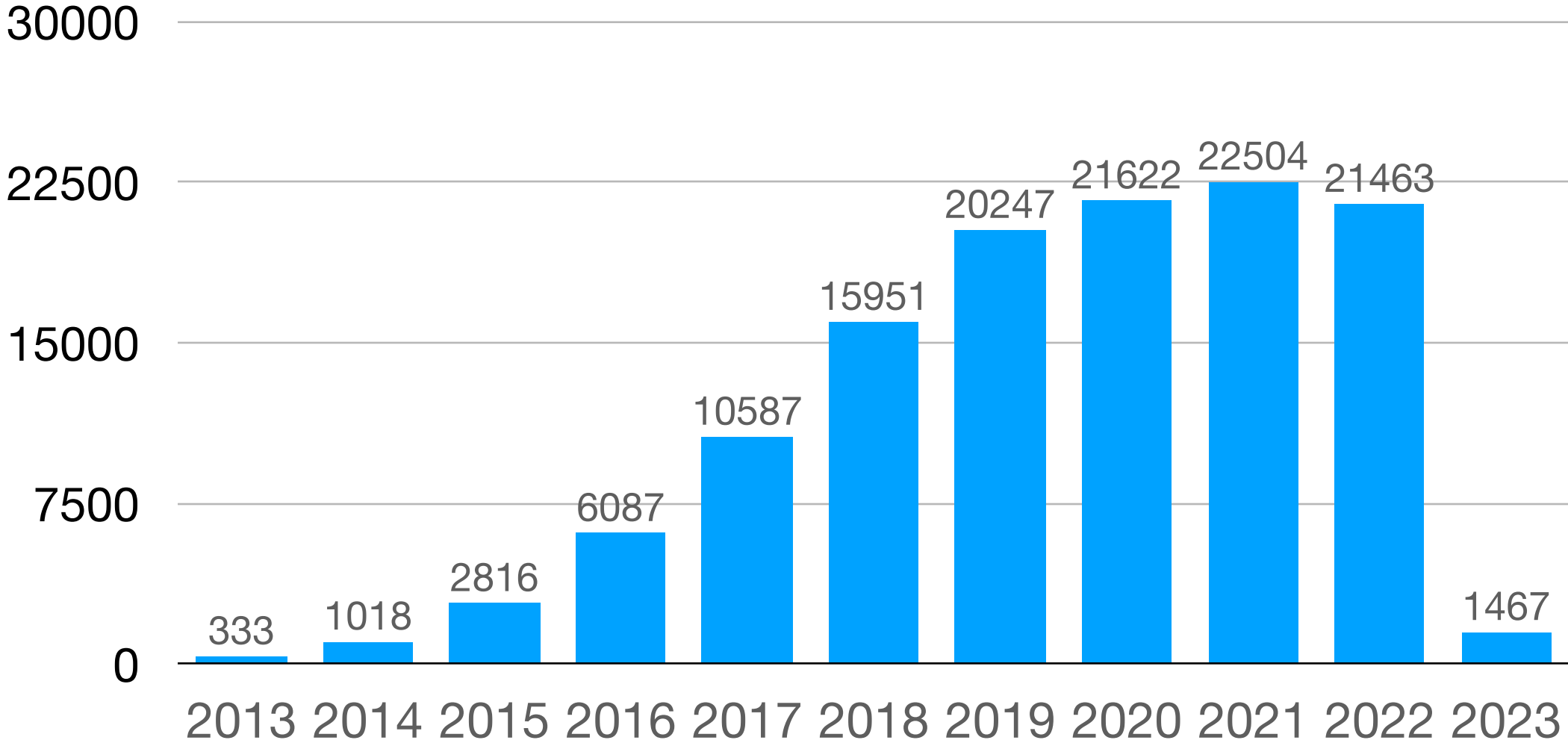
Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input is 150,528-dimensional, and the number of neurons in the network's remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.



AlexNet



AlexNet citations per year (as of 1/31/2023)



Citation Counts:

- Darwin, “On the origin of species”, 1859: **60,117**
- Shannon, “A mathematical theory of communication,” 1948: **140,459**
- Watson and Crick, “Molecular Structure of Nucleic Acids,” 1953: **16,298**

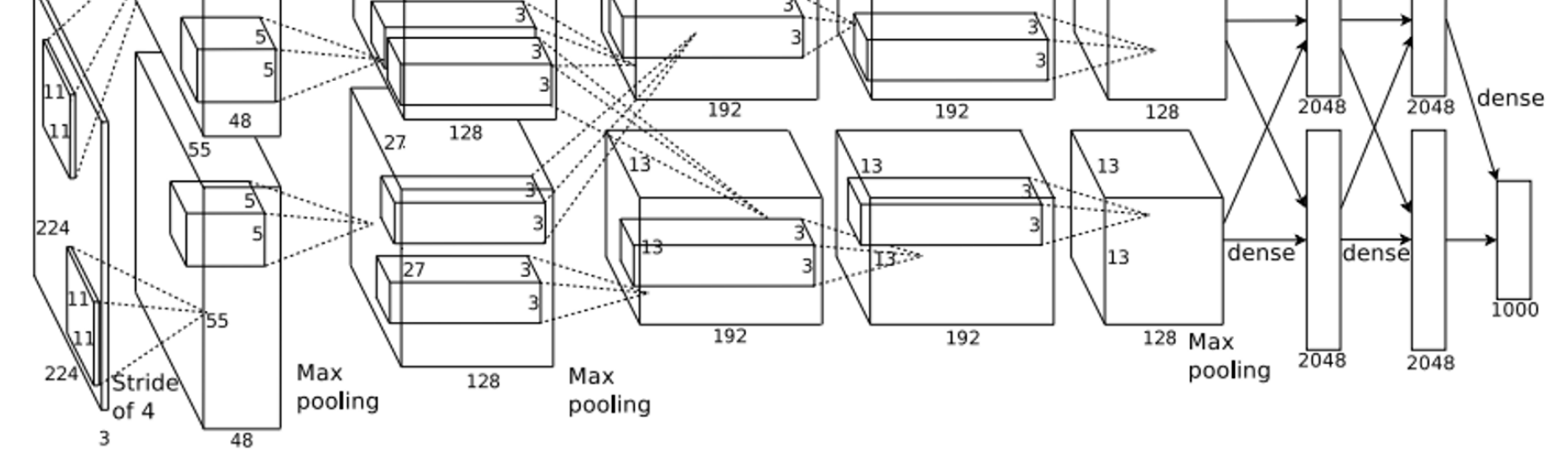
Total citations: **>120,000**



Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012.



AlexNet

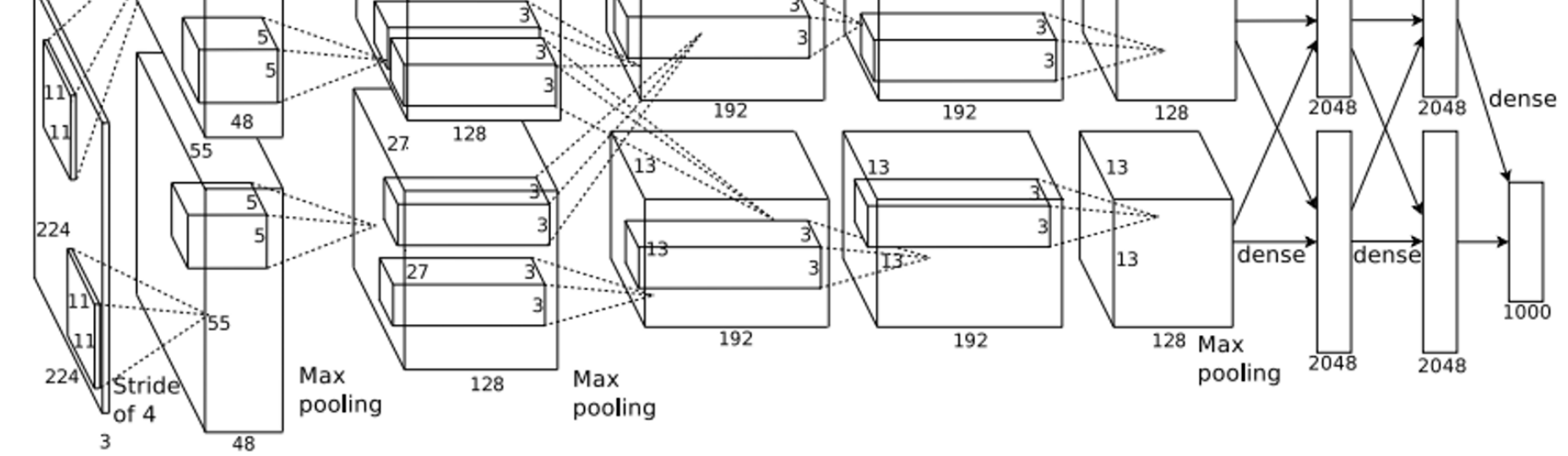


| Layer | Input size | | Layer | | | | Output size | |
|-------|------------|-----|---------|--------|--------|-----|-------------|-----|
| | C | H/W | Filters | Kernel | Stride | Pad | C | H/W |
| Conv1 | 3 | 227 | 64 | 11 | 4 | 2 | ? | |





AlexNet



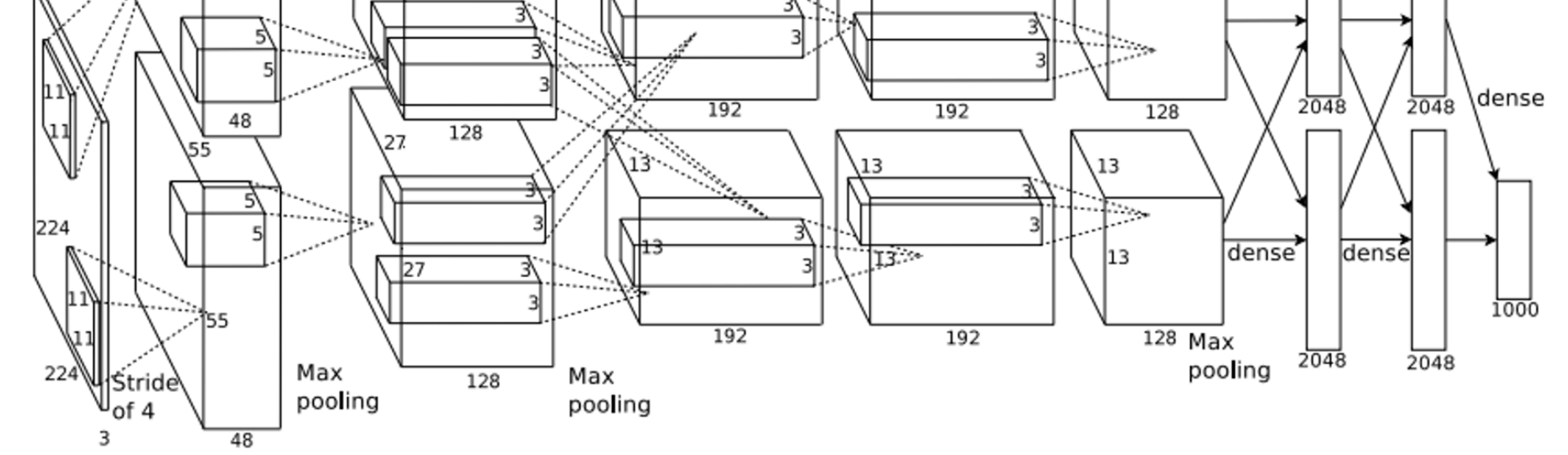
| Layer | Input size | | Layer | | | | Output size | |
|-------|------------|-----|---------|--------|--------|-----|-------------|-----|
| | C | H/W | Filters | Kernel | Stride | Pad | C | H/W |
| Conv1 | 3 | 227 | 64 | 11 | 4 | 2 | 64 | ? |

Recall: Output channels = number of filters





AlexNet



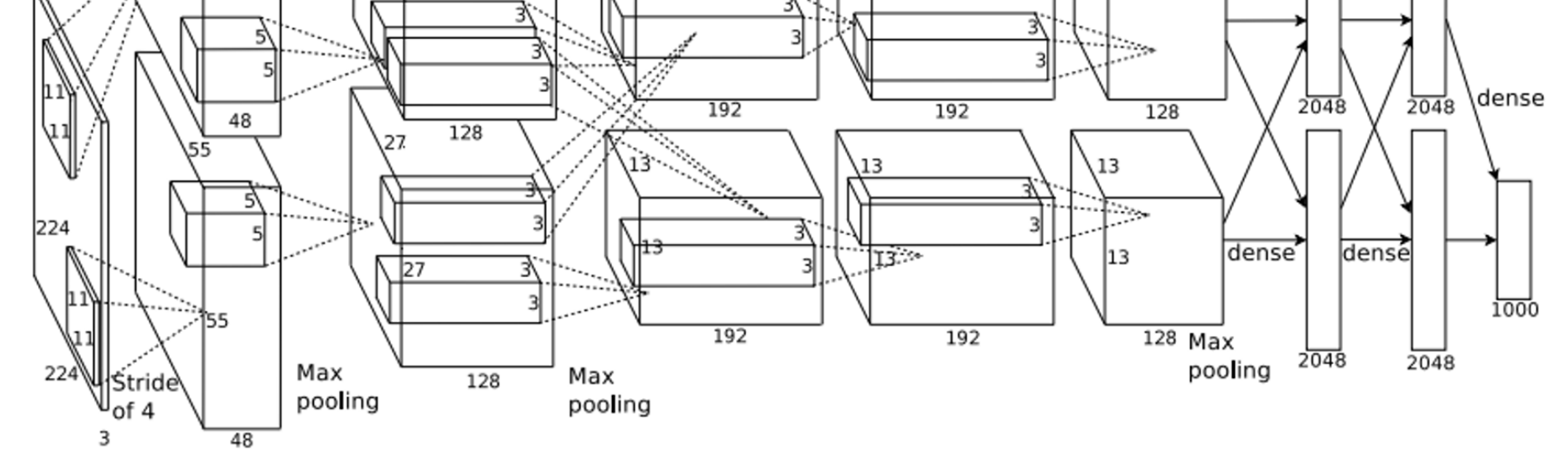
| Layer | Input size | | Layer | | | | Output size | |
|-------|------------|-----|---------|--------|--------|-----|-------------|-----|
| | C | H/W | Filters | Kernel | Stride | Pad | C | H/W |
| Conv1 | 3 | 227 | 64 | 11 | 4 | 2 | 64 | 56 |

$$\begin{aligned}
 \text{Recall: } W' &= (W - K + 2P) / S + 1 \\
 &= (227 - 11 + 2 \times 2) / 4 + 1 \\
 &= 220 / 4 + 1 = 56
 \end{aligned}$$





AlexNet

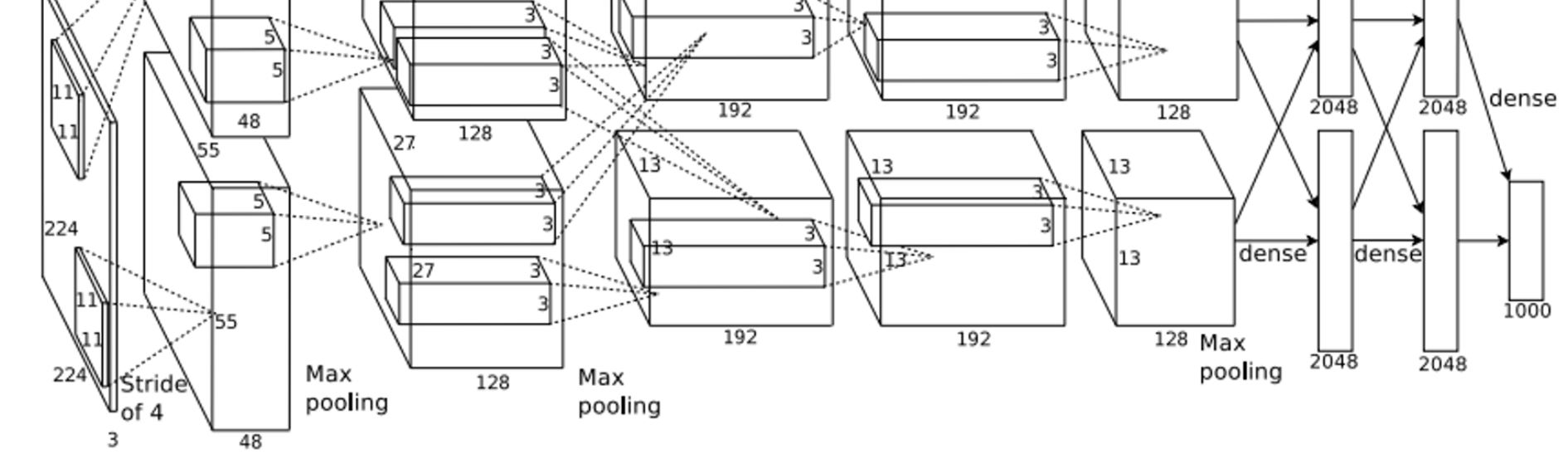


| Layer | Input size | | Layer | | | | Output size | | Memory (KB) |
|-------|------------|-----|---------|--------|--------|-----|-------------|-----|-------------|
| | C | H/W | Filters | Kernel | Stride | Pad | C | H/W | |
| Conv1 | 3 | 227 | 64 | 11 | 4 | 2 | 64 | 56 | ? |





AlexNet



| Layer | Input size | | Layer | | | | Output size | | Memory (KB) |
|-------|------------|-----|---------|--------|--------|-----|-------------|-----|-------------|
| | C | H/W | Filters | Kernel | Stride | Pad | C | H/W | |
| Conv1 | 3 | 227 | 64 | 11 | 4 | 2 | 64 | 56 | 784 |

$$\begin{aligned} \text{Number of output elements} &= C \times H' \times W' \\ &= 64 \times 56 \times 56 = 200,704 \end{aligned}$$

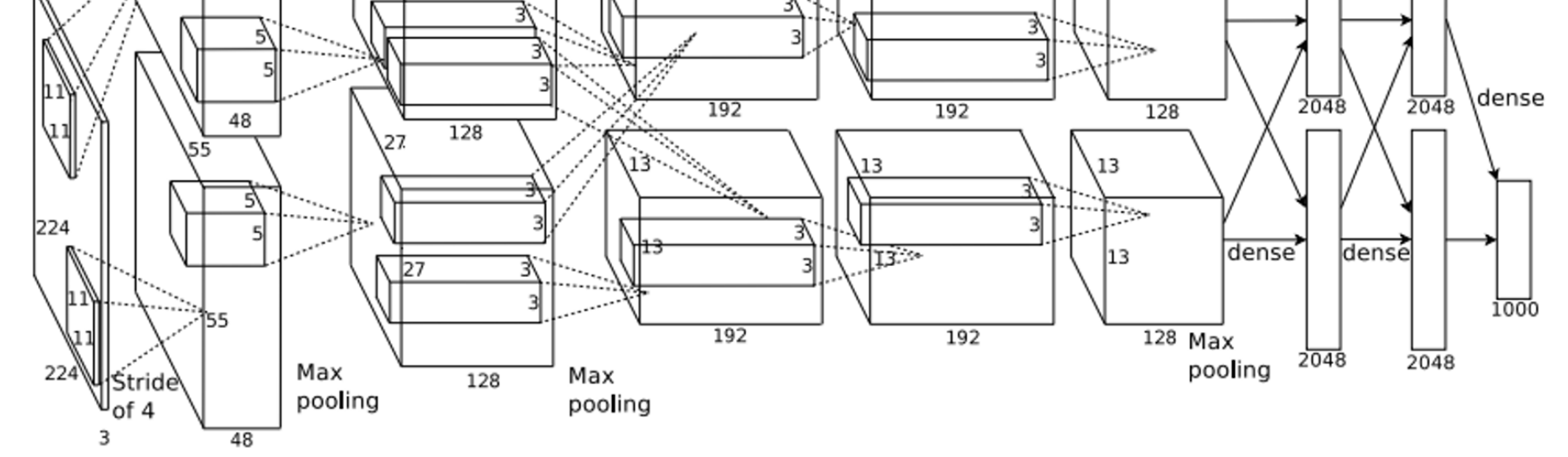
$$\text{Bytes per element} = 4 \text{ (for 32-bit floating point)}$$

$$\begin{aligned} \text{KB} &= (\text{number of elements}) \times (\text{bytes per elem}) / 1024 \\ &= 200704 \times 4 / 1024 \\ &= \mathbf{784} \end{aligned}$$





AlexNet

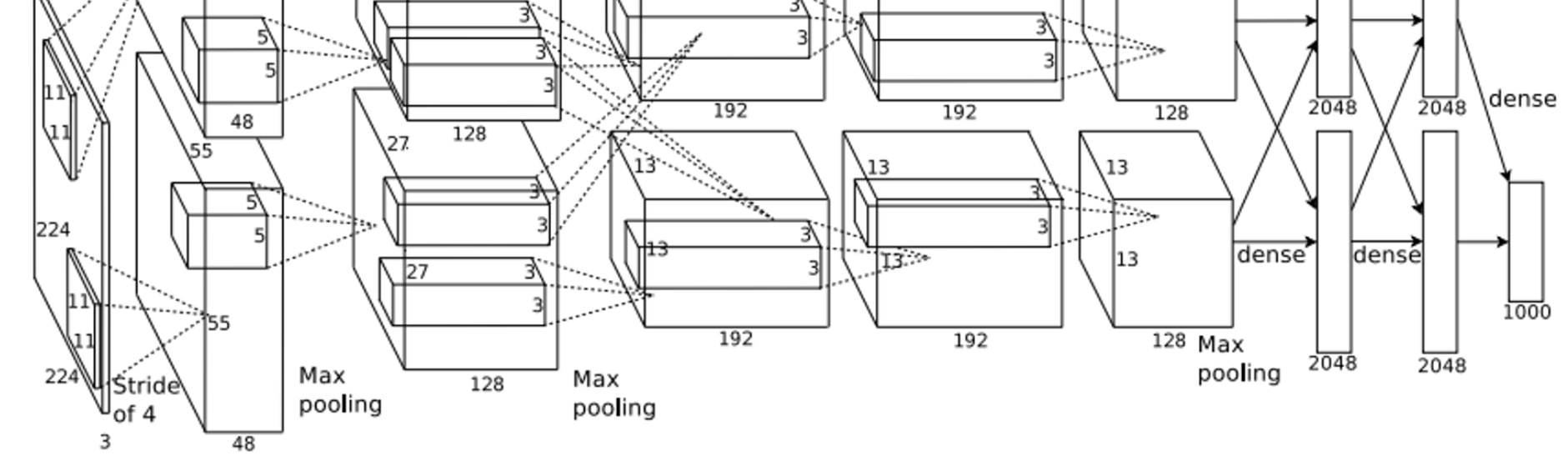


| Layer | Input size | | Layer | | | | Output size | | | |
|-------|------------|-----|---------|--------|--------|-----|-------------|-----|-------------|------------|
| | C | H/W | Filters | Kernel | Stride | Pad | C | H/W | Memory (KB) | Params (k) |
| Conv1 | 3 | 227 | 64 | 11 | 4 | 2 | 64 | 56 | 784 | ? |





AlexNet



| Layer | Input size | | Layer | | | | Output size | | | |
|-------|------------|-----|---------|--------|--------|-----|-------------|-----|-------------|------------|
| | C | H/W | Filters | Kernel | Stride | Pad | C | H/W | Memory (KB) | Params (k) |
| Conv1 | 3 | 227 | 64 | 11 | 4 | 2 | 64 | 56 | 784 | 23 |

$$\begin{aligned} \text{Weight shape} &= C_{\text{out}} \times C_{\text{in}} \times K \times K \\ &= 64 \times 3 \times 11 \times 11 \end{aligned}$$

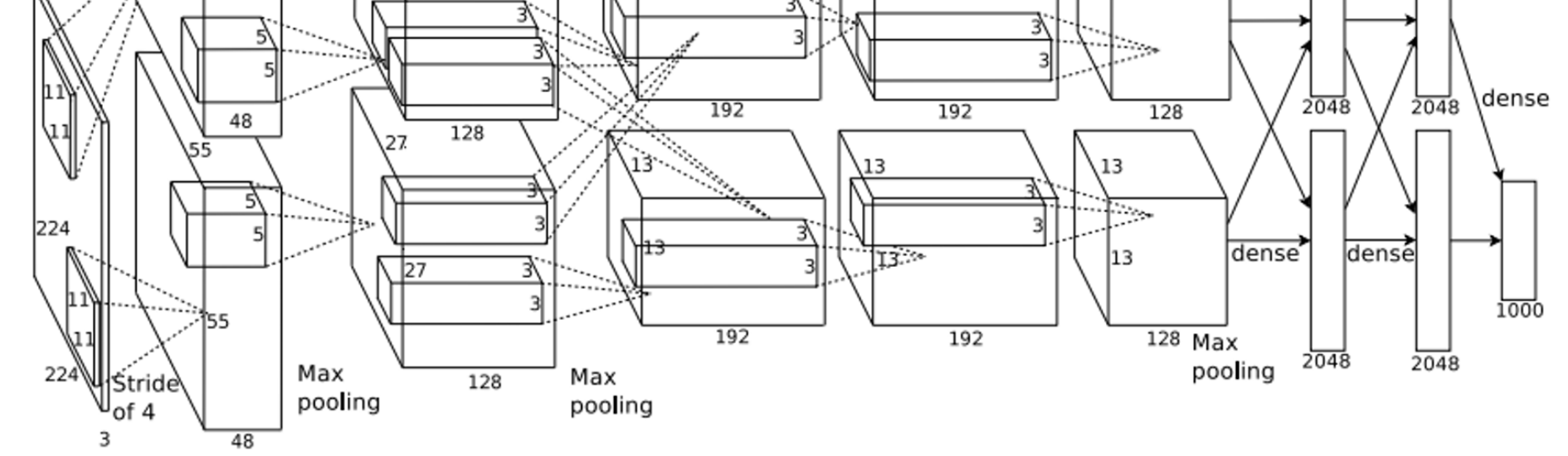
$$\text{Bias shape} = C_{\text{out}} = 64$$

$$\begin{aligned} \text{Number of weights} &= 64 \times 3 \times 11 \times 11 + 64 \\ &= \mathbf{23,296} \end{aligned}$$





AlexNet

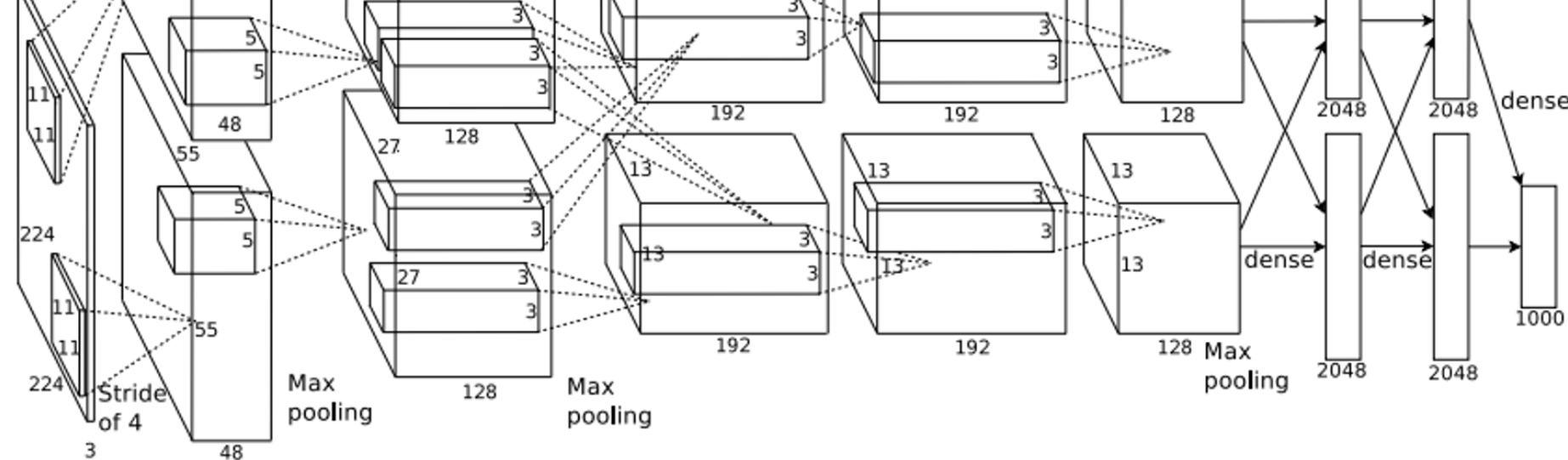


| Layer | Input size | | Layer | | | | Output size | | | | |
|-------|------------|-----|---------|--------|--------|-----|-------------|-----|-------------|------------|----------|
| | C | H/W | Filters | Kernel | Stride | Pad | C | H/W | Memory (KB) | Params (k) | Flop (M) |
| Conv1 | 3 | 227 | 64 | 11 | 4 | 2 | 64 | 56 | 784 | 23 | ? |





AlexNet



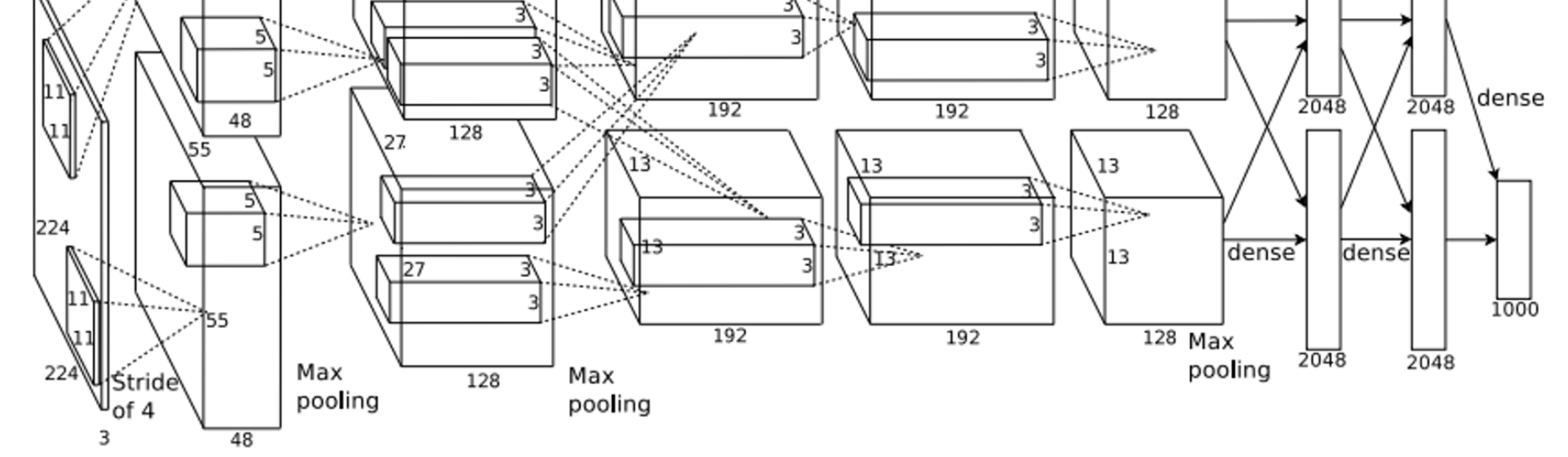
| Layer | Input size | | Layer | | | | Output size | | | | |
|-------|------------|-----|---------|--------|--------|-----|-------------|-----|-------------|------------|----------|
| | C | H/W | Filters | Kernel | Stride | Pad | C | H/W | Memory (KB) | Params (k) | Flop (M) |
| Conv1 | 3 | 227 | 64 | 11 | 4 | 2 | 64 | 56 | 784 | 23 | 73 |

Number of floating point operations (multiply + add)
 = (number of output elements) * (ops per output elem)
 = $(C_{out} \times H' \times W')$ * $(C_{in} \times K \times K)$
 = $(64 * 56 * 56) * (3 * 11 * 11)$
 = $200,704 * 363$
 = **72,855,552**





AlexNet

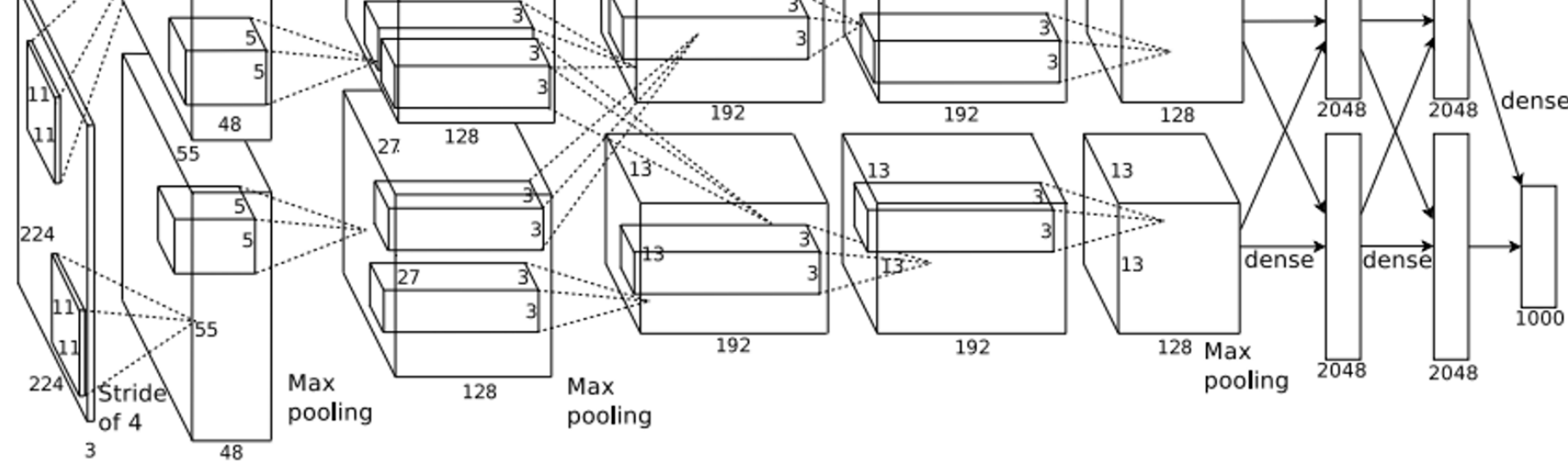


| Layer | Input size | | Layer | | | | Output size | | | | |
|-------|------------|-----|---------|--------|--------|-----|-------------|-----|-------------|------------|----------|
| | C | H/W | Filters | Kernel | Stride | Pad | C | H/W | Memory (KB) | Params (k) | Flop (M) |
| Conv1 | 3 | 227 | 64 | 11 | 4 | 2 | 64 | 56 | 784 | 23 | 73 |
| Pool1 | 64 | 56 | | 3 | 2 | 0 | ? | | | | |





AlexNet



| Layer | Input size | | Layer | | | | Output size | | | | |
|-------|------------|-----|---------|--------|--------|-----|-------------|-----|-------------|------------|----------|
| | C | H/W | Filters | Kernel | Stride | Pad | C | H/W | Memory (KB) | Params (k) | Flop (M) |
| Conv1 | 3 | 227 | 64 | 11 | 4 | 2 | 64 | 56 | 784 | 23 | 73 |
| Pool1 | 64 | 56 | | 3 | 2 | 0 | 64 | 27 | | | |

For pooling layer:

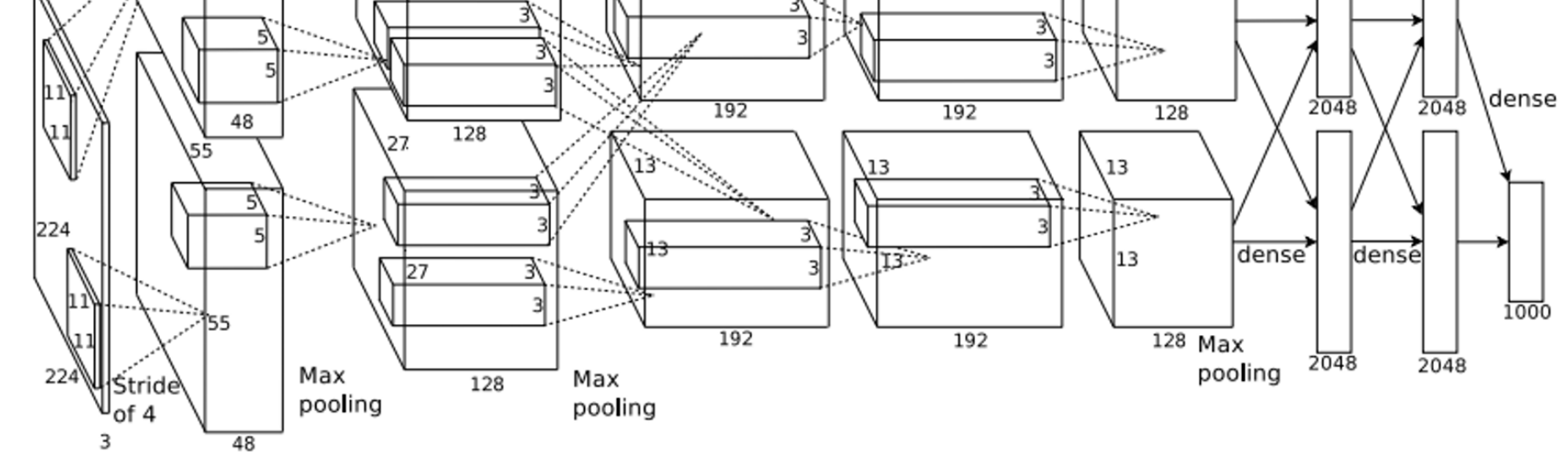
#output channels = #input channels = 64

$$\begin{aligned}
 W' &= \text{floor}((W-K)/S+1) \\
 &= \text{floor}(53/2 + 1) = \text{floor}(27.5) = \mathbf{27}
 \end{aligned}$$





AlexNet



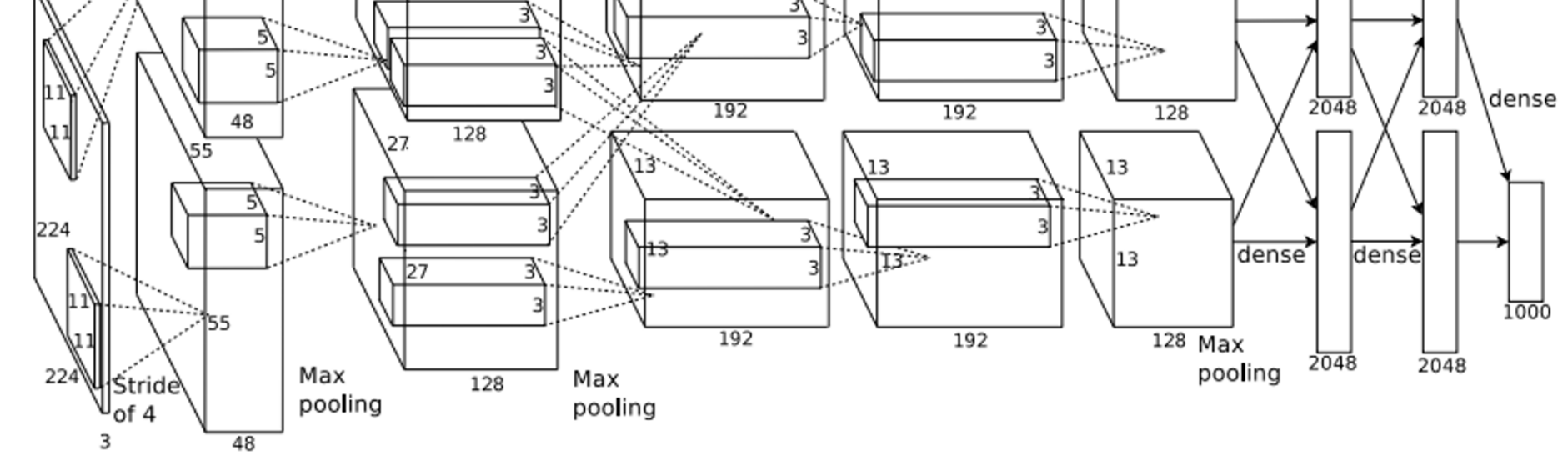
| Layer | Input size | | Layer | | | | Output size | | | | |
|-------|------------|-----|---------|--------|--------|-----|-------------|-----|-------------|------------|----------|
| | C | H/W | Filters | Kernel | Stride | Pad | C | H/W | Memory (KB) | Params (k) | Flop (M) |
| Conv1 | 3 | 227 | 64 | 11 | 4 | 2 | 64 | 56 | 784 | 23 | 73 |
| Pool1 | 64 | 56 | | 3 | 2 | 0 | 64 | 27 | 182 | ? | |

#output elms = $C_{out} \times H' \times W'$
 Bytes per elem = 4
 $KB = C_{out} \times H' \times W' \times 4 / 1024$
 $= 64 * 27 * 27 * 4 / 1024$
 $= \mathbf{182.25}$





AlexNet



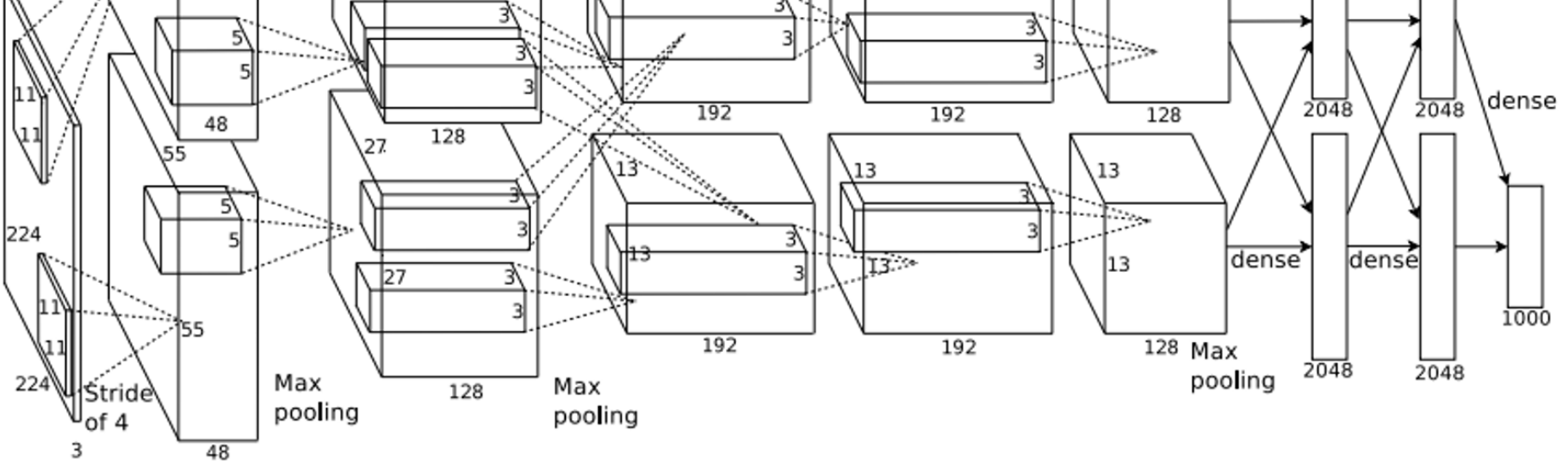
| Layer | Input size | | Layer | | | | Output size | | | | |
|-------|------------|-----|---------|--------|--------|-----|-------------|-----|-------------|------------|----------|
| | C | H/W | Filters | Kernel | Stride | Pad | C | H/W | Memory (KB) | Params (k) | Flop (M) |
| Conv1 | 3 | 227 | 64 | 11 | 4 | 2 | 64 | 56 | 784 | 23 | 73 |
| Pool1 | 64 | 56 | | 3 | 2 | 0 | 64 | 27 | 182 | 0 | 0 |

Pooling layers have no learnable parameters!





AlexNet

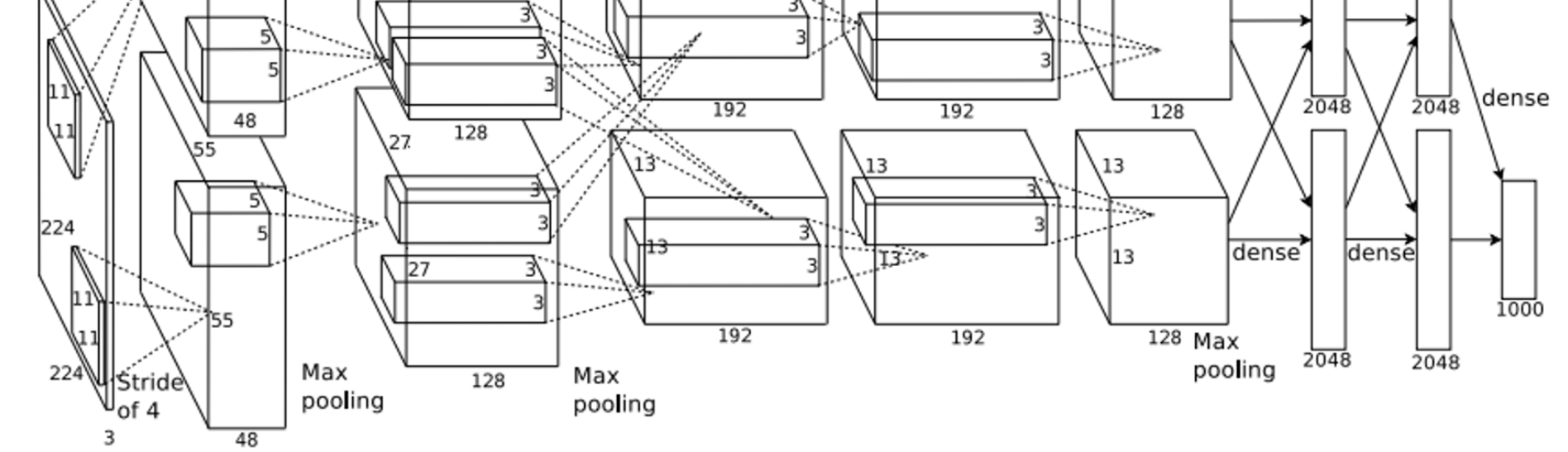


| Layer | Input size | | Layer | | | | Output size | | | | |
|-------|------------|-----|---------|--------|--------|-----|-------------|-----|-------------|------------|----------|
| | C | H/W | Filters | Kernel | Stride | Pad | C | H/W | Memory (KB) | Params (k) | Flop (M) |
| Conv1 | 3 | 227 | 64 | 11 | 4 | 2 | 64 | 56 | 784 | 23 | 73 |
| Pool1 | 64 | 56 | | 3 | 2 | 0 | 64 | 27 | 182 | 0 | 0 |

Floating-point ops for pooling layer
 = (number of output positions) * (flops per output position)
 = (C_{out} x H' x W') x (K x K)
 = (64 * 27 * 27) * (3 * 3)
 = 419,904
 = **0.4 MFLOP**



AlexNet



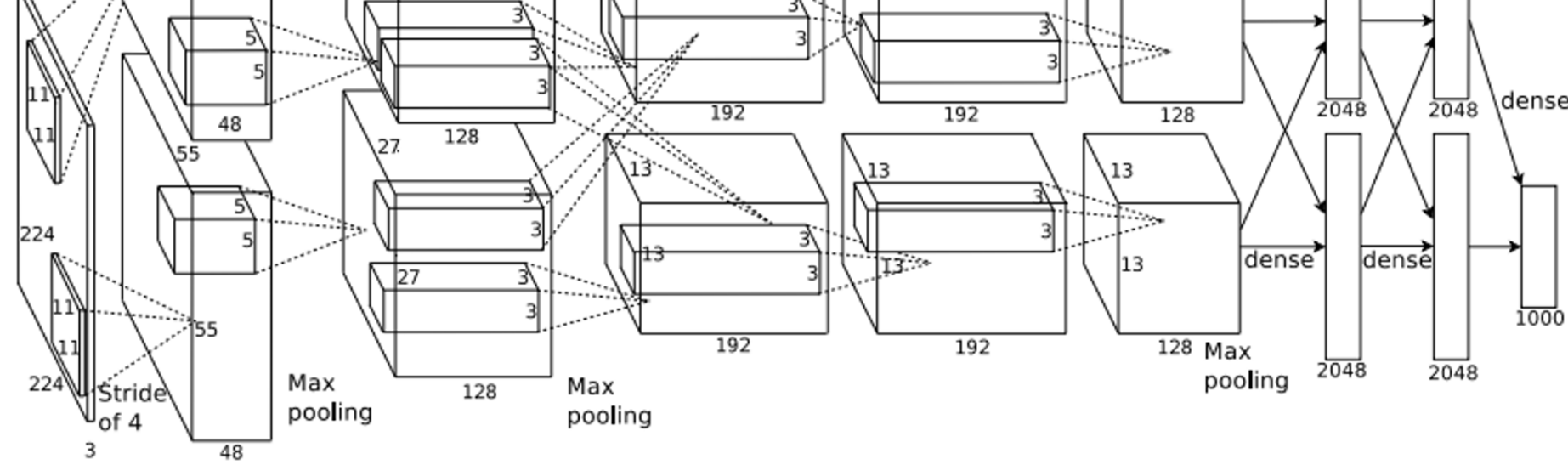
| Layer | Input size | | Layer | | | | Output size | | Memory (KB) | Params (k) | Flop (M) |
|---------|------------|-----|---------|--------|--------|-----|-------------|-----|-------------|------------|----------|
| | C | H/W | Filters | Kernel | Stride | Pad | C | H/W | | | |
| Conv1 | 3 | 227 | 64 | 11 | 4 | 2 | 64 | 56 | 784 | 23 | 73 |
| Pool1 | 64 | 56 | | 3 | 2 | 0 | 64 | 27 | 182 | 0 | 0 |
| Conv2 | 64 | 27 | 192 | 5 | 1 | 2 | 192 | 27 | 547 | 307 | 224 |
| Pool2 | 192 | 27 | | 3 | 2 | 0 | 192 | 13 | 127 | 0 | 0 |
| Conv3 | 192 | 13 | 384 | 3 | 1 | 1 | 384 | 13 | 254 | 664 | 112 |
| Conv4 | 384 | 13 | 256 | 3 | 1 | 1 | 256 | 13 | 169 | 885 | 145 |
| Conv5 | 256 | 13 | 256 | 3 | 1 | 1 | 256 | 13 | 169 | 590 | 100 |
| Pool5 | 256 | 13 | | 3 | 2 | 0 | 256 | 6 | 36 | 0 | 0 |
| Flatten | 256 | 6 | | | | | 9216 | | 36 | 0 | 0 |

$$\begin{aligned}
 \text{Flatten output size} &= C_{in} \times H \times W \\
 &= 256 * 6 * 6 \\
 &= \mathbf{9216}
 \end{aligned}$$





AlexNet



| Layer | Input size | | Layer | | | | Output size | | Memory (KB) | Params (k) | Flop (M) |
|---------|------------|-----|---------|--------|--------|-----|-------------|-----|-------------|------------|----------|
| | C | H/W | Filters | Kernel | Stride | Pad | C | H/W | | | |
| Conv1 | 3 | 227 | 64 | 11 | 4 | 2 | 64 | 56 | 784 | 23 | 73 |
| Pool1 | 64 | 56 | | 3 | 2 | 0 | 64 | 27 | 182 | 0 | 0 |
| Conv2 | 64 | 27 | 192 | 5 | 1 | 2 | 192 | 27 | 547 | 307 | 224 |
| Pool2 | 192 | 27 | | 3 | 2 | 0 | 192 | 13 | 127 | 0 | 0 |
| Conv3 | 192 | 13 | 384 | 3 | 1 | 1 | 384 | 13 | 254 | 664 | 112 |
| Conv4 | 384 | 13 | 256 | 3 | 1 | 1 | 256 | 13 | 169 | 885 | 145 |
| Conv5 | 256 | 13 | 256 | 3 | 1 | 1 | 256 | 13 | 169 | 590 | 100 |
| Pool5 | 256 | 13 | | 3 | 2 | 0 | 256 | 6 | 36 | 0 | 0 |
| Flatten | 256 | 6 | | | | | 9216 | | 36 | 0 | 0 |
| FC6 | 9216 | | 4096 | | | | 4096 | | 16 | 37749 | 38 |

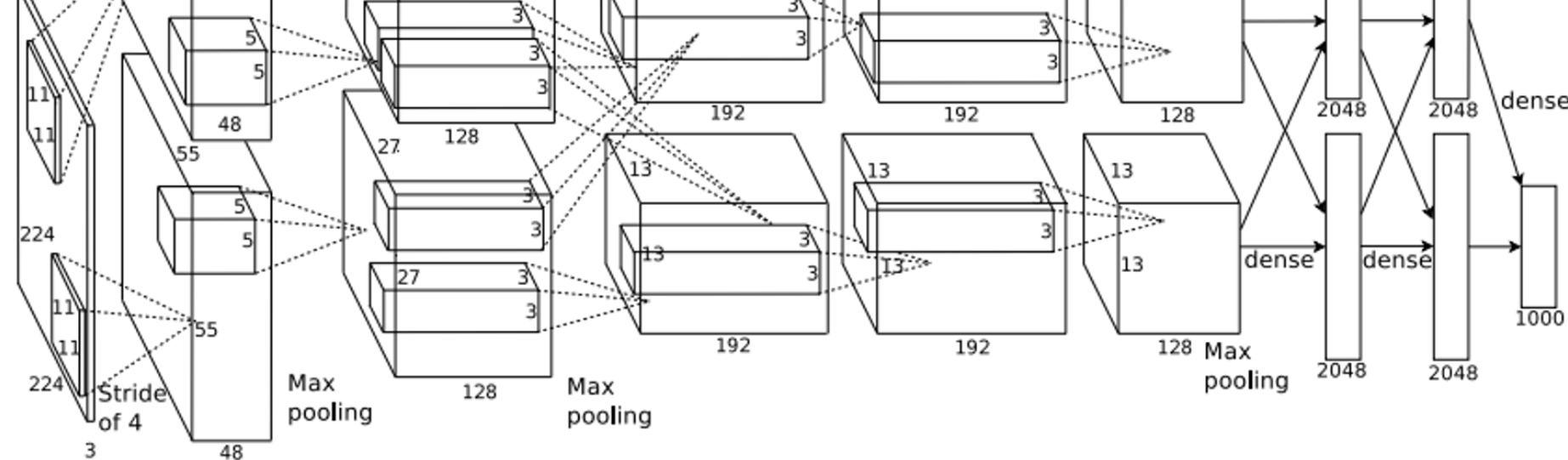
$$\begin{aligned}
 \text{FC params} &= C_{in} * C_{out} + C_{out} \\
 &= 9216 * 4096 + 4096 \\
 &= 37,725,832
 \end{aligned}$$

$$\begin{aligned}
 \text{FC flops} &= C_{in} * C_{out} \\
 &= 9216 * 4096 \\
 &= 37,748,736
 \end{aligned}$$





AlexNet

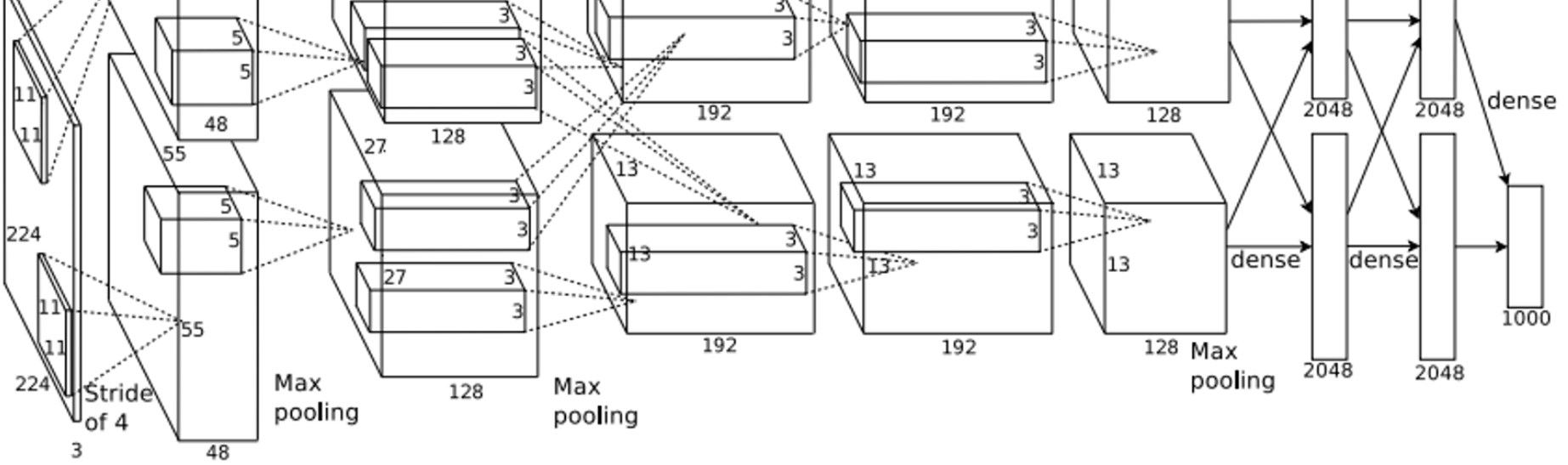


| | Input size | | Layer | | | | Output size | | | | |
|---------|------------|-----|---------|--------|--------|-----|-------------|-----|-------------|------------|----------|
| Layer | C | H/W | Filters | Kernel | Stride | Pad | C | H/W | Memory (KB) | Params (k) | Flop (M) |
| Conv1 | 3 | 227 | 64 | 11 | 4 | 2 | 64 | 56 | 784 | 23 | 73 |
| Pool1 | 64 | 56 | | 3 | 2 | 0 | 64 | 27 | 182 | 0 | 0 |
| Conv2 | 64 | 27 | 192 | 5 | 1 | 2 | 192 | 27 | 547 | 307 | 224 |
| Pool2 | 192 | 27 | | 3 | 2 | 0 | 192 | 13 | 127 | 0 | 0 |
| Conv3 | 192 | 13 | 384 | 3 | 1 | 1 | 384 | 13 | 254 | 664 | 112 |
| Conv4 | 384 | 13 | 256 | 3 | 1 | 1 | 256 | 13 | 169 | 885 | 145 |
| Conv5 | 256 | 13 | 256 | 3 | 1 | 1 | 256 | 13 | 169 | 590 | 100 |
| Pool5 | 256 | 13 | | 3 | 2 | 0 | 256 | 6 | 36 | 0 | 0 |
| Flatten | 256 | 6 | | | | | 9216 | | 36 | 0 | 0 |
| FC6 | 9216 | | 4096 | | | | 4096 | | 16 | 37749 | 38 |
| FC7 | 4096 | | 4096 | | | | 4096 | | 16 | 16777 | 17 |
| FC8 | 4096 | | 1000 | | | | 1000 | | 4 | 4096 | 4 |





AlexNet



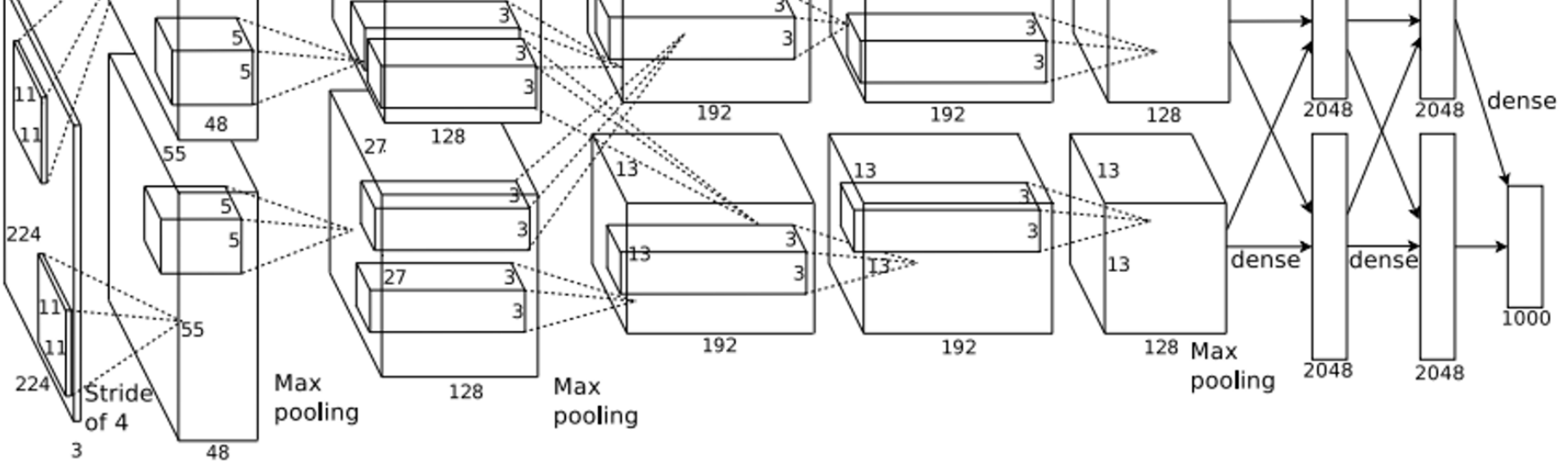
How to choose this? Trial and error :(

| Layer | Input size | | Layer | | | | Output size | | Memory (KB) | Params (k) | Flop (M) |
|---------|------------|-----|---------|--------|--------|-----|-------------|-----|-------------|------------|----------|
| | C | H/W | Filters | Kernel | Stride | Pad | C | H/W | | | |
| Conv1 | 3 | 227 | 64 | 11 | 4 | 2 | 64 | 56 | 784 | 23 | 73 |
| Pool1 | 64 | 56 | | 3 | 2 | 0 | 64 | 27 | 182 | 0 | 0 |
| Conv2 | 64 | 27 | 192 | 5 | 1 | 2 | 192 | 27 | 547 | 307 | 224 |
| Pool2 | 192 | 27 | | 3 | 2 | 0 | 192 | 13 | 127 | 0 | 0 |
| Conv3 | 192 | 13 | 384 | 3 | 1 | 1 | 384 | 13 | 254 | 664 | 112 |
| Conv4 | 384 | 13 | 256 | 3 | 1 | 1 | 256 | 13 | 169 | 885 | 145 |
| Conv5 | 256 | 13 | 256 | 3 | 1 | 1 | 256 | 13 | 169 | 590 | 100 |
| Pool5 | 256 | 13 | | 3 | 2 | 0 | 256 | 6 | 36 | 0 | 0 |
| Flatten | 256 | 6 | | | | | 9216 | | 36 | 0 | 0 |
| FC6 | 9216 | | 4096 | | | | 4096 | | 16 | 37749 | 38 |
| FC7 | 4096 | | 4096 | | | | 4096 | | 16 | 16777 | 17 |
| FC8 | 4096 | | 1000 | | | | 1000 | | 4 | 4096 | 4 |





AlexNet



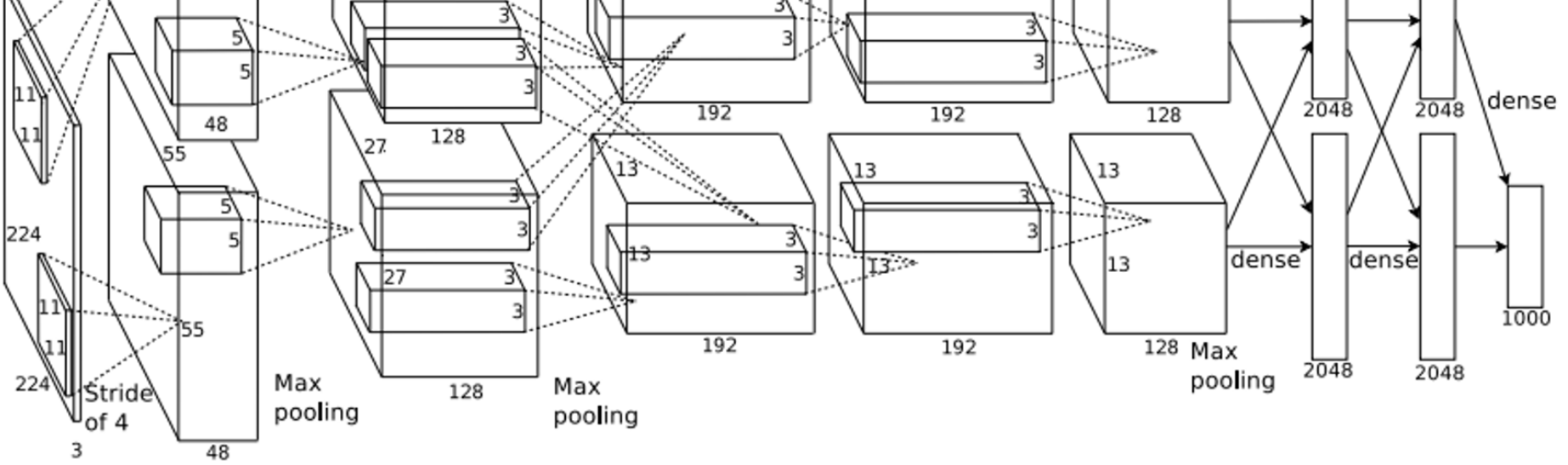
| Layer | Input size | | Layer | | | | Output size | | Memory (KB) | Params (k) | Flop (M) |
|---------|------------|-----|---------|--------|--------|-----|-------------|-----|-------------|------------|----------|
| | C | H/W | Filters | Kernel | Stride | Pad | C | H/W | | | |
| Conv1 | 3 | 227 | 64 | 11 | 4 | 2 | 64 | 56 | 784 | 23 | 73 |
| Pool1 | 64 | 56 | | 3 | 2 | 0 | 64 | 27 | 182 | 0 | 0 |
| Conv2 | 64 | 27 | 192 | 5 | 1 | 2 | 192 | 27 | 547 | 307 | 224 |
| Pool2 | 192 | 27 | | 3 | 2 | 0 | 192 | 13 | 127 | 0 | 0 |
| Conv3 | 192 | 13 | 384 | 3 | 1 | 1 | 384 | 13 | 254 | 664 | 112 |
| Conv4 | 384 | 13 | 256 | 3 | 1 | 1 | 256 | 13 | 169 | 885 | 145 |
| Conv5 | 256 | 13 | 256 | 3 | 1 | 1 | 256 | 13 | 169 | 590 | 100 |
| Pool5 | 256 | 13 | | 3 | 2 | 0 | 256 | 6 | 36 | 0 | 0 |
| Flatten | 256 | 6 | | | | | 9216 | | 36 | 0 | 0 |
| FC6 | 9216 | | 4096 | | | | 4096 | | 16 | 37749 | 38 |
| FC7 | 4096 | | 4096 | | | | 4096 | | 16 | 16777 | 17 |
| FC8 | 4096 | | 1000 | | | | 1000 | | 4 | 4096 | 4 |

Interesting trends here!





AlexNet

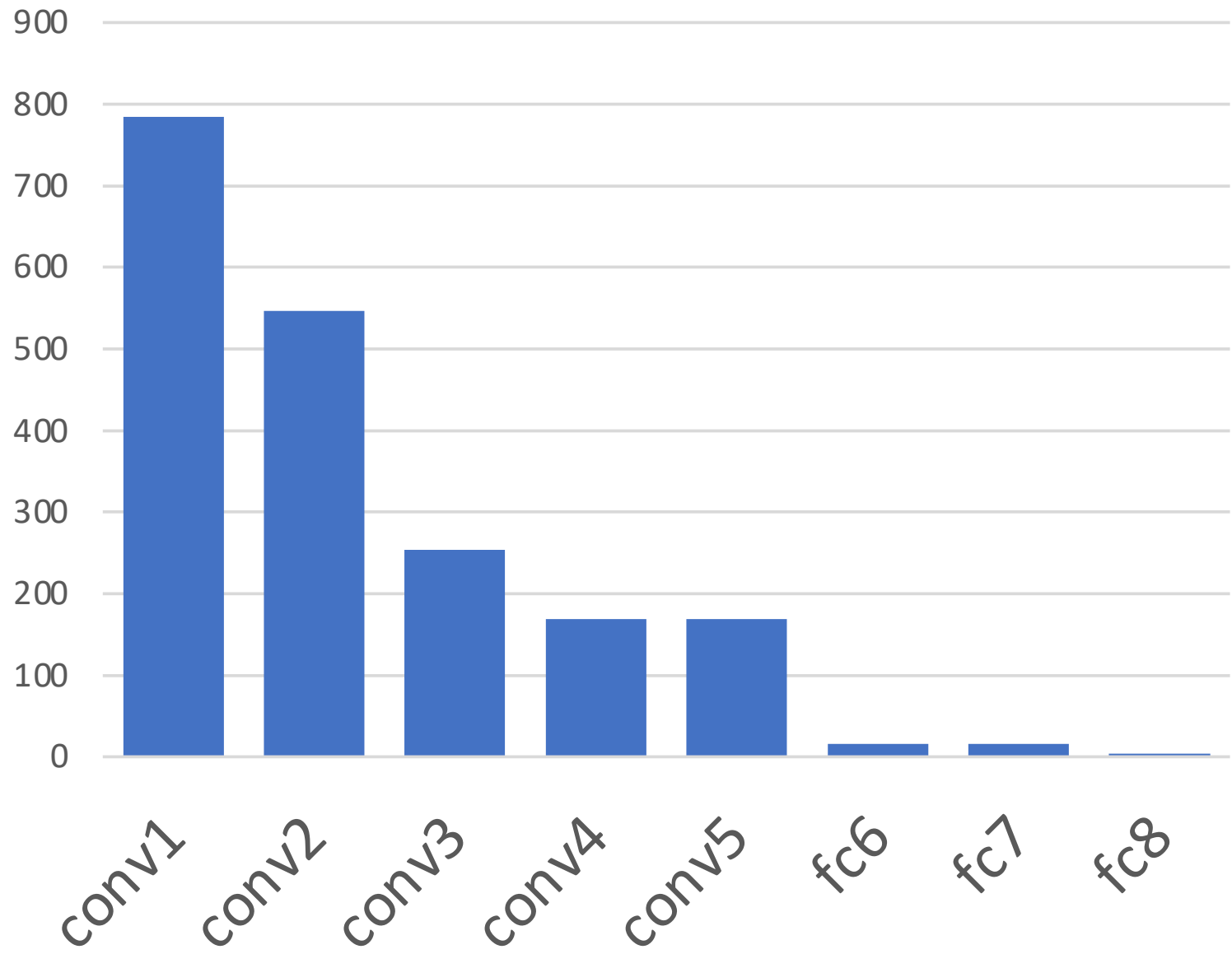


Most of the **memory usage** in the early convolution layers

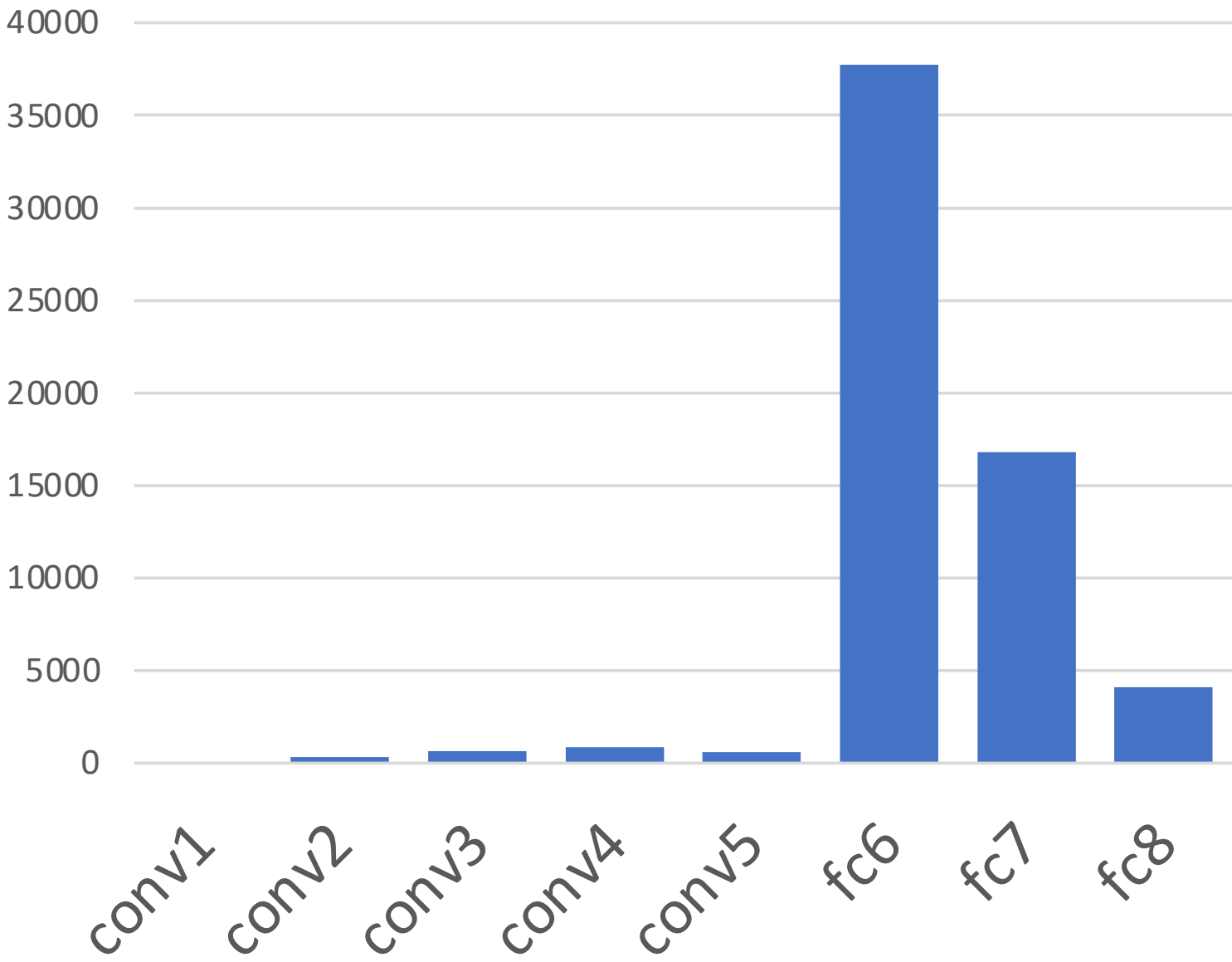
Nearly all **parameters** are in the fully-connected layers

Most **floating-point ops** occur in the convolution layers

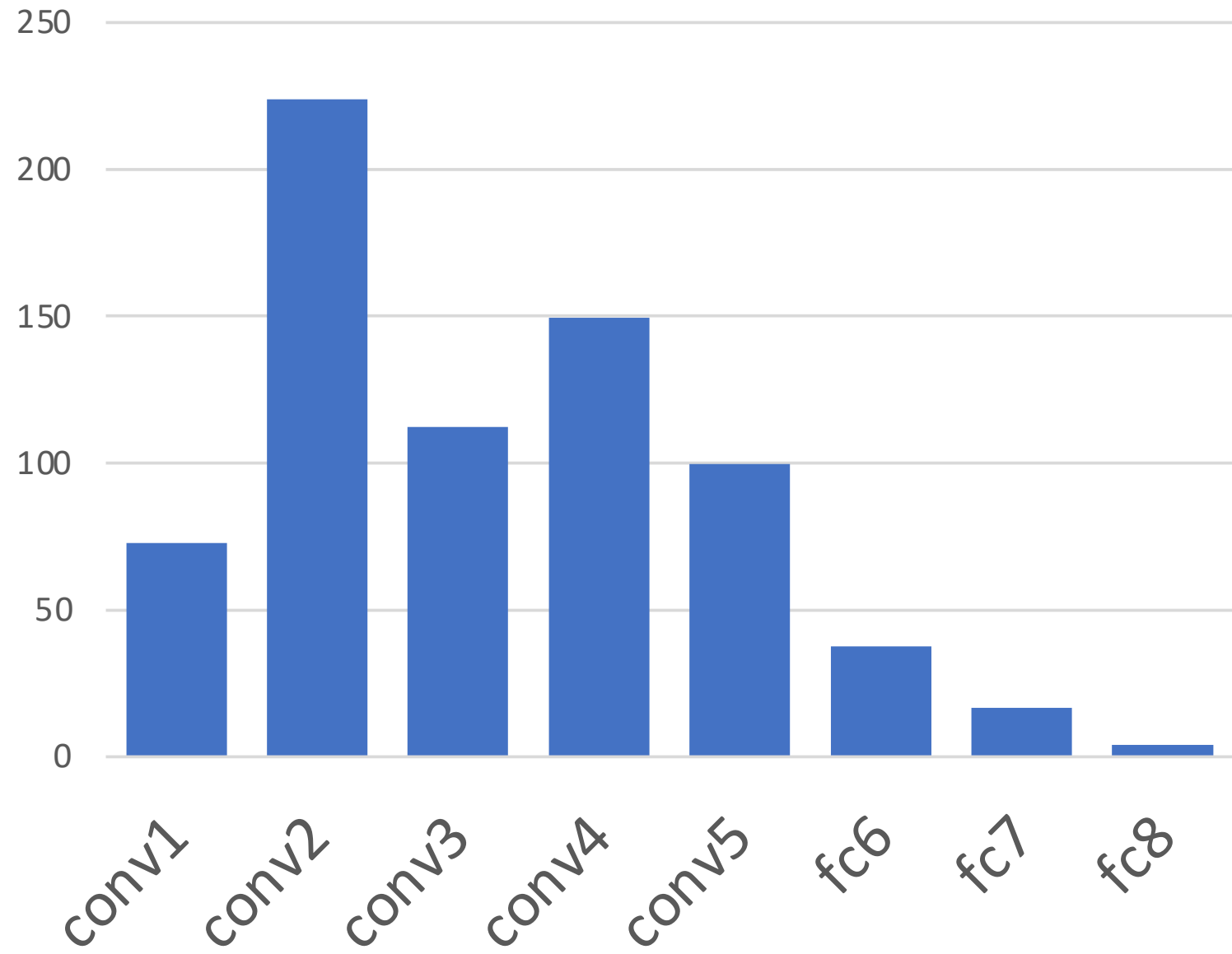
Memory (KB)



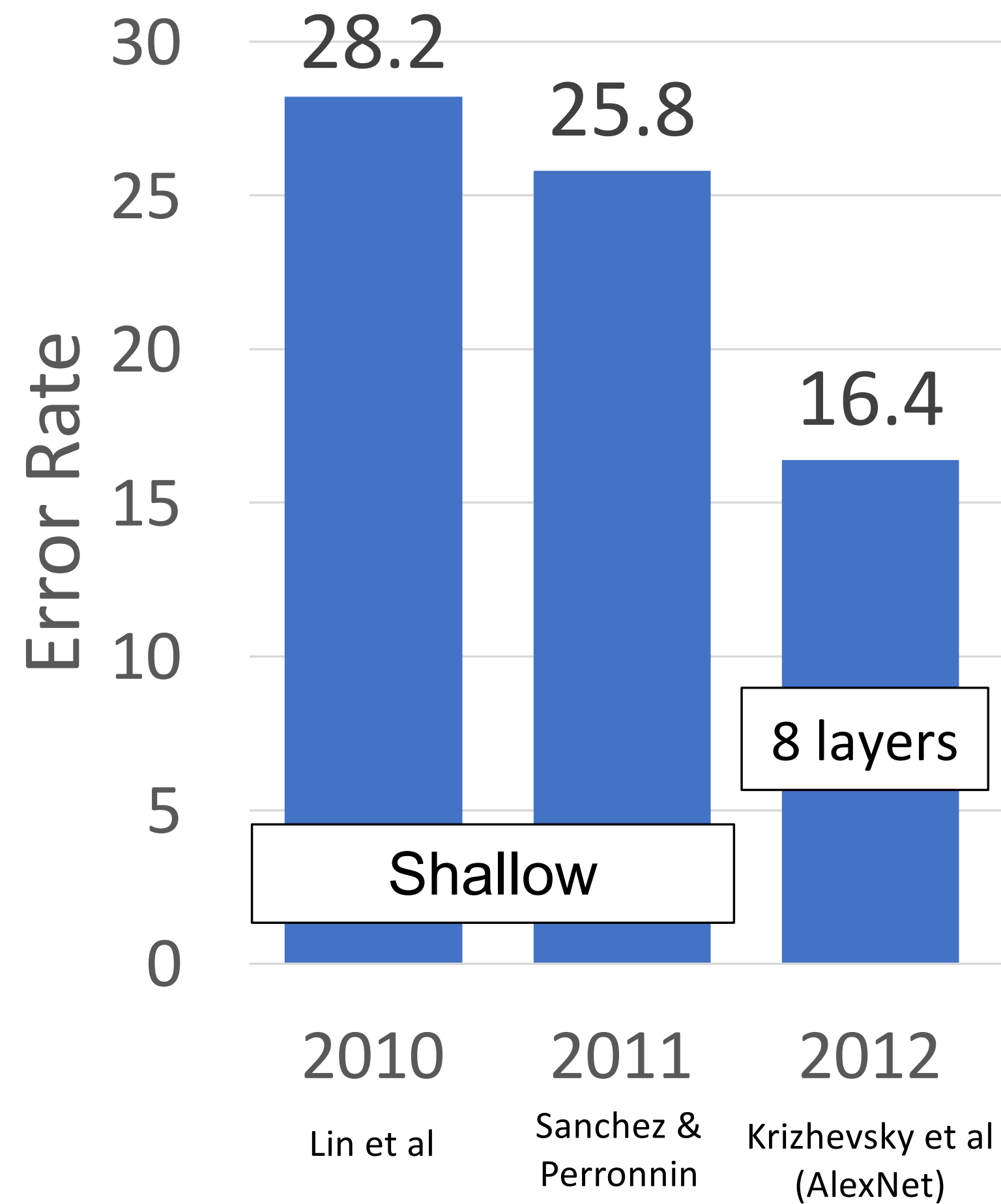
Params (K)



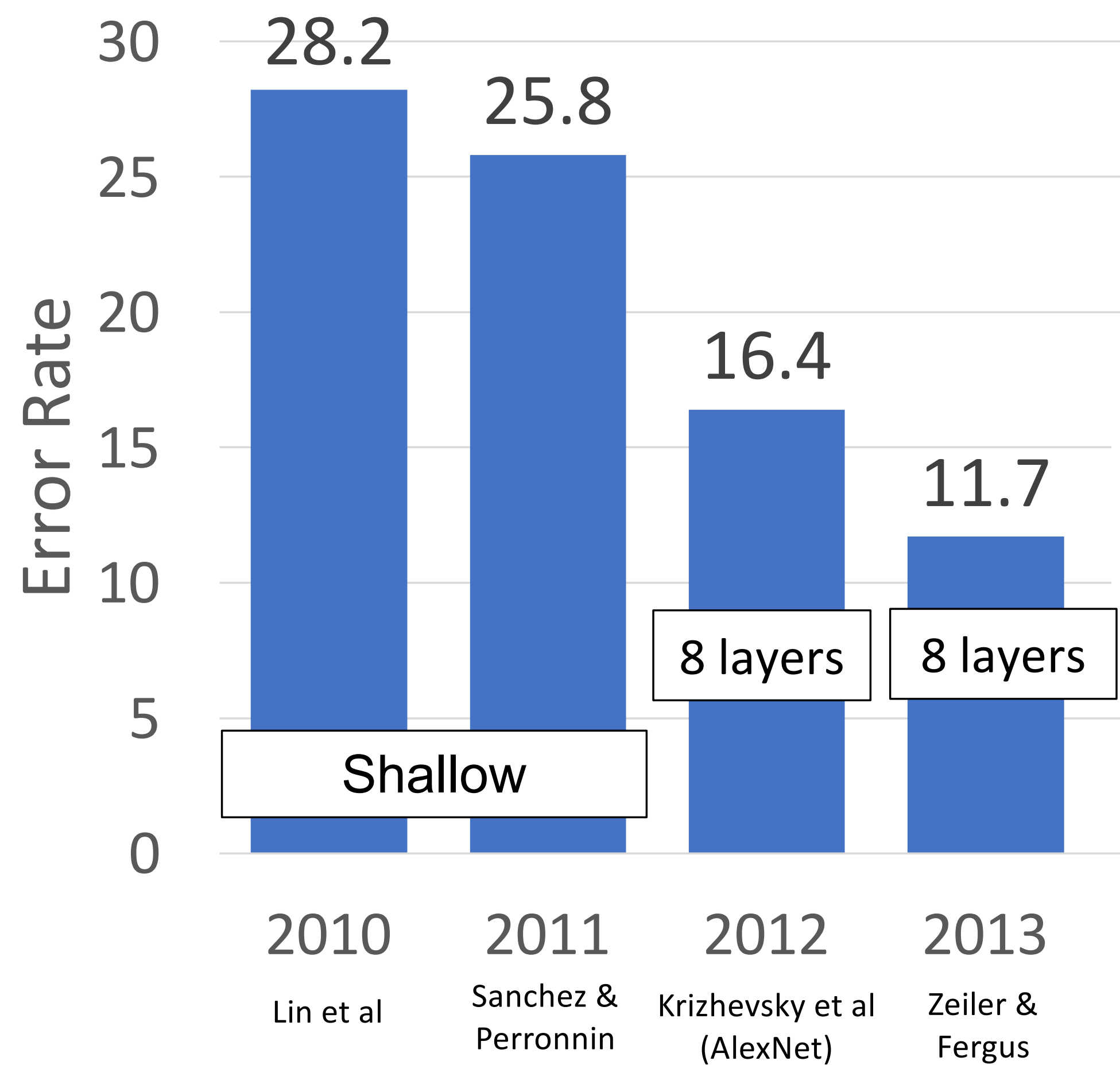
MFLOP



ImageNet Classification Challenge

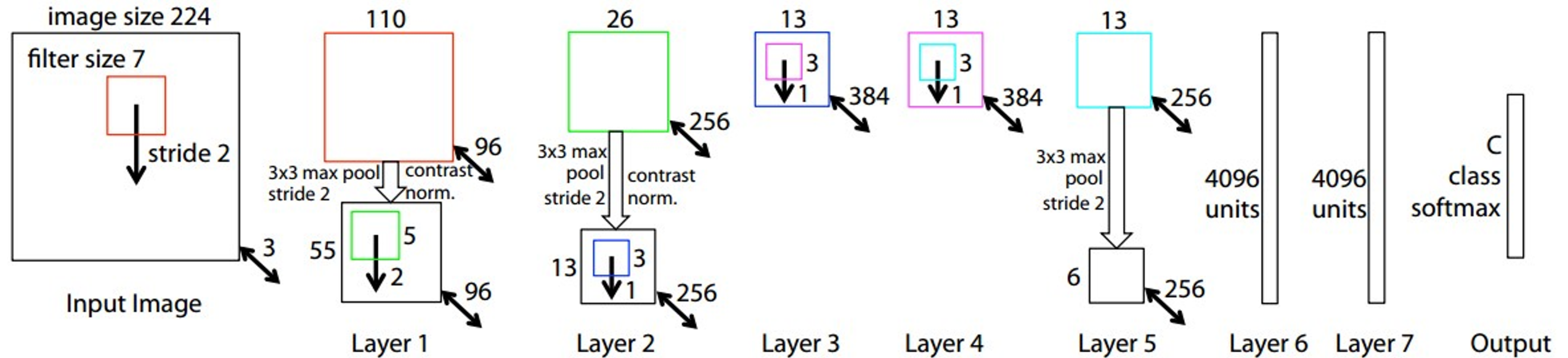


ImageNet Classification Challenge



ZFNet: A Bigger AlexNet

ImageNet top 5 error: 16.4% -> 11.7%



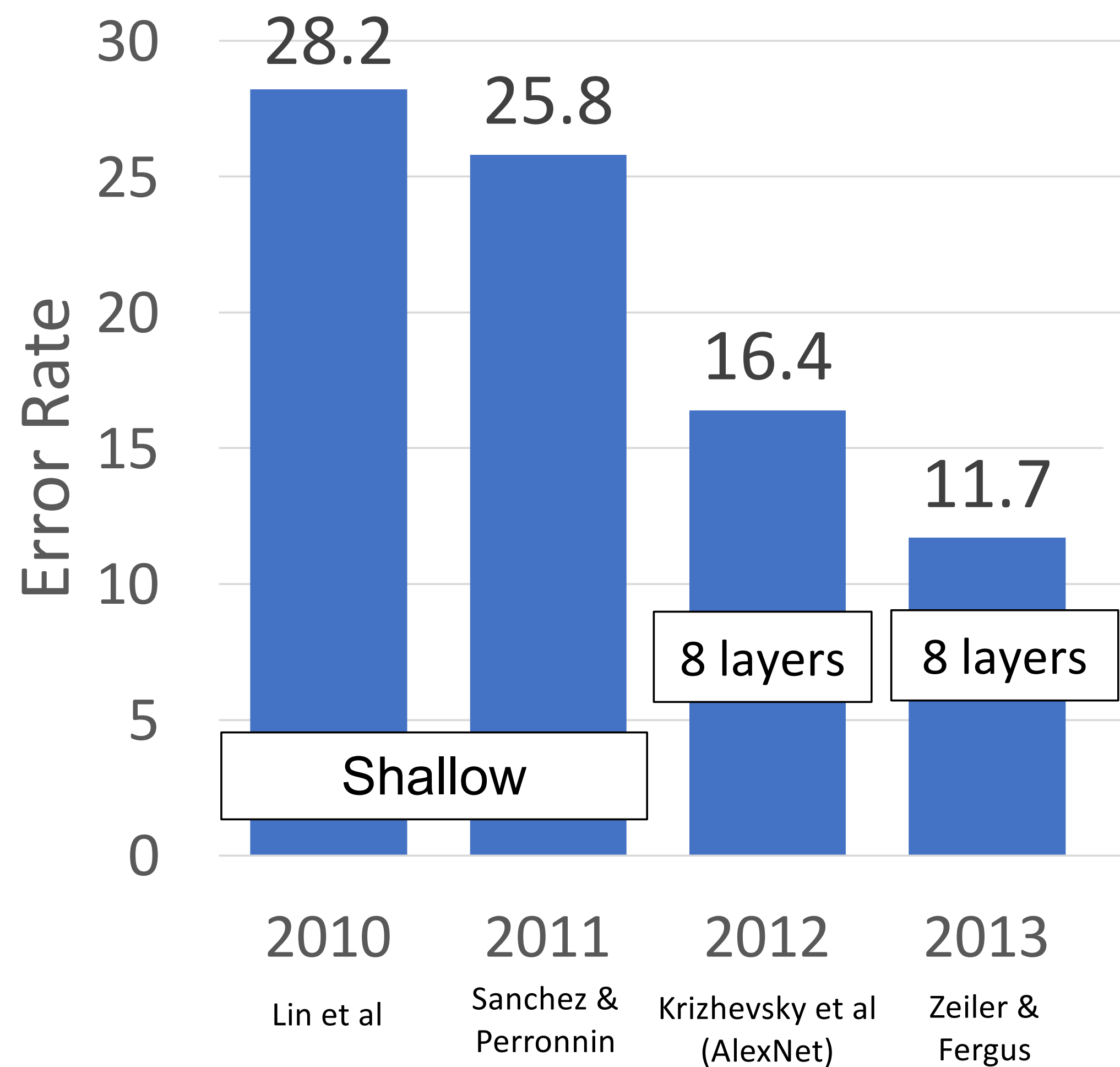
AlexNet but:

Conv1: change from (11x11 stride 4) to (7x7 stride 2)

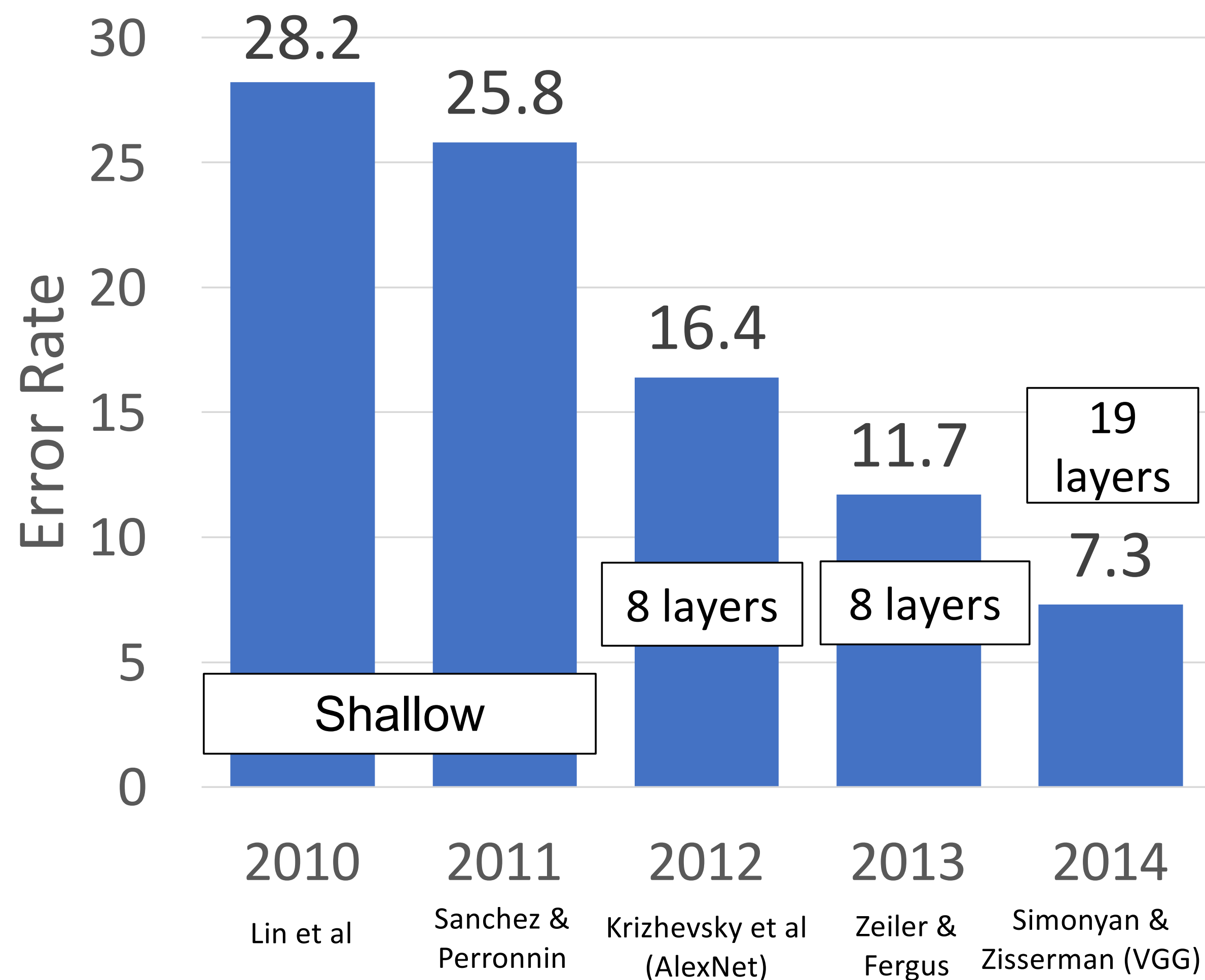
Conv3,4,5: instead of 384, 384, 256 filters use 512, 1024, 512

More trial and error :(

ImageNet Classification Challenge



ImageNet Classification Challenge

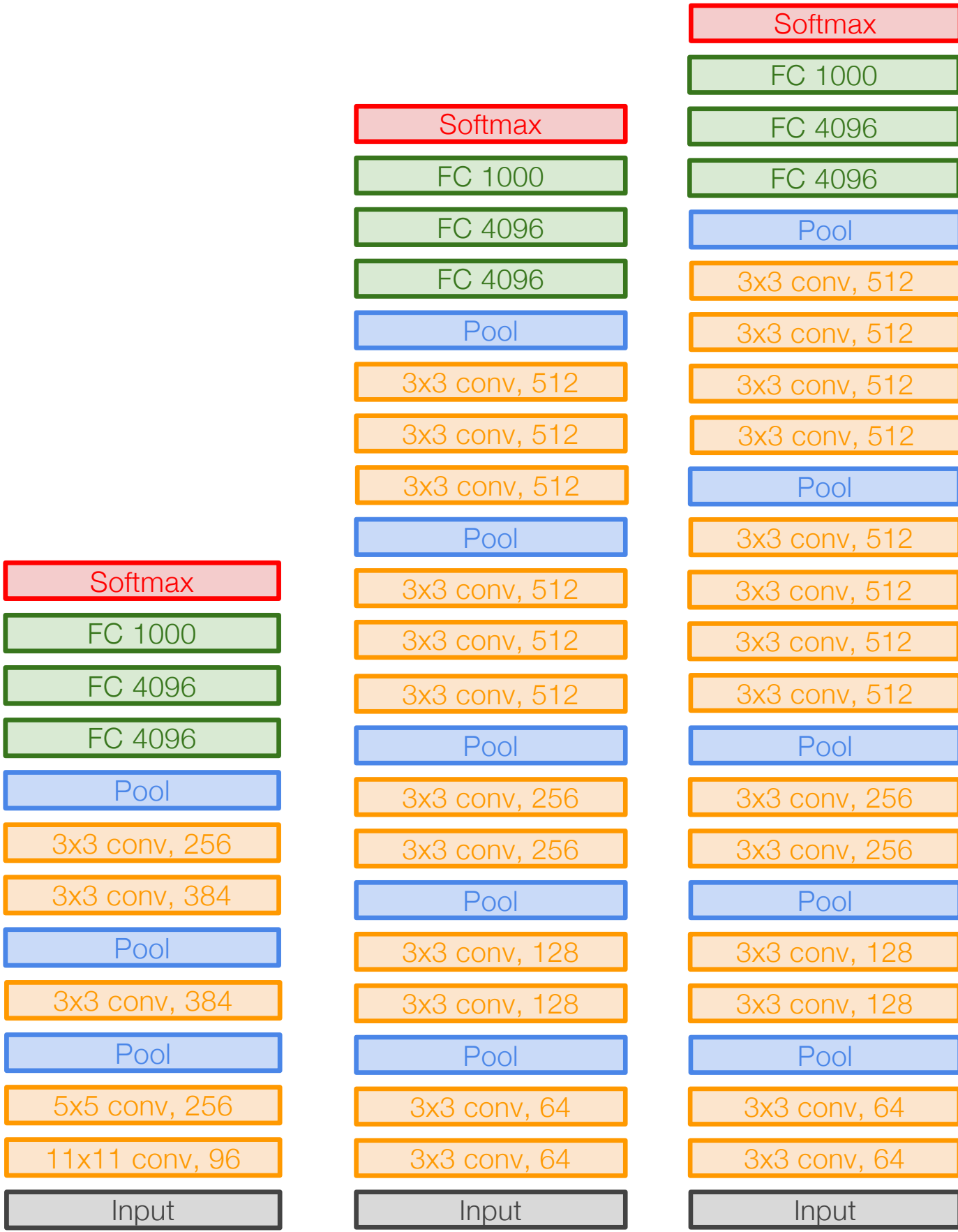




VGG: Deeper Networks, Regular Design

VGG Design rules:

- All conv are 3x3 stride 1 pad 1
- All max pool are 2x2 stride 2
- After pool, double #channels



AlexNet

VGG16

VGG19





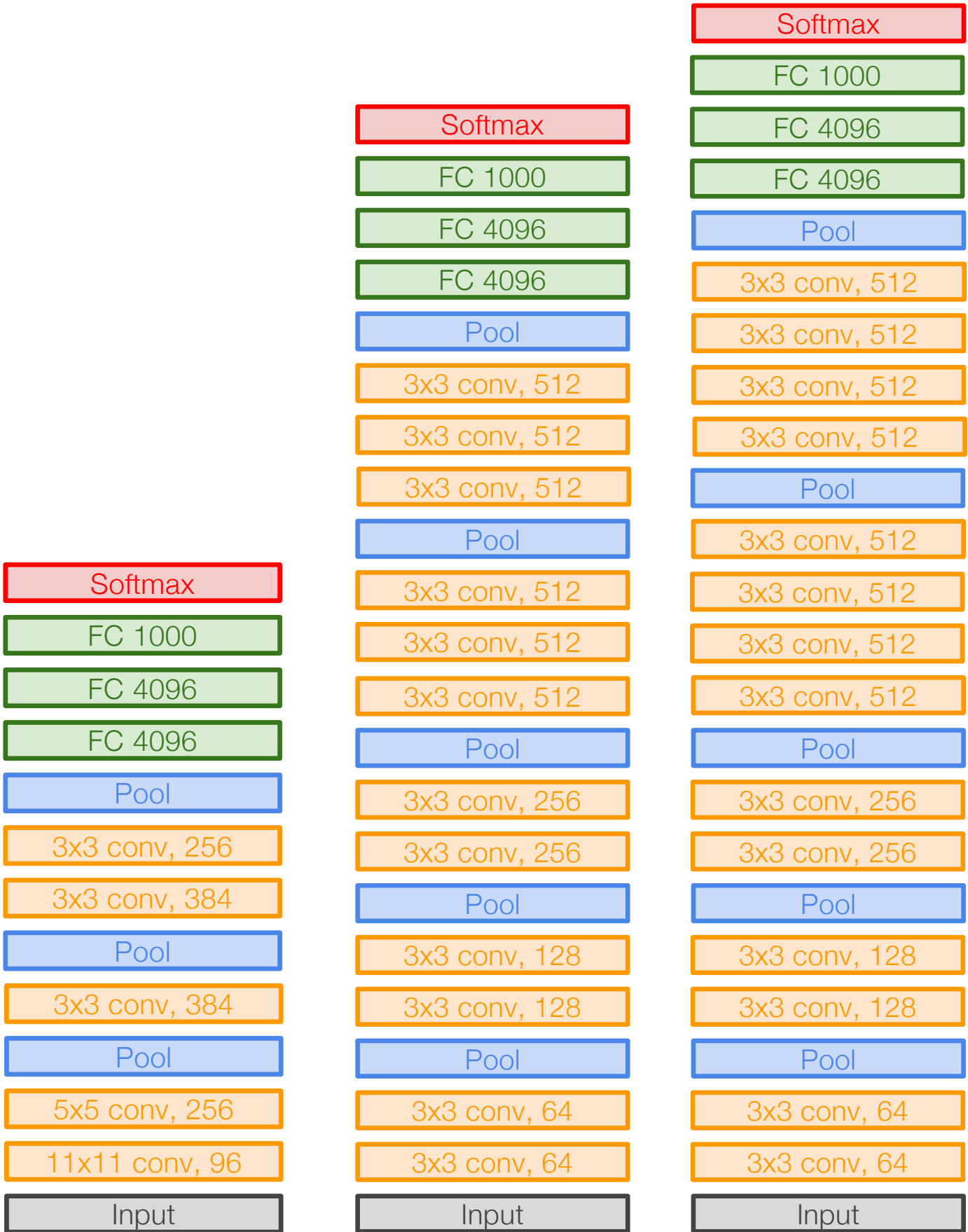
VGG: Deeper Networks, Regular Design

VGG Design rules:

- All conv are 3x3 stride 1 pad 1
- All max pool are 2x2 stride 2
- After pool, double #channels

Network has 5 convolution **stages:**

- Stage 1: conv-conv-pool
- Stage 2: conv-conv-pool
- Stage 3: conv-conv-pool
- Stage 4: conv-conv-conv-[conv]-pool
- Stage 5: conv-conv-conv-[conv]-pool



AlexNet

VGG16

VGG19





VGG: Deeper Networks, Regular Design

VGG Design rules:

All conv are 3x3 stride 1 pad 1

All max pool are 2x2 stride 2

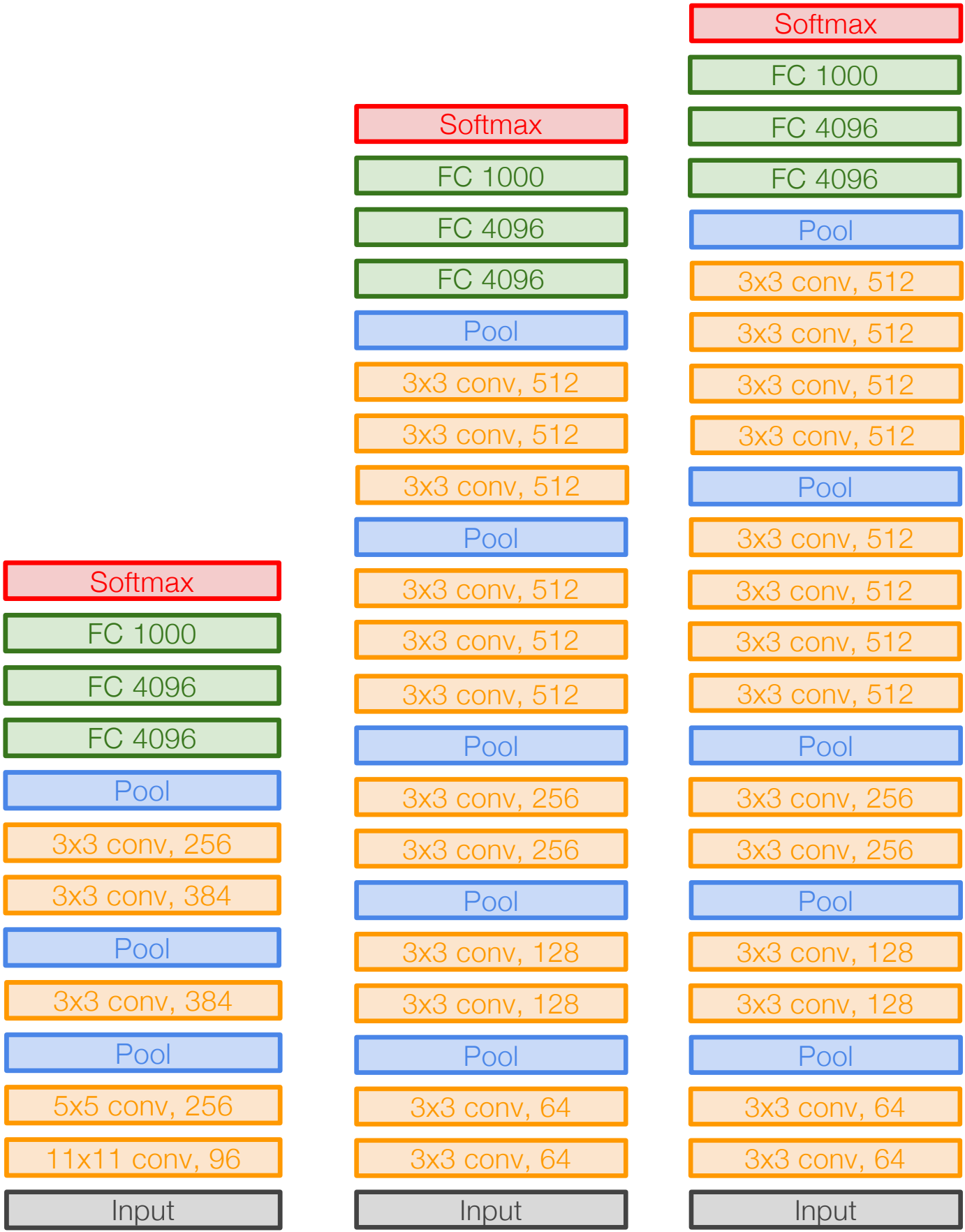
After pool, double #channels

Option 1:

Conv(5x5, C->C)

Params: $25C^2$

FLOPs: $25C^2HW$



AlexNet

VGG16

VGG19





VGG: Deeper Networks, Regular Design

VGG Design rules:

All conv are 3x3 stride 1 pad 1

All max pool are 2x2 stride 2

After pool, double #channels

Option 1:

Conv(5x5, C->C)

Params: $25C^2$

FLOPs: $25C^2HW$

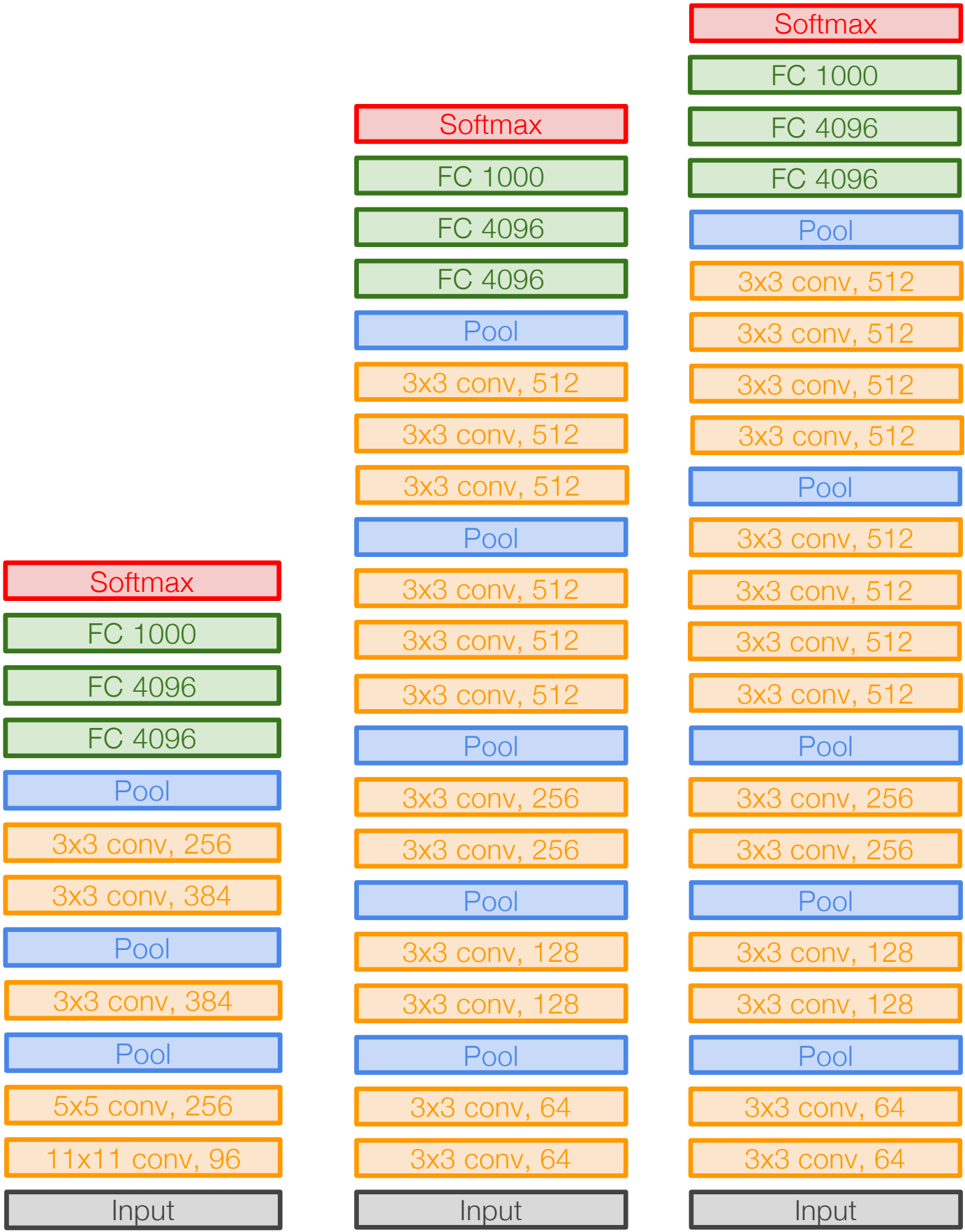
Option 2:

Conv(3x3, C->C)

Conv(3x3, C->C)

Params: $18C^2$

FLOPs: $18C^2HW$



AlexNet

VGG16

VGG19



VGG: Deeper Networks, Regular Design

VGG Design rules:

- All conv are 3x3 stride 1 pad 1
- All max pool are 2x2 stride 2
- After pool, double #channels

Two 3x3 conv has same receptive field as a single 5x5 conv, but has fewer parameters and takes less computation!

Option 1:

Conv(5x5, C->C)

Params: $25C^2$

FLOPs: $25C^2HW$

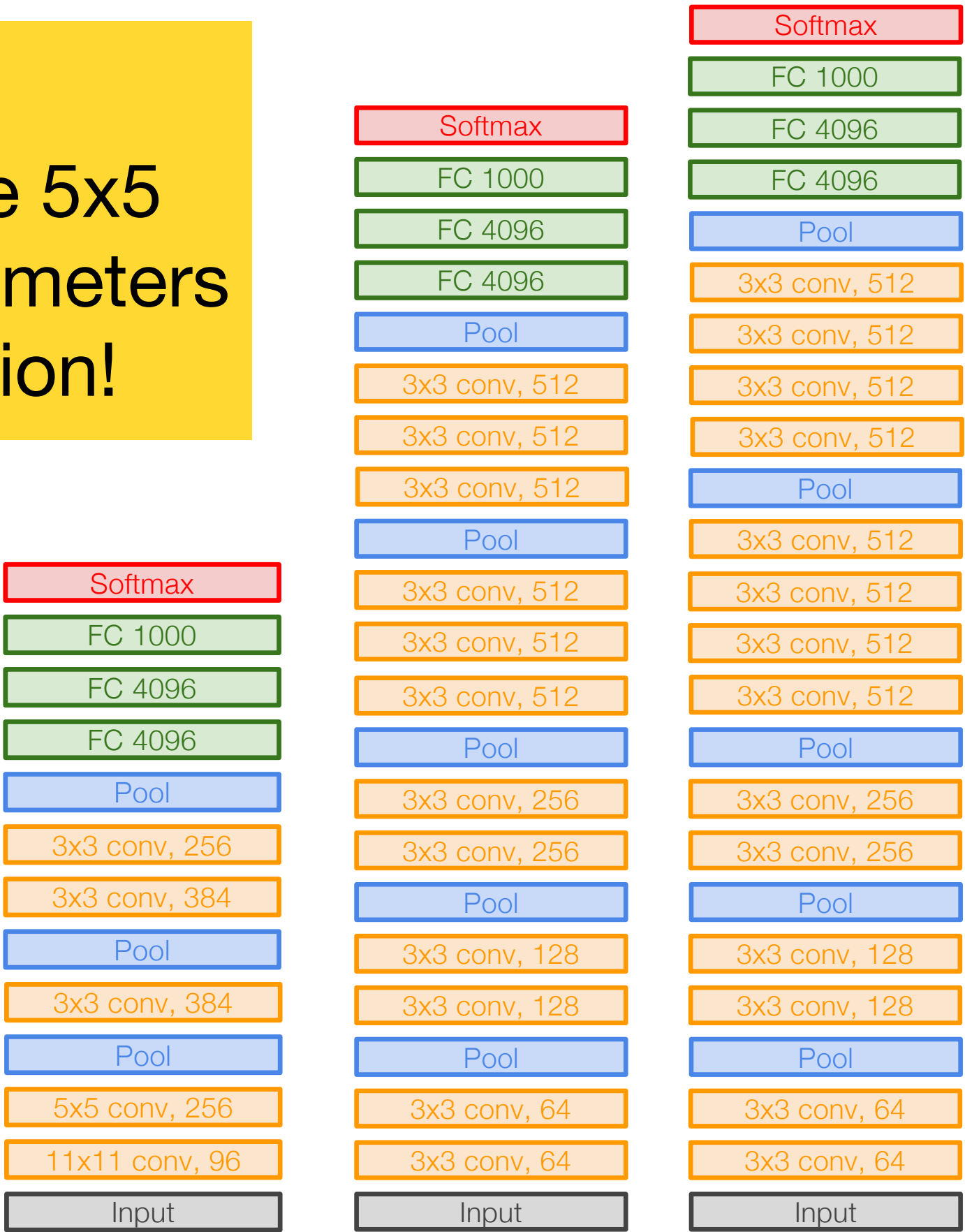
Option 2:

Conv(3x3, C->C)

Conv(3x3, C->C)

Params: $18C^2$

FLOPs: $18C^2HW$



AlexNet

VGG16

VGG19



VGG: Deeper Networks, Regular Design

VGG Design rules:

All conv are 3x3 stride 1 pad 1

All max pool are 2x2 stride 2

After pool, double #channels

Option 1:

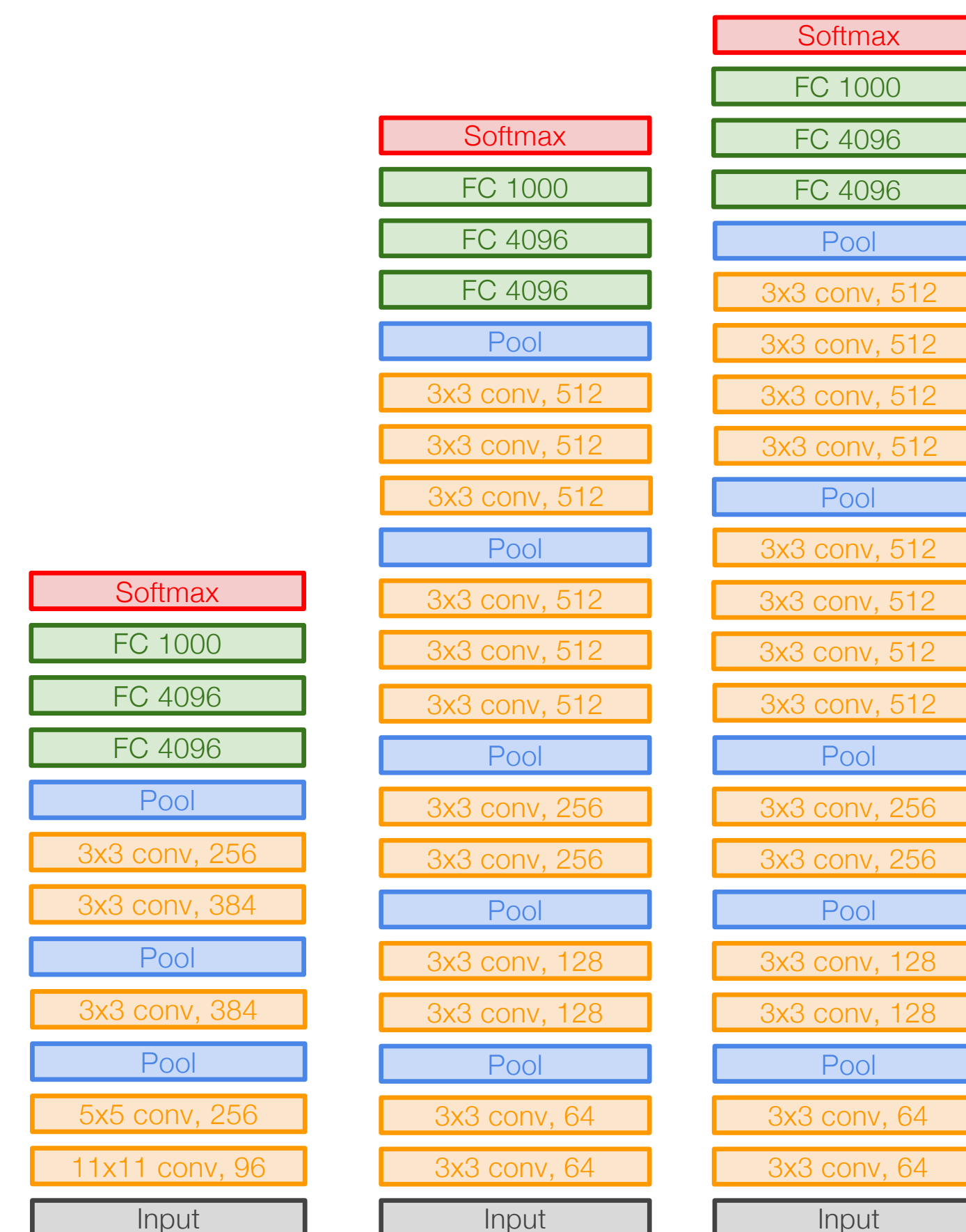
Input: C x 2H x 2W

Layer: Conv(3x3, C->C)

Memory: 4HWC

Params: $9C^2$

FLOPs: $36HWC^2$



AlexNet

VGG16

VGG19



VGG: Deeper Networks, Regular Design

VGG Design rules:

All conv are 3x3 stride 1 pad 1

All max pool are 2x2 stride 2

After pool, double #channels

Option 1:

Input: $C \times 2H \times 2W$

Layer: Conv(3x3, $C \rightarrow C$)

Memory: $4HWC$

Params: $9C^2$

FLOPs: $36HWC^2$

Option 2:

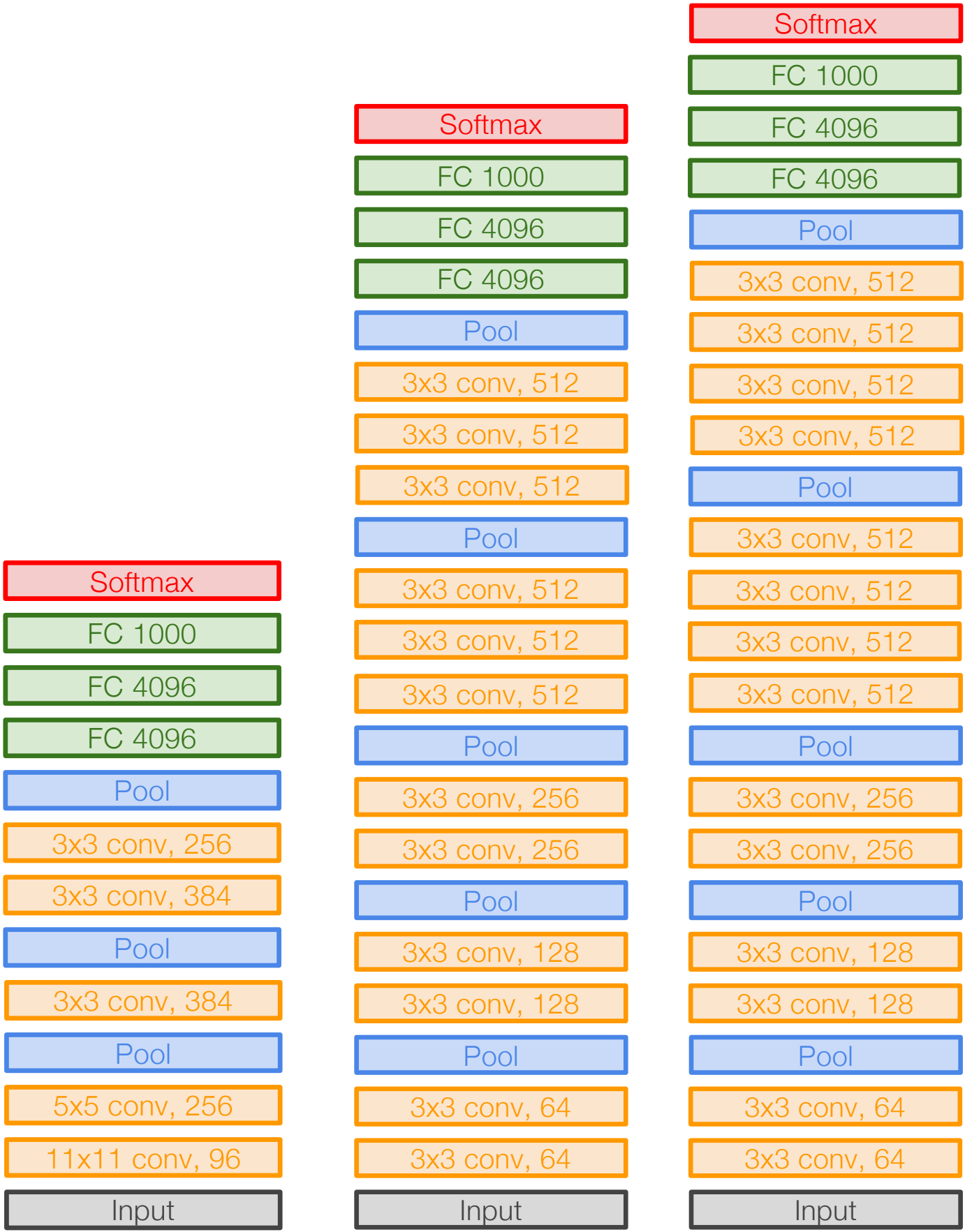
Input: $2C \times H \times W$

Layer: Conv(3x3, $2C \rightarrow 2C$)

Memory: $2HWC$

Params: $36C^2$

FLOPs: $36HWC^2$



AlexNet

VGG16

VGG19





VGG: Deeper Networks, Regular Design

VGG Design rules:

- All conv are 3x3 stride 1 pad 1
- All max pool are 2x2 stride 2
- After pool, double #channels

Conv layers at each spatial resolution take the same amount of computation!

Option 1:

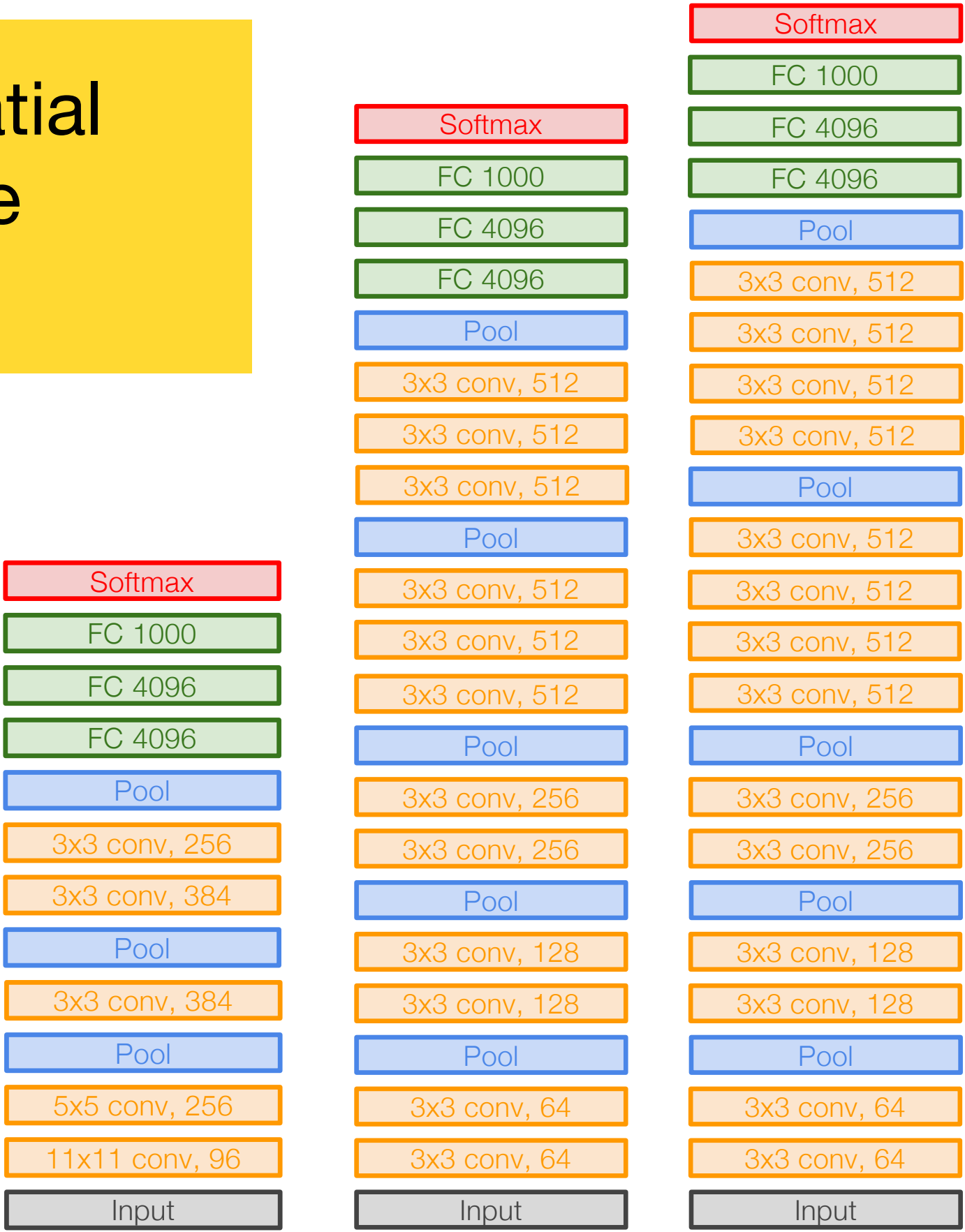
Input: $C \times 2H \times 2W$
 Layer: Conv(3x3, $C \rightarrow C$)

Memory: $4HWC$
 Params: $9C^2$
 FLOPs: $36HWC^2$

Option 2:

Input: $2C \times H \times W$
 Layer: Conv(3x3, $2C \rightarrow 2C$)

Memory: $2HWC$
 Params: $36C^2$
 FLOPs: $36HWC^2$

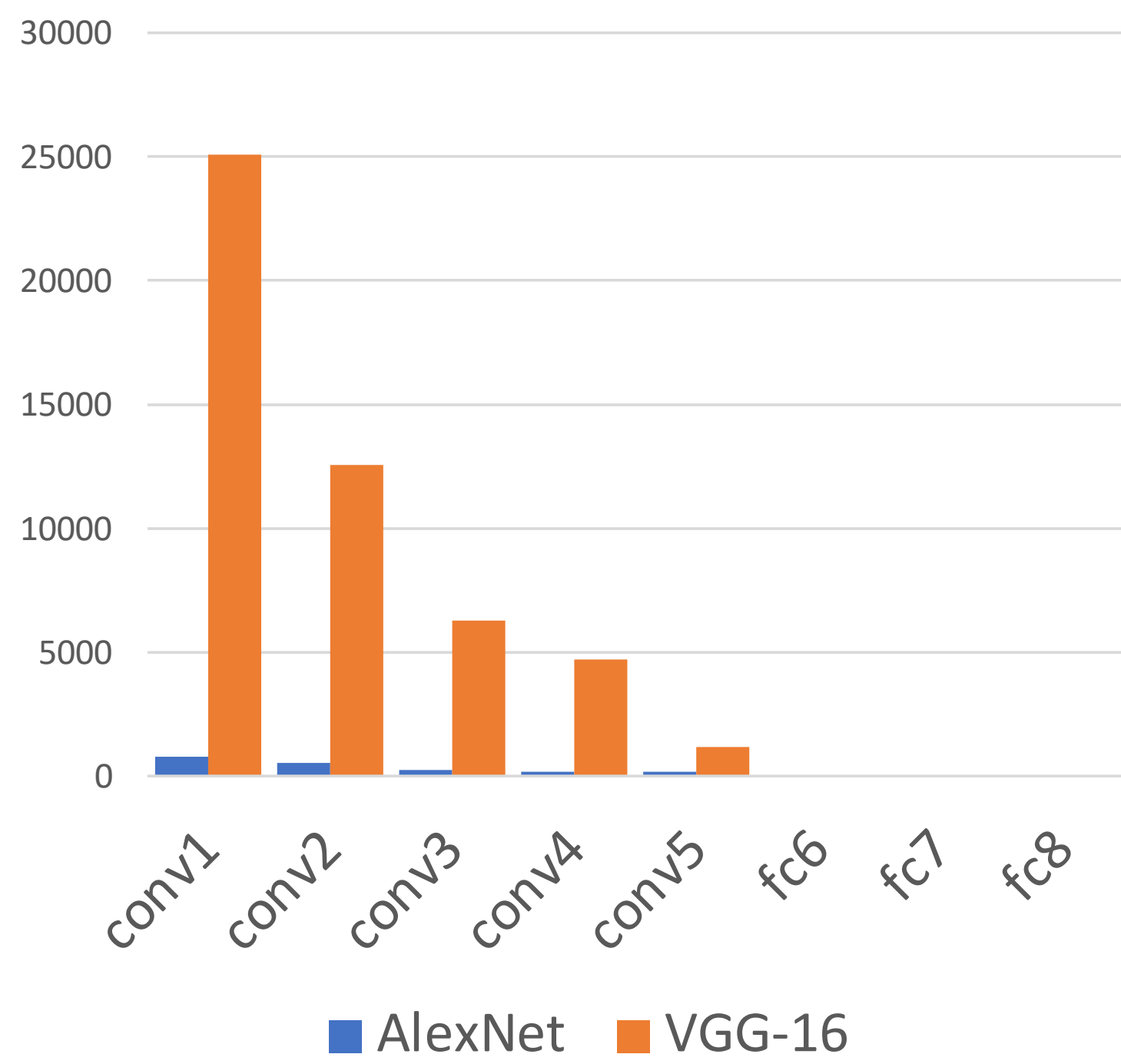


AlexNet VGG16 VGG19



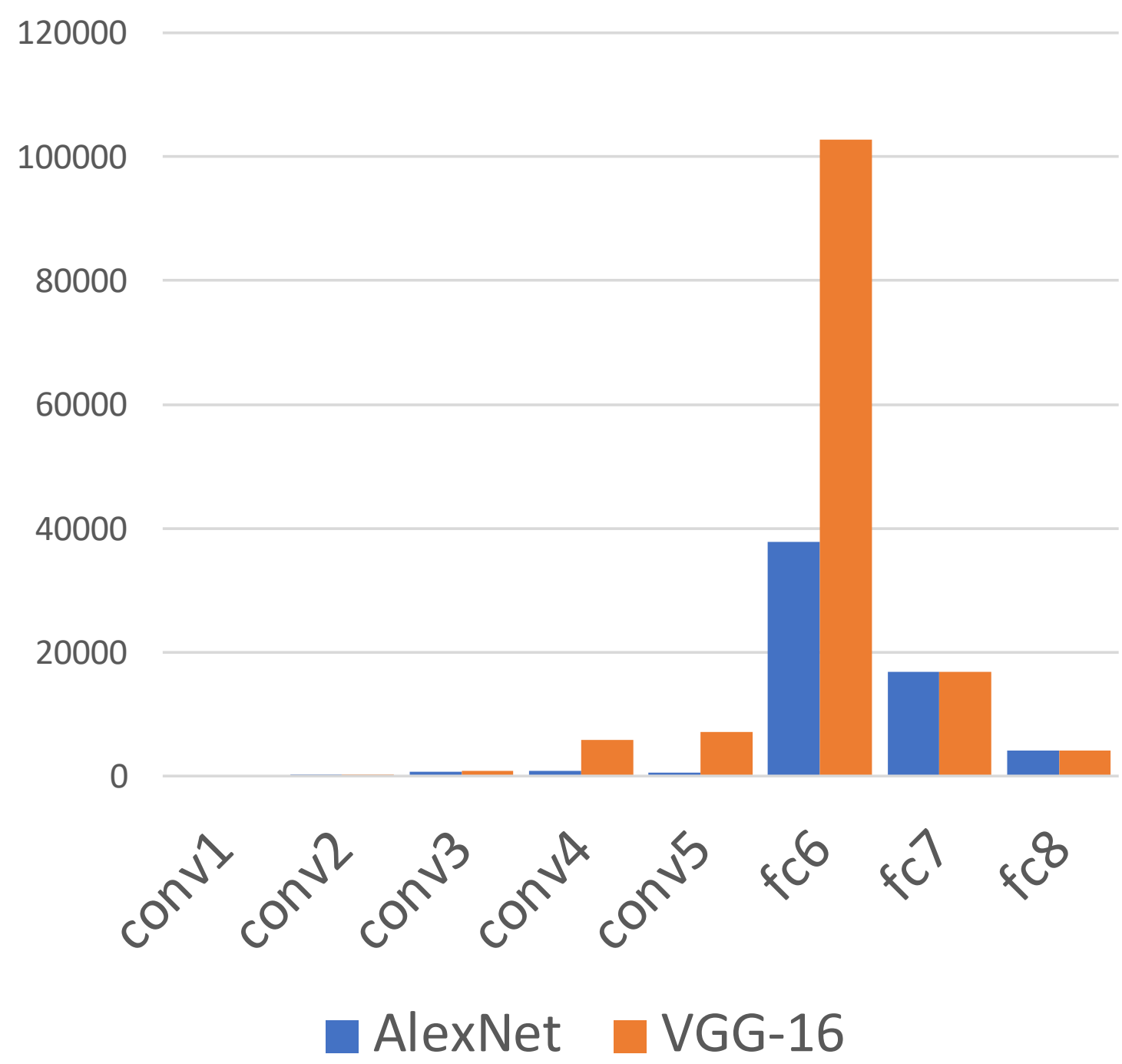
AlexNet vs VGG-16: Much bigger network!

AlexNet vs VGG-16
(Memory, KB)



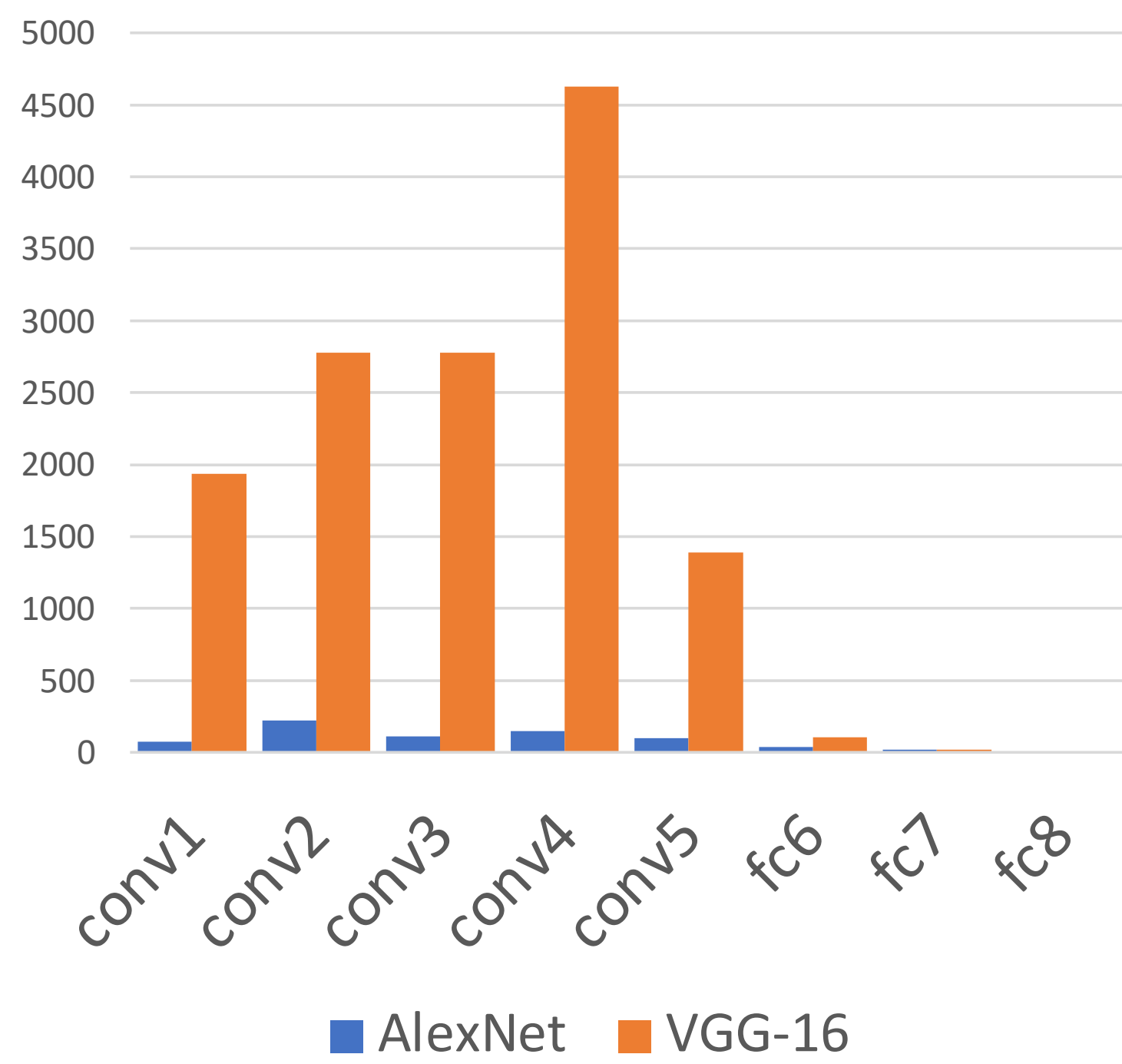
AlexNet total: 1.9MB
VGG-16 total: 48.6MB (25x)

AlexNet vs VGG-16
(Params, M)



AlexNet total: 61M
VGG-16 total: 138M (2.3x)

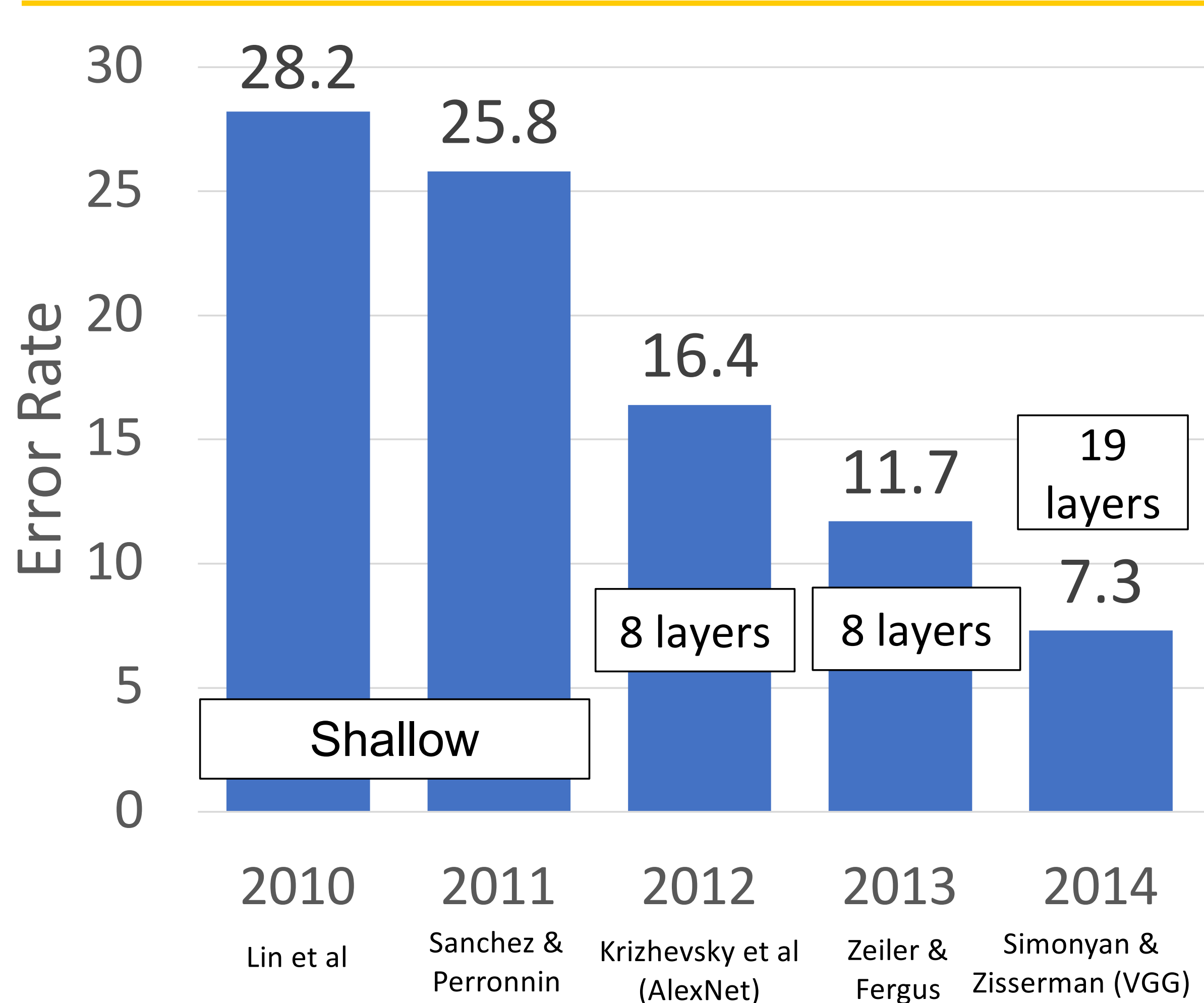
AlexNet vs VGG-16
(MFLOPs)



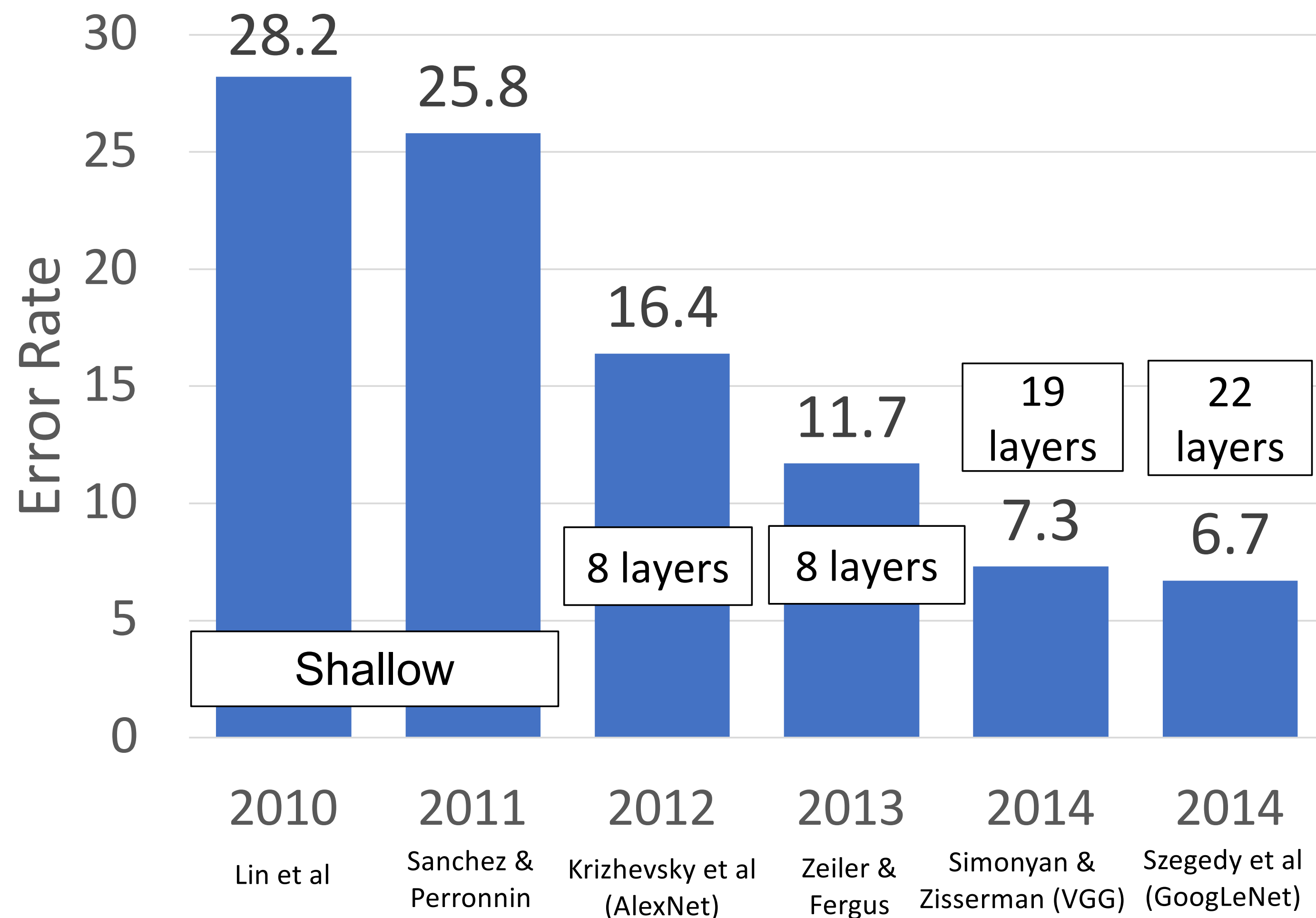
AlexNet total: 0.7 GFLOP
VGG-16 total: 13.6 GFLOP (19.4x)



ImageNet Classification Challenge

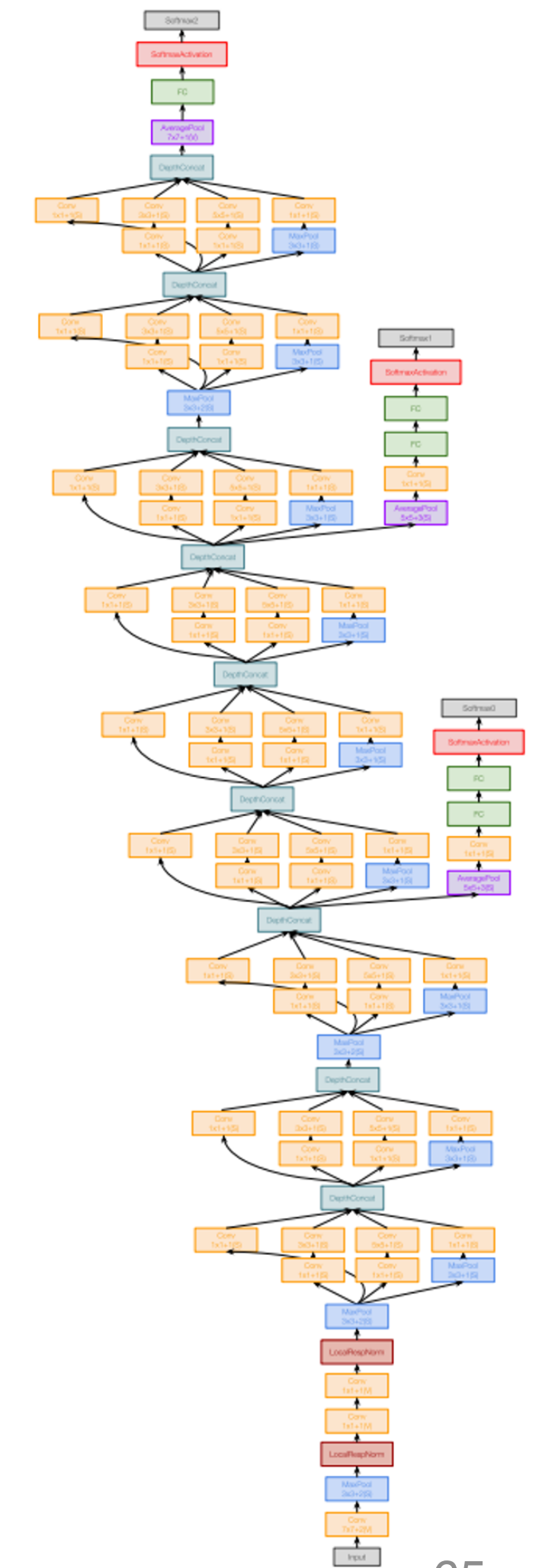


ImageNet Classification Challenge



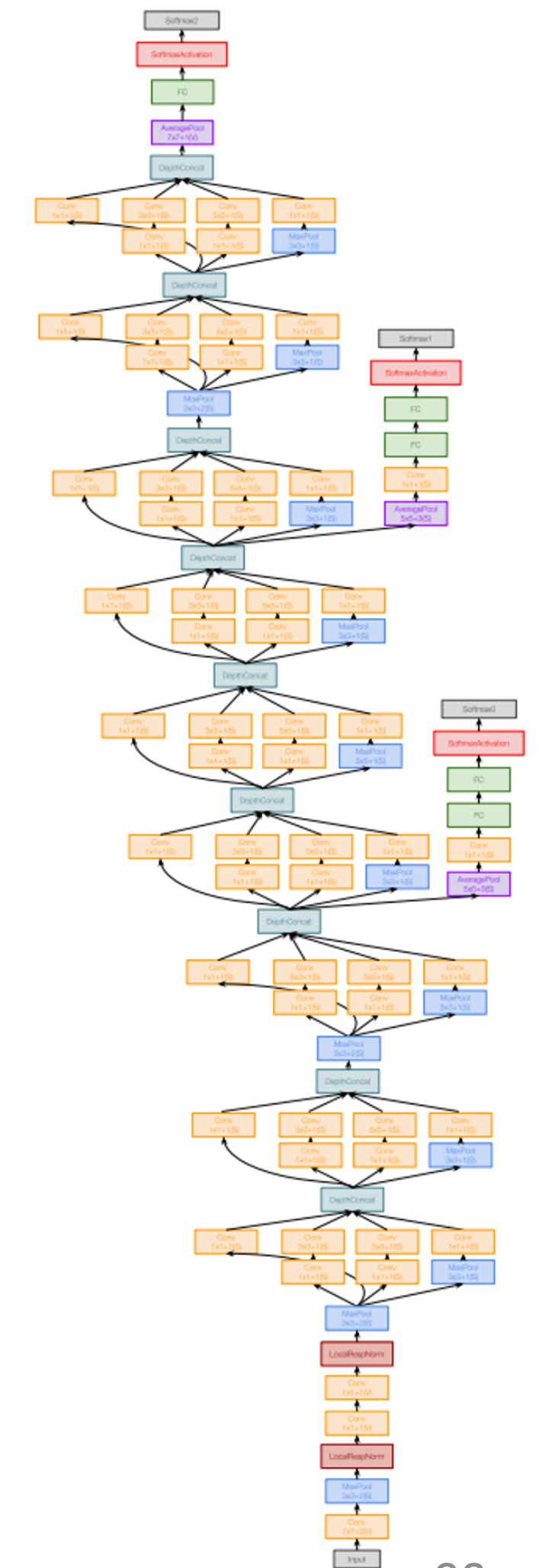
GoogLeNet: Focus on Efficiency

Many innovations for efficiency: reduce parameter count, memory usage, and computation



GoogLeNet: Aggressive Stem

Stem network at the start aggressively downsamples input
(Recall in VGG-16: Most of the compute was at the start)





GoogLeNet: Aggressive Stem

Stem network at the start aggressively downsamples input
(Recall in VGG-16: Most of the compute was at the start)

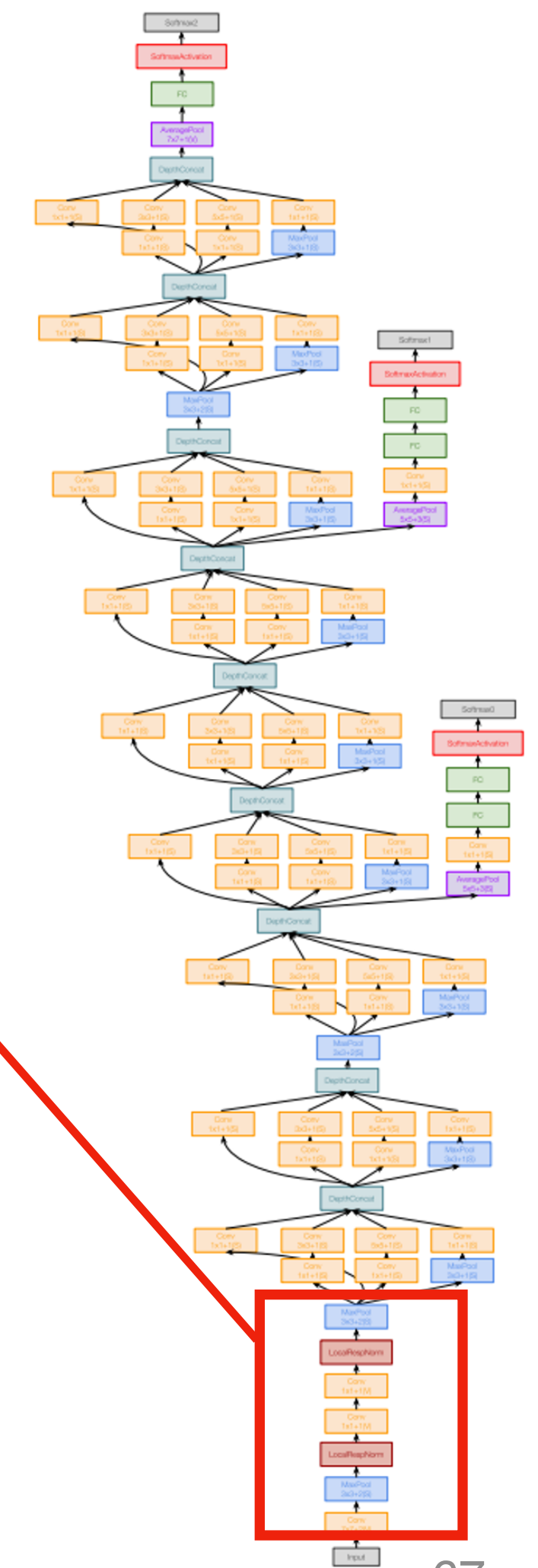
| Layer | Input size | | Layer | | | | Output size | | | | |
|----------|------------|-----|---------|--------|-------|-----|-------------|-----|--------|--------|----------|
| | C | H/W | Filters | Kernel | Strid | Pad | C | H/W | Memory | Params | Flop (M) |
| Conv | 3 | 224 | 64 | 7 | 2 | 3 | 64 | 112 | 3136 | 9 | 118 |
| Max-pool | 64 | 112 | | 3 | 2 | 1 | 64 | 56 | 784 | 0 | 2 |
| Conv | 64 | 56 | 64 | 1 | 1 | 0 | 64 | 56 | 784 | 4 | 13 |
| Conv | 64 | 56 | 192 | 3 | 1 | 1 | 192 | 56 | 2352 | 111 | 347 |
| Max-pool | 192 | 56 | | 3 | 2 | 1 | 192 | 28 | 588 | 0 | 1 |

Total from 224 to 28 spatial resolution:

Memory: 7.5 MB

Params: 124K

MFLOP: 418



GoogLeNet: Aggressive Stem

Stem network at the start aggressively downsamples input
 (Recall in VGG-16: Most of the compute was at the start)

| Layer | Input size | | Layer | | | | Output size | | | | |
|----------|------------|-----|---------|--------|-------|-----|-------------|-----|--------|--------|----------|
| | C | H/W | Filters | Kernel | Strid | Pad | C | H/W | Memory | Params | Flop (M) |
| Conv | 3 | 224 | 64 | 7 | 2 | 3 | 64 | 112 | 3136 | 9 | 118 |
| Max-pool | 64 | 112 | | 3 | 2 | 1 | 64 | 56 | 784 | 0 | 2 |
| Conv | 64 | 56 | 64 | 1 | 1 | 0 | 64 | 56 | 784 | 4 | 13 |
| Conv | 64 | 56 | 192 | 3 | 1 | 1 | 192 | 56 | 2352 | 111 | 347 |
| Max-pool | 192 | 56 | | 3 | 2 | 1 | 192 | 28 | 588 | 0 | 1 |

Total from 224 to 28 spatial resolution:

Memory: 7.5 MB

Params: 124K

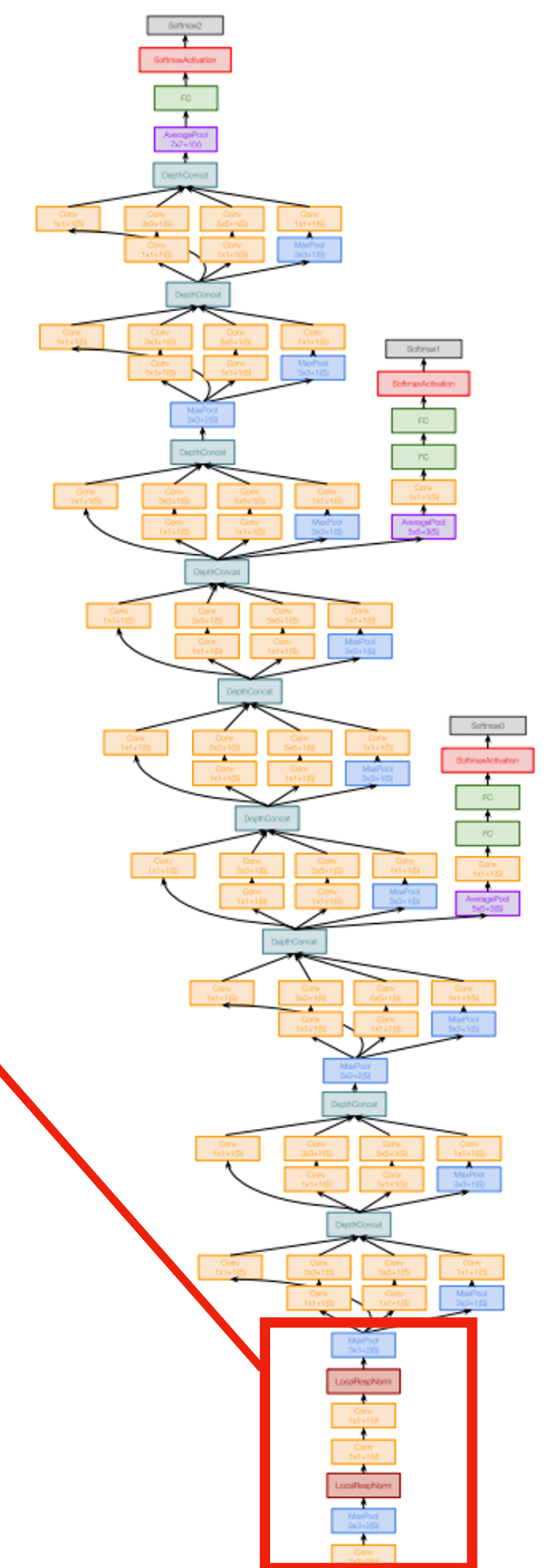
MFLOP: 418

Compare VGG-16:

Memory: 42.9 MB (5.7x)

Params: 1.1M (8.9x)

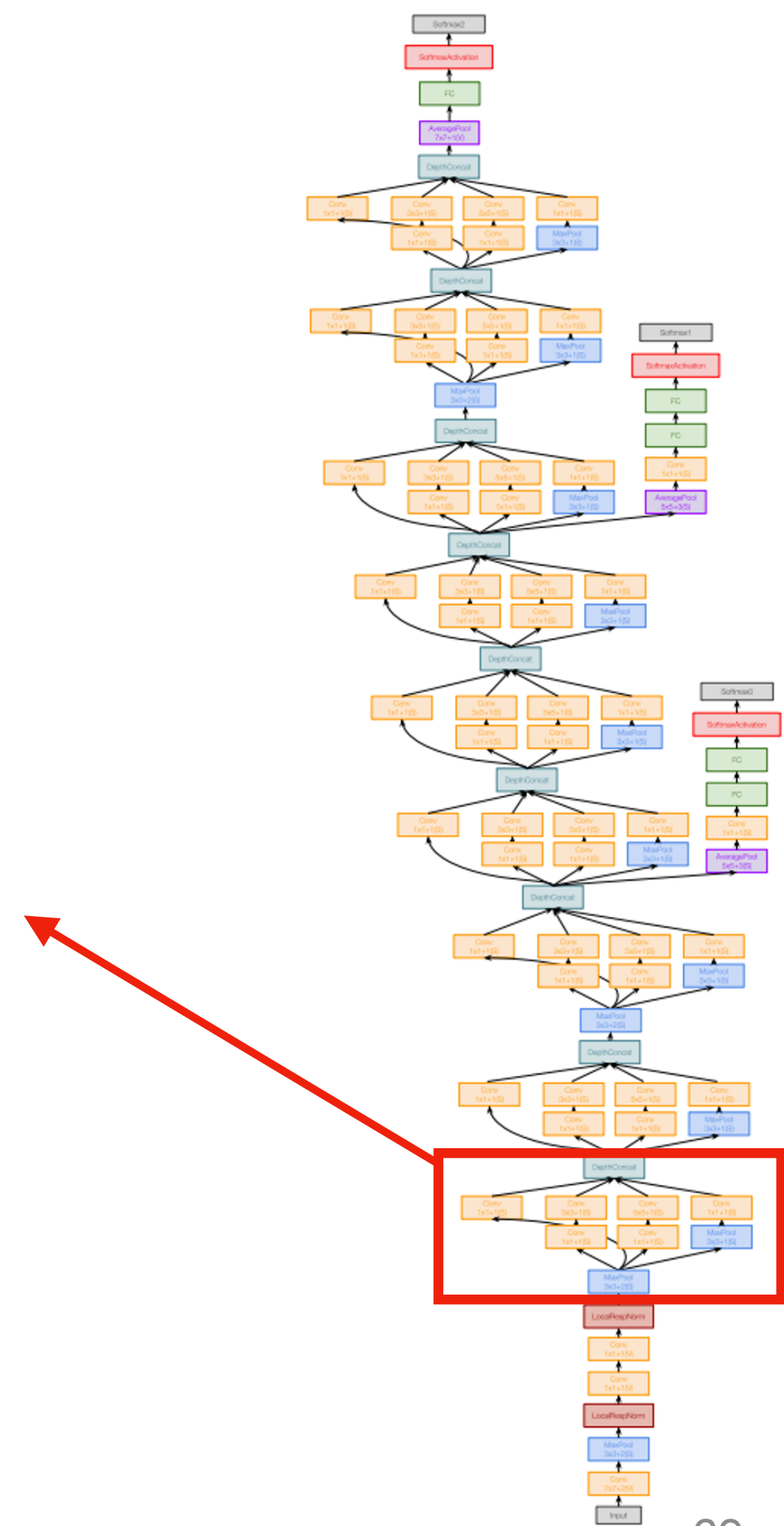
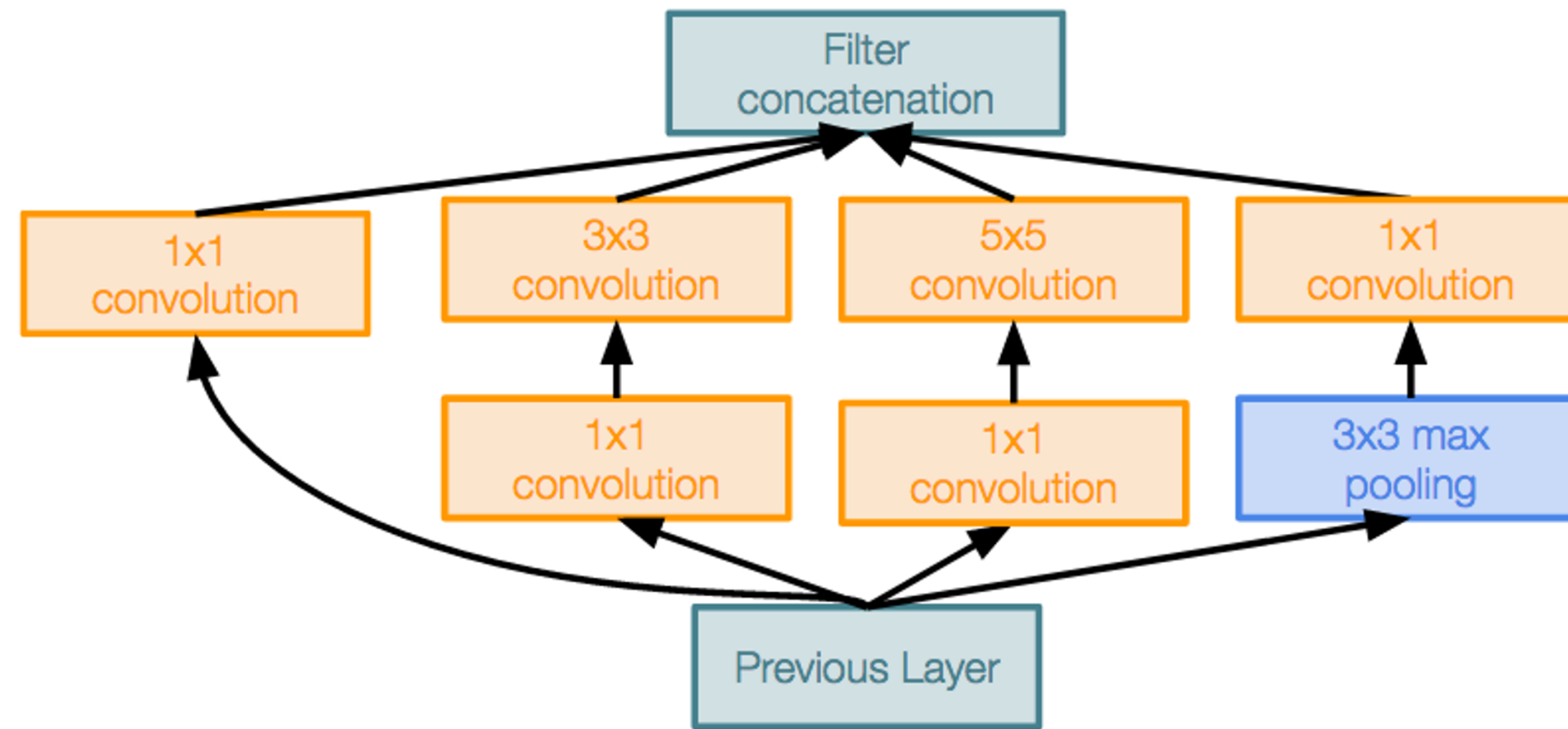
MFLOP: 7485 (17.8x)



GoogLeNet: Inception Module

Inception module: Local unit with parallel branches

Local structure repeated many times throughout the network

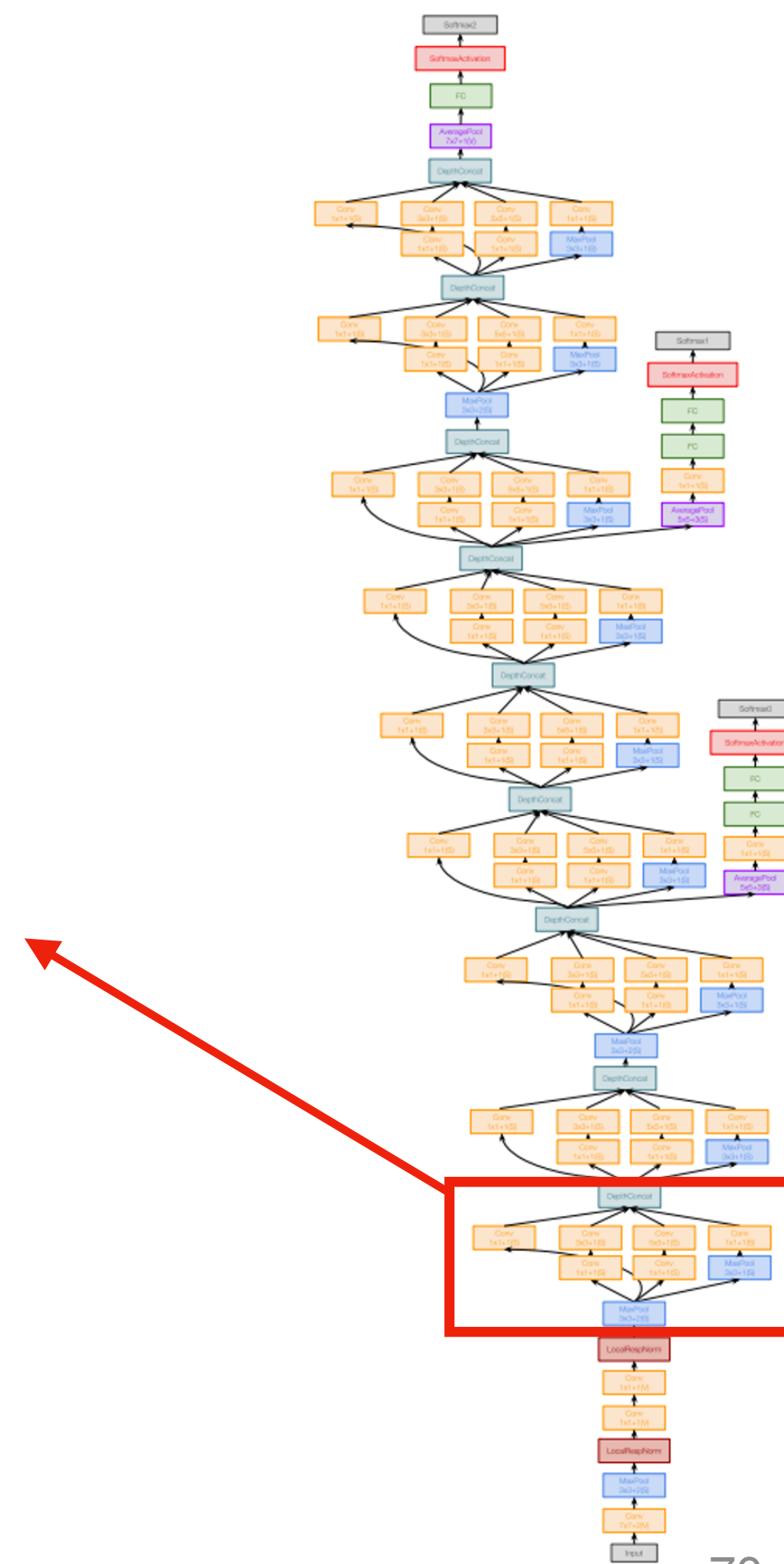
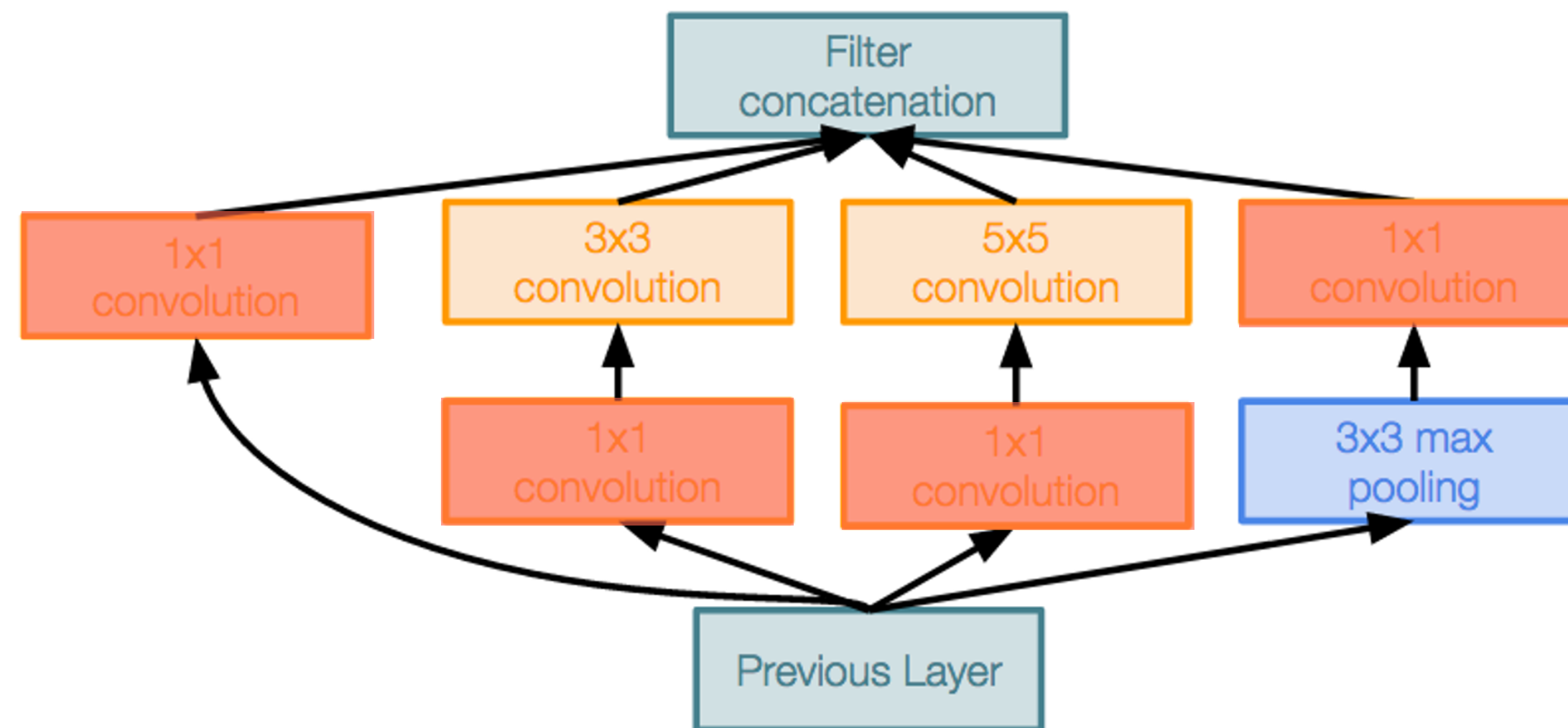


GoogLeNet: Inception Module

Inception module: Local unit with parallel branches

Local structure repeated many times throughout the network

Uses 1x1 “Bottleneck” layers to reduce channel dimension before expensive conv (we will revisit this with ResNet!)



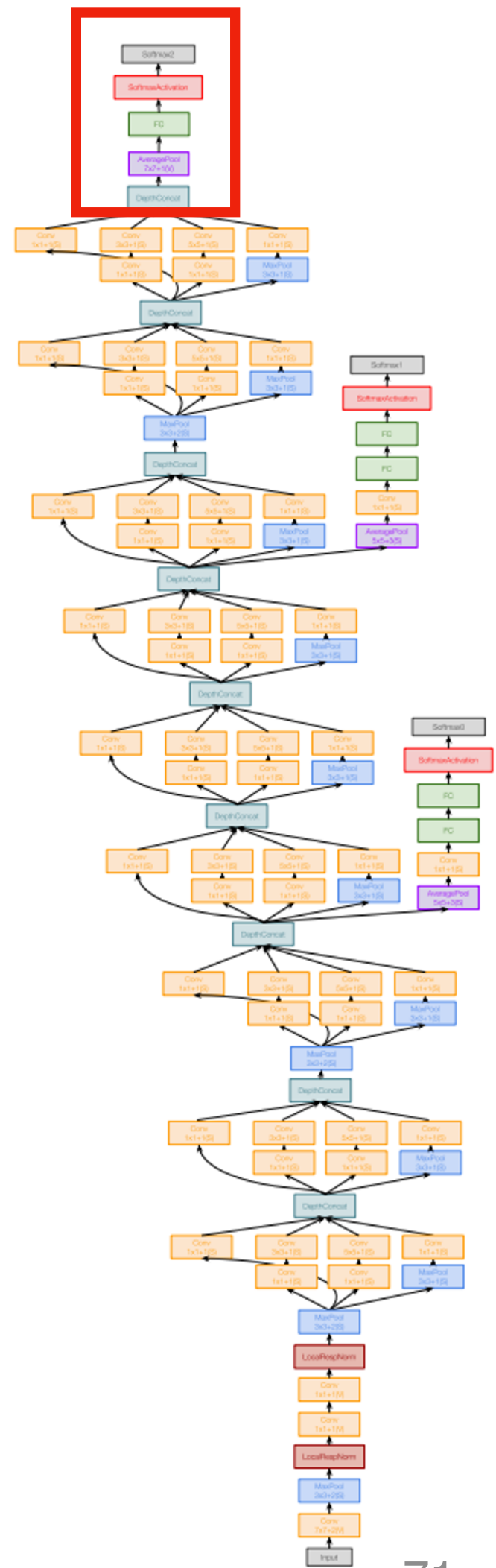


GoogLeNet: Global Average Pooling

No large FC layers at the end!

Instead use **global average pooling** to collapse spatial dimensions, and one linear layer to produce class scores
(Recall VGG-16: Most parameters were in the FC layers!)

| Layer | Input size | | Layer | | | | Output size | | Memory (KB) | Params | Flop (M) |
|----------|------------|-----|---------|--------|--------|-----|-------------|-----|-------------|--------|----------|
| | C | H/W | Filters | Kernel | Stride | Pad | C | H/W | | | |
| avg-pool | 1024 | 7 | | 7 | 1 | 0 | 1024 | 1 | 4 | 0 | 0 |
| fc | 1024 | | 1000 | | | | 1000 | 0 | 0 | 1025 | 1 |





GoogLeNet: Global Average Pooling

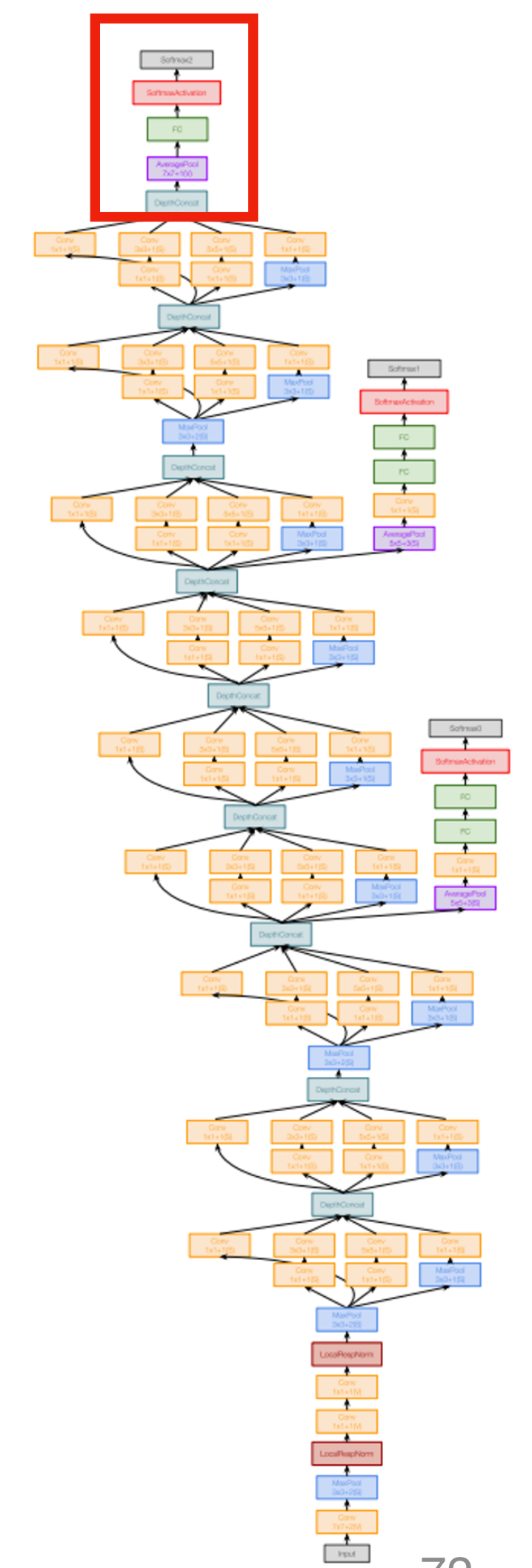
No large FC layers at the end!

Instead use **global average pooling** to collapse spatial dimensions, and one linear layer to produce class scores
(Recall VGG-16: Most parameters were in the FC layers!)

| Layer | Input size | | Layer | | | | Output size | | | | |
|----------|------------|-----|---------|--------|--------|-----|-------------|-----|-------------|--------|----------|
| | C | H/W | Filters | Kernel | Stride | Pad | C | H/W | Memory (KB) | Params | Flop (M) |
| avg-pool | 1024 | 7 | | 7 | 1 | 0 | 1024 | 1 | 4 | 0 | 0 |
| fc | 1024 | | 1000 | | | | 1000 | 0 | 0 | 1025 | 1 |

Compare with VGG-16:

| Layer | Input size | | Layer | | | | Output size | | | | |
|---------|------------|-----|---------|--------|--------|-----|-------------|-----|-------------|--------|----------|
| | C | H/W | Filters | Kernel | Stride | Pad | C | H/W | Memory (KB) | Params | Flop (M) |
| Flatten | 512 | 7 | | | | | 25088 | | 98 | | |
| FC6 | 25088 | | | 4096 | | | 4096 | | 16 | 102760 | 103 |
| FC7 | 4096 | | | 4096 | | | 4096 | | 16 | 16777 | 17 |
| FC8 | 4096 | | | 1000 | | | 1000 | | 4 | 4096 | 4 |

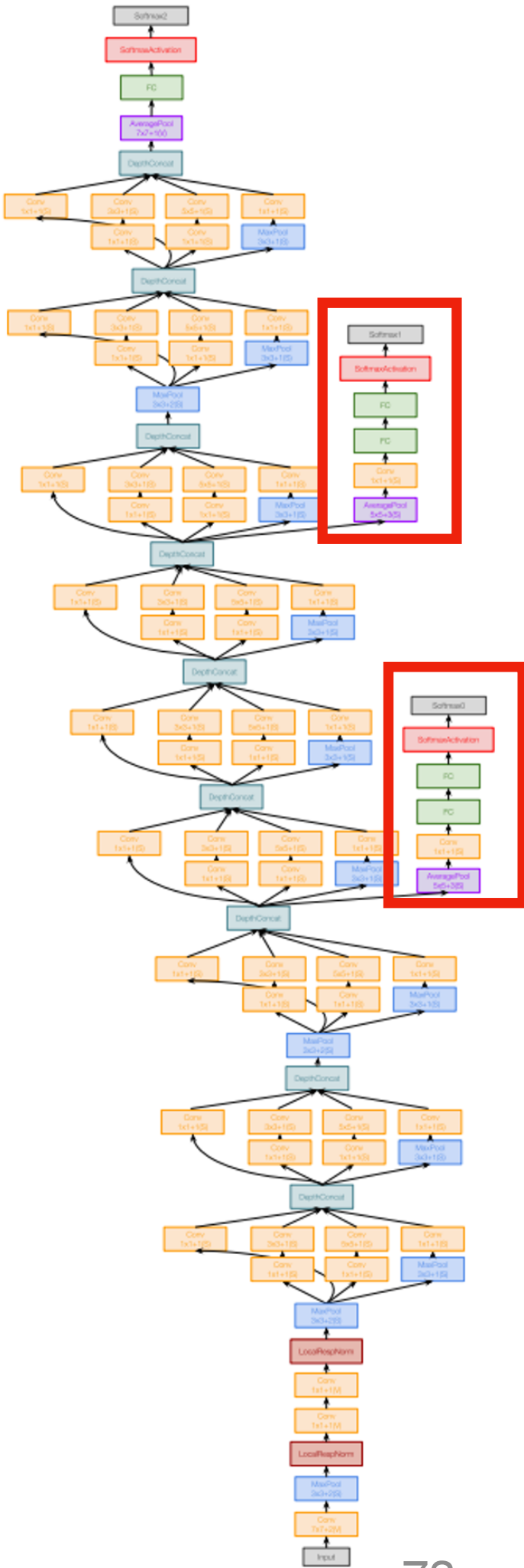


GoogLeNet: Auxiliary Classifiers

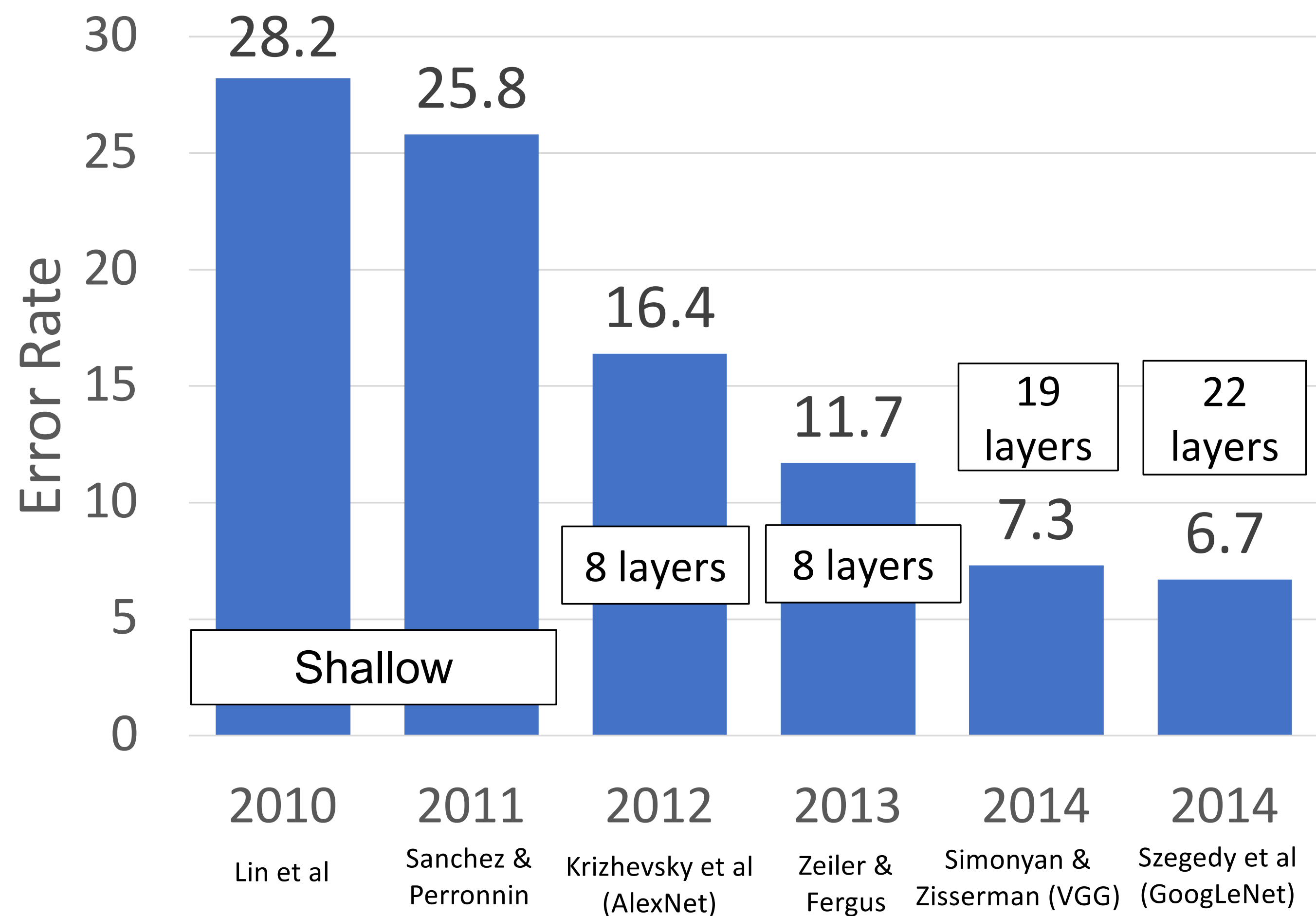
Training using loss at the end of the network didn't work well: Network is too deep, gradients don't propagate cleanly

As a hack, attach "auxiliary classifiers" at several intermediate points in the network that also try to classify the image and receive loss

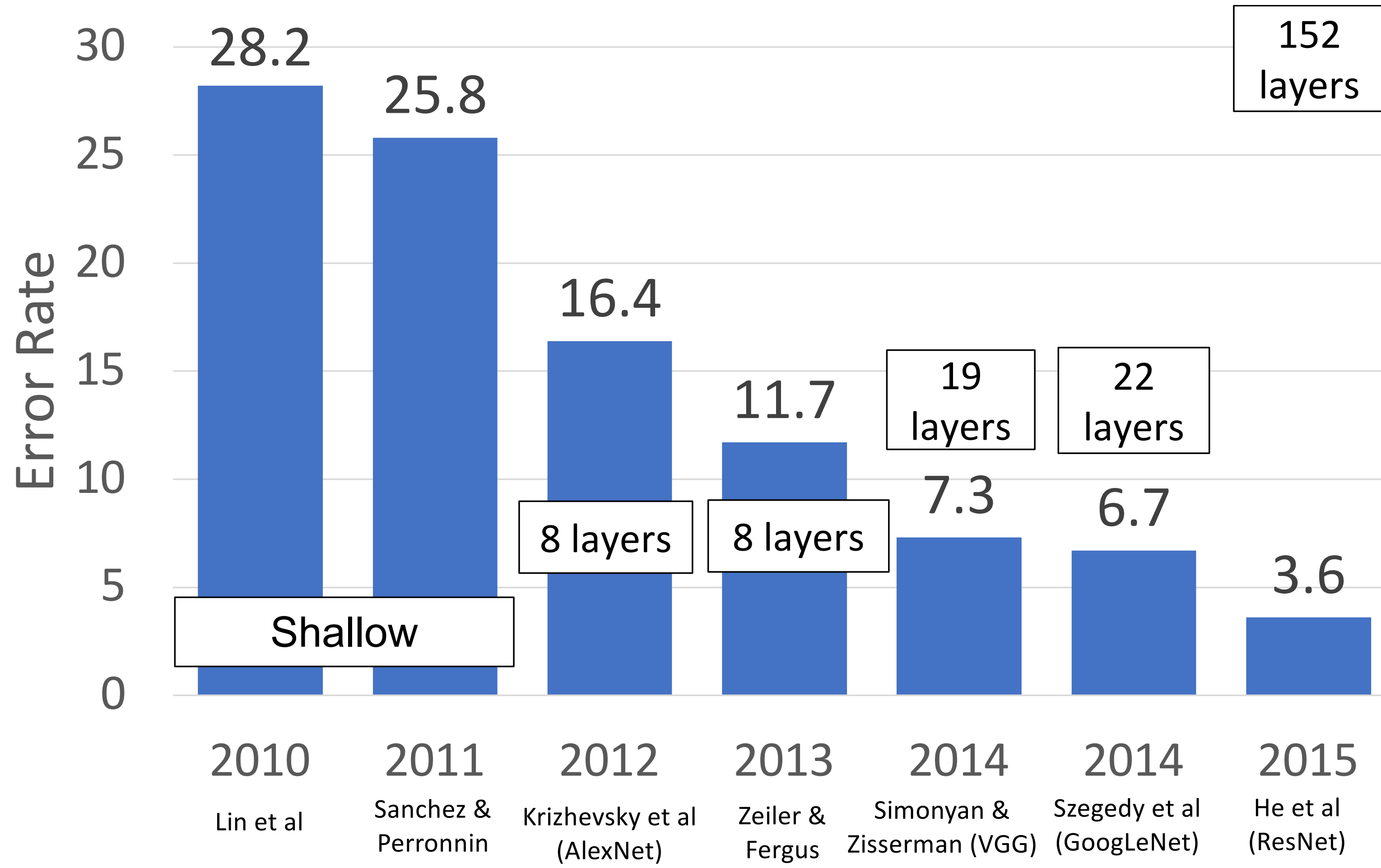
GoogLeNet was before batch normalization! With BatchNorm, we no longer need to use this trick



ImageNet Classification Challenge



ImageNet Classification Challenge

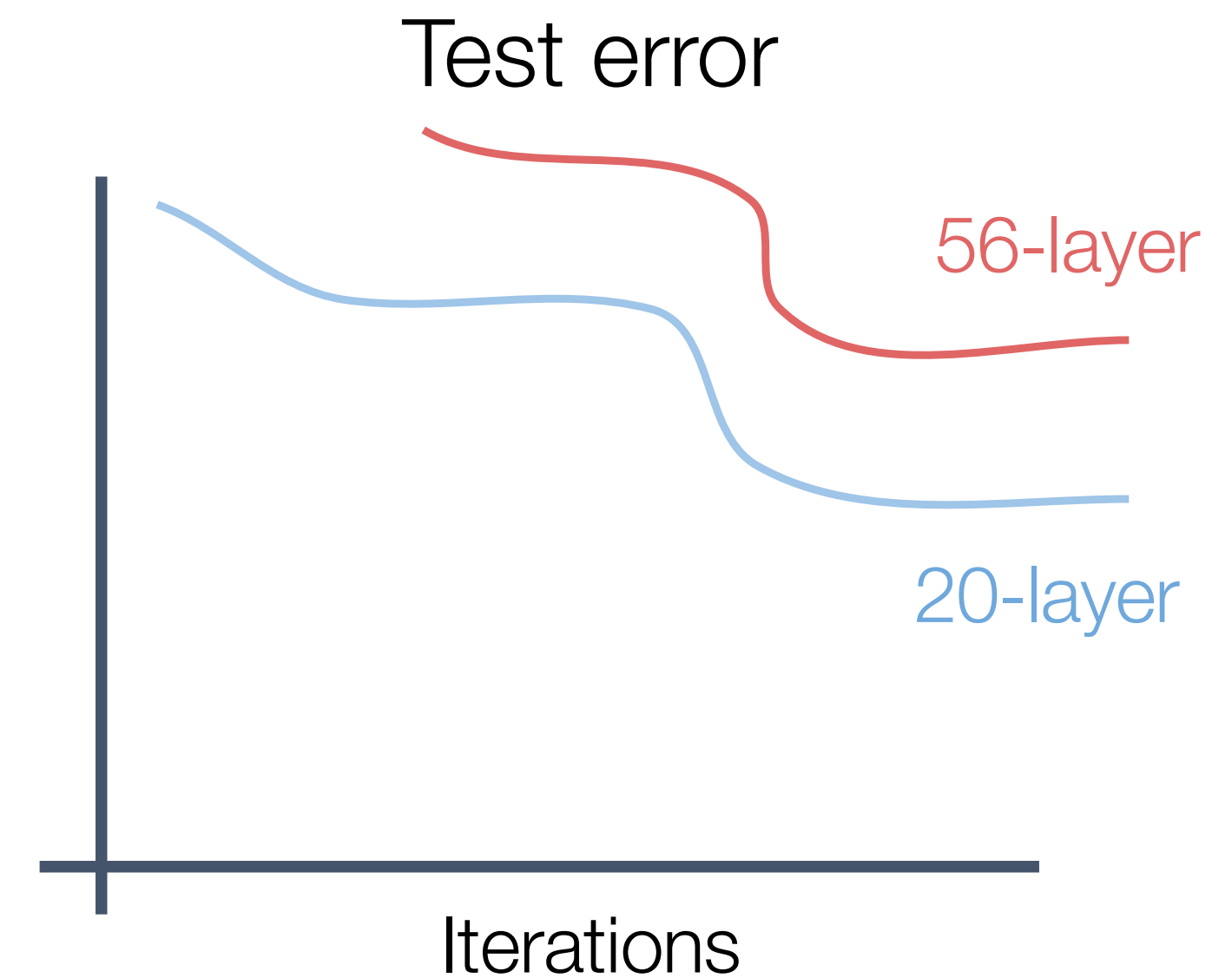


Residual Networks

Once we have Batch Normalization, we can train networks with 10+ layers.
What happens as we go deeper?

Deeper model does worse than shallow model!

Initial guess: Deep model is **overfitting** since it is much bigger than the other model



Residual Networks

Once we have Batch Normalization, we can train networks with 10+ layers.
What happens as we go deeper?



In fact the deep model seems to be **underfitting** since it also performs worse than the shallow model on the training set! It is actually **underfitting**



Residual Networks

A deeper model can emulate a shallower model: copy layers from shallower model, set extra layers to identity

Thus deeper models should do at least as good as shallow models

Hypothesis: This is an optimization problem. Deeper models are harder to optimize, and in particular don't learn identity functions to emulate shallow models



Residual Networks

A deeper model can emulate a shallower model: copy layers from shallower model, set extra layers to identity

Thus deeper models should do at least as good as shallow models

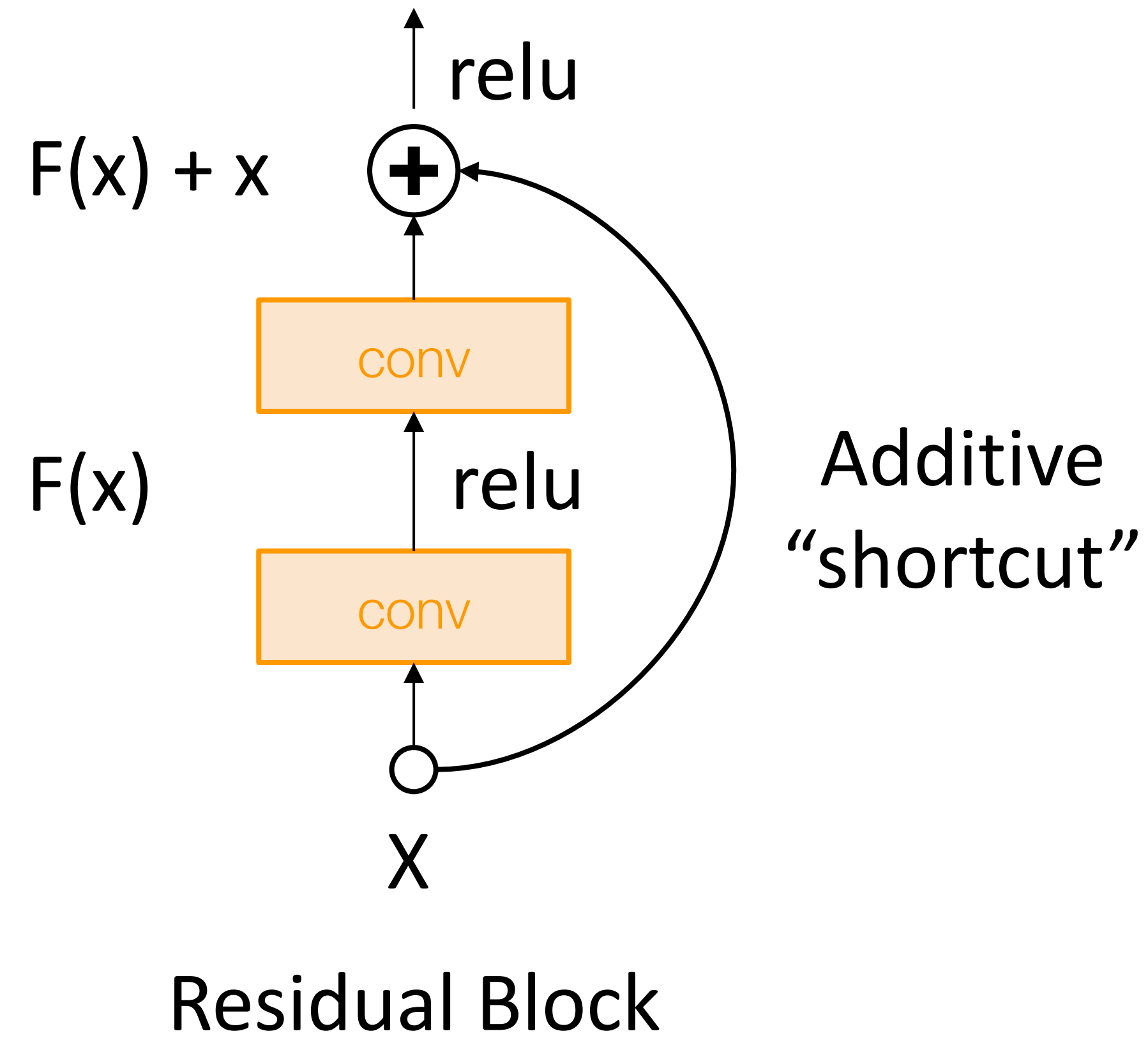
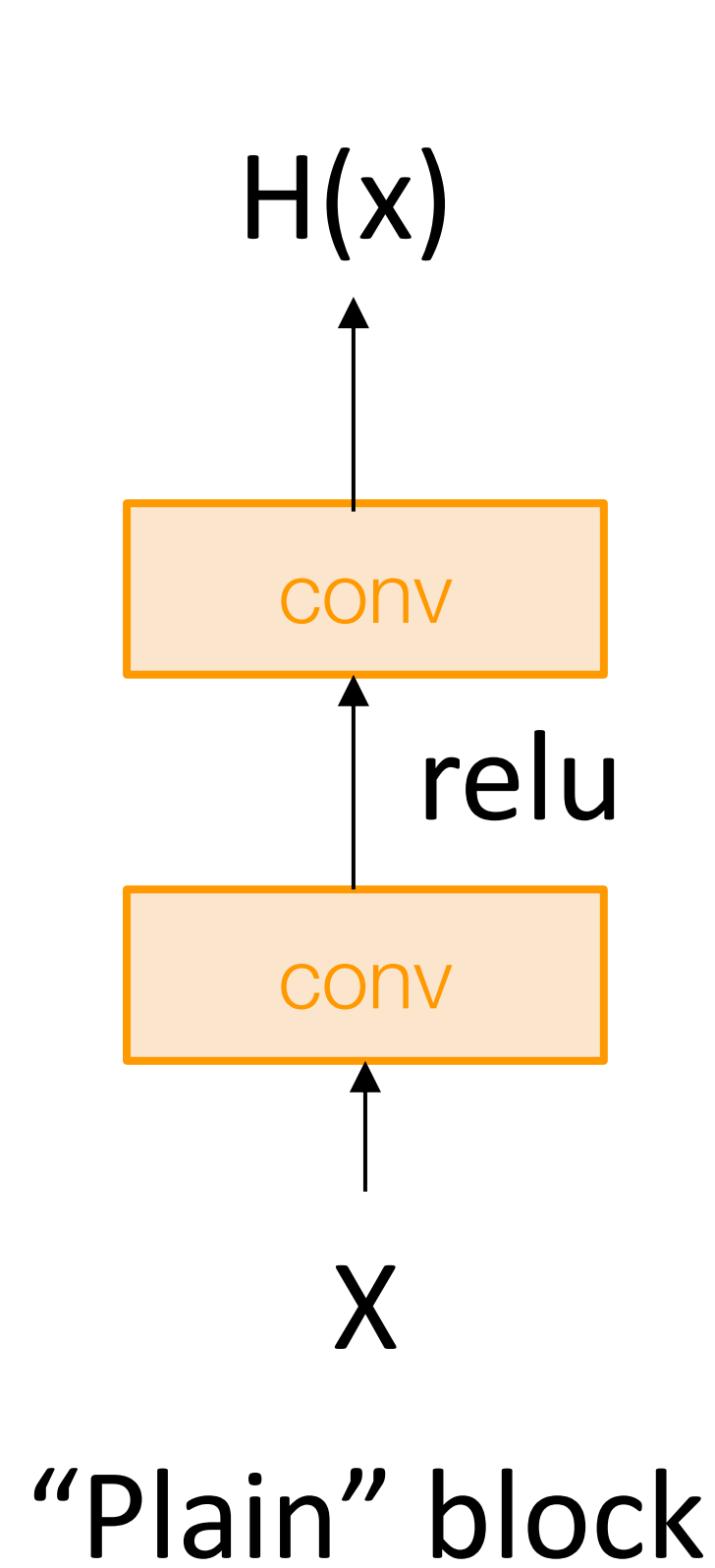
Hypothesis: This is an optimization problem. Deeper models are harder to optimize, and in particular don't learn identity functions to emulate shallow models

Solution: Change the network so learning identity functions with extra layers is easy!



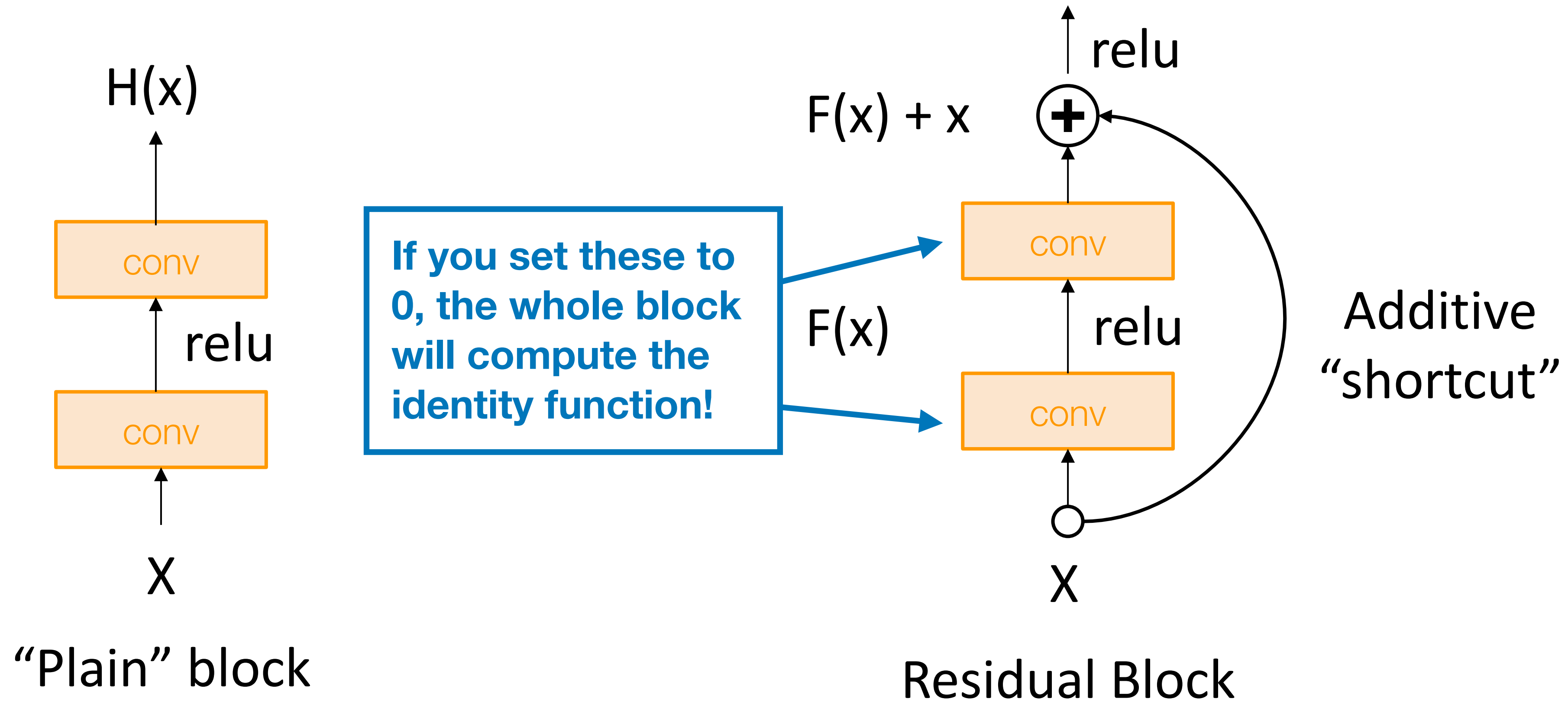
Residual Networks

Solution: Change the network so learning identity functions with extra layers is easy!



Residual Networks

Solution: Change the network so learning identity functions with extra layers is easy!

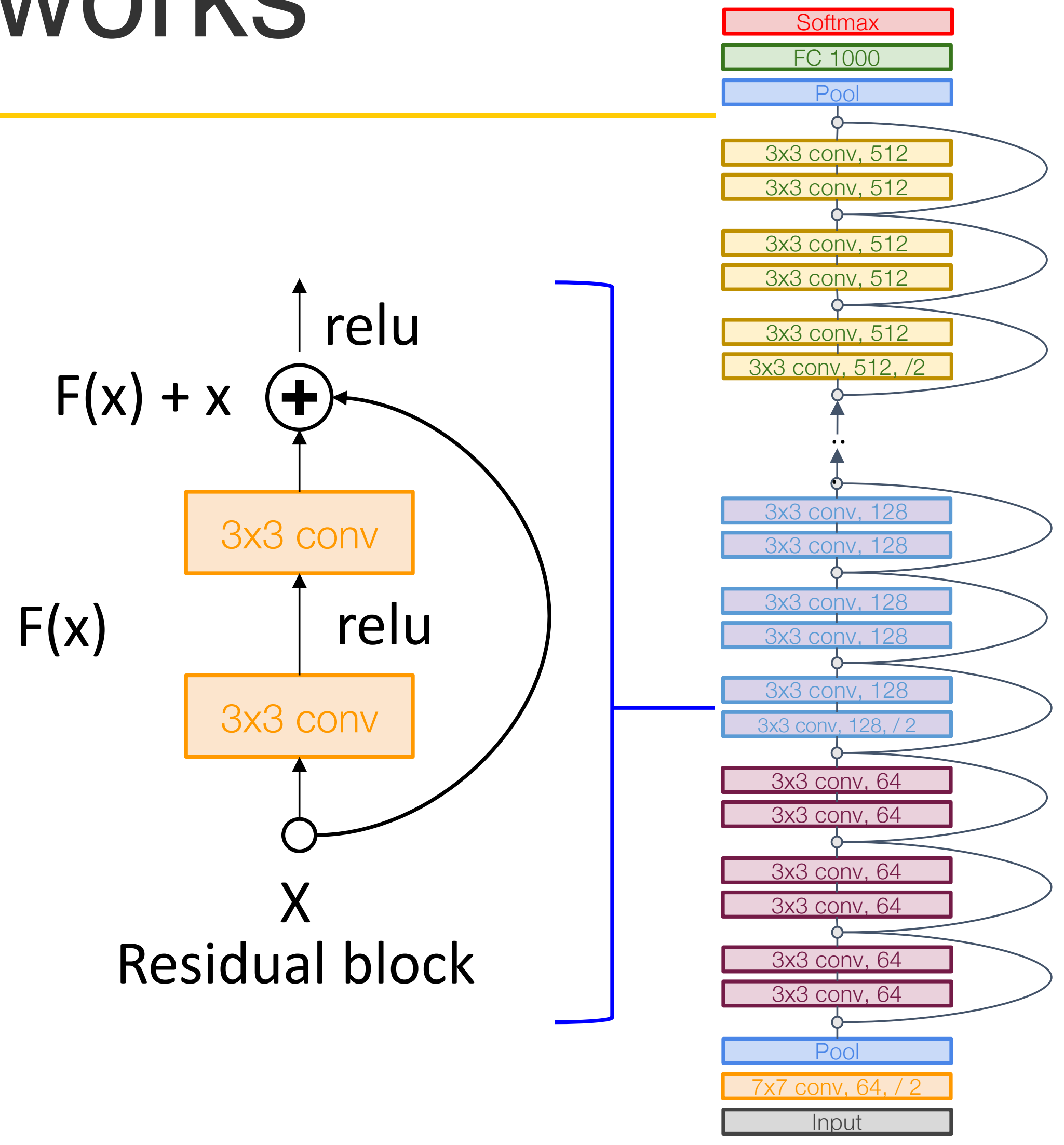


Residual Networks

A residual network is a stack of many residual blocks

Regular design, like VGG: each residual block has two 3x3 conv

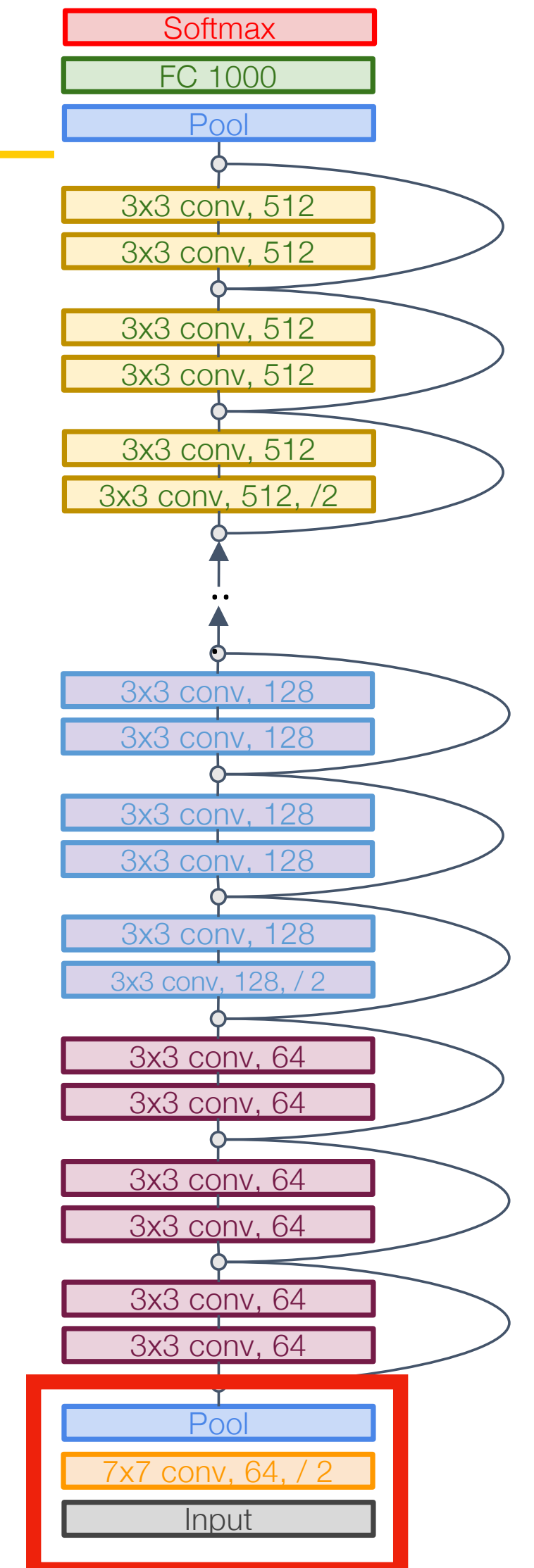
Network is divided into **stages**: the first block of each stage halves the resolution (with stride-2 conv) and doubles the number of channels



Residual Networks

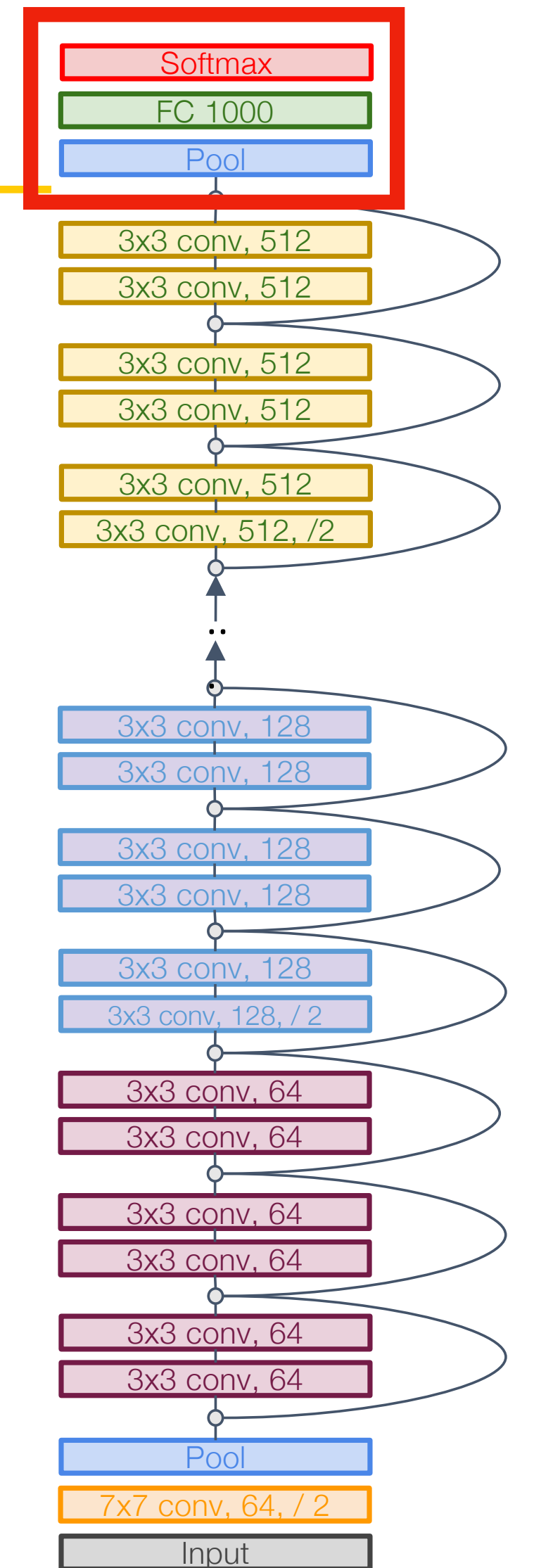
Uses the same aggressive **stem** as GoogleNet to downsample the input 4x before applying residual blocks:

| Layer | Input size | | Layer | | | | Output size | | | Memory (KB) | Params | Flop (M) |
|----------|------------|-----|---------|--------|--------|-----|-------------|-----|------|-------------|--------|----------|
| | C | H/W | Filters | Kernel | Stride | Pad | C | H/W | | | | |
| Conv | 3 | 224 | 64 | 7 | 2 | 3 | 64 | 112 | 3136 | 9 | 118 | |
| Max-pool | 64 | 112 | | 3 | 2 | 1 | 64 | 56 | 784 | 0 | 2 | |



Residual Networks

Like GoogLeNet, no big fully-connected-layers: Instead use **global average pooling** and a single linear layer at the end



Residual Networks

ResNet-18:

Stem: 1 conv layer

Stage 1 (C=64): 2 res. block = 4 conv

Stage 2 (C=128): 2 res. block = 4 conv

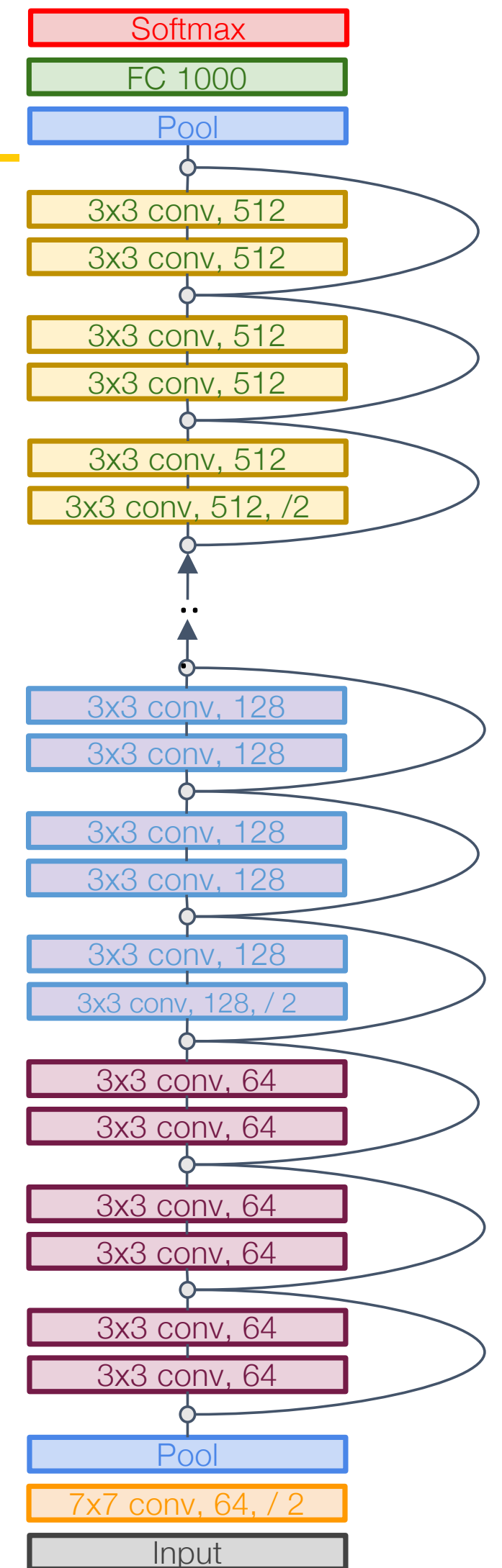
Stage 3 (C=256): 2 res. block = 4 conv

Stage 4 (C=512): 2 res. block = 4 conv

Linear

ImageNet top-5 error: 10.92

GFLOP: 1.8



Residual Networks

ResNet-18:

Stem: 1 conv layer

Stage 1 (C=64): 2 res. block = 4 conv

Stage 2 (C=128): 2 res. block = 4 conv

Stage 3 (C=256): 2 res. block = 4 conv

Stage 4 (C=512): 2 res. block = 4 conv

Linear

ImageNet top-5 error: 10.92

GFLOP: 1.8

ResNet-34:

Stem: 1 conv layer

Stage 1: 3 res. block = 6 conv

Stage 2: 4 res. block = 8 conv

Stage 3: 6 res. block = 12 conv

Stage 4: 3 res. block = 6 conv

Linear

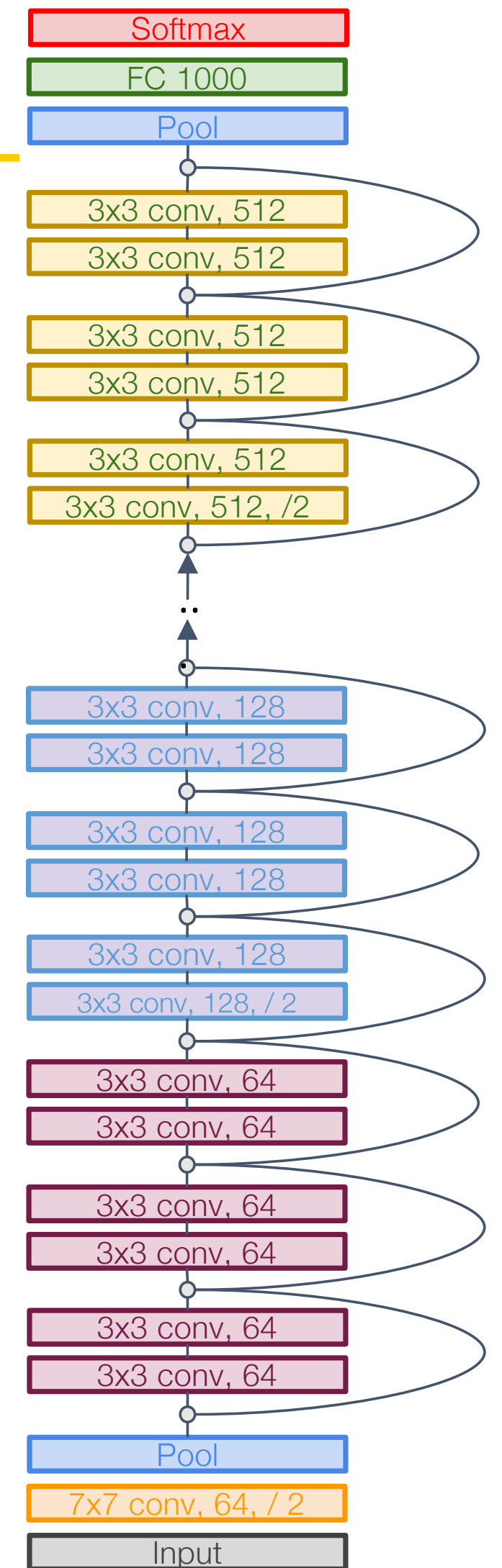
ImageNet top-5 error: 8.58

GFLOP: 3.6

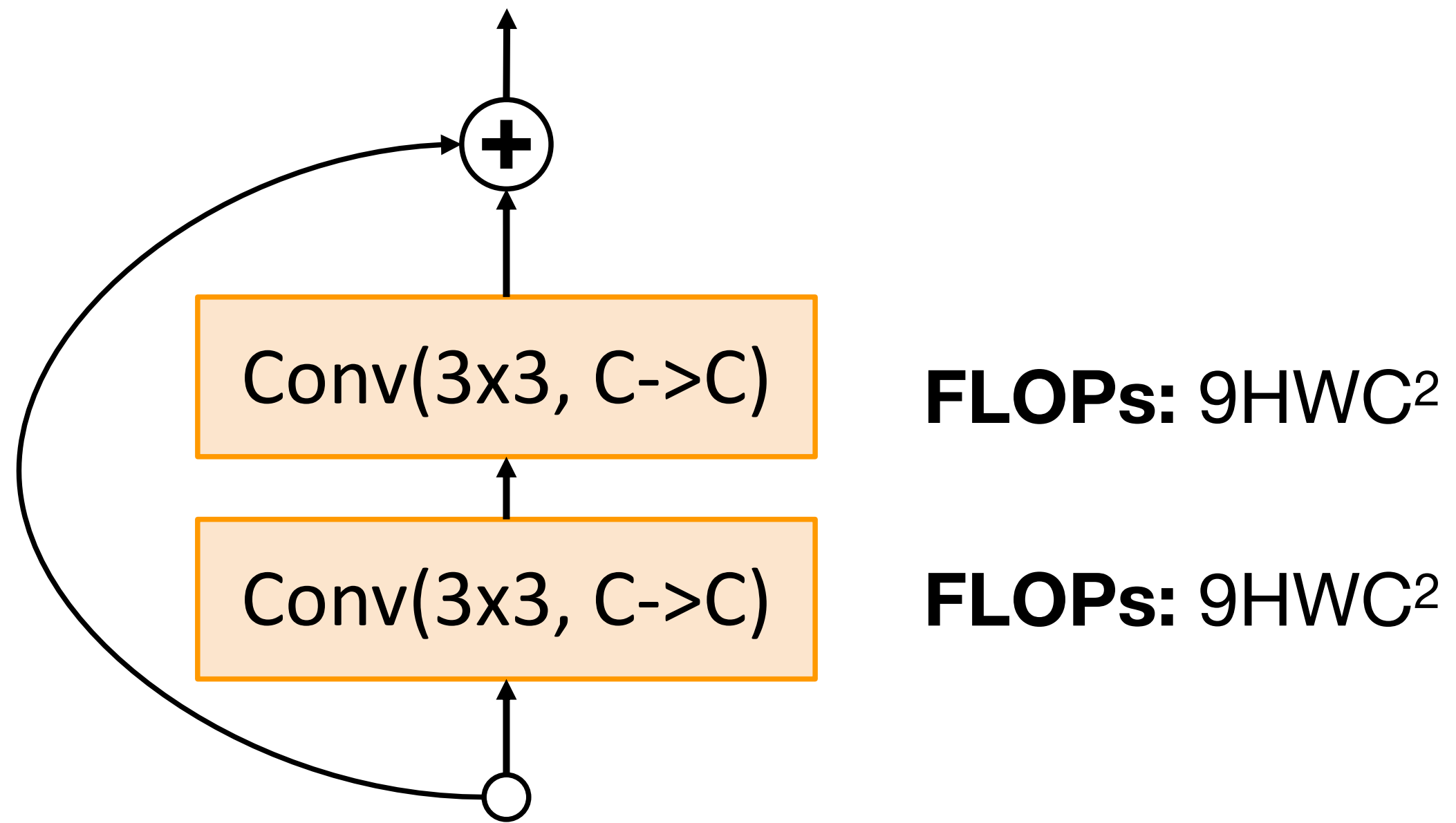
VGG-16:

ImageNet top-5 error: 9.62

GFLOP: 13.6



Residual Networks: Basic Block

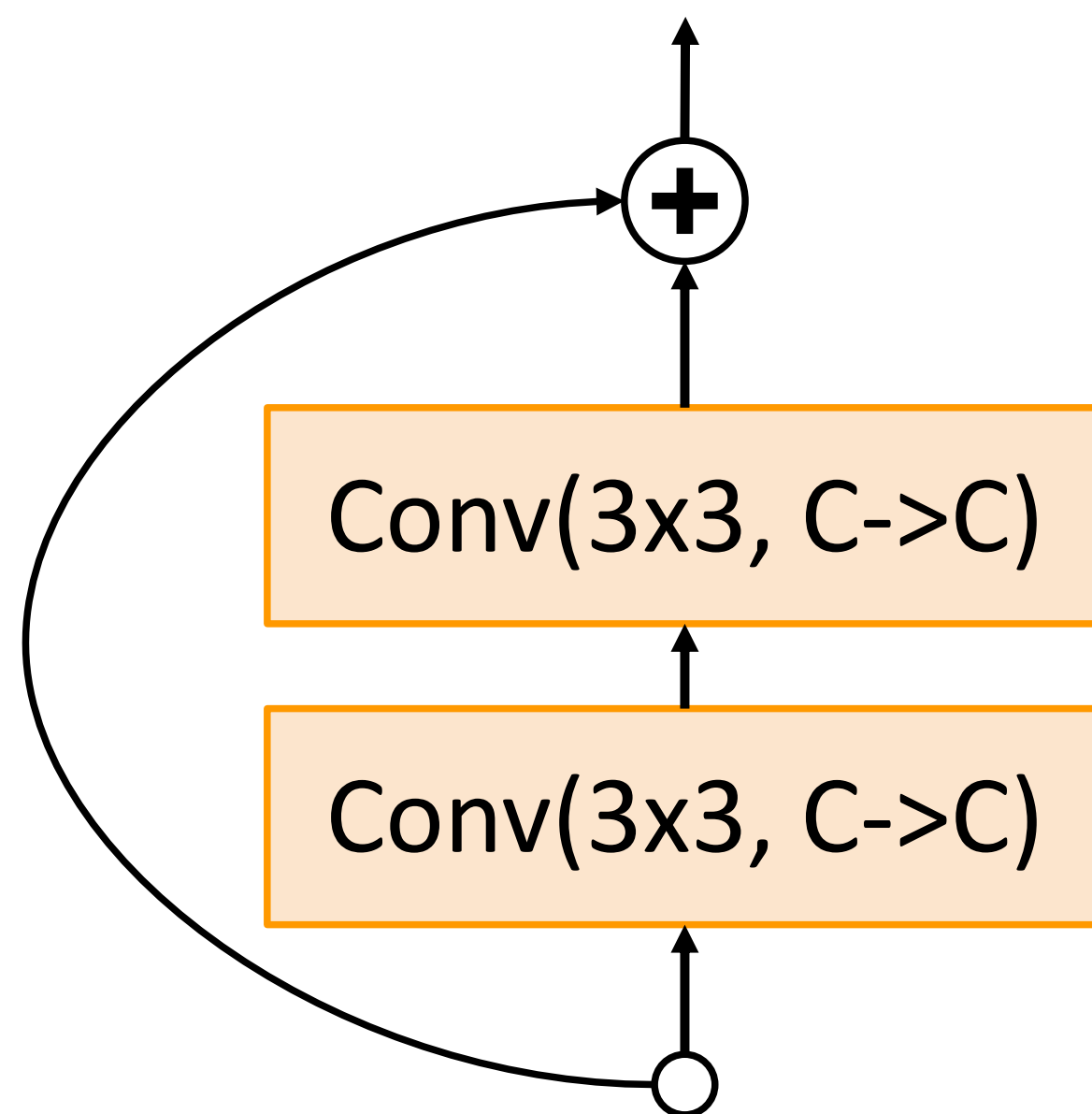


“Basic”
Residual block

Total FLOPs:
 $18HWC^2$



Residual Networks: Bottleneck Block

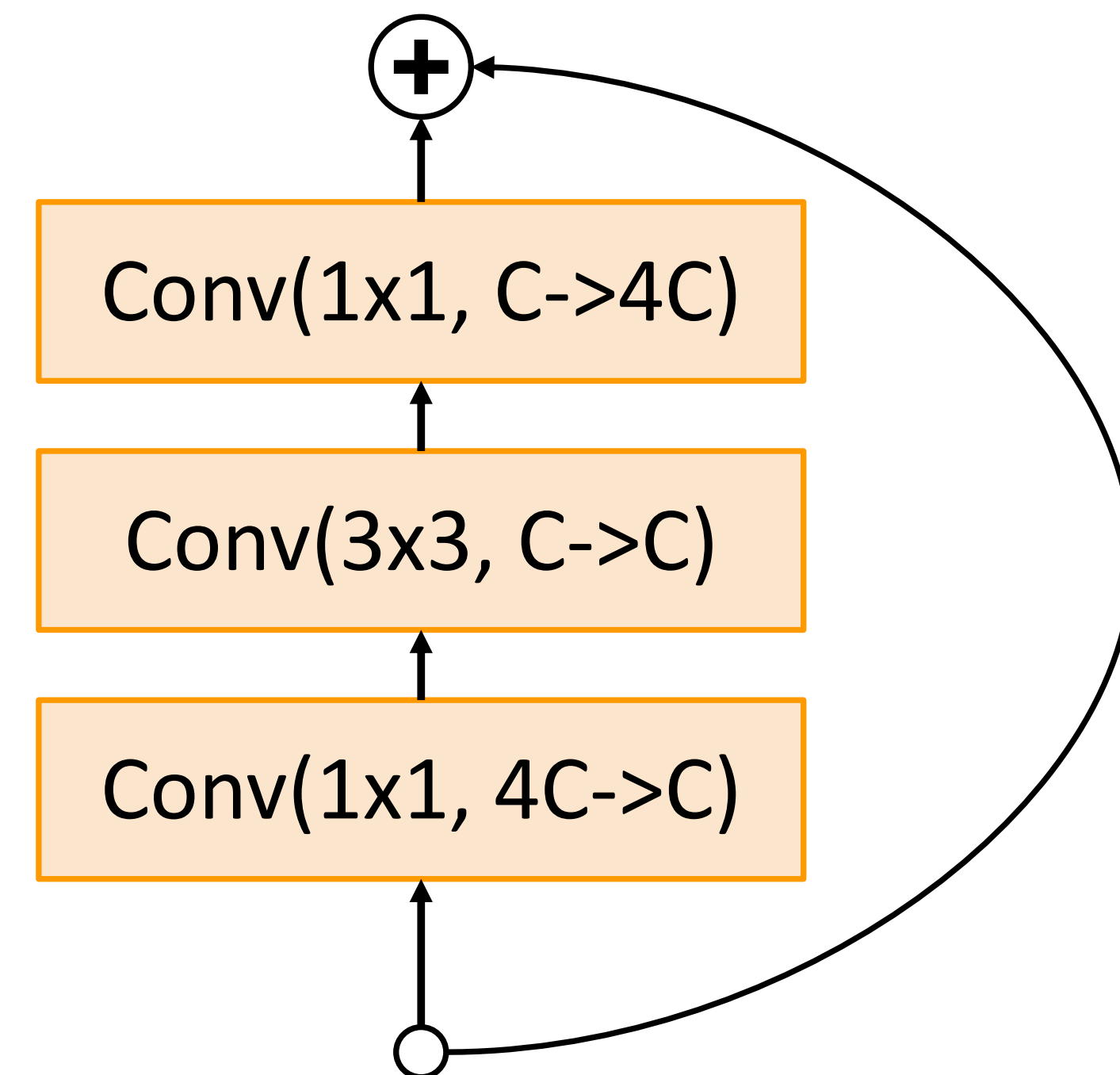


“Basic”
Residual block

FLOPs: $9HWC^2$

FLOPs: $9HWC^2$

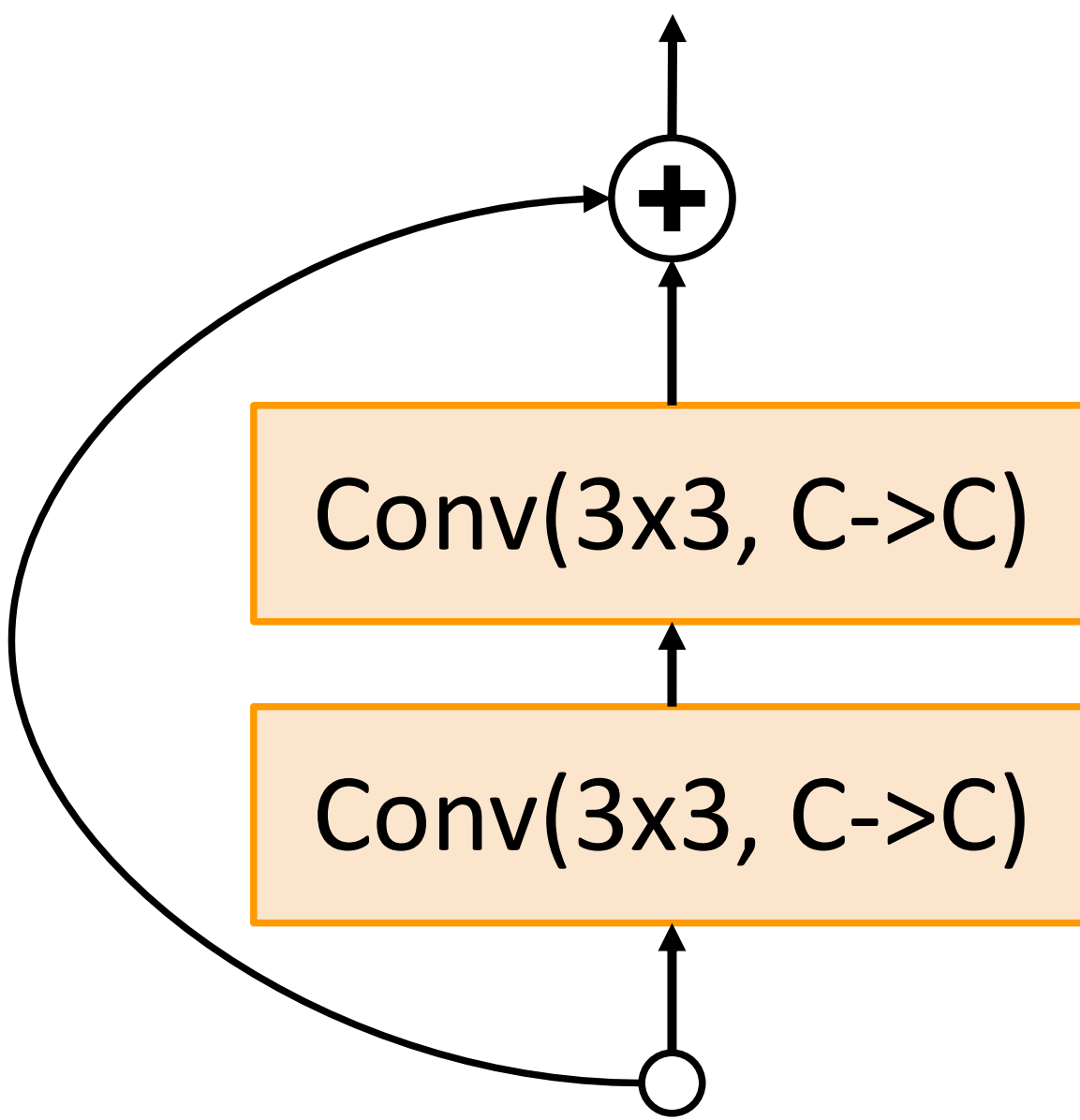
Total FLOPs:
 $18HWC^2$



“Bottleneck”
Residual block

Residual Networks: Bottleneck Block

More layers, less computational cost!



“Basic”
Residual block

FLOPs: $9HWC^2$

FLOPs: $9HWC^2$

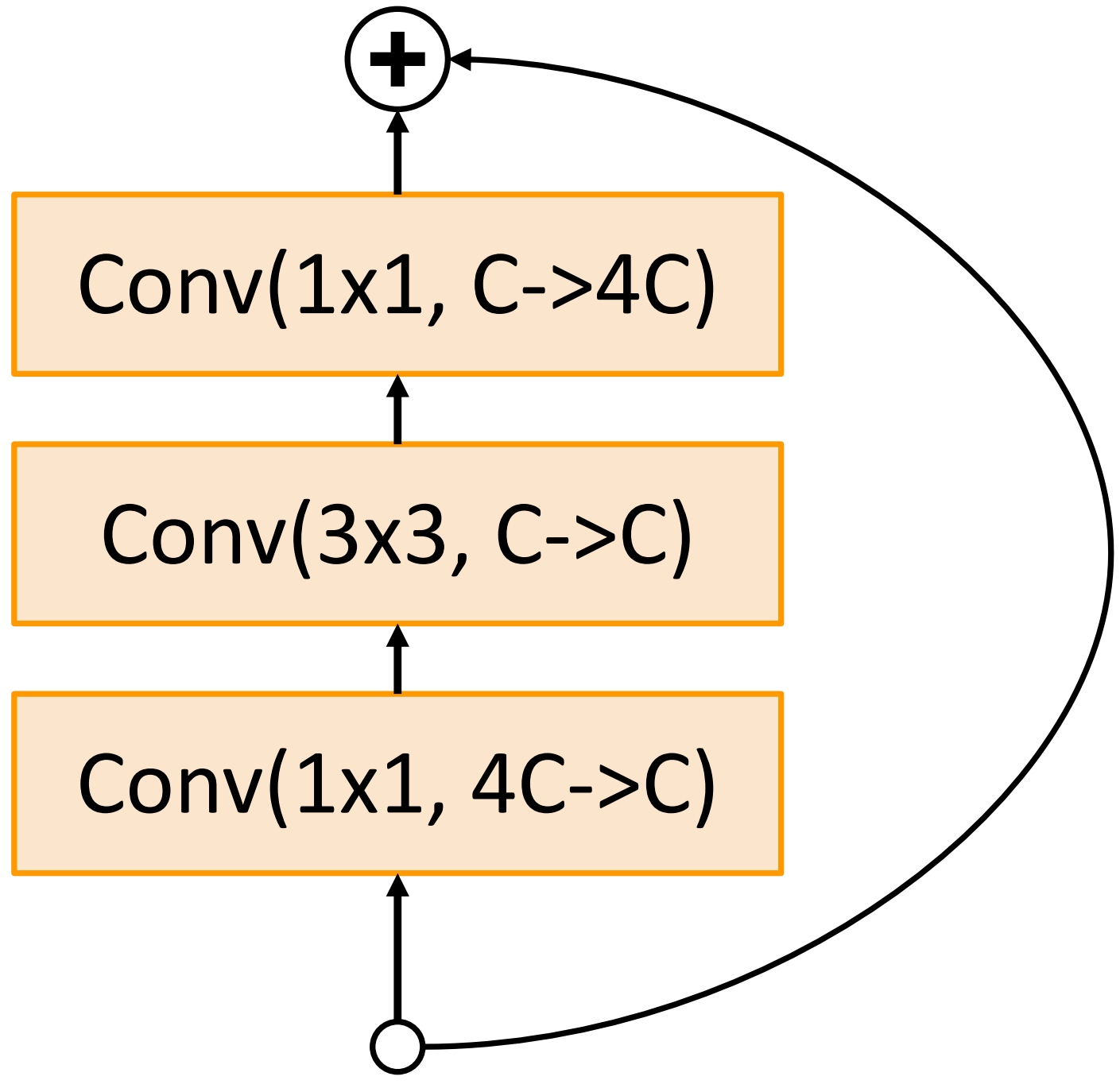
Total FLOPs:
 $18HWC^2$

FLOPs: $4HWC^2$

FLOPs: $9HWC^2$

FLOPs: $4HWC^2$

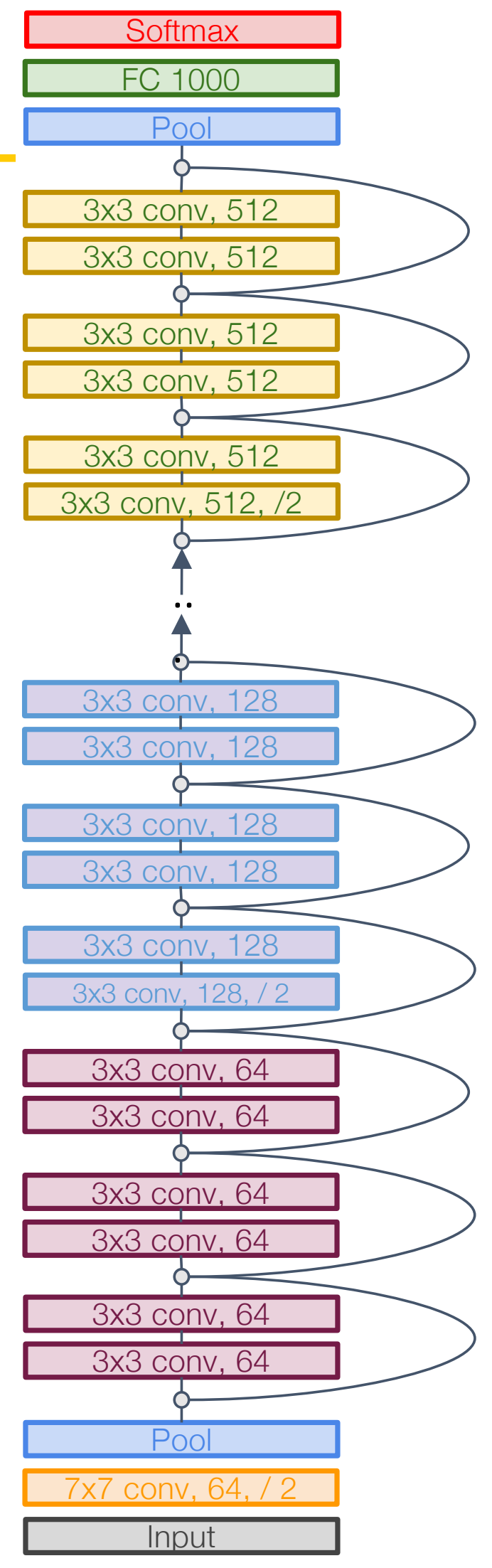
Total FLOPs:
 $17HWC^2$



“Bottleneck”
Residual block



Residual Networks



| | | | Stage 1 | | Stage 2 | | Stage 3 | | Stage 4 | | | | |
|------------------|------------|-------------|---------|--------|---------|--------|---------|--------|---------|--------|-----------|-------|-----------|
| | Block type | Stem layers | Blocks | Layers | Blocks | Layers | Blocks | Layers | Blocks | Layers | FC Layers | GFLOP | Image Net |
| ResNet-18 | Basic | 1 | 2 | 4 | 2 | 4 | 2 | 4 | 2 | 4 | 1 | 1.8 | 10.92 |
| ResNet-34 | Basic | 1 | 3 | 6 | 4 | 8 | 6 | 12 | 3 | 6 | 1 | 3.6 | 8.58 |

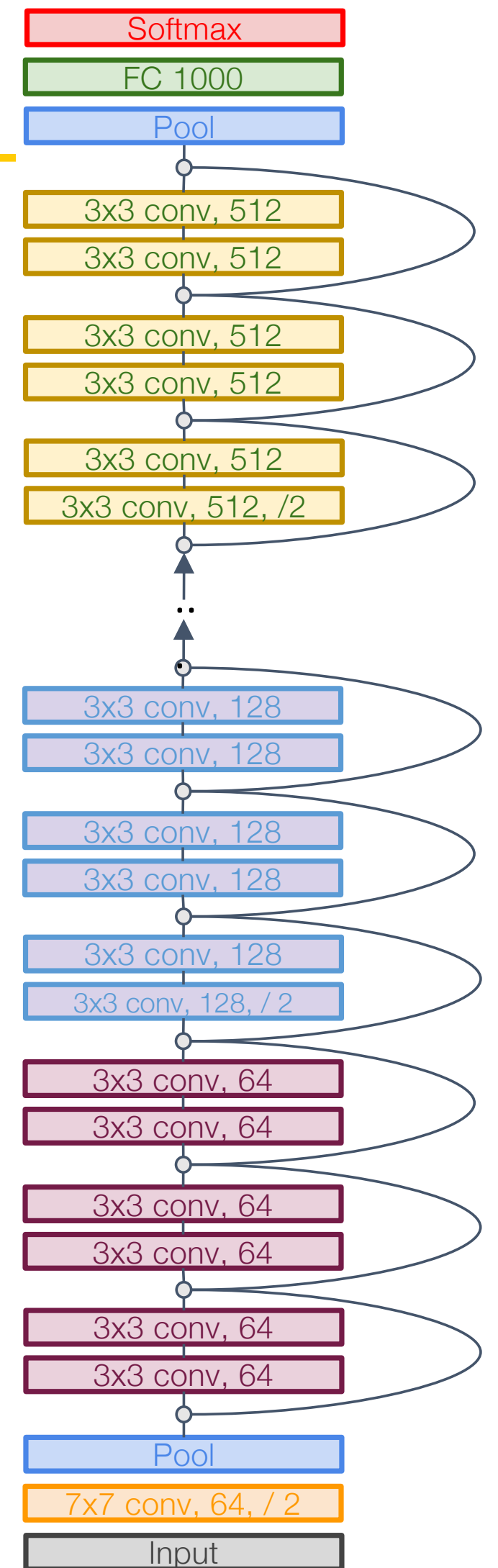


He et al, "Deep Residual Learning for Image Recognition", CVPR 2016
 Error rates are 224x224 single-crop testing, reported by [torchvision](#)

Residual Networks

ResNet-50 is the same as ResNet-34, but replaces Basic blocks with Bottleneck Blocks. This is a great baseline architecture for many tasks even today!

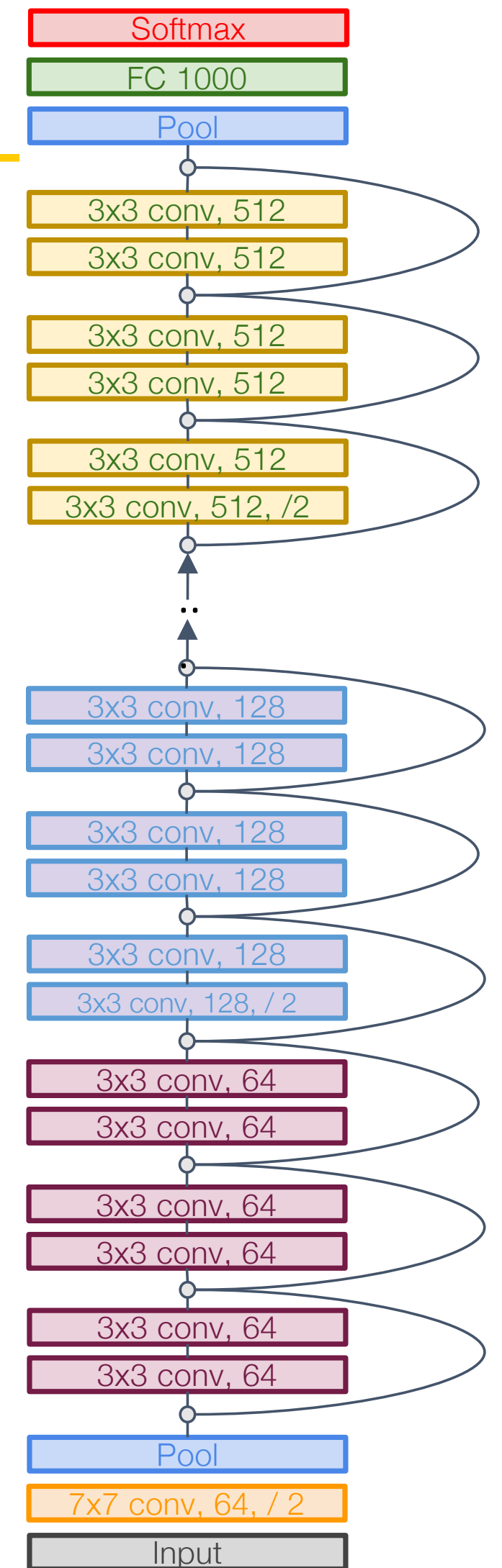
| | | | Stage 1 | | Stage 2 | | Stage 3 | | Stage 4 | | | | |
|------------------|------------|-------------|---------|--------|---------|--------|---------|--------|---------|--------|-----------|-------|-----------|
| | Block type | Stem layers | Blocks | Layers | Blocks | Layers | Blocks | Layers | Blocks | Layers | FC Layers | GFLOP | Image Net |
| ResNet-18 | Basic | 1 | 2 | 4 | 2 | 4 | 2 | 4 | 2 | 4 | 1 | 1.8 | 10.92 |
| ResNet-34 | Basic | 1 | 3 | 6 | 4 | 8 | 6 | 12 | 3 | 6 | 1 | 3.6 | 8.58 |
| ResNet-50 | Bottle | 1 | 3 | 9 | 4 | 12 | 6 | 18 | 3 | 9 | 1 | 3.8 | 7.13 |



Residual Networks

Deeper ResNet-101 and ResNet-152 models are more accurate, but also more computationally heavy

| | | | Stage 1 | | Stage 2 | | Stage 3 | | Stage 4 | | | | |
|-------------------|------------|-------------|---------|--------|---------|--------|---------|--------|---------|--------|-----------|-------|-----------|
| | Block type | Stem layers | Blocks | Layers | Blocks | Layers | Blocks | Layers | Blocks | Layers | FC Layers | GFLOP | Image Net |
| ResNet-18 | Basic | 1 | 2 | 4 | 2 | 4 | 2 | 4 | 2 | 4 | 1 | 1.8 | 10.92 |
| ResNet-34 | Basic | 1 | 3 | 6 | 4 | 8 | 6 | 12 | 3 | 6 | 1 | 3.6 | 8.58 |
| ResNet-50 | Bottle | 1 | 3 | 9 | 4 | 12 | 6 | 18 | 3 | 9 | 1 | 3.8 | 7.13 |
| ResNet-101 | Bottle | 1 | 3 | 9 | 4 | 12 | 23 | 69 | 3 | 9 | 1 | 7.6 | 6.44 |
| ResNet-152 | Bottle | 1 | 3 | 9 | 8 | 24 | 36 | 108 | 3 | 9 | 1 | 11.3 | 5.94 |



Residual Networks

- Able to train very deep networks
- Deeper networks do better than shallow networks (as expected)
- Swept 1st place in all ILSVRC and COCO 2015 competitions
- Still widely used today

MSRA @ ILSVRC & COCO 2015 Competitions

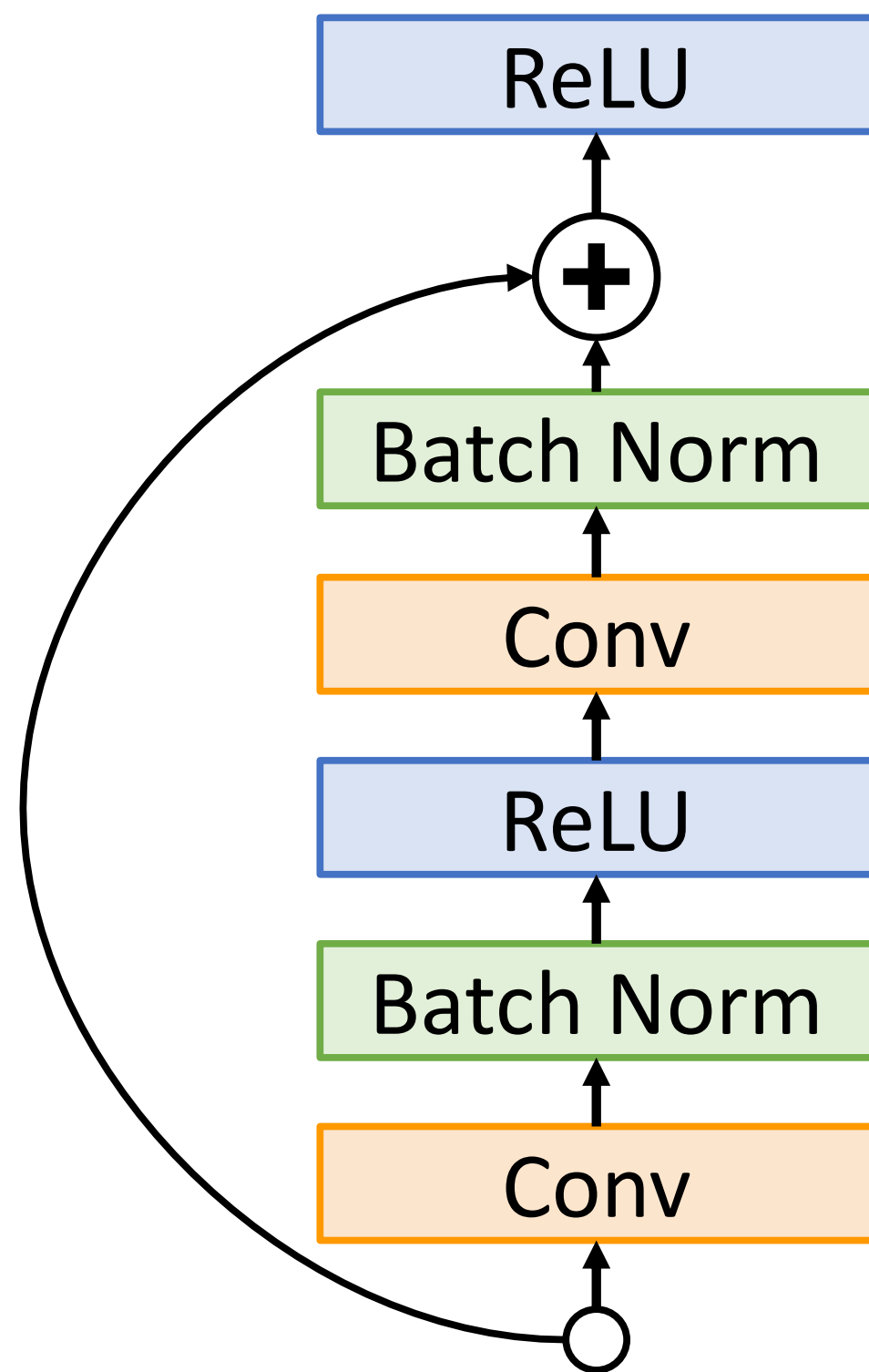
- **1st places in all five main tracks**

- ImageNet Classification: “*Ultra-deep*” (quote Yann) **152-layer** nets
- ImageNet Detection: **16%** better than 2nd
- ImageNet Localization: **27%** better than 2nd
- COCO Detection: **11%** better than 2nd
- COCO Segmentation: **12%** better than 2nd



Improving Residual Networks: Block Design

Original ResNet block



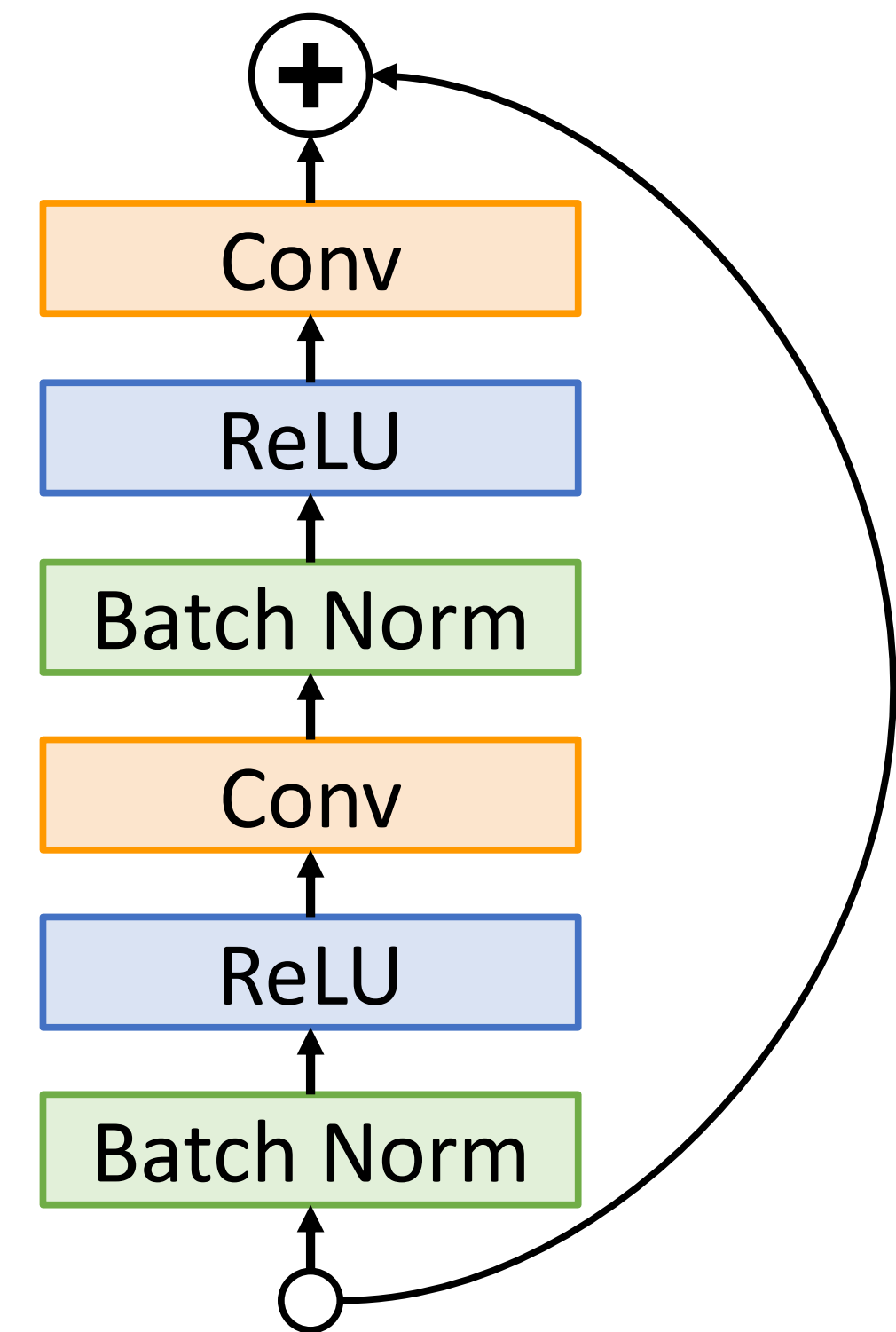
Note ReLU **after** residual:

Cannot actually learn identity function since outputs are nonnegative!

Note ReLU **inside** residual:

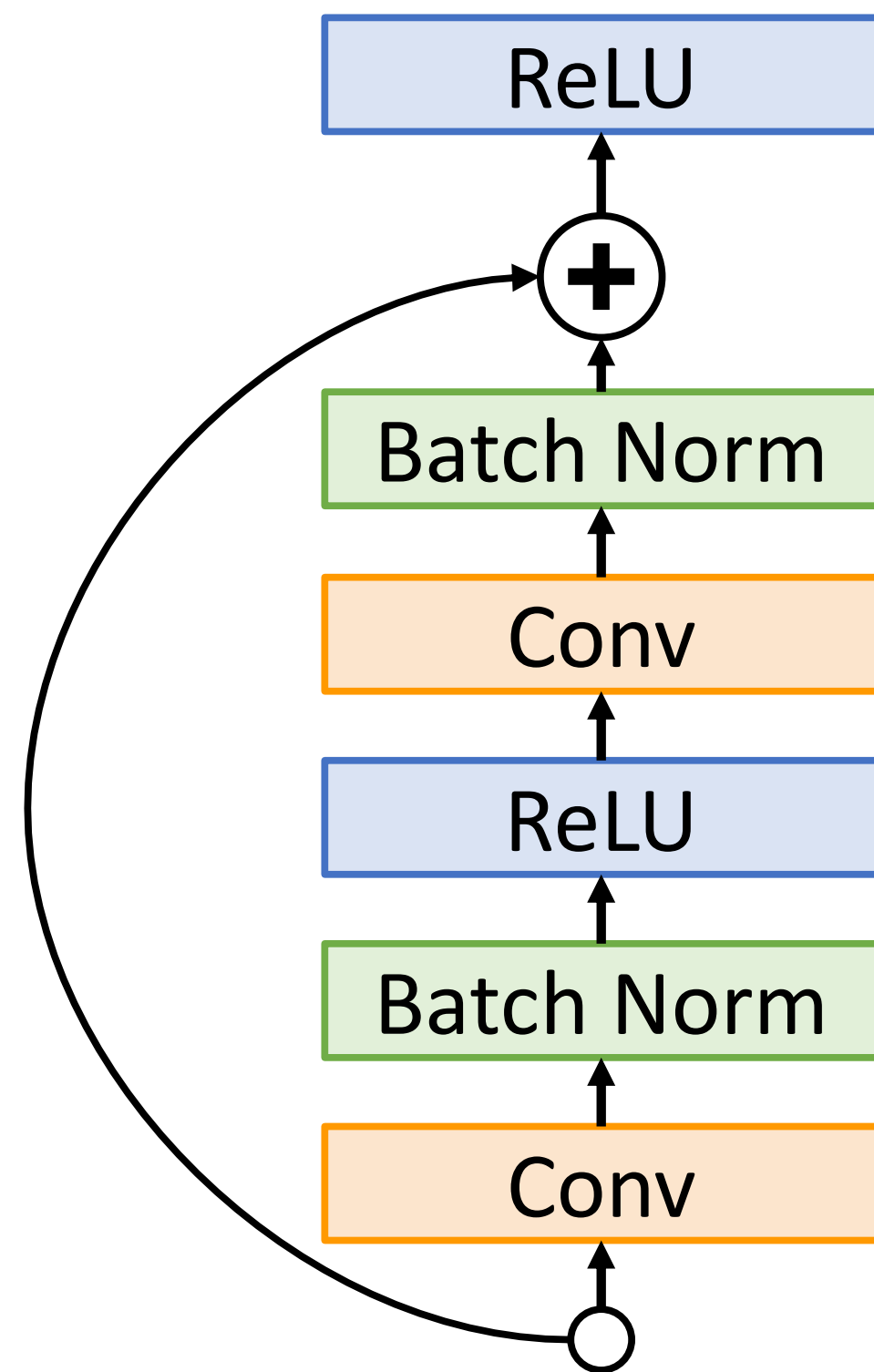
Can learn identity function by setting Conv weights to zero

“Pre-Activation” ResNet Block



Improving Residual Networks: Block Design

Original ResNet block



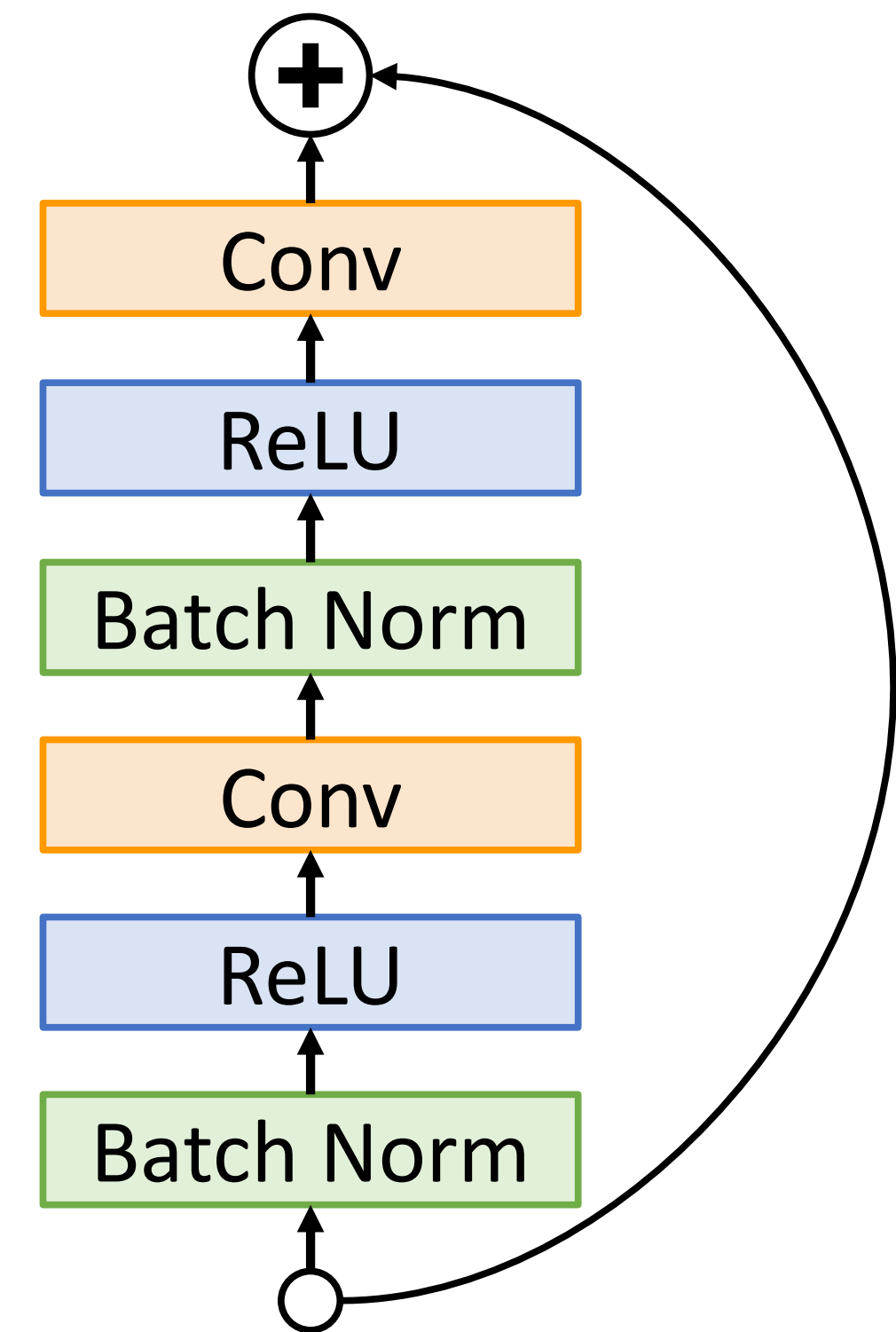
Slight improvement in accuracy
(ImageNet top-1 error)

ResNet-152: 21.3 vs **21.1**

ResNet-200: 21.8 vs **20.7**

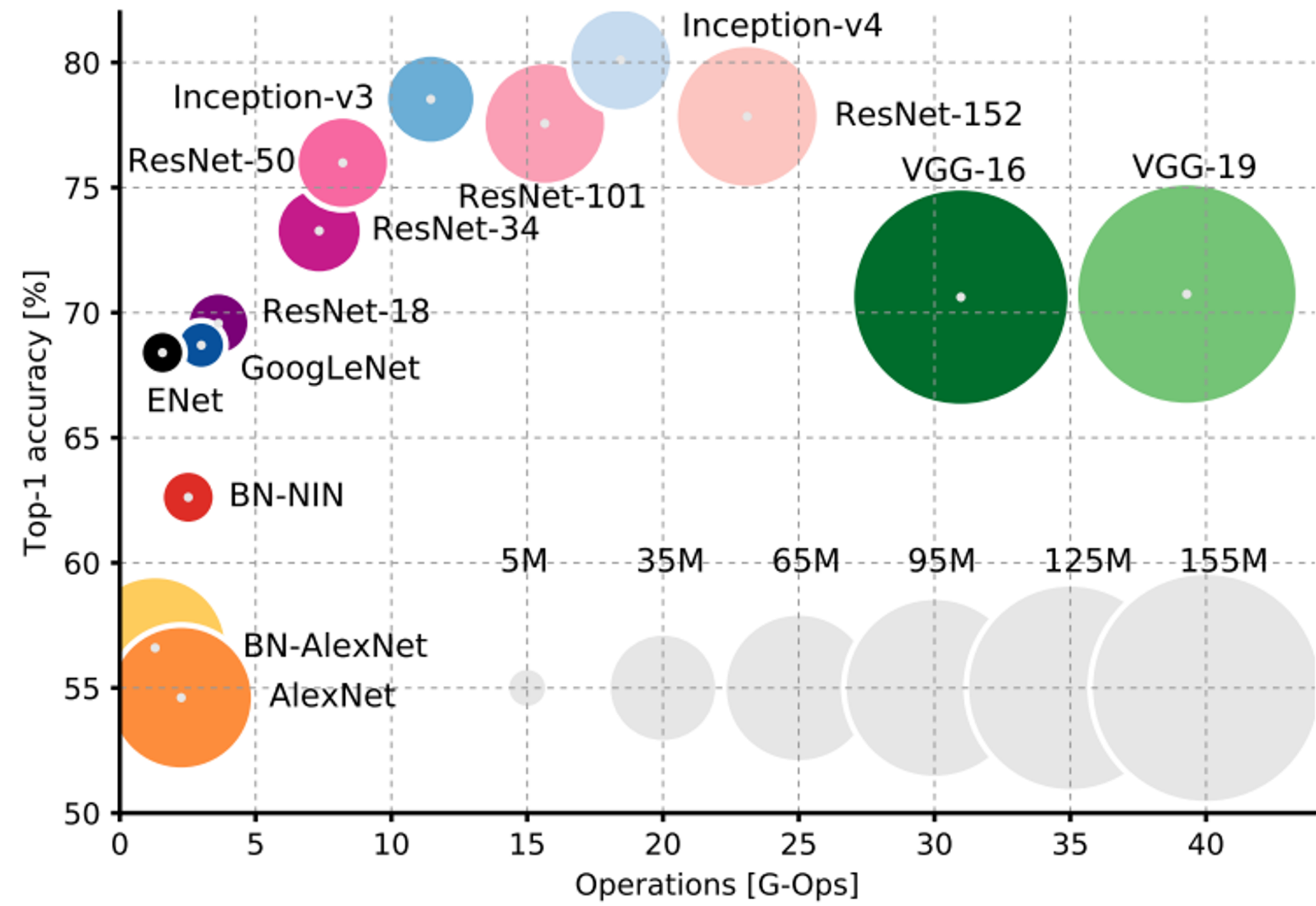
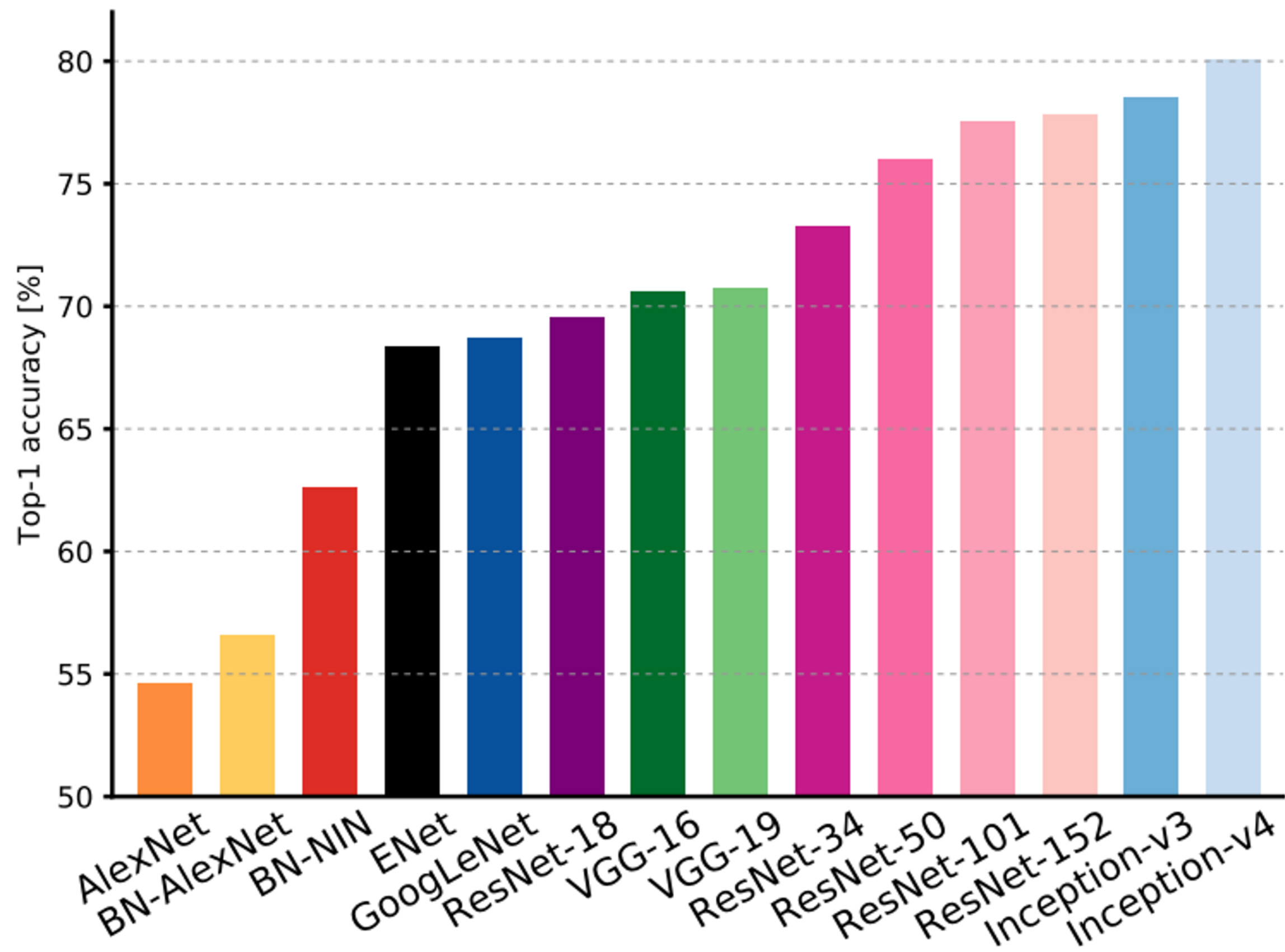
Not actually used that much in
practice

“Pre-Activation” ResNet Block





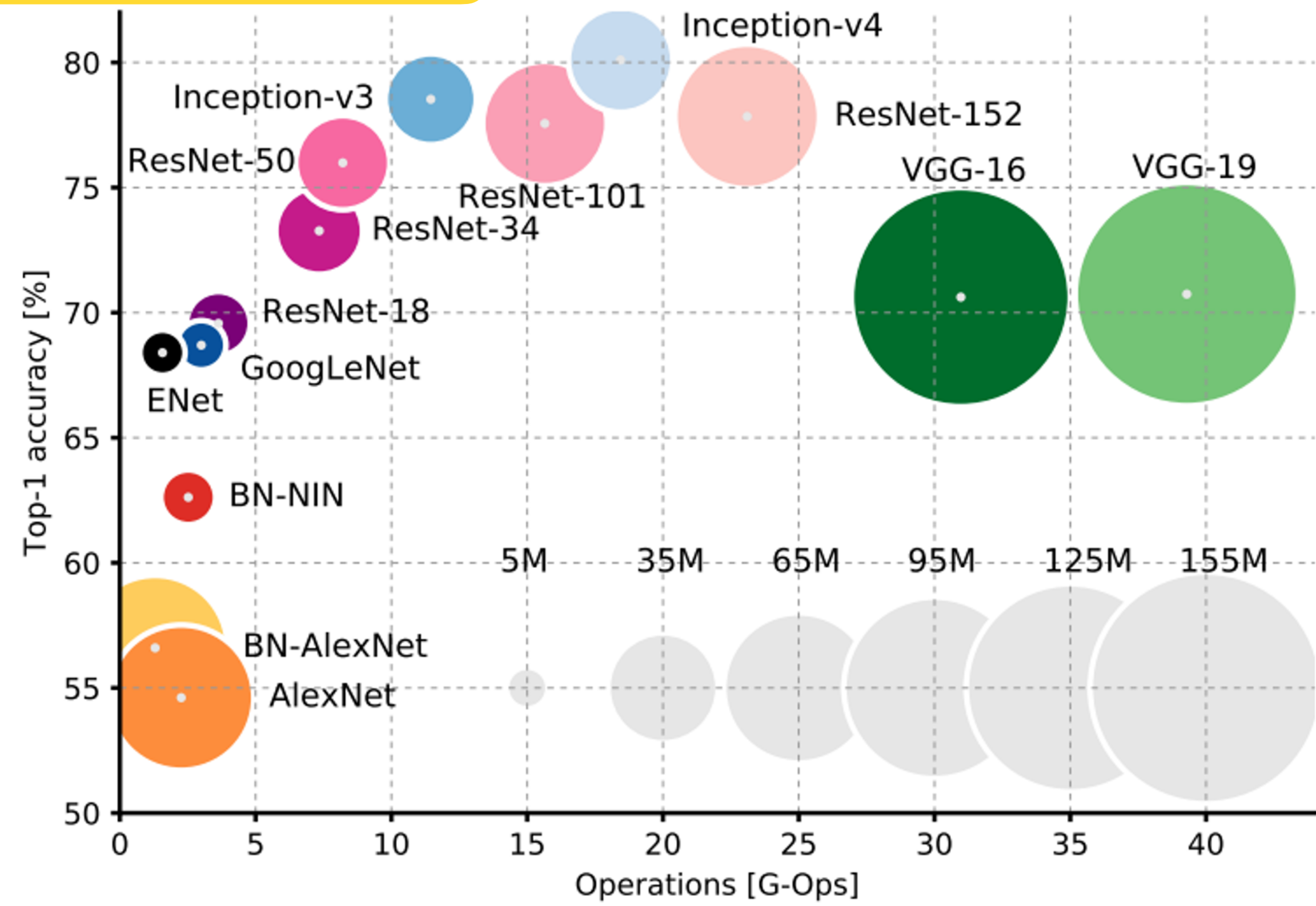
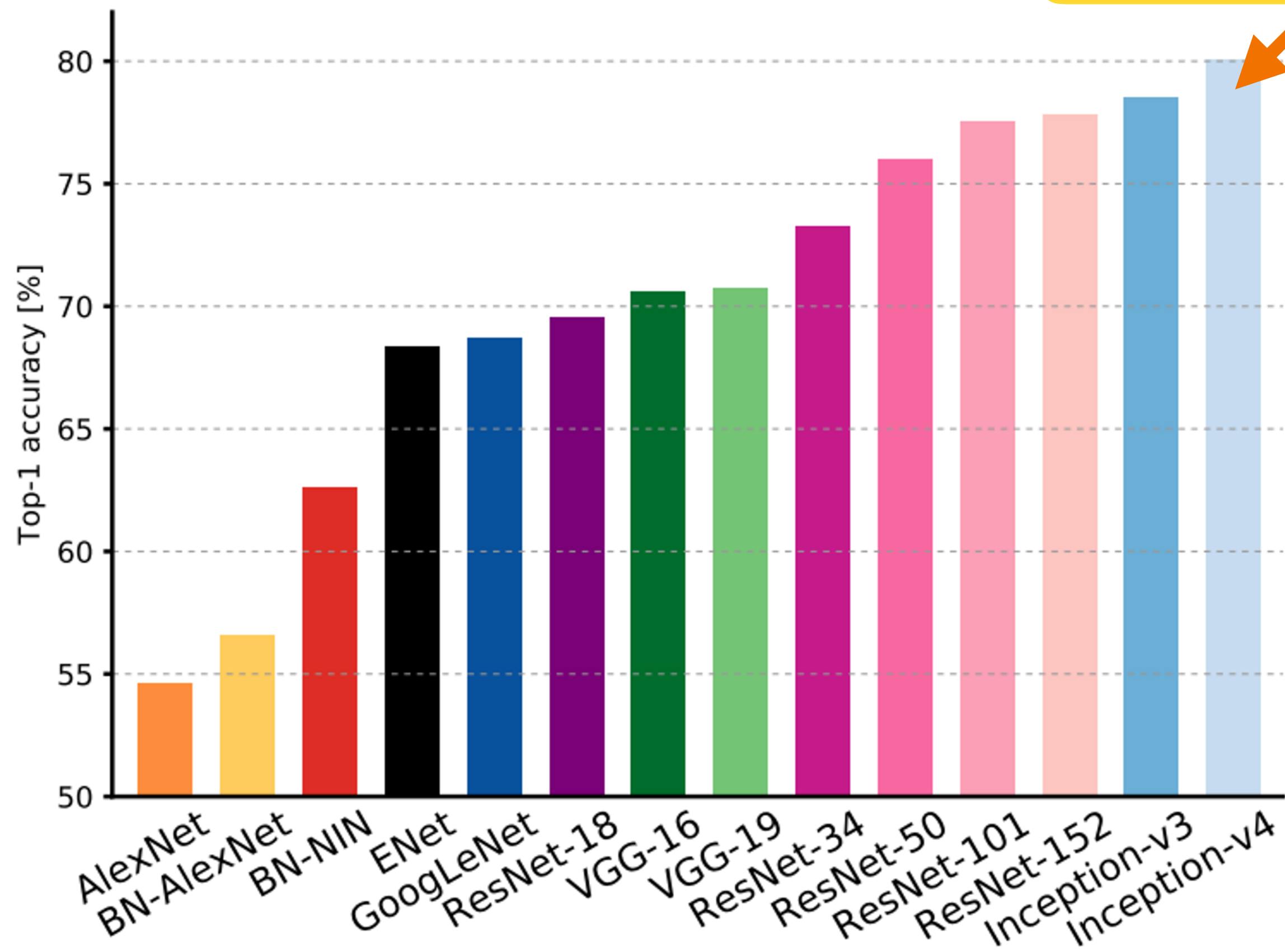
Comparing Complexity





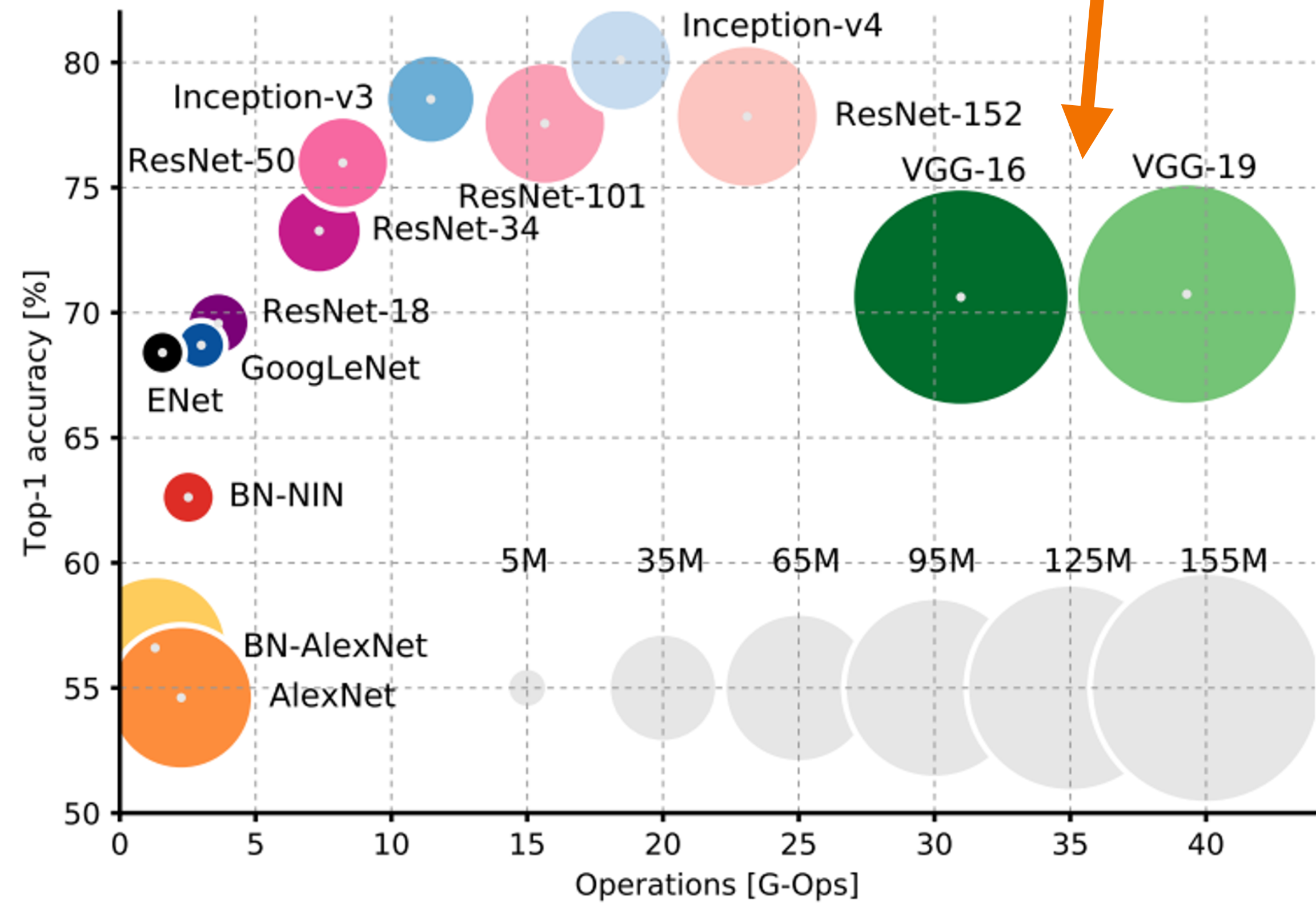
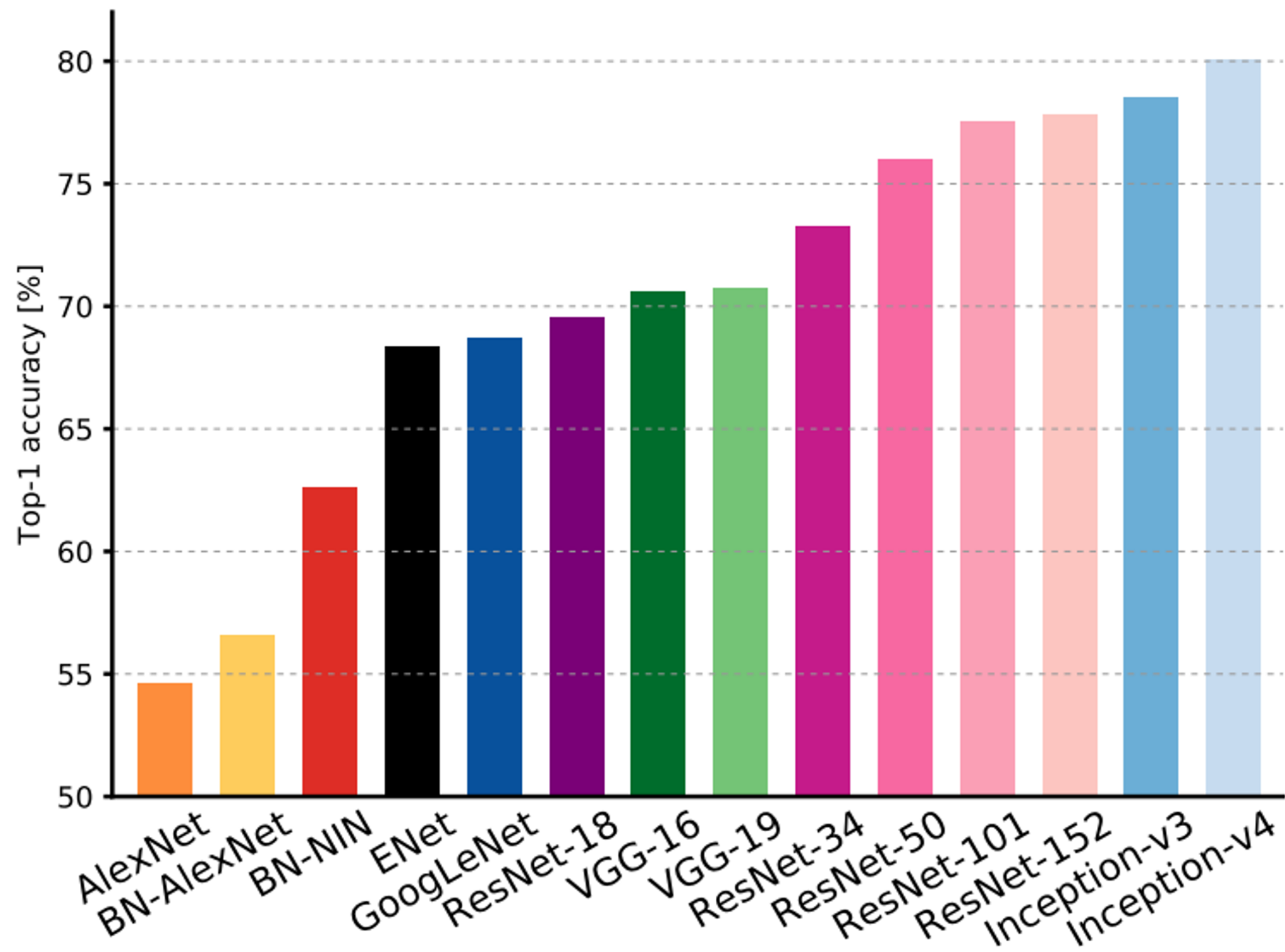
Comparing Complexity

Inception-v4: ResNet + Inception!



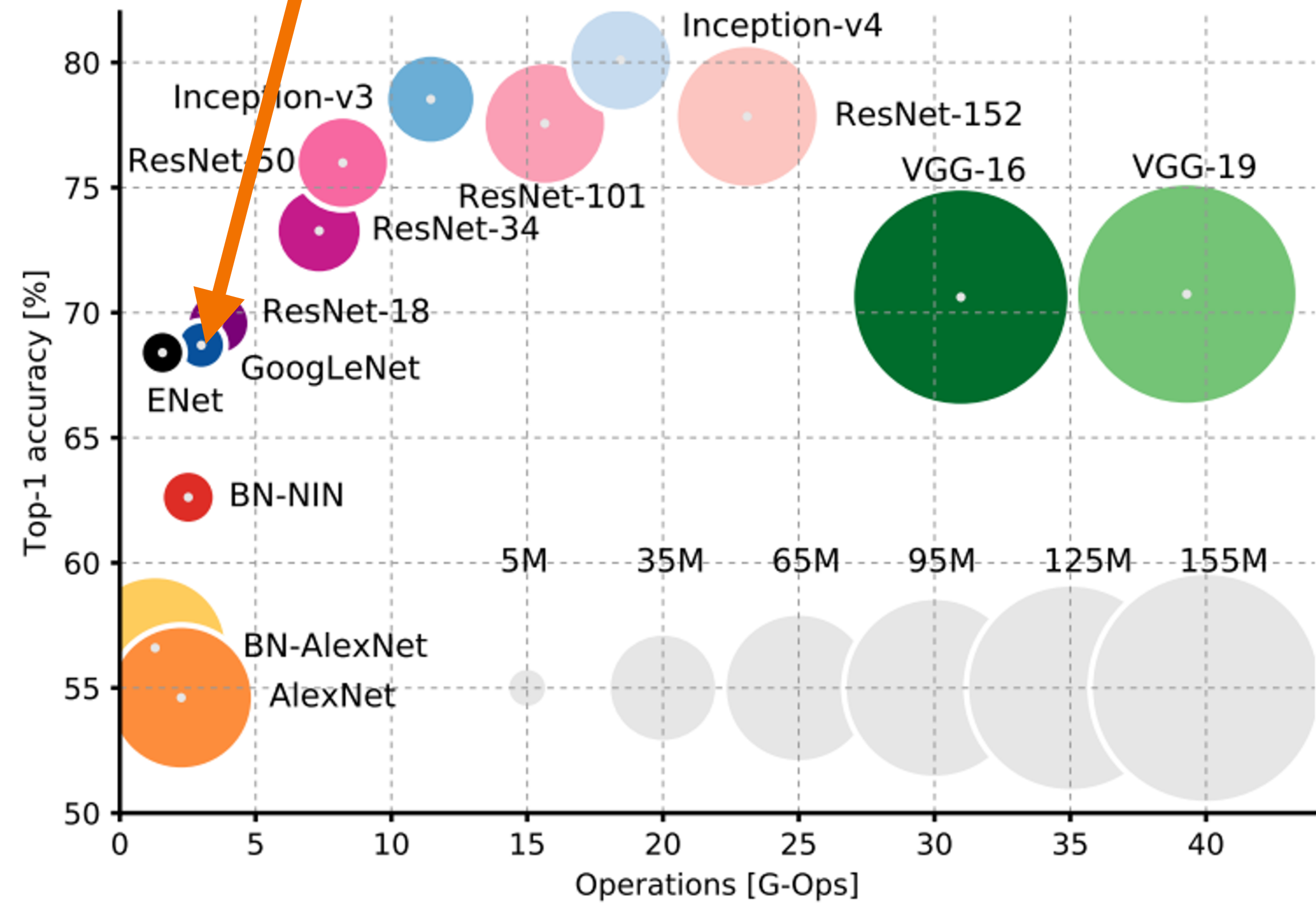
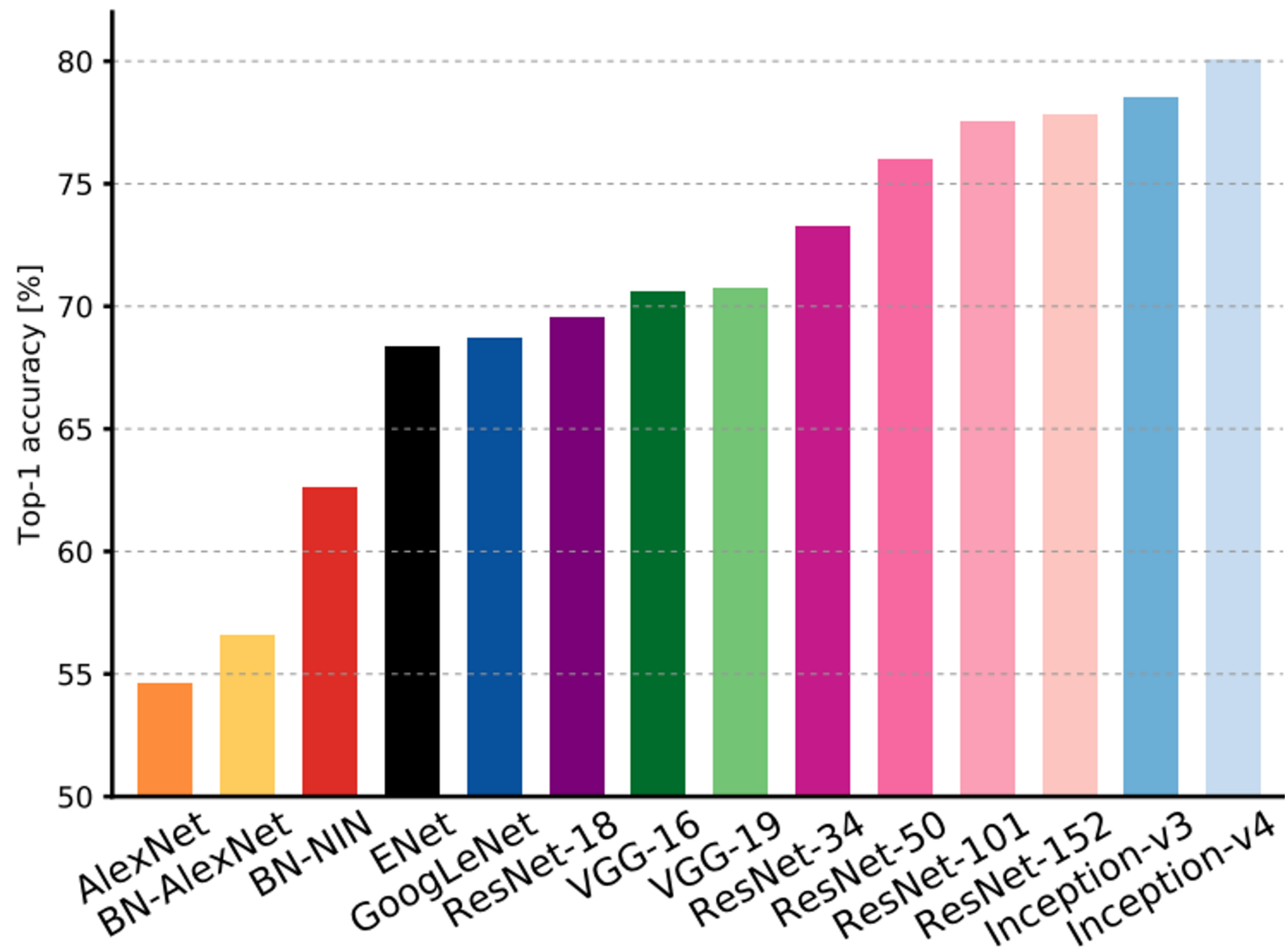
Comparing Complexity

VGG:
Highest memory,
most operations



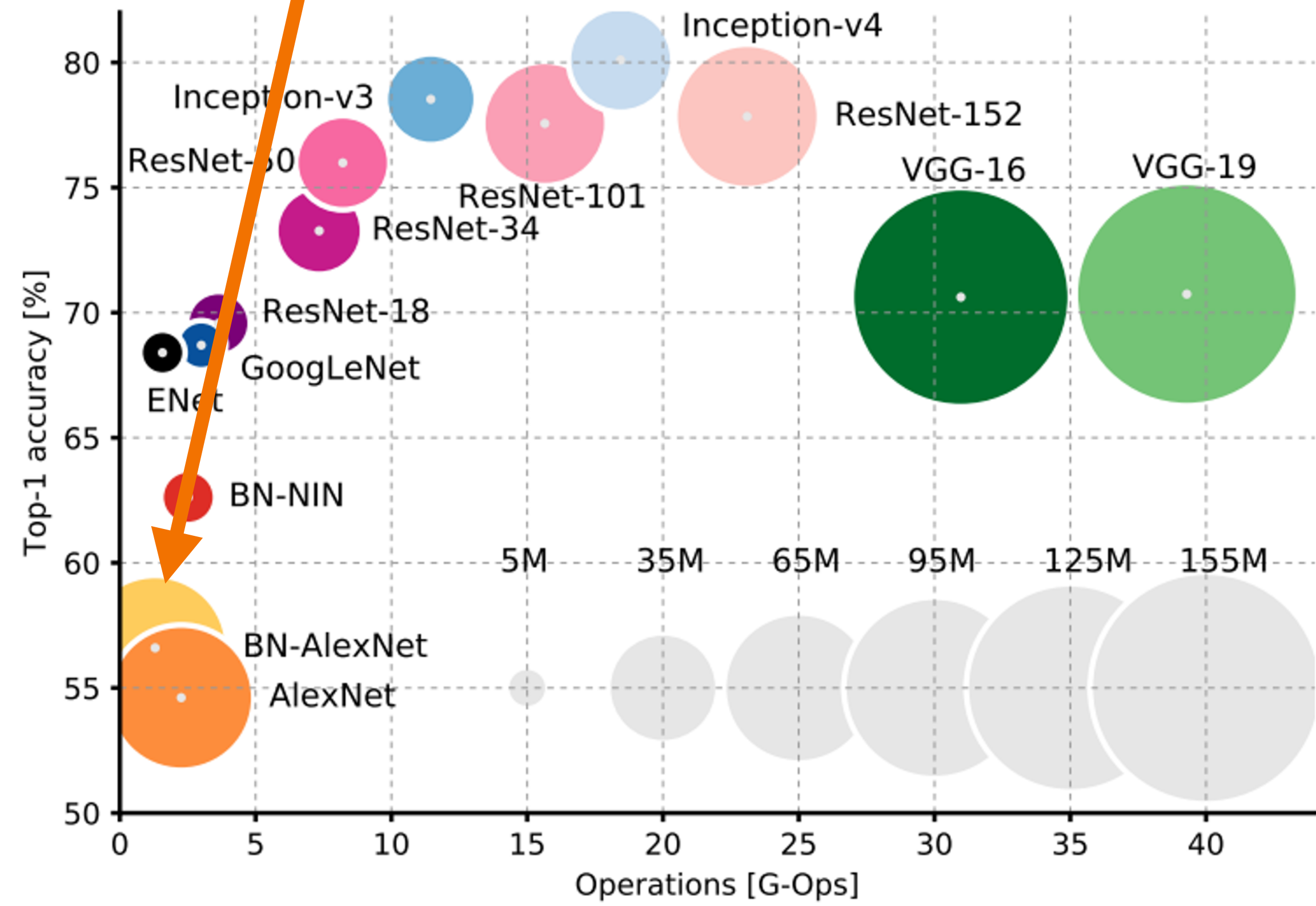
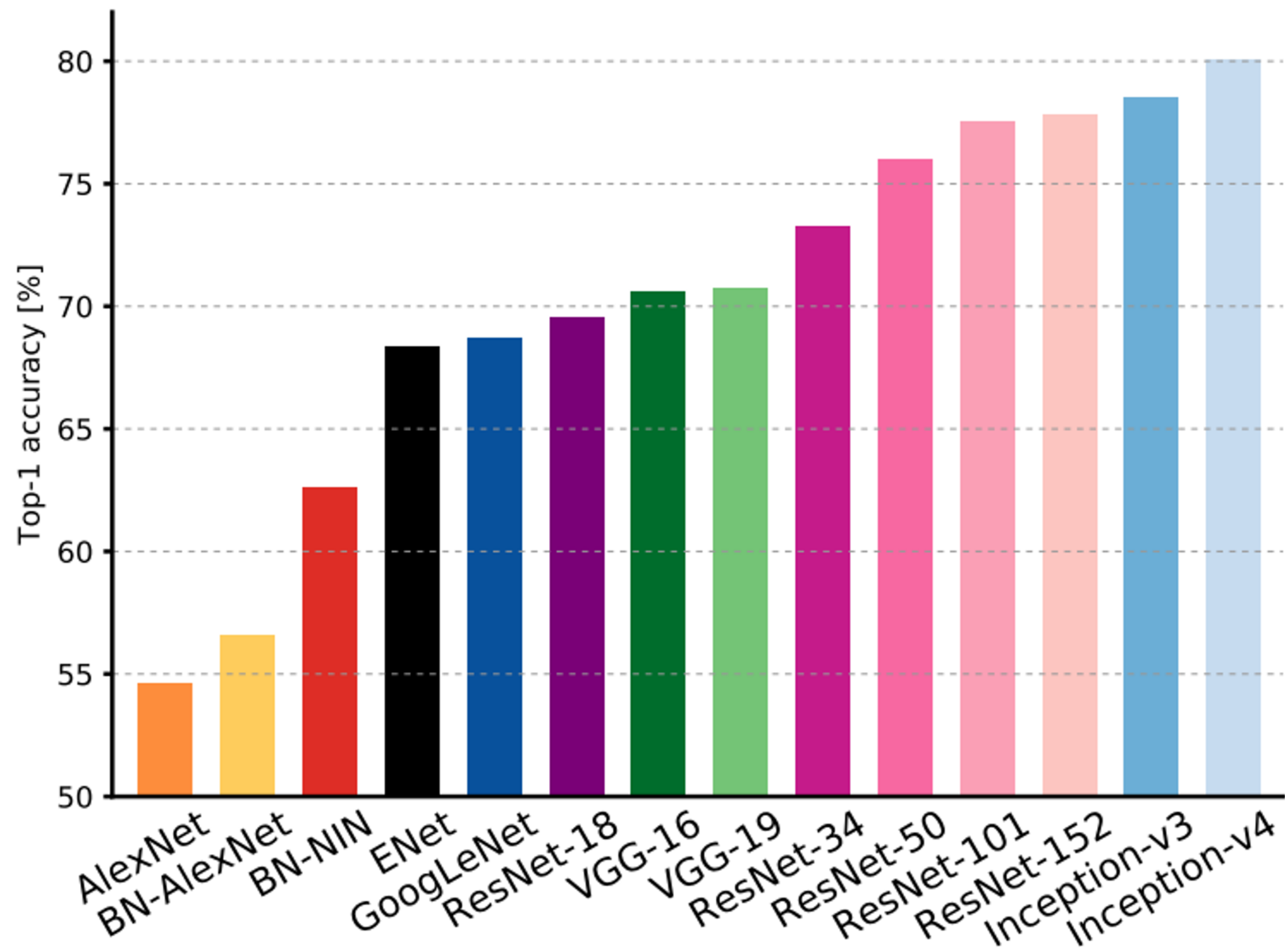
Comparing Complexity

GoogLeNet:
Very efficient!



Comparing Complexity

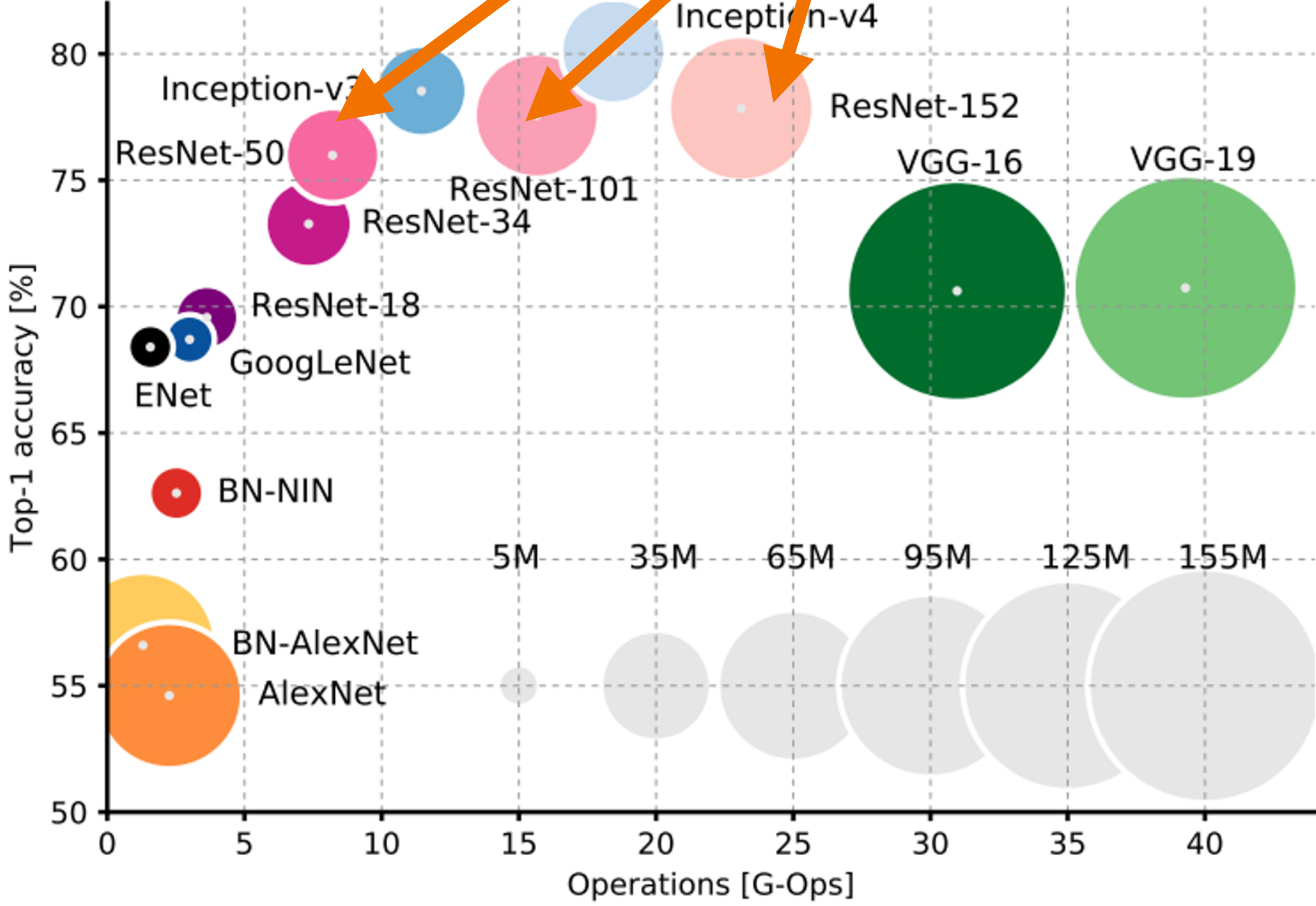
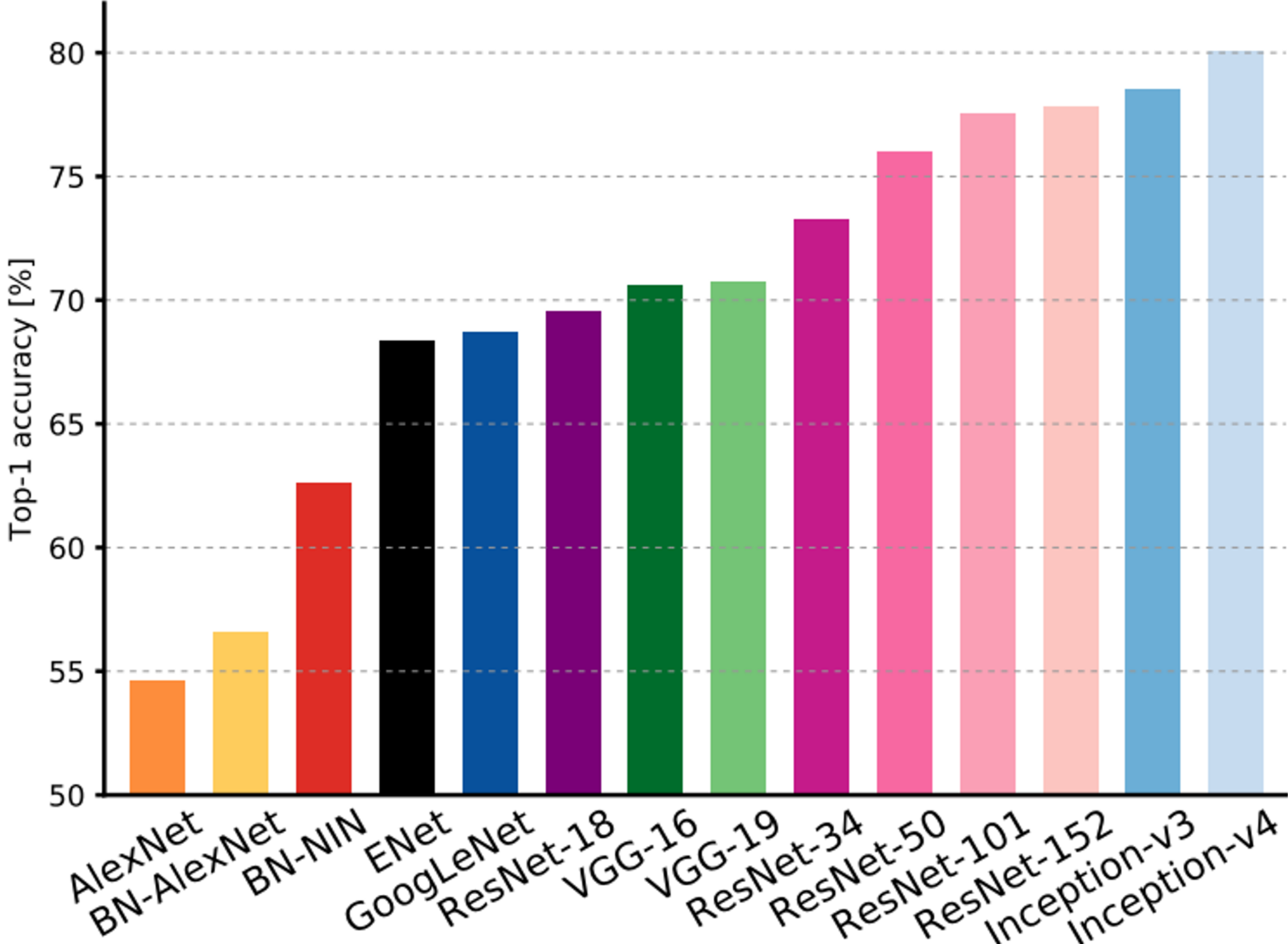
AlexNet: Low compute, lots of parameters



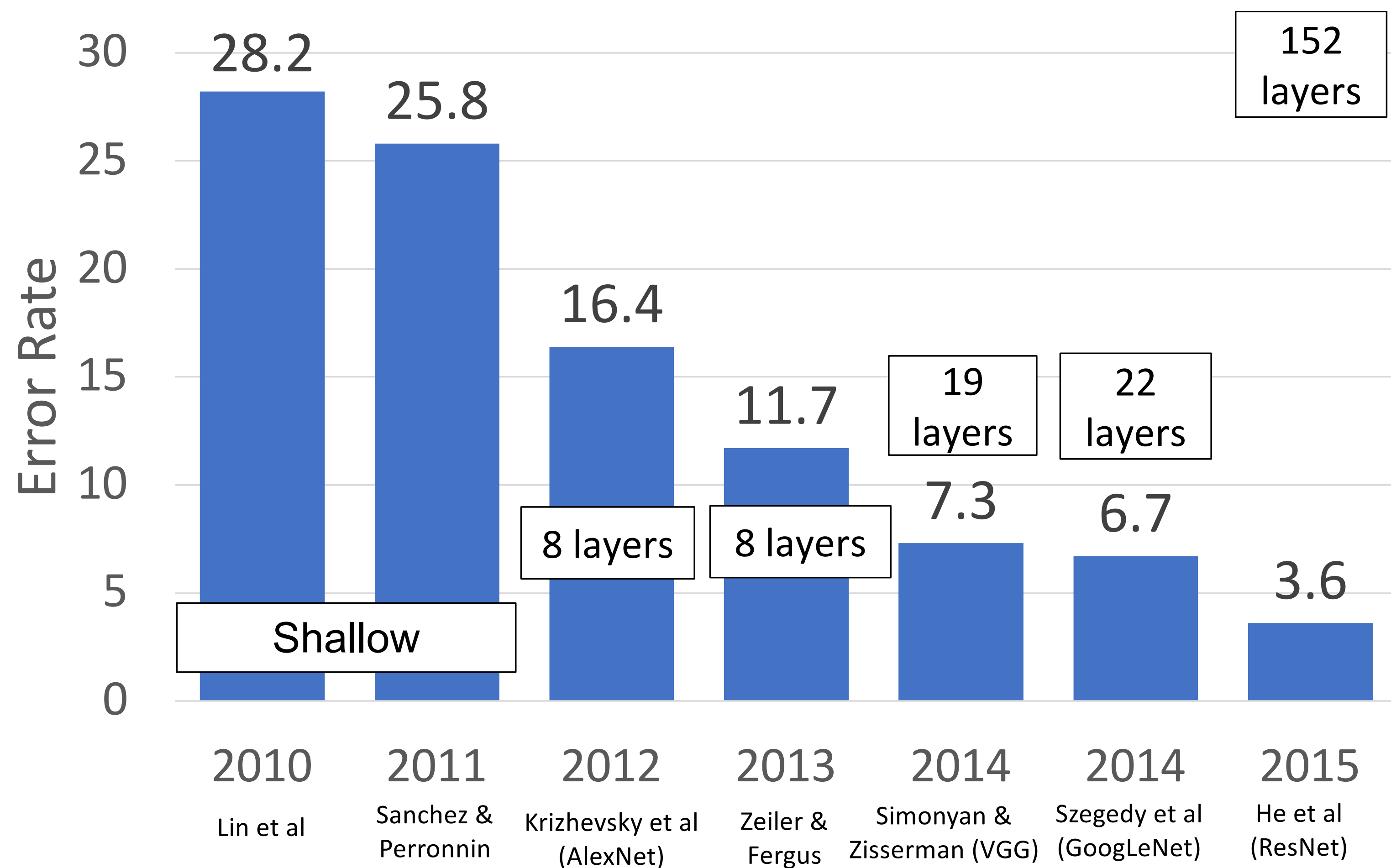


Comparing Complexity

ResNet: Simple design, moderate efficiency, high accuracy



ImageNet Classification Challenge



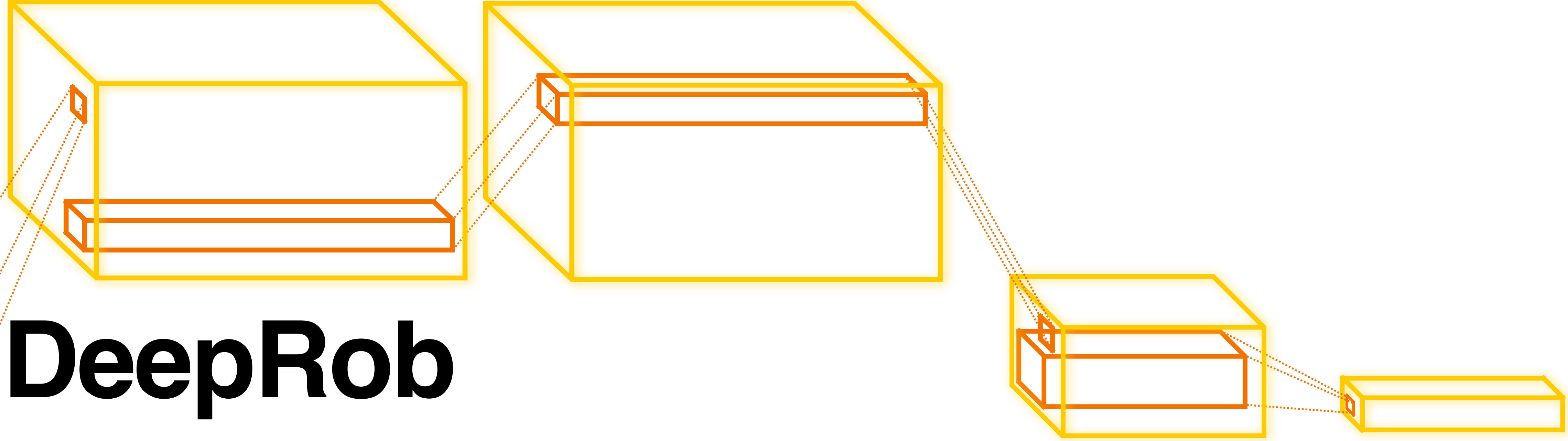
CNN architectures have continued to evolve!

We will see more in Lecture #



Next Time: Training Neural Networks





DeepRob

Lecture 8

CNN Architectures

University of Michigan and University of Minnesota

