

CS 4644-DL / 7643-A: LECTURE 9

DANFEI XU

Topics:

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

Administrative

- PS2/HW2 out: **Difficult assignment. Start early!**
- In-class quiz on Wednesday (Sep 30).
- Closed book/phone/laptop ...
- Sample quiz released on Piazza

CNN Architectures

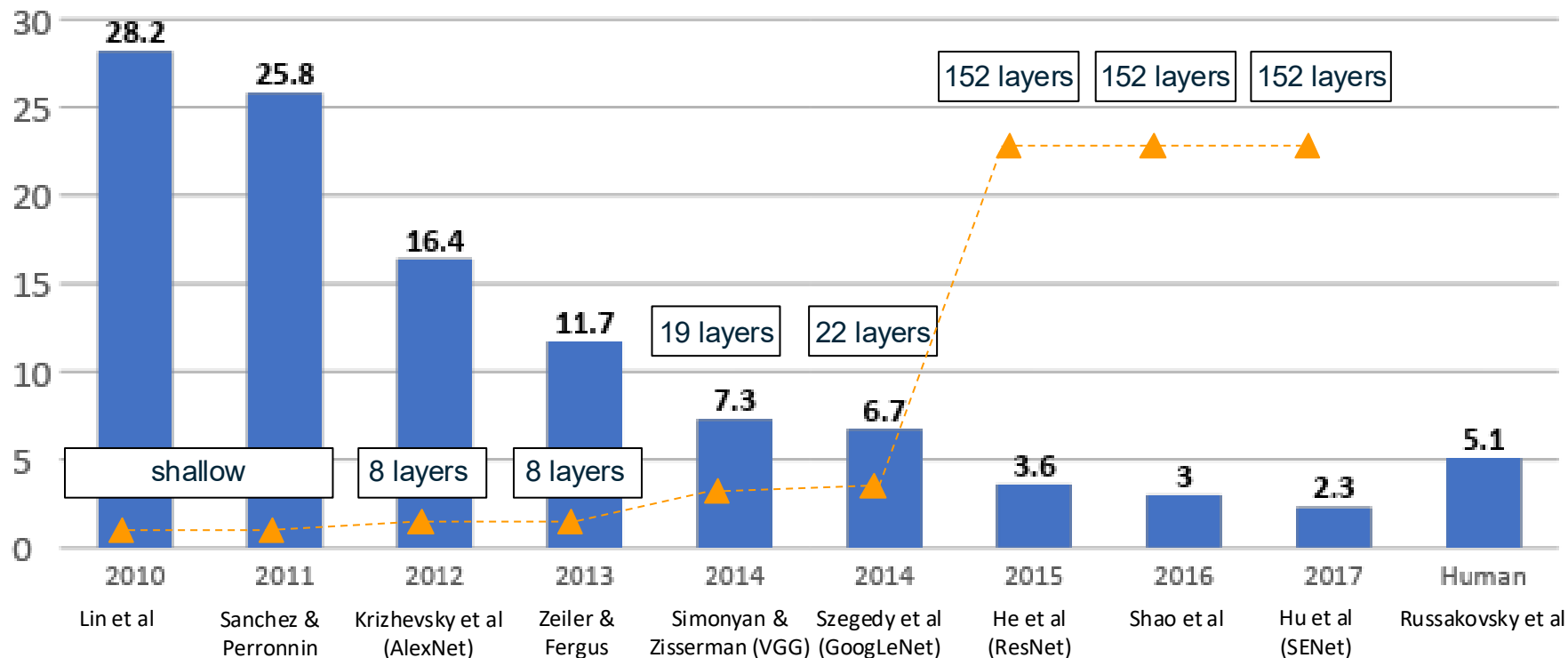
Case Studies

- AlexNet
- VGG
- GoogLeNet
- ResNet

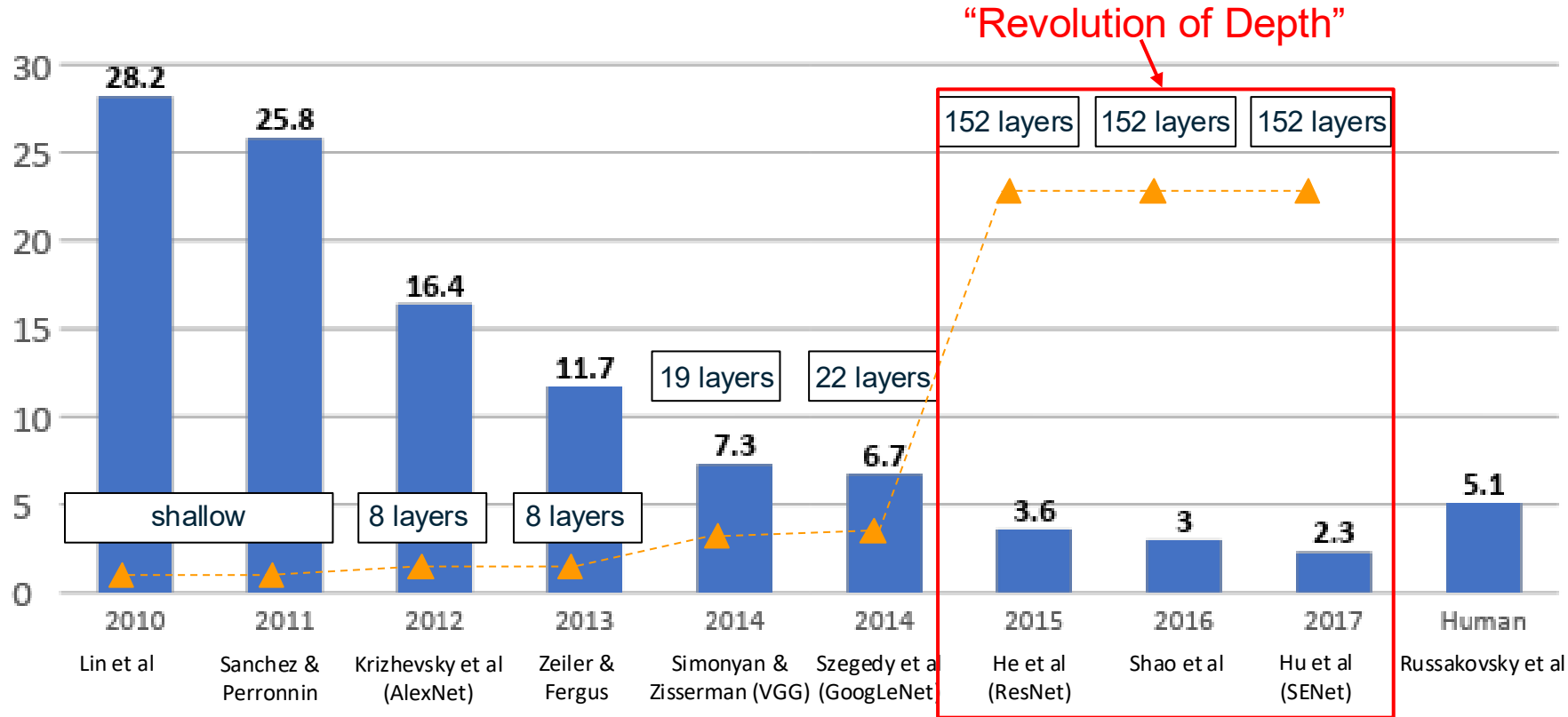
Also....

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

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



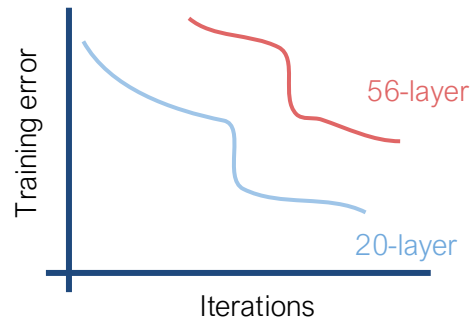
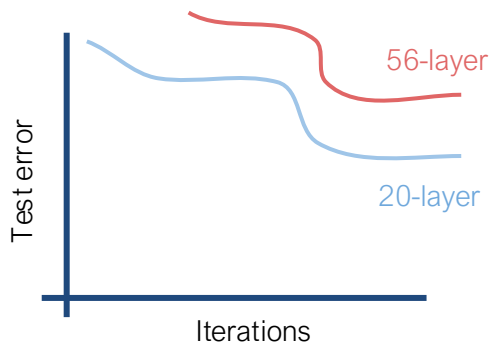
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



Case Study: ResNet

[He et al., 2015]

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

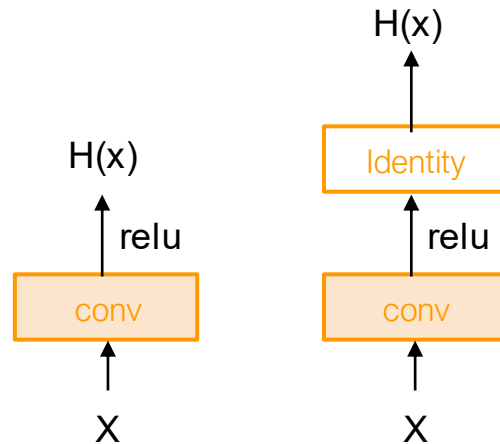
[He et al., 2015]

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

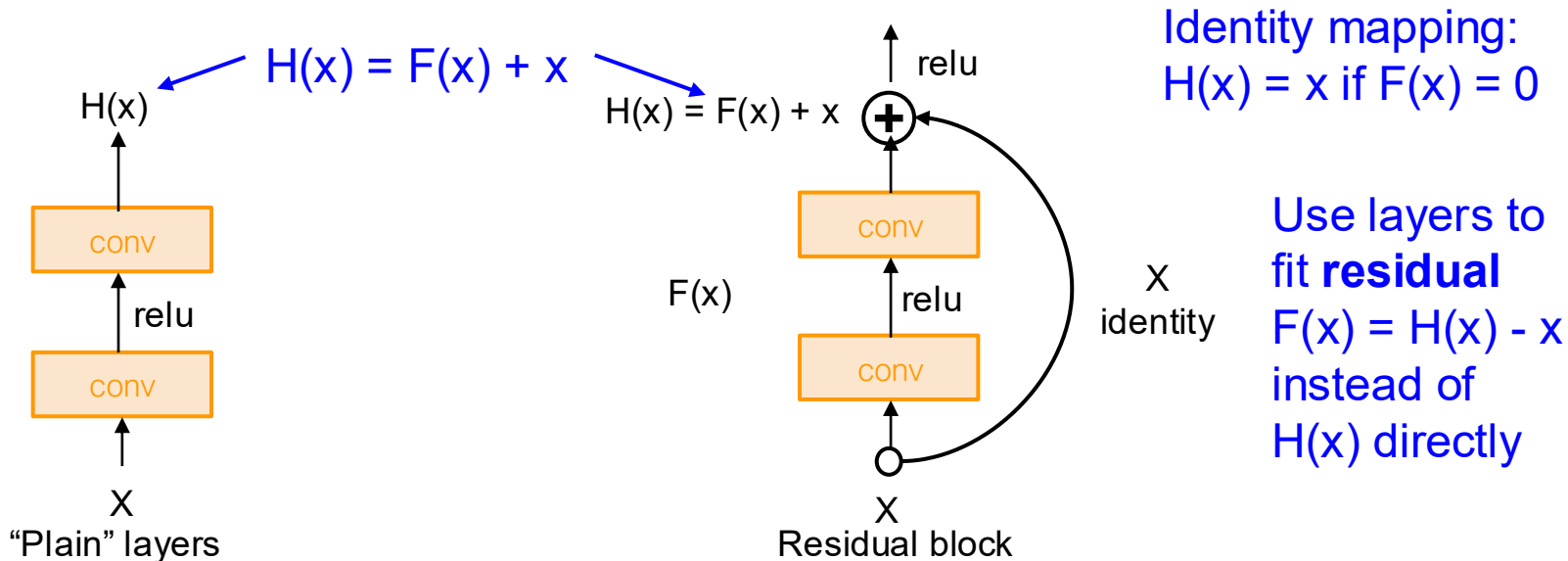
Solution: Change the network so learning identity functions (no-op) as extra layers is easy



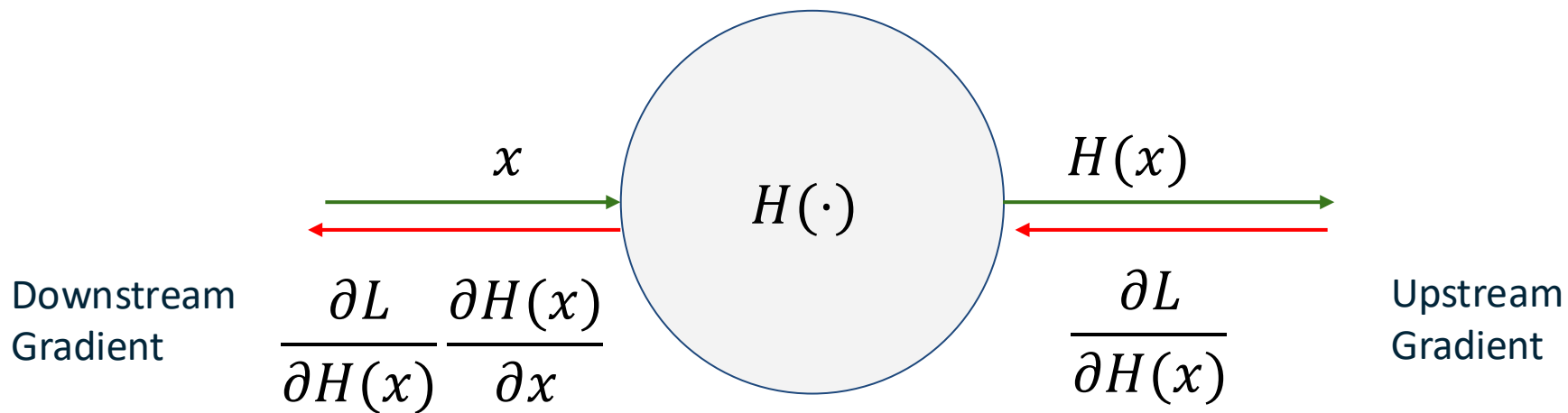
Case Study: ResNet

[He et al., 2015]

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



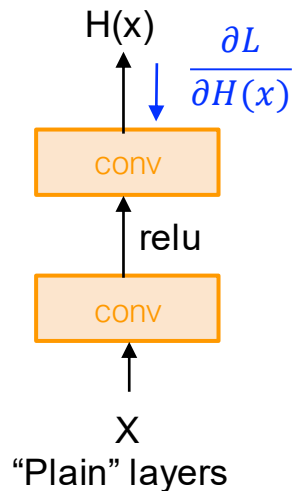
The Vanishing Gradient Problem



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

If W is small, downstream gradient is small.
Each small W in the chain makes gradient progressively smaller ->
Vanishing Gradient during backpropagation

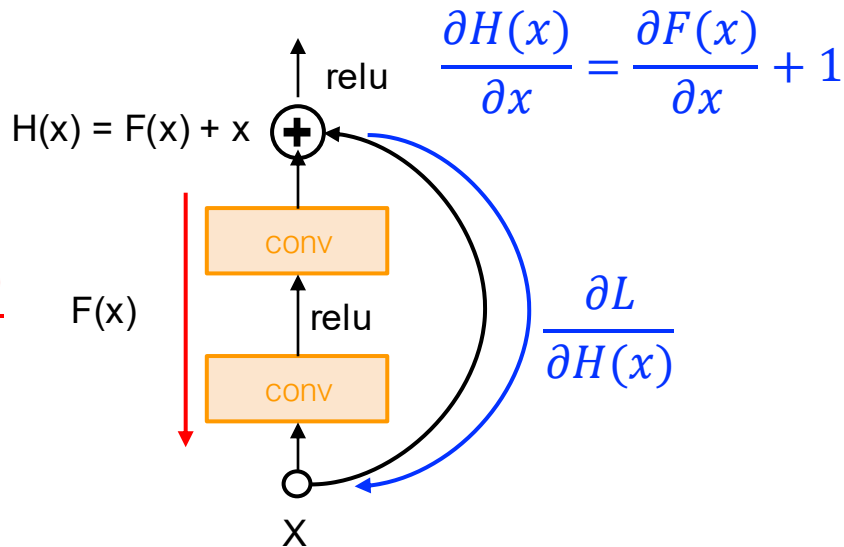
Case Study: ResNet



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

Potentially problematic

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



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

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

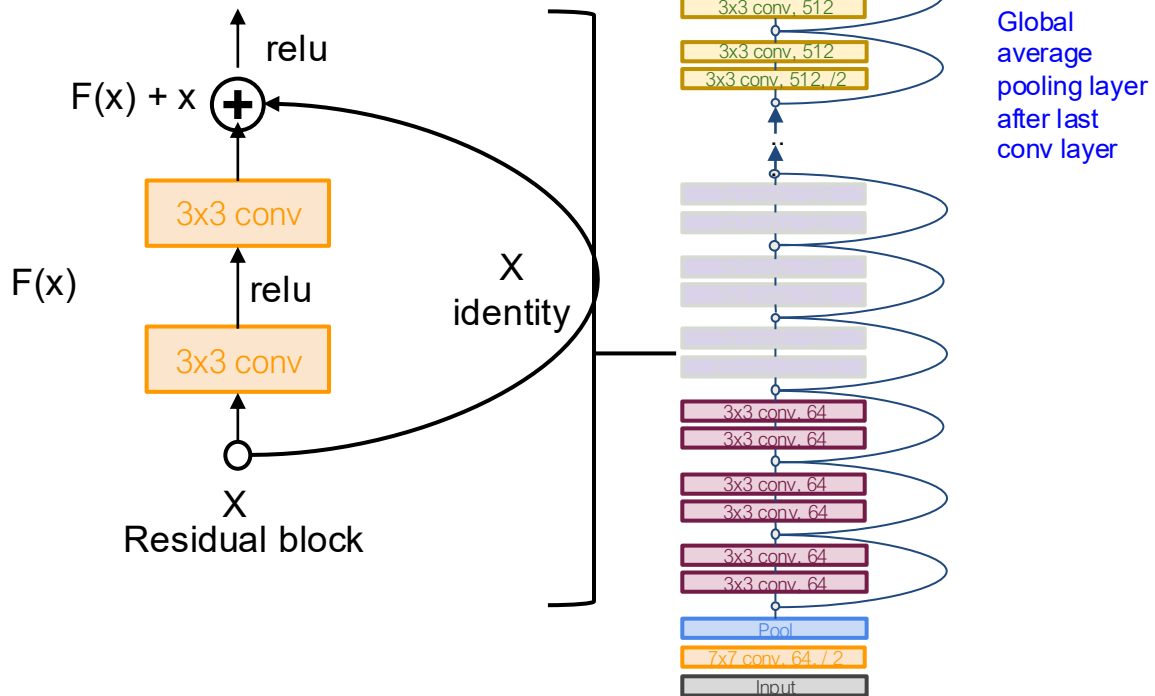
Direct gradient pathway

Case Study: ResNet

[He et al., 2015]

Full ResNet architecture:

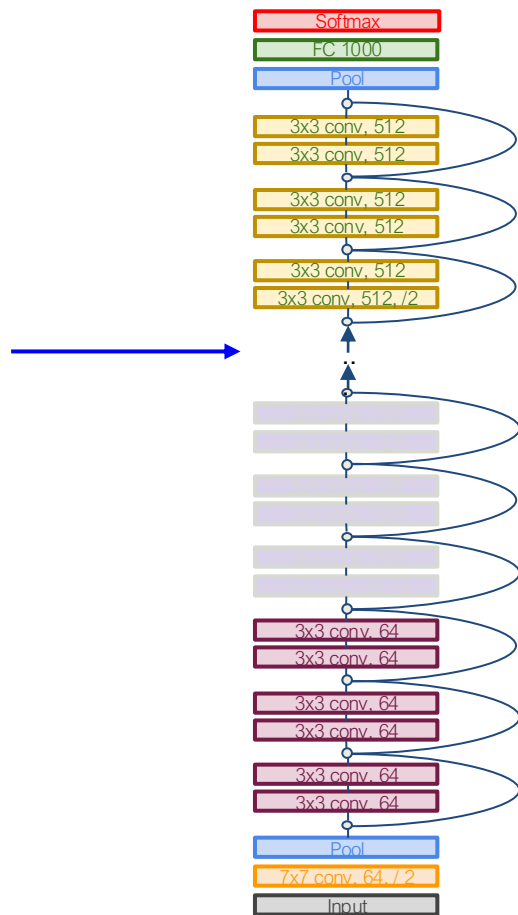
- Stack residual blocks
- Every residual block has two 3x3 conv layers
- Periodically, double # of filters and downsample spatially using stride 2 (/2 in each dimension)
- Reduce the activation volume by half.
- Additional conv layer at the beginning (stem)
- No FC layers at the end (only FC 1000 to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



Case Study: ResNet

[He et al., 2015]

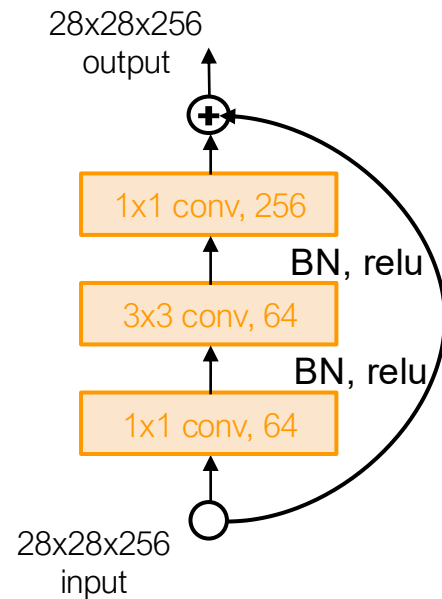
Total depths of 18, 34, 50,
101, or 152 layers for
ImageNet



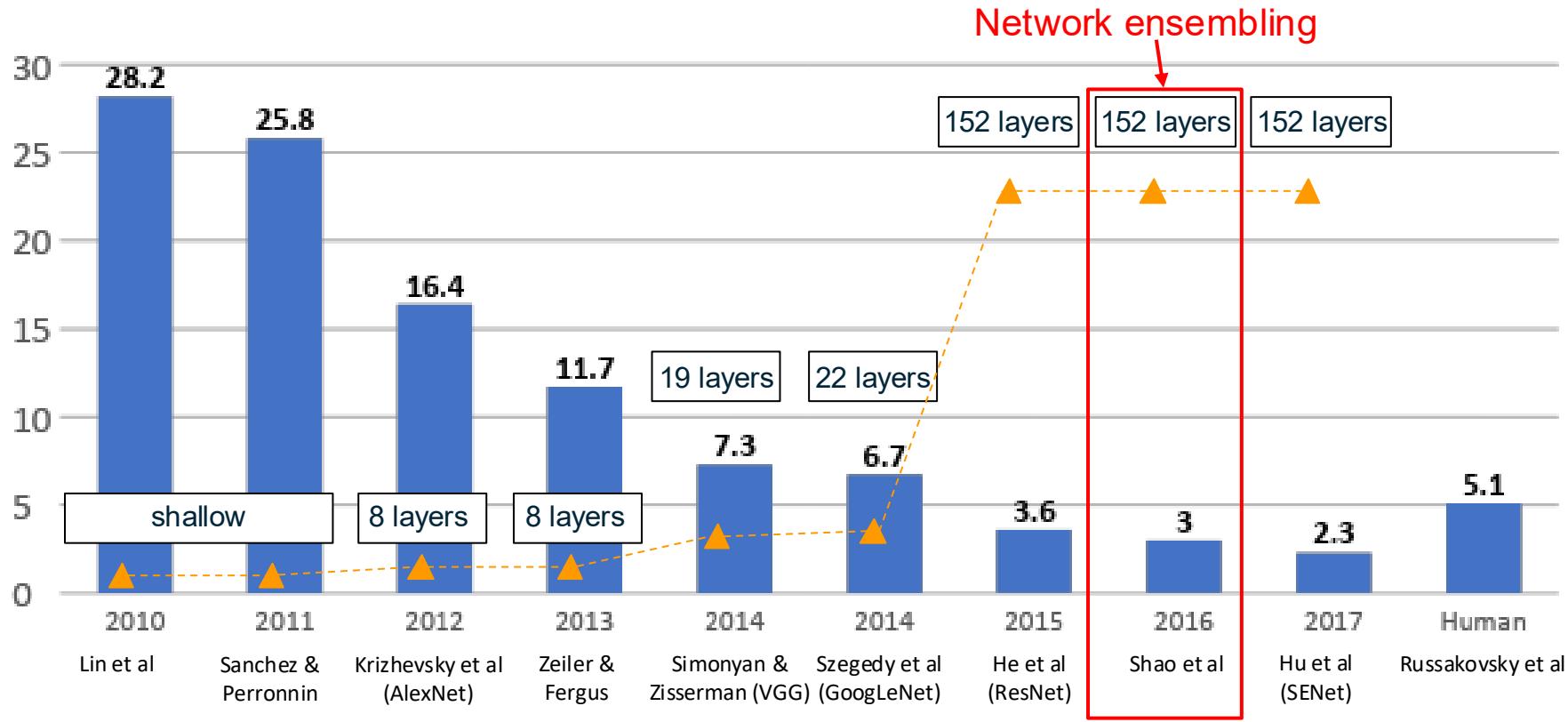
Case Study: ResNet

[He et al., 2015]

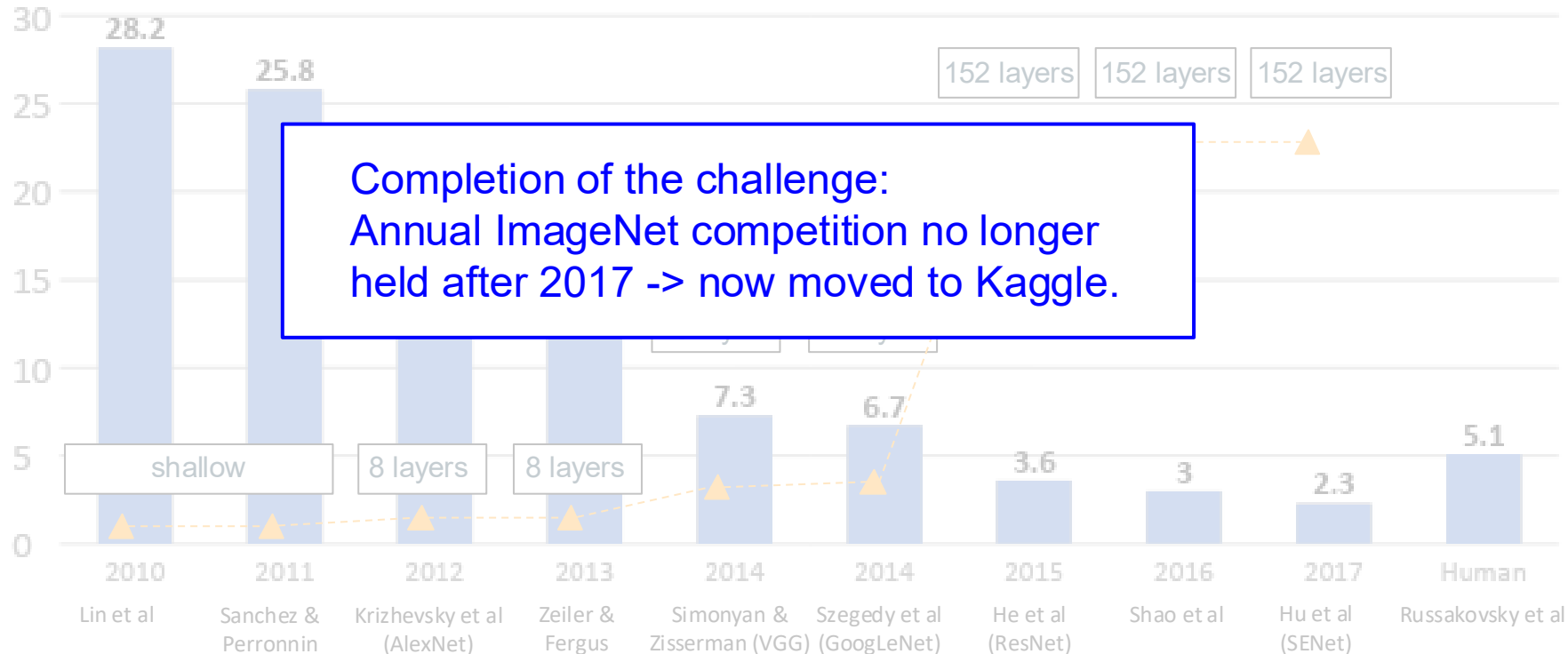
For deeper networks (ResNet-50+), use “bottleneck” layer to improve efficiency (similar to GoogLeNet)



ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



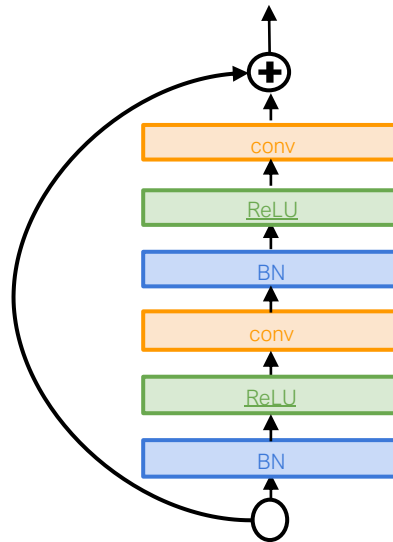
But research into CNN architectures is still flourishing

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

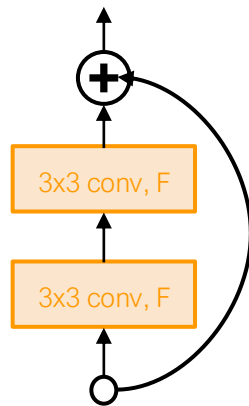


Improving ResNets...

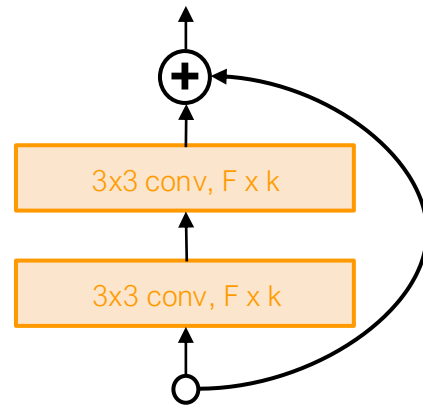
Wide Residual Networks

[Zagoruyko et al. 2016]

- Argues that residuals are the important factor, not depth
- Use wider residual blocks ($F \times k$ 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)



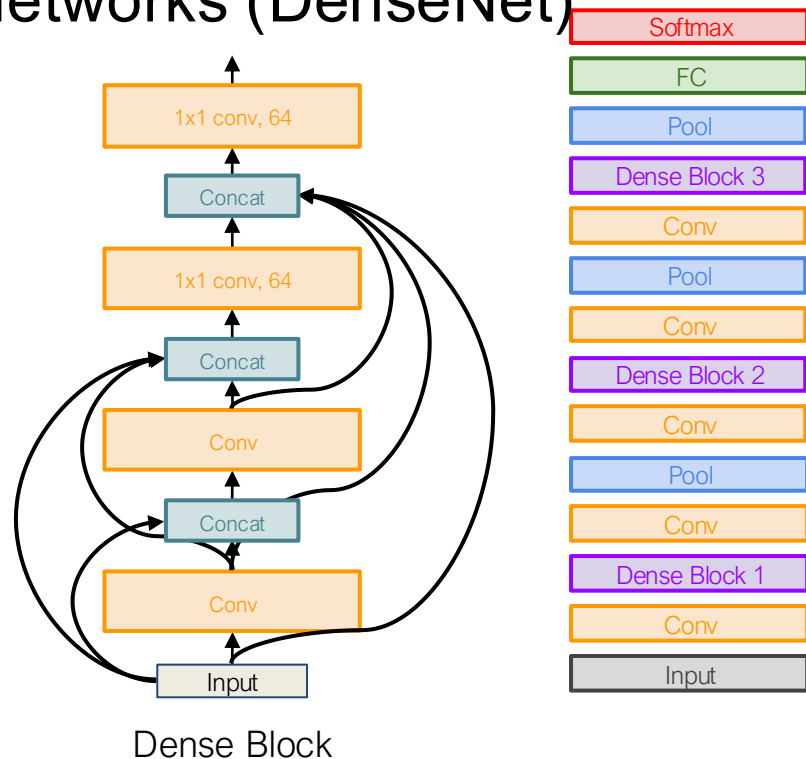
Basic residual block



Wide residual block

Densely Connected Convolutional Networks (DenseNet)

- 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

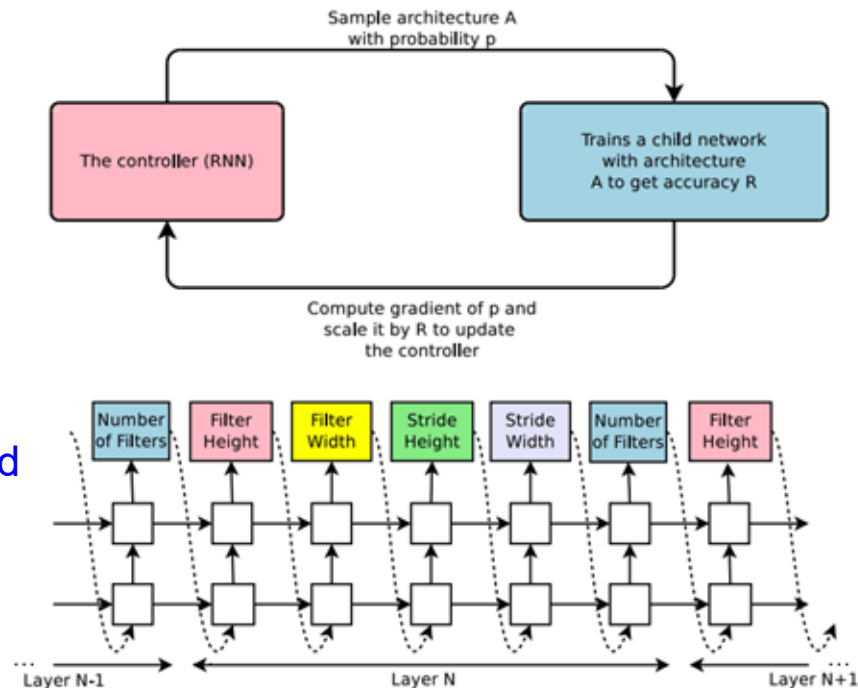


Learning to search for network architectures...

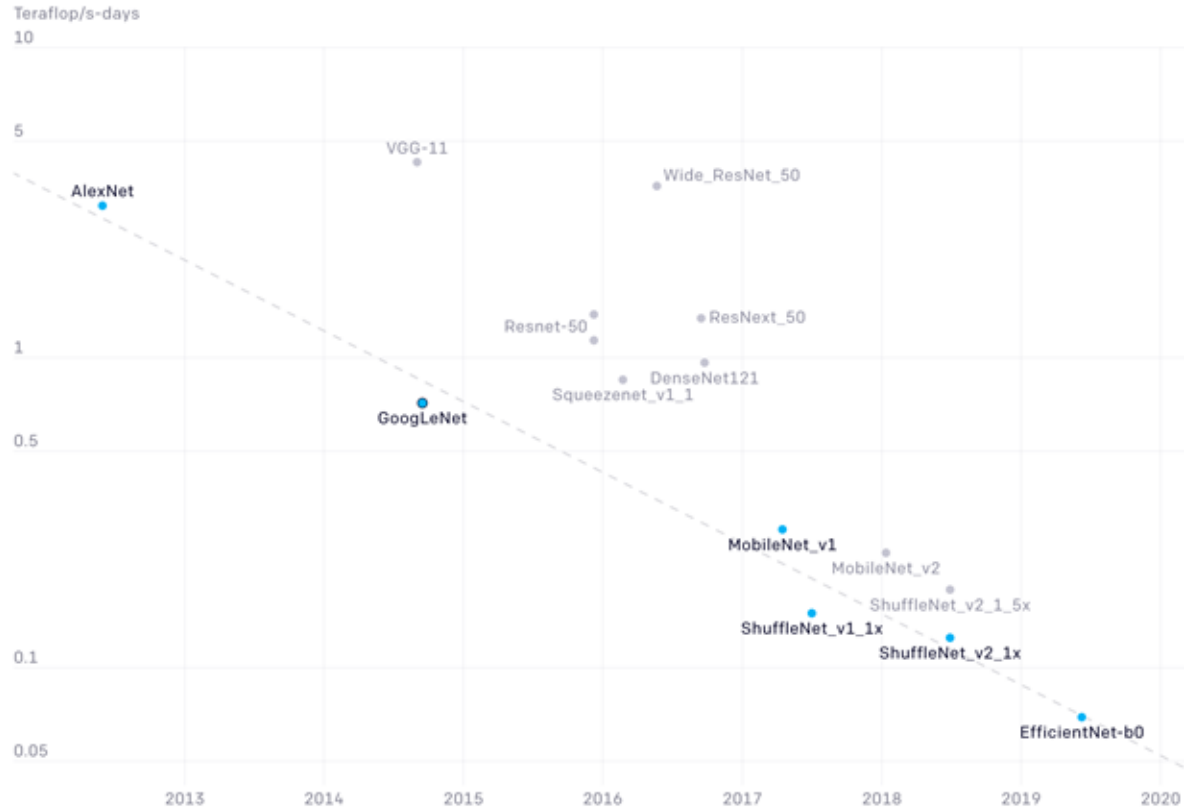
Neural Architecture Search with Reinforcement Learning (NAS)

[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)



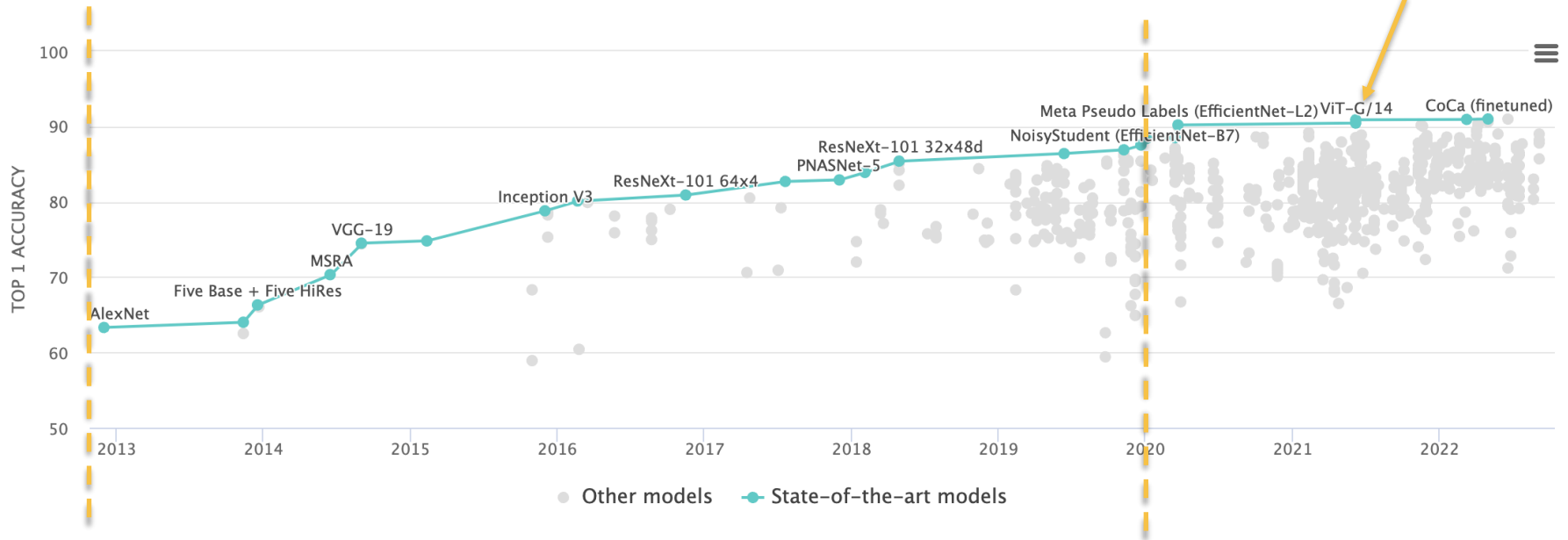
Amount of compute required to reach “AlexNet performance”



<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

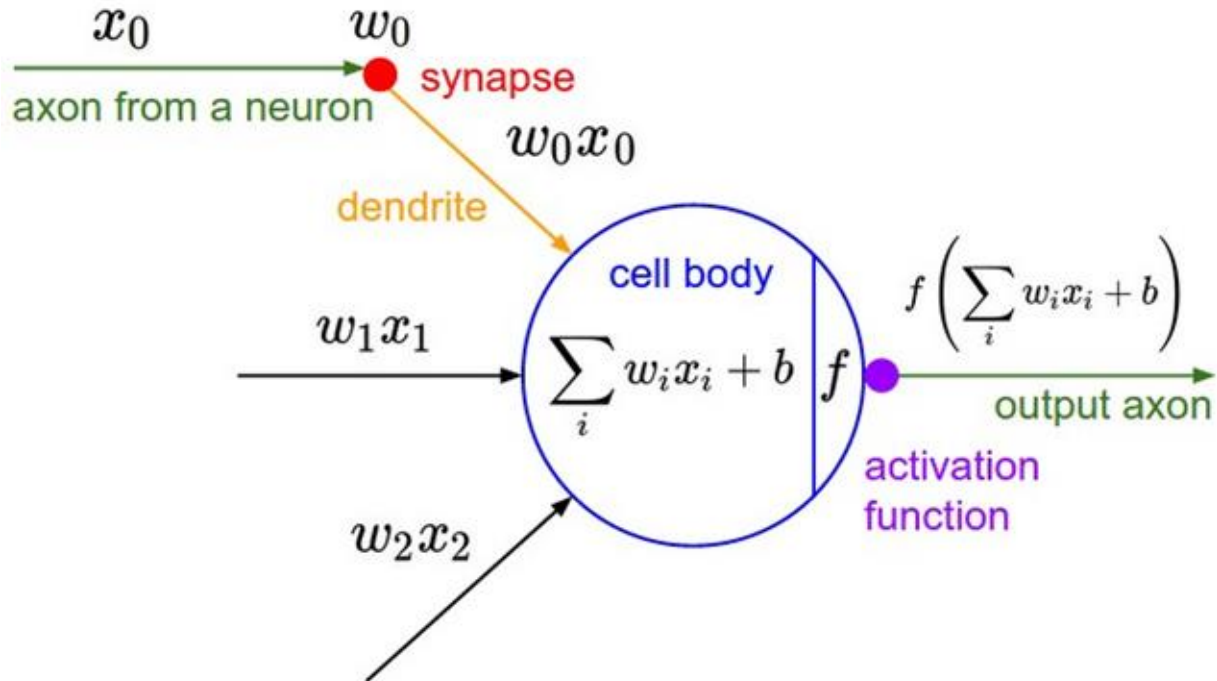
- Details of the non-linear activation functions
- Data normalization
- Weight Initialization
- Batch Normalization
- Regularization
- Advanced Optimization
- Data Augmentation
- Transfer learning
- Hyperparameter Tuning
- Model Ensemble

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

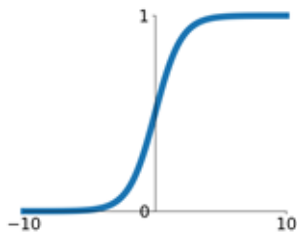
Activation Functions



Activation Functions

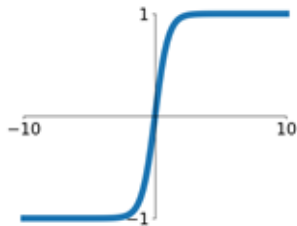
Sigmoid

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



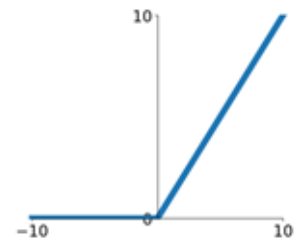
tanh

$$\tanh(x)$$



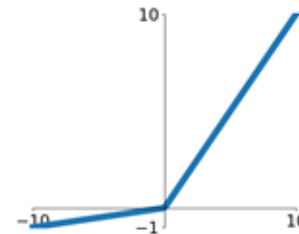
ReLU

$$\max(0, x)$$



Leaky ReLU

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

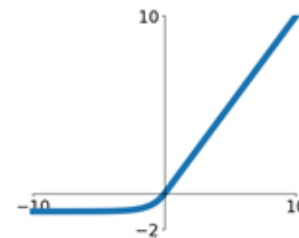


Maxout

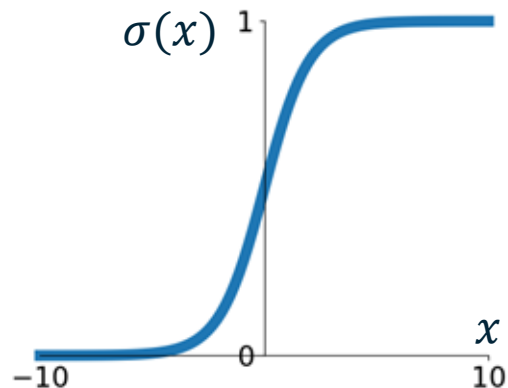
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

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



Activation Functions

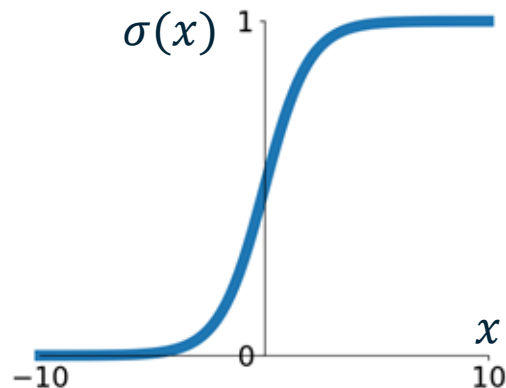


Sigmoid

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

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

Activation Functions



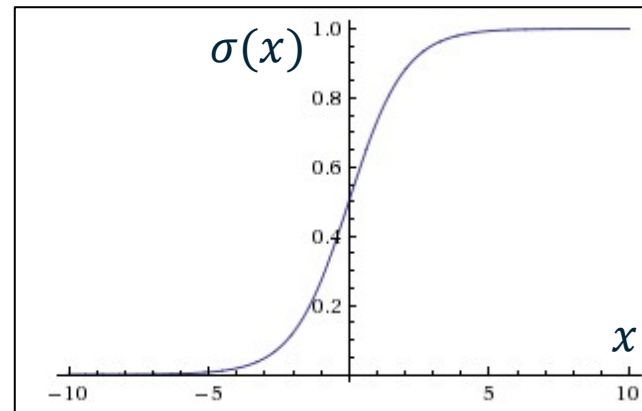
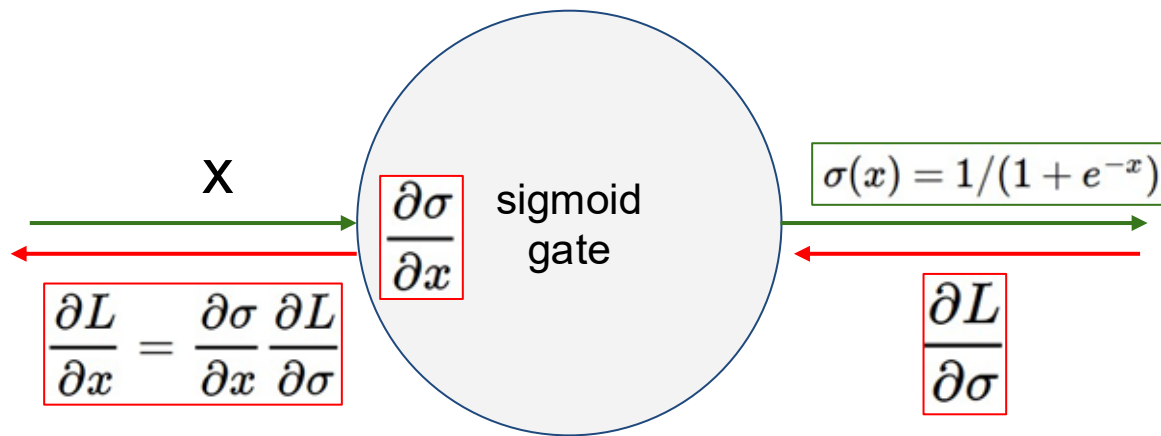
Sigmoid

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

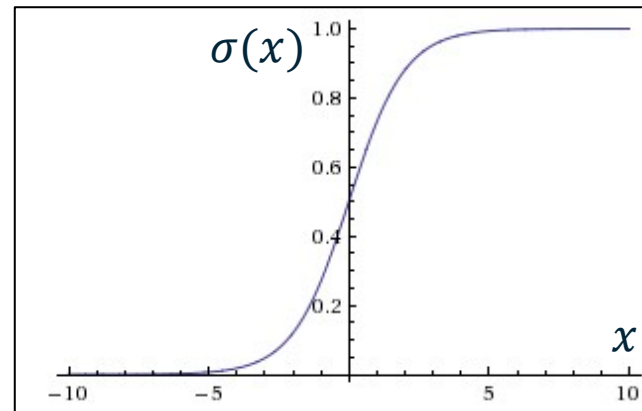
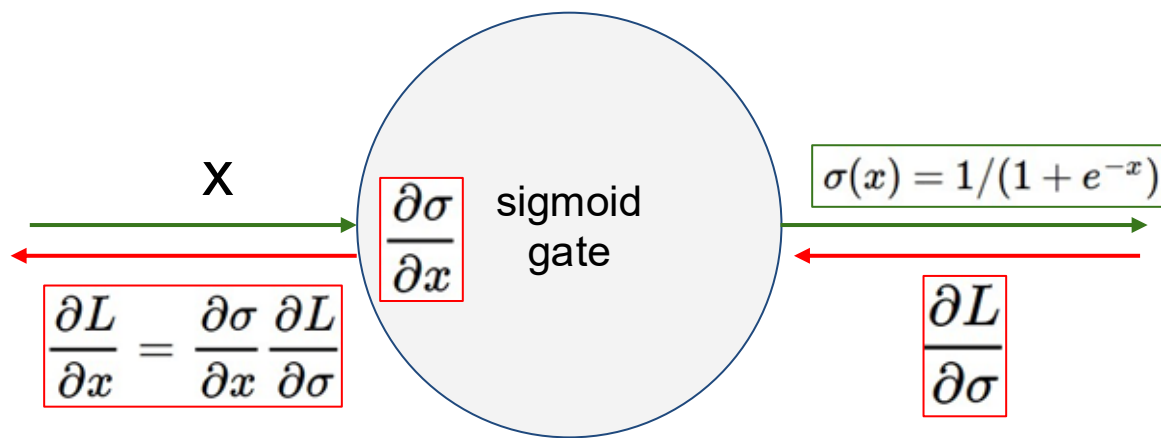
- Squashes numbers to range $[0, 1]$
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

Problems:

1. Saturated neurons “kill” the gradients

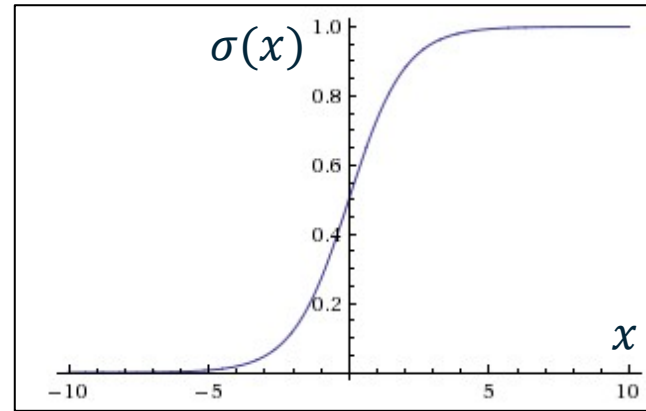
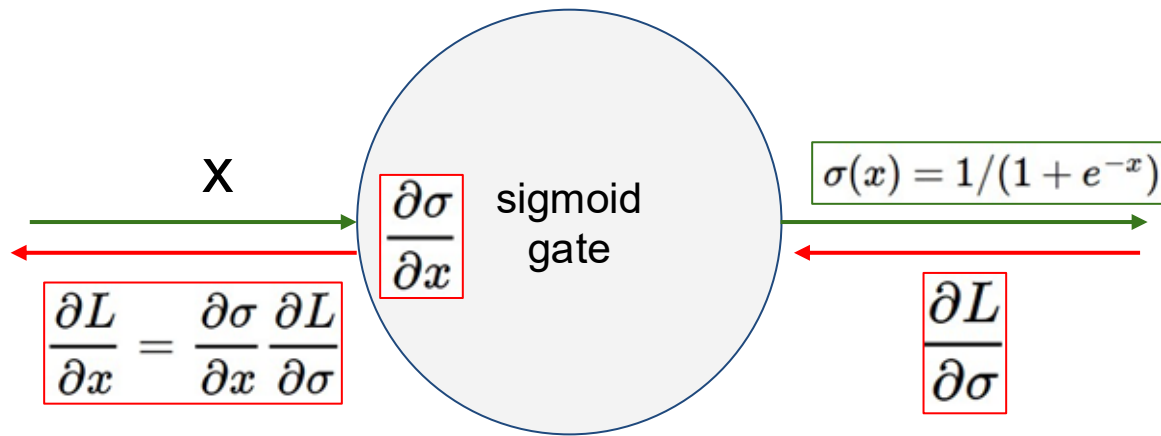


$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x) (1 - \sigma(x))$$



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

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

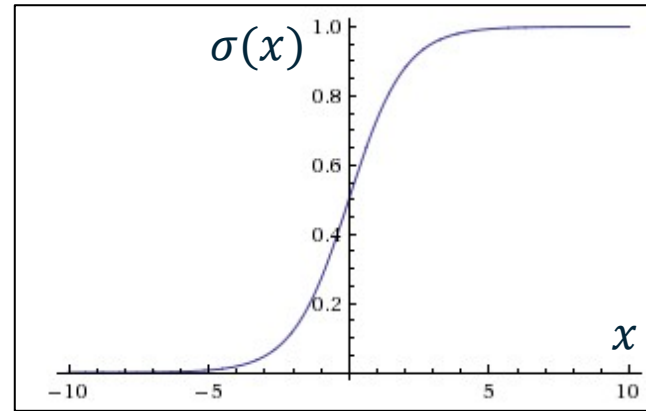
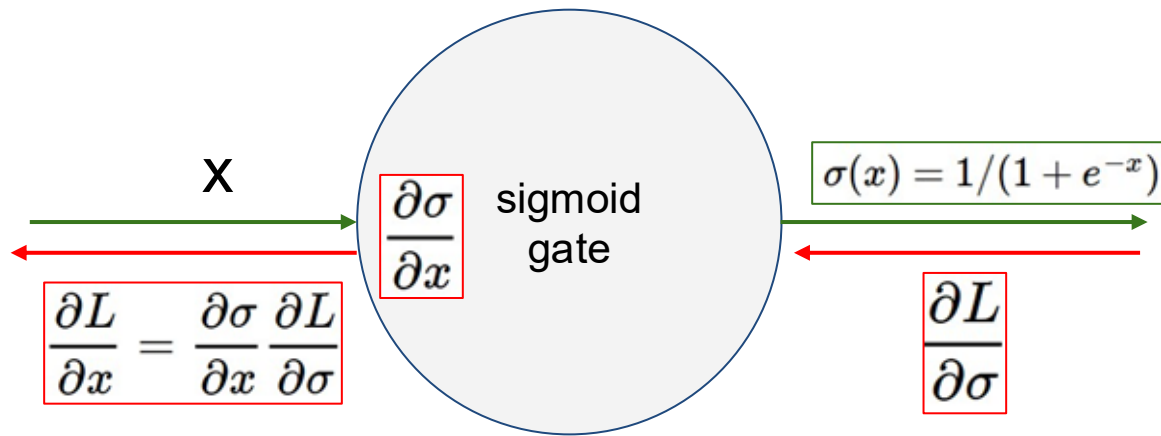


What happens to $\frac{\partial \sigma}{\partial x}$ when $x = -10$?

$$\sigma(x) \approx 0$$

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

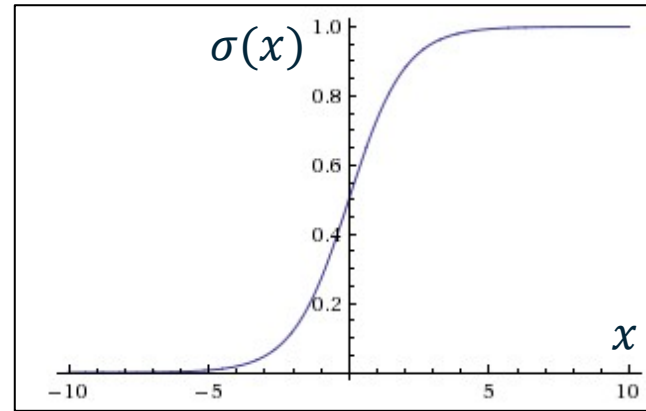
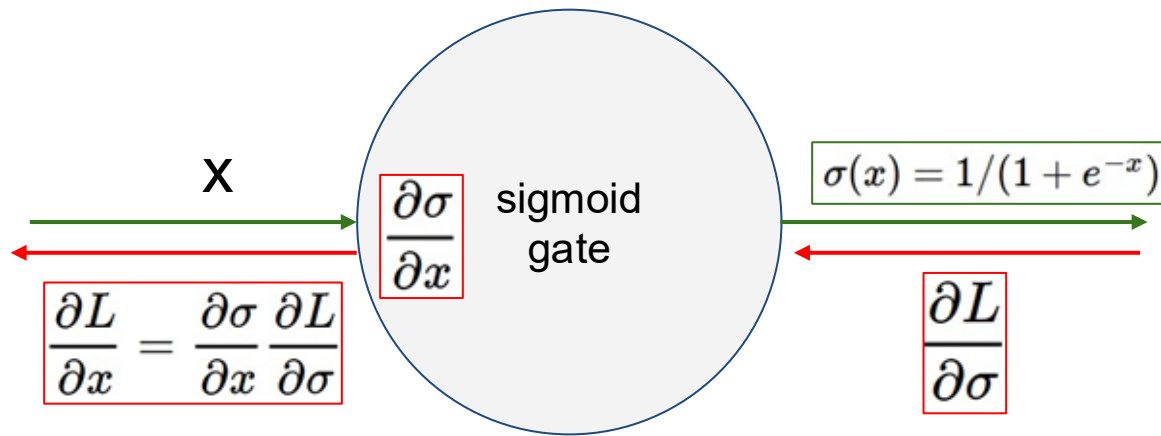
$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x) (1 - \sigma(x))$$



What happens when $x = -10$?

What happens when $x = 10$?

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

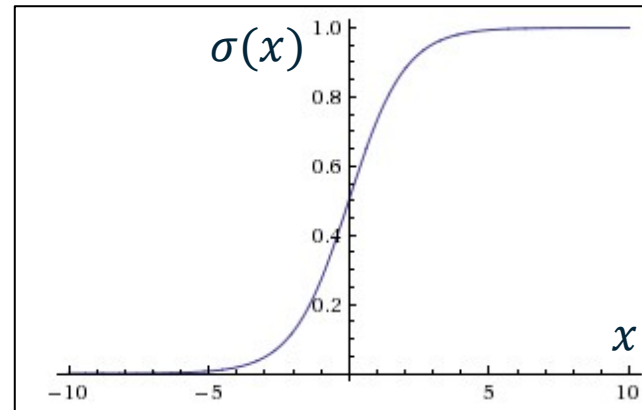
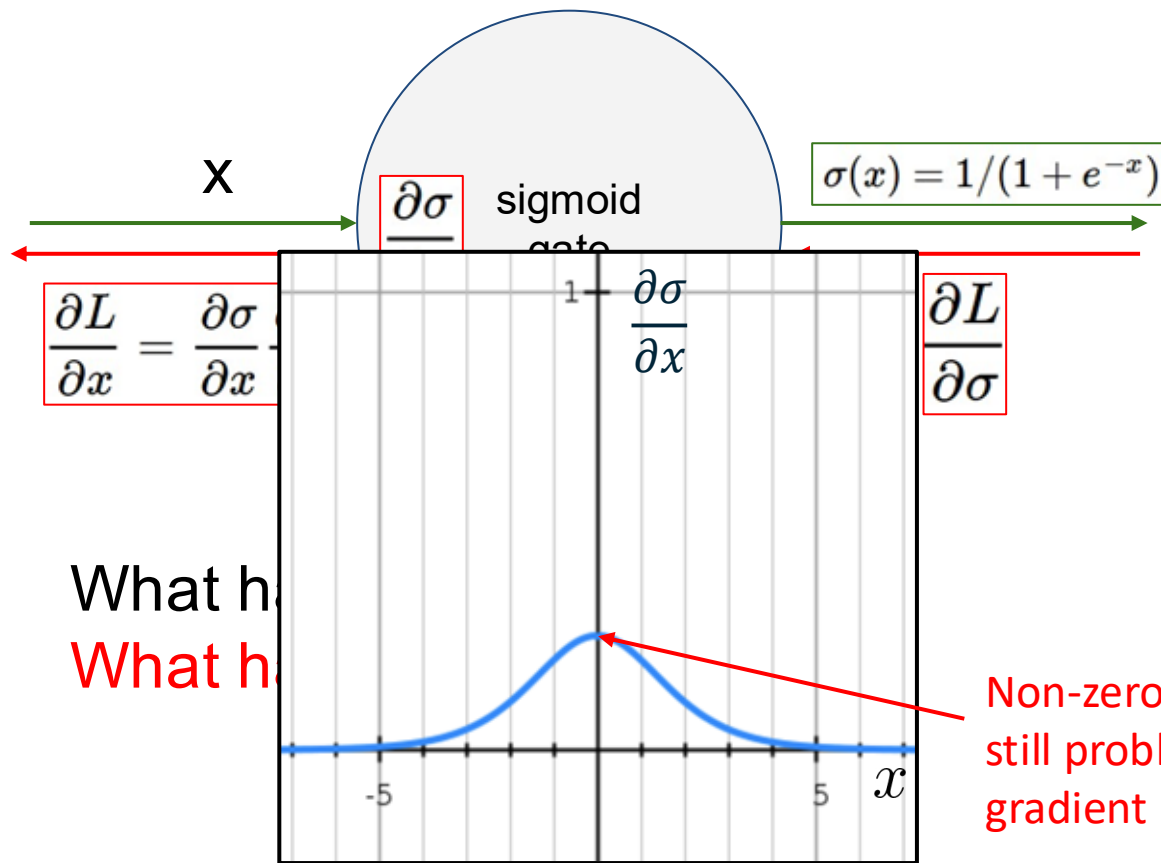


What happens when $x = -10$?

What happens when $x = 10$?

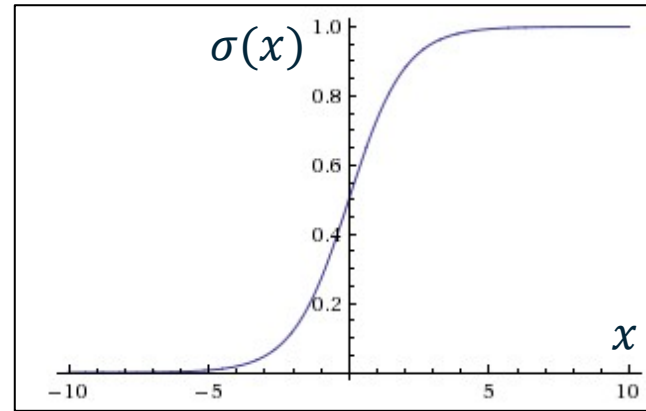
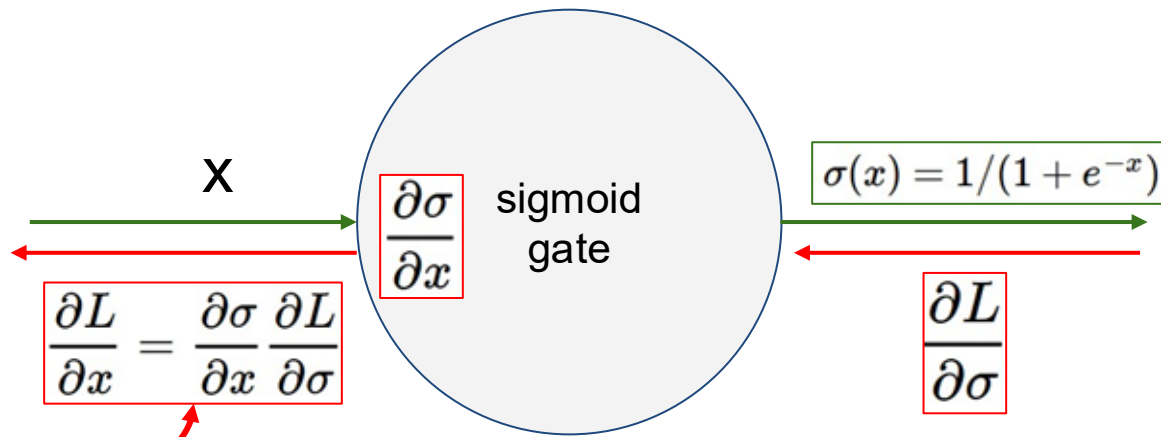
$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x) (1 - \sigma(x))$$

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



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

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



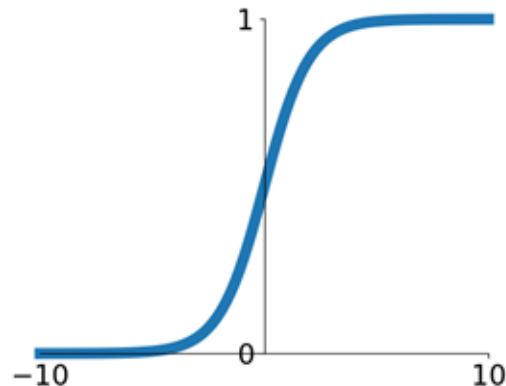
Why is this a problem?

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

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

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

Activation Functions



Sigmoid

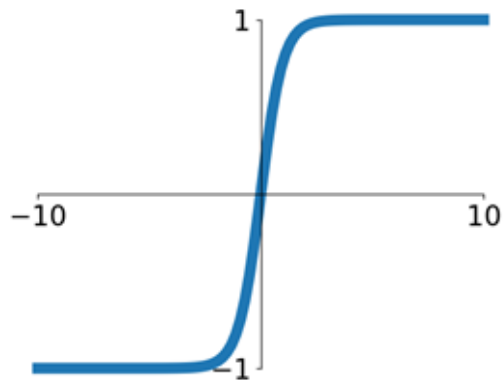
$$\sigma(x) = 1/(1 + e^{-x})$$

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

Problems:

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

Activation Functions

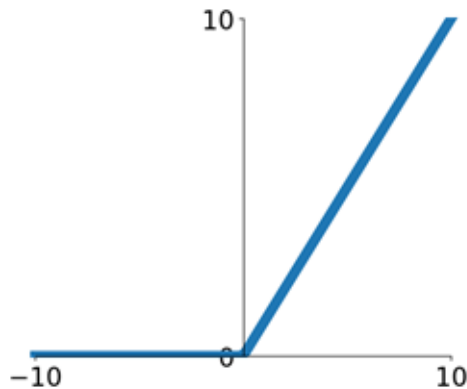


$\tanh(x)$

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

[LeCun et al., 1991]

Activation Functions

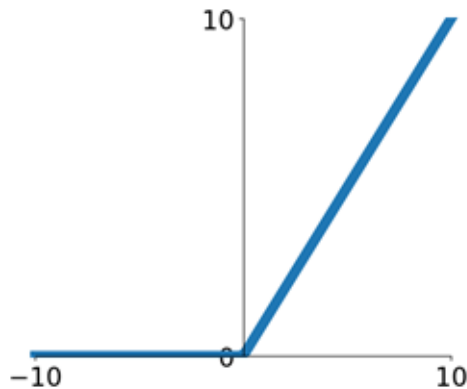


- Computes $f(x) = \max(0, x)$
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)

ReLU

(Rectified Linear Unit)

Activation Functions



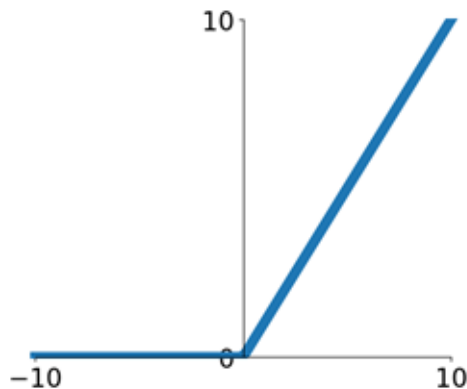
ReLU

(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)

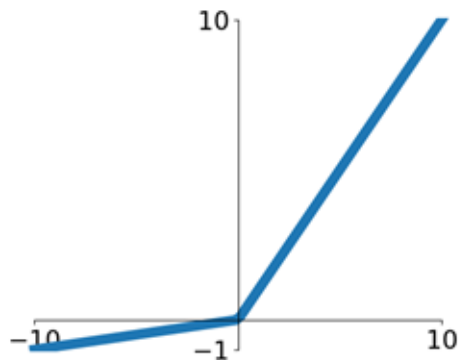
- Computes $f(x) = \max(0, x)$
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- An annoyance:

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”.**

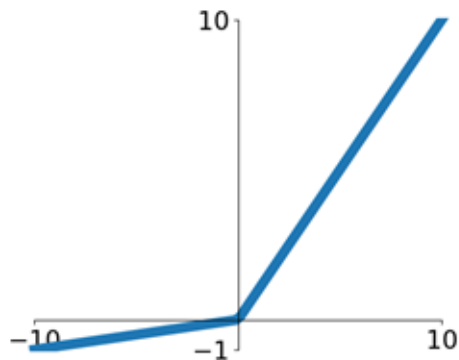
Leaky ReLU

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

Activation Functions

[Mass et al., 2013]

[He et al., 2015]



Leaky ReLU

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

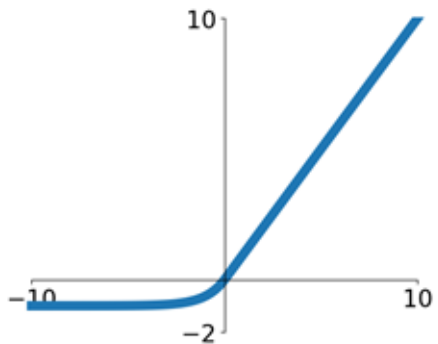
- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- **will not “die”.**

Parametric Rectifier (PReLU)

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

backprop into α
(parameter)

Exponential Linear Units (ELU)



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

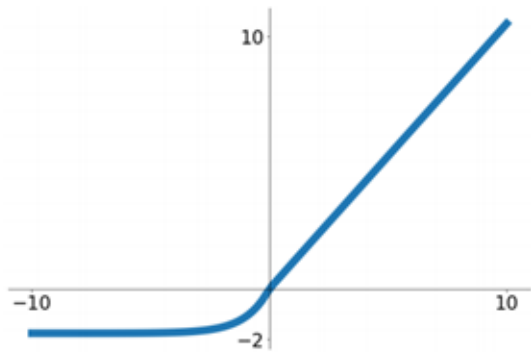
(Alpha default = 1)

- 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)



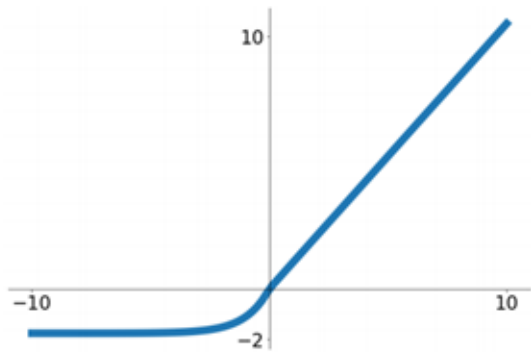
$$f(x) = \begin{cases} \lambda x & \text{if } x > 0 \\ \lambda \alpha (e^x - 1) & \text{otherwise} \end{cases}$$

- Scaled version of 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)



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

$\alpha = 1.6732632423543772848170429916717$

$\lambda = 1.0507009873554804934193349852946$

- Scaled version of 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

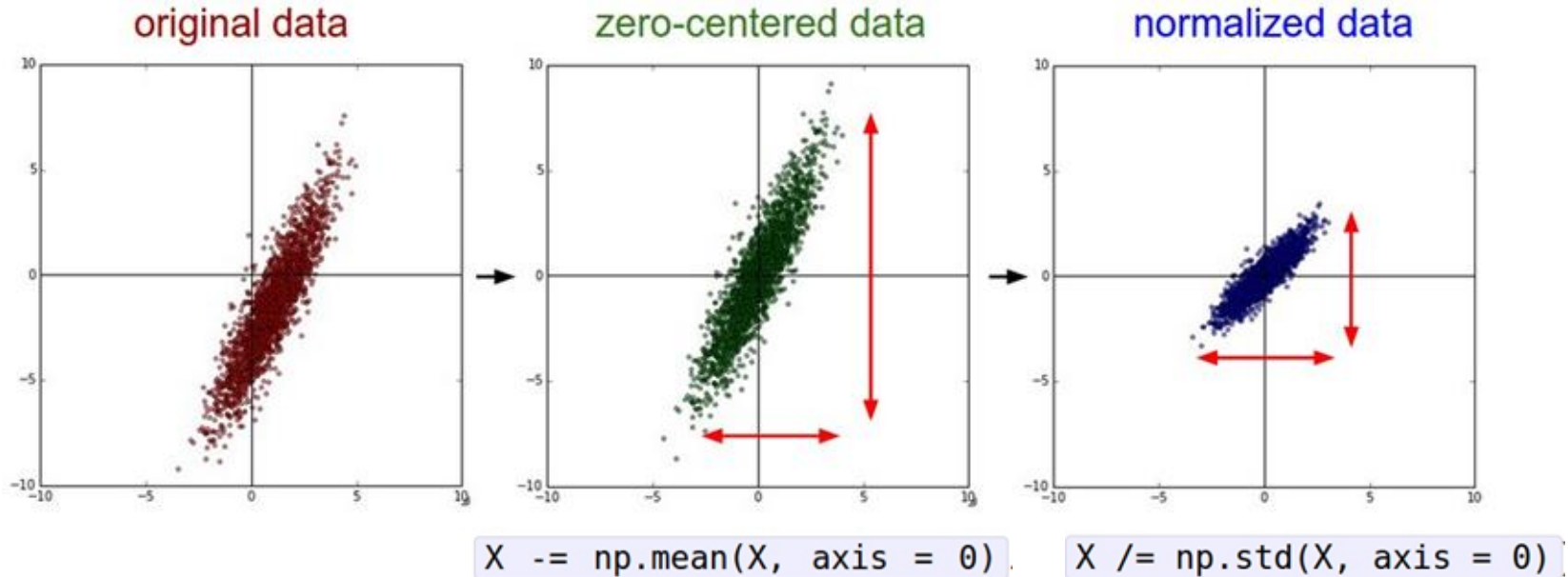
(Klambauer et al, Self-Normalizing Neural Networks, ICLR 2017)

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

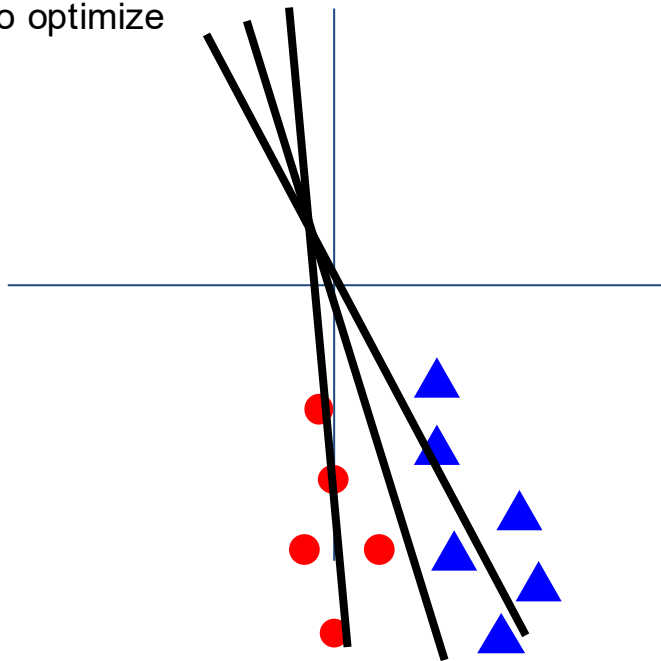
Data Preprocessing



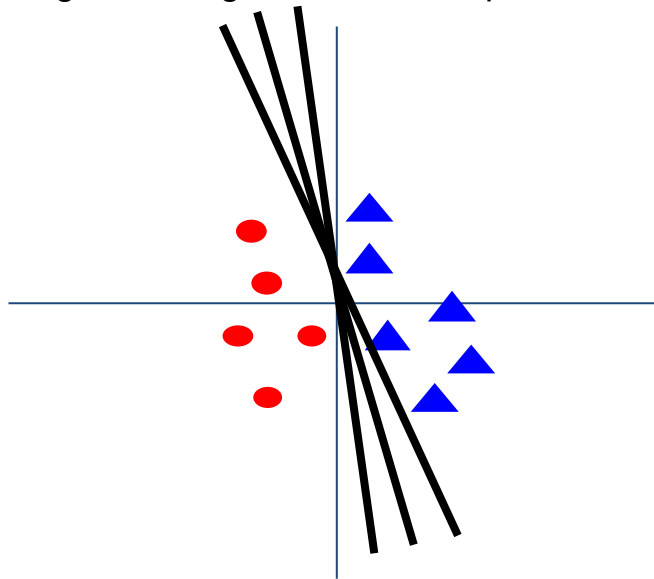
(Assume X [NxD] is data matrix, each example in a row)

Data Preprocessing: example in linear classifier

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

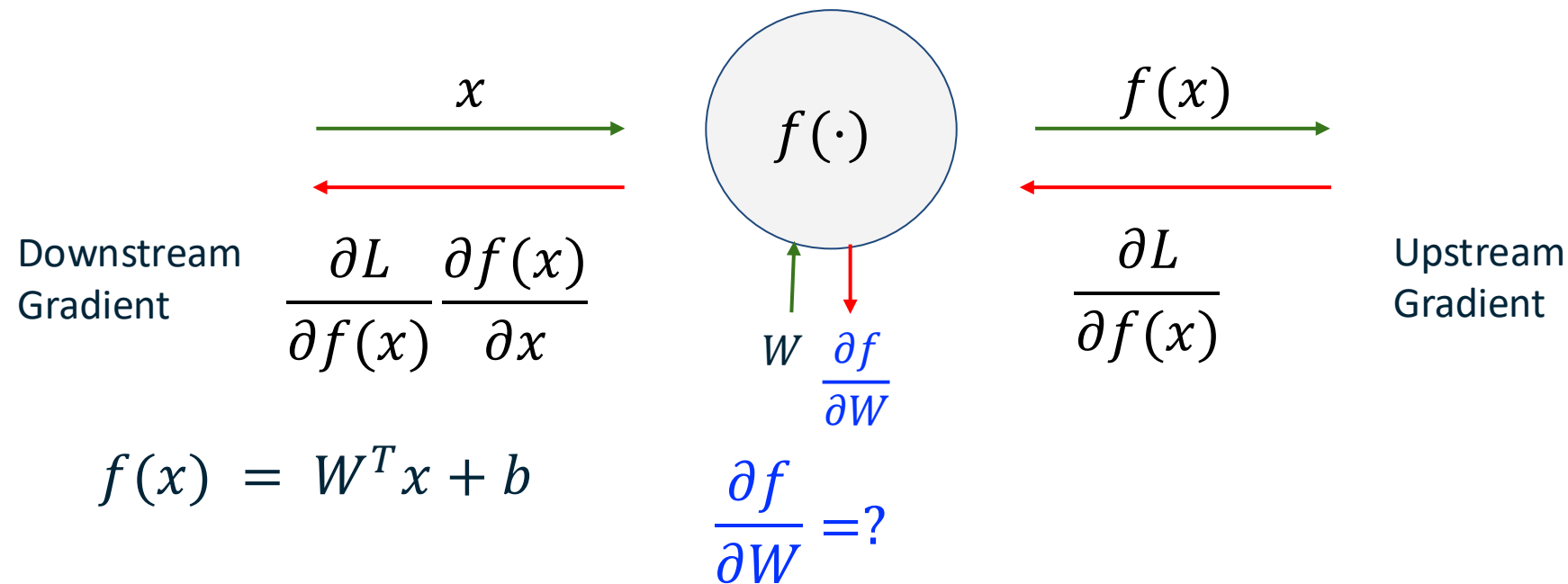


After normalization: less sensitive to small changes in weights; easier to optimize



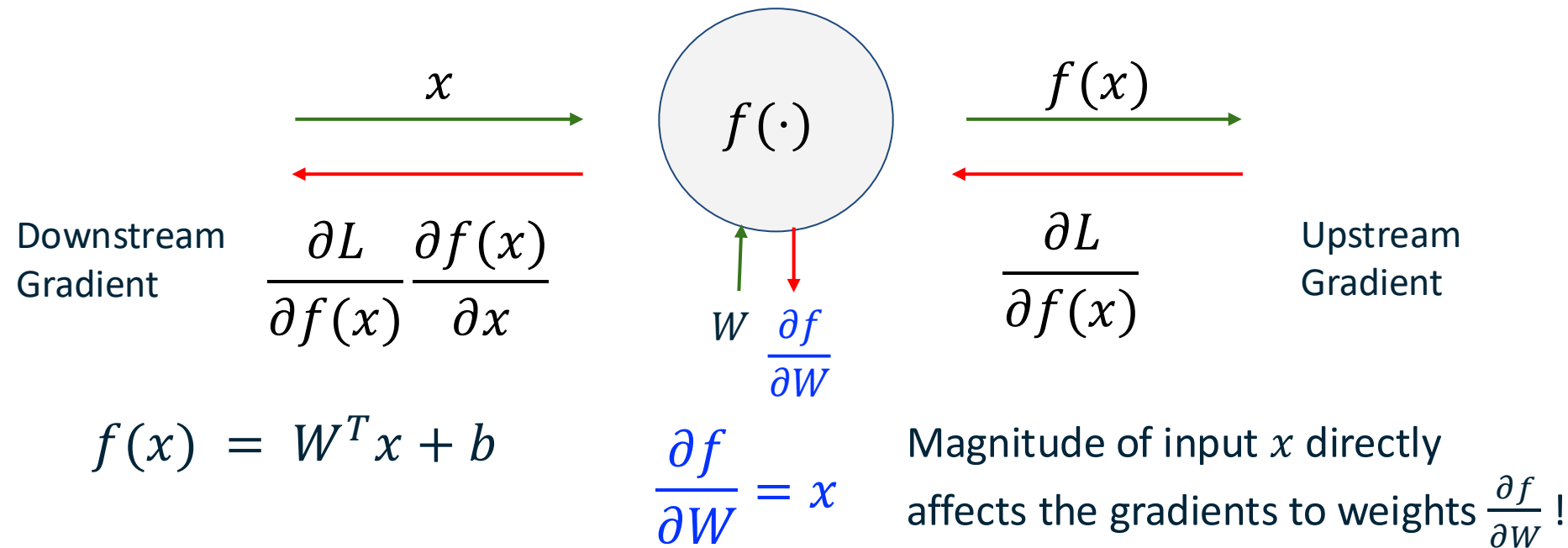
Many different reasons why we might want to normalize the input!

Another example: Input magnitude affects gradient magnitude



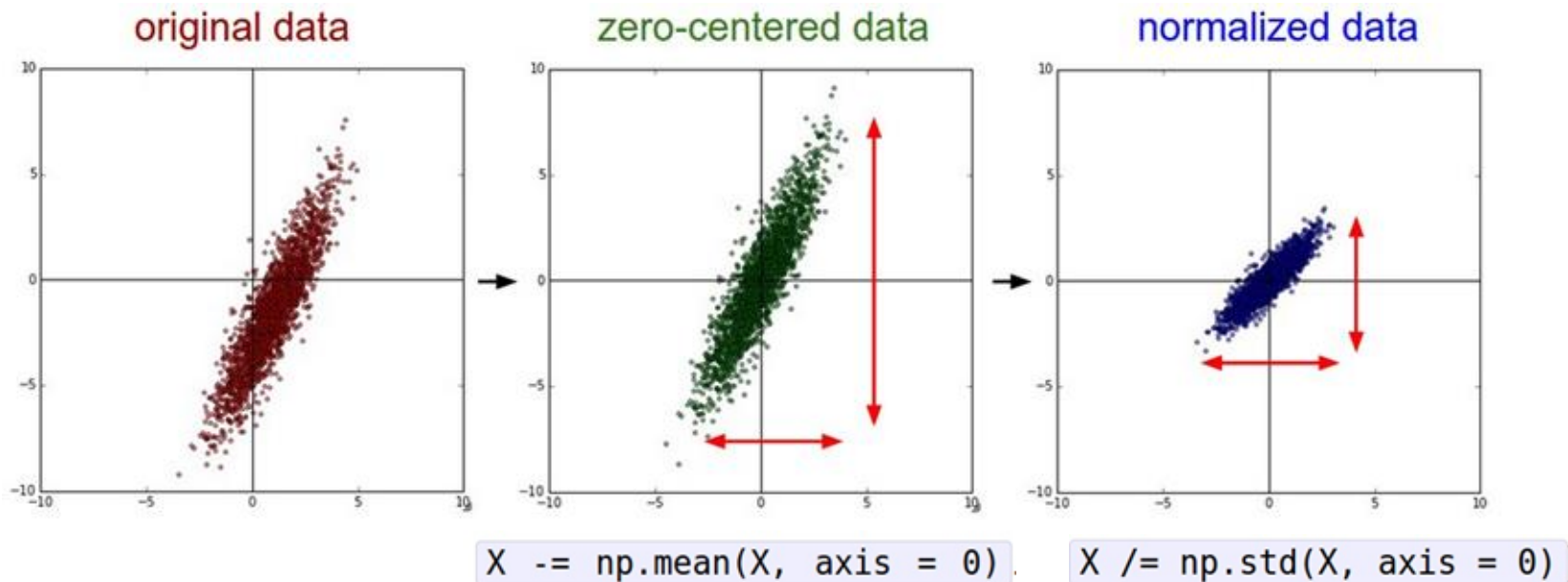
Many different reasons why we might want to normalize the input!

Another example: Input magnitude affects gradient magnitude



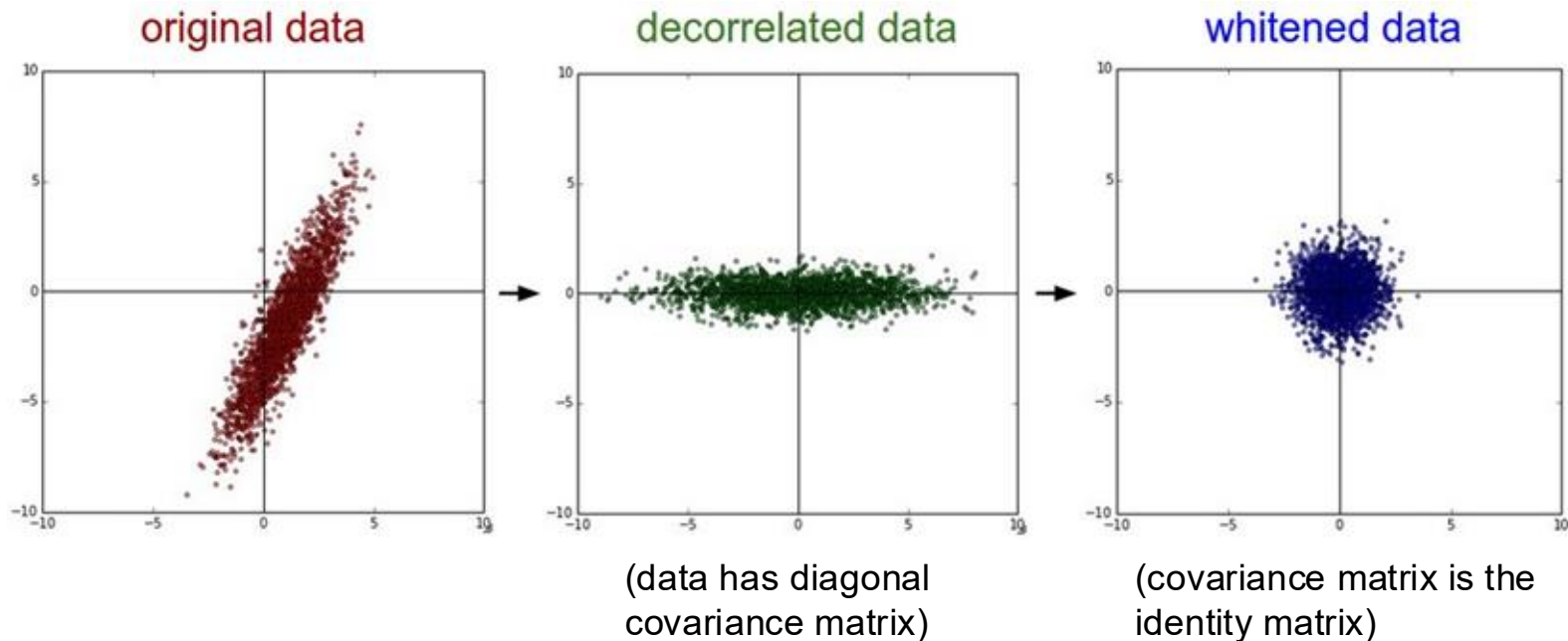
Data Preprocessing

Gaussian normalization is very commonly used



Data Preprocessing

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



Examples: images

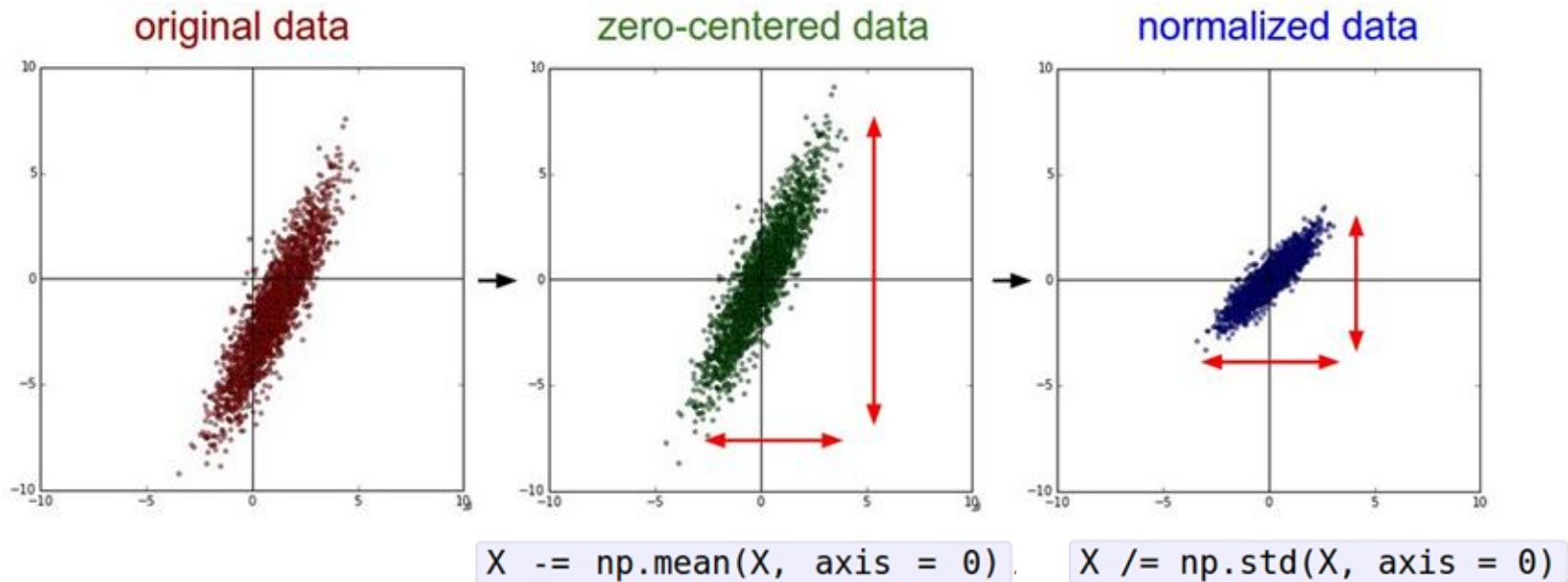
e.g. consider CIFAR-10 example with [32,32,3] images

- Default: Normalize to [0, 1] float (from uint8)
- 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.

Data Preprocessing



What about intermediate features inside the NN after the input?

Batch Normalization

“you want zero-mean unit-variance activations? just make them so.”

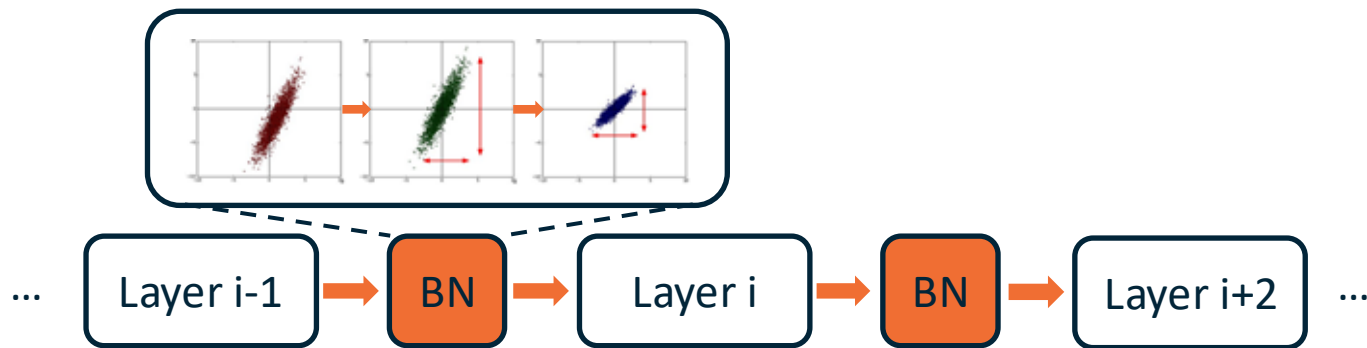
consider a **batch of activations** x at some layer. To make each dimension zero-mean unit-variance, apply:

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

this is a vanilla
differentiable function...

Batch Normalization

“you want zero-mean unit-variance activations? just make them so.”



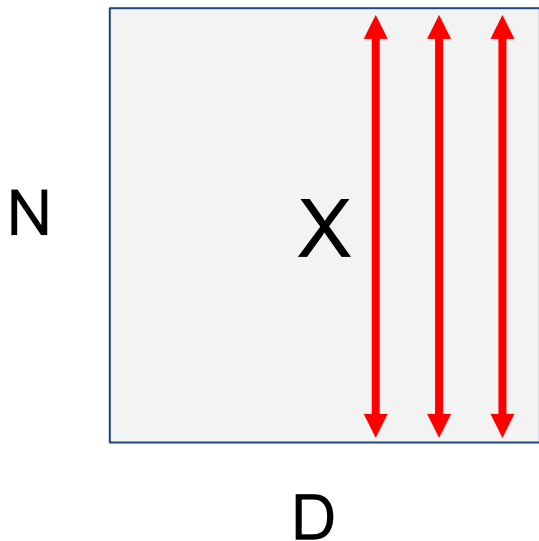
$$\hat{x} = \frac{x - \mathbb{E}[x]}{\sqrt{\text{Var}[x]}}$$

Batch Normalization



[Ioffe and Szegedy, 2015]

Input: $x : N \times D$



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

Per-batch mean,
shape is D

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

Per-batch var,
shape is D

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

Normalized x,
Shape is N x D

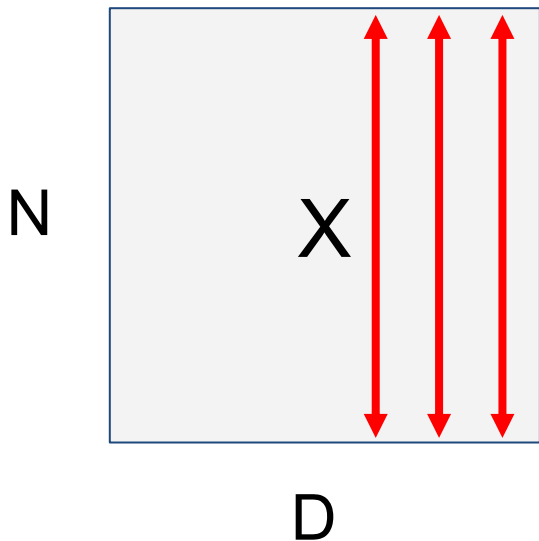
(Prevent div by 0 err)

Batch Normalization



[Ioffe and Szegedy, 2015]

Input: $x : N \times D$



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

Per-batch mean,
shape is D

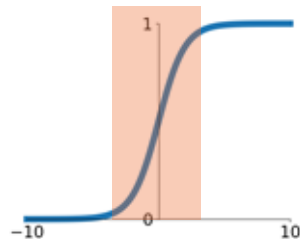
$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (x_{i,j} - \mu_j)^2$$

Per-batch var,
shape is D

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

Normalized x,
Shape is N x D

Problem: What if zero-mean, unit variance is too hard of a constraint?
E.g., inserting a BN before sigmoid will constrain it to (mostly) linear regime

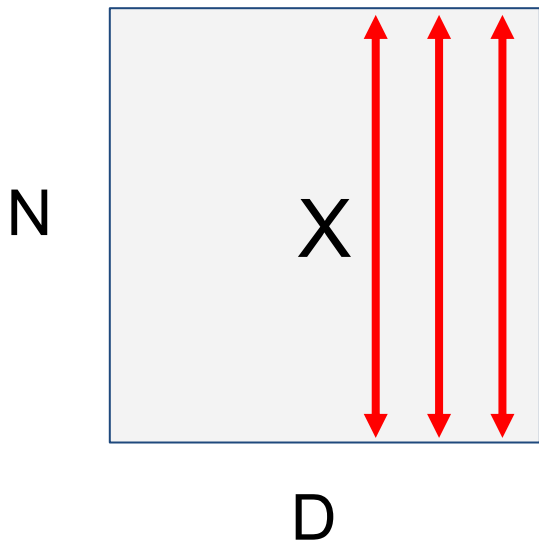


Batch Normalization



[Ioffe and Szegedy, 2015]

Input: $x : N \times D$



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

Per-batch mean,
shape is D

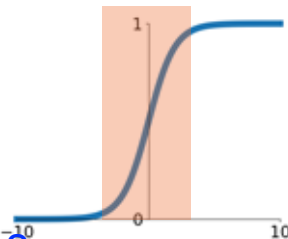
$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (x_{i,j} - \mu_j)^2$$

Per-batch var,
shape is D

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

Normalized x,
Shape is N x D

Problem: What if zero-mean, unit variance is too hard of a constraint?
E.g., inserting a BN before sigmoid will constrain it to (mostly) linear regime
Can we learn the normalization parameters?



Batch Normalization



[Ioffe and Szegedy, 2015]

Input: $x : N \times D$

Learnable scale and shift parameters:

$$\gamma, \beta: \mathbb{R}^D$$

We want to give the model a chance to **adjust batchnorm** if the default is not optimal.

Learning $\gamma = \sigma$ and $\beta = \mu$ will recover the original input batch!

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

Per-batch mean,
shape is D

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

Per-batch var,
shape is D

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

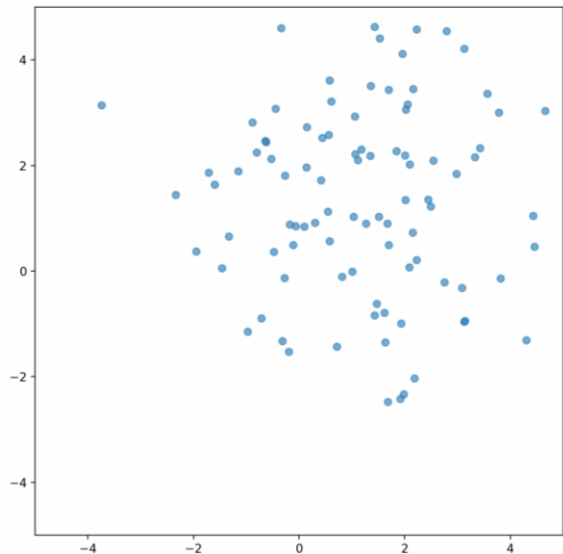
Normalized x,
Shape is N x D

$$y_{i,j} = \underline{\gamma}_j \hat{x}_{i,j} + \underline{\beta}_j$$

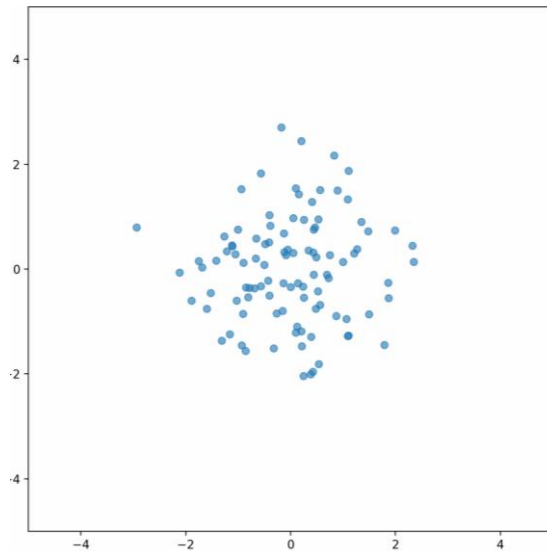
Output,
Shape is N x D

Initialize $\gamma = 1, \beta = 0$

What does it look like?



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



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

$$\gamma = [2, 1.5], \beta = [1, -1]$$

Batch Normalization: Test-Time

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

Input: $x : N \times D$
**Learnable scale and
shift parameters:**

$$\gamma, \beta: \mathbb{R}^D$$

$$\mu_j = \frac{1}{N} \sum_{i=1}^N x_{i,j} \quad \text{Per-batch mean, shape is D}$$
$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (x_{i,j} - \mu_j)^2 \quad \text{Per-batch var, shape is D}$$

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}} \quad \text{Normalized x, Shape is N x D}$$

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j \quad \text{Output, Shape is N x D}$$

Batch Normalization: Test-Time

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

Input: $x : N \times D$
Learnable scale and shift parameters:

$$\gamma, \beta: \mathbb{R}^D$$

Activations become fixed after training. Can calculate training set-wide statistics for inference-time normalization.

At training time, do moving average to save compute.

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

Per-batch mean,
shape is D

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

Per-batch var,
shape is D

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

Normalized x,
Shape is N x D

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

Output,
Shape is N x D

Batch Normalization: Test-Time

Input: $x : N \times D$
Learnable scale and shift parameters:

$$\gamma, \beta : \mathbb{R}^D$$

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

$$\mu_j = \text{(Moving) average of values seen during training}$$

Per-batch mean, shape is D

$$\sigma_j^2 = \text{(Moving) average of values seen during training}$$

Per-batch var, shape is D

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

Normalized x,
Shape is N x D

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

Output,
Shape is N x D

```
import numpy as np

class BatchNorm:
    def __init__(self, num_features, eps=1e-5, momentum=0.1):
        self.num_features = num_features
        self.eps = eps
        self.momentum = momentum

        # Learnable parameters
        self.gamma = np.ones(num_features) # Scale parameter
        self.beta = np.zeros(num_features) # Shift parameter

        # Running statistics (used for inference)
        self.running_mean = np.zeros(num_features)
        self.running_var = np.ones(num_features)
```

You can think of gamma and beta as the layer parameters

```
import numpy as np
```

```
class BatchNorm:
```

```
    def __init__(self, num_features, eps=1e-5, momentum=0.1):  
        self.num_features = num_features  
        self.eps = eps  
        self.momentum = momentum
```

```
        # Learnable parameters
```

```
        self.gamma = np.ones(num_features) # Scale parameter
```

```
        self.beta = np.zeros(num_features) # Shift parameter
```

```
        # Running statistics (used for inference)
```

```
        self.running_mean = np.zeros(num_features)
```

```
        self.running_var = np.ones(num_features)
```

```
    def forward(self, X, training=True):
```

```
        if training:
```

```
            # Training mode
```

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

```
            batch_var = np.var(X, axis=0)
```

```
            # Update running statistics
```

```
            self.running_mean = (1 - self.momentum) * self.running_mean + self.momentum * batch_mean
```

```
            self.running_var = (1 - self.momentum) * self.running_var + self.momentum * batch_var
```

```
            # Normalize
```

```
            X_norm = (X - batch_mean) / np.sqrt(batch_var + self.eps)
```

Use batch statistics during training

Keep running dataset statistics

```
import numpy as np
```

```
class BatchNorm:
```

```
    def __init__(self, num_features, eps=1e-5, momentum=0.1):  
        self.num_features = num_features  
        self.eps = eps  
        self.momentum = momentum
```

```
        # Learnable parameters
```

```
        self.gamma = np.ones(num_features) # Scale parameter  
        self.beta = np.zeros(num_features) # Shift parameter
```

```
        # Running statistics (used for inference)
```

```
        self.running_mean = np.zeros(num_features)  
        self.running_var = np.ones(num_features)
```

```
    def forward(self, X, training=True):
```

```
        if training:
```

```
            # Training mode
```

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

```
            batch_var = np.var(X, axis=0)
```

```
            # Update running statistics
```

```
            self.running_mean = (1 - self.momentum) * self.running_mean + self.momentum * batch_mean
```

```
            self.running_var = (1 - self.momentum) * self.running_var + self.momentum * batch_var
```

```
            # Normalize
```

```
            X_norm = (X - batch_mean) / np.sqrt(batch_var + self.eps)
```

```
        else:
```

```
            # Inference mode
```

```
            X_norm = (X - self.running_mean) / np.sqrt(self.running_var + self.eps)
```

```
        # Scale and shift
```

```
        return self.gamma * X_norm + self.beta
```

Use running statistics during testing

Apply learned scale and shift parameters

Batch Normalization

[Ioffe and Szegedy, 2015]

Q: Should you put batchnorm before or after ReLU?

A: Topic of debate. Original paper says BN→ReLU. Now most commonly ReLU→BN. If BN→ReLU and zero mean, ReLU kills half of the activations, but in practice makes insignificant differences.

Q: Should you normalize the **input** (e.g., images) with batchnorm?

A: No, you already have the fixed & correct dataset statistics, no need to do batchnorm.

Q: How many parameters does a batchnorm layer have?

A: Input dimension * 4: beta, gamma, moving average mu, moving average sigma. Only beta and gamma are trainable parameters.

Batch Normalization

[Ioffe and Szegedy, 2015]

- Makes deep networks **much** easier to train!
 - If you are interested in the theory, read <https://arxiv.org/abs/1805.11604>
 - TL;DR: makes optimization landscape smoother
- Allows higher learning rates, faster convergence
- More useful in deeper networks
- Networks become more robust to initialization
- More robust to range of input
- Zero overhead at test-time: can be fused with conv!
- Behaves differently during training and testing: this is a very common source of bugs!
- Needs large batch size to calculate accurate stats

Batch Normalization for ConvNets

Batch Normalization for
fully-connected networks

$\mathbf{x}: \mathbf{N} \times \mathbf{D}$

Normalize 

$\boldsymbol{\mu}, \boldsymbol{\sigma}: 1 \times \mathbf{D}$

$\boldsymbol{\gamma}, \boldsymbol{\beta}: 1 \times \mathbf{D}$

$$\mathbf{y} = \boldsymbol{\gamma} (\mathbf{x} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \boldsymbol{\beta}$$

Batch Normalization for
convolutional networks
(Spatial Batchnorm, BatchNorm2D)

$\mathbf{x}: \mathbf{N} \times \mathbf{C} \times \mathbf{H} \times \mathbf{W}$

Normalize   

$\boldsymbol{\mu}, \boldsymbol{\sigma}: 1 \times \mathbf{C} \times 1 \times 1$

$\boldsymbol{\gamma}, \boldsymbol{\beta}: 1 \times \mathbf{C} \times 1 \times 1$

$$\mathbf{y} = \boldsymbol{\gamma} (\mathbf{x} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \boldsymbol{\beta}$$

Keep the spatial equivariance property of conv: all locations should be normalized in similar ways

Layer Normalization

Batch Normalization for
fully-connected networks

$$\mathbf{x}: \mathbf{N} \times \mathbf{D}$$

Normalize



$$\boldsymbol{\mu}, \boldsymbol{\sigma}: \mathbf{1} \times \mathbf{D}$$

$$\boldsymbol{\gamma}, \boldsymbol{\beta}: \mathbf{1} \times \mathbf{D}$$

$$\mathbf{y} = \boldsymbol{\gamma}(\mathbf{x} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \boldsymbol{\beta}$$

Layer Normalization for fully-
connected networks

Same behavior at train and test!

$$\mathbf{x}: \mathbf{N} \times \mathbf{D}$$

Normalize



$$\boldsymbol{\mu}, \boldsymbol{\sigma}: \mathbf{N} \times \mathbf{1}$$

$$\boldsymbol{\gamma}, \boldsymbol{\beta}: \mathbf{1} \times \mathbf{D}$$

$$\mathbf{y} = \boldsymbol{\gamma}(\mathbf{x} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \boldsymbol{\beta}$$

Ba, Kiros, and Hinton, "Layer Normalization", arXiv 2016

More flexible (can use $N = 1$!), works well
with sequence models (RNN, Transformers)

Adaptive Layer Normalization

Regular LN:

$$\text{LN}(x) = \gamma \odot \frac{x - \mu(x)}{\sqrt{\sigma^2(x) + \epsilon}} + \beta$$

Adaptive LayerNorm (AdaLN): normalize the activations in the usual way, then let the conditioning signal decide how each feature channel should be scaled and shifted.

$$\text{AdaLN}(x, c) = \gamma(c) \odot \frac{x - \mu(x)}{\sqrt{\sigma^2(x) + \epsilon}} + \beta(c)$$

, where c is typically an instance-specific feature (e.g., time embedding, class label)

$$[\gamma(c), \beta(c)] = \text{MLP}(c).$$

Instance Normalization

Batch Normalization for
convolutional networks

$$\mathbf{x}: \mathbf{N} \times \mathbf{C} \times \mathbf{H} \times \mathbf{W}$$

Normalize



$$\boldsymbol{\mu}, \boldsymbol{\sigma}: 1 \times \mathbf{C} \times 1 \times 1$$

$$\boldsymbol{\gamma}, \boldsymbol{\beta}: 1 \times \mathbf{C} \times 1 \times 1$$

$$\mathbf{y} = \boldsymbol{\gamma} (\mathbf{x} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \boldsymbol{\beta}$$

Instance Normalization for
convolutional networks

Same behavior at train / test!

$$\mathbf{x}: \mathbf{N} \times \mathbf{C} \times \mathbf{H} \times \mathbf{W}$$

Normalize

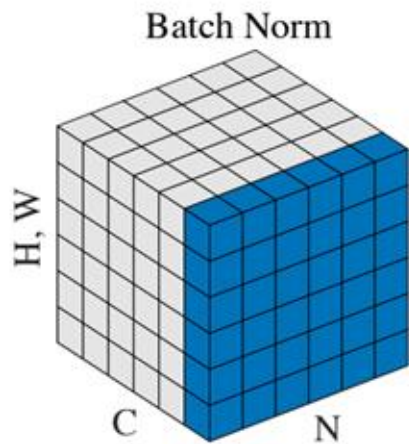


$$\boldsymbol{\mu}, \boldsymbol{\sigma}: \mathbf{N} \times \mathbf{C} \times 1 \times 1$$

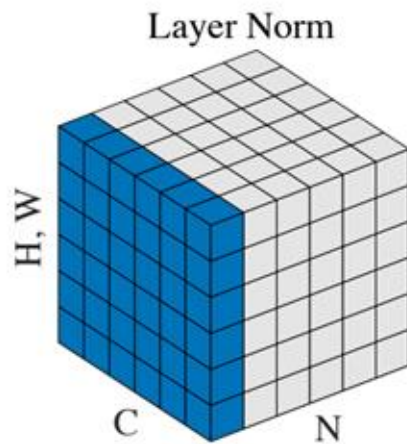
$$\boldsymbol{\gamma}, \boldsymbol{\beta}: 1 \times \mathbf{C} \times 1 \times 1$$

$$\mathbf{y} = \boldsymbol{\gamma} (\mathbf{x} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \boldsymbol{\beta}$$

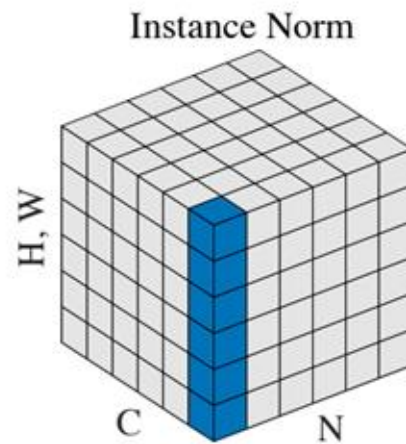
Comparison of Normalization Layers



$N \times C \times H \times W \rightarrow$
 $1 \times C \times 1 \times 1$

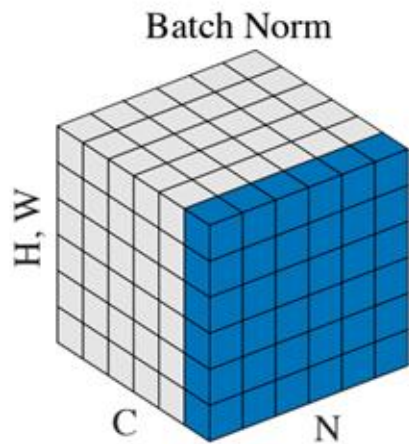


$N \times C \times H \times W \rightarrow$
 $N \times 1 \times 1 \times 1$

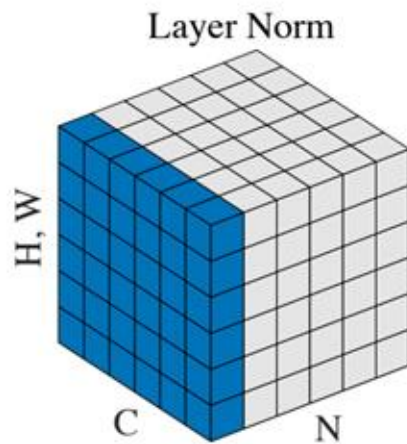


$N \times C \times H \times W \rightarrow$
 $N \times C \times 1 \times 1$

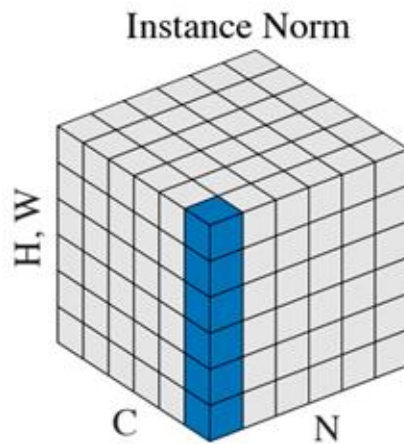
Group Normalization



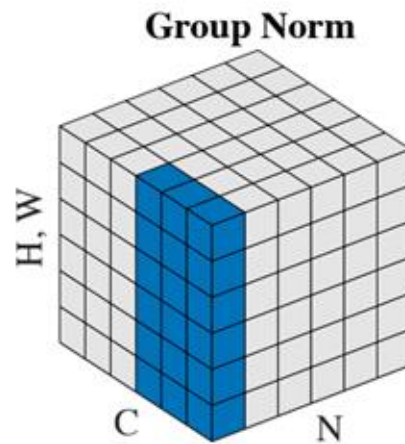
$N \times C \times H \times W \rightarrow$
 $1 \times C \times 1 \times 1$



$N \times C \times H \times W \rightarrow$
 $N \times 1 \times 1 \times 1$



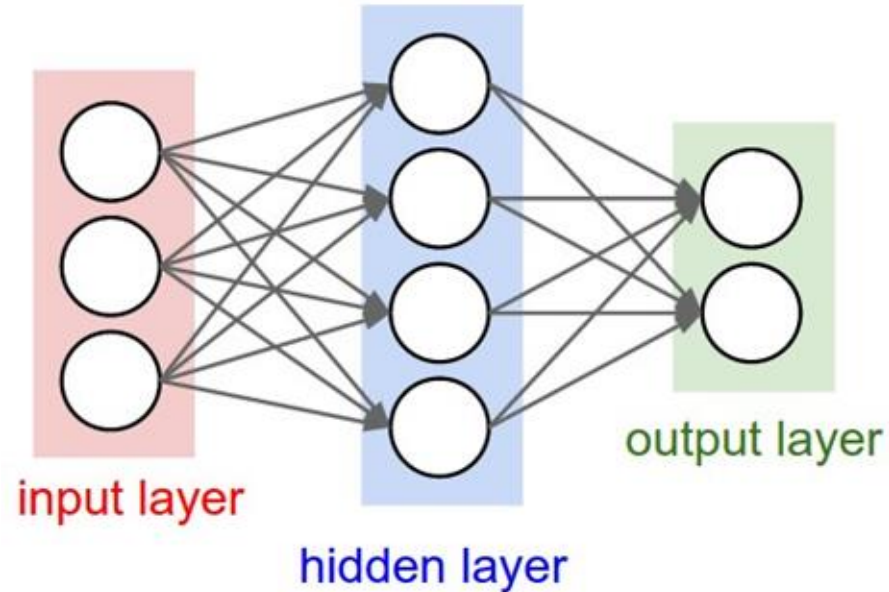
$N \times C \times H \times W \rightarrow$
 $N \times C \times 1 \times 1$



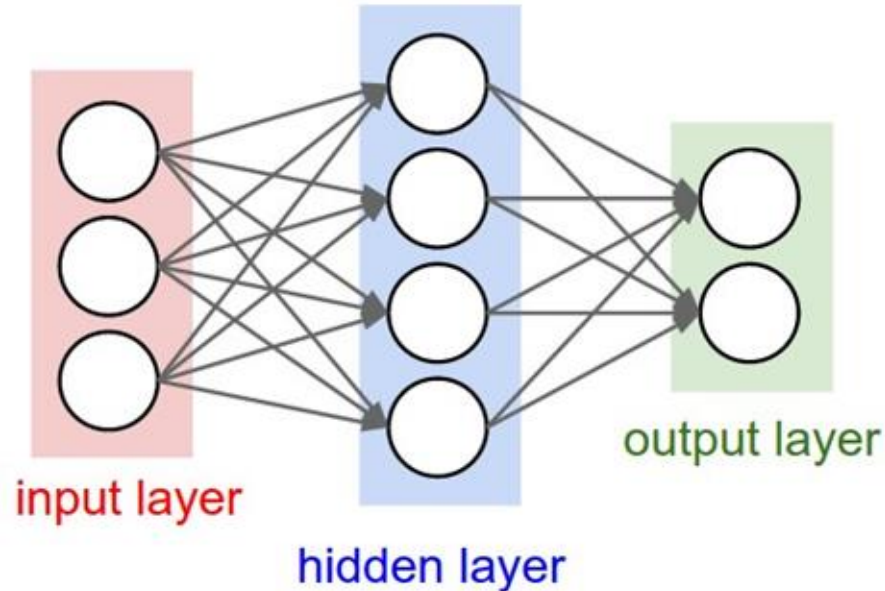
$N \times C \times H \times W \rightarrow$
 $N \times C/G \times 1 \times 1$

Weight Initialization

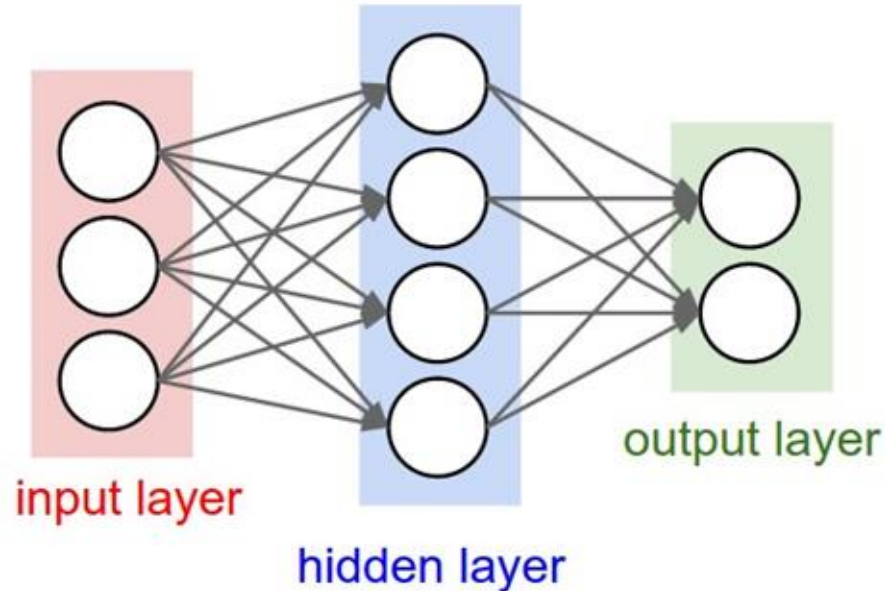
- Q: what happens when W =same initial value is used?



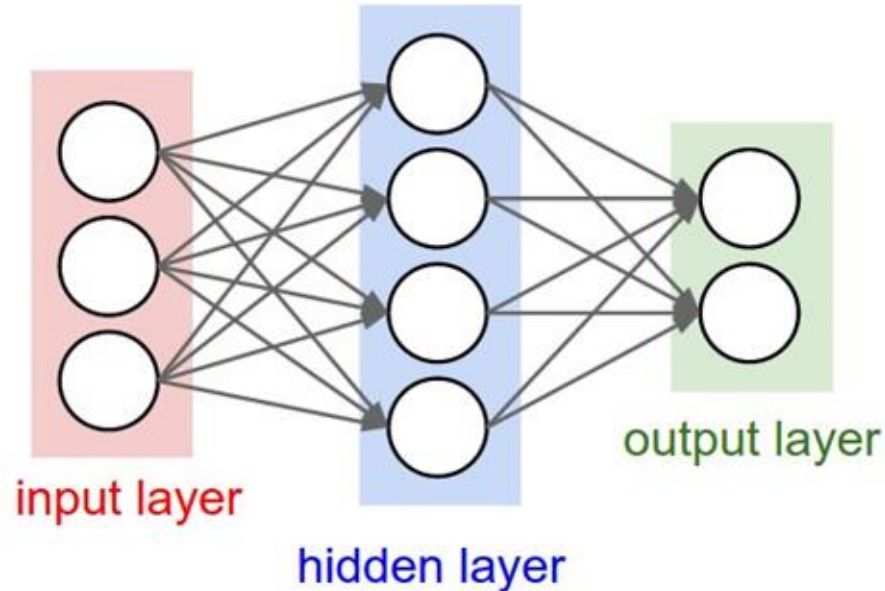
- Q: what happens when W =same initial value is used?
- A: All output will be the same! $w_1^T x = w_2^T x$ if $w_1 = w_2$



- 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

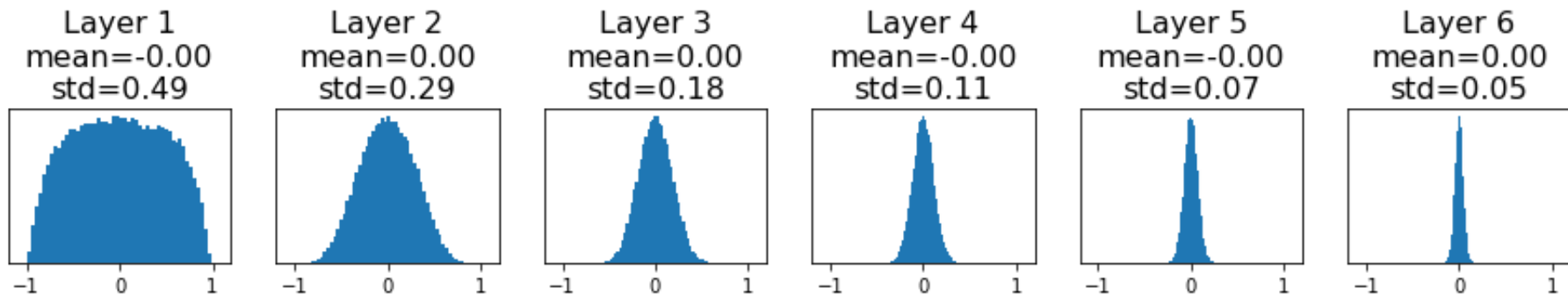
```
dims = [4096] * 7      Forward pass for a 6-layer
hs = []                net with hidden size 4096
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

What will happen to the activations for the last layer?

Weight Initialization: Activation statistics

```
dims = [4096] * 7      Forward pass for a 6-layer
hs = []                net with hidden size 4096
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)
```

All activations tend to zero
for deeper network layers



Visualize distribution of activations

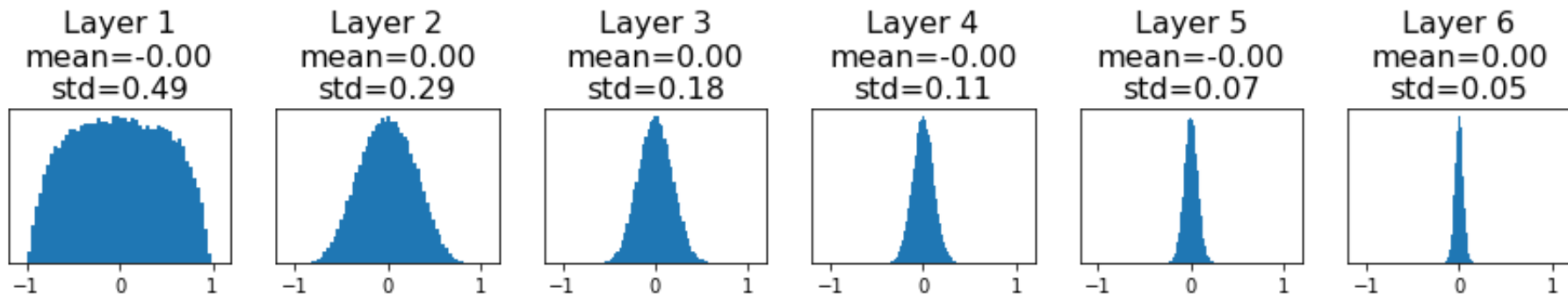
Weight Initialization: Activation statistics

```
dims = [4096] * 7      Forward pass for a 6-layer
hs = []                net with hidden size 4096
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)
```

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

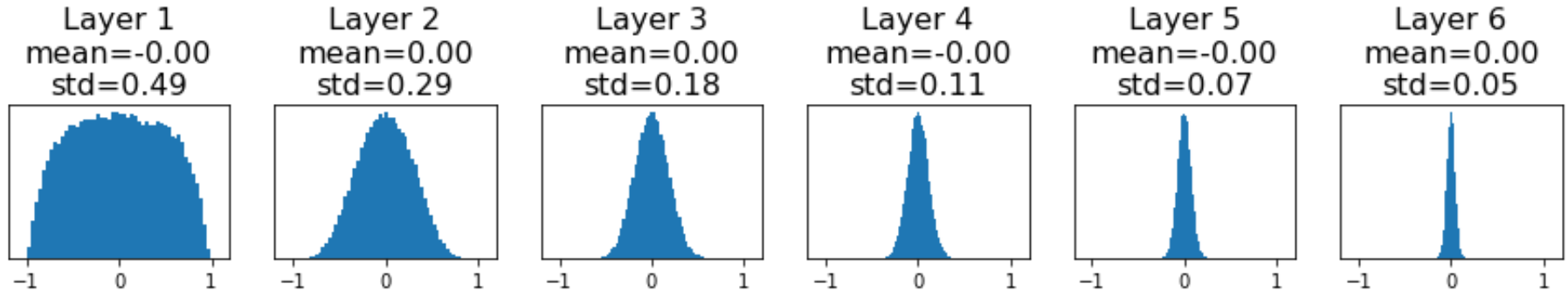
Weight Initialization: Activation statistics

```
dims = [4096] * 7      Forward pass for a 6-layer
hs = []                net with hidden size 4096
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)
```

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    Increase std of initial
hs = []              weights from 0.01 to 0.05
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)
```

Initialize with higher values

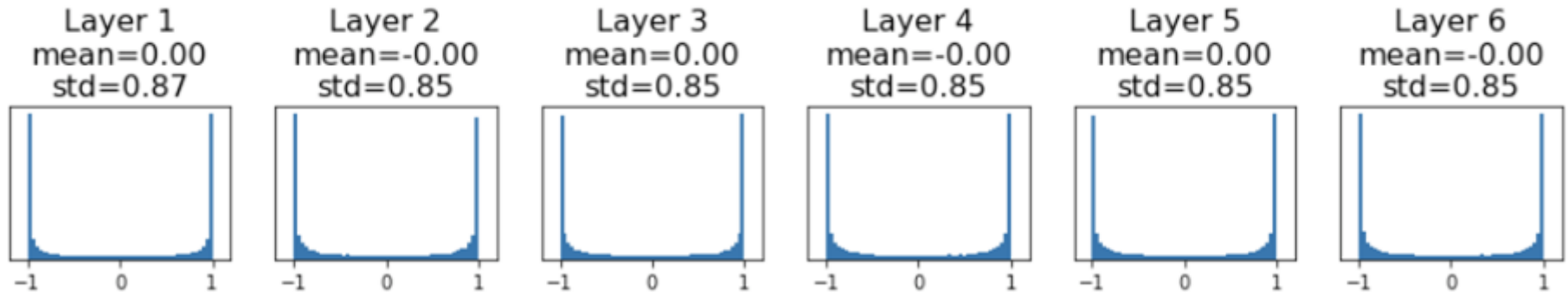
What will happen to the activations for the last layer?

Weight Initialization: Activation statistics

```
dims = [4096] * 7      Increase std of initial
hs = []                weights from 0.01 to 0.05
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)
```

All activations saturate

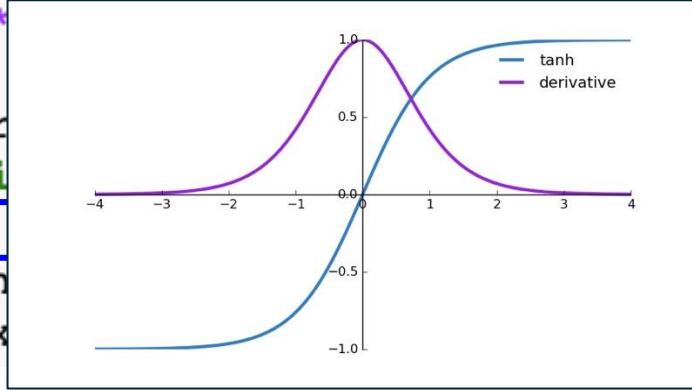
Q: What do the gradients look like?



Visualize distribution of activations

Weight Initialization: Activation statistics

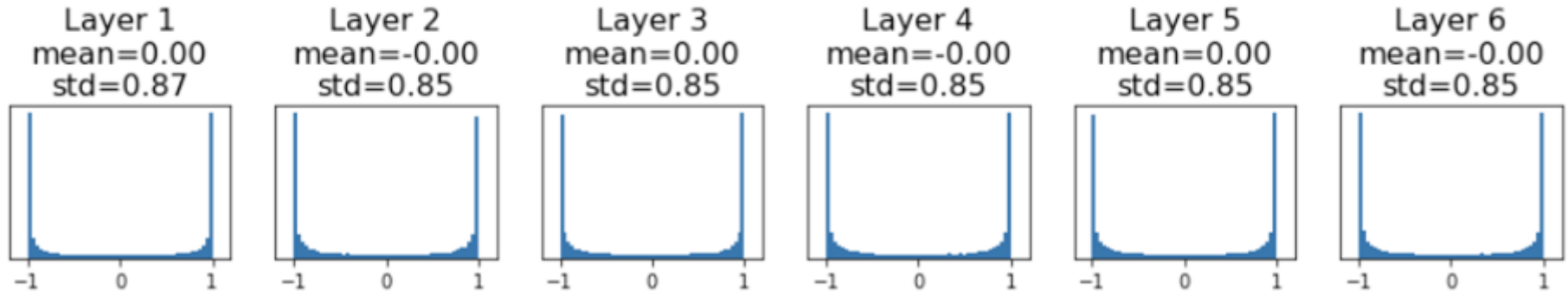
```
dims = [4096] *  
hs = []  
x = np.random.r  
for Din, Dout i  
    W = 0.05 *  
    x = np.tanh  
    hs.append(x
```



All activations saturate

Q: What do the gradients look like?

A: For tanh, large value $\rightarrow$ small gradient



Visualize distribution of activations

Weight Initialization: Activation statistics

```
dims = [4096] * 7    Increase std of initial
hs = []              weights from 0.01 to 0.05
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)
```

All activations saturate

Q: What do the gradients look like?

More generally, *gradient explosion* (high $w \rightarrow$ high output $\rightarrow$ high gradient).

Visualize distribution of activations

Weight Initialization: “Xavier” Initialization

```
dims = [4096] * 7          “Xavier” initialization:
hs = []                    std = 1/sqrt(Din)
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)
```

Goal: try to keep **similar variances/std** across layers

Assume each input contribute similarly to output

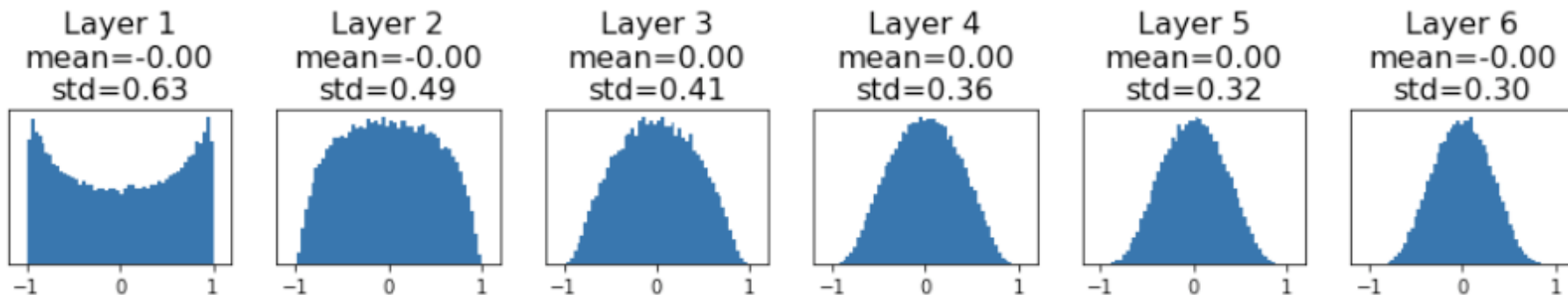
more number of weights needs -> small weight multiplier

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:
 $\text{std} = 1/\sqrt{\text{Din}}$

“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

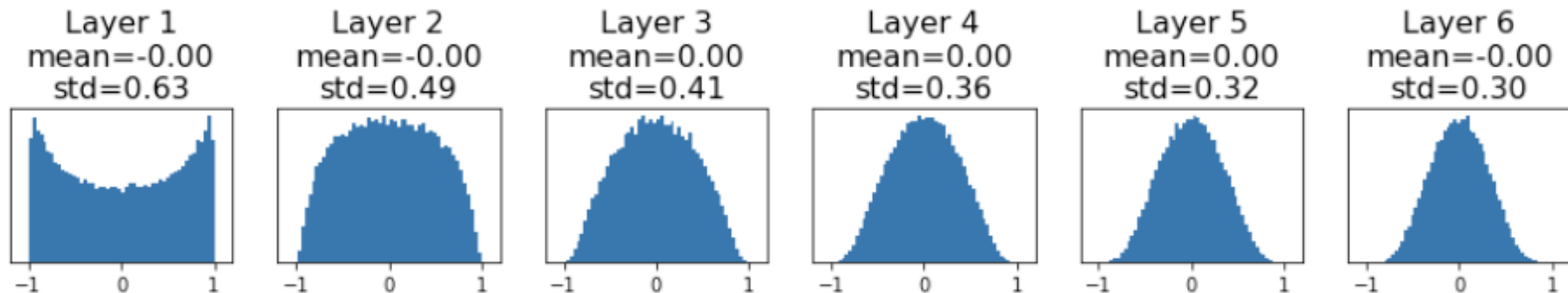
Weight Initialization: “Xavier” Initialization

```
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:
 $\text{std} = 1/\sqrt{D_{\text{in}}}$

“Just right”: Activations are nicely scaled for all layers!

For conv layers, D_{in} is $\text{filter_size}^2 * \text{input_channels}$



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 = $1/\sqrt{D_{in}}$

“Just right”: Activations are nicely scaled for all layers!

For conv layers, D_{in} is $\text{filter_size}^2 * \text{input_channels}$

Let: $y = x_1 w_1 + x_2 w_2 + \dots + x_{D_{in}} w_{D_{in}}$

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 = $1/\sqrt{D_{in}}$

“Just right”: Activations are nicely scaled for all layers!

For conv layers, D_{in} is $\text{filter_size}^2 * \text{input_channels}$

Let: $y = x_1 w_1 + x_2 w_2 + \dots + x_{D_{in}} w_{D_{in}}$

Assume: $\text{Var}(x_1) = \text{Var}(x_2) = \dots = \text{Var}(x_{D_{in}})$

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 = $1/\sqrt{D_{in}}$

“Just right”: Activations are nicely scaled for all layers!

For conv layers, D_{in} is $\text{filter_size}^2 * \text{input_channels}$

Let: $y = x_1 w_1 + x_2 w_2 + \dots + x_{D_{in}} w_{D_{in}}$

Assume: $\text{Var}(x_1) = \text{Var}(x_2) = \dots = \text{Var}(x_{D_{in}})$

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

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 = $1/\sqrt{D_{in}}$

“Just right”: Activations are nicely scaled for all layers!

For conv layers, D_{in} is $\text{filter_size}^2 * \text{input_channels}$

Let: $y = x_1 w_1 + x_2 w_2 + \dots + x_{D_{in}} w_{D_{in}}$

Assume: $\text{Var}(x_1) = \text{Var}(x_2) = \dots = \text{Var}(x_{D_{in}})$

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

$\text{Var}(y) = \text{Var}(x_1 w_1 + x_2 w_2 + \dots + x_{D_{in}} w_{D_{in}})$
[substituting value of y]

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 = $1/\sqrt{D_{in}}$

“Just right”: Activations are nicely scaled for all layers!

For conv layers, D_{in} is $\text{filter_size}^2 * \text{input_channels}$

Let: $y = x_1 w_1 + x_2 w_2 + \dots + x_{D_{in}} w_{D_{in}}$

Assume: $\text{Var}(x_1) = \text{Var}(x_2) = \dots = \text{Var}(x_{D_{in}})$

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

$\text{Var}(y) = \text{Var}(x_1 w_1 + x_2 w_2 + \dots + x_{D_{in}} w_{D_{in}})$

$= \sum \text{Var}(x_i w_i) = D_{in} \text{Var}(x_i w_i)$

[Assume all x_i, w_i are iid] $\sigma_{X+Y}^2 = \sigma_X^2 + \sigma_Y^2$

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 = $1/\sqrt{D_{in}}$

“Just right”: Activations are nicely scaled for all layers!

For conv layers, D_{in} is $\text{filter_size}^2 * \text{input_channels}$

Let: $y = x_1 w_1 + x_2 w_2 + \dots + x_{D_{in}} w_{D_{in}}$

Assume: $\text{Var}(x_1) = \text{Var}(x_2) = \dots = \text{Var}(x_{D_{in}})$

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

$$\begin{aligned}\text{Var}(y) &= \text{Var}(x_1 w_1 + x_2 w_2 + \dots + x_{D_{in}} w_{D_{in}}) \\ &= D_{in} \text{Var}(x_i w_i) \\ &= D_{in} \text{Var}(x_i) \text{Var}(w_i)\end{aligned}$$

[Assume all x_i, w_i are zero mean]

$$\begin{aligned}\text{Var}(XY) &= E(X^2 Y^2) - (E(XY))^2 = \text{Var}(X)\text{Var}(Y) + \cancel{\text{Var}(X)(E(Y))^2} \\ &\quad + \cancel{\text{Var}(Y)(E(X))^2}\end{aligned}$$

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 = $1/\sqrt{D_{in}}$

“Just right”: Activations are nicely scaled for all layers!

For conv layers, D_{in} is $\text{filter_size}^2 * \text{input_channels}$

Let: $y = x_1 w_1 + x_2 w_2 + \dots + x_{D_{in}} w_{D_{in}}$

Assume: $\text{Var}(x_1) = \text{Var}(x_2) = \dots = \text{Var}(x_{D_{in}})$

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

$$\begin{aligned}\text{Var}(y) &= \text{Var}(x_1 w_1 + x_2 w_2 + \dots + x_{D_{in}} w_{D_{in}}) \\ &= D_{in} \text{Var}(x_i w_i) \\ &= D_{in} \text{Var}(x_i) \text{Var}(w_i) \\ &[\text{Assume all } x_i, w_i \text{ are iid}]\end{aligned}$$

So, $\text{Var}(y) = \text{Var}(x_i)$ only when $\text{Var}(w_i) = 1/D_{in}$

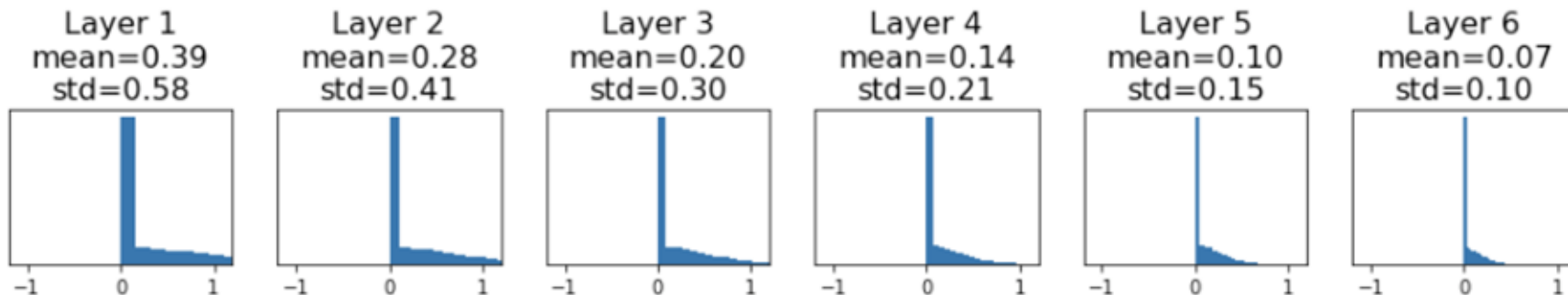
Weight Initialization: What about ReLU?

```
dims = [4096] * 7      Change from tanh to ReLU
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)
```

Weight Initialization: What about ReLU?

```
dims = [4096] * 7      Change from tanh to ReLU
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)
```

Xavier assumes zero centered activation function



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
- Batch normalization
 - BN, Instance Norm, Layer Norm, Group Norm
- Weight Initialization
 - Constant init, random init, Xavier Init, Kaiming Init