THE UNIVERSITY OF BRITISH COLUMBIA

Topics in AI (CPSC 532S):
Multimodal Learning with Vision, Language and Sound

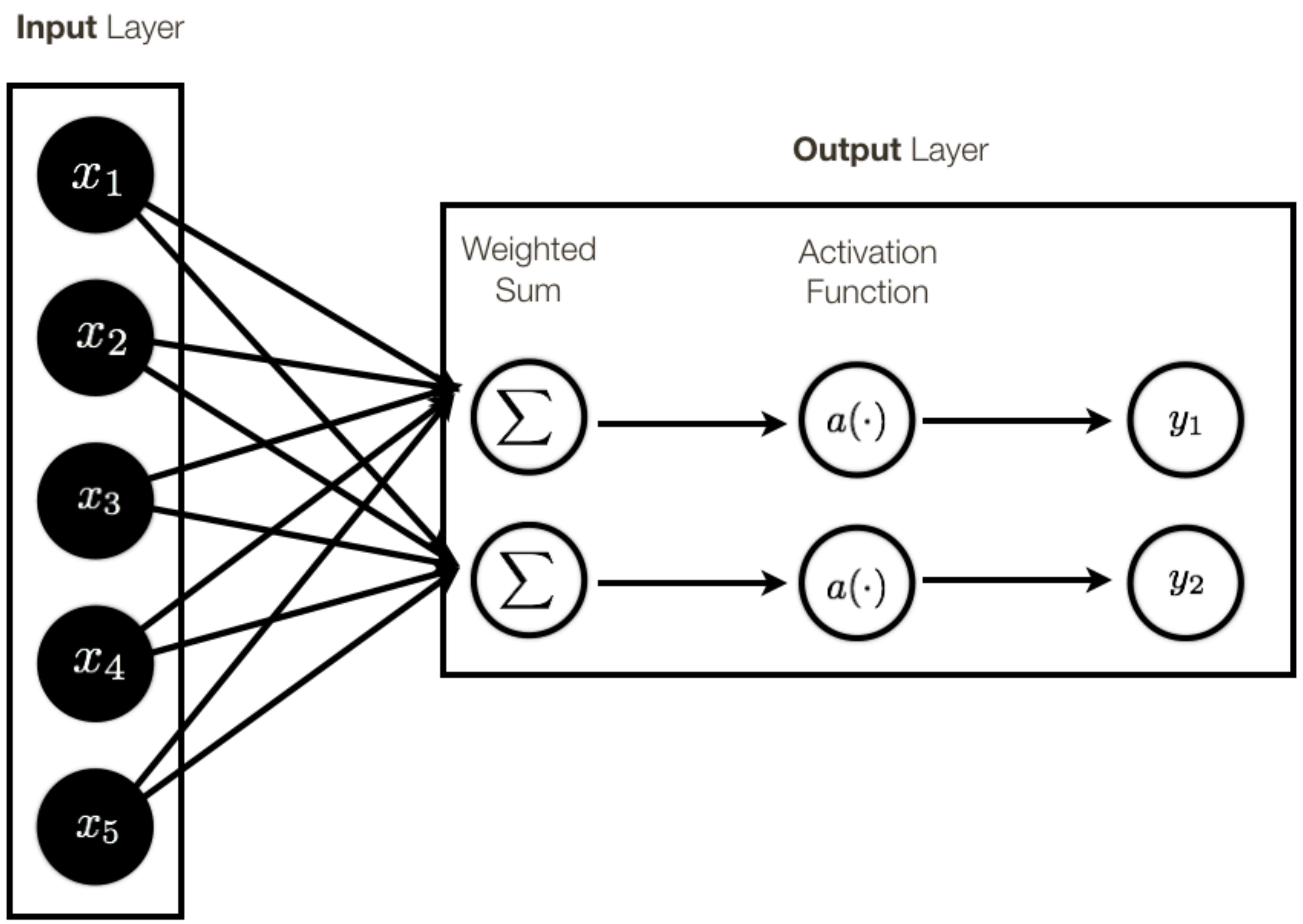
Lecture 3: Introduction to Deep Learning (continued)

Course **Logistics**

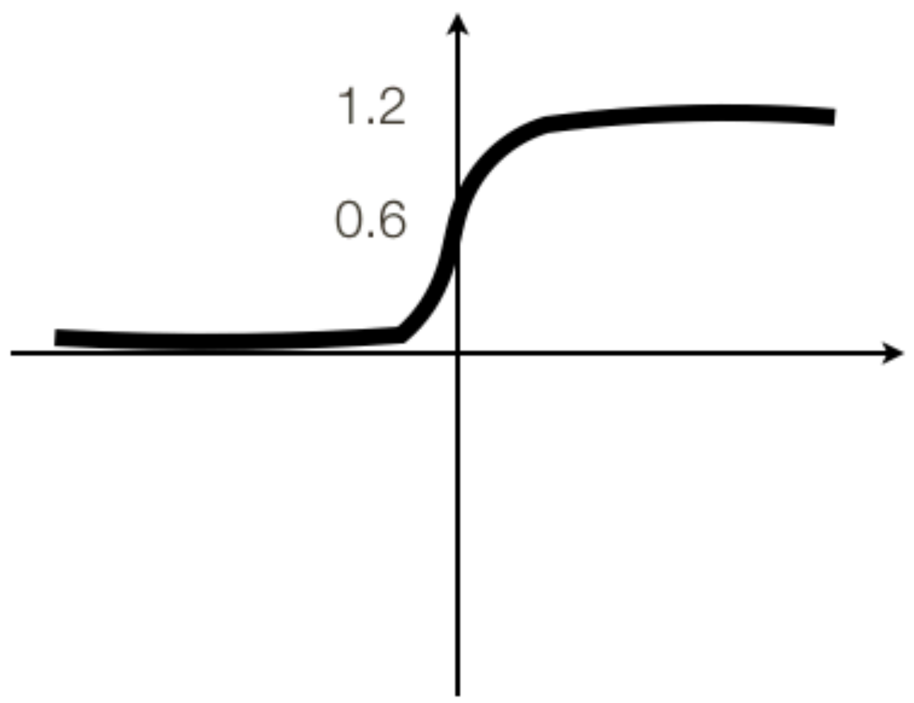
- Course **Registrations**: 3 seats are now available
- **Assignment 1** … any questions?
- My Office Hours — **Friday @ 12:30—1:30pm (hybrid)**

Short **Review** …

- Introduced the basic building block of Neural Networks **(MLP/FC) layer**



$$a(x) = \textbf{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

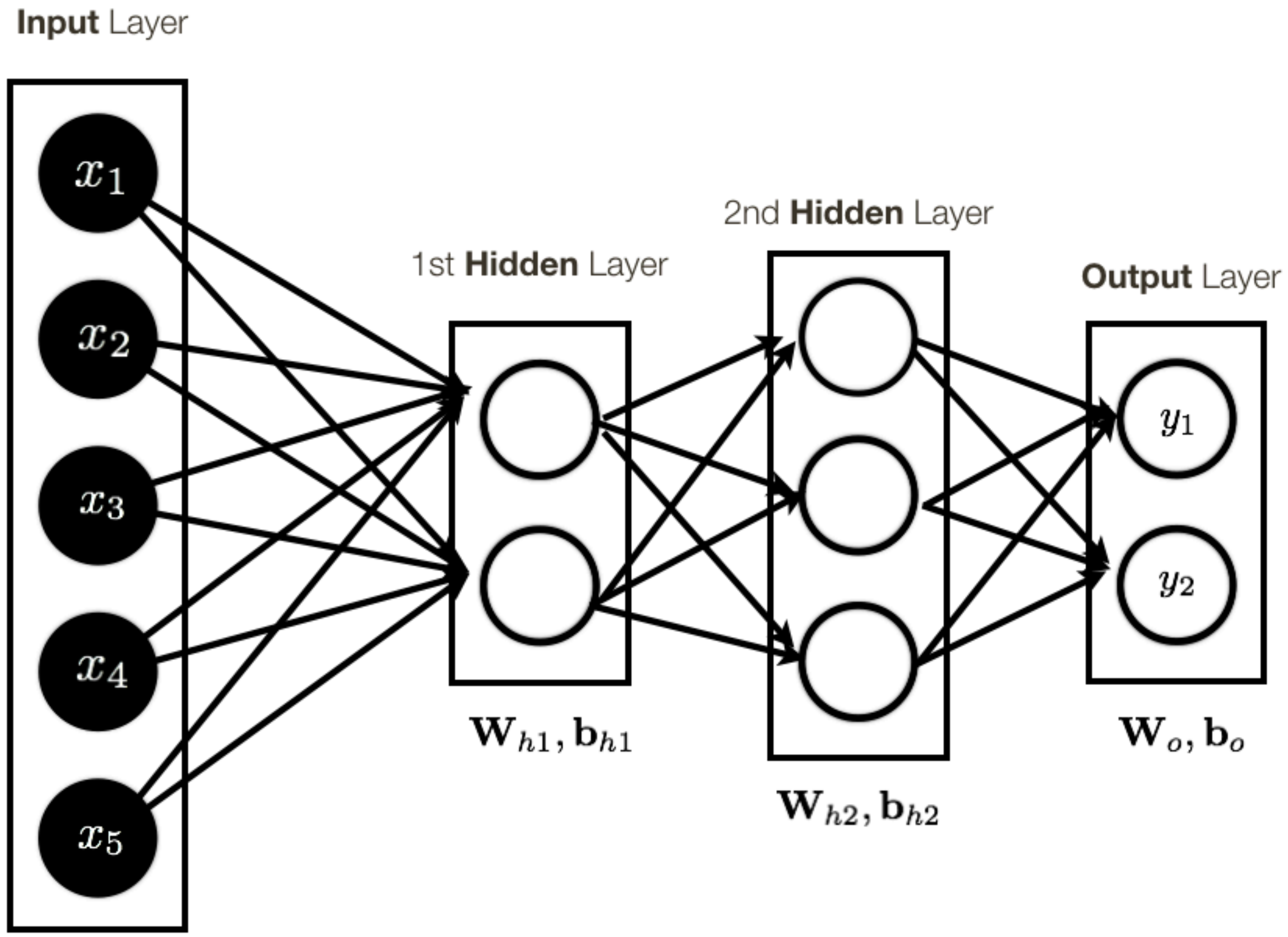


Sigmoid Activation

Short **Review** …

- Introduced the basic building block of Neural Networks **(MLP/FC) layer**

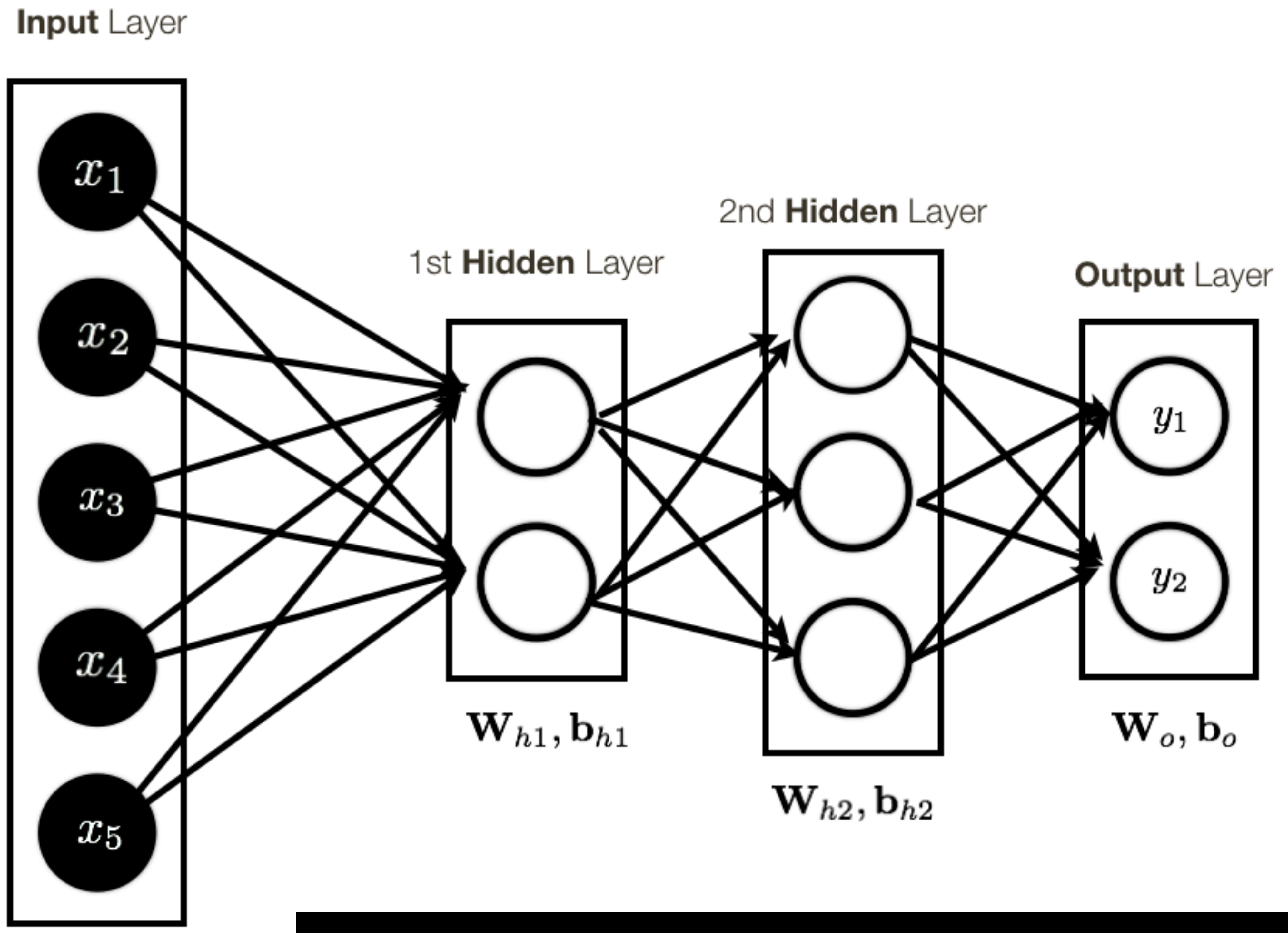
- How do we **stack these layers** up to make a Deep NN



Short **Review** …

- Introduced the basic building block of Neural Networks **(MLP/FC) layer**

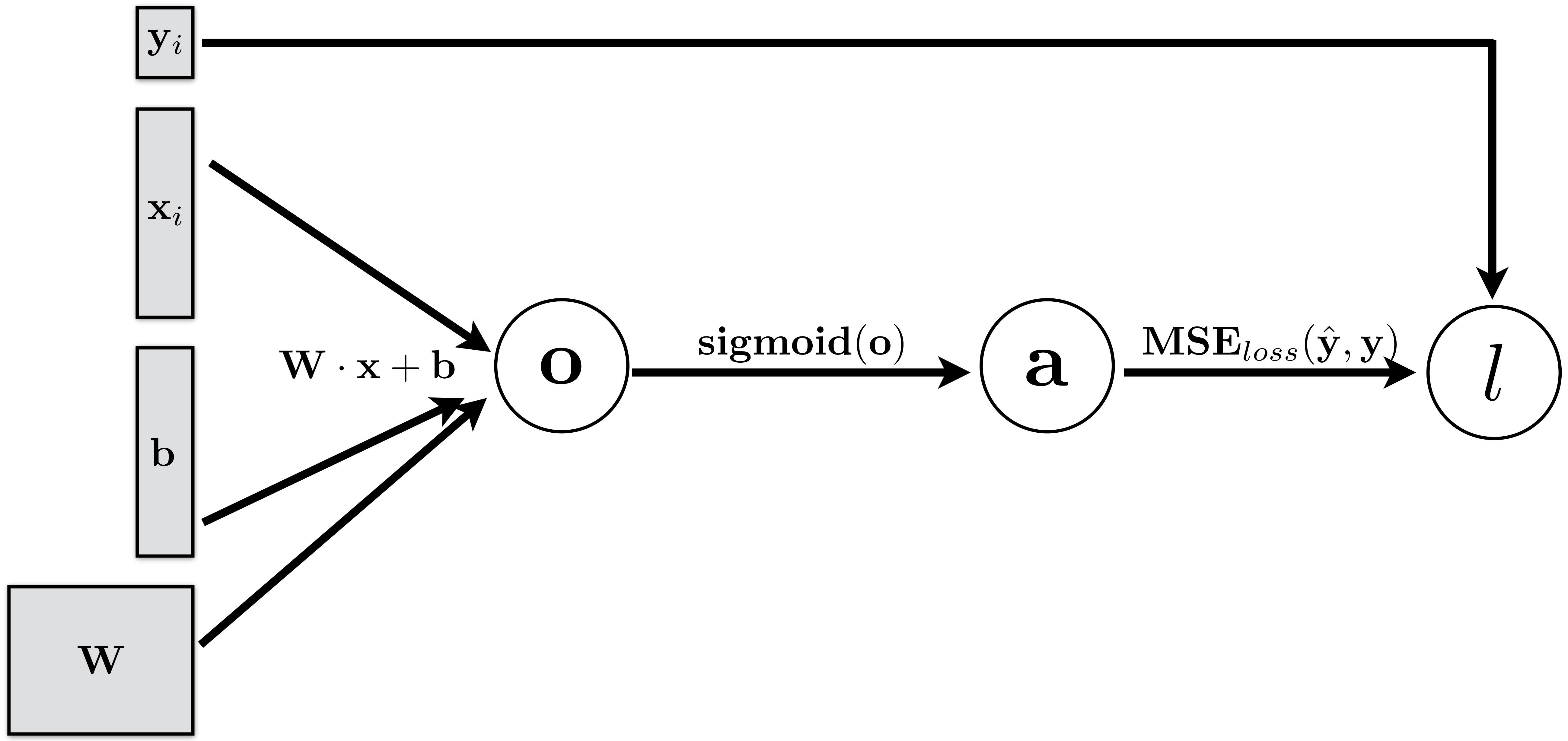
- How do we **stack these layers** up to make a Deep NN



Note: output layer often does not contain activation, or has "activation" function of a different form, to account for the specific **output** we want to produce.

Short **Review** …

- Introduced the basic building block of Neural Networks **(MLP/FC) layer**
- How do we **stack these layers** up to make a Deep NN
- Basic **NN operations** (implemented using **computational graph**)



Short **Review** …

- Introduced the basic building block of Neural Networks **(MLP/FC) layer**
- How do we **stack these layers** up to make a Deep NN
- Basic **NN operations** (implemented using **computational graph**)

Prediction / Inference

Function evaluation

(a.k.a. **ForwardProp**)

Short **Review** …

- Introduced the basic building block of Neural Networks **(MLP/FC) layer**
- How do we **stack these layers** up to make a Deep NN
- Basic **NN operations** (implemented using **computational graph**)

Prediction / Inference

Function evaluation

(a.k.a. **ForwardProp**)

Parameter **Learnings**

(Stochastic) Gradient Descent (needs **derivatives**)

Short **Review** …

- Introduced the basic building block of Neural Networks **(MLP/FC) layer**
- How do we **stack these layers** up to make a Deep NN
- Basic **NN operations** (implemented using **computational graph**)

Prediction / Inference

Function evaluation

(a.k.a. **ForwardProp**)

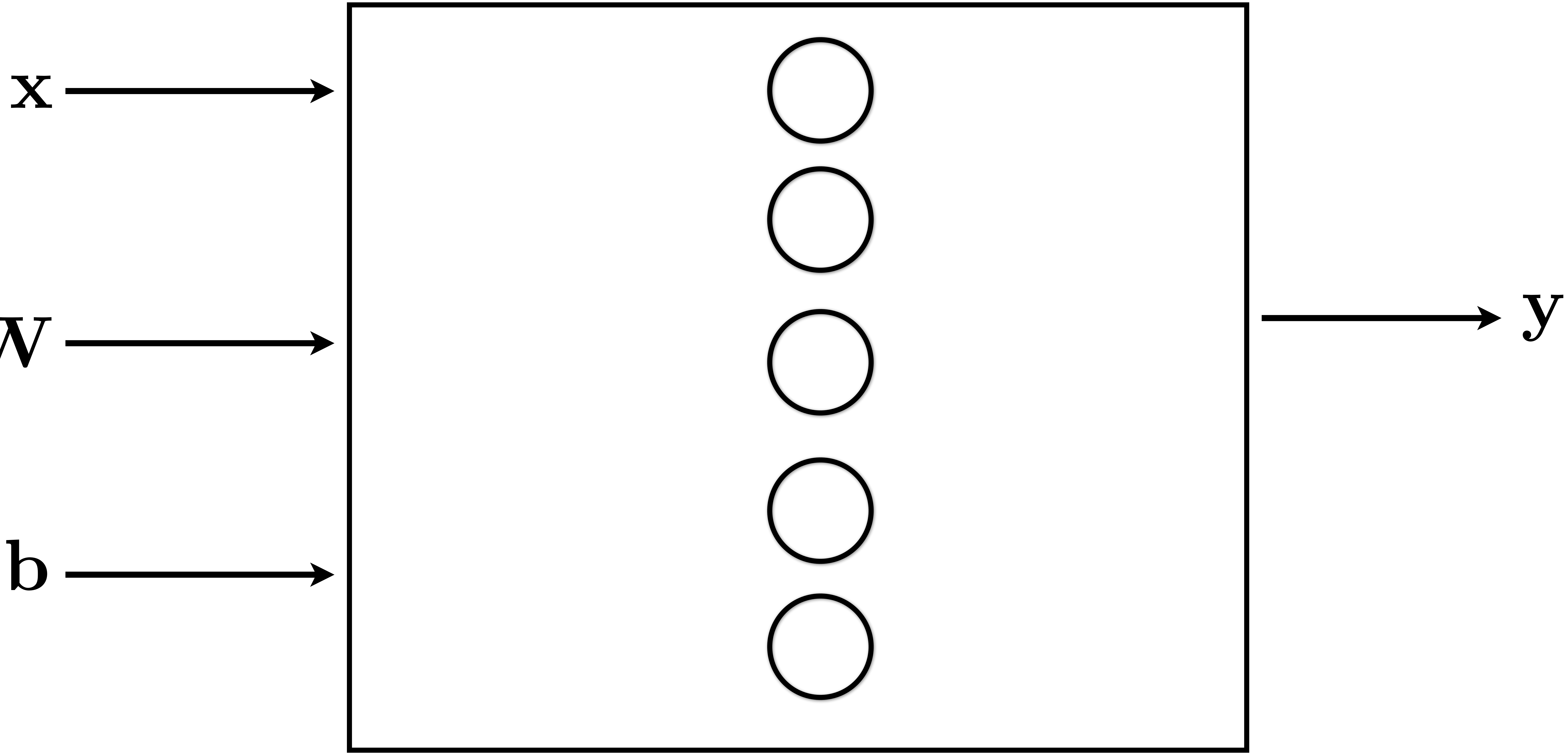
Parameter **Learnings**

(Stochastic) Gradient Descent (needs **derivatives**)

- **Numerical** differentiation (not accurate)
- **Symbolic** differential (intractable)
- AutoDiff **Forward** (computationally expensive)
- AutoDiff **Backward / BackProp**

Backpropagation Practical Issues

$$\mathbf{y} = f(\mathbf{W}, \mathbf{b}, \mathbf{x}) = \textbf{sigmoid}(\mathbf{W} \cdot \mathbf{x} + \mathbf{b})$$

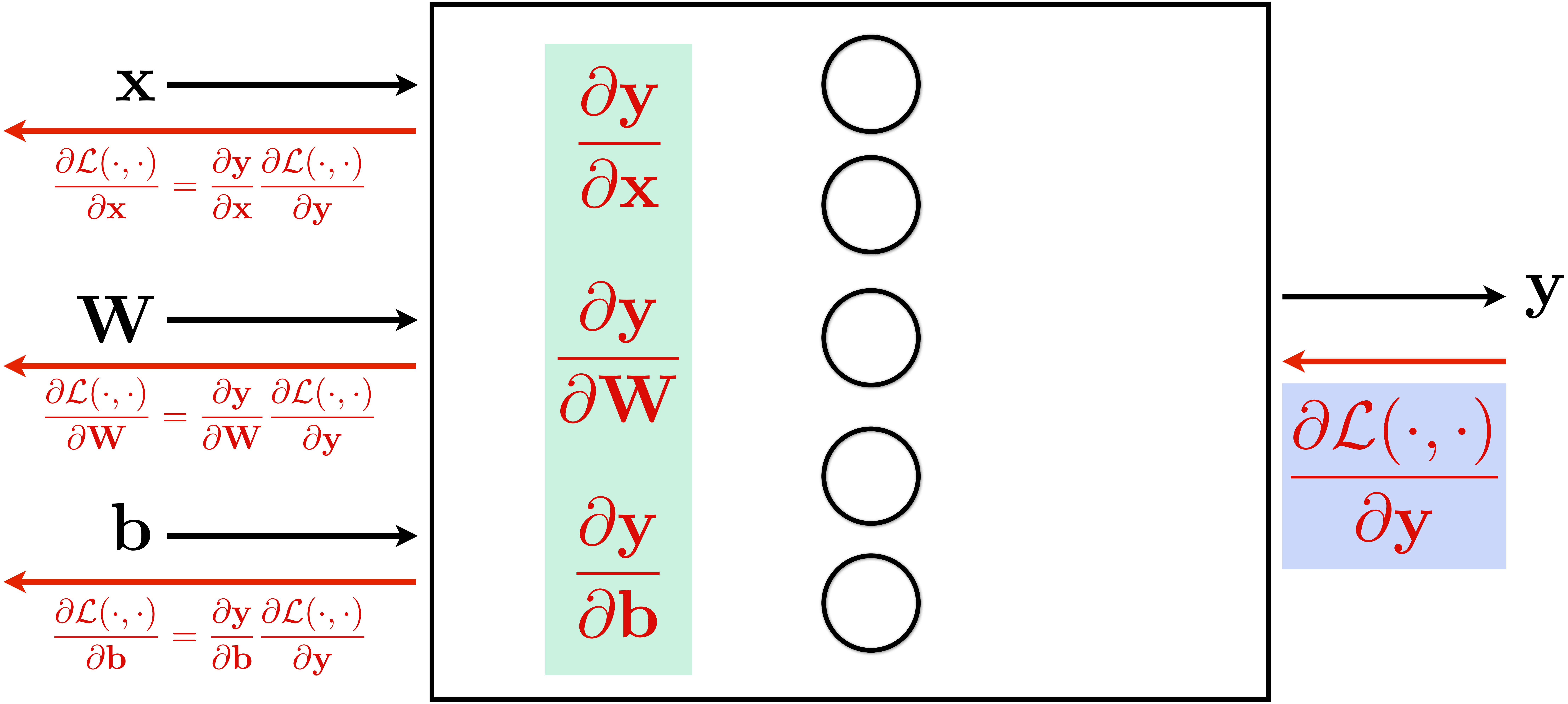


Backpropagation Practical Issues

"**local**" Jacobians
(matrix of partial derivatives, e.g. size |x| x |y|)

"**backprop**" Gradient

$$\mathbf{y} = f(\mathbf{W}, \mathbf{b}, \mathbf{x}) = \mathbf{sigmoid}(\mathbf{W} \cdot \mathbf{x} + \mathbf{b})$$



Jacobian of Sigmoid layer

$\mathbf{x}, \mathbf{y} \in \mathbb{R}^{2048}$

Element-wise sigmoid layer:

Jacobian of Sigmoid layer

$\mathbf{x}, \mathbf{y} \in \mathbb{R}^{2048}$

Element-wise sigmoid layer:

What is the dimension of **Jacobian**?

Jacobian of Sigmoid layer

$\mathbf{x}, \mathbf{y} \in \mathbb{R}^{2048}$

Element-wise sigmoid layer:

What is the dimension of **Jacobian**?

What does it look like?

Jacobian of Sigmoid layer

$\mathbf{x}, \mathbf{y} \in \mathbb{R}^{2048}$

Element-wise sigmoid layer:

What is the dimension of **Jacobian**?

What does it look like?

If we are working with a mini batch of 100 inputs-output pairs, technically Jacobian is a matrix 204,800 x 204,800

Jacobian of Sigmoid layer

In practice this can be made a **LOT** more efficient

- Gradients can be sparse, so can be stored efficiently

- Computations per samples (e.g., in a mini-batch) are independent => can be done in parallel and simply accumulated.

If we are working with a mini batch of 100 inputs-output pairs, technically Jacobian is a matrix 204,800 x 204,800

Jacobian of Sigmoid layer

In practice this can be made a **LOT** more efficient

- Gradients can be sparse, so can be stored efficiently
- Computations per samples (e.g., in a mini-batch) are independent => can be done in parallel and simply accumulated.

What is the dimension of **Jacobian**?

If we are working with a mini batch of 100 inputs-output pairs, technically Jacobian is a matrix 204,800 x 204,800

Jacobian of Sigmoid layer

In practice this can be made a **LOT** more efficient

- Gradients can be sparse, so can be stored efficiently
- Computations per samples (e.g., in a mini-batch) are independent => can be done in parallel and simply accumulated.

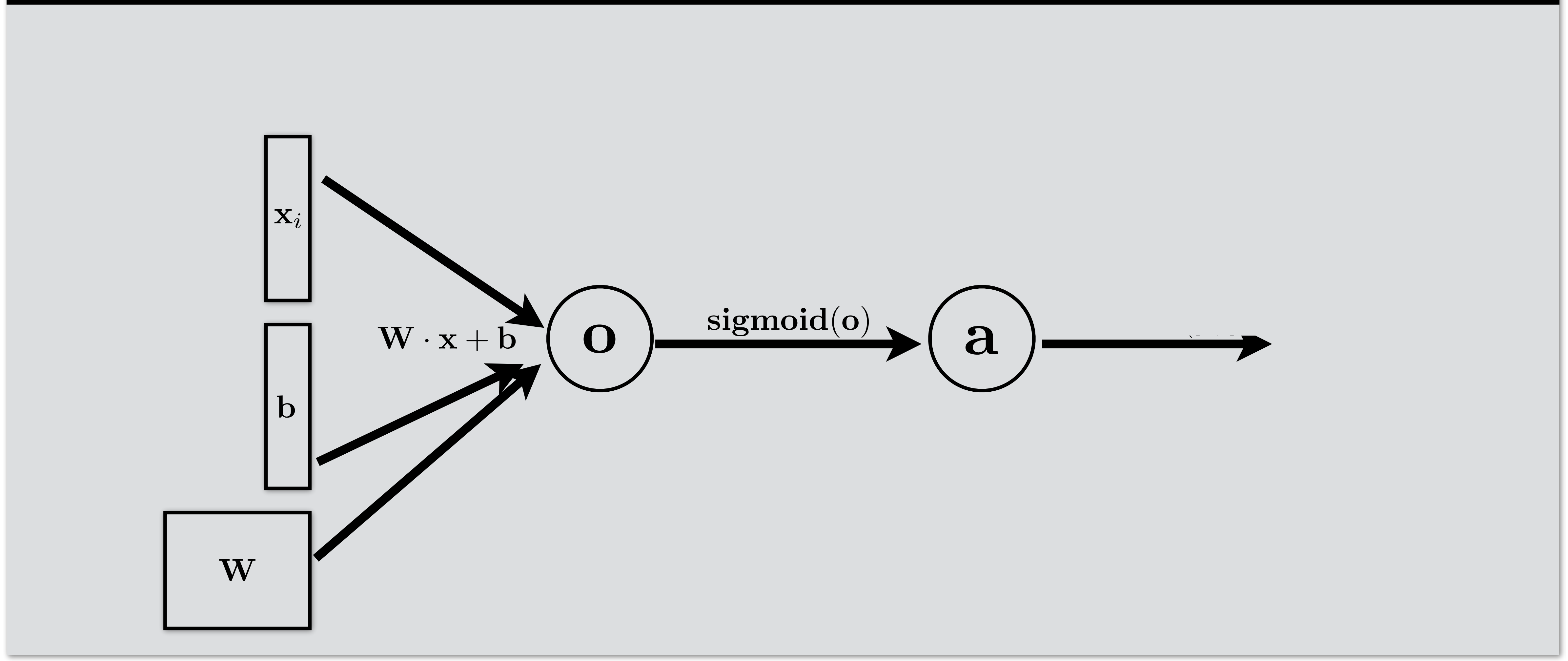
What is the dimension of **Jacobian**?

What does it look like?

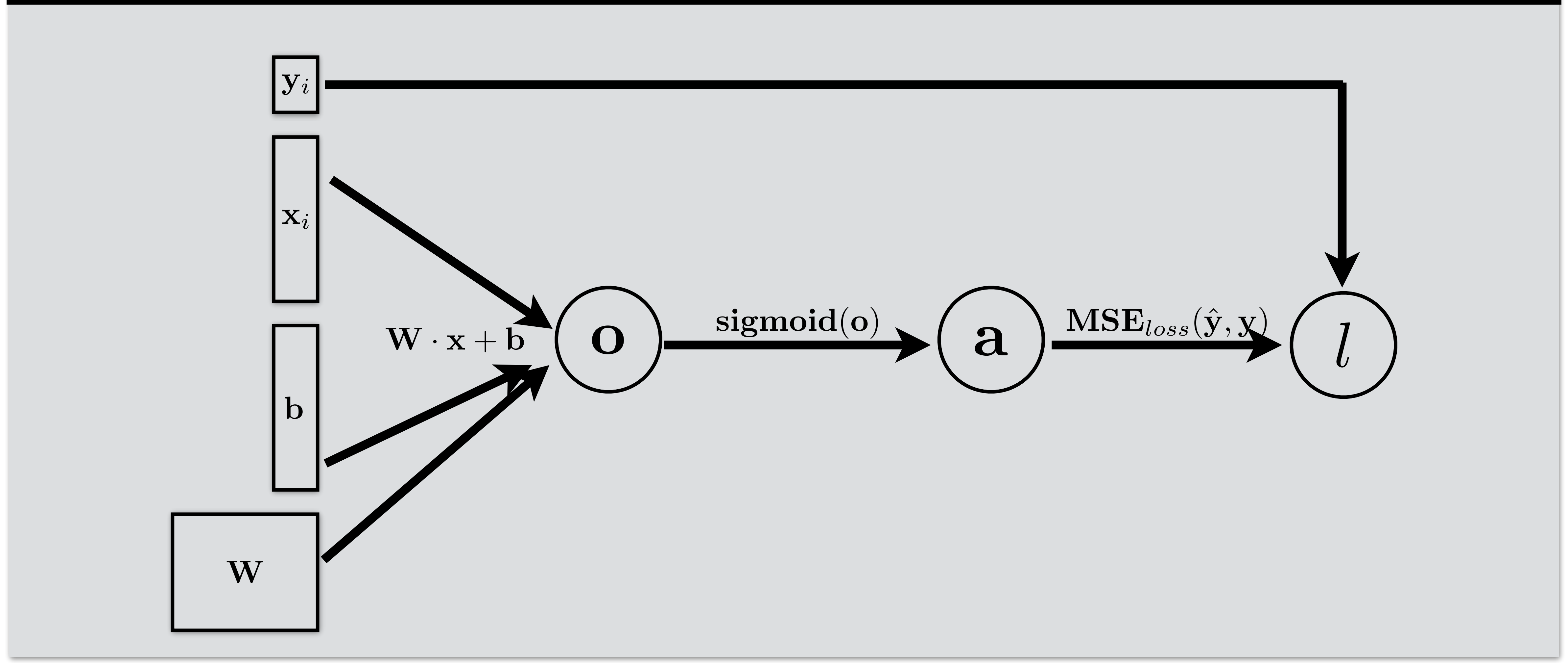
If we are working with a mini batch of 100 inputs-output pairs, technically Jacobian is a matrix 204,800 x 204,800

Short **Review** …

Prediction / Inference



Parameter **Learnings**



Short **Review** …

- Introduced the basic building block of Neural Networks **(MLP/FC) layer**

- How do we **stack these layers** up to make a Deep NN

- Basic **NN operations** (implemented using **computational graph**)

Prediction / Inference

Function evaluation

(a.k.a. **ForwardProp**)

Parameter **Learnings**

(Stochastic) Gradient Descent (needs **derivatives**)

- **Numerical** differentiation (not accurate)
- **Symbolic** differential (intractable)
- AutoDiff **Forward** (computationally expensive)
- AutoDiff **Backward / BackProp**

- Different **activation functions** and saturation problem

Activation Function: Sigmoid

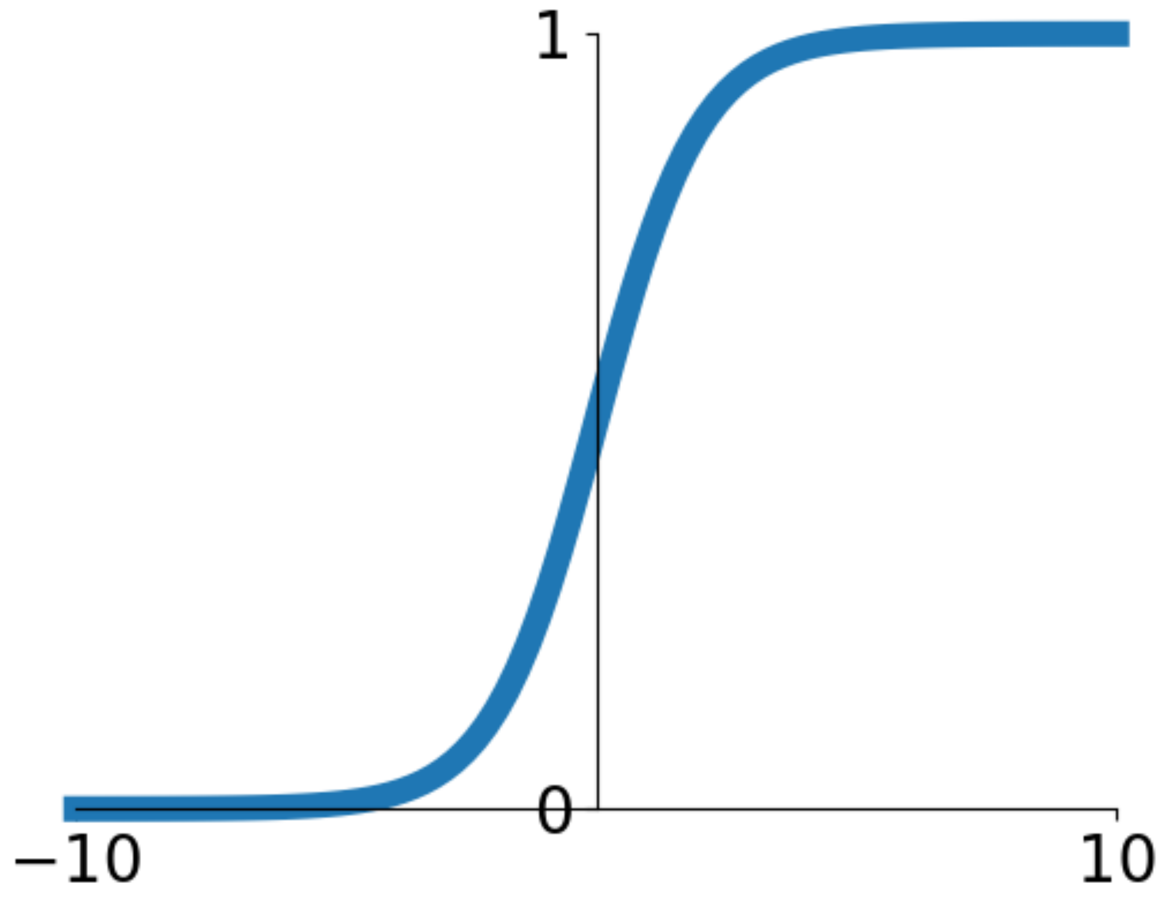
$$a(x) = \textbf{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

Pros:

- Squishes everything in the range [0,1]
- Can be interpreted as "probability"
- Has well defined gradient everywhere

Cons:

- Saturated neurons "kill" the gradients
- Non-zero centered
- Could be expensive to compute



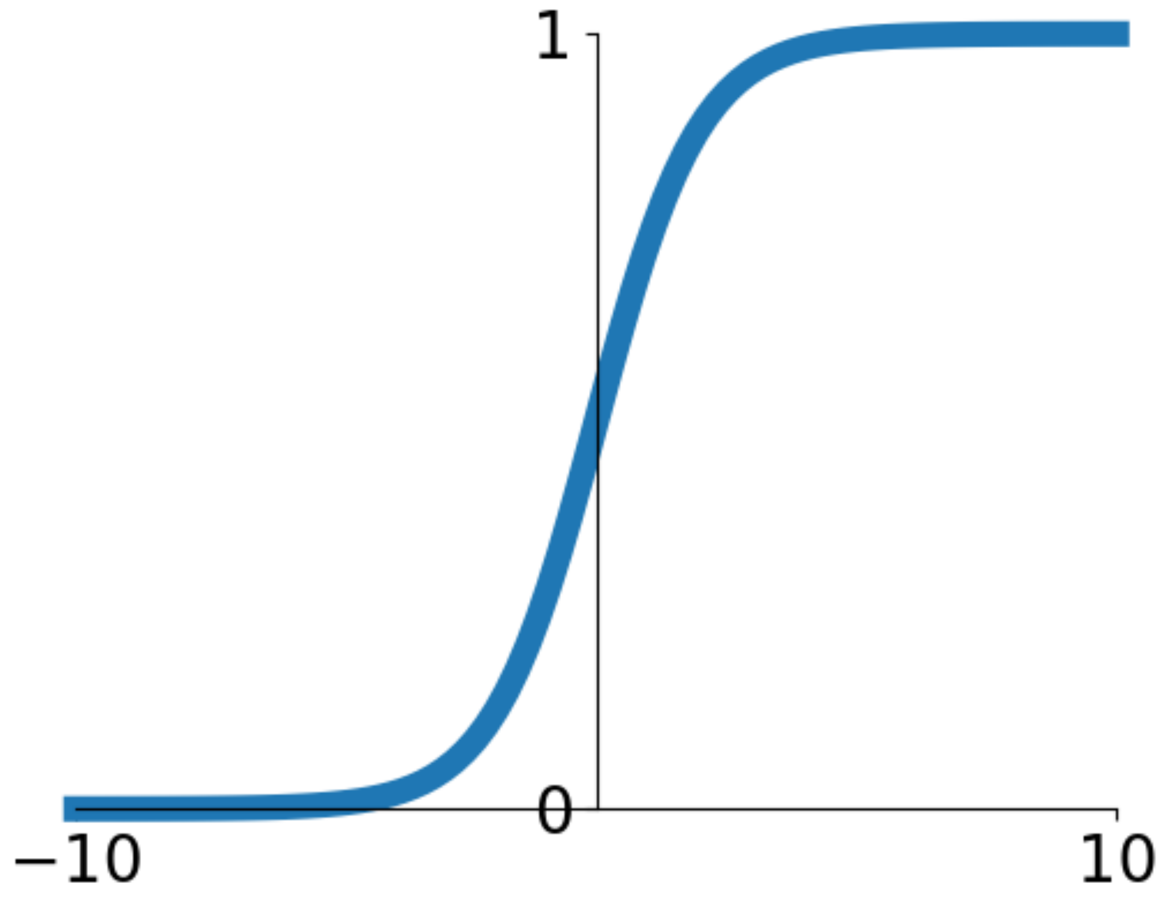
Sigmoid Activation

* slide adopted from Li, Karpathy, Johnson's **CS231n at Stanford**

Activation Function: Sigmoid

Sigmoid Gate

$$a(x) = \mathbf{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$



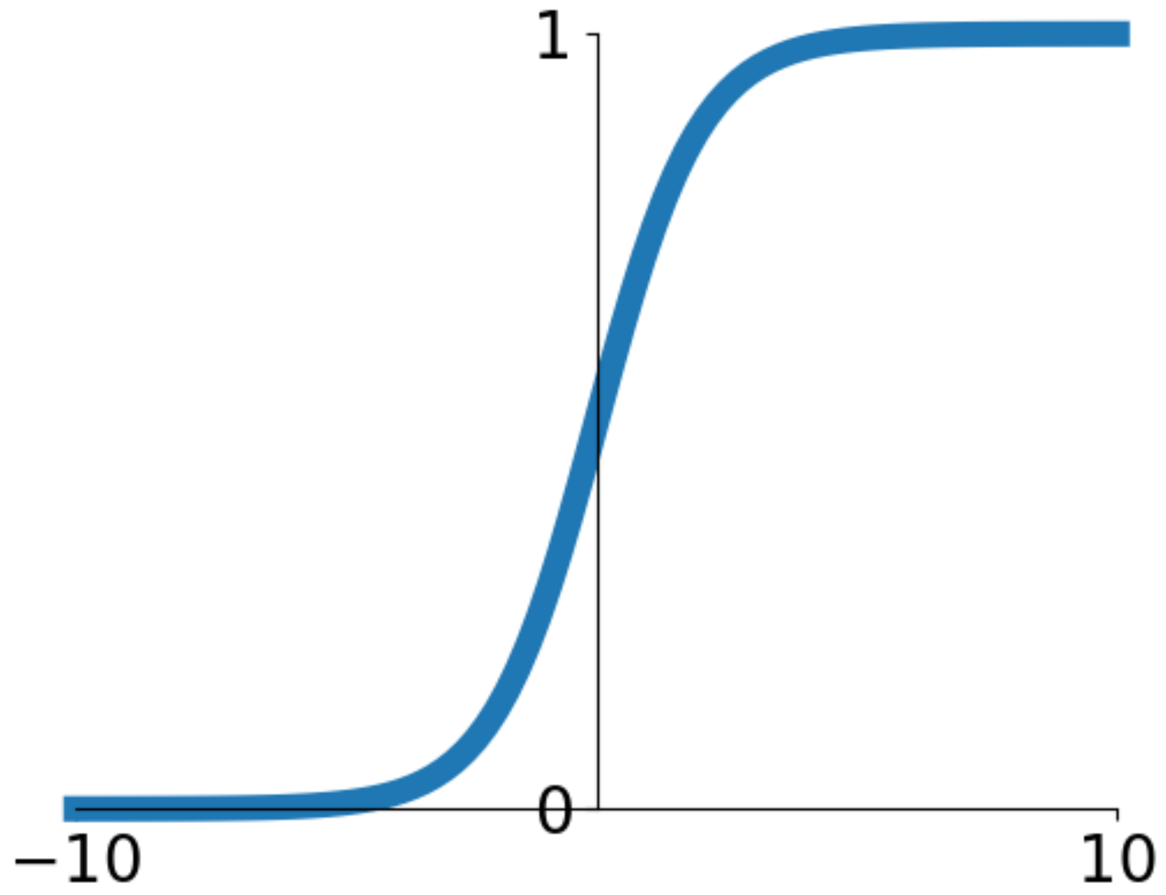
Sigmoid Activation

Cons:

- Saturated neurons **“kill” the gradients**
- Non-zero centered
- Could be expensive to compute

Activation Function: Sigmoid

$$a(x) = \textbf{sigmoid}(x) = \frac{1}{1+e^{-x}}$$



Sigmoid Activation

Cons:

- Saturated neurons **“kill” the gradients**
- Non-zero centered
- Could be expensive to compute

Activation Function: Sigmoid

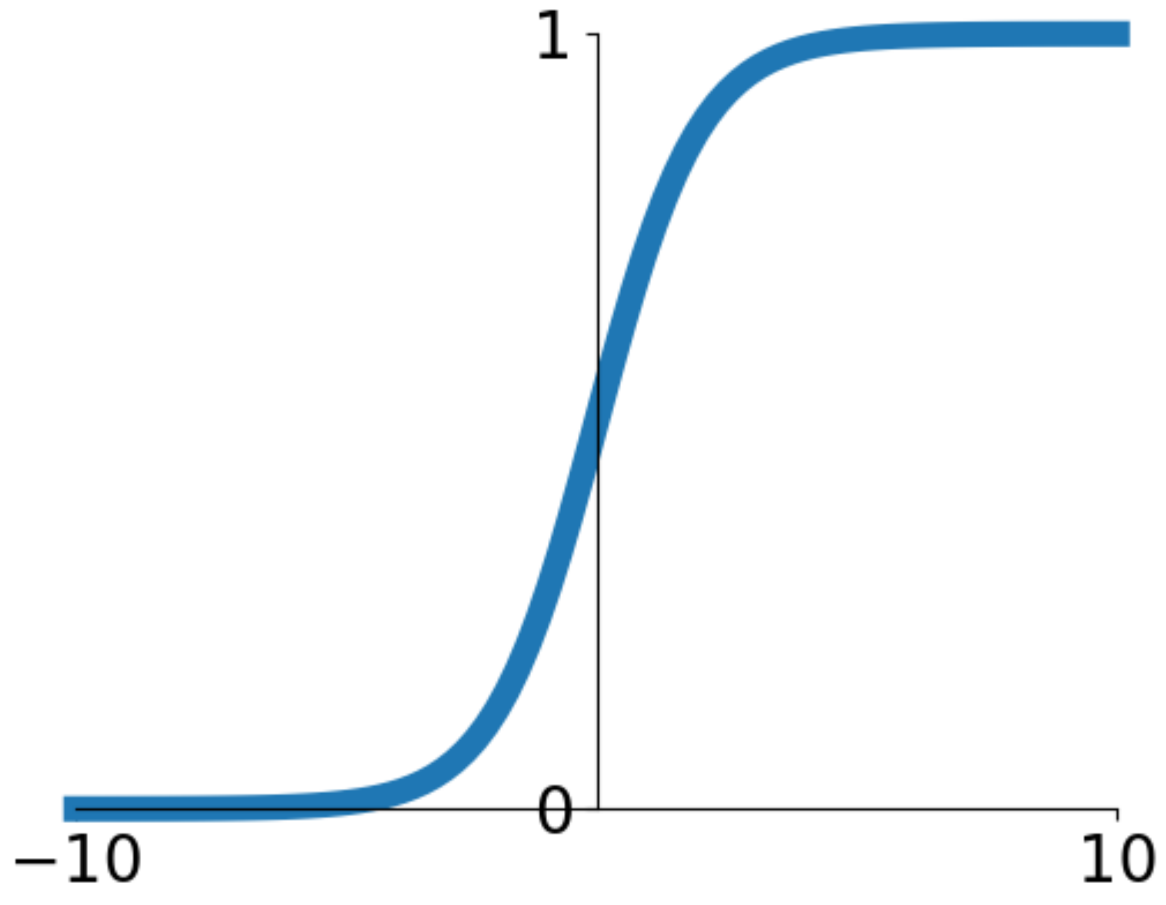
$$a = \textbf{sigmoid}(x) = \frac{1}{1+e^{-x}}$$

x

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \, \textbf{sigmoid}(x)}{\partial x} \frac{\partial \mathcal{L}}{\partial a}$$

$$\frac{\partial \mathcal{L}}{\partial a}$$

$$a(x) = \textbf{sigmoid}(x) = \frac{1}{1+e^{-x}}$$

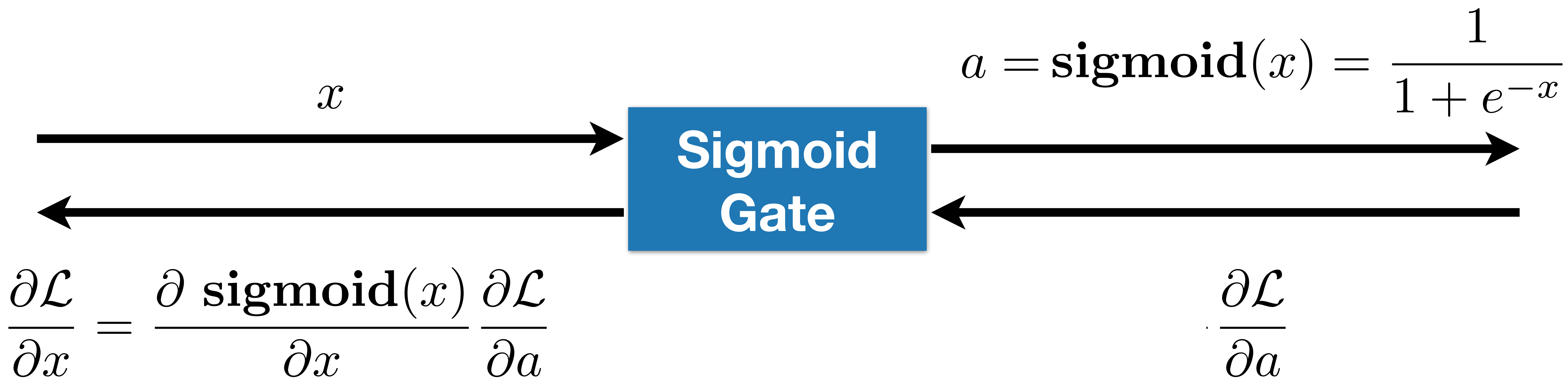


Sigmoid Activation

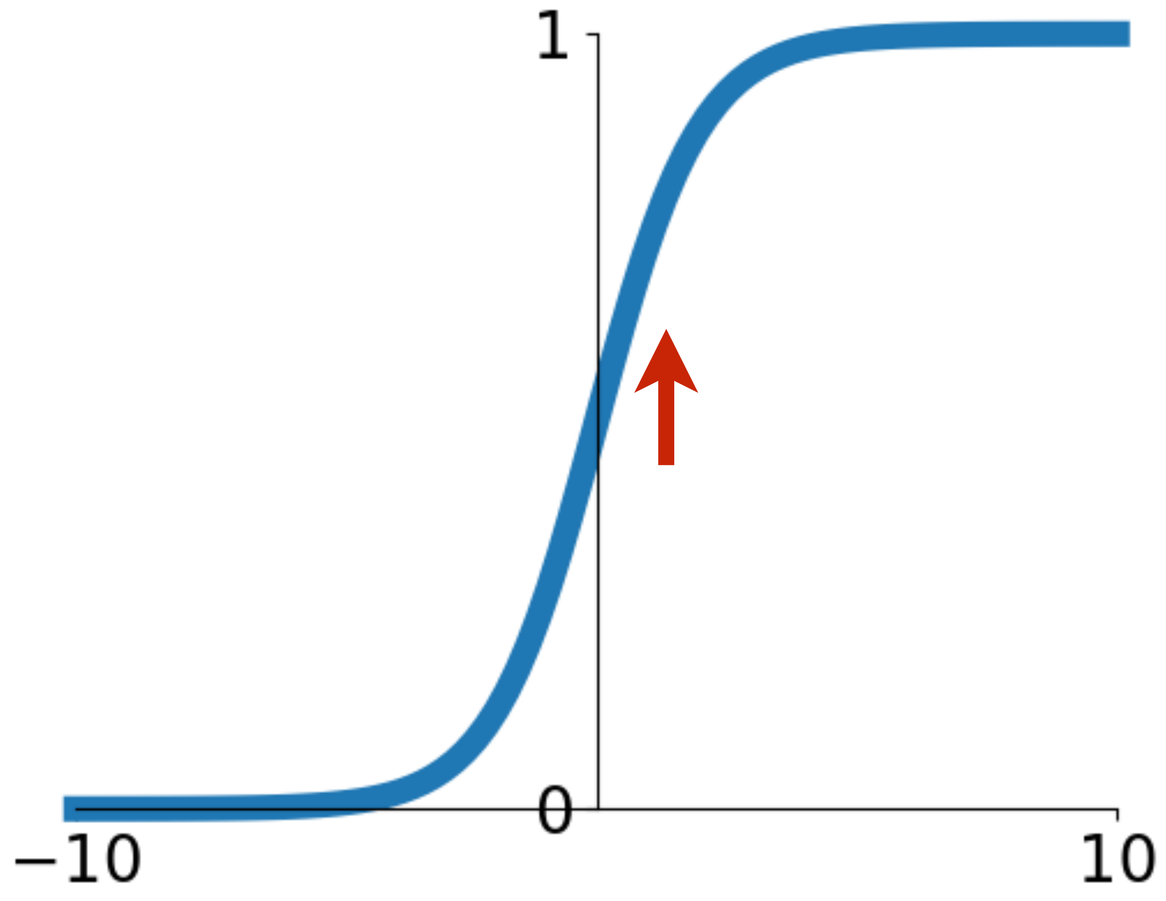
Cons:

- Saturated neurons **“kill” the gradients**
- Non-zero centered
- Could be expensive to compute

Activation Function: Sigmoid



$$a(x) = \mathbf{sigmoid}(x) = \frac{1}{1+e^{-x}}$$



Sigmoid Activation

Cons:

- Saturated neurons **"kill" the gradients**
- Non-zero centered
- Could be expensive to compute

* slide adopted from Li, Karpathy, Johnson's **CS231n at Stanford**

Activation Function: Sigmoid

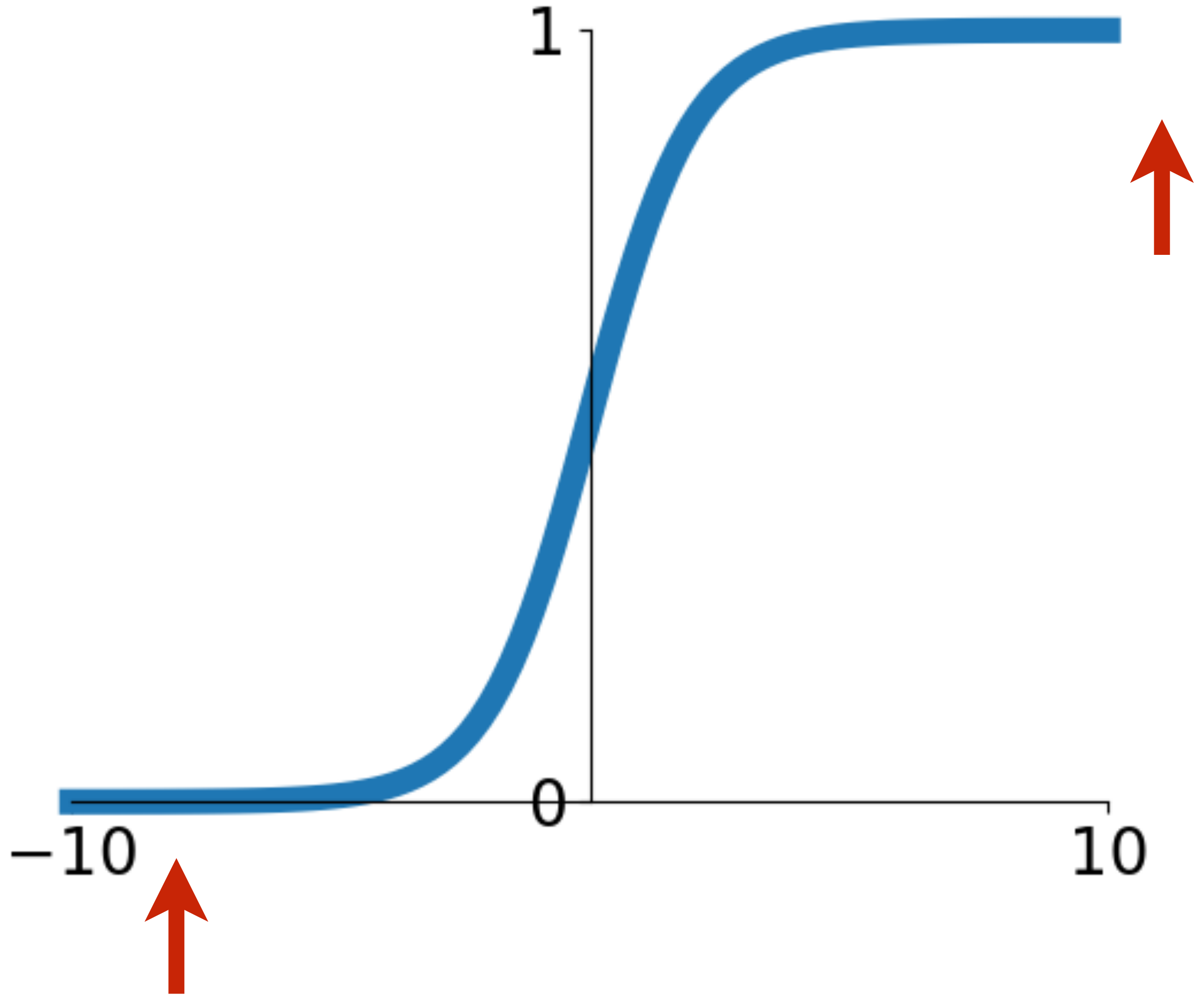
$$x$$

$$a = \textbf{sigmoid}(x) = \frac{1}{1+e^{-x}}$$

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial\, \textbf{sigmoid}(x)}{\partial x}\frac{\partial \mathcal{L}}{\partial a}$$

$$\frac{\partial \mathcal{L}}{\partial a}$$

$$a(x) = \textbf{sigmoid}(x) = \frac{1}{1+e^{-x}}$$



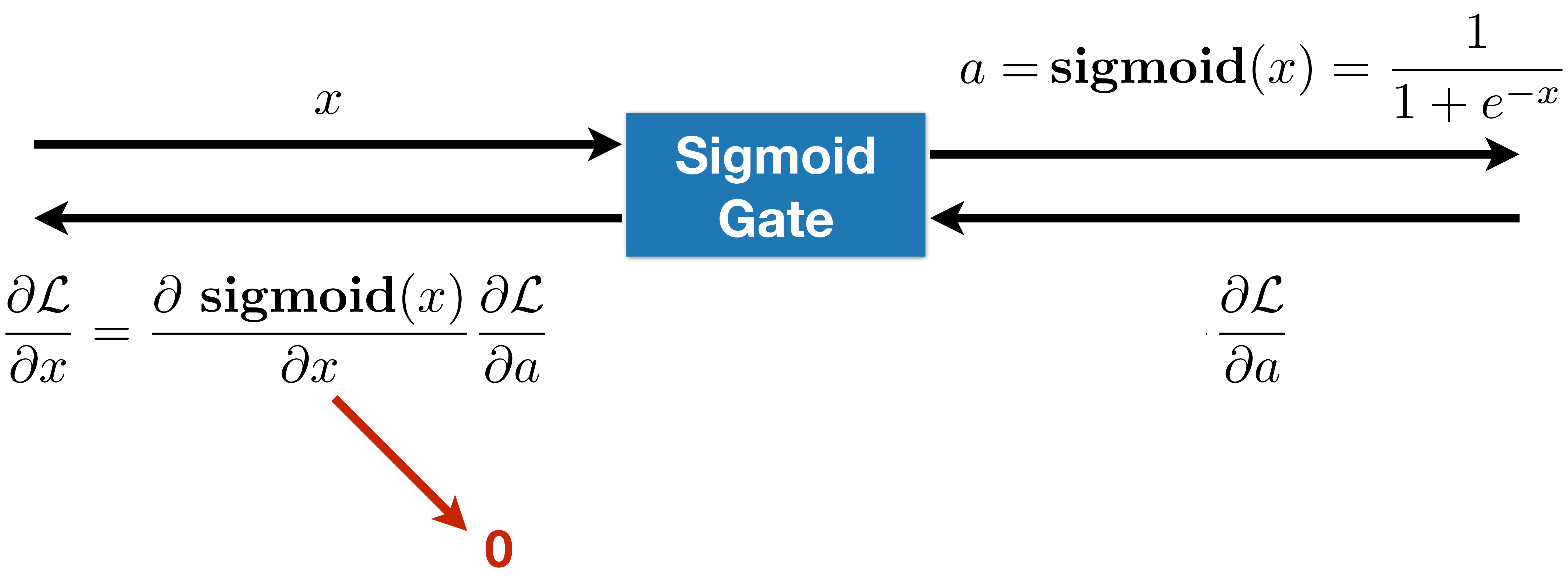
Sigmoid Activation

Cons:

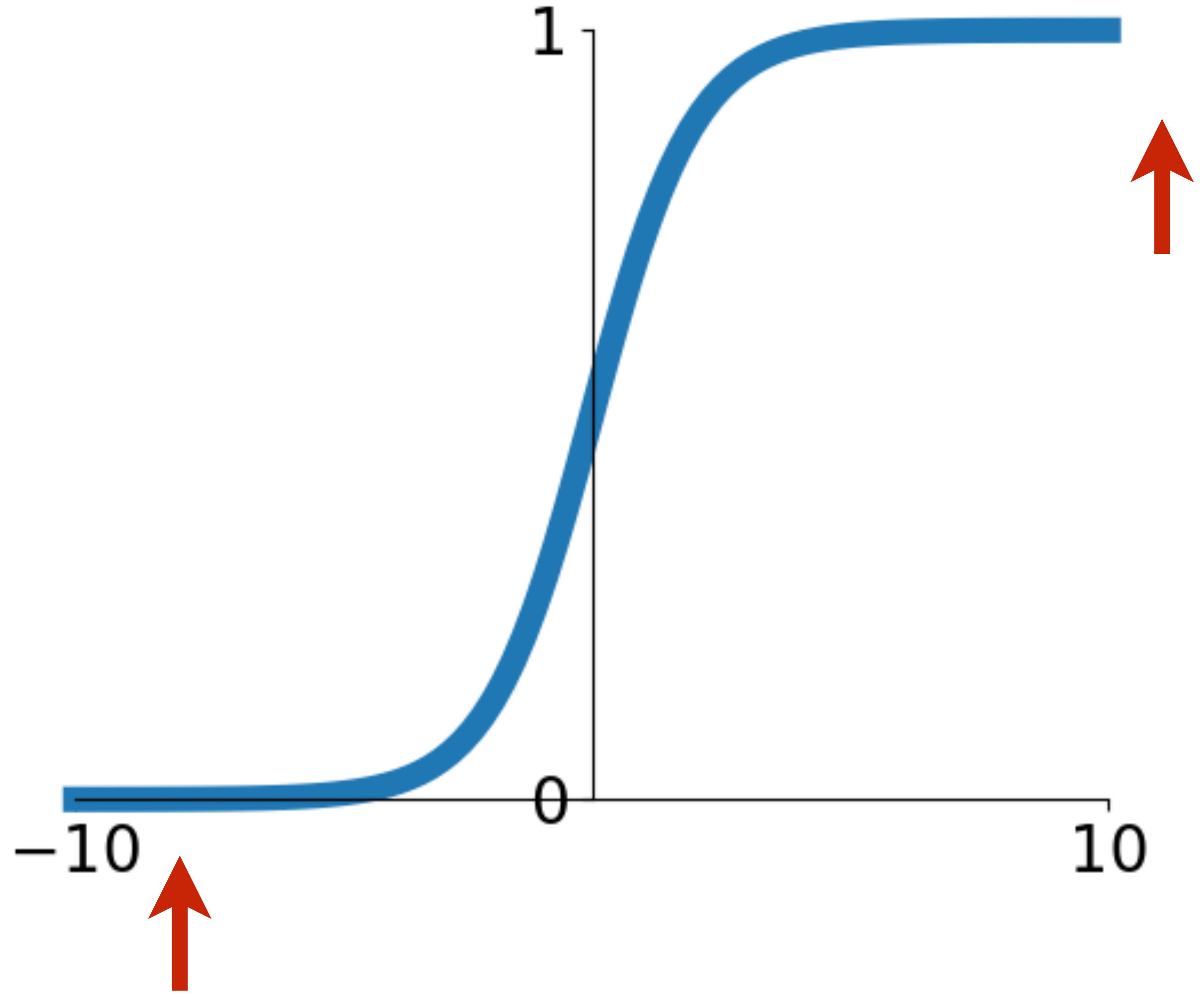
- Saturated neurons **“kill” the gradients**
- Non-zero centered
- Could be expensive to compute

* slide adopted from Li, Karpathy, Johnson’s **CS231n at Stanford**

Activation Function: Sigmoid



$$a(x) = \mathbf{sigmoid}(x) = \frac{1}{1+e^{-x}}$$



Sigmoid Activation

Cons:

- Saturated neurons **“kill” the gradients**
- Non-zero centered
- Could be expensive to compute

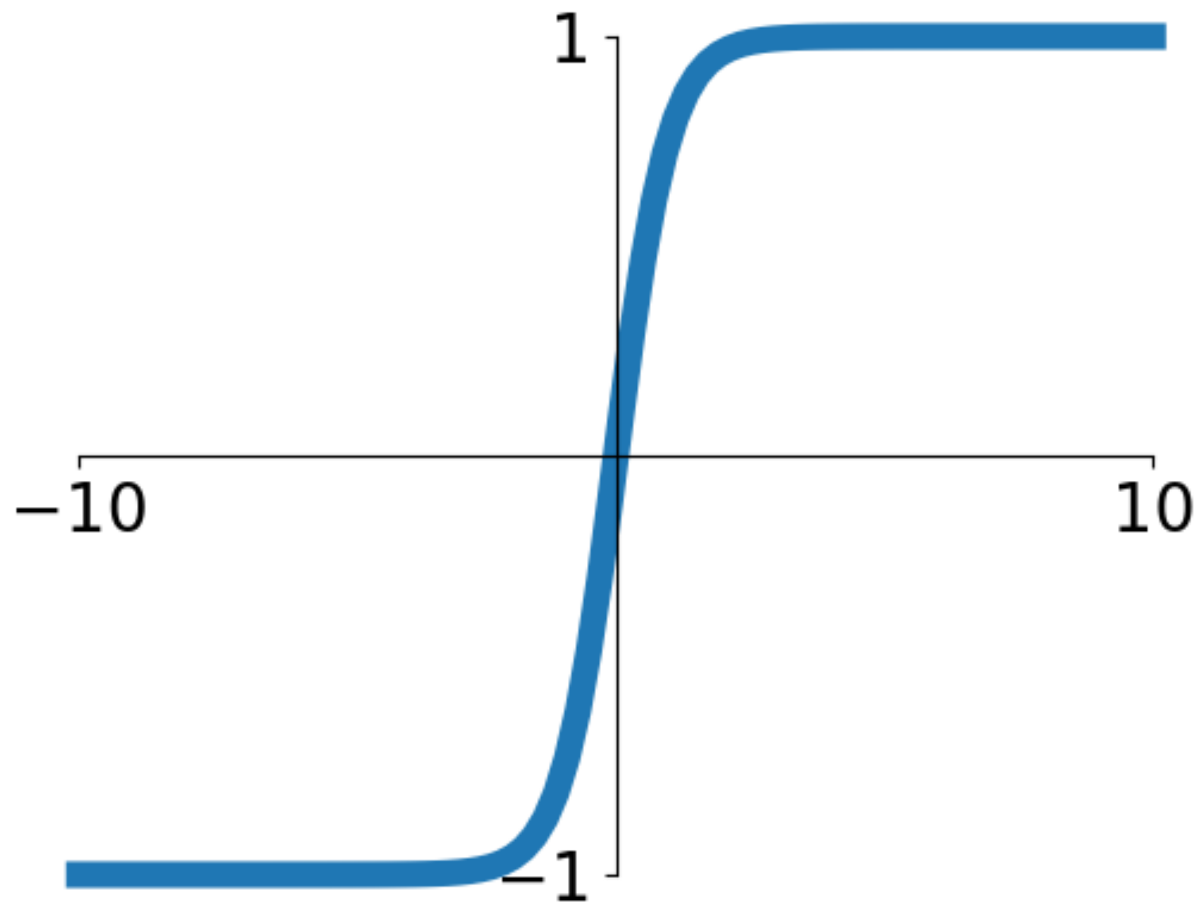
Activation Function: Tanh

$$a(x) = \mathbf{tanh}(x) = 2 \cdot \mathbf{sigmoid}(2x) - 1$$

$$a(x) = \mathbf{tanh}(x) = \frac{2}{1 + e^{-2x}} - 1$$

Pros:

- Squishes everything in the range [-1,1]
- Centered around zero
- Has well defined gradient everywhere



Tanh Activation

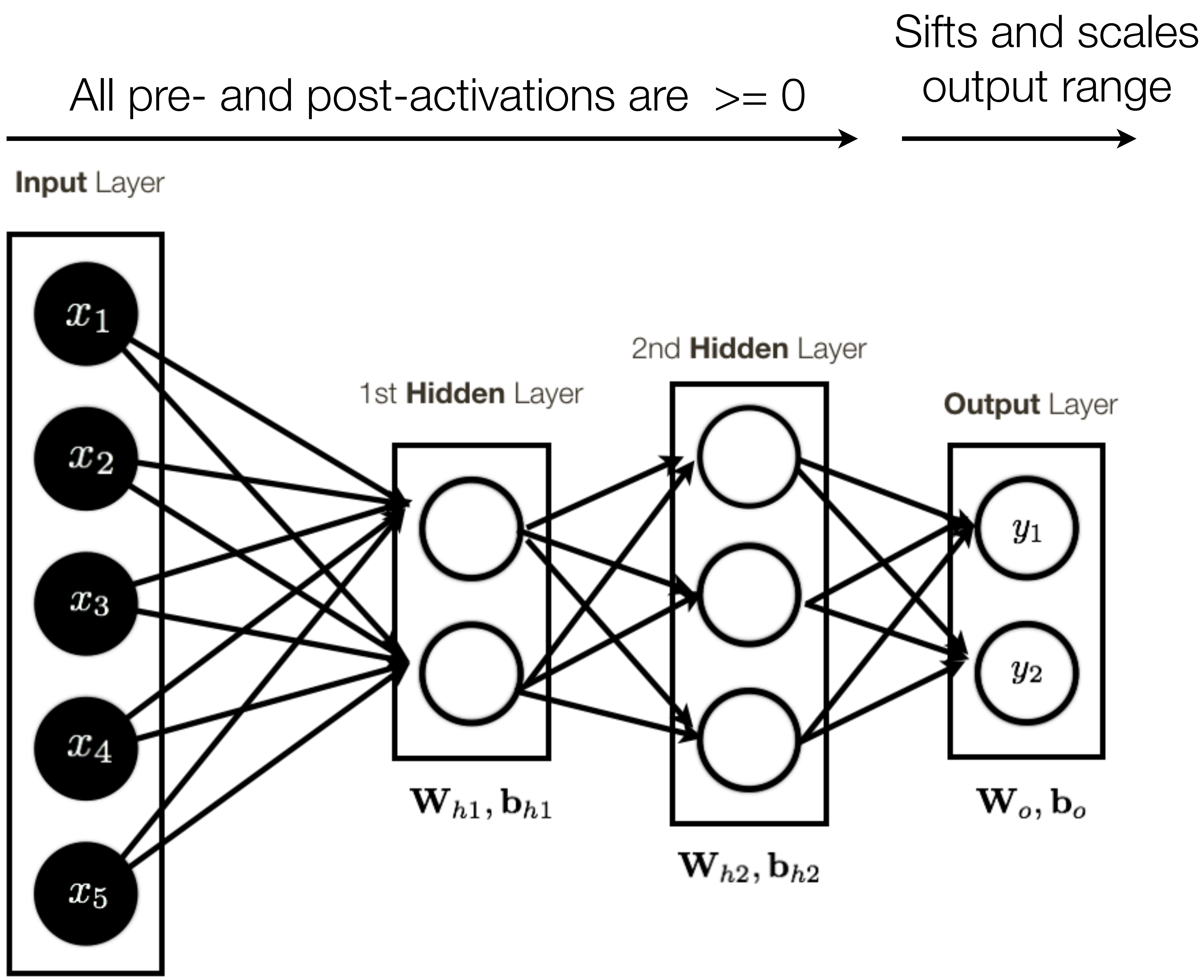
Cons:

- Saturated neurons "kill" the gradients
- Could be expensive to compute

* slide adopted from Li, Karpathy, Johnson's **CS231n at Stanford**

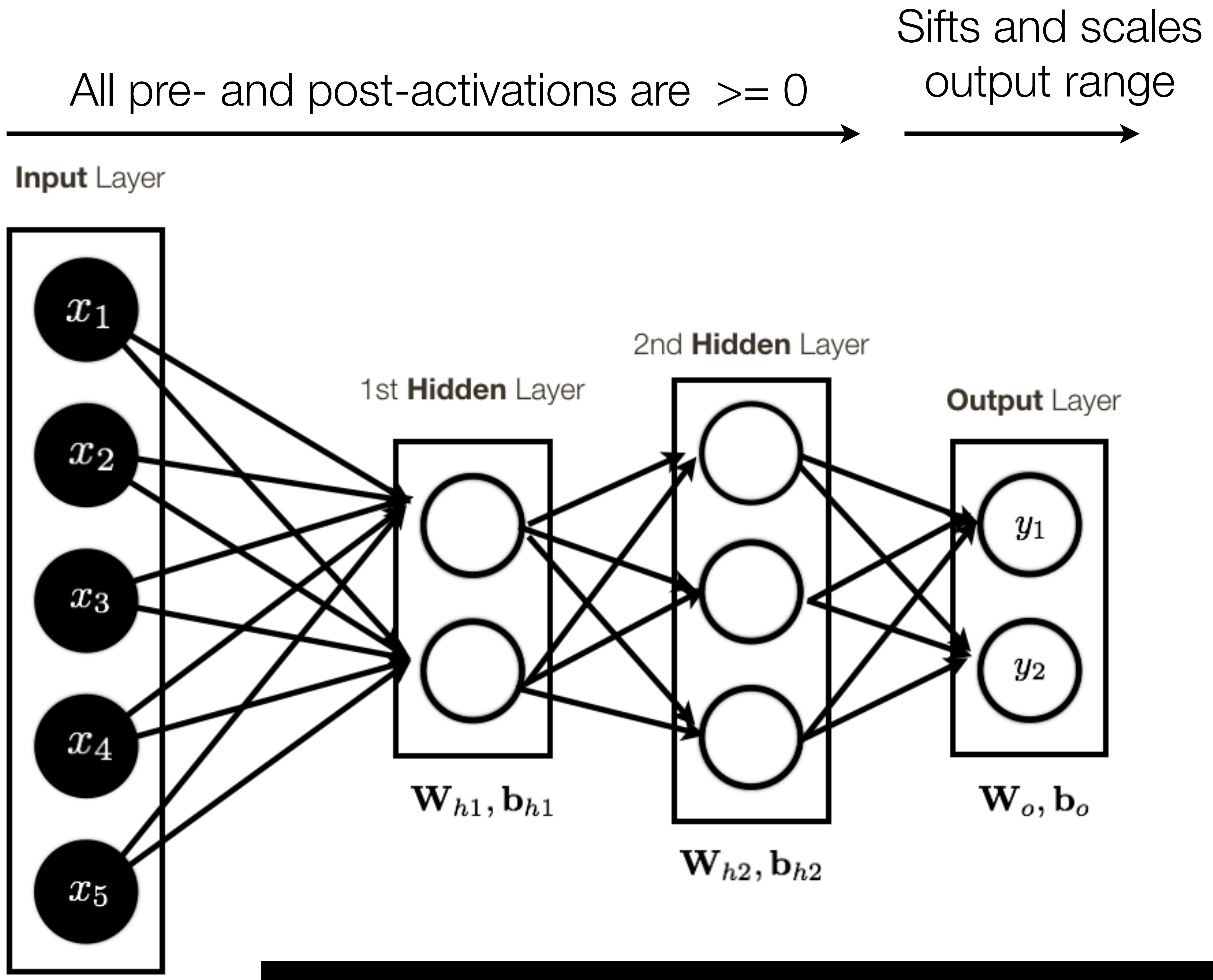
Why **zero-centering** may be good?

Consider a (regression) problem where the predictions can be positive and negative (e.g., cash flow -> you can be loosing money or making money)



Why **zero-centering** may be good?

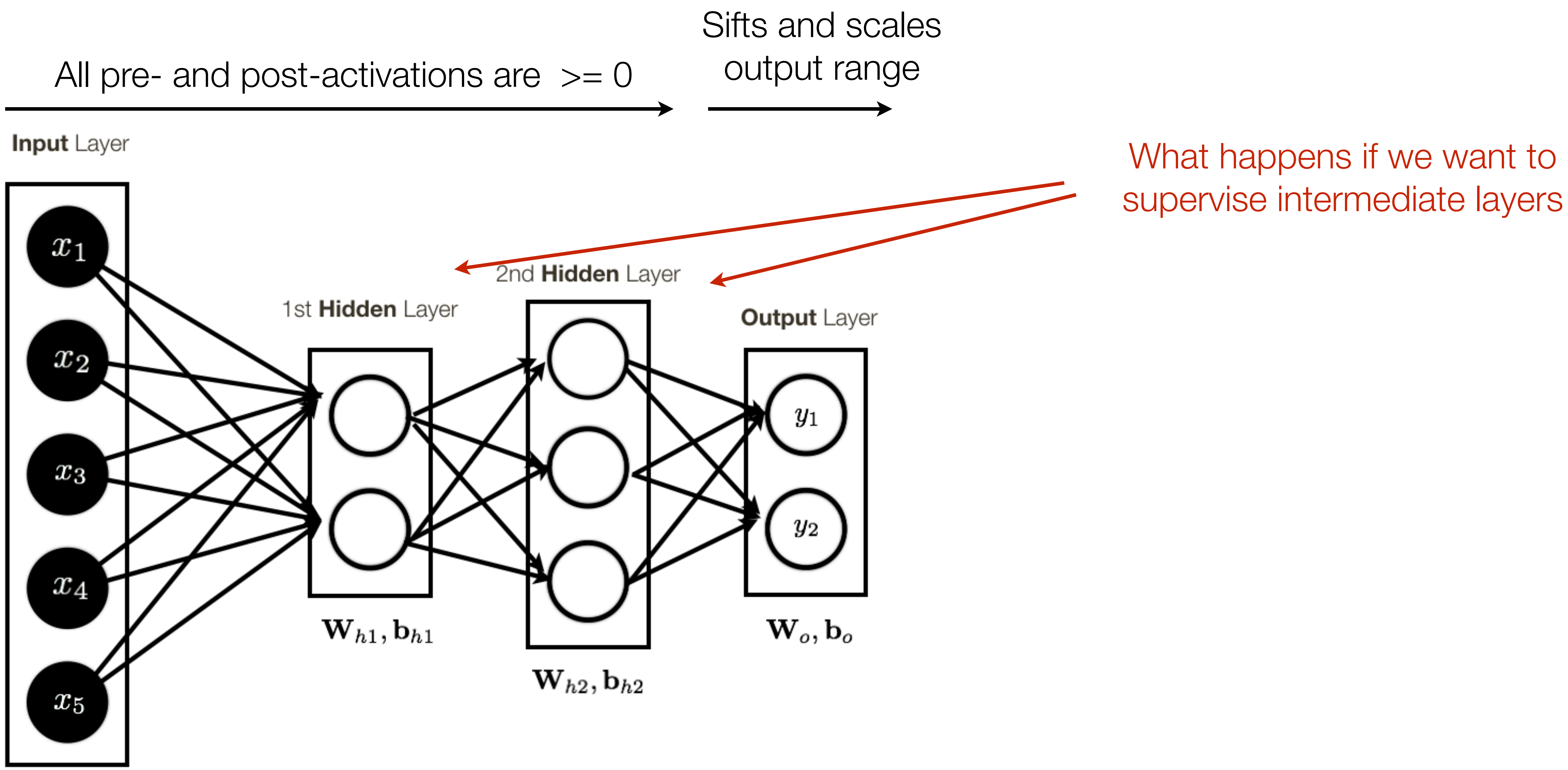
Consider a (regression) problem where the predictions can be positive and negative (e.g., cash flow -> you can be loosing money or making money)



Note: output layer often does not contain activation, or has “activation” function of a different form, to account for the specific **output** we want to produce.

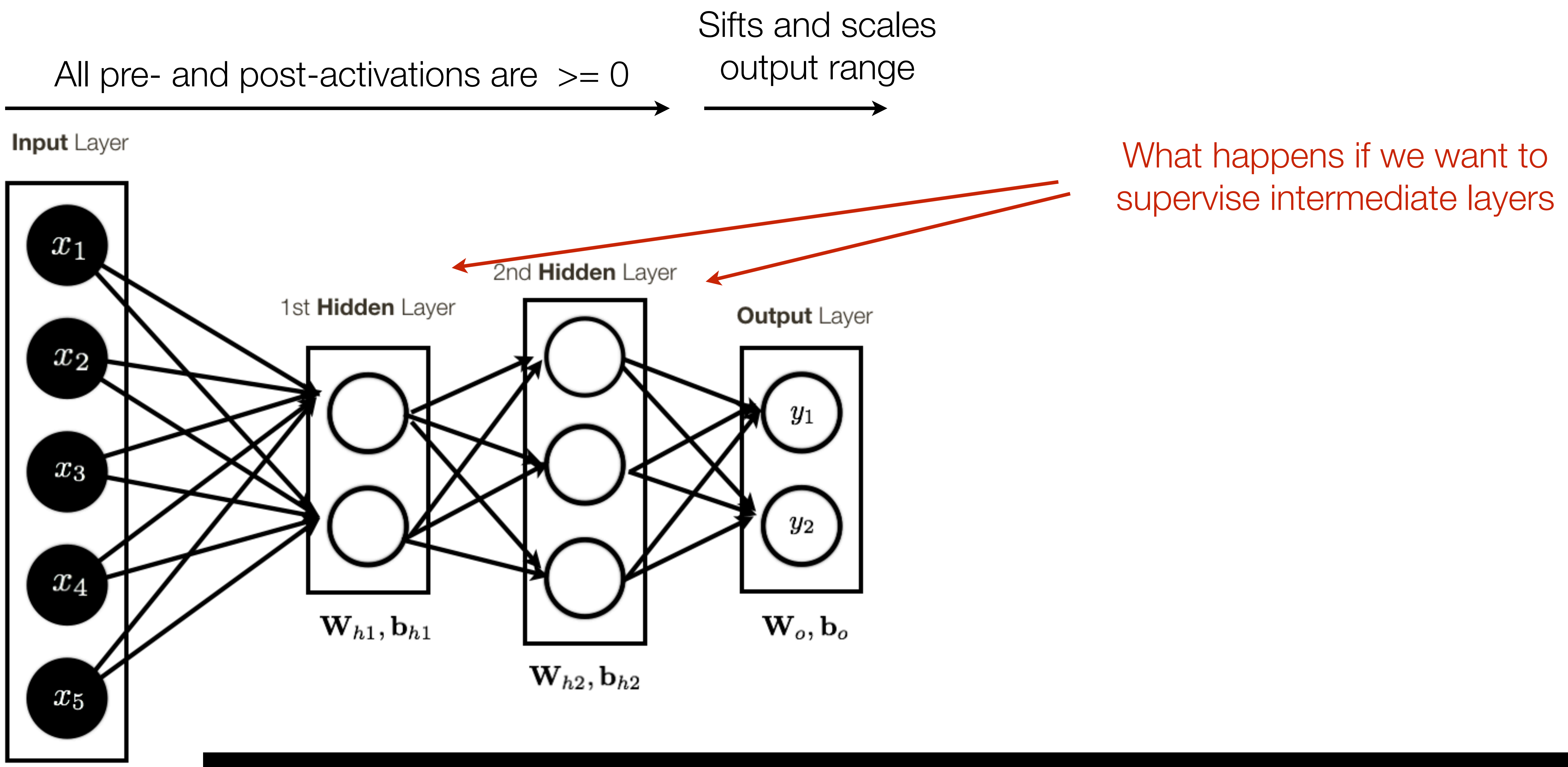
Why **zero-centering** may be good?

Consider a (regression) problem where the predictions can be positive and negative (e.g., cash flow -> you can be loosing money or making money)



Why **zero-centering** may be good?

Consider a (regression) problem where the predictions can be positive and negative (e.g., cash flow -> you can be loosing money or making money)



Note: output layer often does not contain activation, or has "activation" function of a different form, to account for the specific **output** we want to produce.

Activation Function: Rectified Linear Unit (ReLU)

$$a(x) = max(0, x)$$

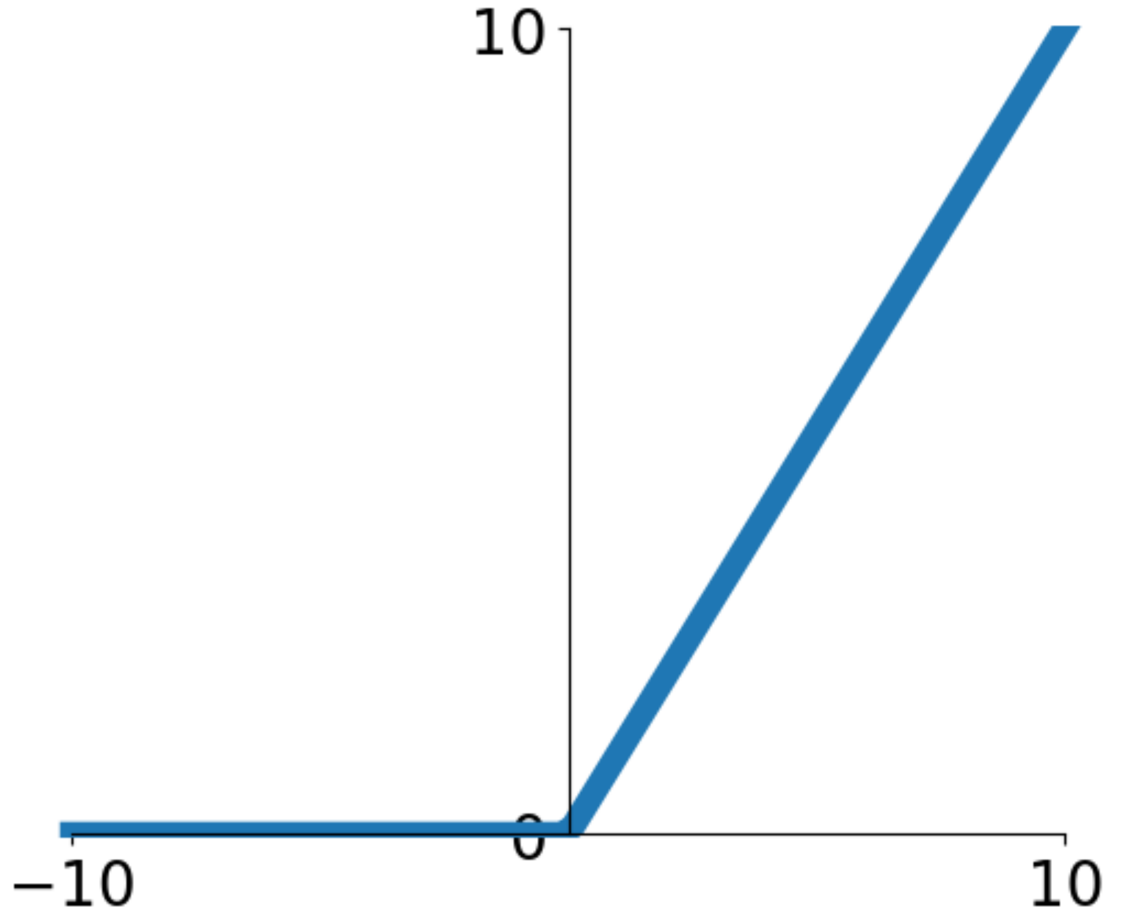
$$a'(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$

Pros:

- Does not saturate (for x > 0)
- Computationally very efficient
- Converges faster in practice (e.g. 6 times faster)

Cons:

- Not zero centered



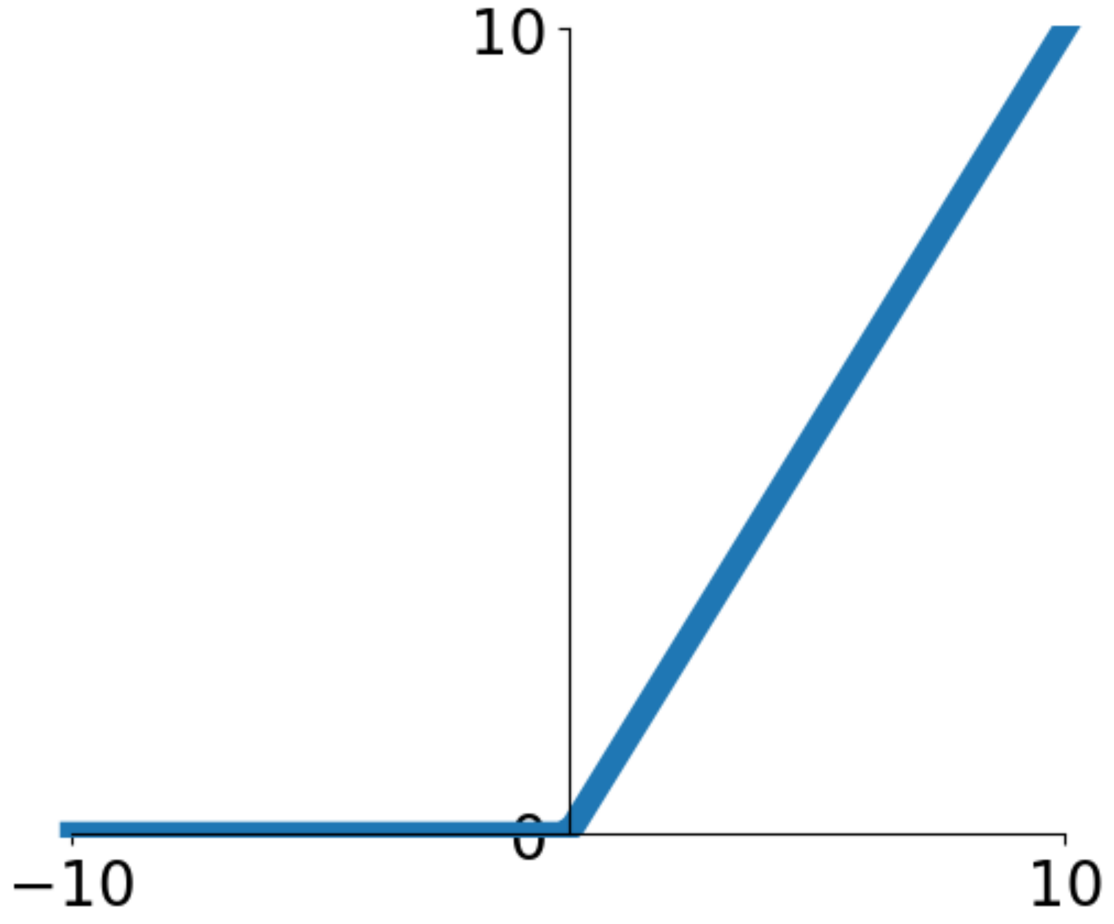
ReLU Activation

Activation Function: Rectified Linear Unit (ReLU)

$$a(x) = max(0, x)$$

$$a'(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$

Question: What do ReLU layers accomplish?



ReLU Activation

* slide adopted from Li, Karpathy, Johnson's **CS231n at Stanford**

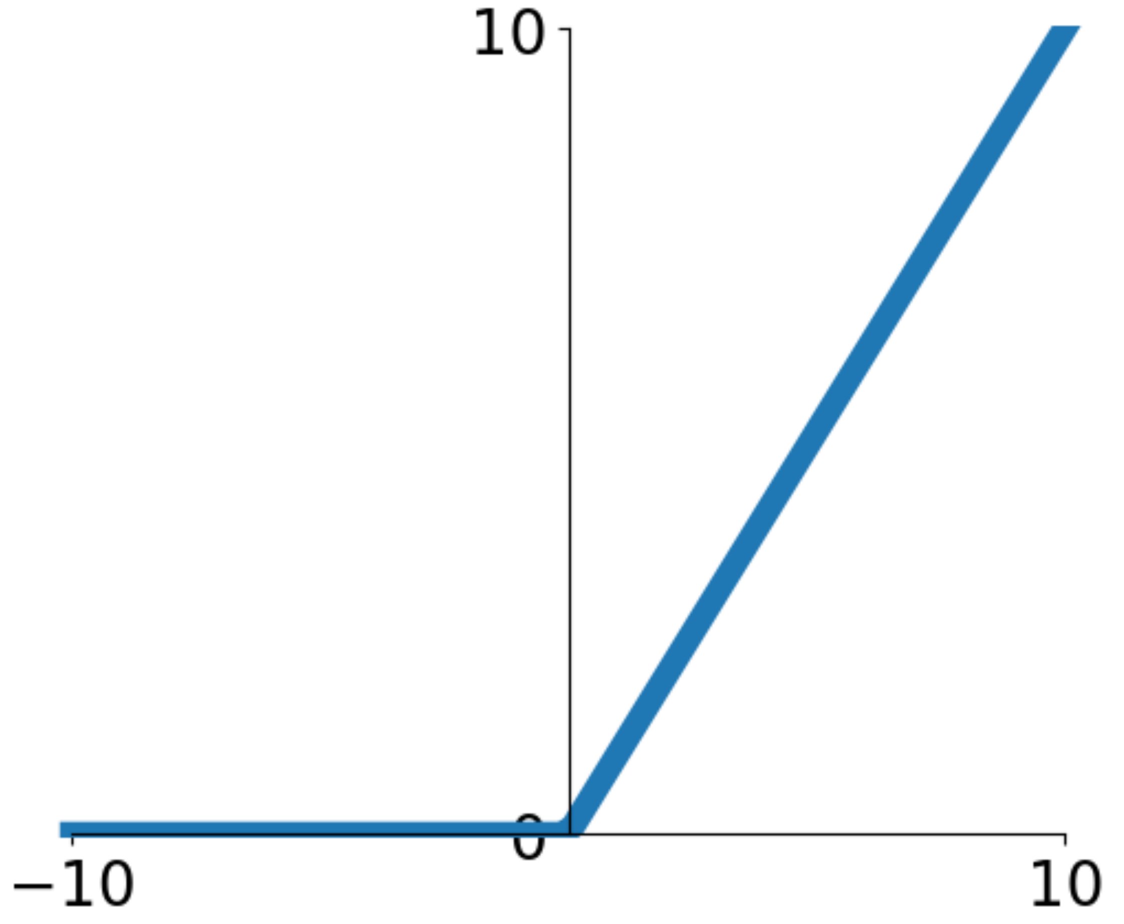
Activation Function: Rectified Linear Unit (ReLU)

$$a(x) = max(0, x)$$

$$a'(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$

Question: What do ReLU layers accomplish?

Answer: Locally linear tiling, function is locally linear



ReLU Activation

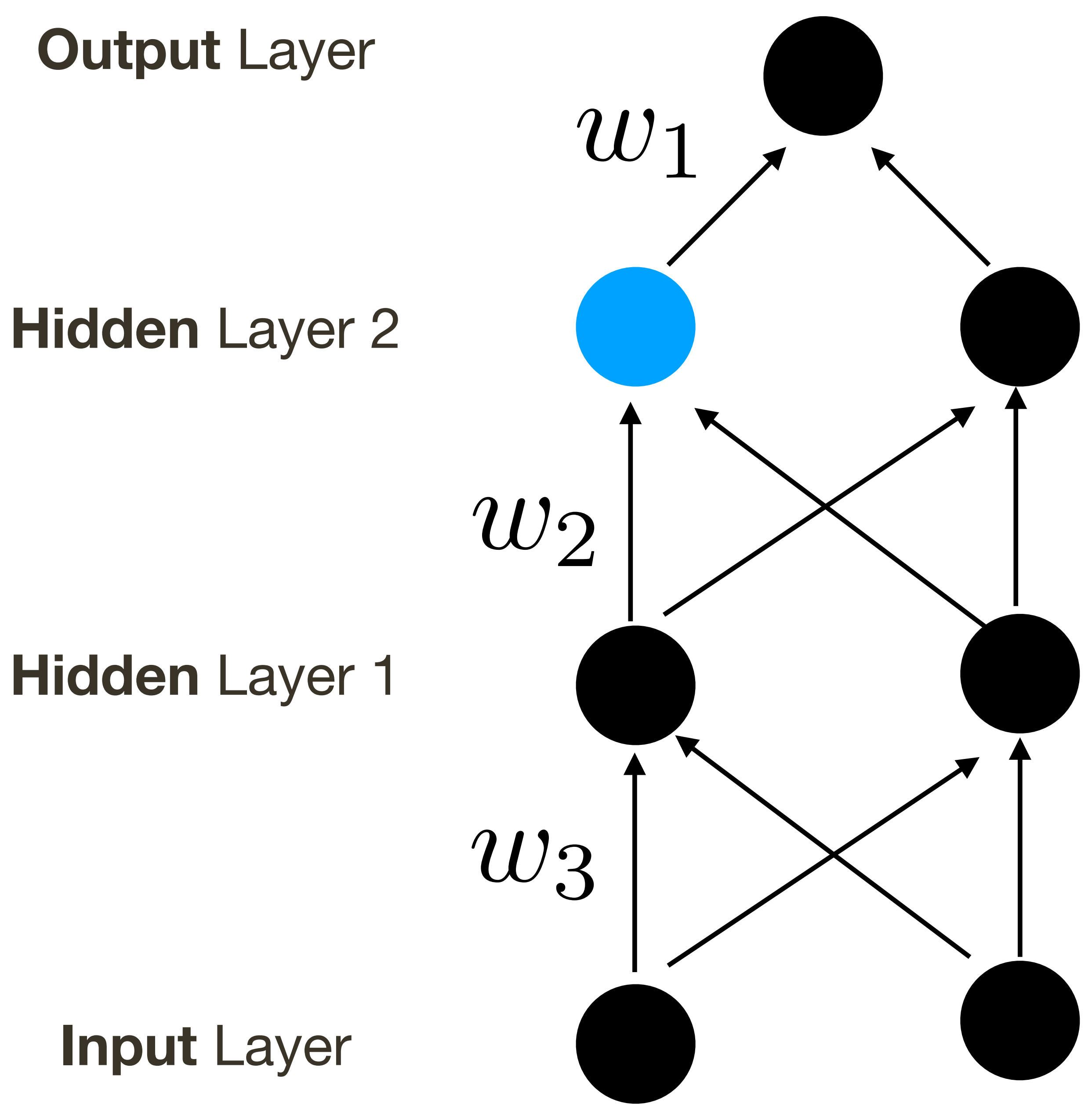
* slide adopted from Li, Karpathy, Johnson's **CS231n at Stanford**

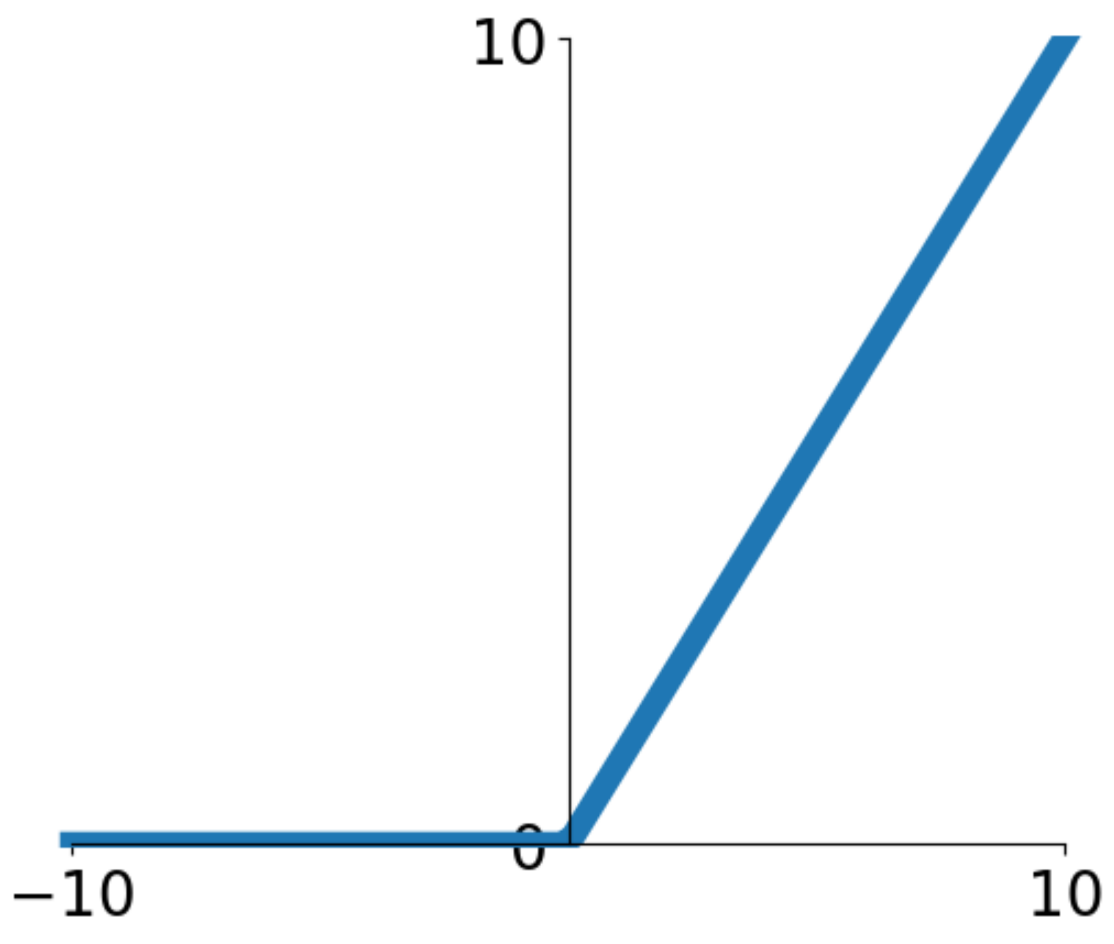
Activation Function: Rectified Linear Unit (ReLU)

ReLU sparcifies activations and derivatives

$$a(x) = max(0, x)$$

$$a'(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$





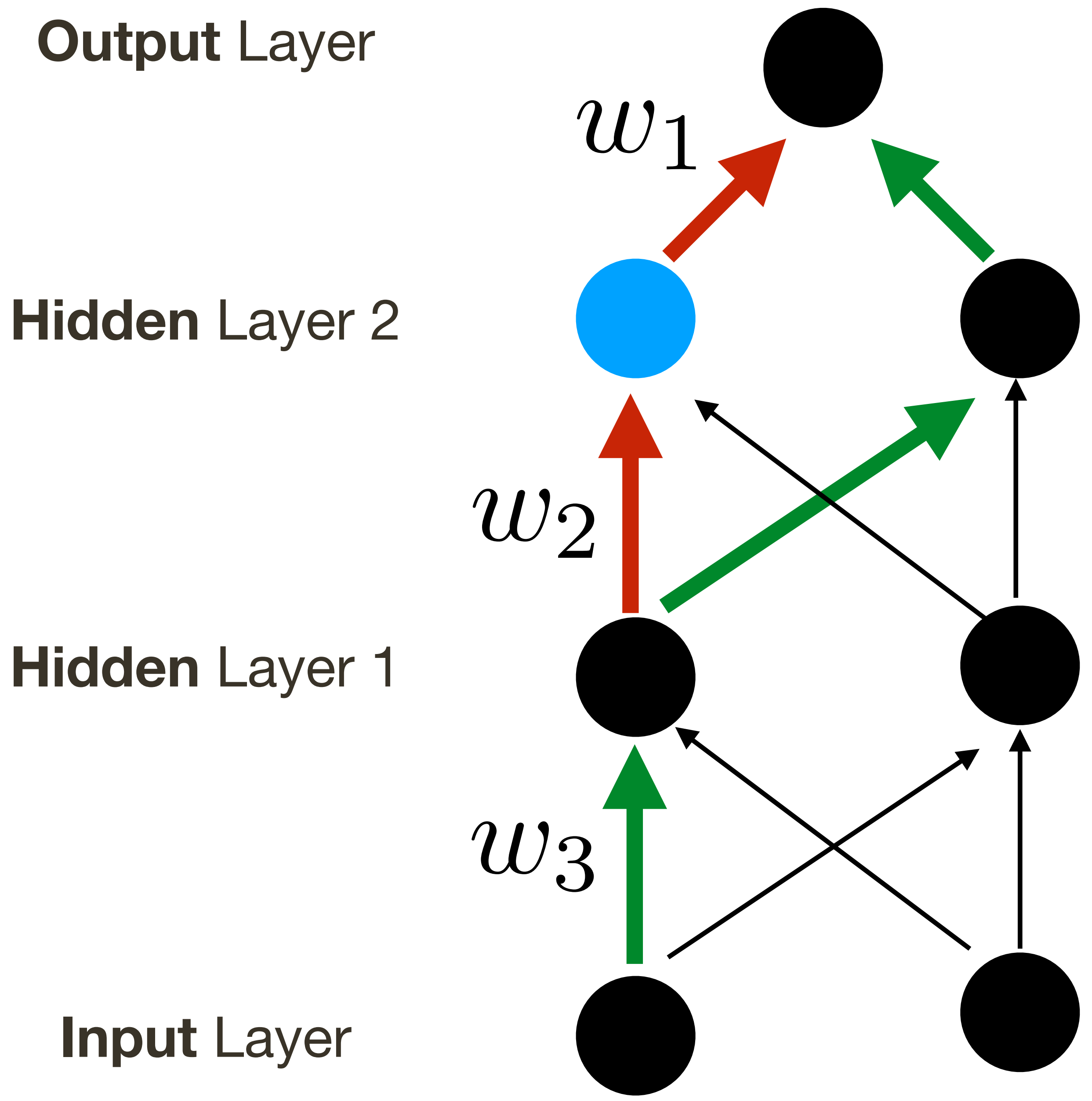
ReLU Activation

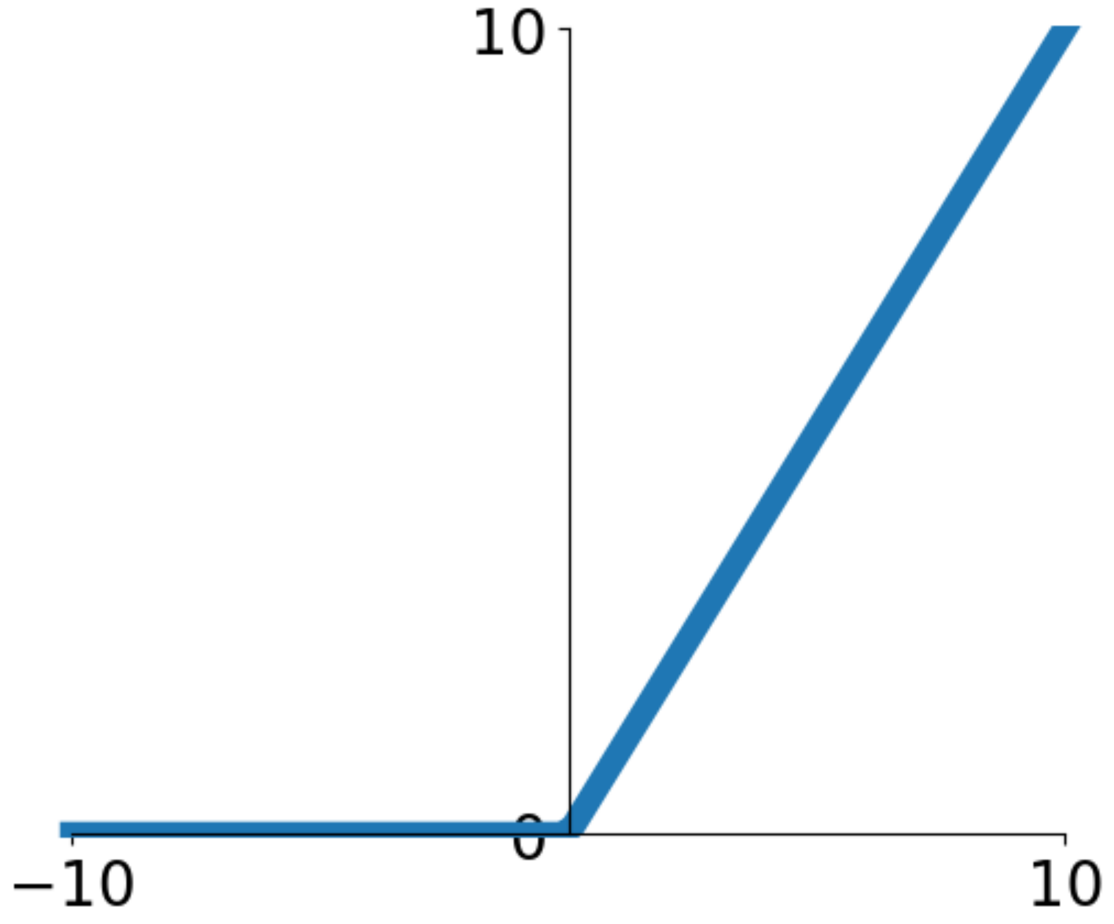
Activation Function: Rectified Linear Unit (ReLU)

ReLU sparcifies activations and derivatives

$$a(x) = max(0, x)$$

$$a'(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$





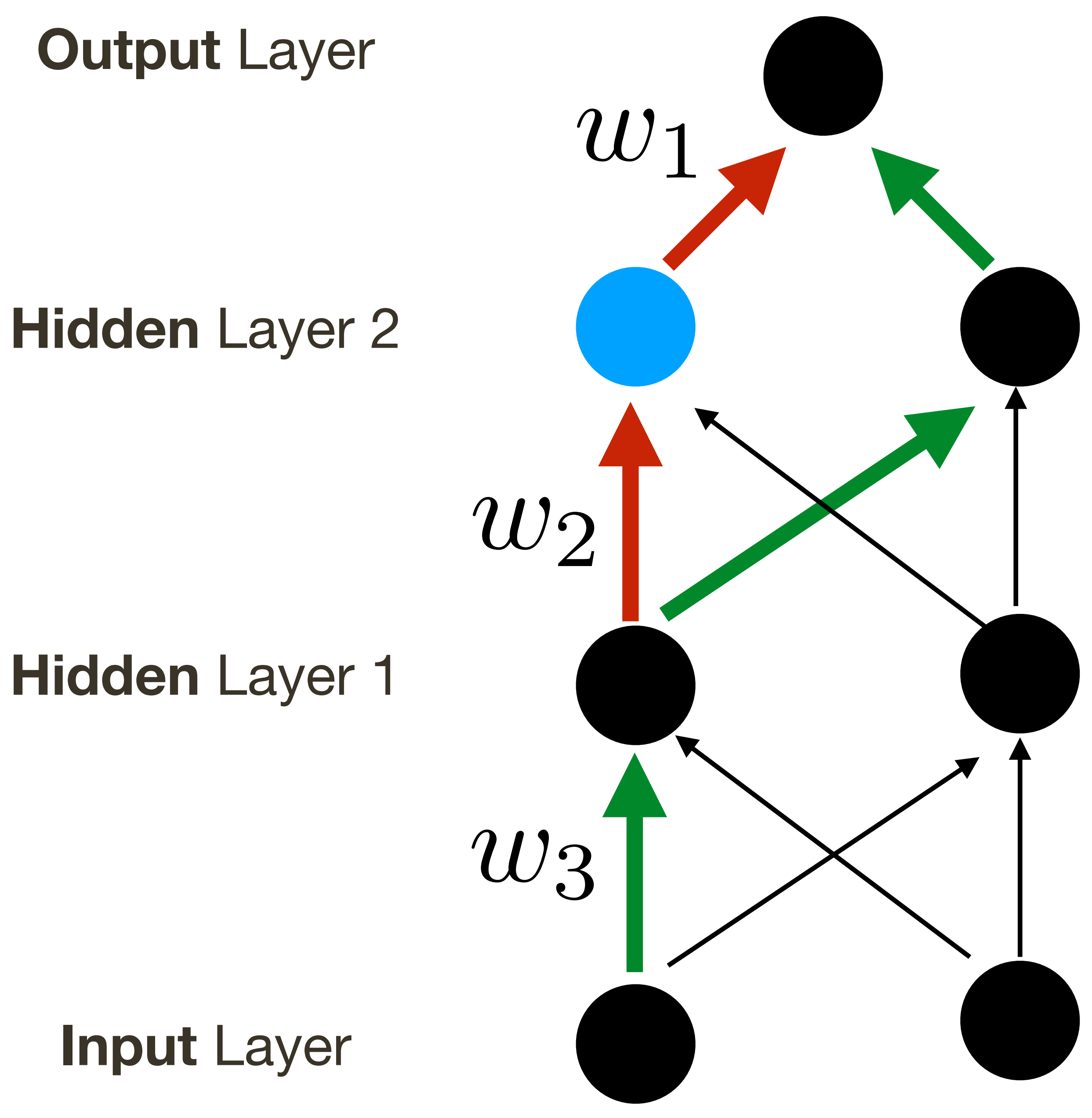
ReLU Activation

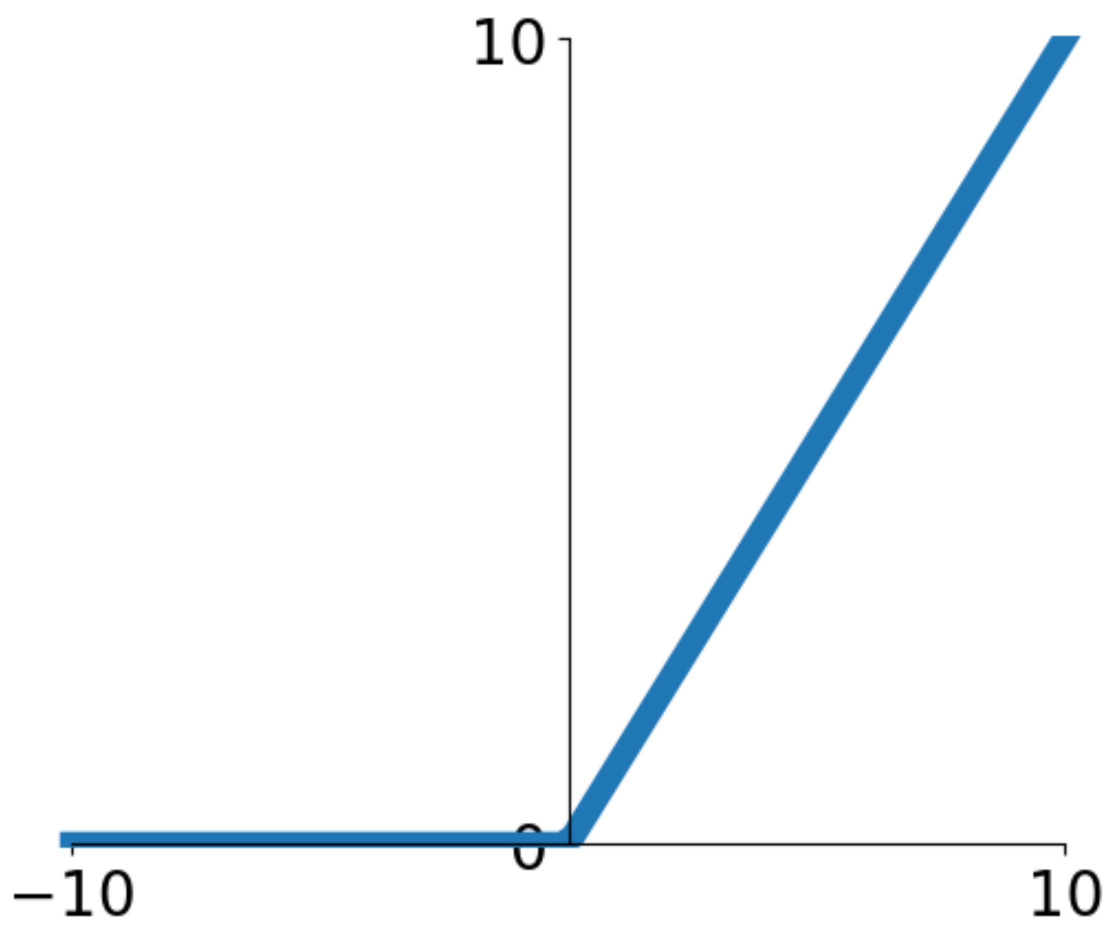
Activation Function: Rectified Linear Unit (ReLU)

ReLU sparcifies activations and derivatives

$$a(x) = max(0, x)$$

$$a'(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$





ReLU Activation

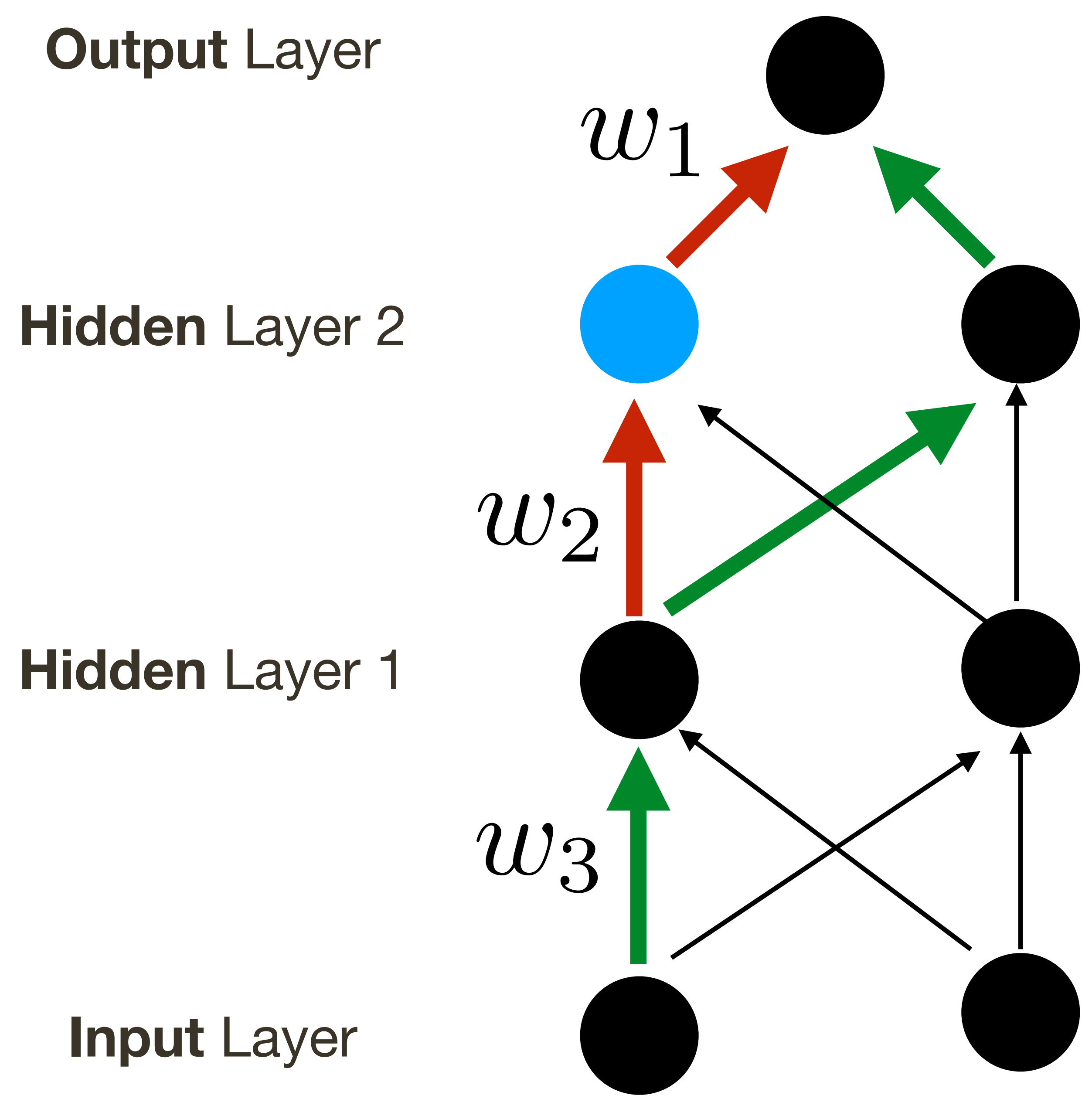
10%-20% of neurons end up being “**dead**” in most standard networks

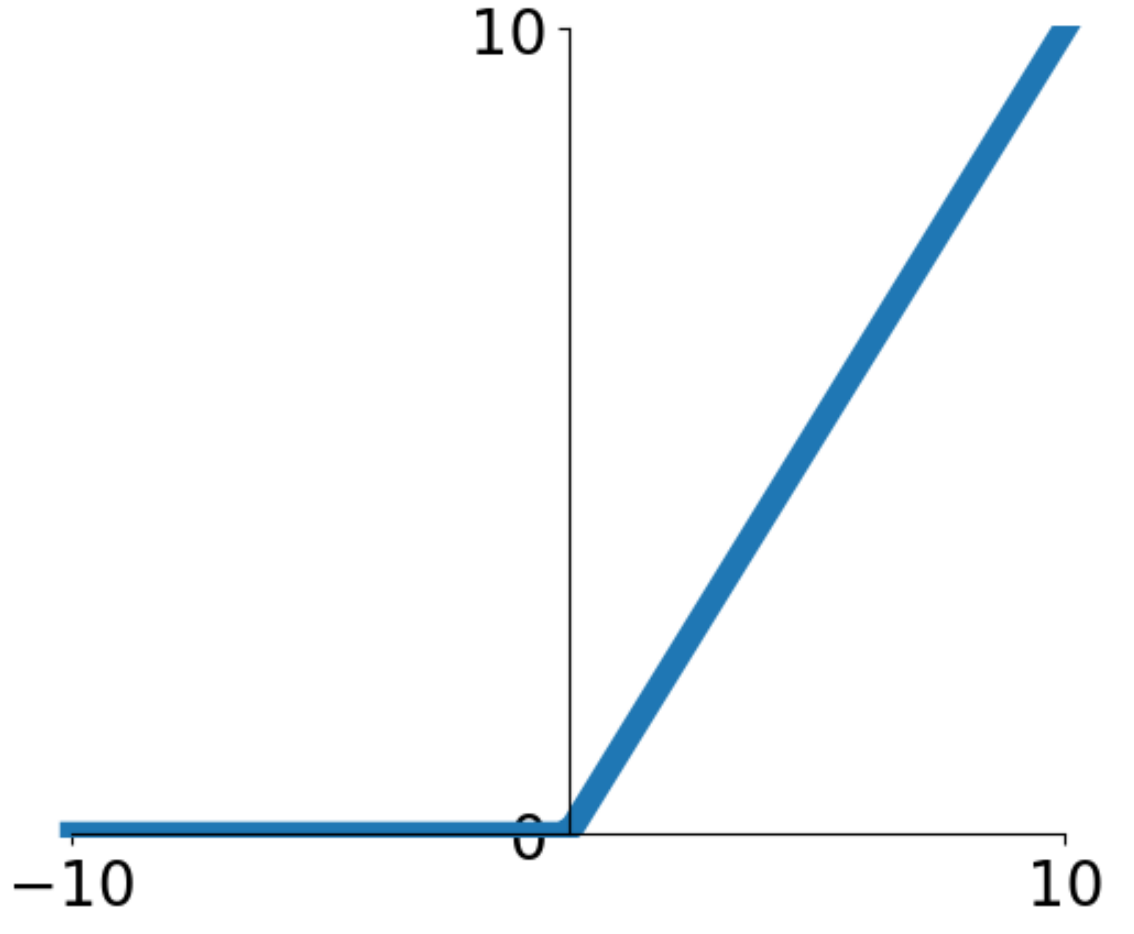
Activation Function: Rectified Linear Unit (ReLU)

ReLU sparcifies activations and derivatives

$$a(x) = max(0, x)$$

$$a'(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$





ReLU Activation

Trick: initialize bias for neurons with ReLU activation to small positive value (0.01)

Initialization

Many tricks for initializations exist. I will not really cover this.

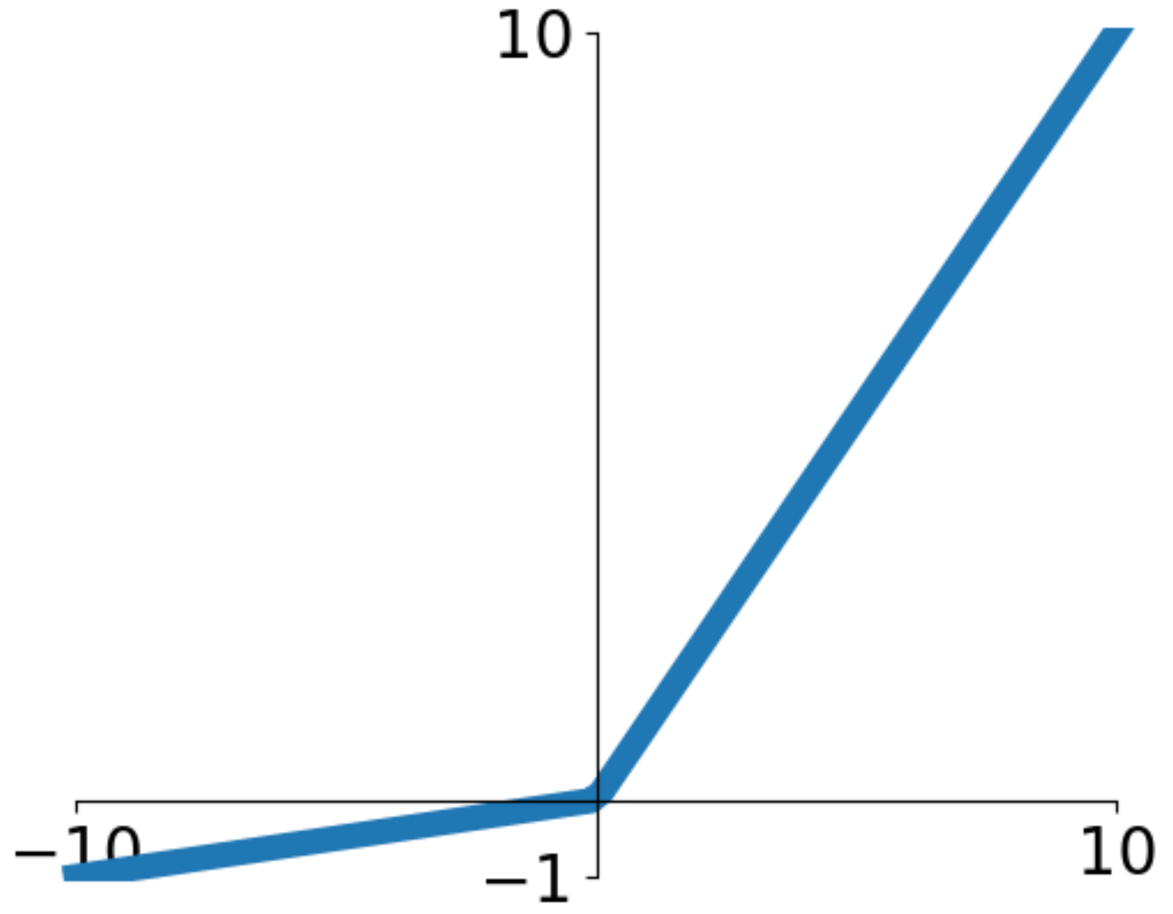
You will partly see why soon …

Activation Function: Leaky / Parametrized ReLU

Leaky: alpha is fixed to a small value (e.g., 0.01)

$$a(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha x & \text{if } x < 0 \end{cases}$$

Parametrized: alpha is optimized as part of the network (BackProp through)



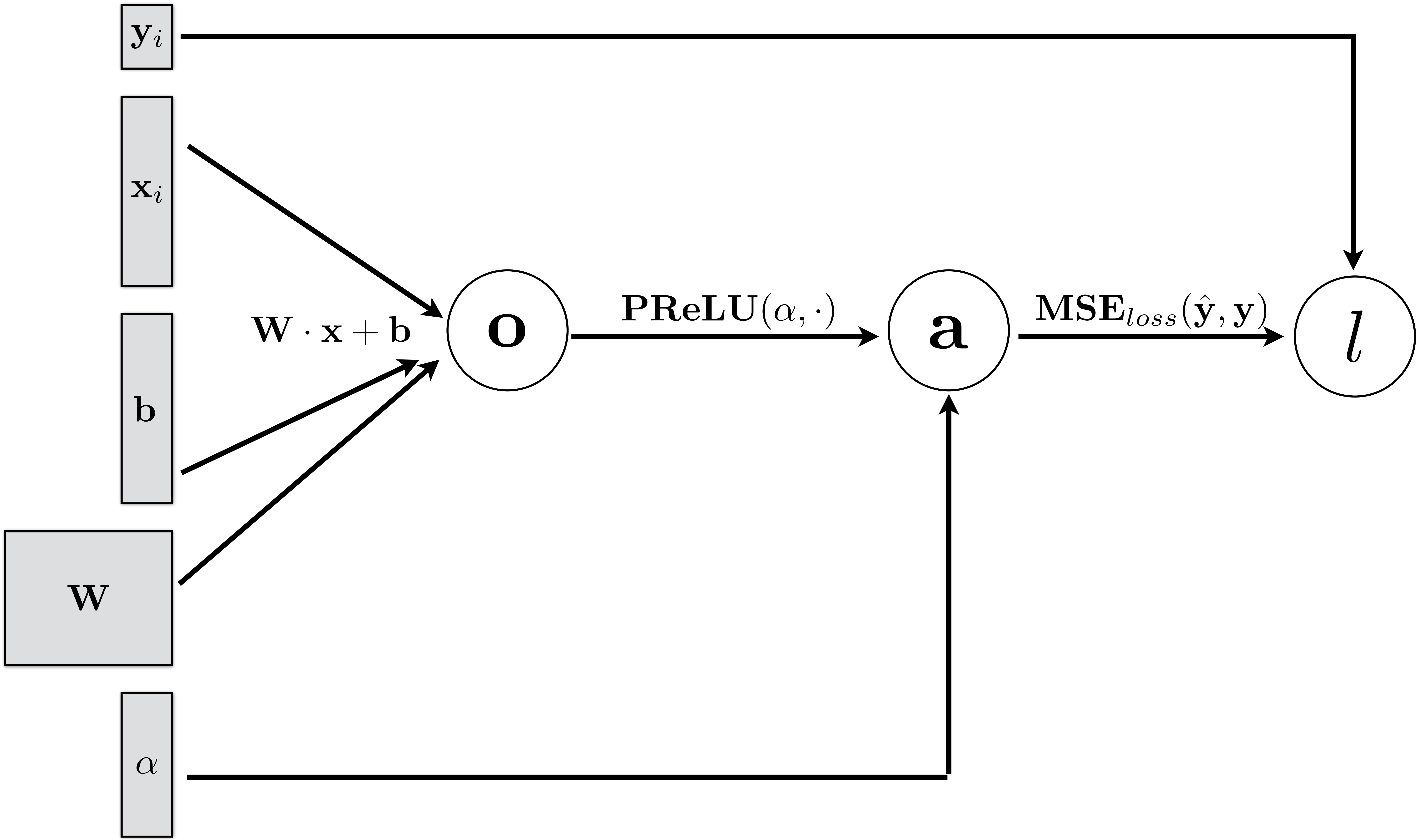
Leaky / Parametrized ReLU Activation

Pros:

- Does not saturate
- Computationally very efficient
- Converges faster in practice (e.g. 6x)

* slide adopted from Li, Karpathy, Johnson's **CS231n at Stanford**

Computational Graph: 1-layer with PReLU



Activation Function: S-shaped ReLU

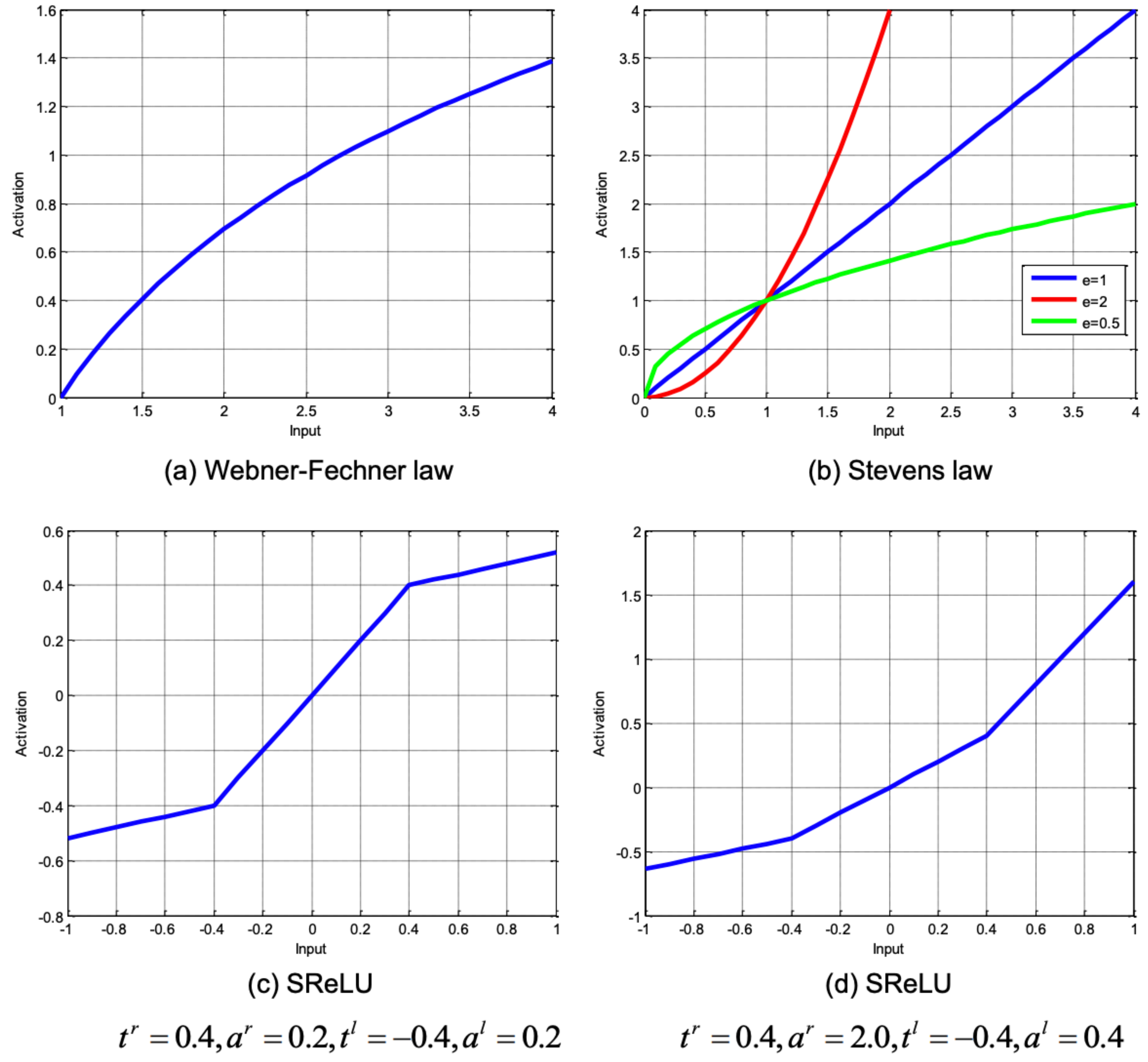
$$a(x) = \begin{cases} \beta_r + \alpha_r(x - \beta_r), & x \geq \beta_r \\ x, & \beta_r \geq x \geq \beta_l \\ \beta_l + \alpha_l(x - \beta_l), & x \leq \beta_l \end{cases}$$

Pros:

- Motivated by neuroscience principles, mainly Webner-Fechner law and Stevens law
- Does not saturate
- Relatively efficient

Cons:

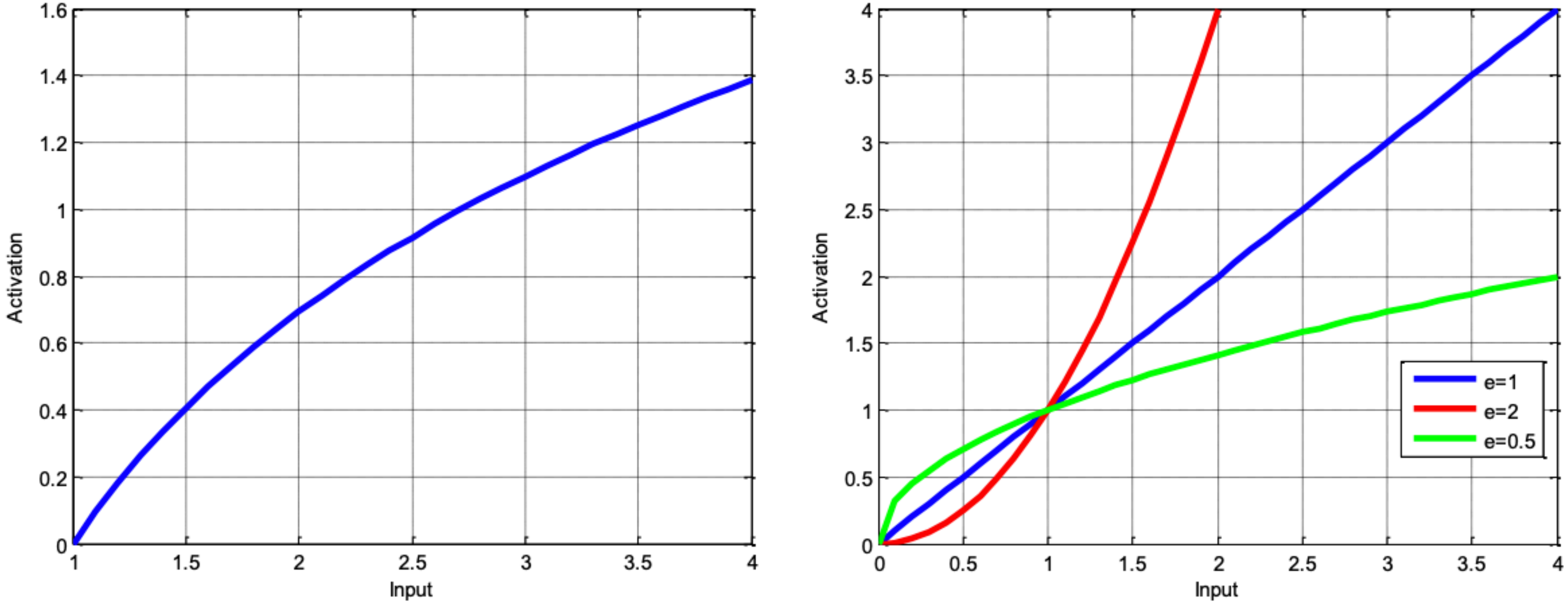
- Need to learn more (4) parameters



[Jin et al., AAAI 2016]

Activation Function: S-shaped ReLU

$$a(x) = \begin{cases} \beta_r + \alpha_r(x - \beta_r), & x \geq \beta_r \\ x, & \beta_r \geq x \geq \beta_l \\ \beta_l + \alpha_l(x - \beta_l), & x \leq \beta_l \end{cases}$$



(a) Webner-Fechner law

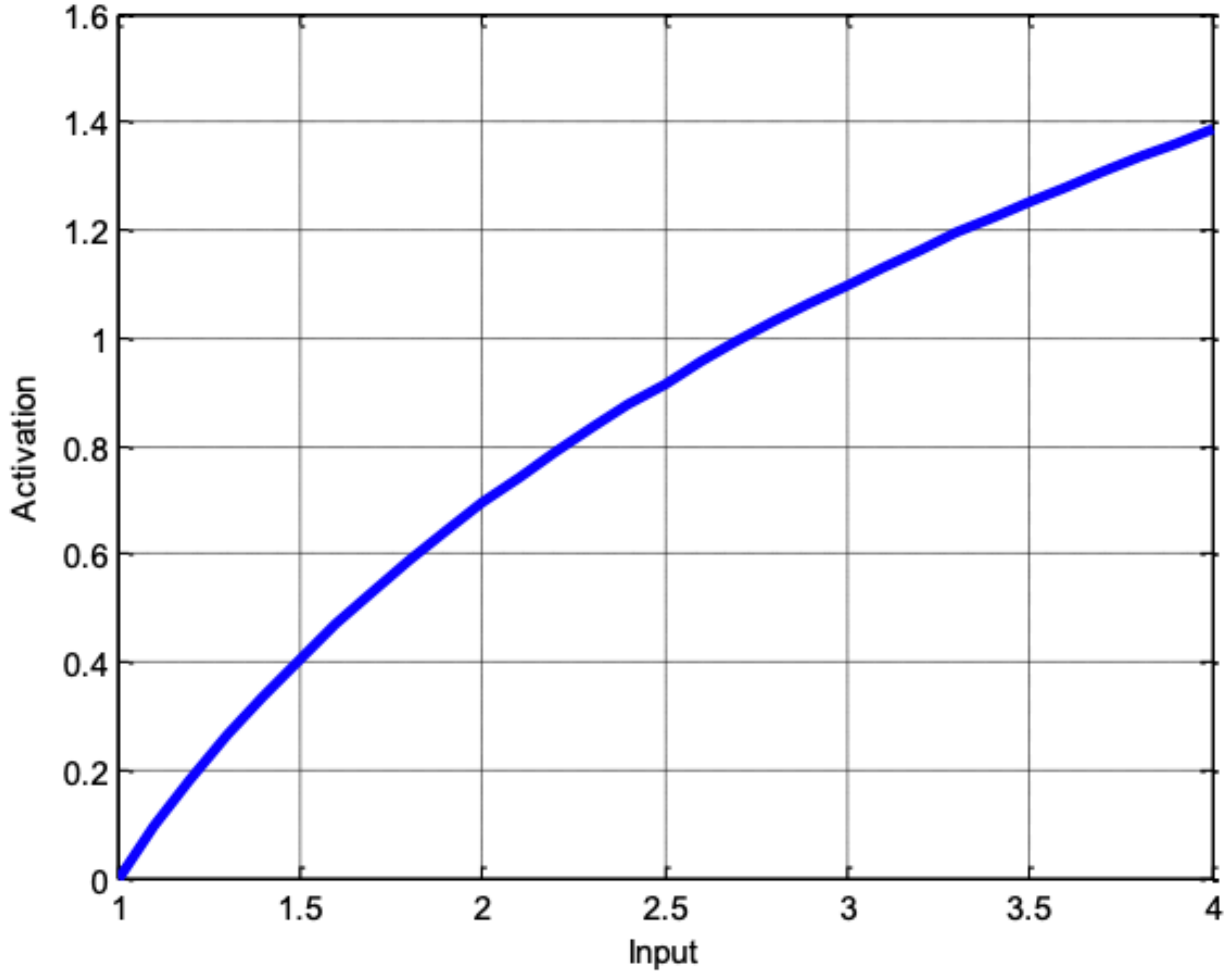
(b) Stevens law

[Jin et al., AAAI 2016]

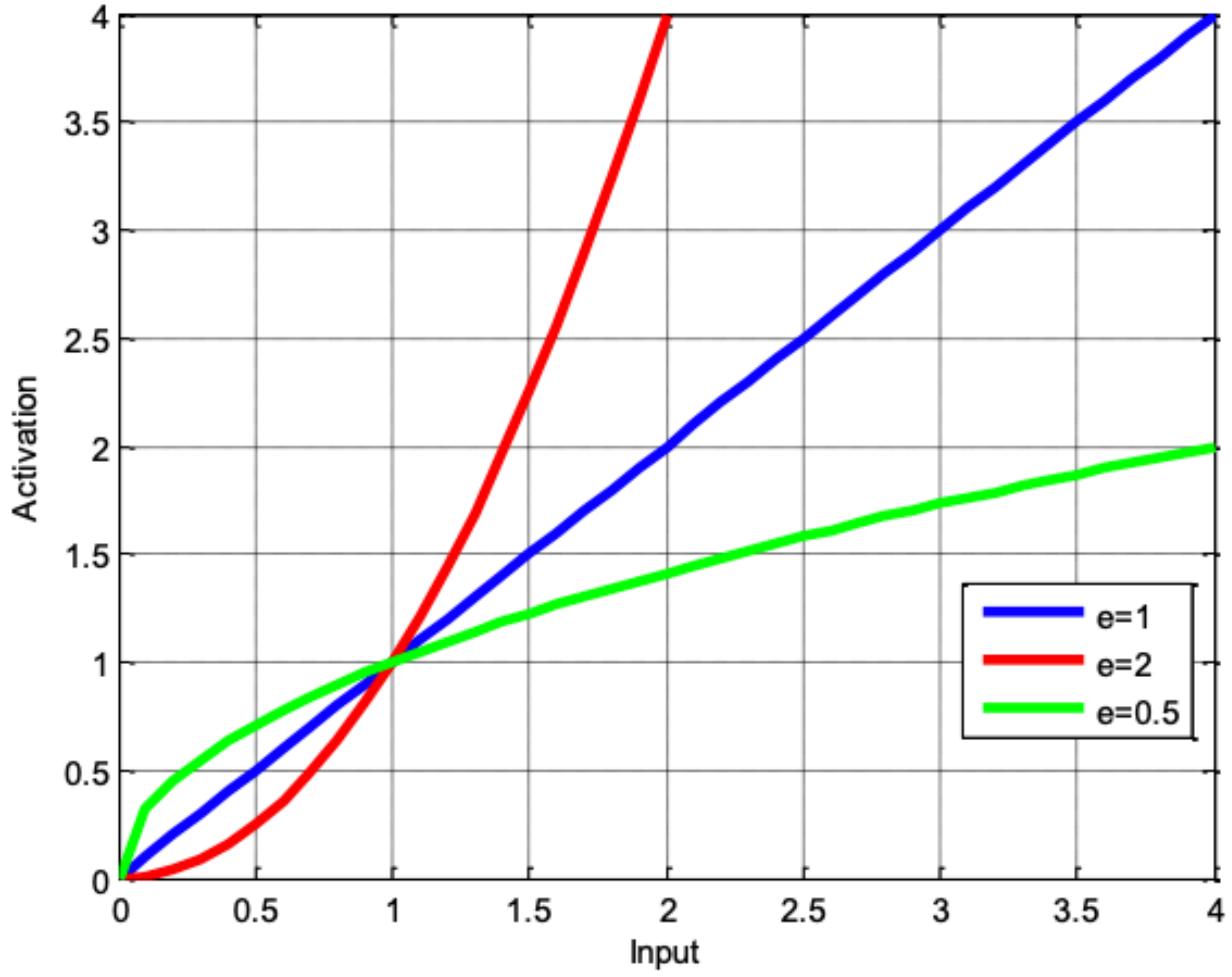
Activation Function: S-shaped ReLU

Why are inputs all positive?

$$a(x) = \begin{cases} \beta_r + \alpha_r(x - \beta_r), & x \geq \beta_r \\ x, & \beta_r \geq x \geq \beta_l \\ \beta_l + \alpha_l(x - \beta_l), & x \leq \beta_l \end{cases}$$



(a) Webner-Fechner law



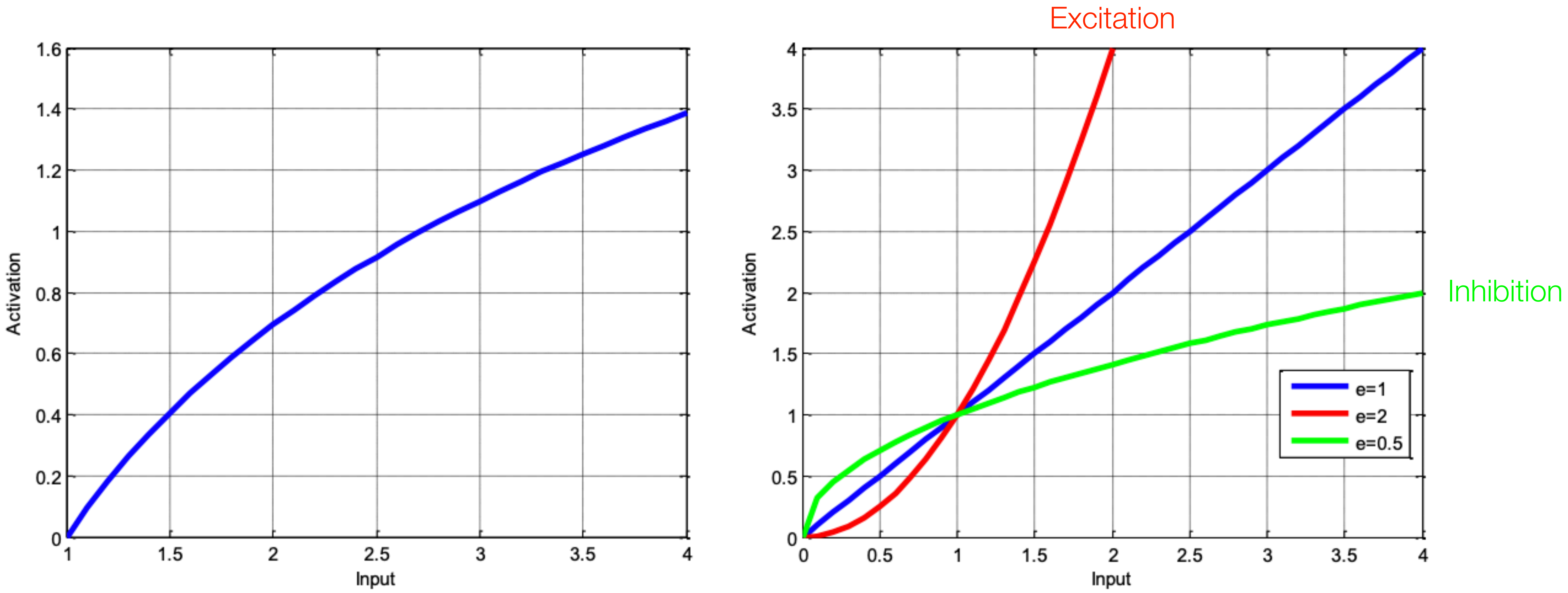
(b) Stevens law

[Jin et al., AAAI 2016]

Activation Function: S-shaped ReLU

Why are inputs all positive?

$$a(x) = \begin{cases} \beta_r + \alpha_r(x - \beta_r), & x \geq \beta_r \\ x, & \beta_r \geq x \geq \beta_l \\ \beta_l + \alpha_l(x - \beta_l), & x \leq \beta_l \end{cases}$$



(a) Webner-Fechner law

(b) Stevens law

[Jin et al., AAAI 2016]

Activation Function: S-shaped ReLU

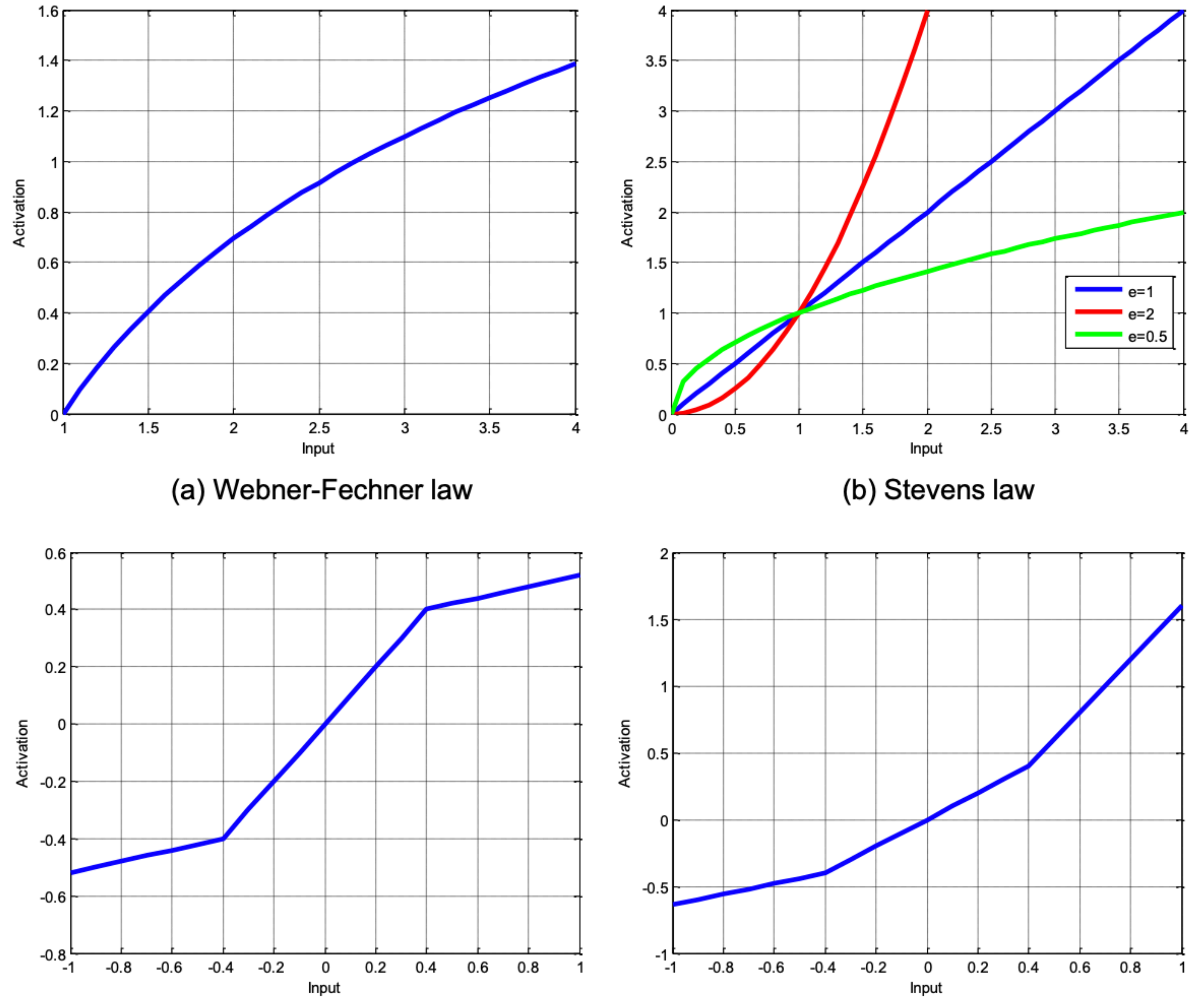
$$a(x) = \begin{cases} \beta_r + \alpha_r(x - \beta_r), & x \geq \beta_r \\ x, & \beta_r \geq x \geq \beta_l \\ \beta_l + \alpha_l(x - \beta_l), & x \leq \beta_l \end{cases}$$

Pros:

- Motivated by neuroscience principles, mainly Webner-Fechner law and Stevens law
- Does not saturate
- Relatively efficient

Cons:

- Need to learn more (4) parameters



(a) Webner-Fechner law

(b) Stevens law

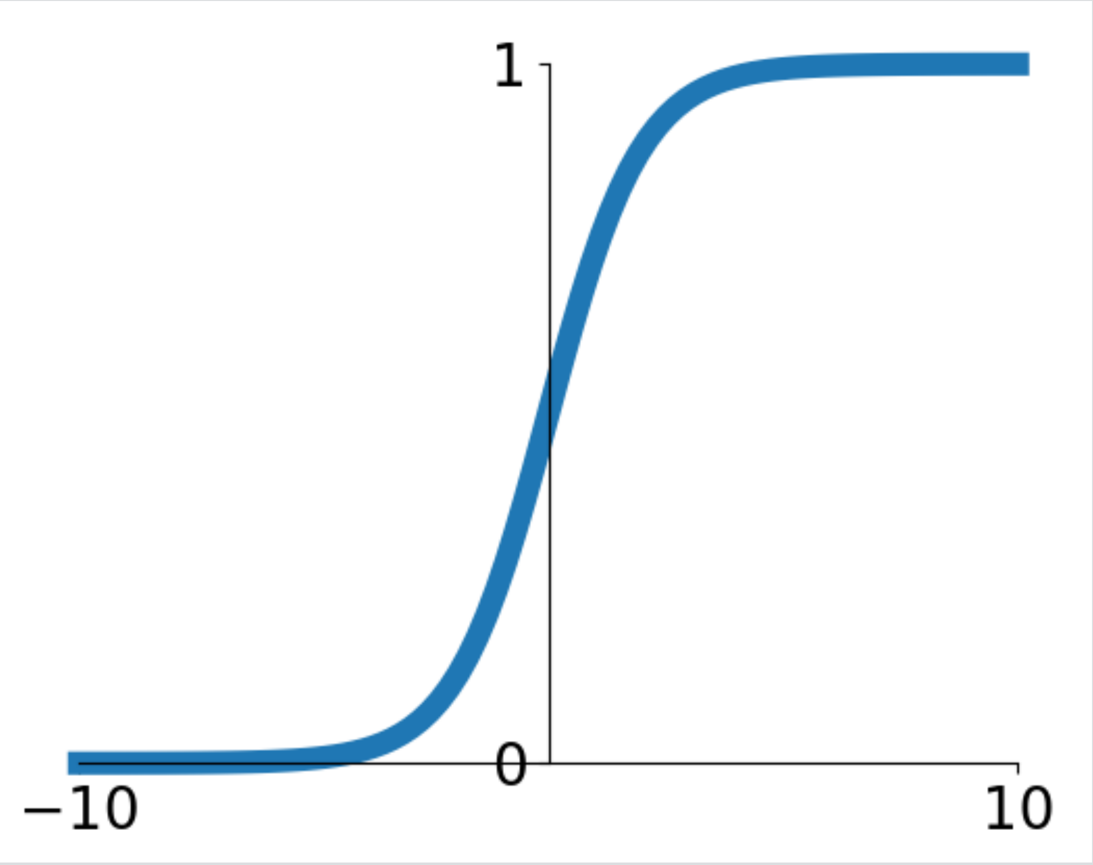
(c) SReLU $t^r = 0.4, a^r = 0.2, t^l = -0.4, a^l = 0.2$

(d) SReLU $t^r = 0.4, a^r = 2.0, t^l = -0.4, a^l = 0.4$

[Jin et al., AAAI 2016]

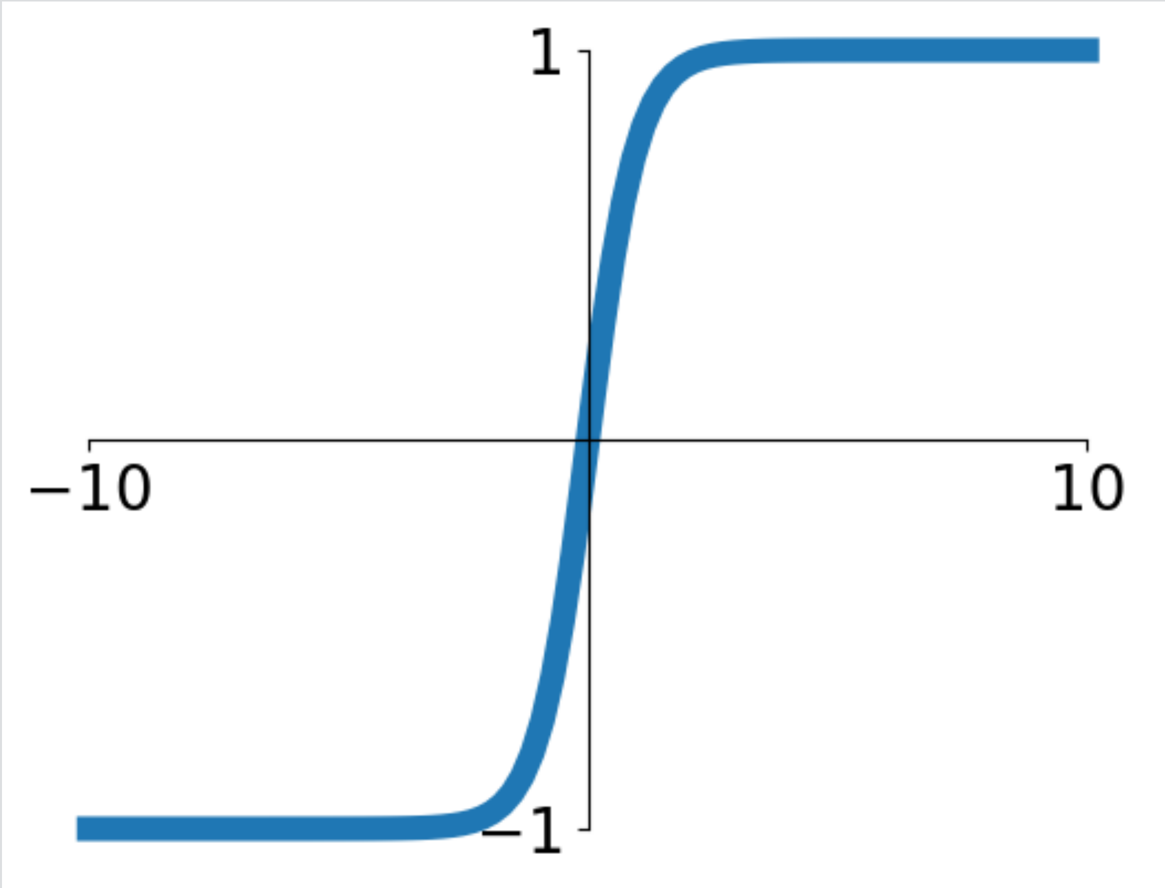
Activation Functions: Review

$$a(x) = \textbf{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$



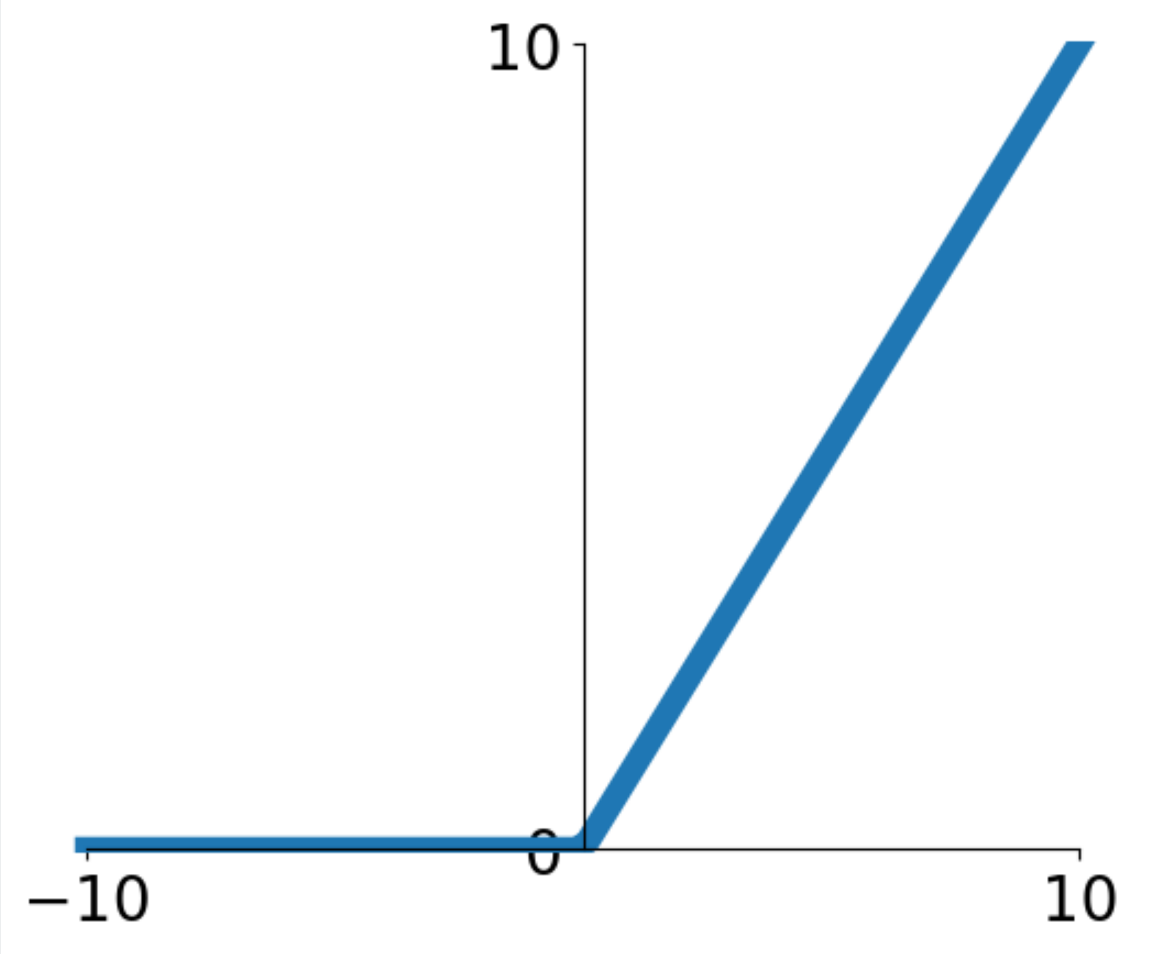
Sigmoid

$$a(x) = \textbf{tanh}(x) = \frac{2}{1 + e^{-2x}} - 1$$



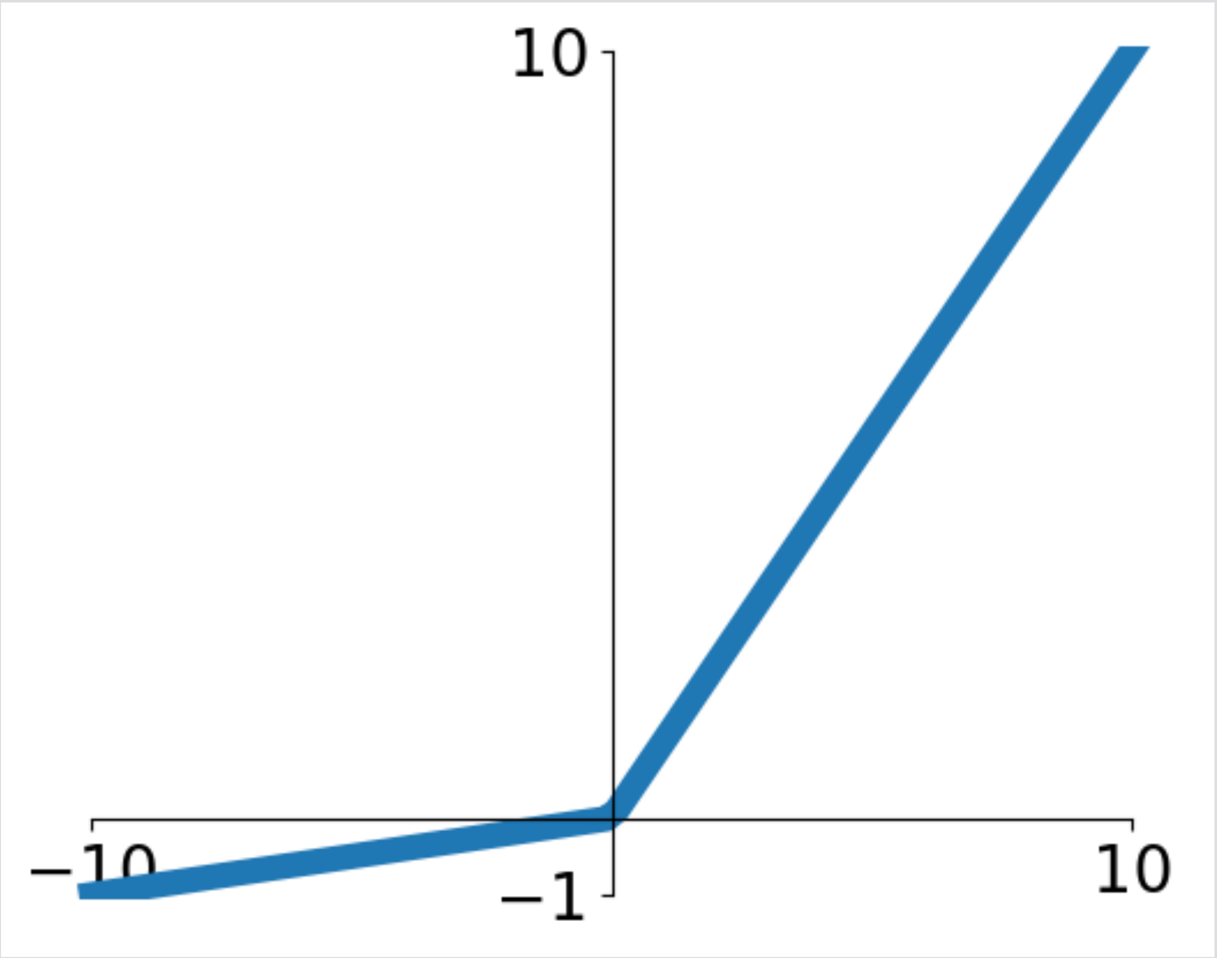
Tanh

$$a(x) = max(0, x)$$



ReLU

$$a(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha x & \text{if } x < 0 \end{cases}$$

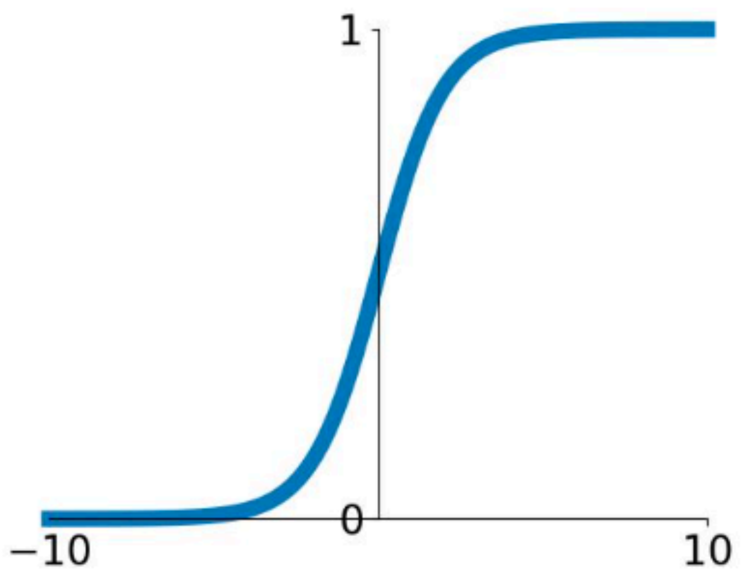


Leaky / Parametrized **ReLU**

Activation Functions: Review

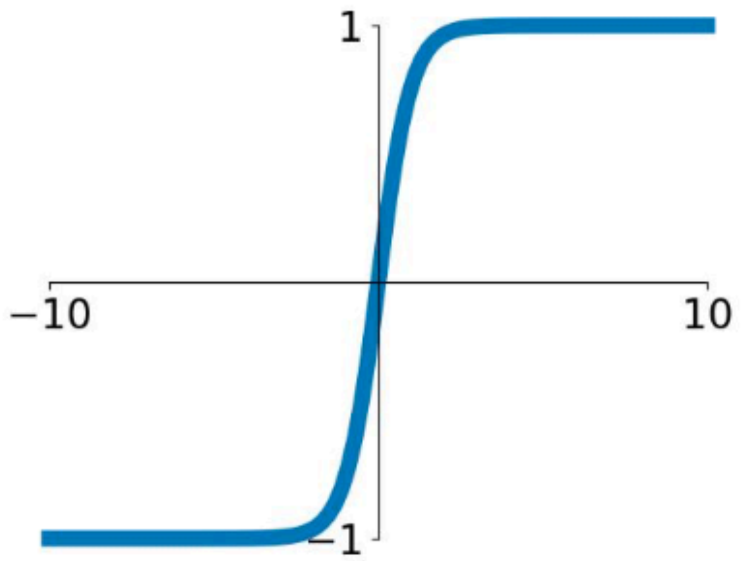
Sigmoid

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



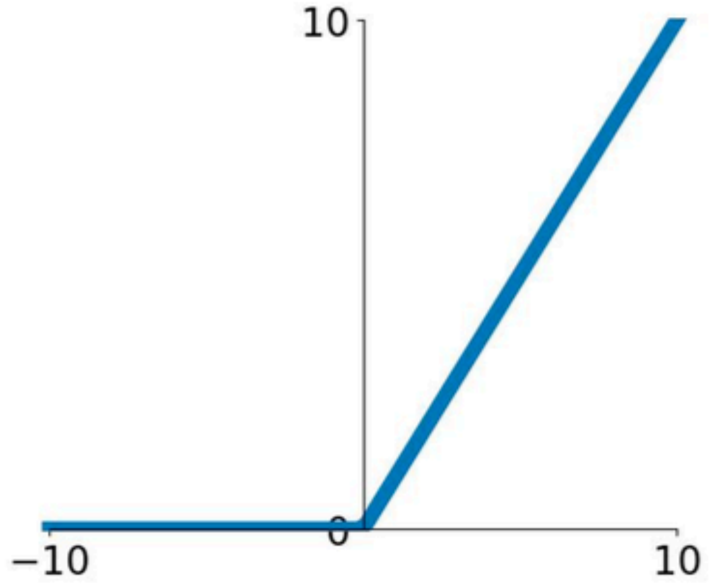
tanh

$$\tanh(x)$$



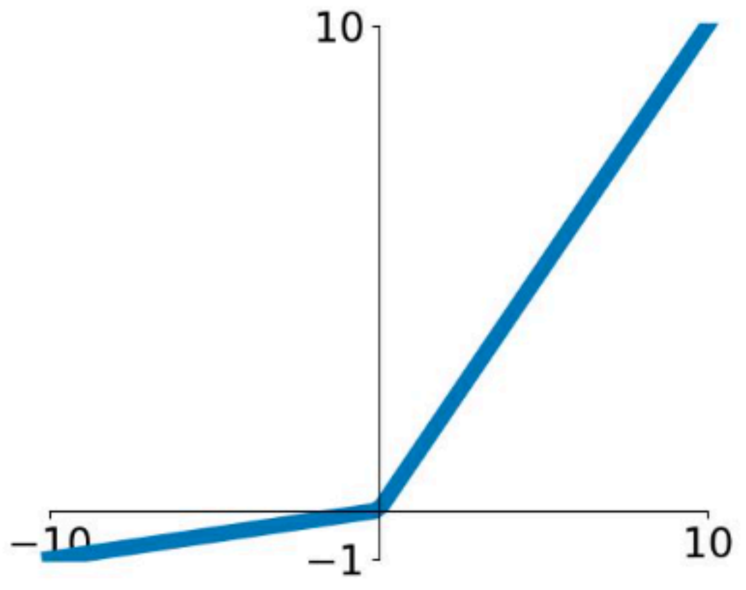
ReLU

$$\max(0, x)$$



Leaky ReLU

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

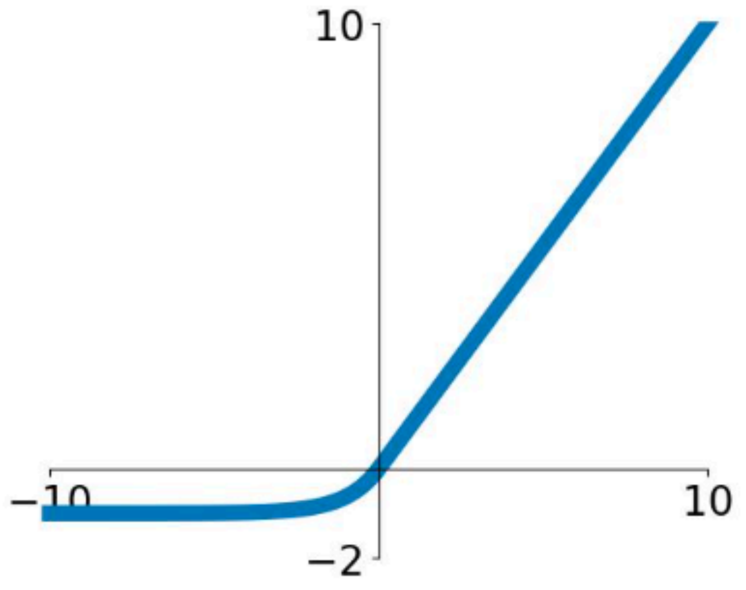


Maxout

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

ELU

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



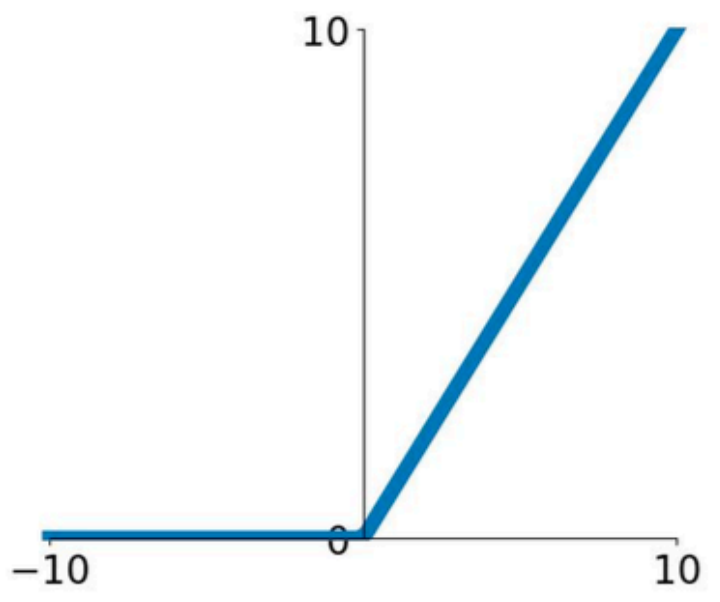
* slide adopted from Li, Karpathy, Johnson's **CS231n at Stanford**

Activation Functions: Review

Good "**default**" choice

ReLU

$\max(0, x)$



* slide adopted from Li, Karpathy, Johnson's **CS231n at Stanford**

Regularization: L2 or L1 on the weights

L2 Regularization: Learn a more (dense) distributed representation

$$R(\mathbf{W}) = ||\mathbf{W}||_2 = \sum_i \sum_j \mathbf{W}_{i,j}^2$$

L1 Regularization: Learn a sparse representation (few non-zero wight elements)

$$R(\mathbf{W}) = ||\mathbf{W}||_1 = \sum_i \sum_j |\mathbf{W}_{i,j}|$$

(others regularizers are also possible)

Regularization: L2 or L1 on the weights

L2 Regularization: Learn a more (dense) distributed representation

$$R(\mathbf{W}) = ||\mathbf{W}||_2 = \sum_i \sum_j \mathbf{W}_{i,j}^2$$

L1 Regularization: Learn a sparse representation (few non-zero wight elements)

$$R(\mathbf{W}) = ||\mathbf{W}||_1 = \sum_i \sum_j |\mathbf{W}_{i,j}|$$

(others regularizers are also possible)

Example:

$$\mathbf{x} = [1, 1, 1, 1]$$

$$\mathbf{W}_1 = [1, 0, 0, 0]$$

$$\mathbf{W}_2 = \left[\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}\right]$$

Regularization: L2 or L1 on the weights

L2 Regularization: Learn a more (dense) distributed representation

$$R(\mathbf{W}) = ||\mathbf{W}||_2 = \sum_i \sum_j \mathbf{W}_{i,j}^2$$

L1 Regularization: Learn a sparse representation (few non-zero wight elements)

$$R(\mathbf{W}) = ||\mathbf{W}||_1 = \sum_i \sum_j |\mathbf{W}_{i,j}|$$

(others regularizers are also possible)

Example:

$$\mathbf{x} = [1, 1, 1, 1]$$

$$\mathbf{W}_1 = [1, 0, 0, 0]$$

$$\mathbf{W}_2 = \left[\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}\right]$$

$$\mathbf{W}_1 \cdot \mathbf{x}^T = \mathbf{W}_2 \cdot \mathbf{x}^T$$

two networks will have identical output

Regularization: L2 or L1 on the weights

L2 Regularization: Learn a more (dense) distributed representation

$$R(\mathbf{W}) = ||\mathbf{W}||_2 = \sum_i \sum_j \mathbf{W}_{i,j}^2$$

L1 Regularization: Learn a sparse representation (few non-zero wight elements)

$$R(\mathbf{W}) = ||\mathbf{W}||_1 = \sum_i \sum_j |\mathbf{W}_{i,j}|$$

(others regularizers are also possible)

Example:

$$\mathbf{x} = [1, 1, 1, 1]$$

$$\mathbf{W}_1 = [1, 0, 0, 0]$$

$$\mathbf{W}_2 = \left[\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}\right]$$

$$\mathbf{W}_1 \cdot \mathbf{x}^T = \mathbf{W}_2 \cdot \mathbf{x}^T$$

two networks will have identical output

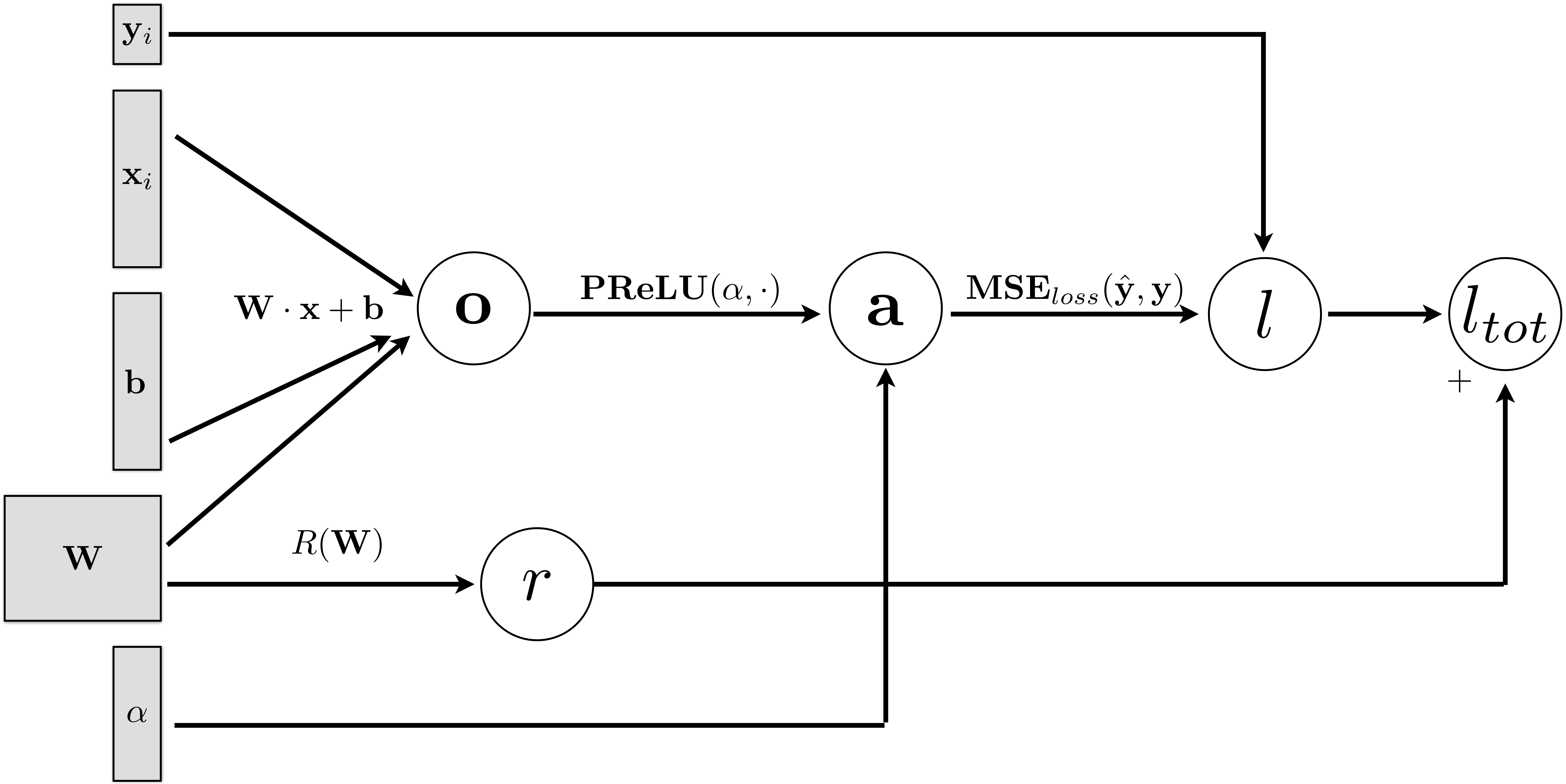
L2 Regularizer:

$$R_{L2}(\mathbf{W}_1) = 1$$
$$R_{L2}(\mathbf{W}_2) = 0.25 \leftarrow$$

L1 Regularizer:

$$R_{L1}(\mathbf{W}_1) = 1 \leftarrow$$
$$R_{L1}(\mathbf{W}_2) = 1 \leftarrow$$

Computational Graph: 1-layer with PReLU + Regularizer



Remember … **Initialization**

Many tricks for initializations exist. I will not really cover this.

Regularization: Batch Normalization

Normalize each mini-batch (using Batch Normalization layer) by subtracting empirically computed mean and dividing by variance for every dimension -> samples are approximately unit Gaussian

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

Benefit:

Improves learning (better gradients, higher learning rate)

[Ioffe and Szegedy, NIPS 2015]

Regularization: Batch Normalization

Normalize each mini-batch (using Batch Normalization layer) by subtracting empirically computed mean and dividing by variance for every dimension -> samples are approximately unit Gaussian

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

Benefit:

Improves learning (better gradients, higher learning rate)

Why?

[Ioffe and Szegedy, NIPS 2015]

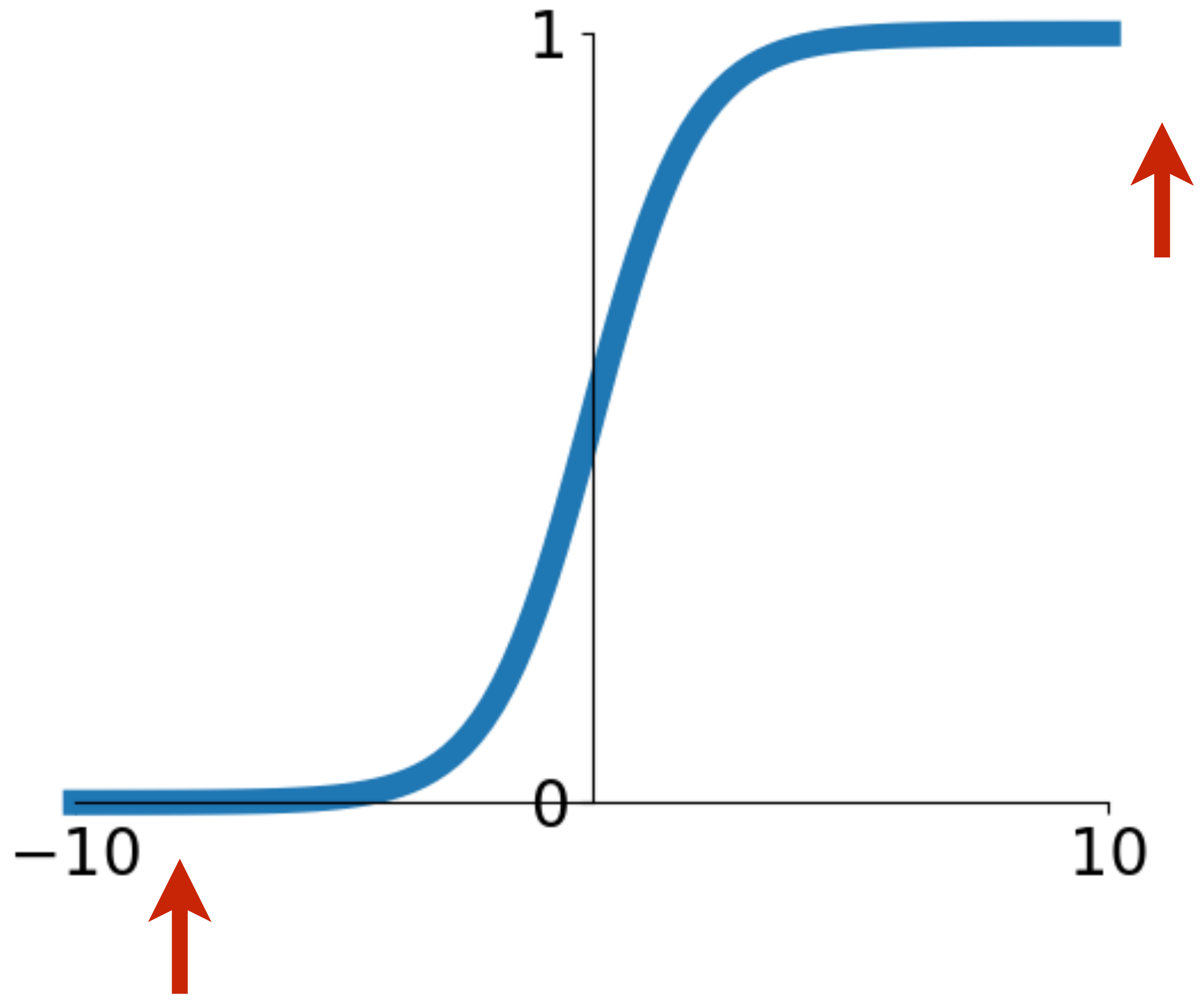
Activation Function: Sigmoid

$$a = \mathbf{sigmoid}(x) = \frac{1}{1+e^{-x}}$$

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial\ \mathbf{sigmoid}(x)}{\partial x}\frac{\partial \mathcal{L}}{\partial a}$$

$$\frac{\partial \mathcal{L}}{\partial a}$$

$$a(x) = \mathbf{sigmoid}(x) = \frac{1}{1+e^{-x}}$$



Sigmoid Activation

Cons:

- Saturated neurons **"kill" the gradients**
- Non-zero centered
- Could be expensive to compute

Regularization: Batch Normalization

Normalize each mini-batch (using Batch Normalization layer) by subtracting empirically computed mean and dividing by variance for every dimension -> samples are approximately unit Gaussian

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

Benefit:

Improves learning (better gradients, higher learning rate)

Typically inserted **before** activation layer

[Ioffe and Szegedy, NIPS 2015]

Regularization: Batch Normalization

Normalize each mini-batch (using Batch Normalization layer) by subtracting empirically computed mean and dividing by variance for every dimension -> samples are approximately unit Gaussian

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

Benefit:

Improves learning (better gradients, higher learning rate)

Typically inserted **before** activation layer

What happens at inference time?

[Ioffe and Szegedy, NIPS 2015]

Regularization: Batch Normalization

Normalize each mini-batch (using Batch Normalization layer) by subtracting empirically computed mean and dividing by variance for every dimension -> samples are approximately unit Gaussian

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

In practice, also learn how to scale and offset:

$$y^{(k)} = \gamma^{(k)} \bar{x}^{(k)} + \beta^{(k)}$$

BN layer parameters

Benefit:

Improves learning (better gradients, higher learning rate, less reliance on initialization)

Typically inserted **before** activation layer

[Ioffe and Szegedy, NIPS 2015]

Regularization: Batch Normalization

Consider what happens at **runtime**, when you are only passing a single sample

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

In practice, also learn how to scale and offset:

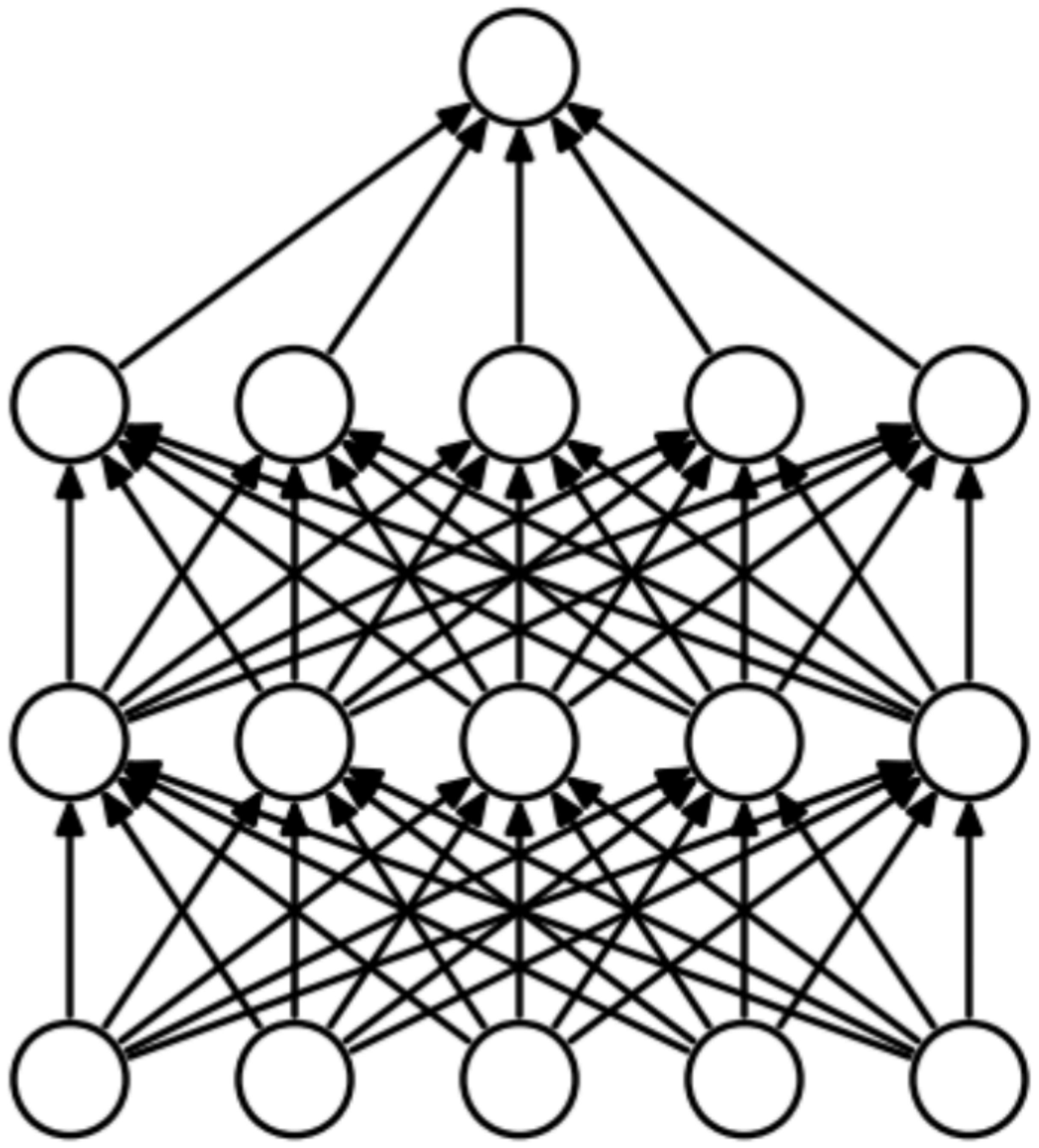
$$y^{(k)} = \gamma^{(k)} \bar{x}^{(k)} + \beta^{(k)}$$

BN layer parameters

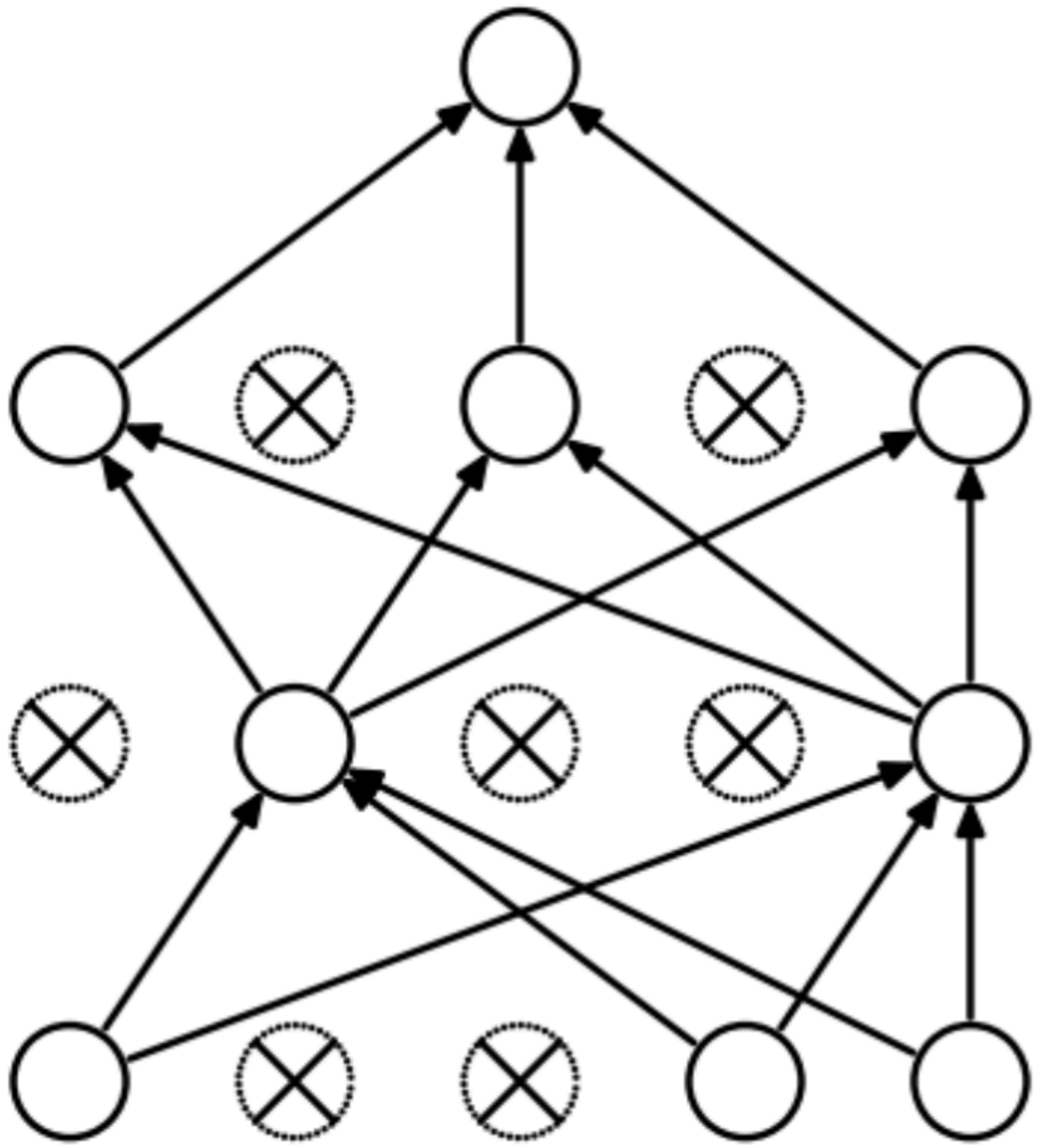
[Ioffe and Szegedy, NIPS 2015]

Regularization: Dropout

Randomly **set some neurons to zero** in the forward pass, with probability proportional to `dropout rate` (between 0 to 1)



Standar Neural Network

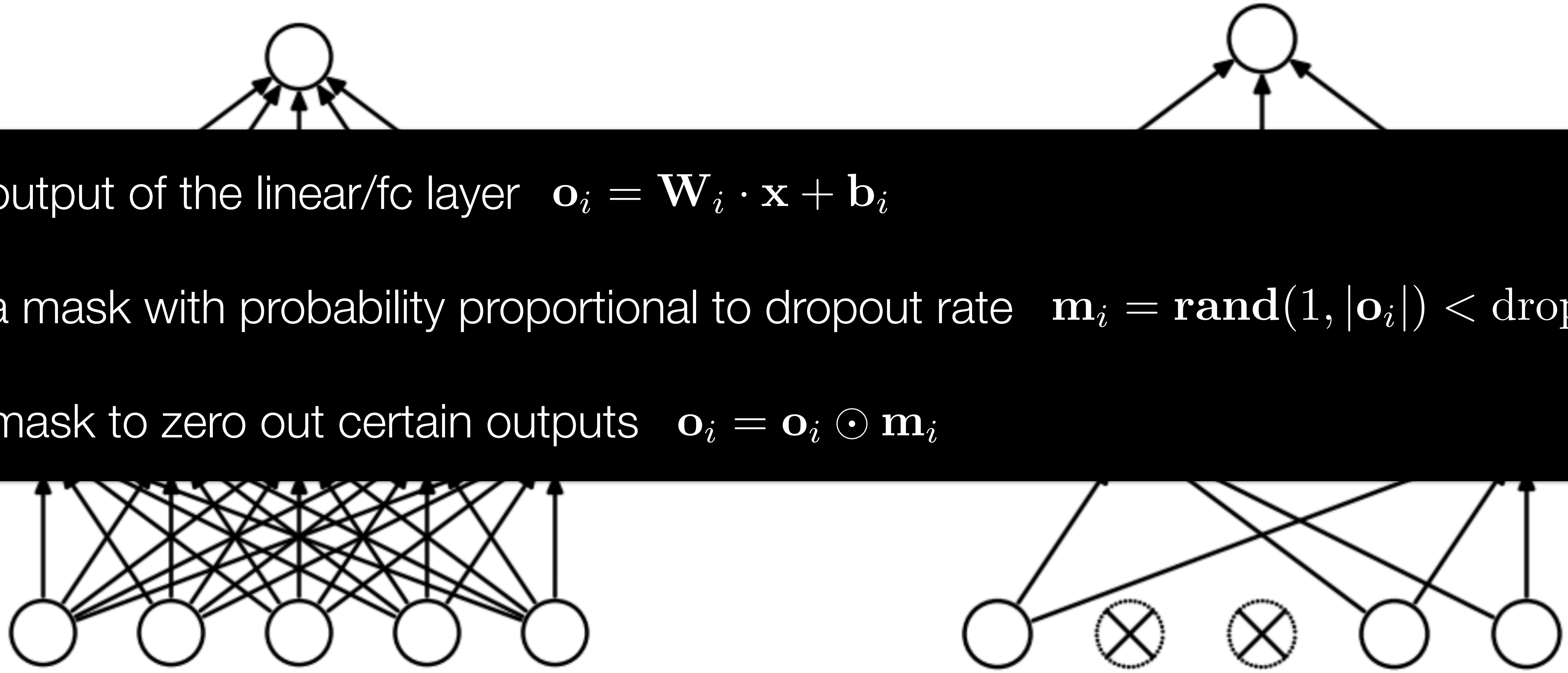


After Applying **Dropout**

[Srivastava et al, JMLR 2014]

Regularization: Dropout

Randomly **set some neurons to zero** in the forward pass, with probability proportional to `dropout rate` (between 0 to 1)



1. Compute output of the linear/fc layer $\mathbf{o}_i = \mathbf{W}_i \cdot \mathbf{x} + \mathbf{b}_i$

2. Compute a mask with probability proportional to dropout rate $\mathbf{m}_i = \mathbf{rand}(1, |\mathbf{o}_i|) < \text{dropout rate}$

3. Apply the mask to zero out certain outputs $\mathbf{o}_i = \mathbf{o}_i \odot \mathbf{m}_i$

Standar Neural Network

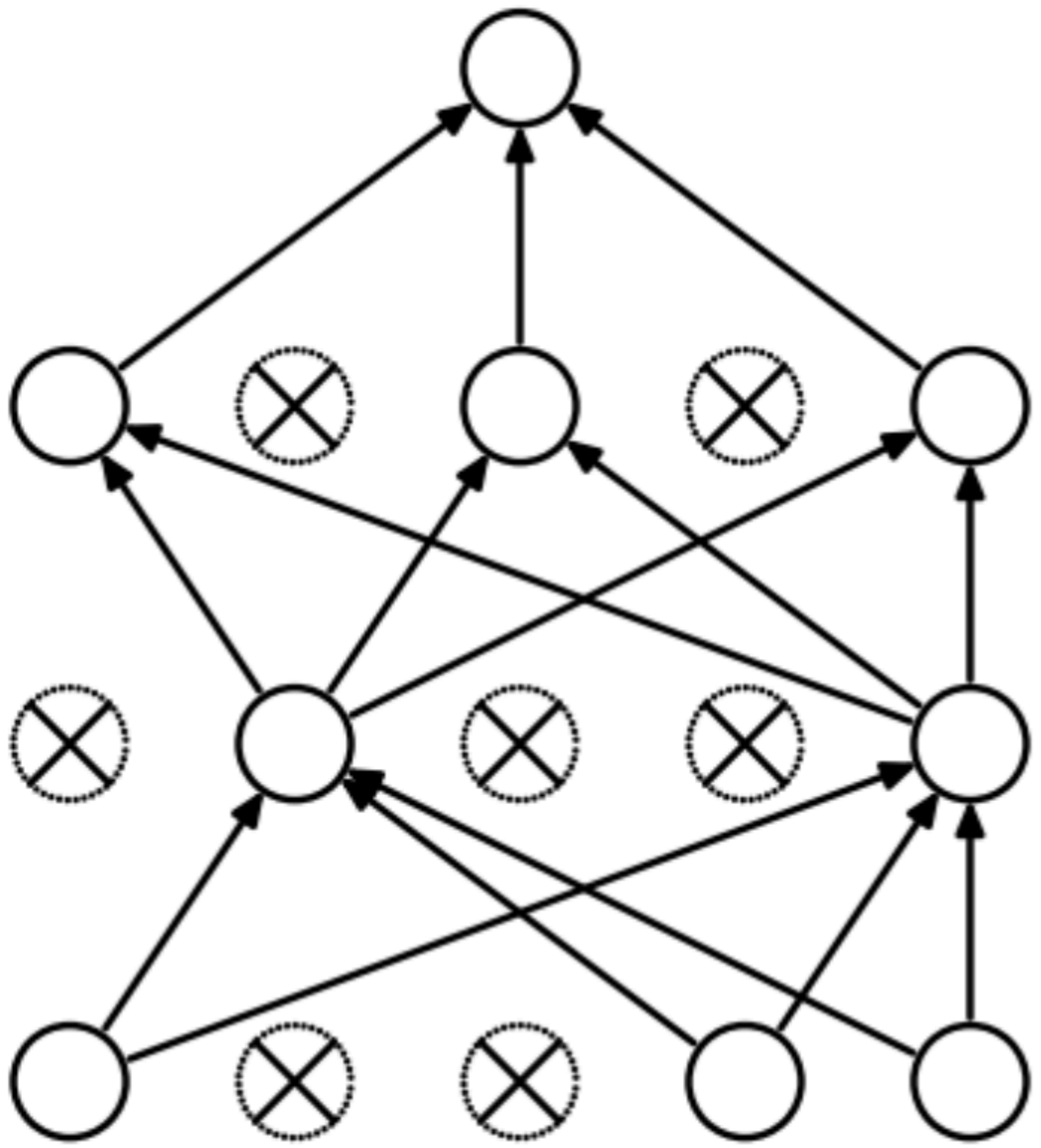
After Applying **Dropout**

[Srivastava et al, JMLR 2014]

* adopted from slides of **CS231n at Stanford**

Regularization: Dropout

Randomly **set some neurons to zero** in the forward pass, with probability proportional to `dropout rate` (between 0 to 1)



After Applying **Dropout**

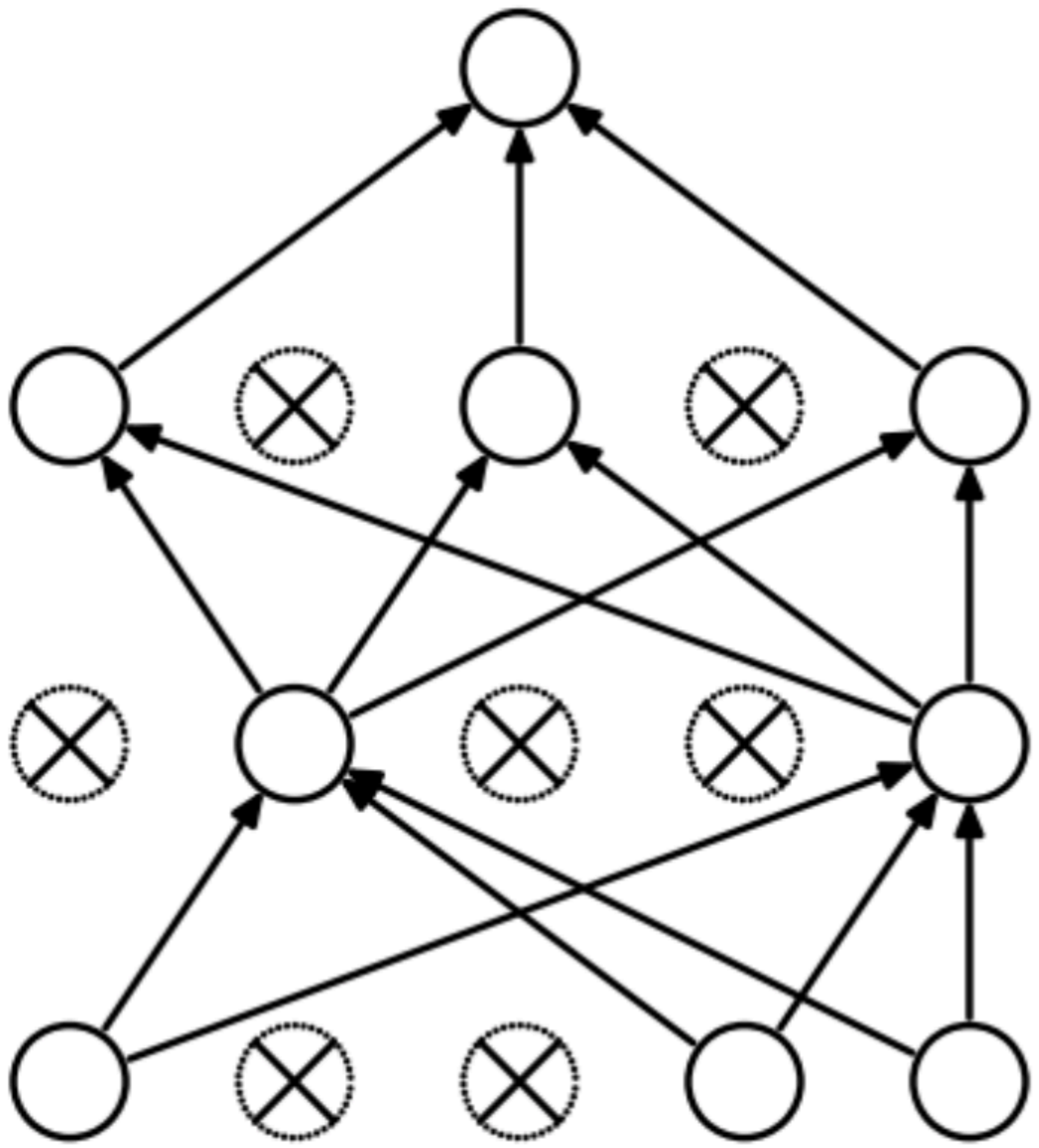
Why is this a good idea?

[Srivastava et al, JMLR 2014]

* adopted from slides of **CS231n at Stanford**

Regularization: Dropout

Randomly **set some neurons to zero** in the forward pass, with probability proportional to `dropout rate` (between 0 to 1)



After Applying **Dropout**

Why is this a good idea?

Dropout is training an **ensemble of models** that share parameters

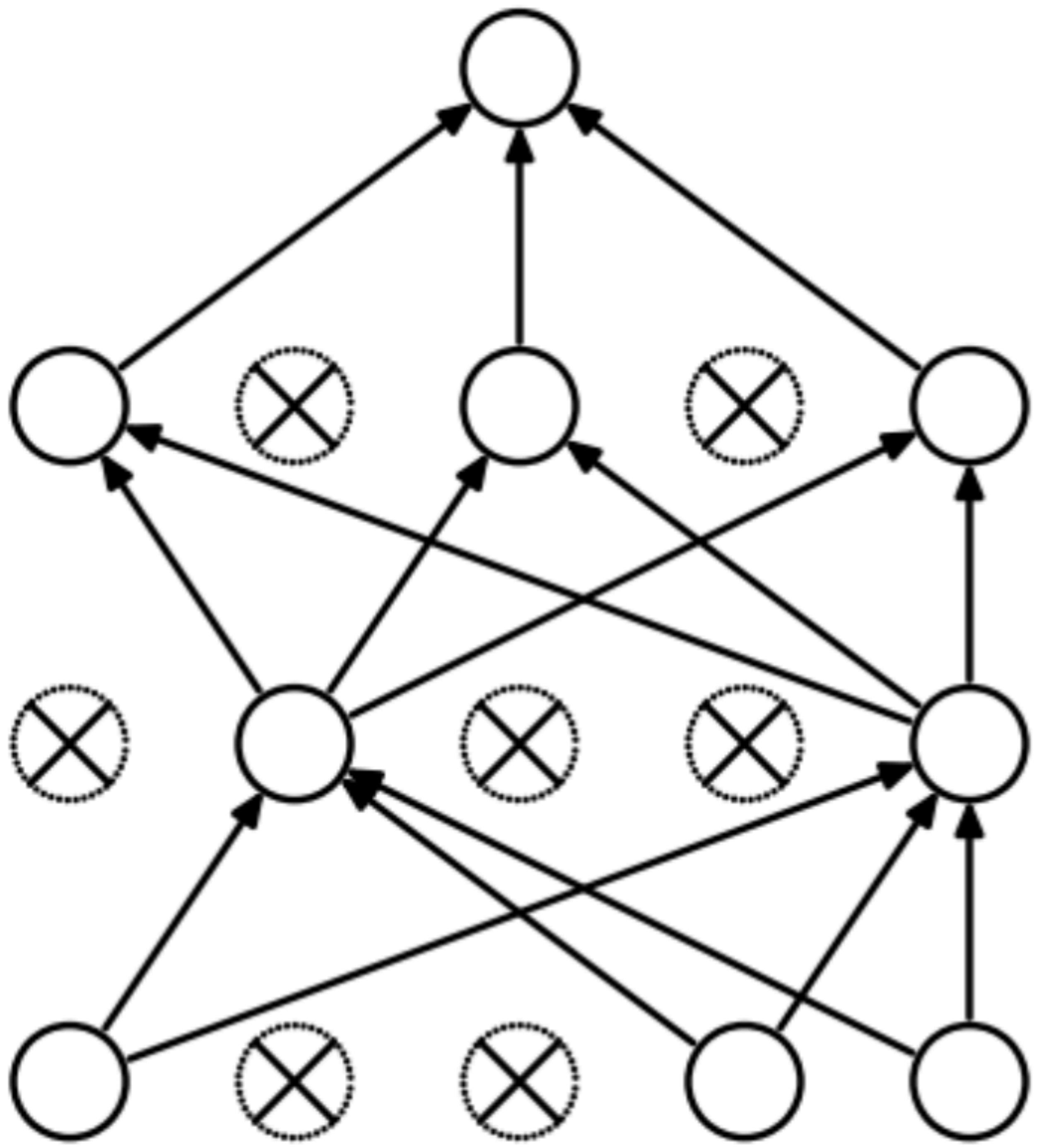
Each binary mask (generated in the forward pass) is one model that is trained on (approximately) one data point

[Srivastava et al, JMLR 2014]

* adopted from slides of **CS231n at Stanford**

Regularization: Dropout (at test time)

Randomly **set some neurons to zero** in the forward pass, with probability proportional to `dropout rate` (between 0 to 1)



After Applying **Dropout**

At test time, **integrate out all the models** in the ensemble

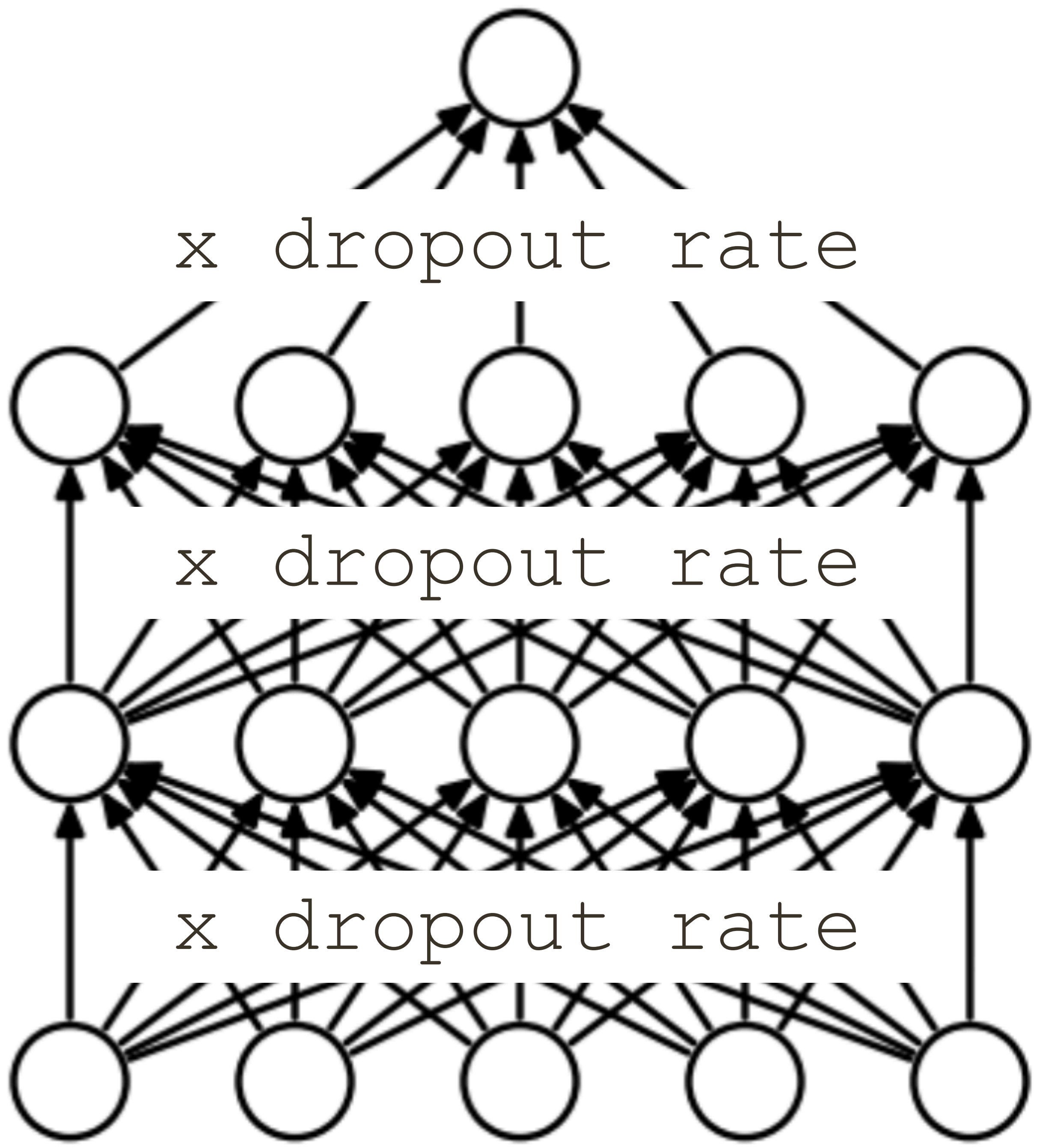
Monte Carlo approximation: many forward passes with different masks and average all predictions

[Srivastava et al, JMLR 2014]

* adopted from slides of **CS231n at Stanford**

Regularization: Dropout (at test time)

Randomly **set some neurons to zero** in the forward pass, with probability proportional to `dropout rate` (between 0 to 1)



At test time, **integrate out all the models** in the ensemble

Monte Carlo approximation: many forward passes with different masks and average all predictions

Equivalent to forward pass with all connections on and **scaling of the outputs** by dropout rate

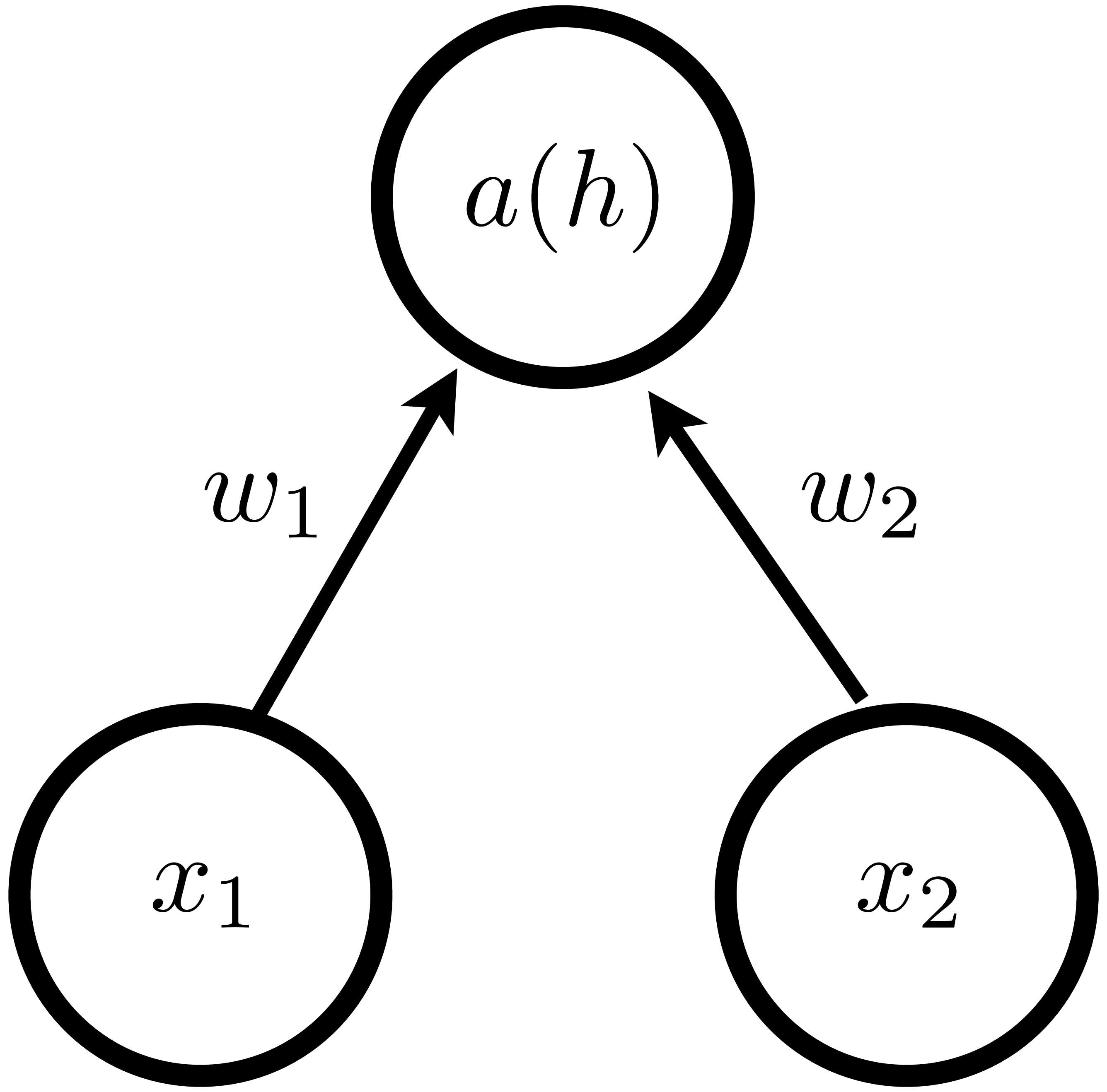
For derivation see Lecture 6 of **CS231n at Stanford**

[Srivastava et al, JMLR 2014]

* adopted from slides of **CS231n at Stanford**

Regularization: Dropout (at test time)

Consider a single neuron

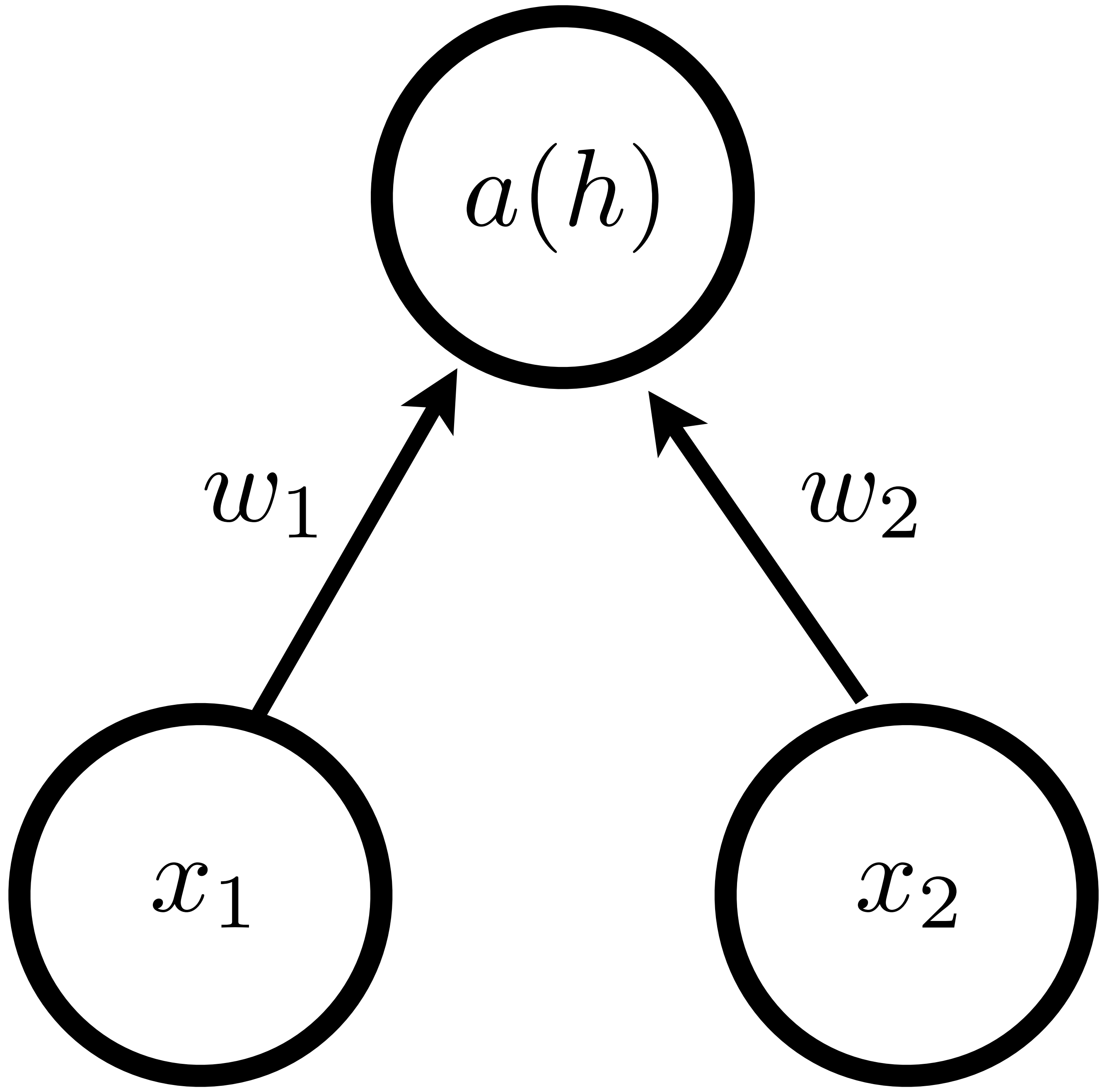


Regularization: Dropout (at test time)

At test time we want to compute **expectation** over input to activation function with respect to exponential number of masks

$$\mathbb{E}_{\mathbf{m}}[h] = \mathbb{E}_{\mathbf{m}}[(\mathbf{W} \cdot \mathbf{x}) \odot \mathbf{m}]$$

Consider a single neuron

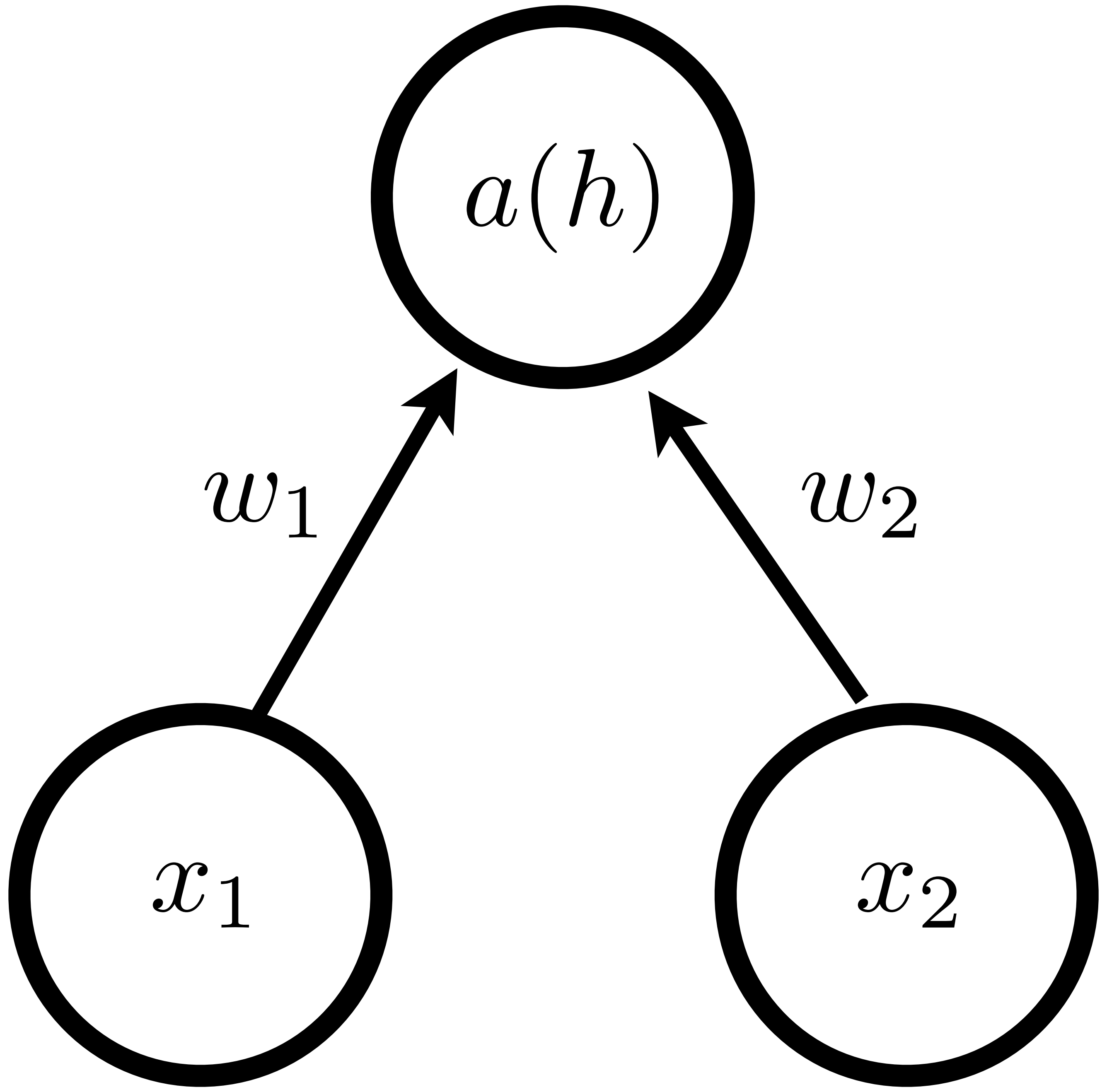


Regularization: Dropout (at test time)

At test time we want to compute **expectation** over input to activation function with respect to exponential number of masks

$$\mathbb{E}_{\mathbf{m}}[h] = \mathbb{E}_{\mathbf{m}}[(\mathbf{W} \cdot \mathbf{x}) \odot \mathbf{m}]$$

Consider a single neuron

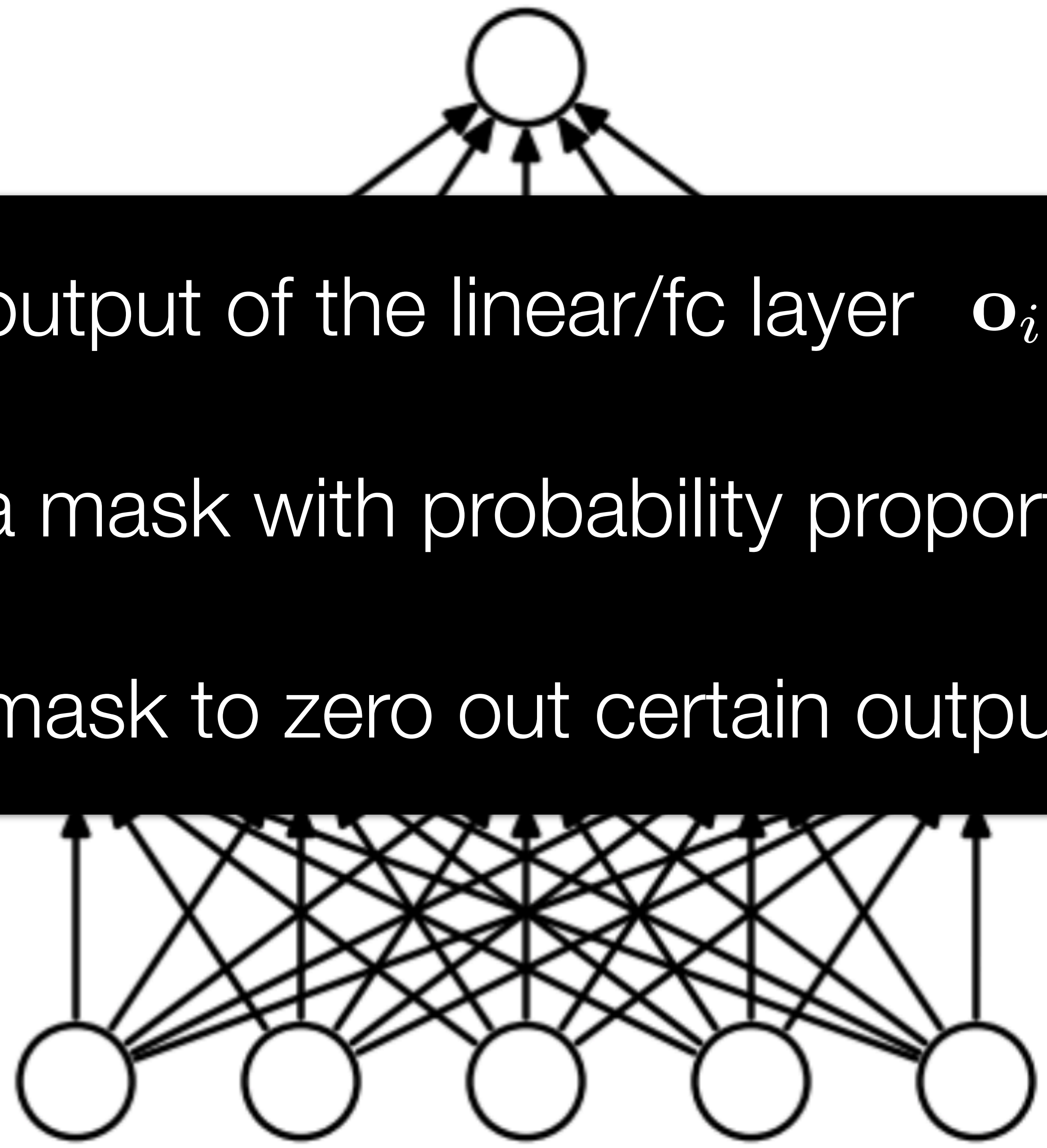


consider dropout rate of p = 0.5

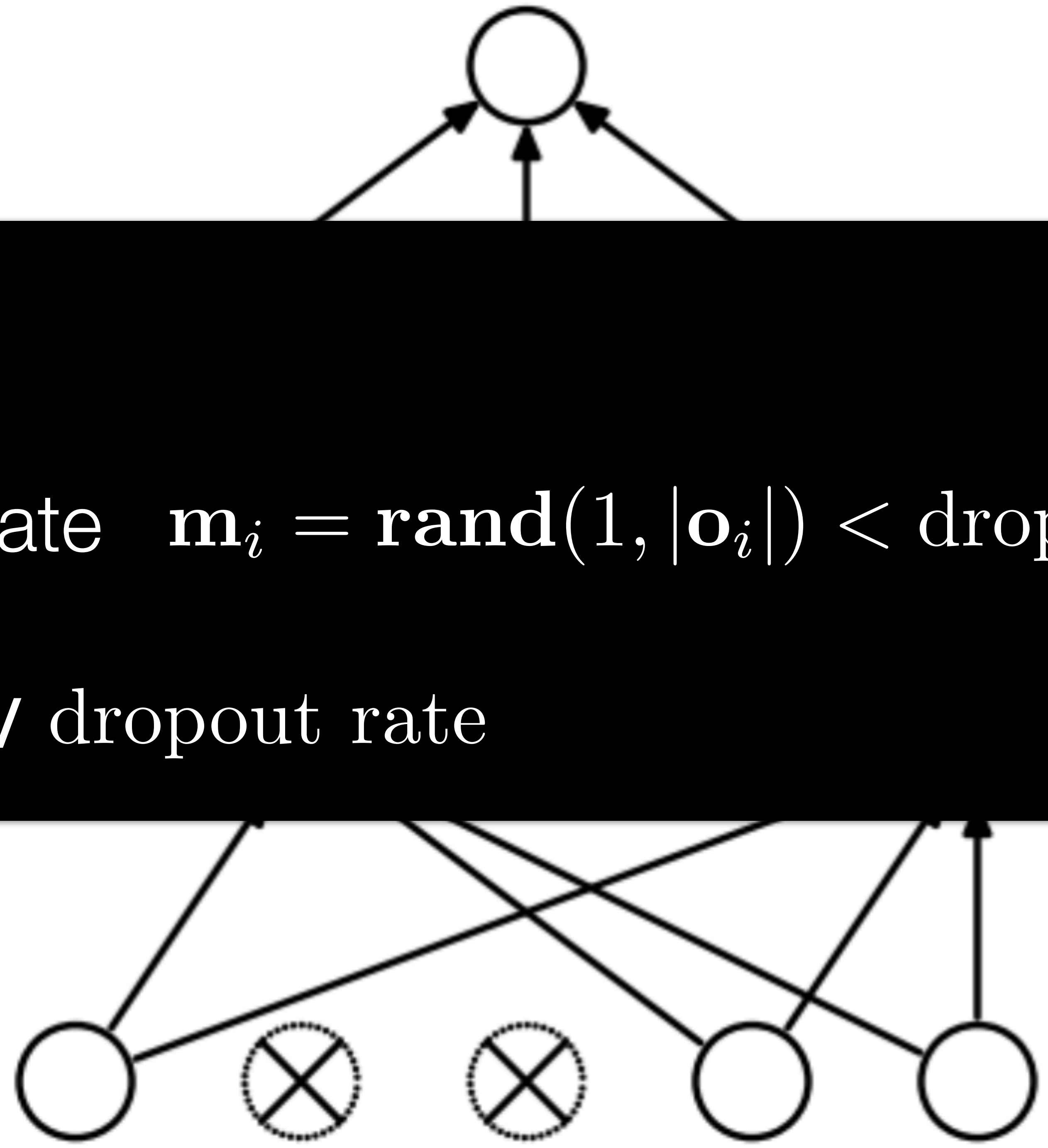
$$\mathbb{E}_{\mathbf{m}}[h] = \mathbb{E}_{(m_1,m_2)}[w_1x_1m_1 + w_2x_2m_2]$$
$$= \frac{1}{4}(w_1x_1 + w_2x_2) + \frac{1}{4}(w_1x_1)\frac{1}{4}(w_2x_2) + \frac{1}{4}(0)$$
$$= \frac{1}{2}(w_1x_1 + w_2x_2)$$

Regularization: Dropout (without change in forward pass)

Randomly **set some neurons to zero** in the forward pass, with probability proportional to `dropout rate` (between 0 to 1)



Standar Neural Network



After Applying **Dropout**

1. Compute output of the linear/fc layer $\mathbf{o}_i = \mathbf{W}_i \cdot \mathbf{x} + \mathbf{b}_i$

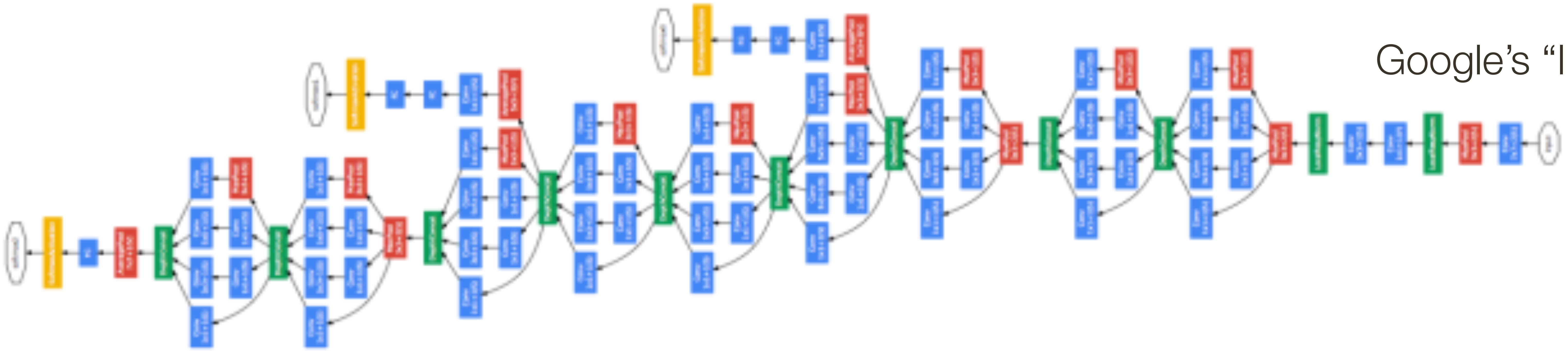
2. Compute a mask with probability proportional to dropout rate $\mathbf{m}_i = \mathbf{rand}(1, |\mathbf{o}_i|) < \text{dropout rate}$

3. Apply the mask to zero out certain outputs $\mathbf{o}_i = \mathbf{o}_i \odot \mathbf{m}_i$ / dropout rate

[Srivastava et al, JMLR 2014]

* adopted from slides of **CS231n at Stanford**

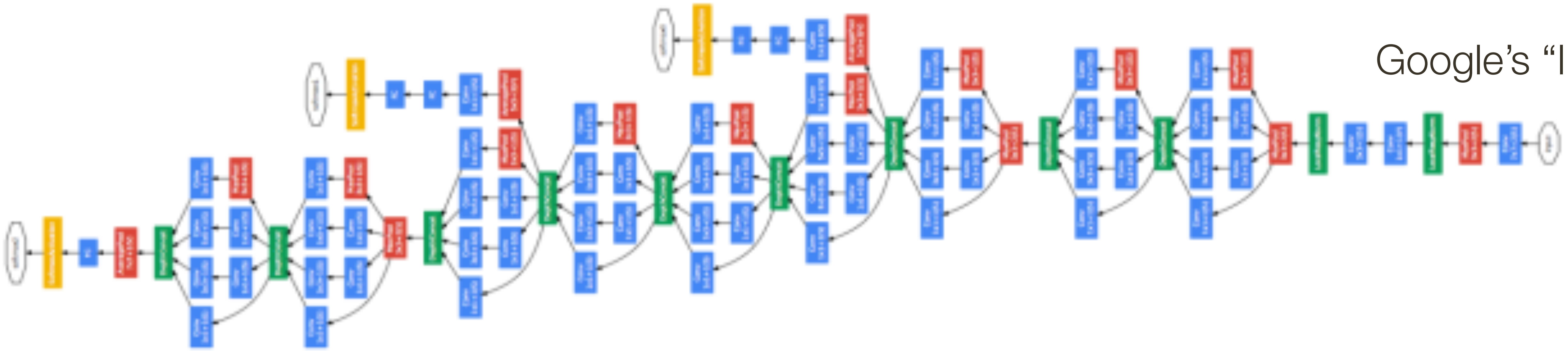
Deep Learning **Terminology**



Google’s “Inception” network

- **Network structure:** number and types of layers, forms of activation functions, dimensionality of each layer and connections (defines computational graph)

Deep Learning **Terminology**

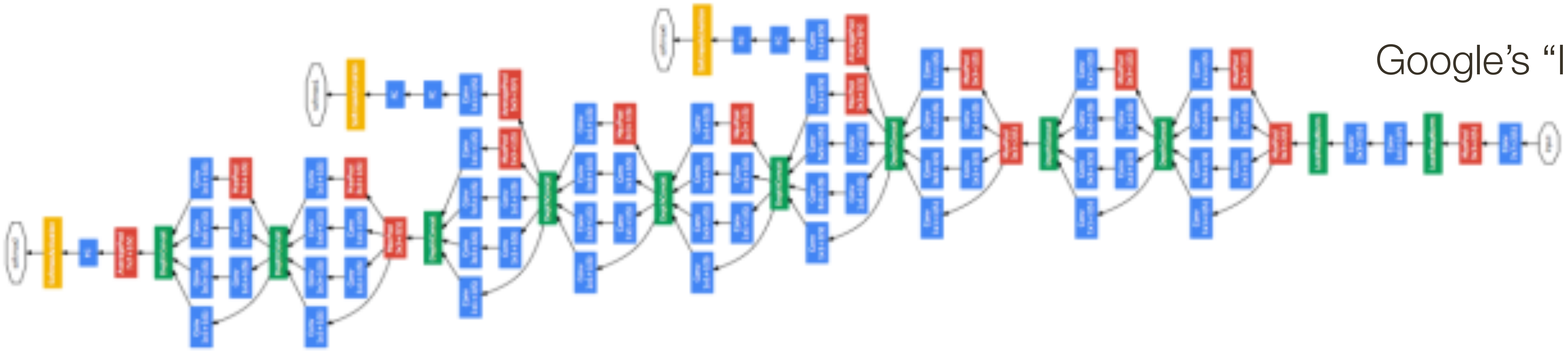


Google's "Inception" network

- **Network structure:** number and types of layers, forms of activation functions, dimensionality of each layer and connections (defines computational graph)

generally kept fixed, requires some knowledge of the problem and NN to sensibly set

Deep Learning **Terminology**



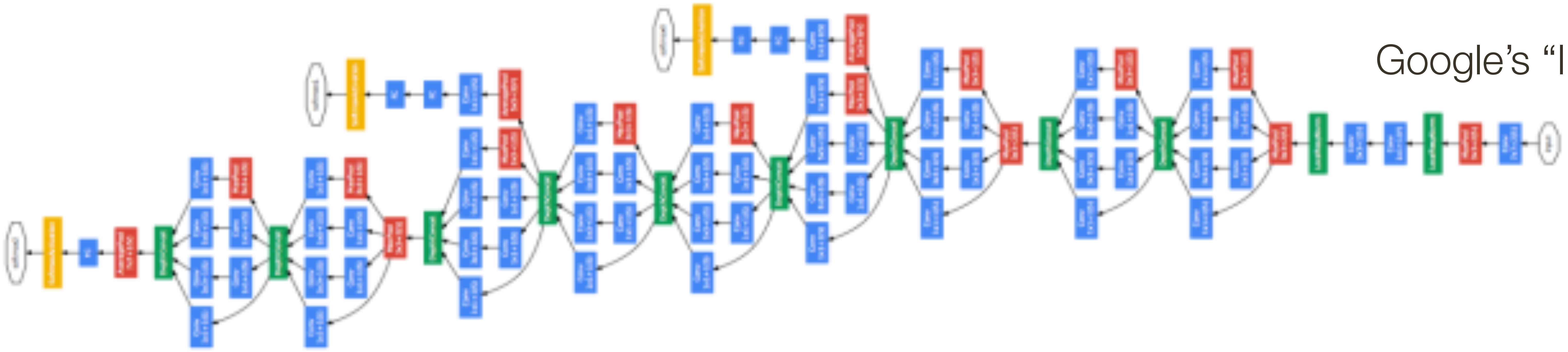
Google's "Inception" network

- **Network structure:** number and types of layers, forms of activation functions, dimensionality of each layer and connections (defines computational graph)

generally kept fixed, requires some knowledge of the problem and NN to sensibly set

deeper = better

Deep Learning **Terminology**



Google's "Inception" network

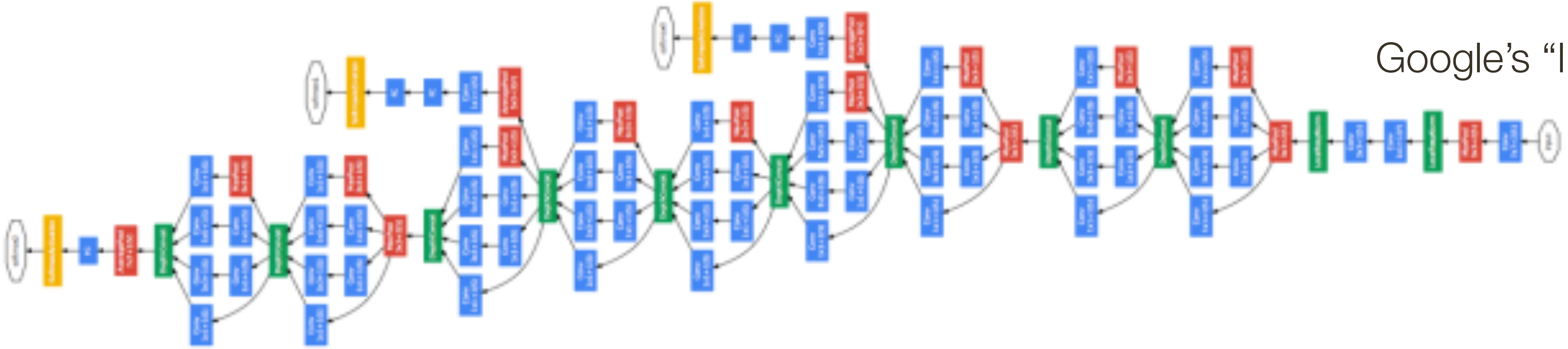
- **Network structure:** number and types of layers, forms of activation functions, dimensionality of each layer and connections (defines computational graph)

generally kept fixed, requires some knowledge of the problem and NN to sensibly set

deeper = better

- **Loss function:** objective function being optimized (`softmax`, `cross entropy`, *etc.*)

Deep Learning **Terminology**



Google's "Inception" network

- **Network structure:** number and types of layers, forms of activation functions, dimensionality of each layer and connections (defines computational graph)

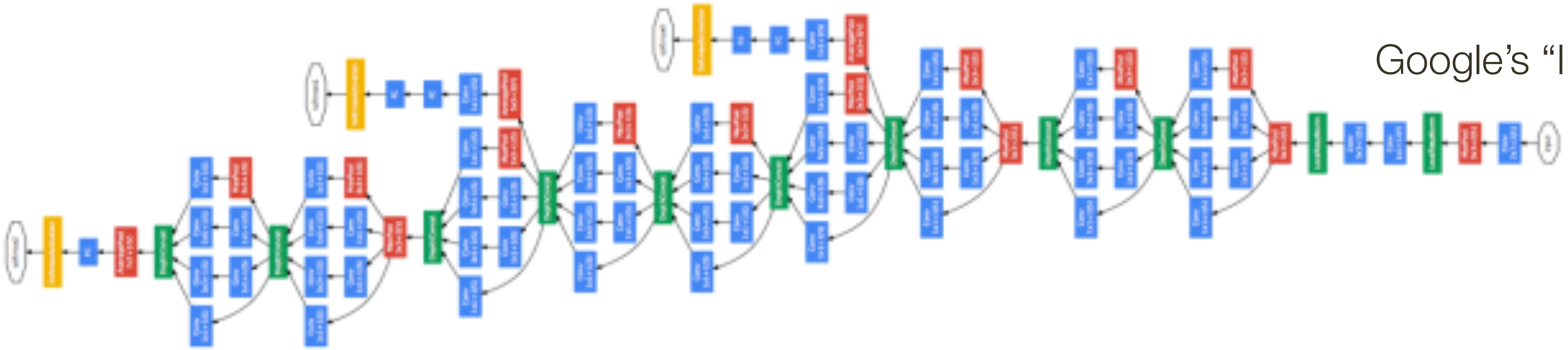
generally kept fixed, requires some knowledge of the problem and NN to sensibly set

deeper = better

- **Loss function:** objective function being optimized (`softmax`, `cross entropy`, *etc.*)

requires knowledge of the nature of the problem

Deep Learning **Terminology**



Google's "Inception" network

- **Network structure:** number and types of layers, forms of activation functions, dimensionality of each layer and connections (defines computational graph)

generally kept fixed, requires some knowledge of the problem and NN to sensibly set

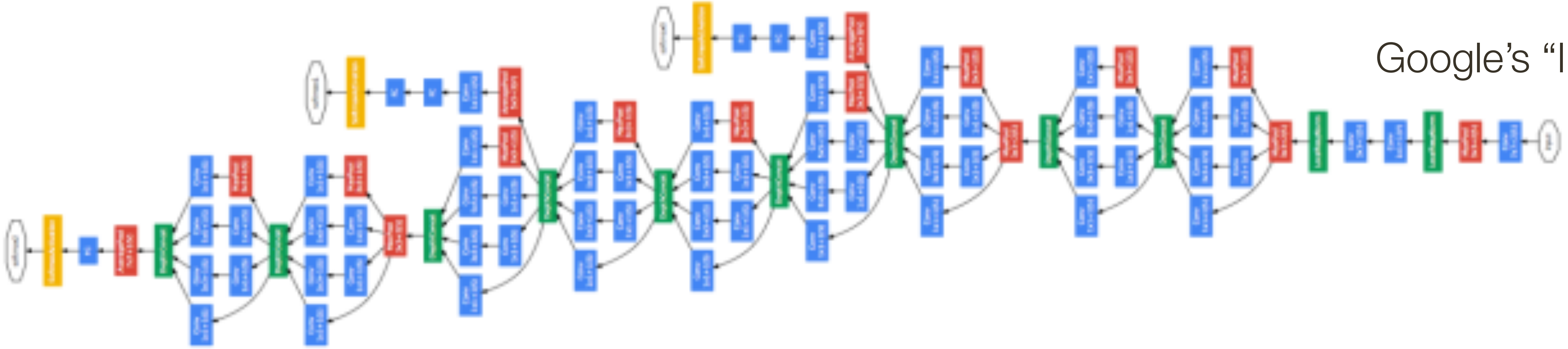
deeper = better

- **Loss function:** objective function being optimized (`softmax`, `cross entropy`, *etc.*)

requires knowledge of the nature of the problem

- **Parameters:** trainable parameters of the network, including weights/biases of linear/fc layers, parameters of the activation functions, *etc.*

Deep Learning **Terminology**



Google's "Inception" network

- **Network structure:** number and types of layers, forms of activation functions, dimensionality of each layer and connections (defines computational graph)

generally kept fixed, requires some knowledge of the problem and NN to sensibly set

deeper = better

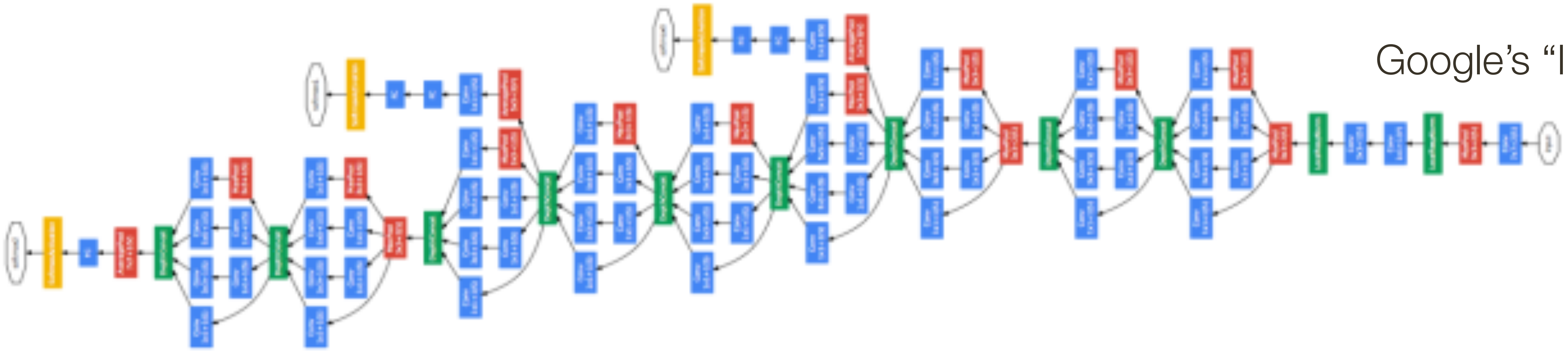
- **Loss function:** objective function being optimized (`softmax`, `cross entropy`, *etc.*)

requires knowledge of the nature of the problem

- **Parameters:** trainable parameters of the network, including weights/biases of linear/fc layers, parameters of the activation functions, *etc.*

optimized using SGD or variants

Deep Learning **Terminology**



Google's "Inception" network

- **Network structure:** number and types of layers, forms of activation functions, dimensionality of each layer and connections (defines computational graph)

generally kept fixed, requires some knowledge of the problem and NN to sensibly set

deeper = better

- **Loss function:** objective function being optimized (`softmax`, `cross entropy`, *etc.*)

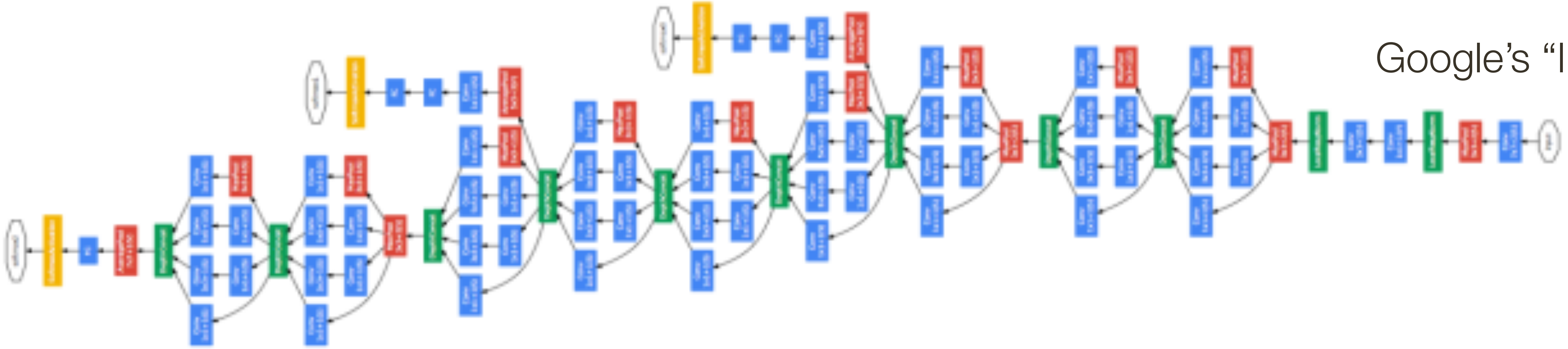
requires knowledge of the nature of the problem

- **Parameters:** trainable parameters of the network, including weights/biases of linear/fc layers, parameters of the activation functions, *etc.*

optimized using SGD or variants

- **Hyper-parameters:** parameters, including for optimization, that are not optimized directly as part of training (*e.g.*, `learning rate`, `batch size`, `drop-out rate`)

Deep Learning **Terminology**



Google's "Inception" network

- **Network structure:** number and types of layers, forms of activation functions, dimensionality of each layer and connections (defines computational graph)

generally kept fixed, requires some knowledge of the problem and NN to sensibly set

deeper = better

- **Loss function:** objective function being optimized (`softmax`, `cross entropy`, *etc.*)

requires knowledge of the nature of the problem

- **Parameters:** trainable parameters of the network, including weights/biases of linear/fc layers, parameters of the activation functions, *etc.*

optimized using SGD or variants

- **Hyper-parameters:** parameters, including for optimization, that are not optimized directly as part of training (*e.g.*, `learning rate`, `batch size`, `drop-out rate`)

grid search

Loss Functions …

This is where all the **fun** is … we will only look a most common ones

Multivariate **Regression**

Input: feature vector $\mathbf{x} \in \mathbb{R}^n$

Output: output vector $\mathbf{y} \in \mathbb{R}^m$

Multivariate **Regression**

Input: feature vector $\mathbf{x} \in \mathbb{R}^n$

Output: output vector $\mathbf{y} \in \mathbb{R}^m$

Neural Network (input + intermediate hidden layers) $f(\mathbf{x}; \Theta) : \mathbb{R}^n \rightarrow \mathbb{R}^k$

with **sigmoid** activations: $\mathbf{0} \leq f(\mathbf{x}; \Theta) \leq \mathbf{1}$

with **Tanh** activations: $-\mathbf{1} \leq f(\mathbf{x}; \Theta) \leq \mathbf{1}$

with **ReLU** activations: $\mathbf{0} \leq f(\mathbf{x}; \Theta)$

Multivariate **Regression**

Input: feature vector $\mathbf{x} \in \mathbb{R}^n$

Output: output vector $\mathbf{y} \in \mathbb{R}^m$

Neural Network (input + intermediate hidden layers) $f(\mathbf{x};\Theta) : \mathbb{R}^n \rightarrow \mathbb{R}^k$

with **sigmoid** activations: $\mathbf{0} \leq f(\mathbf{x};\Theta) \leq \mathbf{1}$

with **Tanh** activations: $-\mathbf{1} \leq f(\mathbf{x};\Theta) \leq \mathbf{1}$

with **ReLU** activations: $\mathbf{0} \leq f(\mathbf{x};\Theta)$

Neural Network (output): linear layer

$$\hat{\mathbf{y}} = g(\mathbf{x};\mathbf{W},\mathbf{b}) = \mathbf{W}f(\mathbf{x};\Theta) + \mathbf{b} : \mathbb{R}^k \rightarrow \mathbb{R}^m$$

Multivariate **Regression**

Input: feature vector $\mathbf{x} \in \mathbb{R}^n$

Output: output vector $\mathbf{y} \in \mathbb{R}^m$

Neural Network (input + intermediate hidden layers) $f(\mathbf{x}; \Theta) : \mathbb{R}^n \to \mathbb{R}^k$

with **sigmoid** activations: $\mathbf{0} \leq f(\mathbf{x}; \Theta) \leq \mathbf{1}$

with **Tanh** activations: $-\mathbf{1} \leq f(\mathbf{x}; \Theta) \leq \mathbf{1}$

with **ReLU** activations: $\mathbf{0} \leq f(\mathbf{x}; \Theta)$

Neural Network (output): linear layer

$$\hat{\mathbf{y}} = g(\mathbf{x}; \mathbf{W}, \mathbf{b}) = \mathbf{W} f(\mathbf{x}; \Theta) + \mathbf{b} : \mathbb{R}^k \to \mathbb{R}^m$$

Loss:

$$\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) = ||\mathbf{y} - \hat{\mathbf{y}}||^2$$

Binary Classification (Bernoulli)

Input: feature vector $\mathbf{x} \in \mathbb{R}^n$

Output: binary label $y \in \{0, 1\}$

Neural Network (input + intermediate hidden layers) $f(\mathbf{x}; \Theta) : \mathbb{R}^n \to \mathbb{R}$

with **sigmoid** activations: $\mathbf{0} \leq f(\mathbf{x}; \Theta) \leq \mathbf{1}$

Binary Classification (Bernoulli)

Input: feature vector $\mathbf{x} \in \mathbb{R}^n$

Output: binary label $y \in \{0, 1\}$

Neural Network (input + intermediate hidden layers) $f(\mathbf{x}; \Theta) : \mathbb{R}^n \to \mathbb{R}$

with **sigmoid** activations: $\mathbf{0} \leq f(\mathbf{x}; \Theta) \leq \mathbf{1}$

Neural Network (output): threshold hidden output (which is a sigmoid)

$$\hat{y} = 1[f(\mathbf{x}; \Theta) > 0.5]$$

Binary Classification (Bernoulli)

Input: feature vector $\mathbf{x} \in \mathbb{R}^n$

Output: binary label $y \in \{0, 1\}$

Neural Network (input + intermediate hidden layers) $f(\mathbf{x}; \Theta) : \mathbb{R}^n \to \mathbb{R}$

with **sigmoid** activations: $\mathbf{0} \leq f(\mathbf{x}; \Theta) \leq \mathbf{1}$

Neural Network (output): threshold hidden output (which is a sigmoid)

$$\hat{y} = 1[f(\mathbf{x}; \Theta) > 0.5]$$

Problem: Not differentiable, probabilistic interpretation maybe desirable

Binary Classification (Bernoulli)

Input: feature vector $\mathbf{x} \in \mathbb{R}^n$

Output: binary label $y \in \{0, 1\}$

Neural Network (input + intermediate hidden layers) $f(\mathbf{x}; \Theta) : \mathbb{R}^n \to \mathbb{R}$

with **sigmoid** activations: $\mathbf{0} \leq f(\mathbf{x}; \Theta) \leq \mathbf{1}$

Neural Network (output): interpret sigmoid output as probability

$$p(y = 1) = f(\mathbf{x}; \Theta)$$

can interpret the score as the log-odds of $y = 1$ (a.k.a. the **logits**)

Binary Classification (Bernoulli)

Input: feature vector $\mathbf{x} \in \mathbb{R}^n$

Output: binary label $y \in \{0, 1\}$

Neural Network (input + intermediate hidden layers) $f(\mathbf{x}; \Theta) : \mathbb{R}^n \to \mathbb{R}$

with **sigmoid** activations: $\mathbf{0} \leq f(\mathbf{x}; \Theta) \leq \mathbf{1}$

Neural Network (output): interpret sigmoid output as probability

$$p(y = 1) = f(\mathbf{x}; \Theta)$$

can interpret the score as the log-odds of $y = 1$ (a.k.a. the **logits**)

Loss: similarity between two distributions

Binary Classification (Bernoulli)

Input: feature vector $\mathbf{x} \in \mathbb{R}^n$

Output: binary label $y \in \{0, 1\}$

We can measure similarity between distribution $p(x)$ and $q(x)$ using cross-entropy

$$H(p, q) = -\mathbb{E}_{x \sim p}[\log q(x)]$$

For discrete distributions this ends up being:

$$H(p, q) = -\sum_{x} p(x) \log q(x)$$

Loss: similarity between two distributions

Binary Classification (Bernoulli)

Input: feature vector $\mathbf{x} \in \mathbb{R}^n$

Output: binary label $y \in \{0, 1\}$

Neural Network (input + intermediate hidden layers) $f(\mathbf{x}; \Theta) : \mathbb{R}^n \to \mathbb{R}$

with **sigmoid** activations: $\mathbf{0} \leq f(\mathbf{x}; \Theta) \leq \mathbf{1}$

Neural Network (output): interpret sigmoid output as probability

$$p(y = 1) = f(\mathbf{x}; \Theta)$$

can interpret the score as the log-odds of $y = 1$ (a.k.a. the **logits**)

Loss: $\mathcal{L}(y, \hat{y}) = -y \log[f(\mathbf{x}; \Theta)] - (1 - y) \log[1 - f(\mathbf{x}; \Theta)]$

Binary Classification (Bernoulli)

Input: feature vector $\mathbf{x} \in \mathbb{R}^n$

Output: binary label $y \in \{0, 1\}$

Neural Network (input + intermediate hidden layers) $f(\mathbf{x}; \Theta) : \mathbb{R}^n \rightarrow \mathbb{R}$

with **sigmoid** activations: $\mathbf{0} \leq f(\mathbf{x}; \Theta) \leq \mathbf{1}$

Neural Network (output): interpret sigmoid output as probability

$$p(y = 1) = f(\mathbf{x}; \Theta)$$

can interpret the score as the log-odds of $y = 1$ (a.k.a. the **logits**)

Loss:

$$\mathcal{L}(y, \hat{y}) = \begin{cases} -log[1 - f(\mathbf{x}; \Theta)] & y = 0 \\ -log[f(\mathbf{x}; \Theta)] & y = 1 \end{cases}$$

Binary Classification (Bernoulli)

Input: feature vector $\mathbf{x} \in \mathbb{R}^n$

Output: binary label $y \in \{0, 1\}$

Neural Network (input + intermediate hidden layers) $f(\mathbf{x}; \Theta) : \mathbb{R}^n \to \mathbb{R}$

with **sigmoid** activations: $\mathbf{0} \leq f(\mathbf{x}; \Theta) \leq \mathbf{1}$

Neural Network (output): interpret sigmoid output as probability

$$p(y = 1) = f(\mathbf{x}; \Theta)$$

Minimizing this **loss** is the same as maximizing **log likelihood** of data

Loss:

$$\mathcal{L}(y, \hat{y}) = \begin{cases} -log[1 - f(\mathbf{x}; \Theta)] & y = 0 \\ -log[f(\mathbf{x}; \Theta)] & y = 1 \end{cases}$$

Binary Classification (Bernoulli)

Input: feature vector $\mathbf{x} \in \mathbb{R}^n$

Output: binary label $y \in \{0, 1\}$

Neural Network (input + intermediate hidden layers) $f(\mathbf{x}; \Theta) : \mathbb{R}^n \to \mathbb{R}^k$

with **ReLU** activations: $\mathbf{0} \leq f(\mathbf{x}; \Theta)$

Binary Classification (Bernoulli)

Input: feature vector $\mathbf{x} \in \mathbb{R}^n$

Output: binary label $y \in \{0, 1\}$

Neural Network (input + intermediate hidden layers) $f(\mathbf{x}; \Theta) : \mathbb{R}^n \to \mathbb{R}^k$

with **ReLU** activations: $\mathbf{0} \leq f(\mathbf{x}; \Theta)$

Neural Network (output): linear layer with one neuron and sigmoid activation