

Lecture 6: Training Neural Networks

Administrative

Assignment 1 due Friday October 20, 11:59pm

- Before submitting your work, please be sure to read the instructions on the course website carefully (Assignments Tab), and follow the steps mentioned in “Submitting your work”. This will ensure Gradescope grades your work correctly.

Assignment 2 will also be released **October 20th**

Administrative

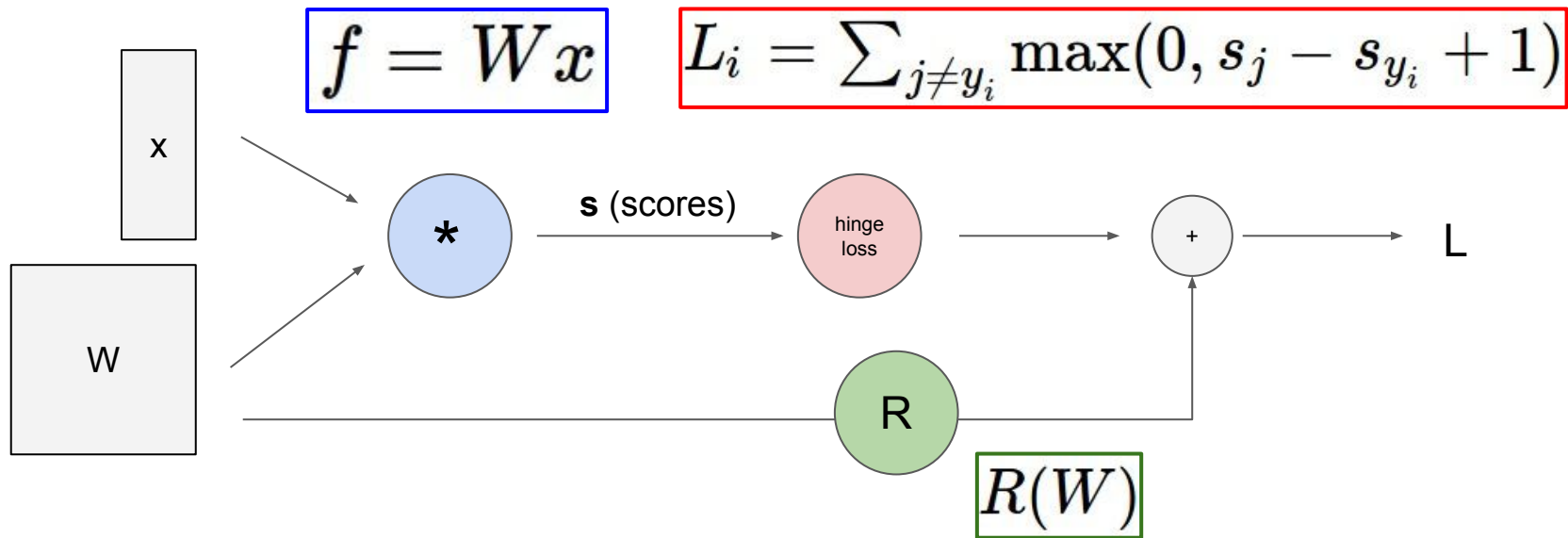
In-class quiz on **Thursday October 26**

Project proposal due **Friday October 27**, 11:59pm

Some resources on project design available on the website

Where we are now...

Computational graphs



Where we are now...

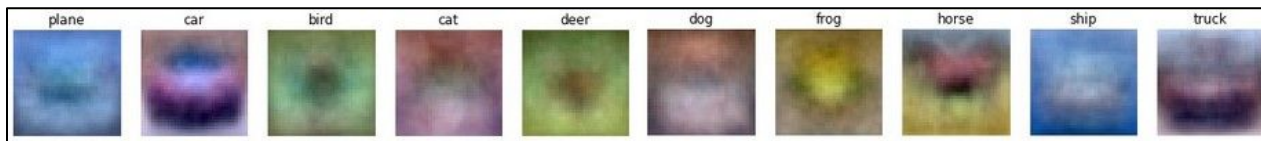
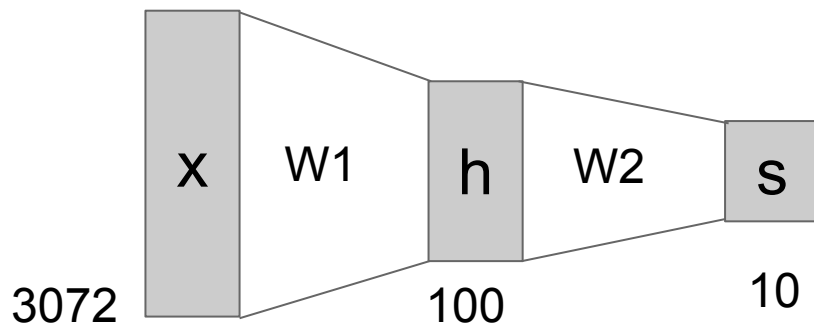
Neural Networks

Linear score function:

$$f = Wx$$

2-layer Neural Network

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



Where we are now...

Convolutional Neural Networks

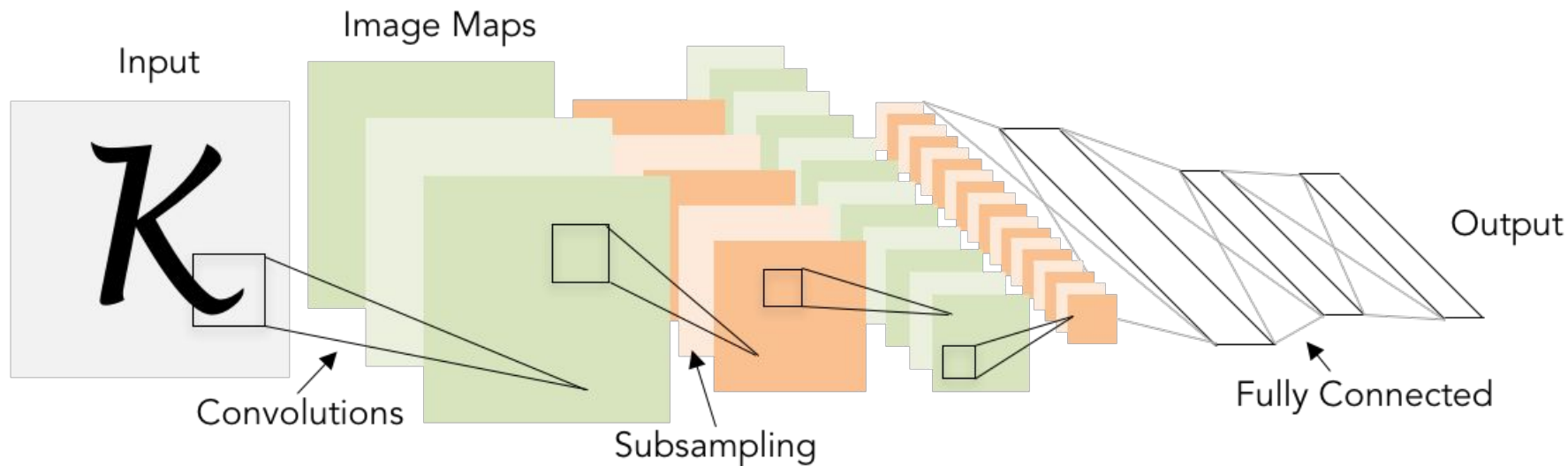
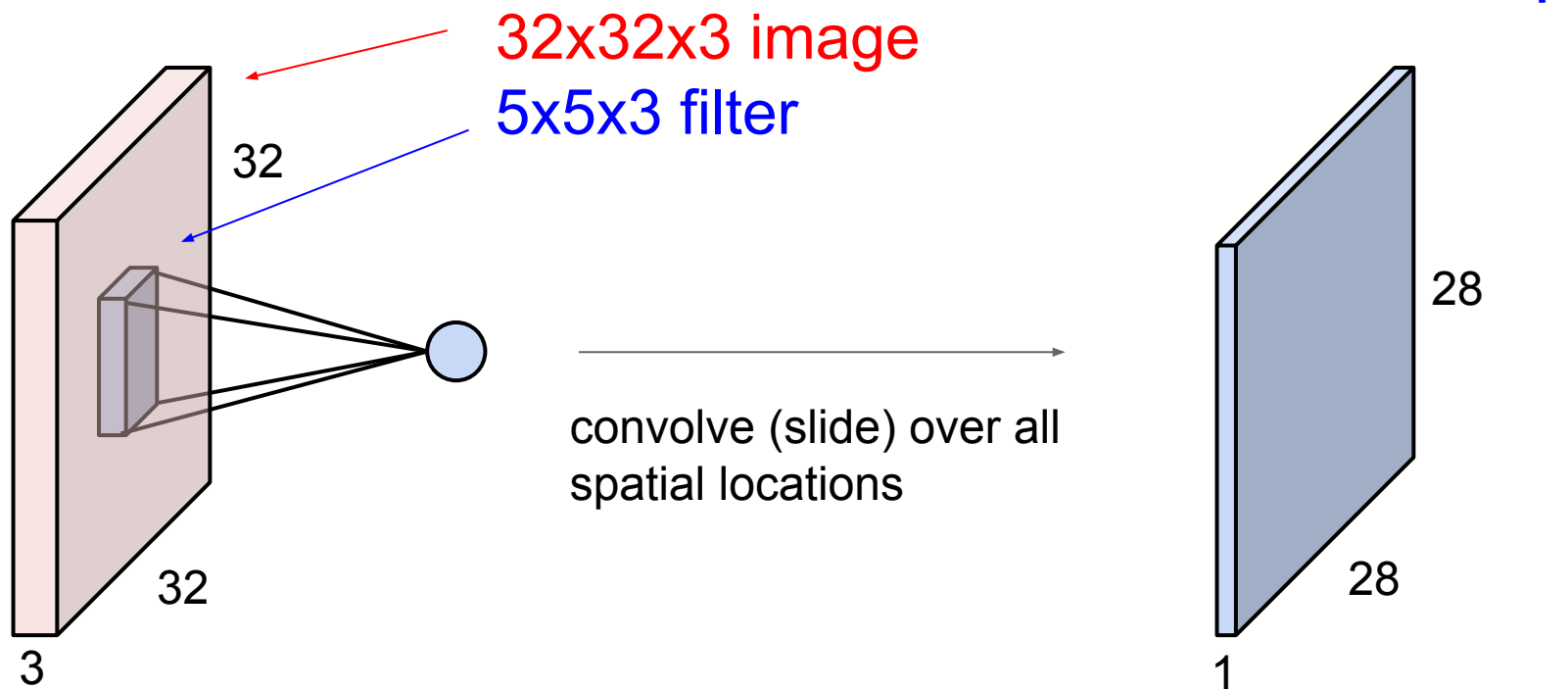


Illustration of LeCun et al. 1998 from CS231n 2017 Lecture 1

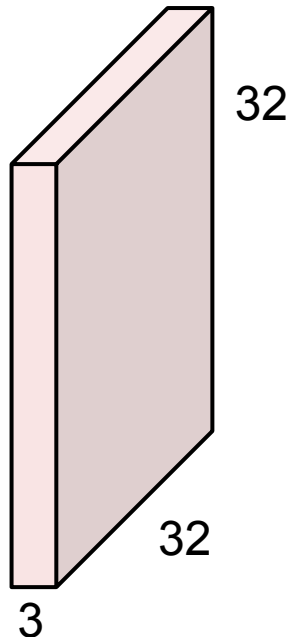
Where we are now...

Convolutional Layer

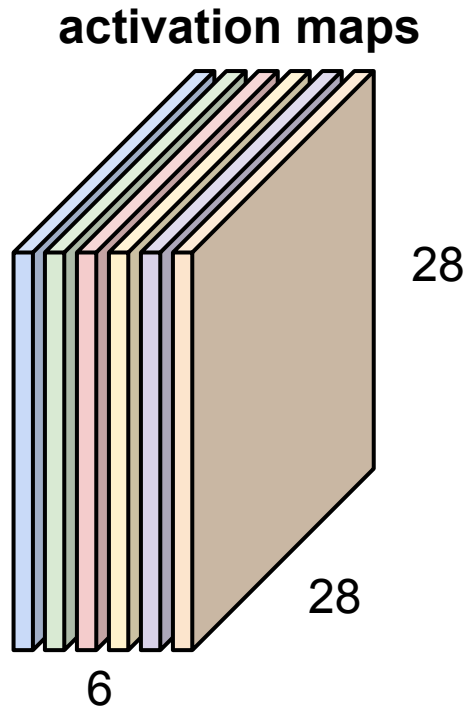


Where we are now...

Convolutional Layer



Convolution Layer

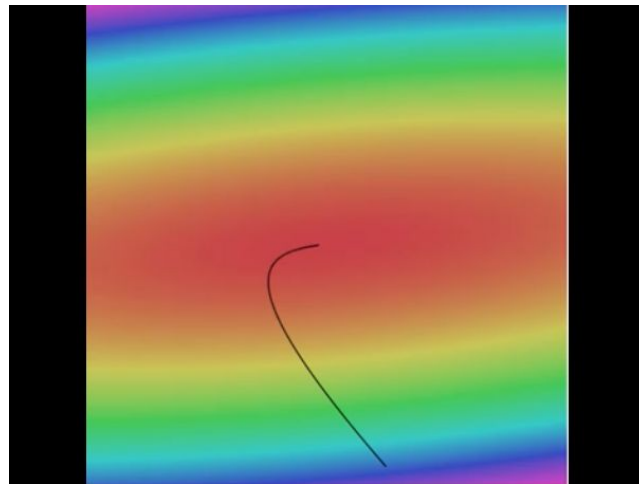
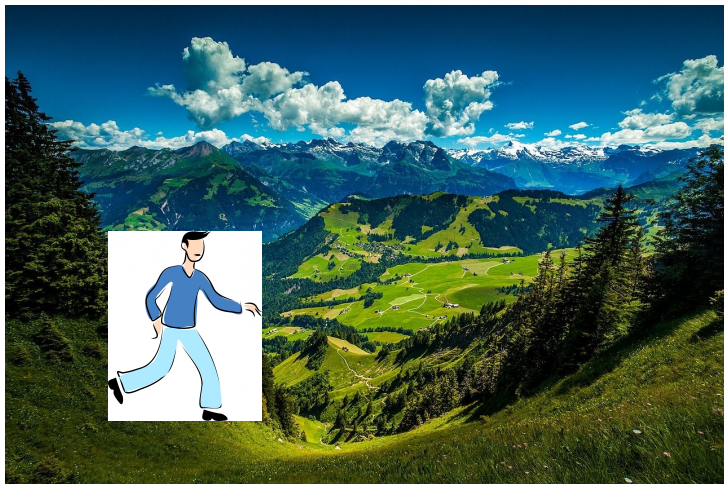


We stack these up to get a “new image” of size 28x28x6!

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

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
```

[Landscape image](#) is [CC0 1.0](#) public domain

[Walking man image](#) is [CC0 1.0](#) public domain

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

Today: 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**

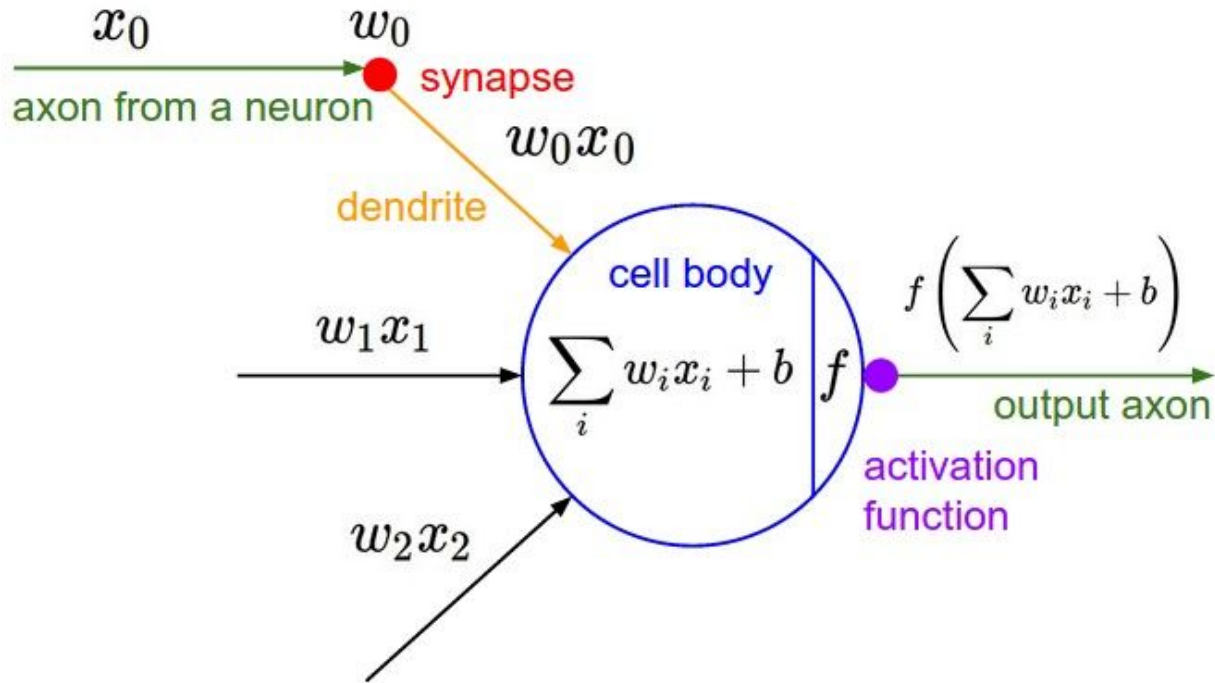
model ensembles, test-time augmentation, transfer learning

Part 1

- Activation Functions
- Data Preprocessing
- Weight Initialization
- Batch Normalization

Activation Functions

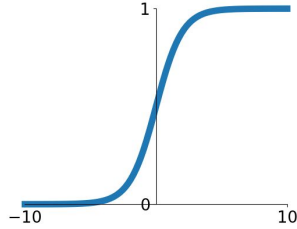
Activation Functions



Activation Functions

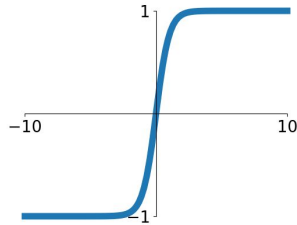
Sigmoid

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



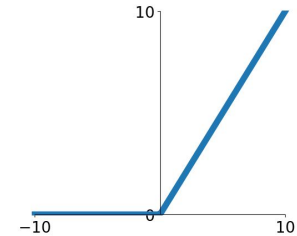
tanh

$$\tanh(x)$$



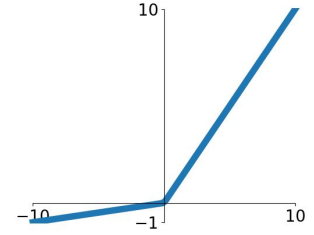
ReLU

$$\max(0, x)$$



Leaky ReLU

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

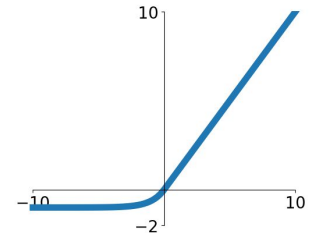


Maxout

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

ELU

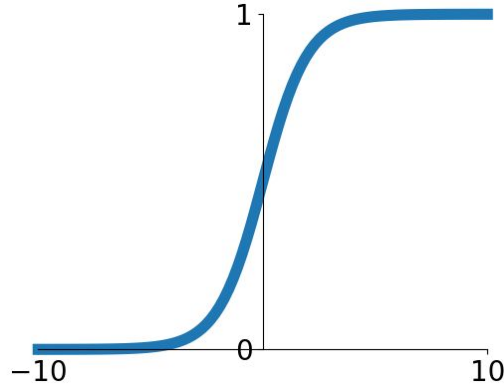
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Activation Functions

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

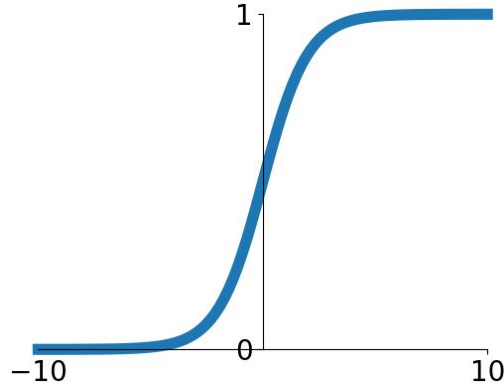
- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron



Sigmoid

Activation Functions

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

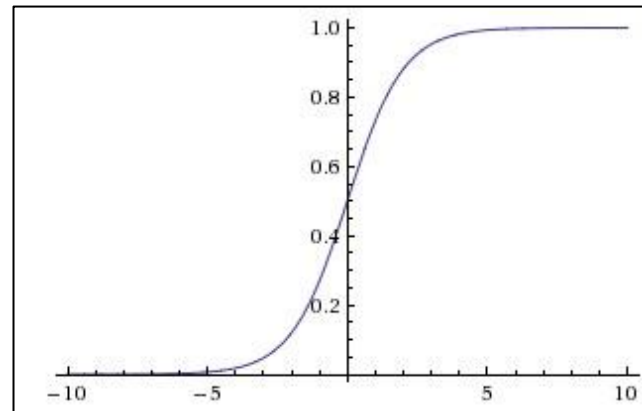
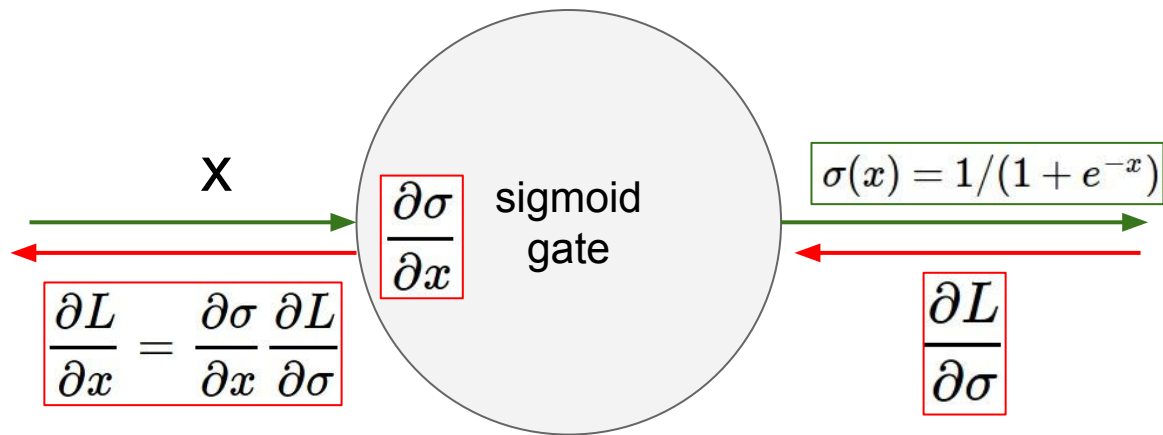


Sigmoid

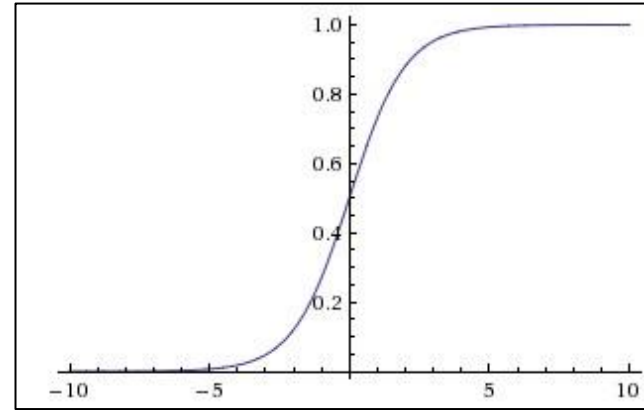
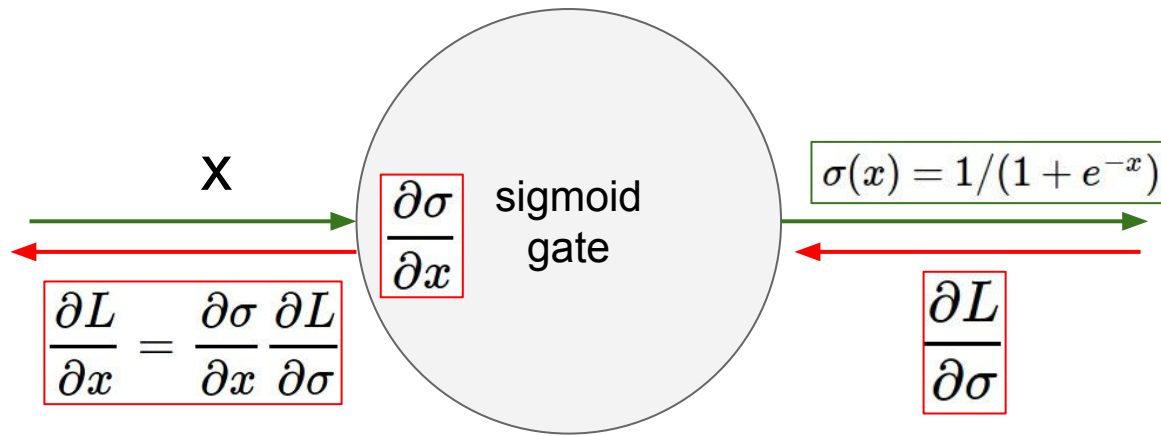
- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

3 problems:

1. Saturated neurons “kill” the gradients

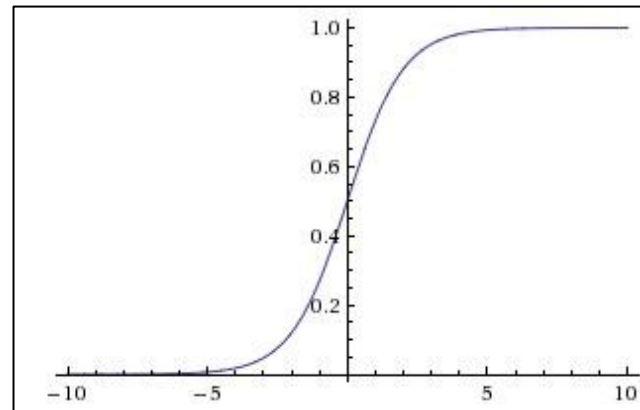
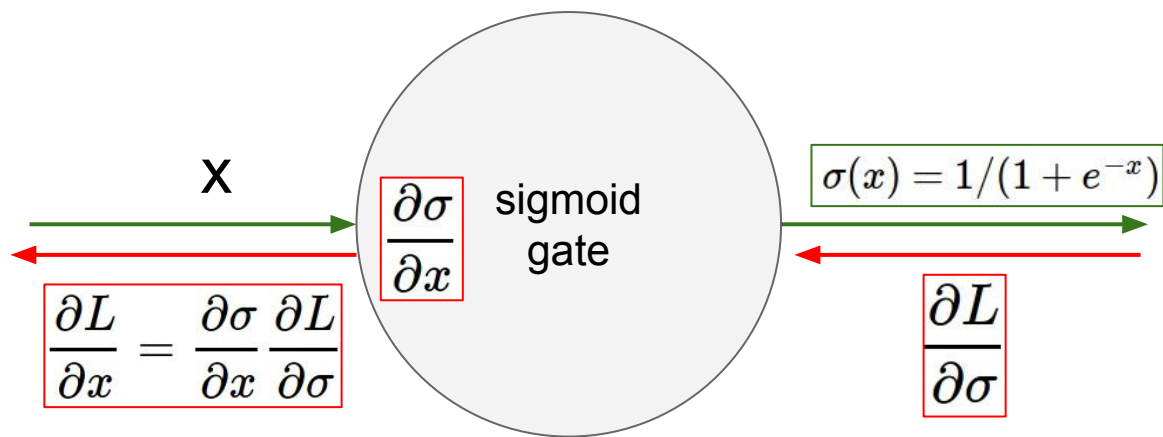


$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x) (1 - \sigma(x))$$



What happens when $x = -10$?

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

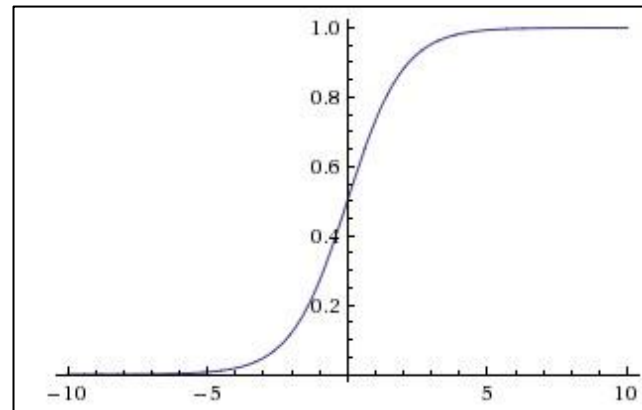
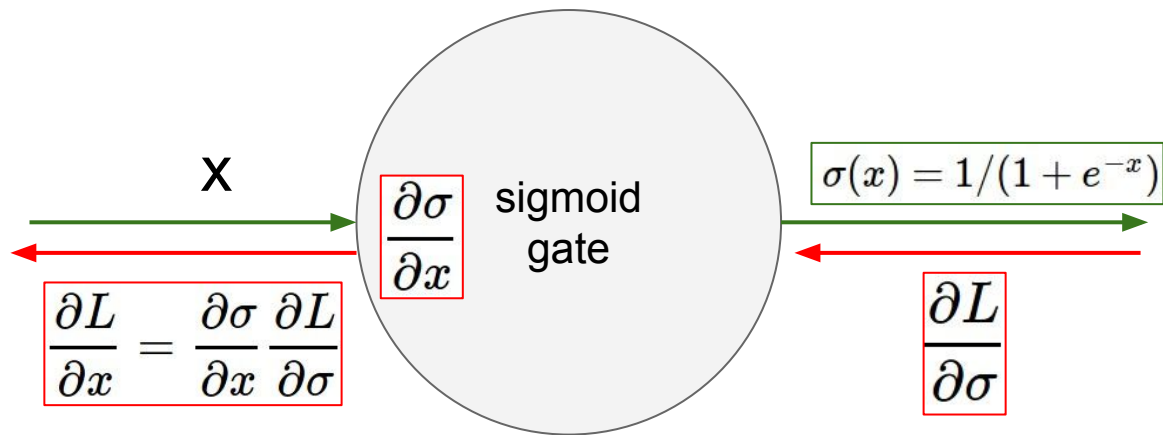


What happens when $x = -10$?

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

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

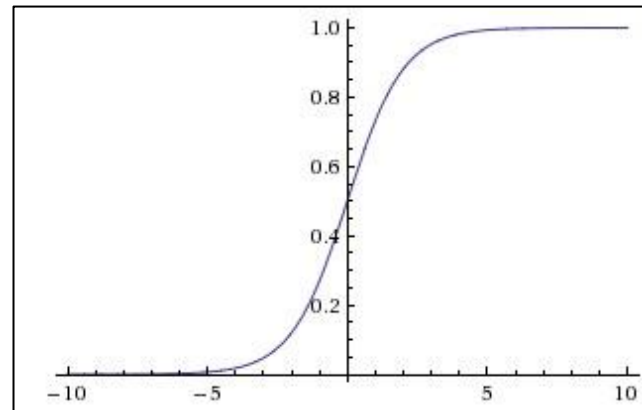
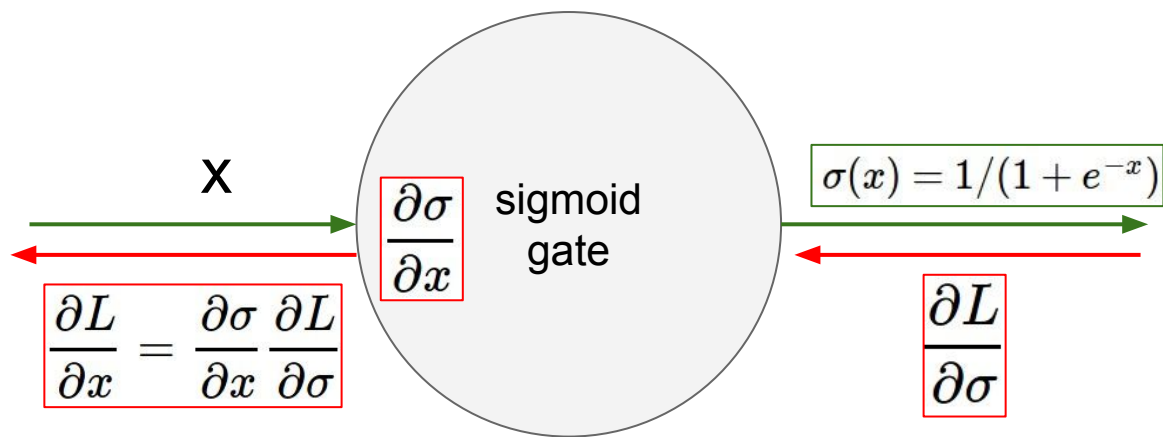
$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x) (1 - \sigma(x))$$



What happens when $x = -10$?

What happens when $x = 0$?

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

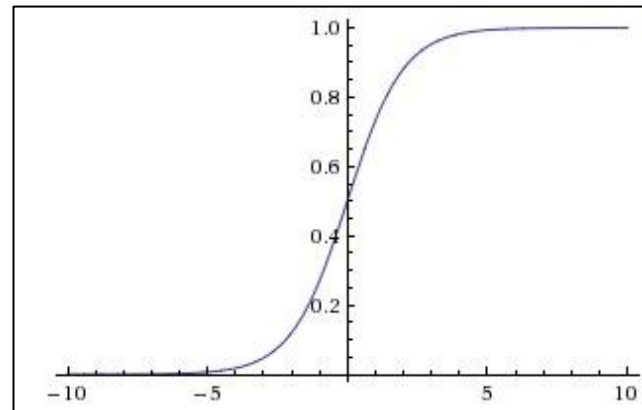
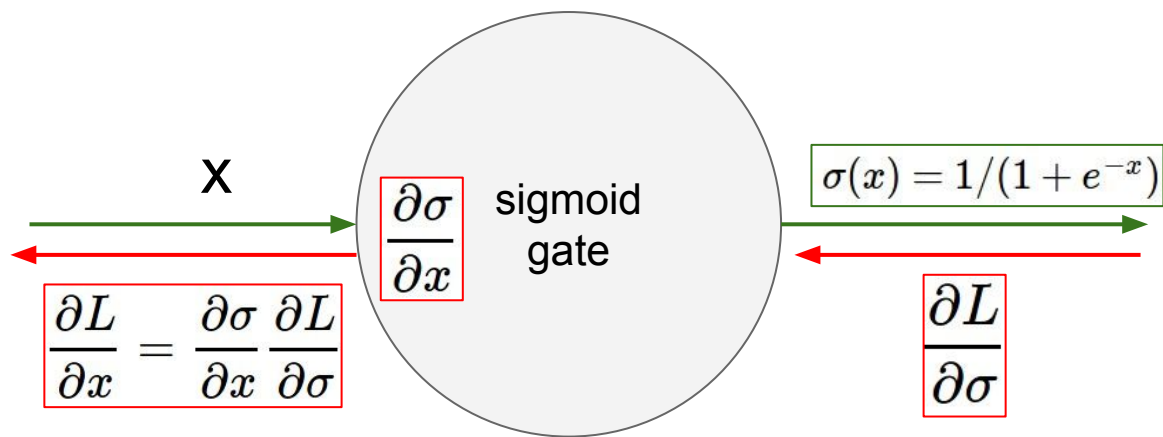


What happens when $x = -10$?

What happens when $x = 0$?

What happens when $x = 10$?

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



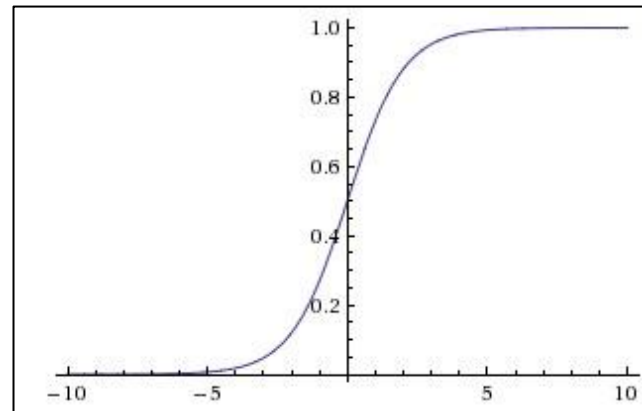
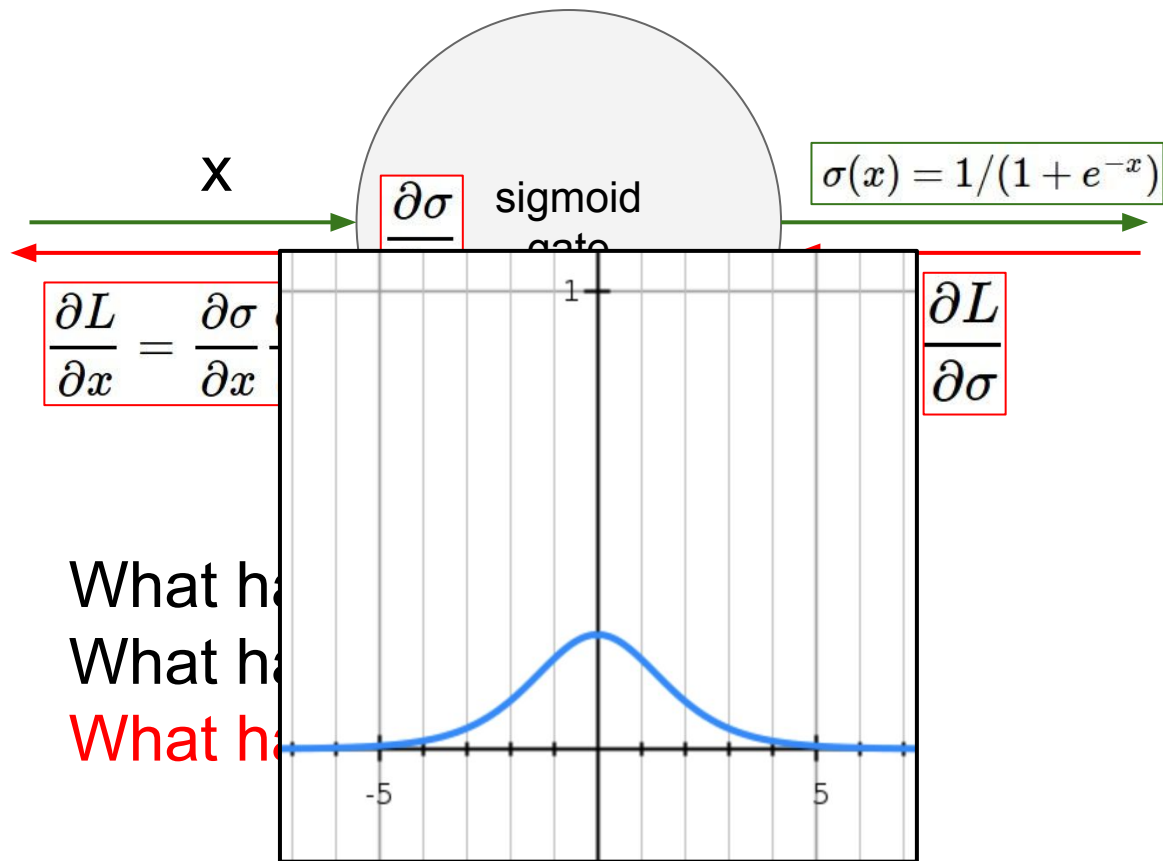
What happens when $x = -10$?

What happens when $x = 0$?

What happens when $x = 10$?

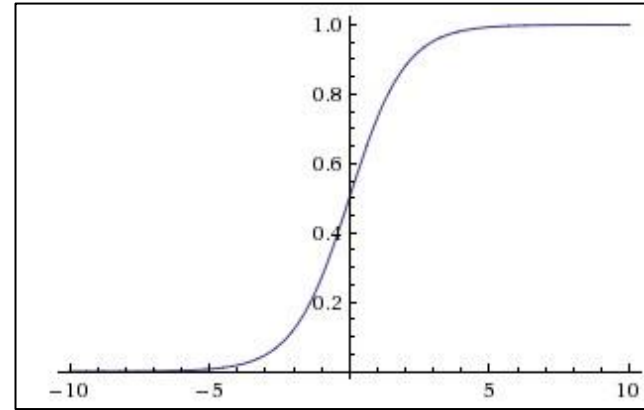
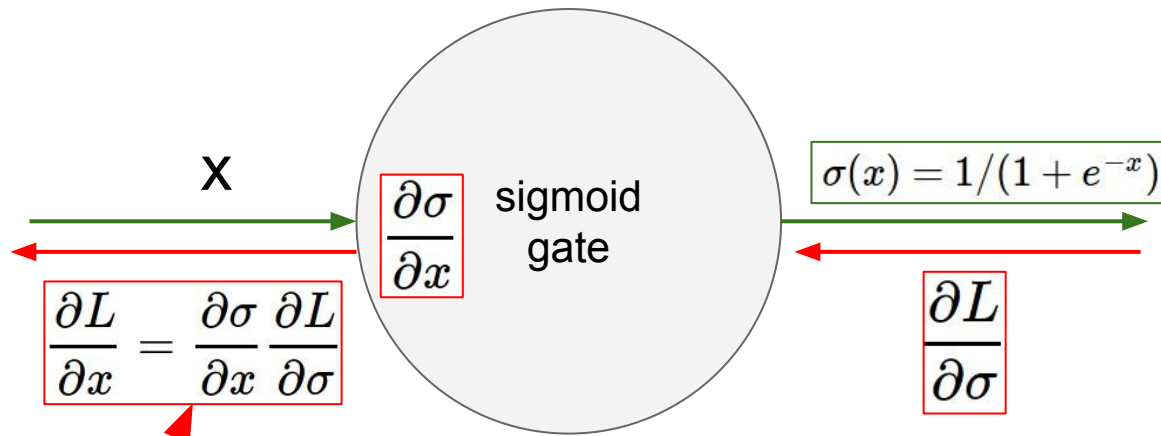
$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x) (1 - \sigma(x))$$

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



What has
 What has
 What has

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



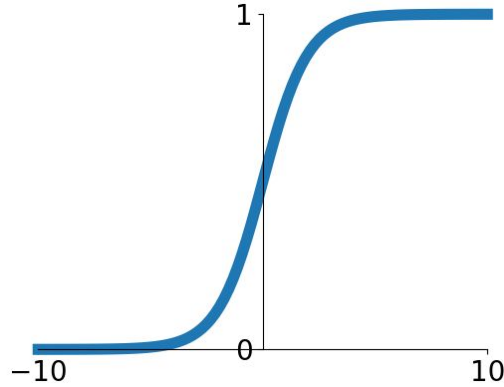
$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x) (1 - \sigma(x))$$

Why is this a problem?

If all the gradients flowing back will be zero and weights will never change

Activation Functions

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



Sigmoid

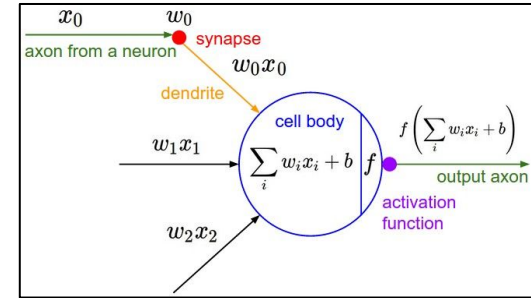
- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

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...

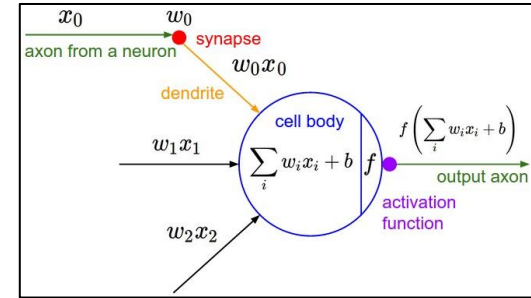
$$f\left(\sum_i w_i x_i + b\right)$$



What can we say about the gradients on \mathbf{w} ?

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

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

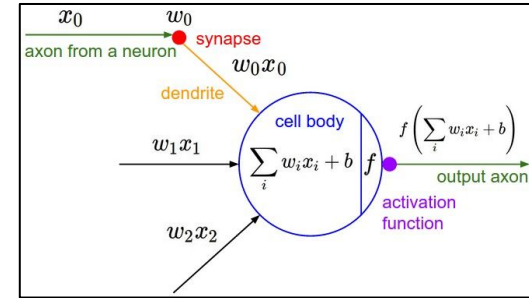


What can we say about the gradients on \mathbf{w} ?

$$\frac{\partial L}{\partial w} = \sigma\left(\sum_i w_i x_i + b\right) \left(1 - \sigma\left(\sum_i w_i x_i + b\right)\right) x \times \textit{upstream_gradient}$$

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

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



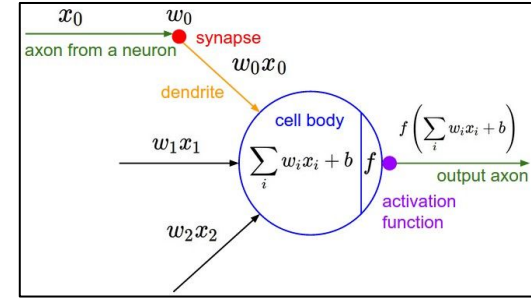
What can we say about the gradients on \mathbf{w} ?

We know that local gradient of sigmoid is always positive

$$\frac{\partial L}{\partial w} = \sigma\left(\sum_i w_i x_i + b\right) \left(1 - \sigma\left(\sum_i w_i x_i + b\right)\right) x \times upstream_gradient$$

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 \mathbf{w} ?

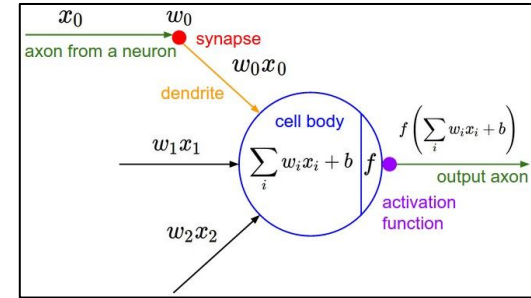
We know that local gradient of sigmoid is always positive

We are assuming x is always positive

$$\frac{\partial L}{\partial w} = \sigma\left(\sum_i w_i x_i + b\right) (1 - \sigma\left(\sum_i w_i x_i + b\right)) x \times \text{upstream_gradient}$$

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 \mathbf{w} ?

We know that local gradient of sigmoid is always positive

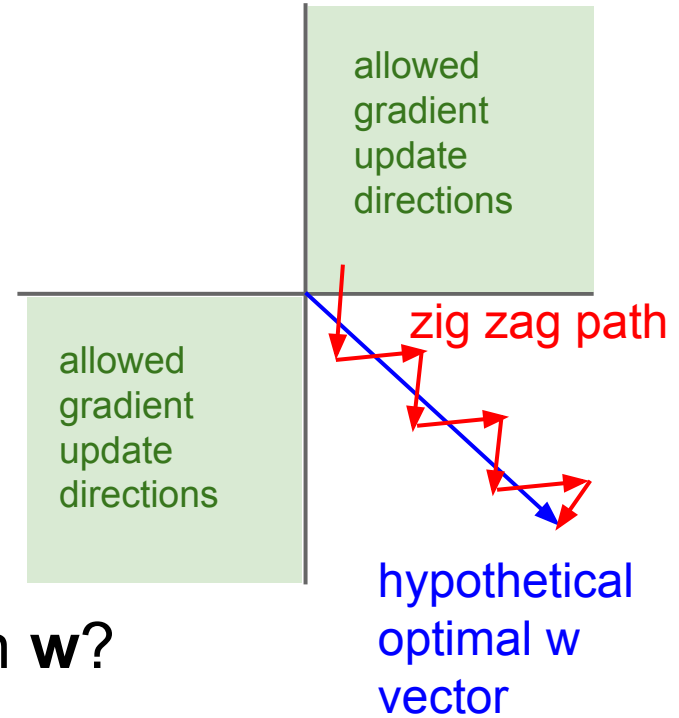
We are assuming x is always positive

So!! Sign of gradient **for all w_i** is the same as the sign of upstream scalar gradient!

$$\frac{\partial L}{\partial w} = \sigma\left(\sum_i w_i x_i + b\right) (1 - \sigma\left(\sum_i w_i x_i + b\right)) x \times \text{upstream_gradient}$$

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

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

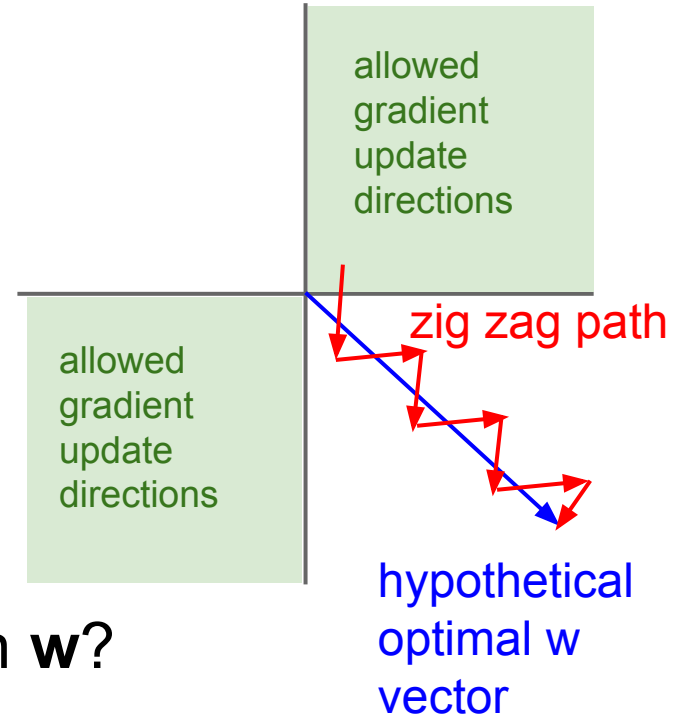


What can we say about the gradients on \mathbf{w} ?

Always all positive or all negative :(

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

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



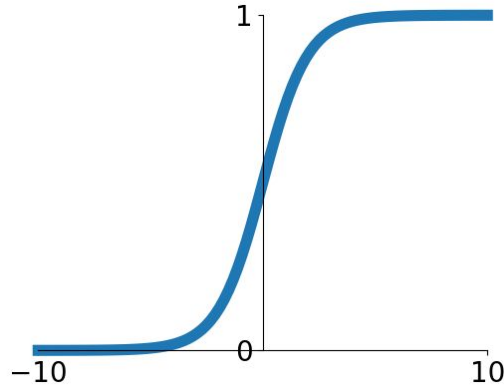
What can we say about the gradients on \mathbf{w} ?

Always all positive or all negative :(

(For a single element! Minibatches help)

Activation Functions

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



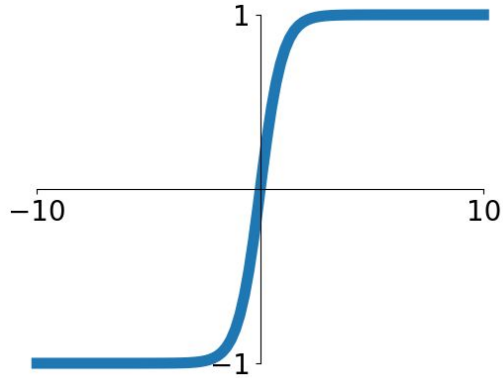
Sigmoid

- 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

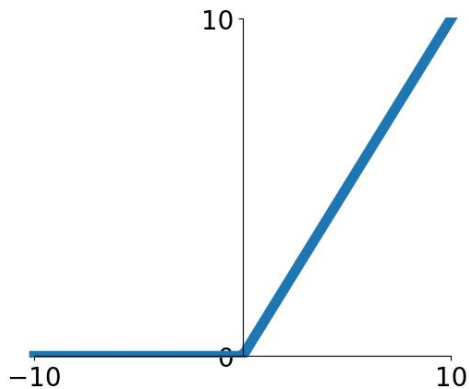


$\tanh(x)$

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

[LeCun et al., 1991]

Activation Functions

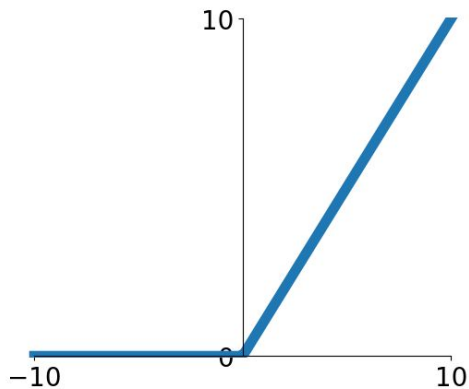


ReLU (Rectified Linear Unit)

- Computes $f(\mathbf{x}) = \max(0, \mathbf{x})$
- 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

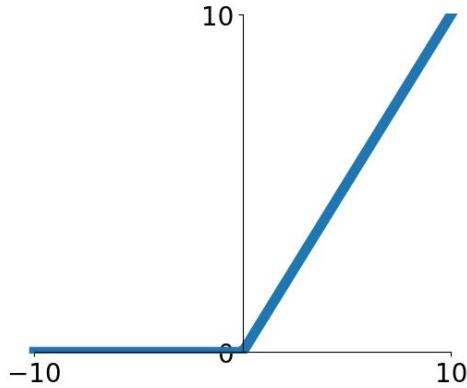


ReLU

(Rectified Linear Unit)

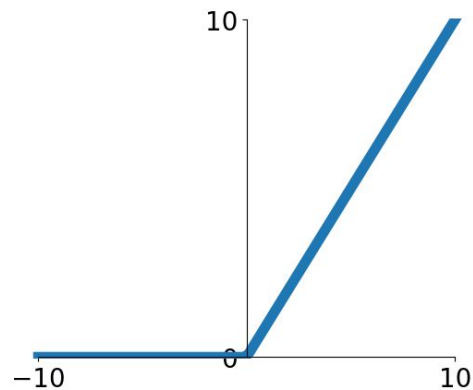
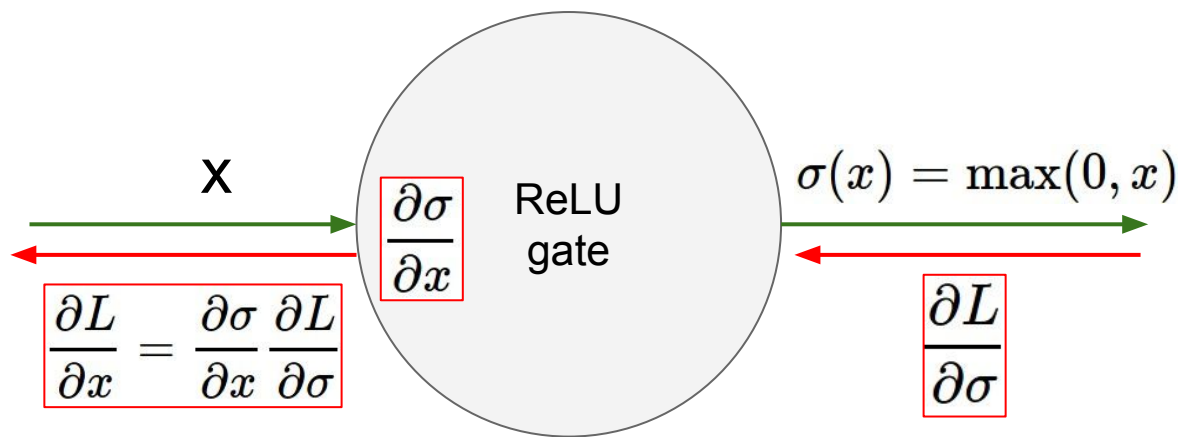
- Computes $f(x) = \max(0, x)$
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- Not zero-centered output

Activation Functions



ReLU (Rectified Linear Unit)

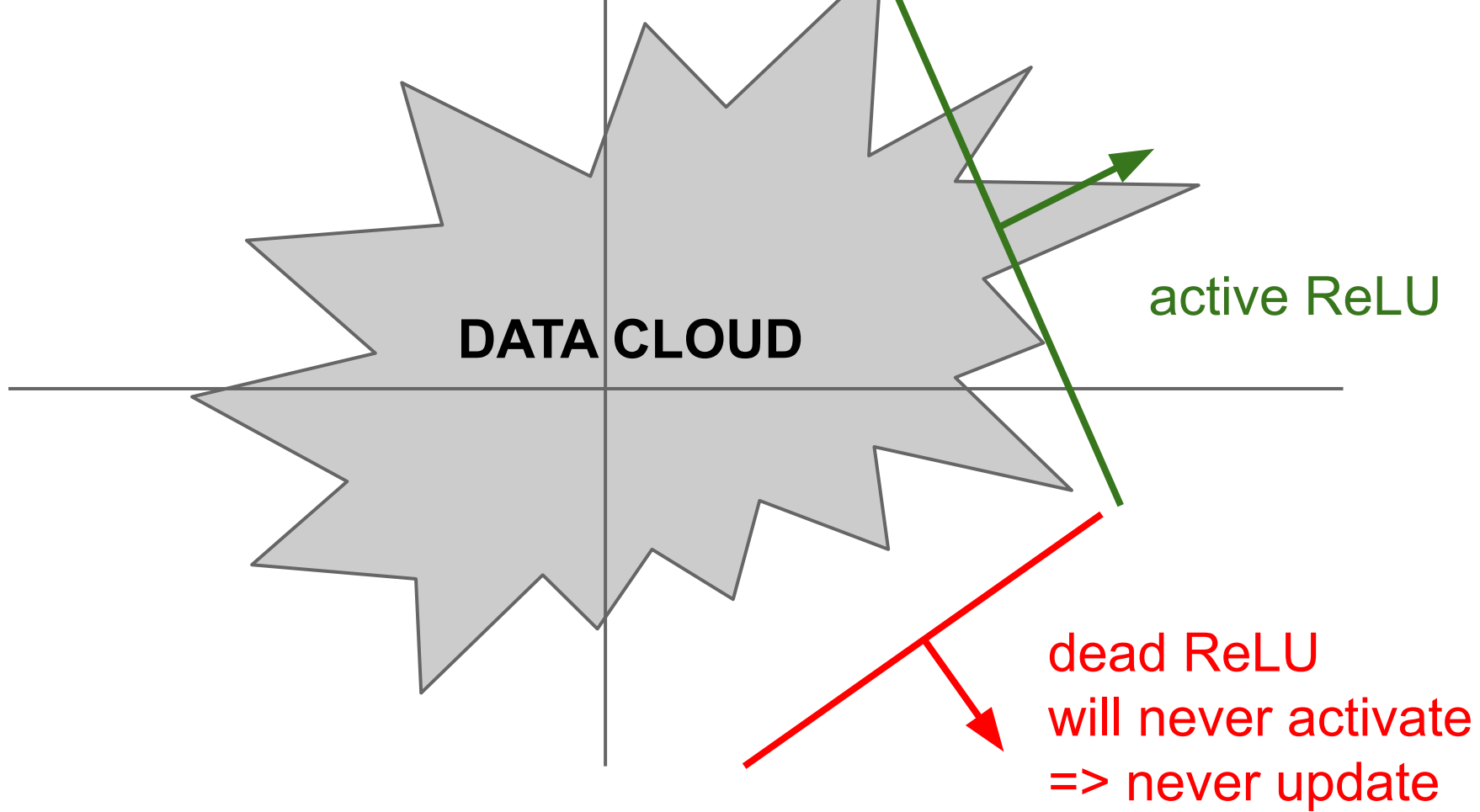
- Computes $f(x) = \max(0, x)$
 - Does not saturate (in +region)
 - Very computationally efficient
 - Converges much faster than sigmoid/tanh in practice (e.g. 6x)
 - Not zero-centered output
 - An annoyance:
- hint: what is the gradient when $x < 0$?

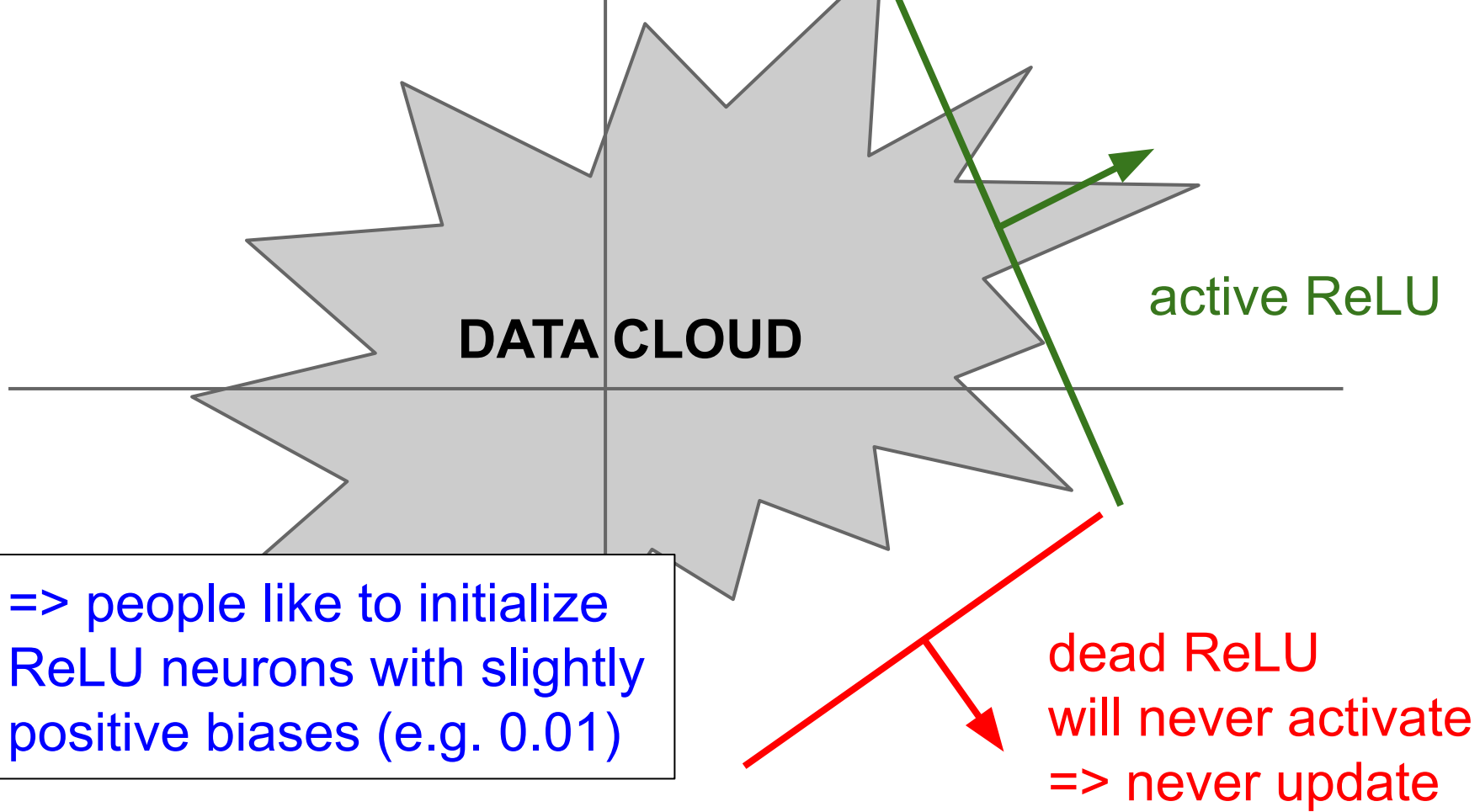


What happens when $x = -10$?

What happens when $x = 0$?

What happens when $x = 10$?



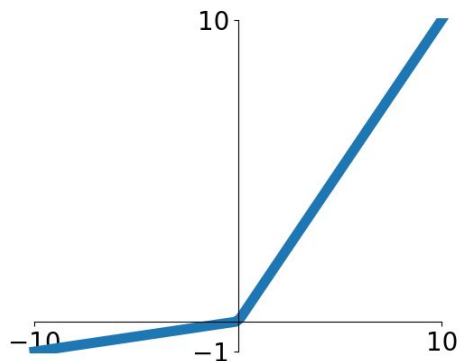


=> people like to initialize ReLU neurons with slightly positive biases (e.g. 0.01)

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”.**

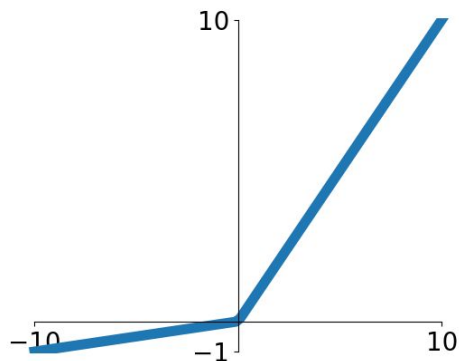
Leaky ReLU

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

Activation Functions

[Mass et al., 2013]

[He et al., 2015]



Leaky ReLU

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

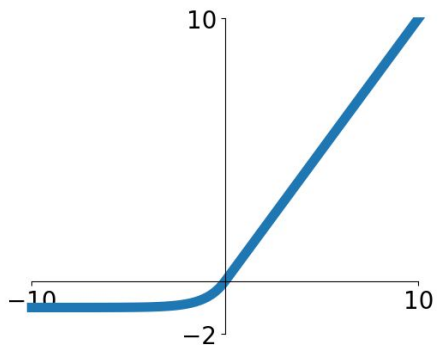
- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- **will not “die”.**

Parametric Rectifier (PReLU)

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

backprop into α
(parameter)

Exponential Linear Units (ELU)

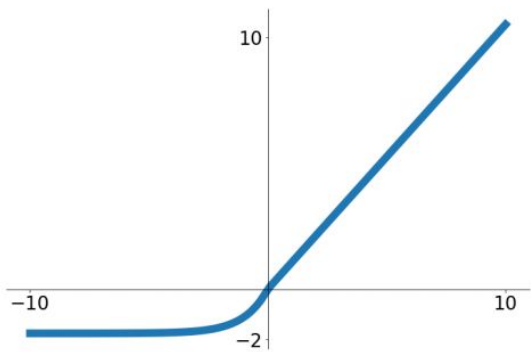


$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha (\exp(x) - 1) & \text{if } x \leq 0 \end{cases}$$

(Alpha default = 1)

- All benefits of ReLU
- Closer to zero mean outputs
- Negative saturation regime compared with Leaky ReLU adds some robustness to noise
- **Computation requires $\exp()$**

Scaled Exponential Linear Units (SELU)



- Scaled version of ELU that works better for deep networks
- “Self-normalizing” property;
- Can train deep SELU networks without BatchNorm
 - (will discuss more later)

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

$\alpha = 1.6733, \lambda = 1.0507$

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/weights :(

Activation Functions

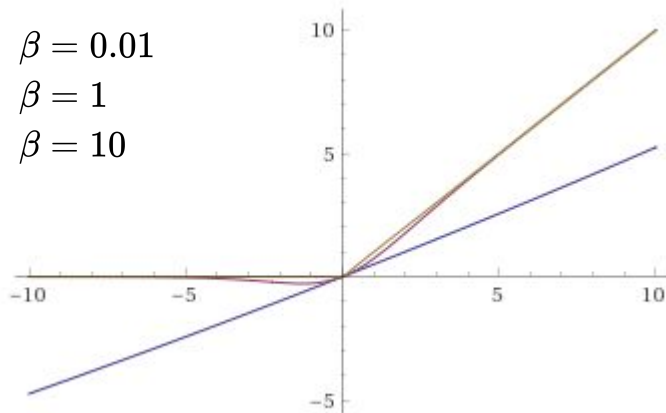
[Ramachandran et al. 2018]

Swish

$$\beta = 0.01$$

$$\beta = 1$$

$$\beta = 10$$

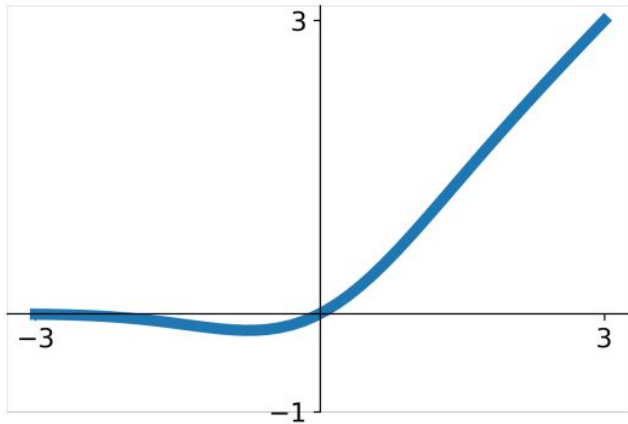


$$f(x) = x\sigma(\beta x)$$

- They trained a neural network to generate and test out different non-linearities.
- Swish outperformed all other options for CIFAR-10 accuracy

Activation Functions

GeLU



$X \sim N(0, 1)$

$$\begin{aligned} \text{gelu}(x) &= xP(X \leq x) = \frac{x}{2}(1 + \text{erf}(x/\sqrt{2})) \\ &\approx x\sigma(1.702x) \end{aligned}$$

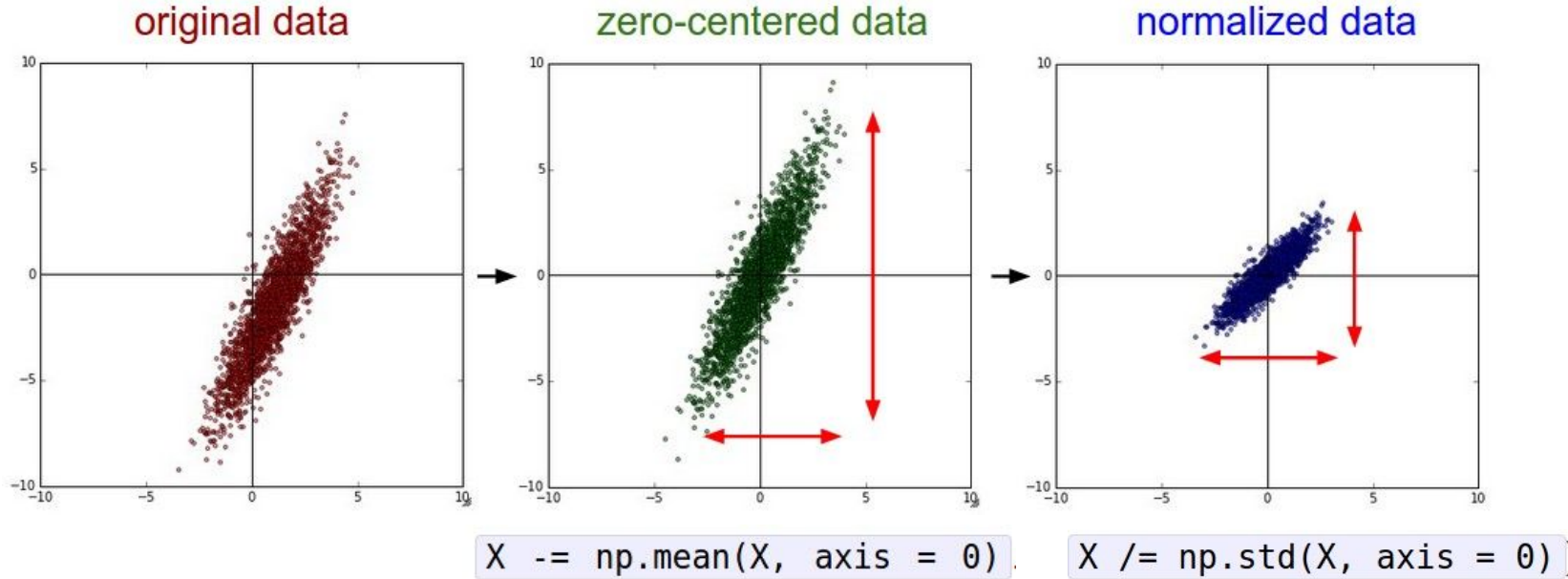
- Idea: Multiply input by 0 or 1 at random; large values more likely to be multiplied by 1, small values more likely to be multiplied by 0 (data-dependent dropout)
- Take expectation over randomness
- Very common in Transformers (BERT, GPT, ViT)

TLDR: In practice:

- Use **ReLU**. Be careful with your learning rates
- Use GeLU is using transformers
- Try out **Leaky ReLU / Maxout / ELU / SELU**
 - To squeeze out some marginal gains
- Don't use **sigmoid** or **tanh**

Data Preprocessing

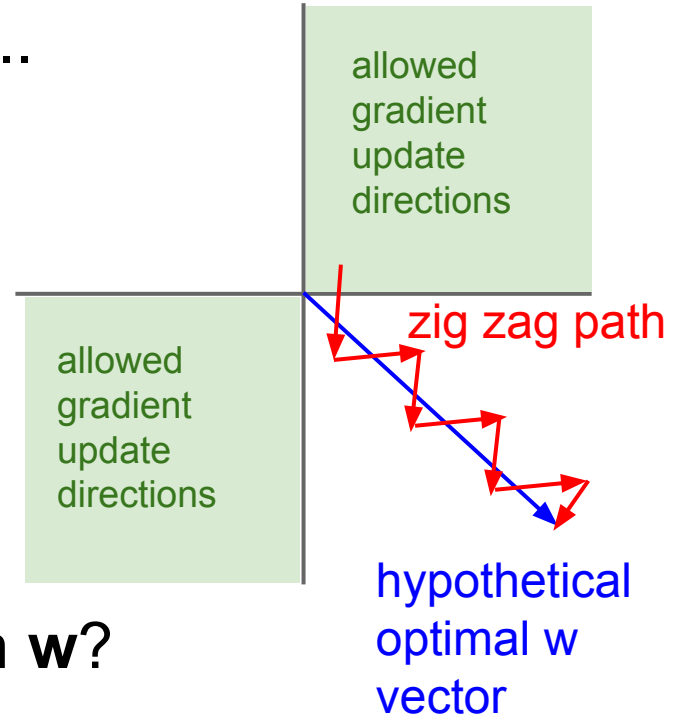
Data Preprocessing



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

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

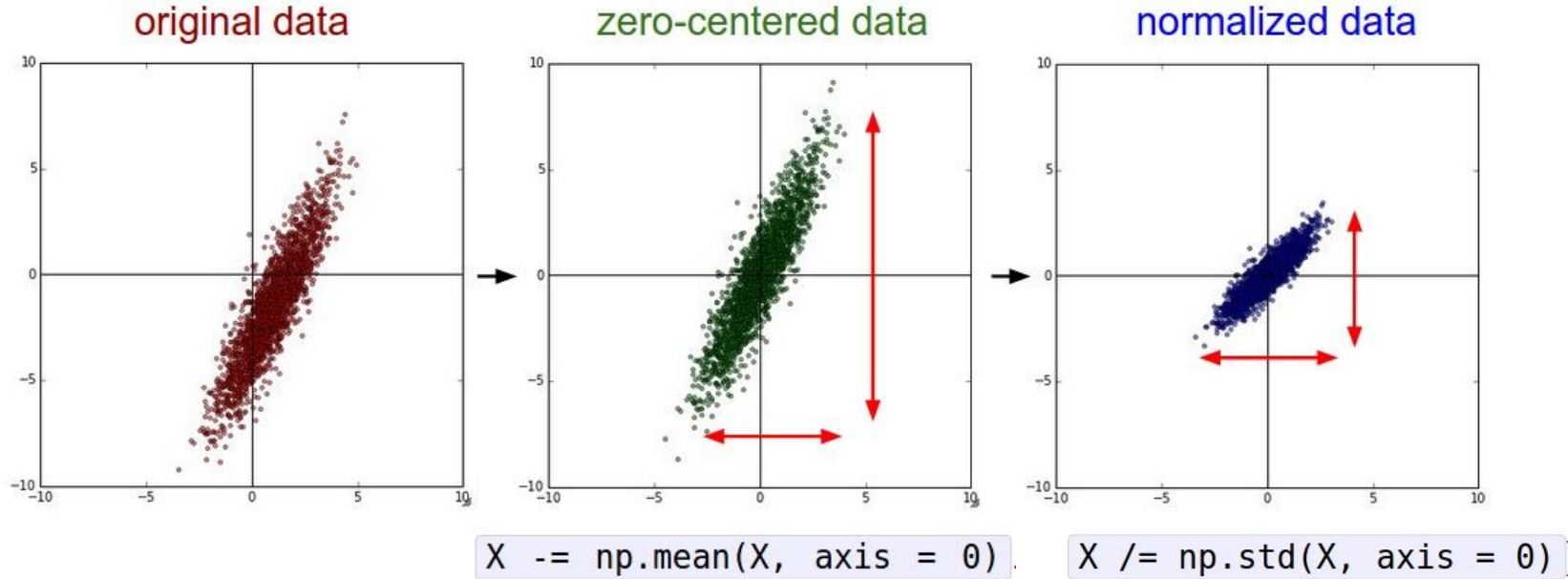
$$f\left(\sum_i w_i x_i + b\right)$$



What can we say about the gradients on \mathbf{w} ?

Always all positive or all negative :(
(this is also why you want zero-mean data!)

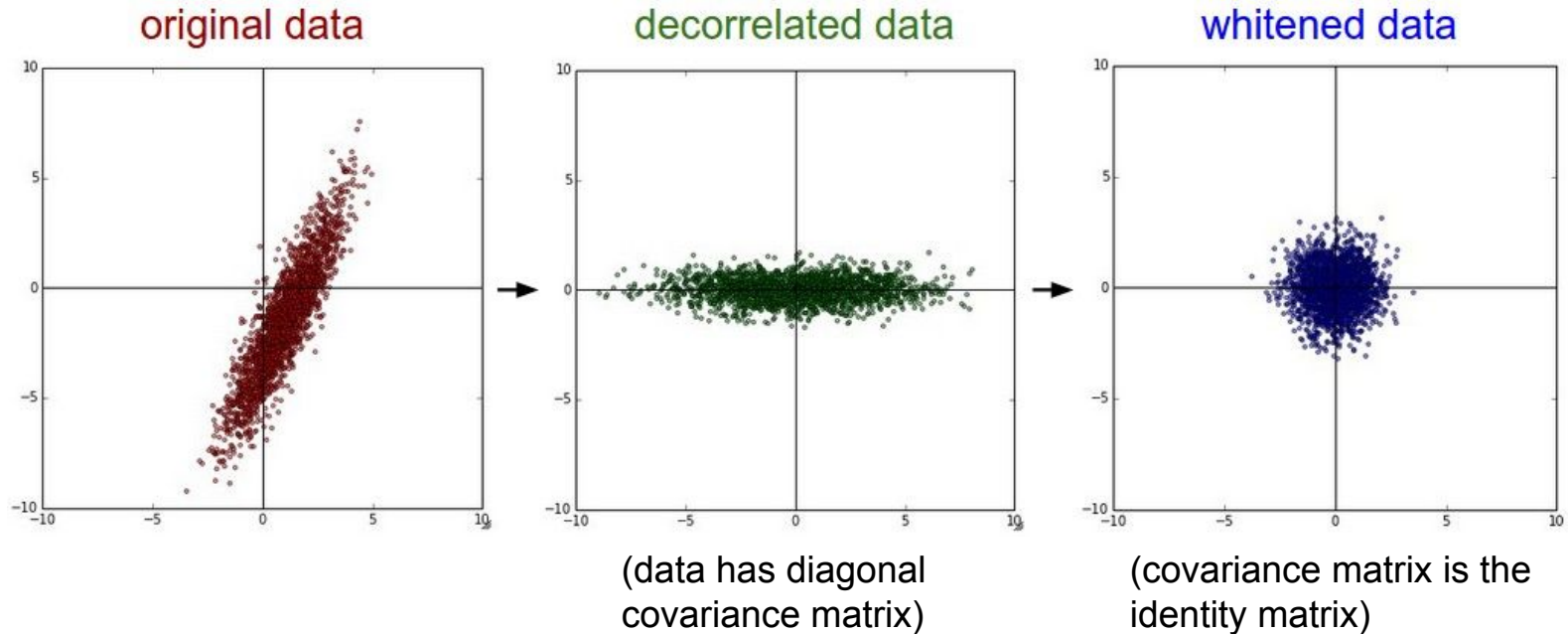
Data Preprocessing



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

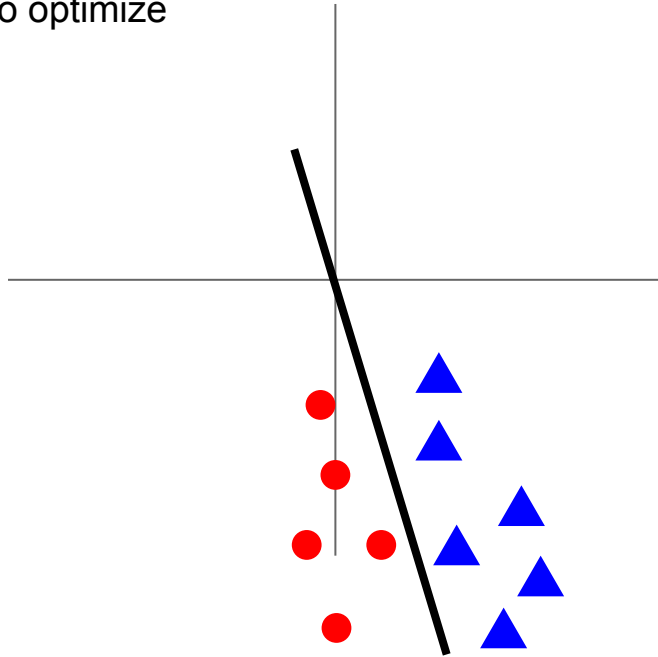
Data Preprocessing

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

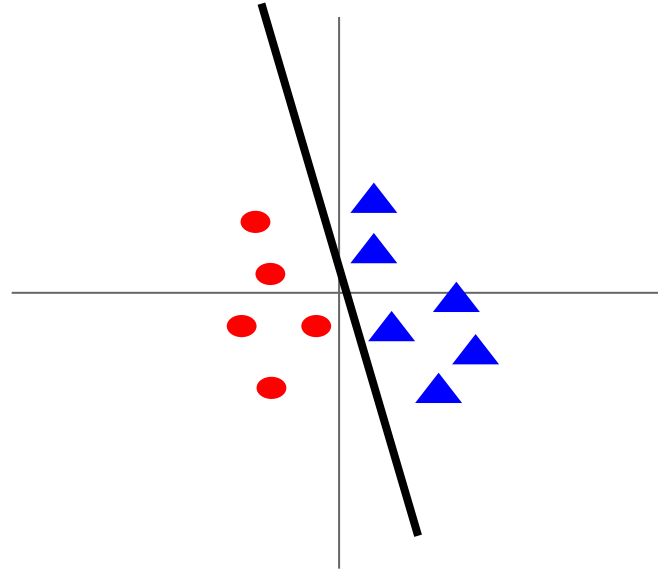


Data Preprocessing

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

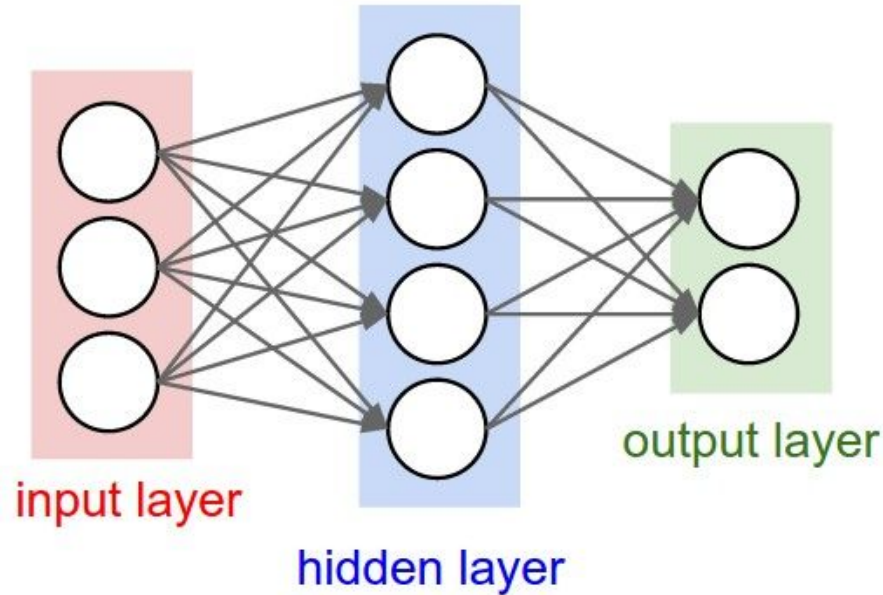
Divide by per-channel std (e.g. ResNet)

(mean along each channel = 3 numbers)

Not common
to do PCA or
whitening

Weight Initialization

- Q: what happens when $W=\text{constant}$ 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      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)
```

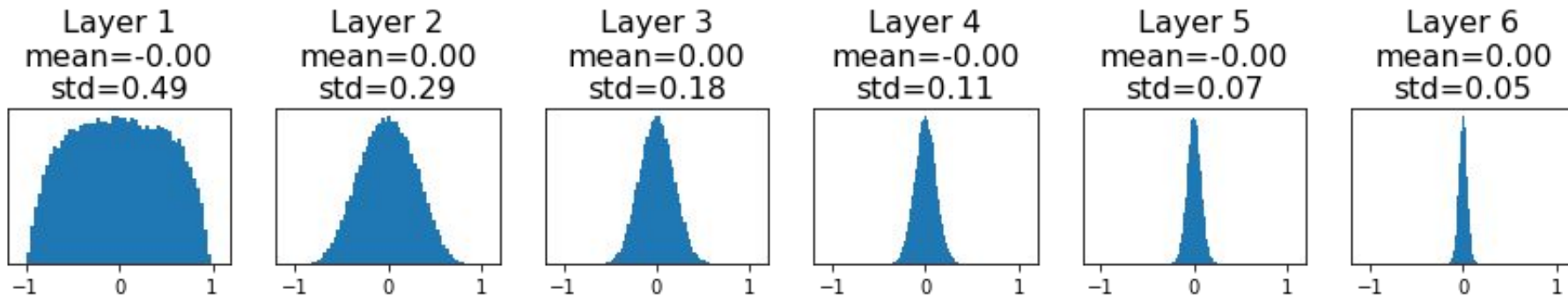
What will happen to the activations for the last layer?

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

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



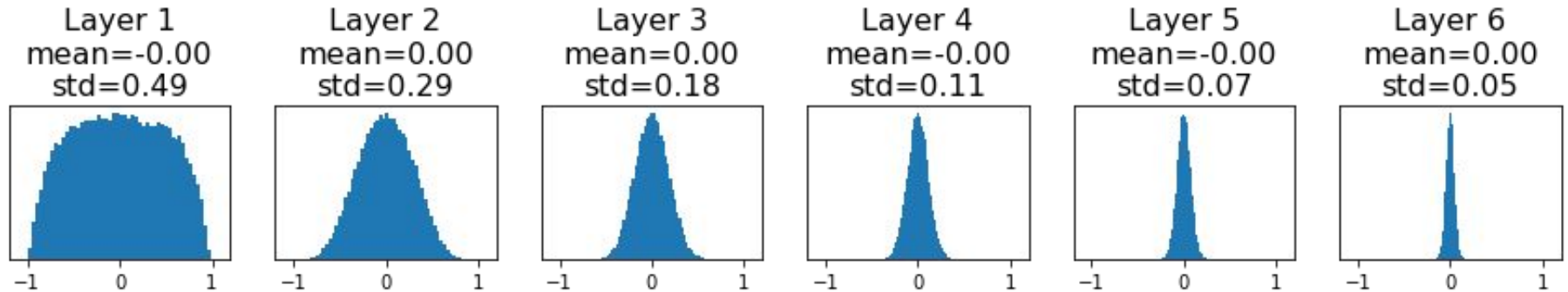
Weight Initialization: Activation statistics

```
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: All zero, no learning = (



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)
```

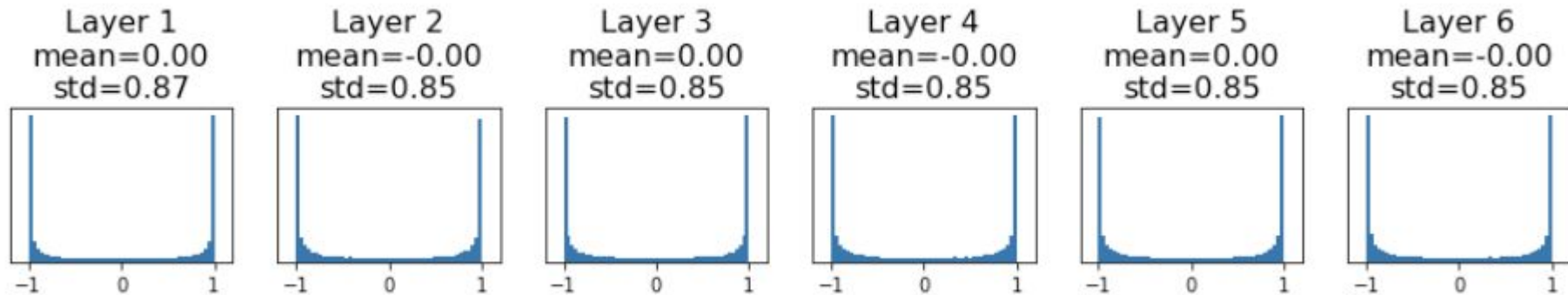
What will happen to the activations for the last layer?

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?



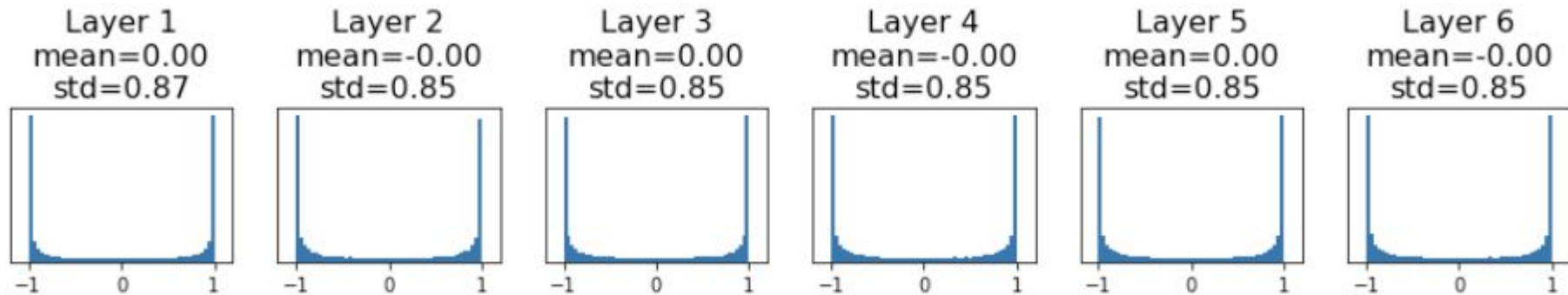
Weight Initialization: Activation statistics

```
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?

A: Local gradients all zero, no learning =(



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)
```

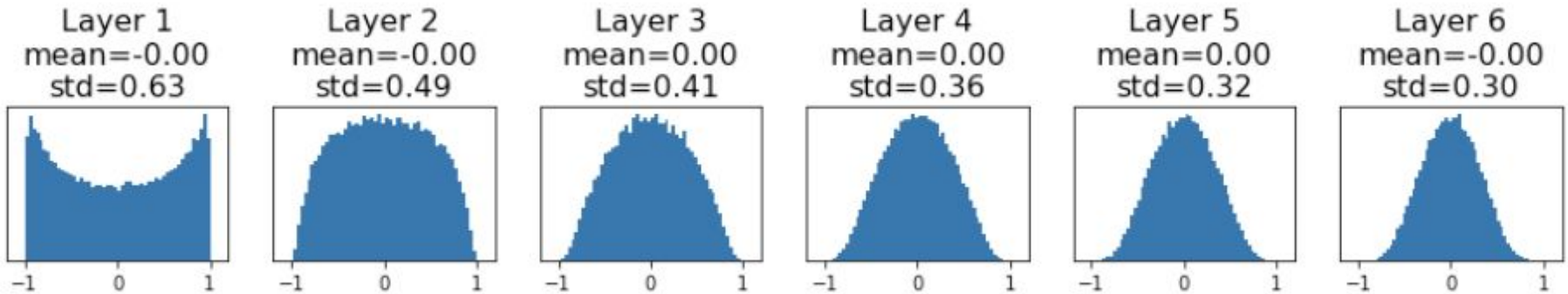
Glorot and Bengio, “Understanding the difficulty of training deep feedforward neural networks”, AISTAT 2010

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

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



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

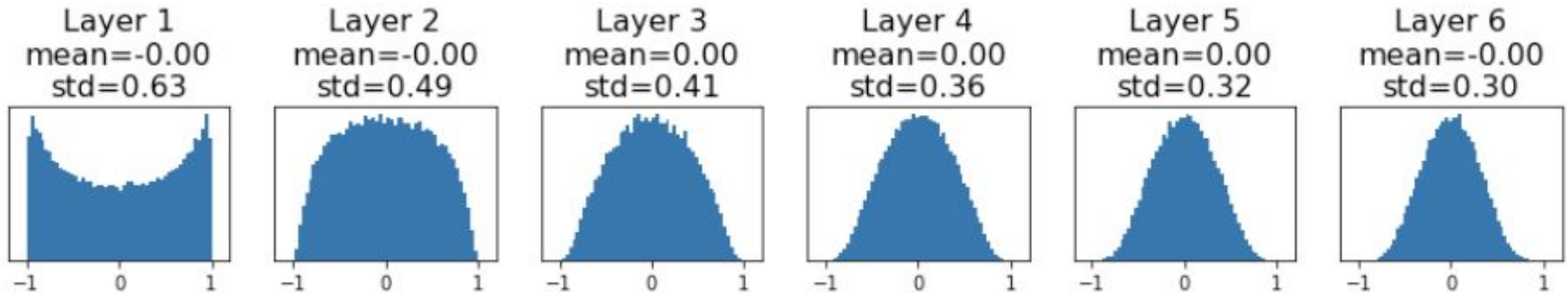
Weight Initialization: “Xavier” Initialization

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

“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

Weight Initialization: “Xavier” Initialization

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

“Xavier” initialization:
std = 1/sqrt(Din)

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

For conv layers, Din is filter_size² * input_channels

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

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

“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: $\text{Var}(y) = \text{Var}(x_i)$

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

[Assume all x_i, w_i are iid]

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]

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

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

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

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

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

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

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

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

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

Weight Initialization: What about ReLU?

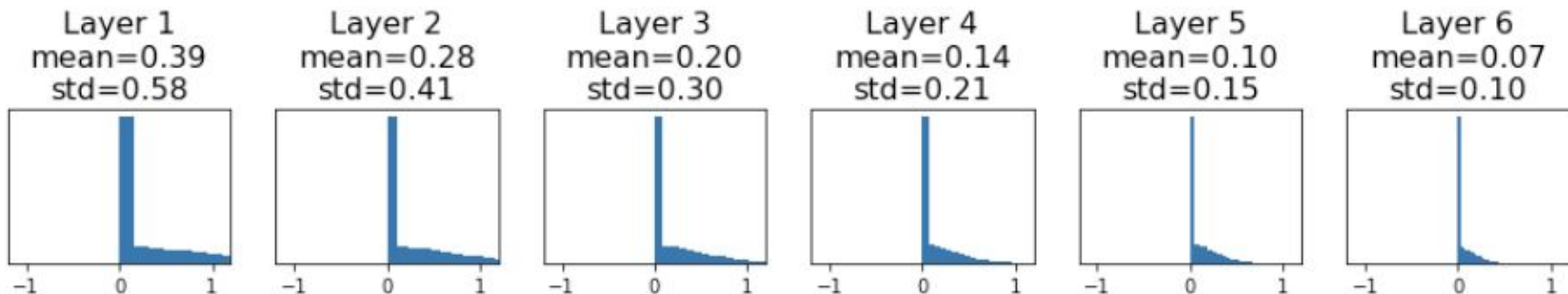
```
dims = [4096] * 7      Change from tanh to ReLU
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

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

Activations collapse to zero again, no learning =(

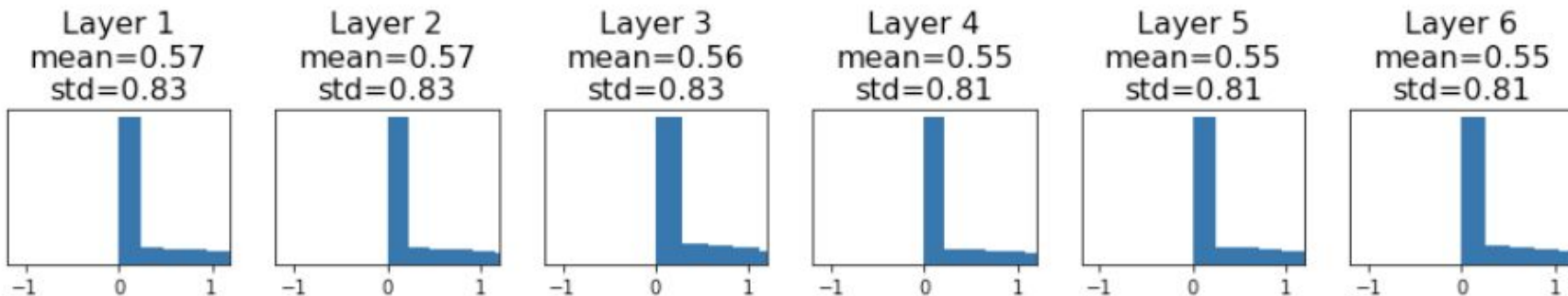


Weight Initialization: Kaiming / MSRA Initialization

```
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)
```

ReLU correction: $\text{std} = \sqrt{2 / \text{Din}}$

“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

[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:

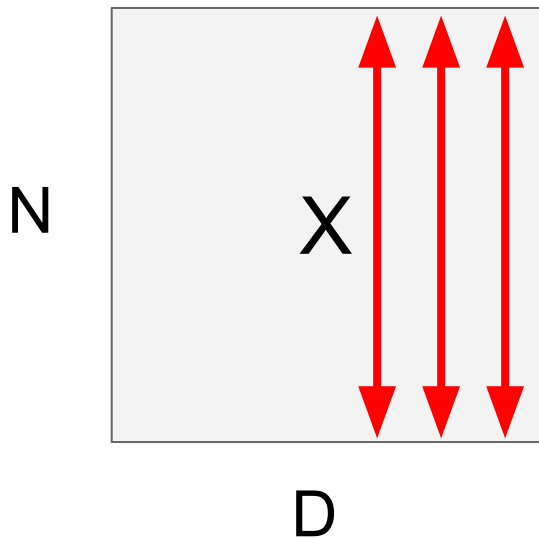
$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

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

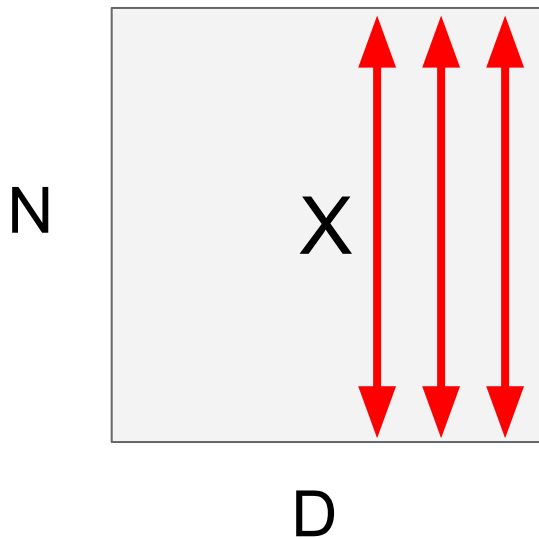
$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

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

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

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

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

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

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!

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

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

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

Batch Normalization: Test-Time

Input: $x : N \times D$

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

Per-channel mean, shape is D

Learnable scale and shift parameters:

$$\gamma, \beta : D$$

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

Per-channel var, shape is D

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

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

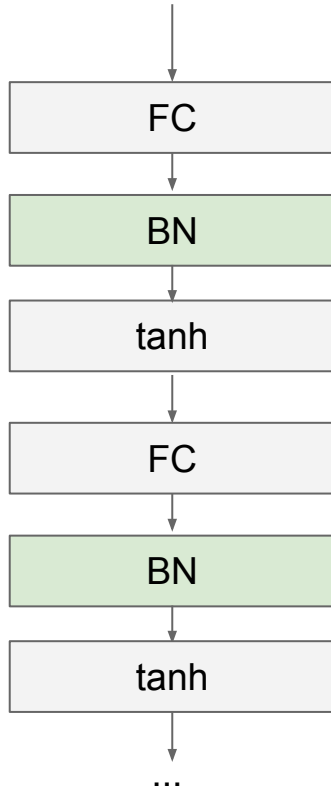
Normalized x, Shape is N x D

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

Output, Shape is N x D

Batch Normalization

[Ioffe and Szegedy, 2015]

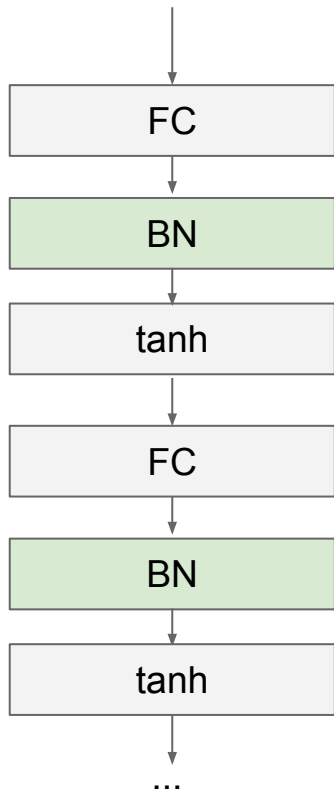


Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

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

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!**

Batch Normalization for ConvNets

Batch Normalization for
fully-connected networks

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

Normalize 

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

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

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

Batch Normalization for
convolutional networks
(Spatial Batchnorm, BatchNorm2D)

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

Normalize   

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

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

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

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!

Can be used in recurrent networks

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

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

Instance Normalization

Batch Normalization for convolutional networks

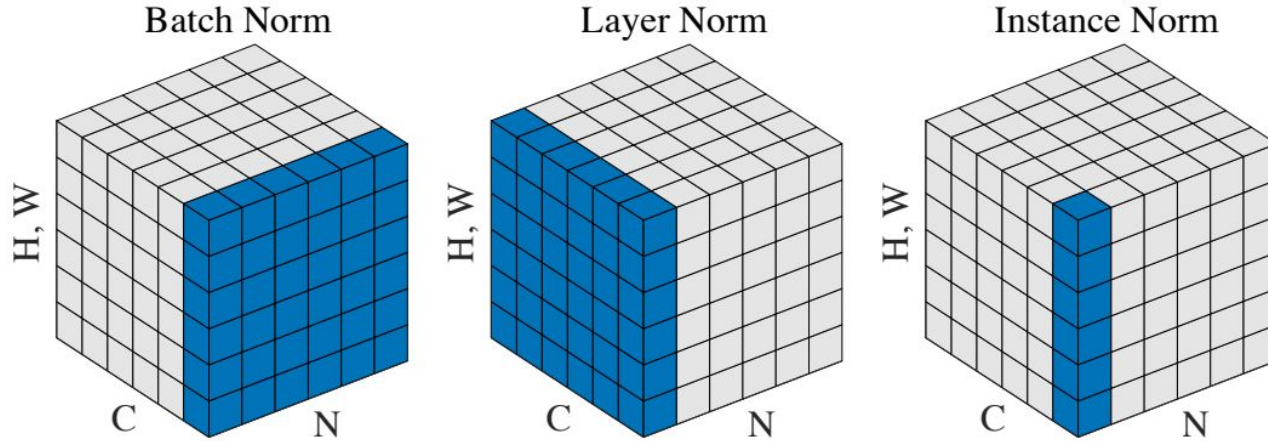
$$\begin{array}{l} \mathbf{x} : \mathbf{N} \times \mathbf{C} \times \mathbf{H} \times \mathbf{W} \\ \text{Normalize} \quad \downarrow \quad \downarrow \quad \downarrow \\ \boldsymbol{\mu}, \boldsymbol{\sigma} : \mathbf{1} \times \mathbf{C} \times \mathbf{1} \times \mathbf{1} \\ \boldsymbol{\gamma}, \boldsymbol{\beta} : \mathbf{1} \times \mathbf{C} \times \mathbf{1} \times \mathbf{1} \\ \mathbf{y} = \boldsymbol{\gamma} (\mathbf{x} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \boldsymbol{\beta} \end{array}$$

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

$$\begin{array}{l} \mathbf{x} : \mathbf{N} \times \mathbf{C} \times \mathbf{H} \times \mathbf{W} \\ \text{Normalize} \quad \quad \quad \downarrow \quad \downarrow \\ \boldsymbol{\mu}, \boldsymbol{\sigma} : \mathbf{N} \times \mathbf{C} \times \mathbf{1} \times \mathbf{1} \\ \boldsymbol{\gamma}, \boldsymbol{\beta} : \mathbf{1} \times \mathbf{C} \times \mathbf{1} \times \mathbf{1} \\ \mathbf{y} = \boldsymbol{\gamma} (\mathbf{x} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \boldsymbol{\beta} \end{array}$$

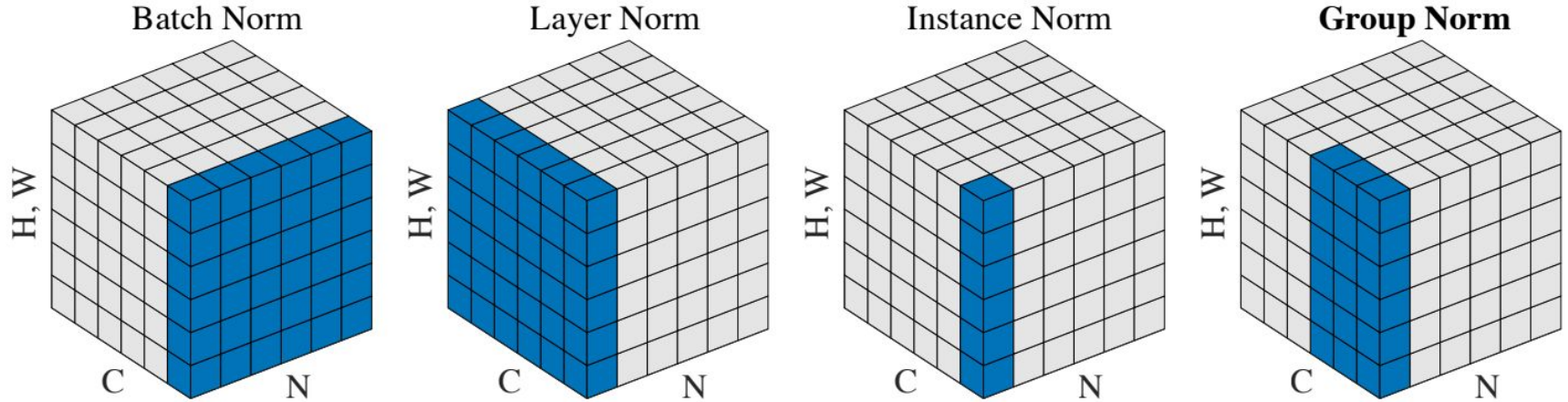
Ulyanov et al, Improved Texture Networks: Maximizing Quality and Diversity in Feed-forward Stylization and Texture Synthesis, CVPR 2017

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 or GeLU)
- Data Preprocessing (images: subtract mean)
- Weight Initialization (use Xavier/He init)
- Batch Normalization (use this!)

Next time:

Training Neural Networks, Part 2

- Parameter update schemes
- Learning rate schedules
- Gradient checking
- Regularization (Dropout etc.)
- Babysitting learning
- Evaluation (Ensembles etc.)
- Hyperparameter Optimization