Lecture 7:
Training Neural Networks, Part I

Administrative: Project Proposal

Due tomorrow, 4/24 on GradeScope

1 person per group needs to submit, but tag all group members

Administrative: Alternate Midterm

See Piazza for form to request alternate midterm time or other midterm accommodations
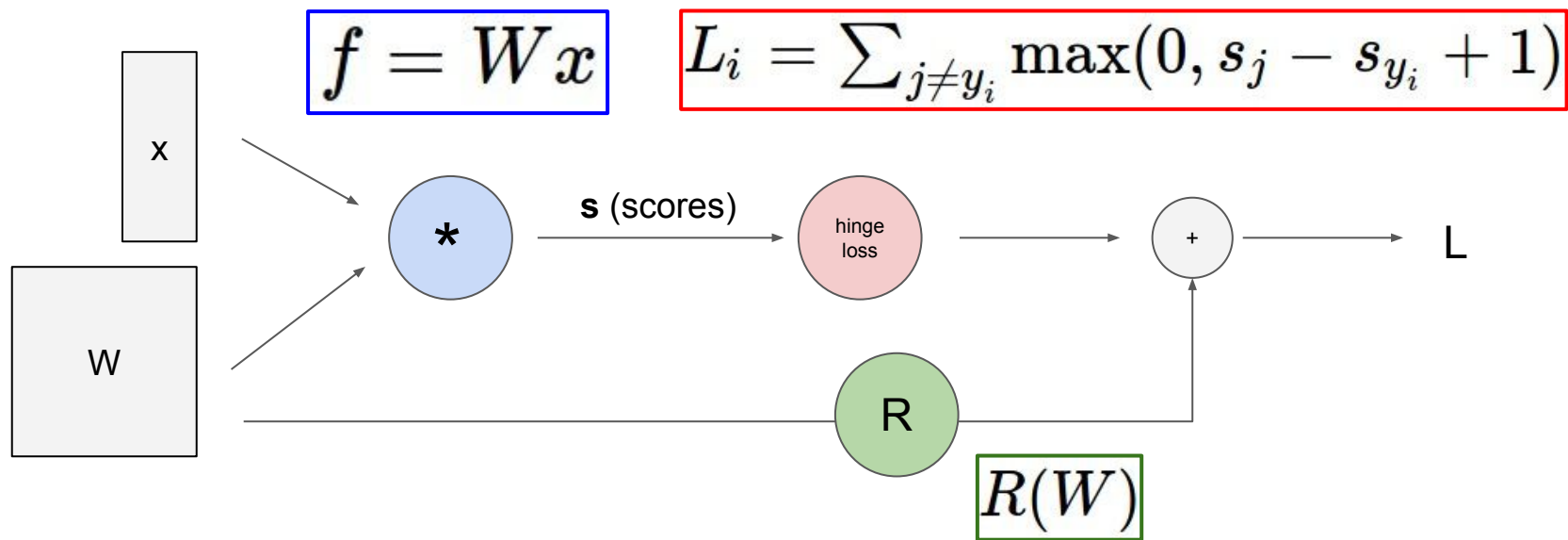
Alternate midterm requests due Thursday!

Administrative: A2

A2 is out, due Wednesday 5/1

We recommend using Google Cloud for the assignment, especially if your local machine uses Windows
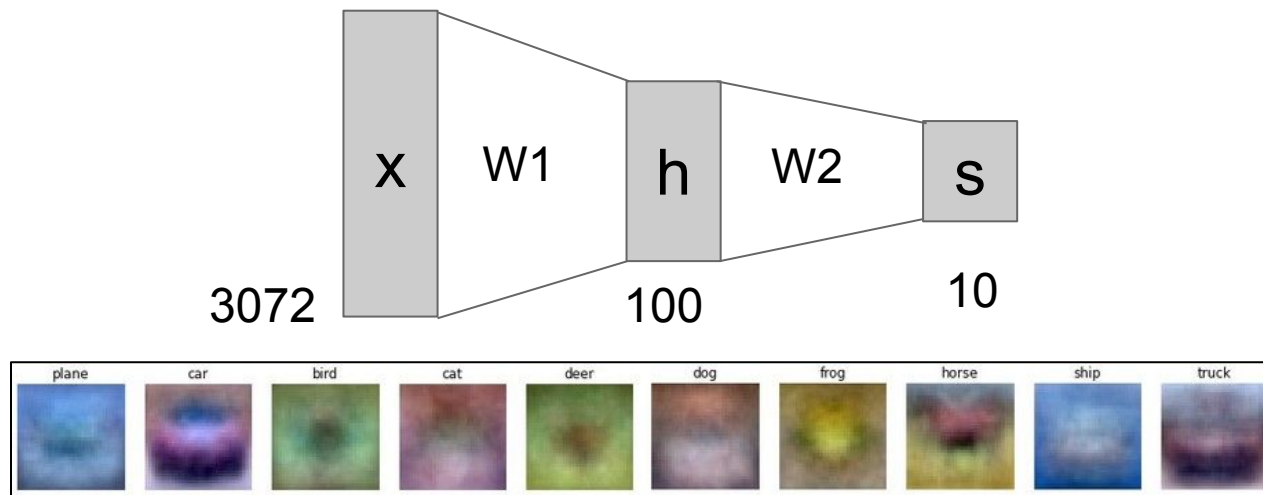
Where we are now...

Computational graphs

$$f = Wx$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

x

W

*

s (scores)

hinge loss

+

L

R

$$R(W)$$

Where we are now...

Neural Networks

Linear score function:

$$f = Wx$$

2-layer Neural Network

$$f = W_2 \max(0, W_1 x)$$

3072   100   10

plane   car   bird   cat   deer   dog   frog   horse   ship   truck

Where we are now...

Convolutional Neural Networks
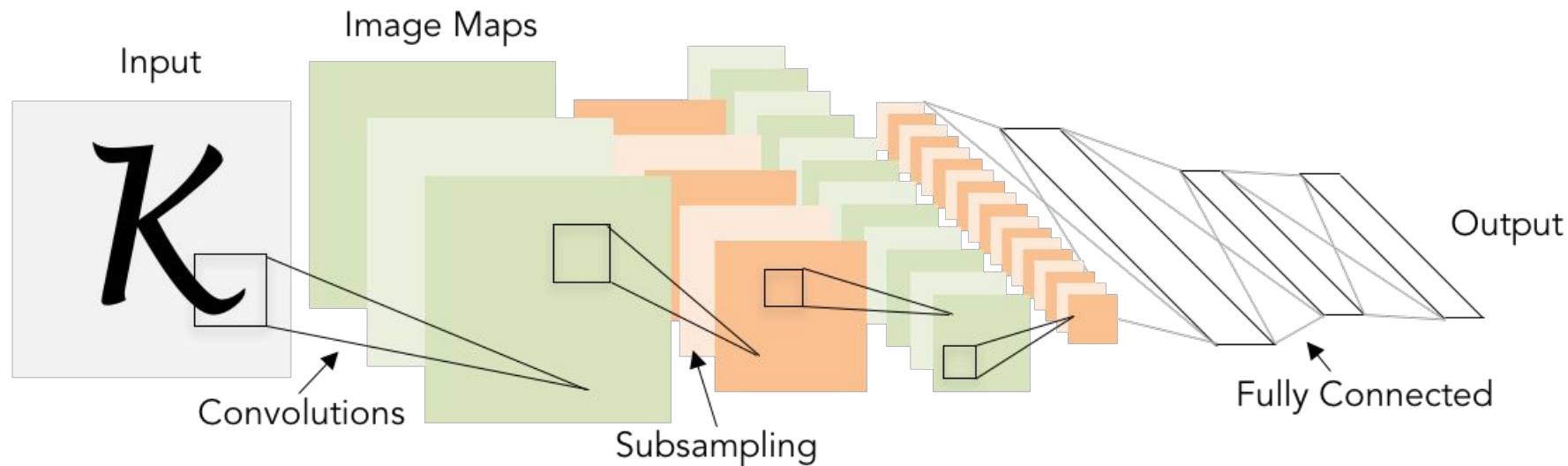
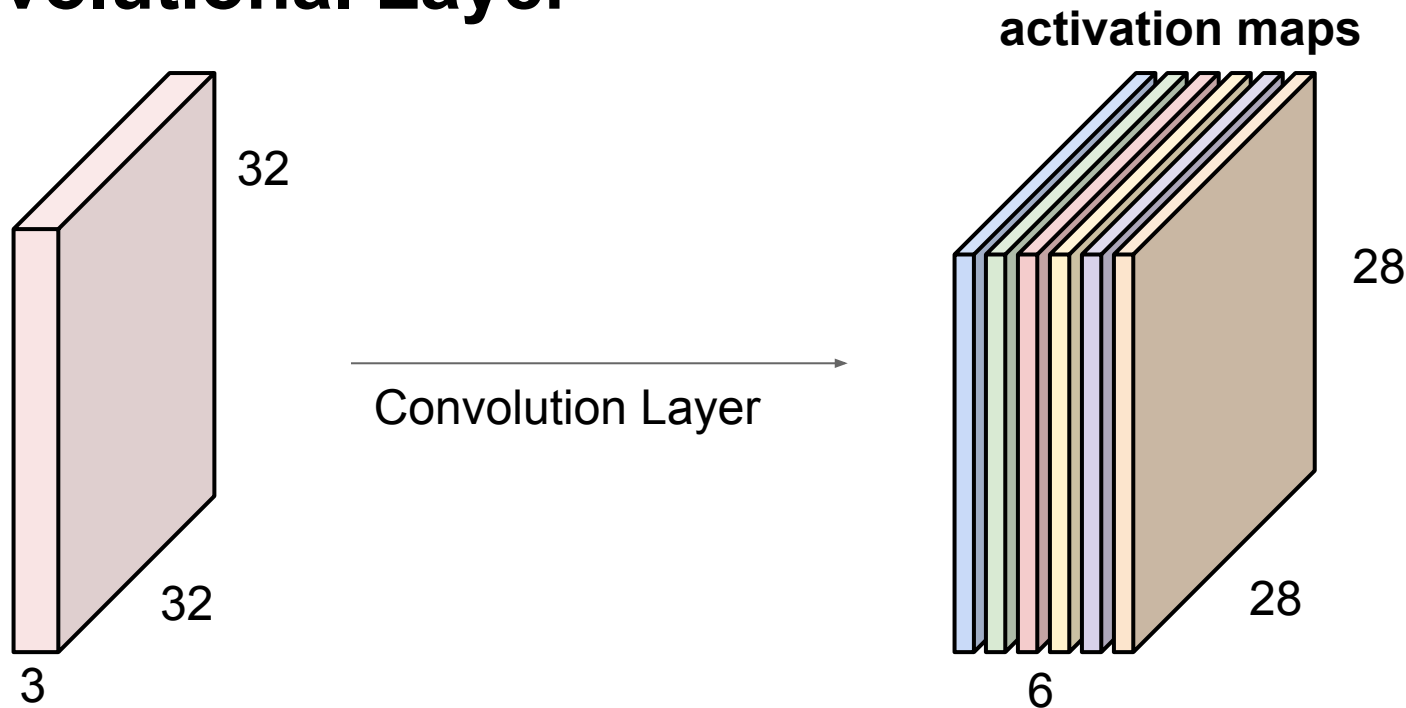Illustration of LeCun et al. 1998 from CS231n 2017 Lecture 1

Where we are now...

Convolutional Layer

32x32x3 image

5x5x3 filter

activation map

convolve (slide) over all spatial locations

32

32

3

28

28

1
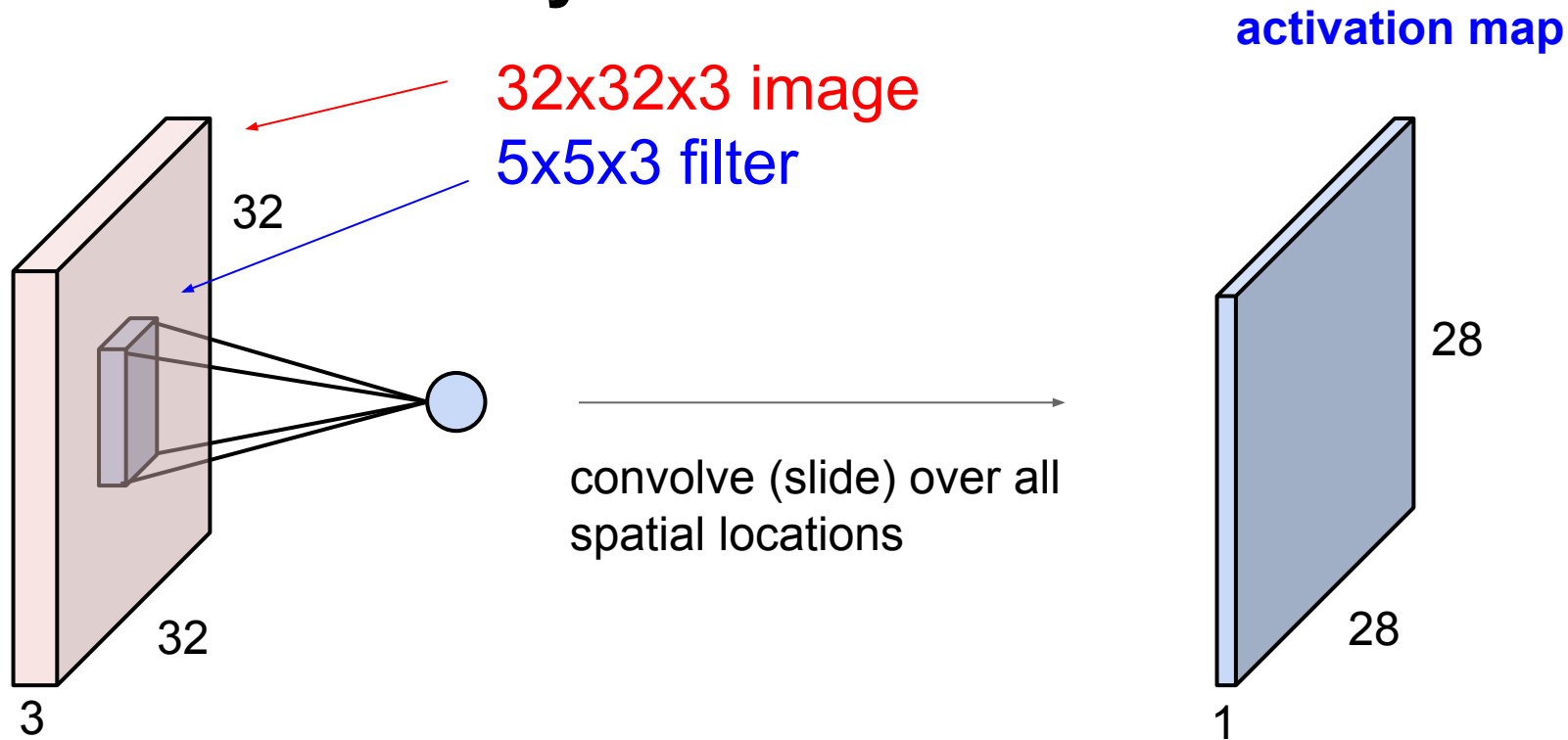
Where we are now...
Convolutional Layer

For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:

activation maps

Convolution Layer

32

32

3

28

28

6

We stack these up to get a "new image" of size 28x28x6!

Where we are now...

Learning network parameters through optimization
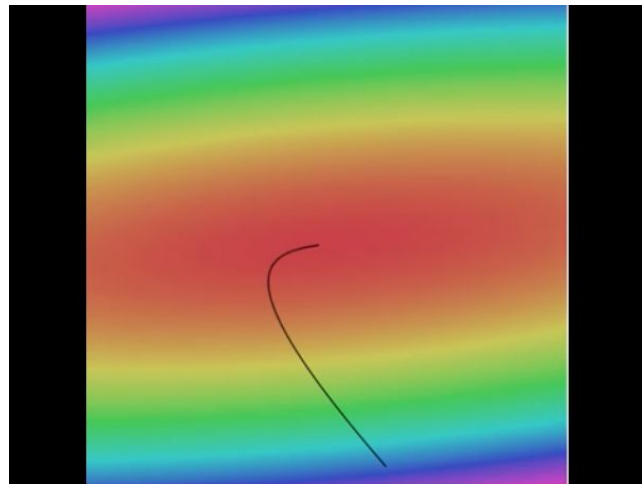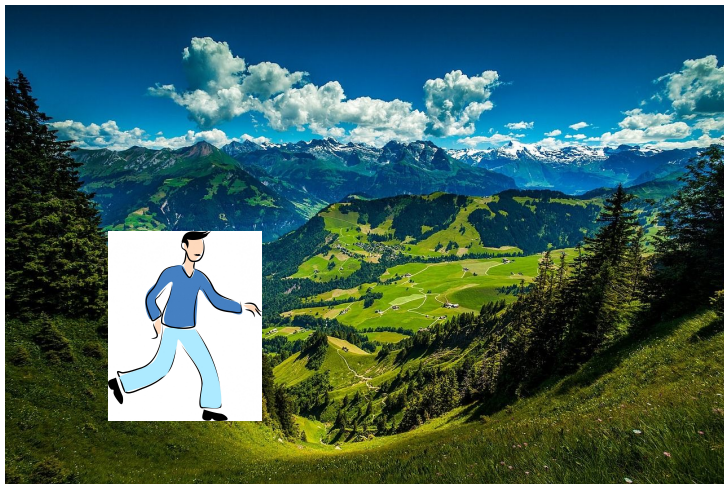

```
# Vanilla Gradient Descent

while True:
    weights_grad = evaluate_gradient(loss_fun, data, weights)
    weights += - step_size * weights_grad # perform parameter update
```

Where we are now...

Mini-batch SGD

Loop:
1. **Sample** a batch of data
2. **Forward** prop it through the graph (network), get loss
3. **Backprop** to calculate the gradients
4. **Update** the parameters using the gradient

Where we are now...

Hardware + Software

PyTorch

TensorFlow

Next: Training Neural Networks

Overview

1. **One time setup**
   activation functions, preprocessing, weight initialization, regularization, gradient checking
2. **Training dynamics**
   babysitting the learning process, parameter updates, hyperparameter optimization
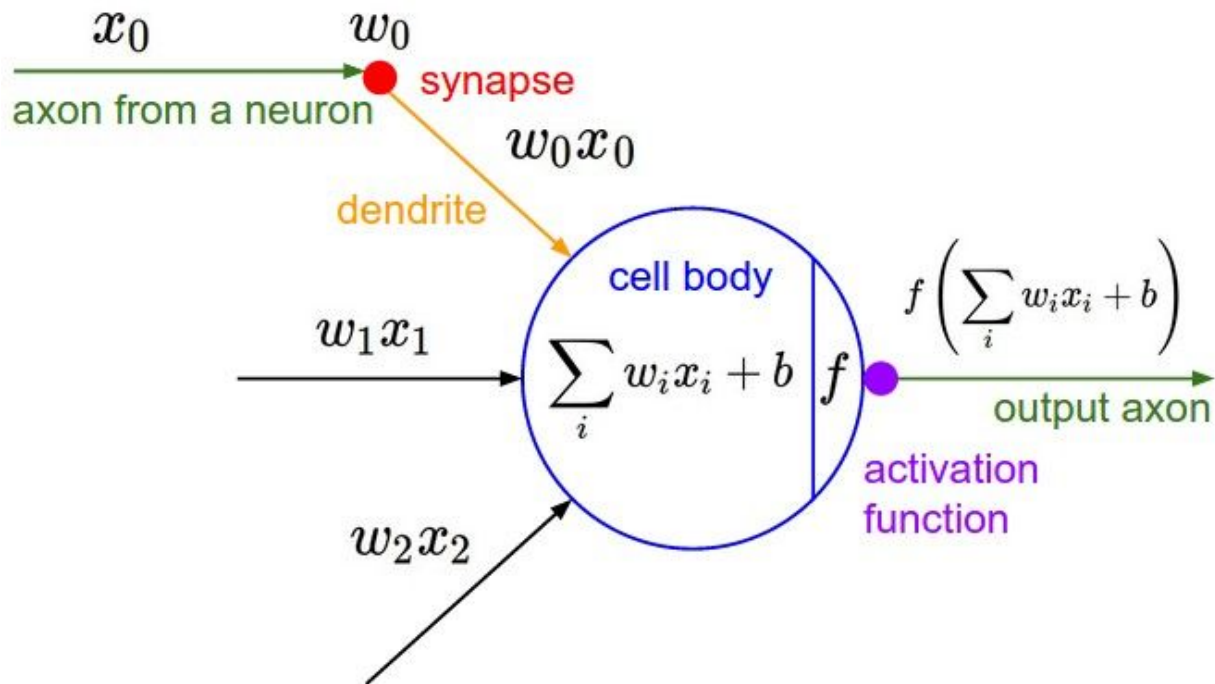3. **Evaluation**
   model ensembles, test-time augmentation

Part 1

- Activation Functions
- Data Preprocessing
- Weight Initialization
- Batch Normalization
- Babysitting the Learning Process
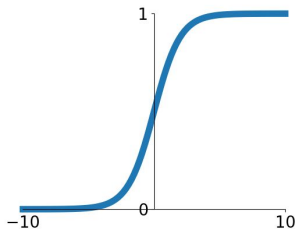- Hyperparameter Optimization

Activation Functions

Activation Functions

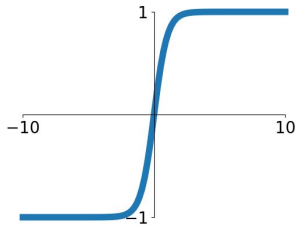# Activation Functions

Sigmoid

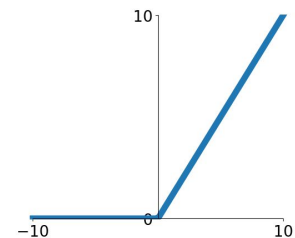$$\sigma(x) = \frac{1}{1+e^{-x}}$$

tanh

$$\tanh(x)$$

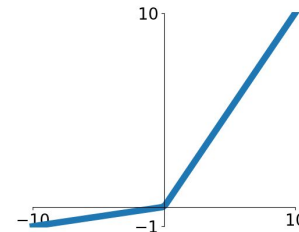**ReLU**

$$\max(0, x)$$

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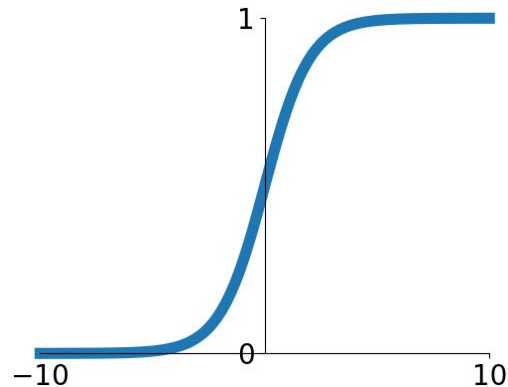
$$\sigma(x) = 1/(1 + e^{-x})$$

Sigmoid

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

Activation Functions

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

- 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

Sigmoid

x

$\dfrac{\partial \sigma}{\partial x}$ sigmoid gate

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

$\dfrac{\partial L}{\partial x} = \dfrac{\partial \sigma}{\partial x}\dfrac{\partial L}{\partial \sigma}$

$\dfrac{\partial L}{\partial \sigma}$

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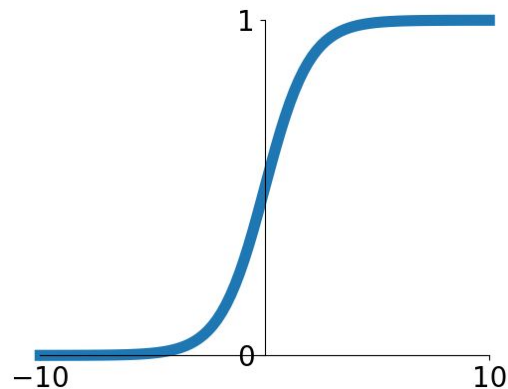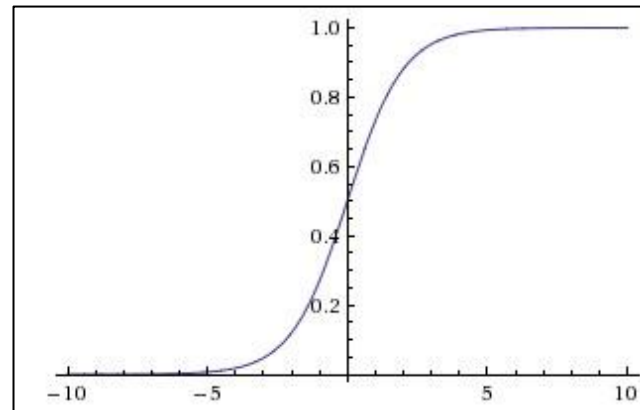
Activation Functions

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

Sigmoid

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

3 problems:

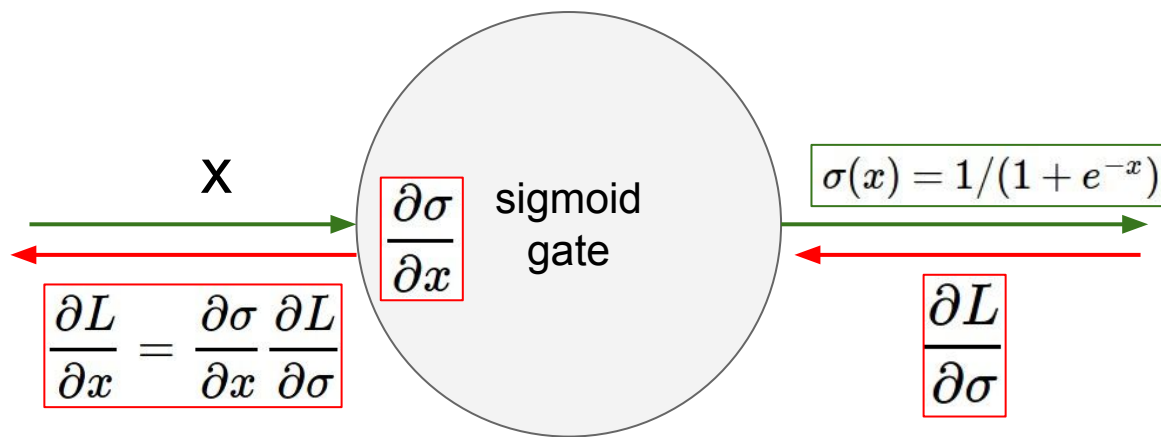1. Saturated neurons "kill" the gradients
2. Sigmoid outputs are not zero-centered

Consider what happens when the input to a neuron is always positive...
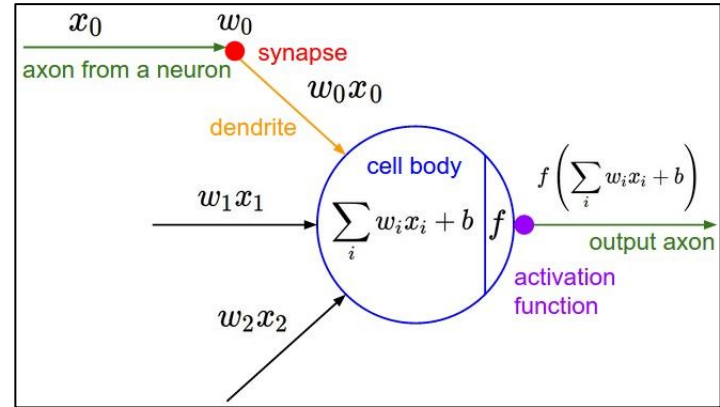
$$f\left(\sum_i w_i x_i + b\right)$$

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

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

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

allowed gradient update directions

zig zag path

allowed gradient update directions

hypothetical optimal w vector
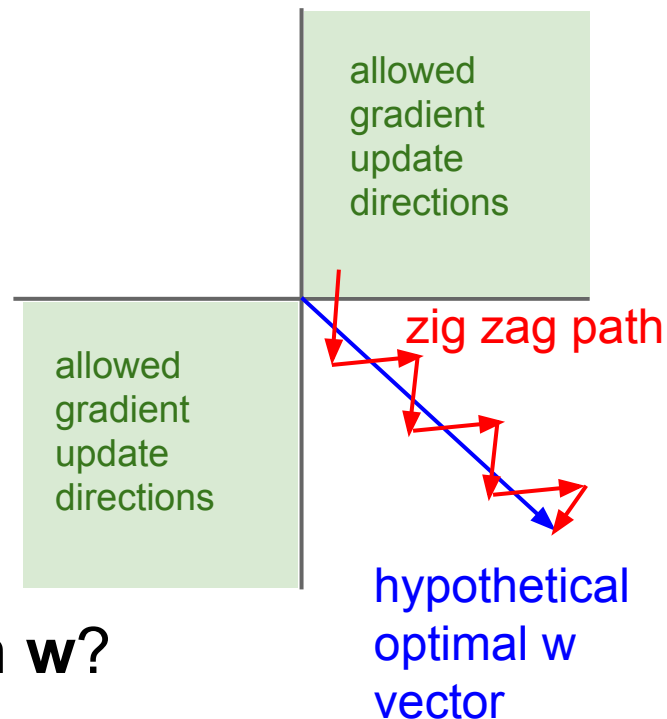
What can we say about the gradients on **w**?
Always all positive or all negative :(

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

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

allowed
gradient
update
directions

zig zag path

allowed
gradient
update
directions

hypothetical
optimal w
vector
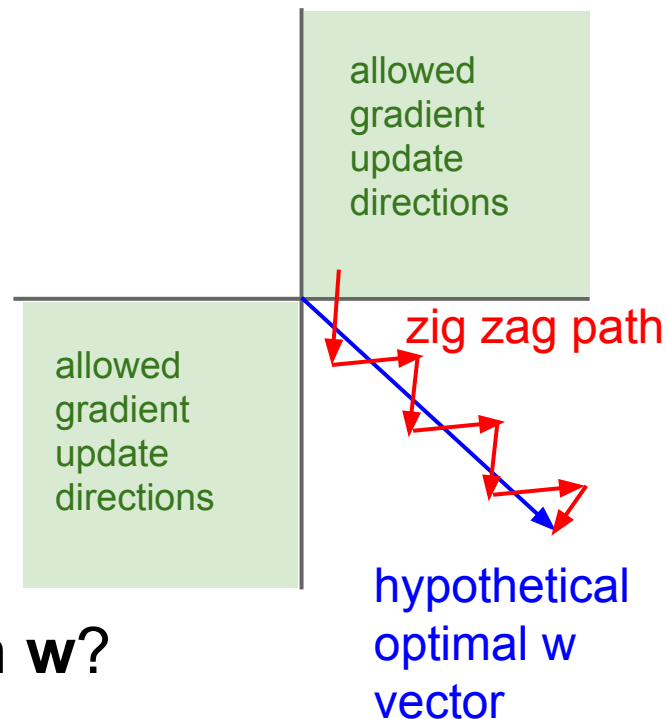
What can we say about the gradients on **w**?
Always all positive or all negative :(
(For a single element! Minibatches help)

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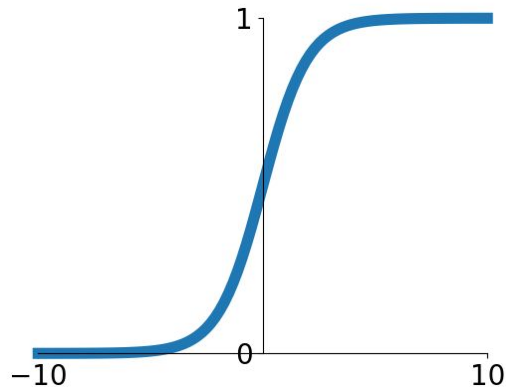
$$\sigma(x) = 1/(1 + e^{-x})$$

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

3 problems:

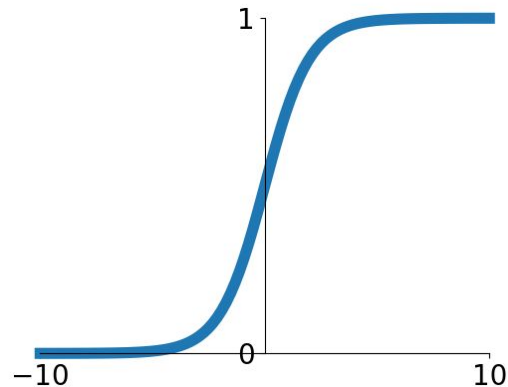1. Saturated neurons "kill" the gradients
2. Sigmoid outputs are not zero-centered
3. exp() is a bit compute expensive

Sigmoid

Activation Functions

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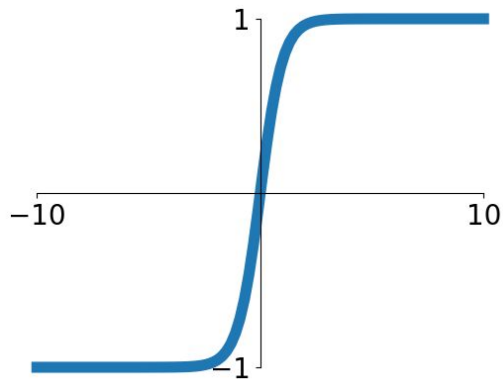
- Squashes numbers to range [-1,1]
- zero centered (nice)
- still kills gradients when saturated :(

[LeCun et al., 1991]

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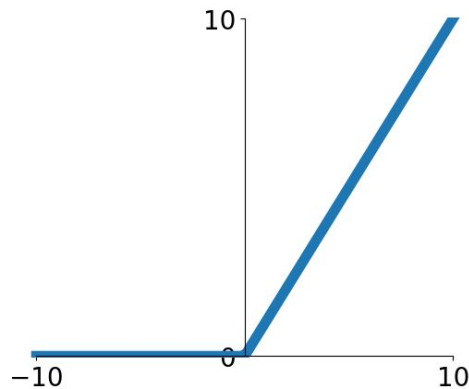
- Computes **f(x) = max(0,x)**

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

ReLU
(Rectified Linear Unit)

[Krizhevsky et al., 2012]

Activation Functions
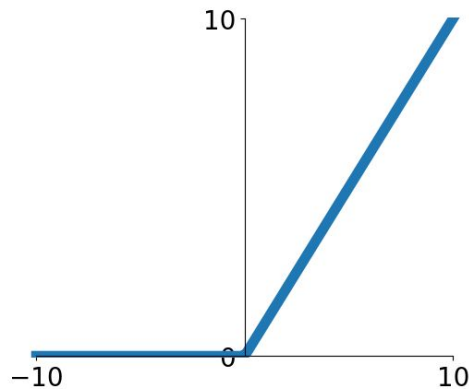
ReLU
(Rectified Linear Unit)

- 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

Activation Functions
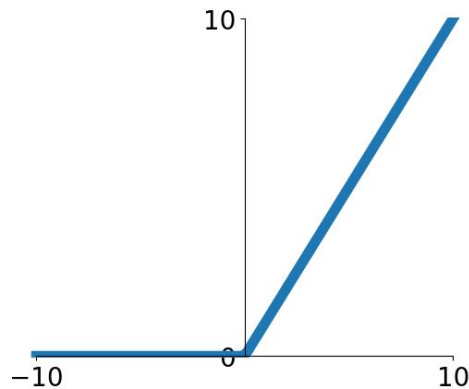
ReLU
(Rectified Linear Unit)

- 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?

$$\frac{\partial \sigma}{\partial x}$$ ReLU gate $$\sigma(x) = \max(0, x)$$

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

$$\frac{\partial L}{\partial \sigma}$$

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

DATA CLOUD

active ReLU

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

dead ReLU
will never activate
=> never update

Activation Functions

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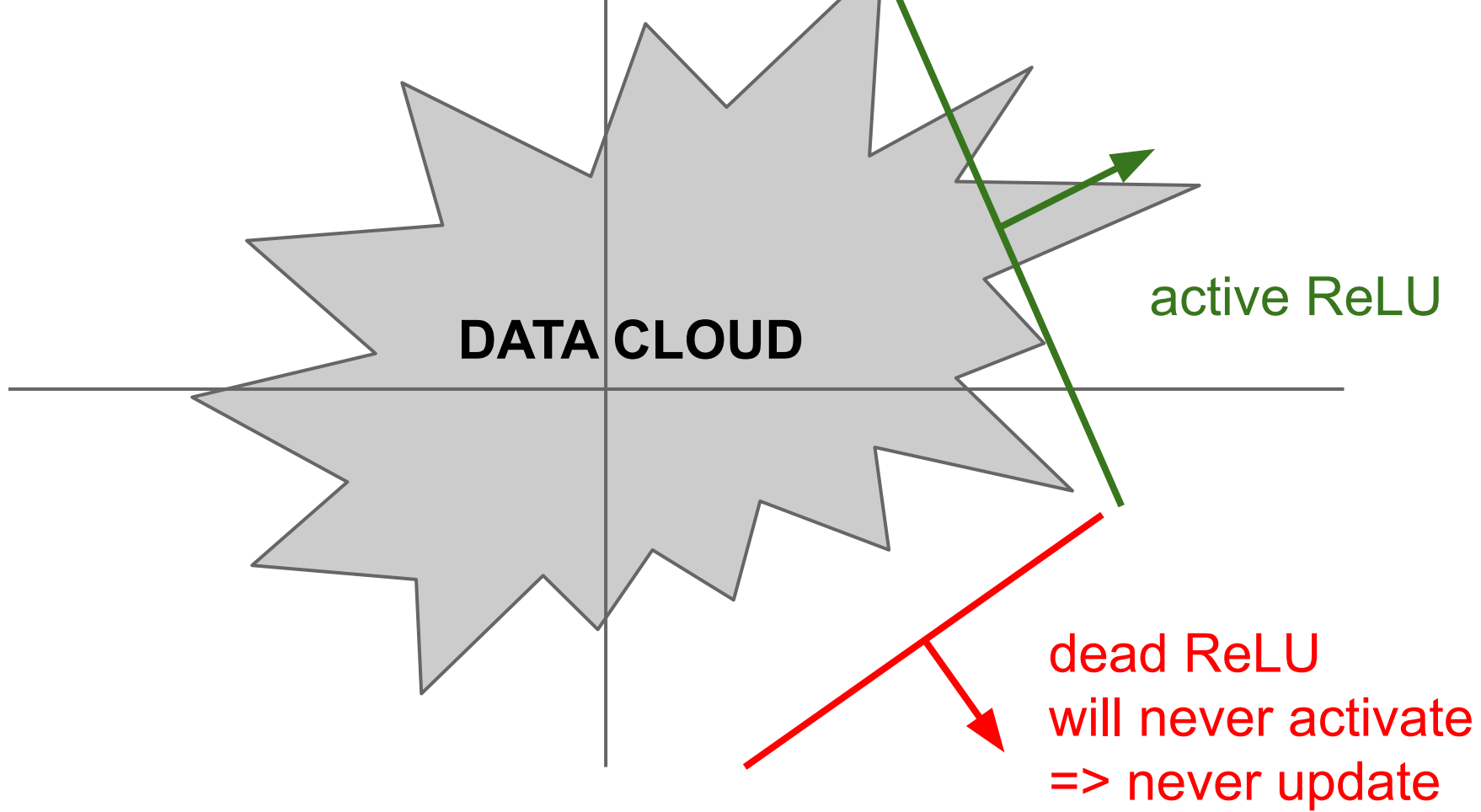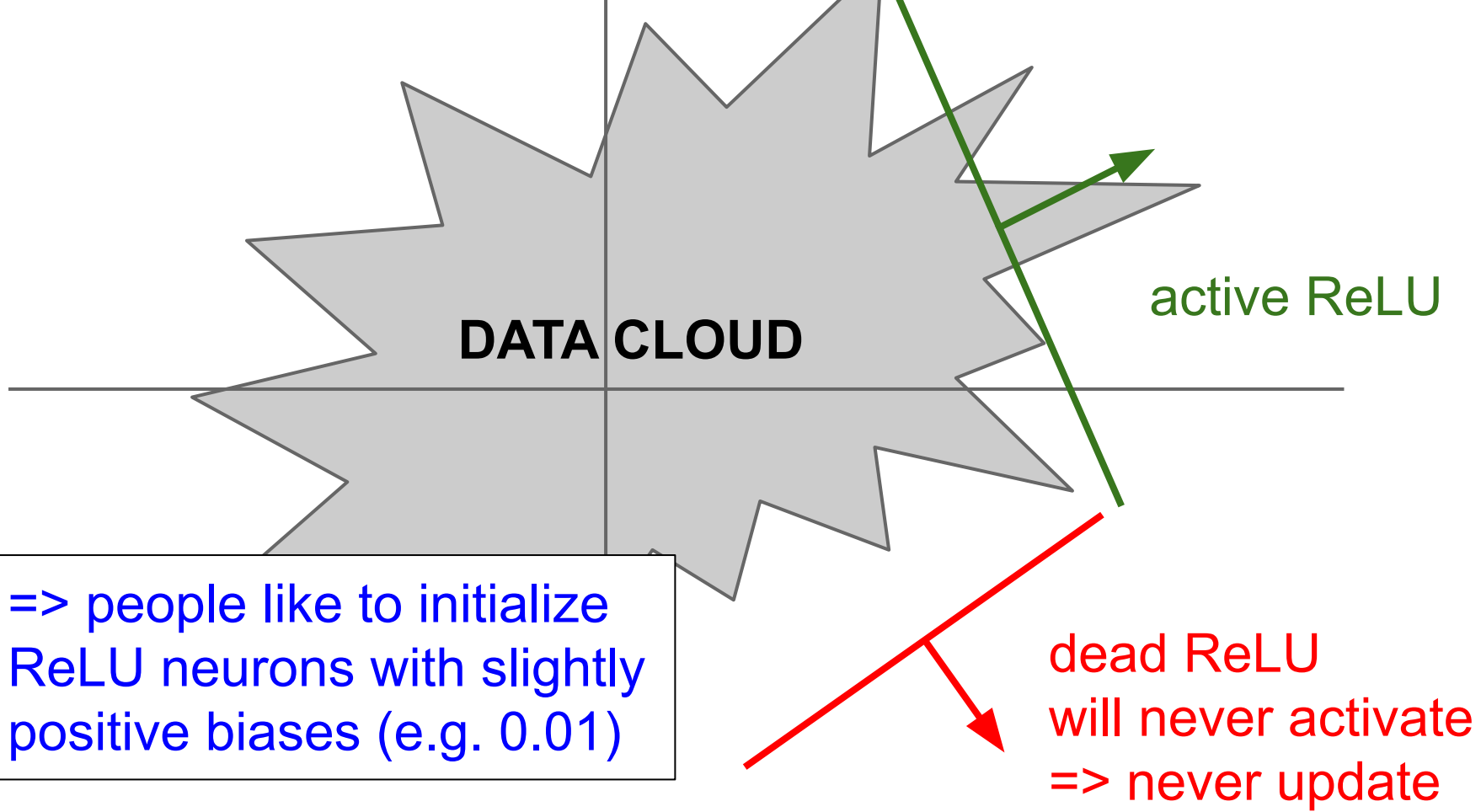
Leaky ReLU

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

Activation Functions

[Mass et al., 2013]
[He et al., 2015]
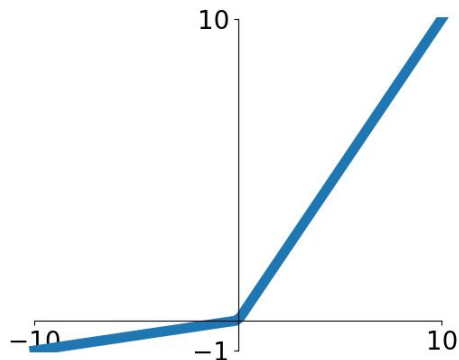
- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- **will not "die".**

Leaky ReLU

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

Parametric Rectifier (PReLU)

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

backprop into \alpha
(parameter)

Activation Functions

Exponential Linear Units (ELU)

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\left(\exp(x) - 1\right) & \text{if } x \leq 0 \end{cases}$$
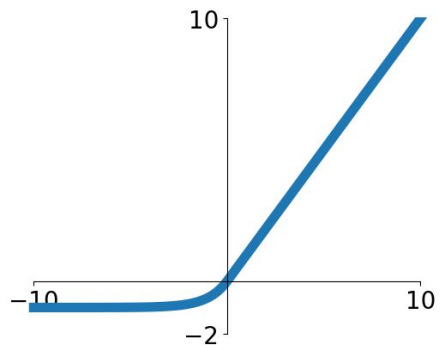
- All benefits of ReLU
- Closer to zero mean outputs
- Negative saturation regime compared with Leaky ReLU adds some robustness to noise

- Computation requires exp()

Maxout "Neuron"

- 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

original data   zero-centered data   normalized data

`X -= np.mean(X, axis = 0)`   `X /= np.std(X, axis = 0)`
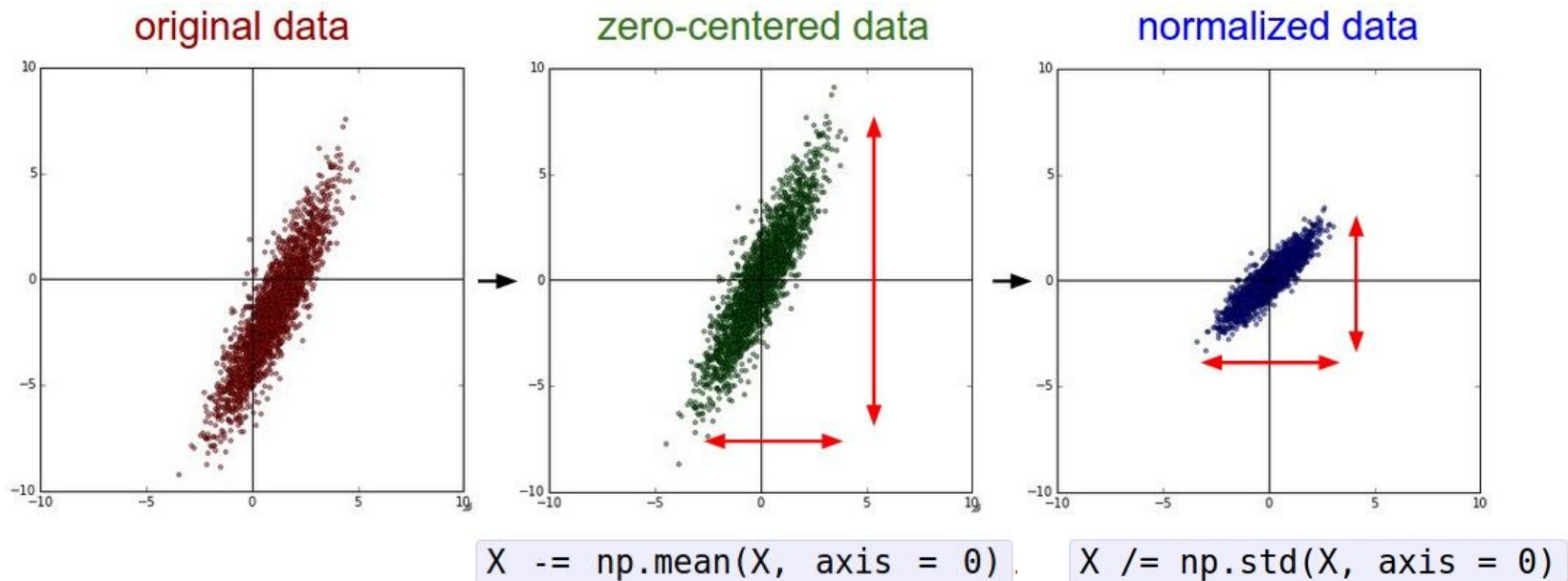
(Assume X [NxD] is data matrix,
each example in a row)

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

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

allowed gradient update directions

allowed gradient update directions

zig zag path

hypothetical optimal w vector
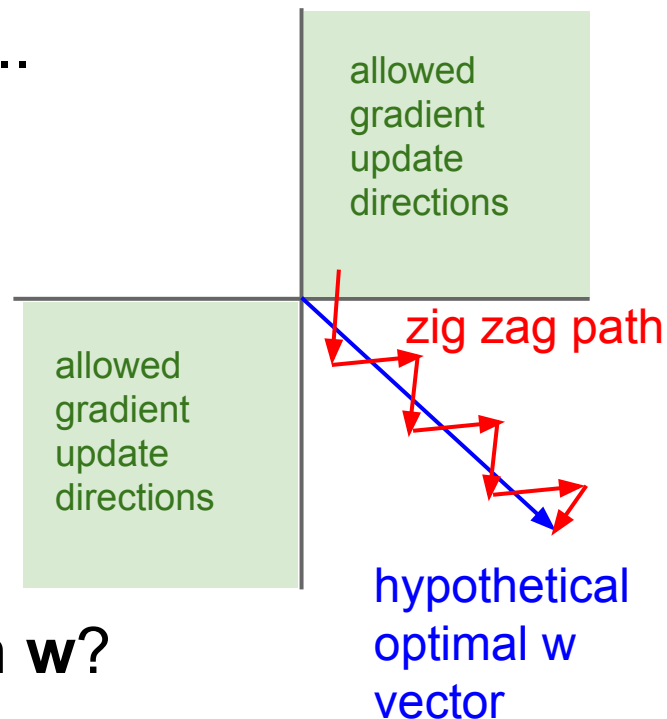
What can we say about the gradients on **w**?
Always all positive or all negative :(
(this is also why you want zero-mean data!)

Data Preprocessing

original data | zero-centered data | normalized data

```
X -= np.mean(X, axis = 0)        X /= np.std(X, axis = 0)
```
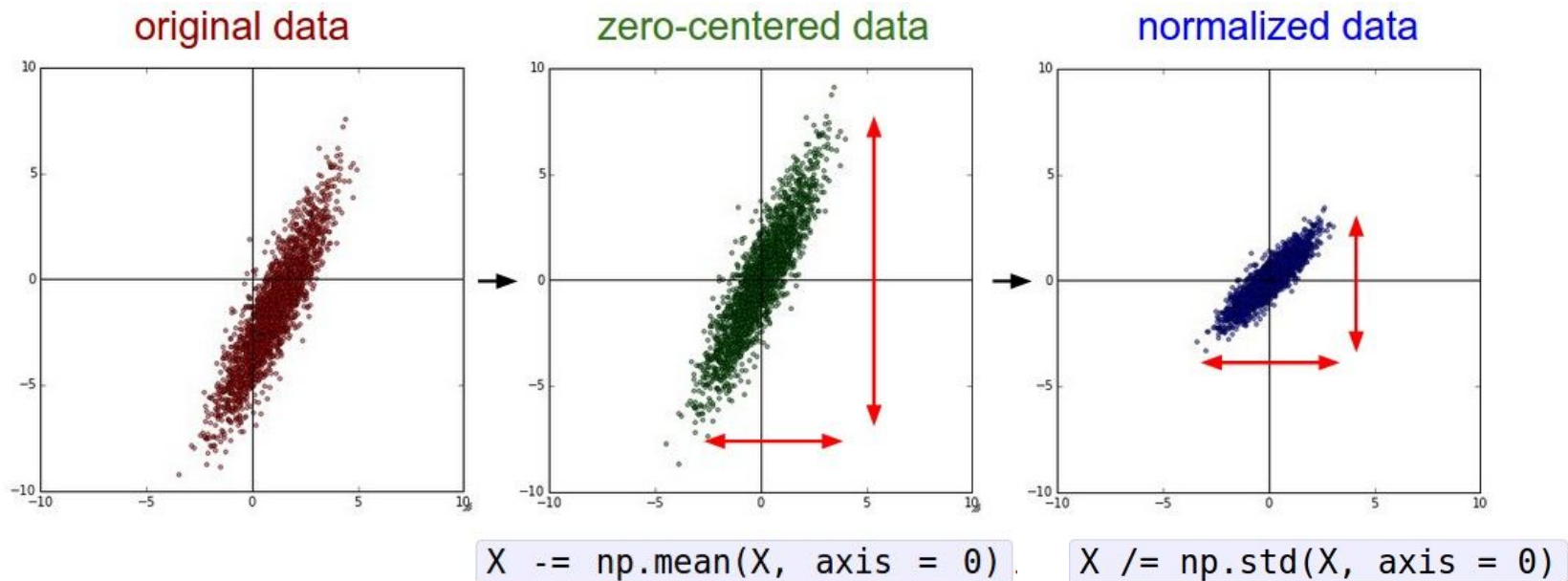
(Assume X [NxD] is data matrix, each example in a row)

Data Preprocessing

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

original data | decorrelated data | whitened data

(data has diagonal covariance matrix)

(covariance matrix is the identity matrix)
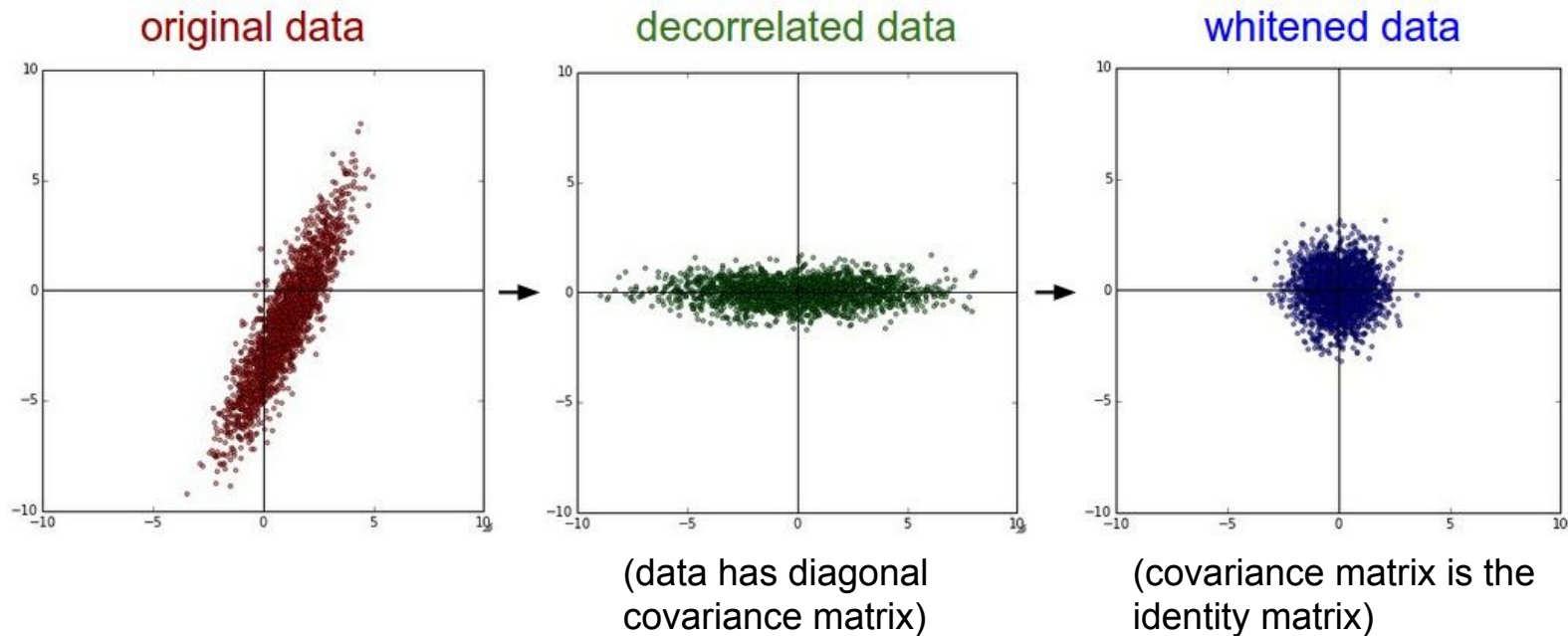
Data Preprocessing

Before normalization: classification loss very sensitive to changes in weight matrix; hard to optimize
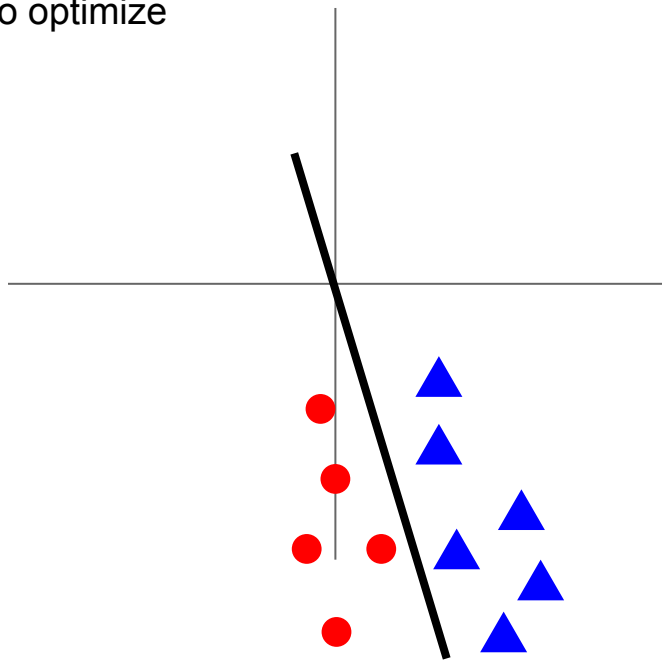
After normalization: less sensitive to small changes in weights; easier to optimize
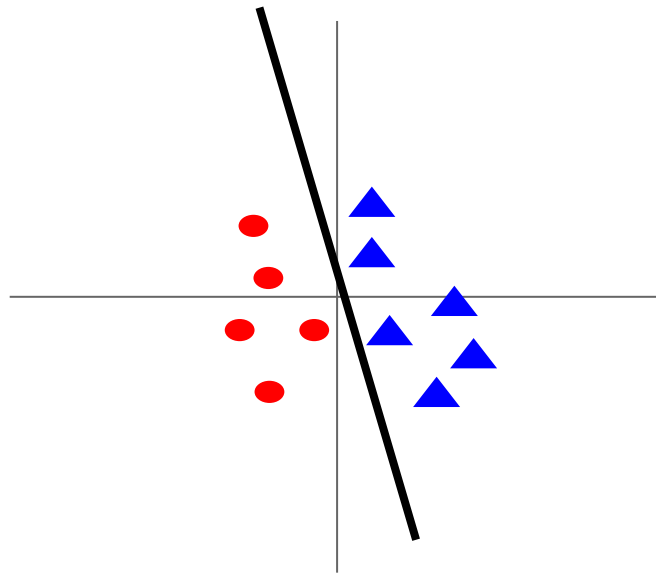
TLDR: In practice for Images: center only

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?
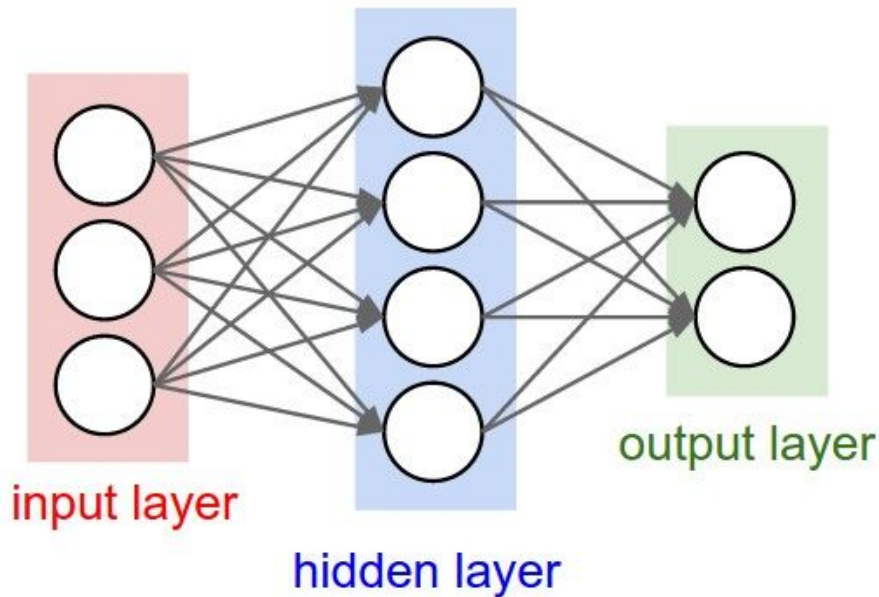
input layer

hidden layer

output layer

- 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?

Layer 1
mean=-0.00
std=0.49

Layer 2
mean=0.00
std=0.29

Layer 3
mean=0.00
std=0.18

Layer 4
mean=-0.00
std=0.11

Layer 5
mean=-0.00
std=0.07

Layer 6
mean=0.00
std=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 = 0.01 * 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 =(

| Layer 1 mean=-0.00 std=0.49 | Layer 2 mean=0.00 std=0.29 | Layer 3 mean=0.00 std=0.18 | Layer 4 mean=-0.00 std=0.11 | Layer 5 mean=-0.00 std=0.07 | Layer 6 mean=0.00 std=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

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?

| Layer 1 mean=0.00 std=0.87 | Layer 2 mean=-0.00 std=0.85 | Layer 3 mean=0.00 std=0.85 | Layer 4 mean=-0.00 std=0.85 | Layer 5 mean=0.00 std=0.85 | Layer 6 mean=-0.00 std=0.85 |

Weight Initialization: Activation statistics

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

Increase std of initial weights from 0.01 to 0.05

All activations saturate

Q: What do the gradients look like?

A: Local gradients all zero, no learning =(

Weight Initialization: "Xavier" Initialization

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

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

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!

Layer 1	Layer 2	Layer 3	Layer 4	Layer 5	Layer 6
mean=-0.00	mean=-0.00	mean=0.00	mean=0.00	mean=0.00	mean=-0.00
std=0.63	std=0.49	std=0.41	std=0.36	std=0.32	std=0.30

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

Weight Initialization: "Xavier" Initialization

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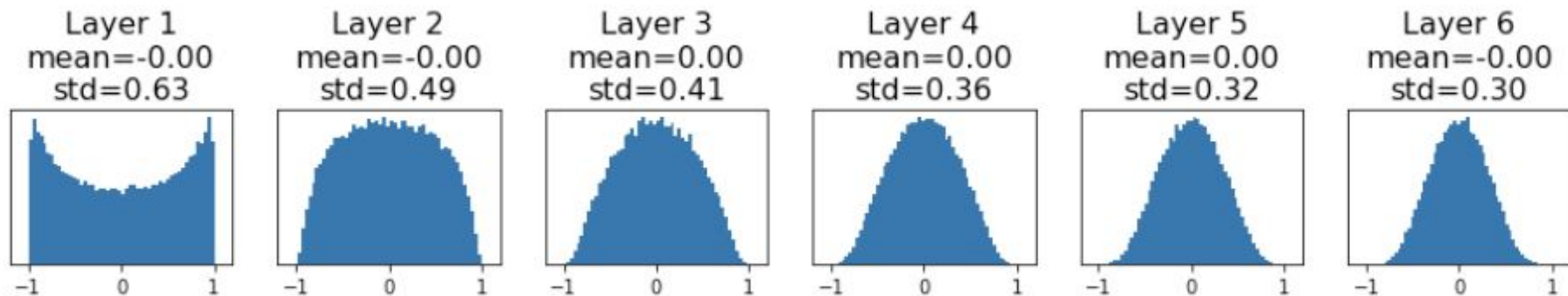
"Xavier" initialization:
std = 1/sqrt(Din)

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

For conv layers, Din is kernel_size^2 * input_channels

| Layer 1 mean=-0.00 std=0.63 | Layer 2 mean=-0.00 std=0.49 | Layer 3 mean=0.00 std=0.41 | Layer 4 mean=0.00 std=0.36 | Layer 5 mean=0.00 std=0.32 | Layer 6 mean=-0.00 std=0.30 |

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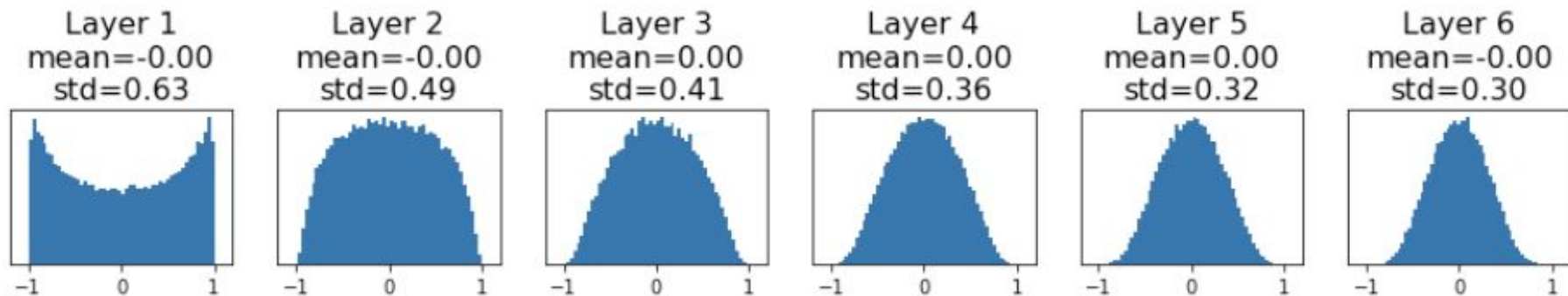
Weight Initialization: "Xavier" Initialization

```
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

Derivation:

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

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

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

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

Weight Initialization: What about ReLU?
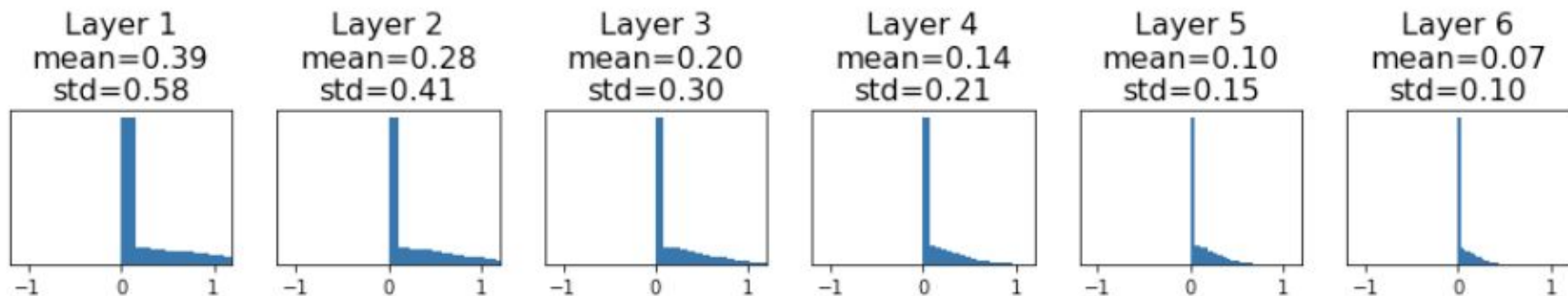
Change from tanh to ReLU

```
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)
```

Weight Initialization: What about ReLU?

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

Change from tanh to ReLU

Xavier assumes zero centered activation function

Activations collapse to zero again, no learning =(

| Layer 1 mean=0.39 std=0.58 | Layer 2 mean=0.28 std=0.41 | Layer 3 mean=0.20 std=0.30 | Layer 4 mean=0.14 std=0.21 | Layer 5 mean=0.10 std=0.15 | Layer 6 mean=0.07 std=0.10 |

Weight Initialization: Kaiming / MSRA 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(2/Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

ReLU correction: std = sqrt(2 / Din)

"Just right": Activations are nicely scaled for all layers!
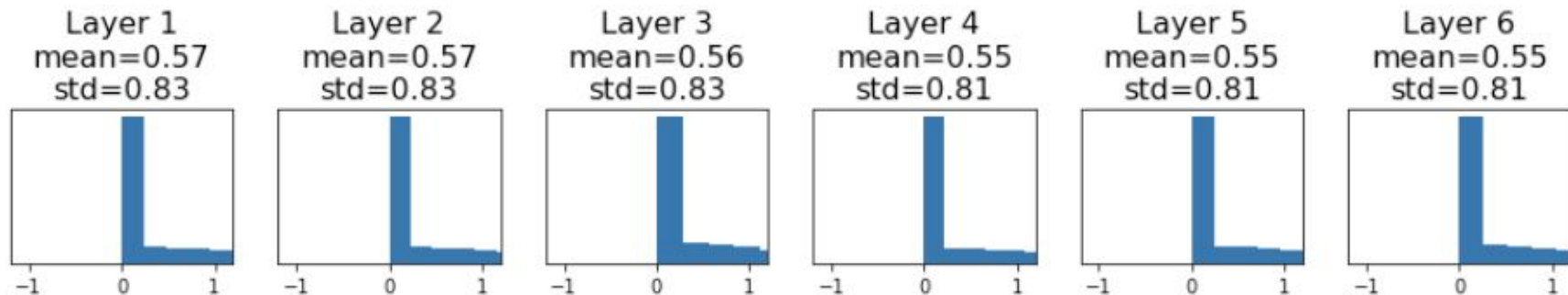
He et al, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification", ICCV 2015

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

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

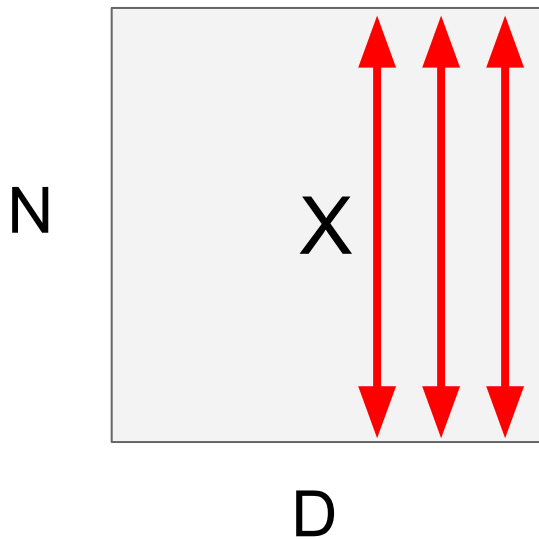consider a batch of activations at some layer. To make each dimension zero-mean unit-variance, apply:

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

this is a vanilla differentiable function...

Batch Normalization

Input: $x : N \times D$

N

X

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

Input: $x : N \times D$

N

X

D

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

Per-channel mean, shape is D

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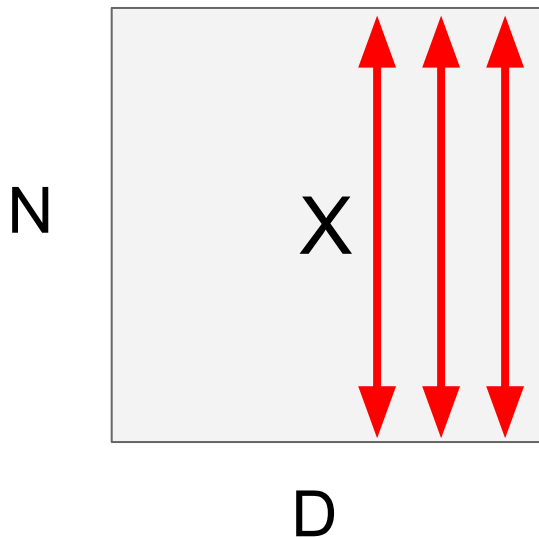
Per-channel var, shape is D

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

Normalized x, Shape is N x D

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

Batch Normalization

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

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

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

$\hat{x}_{i,j} = \dfrac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$ — 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: $x : N \times D$

$\mu_j =$ (Running) average of values seen during training   Per-channel mean, shape is D

Learnable scale and shift parameters:

$\gamma, \beta : D$

$\sigma_j^2 =$ (Running) average of values seen during training   Per-channel var, shape is D

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

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

Batch Normalization for
convolutional networks
(Spatial Batchnorm, BatchNorm2D)

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

Normalize

$$\boldsymbol{\mu}, \boldsymbol{\sigma}: \texttt{1} \times \texttt{D}$$
$$\gamma, \beta: \texttt{1} \times \texttt{D}$$
$$\texttt{y} = \gamma(\texttt{x}-\boldsymbol{\mu})/\sigma+\beta$$

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

Normalize

$$\boldsymbol{\mu}, \boldsymbol{\sigma}: \texttt{1} \times \texttt{C} \times \texttt{1} \times \texttt{1}$$
$$\gamma, \beta: \texttt{1} \times \texttt{C} \times \texttt{1} \times \texttt{1}$$
$$\texttt{y} = \gamma(\texttt{x}-\boldsymbol{\mu})/\sigma+\beta$$
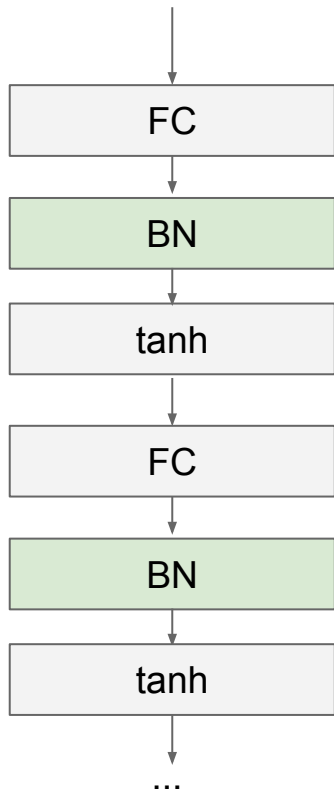
Batch Normalization
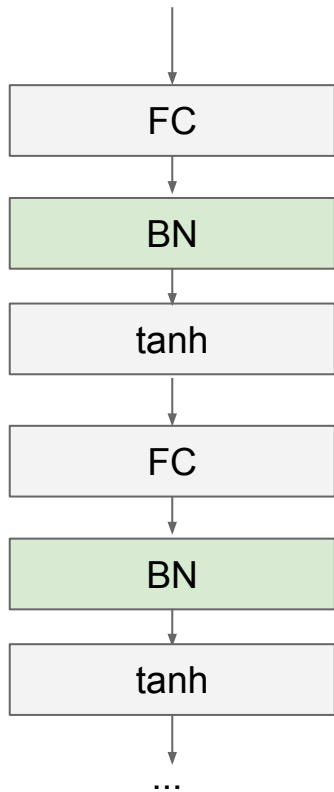
Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

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

Batch Normalization

- 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

Layer Normalization for fully-connected networks
Same behavior at train and test!
Can be used in recurrent networks

$$\texttt{x: N × D}$$

Normalize ↓

$$\boldsymbol{\mu},\boldsymbol{\sigma}: \texttt{1 × D}$$

$$\gamma,\beta: \texttt{1 × D}$$

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

$$\texttt{x: N × D}$$

Normalize ↓

$$\boldsymbol{\mu},\boldsymbol{\sigma}: \texttt{N × 1}$$

$$\gamma,\beta: \texttt{1 × D}$$

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

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

Instance Normalization

Batch Normalization for convolutional networks

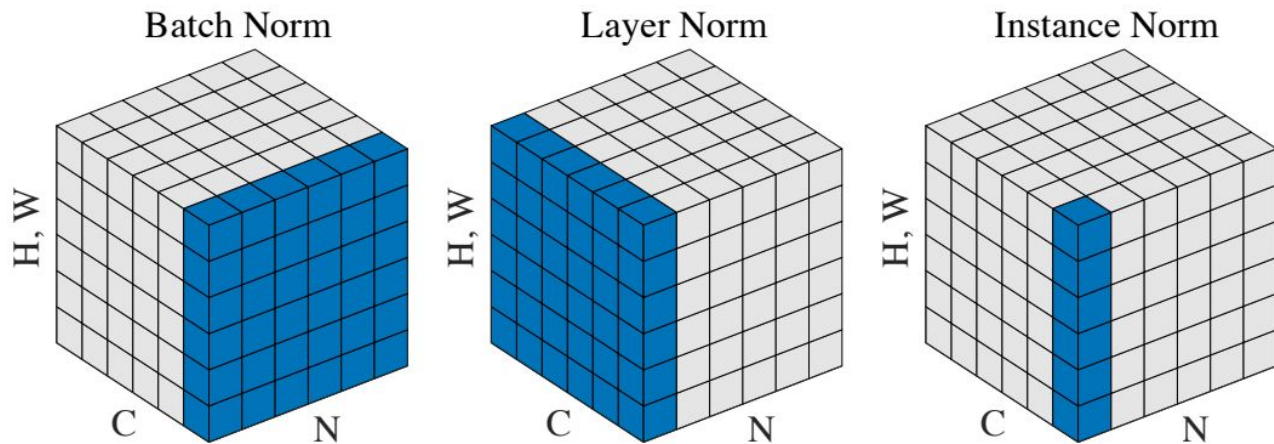**Instance Normalization** for convolutional networks
Same behavior at train / test!

Normalize

$$\text{x: } N×C×H×W$$

$$\mu,\sigma: 1×C×1×1$$

$$\gamma,\beta: 1×C×1×1$$

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

Normalize

$$\text{x: } N×C×H×W$$

$$\mu,\sigma: N×C×1×1$$

$$\gamma,\beta: 1×C×1×1$$

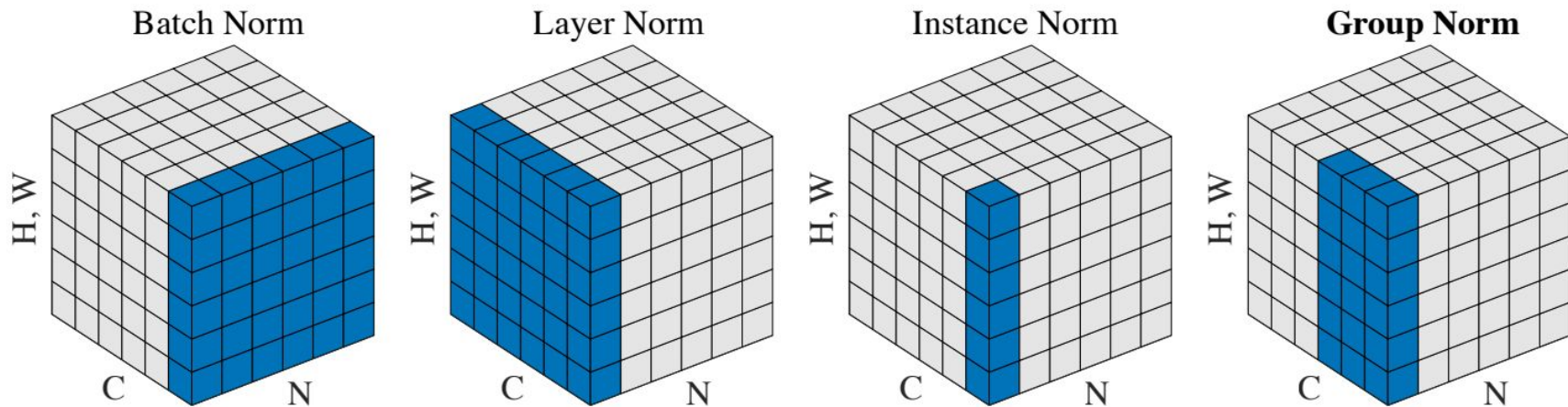$$y = \gamma(x-\mu)/\sigma+\beta$$

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

We looked in detail at:

TLDRs

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

Next time:
Training Neural Networks, Part 2

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