DATA ANALYTICS USING DEEP LEARNING

GT 8803 // FALL 2019 // JOY ARULRAJ

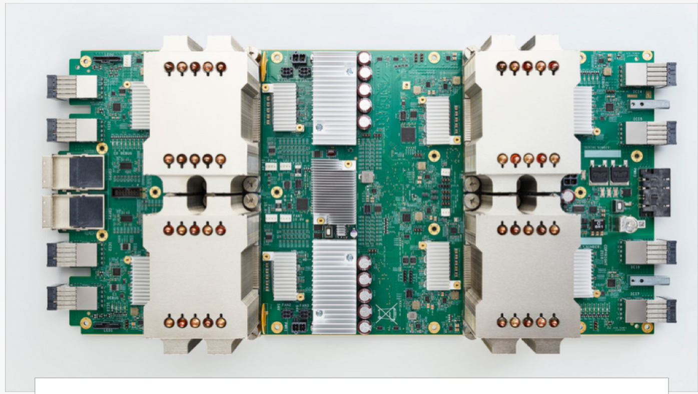
LECTURE #12:TRAINING NEURAL NETWORKS (PT 1)

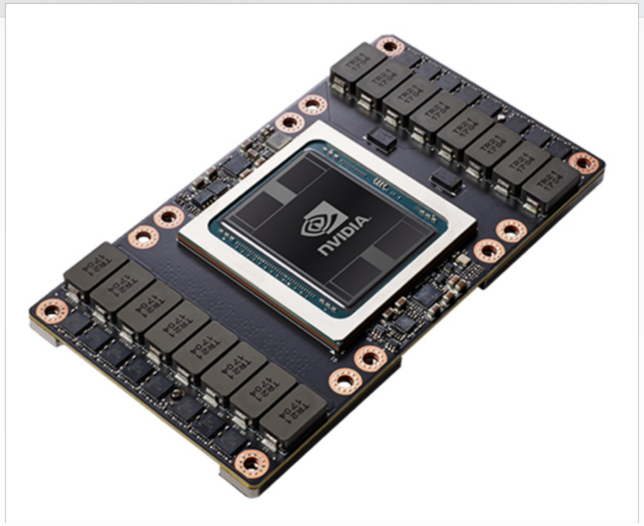
ADMINISTRIVIA

- Reminders
  - Integration with Eva
  - Code reviews
  - Each team must send Pull Requests to Eva

WHERE WE ARE NOW...

HARDWARE + SOFTWARE





PyTorch

TensorFlow

OVERVIEW

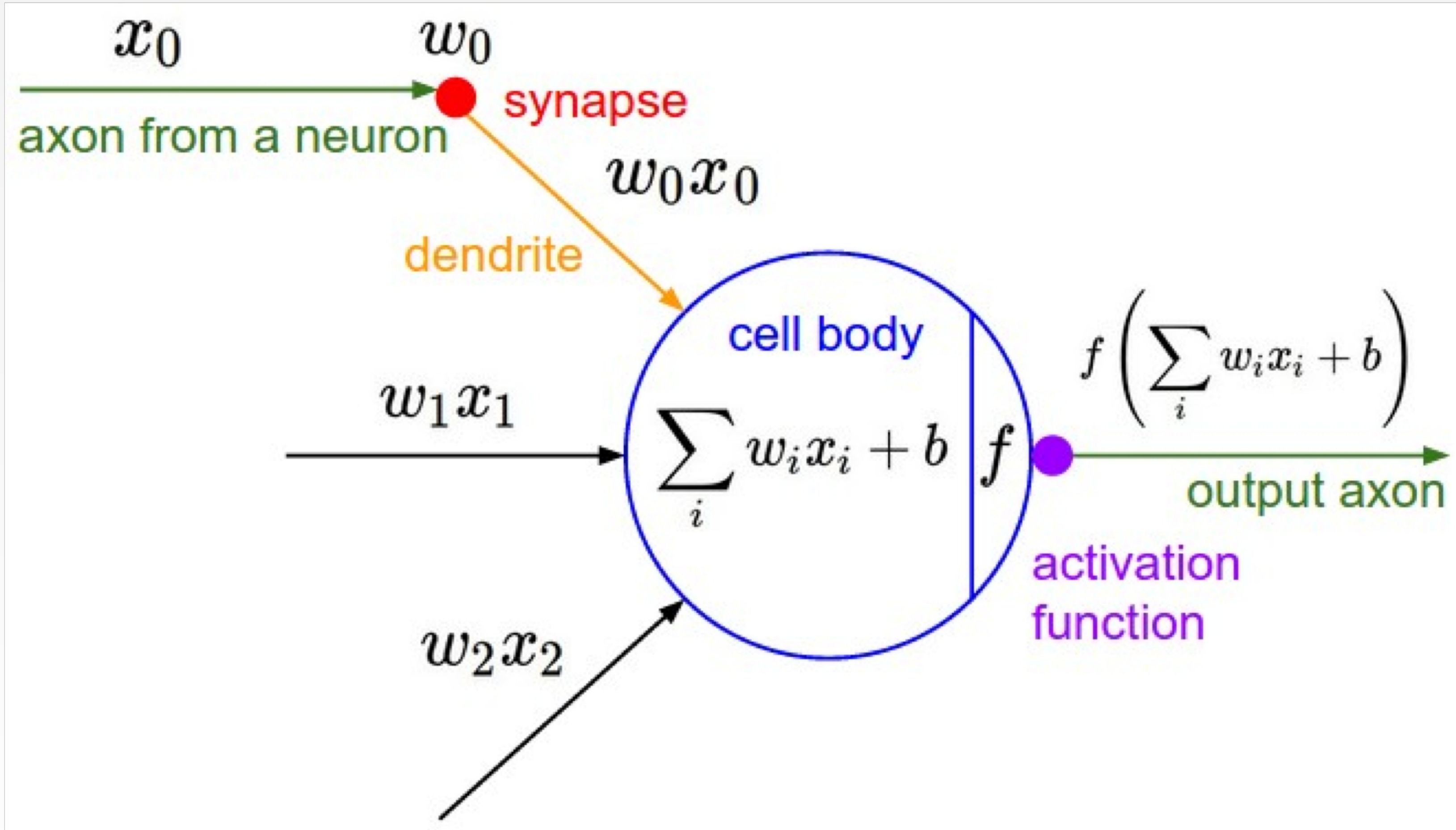
- **One time setup**
  - Activation Functions, Preprocessing, Weight Initialization, Regularization, Gradient Checking
- **Training dynamics**
  - Babysitting the Learning Process, Parameter updates, Hyperparameter Optimization
- **Evaluation**
  - Model ensembles, Test-time augmentation

TODAY'S AGENDA

- Training Neural Networks
  - Activation Functions
  - Data Preprocessing
  - Weight Initialization
  - Batch Normalization

ACTIVATION FUNCTIONS

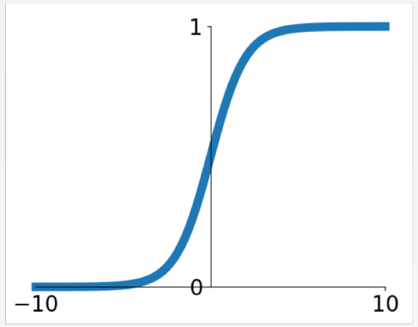
ACTIVATION FUNCTIONS



ACTIVATION FUNCTIONS

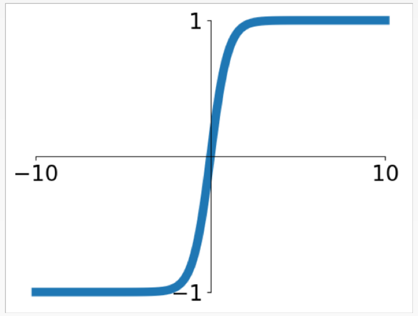
Sigmoid

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



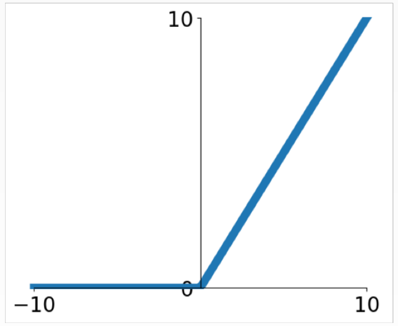
tanh

$\tanh(x)$



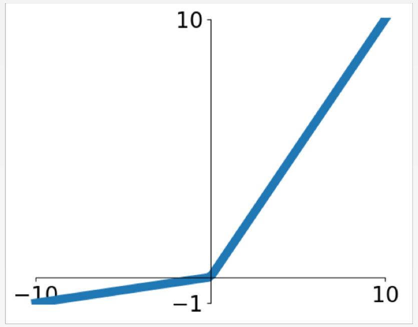
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

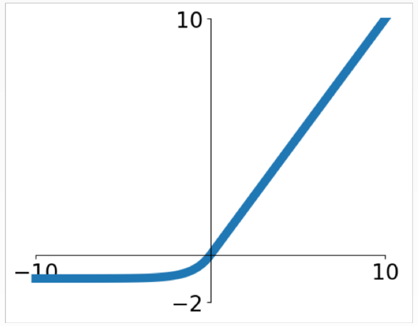


Maxout

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

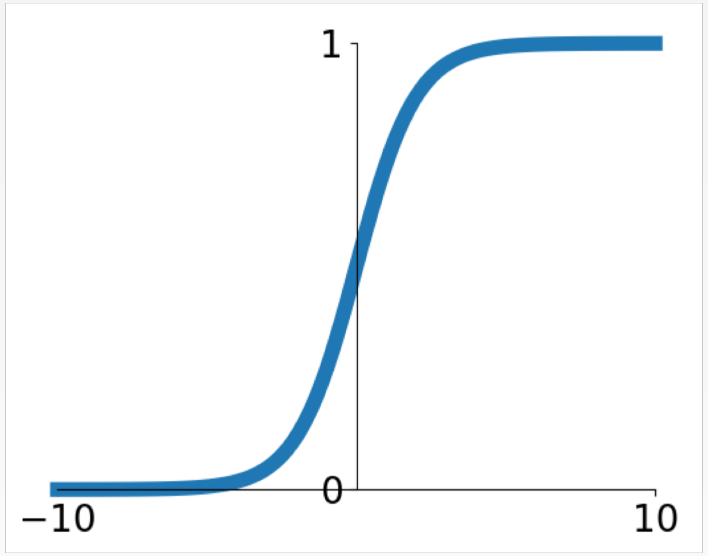
ELU

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



ACTIVATION FUNCTIONS

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

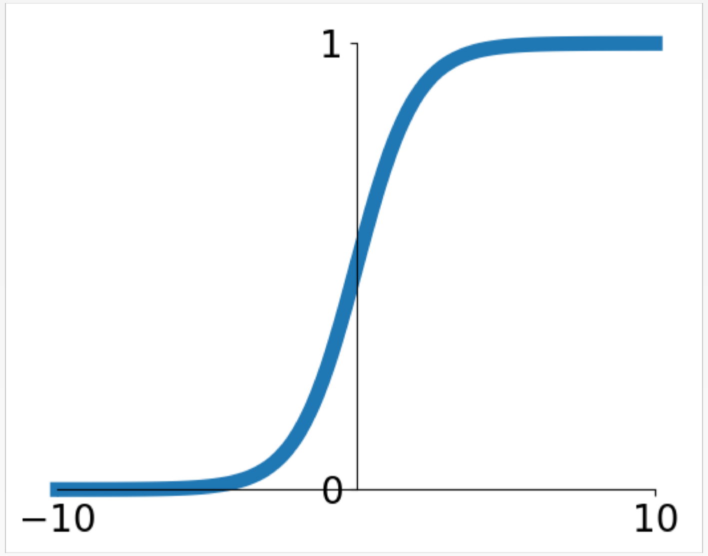


Sigmoid

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

ACTIVATION FUNCTIONS

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

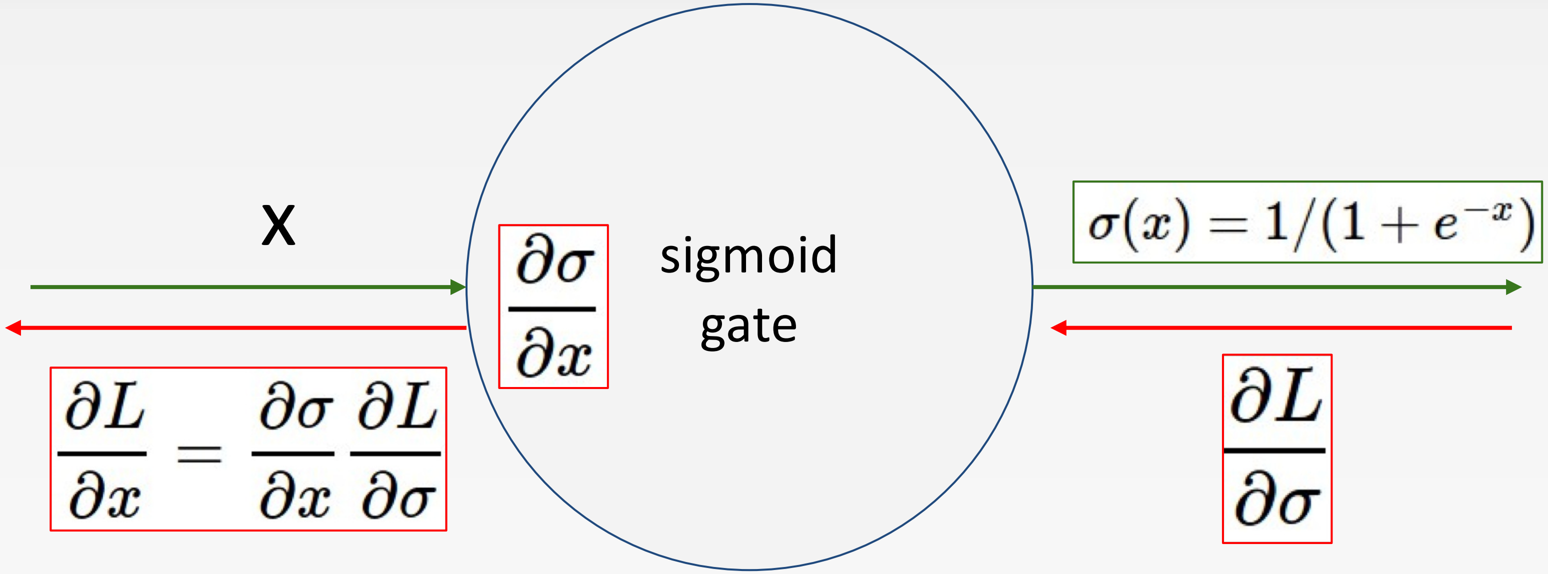


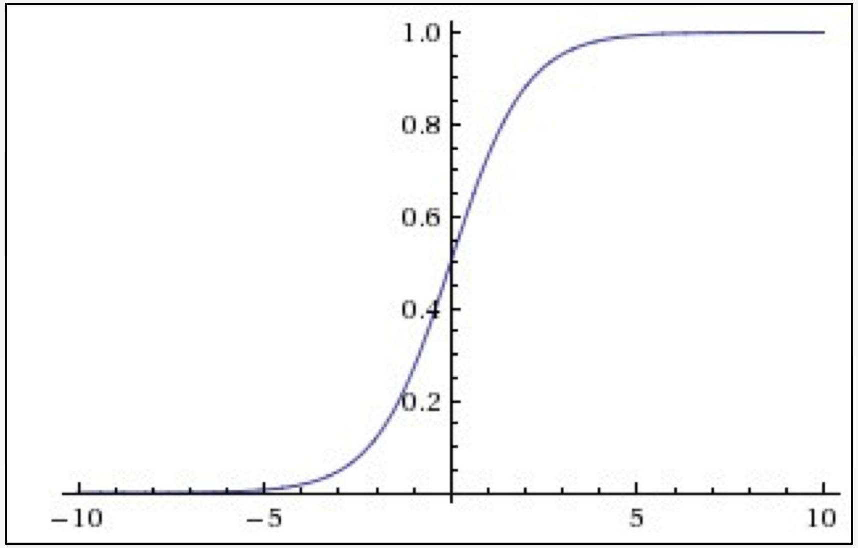
Sigmoid

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

3 problems:

1. Saturated neurons “kill” the gradients

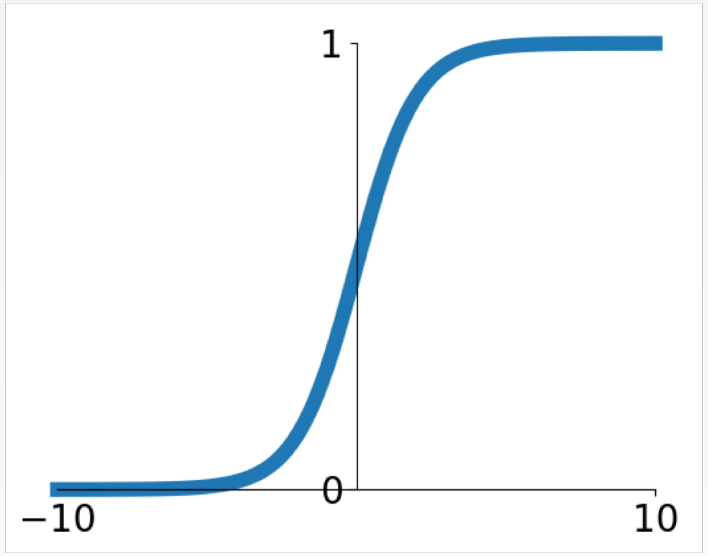




What happens when x = -10?
What happens when x = 0?
What happens when x = 10?

ACTIVATION FUNCTIONS

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



Sigmoid

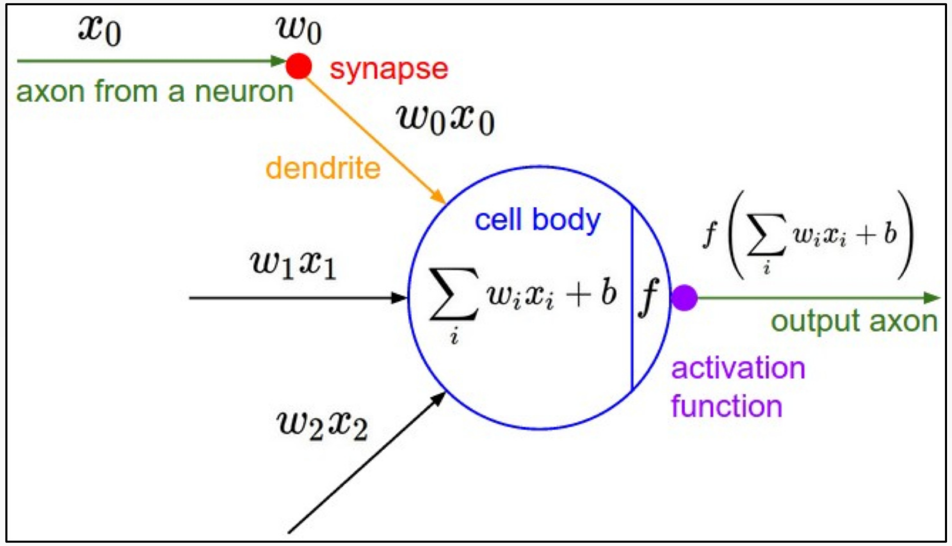
- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron

3 problems:

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

Consider what happens when the input to a neuron is always positive...

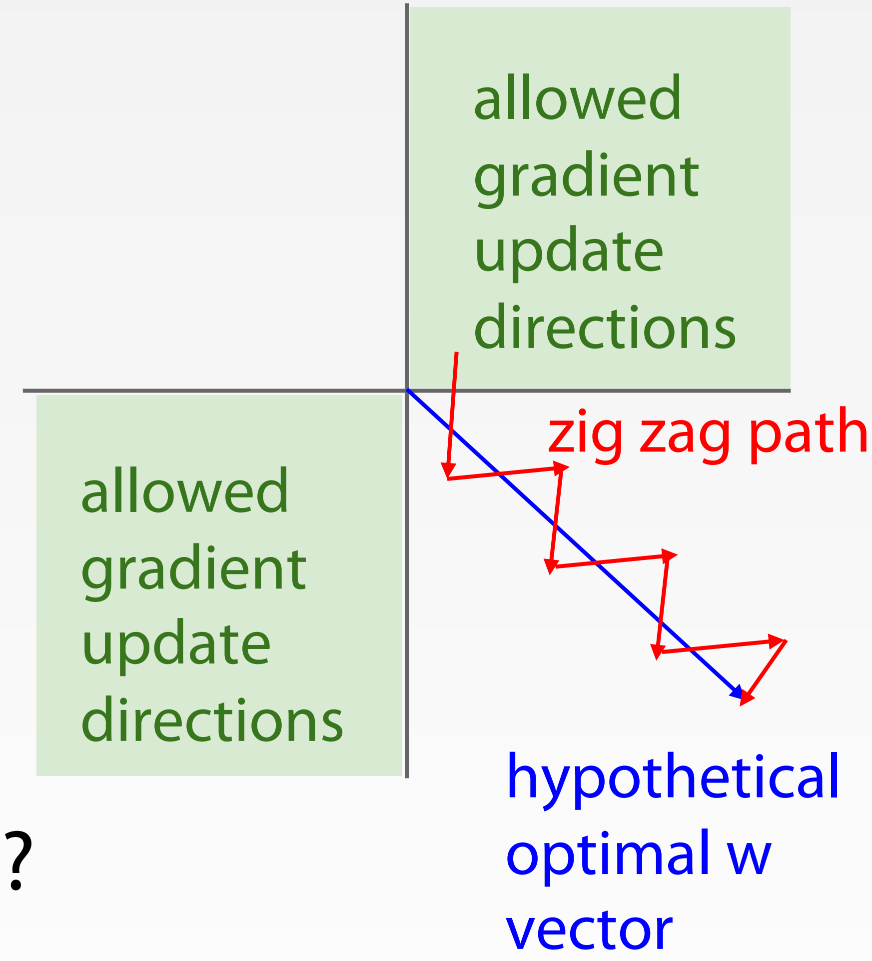
$$f\left(\sum_i w_i x_i + b\right)$$



What can we say about the gradients on **w**?

Consider what happens when the input to a neuron is always positive...

$$f\left(\sum_i w_i x_i + b\right)$$

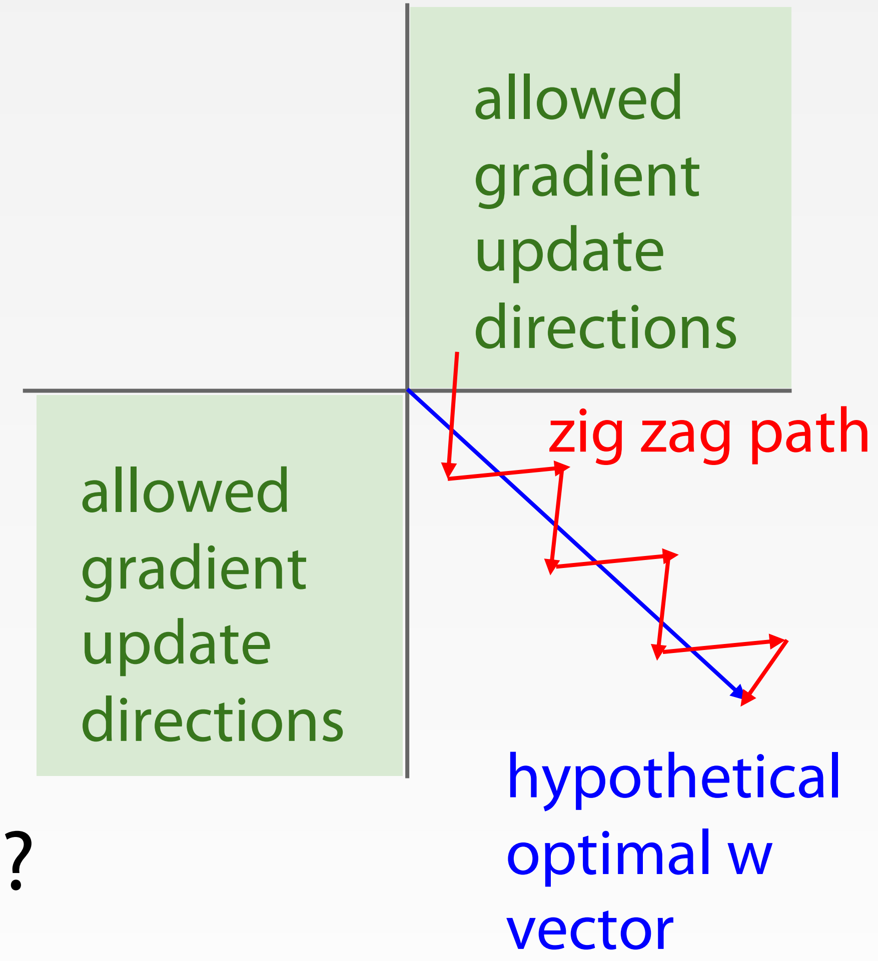


What can we say about the gradients on **w**?

Always all positive or all negative :(

Consider what happens when the input to a neuron is always positive...

$$f\left(\sum_i w_i x_i + b\right)$$



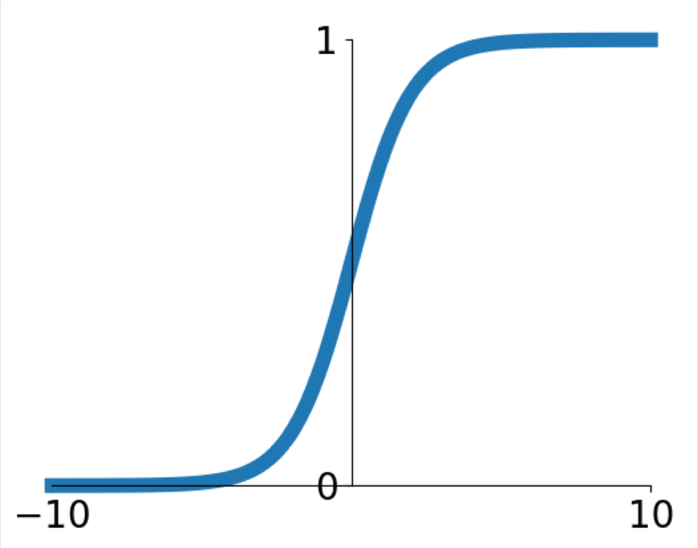
What can we say about the gradients on **w**?

Always all positive or all negative :(

(For a single element! Minibatches help)

ACTIVATION FUNCTIONS

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



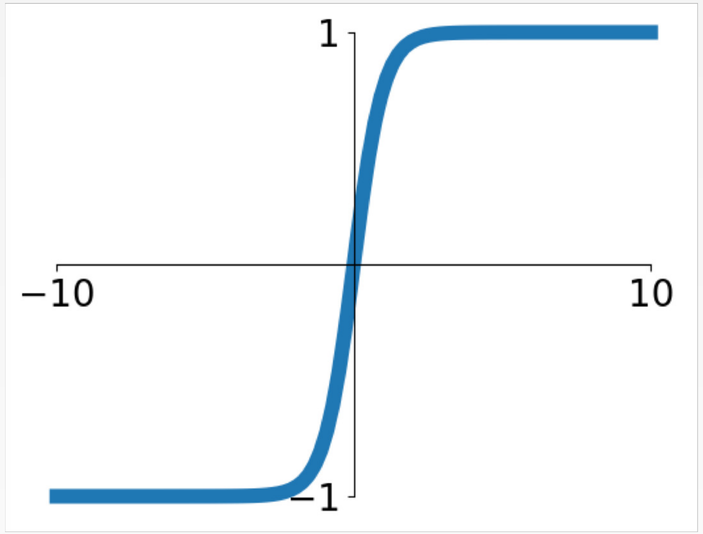
Sigmoid

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

3 problems:

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

ACTIVATION FUNCTIONS

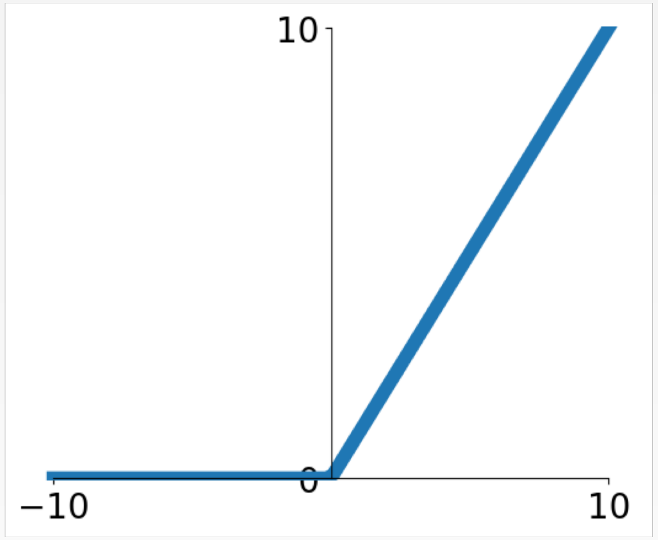


tanh(x)

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

[LeCun et al., 1991]

ACTIVATION FUNCTIONS



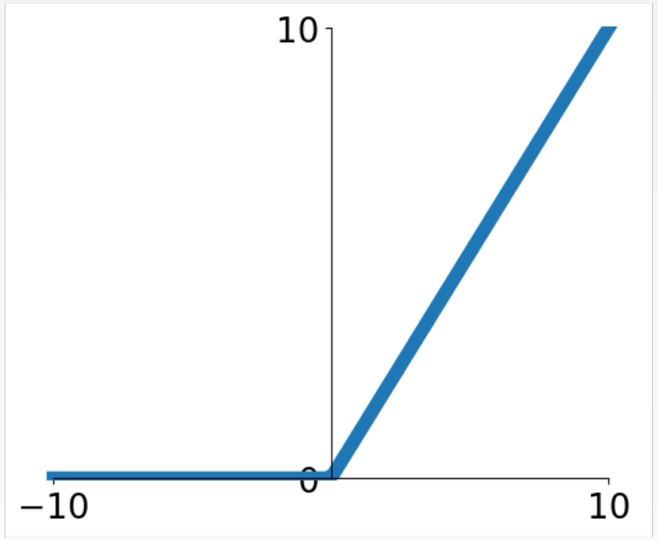
ReLU
(Rectified Linear Unit)

- 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)

[Krizhevsky et al., 2012]

ACTIVATION FUNCTIONS



ReLU
(Rectified Linear Unit)

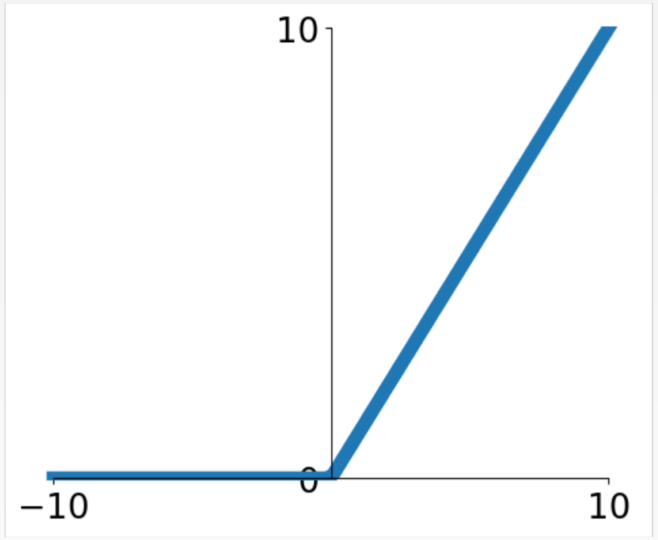
- Computes **f(x) = max(0,x)**

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

- Not zero-centered output

[Krizhevsky et al., 2012]

ACTIVATION FUNCTIONS



ReLU
(Rectified Linear Unit)

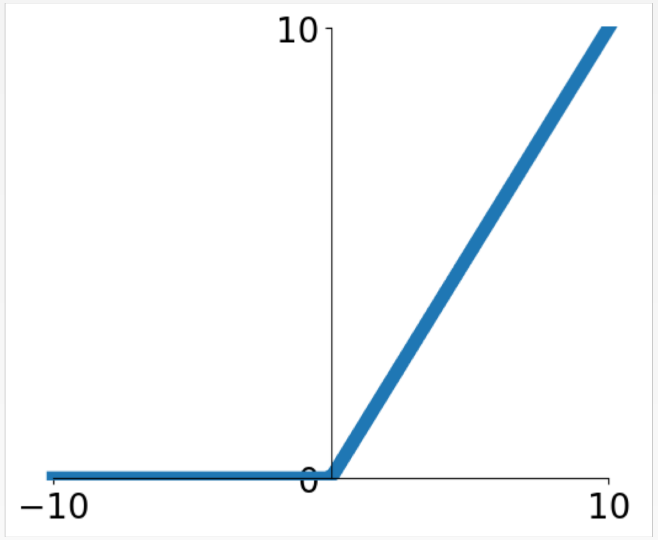
- Computes **f(x) = max(0,x)**

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

- Not zero-centered output

[Krizhevsky et al., 2012]

ACTIVATION FUNCTIONS



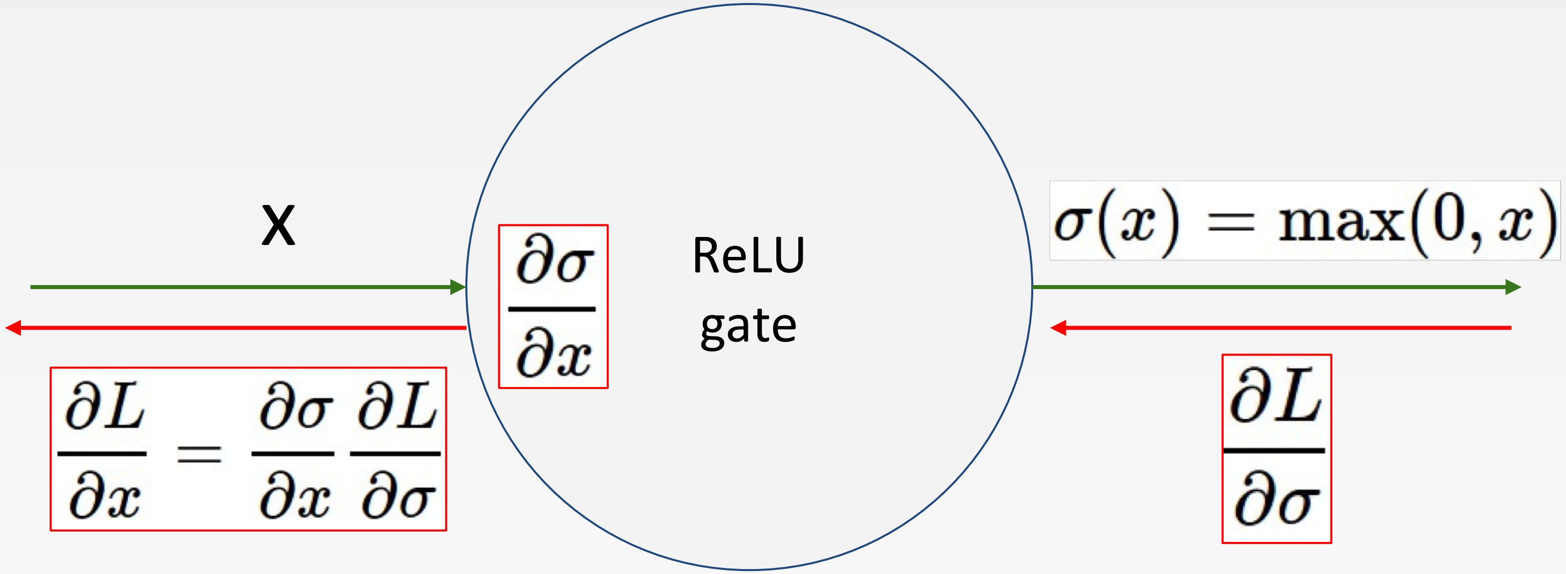
ReLU
(Rectified Linear Unit)

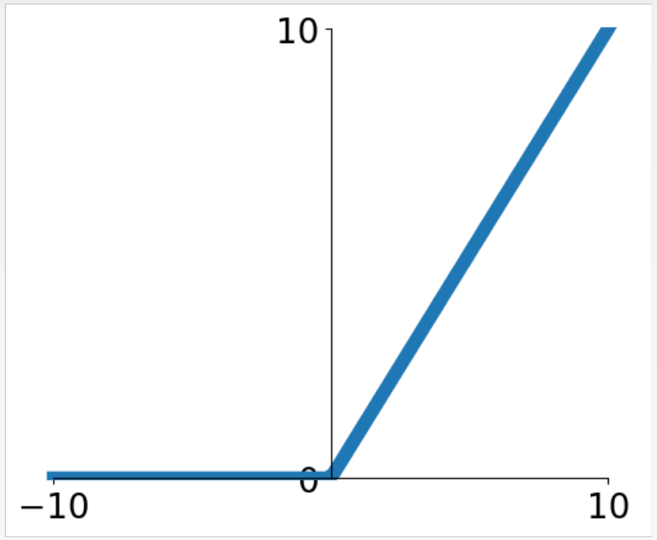
- Computes **f(x) = max(0,x)**

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

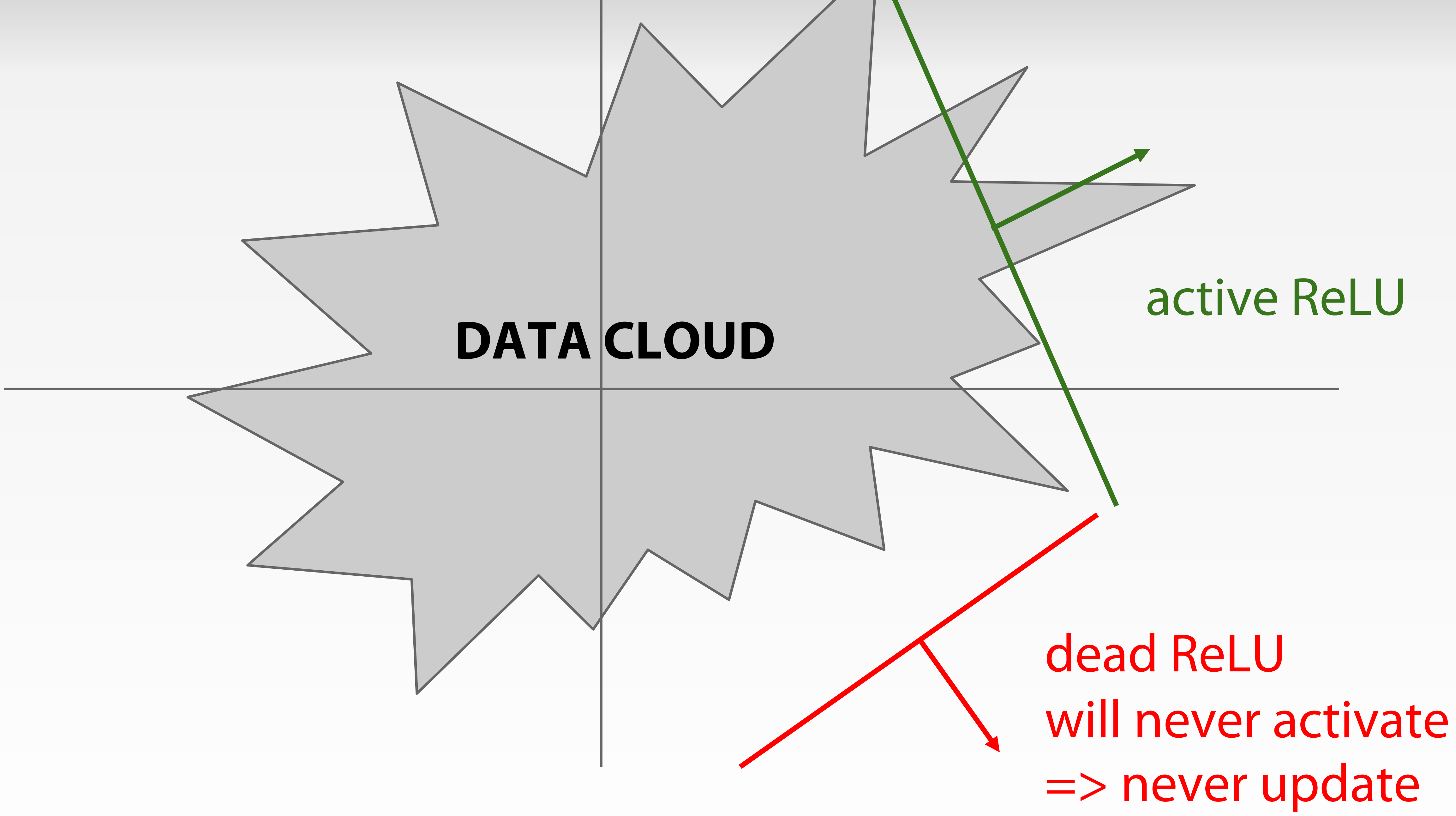
- Not zero-centered output
- An annoyance:

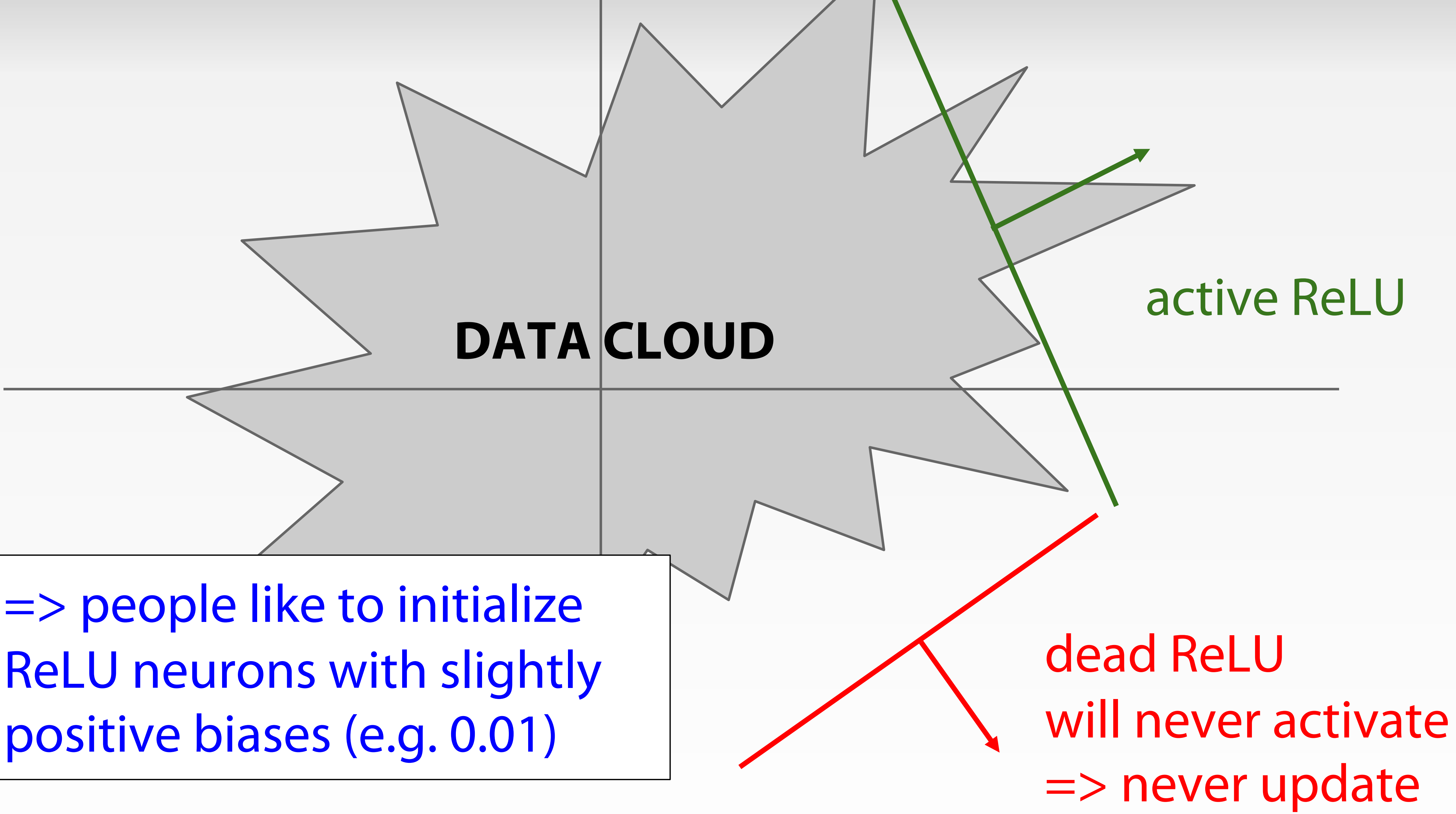
hint: what is the gradient when x < 0?





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

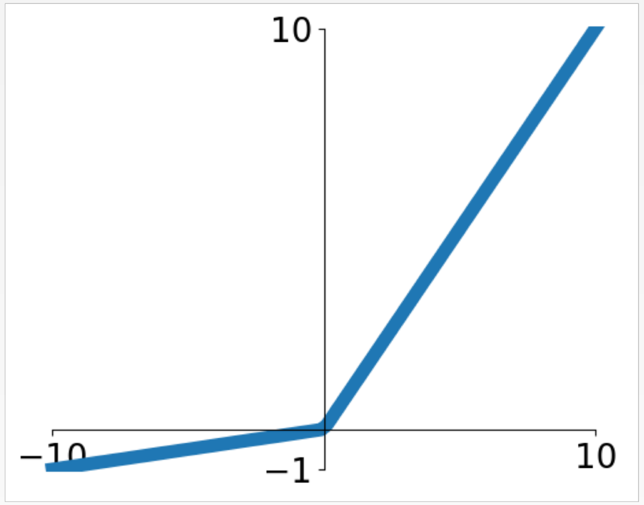

DATA CLOUD
active ReLU
dead ReLU
will never activate
=> never update



=> people like to initialize ReLU neurons with slightly positive biases (e.g. 0.01)

ACTIVATION FUNCTIONS

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



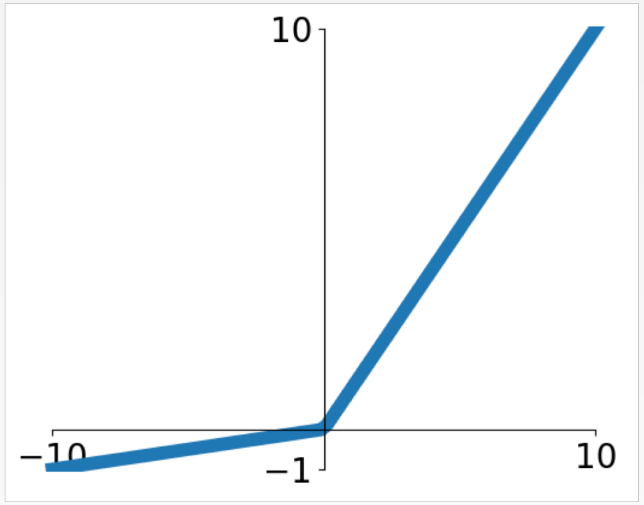
- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- **will not "die".**

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)

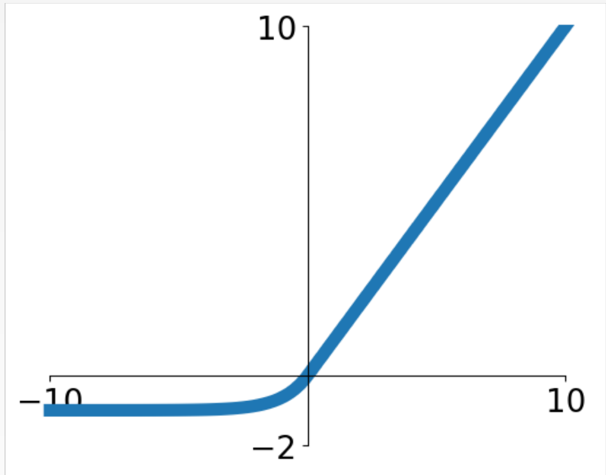
$$f(x) = \max(\alpha x, x)$$

backprop into \alpha (parameter)

ACTIVATION FUNCTIONS

[Clevert et al., 2015]

Exponential Linear Units (ELU)



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

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

- Computation requires exp()

MAXOUT "NEURON"

[Goodfellow et al., 2013]

- Does not have the basic form of dot product -> nonlinearity
- Generalizes ReLU and Leaky ReLU
- Linear Regime! Does not saturate! Does not die!

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

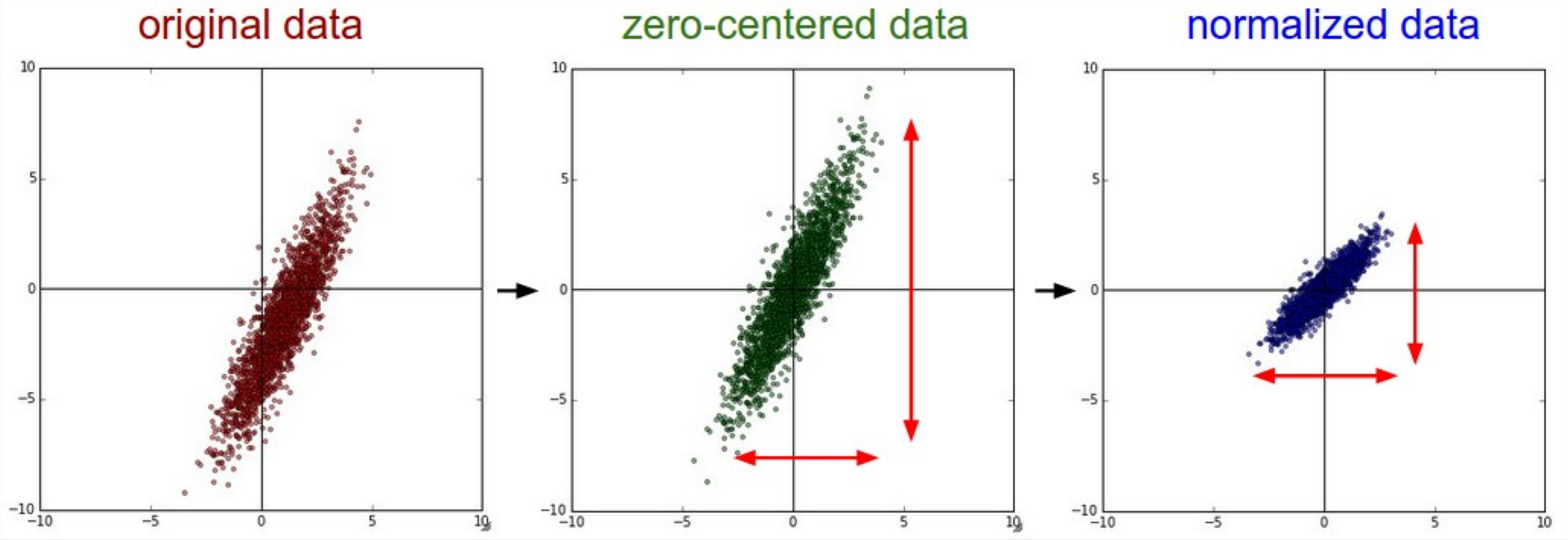
Problem: doubles the number of parameters/neuron :(

TLDR: IN PRACTICE:

- Use ReLU. Be careful with your learning rates
- Try out Leaky ReLU / Maxout / ELU
- Try out tanh but don't expect much
- Don't use sigmoid

DATA PREPROCESSING

DATA PREPROCESSING



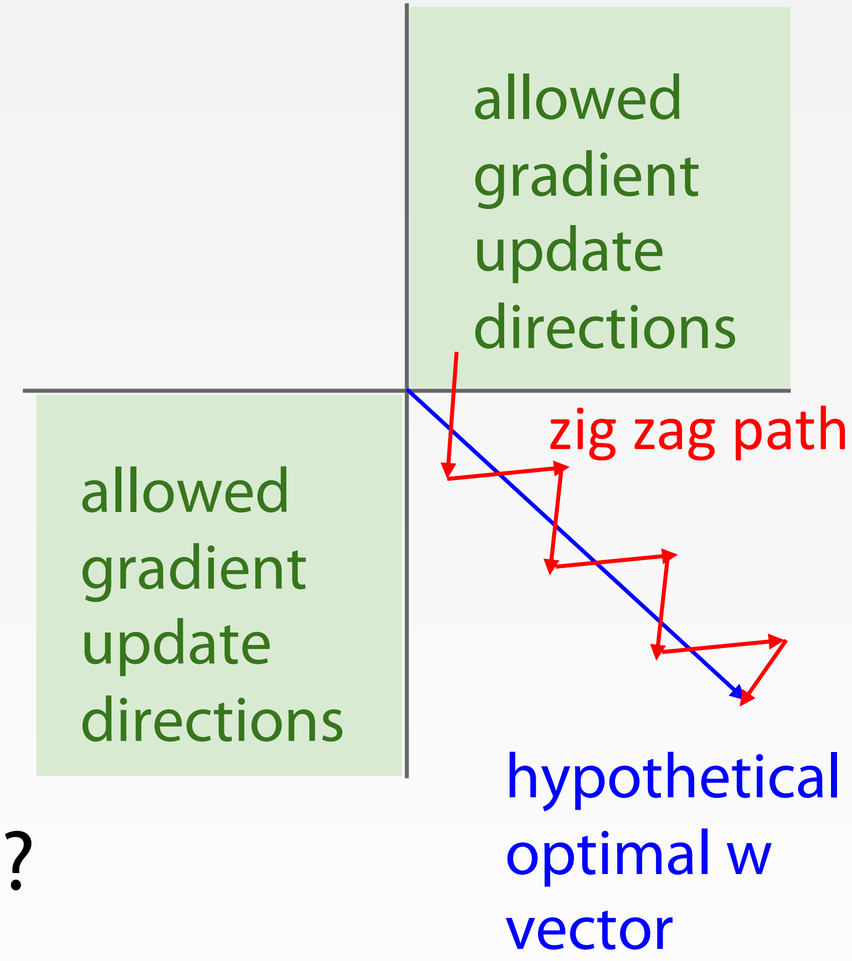

```
X -= np.mean(X, axis = 0)
```

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

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

Remember: Consider what happens when the input to a neuron is always positive...

$$f\left(\sum_i w_i x_i + b\right)$$

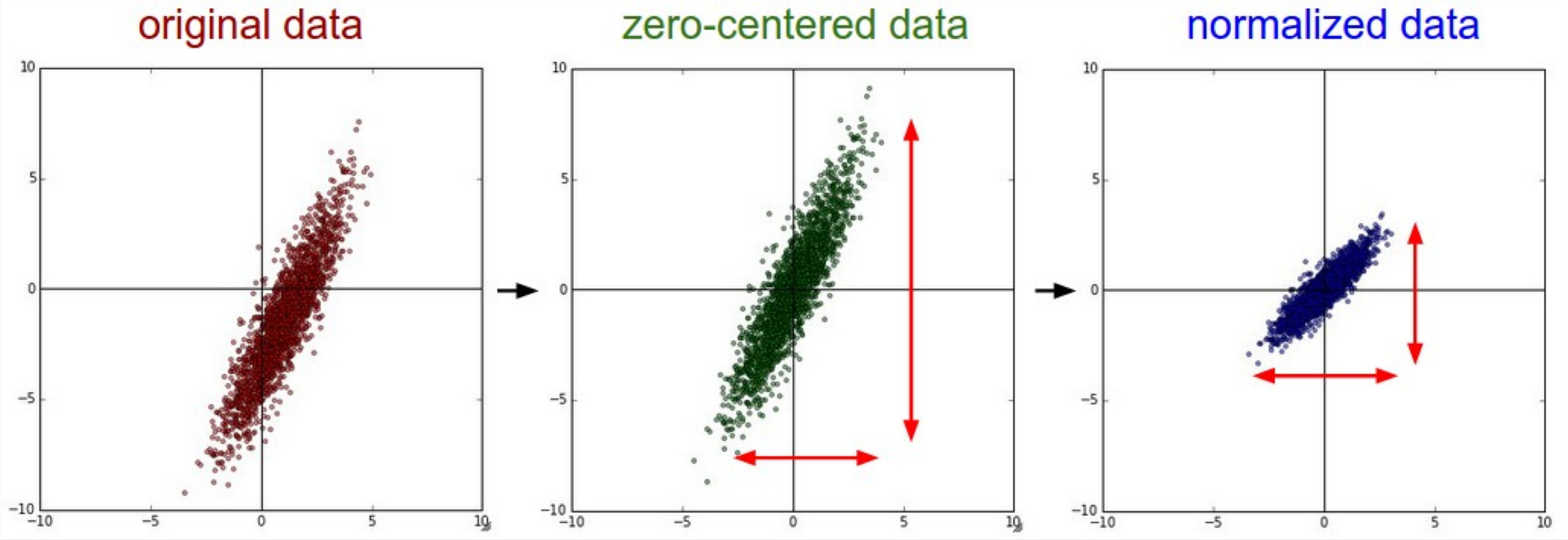


What can we say about the gradients on **w**?

Always all positive or all negative :(

(this is also why you want zero-mean data!)

DATA PREPROCESSING

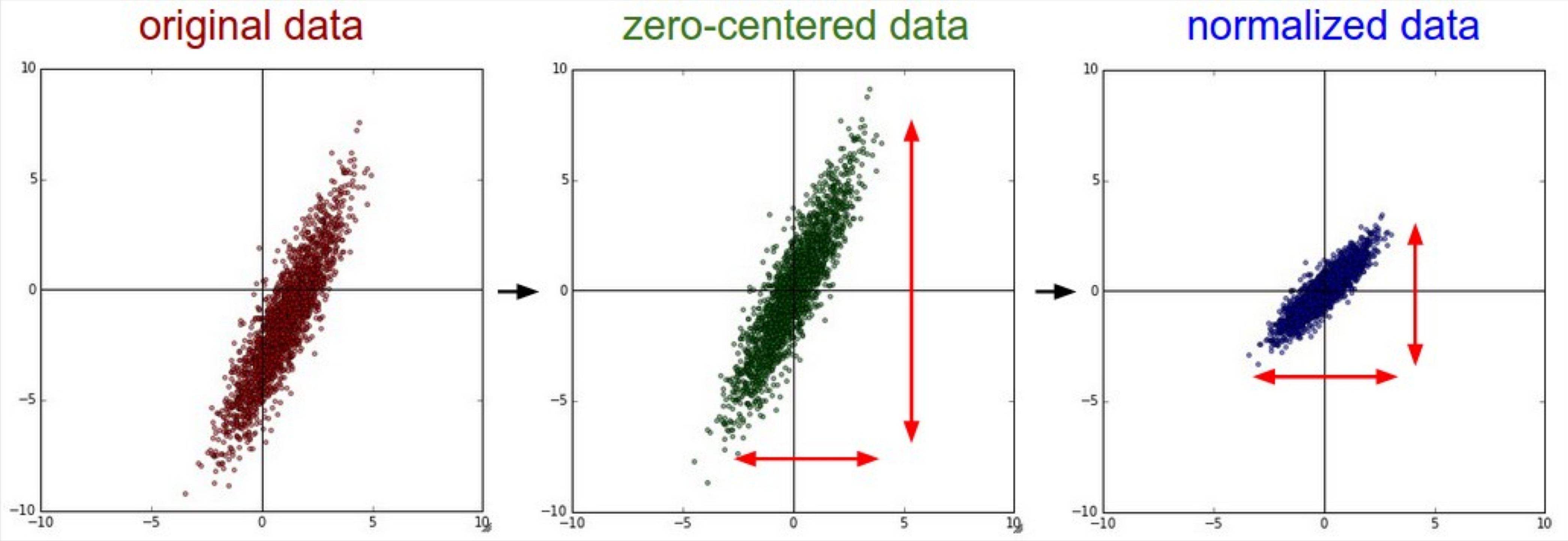



```
X -= np.mean(X, axis = 0)
```

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

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

DATA PREPROCESSING



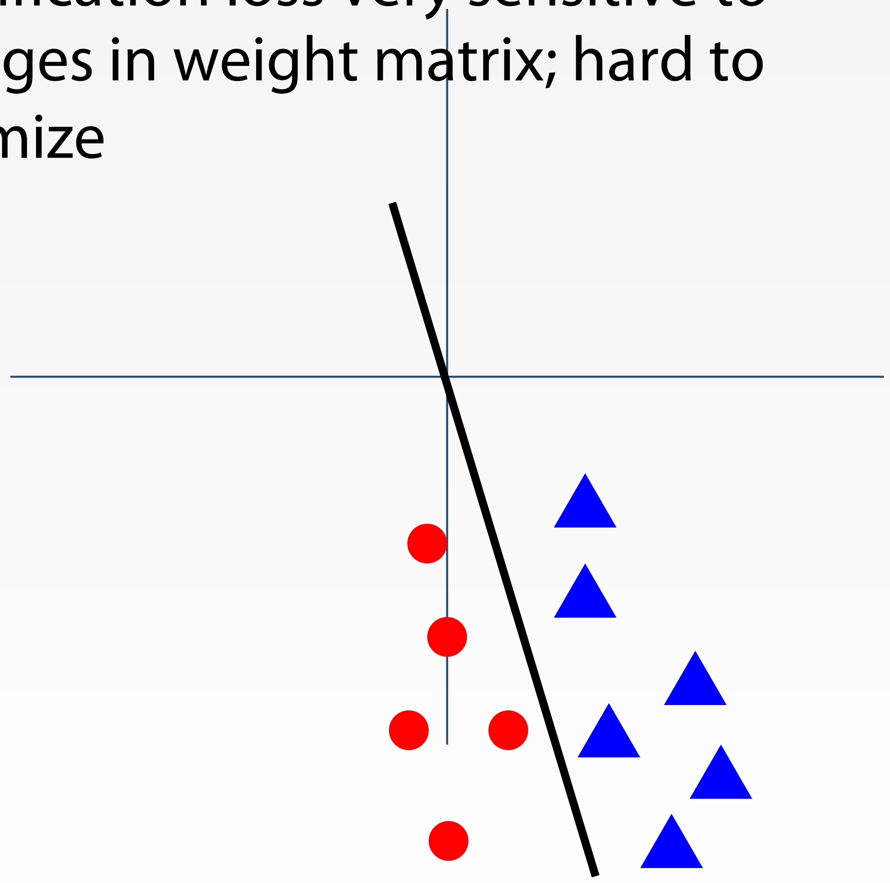
(data has diagonal covariance matrix)

(covariance matrix is the identity matrix)

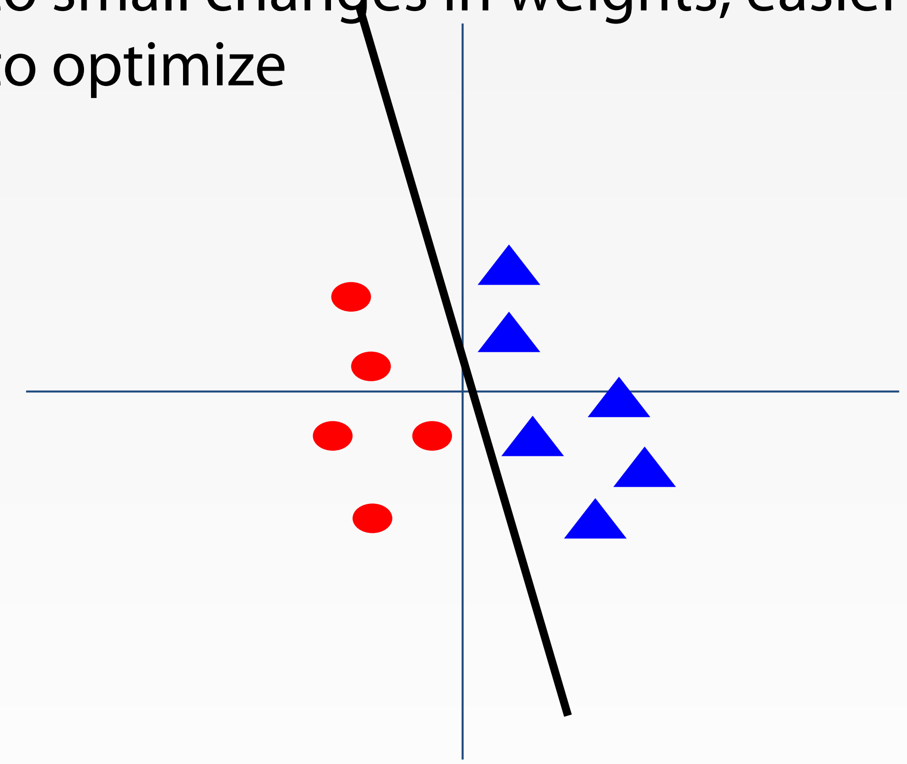
In practice, you may also see **PCA** and **Whitening** of the data

DATA PREPROCESSING

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



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



TLDR: IN PRACTICE FOR IMAGES: CENTER ONLY

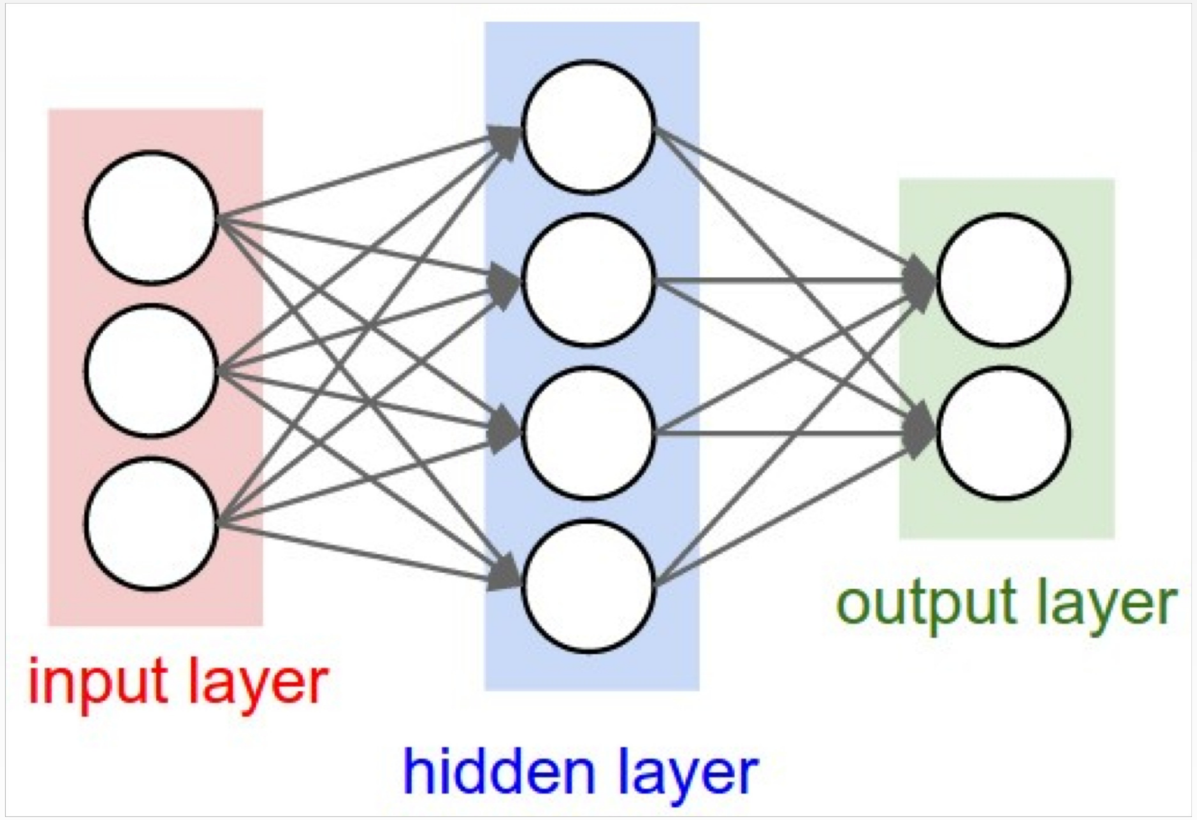
e.g. consider CIFAR-10 example with [32,32,3] images

- Subtract the mean image (e.g. AlexNet)
  (mean image = [32,32,3] array)
- Subtract per-channel mean (e.g. VGGNet)
  (mean along each channel = 3 numbers)
- Subtract per-channel mean and Divide by per-channel std (e.g. ResNet)
  (mean along each channel = 3 numbers)

Not common to do PCA or whitening

WEIGHT INITIALIZATION

Q: what happens when W=constant init is used?



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

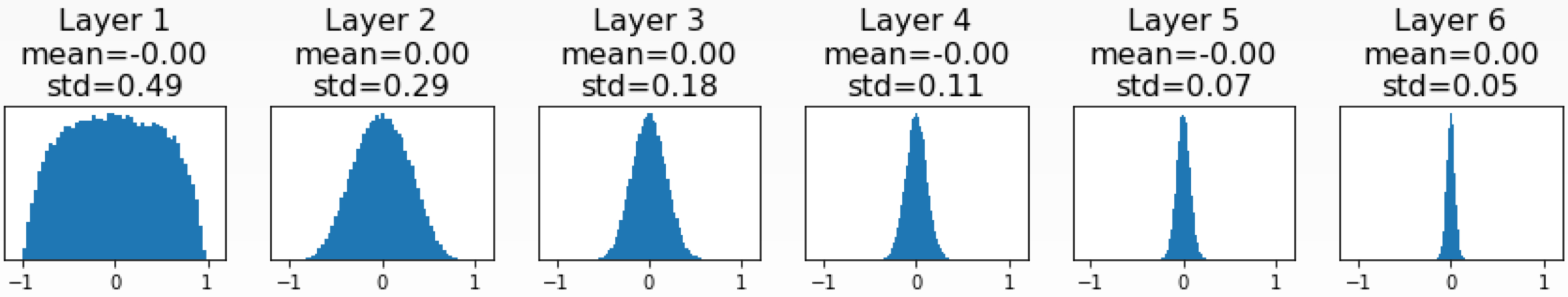
WEIGHT INITIALIZATION: ACTIVATION STATISTICS

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

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

All activations tend to zero for deeper network layers

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



WEIGHT INITIALIZATION: ACTIVATION STATISTICS

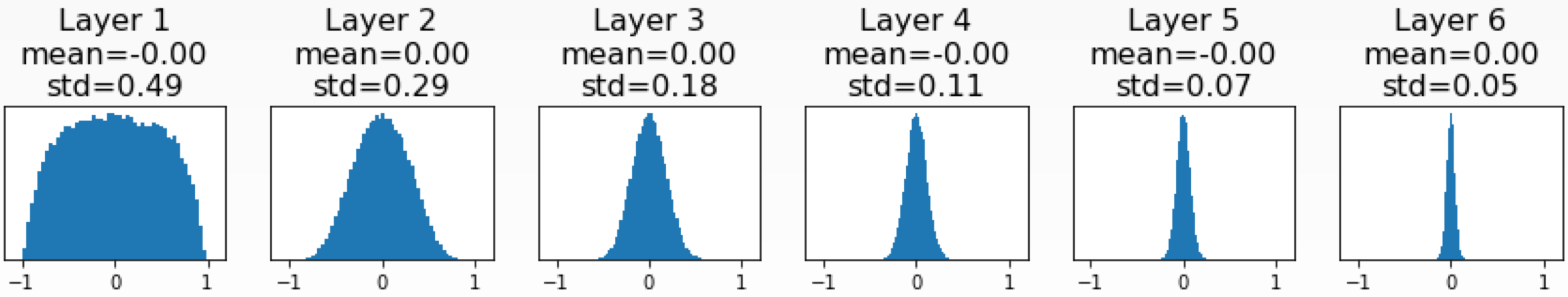
```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

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

All activations tend to zero for deeper network layers

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

A: All zero, no learning =(



WEIGHT INITIALIZATION: ACTIVATION STATISTICS

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

Increase std of initial weights from 0.01 to 0.05

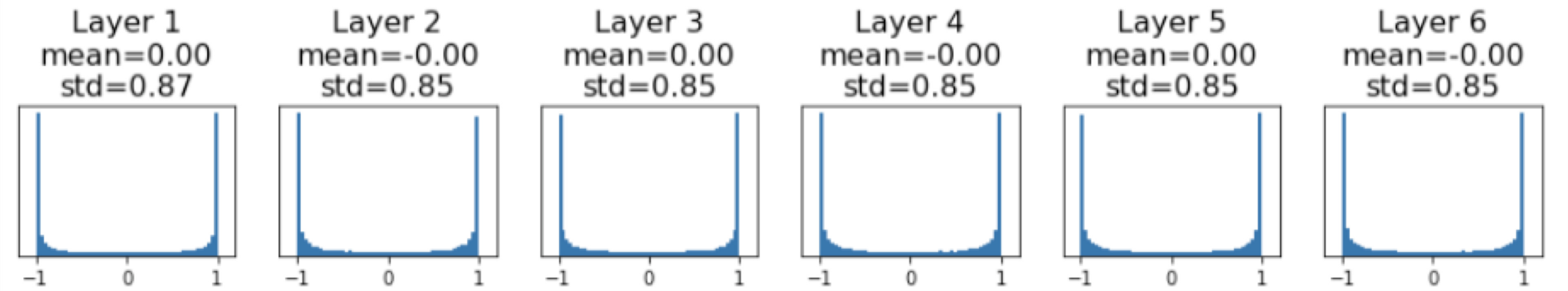
WEIGHT INITIALIZATION: ACTIVATION STATISTICS

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

Increase std of initial weights from 0.01 to 0.05

All activations saturate

Q: What do the gradients look like?



WEIGHT INITIALIZATION: ACTIVATION STATISTICS

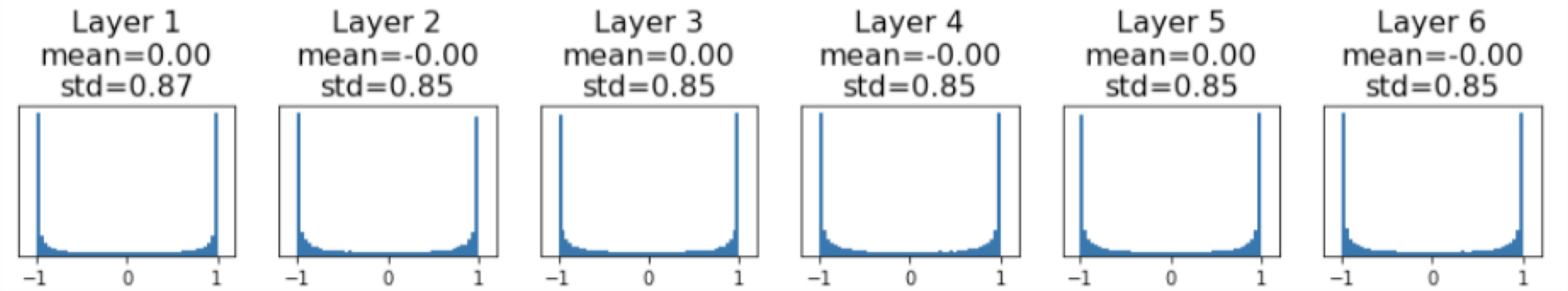
```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.05 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Increase std of initial weights from 0.01 to 0.05

All activations saturate

Q: What do the gradients look like?

A: Local gradients all zero, no learning =(



WEIGHT INITIALIZATION: "XAVIER" INITIALIZATION

```
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)
```

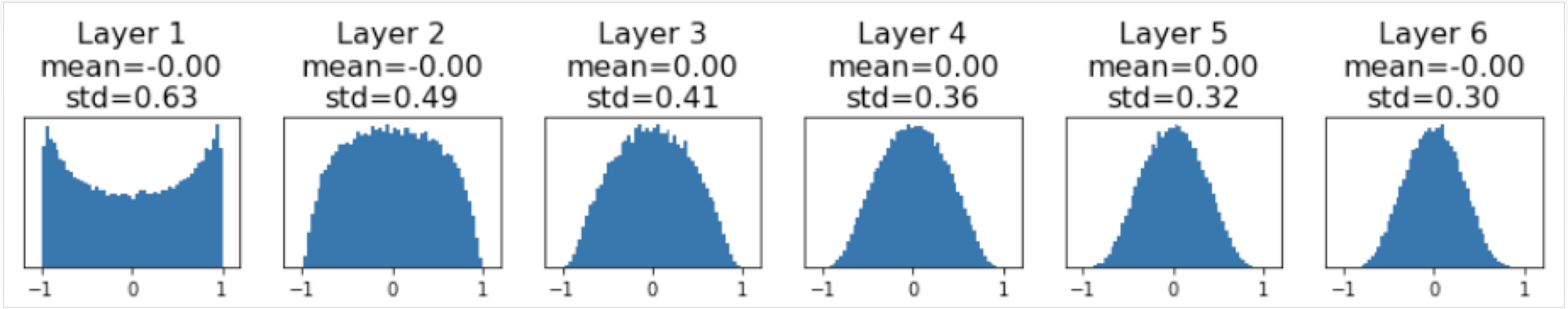
"Xavier" initialization:
std = 1/sqrt(Din)

WEIGHT INITIALIZATION: "XAVIER" INITIALIZATION

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

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

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



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

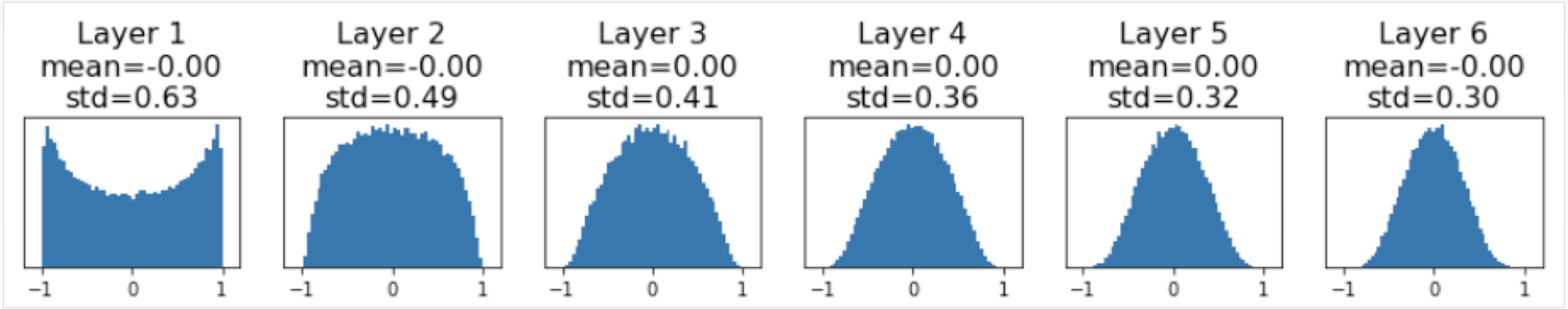
WEIGHT INITIALIZATION: "XAVIER" INITIALIZATION

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

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

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

For conv layers, Din is kernel_size^2 * input_channels



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 $\text{kernel\_size}^2 * \text{input\_channels}$

$y = Wx$
$h = f(y)$

Derivation:

$\mathrm{Var}(y_i) = \mathrm{Din} * \mathrm{Var}(x_i w_i)$ [Assume x, w are iid]
$= \mathrm{Din} * (E[x_i^2]E[w_i^2] - E[x_i]^2 E[w_i]^2)$ [Assume x, w independant]
$= \mathrm{Din} * \mathrm{Var}(x_i) * \mathrm{Var}(w_i)$ [Assume x, w are zero-mean]

If $\mathrm{Var}(w_i) = 1/\mathrm{Din}$ then $\mathrm{Var}(y_i) = \mathrm{Var}(x_i)$

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

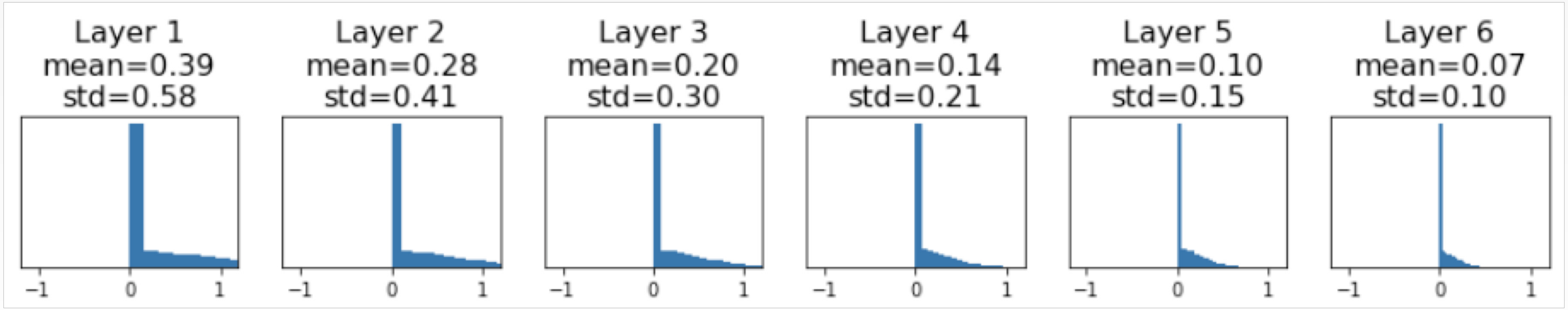
WEIGHT INITIALIZATION: WHAT ABOUT RELU?

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

Change from tanh to ReLU

Xavier assumes zero centered activation function

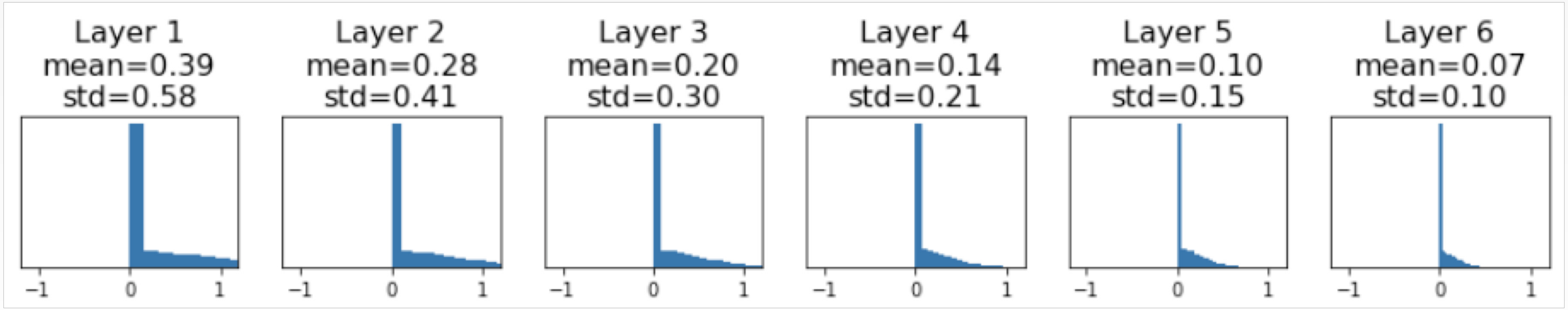
Activations collapse to zero again, no learning =(



WEIGHT INITIALIZATION: KAIMING/MSRA INITIALIZATION

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) * np.sqrt(2/Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

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



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

PROPER INITIALIZATION IS AN ACTIVE AREA OF RESEARCH…

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

BATCH NORMALIZATION

BATCH NORMALIZATION

[Ioffe and Szegedy, 2015]

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

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

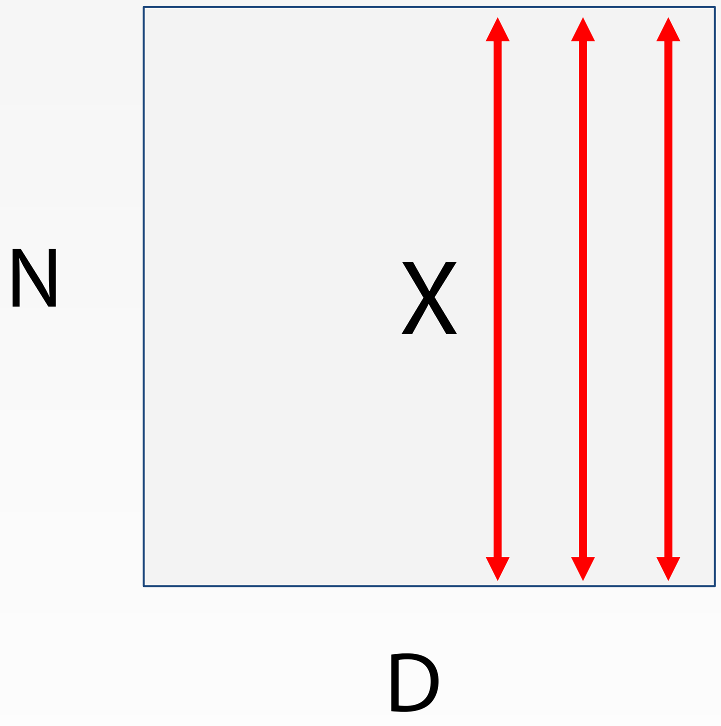
$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

this is a vanilla differentiable function...

BATCH NORMALIZATION

[Ioffe and Szegedy, 2015]

Input: $x : N \times D$



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

Per-channel mean, shape is D

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

Per-channel var, shape is D

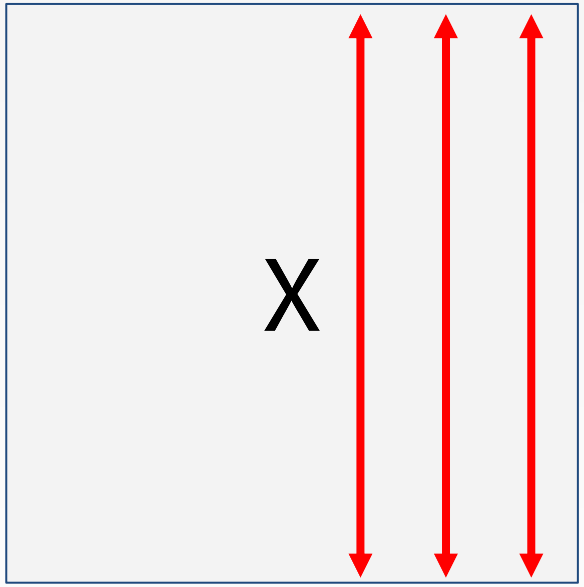
$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Normalized x, Shape is N x D

BATCH NORMALIZATION

[Ioffe and Szegedy, 2015]

Input: $x : N \times D$



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

Per-channel mean, shape is D

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

Per-channel var, shape is D

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

Normalized x, Shape is N x D

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

BATCH NORMALIZATION

[Ioffe and Szegedy, 2015]

Input: $x : N \times D$

Learnable scale and shift parameters:

$\gamma, \beta : D$

Learning $\gamma = \sigma$, $\beta = \mu$ will recover the identity function!

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

Per-channel mean, shape is D

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

Per-channel var, shape is D

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

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

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

Input: $x : N \times D$

Learnable scale and shift parameters:

$\gamma, \beta : D$

Learning $\gamma = \sigma$, $\beta = \mu$ will recover the identity function!

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

Per-channel mean, shape is D

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

Per-channel var, shape is D

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

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:

$\gamma, \beta : D$

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

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

Per-channel mean, shape is D

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

Per-channel var, shape is D

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

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 FOR CONVNETS

Batch Normalization for **fully-connected** networks

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

Normalize

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

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

$$y = \gamma(x-\mu)/\sigma+\beta$$

Batch Normalization for **convolutional** networks
(Spatial Batchnorm, BatchNorm2D)

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

Normalize

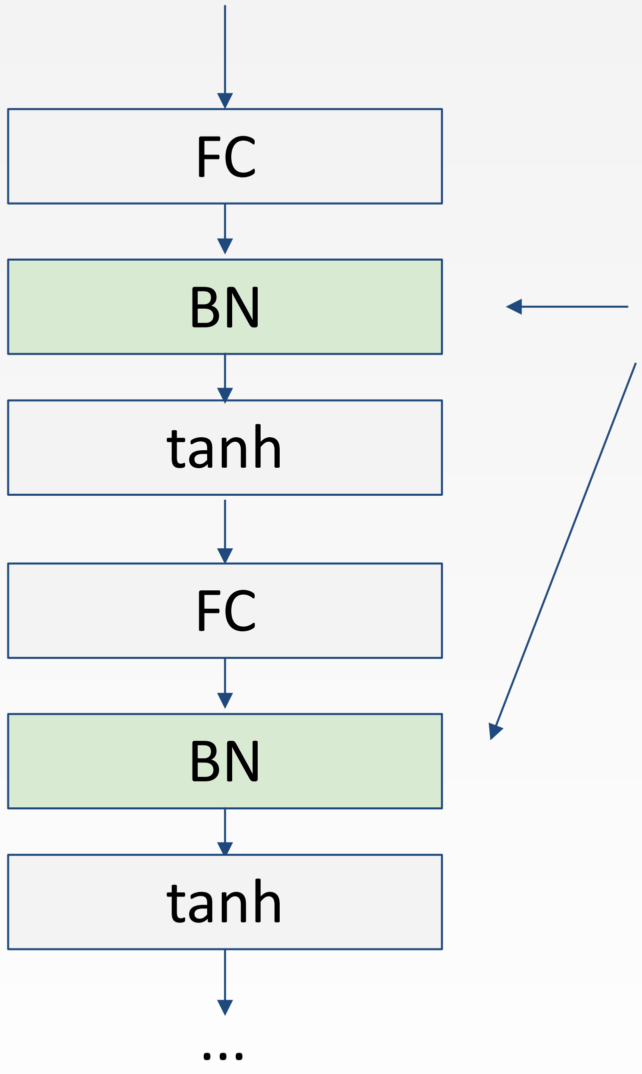
$$\boldsymbol{\mu},\boldsymbol{\sigma}:\ 1{\times}C{\times}1{\times}1$$

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

$$y = \gamma(x-\mu)/\sigma+\beta$$

BATCH NORMALIZATION

[Ioffe and Szegedy, 2015]

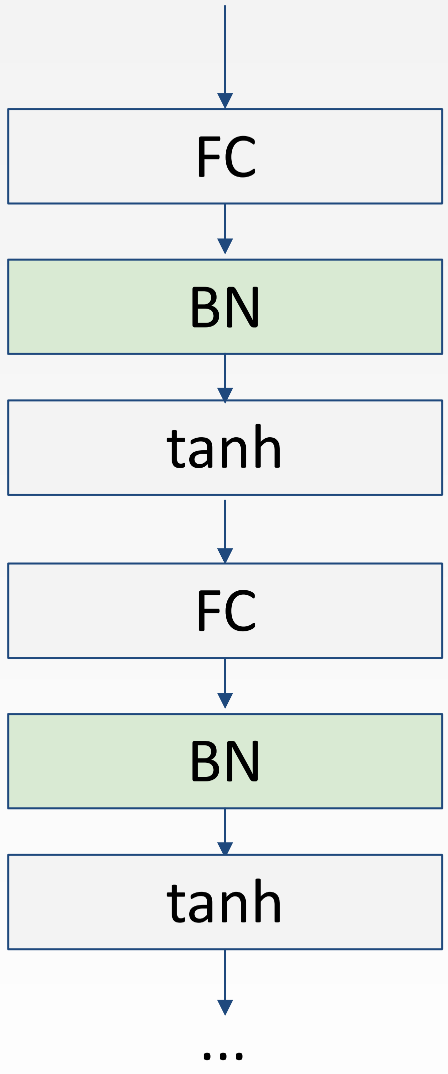


Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

BATCH NORMALIZATION

[Ioffe and Szegedy, 2015]



- Makes deep networks **much** easier to train!
- Improves gradient flow
- Allows higher learning rates, faster convergence
- Networks become more robust to initialization
- Acts as regularization during training
- Zero overhead at test-time: can be fused with conv!
- Behaves differently during training and testing: this is a very common source of bugs!

LAYER NORMALIZATION

Batch Normalization for fully-connected networks

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

Normalize

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

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

$$y = \gamma(x-\mu)/\sigma+\beta$$

Layer Normalization for fully-connected networks

Same behavior at train and test!

Can be used in recurrent networks

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

Normalize

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

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

$$y = \gamma(x-\mu)/\sigma+\beta$$

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

INSTANCE NORMALIZATION

Batch Normalization for convolutional networks

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

Normalize

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

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

$$y = \gamma(x-\mu)/\sigma+\beta$$

Instance Normalization for convolutional networks

Same behavior at train / test!

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

Normalize

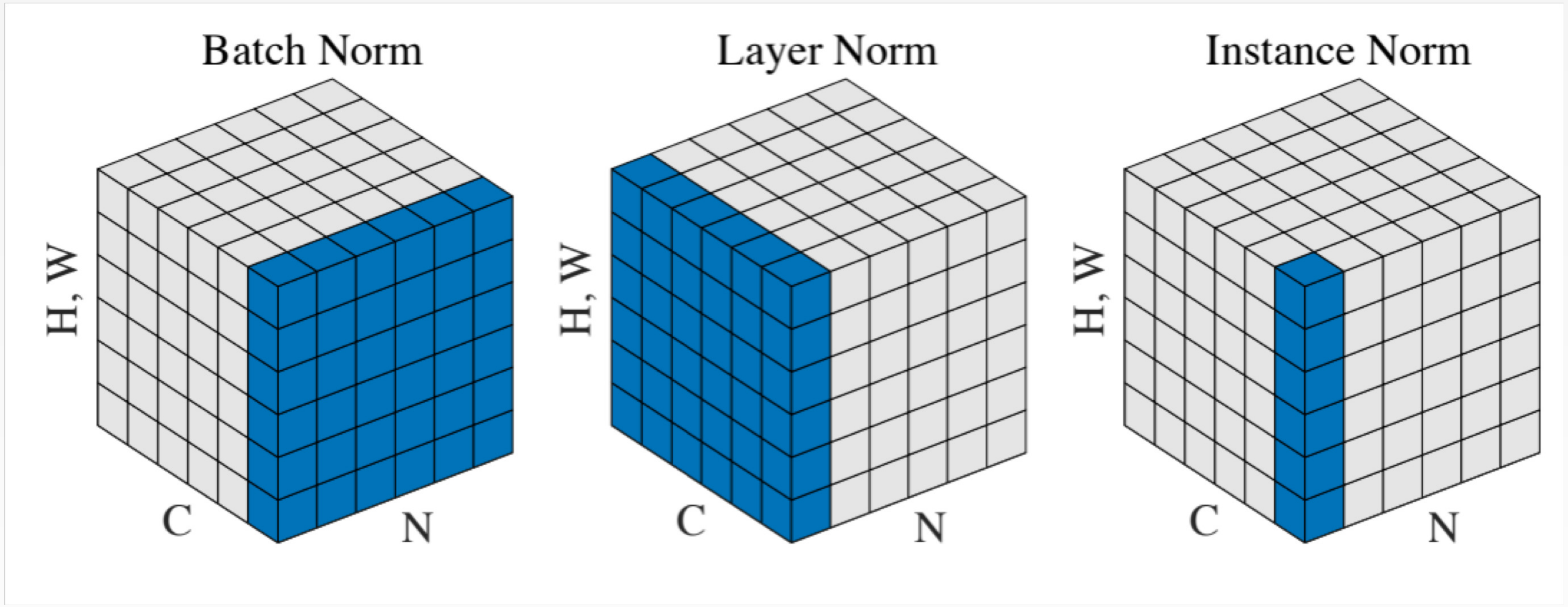
$$\mu,\sigma: N\times C\times 1\times 1$$

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

$$y = \gamma(x-\mu)/\sigma+\beta$$

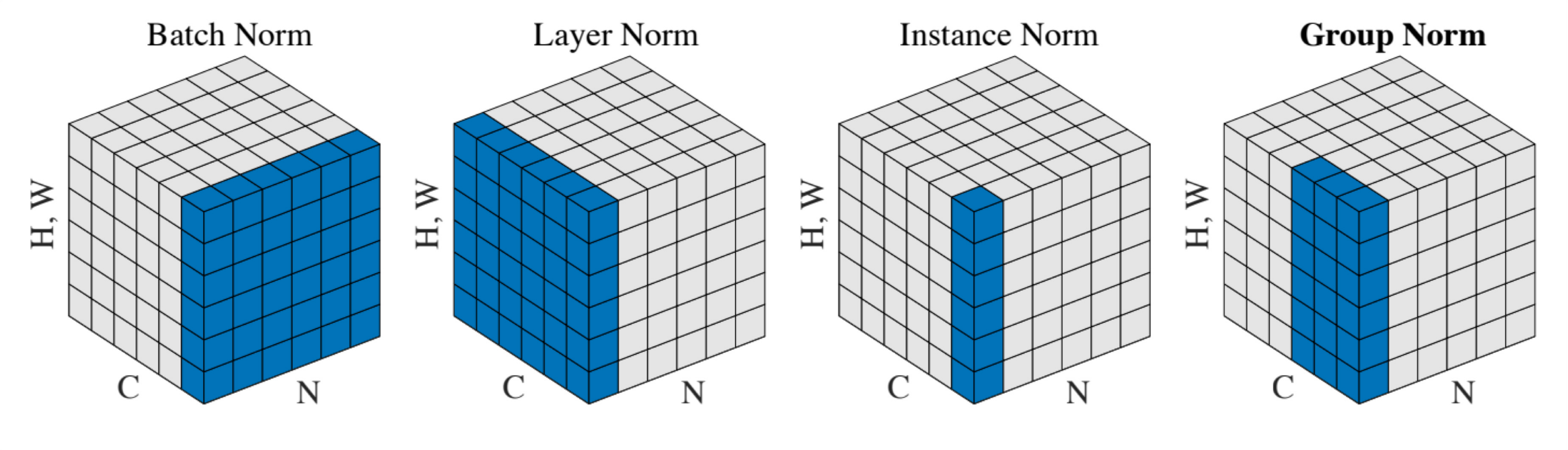
Ulyanov et al, Improved Texture Networks: Maximizing Quality and Diversity in Feed-forward Stylization and Texture Synthesis, CVPR 2017

COMPARISON OF NORMALIZATION LAYERS



Wu and He, “Group Normalization”, ECCV 2018

GROUP NORMALIZATION



Wu and He, "Group Normalization", ECCV 2018

SUMMARY

TLDRs

We looked in detail at:

- Activation Functions (use ReLU)
- Data Preprocessing (images: subtract mean)
- Weight Initialization (use Xavier/He init)
- Batch Normalization (use)

NEXT LECTURE

- Training Neural Networks (Part II)
  - Parameter update schemes
  - Learning rate schedules
  - Gradient checking
  - Regularization (Dropout etc.)
  - Babysitting learning
  - Hyperparameter search
  - Evaluation (Ensembles etc.)
  - Transfer learning / fine-tuning