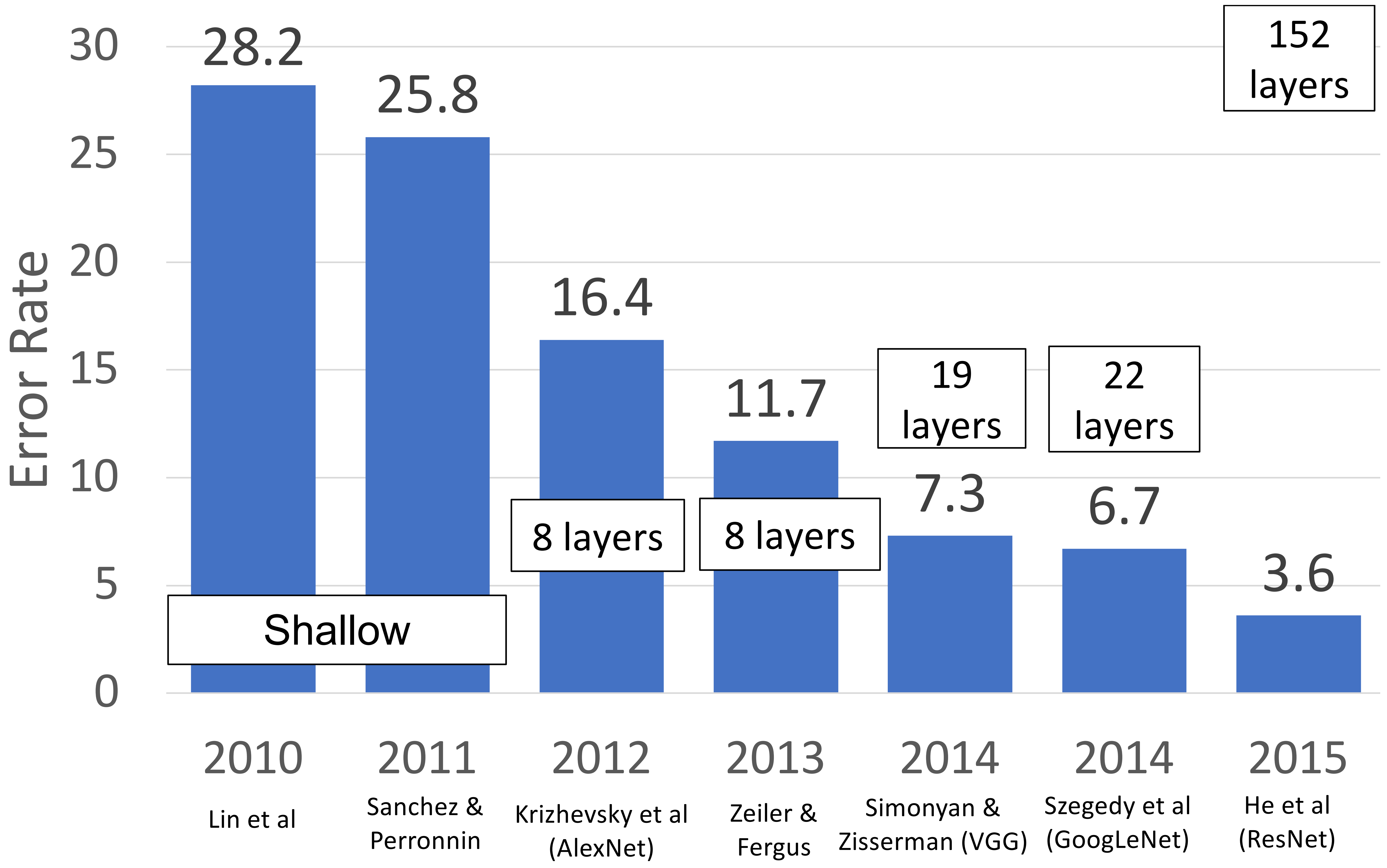
DeepRob

Lecture 9
Training Neural Networks I
University of Minnesota

Project 2—Updates

- Instructions available on the website
  - Here: https://rpm-lab.github.io/CSCI5980-F24-DeepRob/projects/project2/
- Implement two-layer neural network and generalize to FCN
- **Autograder fixed!**
- **Due Monday, October 14th, 11:59 PM CT**

Recap: CNN Architectures for ImageNet Classification

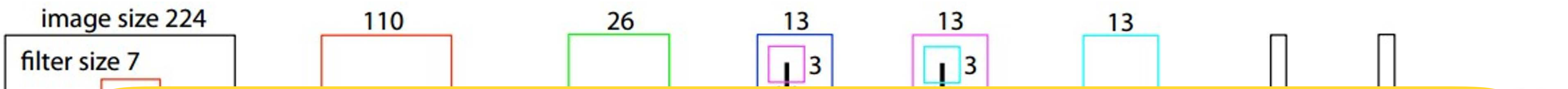


Questions from the previous lecture

- **Computation for Forward pass vs Backward pass**
  - Backward pass in a neural network takes significantly more compute than the forward pass (*computing gradients and propagating them back through the network*)
  - Forward pass compute time is used to compare networks as we care about the inference time (*after training*)

- **AlexNet memory requirement**
  - Input size of 227 x 227 pixels
  - Batch size of 128 images
  - ~2.3 gigabytes for storing the activations of all the layers during the forward pass
  - During training, additional memory is required to store intermediate results for backpropagation, weight updates, and other operations.

Questions from the previous lecture

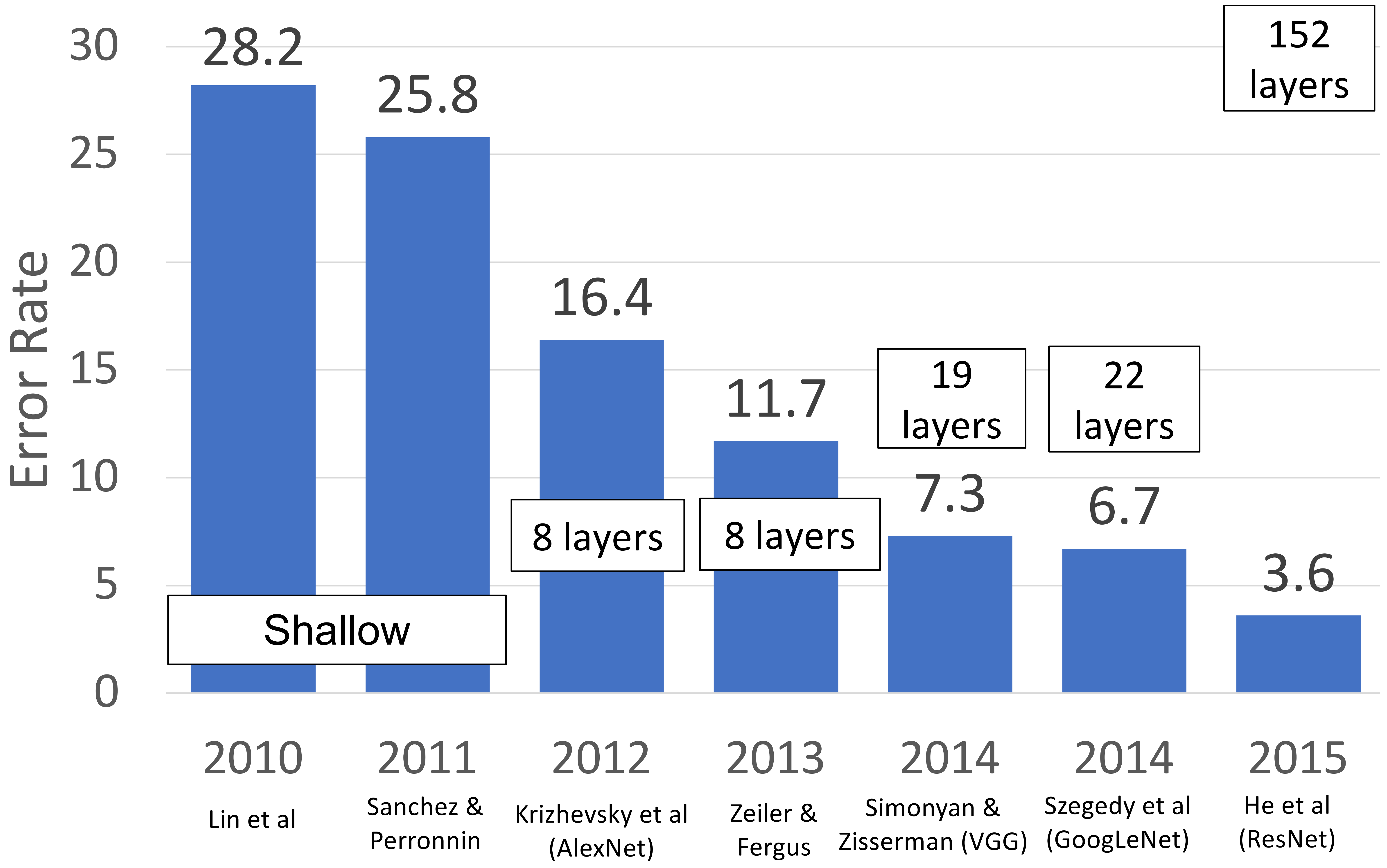


- **Difference between ZFNet and AlexNet**
  - Bigger networks (more params, more memory)
  - 11x11 stride 4 to 7x7 stride 2 -> Less aggressively downsampling the spatial dimensions, higher spatial resolution -> more receptive fields -> more compute.
  - Increase in the number of filters in the later layers -> more learnable parameters -> more memory and more compute.

More trial and error :(

Zeiler and Fergus, “Visualizing and Understanding Convolutional Networks”, ECCV 2014

Recap: CNN Architectures for ImageNet Classification



Residual Networks

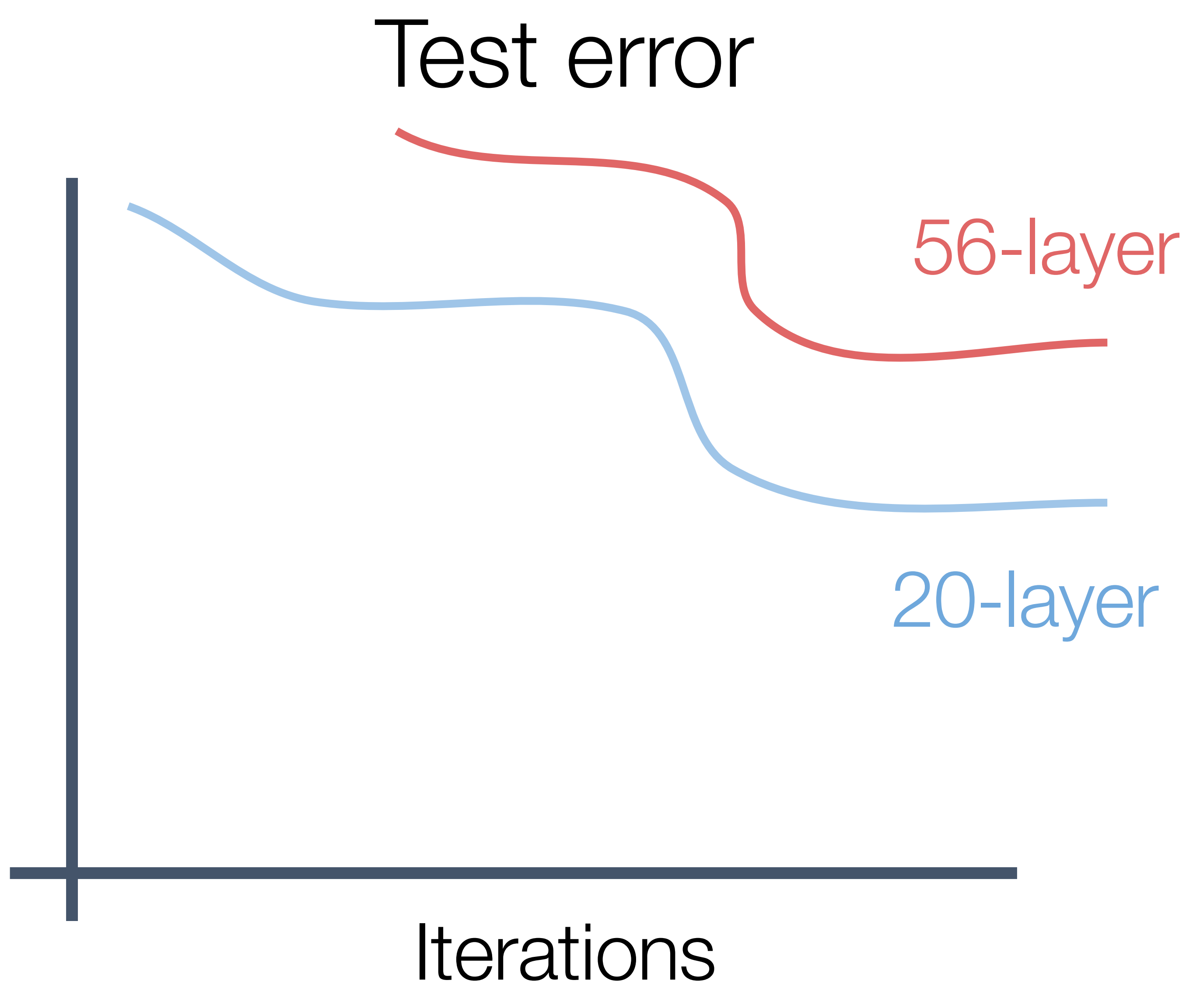
Once we have Batch Normalization, we can train networks with 10+ layers.
What happens as we go deeper?

He et al, "Deep Residual Learning for Image Recognition", CVPR 2016

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!



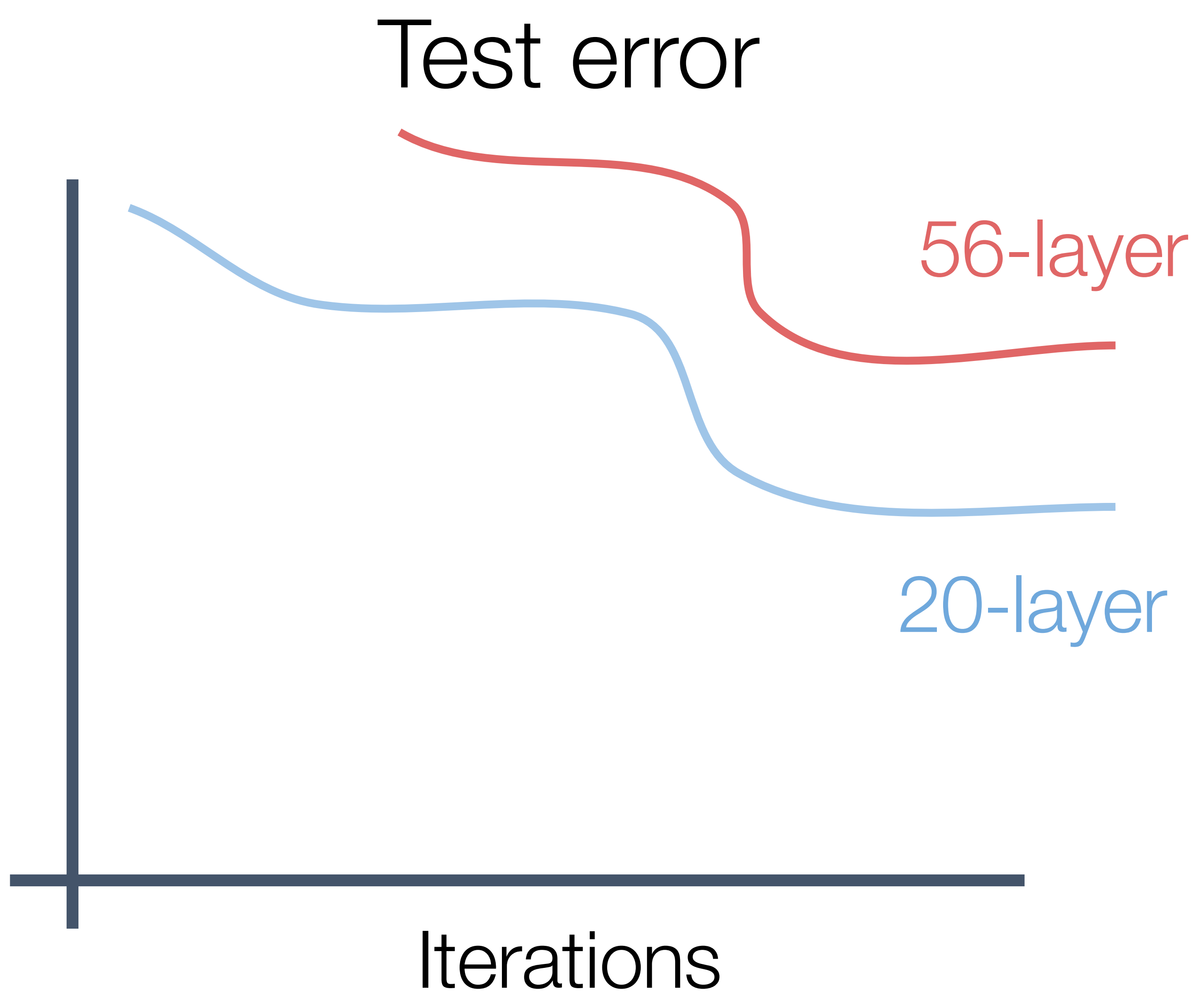
He et al, “Deep Residual Learning for Image Recognition”, CVPR 2016

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

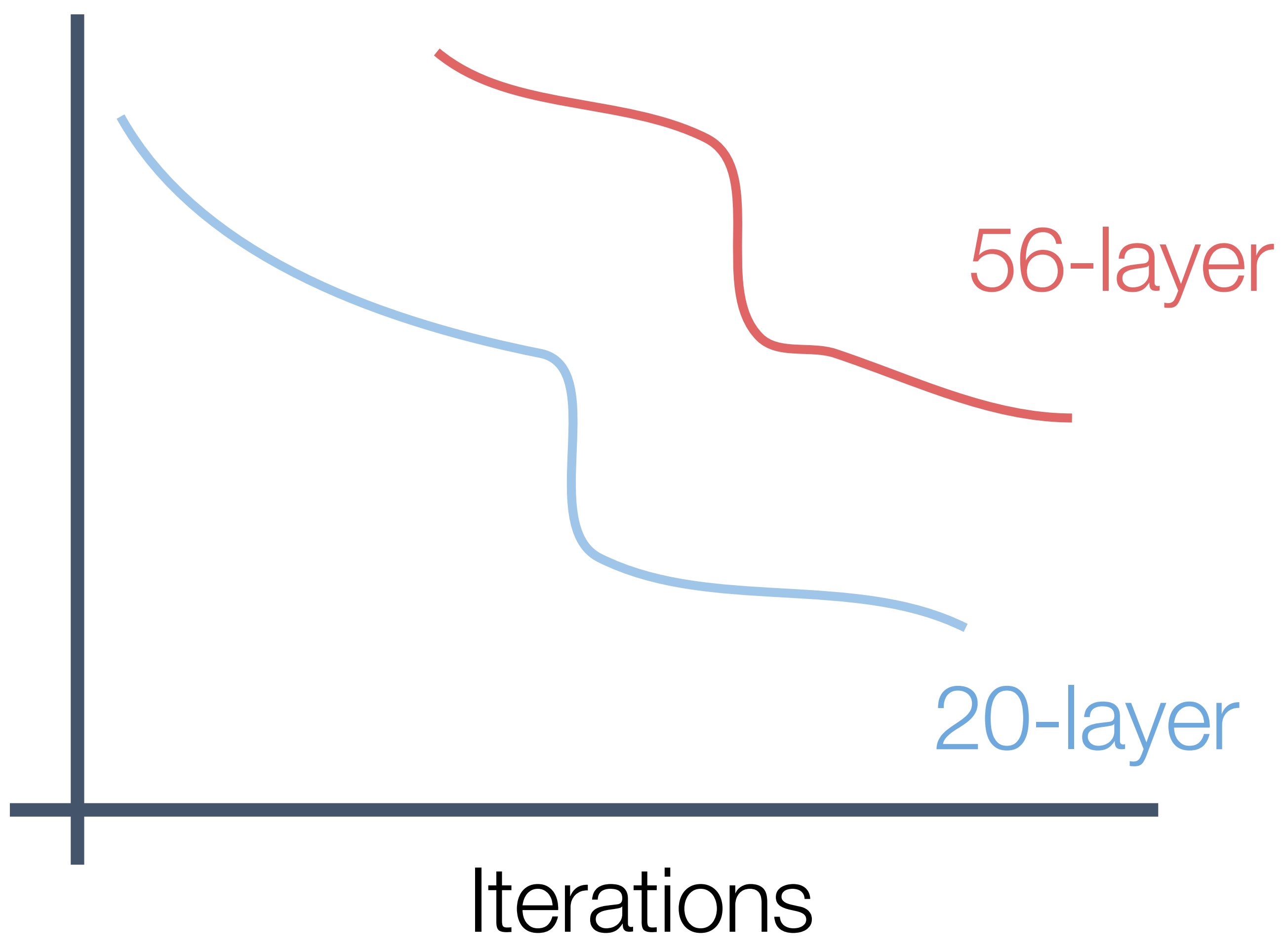


He et al, “Deep Residual Learning for Image Recognition”, CVPR 2016

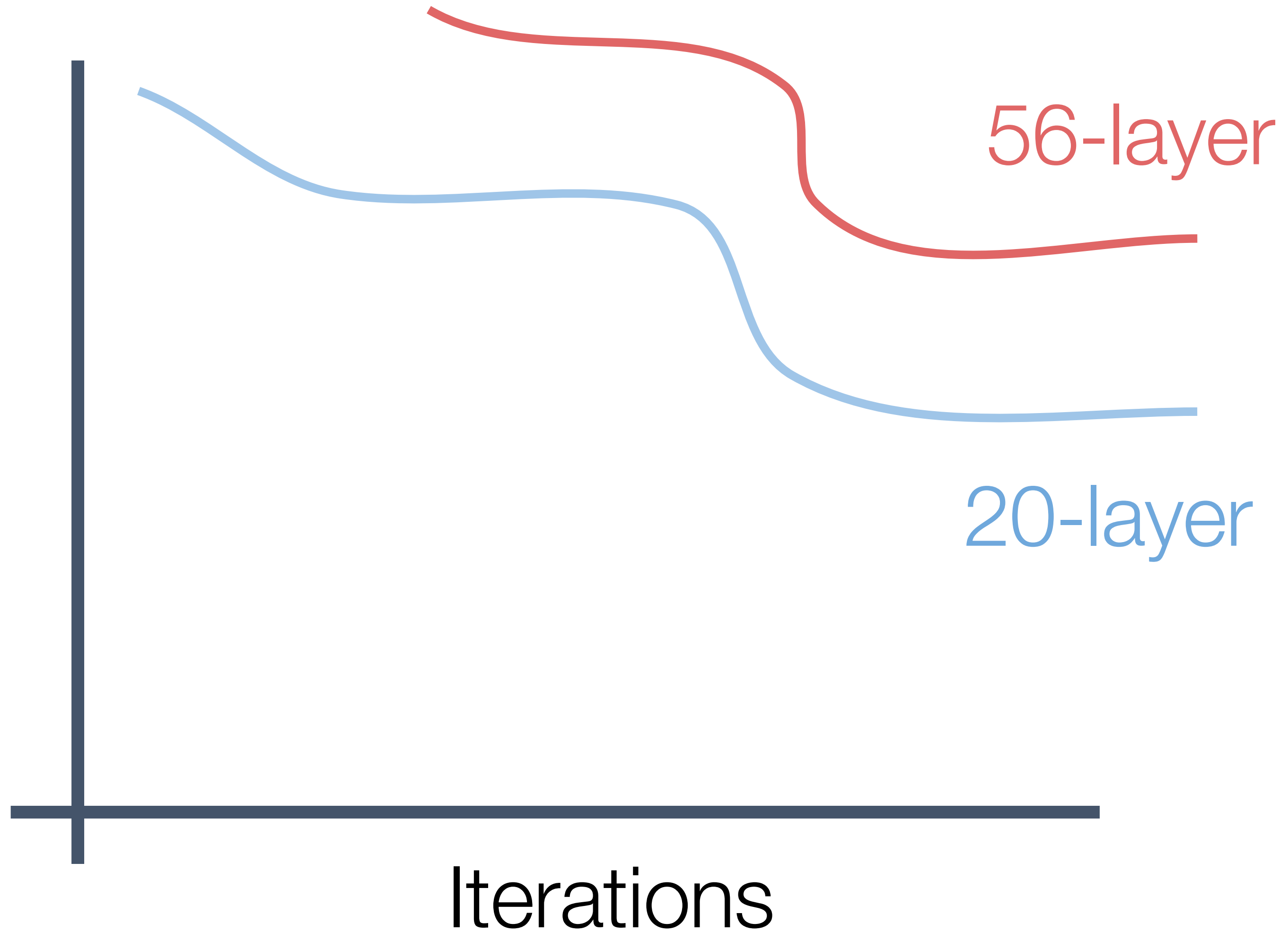
Residual Networks

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

Training error



Test error



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**

He et al, "Deep Residual Learning for Image Recognition", CVPR 2016

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*

He et al, “Deep Residual Learning for Image Recognition”, CVPR 2016

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

He et al, "Deep Residual Learning for Image Recognition", CVPR 2016

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*

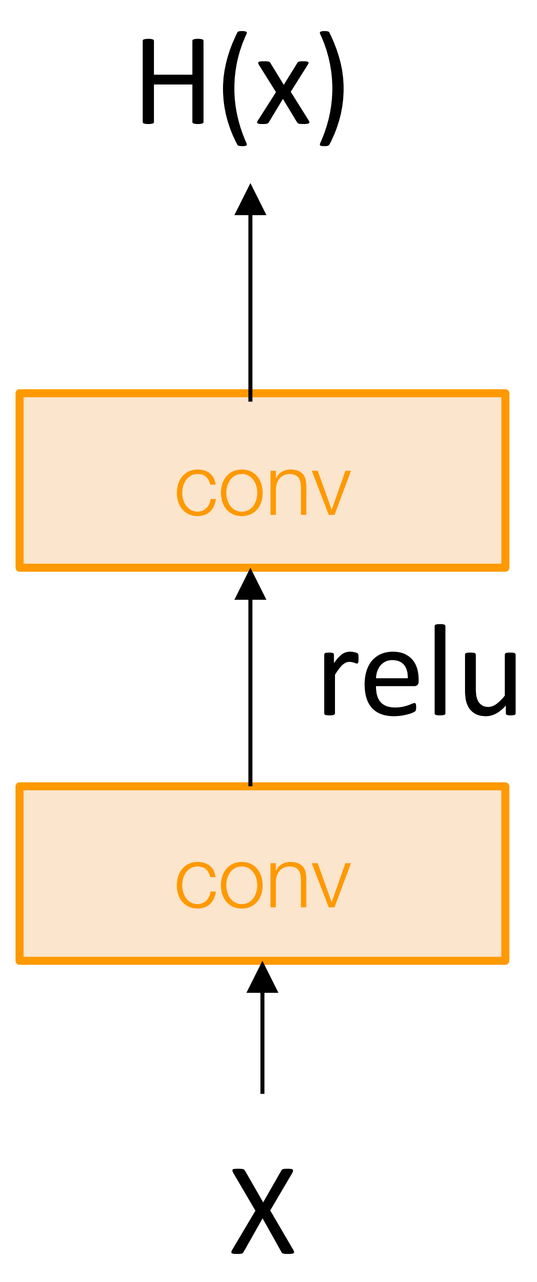
Hypothesis: This is an optimization problem. Deeper models are harder to optimize, and in particular don’t learn identity functions to emulate shallow models

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

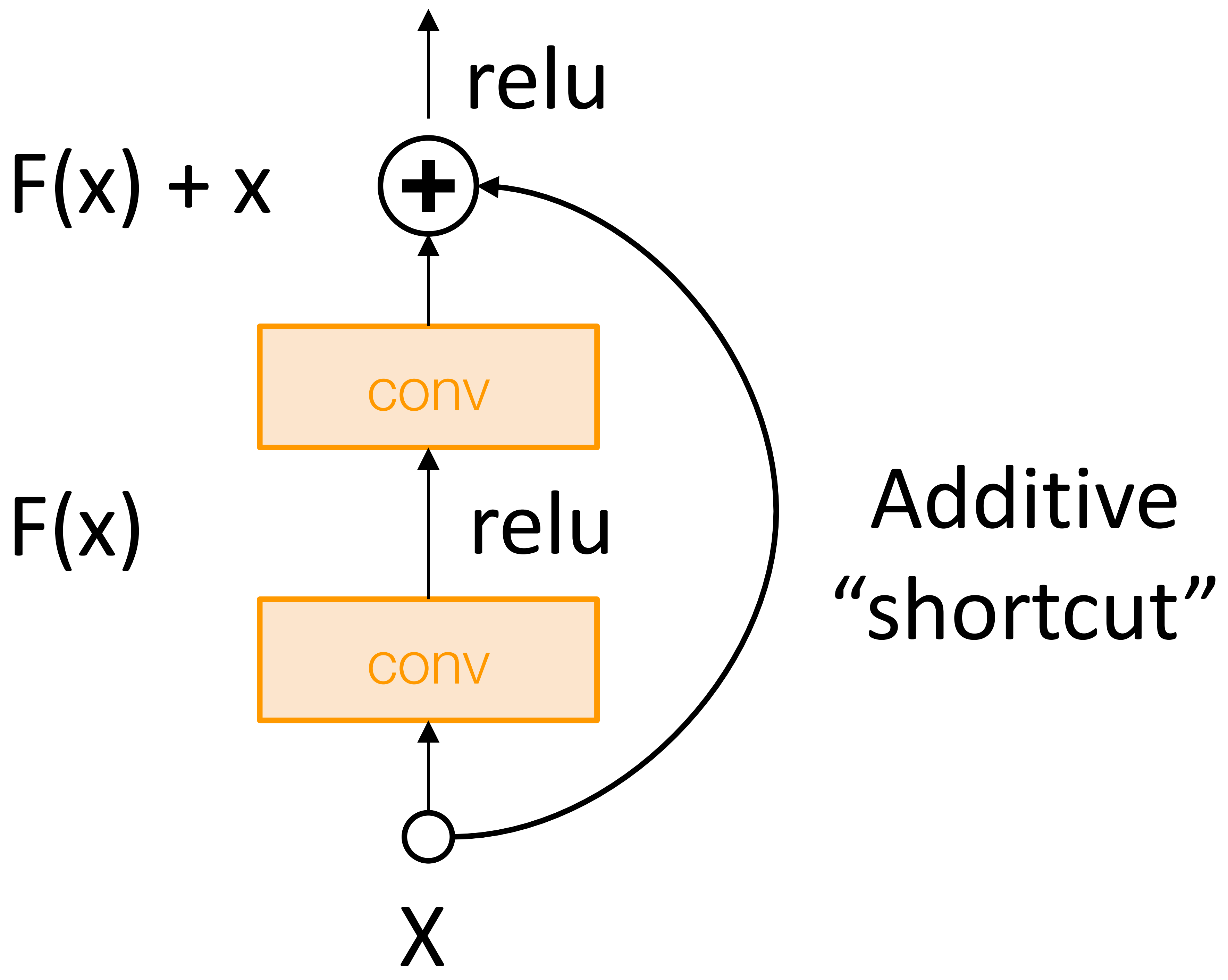
He et al, “Deep Residual Learning for Image Recognition”, CVPR 2016

Residual Networks

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



"Plain" block

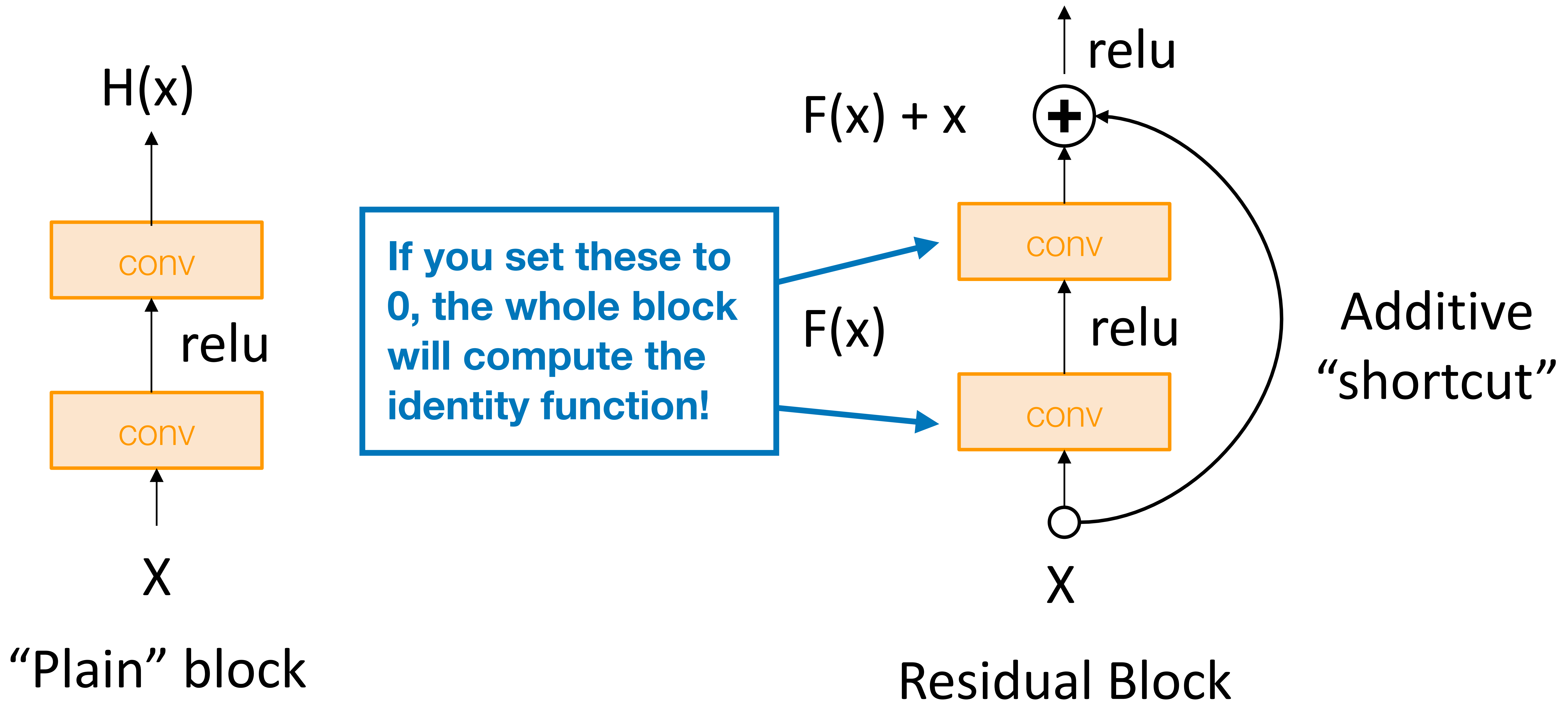


Residual Block

He et al, "Deep Residual Learning for Image Recognition", CVPR 2016

Residual Networks

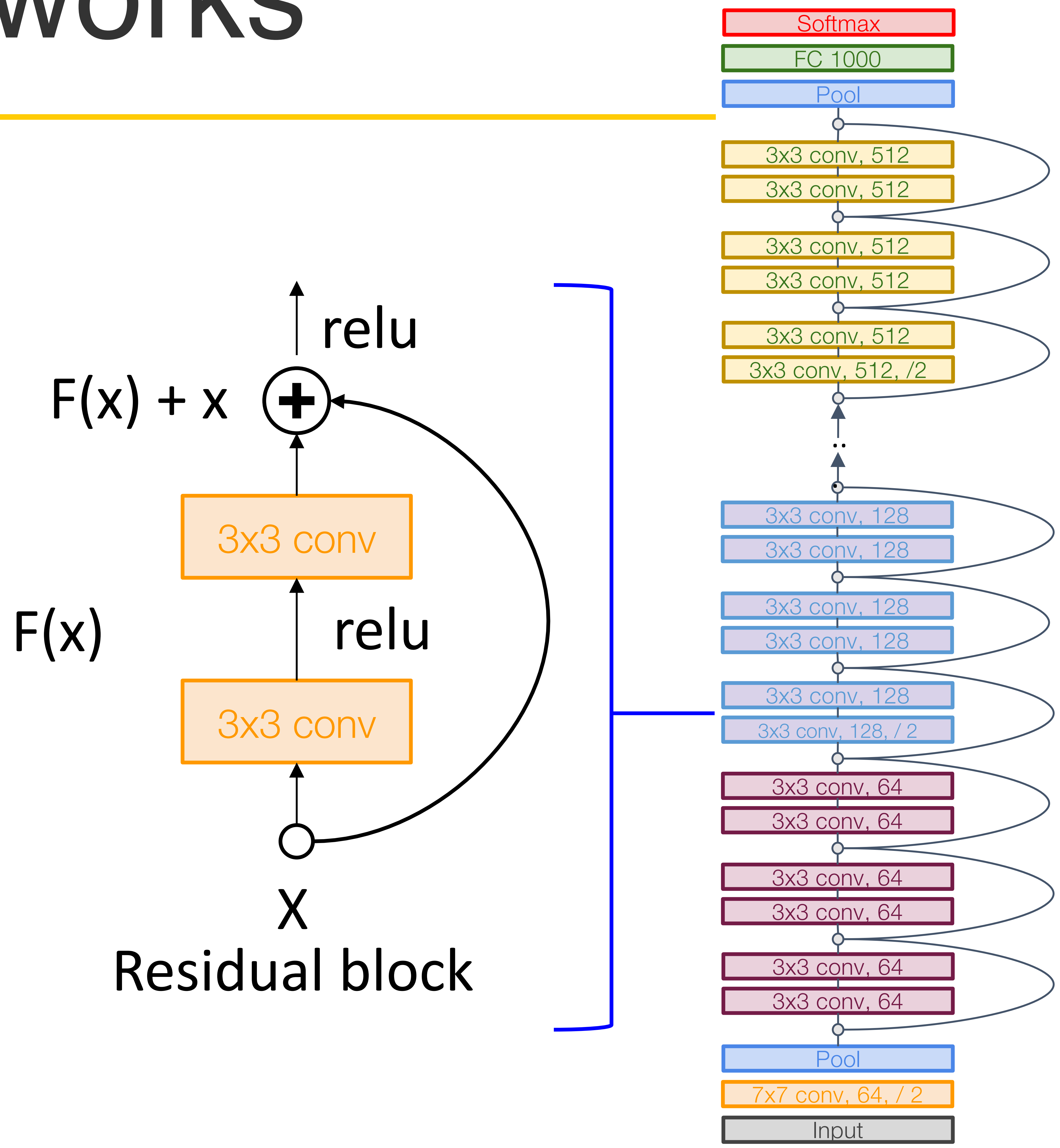
Solution: Change the network so learning identity functions with extra layers is easy!



He et al, "Deep Residual Learning for Image Recognition", CVPR 2016

Residual Networks

A residual network is a stack of many residual blocks

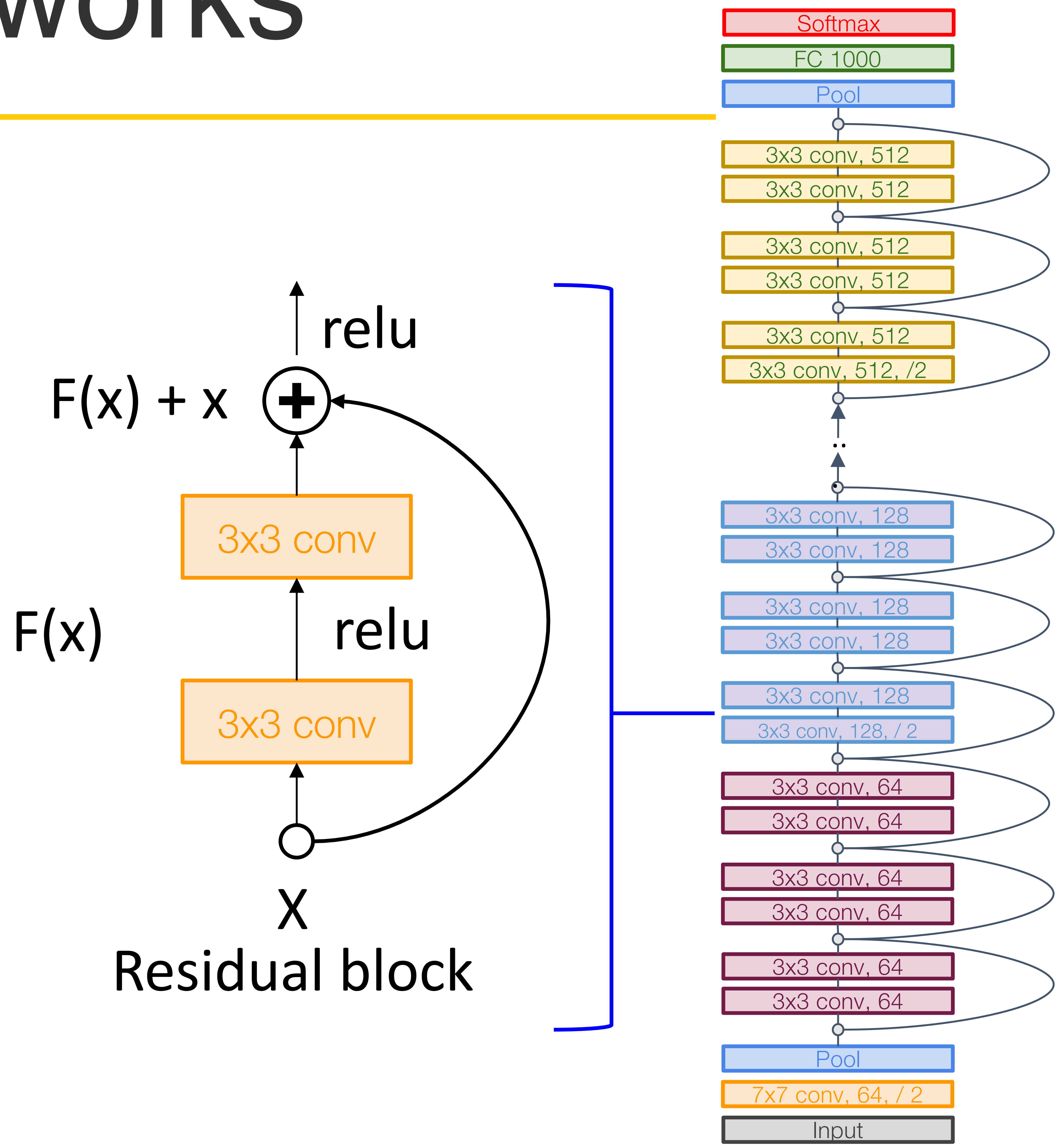


He et al, "Deep Residual Learning for Image Recognition", CVPR 2016

Residual Networks

A residual network is a stack of many residual blocks

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



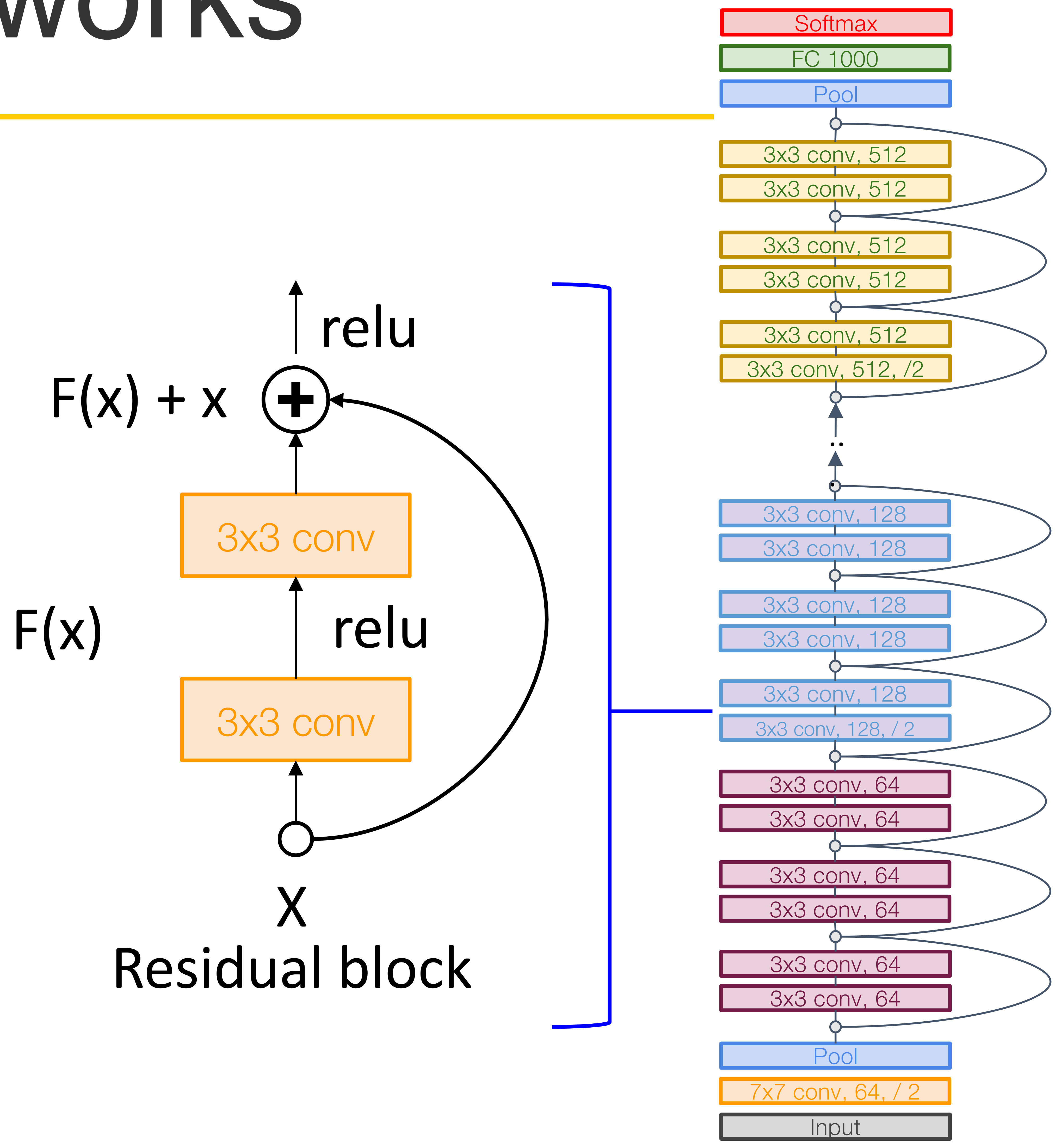
He et al, "Deep Residual Learning for Image Recognition", CVPR 2016

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

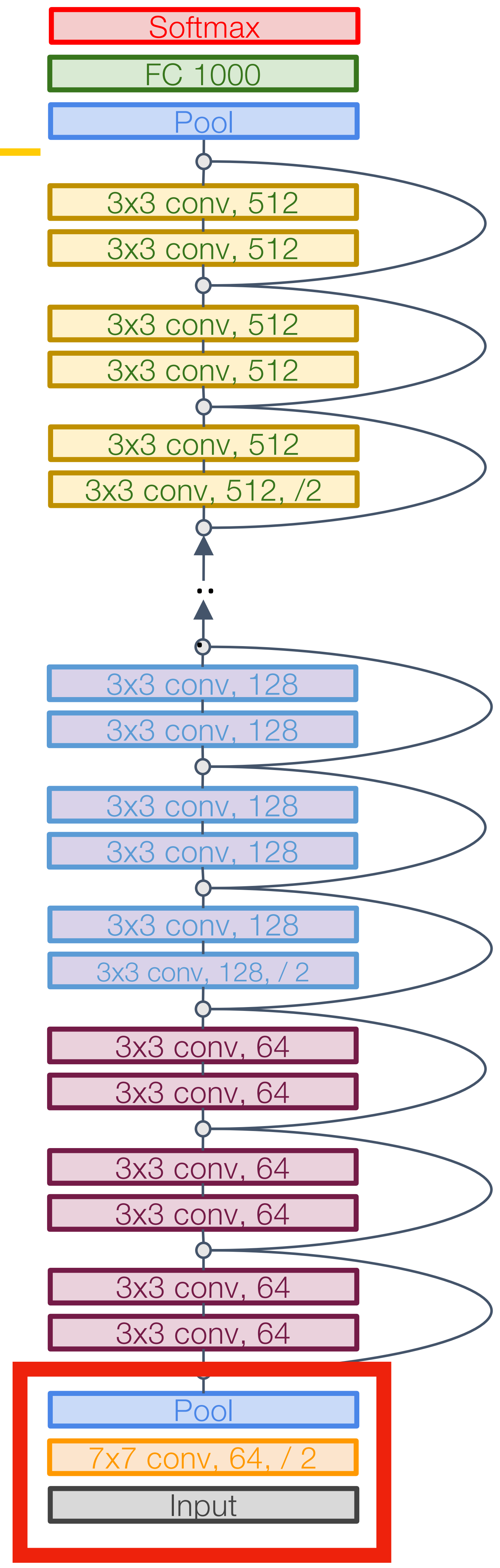


He et al, “Deep Residual Learning for Image Recognition”, CVPR 2016

Residual Networks

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

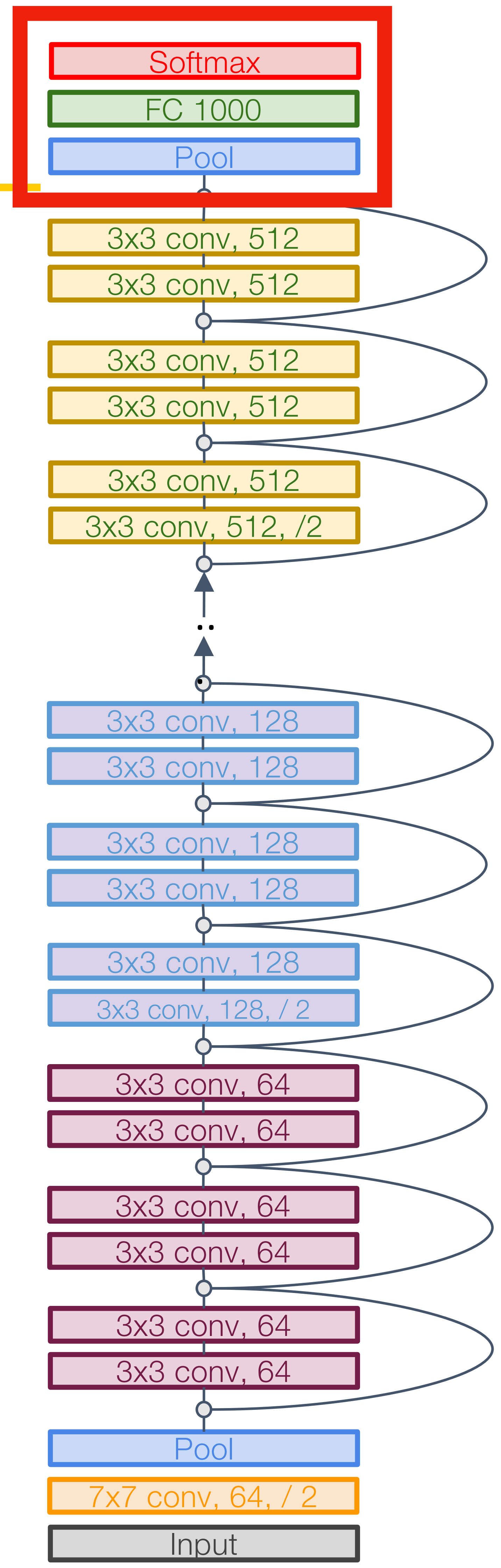
	Input size		Layer				Output size				
Layer	C	H/W	Filters	Kernel	Stride	Pad	C	H/W	Memory (KB)	Params	Flop (M)
Conv	3	224	64	7	2	3	64	112	3136	9	118
Max-pool	64	112		3	2	1	64	56	784	0	2



He et al, “Deep Residual Learning for Image Recognition”, CVPR 2016

Residual Networks

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

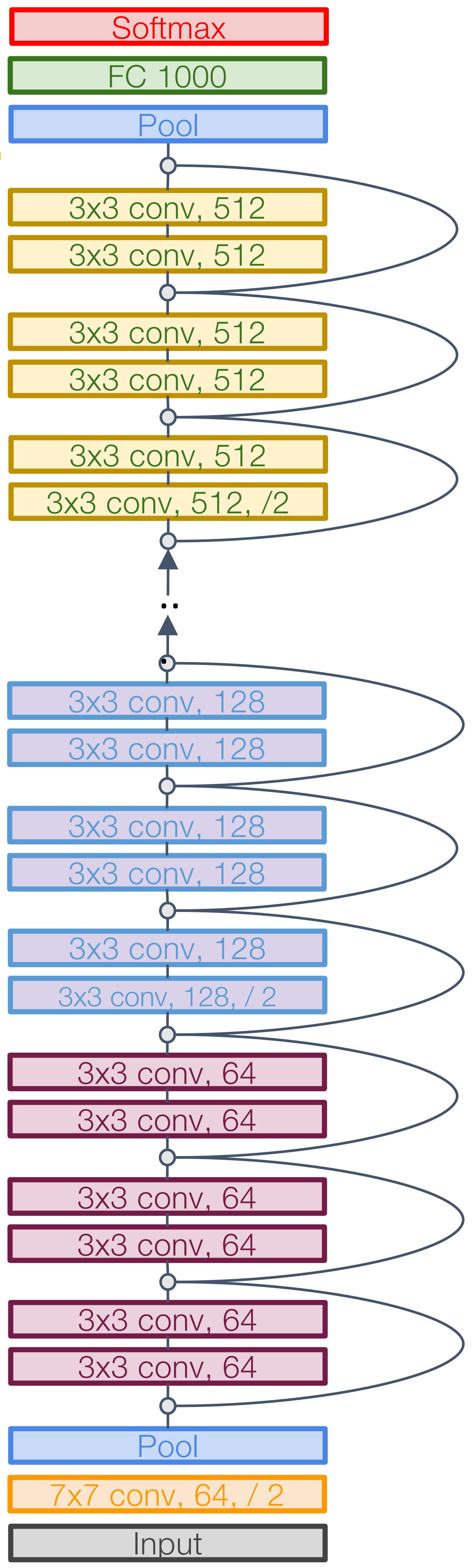


He et al, "Deep Residual Learning for Image Recognition", CVPR 2016

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



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

Residual Networks

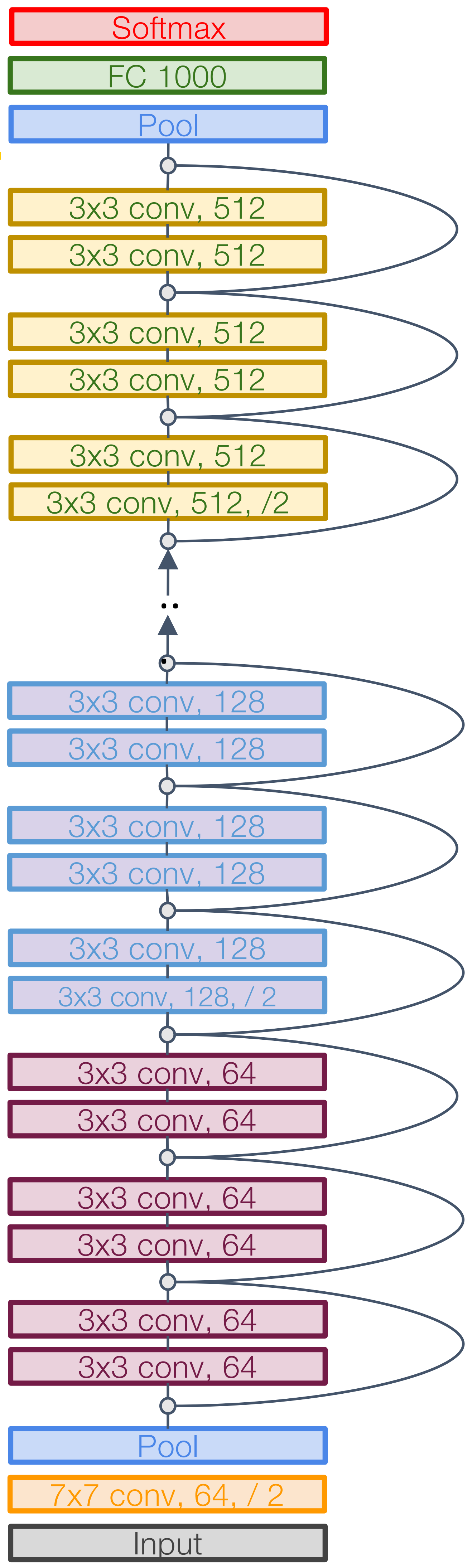
ResNet-18:
Stem: 1 conv layer
Stage 1 (C=64): 2 res. block = 4 conv
Stage 2 (C=128): 2 res. block = 4 conv
Stage 3 (C=256): 2 res. block = 4 conv
Stage 4 (C=512): 2 res. block = 4 conv
Linear

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

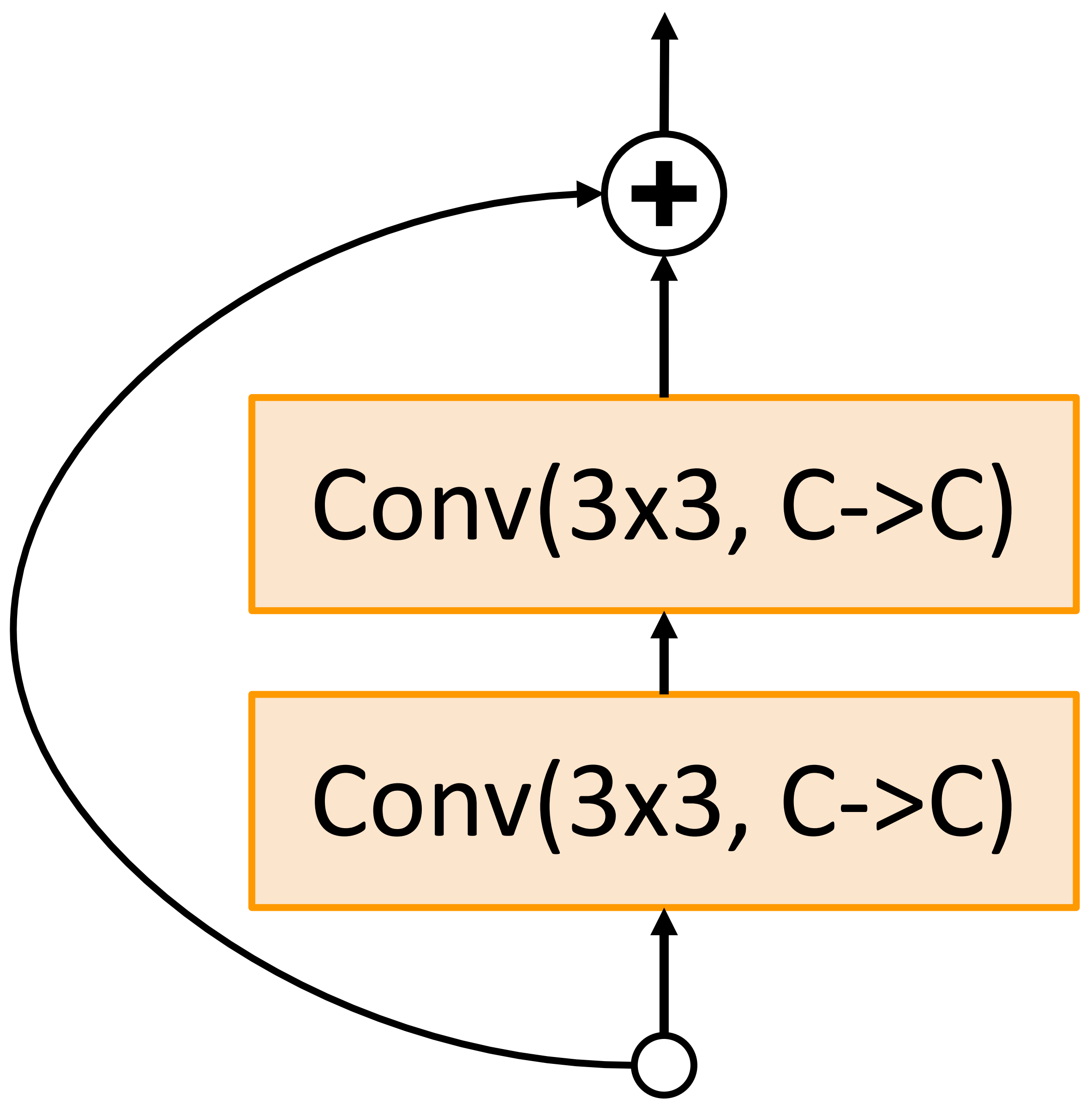
ImageNet top-5 error: 8.58
GFLOP: 3.6

VGG-16:
ImageNet top-5 error: 9.62
GFLOP: 13.6



He et al, “Deep Residual Learning for Image Recognition”, CVPR 2016
Error rates are 224x224 single-crop testing, reported by torchvision

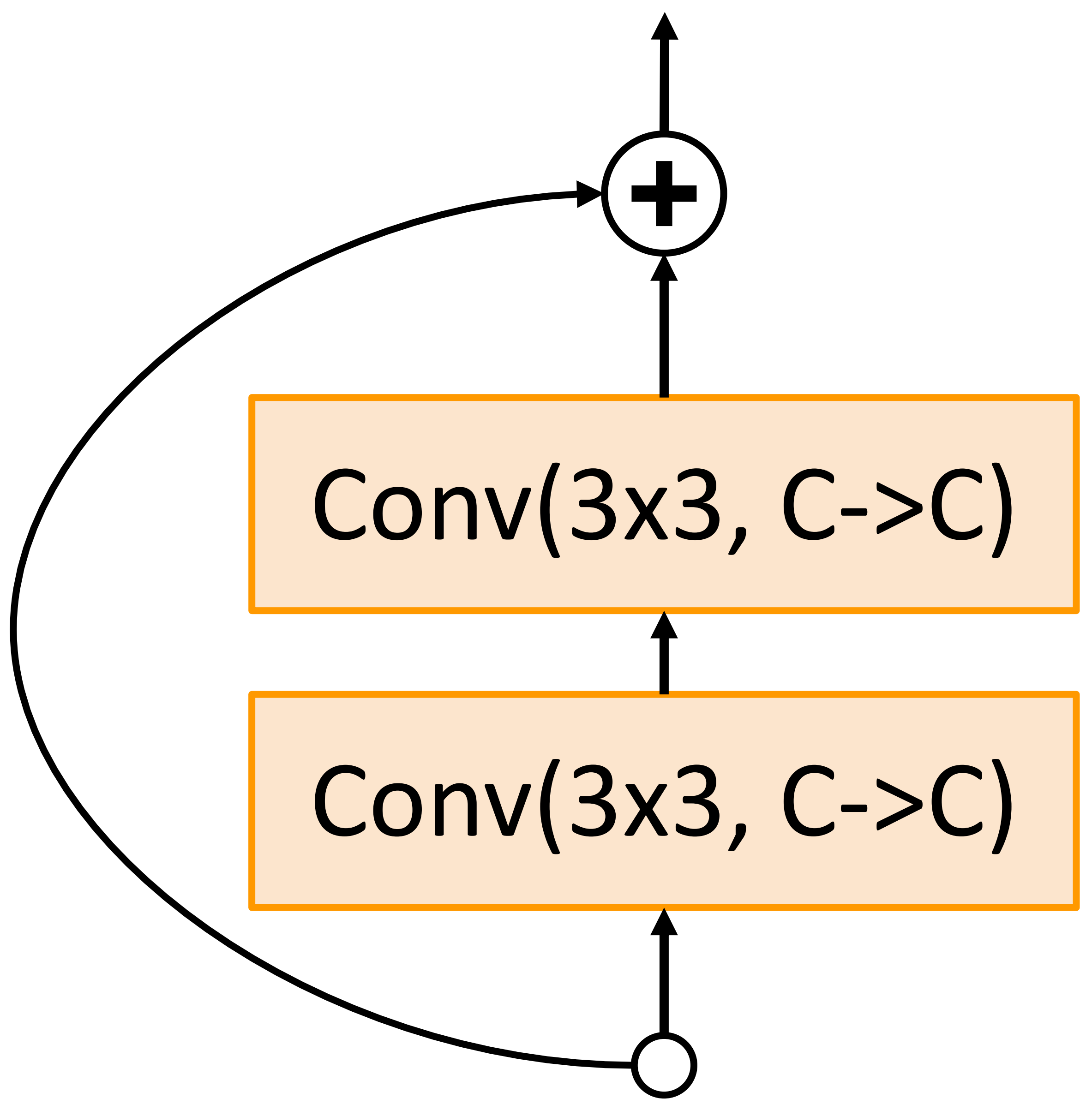
Residual Networks: Basic Block



"Basic"
Residual block

He et al, "Deep Residual Learning for Image Recognition", CVPR 2016

Residual Networks: Basic Block



FLOPs: $9HWC^2$

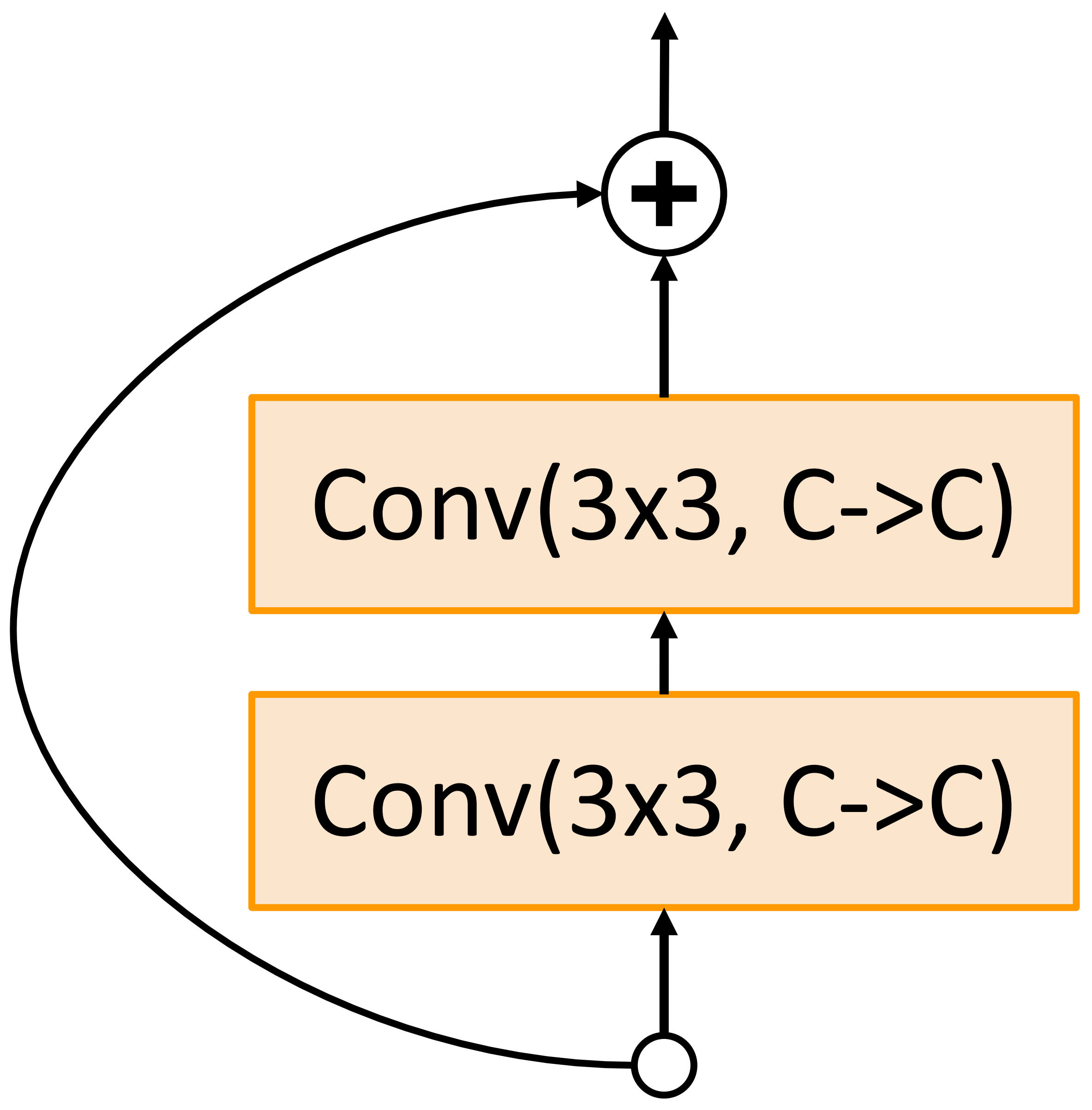
FLOPs: $9HWC^2$

"Basic"
Residual block

Total FLOPs:
$18HWC^2$

He et al, "Deep Residual Learning for Image Recognition", CVPR 2016

Residual Networks: Bottleneck Block

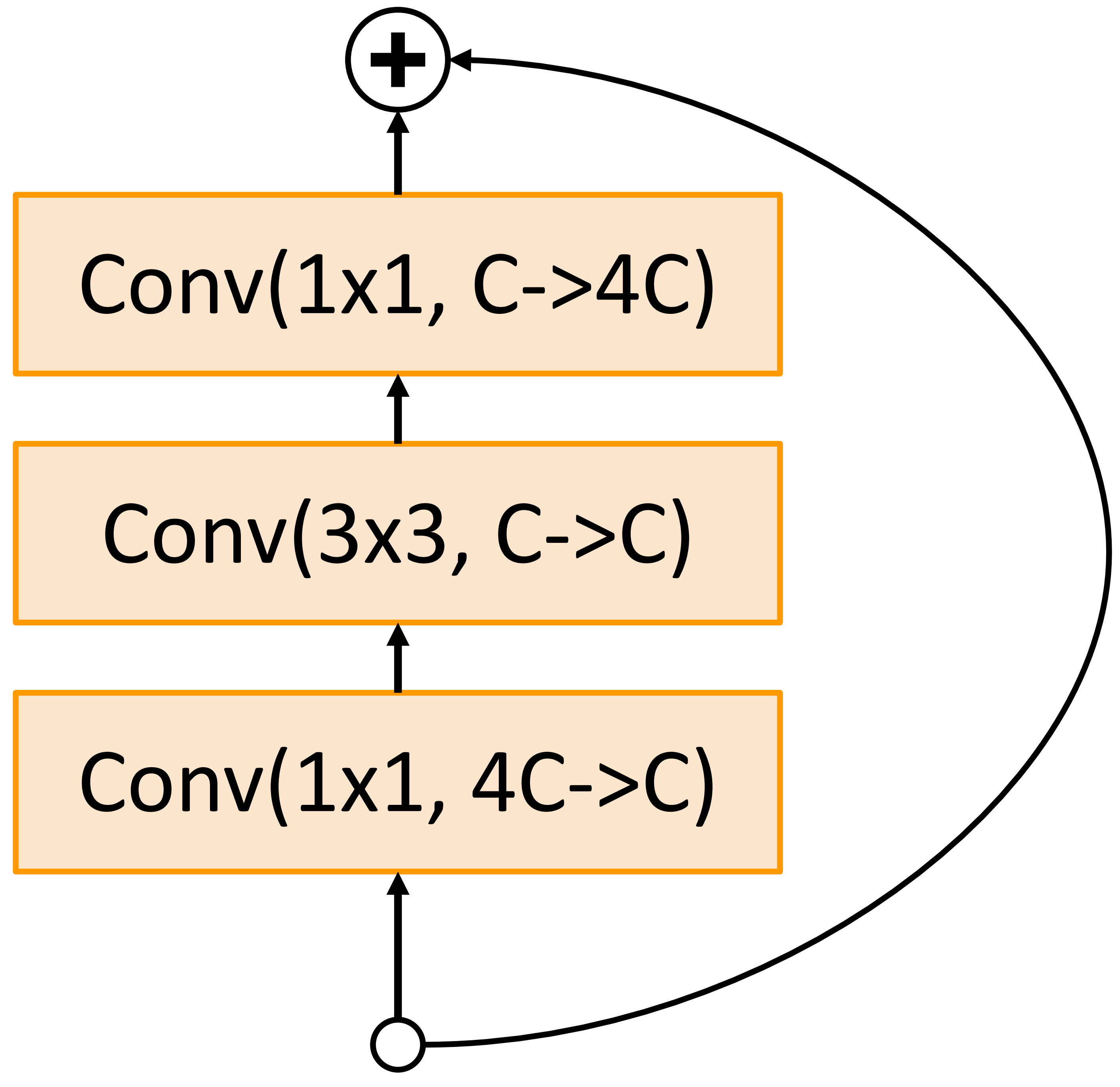


FLOPs: $9HWC^2$

FLOPs: $9HWC^2$

"Basic"
Residual block

Total FLOPs:
$18HWC^2$

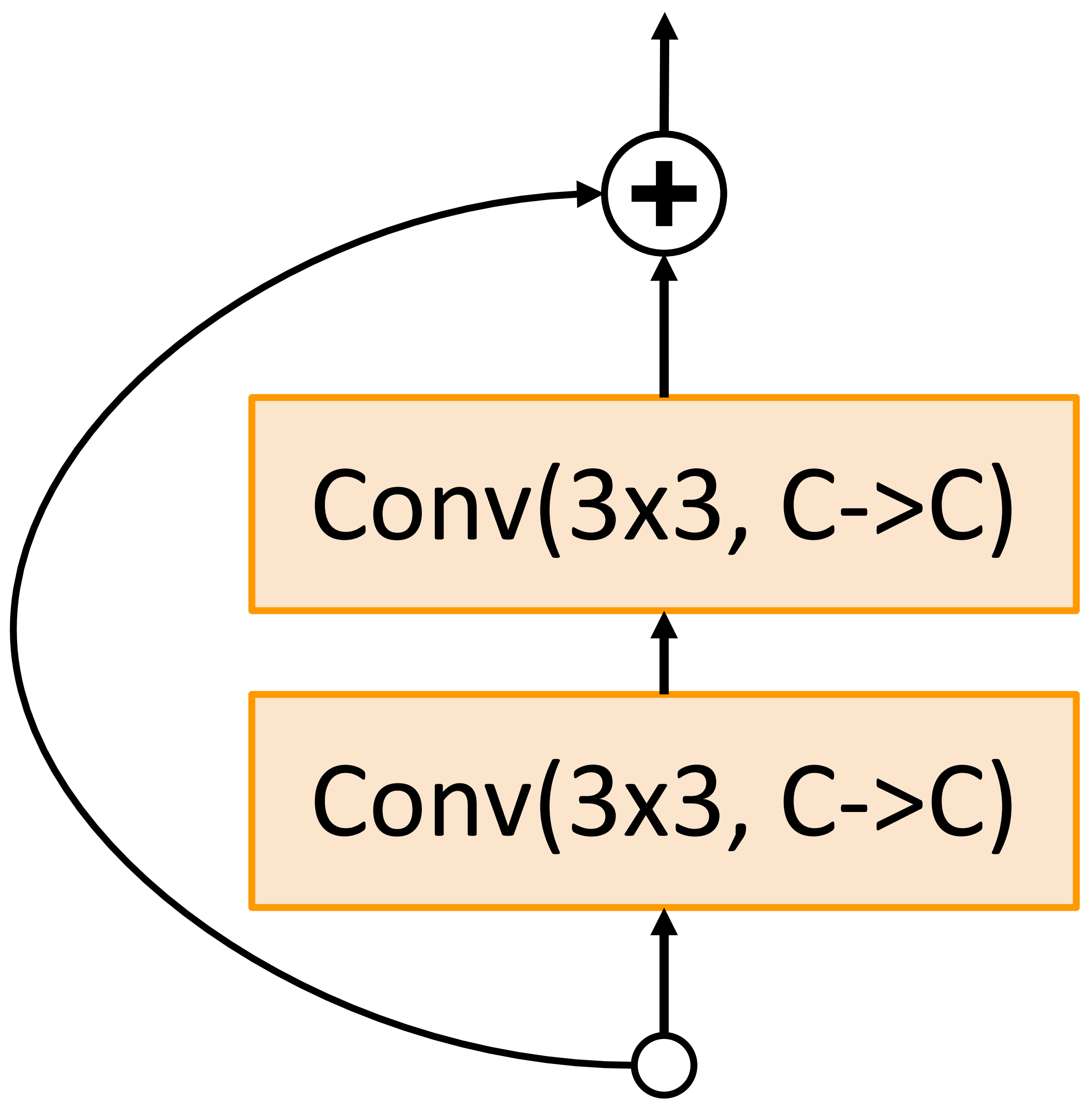


"Bottleneck"
Residual block

He et al, "Deep Residual Learning for Image Recognition", CVPR 2016

Residual Networks: Bottleneck Block

More layers, less computational cost!



"Basic"
Residual block

FLOPs: $9HWC^2$

FLOPs: $9HWC^2$

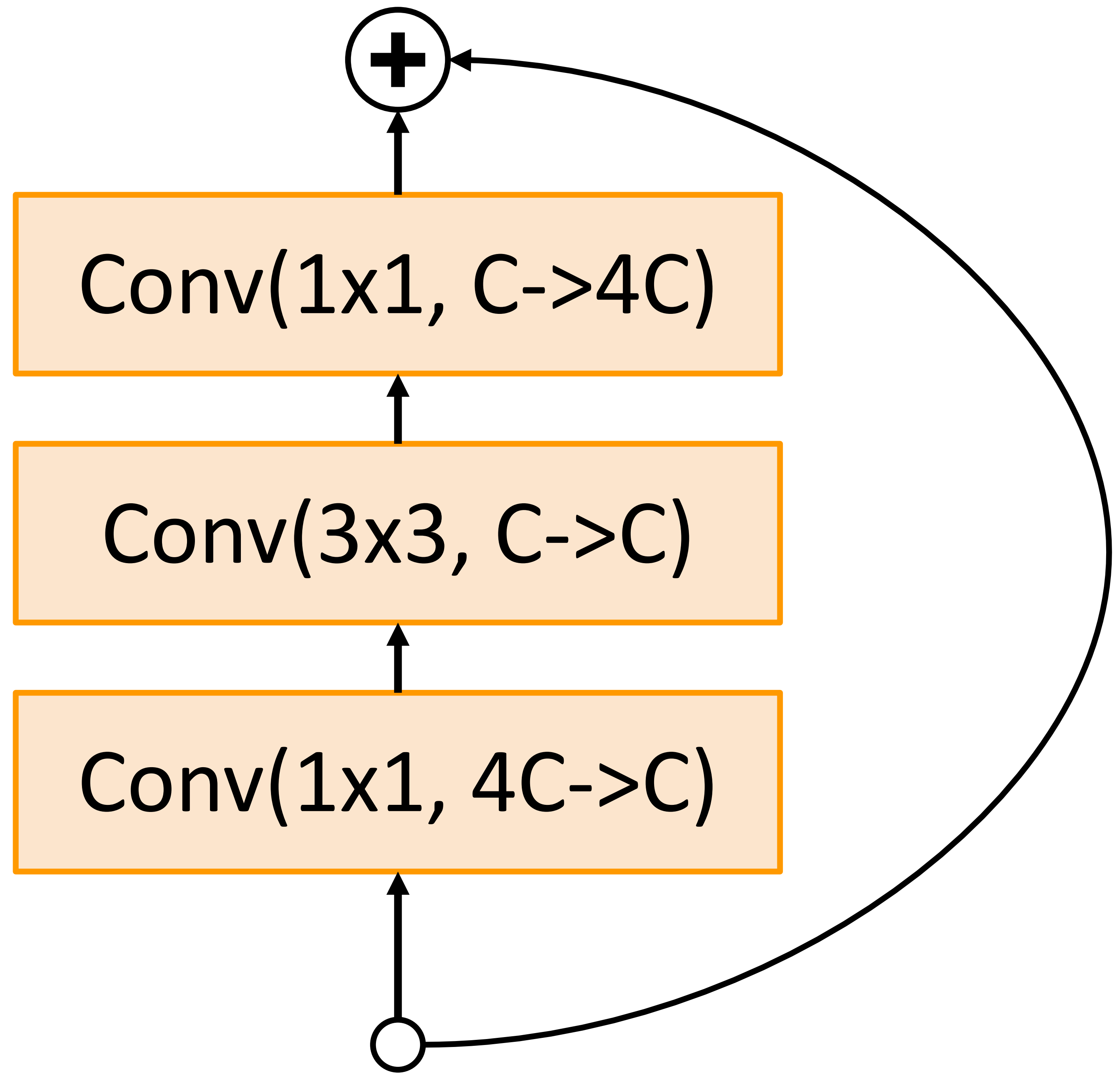
Total FLOPs:
$18HWC^2$

FLOPs: $4HWC^2$

FLOPs: $9HWC^2$

FLOPs: $4HWC^2$

Total FLOPs:
$17HWC^2$



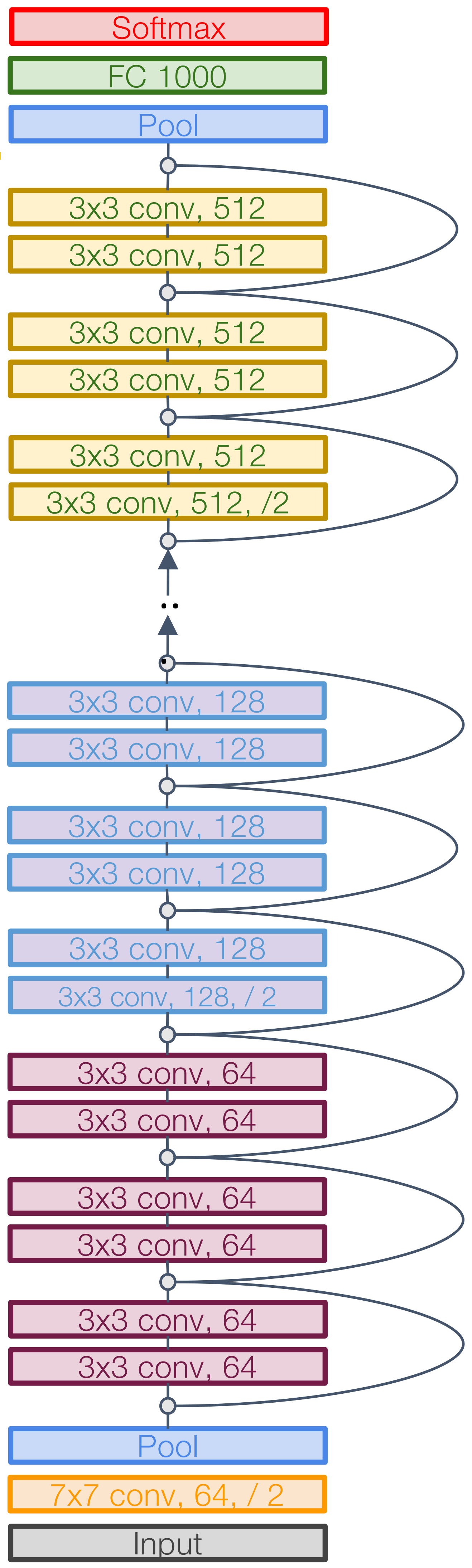
"Bottleneck"
Residual block

He et al, "Deep Residual Learning for Image Recognition", CVPR 2016

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**	**ImageNet**
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



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

Residual Networks

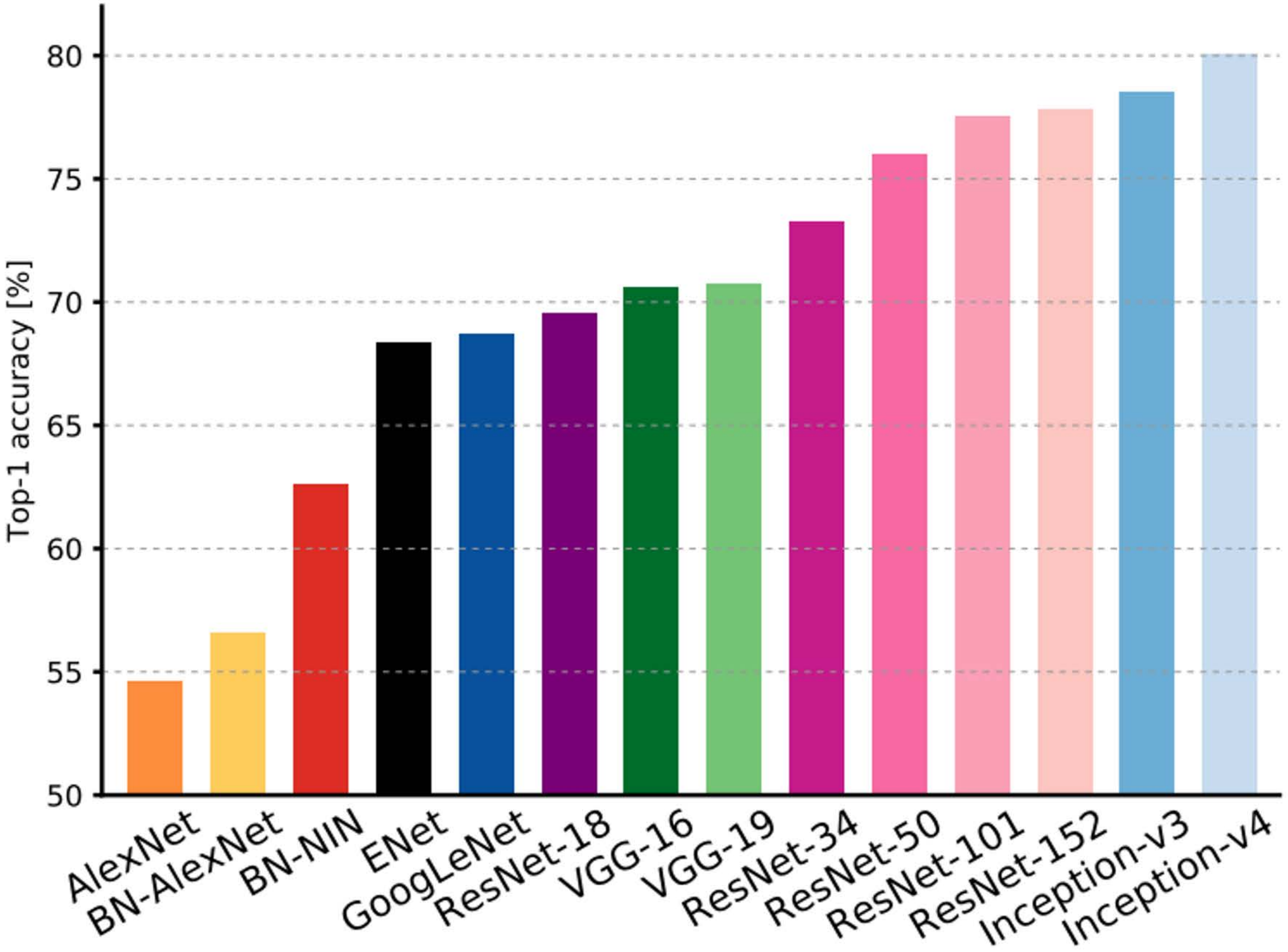
- Able to train very deep networks
- Deeper networks do better than shallow networks (as expected)
- Swept 1st place in all ILSVRC and COCO 2015 competitions
- Still widely used today

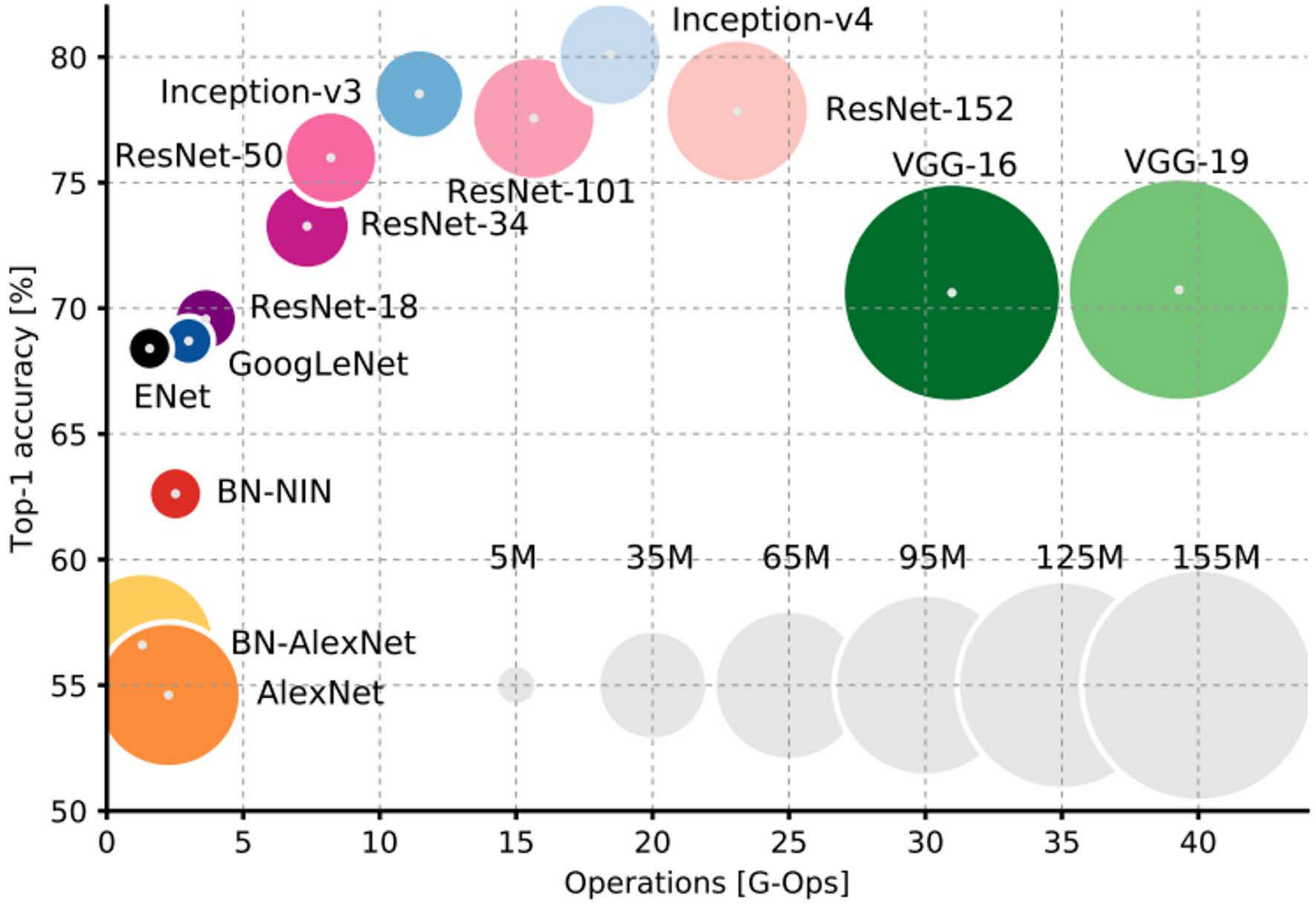
MSRA @ ILSVRC & COCO 2015 Competitions

- **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

He et al, "Deep Residual Learning for Image Recognition", CVPR 2016

Comparing Complexity

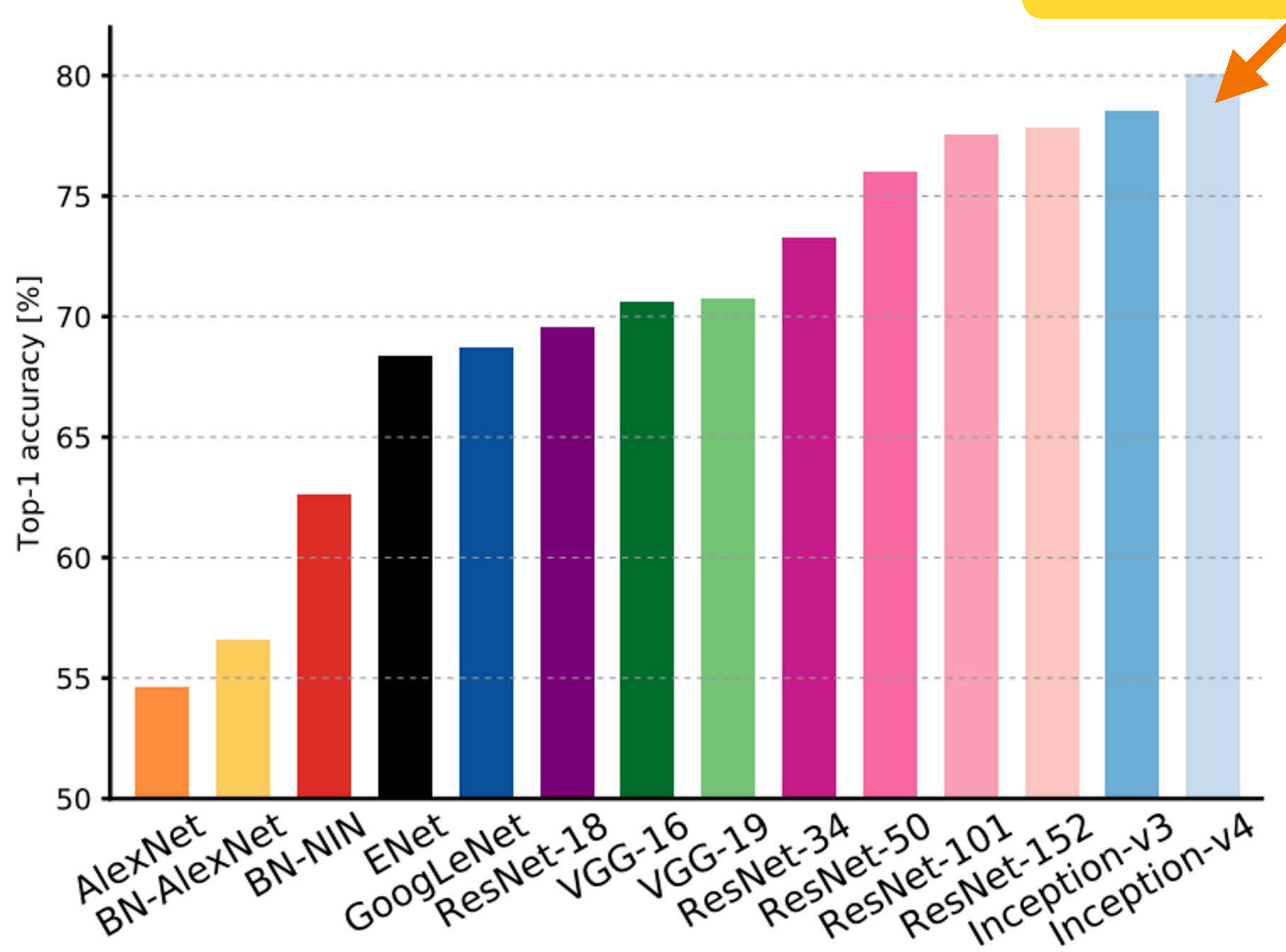


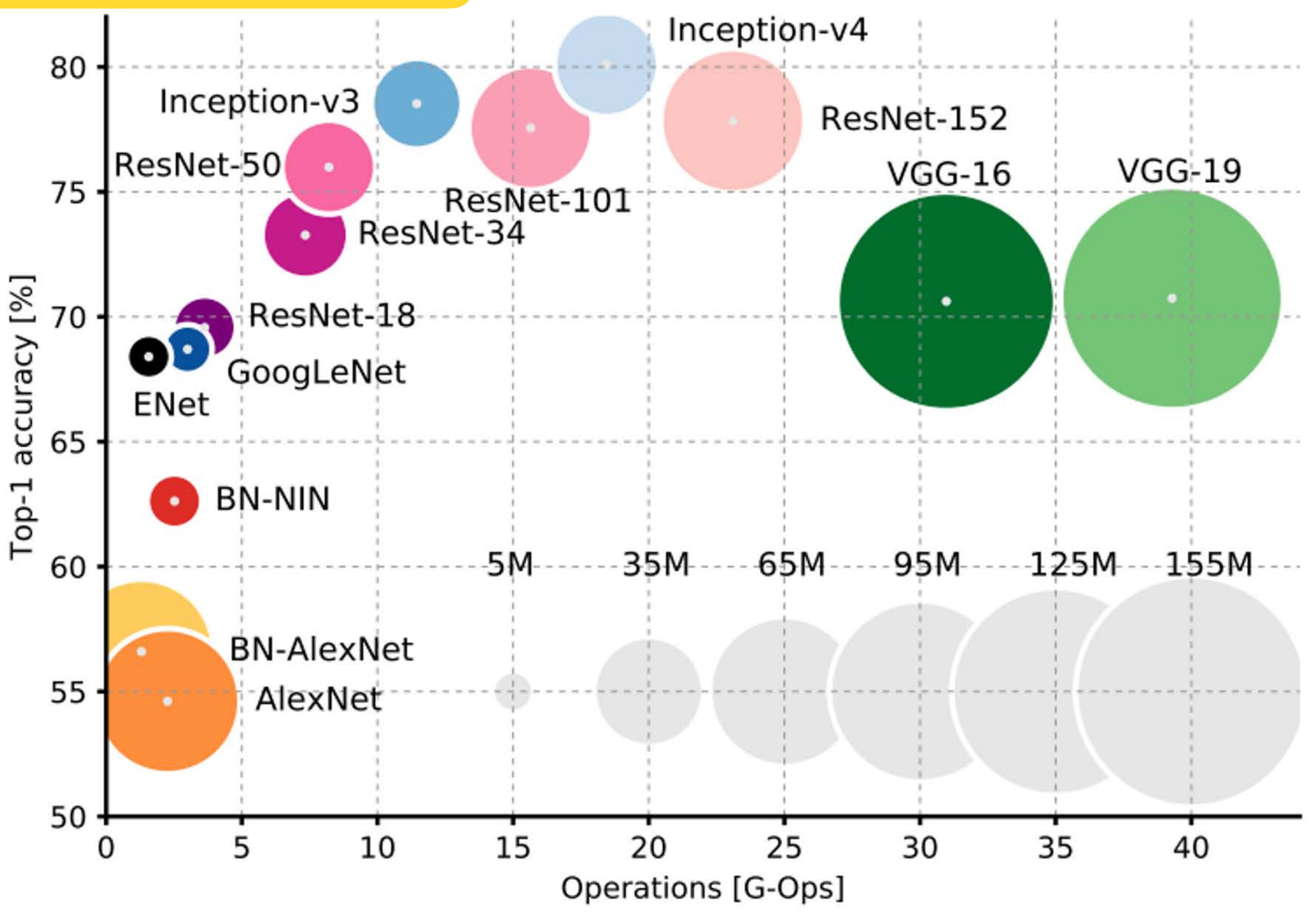


Canziani et al, "An analysis of deep neural network models for practical applications", 2017

Comparing Complexity

Inception-v4: ResNet + Inception!

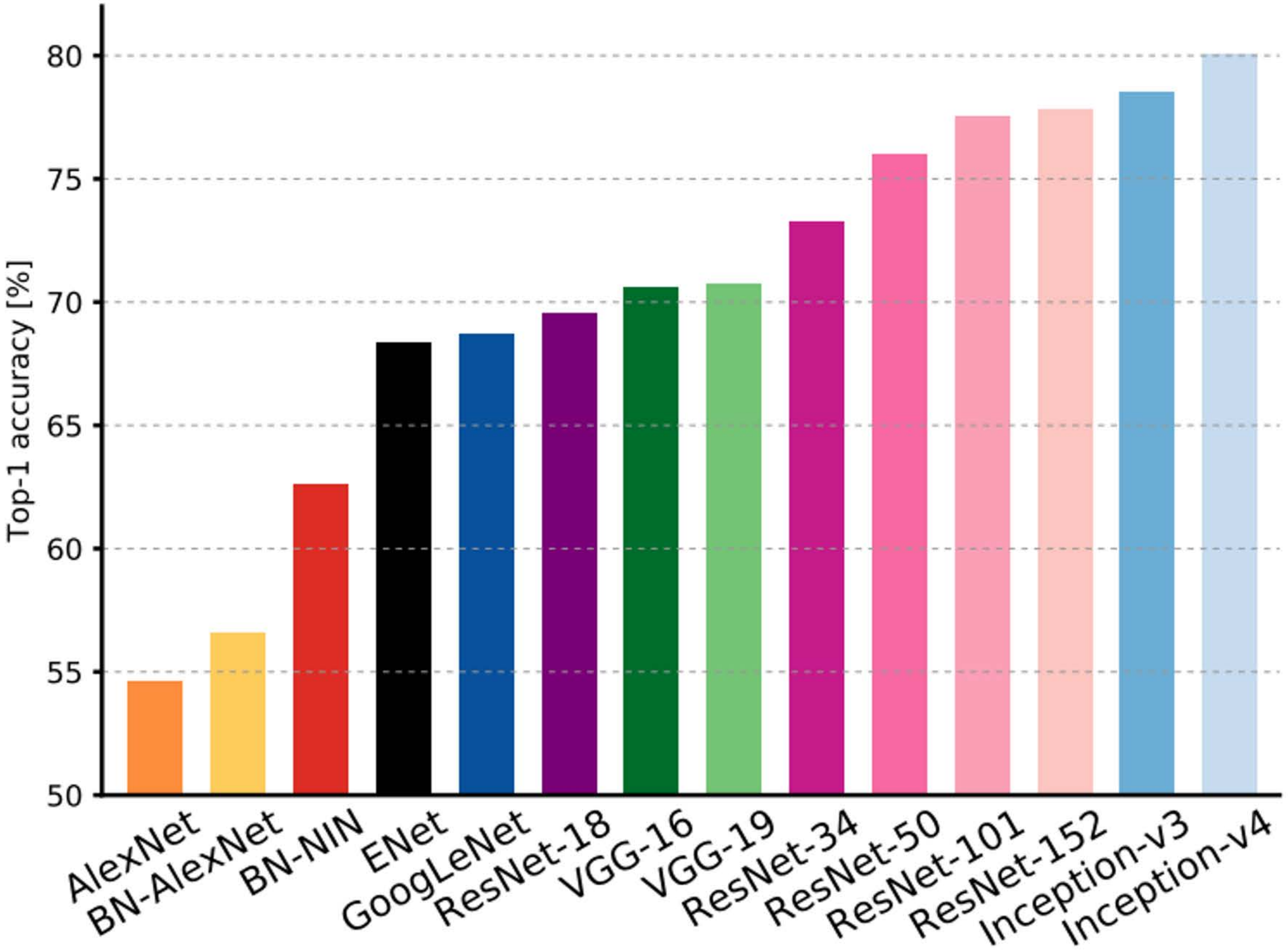


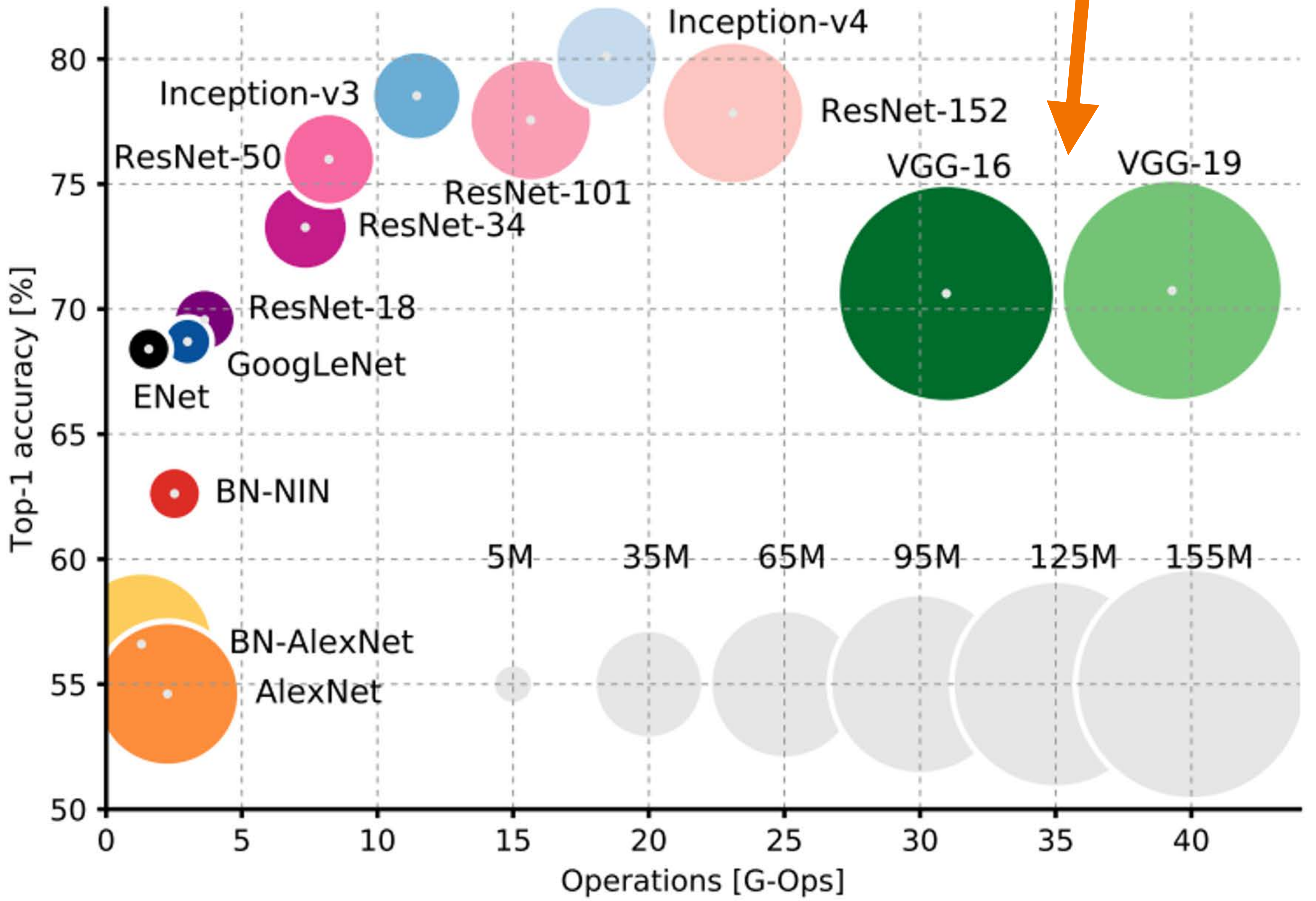


Canziani et al, “An analysis of deep neural network models for practical applications”, 2017

Comparing Complexity

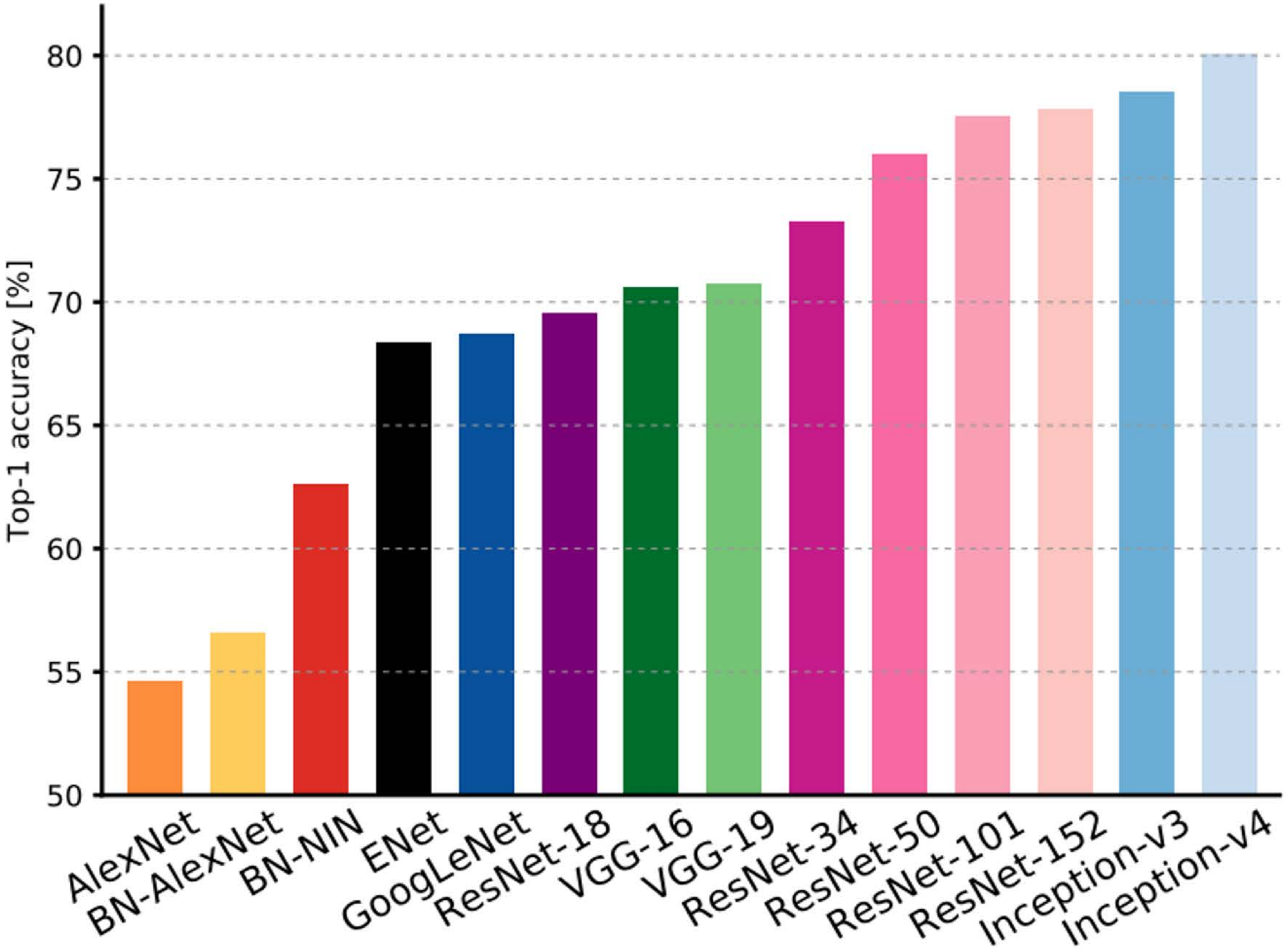
VGG:
Highest memory,
most operations

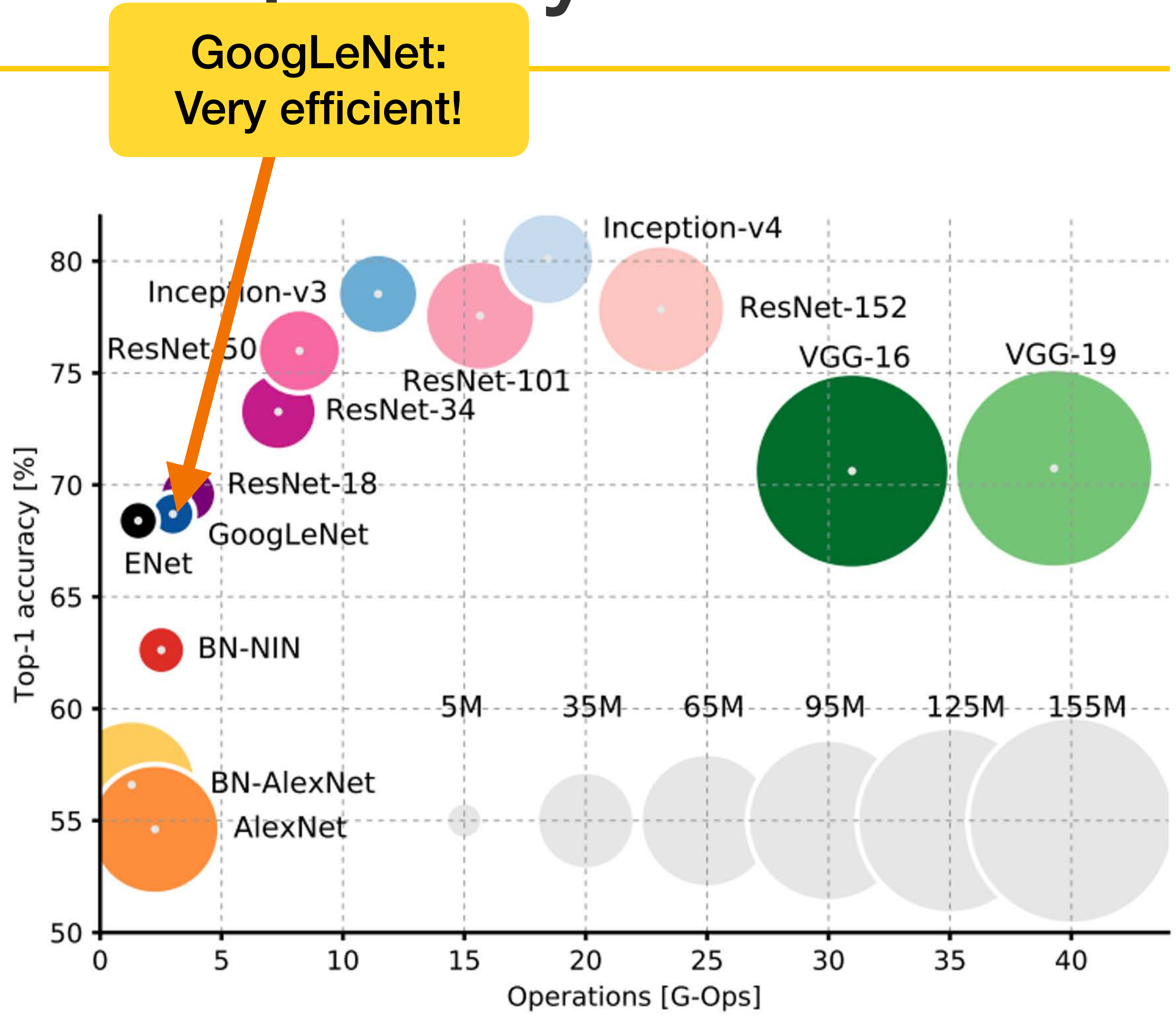




Canziani et al, “An analysis of deep neural network models for practical applications”, 2017

Comparing Complexity

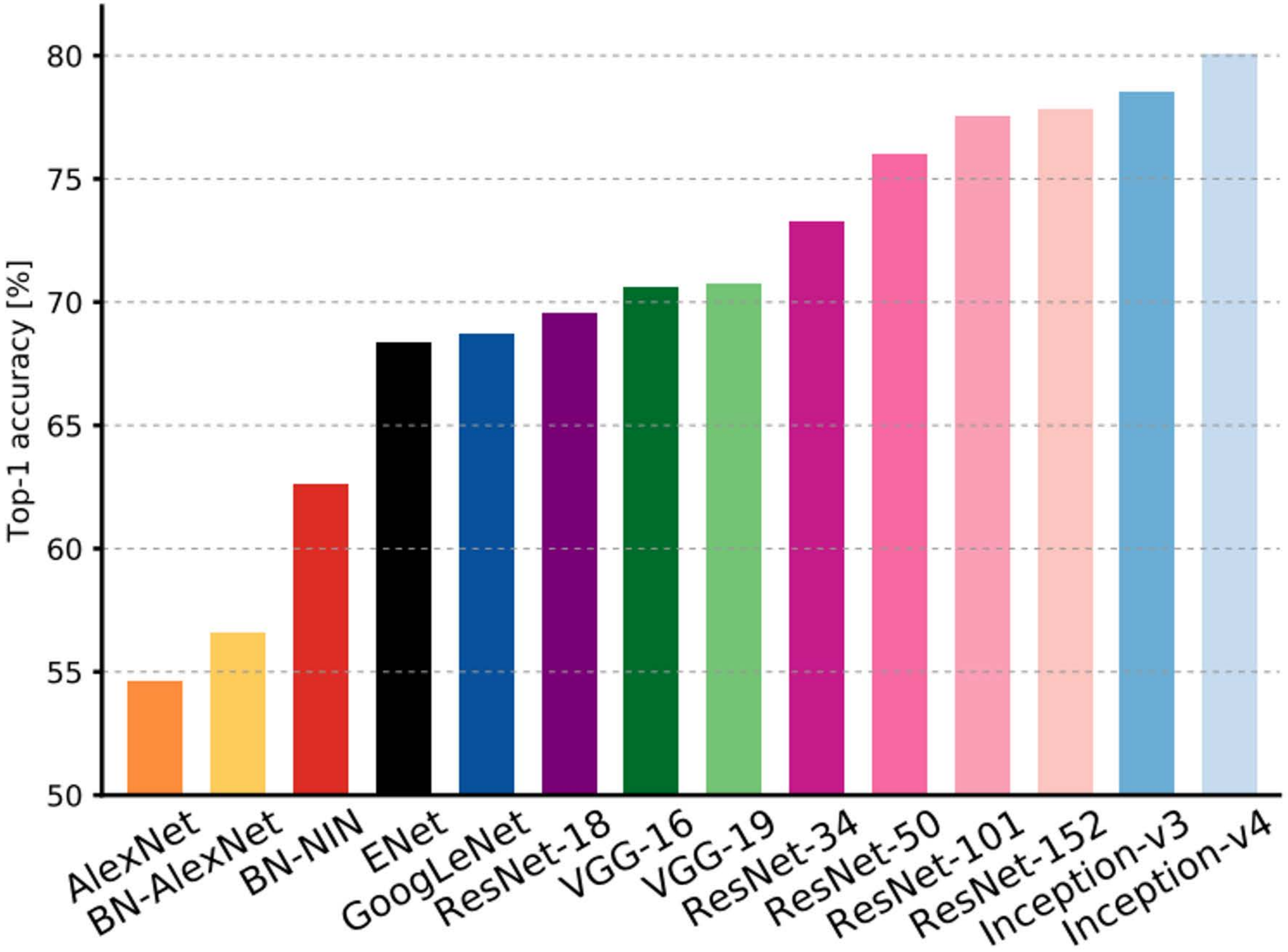


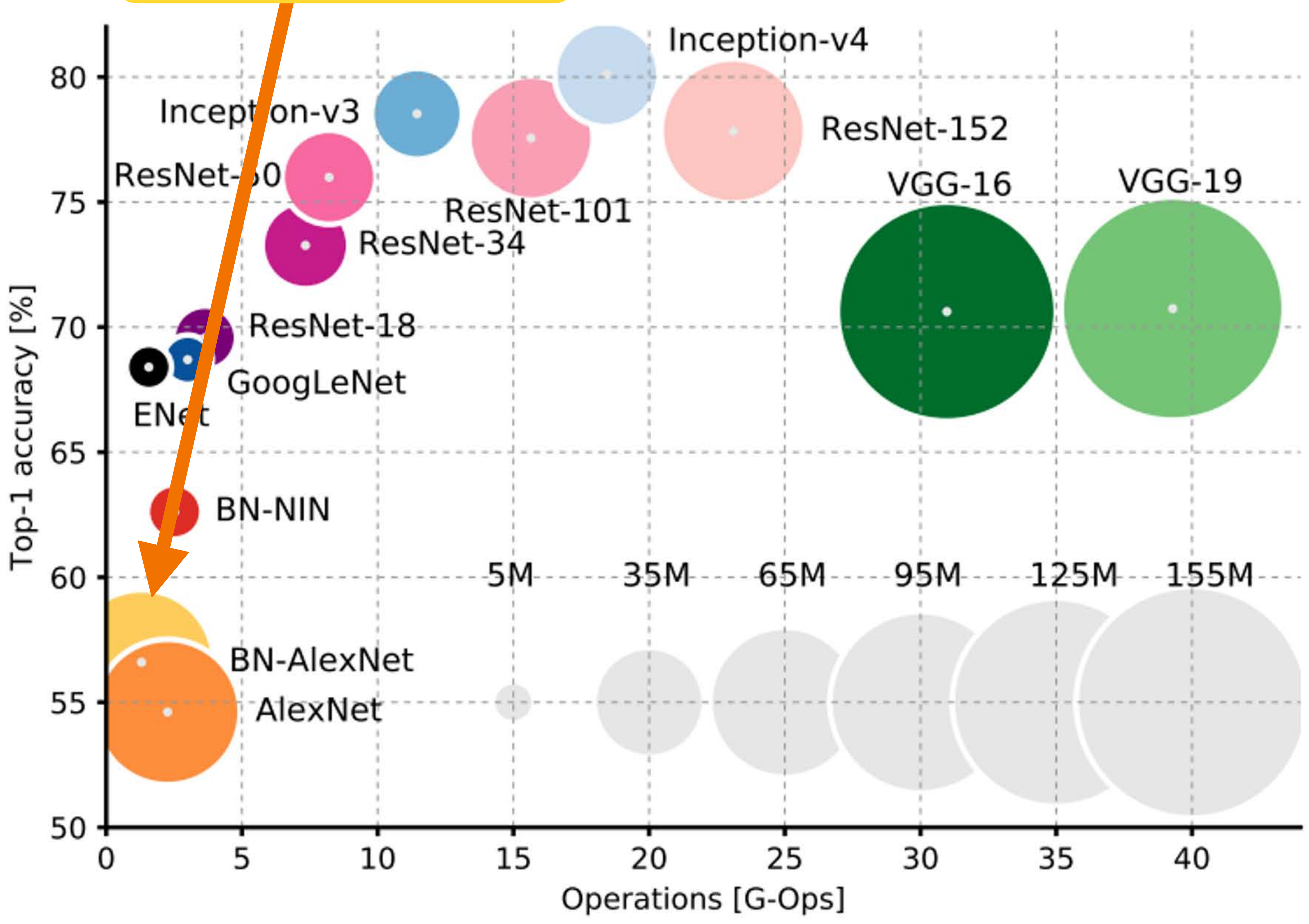


Canziani et al, "An analysis of deep neural network models for practical applications", 2017

Comparing Complexity

AlexNet: Low compute, lots of parameters

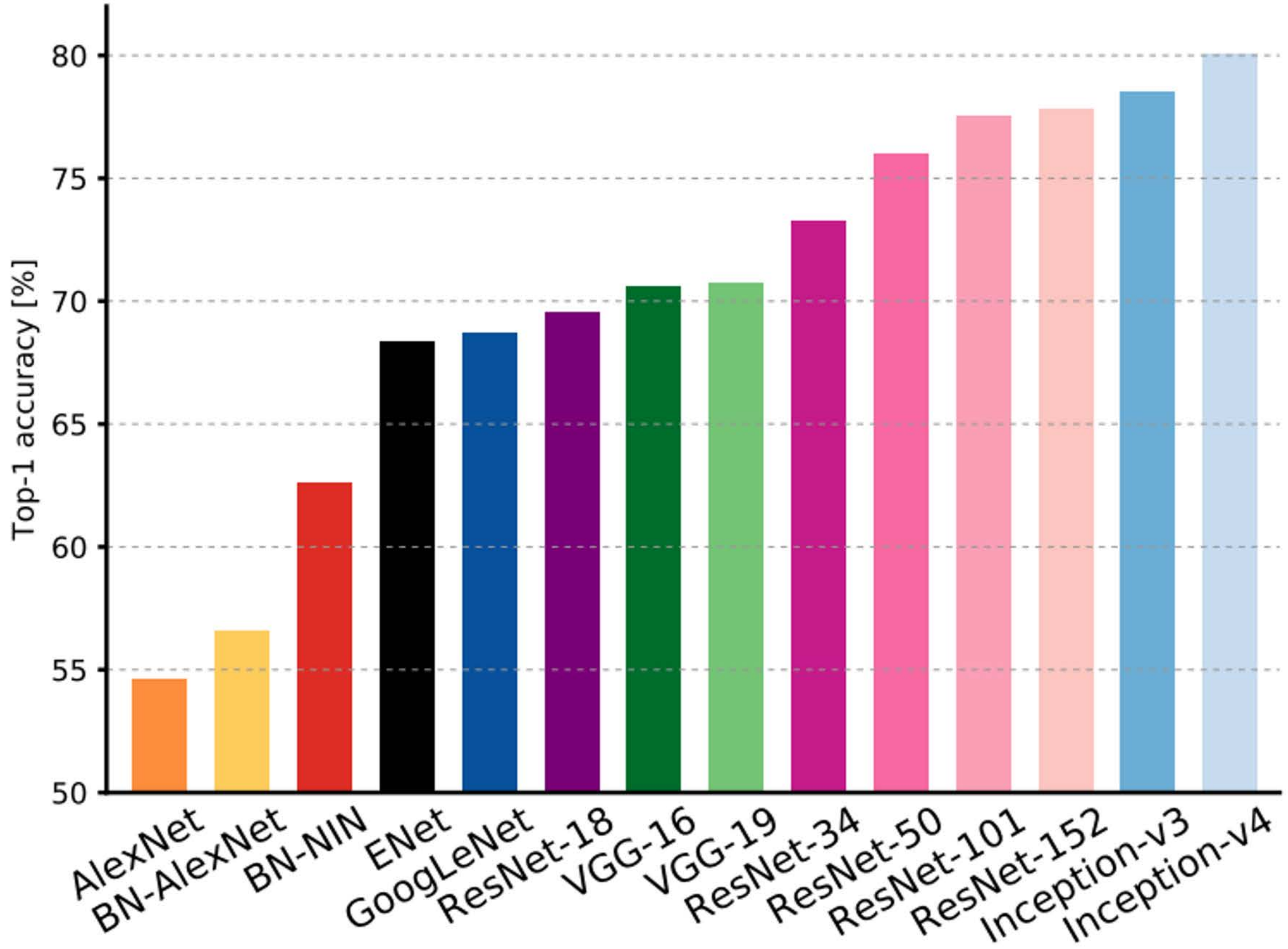


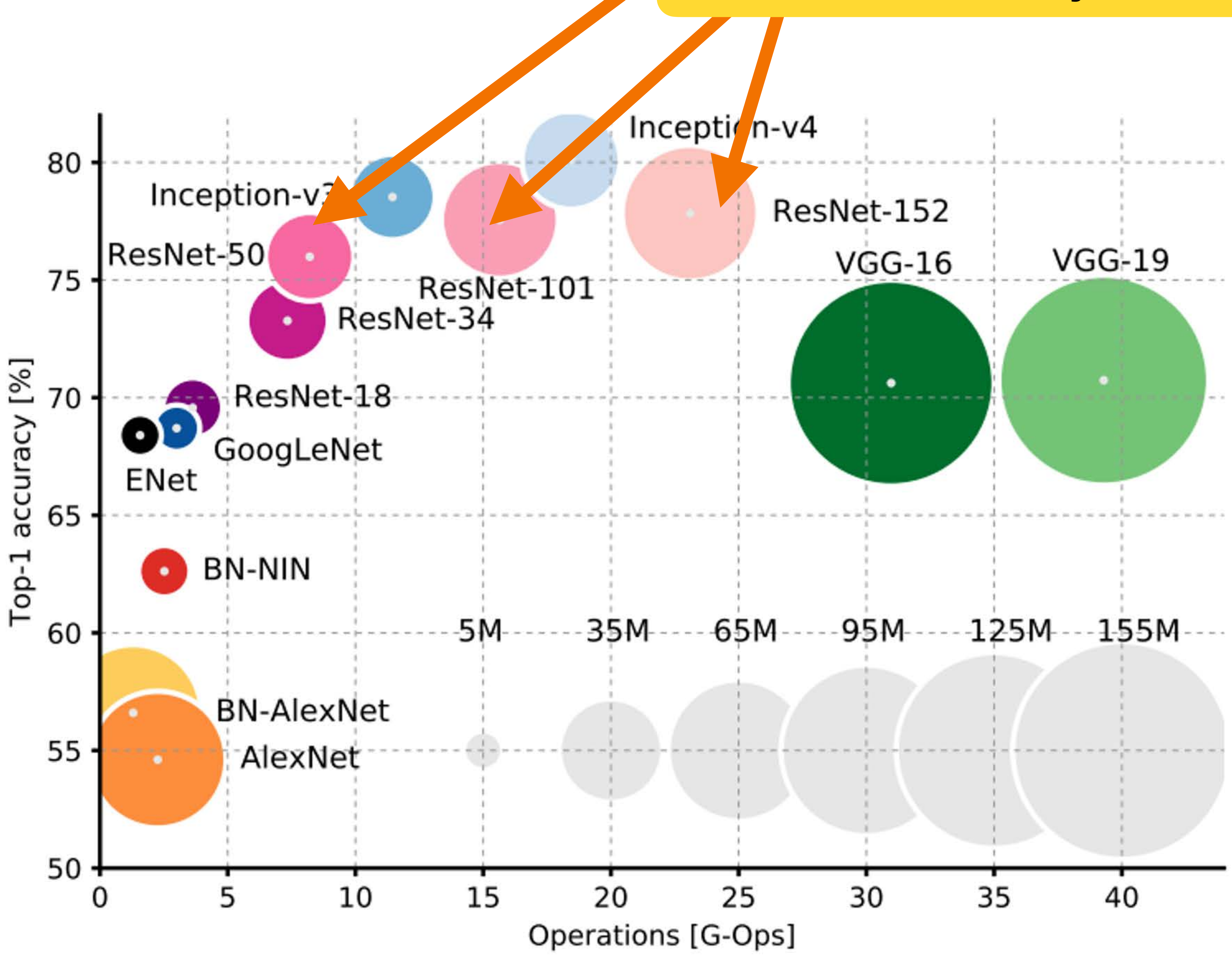


Canziani et al, "An analysis of deep neural network models for practical applications", 2017

Comparing Complexity

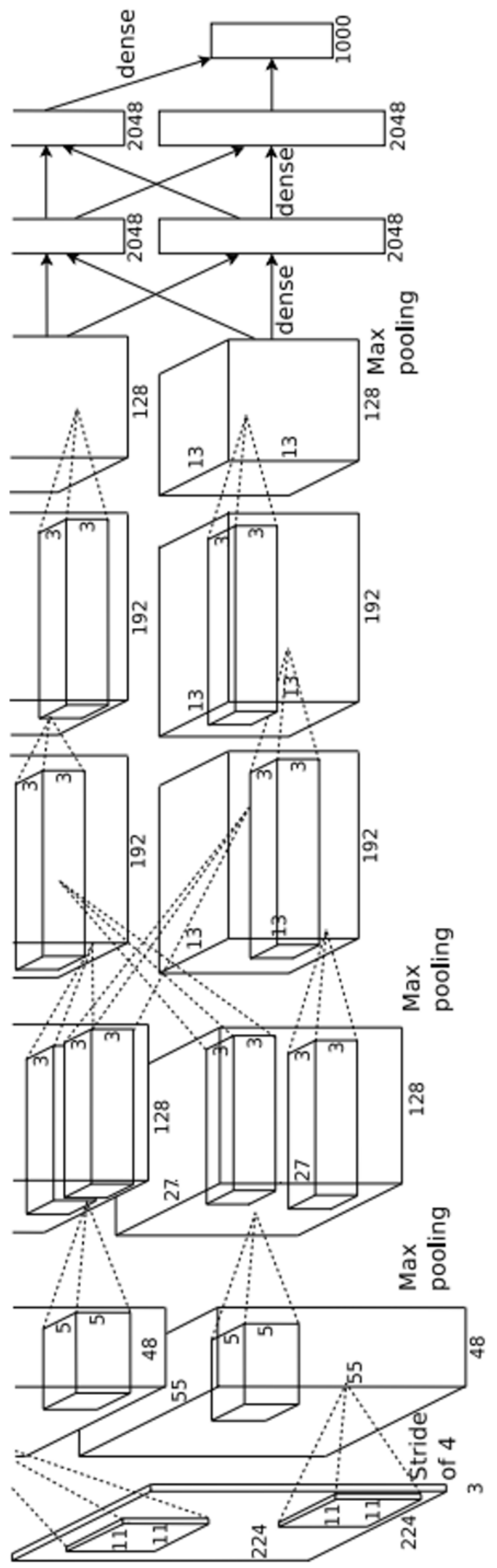
ResNet: Simple design, moderate efficiency, high accuracy





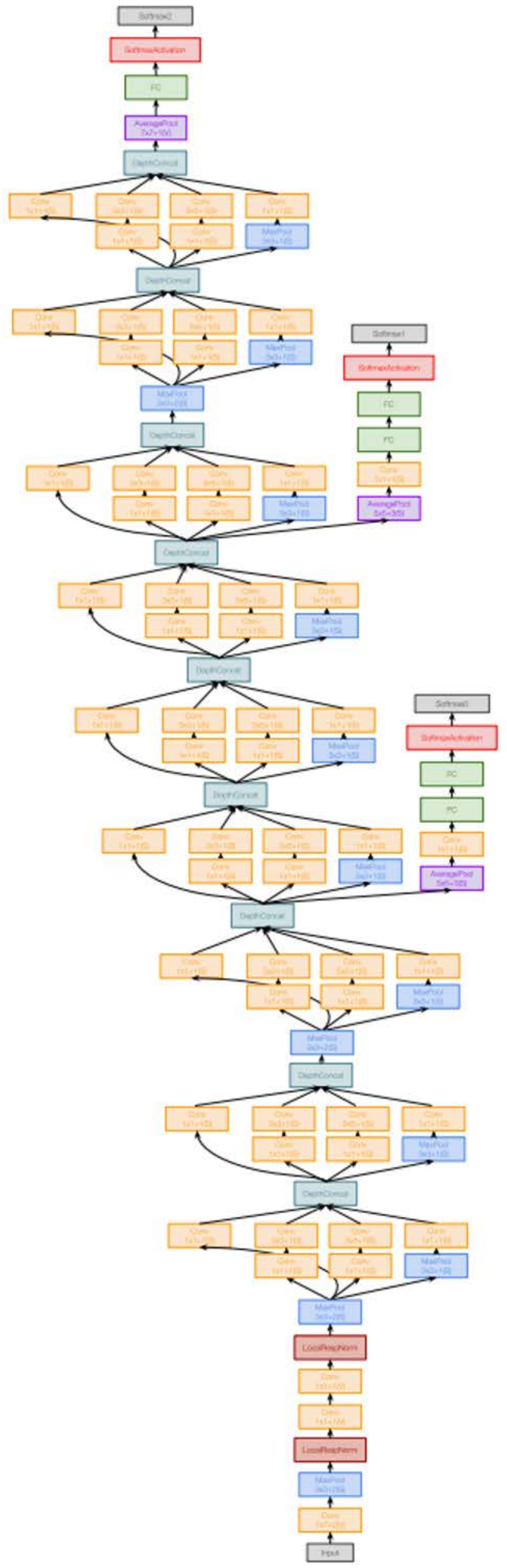
Canziani et al, "An analysis of deep neural network models for practical applications", 2017

Recap

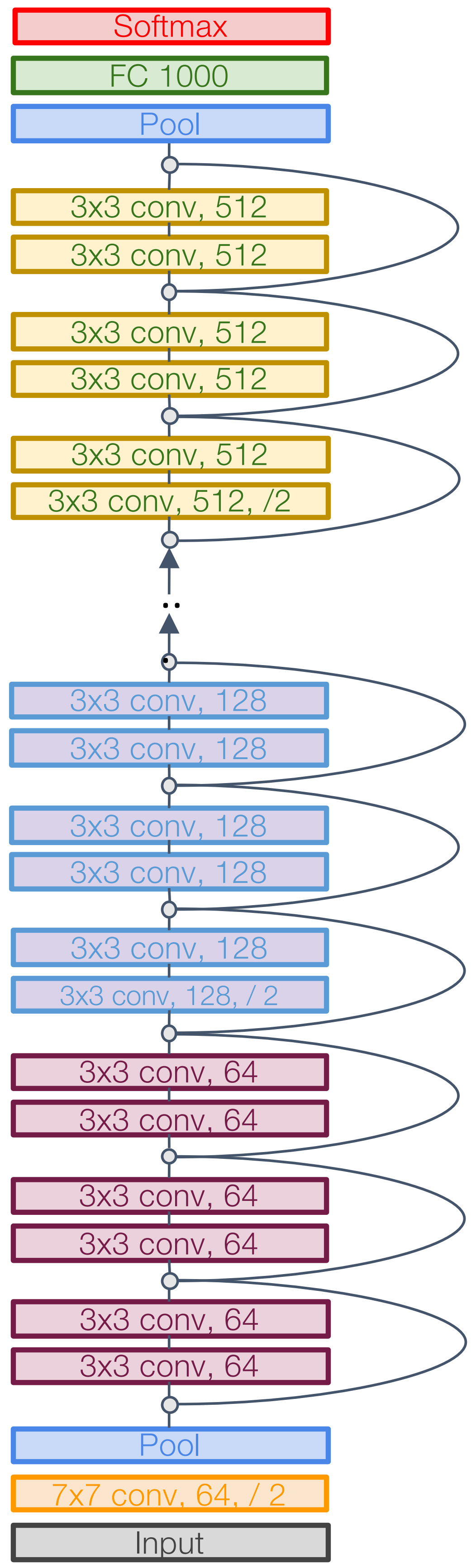


AlexNet

VGG



GoogLeNet

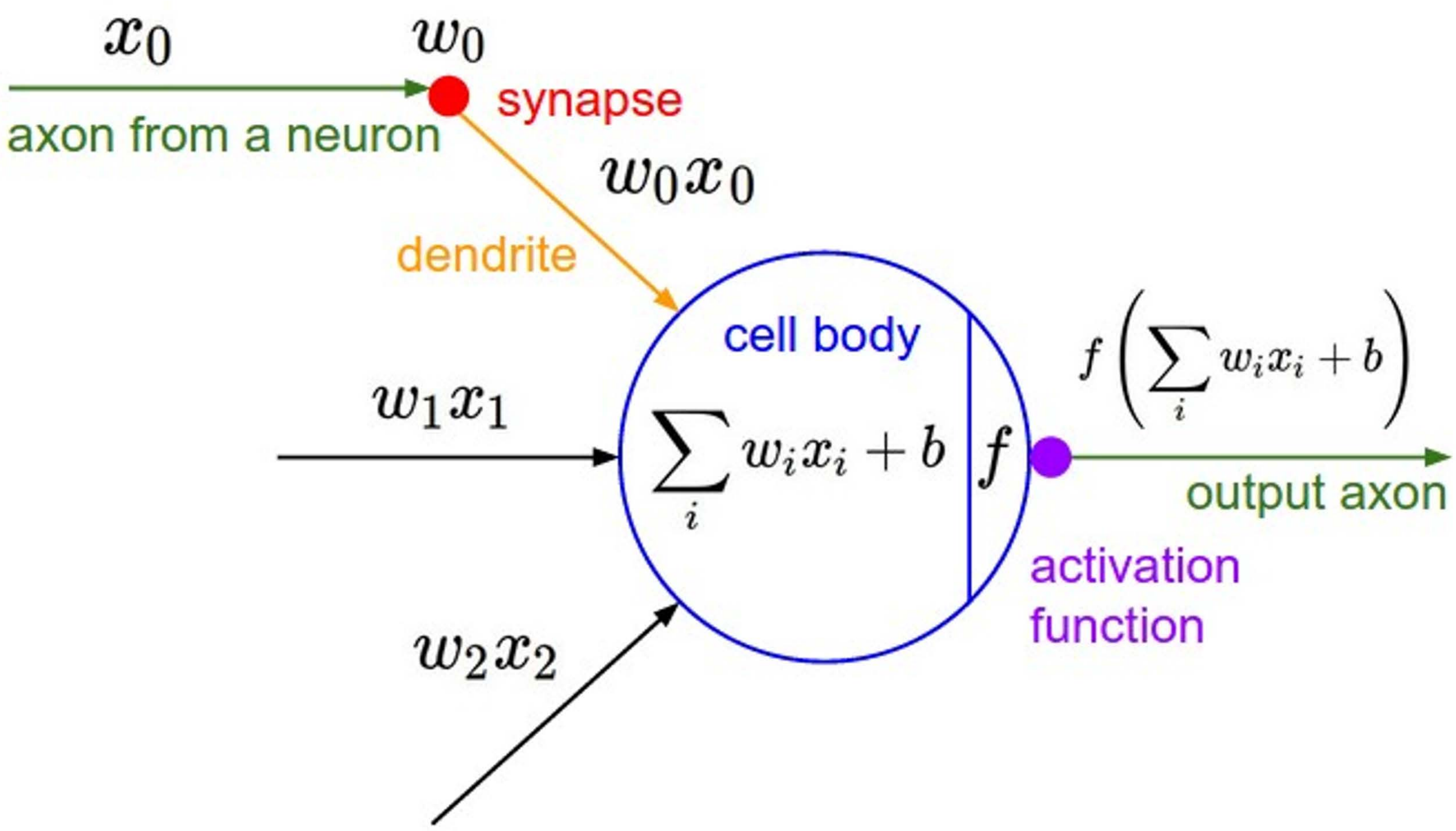


ResNet

Overview

1. One time setup: **Today**
    - Activation functions, data preprocessing, weight initialization, regularization
2. Training dynamics: **Next time**
    - Learning rate schedules; large-batch training; hyperparameter optimization
3. After training:
    - Model ensembles, transfer learning

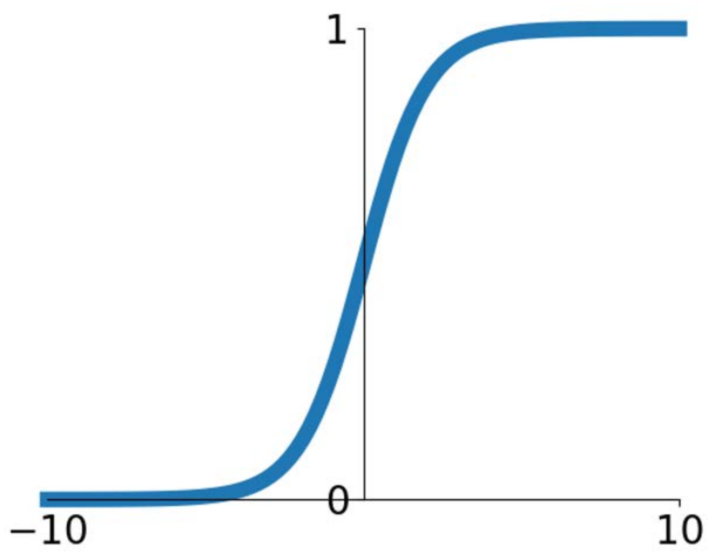
Activation Functions



Activation Functions

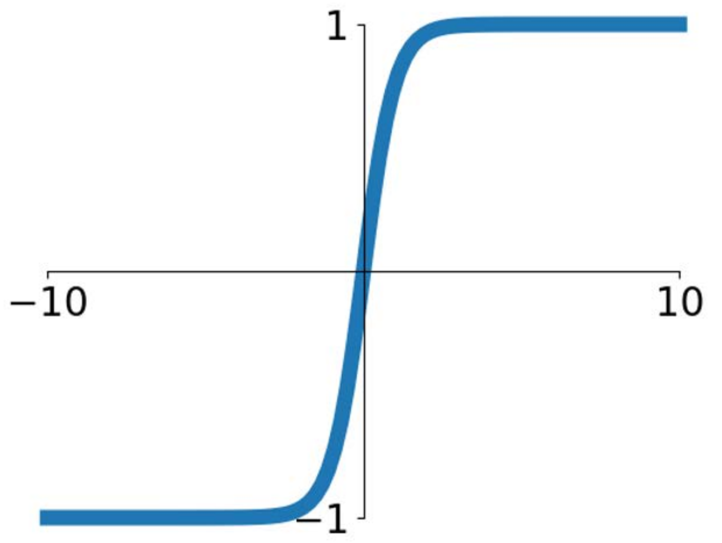
Sigmoid

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$



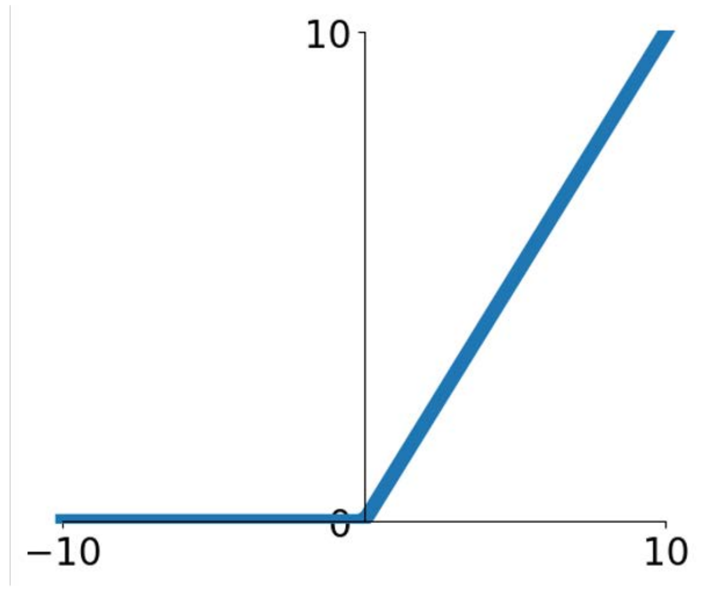
tanh

$$\tanh(x)$$



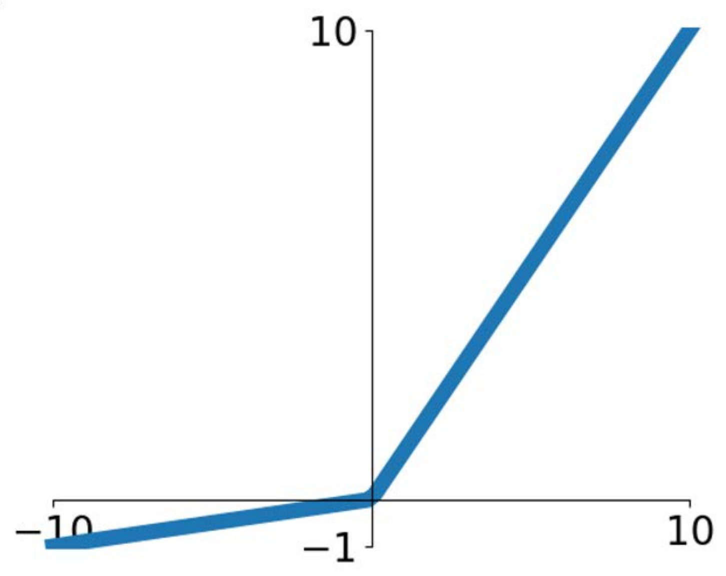
ReLU

$$\max(0,x)$$



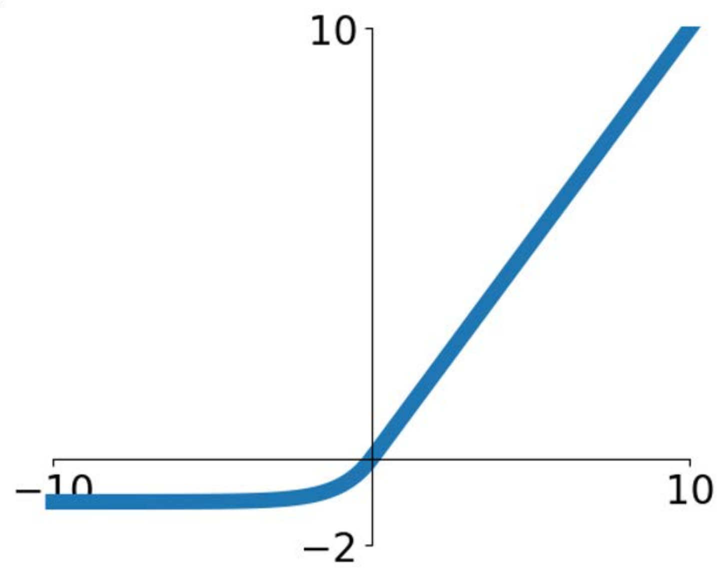
Leaky ReLU

$$\max(0.1x, x)$$



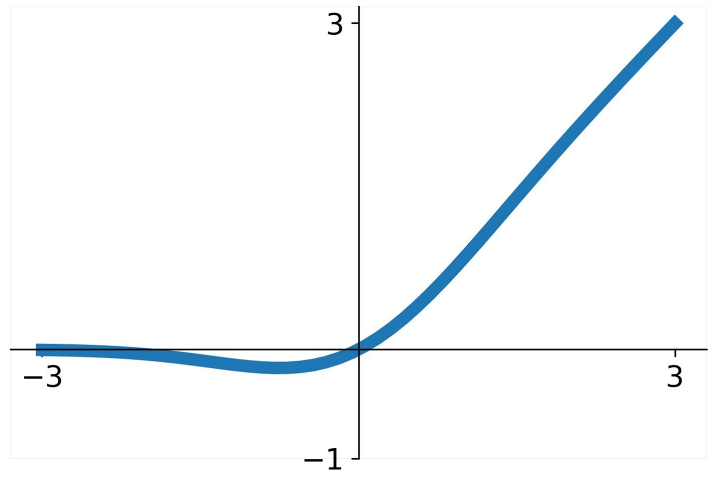
ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(\exp^x - 1) & x < 0 \end{cases}$$



GELU

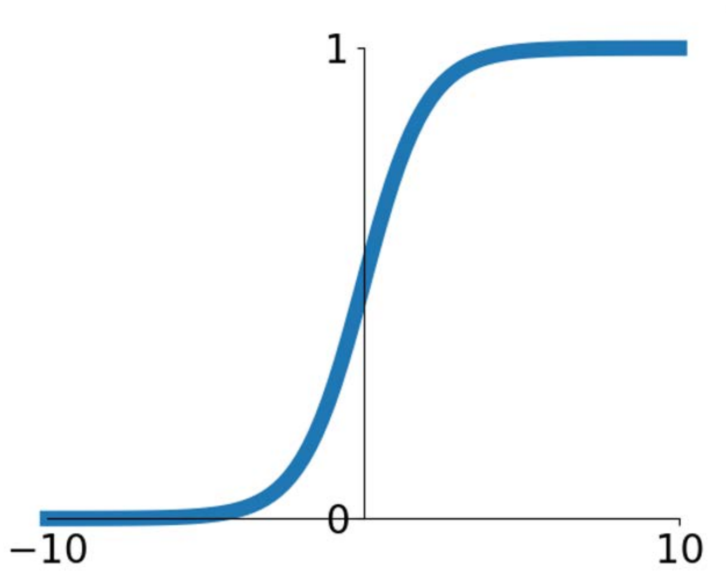
$$\approx x\alpha(1.702x)$$



Activation Functions: Sigmoid

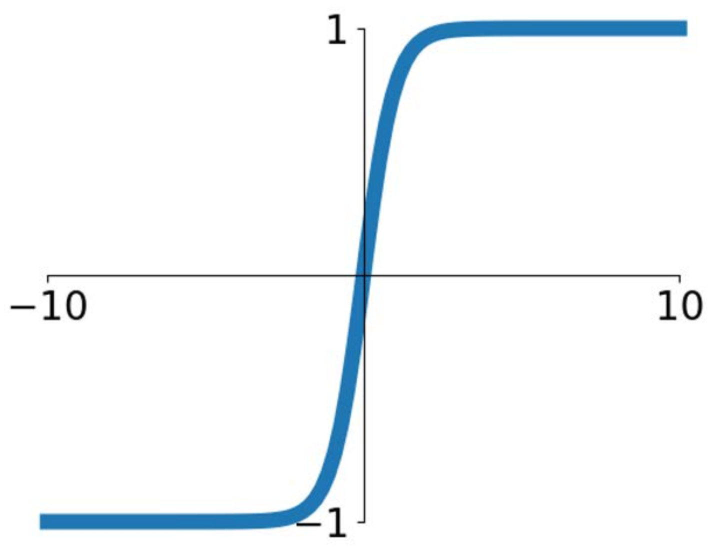
Sigmoid

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$



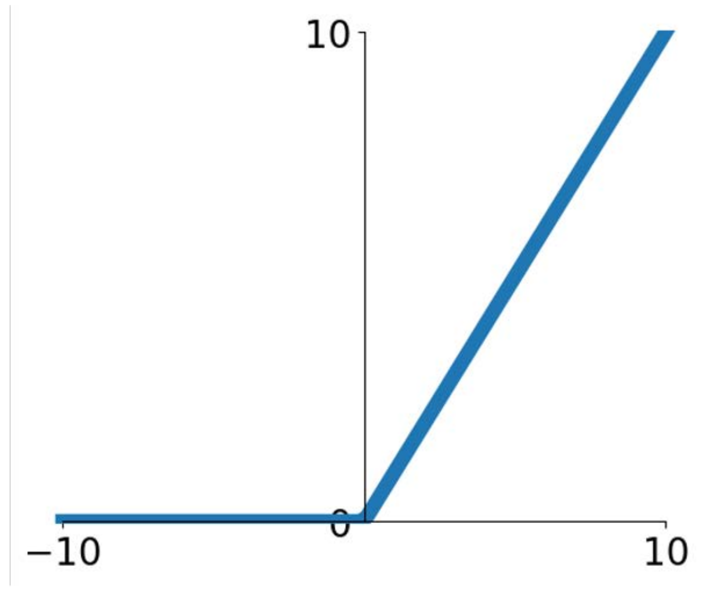
tanh

$$\tanh(x)$$



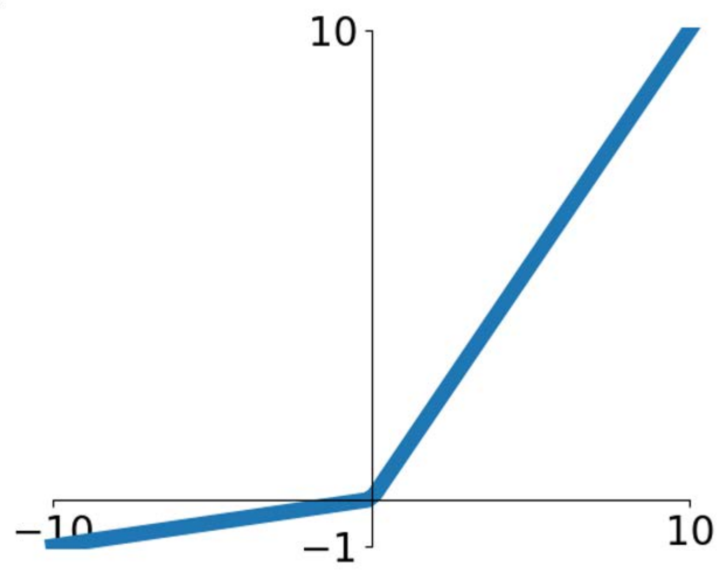
ReLU

$$\max(0,x)$$



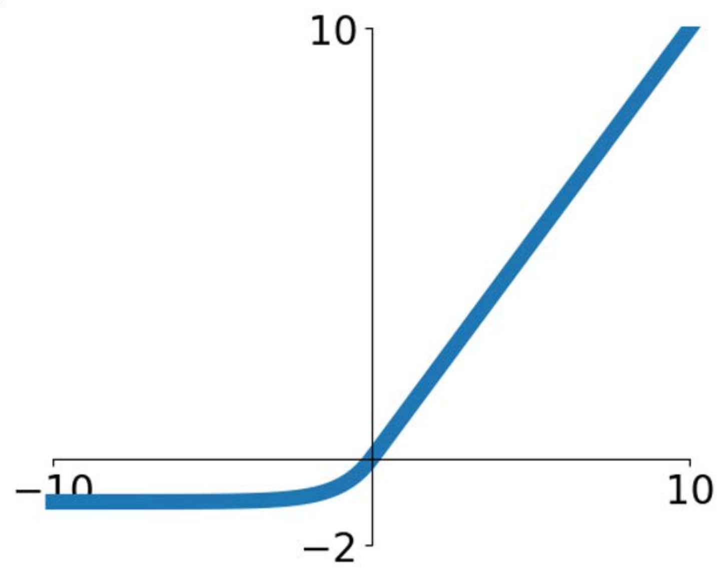
Leaky ReLU

$$\max(0.1x, x)$$



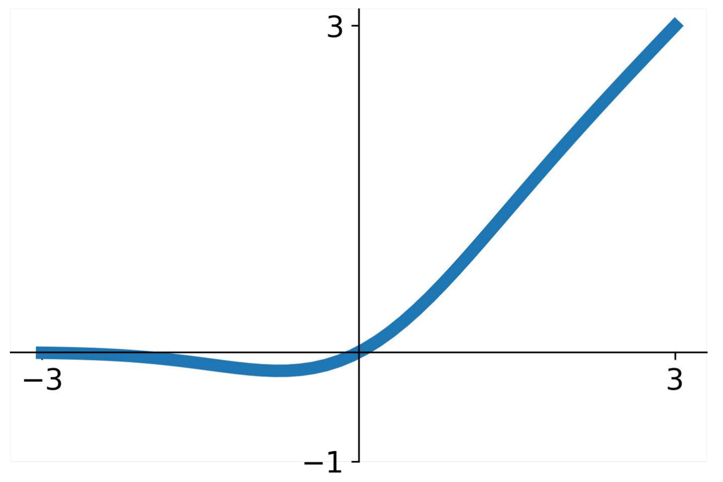
ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(\exp^x - 1) & x < 0 \end{cases}$$

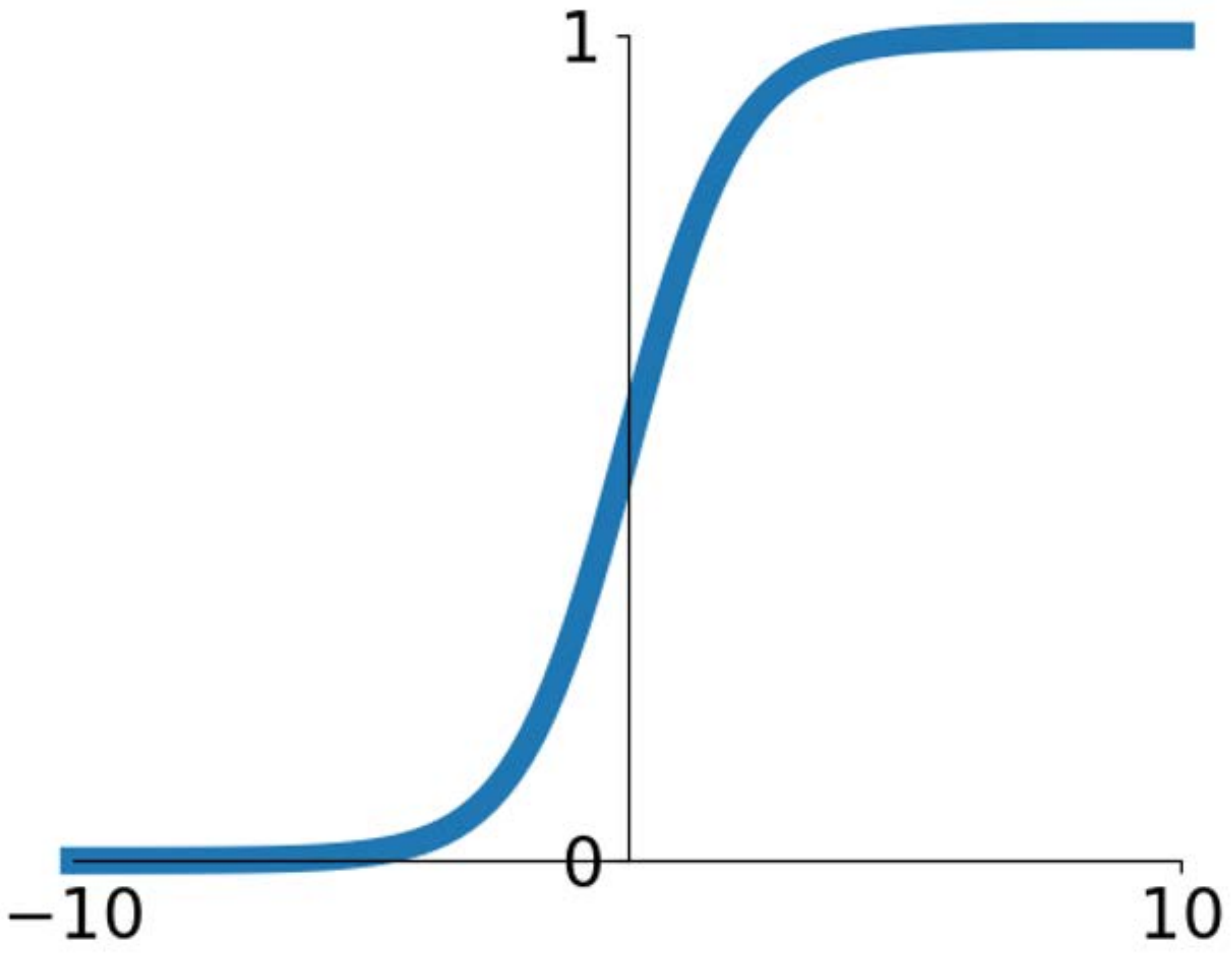


GELU

$$\approx x\alpha(1.702x)$$



Activation Functions: Sigmoid

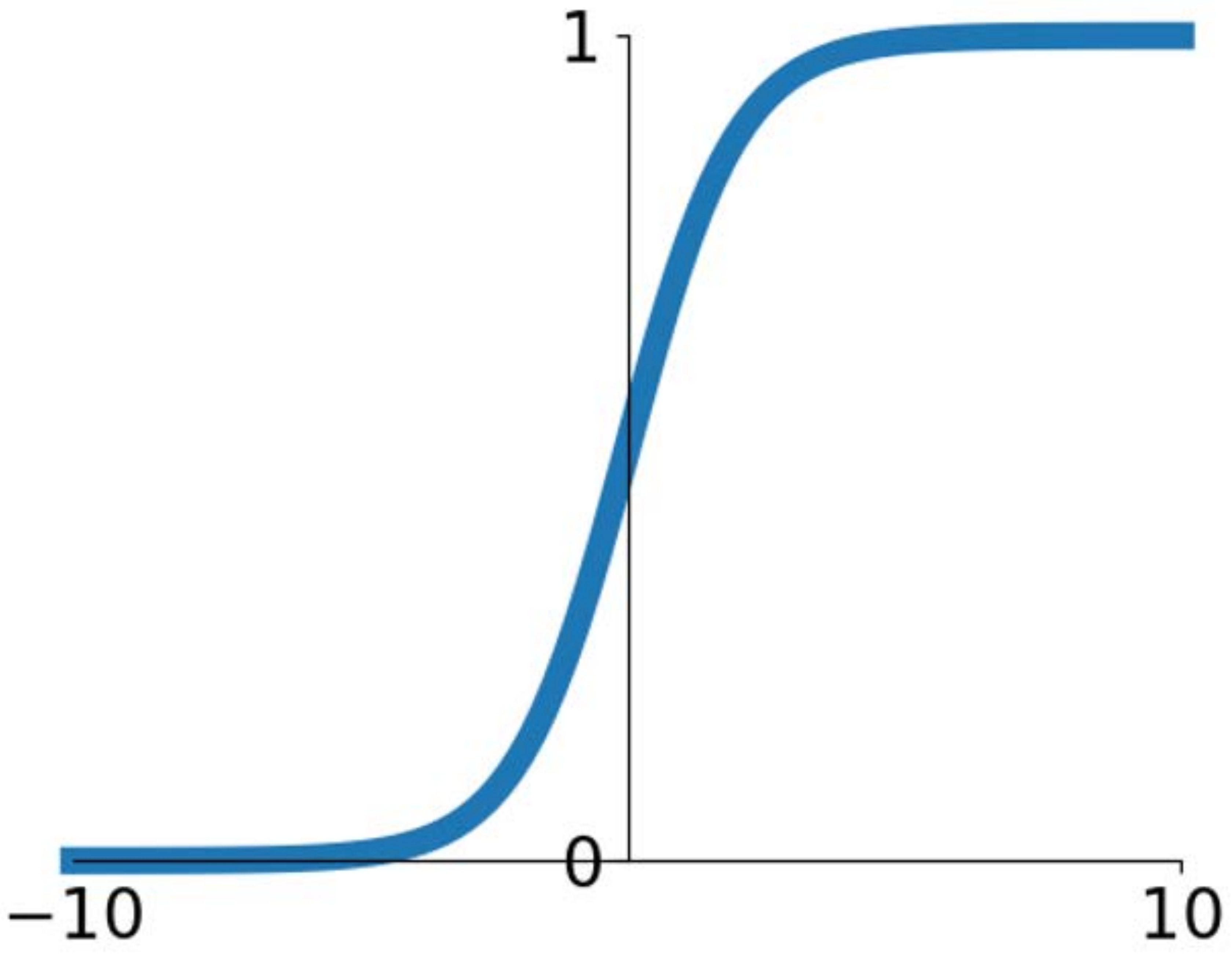


Sigmoid

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- Squashes numbers to range [0, 1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

Activation Functions: Sigmoid



Sigmoid

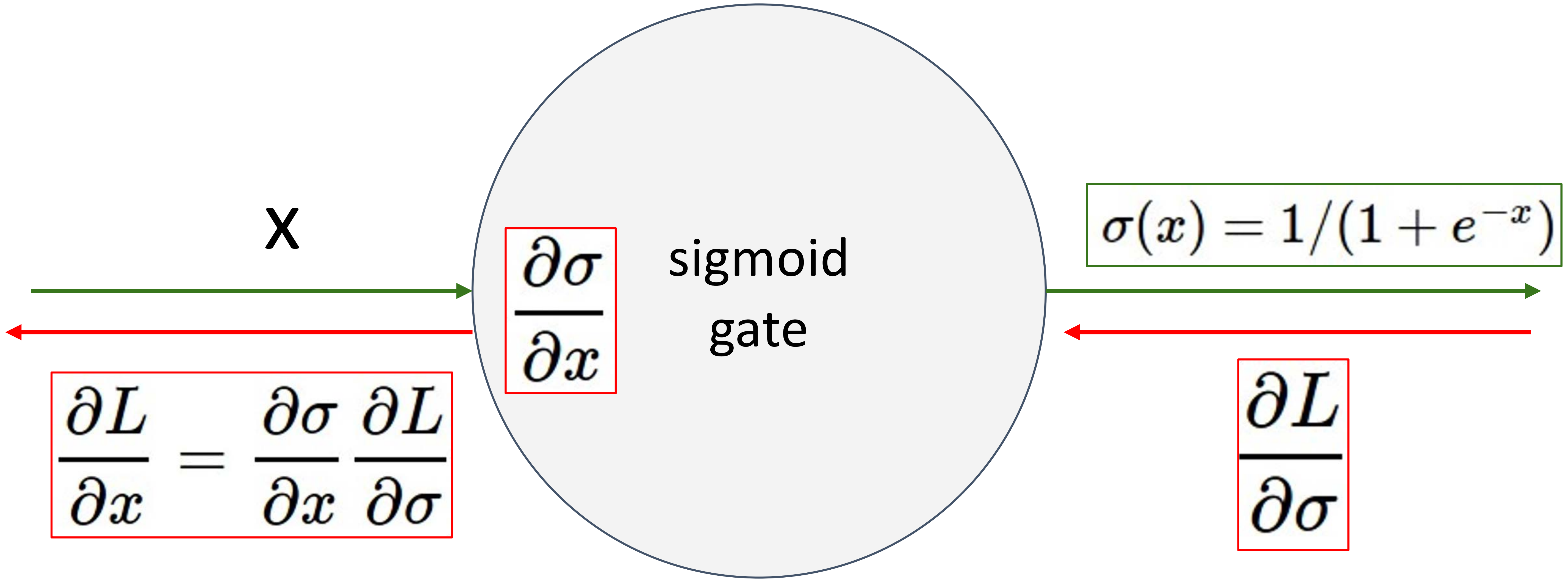
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

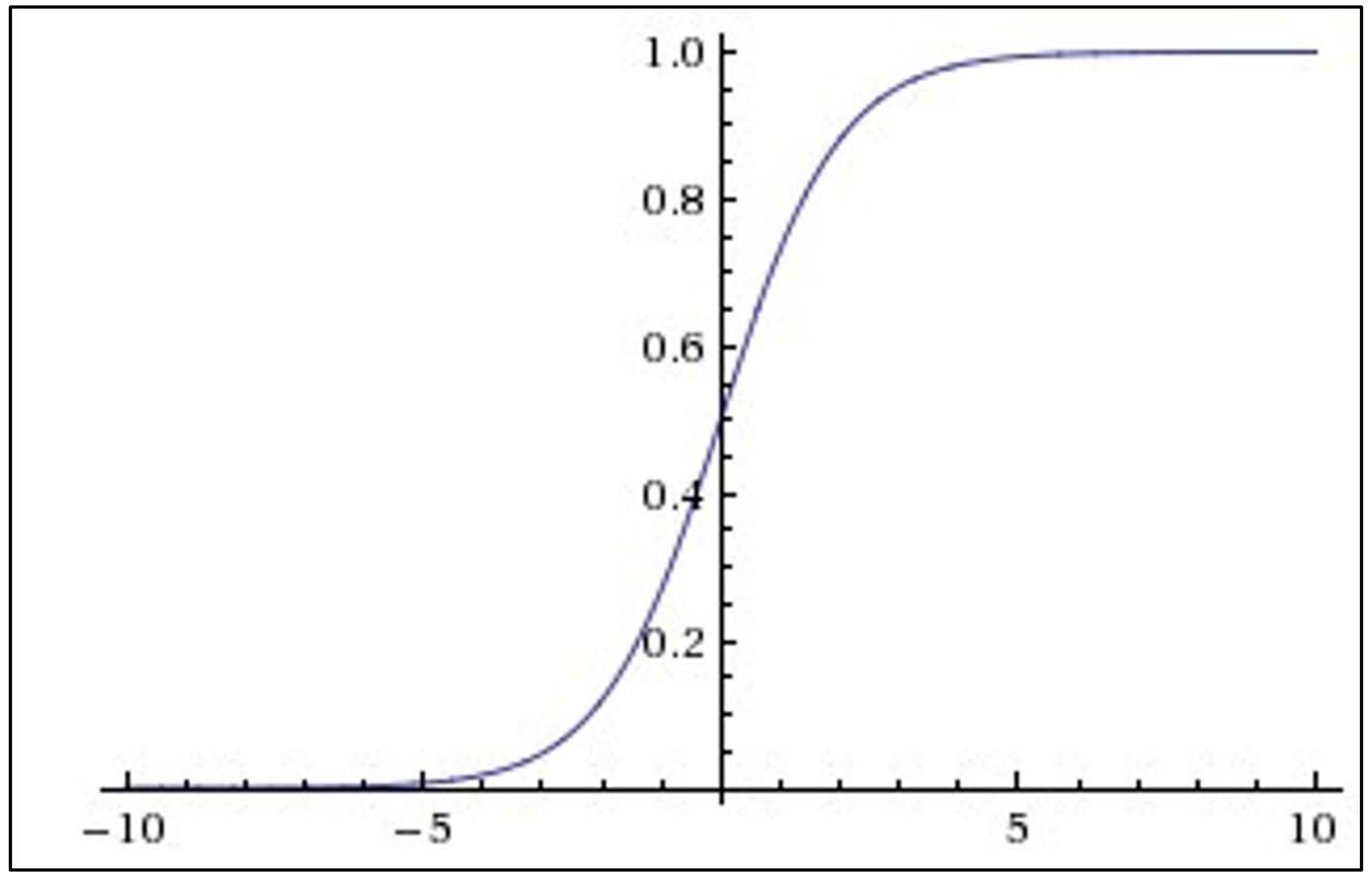
- Squashes numbers to range [0, 1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron

3 problems:

1. Saturated neurons "kill" the gradients

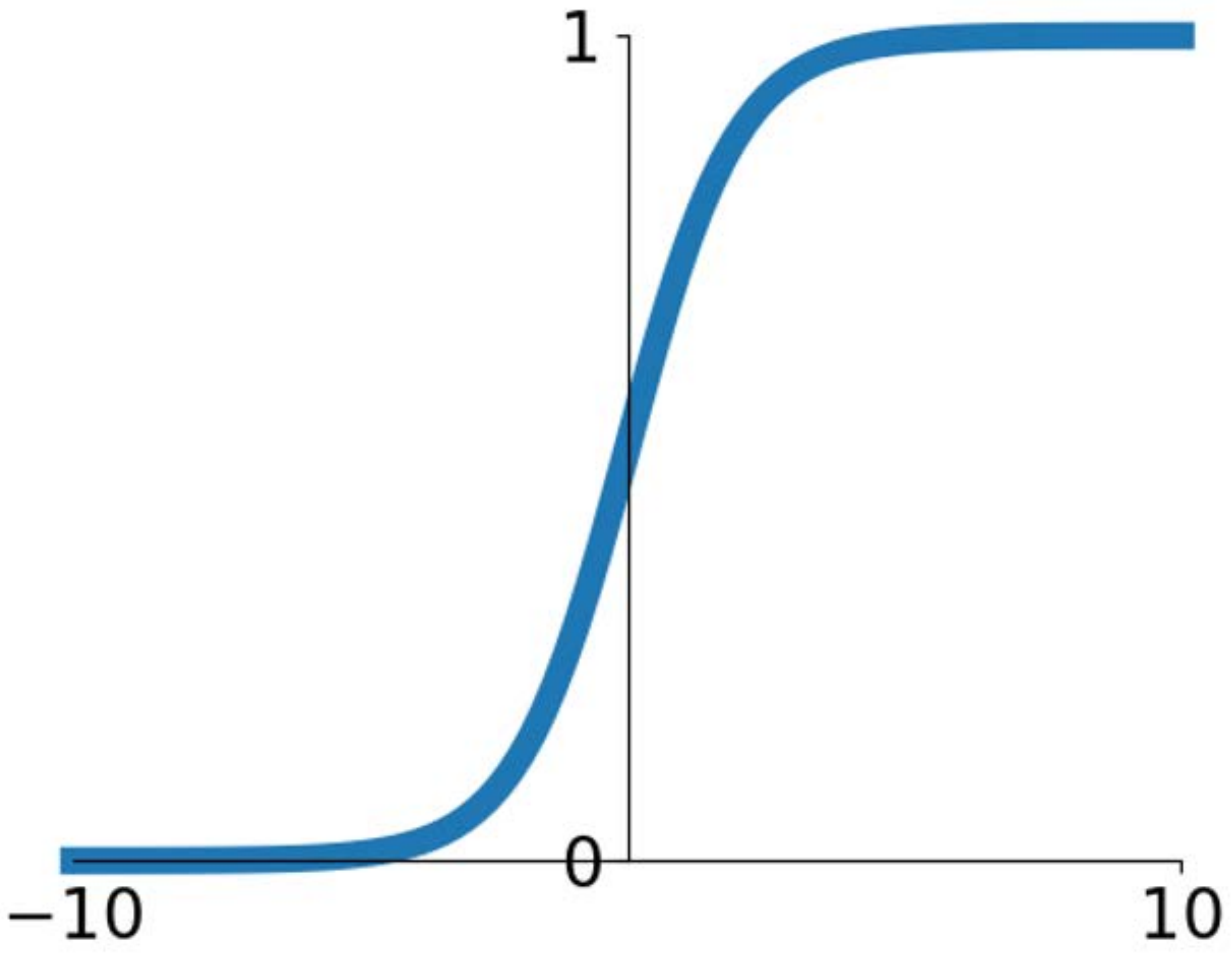
Activation Functions: Sigmoid





- What happens when x = -10?
- What happens when x = 0?
- What happens when x = 10?

Activation Functions: Sigmoid



Sigmoid

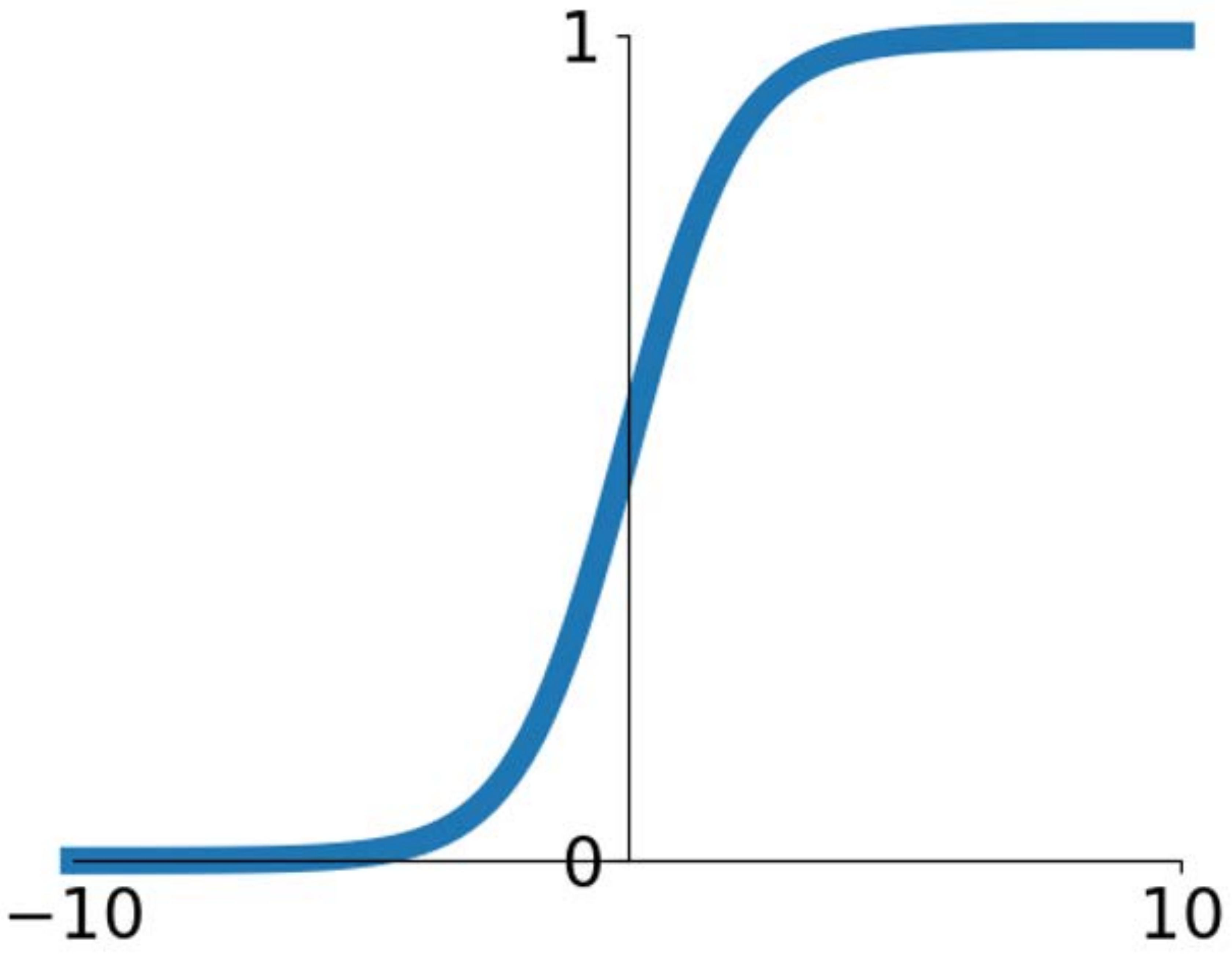
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- Squashes numbers to range [0, 1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron

3 problems:

1. Saturated neurons "kill" the gradients

Activation Functions: Sigmoid



Sigmoid

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- Squashes numbers to range [0, 1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron

3 problems:

1. Saturated neurons "kill" the gradients
2. Sigmoid outputs are not zero-centered

Activation Functions: Sigmoid

Consider what happens when nonlinearity is always positive

$$h_i^{(\ell)} = \sum_j w_{i,j}^{(\ell)} \sigma(h_j^{\ell-1}) + b_i^{(\ell)}$$

$h_i^{(\ell)}$ is the ith element of the hidden layer at layer ℓ (before activation)

$w^{(\ell)}, b^{(\ell)}$ are the weights and bias of layer ℓ

What can we say about the gradients on $w^{(\ell)}$?

Activation Functions: Sigmoid

Consider what happens when nonlinearity is always positive

$$h_i^{(\ell)} = \sum_j w_{i,j}^{(\ell)} \sigma(h_j^{\ell-1}) + b_i^{(\ell)}$$

Local gradient · Upstream gradient

$$\frac{\partial L}{\partial w_{i,j}^{(\ell)}} = \frac{\partial h_i^{(\ell)}}{\partial w_{i,j}^{(\ell)}} \cdot \frac{\partial L}{\partial h_i^{(\ell)}}$$

$h_i^{(\ell)}$ is the ith element of the hidden layer at layer ℓ (before activation)

$w^{(\ell)}, b^{(\ell)}$ are the weights and bias of layer ℓ

What can we say about the gradients on $w^{(\ell)}$?

Activation Functions: Sigmoid

Consider what happens when nonlinearity is always positive

$$h_i^{(\ell)} = \sum_j w_{i,j}^{(\ell)} \sigma(h_j^{\ell-1}) + b_i^{(\ell)}$$

$h_i^{(\ell)}$ is the ith element of the hidden layer at layer ℓ (before activation)

$w^{(\ell)}, b^{(\ell)}$ are the weights and bias of layer ℓ

What can we say about the gradients on $w^{(\ell)}$?

Gradients on all $w_{i,j}^{(\ell)}$ have the same sign as upstream gradient $\partial L / \partial h_i^{(\ell)}$

Local gradient · Upstream gradient

$$\frac{\partial L}{\partial w_{i,j}^{(\ell)}} = \frac{\partial h_i^{(\ell)}}{\partial w_{i,j}^{(\ell)}} \cdot \frac{\partial L}{\partial h_i^{(\ell)}}$$

$$= \sigma(h_j^{(\ell-1)}) \cdot \frac{\partial L}{\partial h_i^{(\ell)}}$$

Activation Functions: Sigmoid

Consider what happens when nonlinearity is always positive

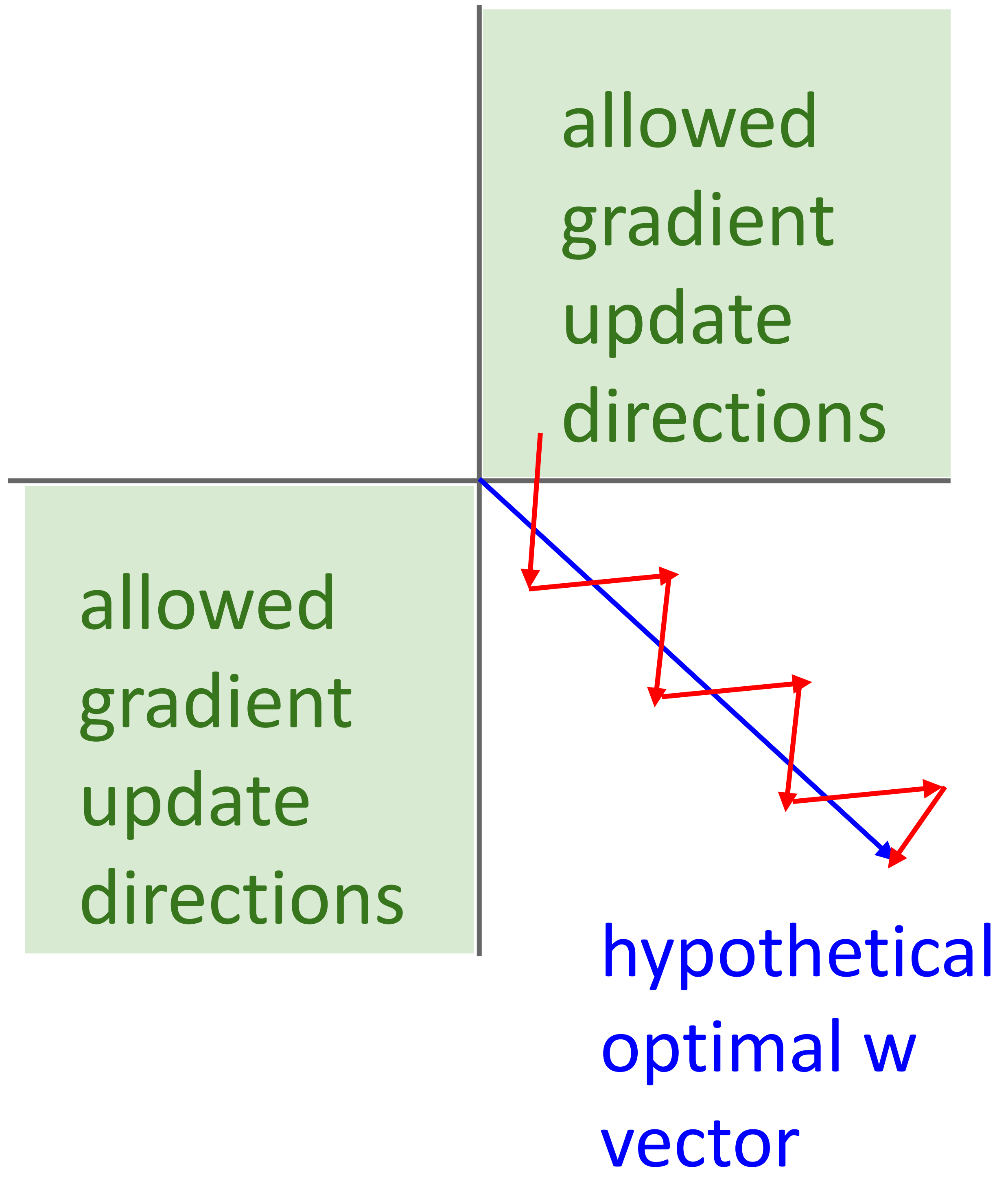
$$h_i^{(\ell)} = \sum_j w_{i,j}^{(\ell)} \sigma(h_j^{\ell-1}) + b_i^{(\ell)}$$

$h_i^{(\ell)}$ is the ith element of the hidden layer at layer ℓ (before activation)

$w^{(\ell)}, b^{(\ell)}$ are the weights and bias of layer ℓ

What can we say about the gradients on $w^{(\ell)}$?

Gradients on all $w_{i,j}^{(\ell)}$ have the same sign as upstream gradient $\partial L / \partial h_i^{(\ell)}$



Gradients on rows of w can only point in some directions; needs to "zigzag" to move in other directions

Activation Functions: Sigmoid

Consider what happens when nonlinearity is always positive

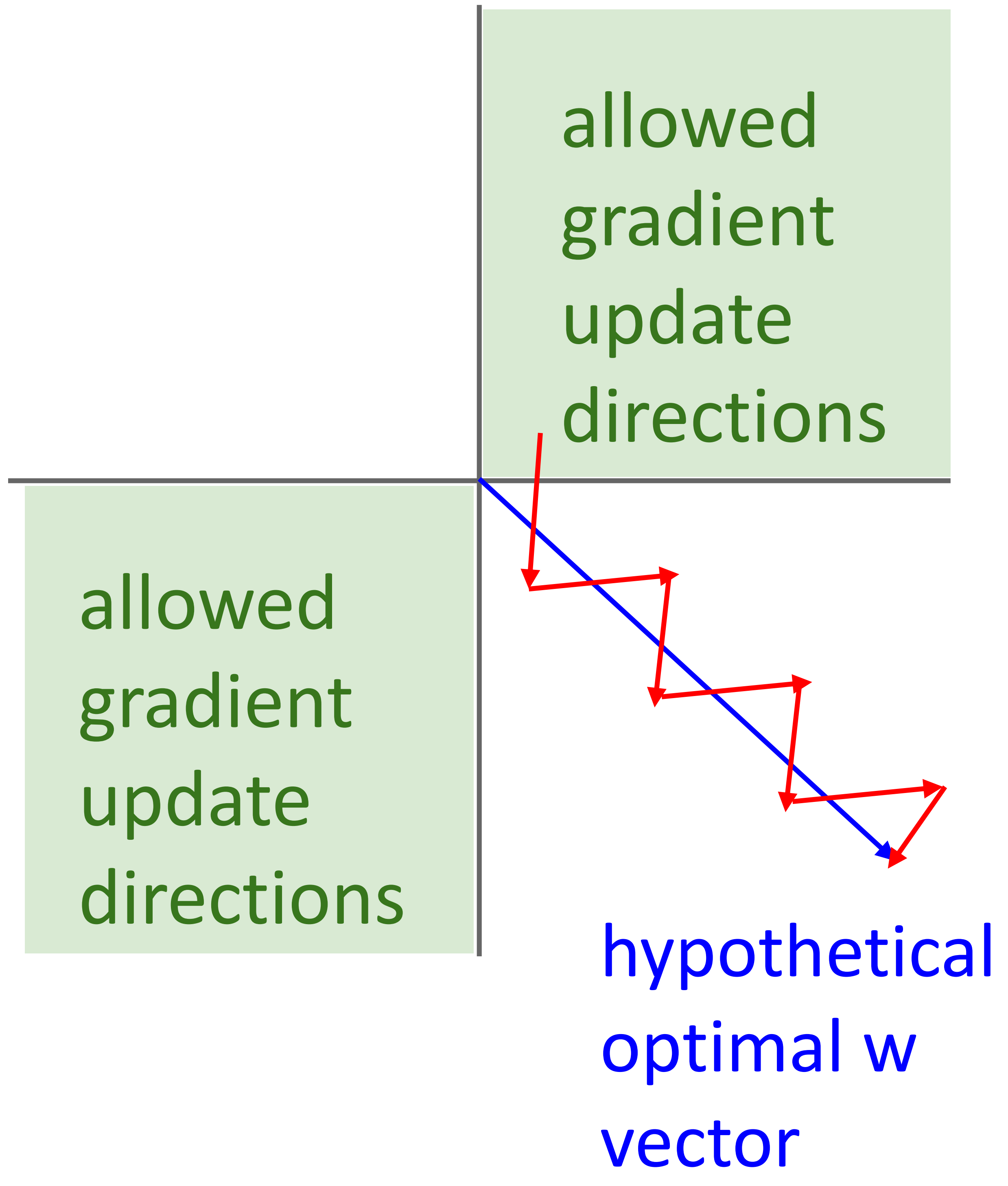
$$h_i^{(\ell)} = \sum_j w_{i,j}^{(\ell)} \sigma(h_j^{\ell-1}) + b_i^{(\ell)}$$

$h_i^{(\ell)}$ is the ith element of the hidden layer at layer ℓ (before activation)

$w^{(\ell)}, b^{(\ell)}$ are the weights and bias of layer ℓ

What can we say about the gradients on $w^{(\ell)}$?

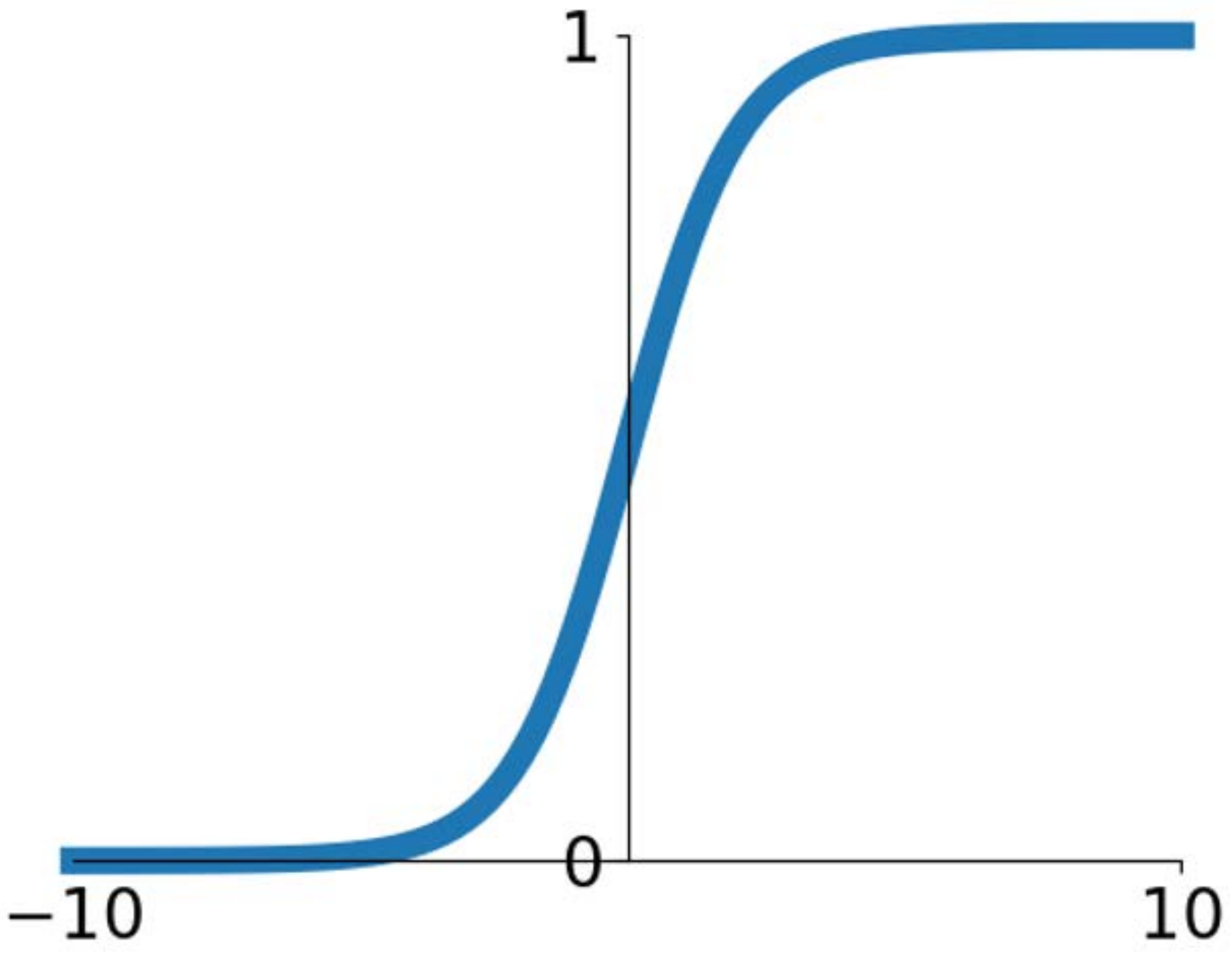
Gradients on all $w_{i,j}^{(\ell)}$ have the same sign as upstream gradient $\partial L / \partial h_i^{(\ell)}$



Not that bad in practice:

- Only true for a single example, mini batches help
- BatchNorm can also avoid this

Activation Functions: Sigmoid



Sigmoid

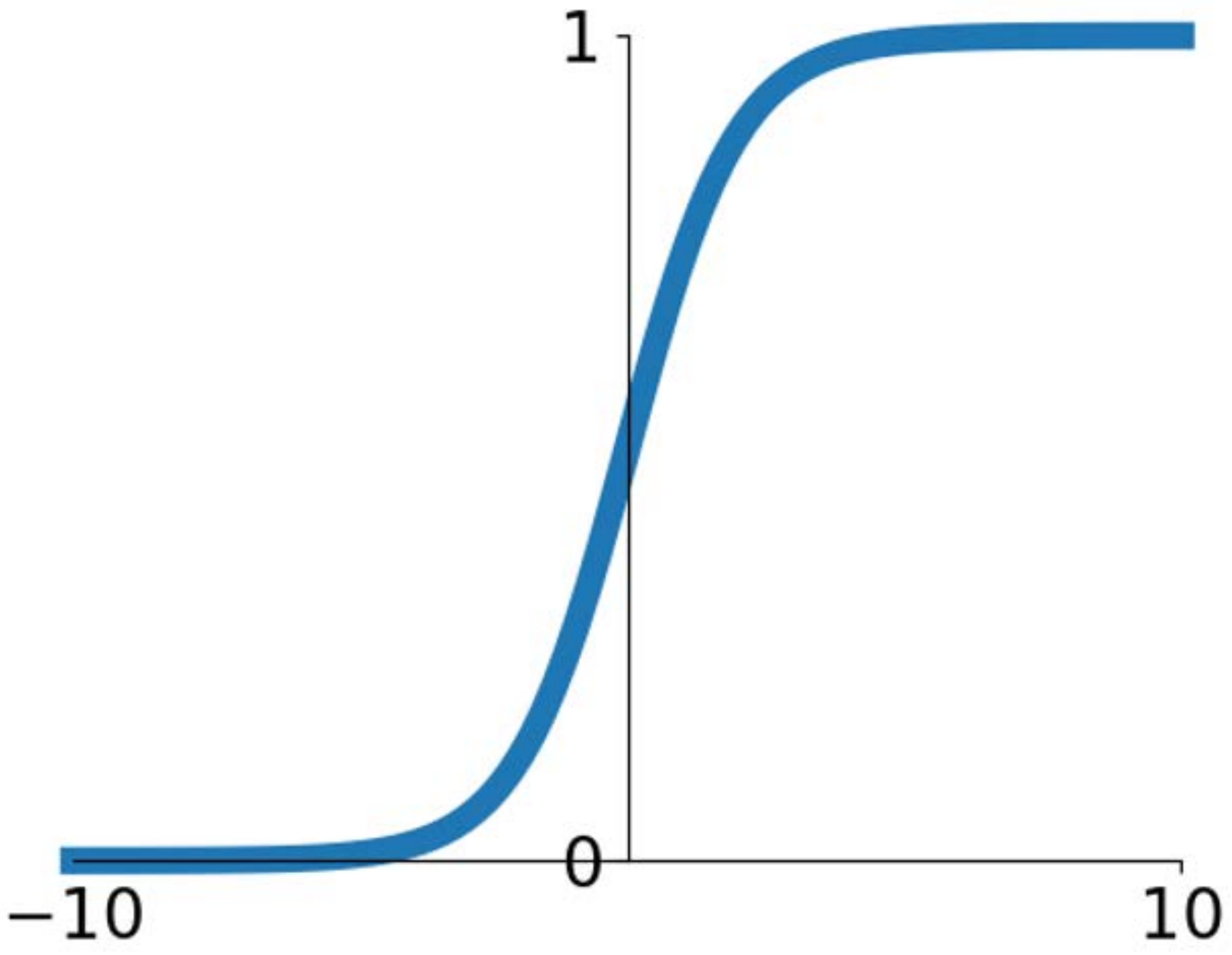
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- Squashes numbers to range [0, 1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron

3 problems:

1. Saturated neurons "kill" the gradients
2. Sigmoid outputs are not zero-centered

Activation Functions: Sigmoid



Sigmoid

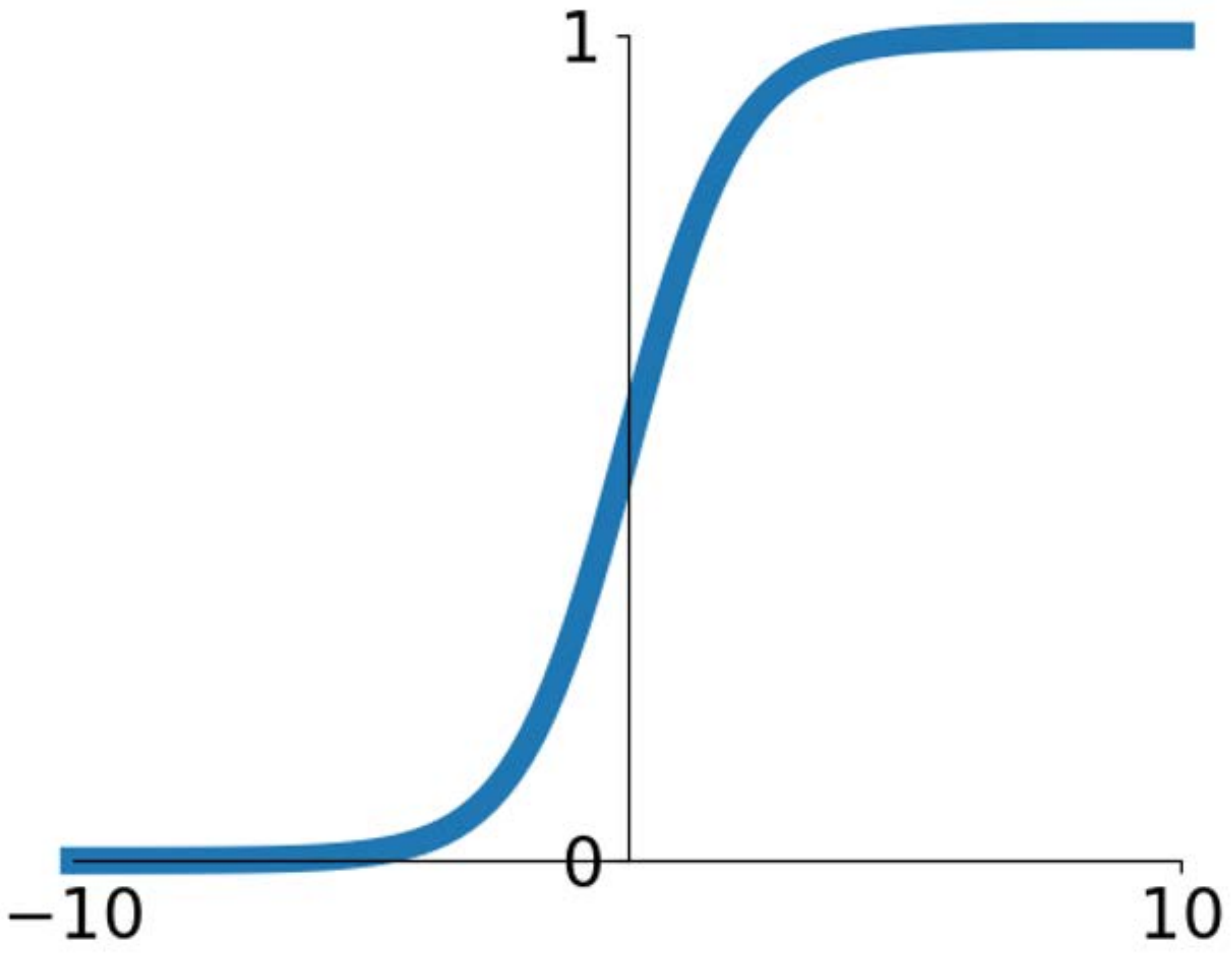
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- Squashes numbers to range [0, 1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron

3 problems:

1. Saturated neurons "kill" the gradients
2. Sigmoid outputs are not zero-centered
3. exp() is a bit compute expensive

Activation Functions: Sigmoid



Sigmoid

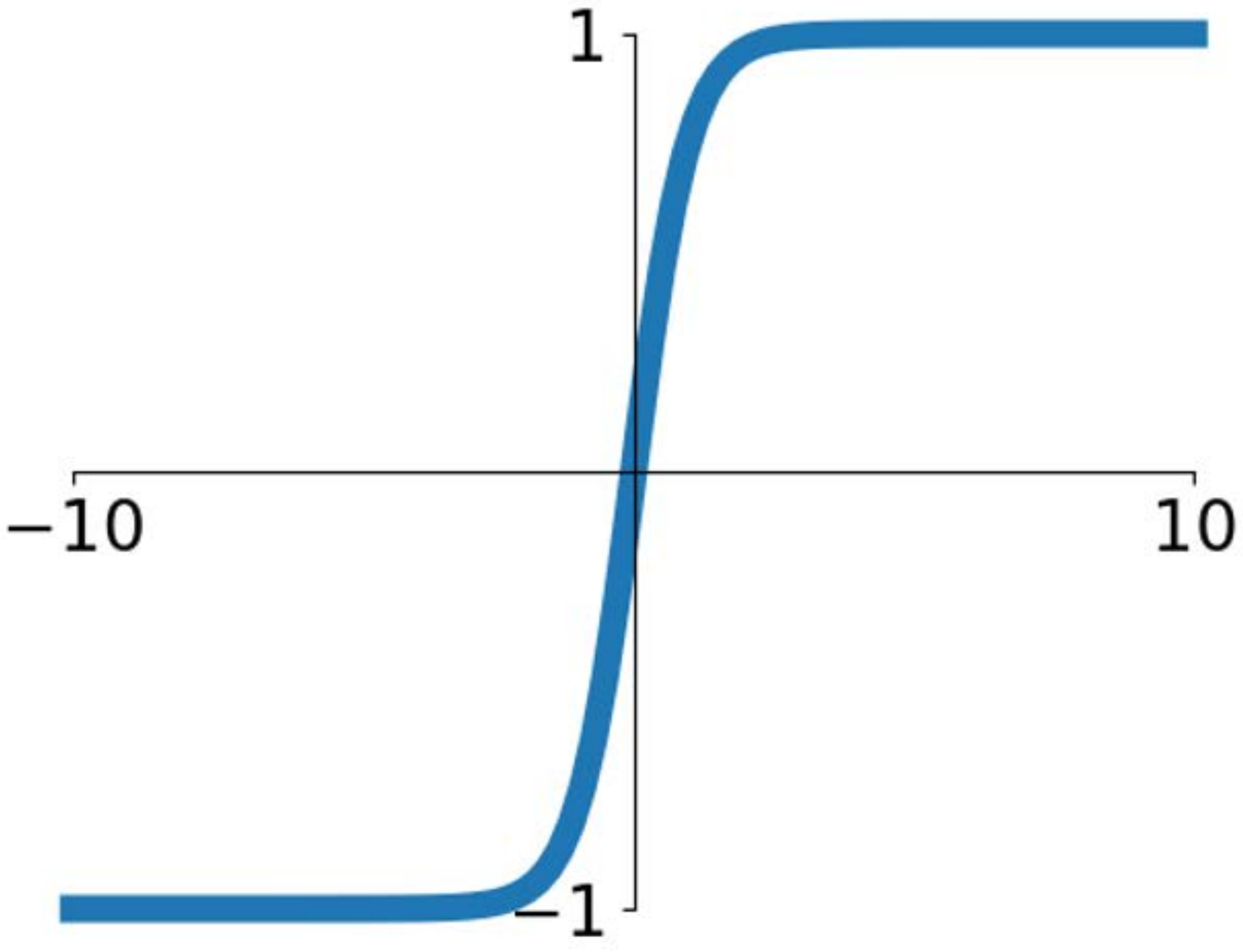
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- Squashes numbers to range [0, 1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

3 problems: **Worst problem in practice**

1. **Saturated neurons “kill” the gradients**
2. Sigmoid outputs are not zero-centered
3. exp() is a bit compute expensive

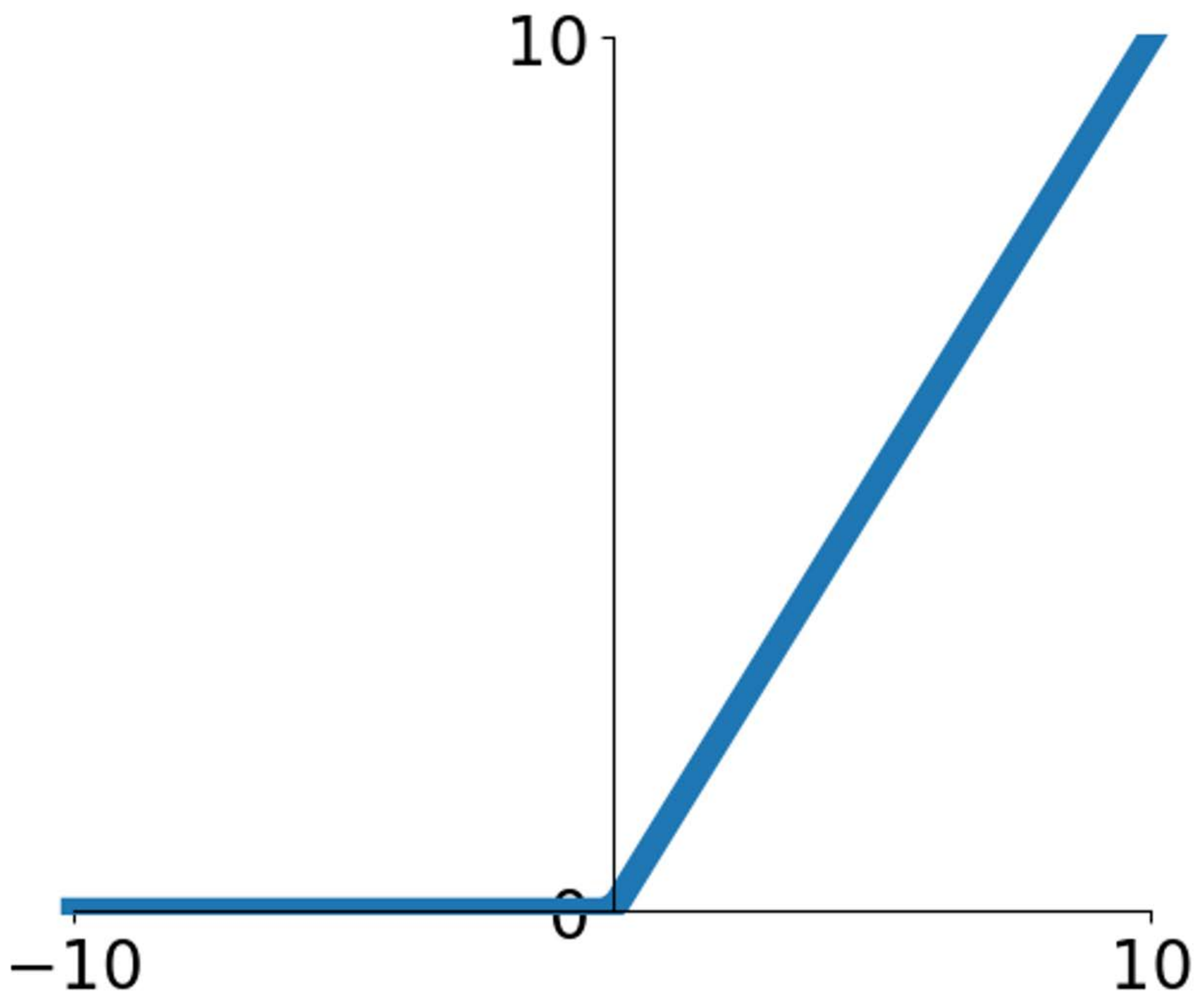
Activation Functions: tanh



tanh(x)

- Squashes numbers to range [-1, 1]
- Zero centered (nice)
- Still kills gradients when saturated :(

Activation Functions: ReLU

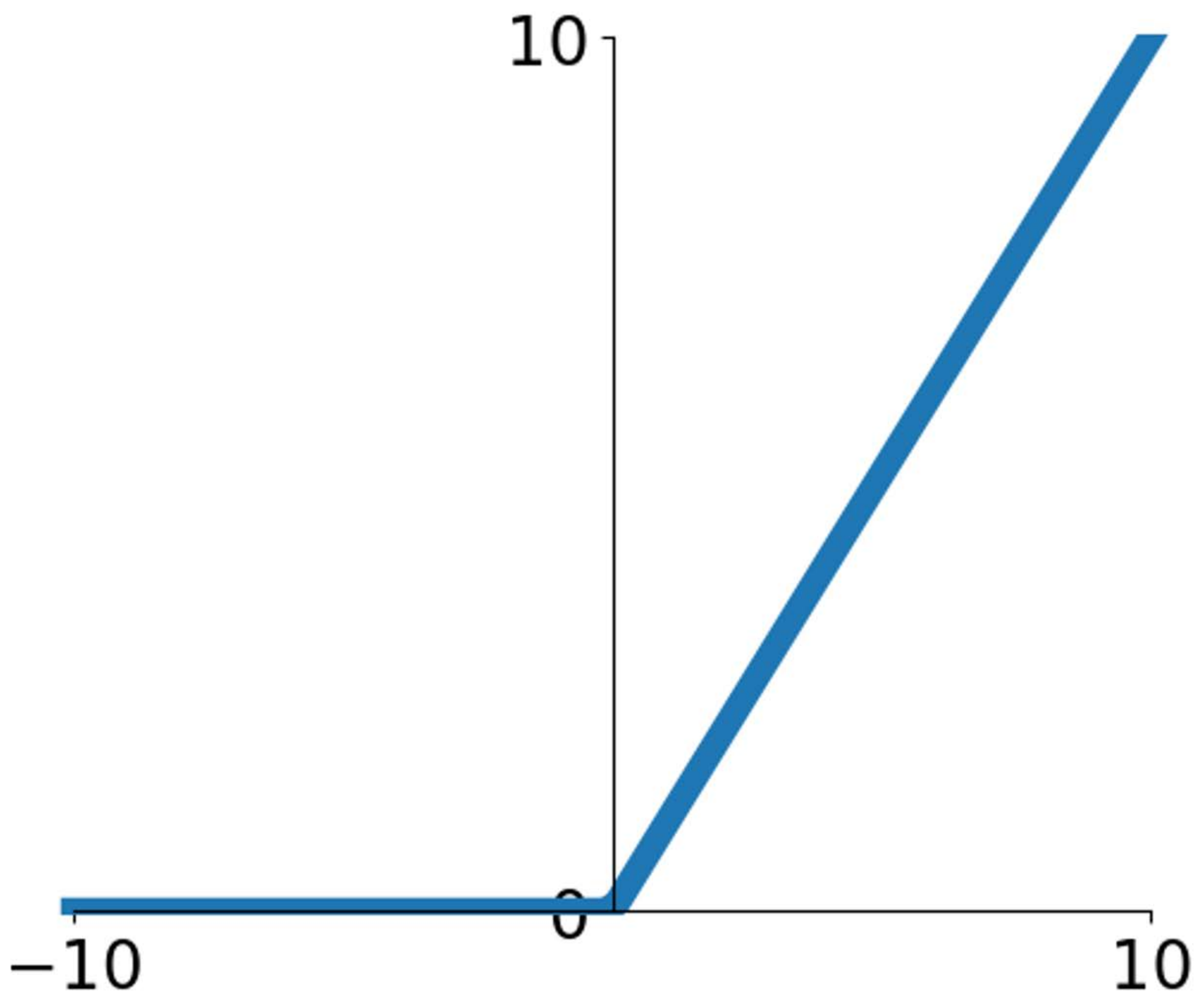


ReLU
(Rectified Linear Unit)

$$f(x) = \max(0, x)$$

- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid and tanh in practice (e.g. 6x)

Activation Functions: ReLU



ReLU
(Rectified Linear Unit)

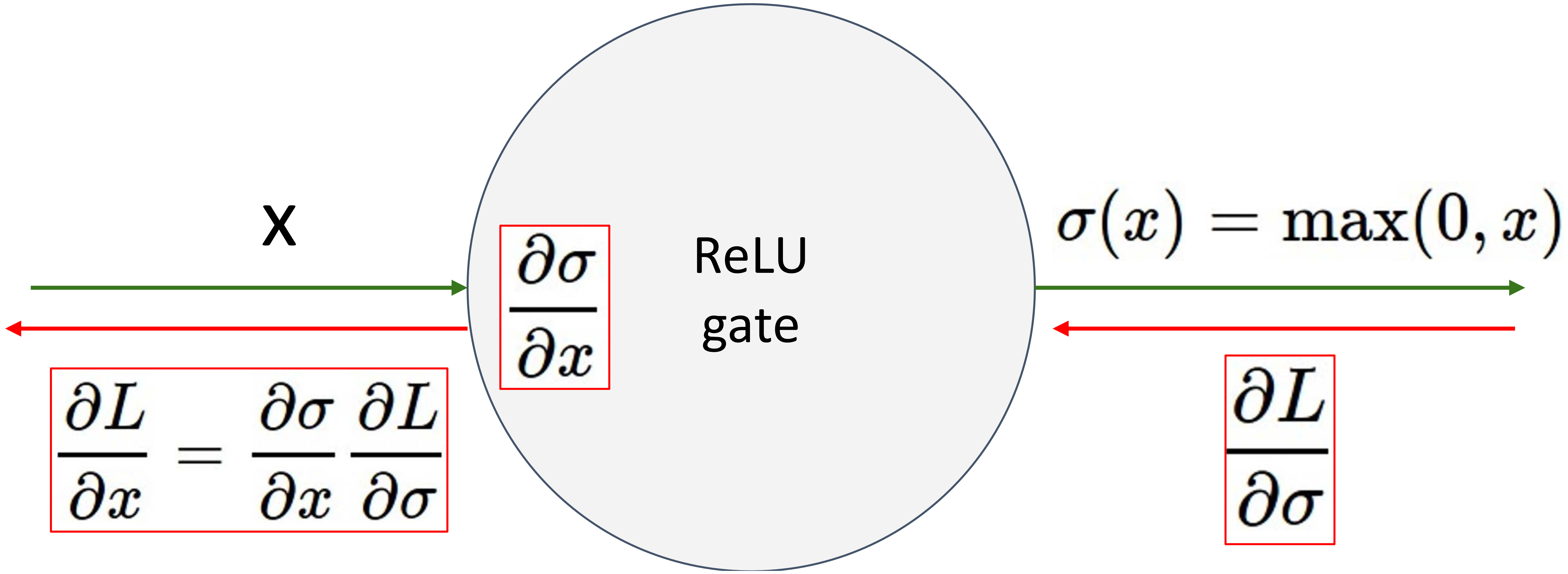
$$f(x) = \max(0,x)$$

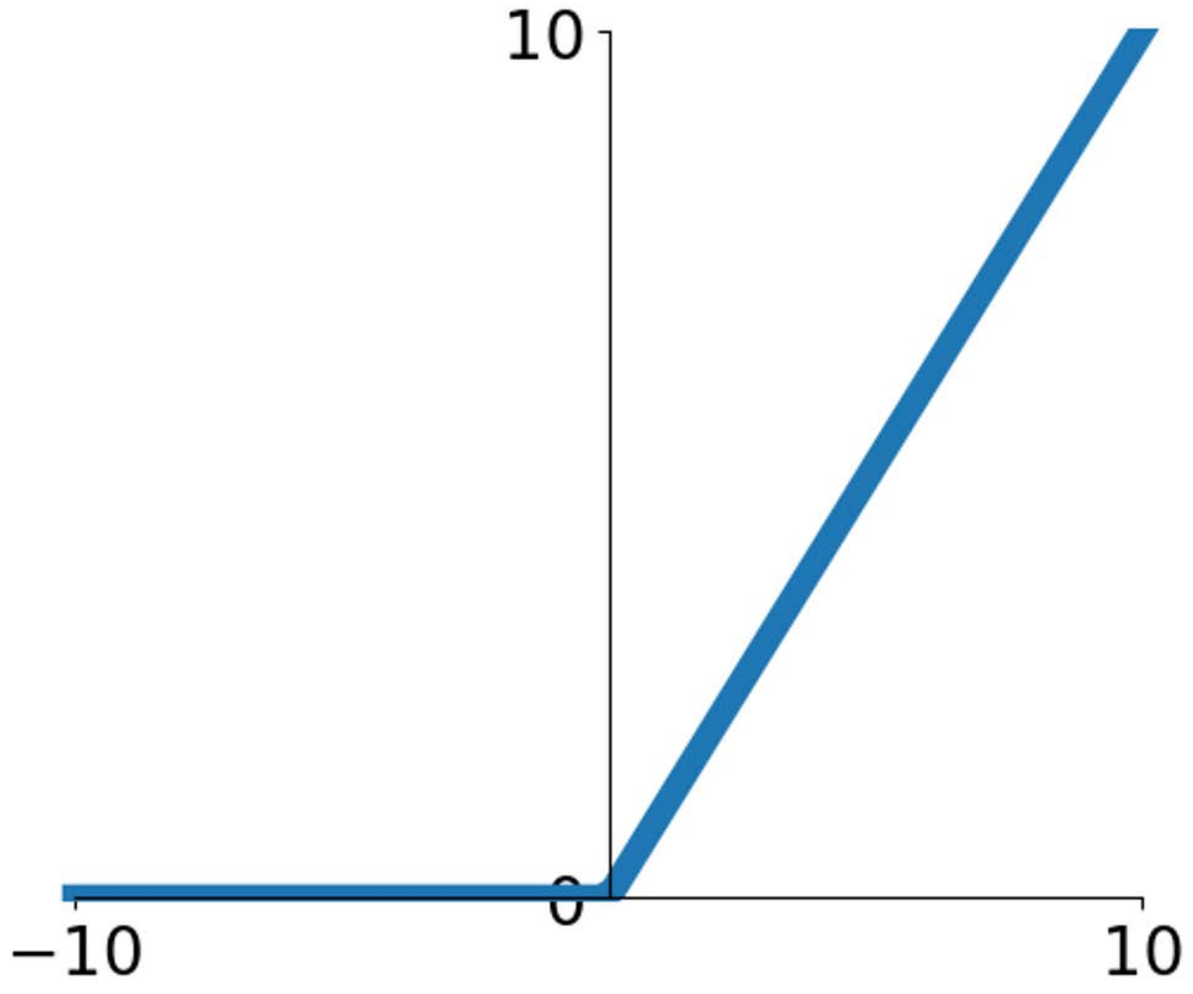
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid and tanh in practice (e.g. 6x)

- Not zero-centered output
- An annoyance:

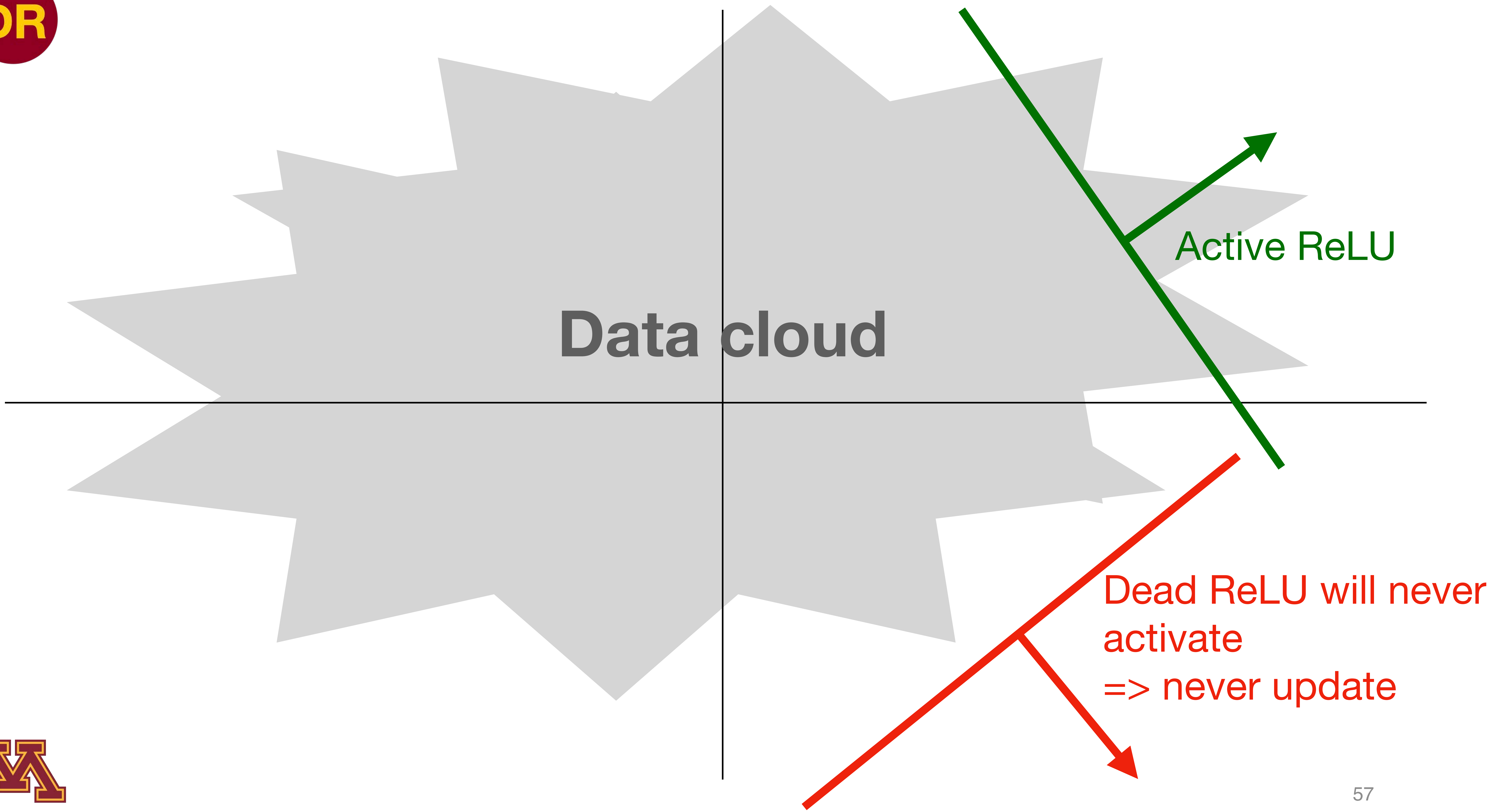
Hint: what is the gradient when x<0?

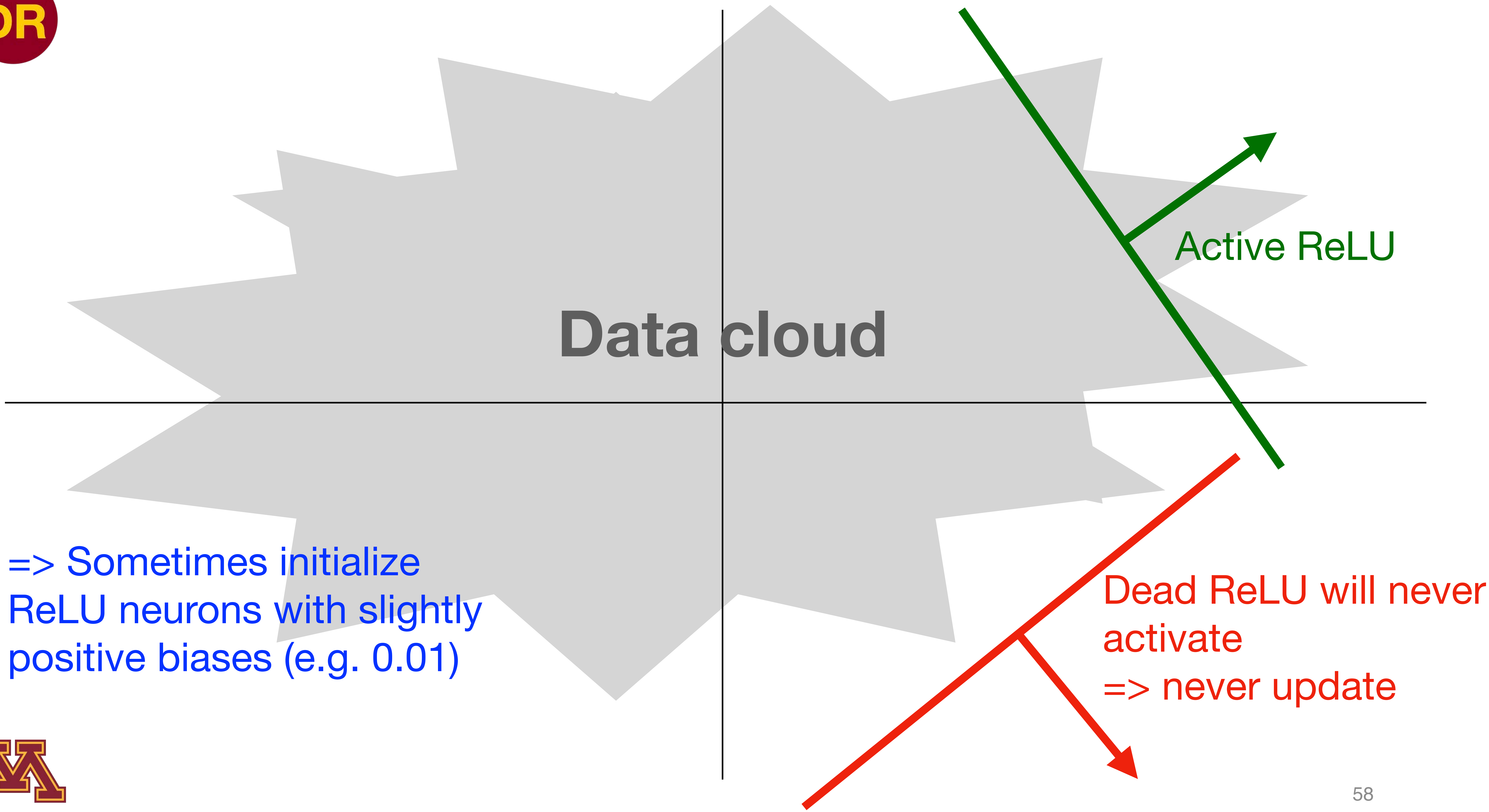
Activation Functions: ReLU





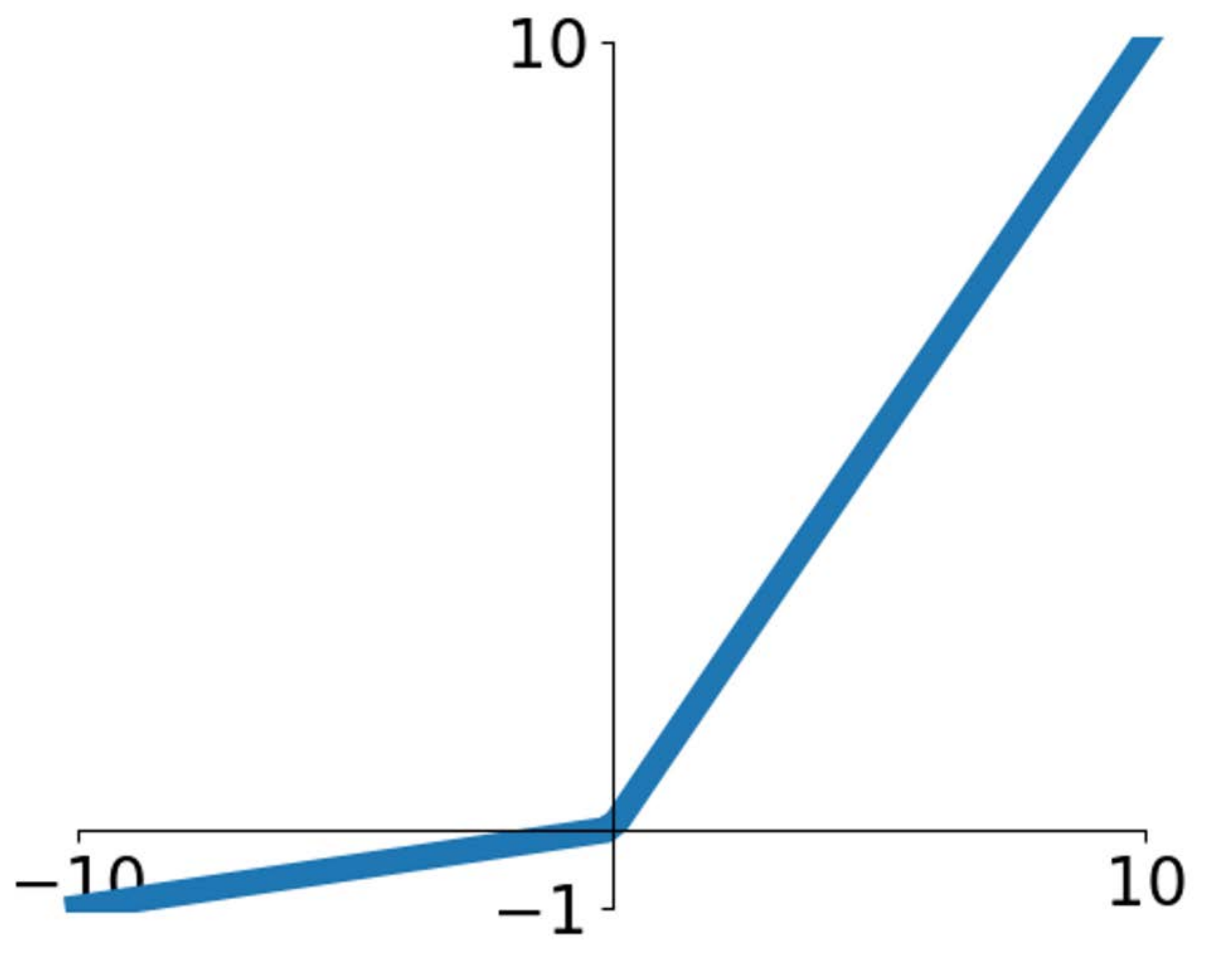
- What happens when x = -10?
- What happens when x = 0?
- What happens when x = 10?


Data cloud
Active ReLU
Dead ReLU will never activate
=> never update



=> Sometimes initialize
ReLU neurons with slightly
positive biases (e.g. 0.01)

Activation Functions: Leaky ReLU



Leaky ReLU

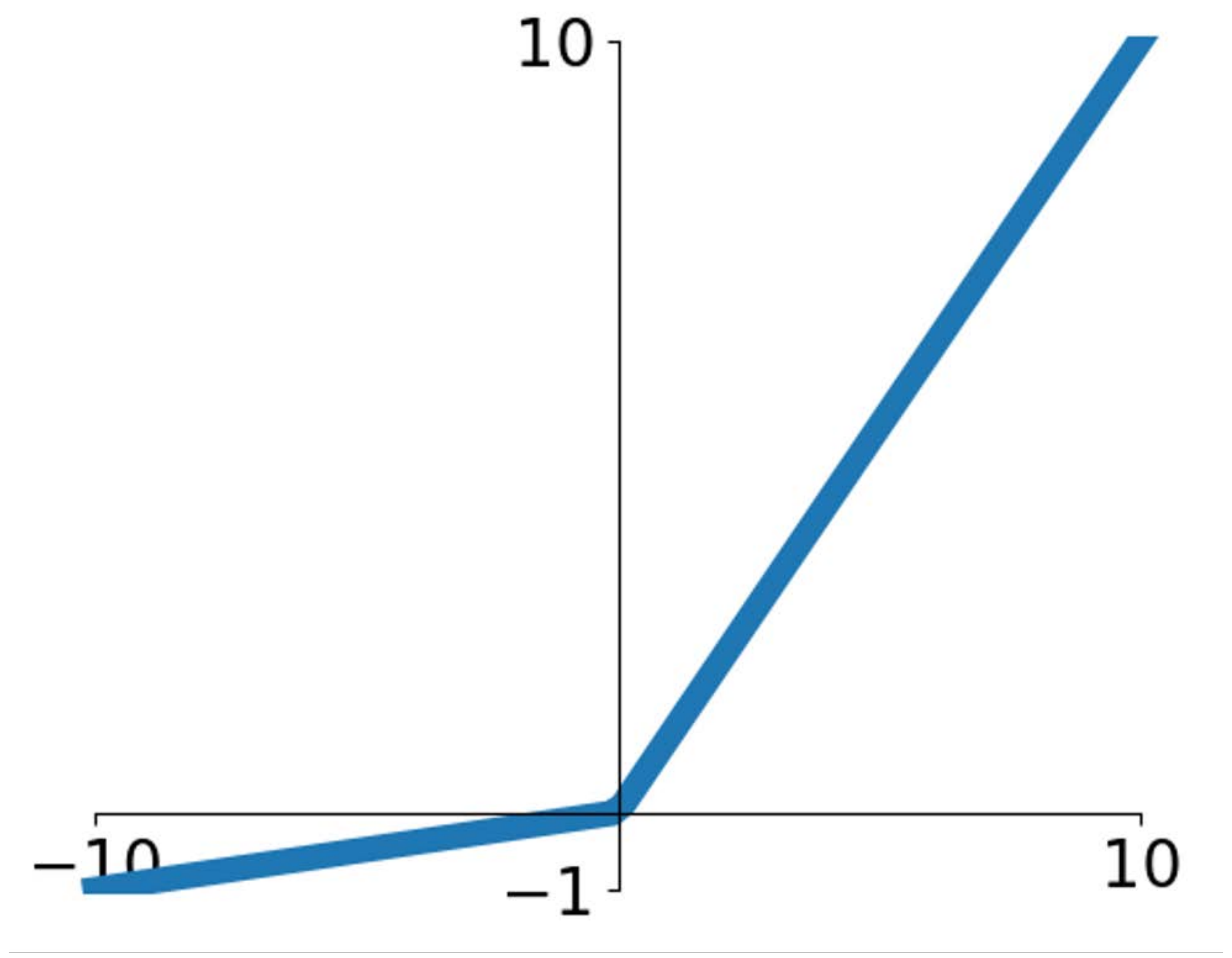
$$f(x) = \max(\alpha x, x)$$

α is a hyperparameter, often $\alpha = 0.1$

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid and tanh in practice (e.g. 6x)
- **Will not "die"**

Maas et al, "Rectifier Nonlinearities Improve Neural Network Acoustic Models", ICML 2013

Activation Functions: Leaky ReLU



- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid and tanh in practice (e.g. 6x)
- **Will not "die"**

Leaky ReLU

$f(x) = \max(\alpha x, x)$

α is a hyperparameter, often $\alpha = 0.1$

Maas et al, "Rectifier Nonlinearities Improve Neural Network Acoustic Models", ICML 2013

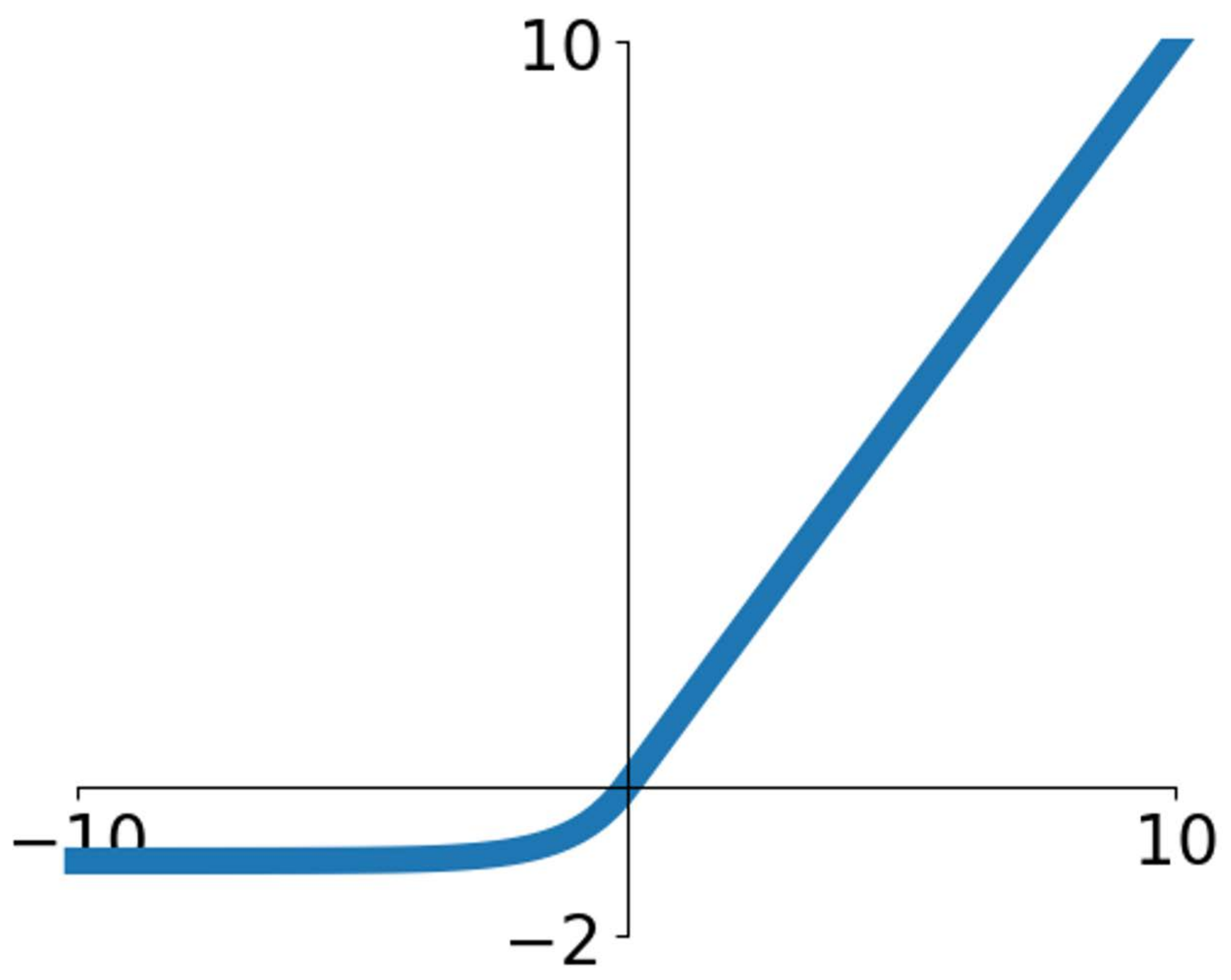
Parametric ReLU (PReLU)

$f(x) = \max(\alpha x, x)$

α is learned via backprop

He et al, "Delving Deep into Rectifiers: Surpassing Human- Level Performance on ImageNet Classification", ICCV 2015

Activation Functions: Exponential Linear Unit (ELU)

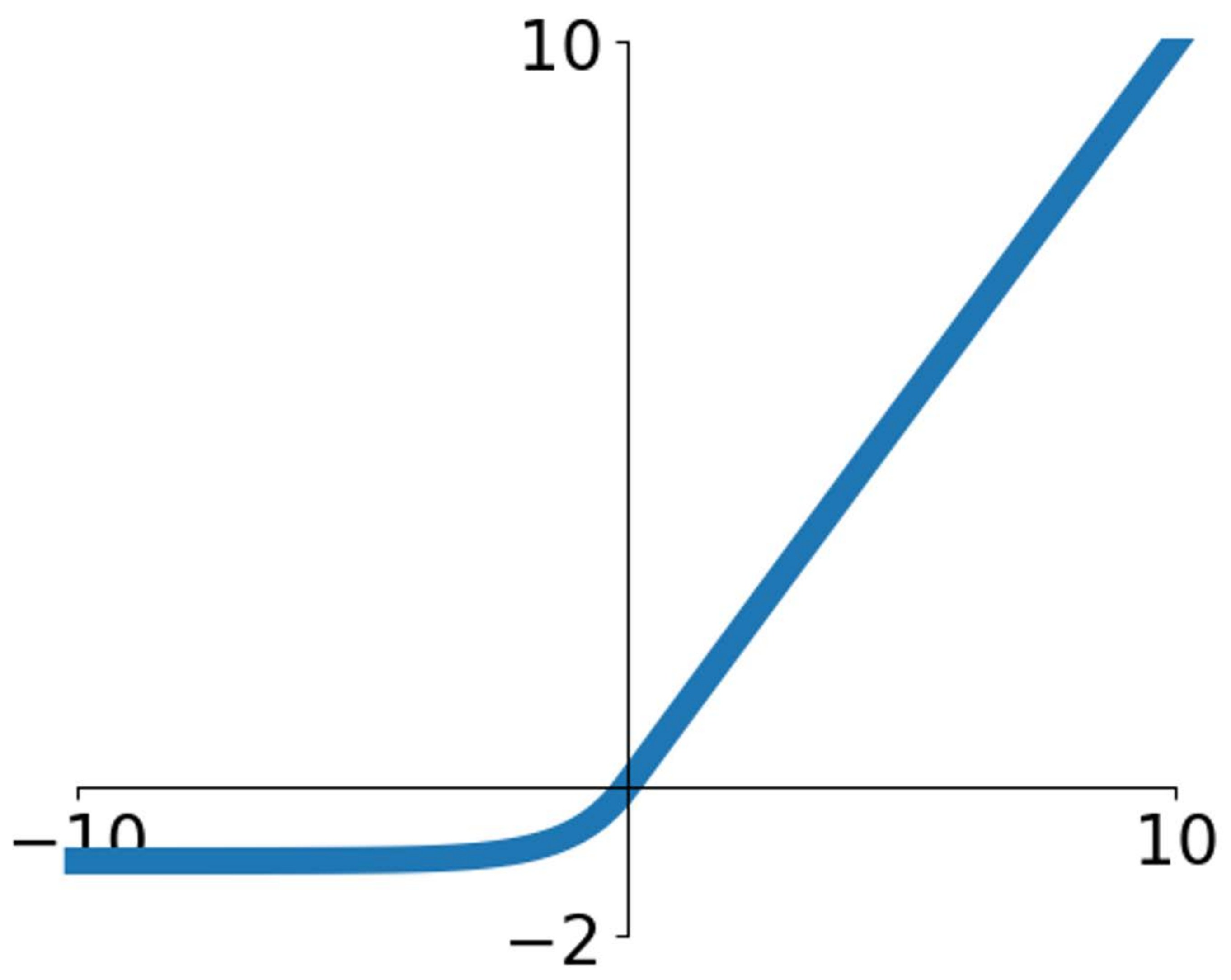


- All benefits of ReLU
- Closer to zero means outputs
- Negative saturation regime compared with Leaky ReLU adds some robustness to noise

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{if } x \leq 0 \end{cases}$$

(Default $\alpha = 1$)

Activation Functions: Exponential Linear Unit (ELU)



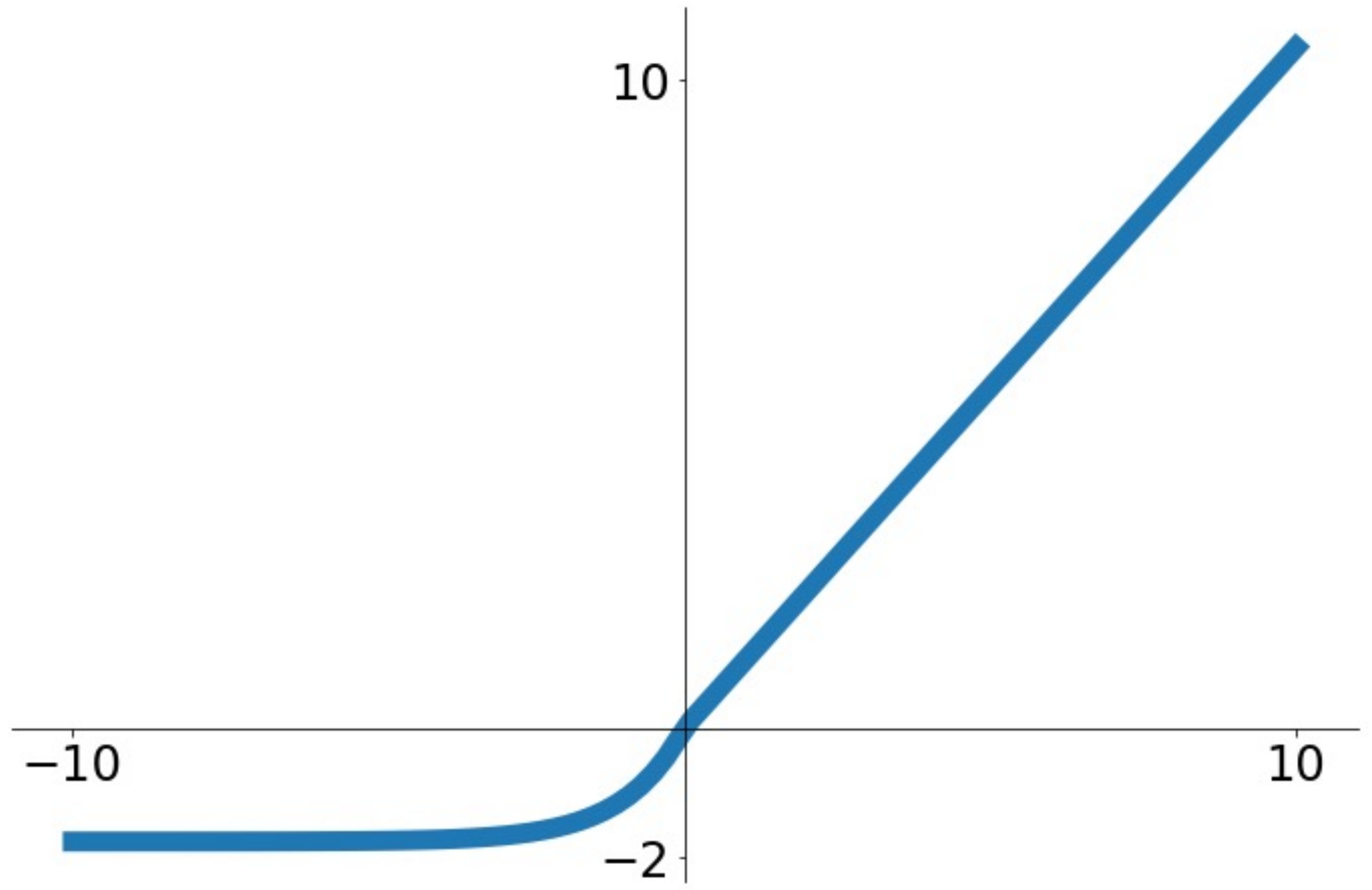
$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{if } x \leq 0 \end{cases}$$

(Default $\alpha = 1$)

- All benefits of ReLU
- Closer to zero means outputs
- Negative saturation regime compared with Leaky ReLU adds some robustness to noise

- Computation requires exp()

Activation Functions: Scale Exponential Linear Unit (SELU)



- Scaled version of ELU that works better for deep networks "Self-Normalizing" property; can train deep SELU networks without BatchNorm

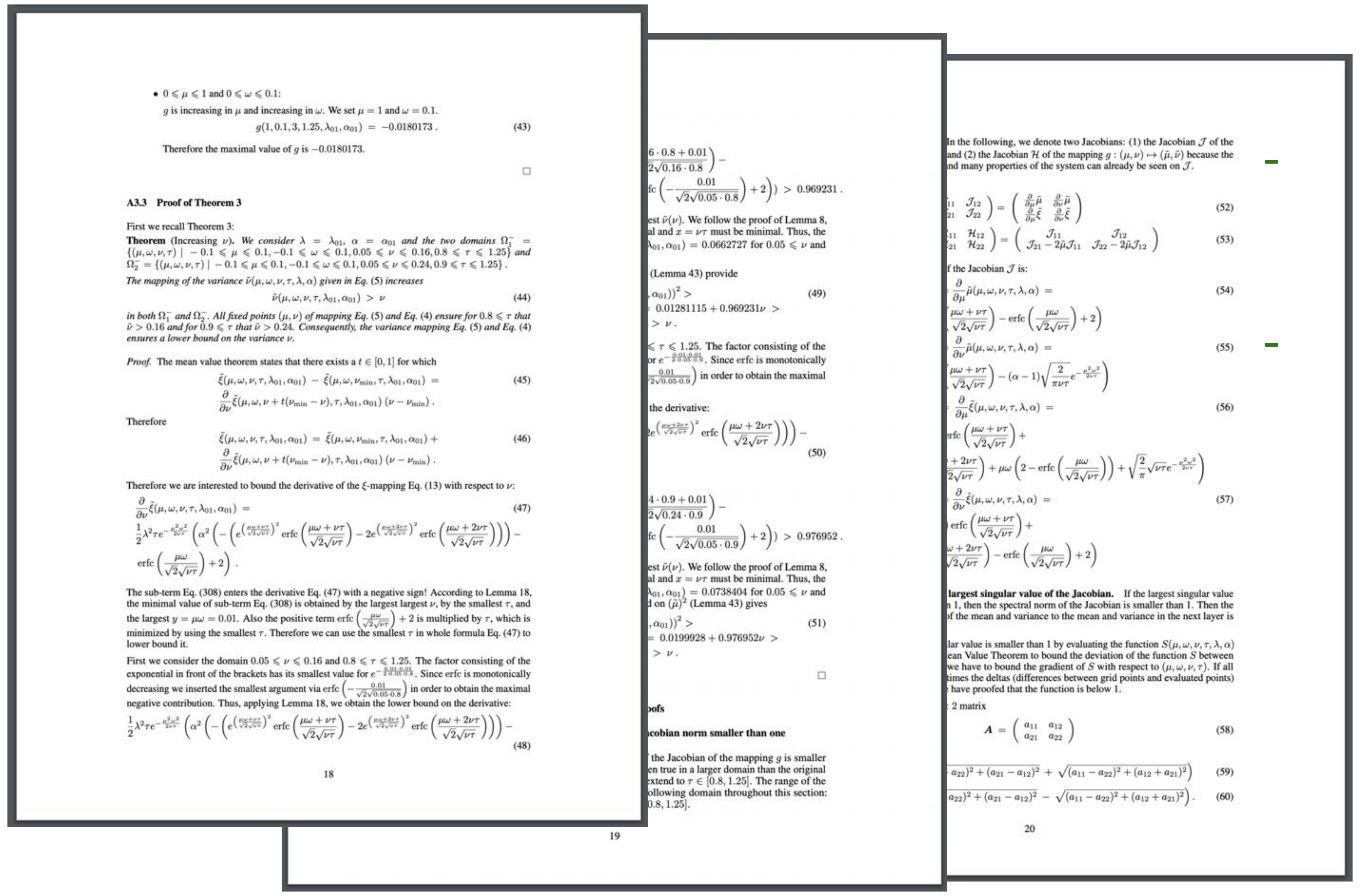
$$selu(x) = \begin{cases} \lambda x & \text{if } x > 0 \\ \lambda \alpha (e^x - 1) & \text{if } x \leq 0 \end{cases}$$

$\alpha = 1.6732632423543772848170429916717$

$\lambda = 1.0507009873554804934193349852946$

Klambauer et al, Self-Normalizing Neural Networks, ICLR 2017

Activation Functions: Scale Exponential Linear Unit (SELU)



$$selu(x) = \begin{cases} \lambda x & \text{if } x > 0 \\ \lambda\alpha(e^x - 1) & \text{if } x \le 0 \end{cases}$$

$\alpha = 1.6732632423543772848170429916717$

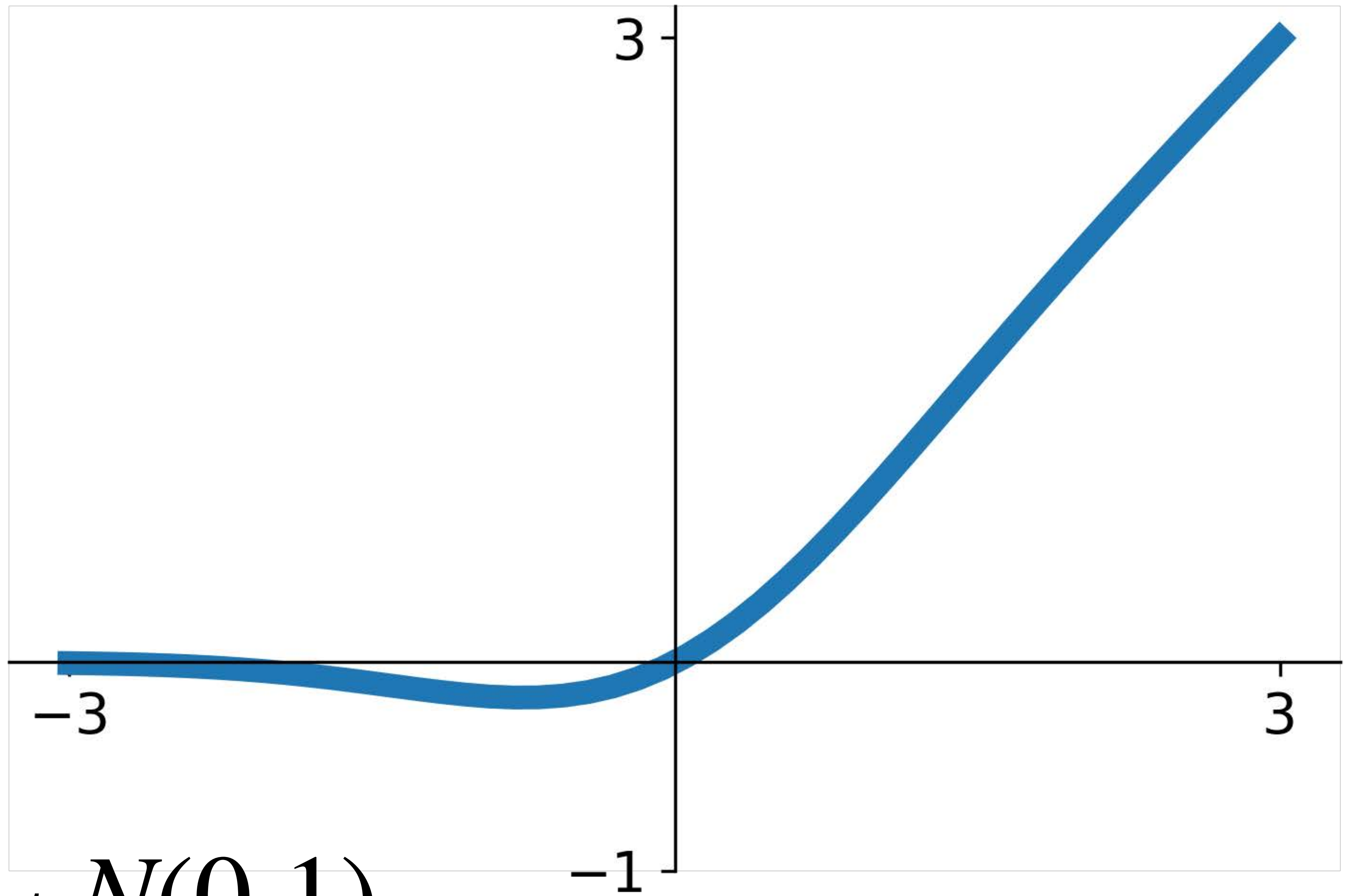
$\lambda = 1.0507009873554804934193349852946$

- Scaled version of ELU that works better for deep networks "Self-Normalizing" property; can train deep SELU networks without BatchNorm

- Derivation takes 91 pages of math in appendix…

Klambauer et al, Self-Normalizing Neural Networks, ICLR 2017

Activation Functions: Gaussian Error Linear Unit (GELU)



$$X \sim N(0,1)$$

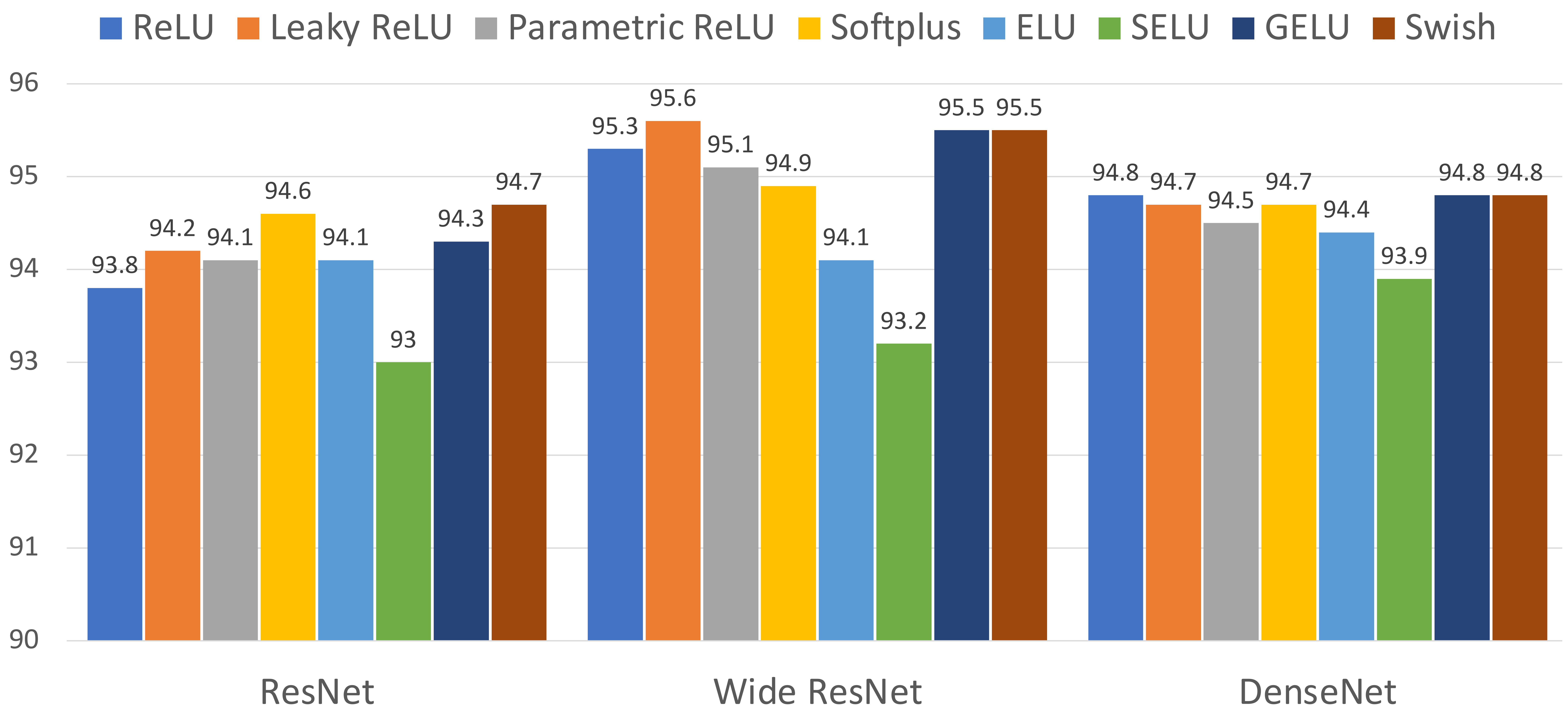
$$\text{gelu}(x) = xP(X \le x) = \frac{x}{2}(1 + \text{erf}(x/\sqrt{2}))$$

$$\approx x\sigma(1.702x)$$

- **Idea**: Multiply input by 0 or 1 at random; large values more likely to be multiplied by 1, small values more likely to be multiplied by 0 (data-dependent dropout)
- Take expectation over randomness
- Very common in Transformers (BERT, GPT, ViT)

Hendrycks and Gimpel, Gaussian Error Linear Units (GELUs), 2016

Accuracy on CIFAR10



Ramachandran et al, "Searching for activation functions", ICLR Workshop 2018

Activation Functions: Summary

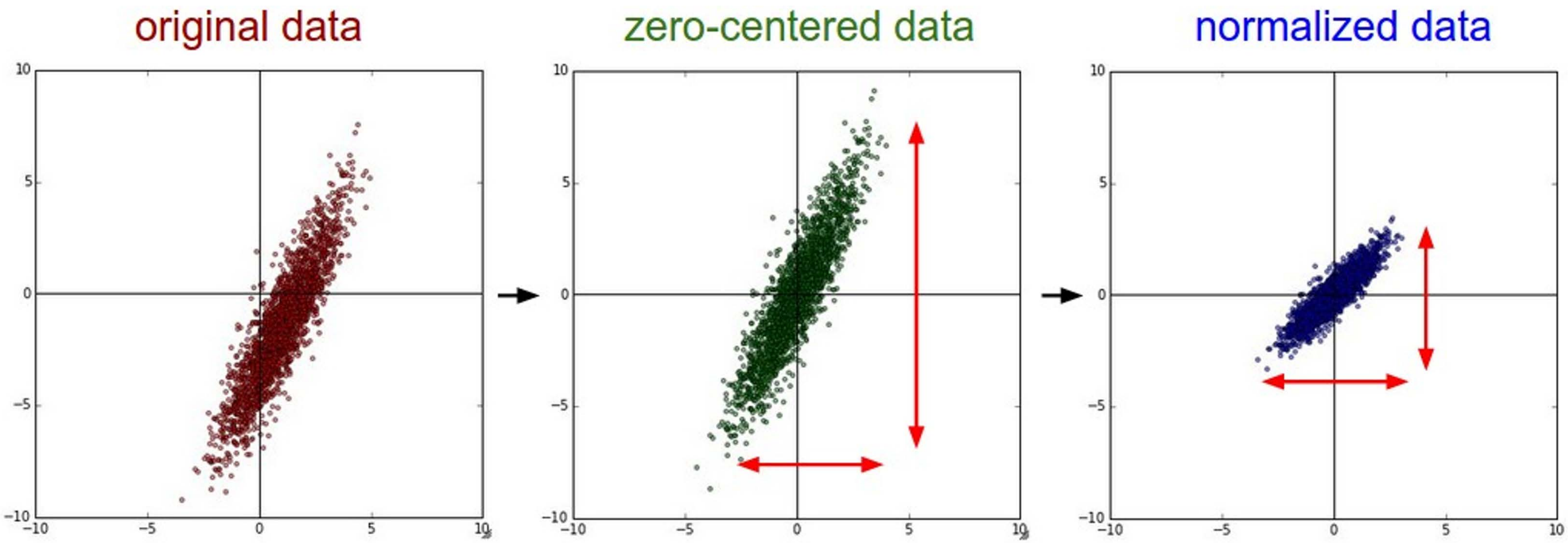
- Don’t think too hard. Just use **ReLU**
- Try out **Leaky ReLU / ELU / SELU / GELU** if you need to squeeze that last 0.1%
- Don’t use sigmoid or tanh

Some (very) recent architectures use GeLU instead of ReLU, but the gains are minimal

Dosovitskiy et al, “An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale”, ICLR 2021
Liu et al, “A ConvNet for the 2020s”, arXiv 2022

Data preprocessing

Data preprocessing

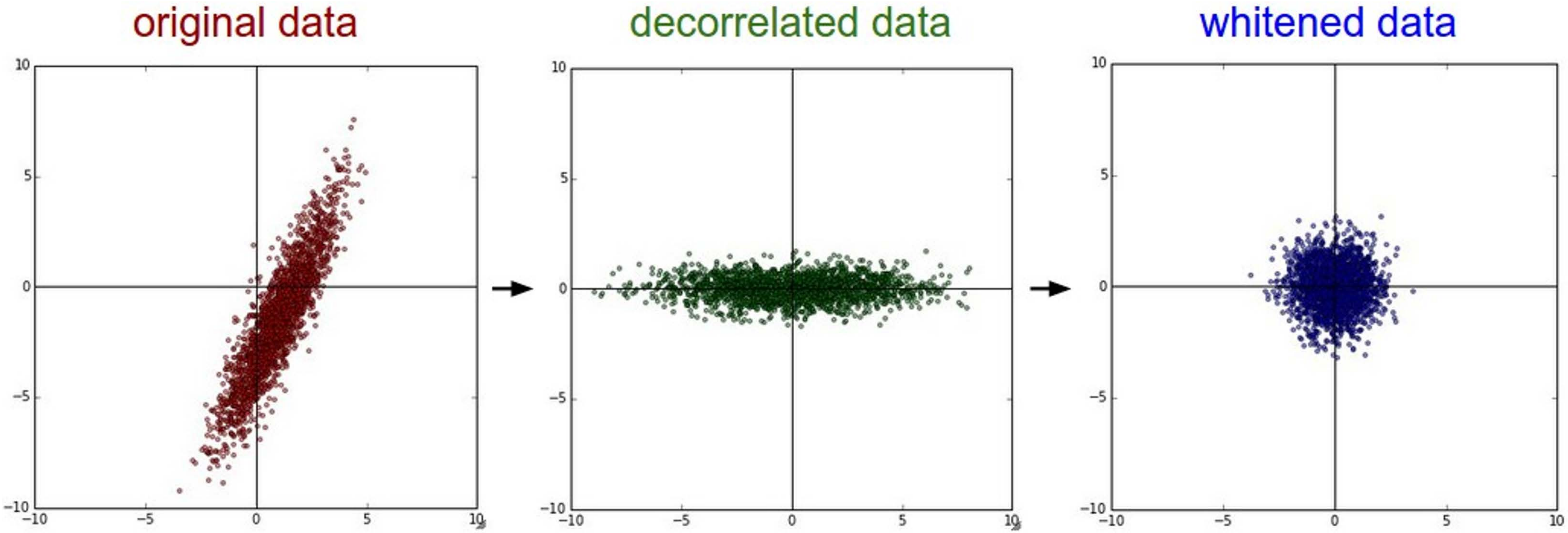



```
X -= np.mean(X, axis = 0)
X /= np.std(X, axis = 0)
```

(Assume X[NxD] is data matrix, each example in a row)

Data preprocessing

In practice, you may also see PCA and Whitening of the data

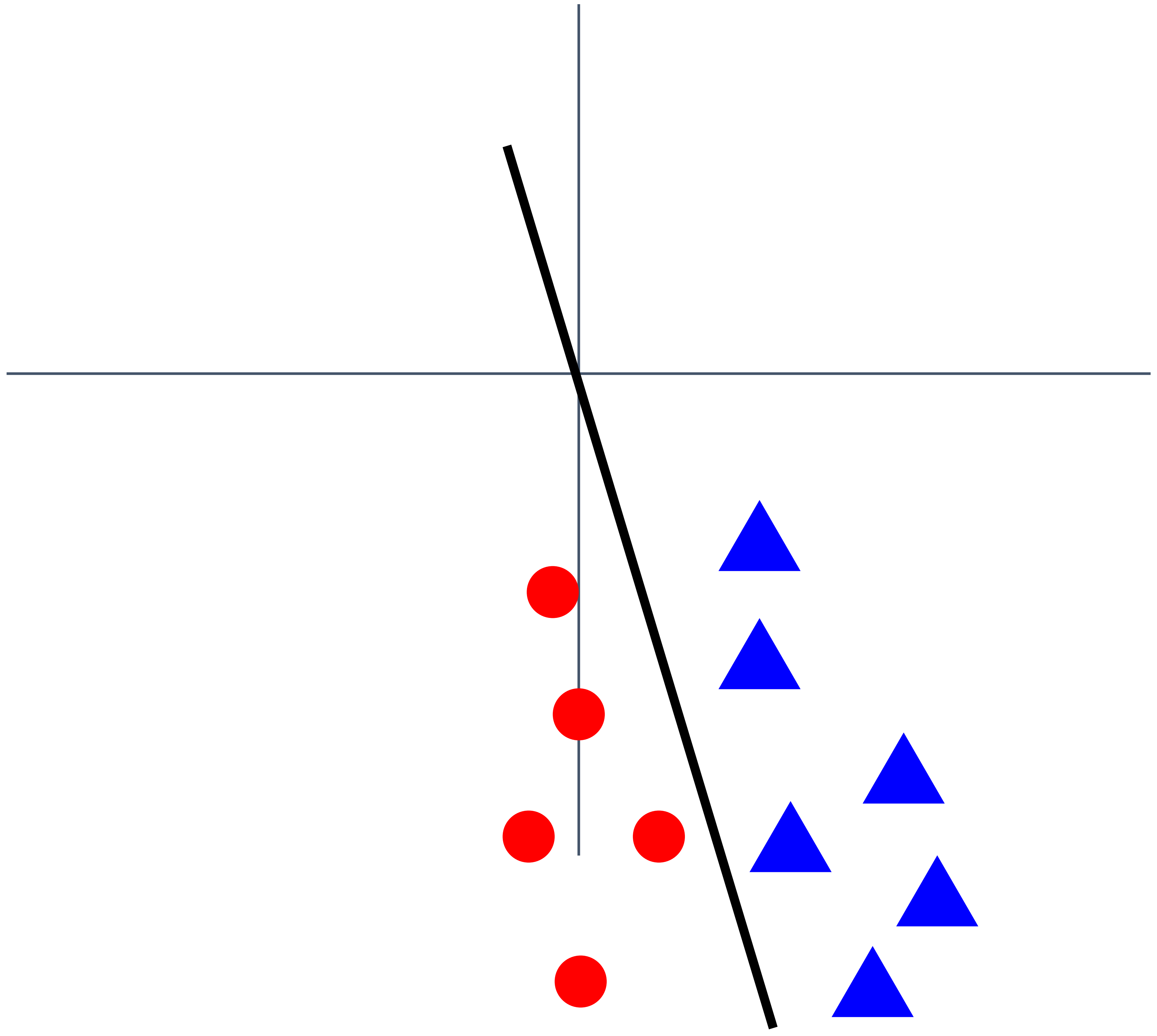


(Data has diagonal covariance matrix)

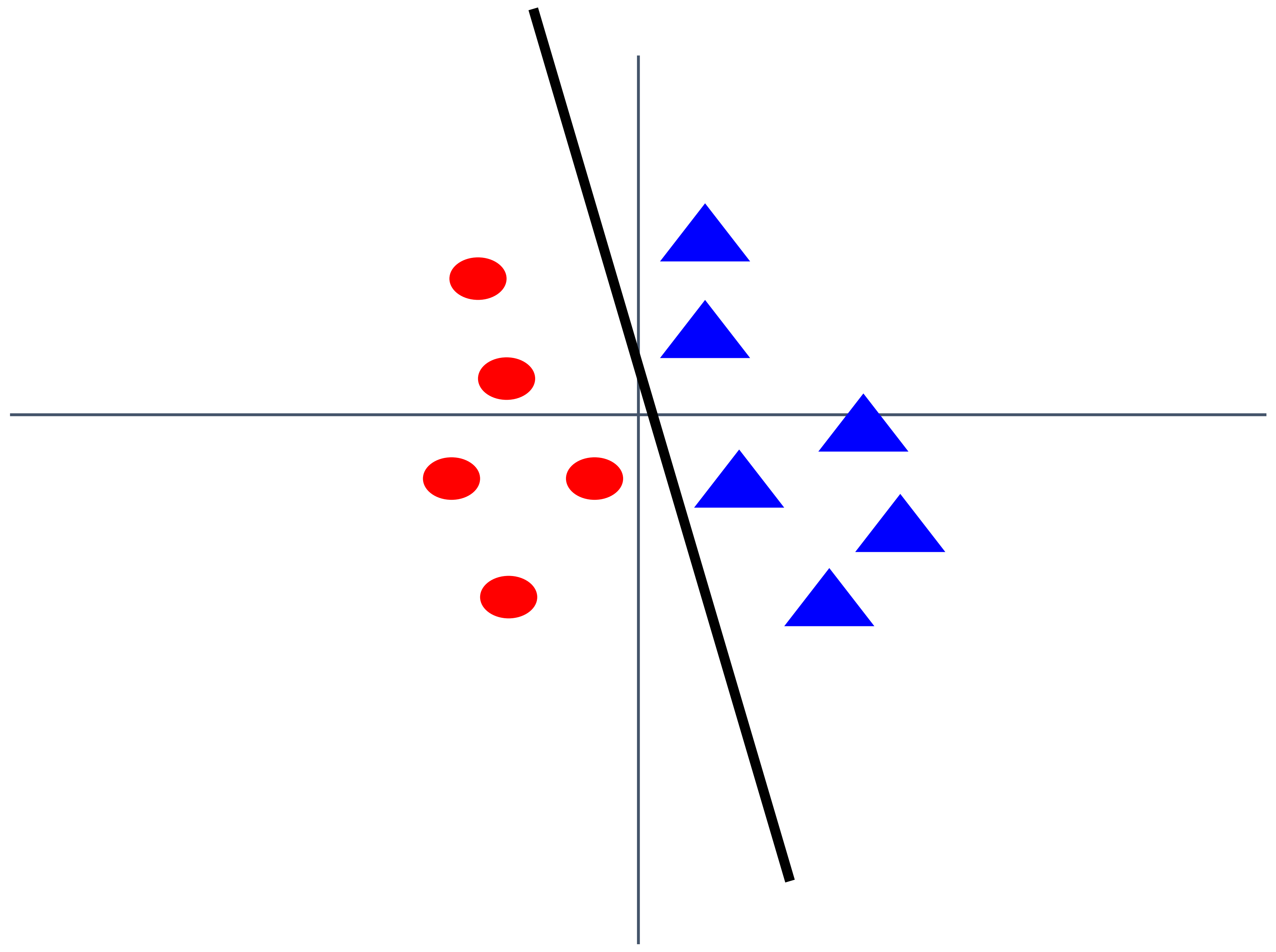
(Covariance matrix is the identity matrix)

Data preprocessing

Before normalization: Classification loss very sensitive to changes in weight matrix; hard to optimize



After normalization: less sensitive to small changes in weights; easier to optimize



Data preprocessing for Images

e.g. consider CIFAR-10 example with [32, 32, 3] images

- Subtract the mean image (e.g. AlexNet)
  (mean image = [32, 32, 3] array)

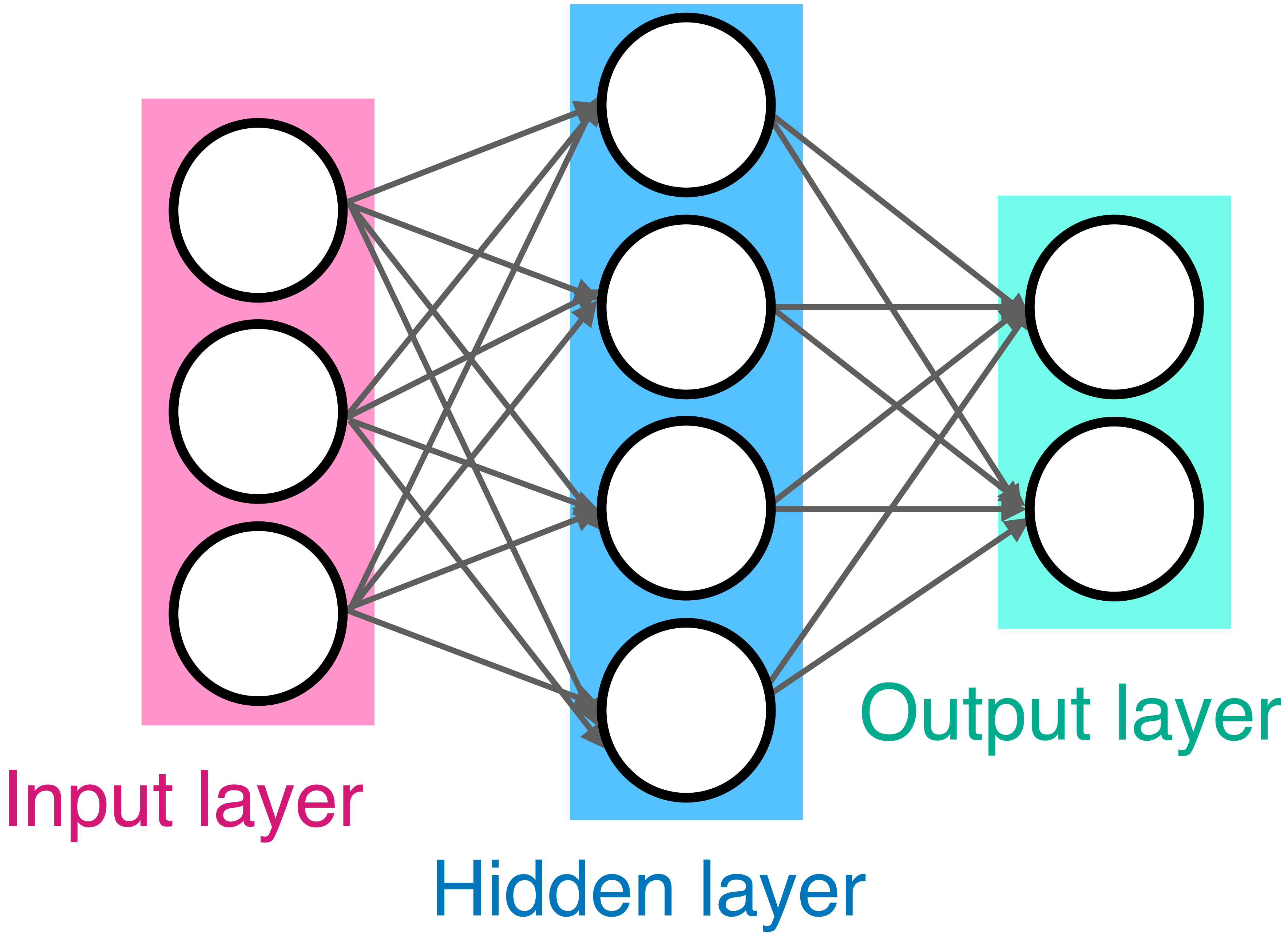
- Subtract per-channel mean (e.g. VGGNet)
  (mean along each channel = 3 numbers)

- Subtract per-channel mean and Divide by per-channel std (e.g. ResNet)
  (mean along each channel = 3 numbers)

Not common to do PCA or whitening

Weight initialization

Weight initialization



Q: What happens if we initialize all W=0, b=0?

A: All outputs are 0, all gradients are the same! No “symmetry breaking”

Weight initialization

Next idea: **small random numbers** (Gaussian with zero mean, std=0.01)

```
W = 0.01 * np.random.randn(Din, Dout)
```

Weight initialization

Next idea: **small random numbers** (Gaussian with zero mean, std=0.01)

```
W = 0.01 * np.random.randn(Din, Dout)
```

Works ~okay for small networks, but problems with deeper networks.

Weight initialization: Activation statistics

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Forward pass for a 6-layer net with hidden size 4096

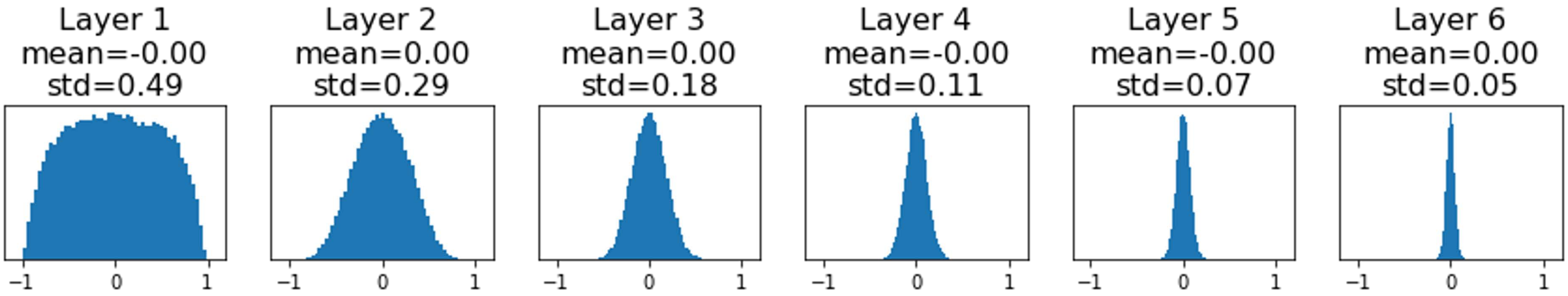
Weight initialization: Activation statistics

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Forward pass for a 6-layer net with hidden size 4096

All activations tend to zero for deeper network layers

Q: What do the gradients dL/dW look like?



Weight initialization: Activation statistics

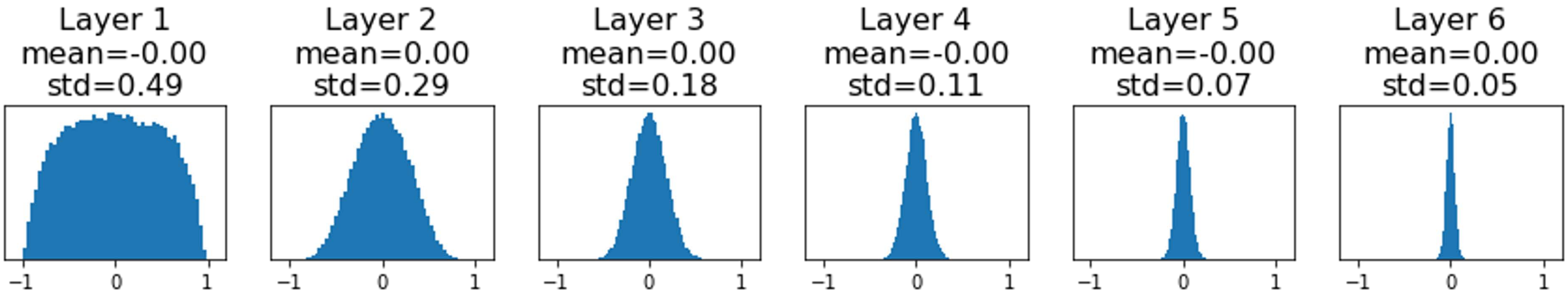
```python
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Forward pass for a 6-layer net with hidden size 4096

All activations tend to zero for deeper network layers

Q: What do the gradients dL/dW look like?

A: All zero, no learning :(



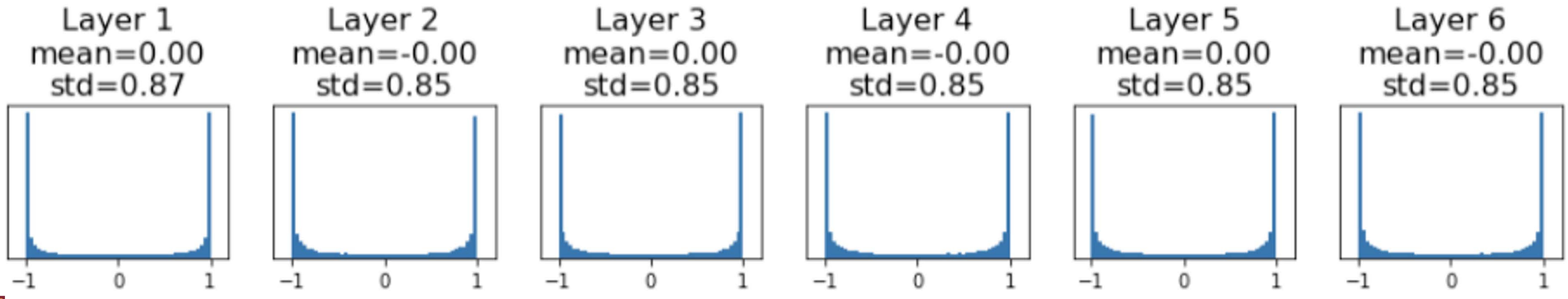
Weight initialization: Activation statistics

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.05 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Increase std of initial weights from 0.01 to 0.05

All activations saturate

Q: What do the gradients look like?



Weight initialization: Activation statistics

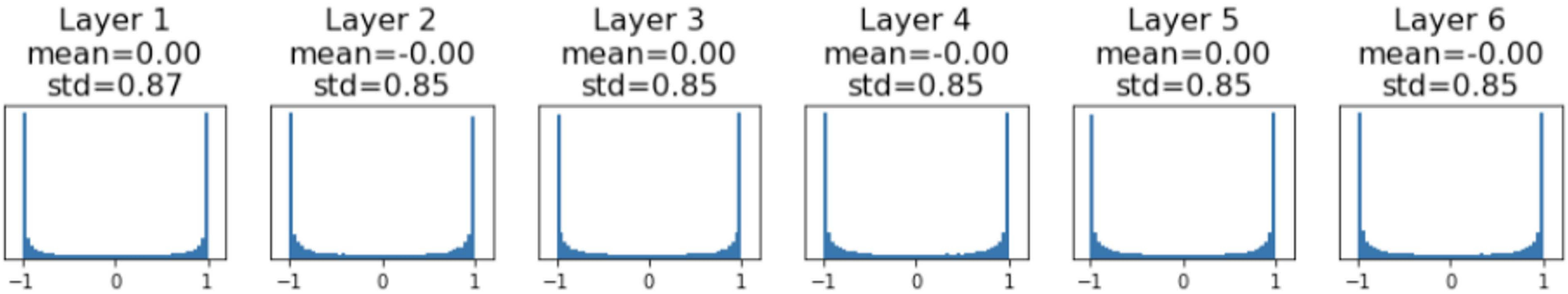
```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.05 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Increase std of initial weights from 0.01 to 0.05

All activations saturate

Q: What do the gradients look like?

A: Local gradients all zero, no learning :(



Weight initialization: Xavier Initialization

```python
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

"Xavier" initialization:
std = 1/sqrt(Din)

"Just right": Activations are nicely scaled for all layers!

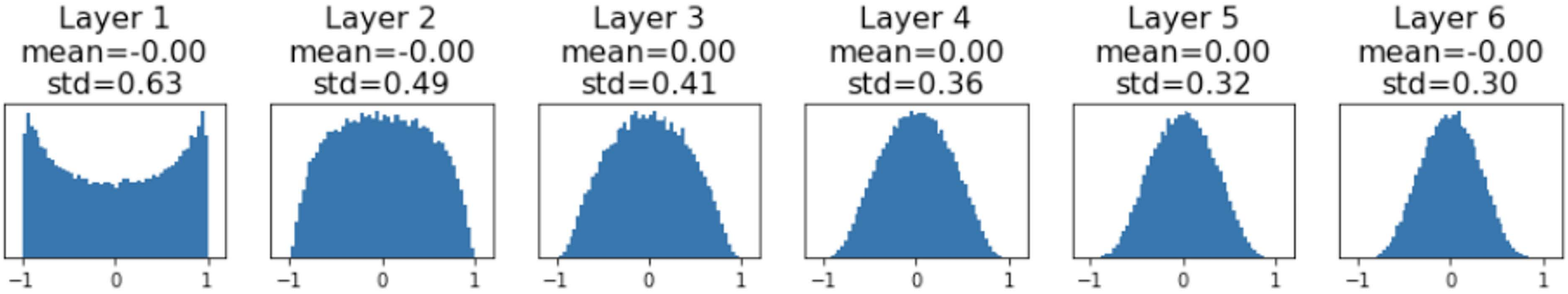
Glorot and Bengio, "Understanding the difficulty of training deep feedforward neural networks", AISTAT 2010

Weight initialization: Xavier Initialization

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

"Xavier" initialization:
std = 1/sqrt(Din)

"Just right": Activations are nicely scaled for all layers!



Glorot and Bengio, "Understanding the difficulty of training deep feedforward neural networks", AISTAT 2010

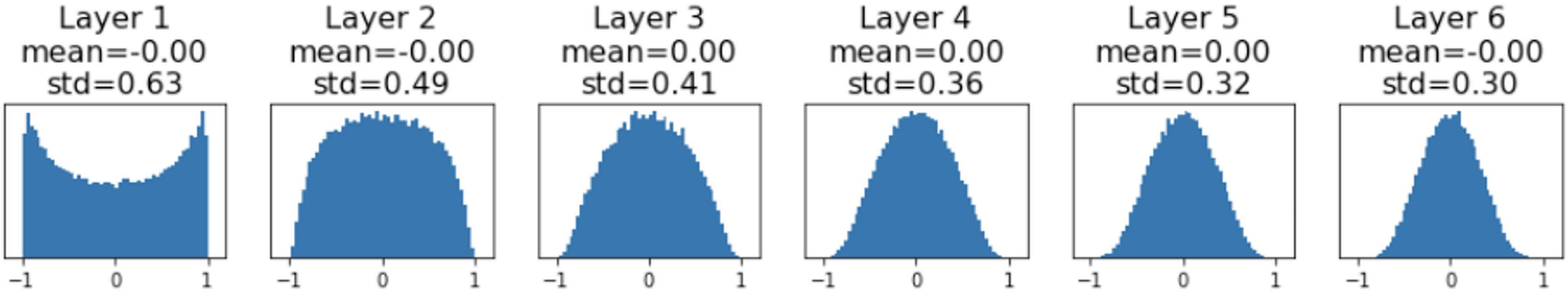
Weight initialization: Xavier Initialization

```python
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

"Xavier" initialization: std = 1/sqrt(Din)

"Just right": Activations are nicely scaled for all layers!

For conv layers, Din is $kernel\_size^2$ x input_channels



Glorot and Bengio, "Understanding the difficulty of training deep feedforward neural networks", AISTAT 2010

Weight initialization: What about ReLU?

```python
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

Change from tanh to ReLU

Xavier assumes zero centered activation function

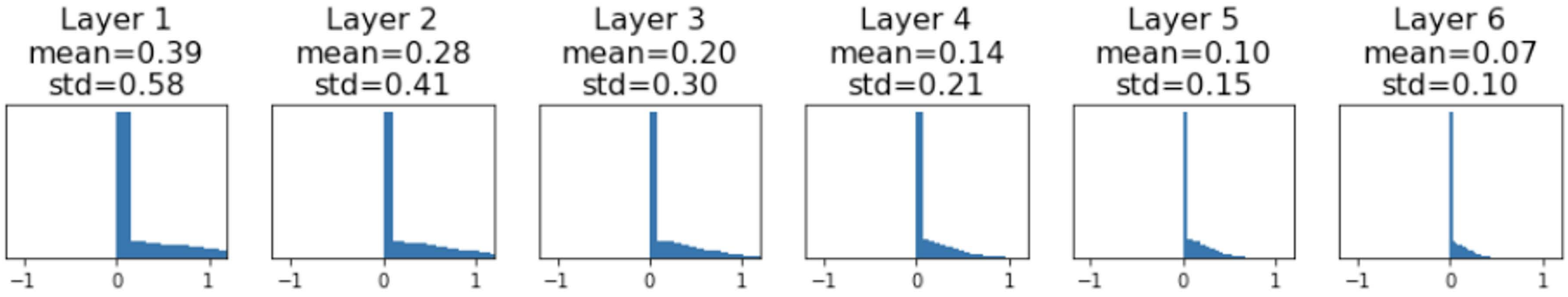
Weight initialization: What about ReLU?

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

Change from tanh to ReLU

Xavier assumes zero centered activation function

Activations collapse to zero again, no learning :(

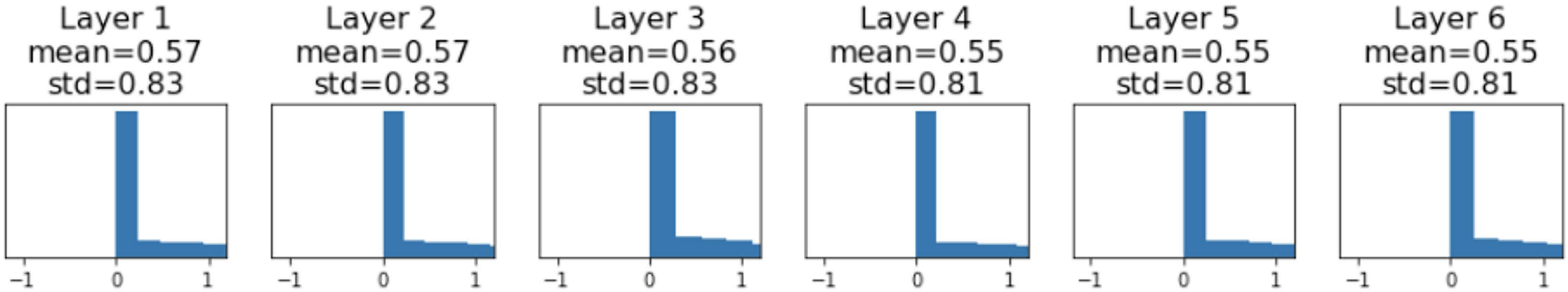


Weight initialization: Kaiming / MSRA initialization

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

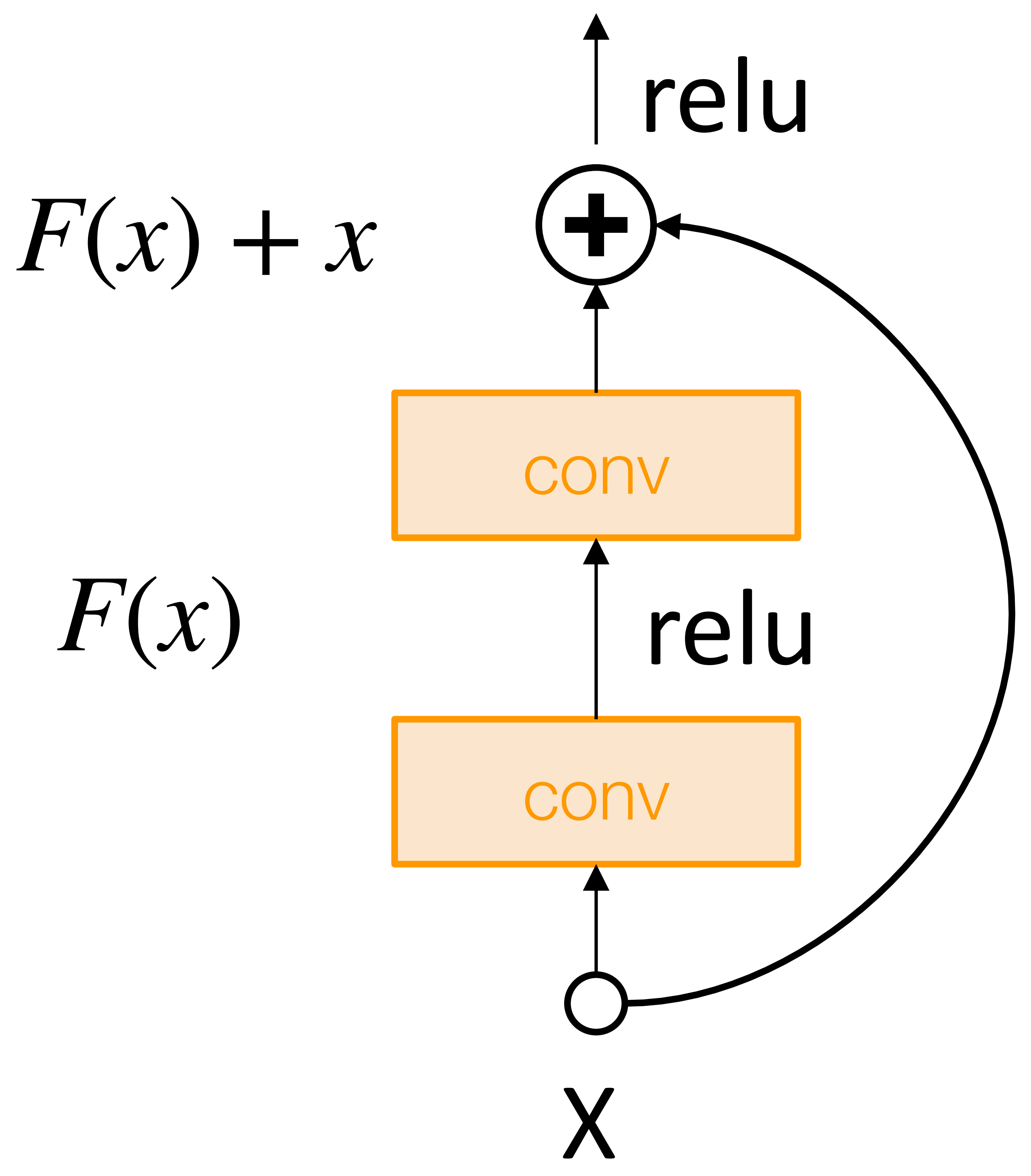
ReLU correction: std = sqrt(2 / Din)

"Just right" - activations nicely scaled for all layers



He et al., "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification", ICCV 2015

Weight initialization: Residual Networks



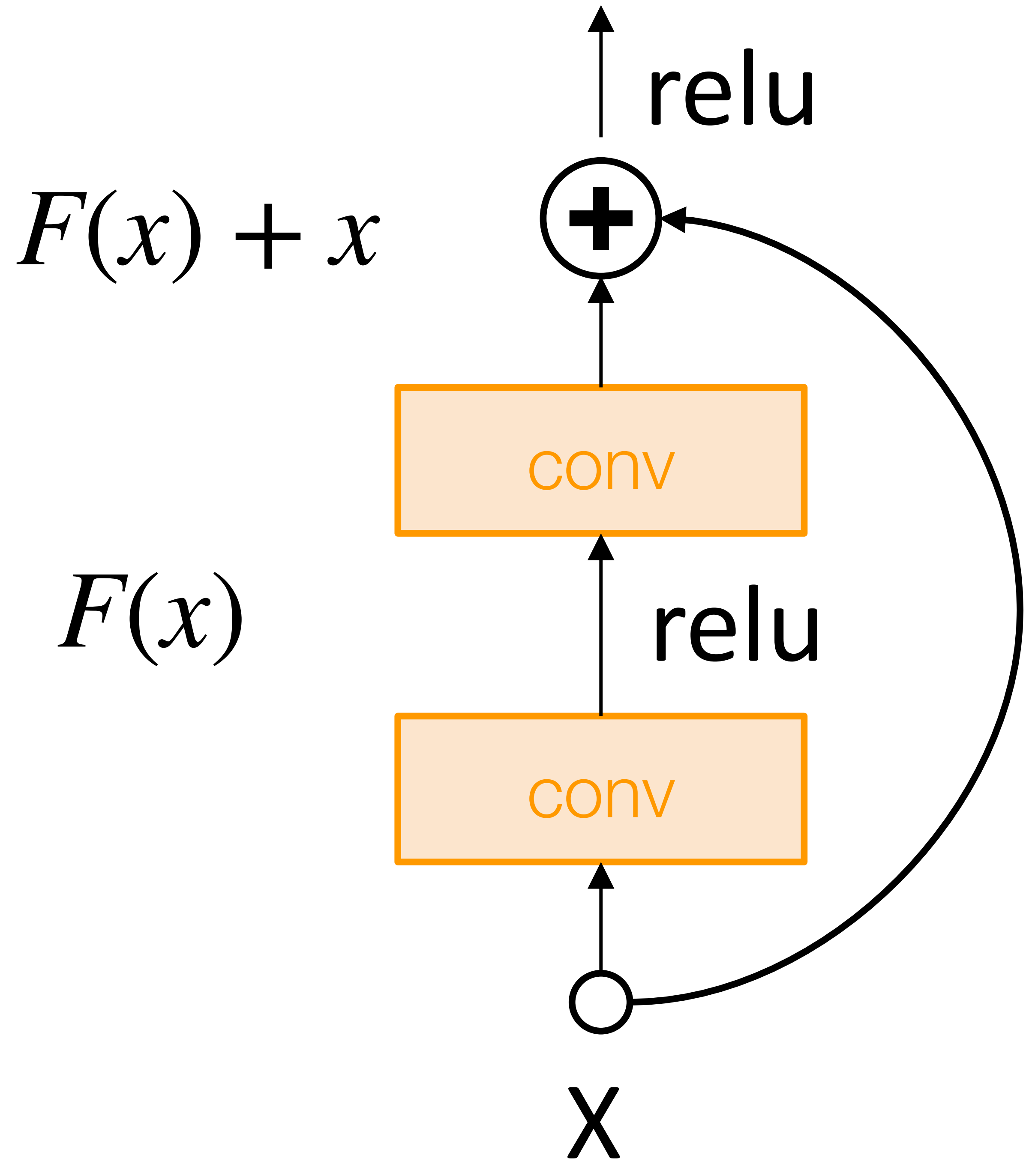
Residual Block

If we initialize with MSRA: then
$Var(F(x)) = Var(x)$

But then $Var(F(x) + x) > Var(x)$
variance grows with each block!

Zhang et al, "Fixup Initialization: Residual Learning Without Normalization", ICLR 2019

Weight initialization: Residual Networks



Residual Block

If we initialize with MSRA: then $Var(F(x)) = Var(x)$

But then $Var(F(x) + x) > Var(x)$ variance grows with each block!

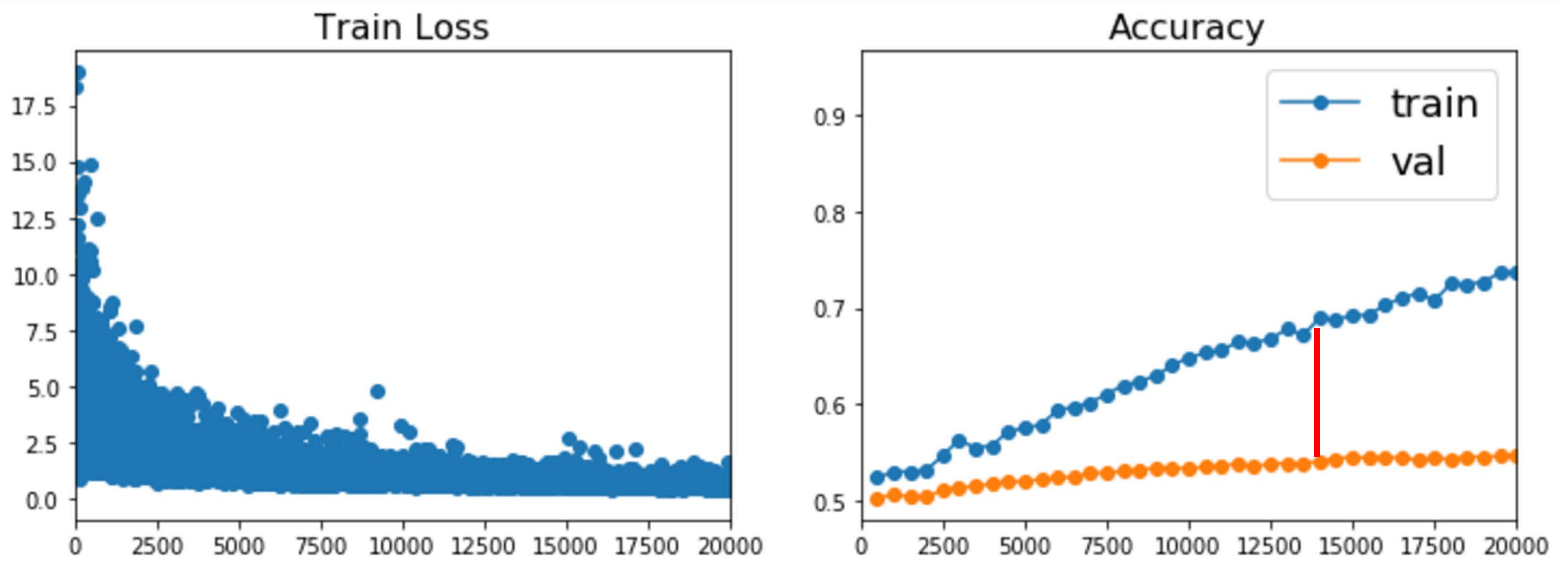
Solution: Initialize first conv with MSRA, initialize second conv to zero. Then $Var(F(x) + x) = Var(x)$

Zhang et al, "Fixup Initialization: Residual Learning Without Normalization", ICLR 2019

Proper initialization is an active area of research

- *Understanding the difficulty of training deep feedforward neural networks* by Glorot and Bengio, 2010
- *Exact solutions to the nonlinear dynamics of learning in deep linear neural networks* by Saxe et al, 2013
- *Random walk initialization for training very deep feedforward networks* by Sussillo and Abbott, 2014
- *Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification* by He et al., 2015
- *Data-dependent Initializations of Convolutional Neural Networks* by Krähenbühl et al., 2015
- *All you need is a good init*, Mishkin and Matas, 2015
- *Fixup Initialization: Residual Learning Without Normalization*, Zhang et al, 2019
- *The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks*, Frankle and Carbin, 2019

Now your model is training … but it overfits!



Regularization

Summary

1. One time setup: **Today**
   - Activation functions, data preprocessing, weight initialization, regularization
2. Training dynamics: **Next time**
   - Learning rate schedules; large-batch training; hyperparameter optimization
3. After training:
   - Model ensembles, transfer learning

Next Time: Training Neural Networks II

Reminder: Form your final project teams

- Read the individual brainstorming documents from other students in the google-folder.
- Talk to your fellow classmates.
  - Discuss your project idea with them.
  - Start working toward more concrete project as a team.
    - Adapt/Modify/Narrow down your ideas a team.
    - *Talk to Karthik during his OH to see the feasibility.*
  - Pick a few lecture topics from the list (provided here).
  - Pick 3 papers to read.
    - To reimplement as your project.
    - To help your project.
- Form a team of 2-3 students by 10/07 EOD using the google-sheet.
  - You do not have to finalize your project by this date.
  - You should finalize your group.

Visit RPM Lab!

DeepRob

Lecture 9
Training Neural Networks I
University of Minnesota