CS 4644-DL / 7643-A: LECTURE 9
DANFEI XU

Topics:

- Convolutional Neural Networks Architectures (cont.)
- Training Neural Networks (Part 1)

Administrative

- PS1/HW1 due **today** (grace period till Sep 19st)
- PS2/HW2 out: **Difficult assignment. Start early!**
- Project proposal due **Sep 24th**. **No extension**

CNN Architectures

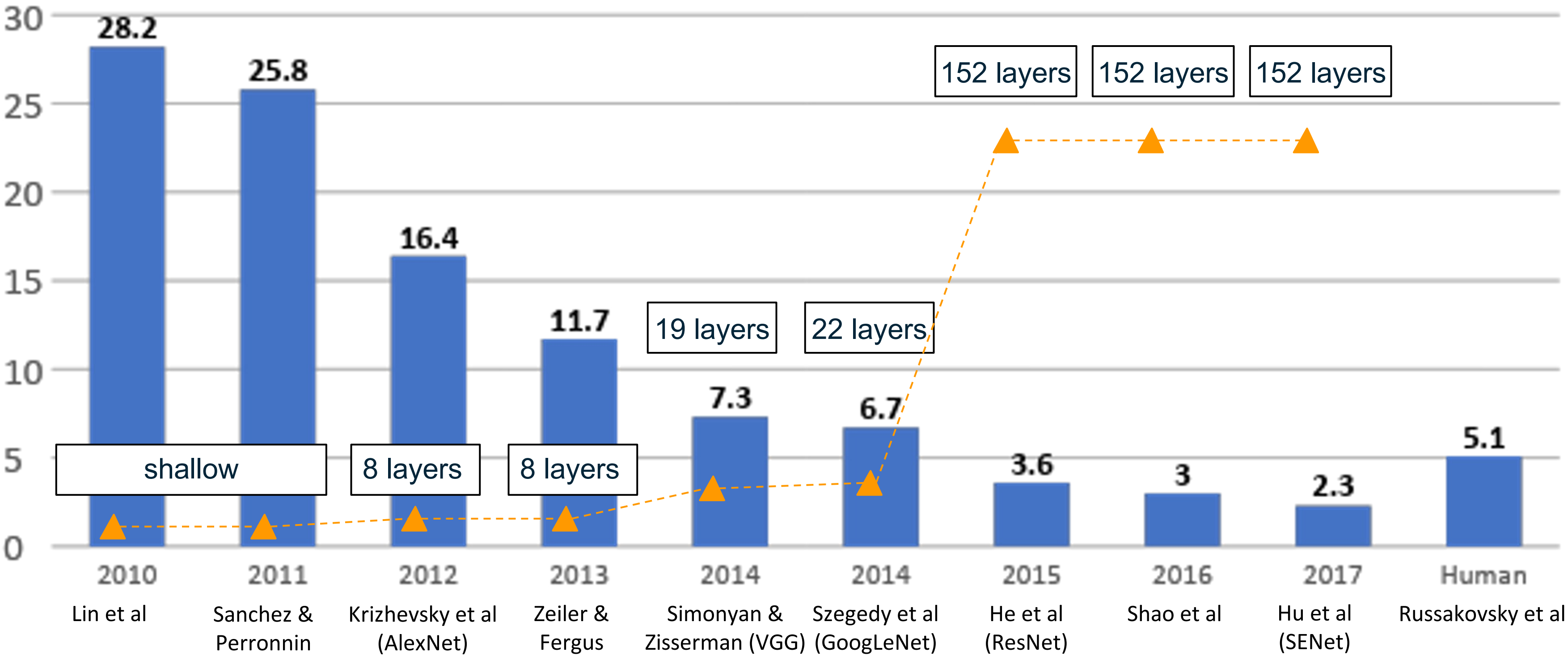
Case Studies

- AlexNet
- VGG
- GoogLeNet
- ResNet

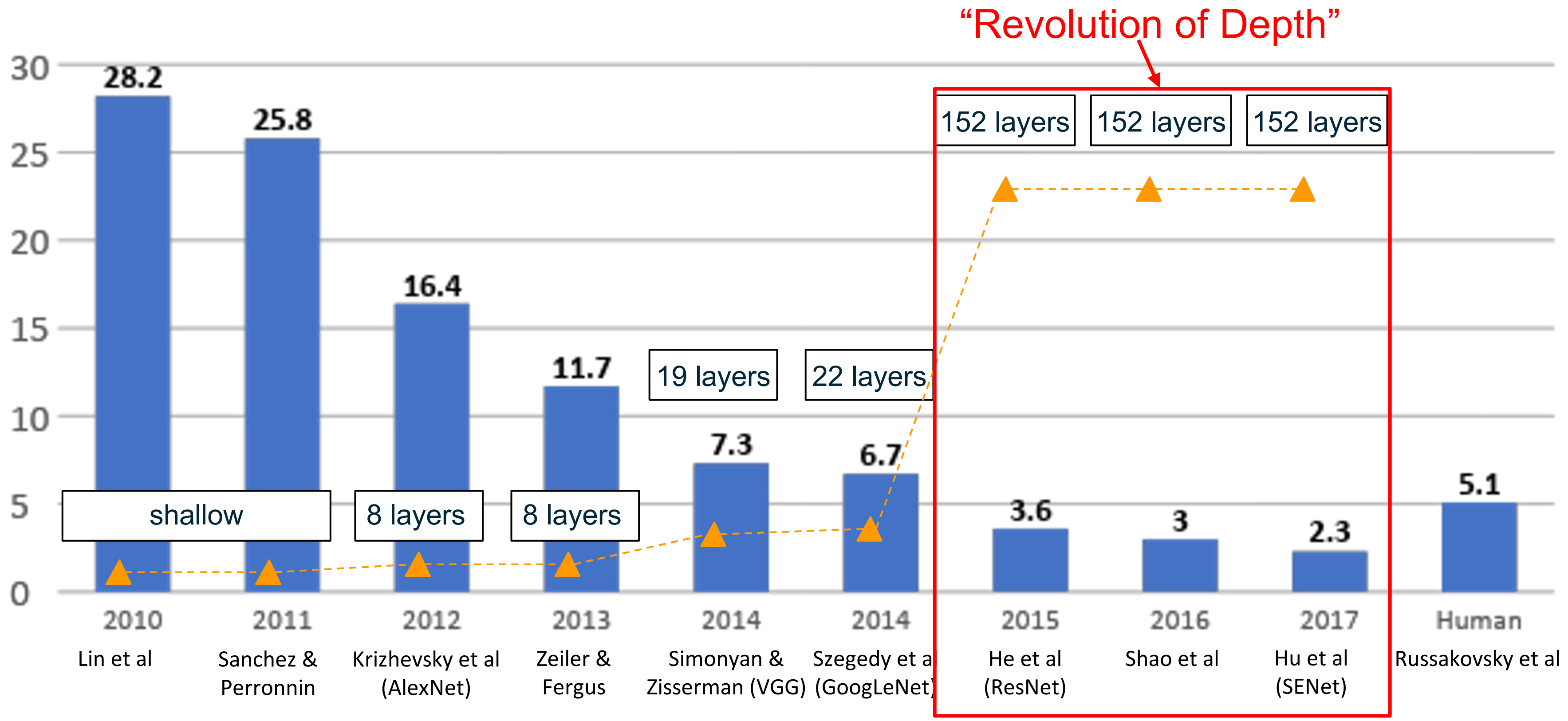
Also....

- SENet
- Wide ResNet
- ResNeXT
- DenseNet
- MobileNets
- NASNet
- EfficientNet

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

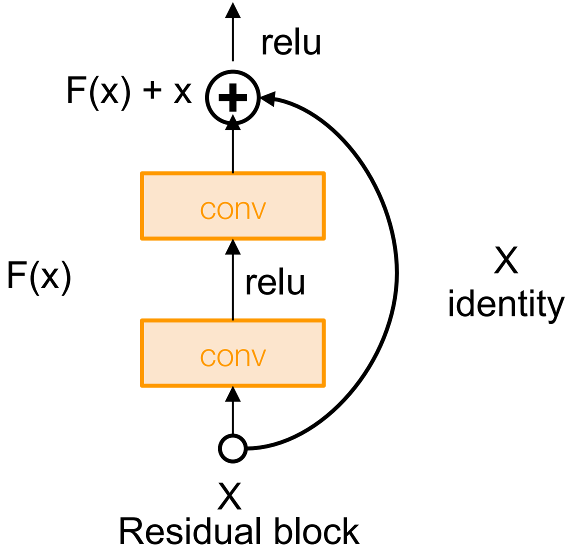


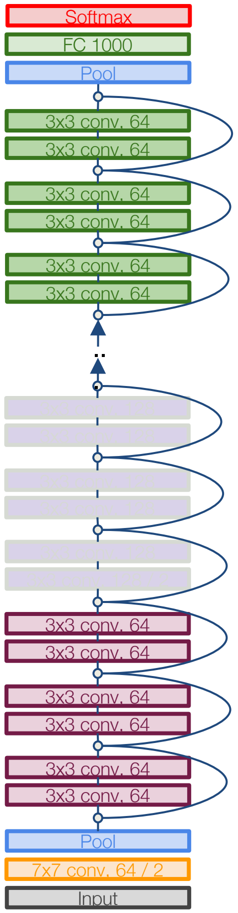
Case Study: ResNet

[He et al., 2015]

Very deep networks using residual connections

- 152-layer model for ImageNet
- ILSVRC'15 classification winner (3.57% top 5 error)
- Swept all classification and detection competitions in ILSVRC'15 and COCO'15!





Case Study: ResNet

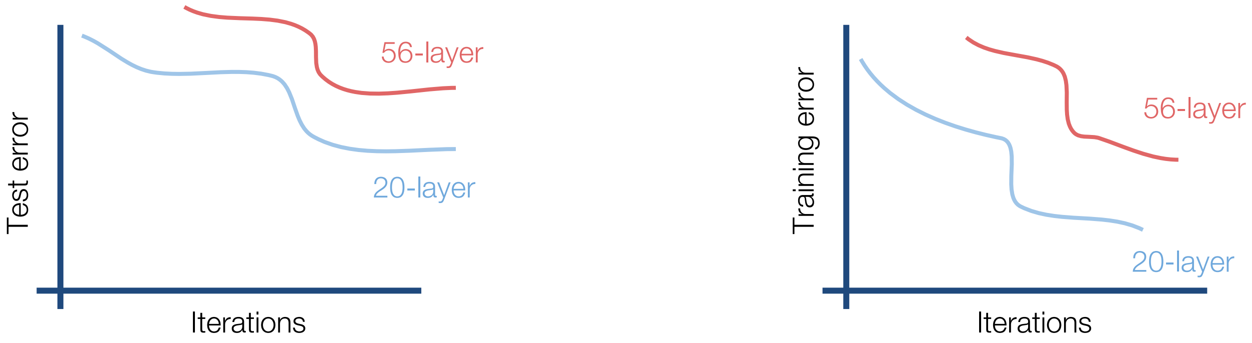
[He et al., 2015]

What happens when we continue stacking deeper layers on a “plain” convolutional neural network?

Case Study: ResNet

[He et al., 2015]

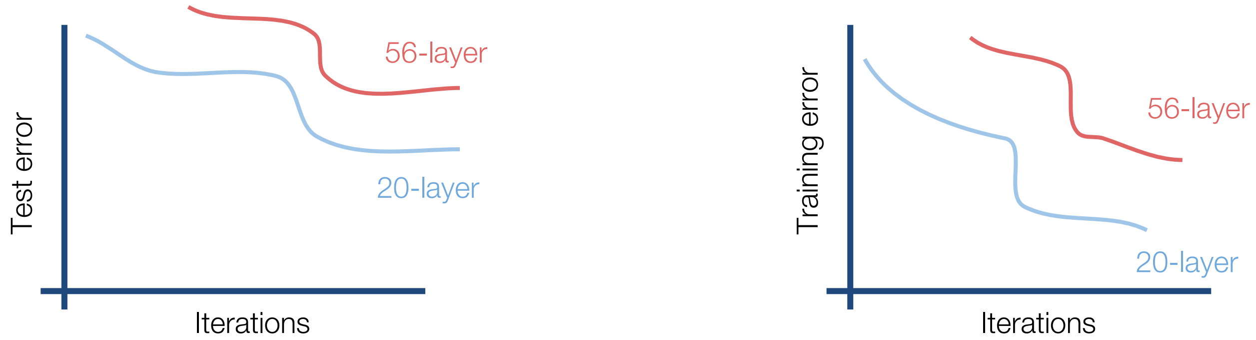
What happens when we continue stacking deeper layers on a “plain” convolutional neural network?



Case Study: ResNet

[He et al., 2015]

What happens when we continue stacking deeper layers on a “plain” convolutional neural network?



56-layer model performs worse on both test and training error

-> The deeper model performs worse, but it’s not caused by overfitting!

Case Study: ResNet

[He et al., 2015]

Fact: Deep models have more representation power (more parameters) than shallower models.

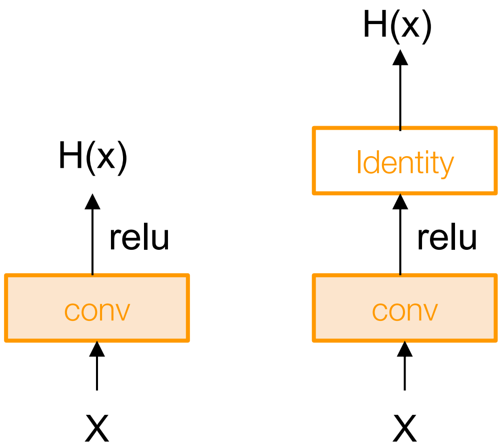
Hypothesis: the problem is an *optimization* problem, **deeper models are harder to optimize**

Case Study: ResNet

[He et al., 2015]

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 <u>as good as</u> shallow models



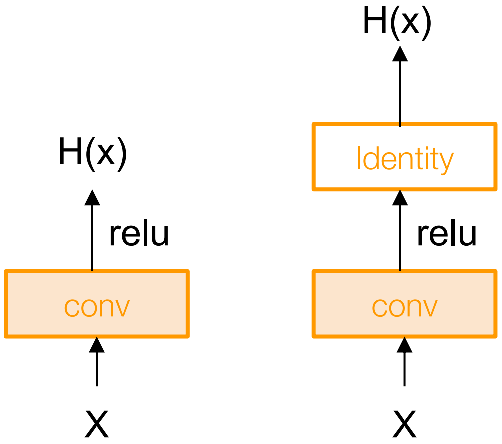
Case Study: ResNet

[He et al., 2015]

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

Deeper models are harder to optimize. They don't learn identity functions (no-op) to emulate shallow models



Case Study: ResNet

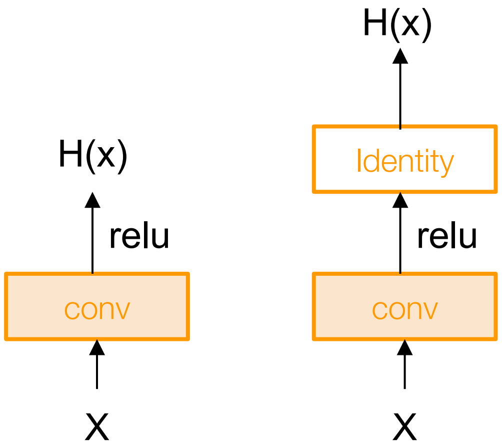
[He et al., 2015]

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

Deeper models are harder to optimize. They don't learn identity functions (no-op) to emulate shallow models

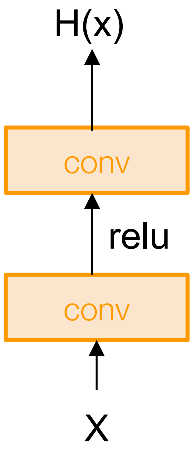
Solution: Change the network so learning identity functions (no-op) as extra layers is easy



Case Study: ResNet

[He et al., 2015]

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

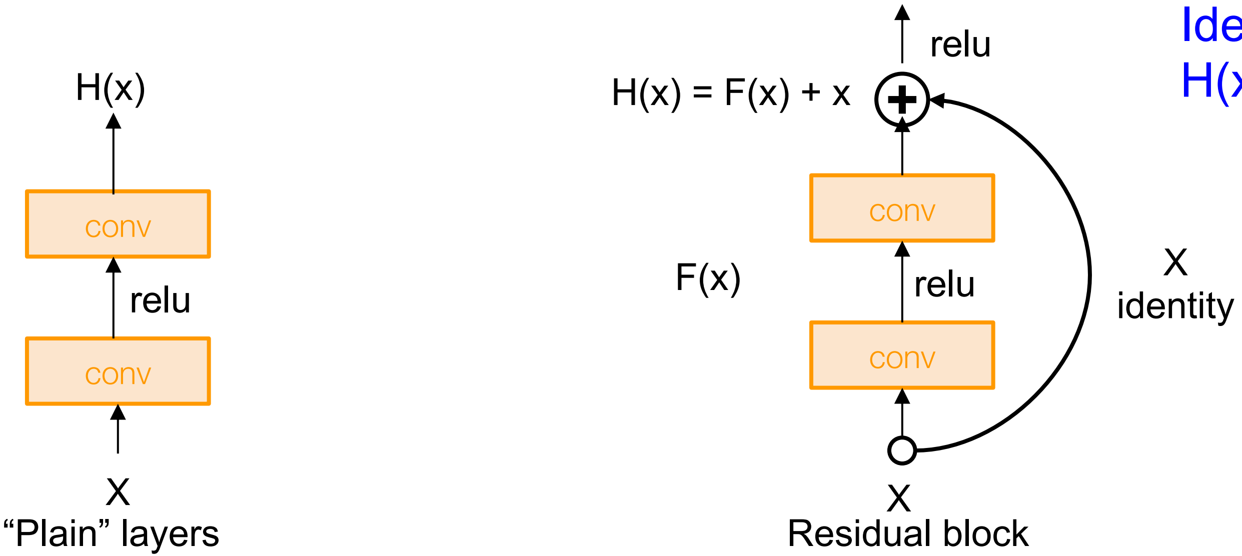


“Plain” layers

Case Study: ResNet

[He et al., 2015]

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

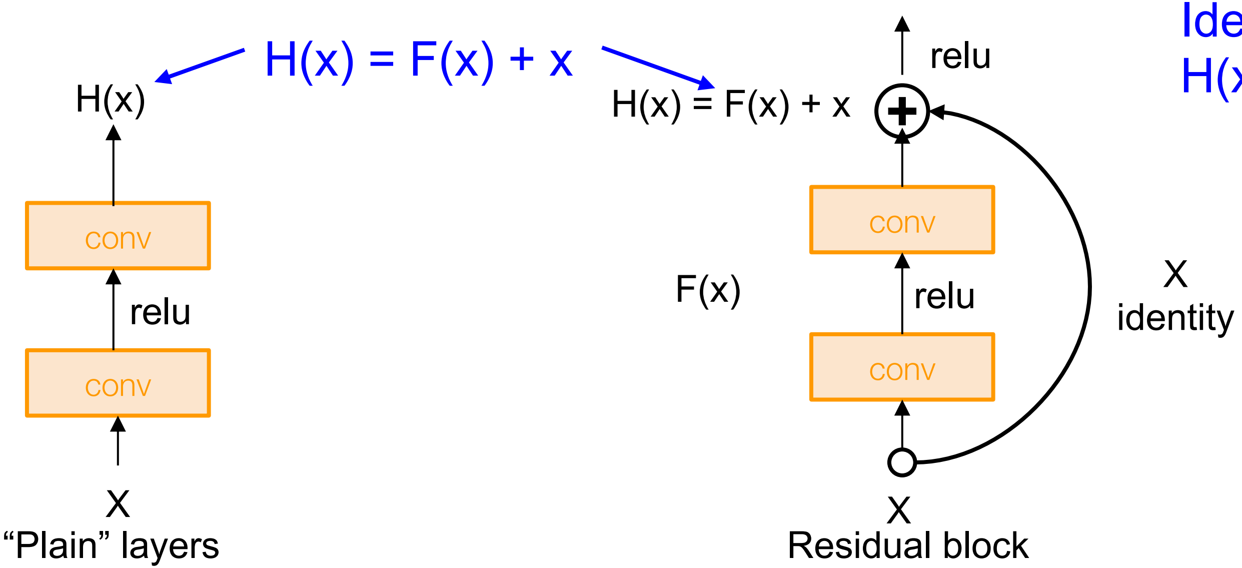


Identity mapping:
H(x) = x if F(x) = 0

Case Study: ResNet

[He et al., 2015]

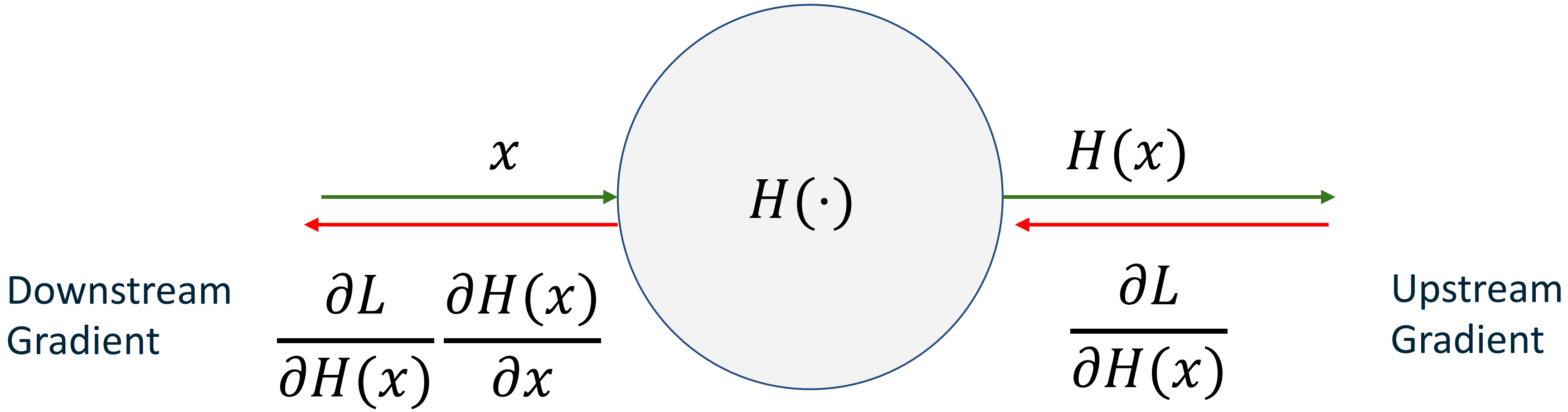
Solution: Change the network so learning identity functions as extra layers is easy



Identity mapping:
H(x) = x if F(x) = 0

Use layers to fit **residual**
F(x) = H(x) - x
instead of
H(x) directly

The Vanishing Gradient Problem



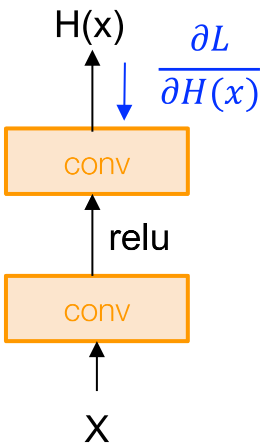
$$H(x) = W^T x + b$$

$$\frac{\partial H(x)}{\partial x} = W^T$$

If W is small, downstream gradient is small.

Each small W in the chain makes gradient *progressively smaller ->*
Vanishing Gadient during backpropagation

Case Study: ResNet

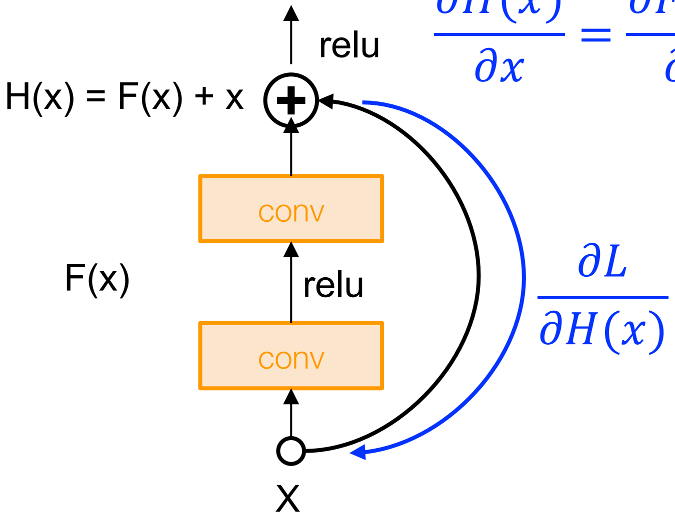


"Plain" layers

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial H(x)} \frac{\partial H(x)}{\partial x}$$

Potentially problematic

$$\frac{\partial H(x)}{\partial x} = \frac{\partial F(x)}{\partial x} + 1$$



$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial H(x)} \frac{\partial H(x)}{\partial x}$$

$$= \frac{\partial L}{\partial H(x)} \frac{\partial F(x)}{\partial x} + \frac{\partial L}{\partial H(x)}$$

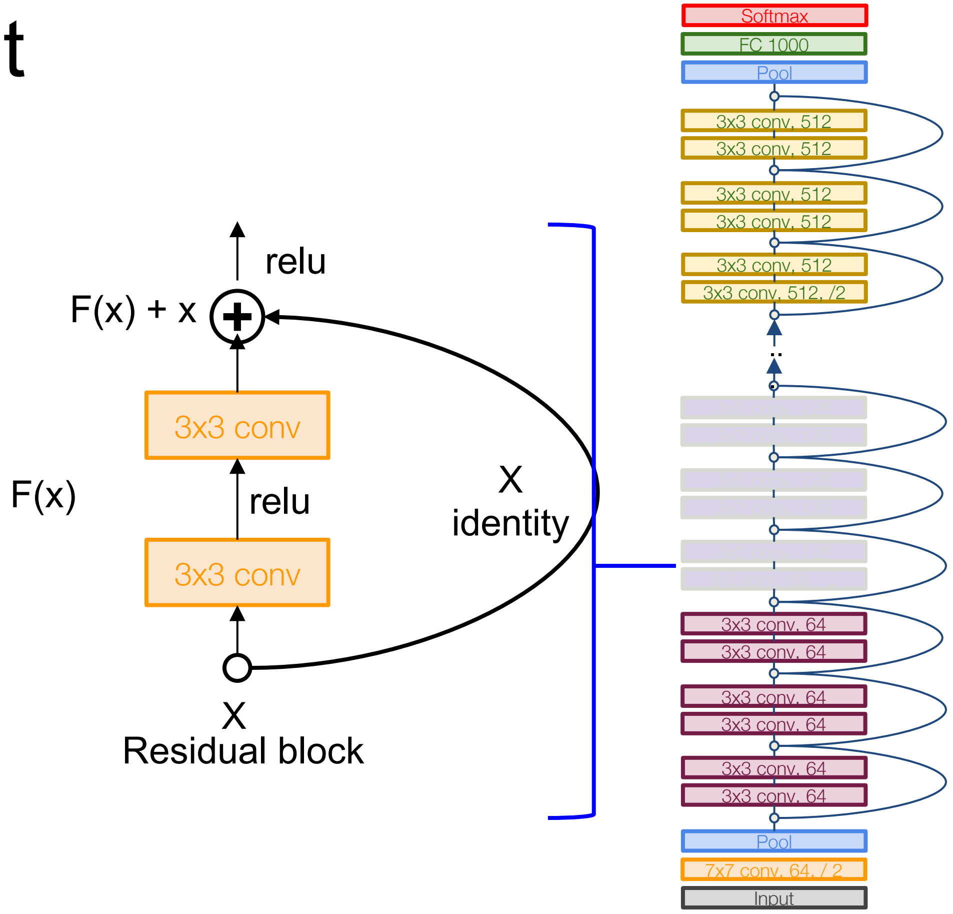
Direct gradient pathway

Case Study: ResNet

[He et al., 2015]

Full ResNet architecture:

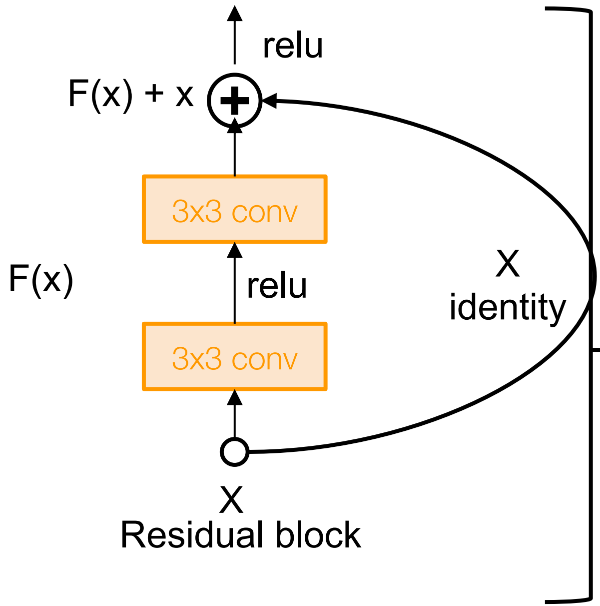
- Stack residual blocks
- Every residual block has two 3x3 conv layers

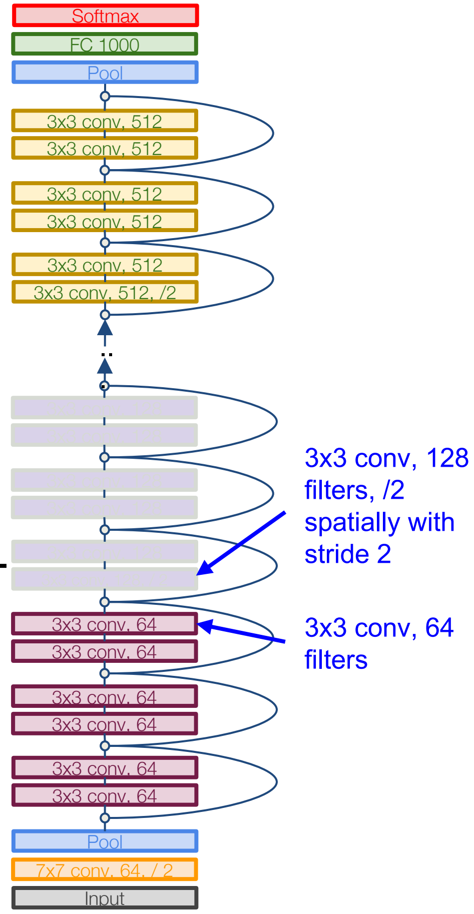


Case Study: ResNet

[He et al., 2015]

Full ResNet architecture:
- Stack residual blocks
- Every residual block has two 3x3 conv layers
- Periodically, double # of filters and downsample spatially using stride 2 (/2 in each dimension) Reduce the activation volume by half.

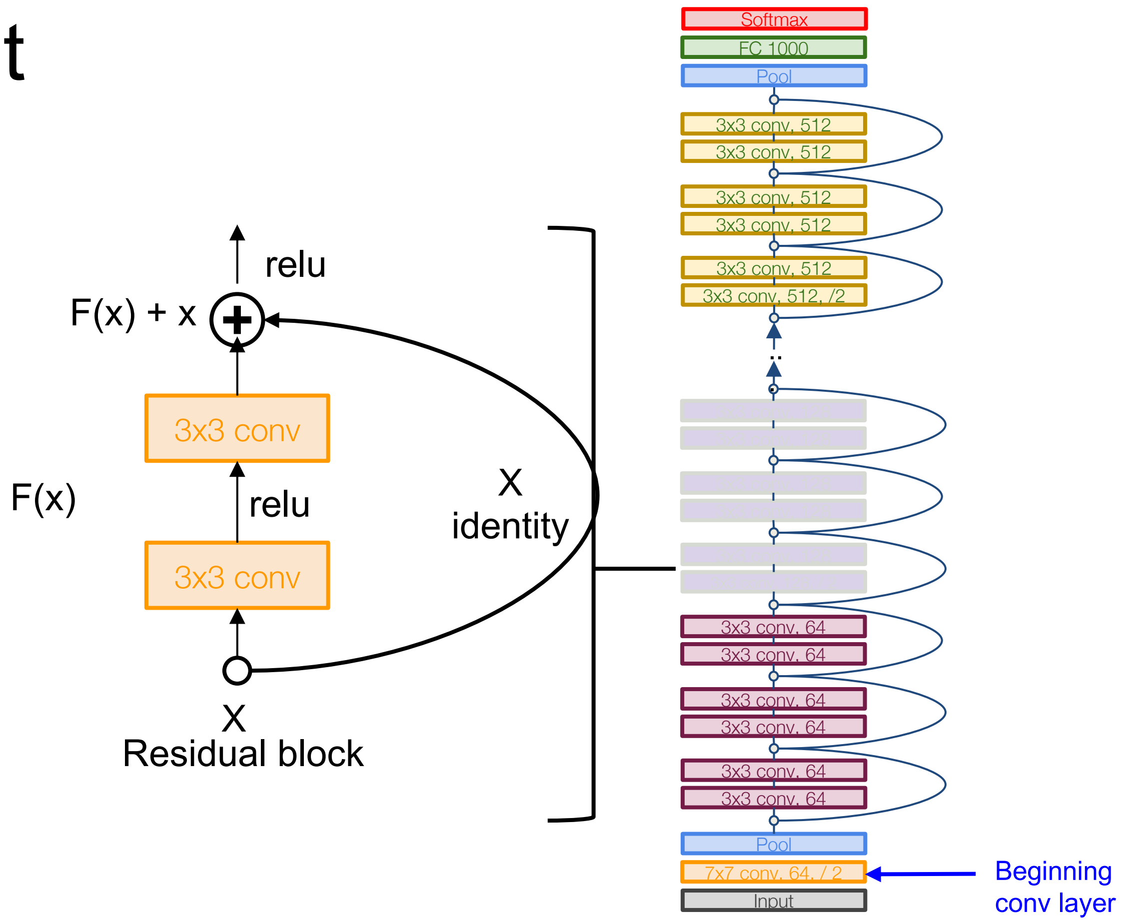




Case Study: ResNet

[He et al., 2015]

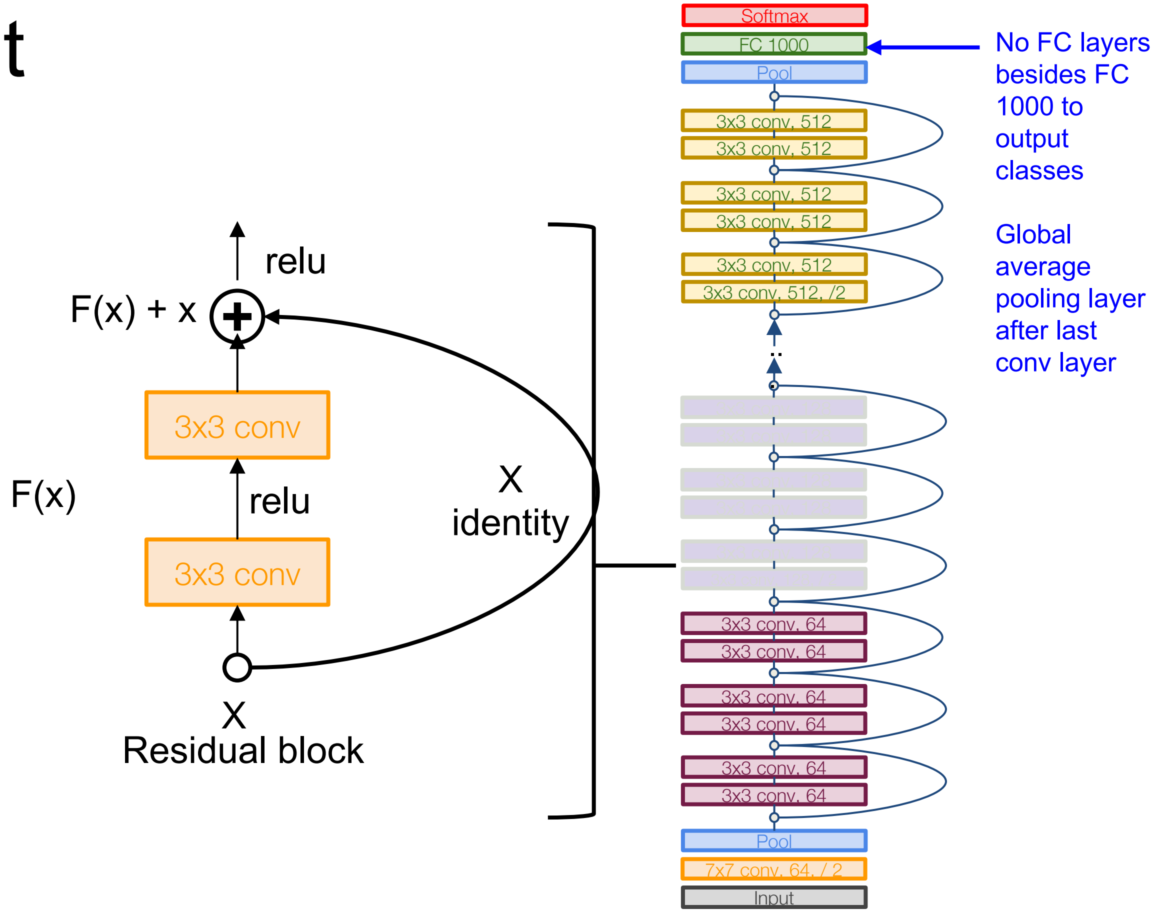
Full ResNet architecture:
- Stack residual blocks
- Every residual block has two 3x3 conv layers
- Periodically, double # of filters and downsample spatially using stride 2 (/2 in each dimension) Reduce the activation volume by half.
- Additional conv layer at the beginning (stem)



Case Study: ResNet

[He et al., 2015]

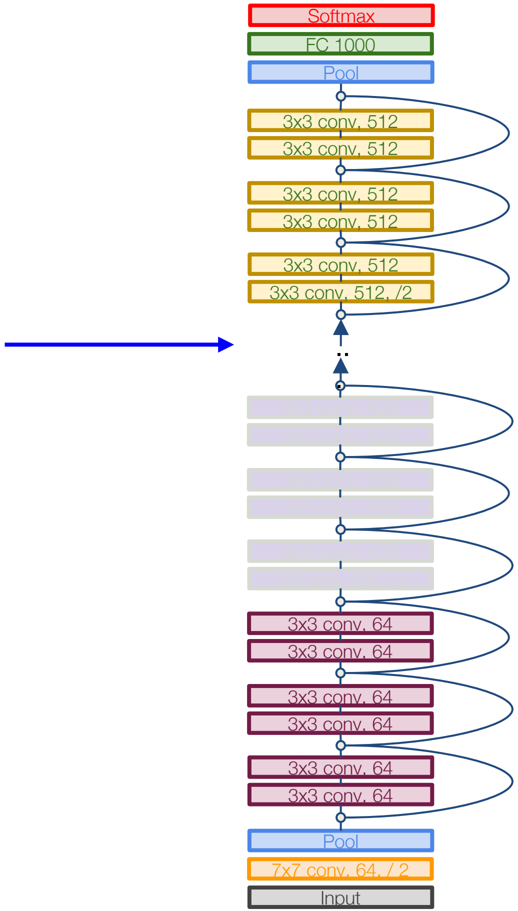
Full ResNet architecture:
- Stack residual blocks
- Every residual block has two 3x3 conv layers
- Periodically, double # of filters and downsample spatially using stride 2 (/2 in each dimension) Reduce the activation volume by half.
- Additional conv layer at the beginning (stem)
- No FC layers at the end (only FC 1000 to output classes)



Case Study: ResNet

[He et al., 2015]

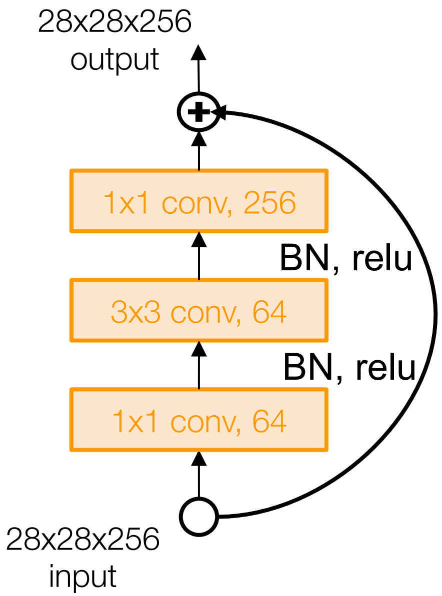
Total depths of 18, 34, 50, 101, or 152 layers for ImageNet



Case Study: ResNet

[He et al., 2015]

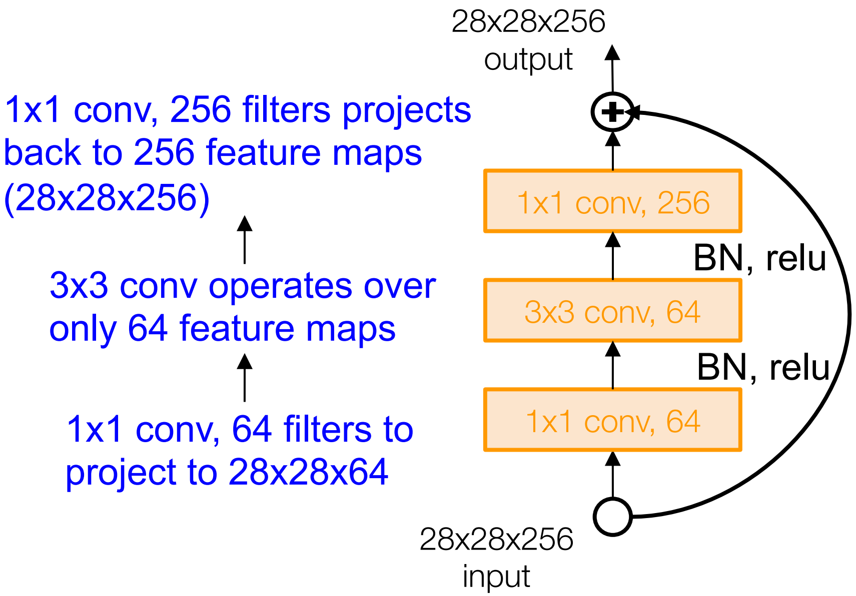
For deeper networks (ResNet-50+), use "bottleneck" layer to improve efficiency (similar to GoogLeNet)



Case Study: ResNet

[He et al., 2015]

For deeper networks (ResNet-50+), use "bottleneck" layer to improve efficiency (similar to GoogLeNet)



Case Study: ResNet

[He et al., 2015]

Training ResNet in practice:

- Batch Normalization after every CONV layer (this lecture)
- Xavier initialization from He et al. (this lecture)
- SGD + Momentum (this lecture)
- Learning rate: 0.1, divided by 10 when validation error plateaus
- Mini-batch size 256
- Weight decay of 1e-5
- No dropout used

Case Study: ResNet

[He et al., 2015]

Experimental Results

- Able to train very deep networks without degrading (152 layers on ImageNet, 1202 on Cifar)
- Deeper networks now achieve lower training error as expected
- Swept 1st place in all ILSVRC and COCO 2015 competitions

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

Case Study: ResNet

[He et al., 2015]

Experimental Results

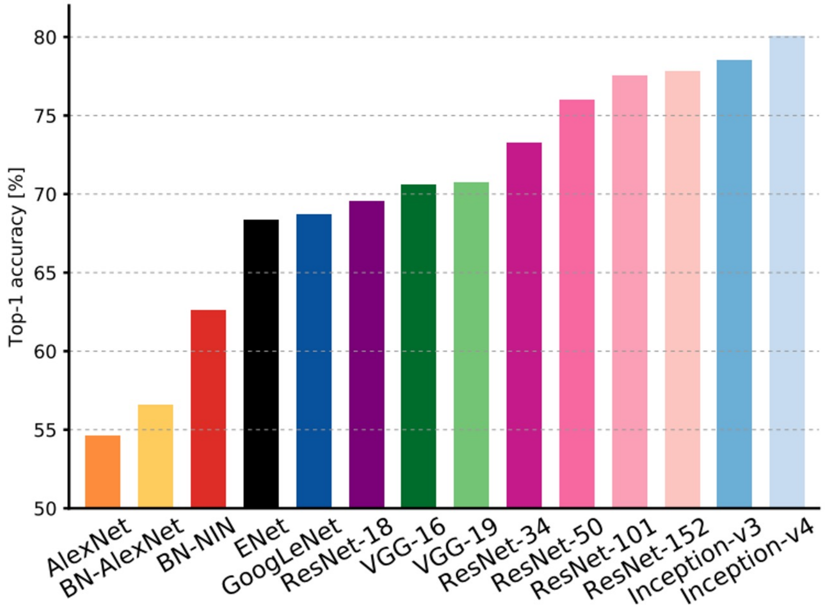
- Able to train very deep networks without degrading (152 layers on ImageNet, 1202 on Cifar)
- Deeper networks now achieve lower training error as expected
- Swept 1st place in all ILSVRC and COCO 2015 competitions

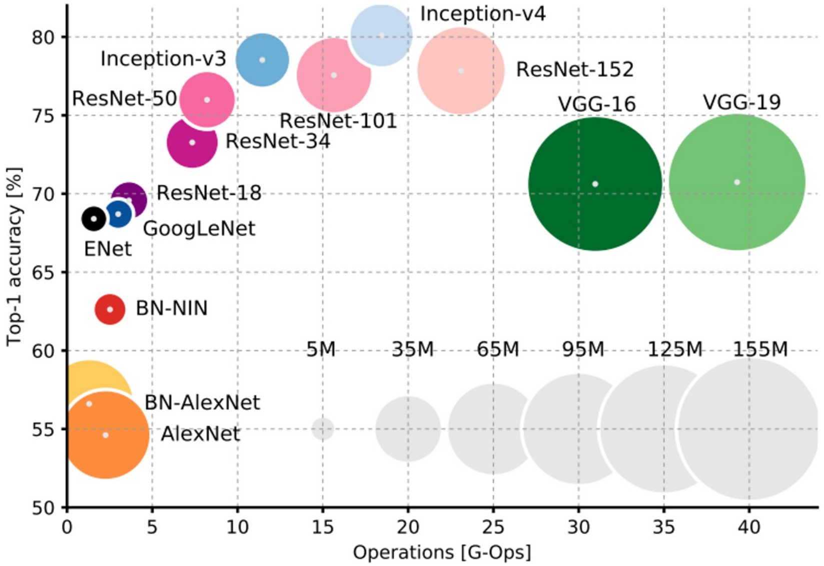
MSRA @ ILSVRC & COCO 2015 Competitions

- **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

ILSVRC 2015 classification winner (3.6% top 5 error) -- better than "human performance"! (Russakovsky 2014)

Comparing complexity...



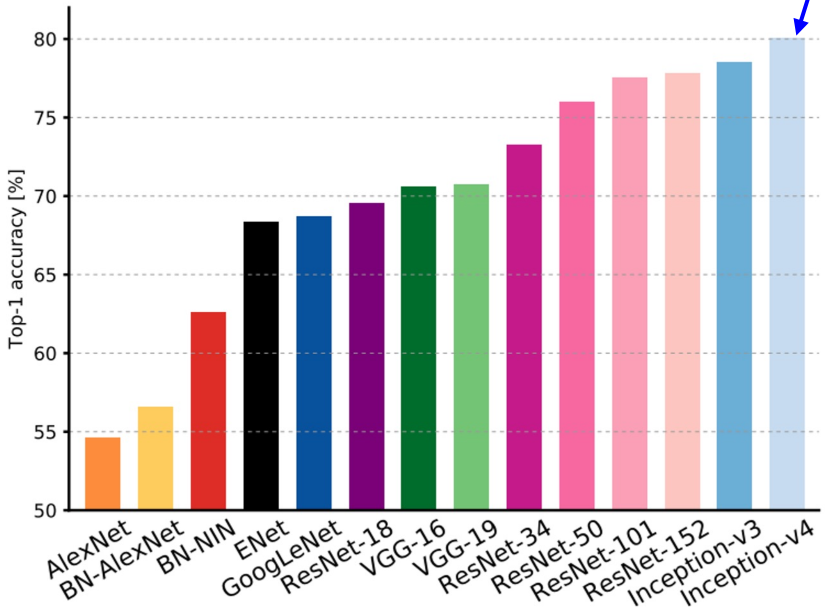


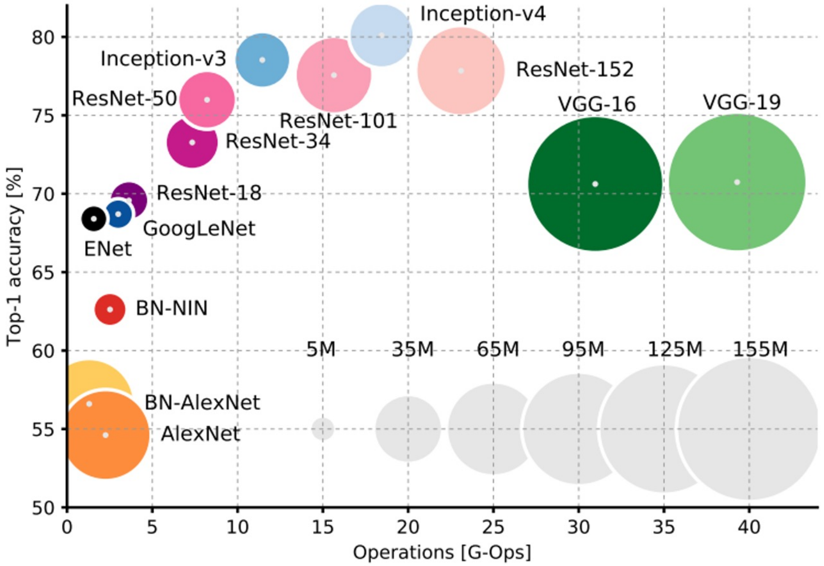
An Analysis of Deep Neural Network Models for Practical Applications, 2017.

Figures copyright Alfredo Canziani, Adam Paszke, Eugenio Culurciello, 2017. Reproduced with permission.

Comparing complexity...

Inception-v4: Resnet + Inception!



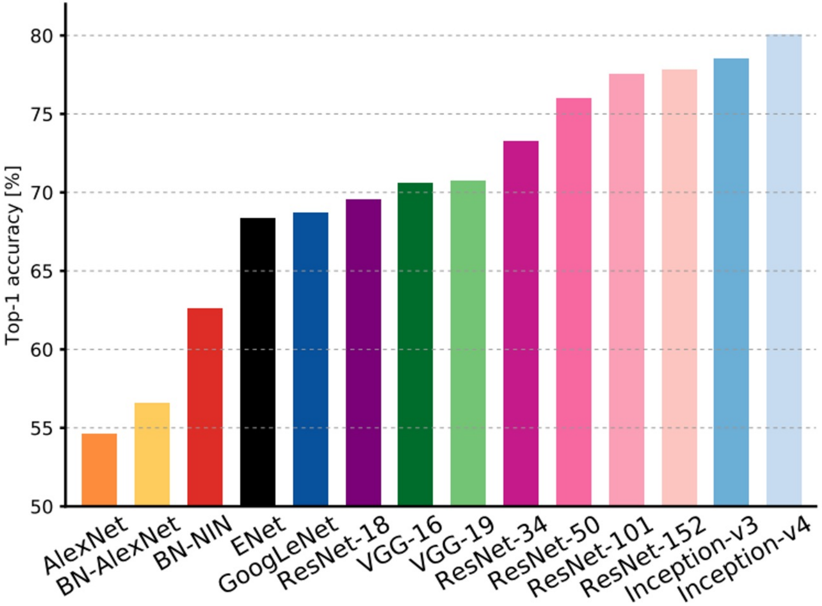


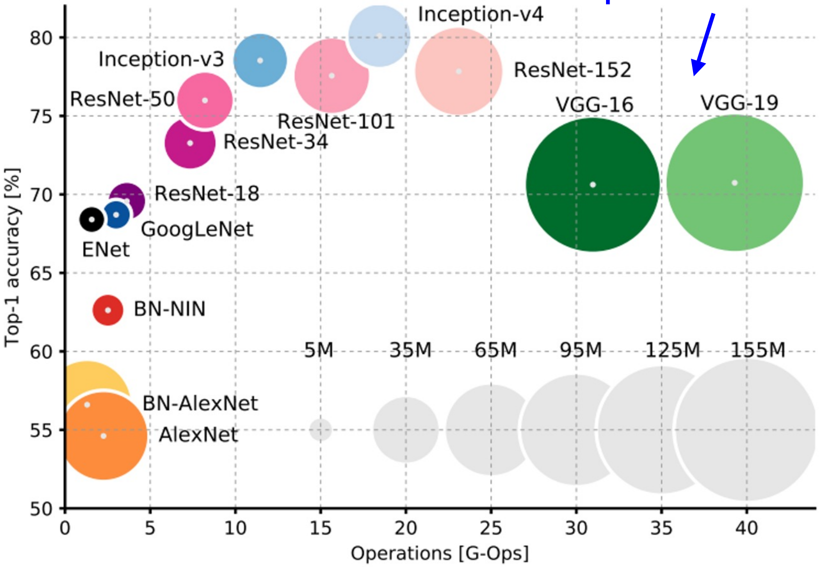
An Analysis of Deep Neural Network Models for Practical Applications, 2017.

Figures copyright Alfredo Canziani, Adam Paszke, Eugenio Culurciello, 2017. Reproduced with permission.

Comparing complexity...

VGG: most parameters, most operations



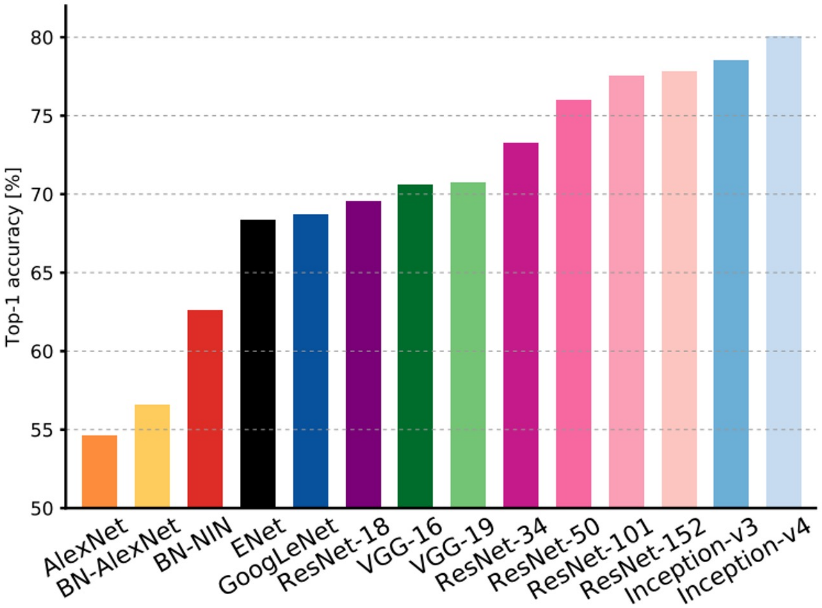


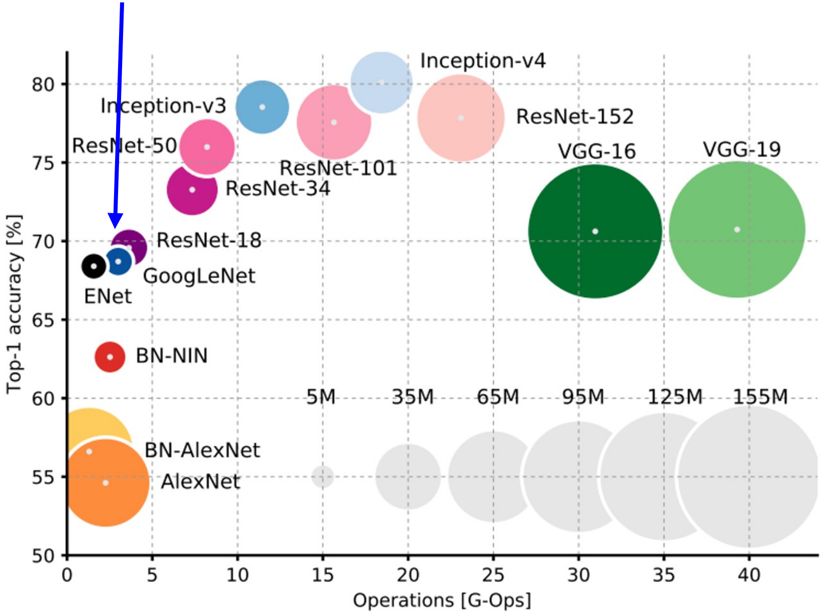
An Analysis of Deep Neural Network Models for Practical Applications, 2017.

Figures copyright Alfredo Canziani, Adam Paszke, Eugenio Culurciello, 2017. Reproduced with permission.

Comparing complexity...

GoogLeNet:
most efficient



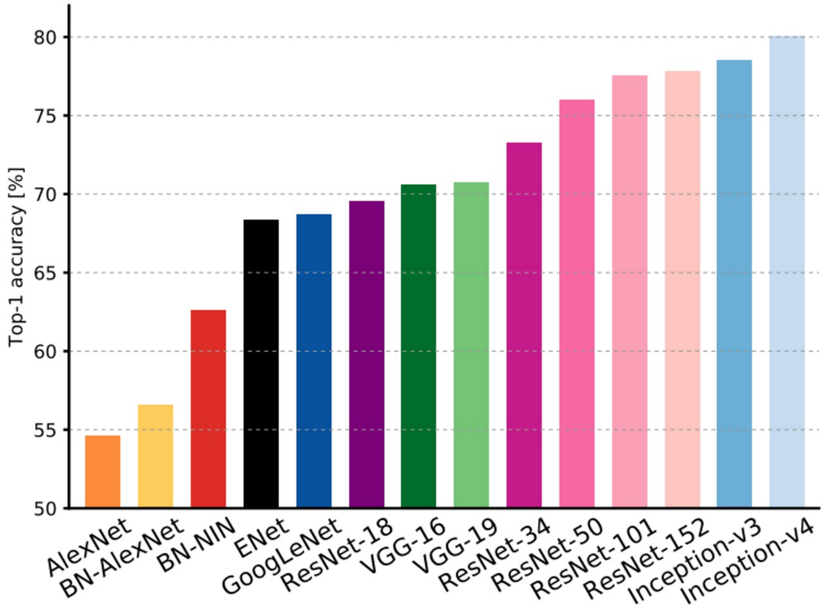


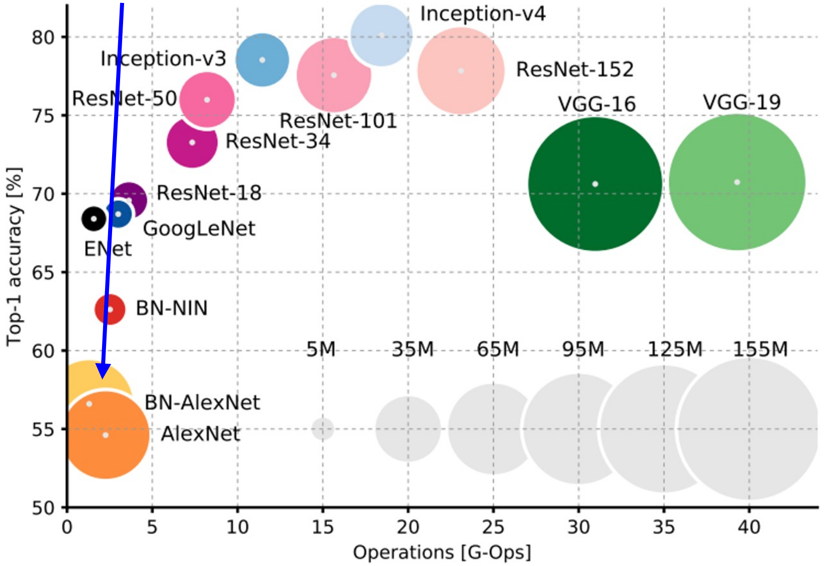
An Analysis of Deep Neural Network Models for Practical Applications, 2017.

Figures copyright Alfredo Canziani, Adam Paszke, Eugenio Culurciello, 2017. Reproduced with permission.

Comparing complexity...

AlexNet:
Smaller compute, still memory heavy, lower accuracy



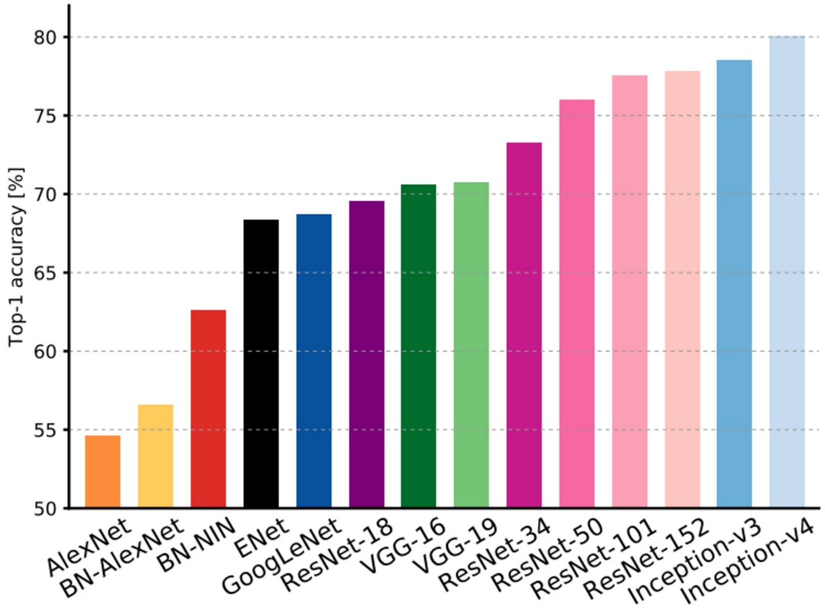


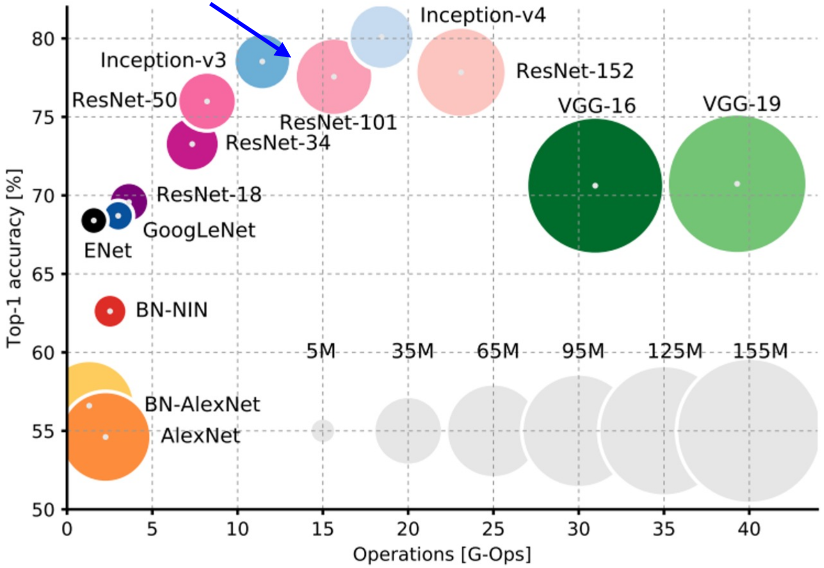
An Analysis of Deep Neural Network Models for Practical Applications, 2017.

Figures copyright Alfredo Canziani, Adam Paszke, Eugenio Culurciello, 2017. Reproduced with permission.

Comparing complexity...

ResNet:
Moderate efficiency depending on model, highest accuracy

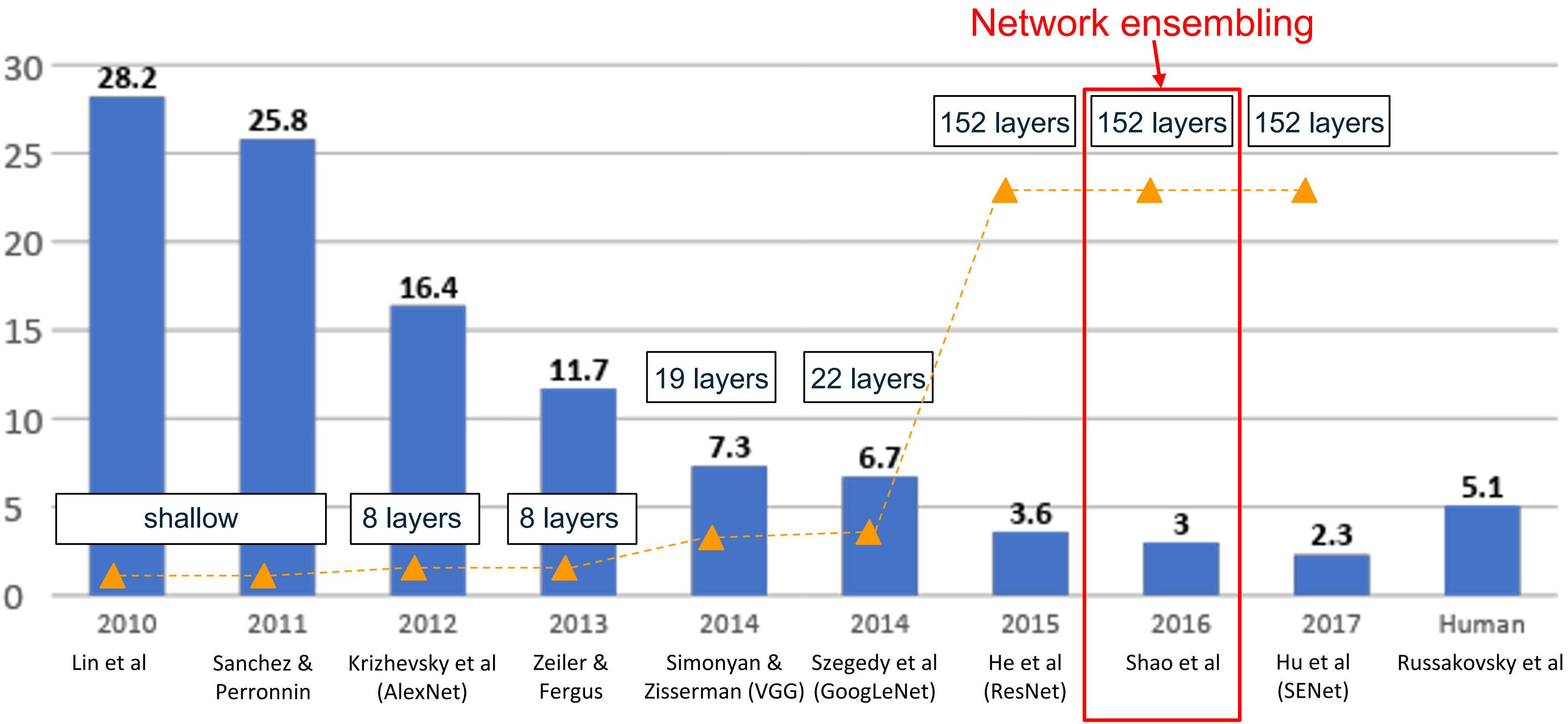




An Analysis of Deep Neural Network Models for Practical Applications, 2017.

Figures copyright Alfredo Canziani, Adam Paszke, Eugenio Culurciello, 2017. Reproduced with permission.

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



Improving ResNets...

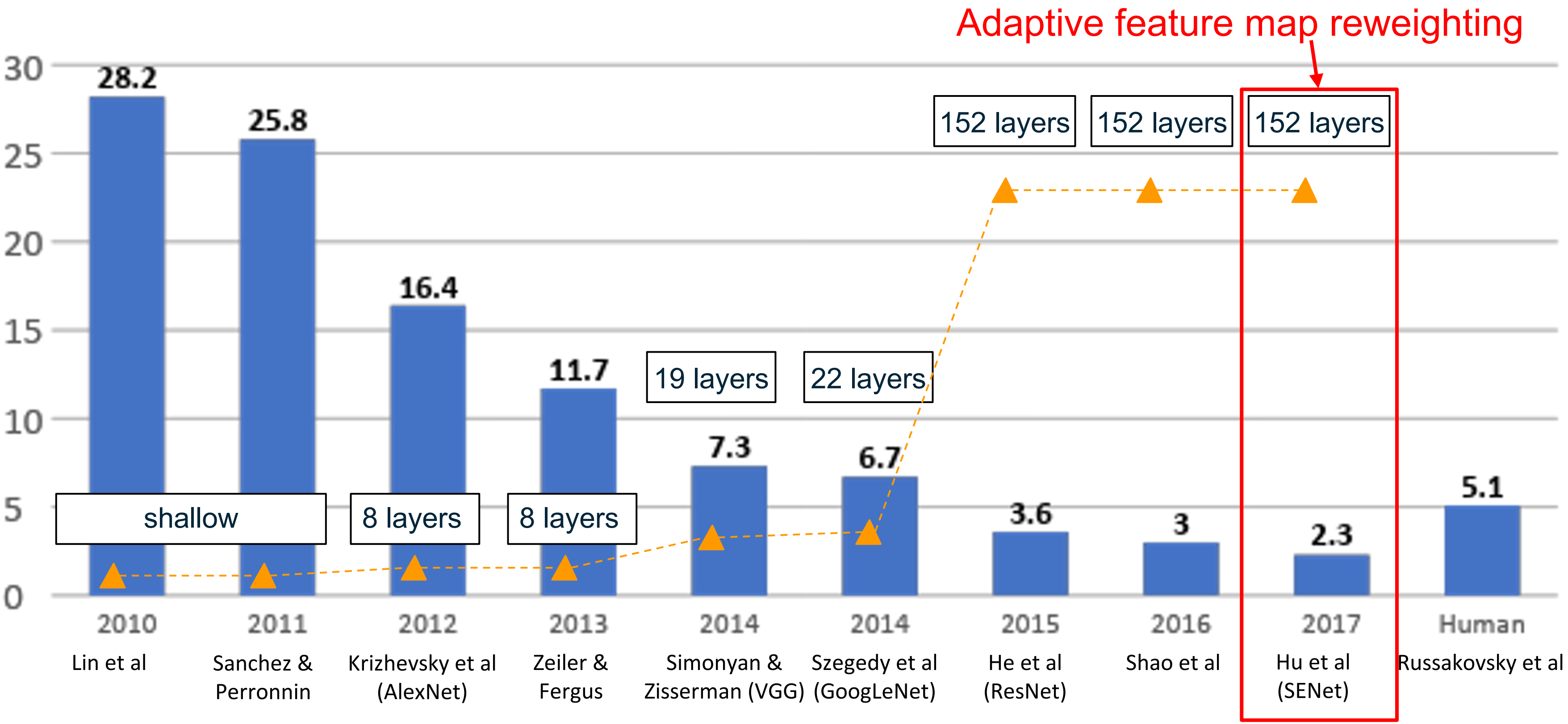
“Good Practices for Deep Feature Fusion”

[Shao et al. 2016]

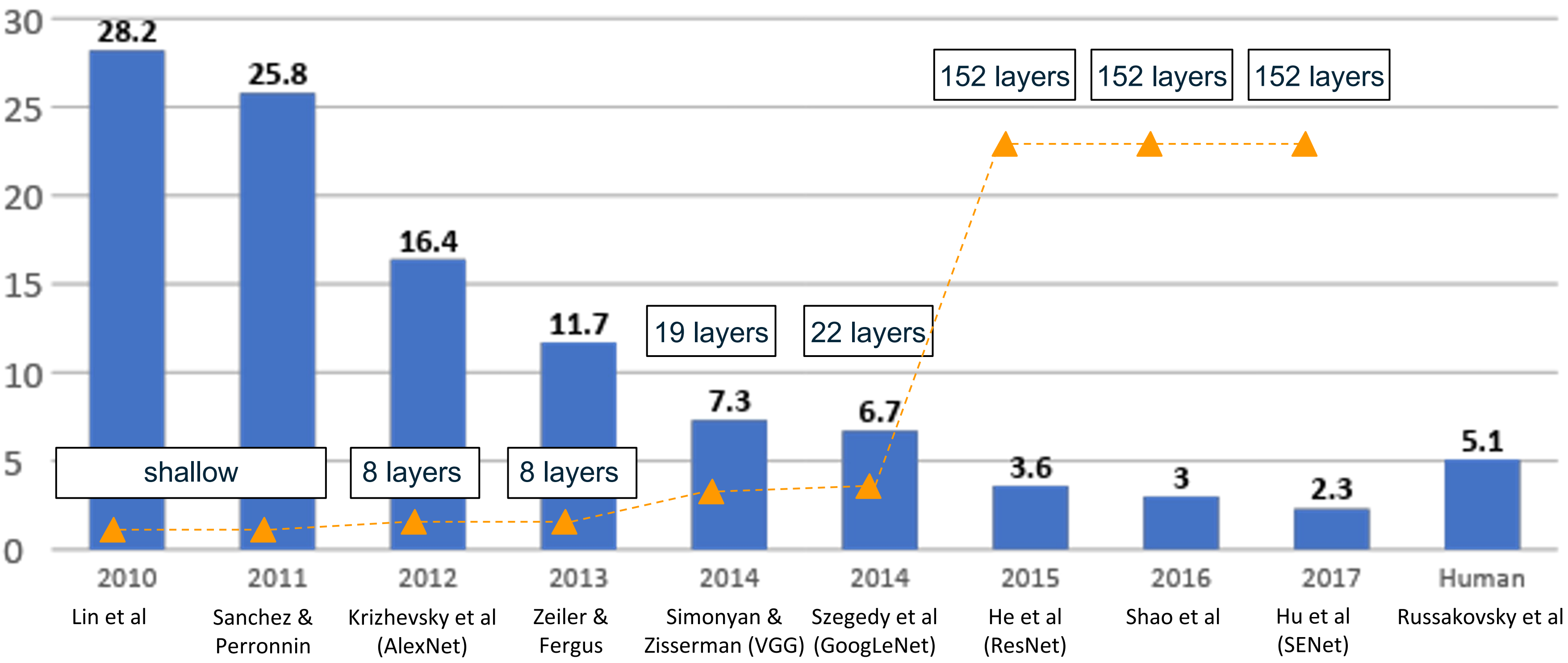
- Multi-scale ensembling of Inception, Inception-Resnet, Resnet, Wide Resnet models
- ILSVRC’16 classification winner

	Inception-v3	Inception-v4	Inception-Resnet-v2	Resnet-200	Wrn-68-3	Fusion（Val.）	Fusion（Test）
Err. (%)	4.20	4.01	3.52	4.26	4.65	2.92 (-0.6)	2.99

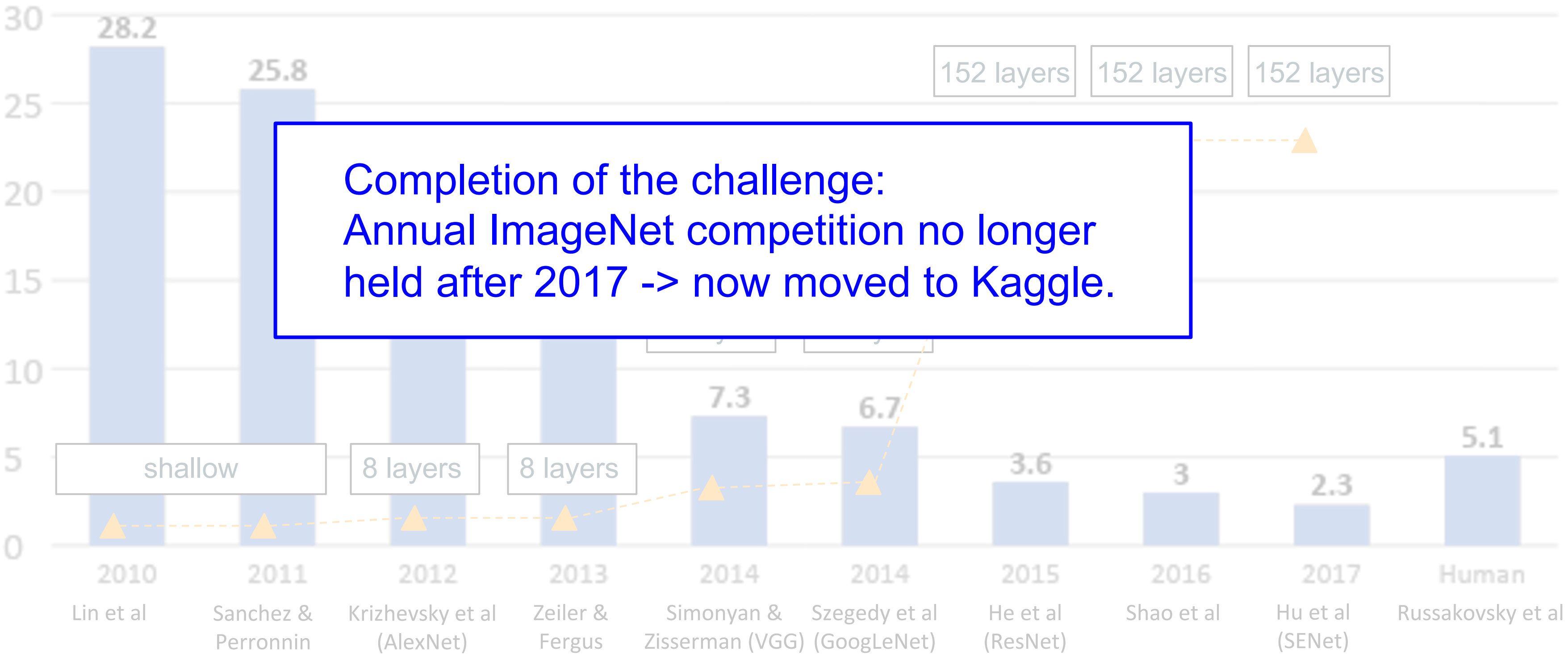
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



Completion of the challenge:
Annual ImageNet competition no longer held after 2017 -> now moved to Kaggle.

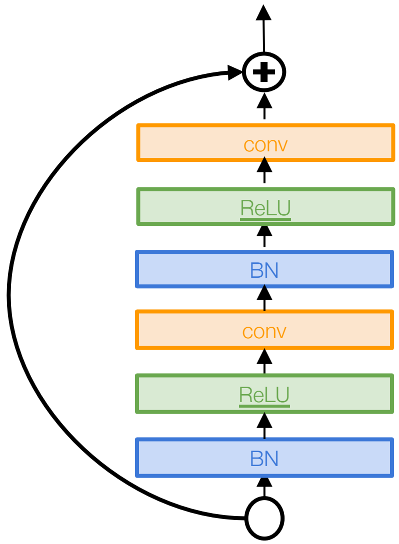
But research into CNN architectures is still flourishing

Improving ResNets...

Identity Mappings in Deep Residual Networks

[He et al. 2016]

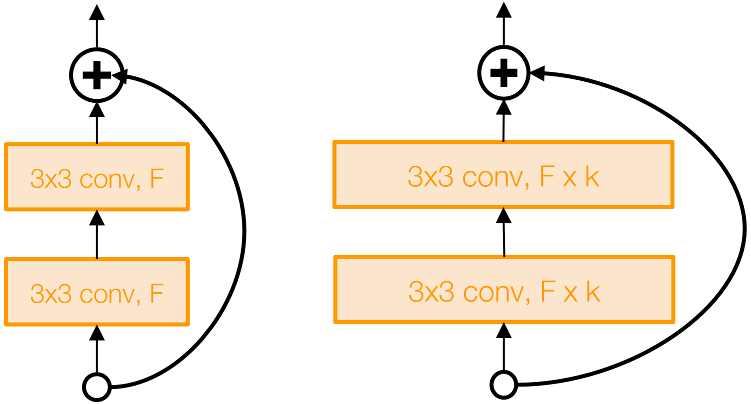
- Improved ResNet block design from creators of ResNet
- Creates a more direct path for propagating information throughout network
- Gives better performance



Wide Residual Networks

[Zagoruyko et al. 2016]

- Argues that residuals are the important factor, not depth
- Use wider residual blocks (F x k filters instead of F filters in each layer)
- 50-layer wide ResNet outperforms 152-layer original ResNet
- Increasing width instead of depth more computationally efficient (parallelizable)



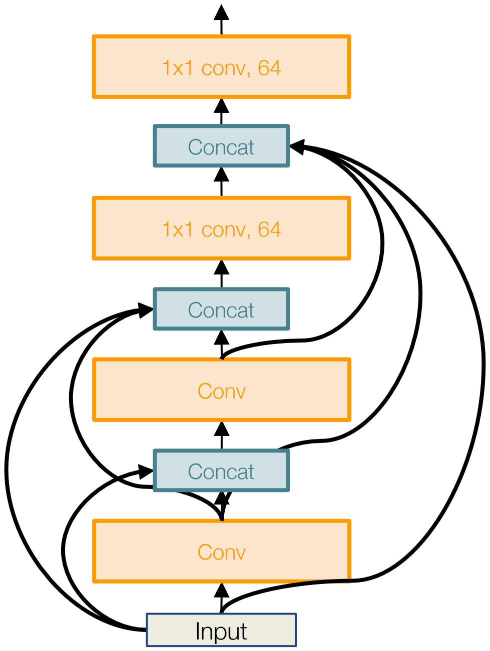
Basic residual block   Wide residual block

Other ideas...

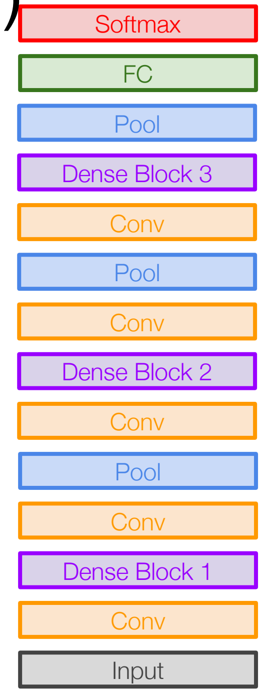
Densely Connected Convolutional Networks (DenseNet)

[Huang et al. 2017]

- Dense blocks where each layer is connected to every other layer through concatenation
- Different way to address vanishing gradient (concat vs. residual) .
- Multi-layer feature aggregation
- Showed that shallow 50-layer network can outperform deeper 152 layer ResNet



Dense Block

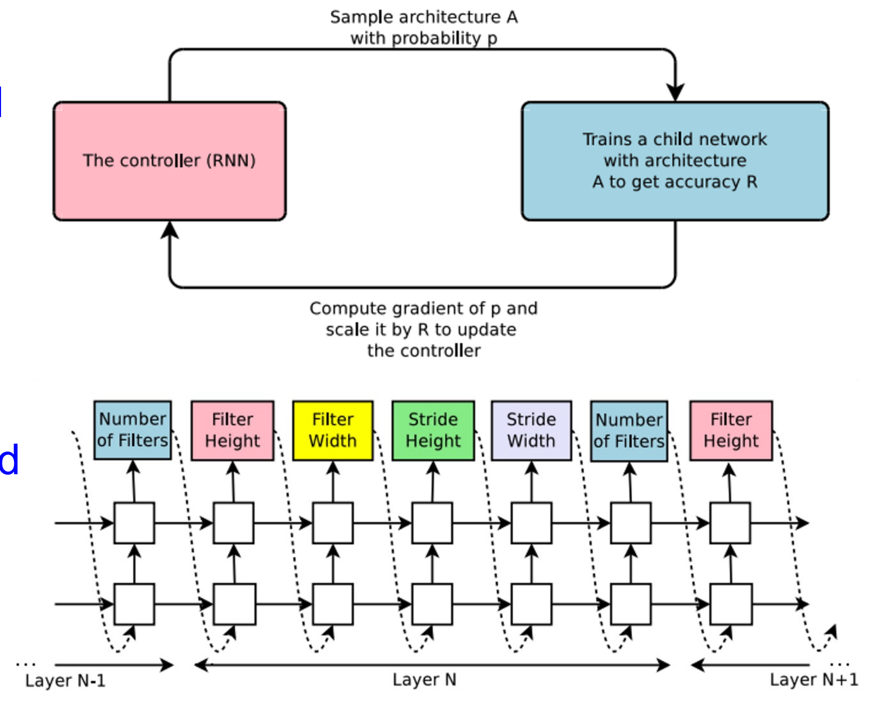


Learning to search for network architectures...

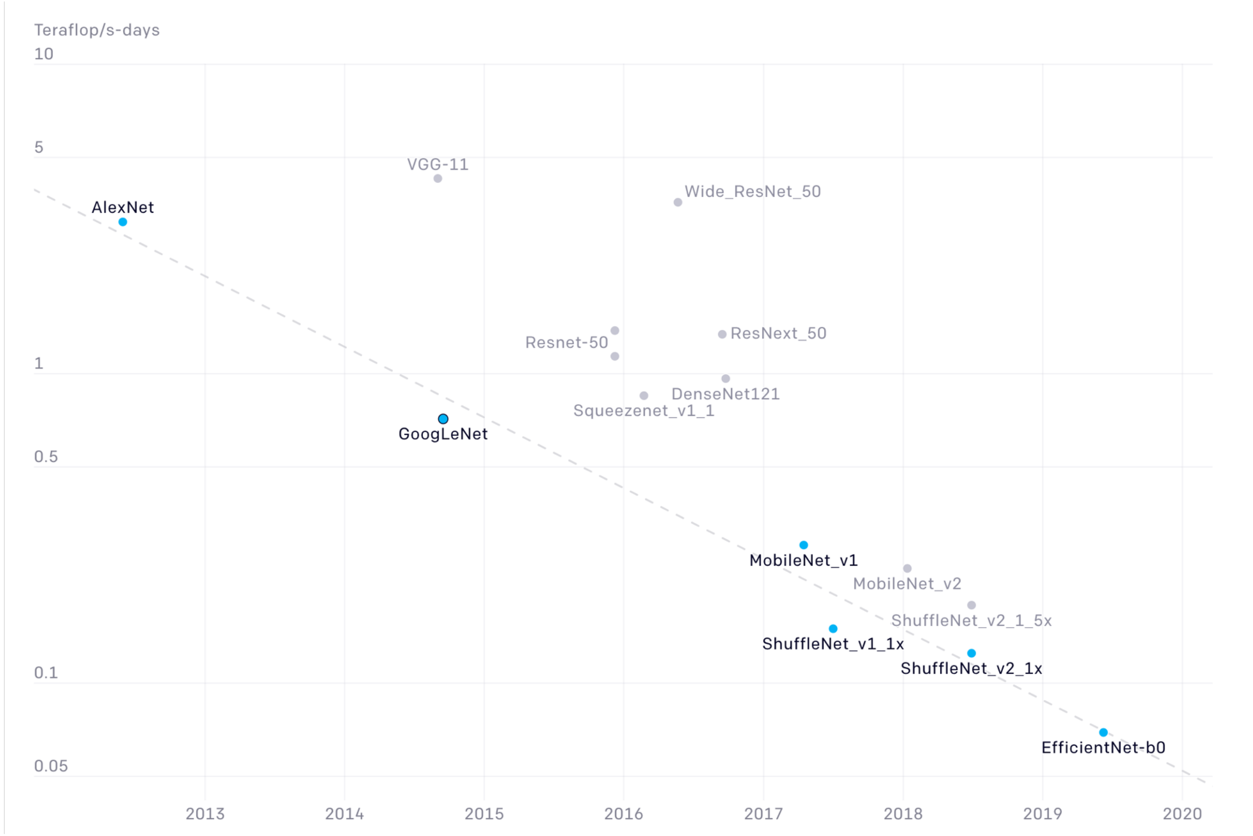
Neural Architecture Search with Reinforcement Learning (NAS)

[Zoph et al. 2016]

- "Controller" network that learns to design a good network architecture (output a string corresponding to network design)
- Iterate:
    1) Sample an architecture from search space
    2) Train the architecture to get a "reward" R corresponding to accuracy
    3) Compute gradient of sample probability, and scale by R to perform controller parameter update (i.e. increase likelihood of good architecture being sampled, decrease likelihood of bad architecture)



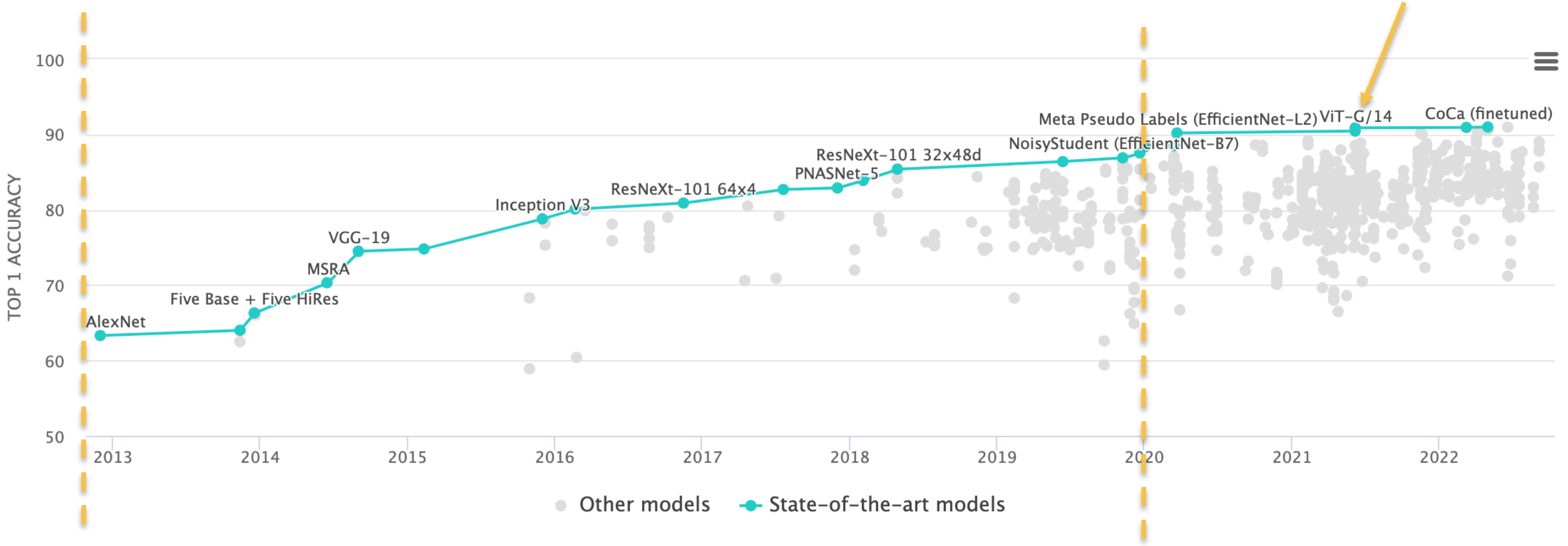
Amount of compute required to reach “AlexNet performance”



https://openai.com/blog/ai-and-efficiency/

This Lecture

Transformer (later this sem.)



https://paperswithcode.com/sota/image-classification-on-imagenet

What we have learned so far …

Deep Neural Networks:

- What they are (composite parametric, non-linear functions)
- Where they come from (biological inspiration, brief history of ANN)
- How they are optimized, in principle (analytical gradient via computational graphs, backpropagation)
- What they look like in practice (Deep ConvNets)

Next few lectures:

Training Deep Neural Networks

- Details of the non-linear activation functions
- Data normalization
- Weight Initialization
- Batch Normalization
- Regularization
- Advanced Optimization
- Data Augmentation
- Transfer learning
- Hyperparameter Tuning
- Model Ensemble

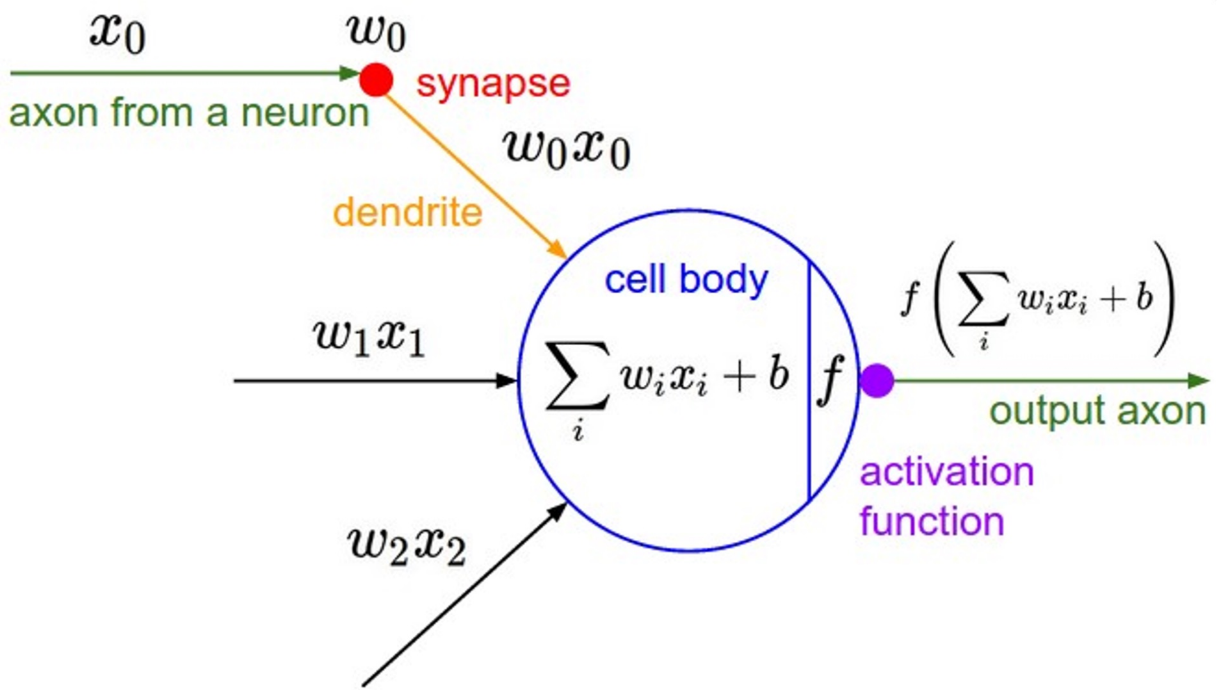
This lecture:

Training Deep Neural Networks
- Details of the non-linear activation functions
- Data normalization
- Weight Initialization
- Batch Normalization
- Regularization
- Advanced Optimization
- Data Augmentation
- Transfer learning
- Hyperparameter Tuning
- Model Ensemble

Today: Training Deep NNs (Part 1)

- Details of the non-linear activation functions
- Data normalization
- Weight Initialization

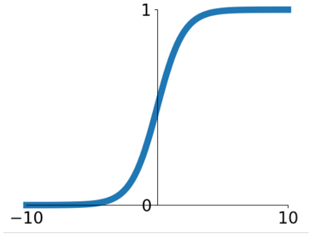
Activation Functions



Activation Functions

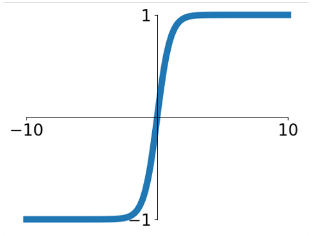
Sigmoid

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



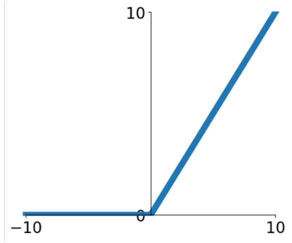
tanh

$$\tanh(x)$$



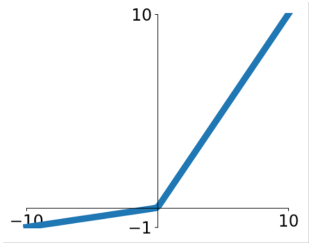
ReLU

$$\max(0, x)$$



Leaky ReLU

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

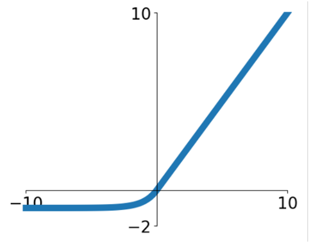


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

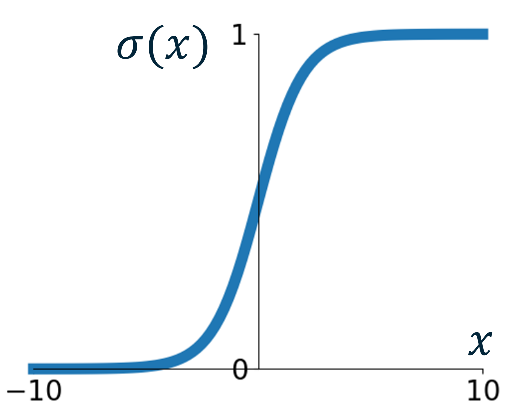
ELU

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



Activation Functions

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

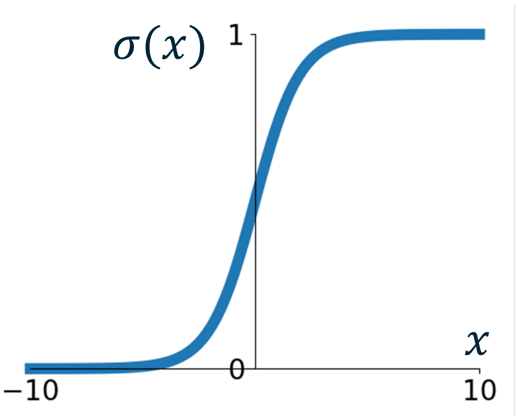


Sigmoid

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

Activation Functions

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

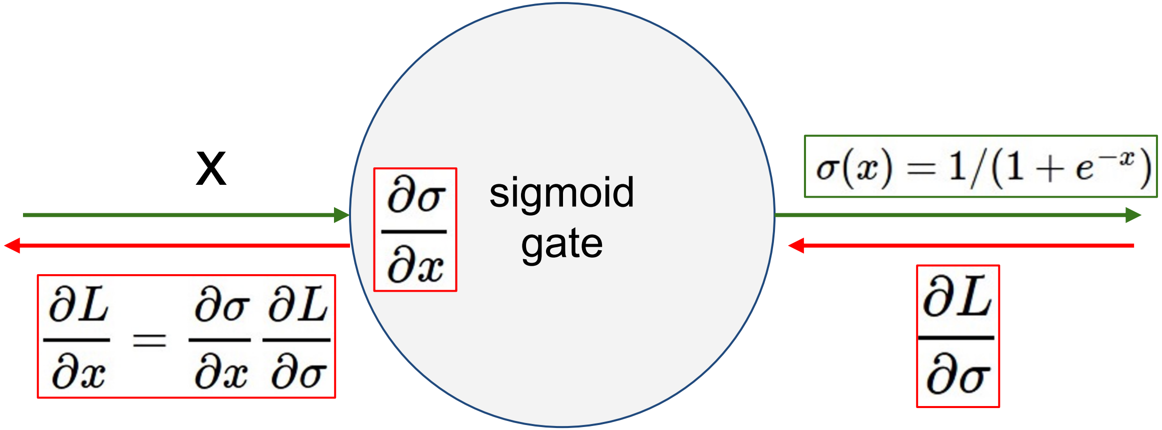


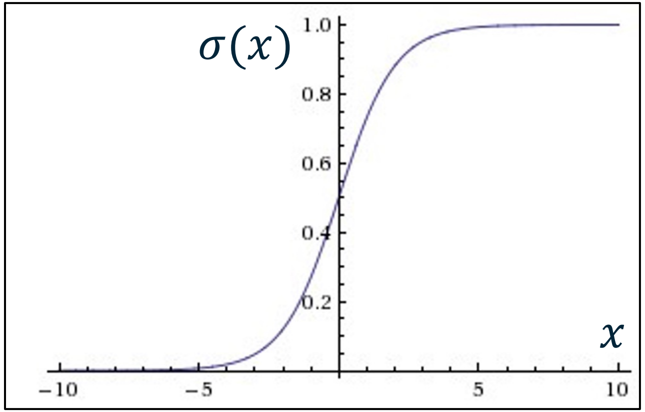
Sigmoid

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

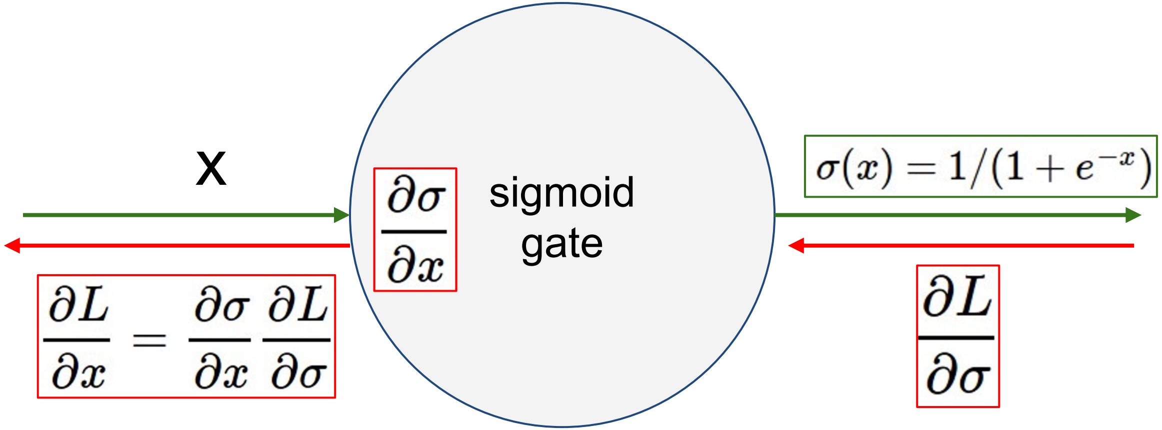
Problems:

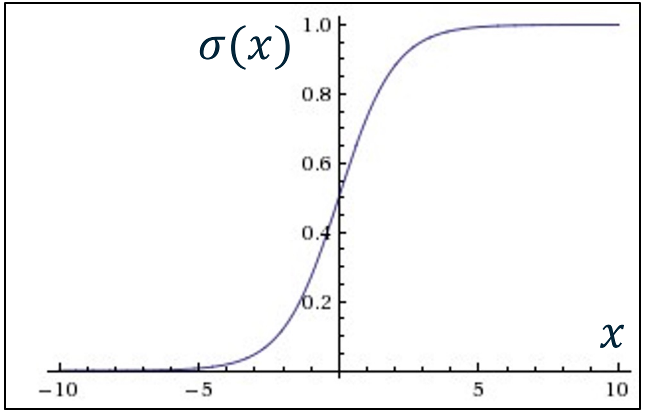
1. Saturated neurons “kill” the gradients





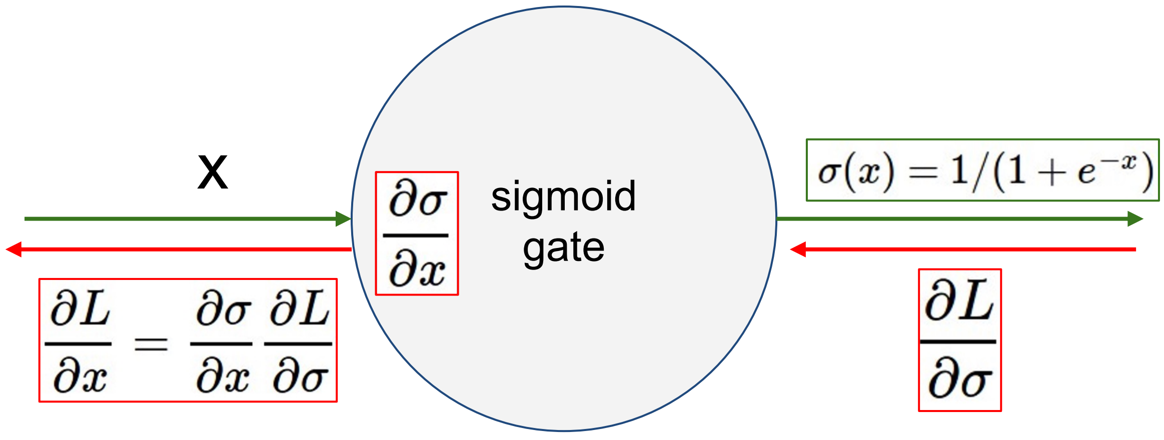
$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)\,(1 - \sigma(x))$$

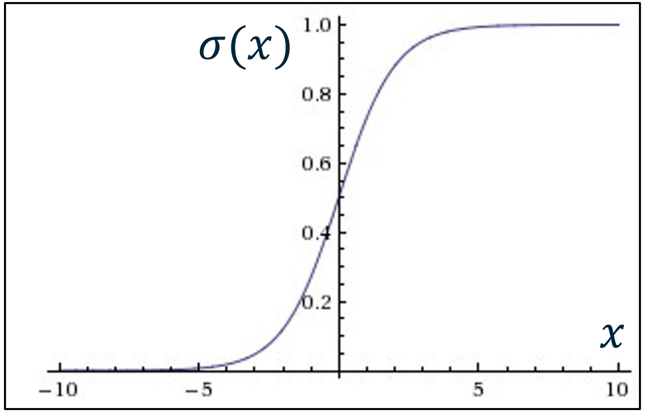




What happens to $\frac{\partial \sigma}{\partial x}$ when $x = -10$?

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)\,(1 - \sigma(x)\,)$$



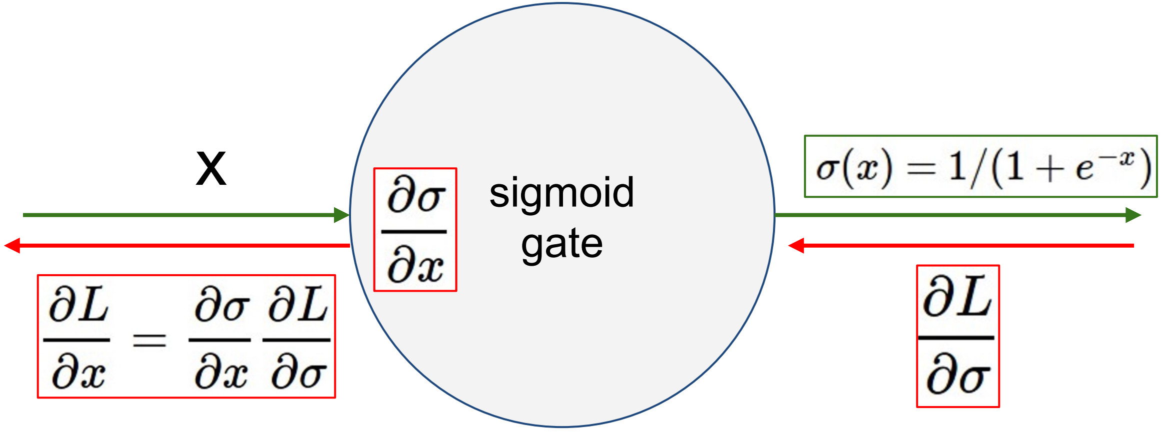


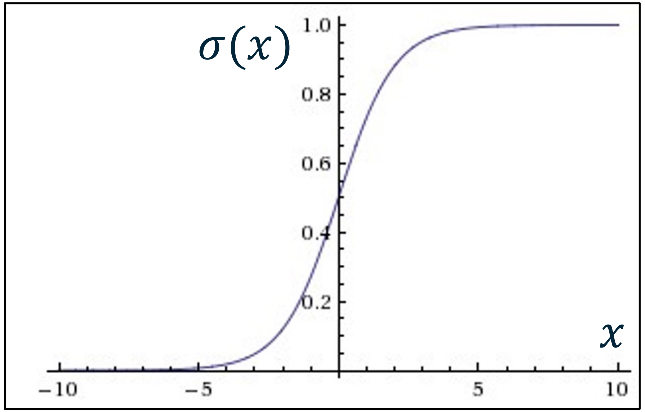
$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)\,(1 - \sigma(x)\,)$$

What happens to $\frac{\partial \sigma}{\partial x}$ when $x = -10$?

$$\sigma(x) = \sim 0$$

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)\,(1 - \sigma(x)\,) = 0(1-0) = 0$$

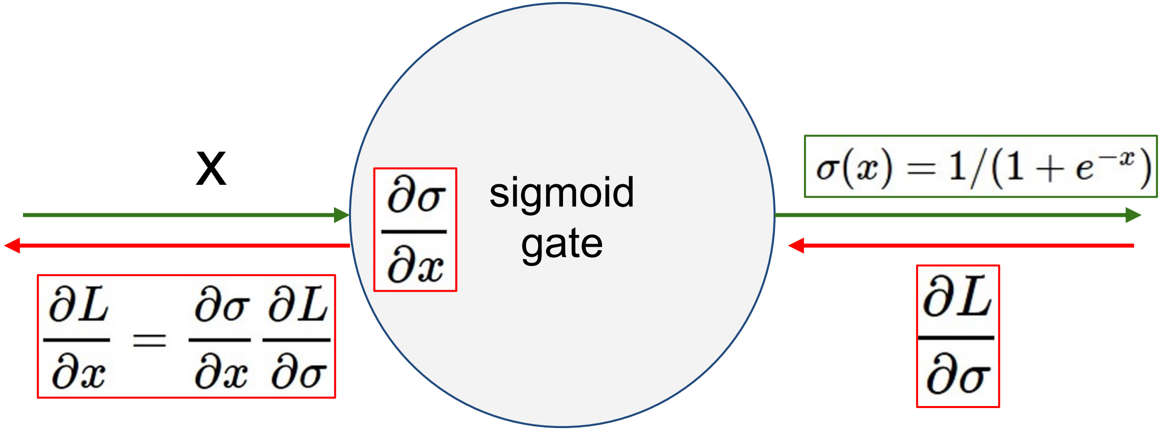


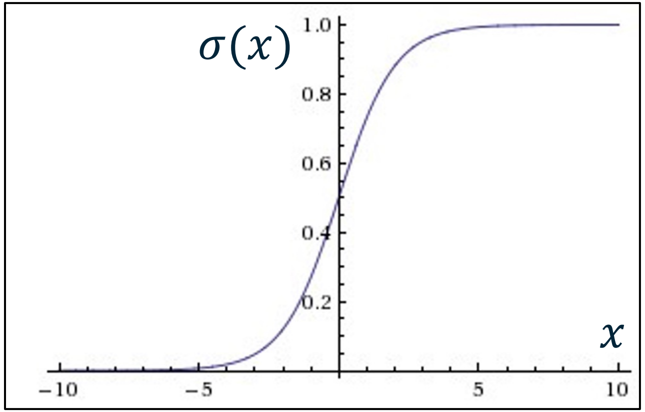


What happens when x = -10?

What happens when x = 10?

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)\,(1 - \sigma(x)\,)$$



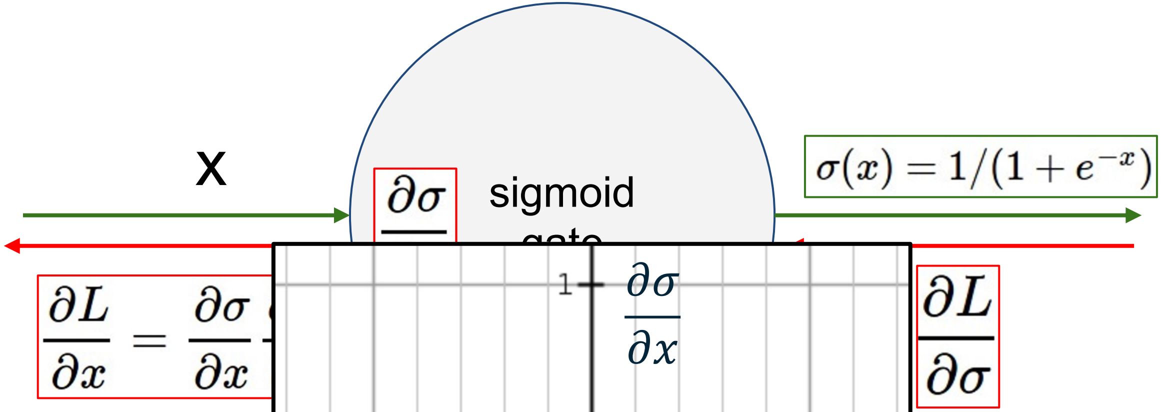


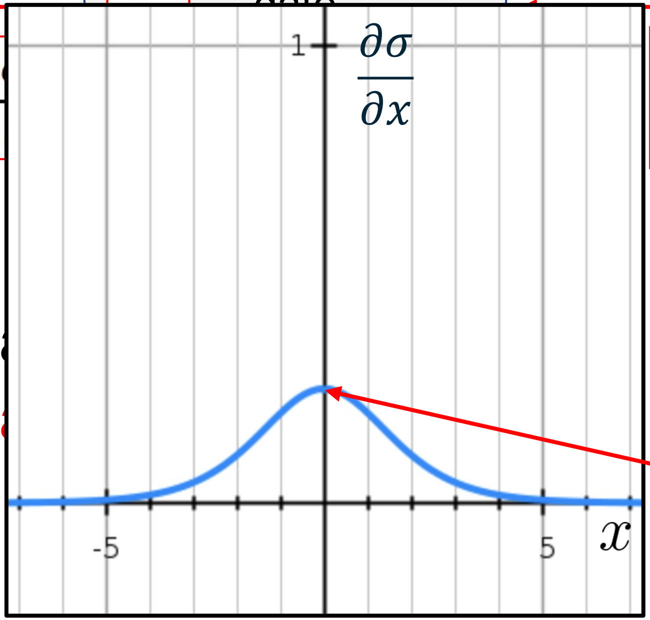
What happens when x = -10?

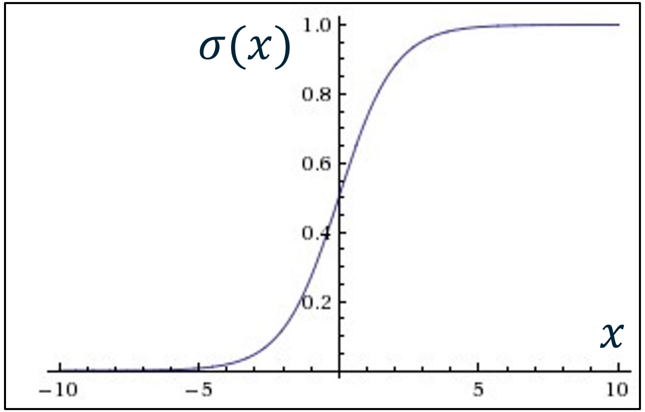
What happens when x = 10?

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)\,(1 - \sigma(x))$$

$$\sigma(x) = \sim 1 \qquad \frac{\partial \sigma(x)}{\partial x} = \sigma(x)\,(1 - \sigma(x)) = 1(1-1) = 0$$





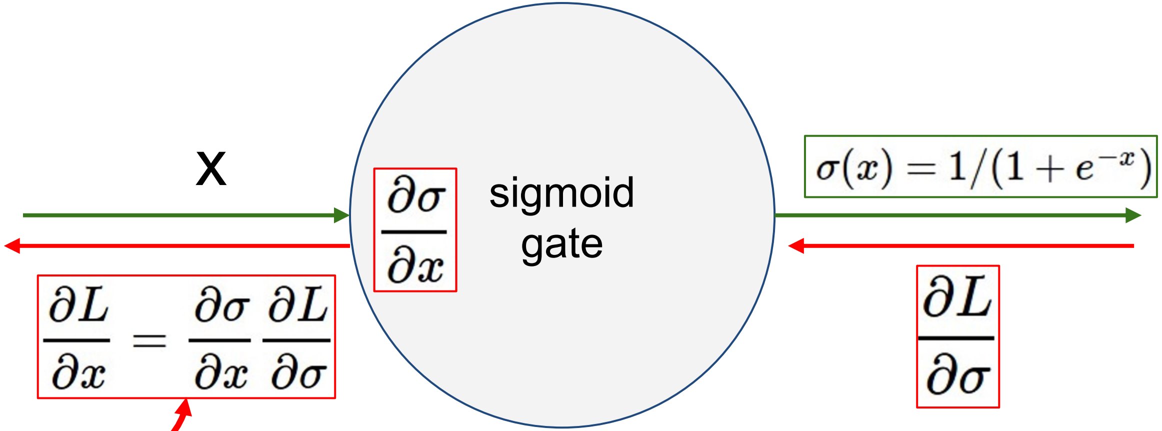


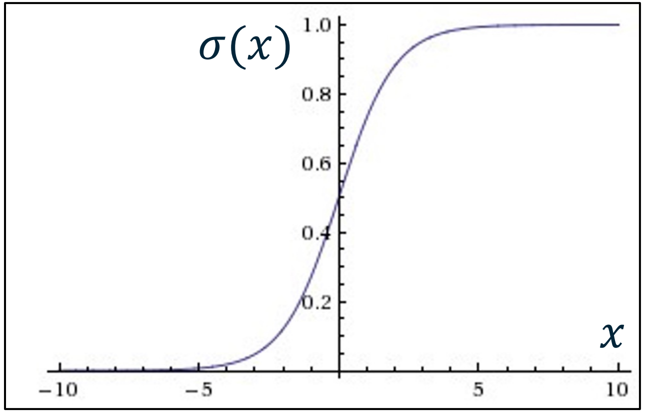
What h

What h

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)\,(1 - \sigma(x)\,)$$

Non-zero but small (~0.269):
still problematic, causes vanishing gradient





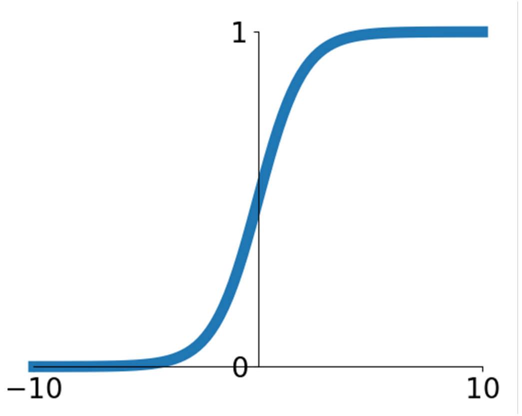
$$\frac{\partial\sigma(x)}{\partial x} = \sigma(x)\,(1-\sigma(x))$$

Why is this a problem?

If all the gradients flowing back is small, the weights will change slowly / never change (aka "Vanishing Gradient")

Activation Functions

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



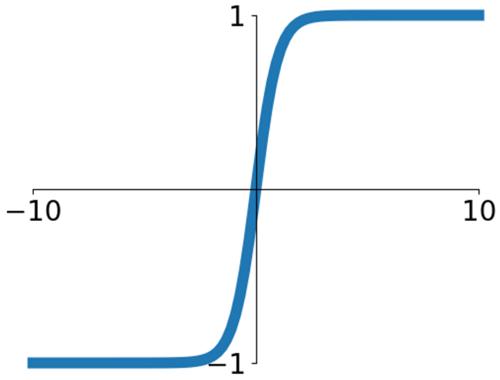
Sigmoid

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

Problems:

1. **Saturated neurons “kill” the gradients**
2. exp() is a bit compute expensive

Activation Functions

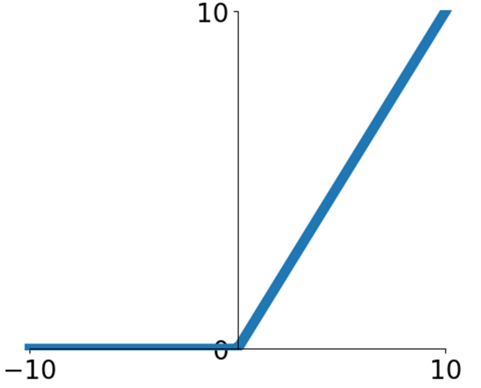


tanh(x)

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

[LeCun et al., 1991]

Activation Functions

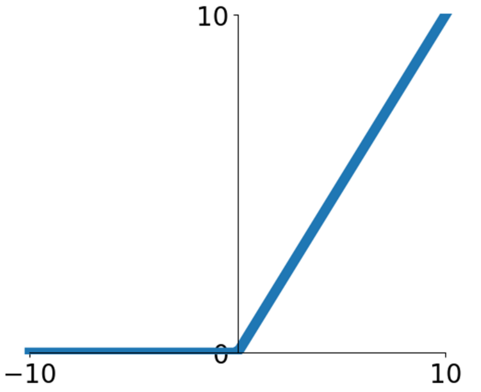


ReLU
(Rectified Linear Unit)

- Computes **f(x) = max(0,x)**

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

Activation Functions



ReLU
(Rectified Linear Unit)

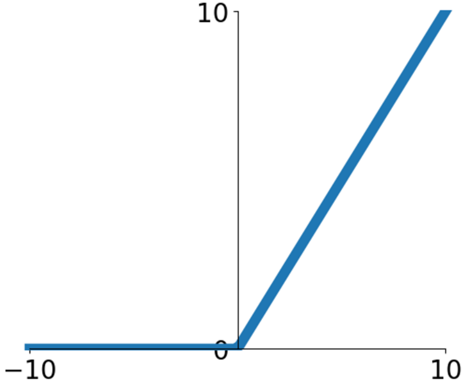
- Computes **f(x) = max(0,x)**

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

- An annoyance:

hint: what is the gradient when x < 0?

Activation Functions



ReLU
(Rectified Linear Unit)

- Computes **f(x) = max(0,x)**

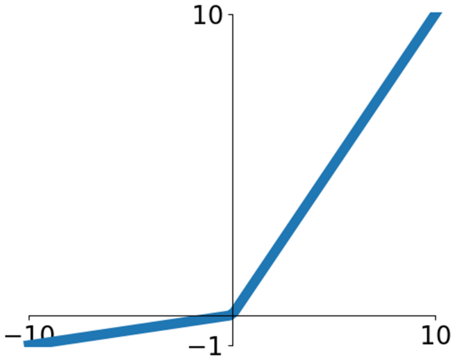
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)

- An annoyance:

hint: what is the gradient when x < 0?
Always 0 -> no update in weights -> stays 0, A.K.A. “dead ReLU”

Activation Functions

[Mass et al., 2013]
[He et al., 2015]



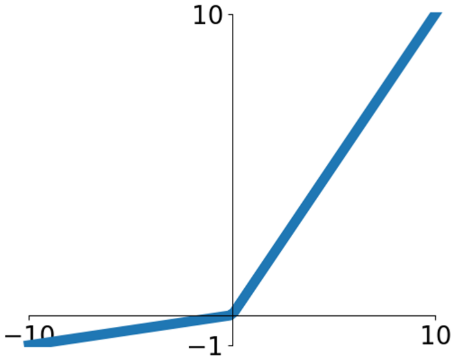
- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- **will not "die".**

Leaky ReLU

$$f(x) = \max(0.01x, x)$$

Activation Functions

[Mass et al., 2013]
[He et al., 2015]



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

Leaky ReLU

$$f(x) = \max(0.01x, x)$$

Parametric Rectifier (PReLU)

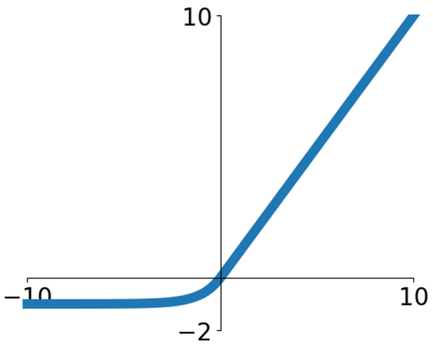
$$f(x) = \max(\alpha x, x)$$

backprop into \alpha
(parameter)

Activation Functions

[Clevert et al., 2015]

Exponential Linear Units (ELU)



$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\left(\exp(x) - 1\right) & \text{if } x \le 0 \end{cases}$$

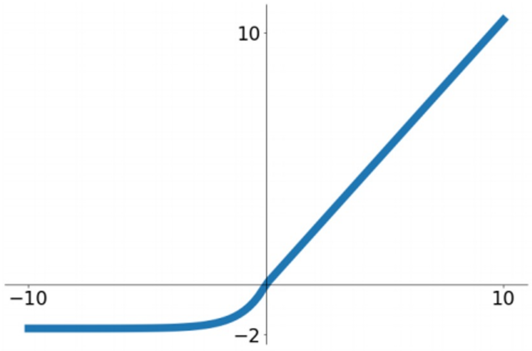
(Alpha default = 1)

- All benefits of ReLU
- Negative saturation encodes presence of features (all goes to $-\alpha$), not magnitude
- Similar in backprop (αe^{x} when x is negative)
- Compared with Leaky ReLU: smooth gradient at 0 (no kink), better optimization landscape

Activation Functions

[Klambauer et al. ICLR 2017]

Scaled Exponential Linear Units (SELU)



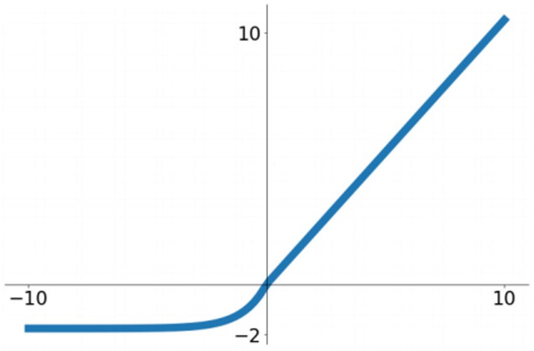
$$f(x) = \begin{cases} \lambda x & \text{if } x > 0 \\ \lambda\alpha(e^x - 1) & \text{otherwise} \end{cases}$$

- Scaled versionof ELU that works better for deep networks
- “Self-normalizing” property: under certain condition, the output of a feedforward network stays around zero-mean and unit variance

Activation Functions

[Klambauer et al. ICLR 2017]

Scaled Exponential Linear Units (SELU)



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

α = 1.6732632423543772848170429916717
λ = 1.0507009873554804934193349852946

- Scaled versionof ELU that works better for deep networks
- "Self-normalizing" property: under certain condition, the output of a feedforward network stays around zero-mean and unit variance

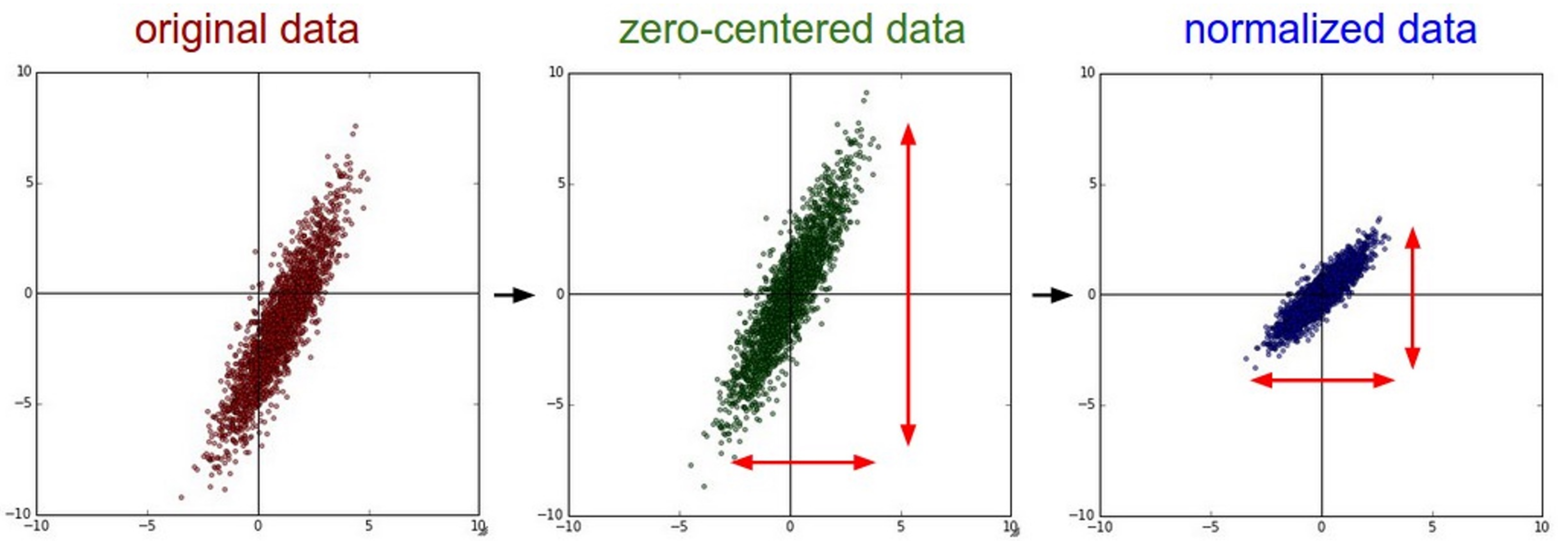
(Klambauer et al, Self-Normalizing Neural Networks, ICLR 2017)

TLDR: In practice:

- Many possible choices beyond what we've talked here, but …
- Use ReLU. Be careful with your learning rates
- Try out Leaky ReLU / ELU / SELU / GELU
  - To squeeze out some marginal gains
- Don't use sigmoid or tanh

Data Preprocessing

Data Preprocessing



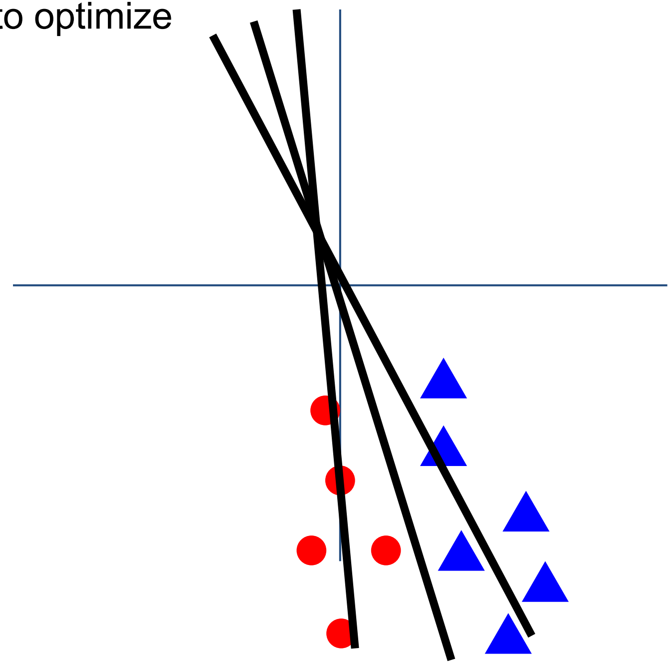

```
X -= np.mean(X, axis = 0)
```

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

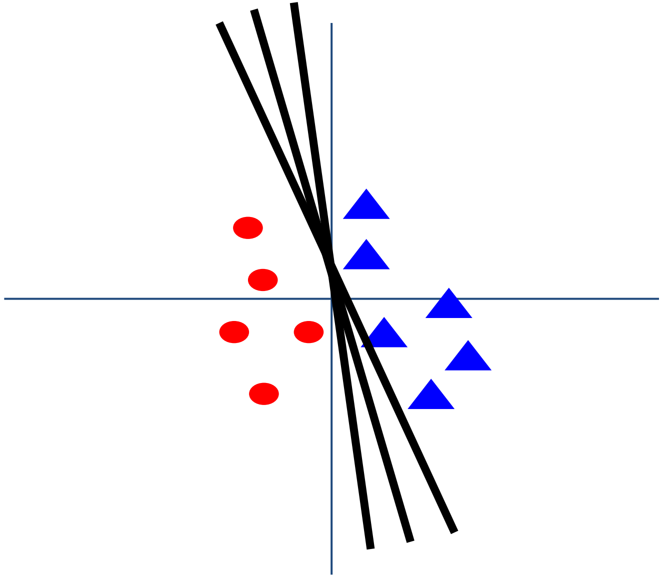
(Assume X [NxD] is data matrix, each example in a row)

Data Preprocessing: example in linear classifier

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

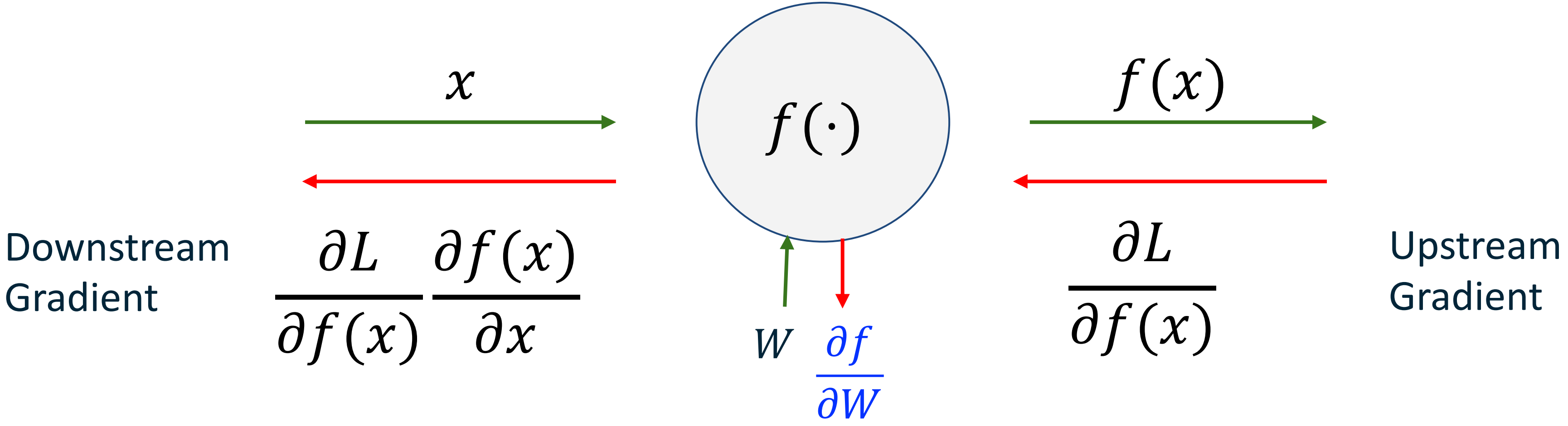


After normalization: less sensitive to small changes in weights; easier to optimize



Many different reasons why we might want to normalize the input!

Another example: Input magnitude affects gradient magnitude

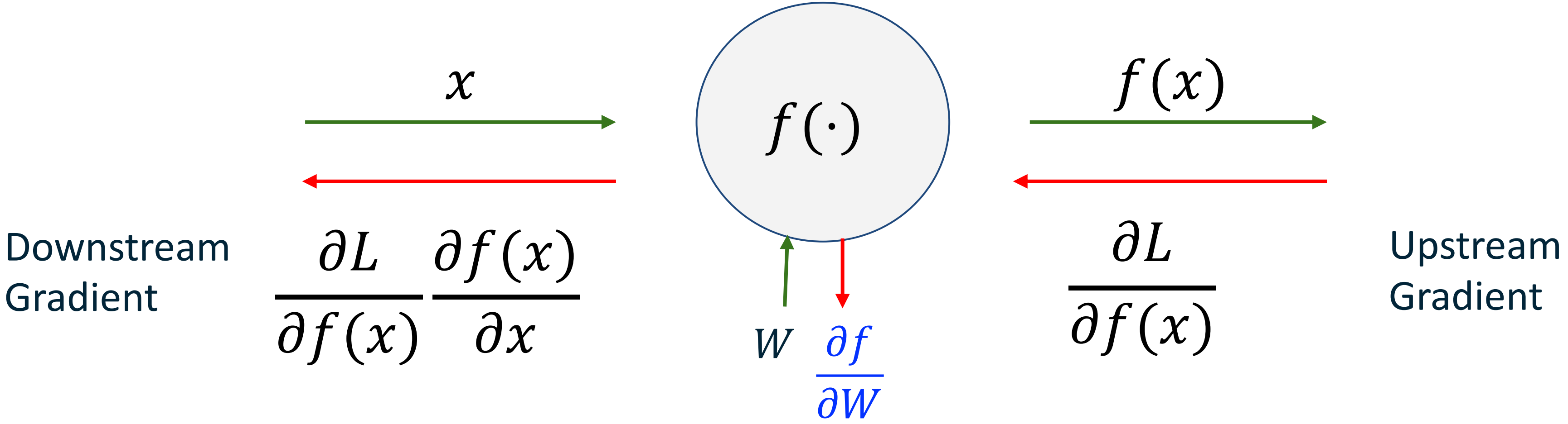


$$f(x) = W^T x + b$$

$$\frac{\partial f}{\partial W} = ?$$

Many different reasons why we might want to normalize the input!

Another example: Input magnitude affects gradient magnitude



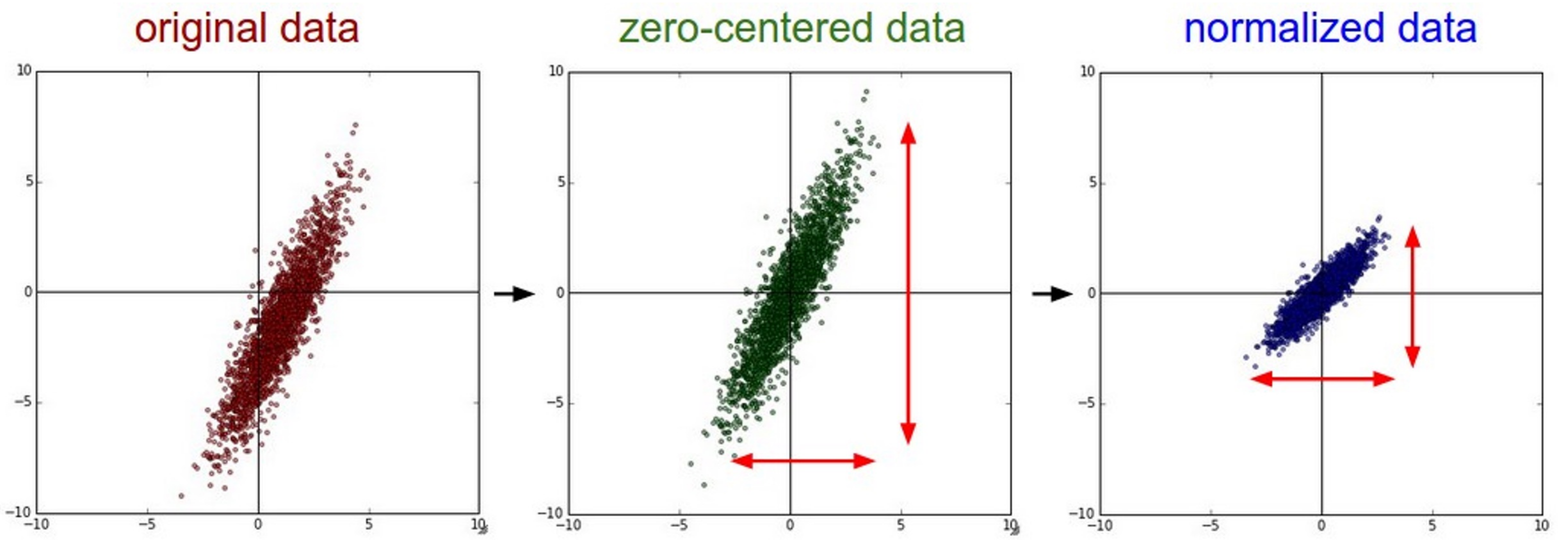
$$f(x) = W^T x + b$$

$$\frac{\partial f}{\partial W} = x$$

Magnitude of input x directly affects the gradients to weights $\frac{\partial f}{\partial W}$!

Data Preprocessing

Gaussian normalization is very commonly used

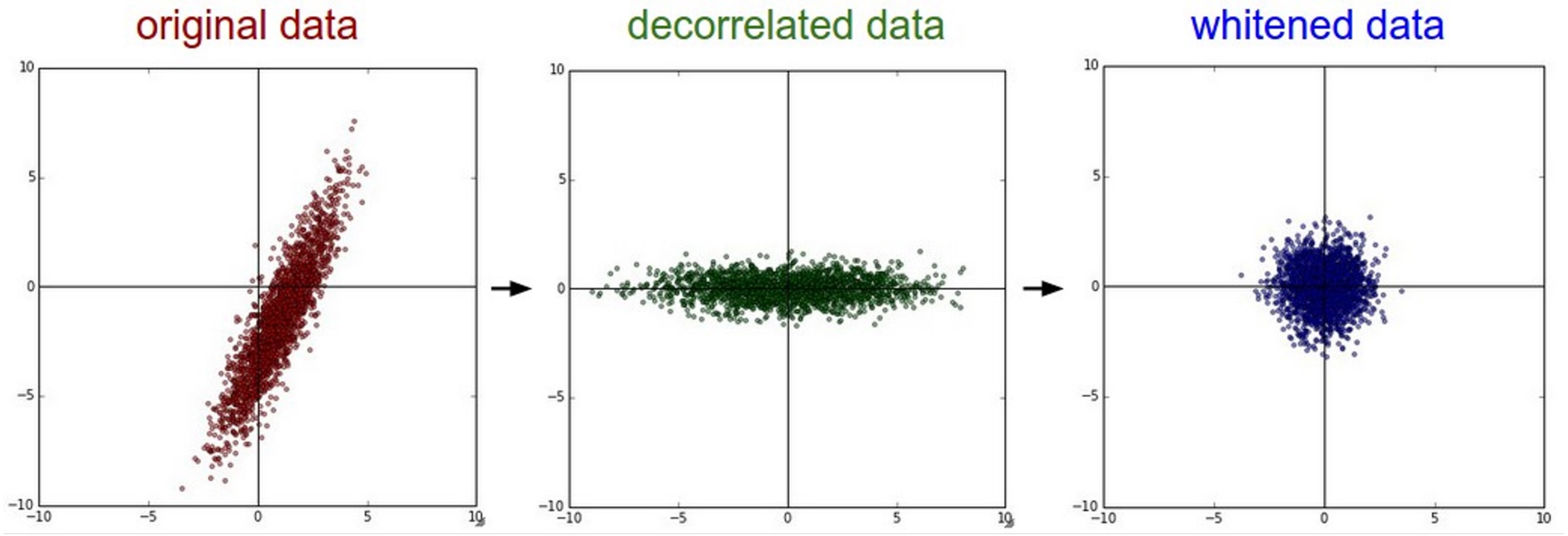



```
X -= np.mean(X, axis = 0)
```

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

Data Preprocessing

In practice, you could also **PCA** and **Whitening** of the data



(data has diagonal covariance matrix)

(covariance matrix is the identity matrix)

Examples: images

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

- Subtract the per-pixel mean(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)

Examples: other domains

- **Natural language processing:** Normalize word embeddings like Word2Vec or GloVe vectors so that they have a unit norm
- **Graph Neural Networks (GNN):** the feature vector of a node might be scaled by the inverse of its degree or the square root of its degree.
- **Audio data**: Spectral normalize waveforms to ensure that the frequency components are on a similar scale.
- **Reinforcement learning**: reward can be normalized to stabilize learning.

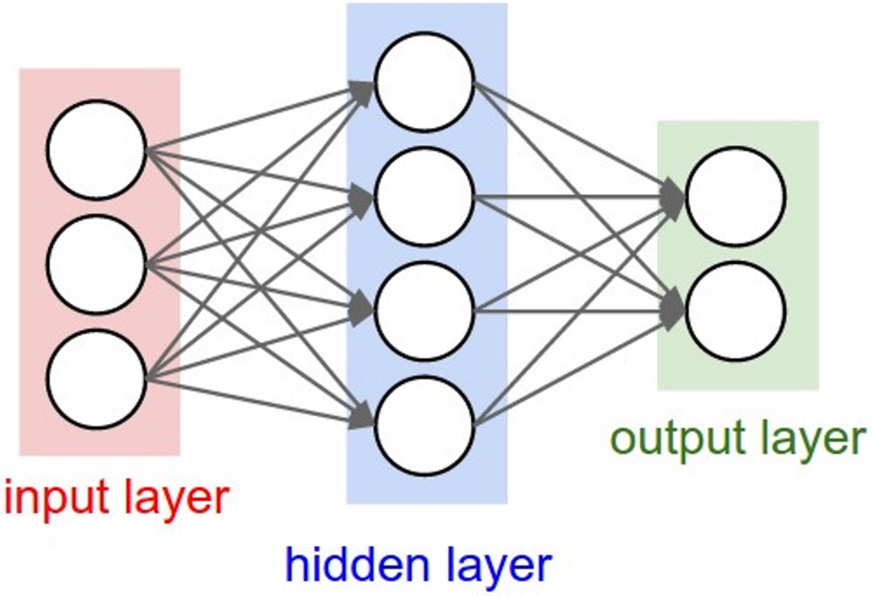
Next time:

Training Deep Neural Networks

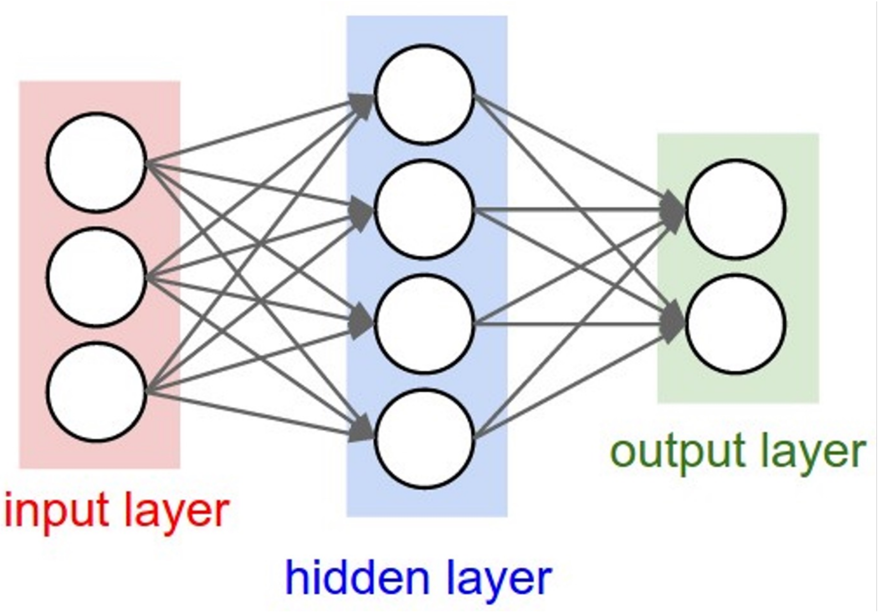
- Details of the non-linear activation functions
- Data normalization
- Weight Initialization
- Batch Normalization
- Advanced Optimization
- Regularization
- Data Augmentation
- Transfer learning
- Hyperparameter Tuning
- Model Ensemble

Weight Initialization

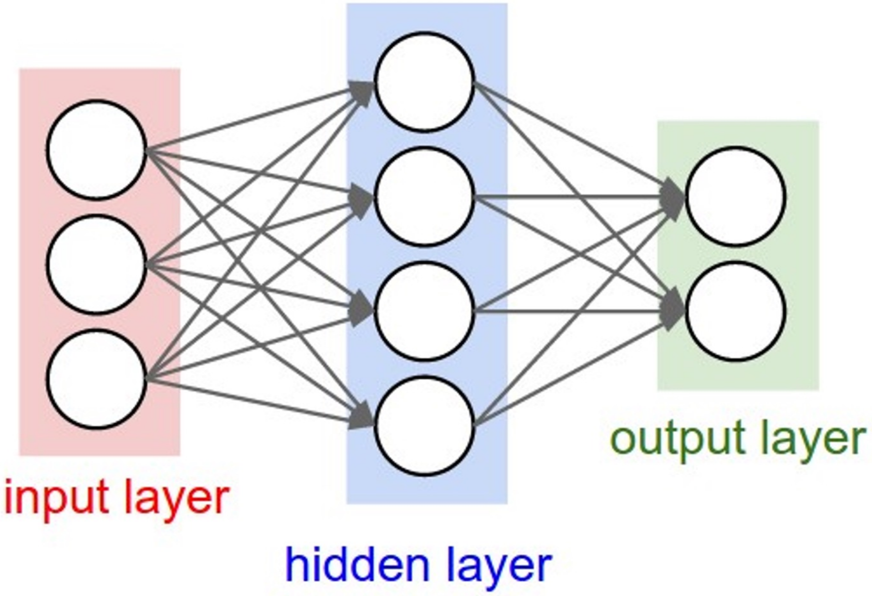
- Q: what happens when W=same initial value is used?



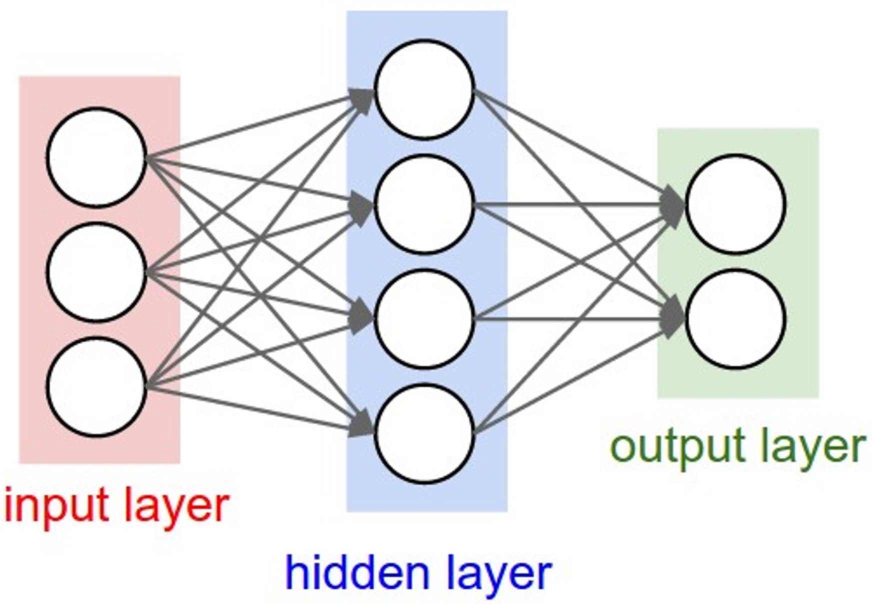
- Q: what happens when W=same initial value is used?
- A: All output will be the same! $w_1^T x = w_2^T x$ if $w_1 = w_2$



- Q: what if $w_1 = 0$ and $w_2 = 100000$?
- A: Output will have extremely different values! Vanishing / exploding gradient



Weight initialization: goal is to **maintain both diversity and variance** of layer output throughout the network, at least at the beginning of the training



- First idea: **Small random numbers**
(gaussian with zero mean and 1e-2 standard deviation)

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

- First idea: **Small random numbers**
(gaussian with zero mean and 1e-2 standard deviation)

```
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

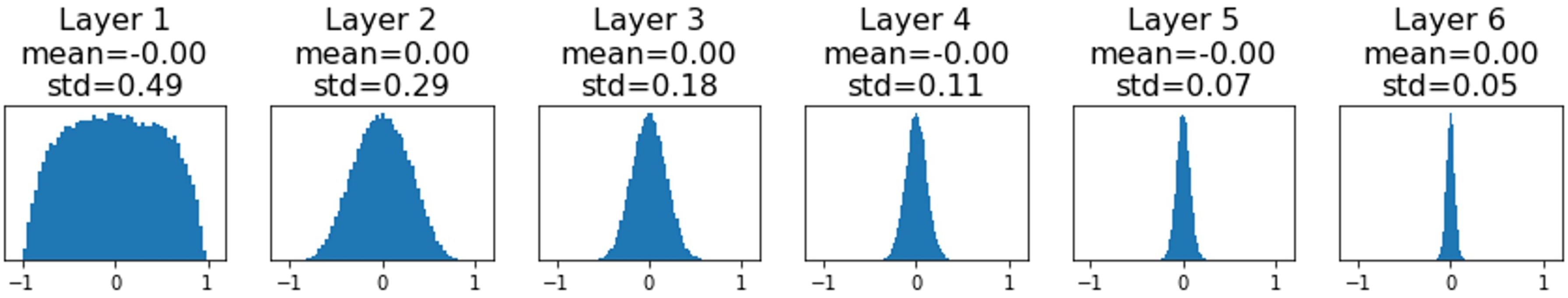
What will happen to the activations for the last layer?

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



Visualize distribution of activations

Weight Initialization: Activation statistics

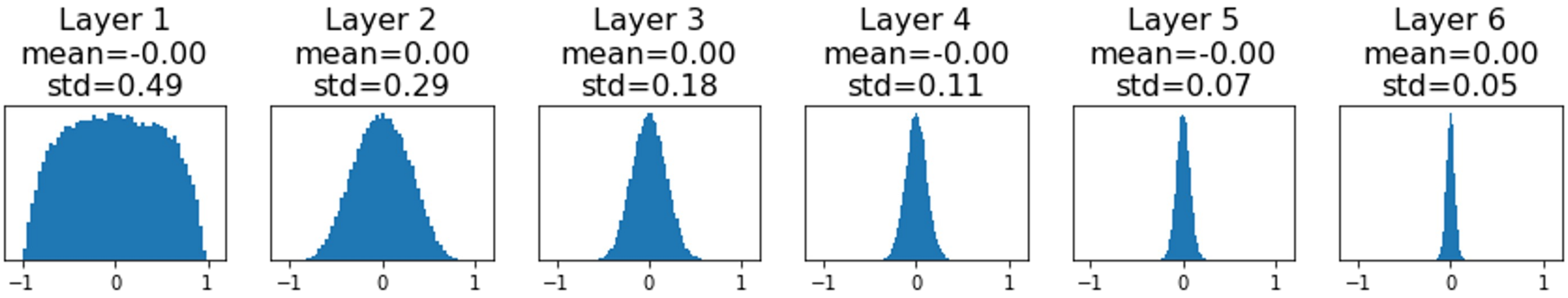
```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

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?

Hint: $\frac{\partial L}{\partial w} = x^T \left(\frac{\partial L}{\partial y}\right)$



Visualize distribution of activations

Weight Initialization: Activation statistics

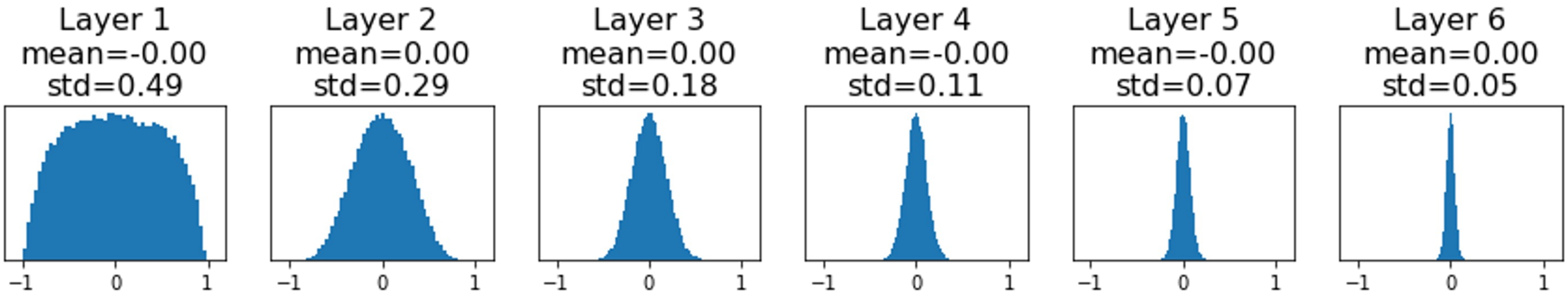
```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

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?

A: Very small, slow learning



Visualize distribution of activations

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

Initialize with higher values

What will happen to the activations for the last layer?

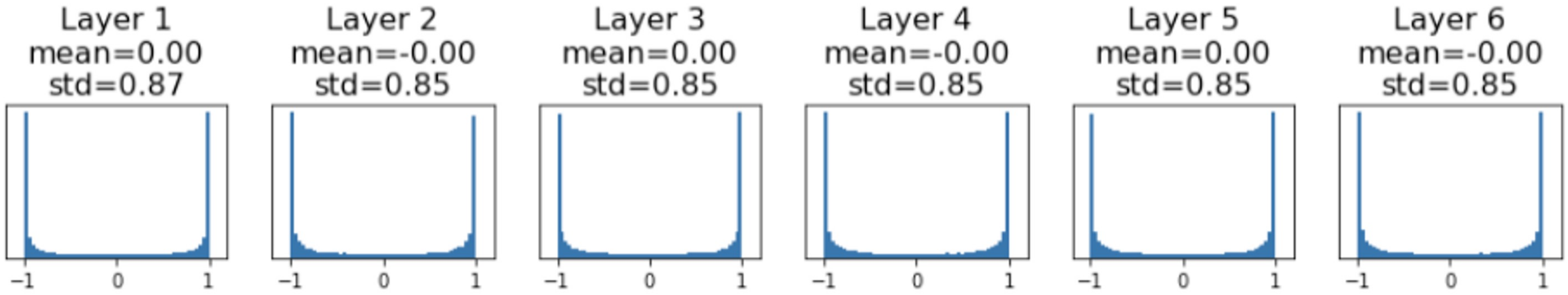
Weight Initialization: Activation statistics

```
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

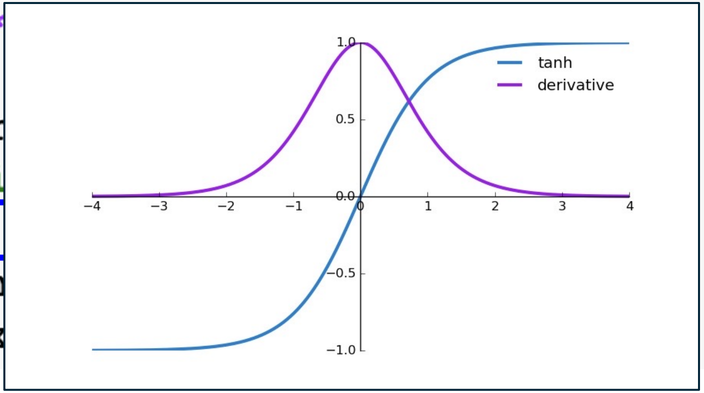
Q: What do the gradients look like?



Visualize distribution of activations

Weight Initialization: Activation statistics

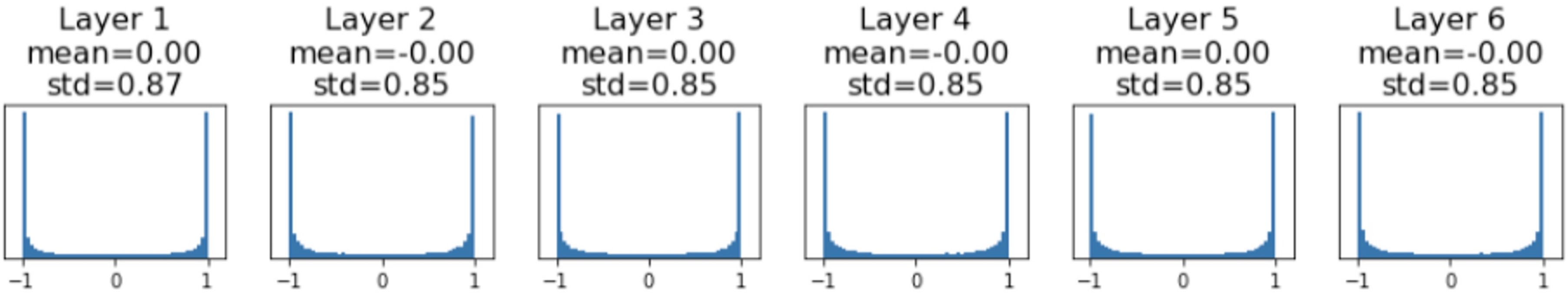
```
dims = [4096] *
hs = []
x = np.random.r
for Din, Dout i
    W = 0.05 *
    x = np.tanh
    hs.append(x
```



All activations saturate

Q: What do the gradients look like?

A: For tanh, large value -> small gradient



Visualize distribution of activations

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?

More generally, *gradient explosion* (high w-> high output -> high gradient).

Visualize distribution of activations

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)

Assume each input contribute similarly to output

more number of weights needs -> small weight multiplier

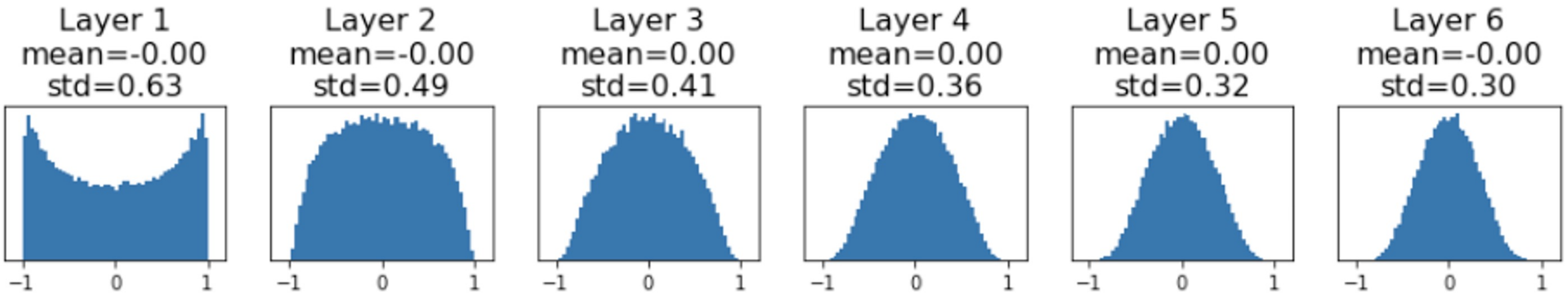
Glorot and Bengio, “Understanding the difficulty of training deep feedforward neural networks”, AISTAT 2010

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

Visualize distribution of activations

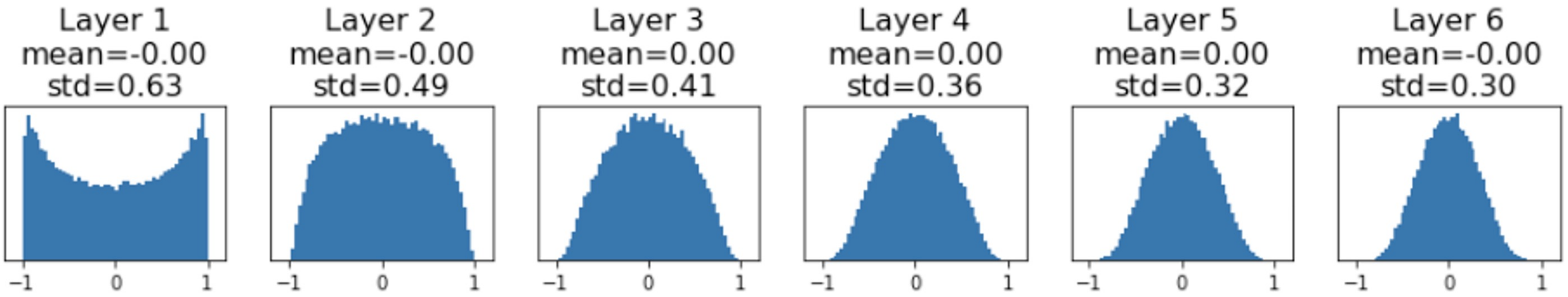
Weight Initialization: “Xavier” Initialization

```
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!

For conv layers, Din is filter_size^2 * input_channels



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

Visualize distribution of activations

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!

For conv layers, Din is $\text{filter\_size}^2 * \text{input\_channels}$

Let: $y = x_1w_1+x_2w_2+\ldots+x_{Din}w_{Din}$

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!

For conv layers, Din is filter_size^2 * input_channels

Let: $y = x_1 w_1 + x_2 w_2 + \ldots + x_{Din} w_{Din}$

Assume: $Var(x_1) = Var(x_2) = \ldots = Var(x_{Din})$

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 filter_size^2 * input_channels

Let: $y = x_1w_1 + x_2w_2 + \ldots + x_{Din}w_{Din}$

Assume: $Var(x_1) = Var(x_2) = \ldots = Var(x_{Din})$

We want: $Var(y) = Var(x_i)$

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!

For conv layers, Din is $\text{filter\_size}^2 * \text{input\_channels}$

Let: $y = x_1w_1+x_2w_2+\ldots+x_{Din}w_{Din}$

Assume: $Var(x_1) = Var(x_2)= \ldots=Var(x_{Din})$

We want: $Var(y) = Var(x_i)$

$Var(y) = Var(x_1w_1+x_2w_2+\ldots+x_{Din}w_{Din})$
[substituting value of y]

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!

For conv layers, Din is filter_size^2 * input_channels

Let: $y = x_1w_1+x_2w_2+\ldots+x_{Din}w_{Din}$

Assume: $Var(x_1) = Var(x_2)= \ldots =Var(x_{Din})$

We want: $Var(y) = Var(x_i)$

$Var(y) = Var(x_1w_1+x_2w_2+\ldots+x_{Din}w_{Din})$

$= \Sigma\, Var(x_iw_i) = Din\, Var(x_iw_i)$

[Assume all x_i, w_i are iid] $\sigma^2_{X+Y} = \sigma^2_X + \sigma^2_Y$

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!

For conv layers, Din is $\text{filter\_size}^2 * \text{input\_channels}$

Let: $y = x_1w_1+x_2w_2+\ldots+x_{Din}w_{Din}$

Assume: $\mathrm{Var}(x_1) = \mathrm{Var}(x_2)= \ldots=\mathrm{Var}(x_{Din})$

We want: $\mathrm{Var}(y) = \mathrm{Var}(x_i)$

$$\begin{aligned}\mathrm{Var}(y) &= \mathrm{Var}(x_1w_1+x_2w_2+\ldots+x_{Din}w_{Din})\\ &= \mathrm{Din}\ \mathrm{Var}(x_iw_i)\\ &= \mathrm{Din}\ \mathrm{Var}(x_i)\ \mathrm{Var}(w_i)\end{aligned}$$

[Assume all x_i, w_i are zero mean]

$$\mathrm{Var}(XY) = E(X^2Y^2) - (E(XY))^2 = \mathrm{Var}(X)\mathrm{Var}(Y) + \cancel{\mathrm{Var}(X)(E(Y))^2} + \cancel{\mathrm{Var}(Y)(E(X))^2}$$

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!

For conv layers, Din is filter_size^2 * input_channels

Let: $y = x_1 w_1 + x_2 w_2 + \ldots + x_{Din} w_{Din}$

Assume: $Var(x_1) = Var(x_2) = \ldots = Var(x_{Din})$

We want: $Var(y) = Var(x_i)$

$$Var(y) = Var(x_1 w_1 + x_2 w_2 + \ldots + x_{Din} w_{Din})$$
$$= Din\ Var(x_i w_i)$$
$$= Din\ Var(x_i)\ Var(w_i)$$

[Assume all x_i, w_i are iid]

So, $Var(y) = Var(x_i)$ only when $Var(w_i) = 1/Din$

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

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)
```

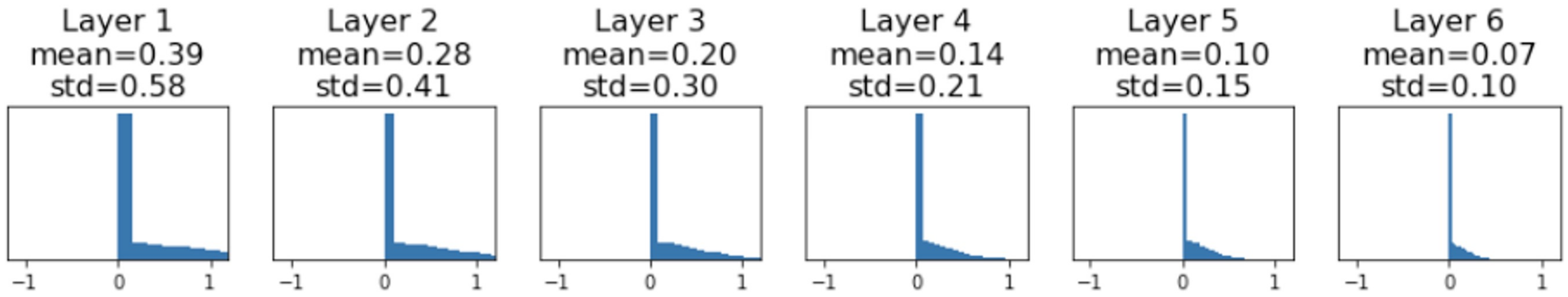
Change from tanh to ReLU

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



Visualize distribution of activations

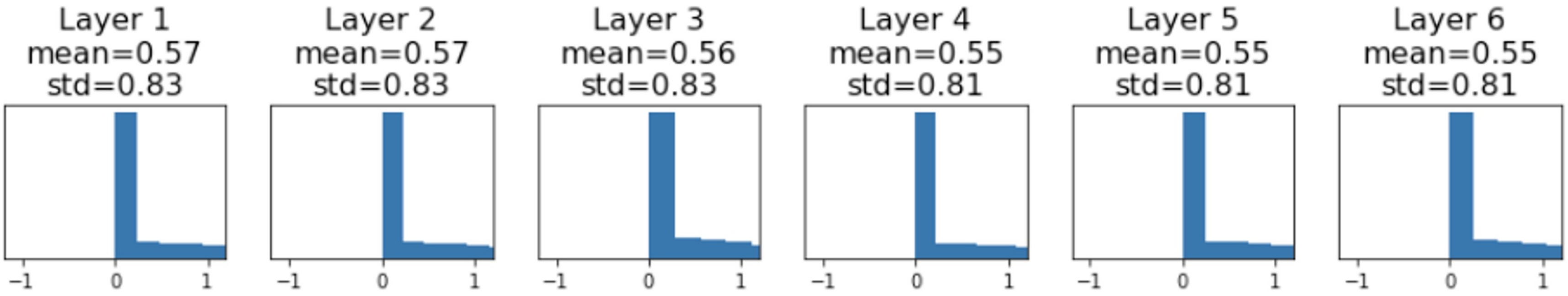
Weight Initialization: Kaiming / MSRA Initialization

```
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(2/Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

ReLU correction: std = sqrt(2 / Din)

Issue: Half of the activation get killed.

Solution: make the non-zero output variance twice as large as input



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

Visualize distribution of activations

Proper initialization is still 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

Summary

Training Deep Neural Networks

- Details of the non-linear activation functions
    - Sigmoid, Tanh, ReLU, LeakyRELU, ELU, SELU
- Data normalization
    - Zero-centering, image normalization
- Weight Initialization
    - Constant init, random init, Xavier Init, Kaiming Init