9. Training Multilayer Perceptrons

Dongwoo Kim

POSTECH

Machine Learning Laboratory POSTECH

*Slides courtesy of CS231 Stanford

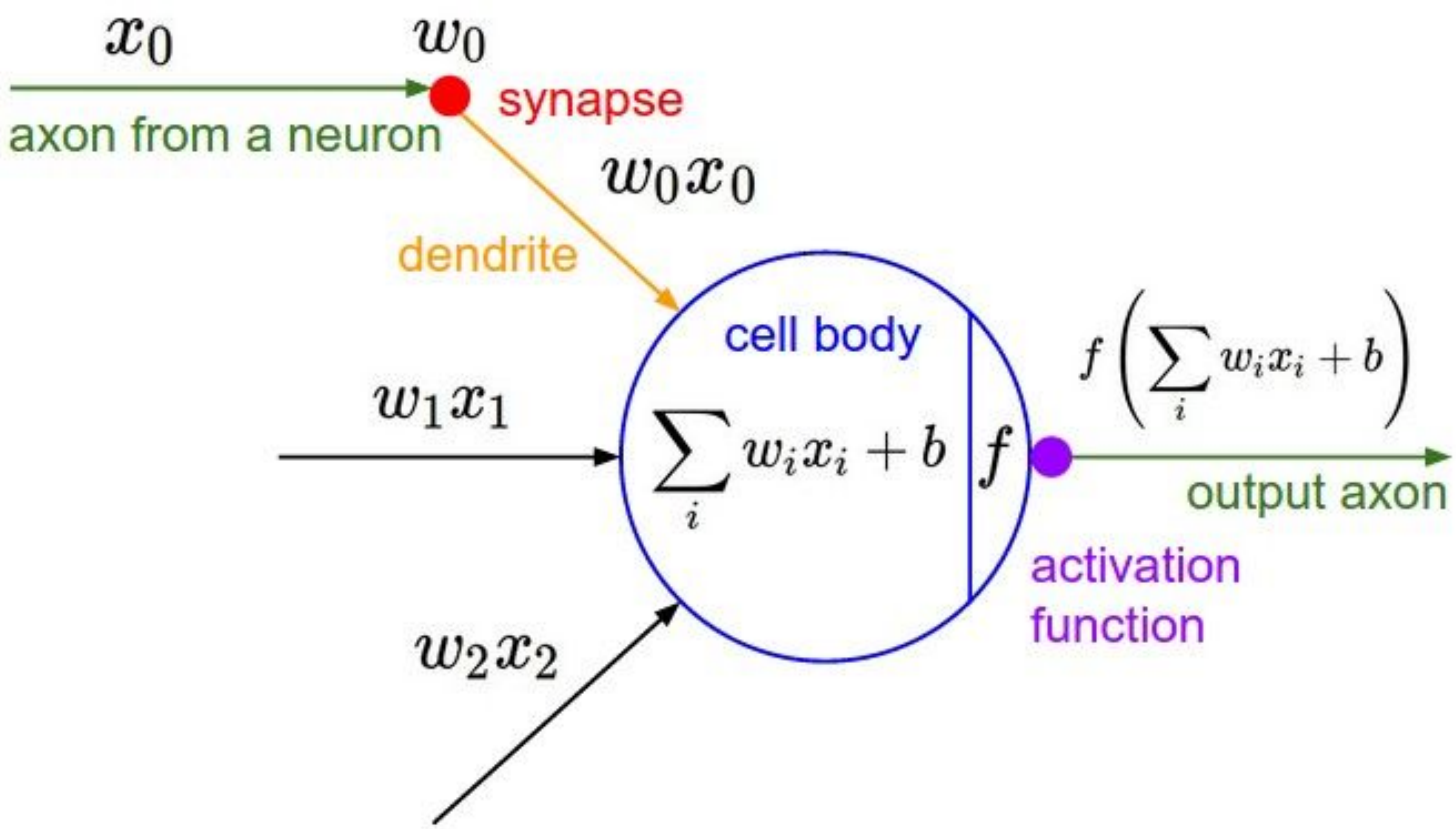
Recap

- Last time, we learned
  - Neural networks (Multi-layer perceptrons, feed-forward neural networks)
  - Back-propagation - to obtain gradient
  - https://youtu.be/tIeHLnjs5U8?t=36
- Today, we will learn several techniques to improve the performance of neural networks.

Outline

- **Activation Functions**
- Weight Initialization
- Batch Normalization

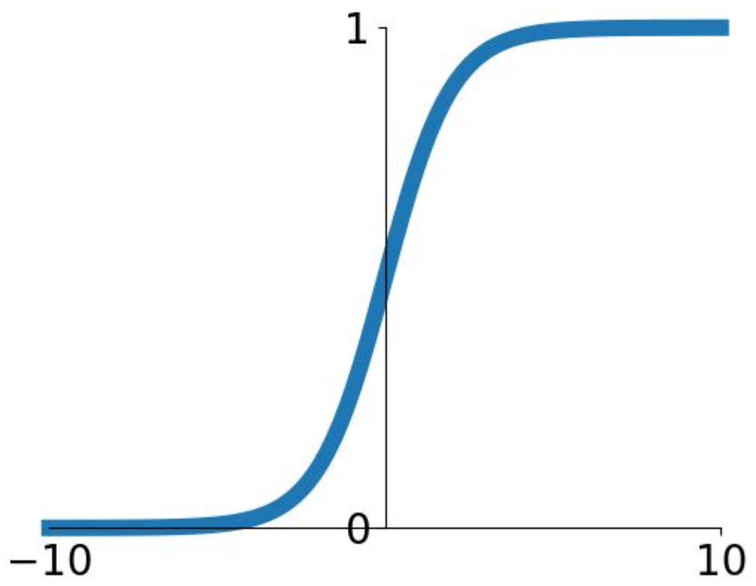
Activation Functions



Activation Functions

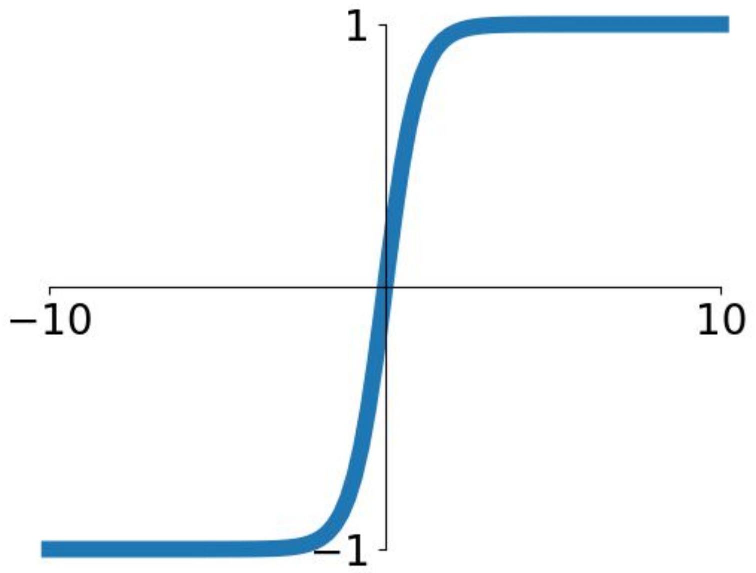
Sigmoid

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



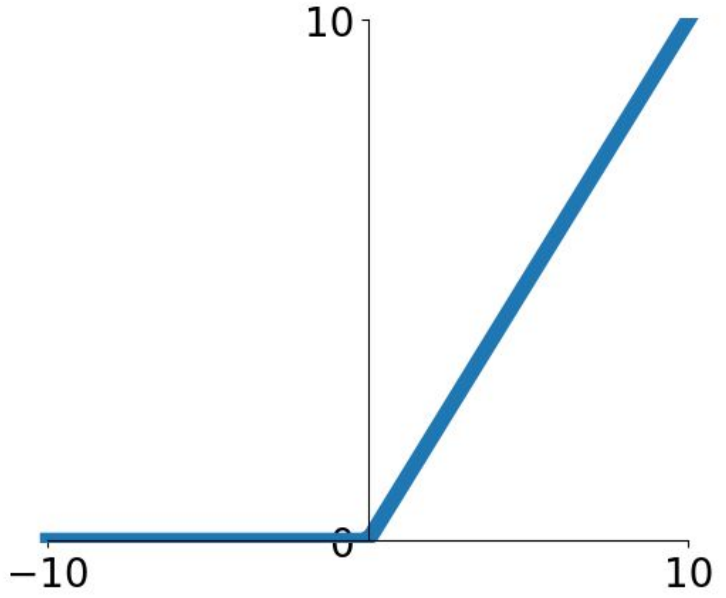
tanh

$\tanh(x)$



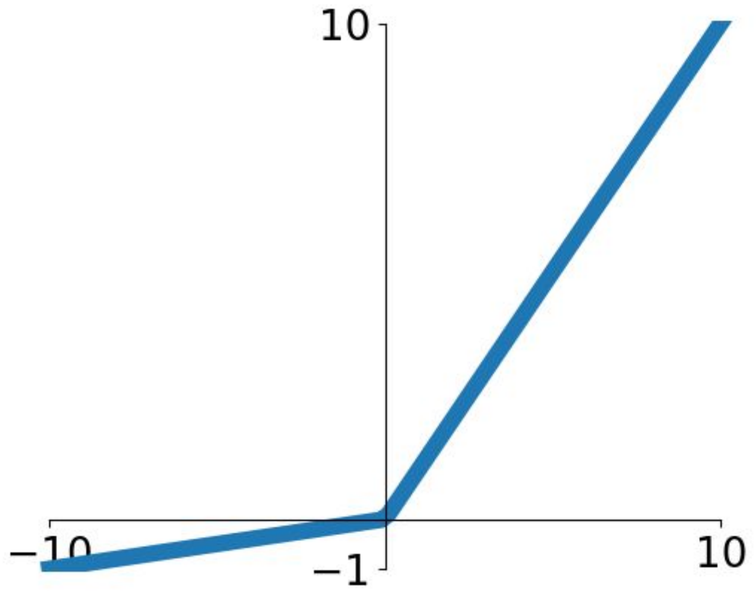
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

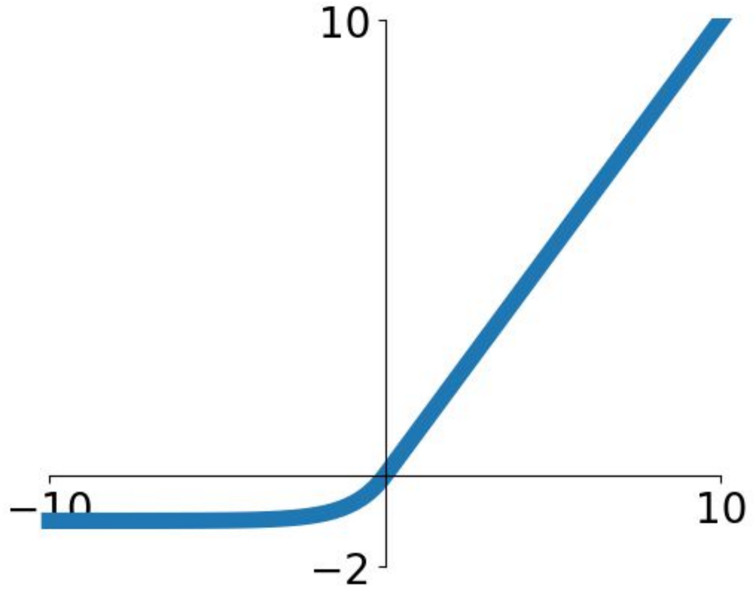


Maxout

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

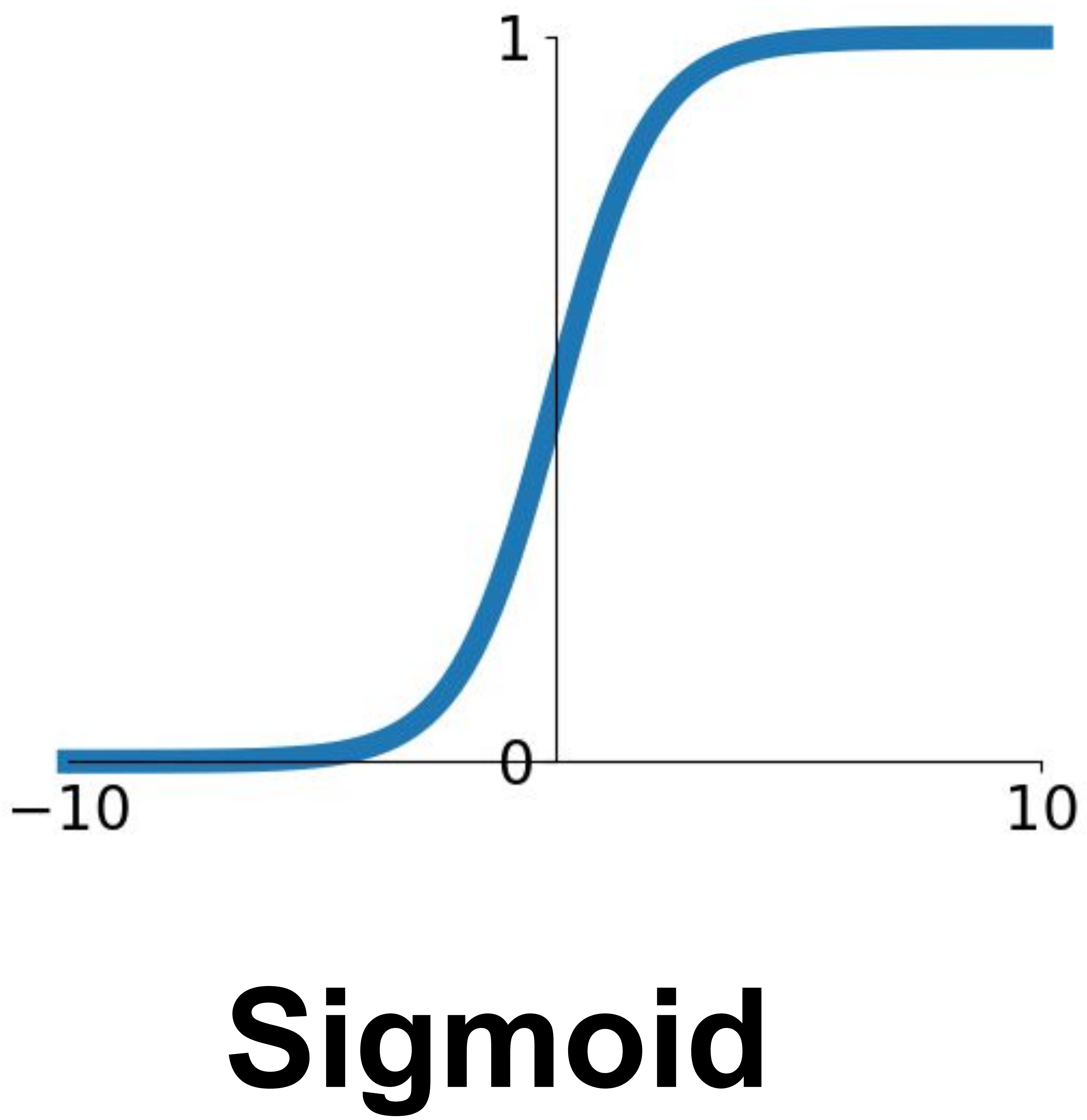
ELU

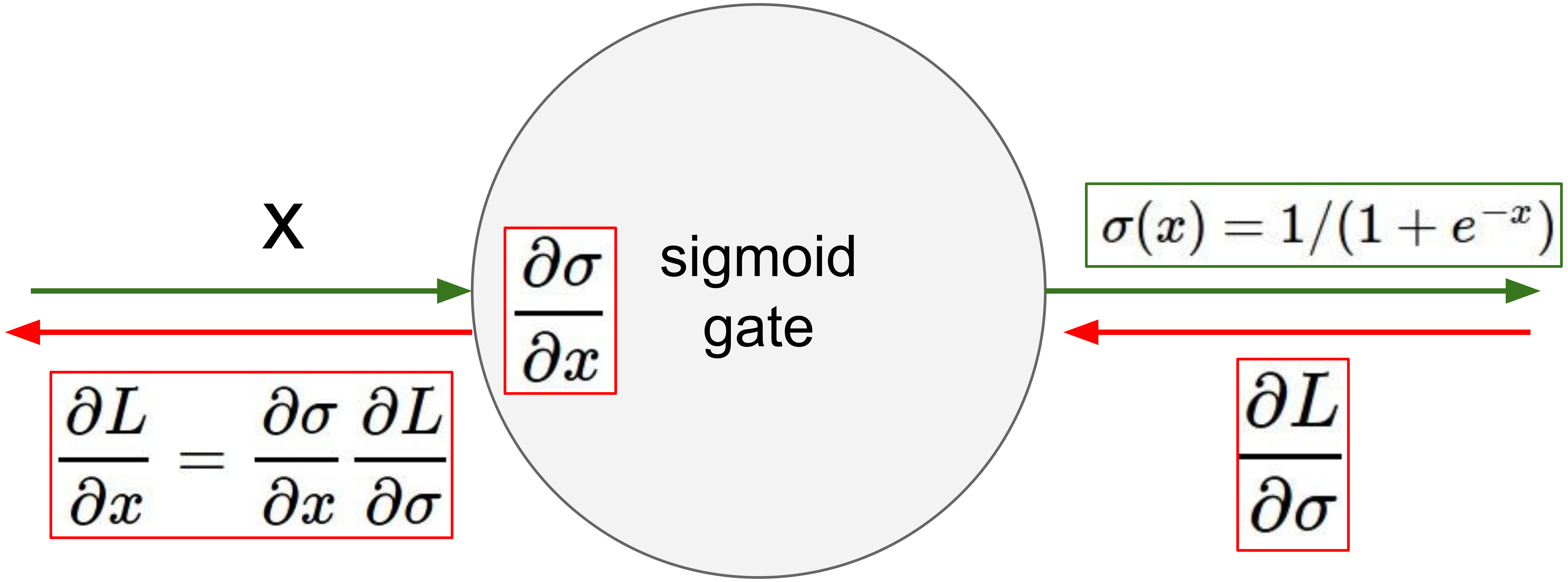
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

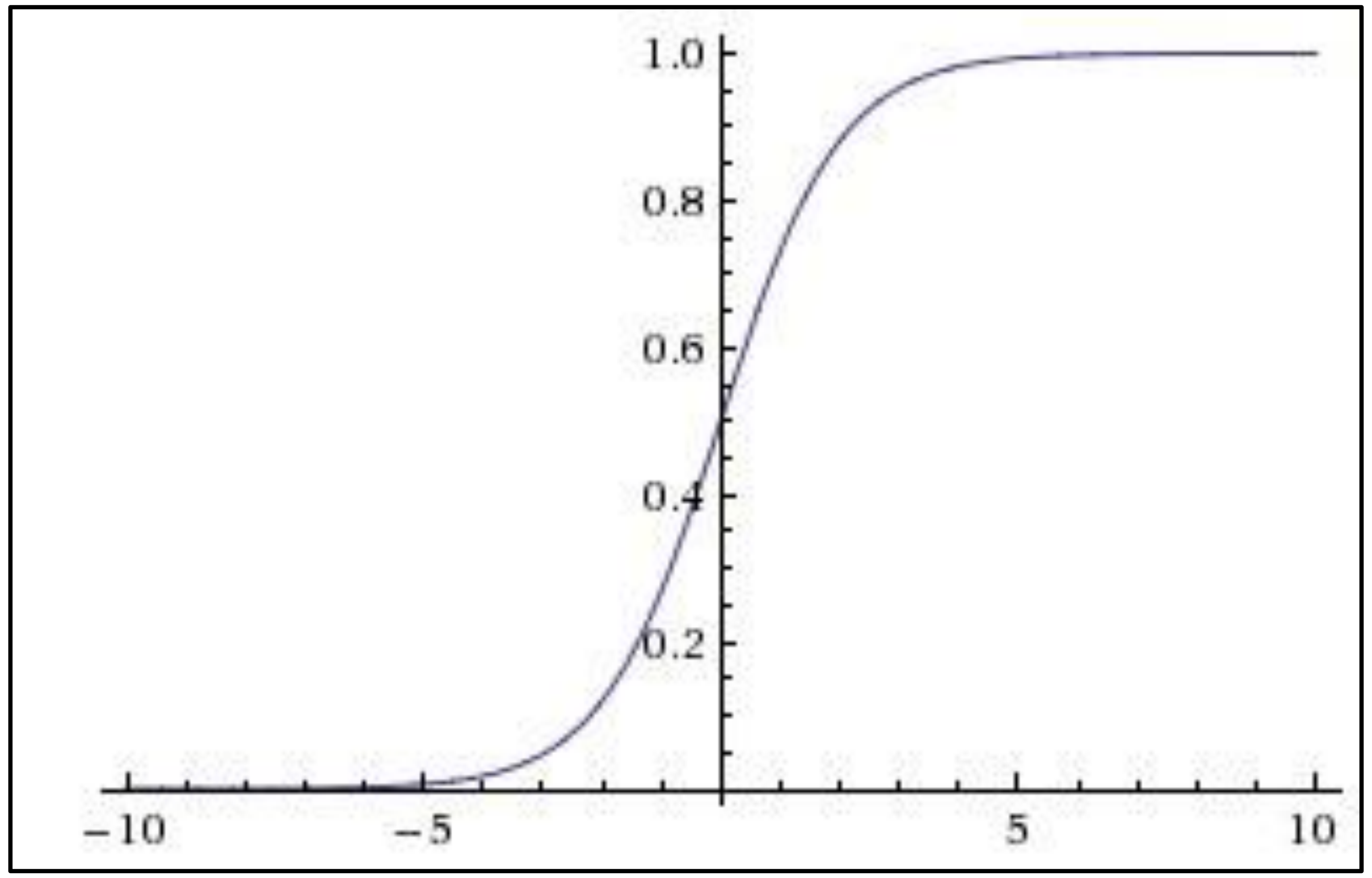


- $\sigma(x) = 1/(1 + e^{-x})$
- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron
- 3 problems:

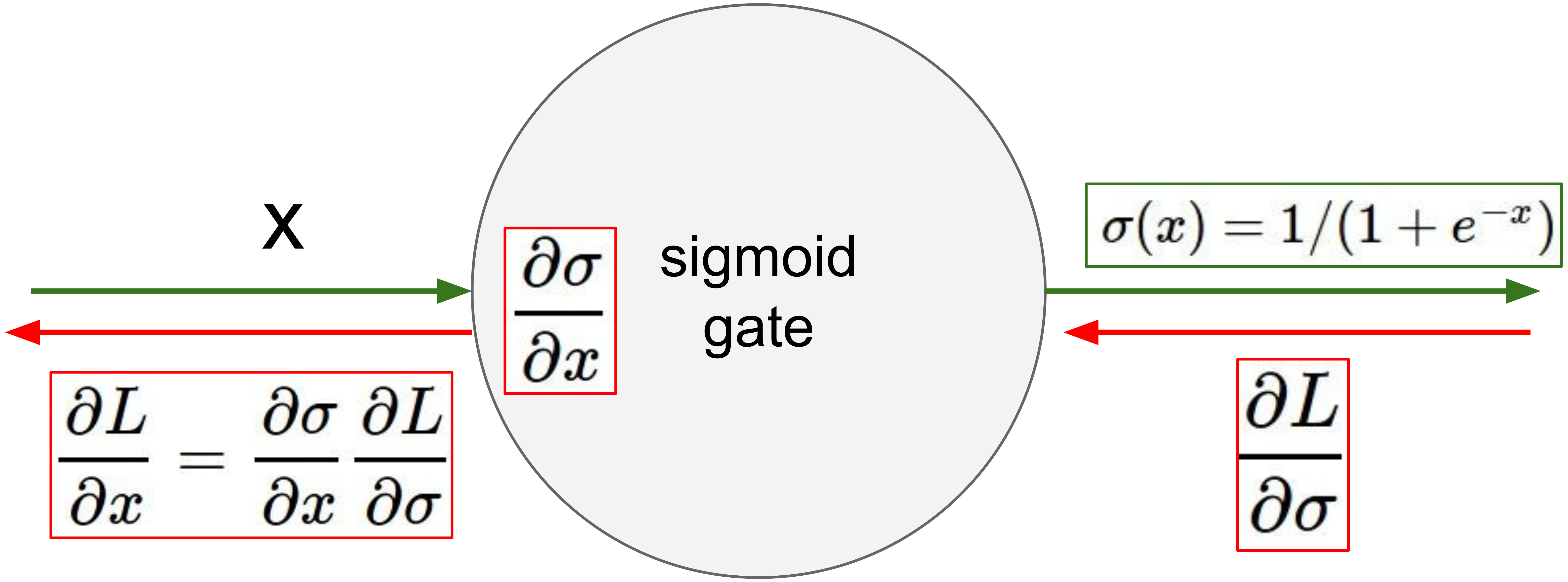
1. Saturated neurons "kill" the gradients

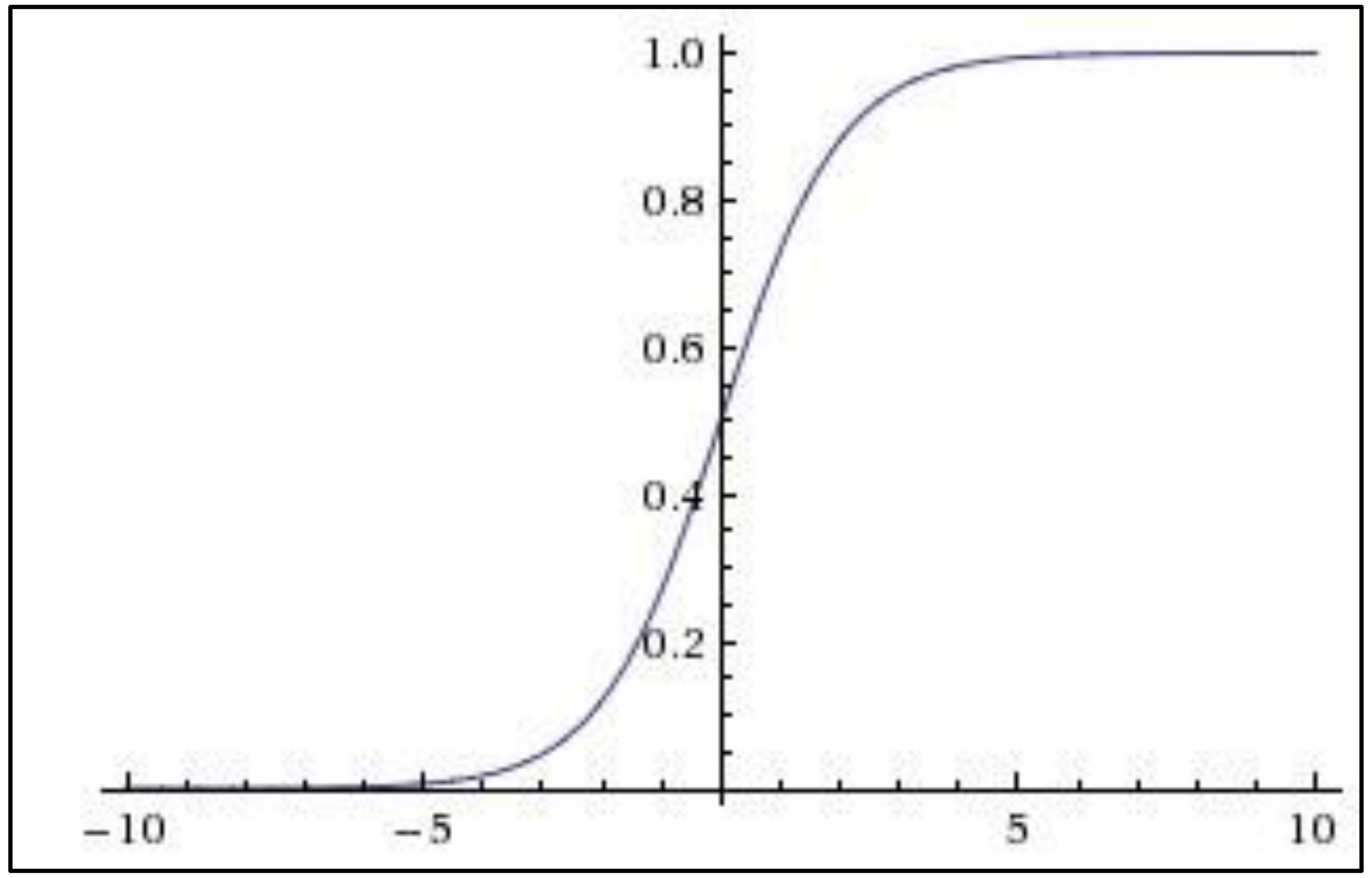






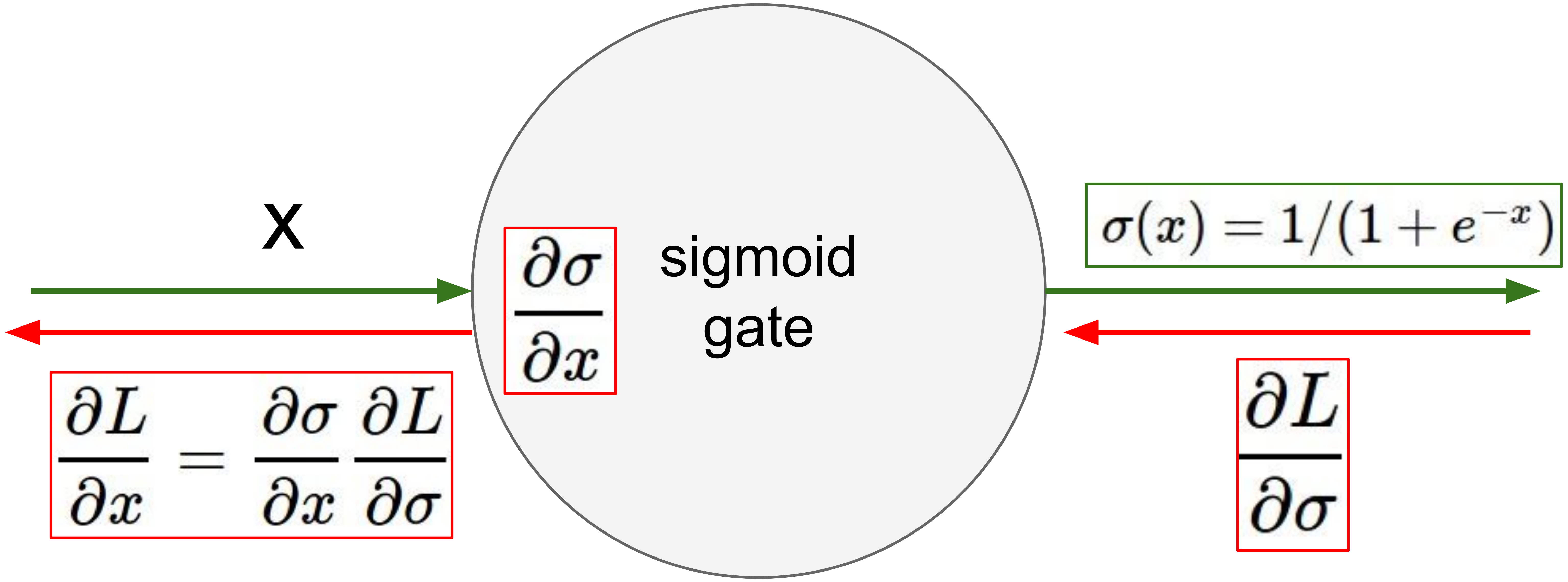
$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1-\sigma(x))$$

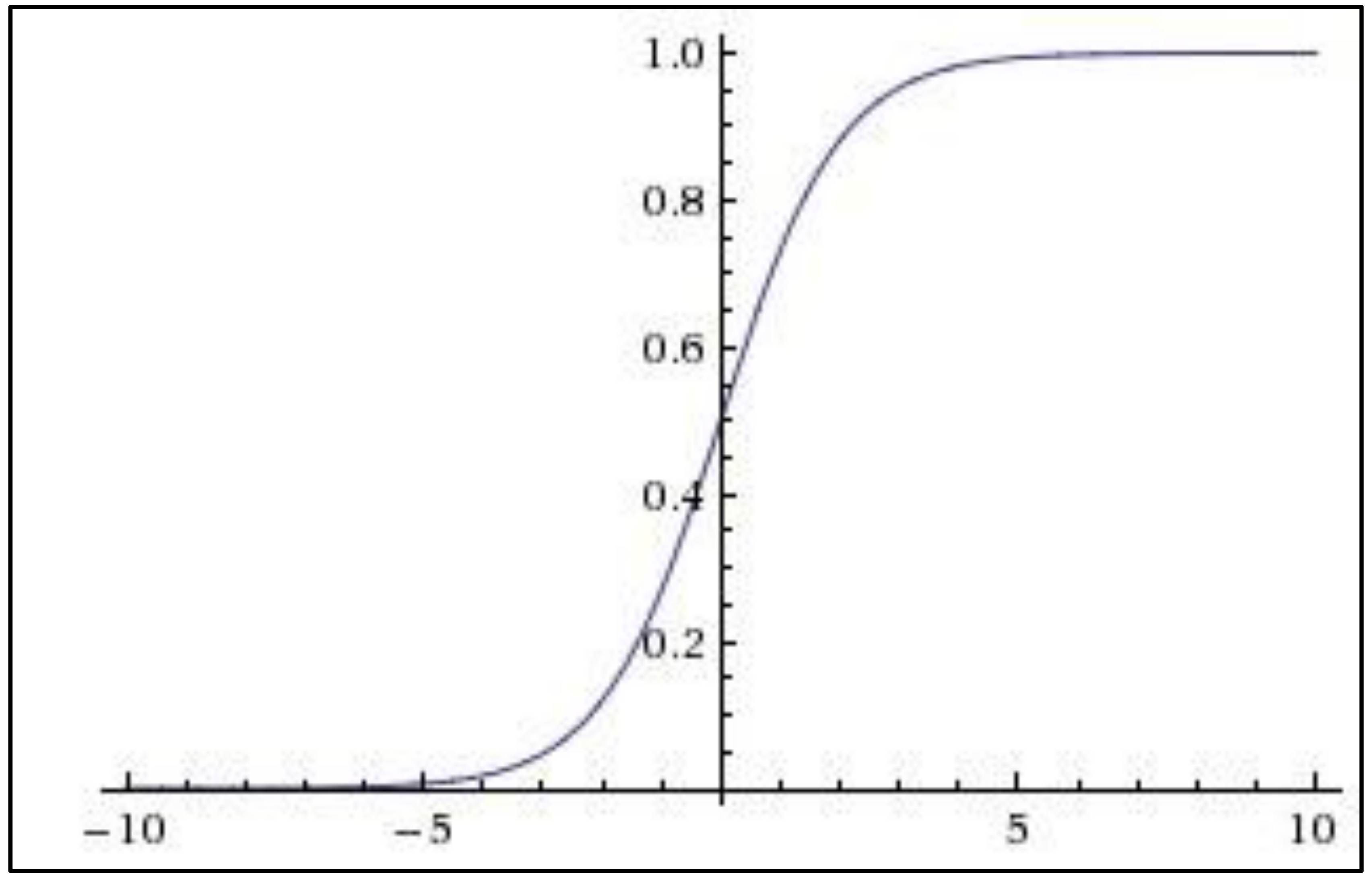




$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1 - \sigma(x))$$

What happens when x = -10



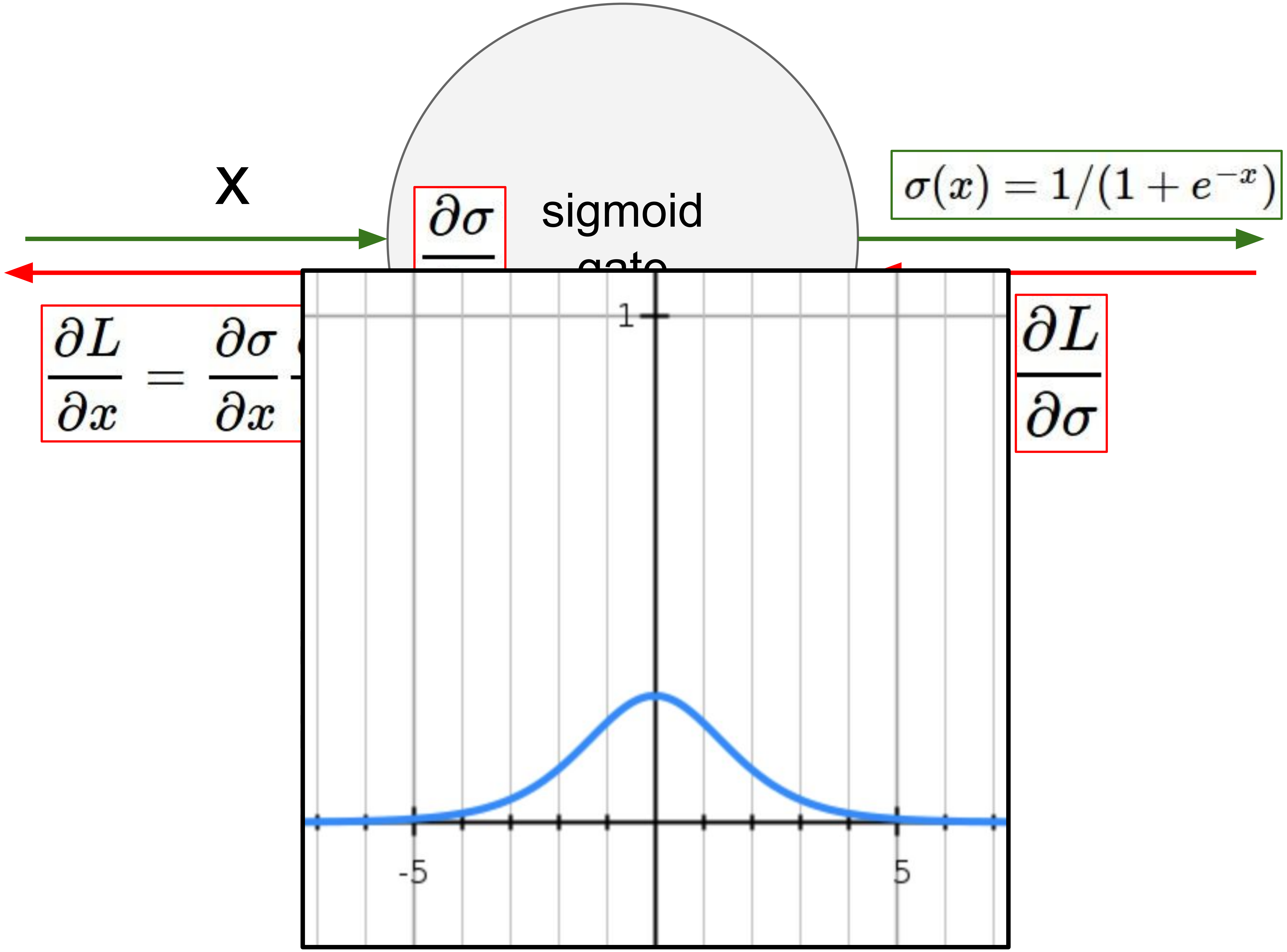


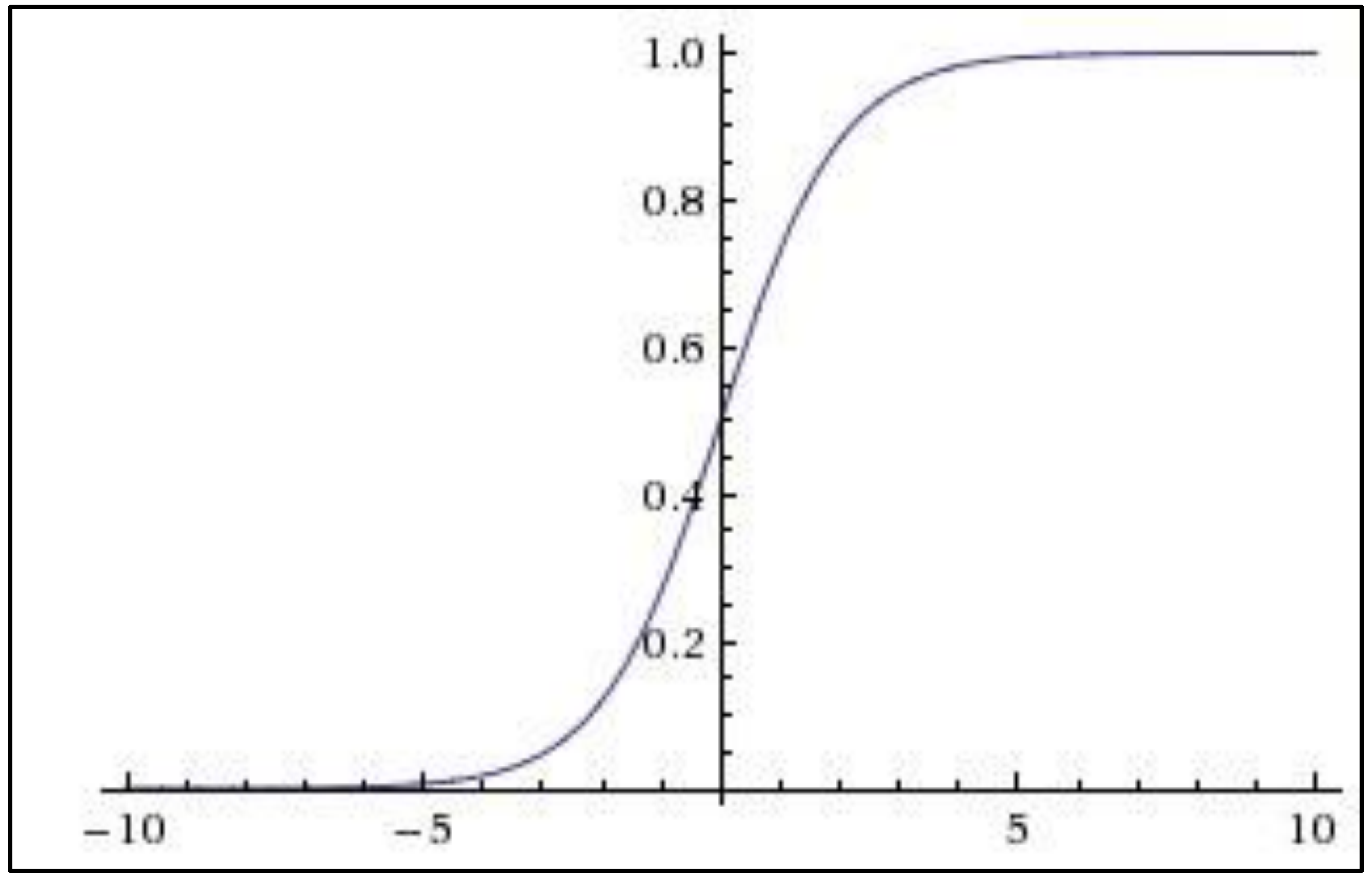
$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1-\sigma(x))$$

What happens when x = -10

What happens when x = 0

What happens when x = 10



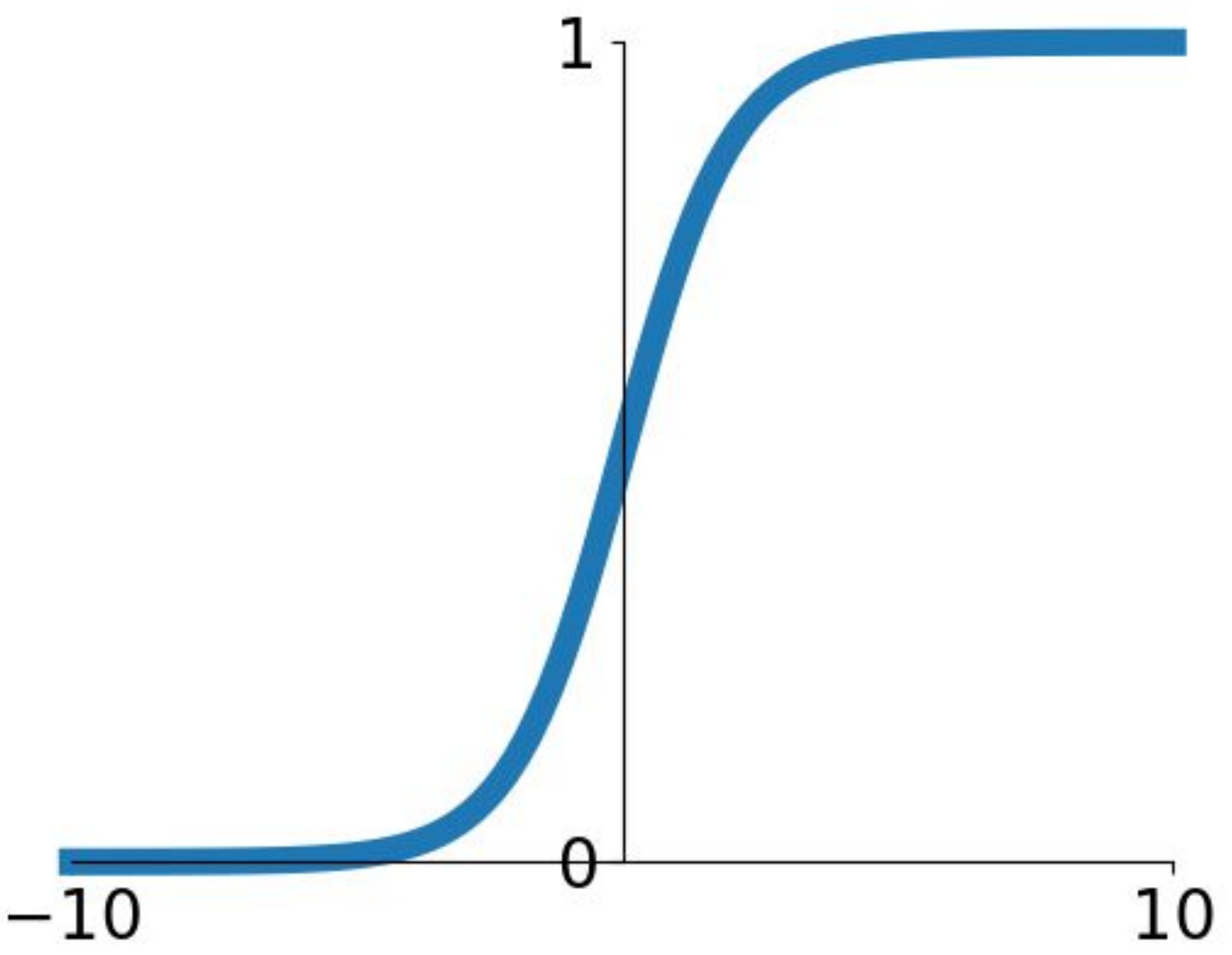


$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1-\sigma(x))$$

Activation Functions - Sigmoid

- $\sigma(x) = 1/(1 + e^{-x})$
- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron
- 3 problems:

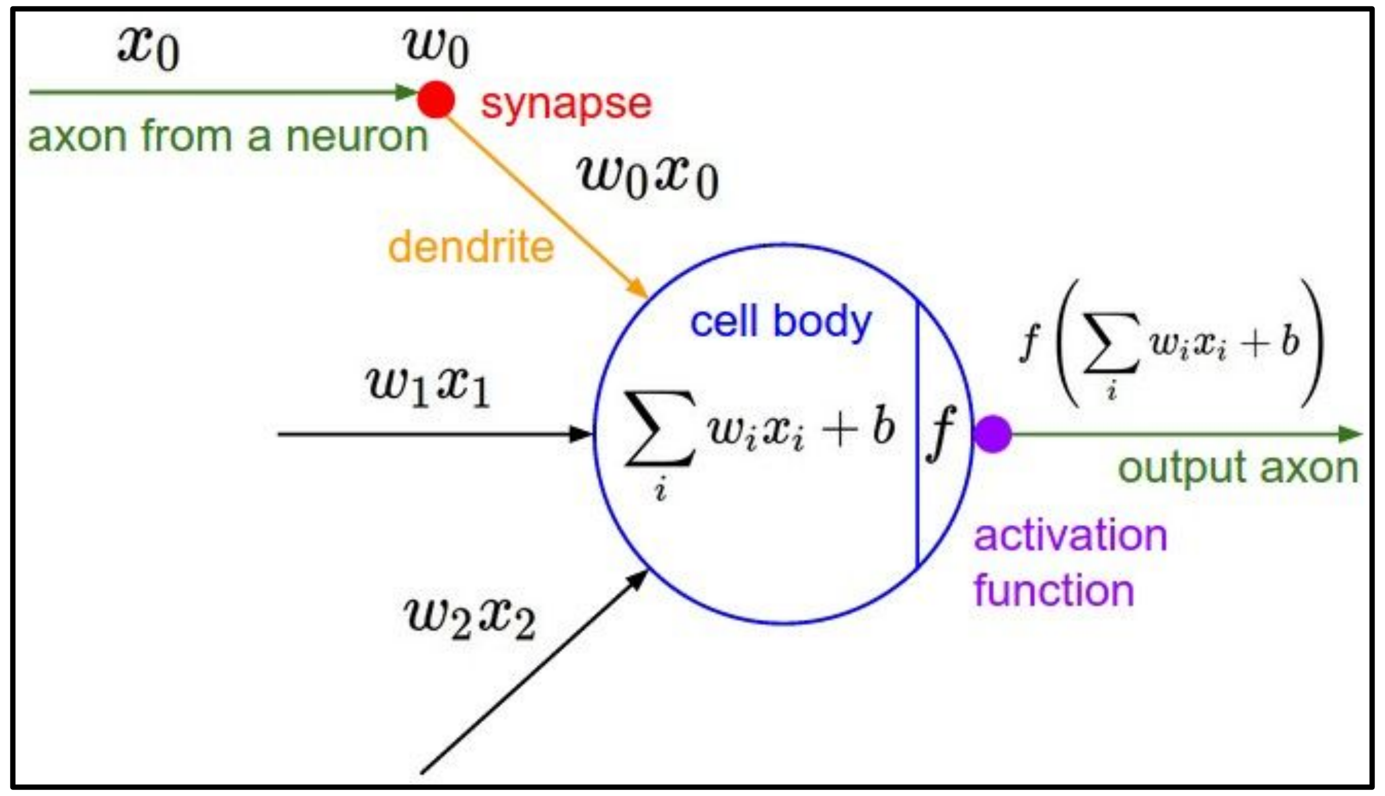
1. Saturated neurons "kill" the gradients
2. Sigmoid outputs are not zero-centered



Sigmoid

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

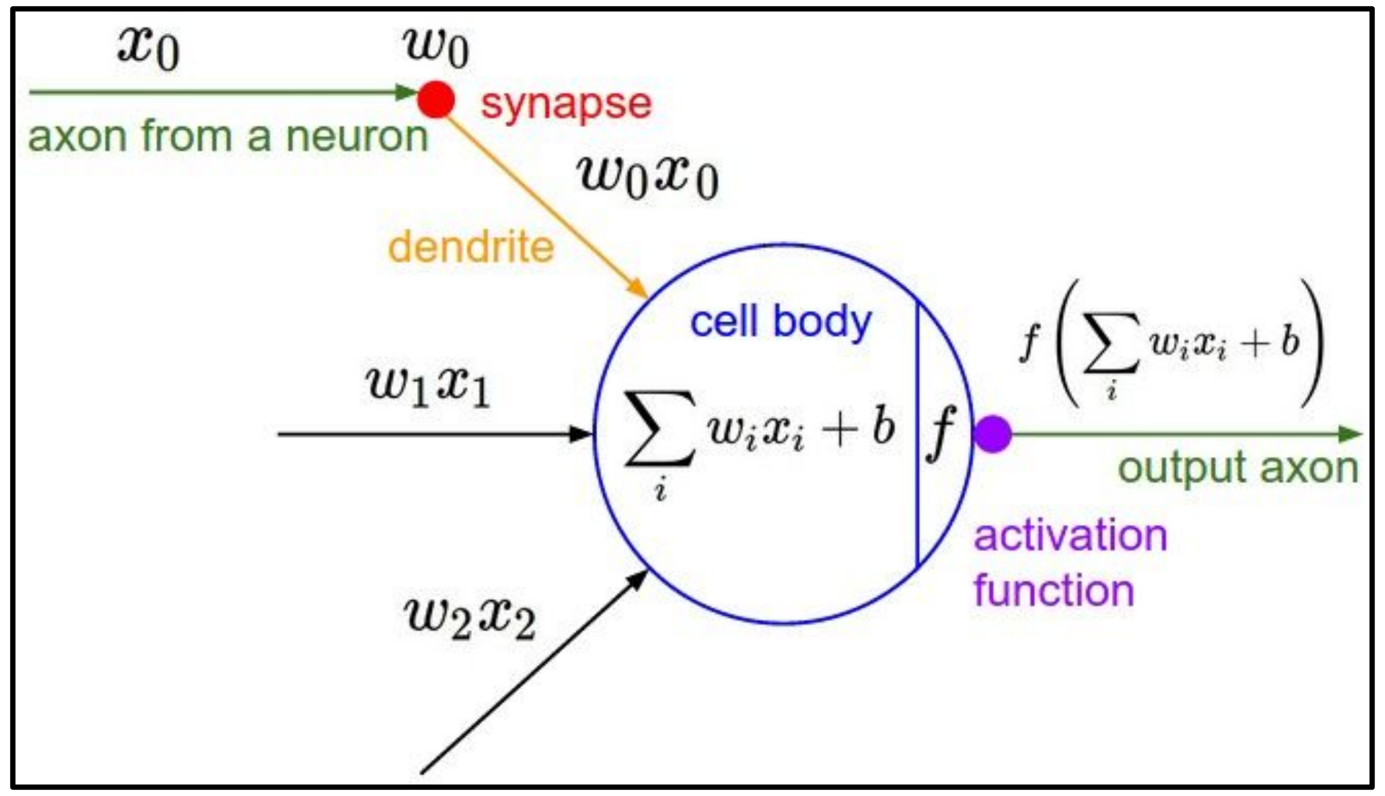
$$f(\sum_i w_i x_i + b)$$



- What can we say about the gradient on w_i?

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

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



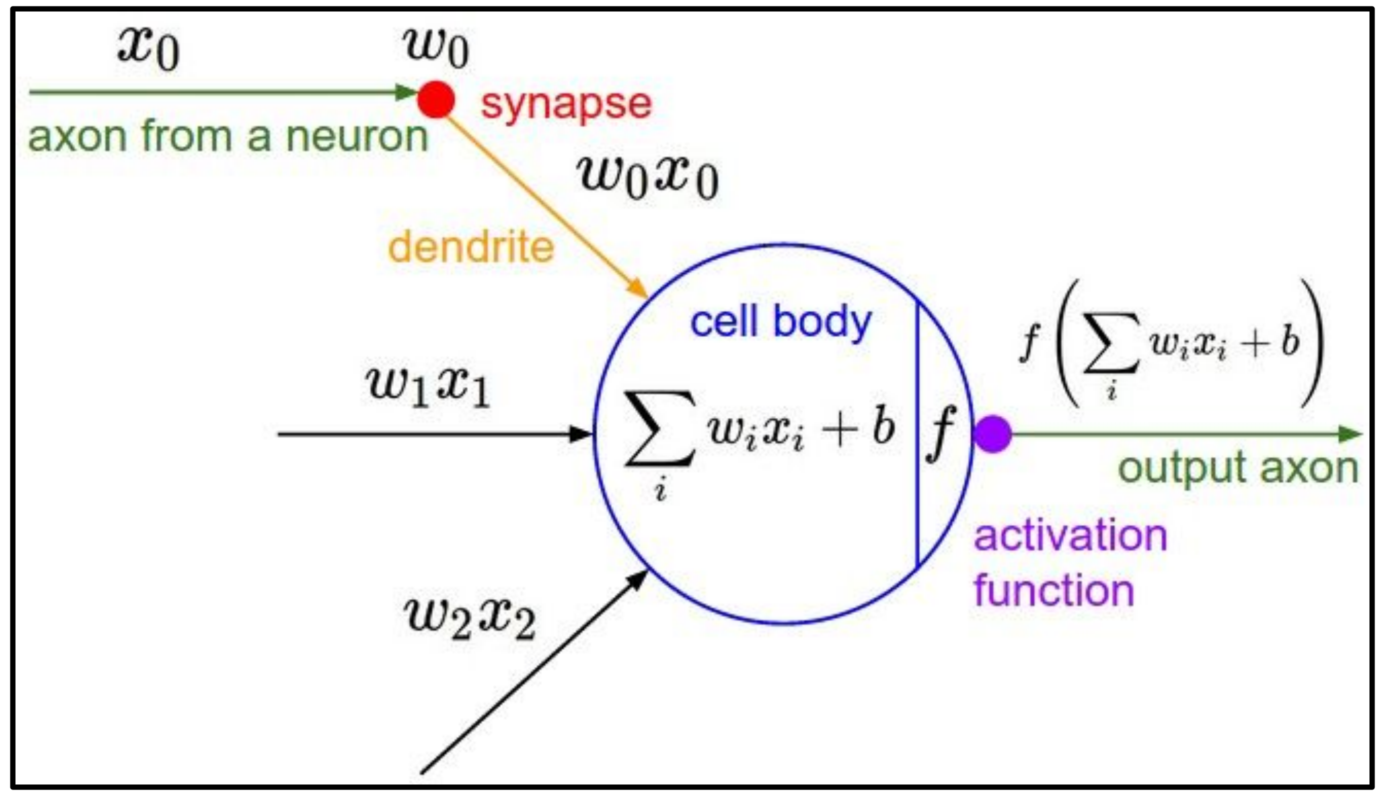
- What can we say about the gradient on w_i?

- We know the local gradient of sigmoid is always positive

$$\frac{\partial L}{\partial w} = \boxed{\sigma(\sum_i w_i x_i + b)(1 - \sigma(\sum_i w_i x_i + b))} x \times \text{upstream gradient}$$

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

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

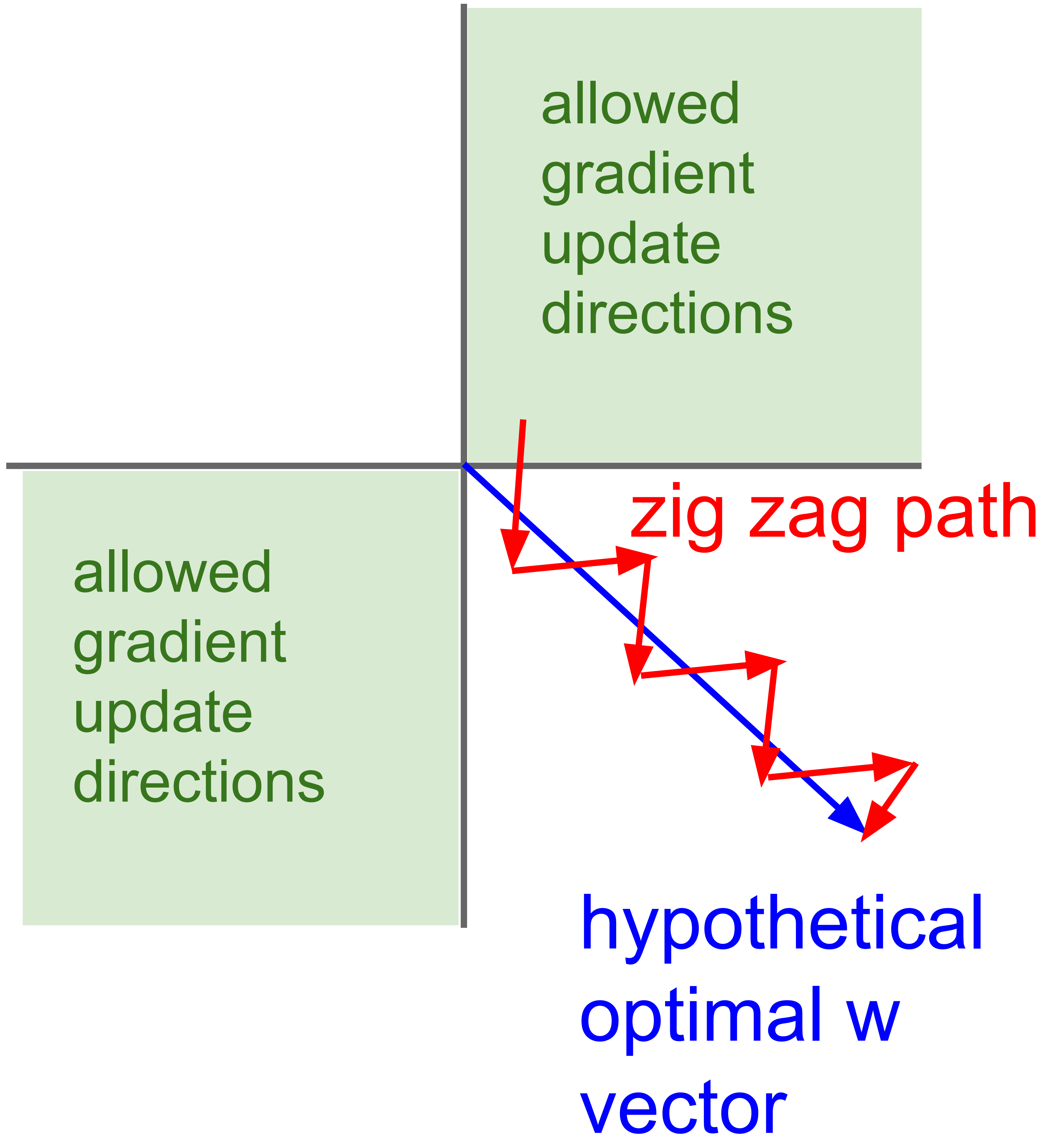


- What can we say about the gradient on w_i?
- We know the local gradient of sigmoid is always positive
- x is always positive (if x is the output of previous layer)
- Therefore, sign of gradient for all w_i is the same as the sign of upstream *scalar* gradient!

$$\frac{\partial L}{\partial w} = \boxed{\sigma(\sum_i w_i x_i + b)(1 - \sigma(\sum_i w_i x_i + b))}\boxed{x} \times \text{upstream gradient}$$

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

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

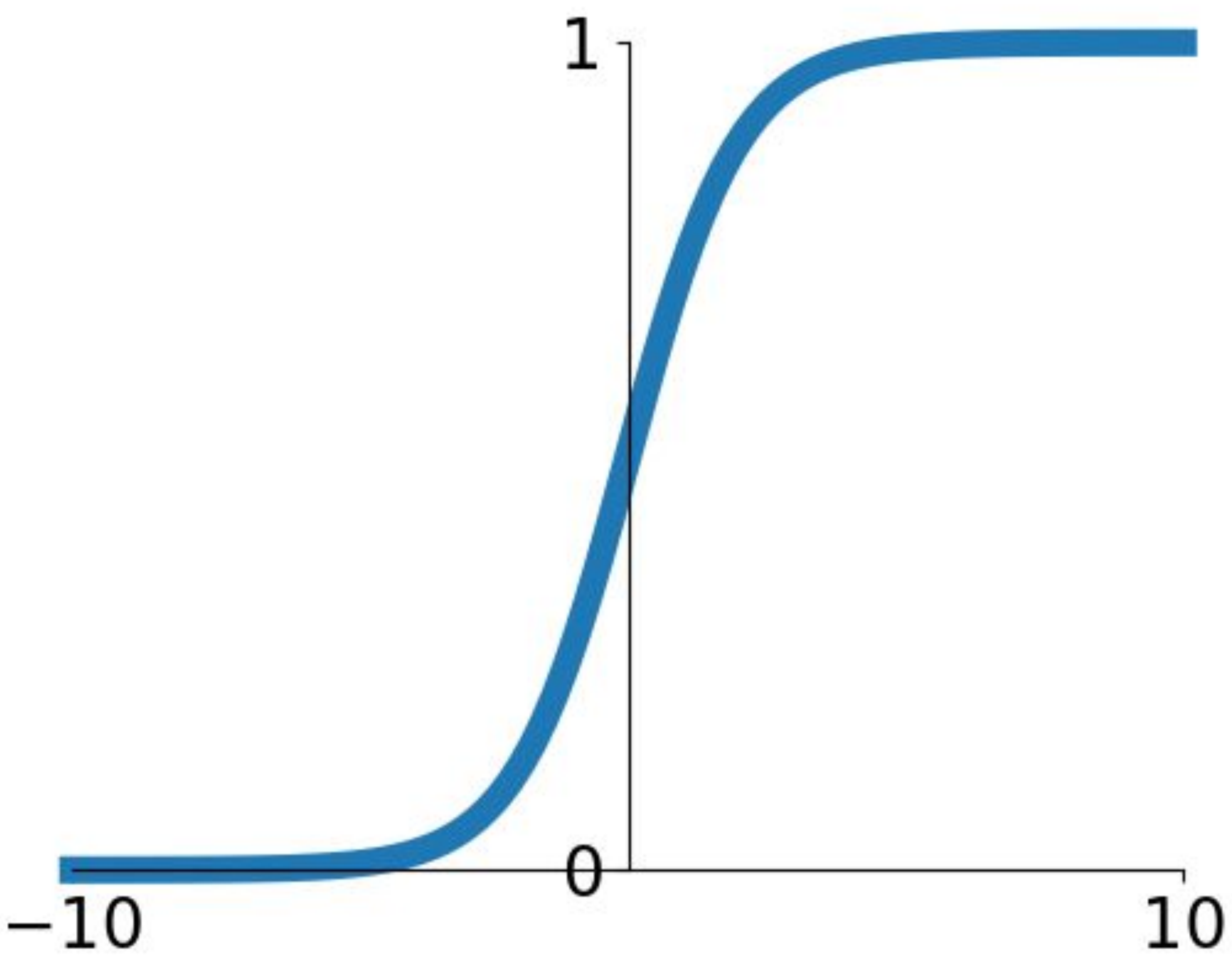


- What can we say about the gradients on w_i?
- Always all positive or all negative :(
- (For a single element! mini-batches help)

Activation Functions - Sigmoid

- $\sigma(x) = 1/(1+e^{-x})$
- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron
- 3 problems:

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



Sigmoid

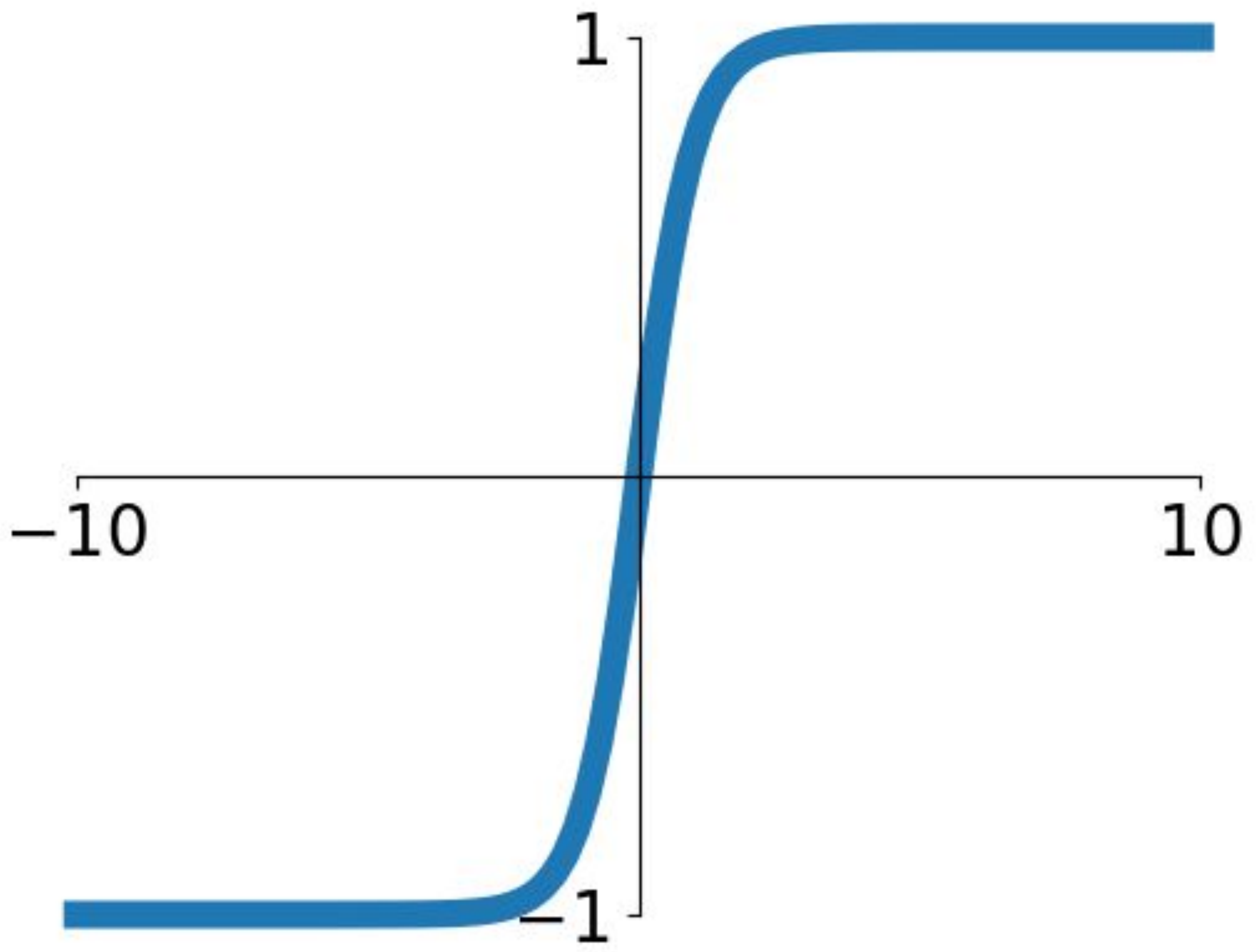
What if we have very deep layers?
- vanishing gradients

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

Sigmoid

Activation Functions - Hyperbolic Tangent

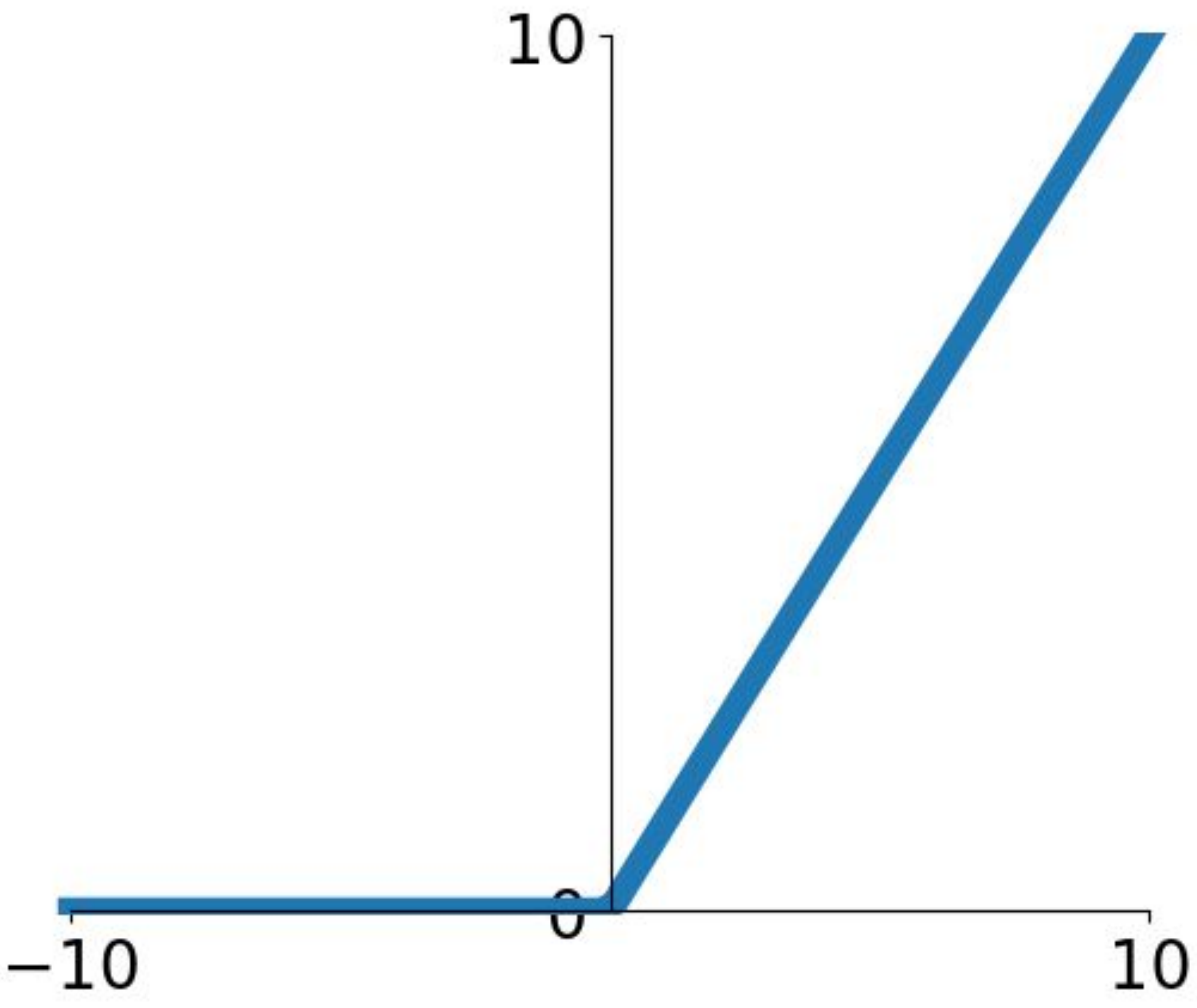
- Squashes numbers to range [-1,1]
- zero centered (nice)
- still kills gradients when saturated :(



tanh(x)

Activation Functions - Rectified Linear Unit

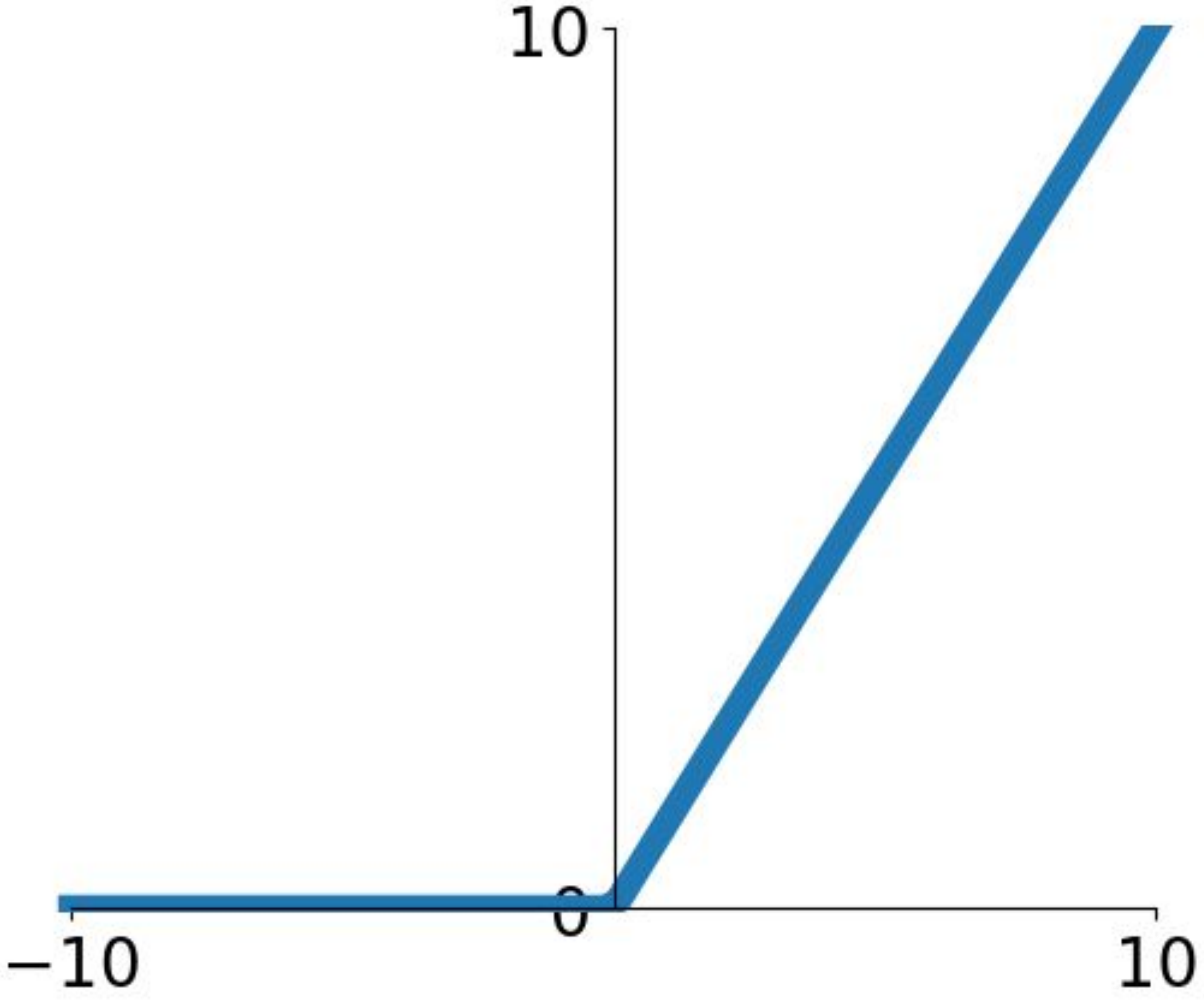
- Computes f(x) = max(0,x)
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)



ReLU
(Rectified Linear Unit)

Activation Functions - Rectified Linear Unit

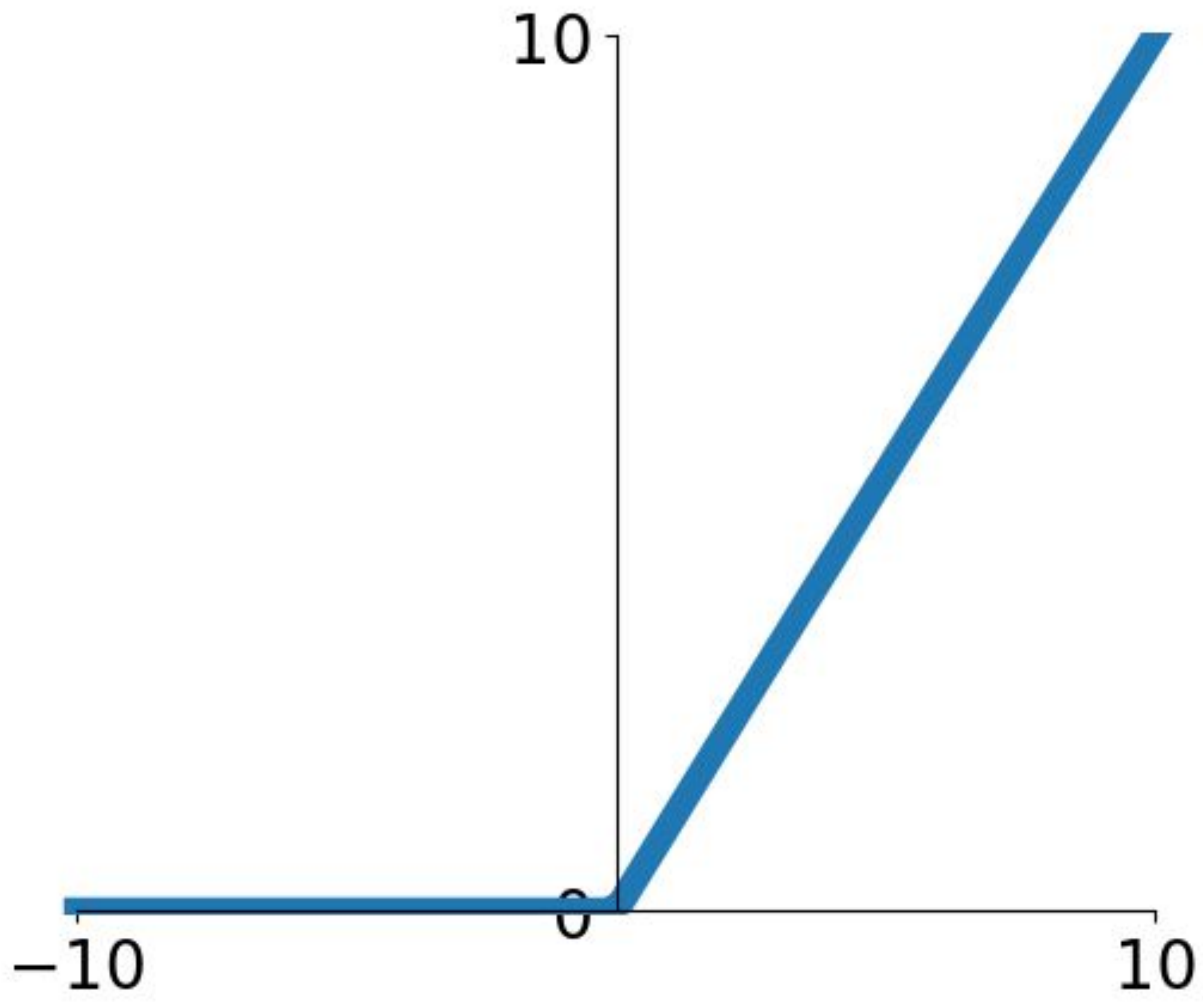
- Computes f(x) = max(0,x)
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- Sparse gradients



ReLU
(Rectified Linear Unit)

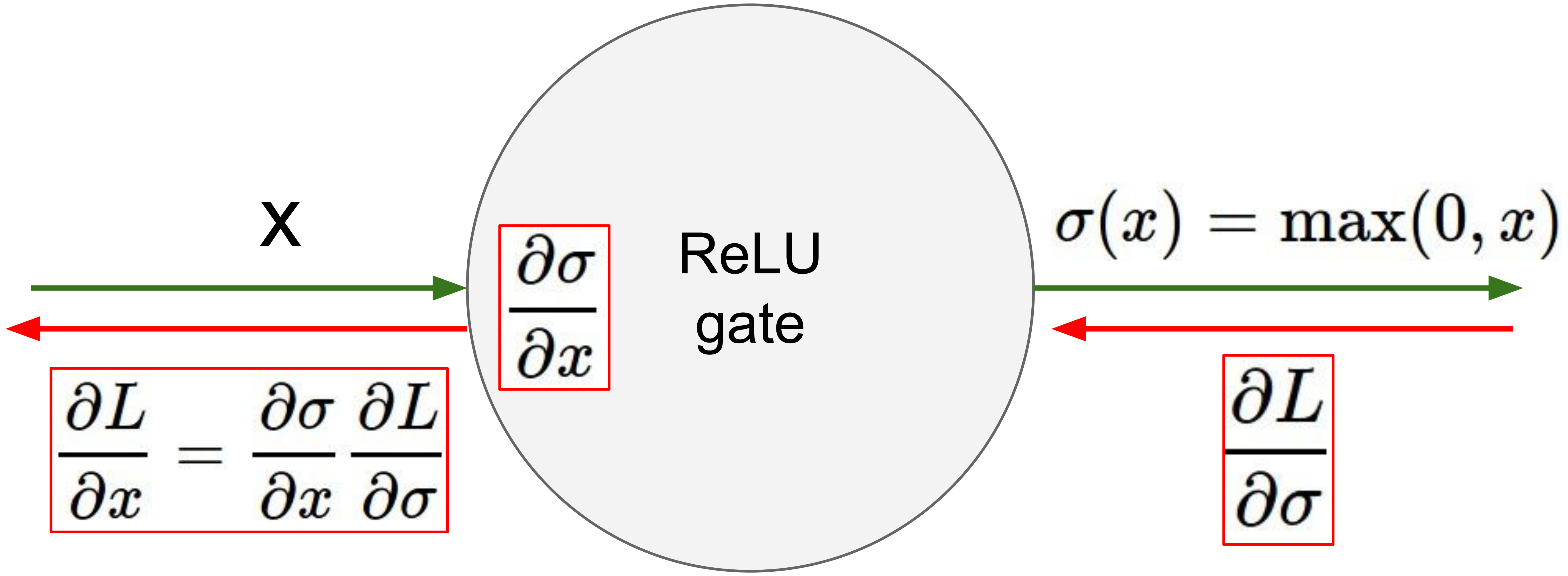
Activation Functions - Rectified Linear Unit

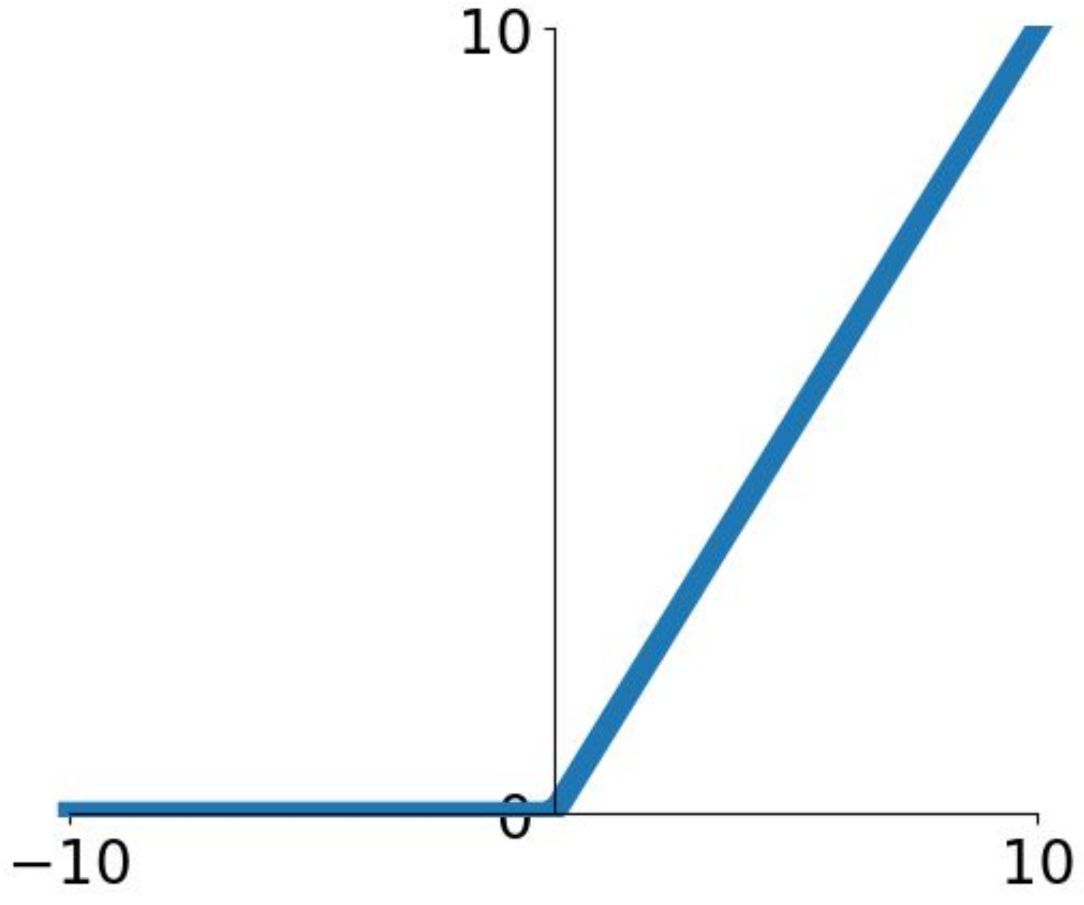
- Computes f(x) = max(0,x)
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- Sparse gradients
- Not zero centered output
- An annoyance
  - What is the gradient when x<0?



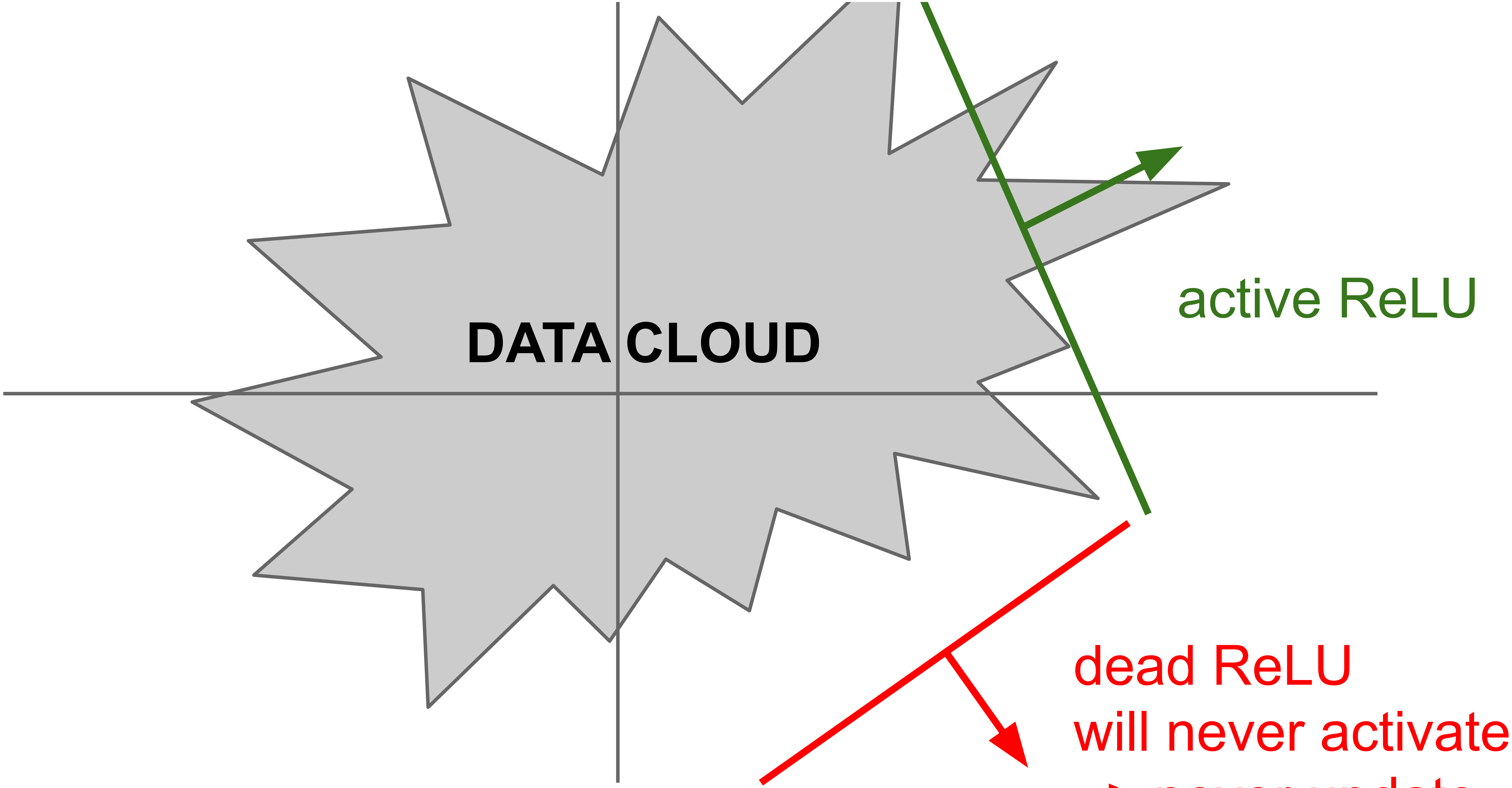
ReLU
(Rectified Linear Unit)

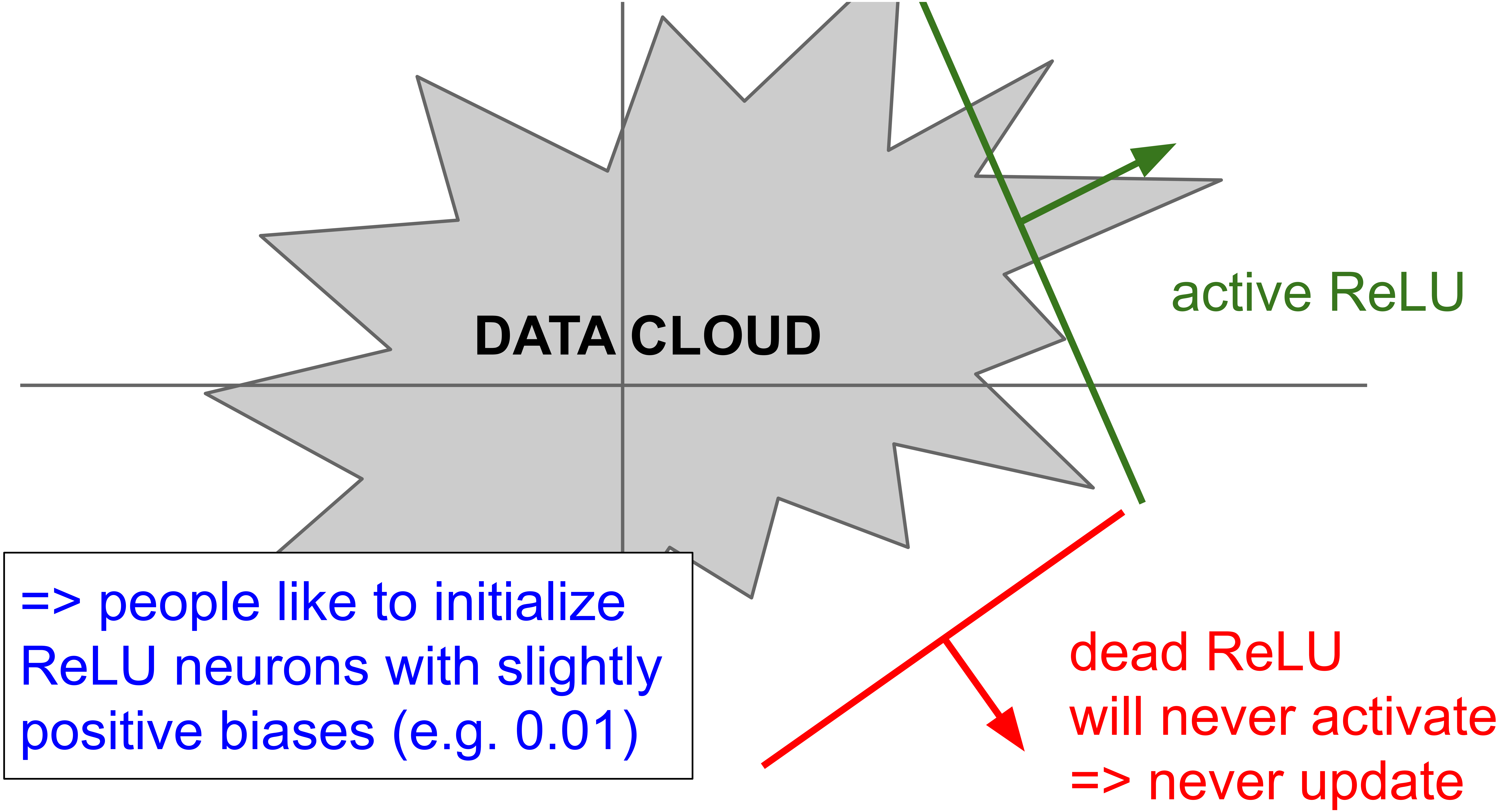
Activation Functions - Rectified Linear Unit





- 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

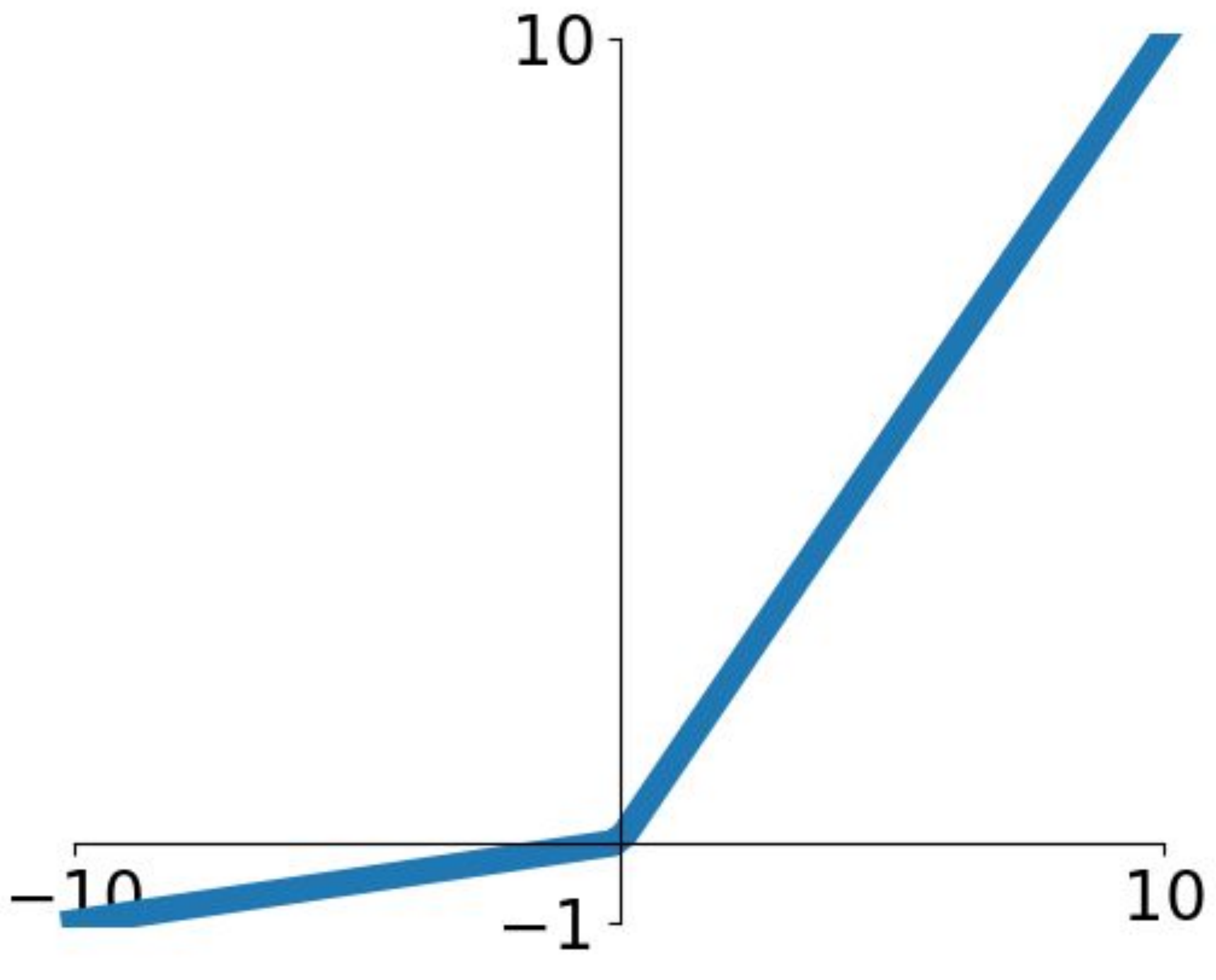


=> people like to initialize ReLU neurons with slightly positive biases (e.g. 0.01)

Activation Functions - Leaky ReLU

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

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

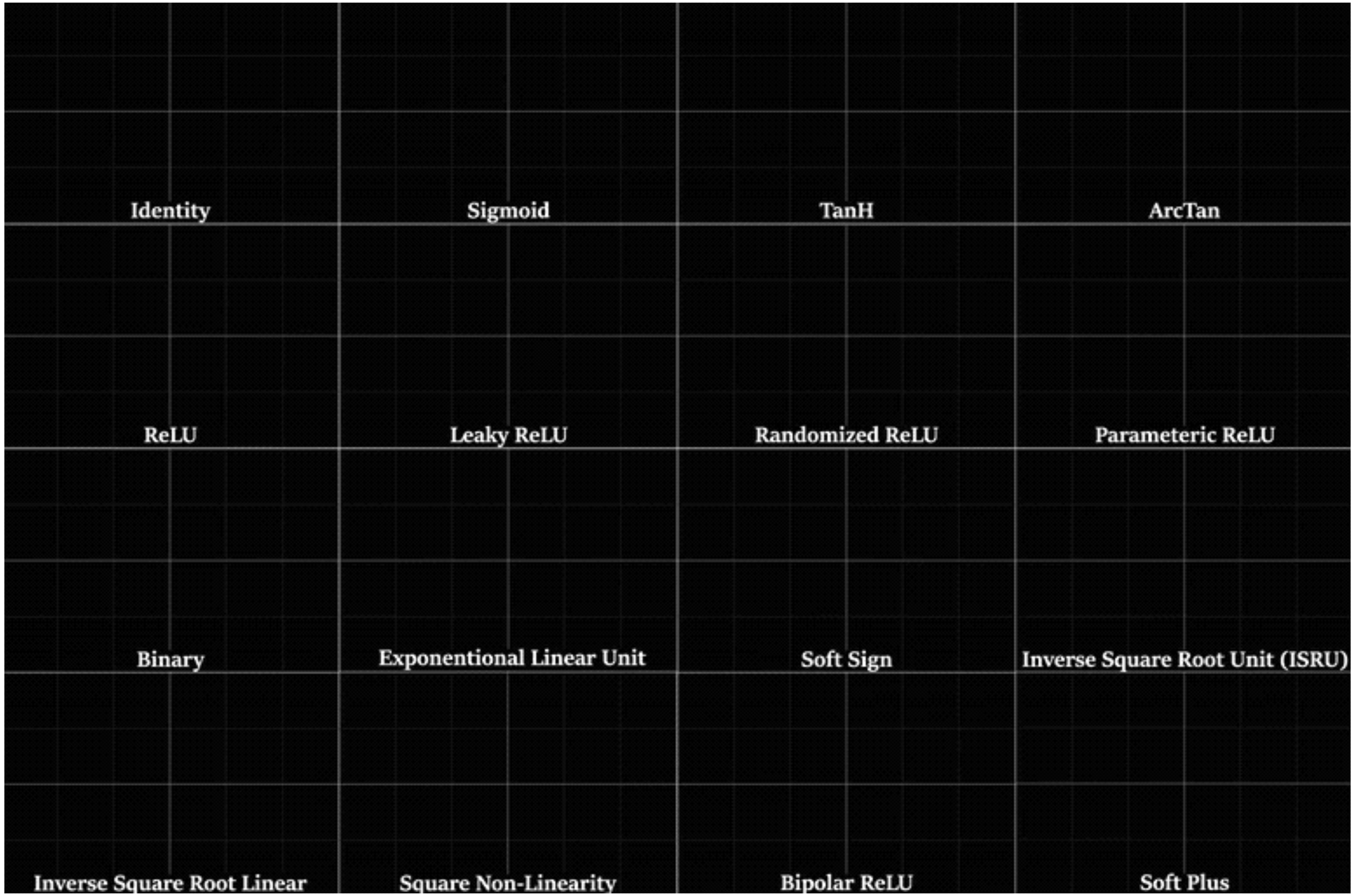


Leaky ReLU

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

TDLR: In practice:

- Use ReLU. Be careful with your learning rates
- Don't use sigmoid or tanh
- There are many more: https://mlfromscratch.com/activation-functions-explained/

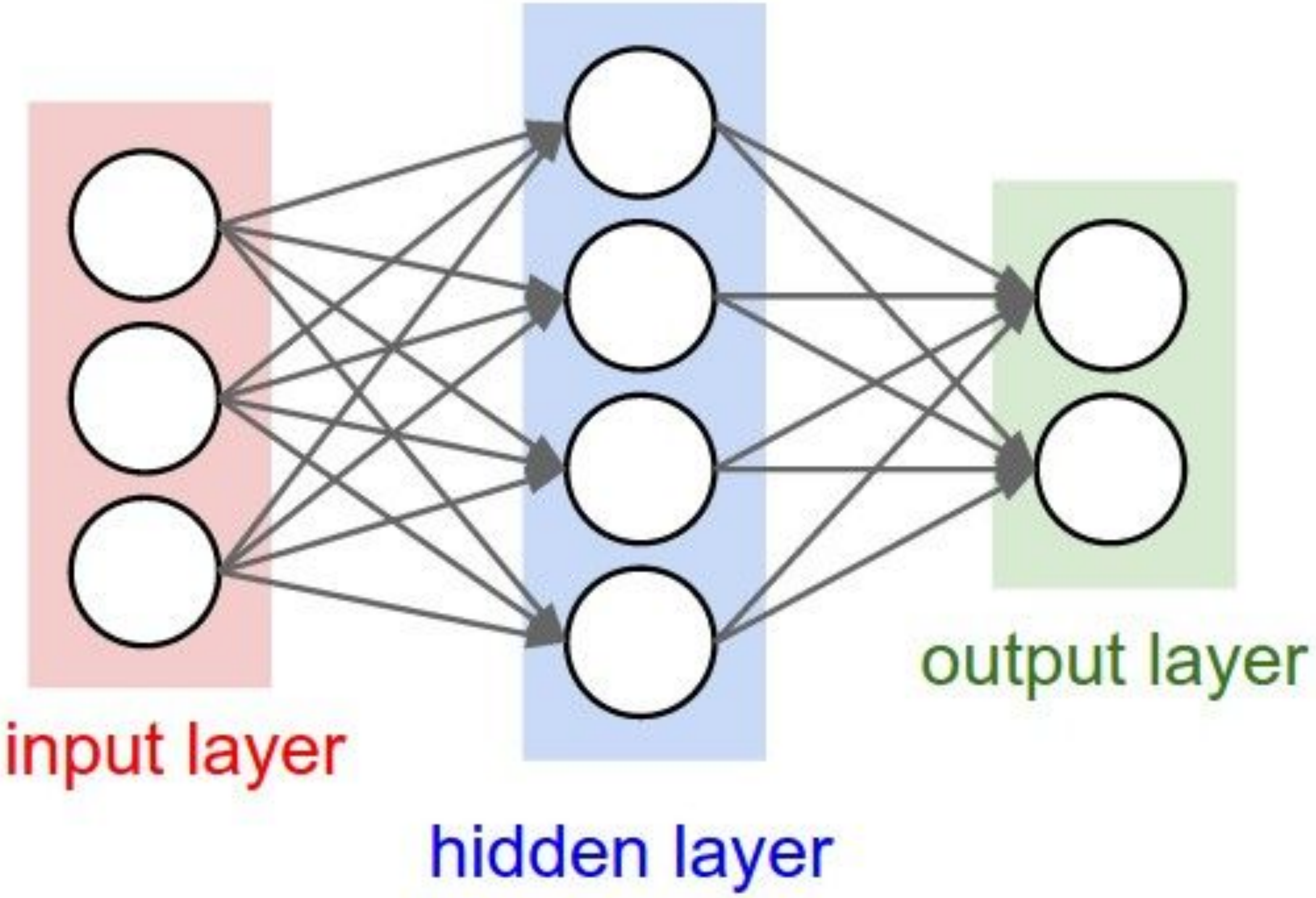


Outline

- Activation Functions
- **Weight Initialization**
- Batch Normalization

Weight Initialization

- Q: what happens when W=constant initialization is used?



Weight Initialization

- First idea: small random numbers
  - Gaussian with zero mean and 1e-2 standard deviation

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

Weight Initialization

- 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 problem with deeper networks.

Weight Initialization: Activation Statistics

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

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

- What will happen to the activation for the last layer?

Weight Initialization: Activation Statistics

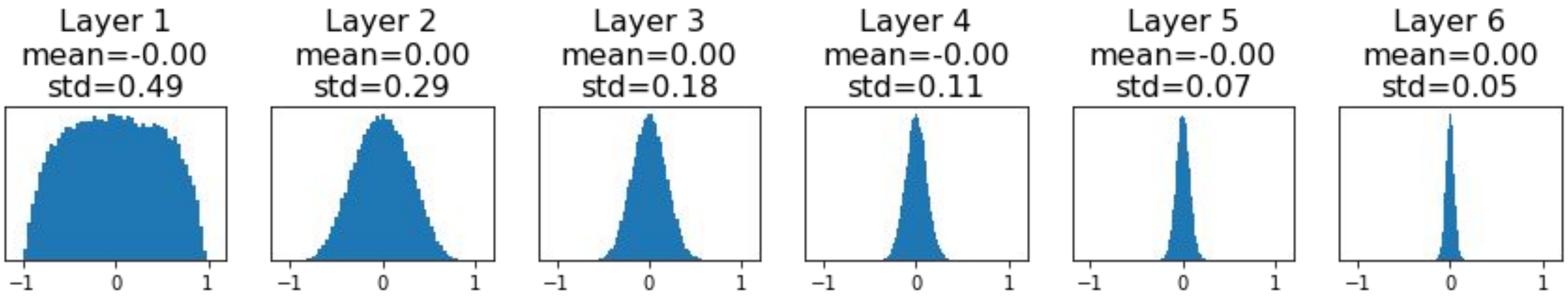
```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

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

All activations tend to zero for deeper network layers

Q: What do the gradient look like?

- What will happen to the activation for the last layer?



Weight Initialization: Activation Statistics

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

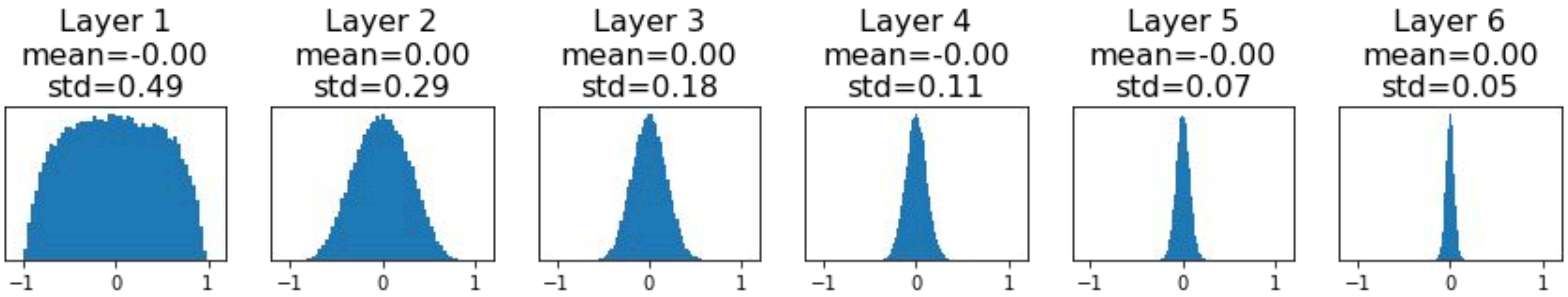
Forward pass for a 6-layer net with hidden size 4096

All activations tend to zero for deeper network layers

Q: What do the gradient look like?

A: All zero, no learning

- What will happen to the activation for the last layer?



Weight Initialization: Activation Statistics

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

Increase std of initial weights from 0.01 to 0.05

- What will happen to the activation for the last layer?

Weight Initialization: Activation Statistics

```python
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)
```

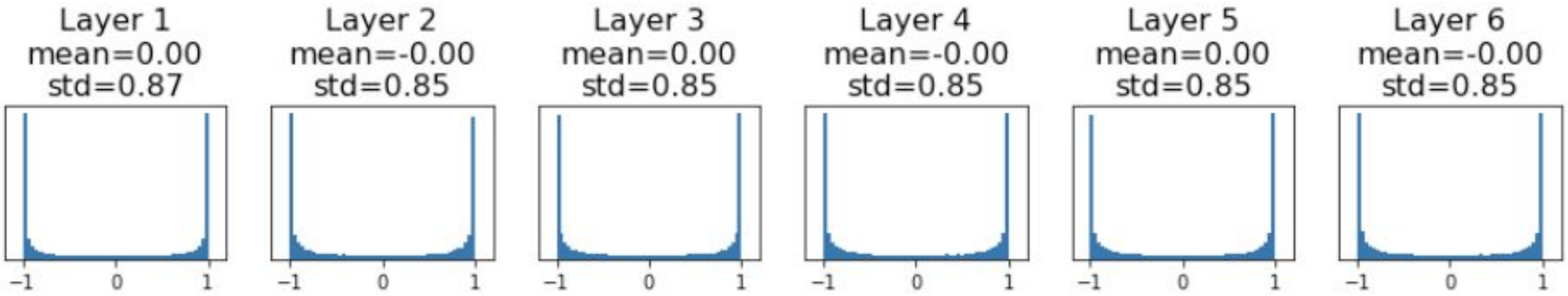
Increase std of initial weights from 0.01 to 0.05

All activations are saturated

Q: What do the gradient look like?

A: Local gradient all zero, no learning

- What will happen to the activation for the last layer?



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)

- What will happen to the activation for the last layer?

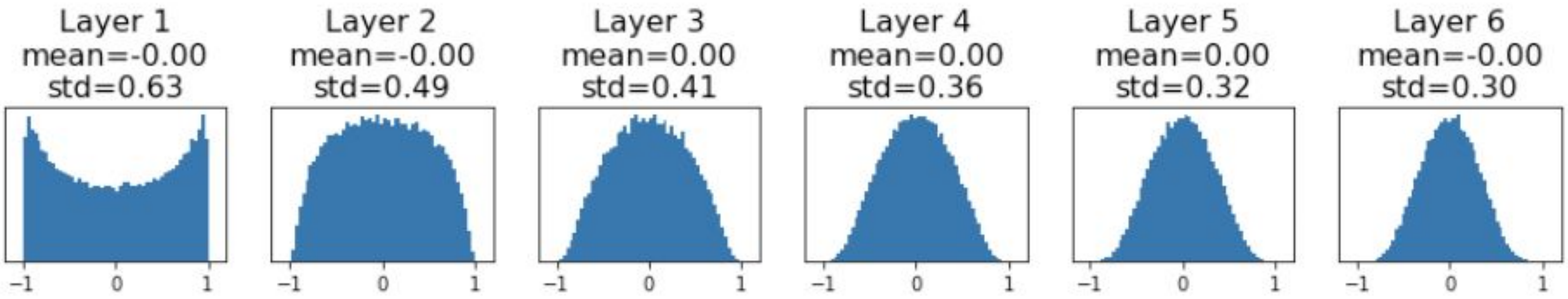
Weight Initialization: “Xavier” Initialization

```
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!

- What will happen to the activation for the last layer?



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!

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

$$\begin{aligned} \mathrm{Var}(\mathrm{y_i}) &= \mathrm{Din} * \mathrm{Var}(\mathrm{x_i w_i}) \\ &= \mathrm{Din} * \left(\mathrm{E}\left[\mathrm{x_i^2}\right]\mathrm{E}\left[\mathrm{w_i^2}\right] - \mathrm{E}\left[\mathrm{x_i}\right]^2 \mathrm{E}\left[\mathrm{w_i}\right]^2\right) \\ &= \mathrm{Din}^* \mathrm{Var}(\mathrm{x_i})^* \mathrm{Var}(\mathrm{w_i}) \end{aligned}$$

→ assume x and w are iid

→ assume x and w 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(Din)

Just right: activations are nicely scaled for all layers!

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

$$\begin{aligned}
\mathrm{Var}(\mathrm{y_i}) &= \mathrm{Din} * \mathrm{Var}(\mathrm{x_i w_i}) \\
&= \mathrm{Din} * \left(\mathrm{E}\left[\mathrm{x_i^2}\right] \mathrm{E}\left[\mathrm{w_i^2}\right] - \mathrm{E}\left[\mathrm{x_i}\right]^2 \mathrm{E}\left[\mathrm{w_i}\right]^2\right) \\
&= \mathrm{Din}^* \, \mathrm{Var}(\mathrm{x_i})^* \, \mathrm{Var}(\mathrm{w_i})
\end{aligned}$$

→ assume x and w are iid

→ assume x and w are zero mean

$$\boxed{\text{If } \mathrm{Var}(w_j) = 1/\mathrm{Din} \text{ then } \mathrm{Var}(y_j) = \mathrm{Var}(x_i)}$$

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

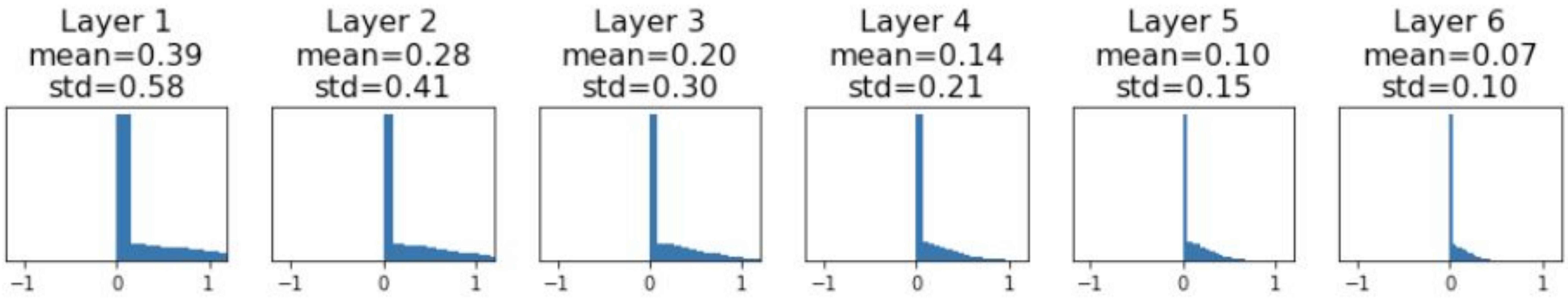
Weight Initialization: What about ReLU?

```
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

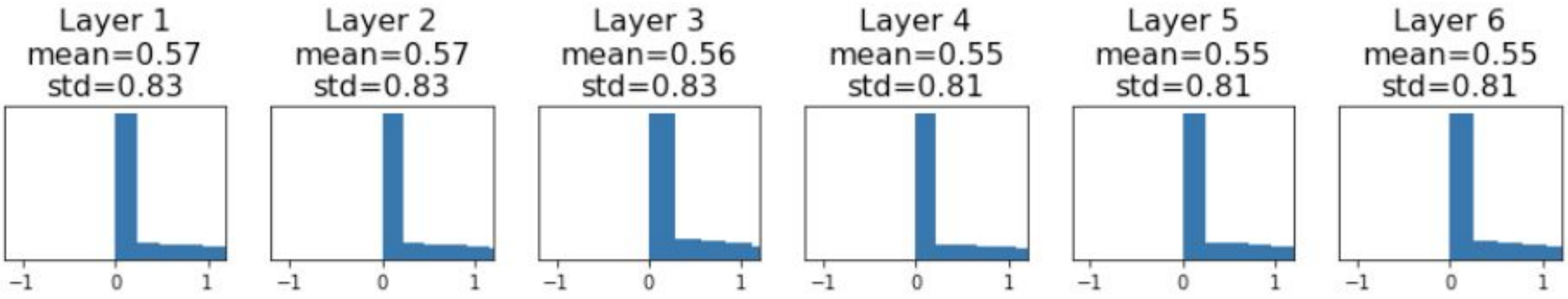


Weight Initialization: Kaiming / MSRA Initialization

```
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)
```

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

Outline

- Activation Functions
- Weight Initialization
- **Batch Normalization**

Batch Normalization - Ioffe and Szegedy, 2015

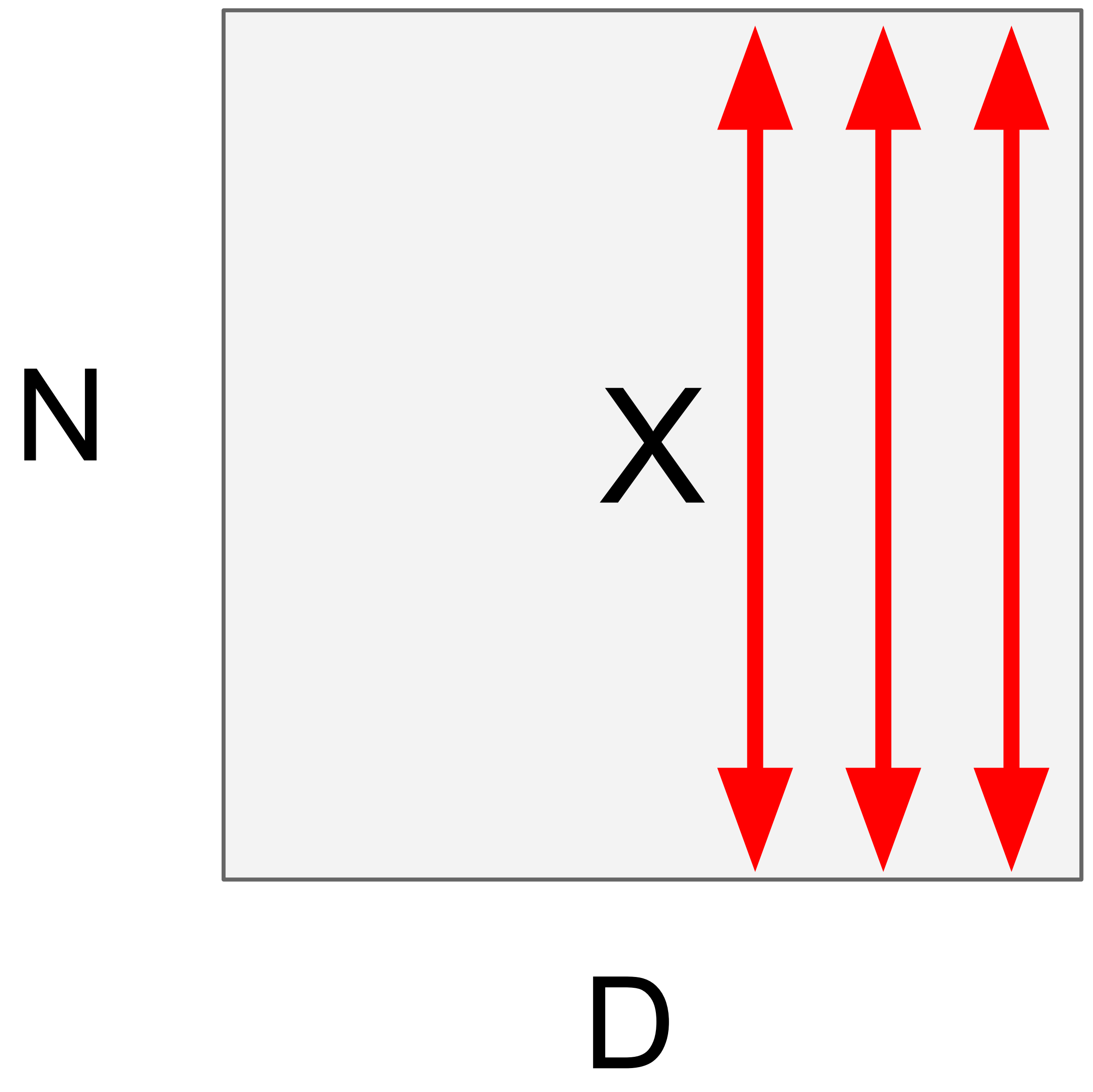
- "You want zero-mean unit-variance activations? Just make them so."
- Consider a batch of activations at some layer. To make each dimension zero-mean unit variance, apply.

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

This is a vanilla differentiable function

Batch Normalization

- Input: $X \in \mathbb{R}^{N \times D}$



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

Per-feature mean.
Shape is *D*

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

Per-feature var.
Shape is *D*

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

Normalized x.
Shape is $N \times D$

Batch Normalization

- Input: $X \in \mathbb{R}^{N \times D}$
- What if zero-mean, unit var is too hard of a constraints?
- Learnable scale and shift parameters:

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

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

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

Per-feature mean. Shape is *D*

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

Per-feature var. Shape is *D*

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

Normalized x. Shape is *N*×*D*

$$y_{ij} = \gamma_j \hat{x}_{ij} + \beta_j$$

Output. Shape is *N*×*D*

Batch Normalization

Estimations depends on mini batch
Can't do this at test time!

- Input: $X \in \mathbb{R}^{N \times D}$
- What if zero-mean, unit var is too hard of a constraints?
- Learnable scale and shift parameters:

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

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

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

Per-feature mean. Shape is *D*

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

Per-feature var. Shape is *D*

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

Normalized x. Shape is *N*×*D*

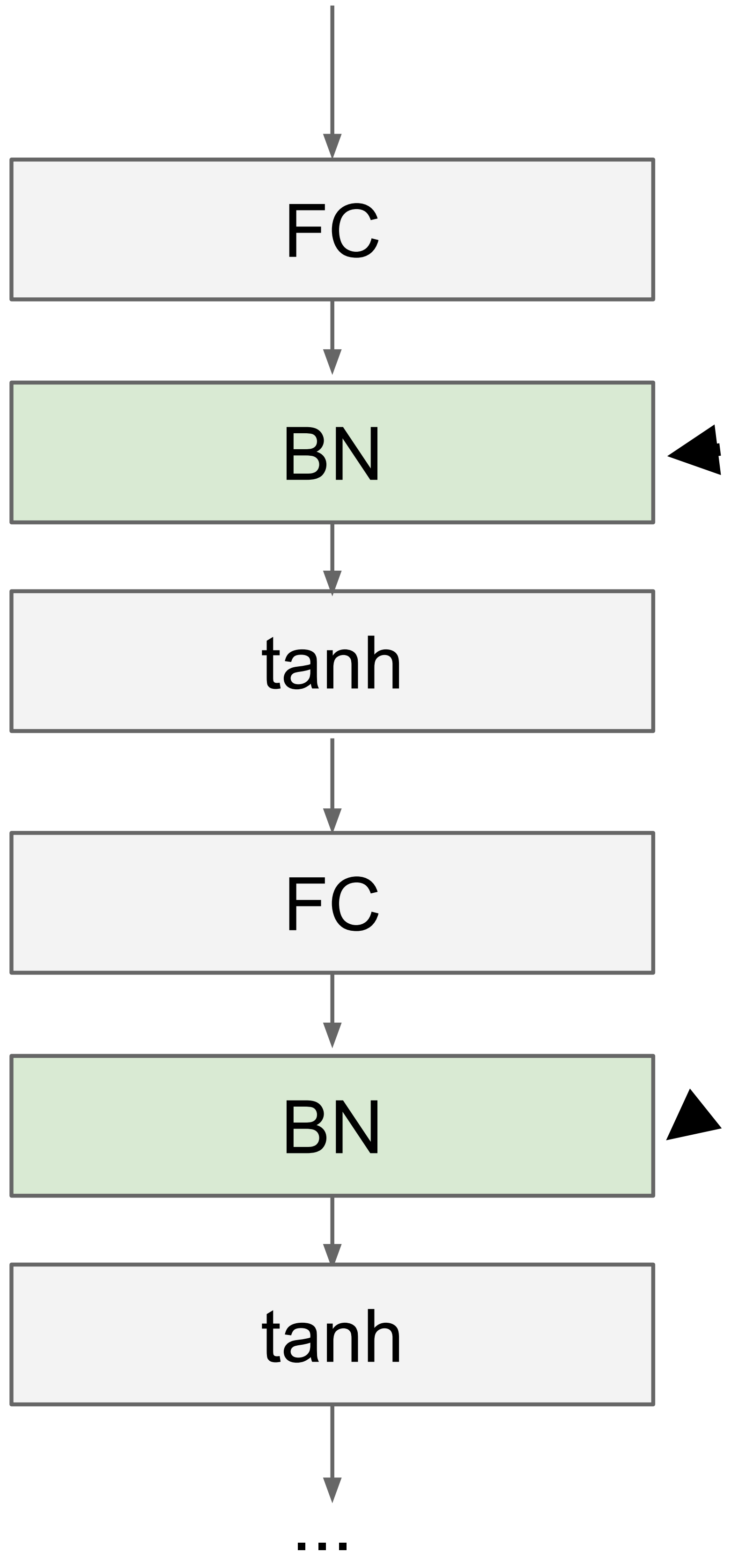
$$y_{ij} = \gamma_j \hat{x}_{ij} + \beta_j$$

Output. Shape is *N*×*D*

Batch Normalization: Test-time

- Set μ_j to be the average of values seen during training.
- Set σ_j^2 to be the average of values seen during training.
- During testing batch norm becomes a linear operator.
- Can be fused with previous fully-connected layer.

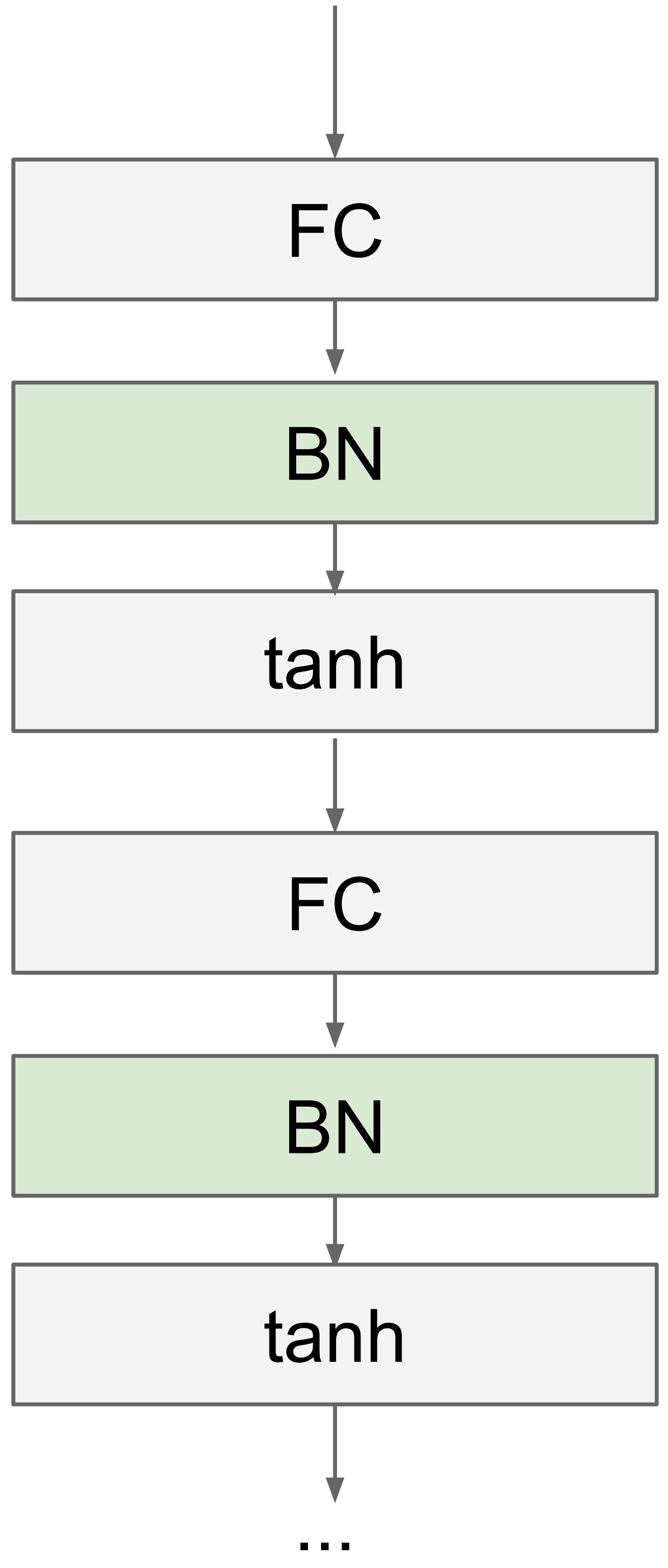
Batch Normalization



Usually inserted after fully connected or convolutional layers (will learn) and before nonlinearity

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

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
- Behaves differently during training and testing

Outline

- Activation Functions
- Weight Initialization
- Batch Normalization
- There are many more practical tips that we haven't discussed.
  - Fancy optimizer
  - Learning rate schedulers
  - Regularizer: Dropout
- CSED/AIGS538: Deep Learning!