

Chapter 5-2a:

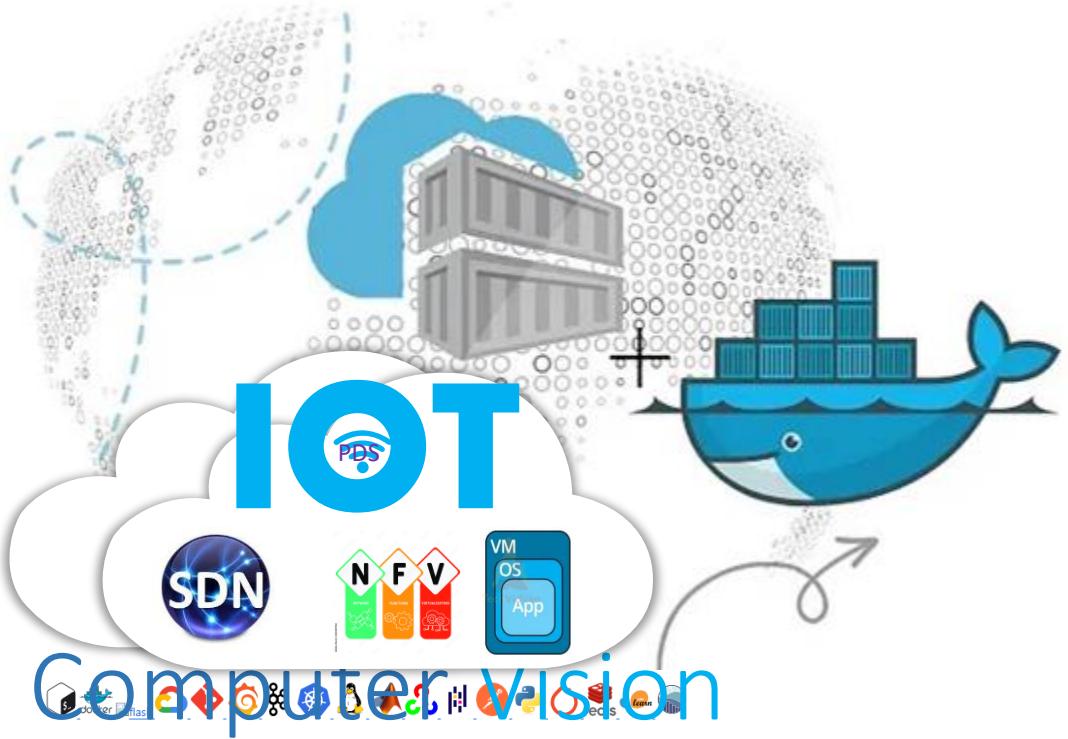
Neural Network

https://github.com/amelcharolinesgn2/IoT_simulator-mqtt-NodeRed

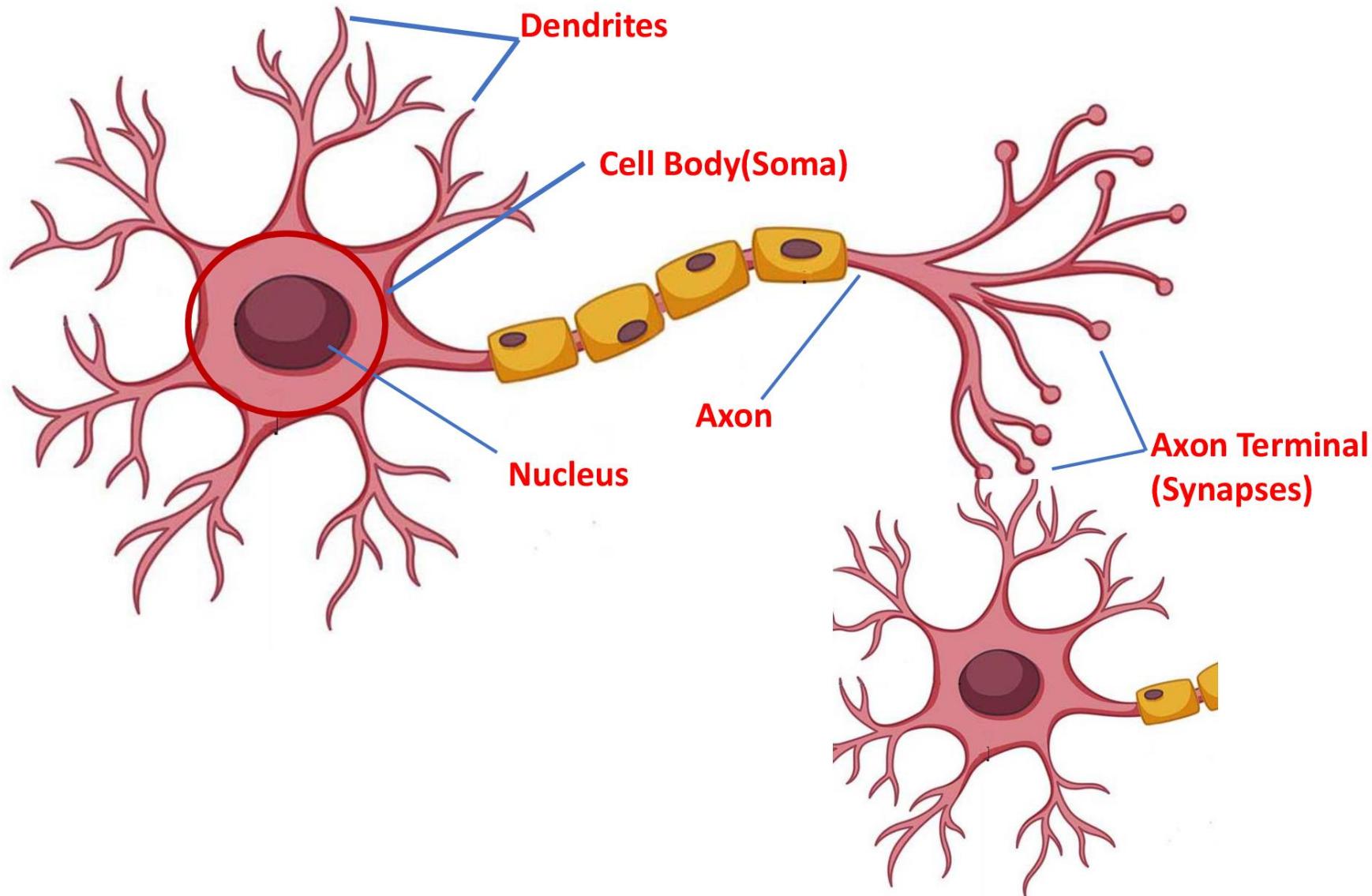
<https://github.com/siagianp>
<https://www.youtube.com/@AmelOline/videos>

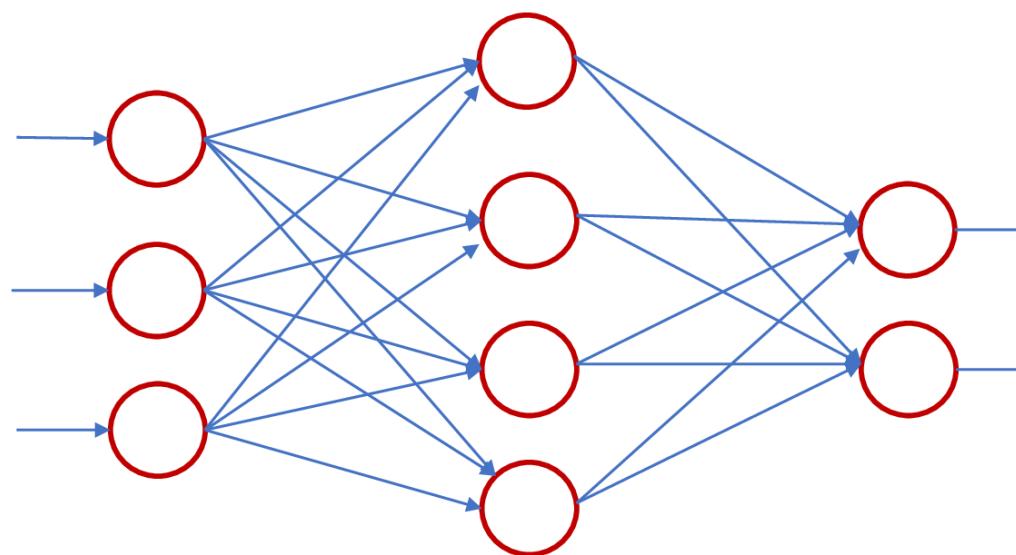
Outlines

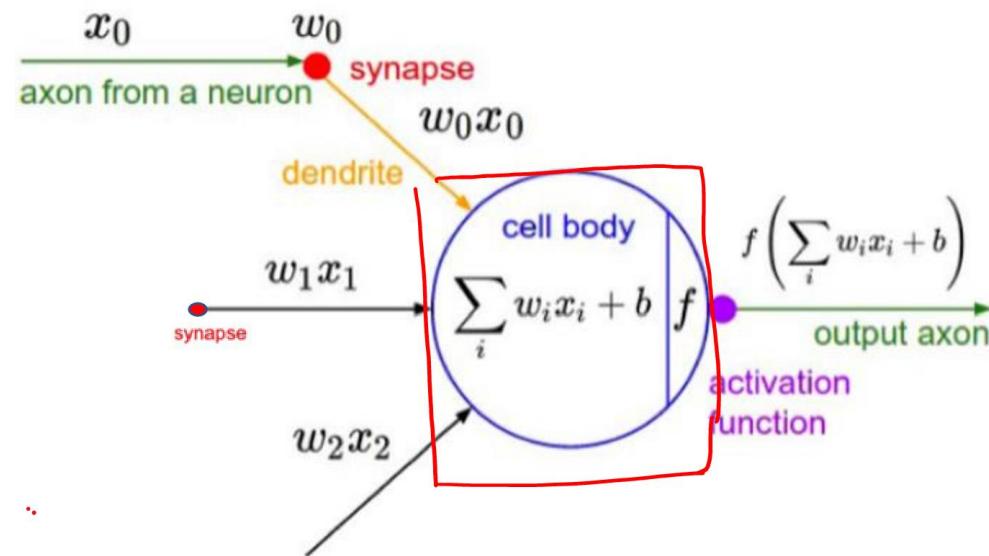
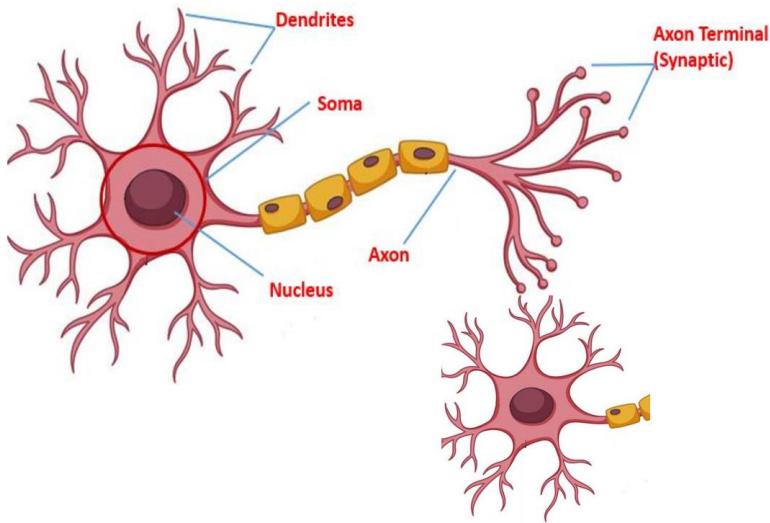
- Network of Neuron in Brain
- Neural Networks
- Training Neural Networks
- Mini Batch SGD

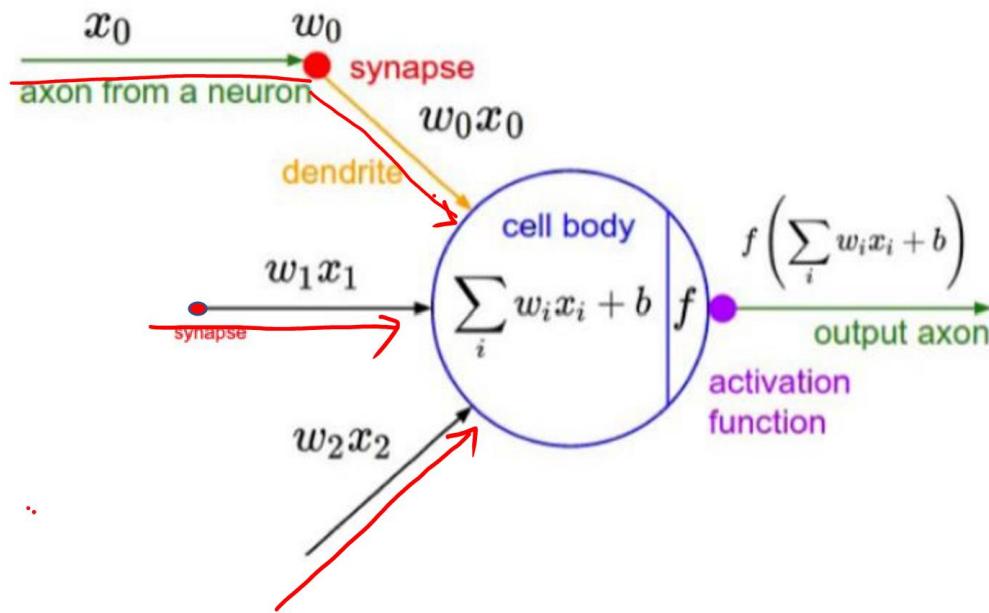
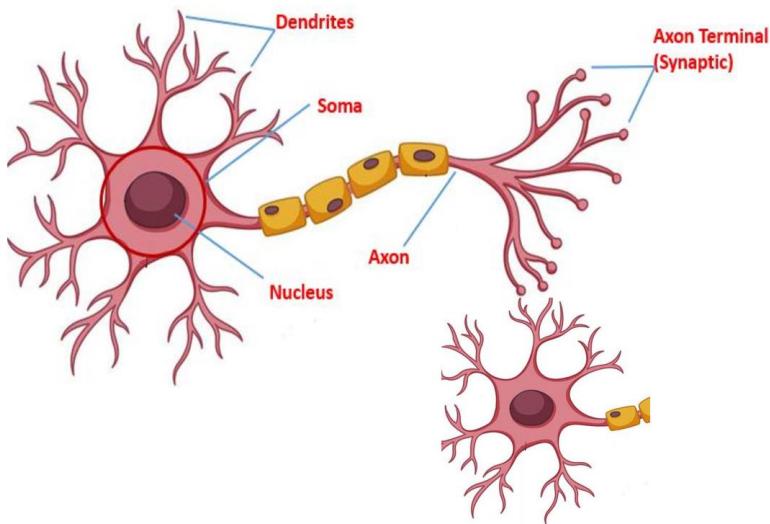


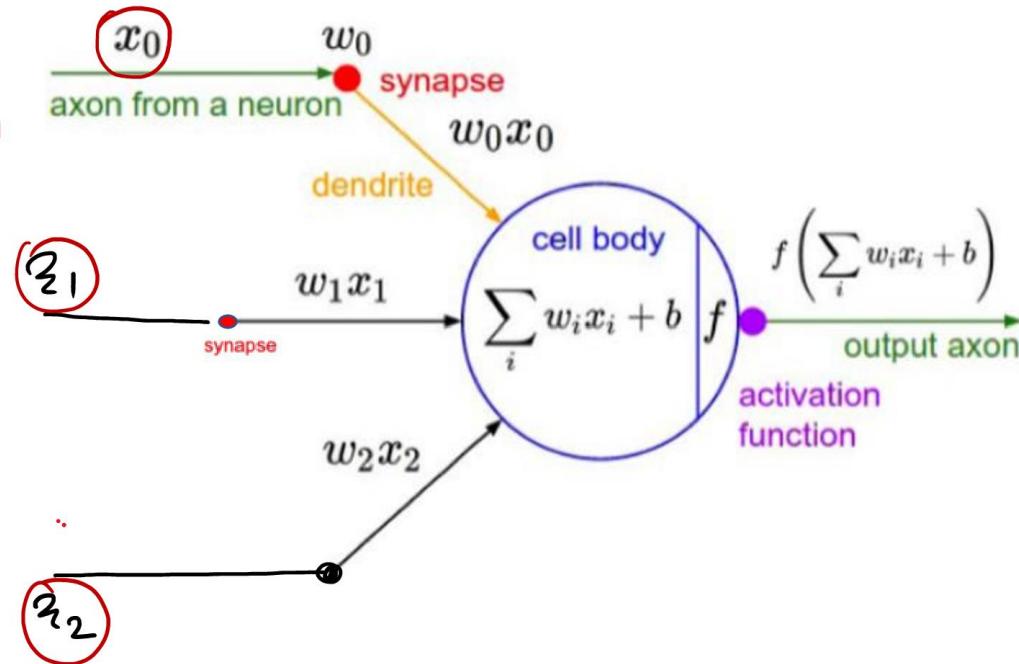
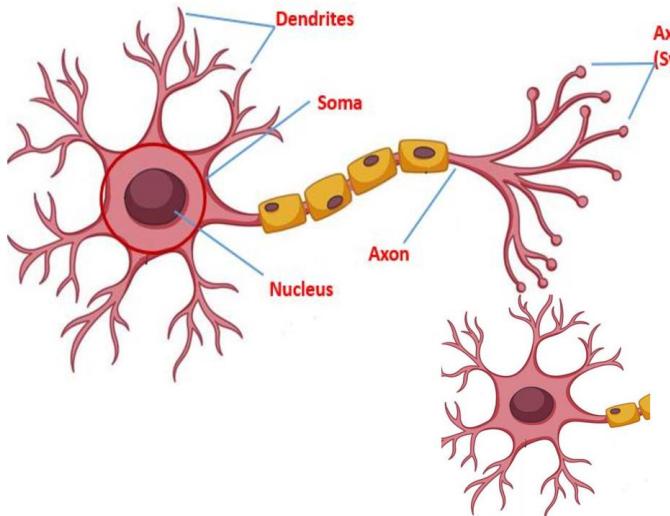
Network of Neuron in Brain

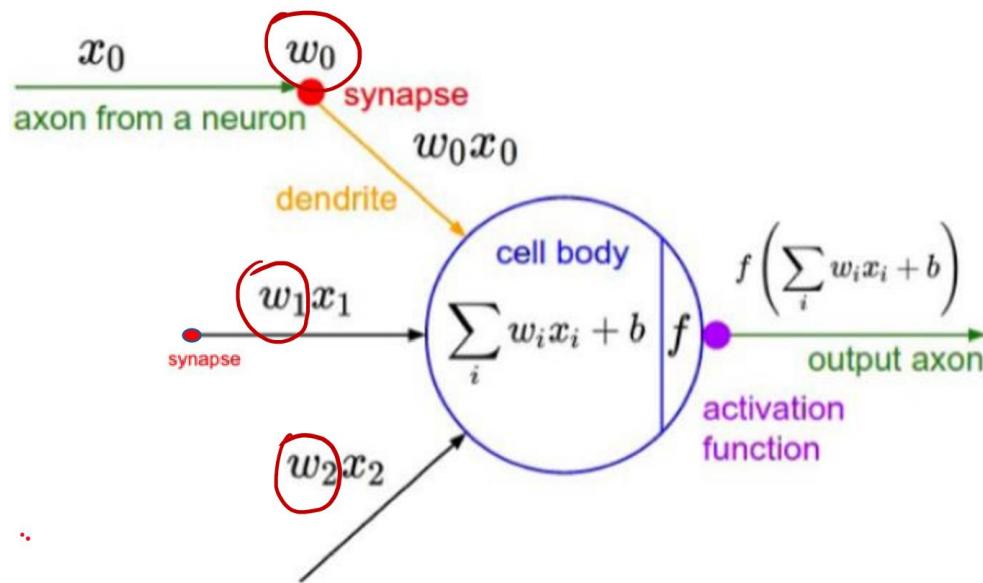
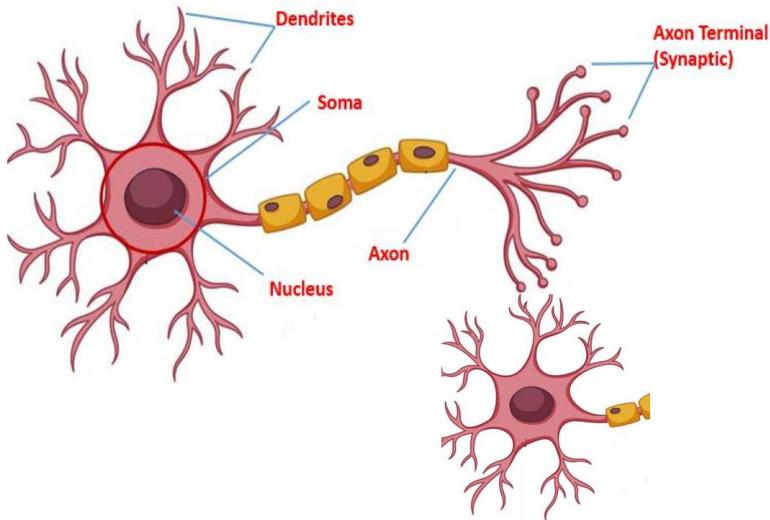


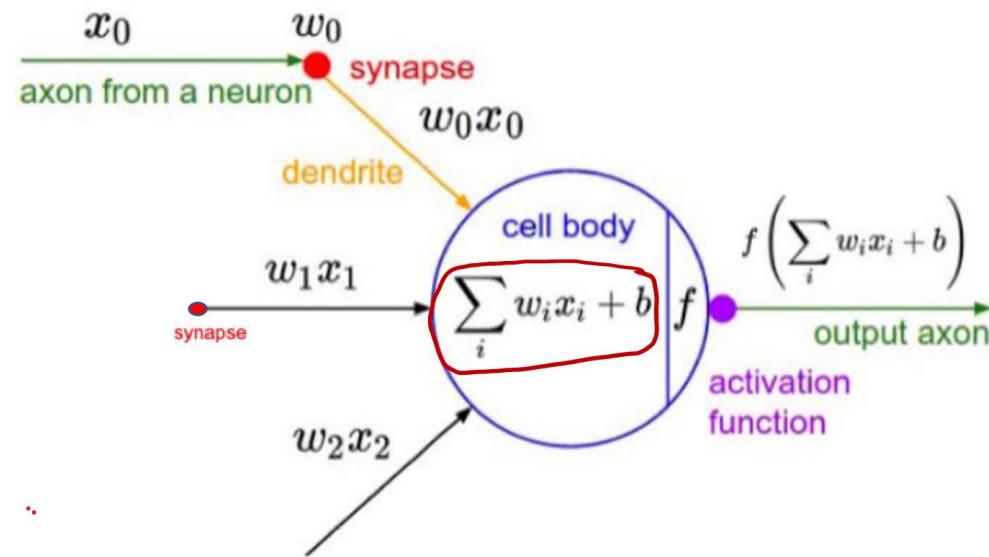
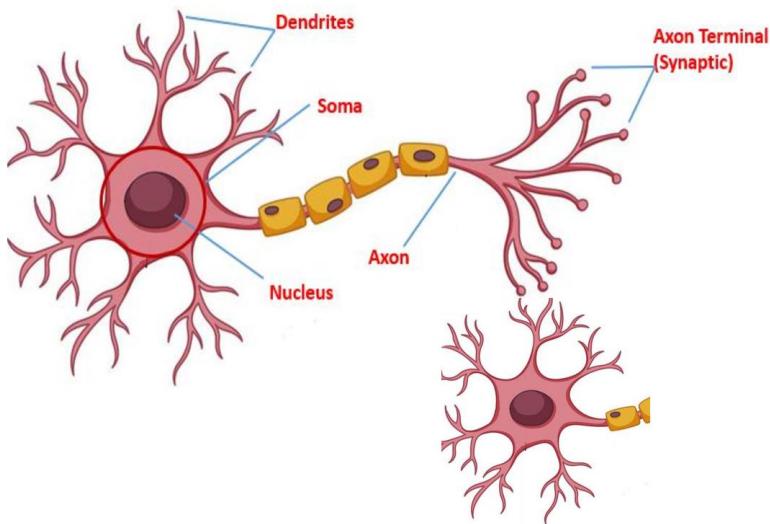


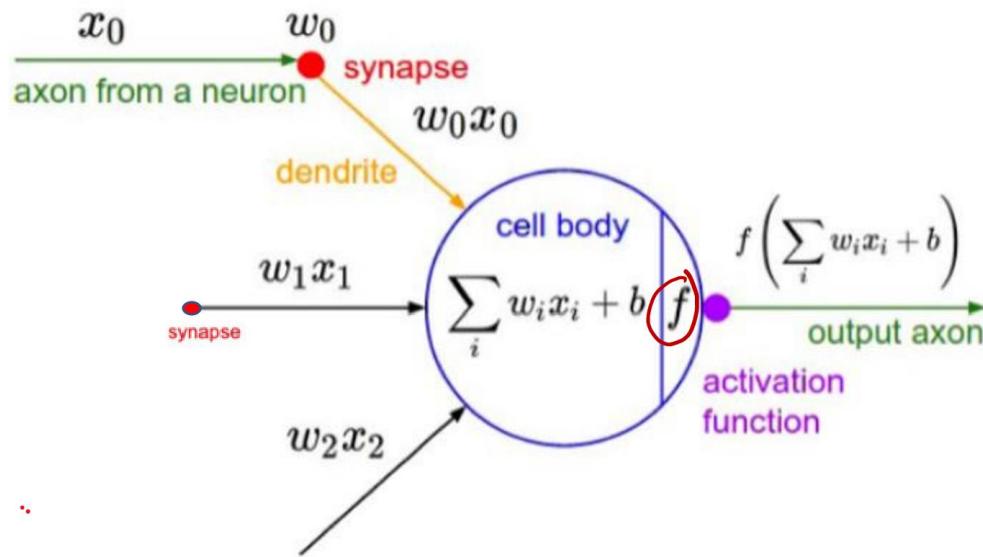
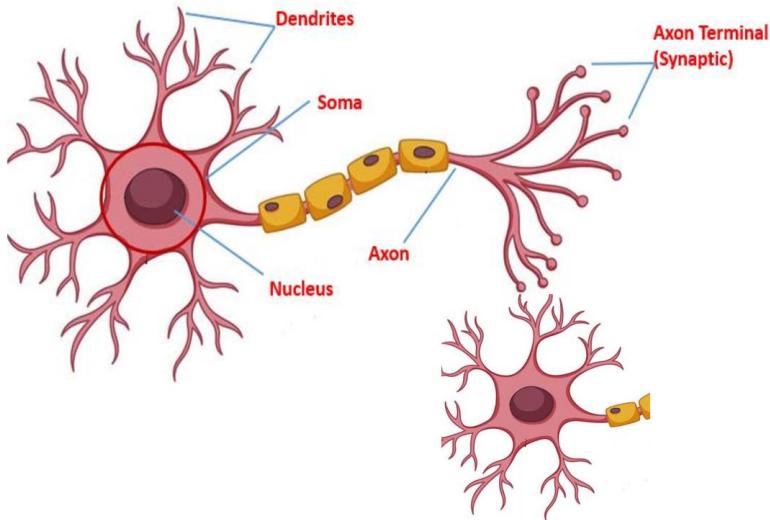


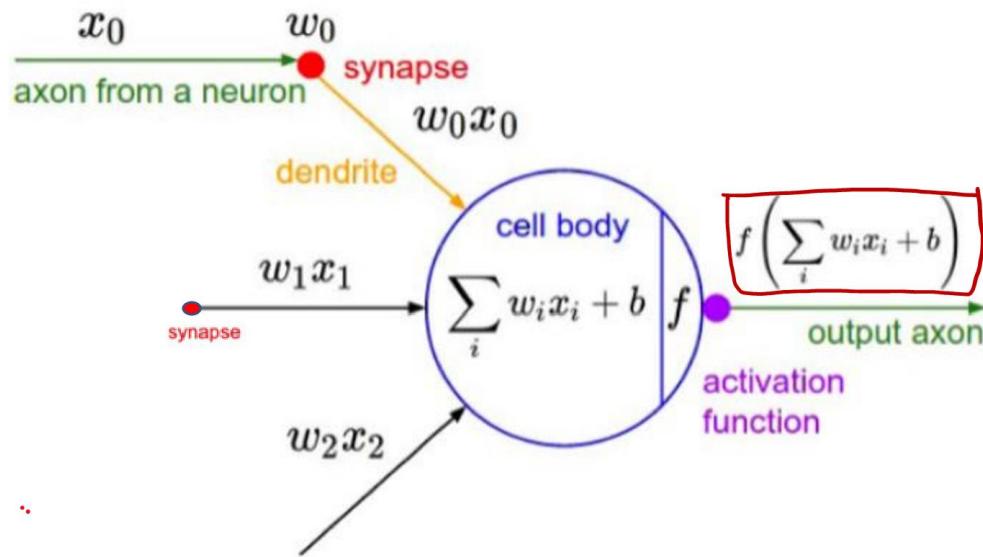
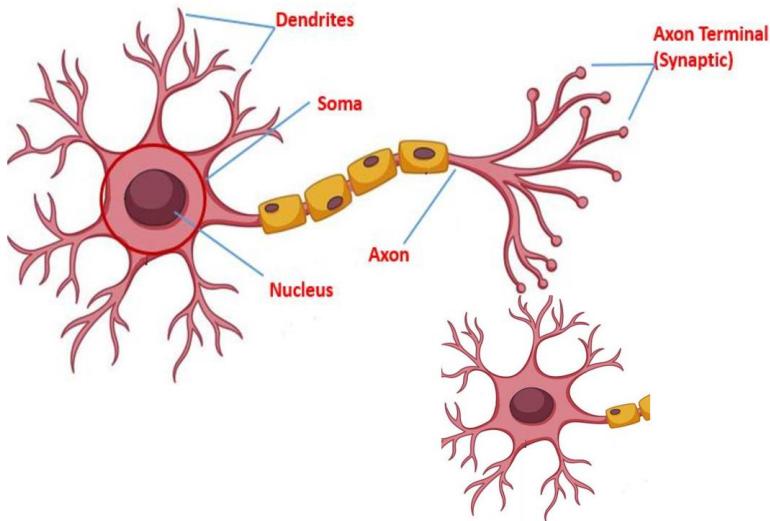






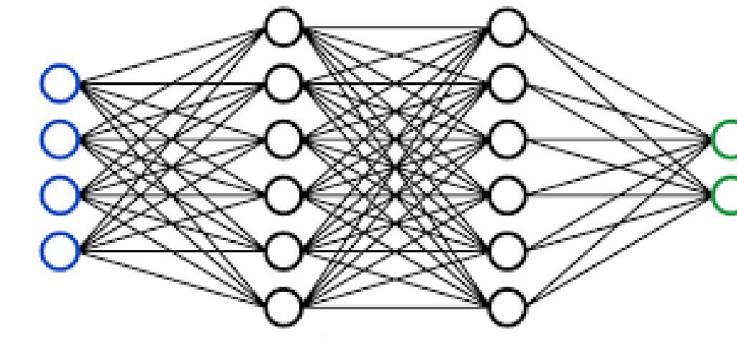
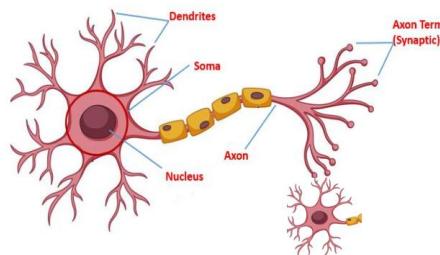




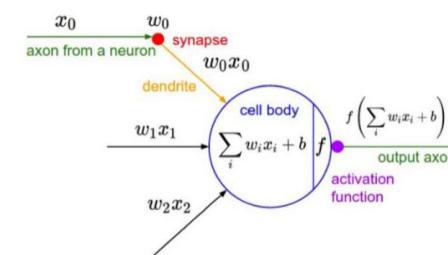




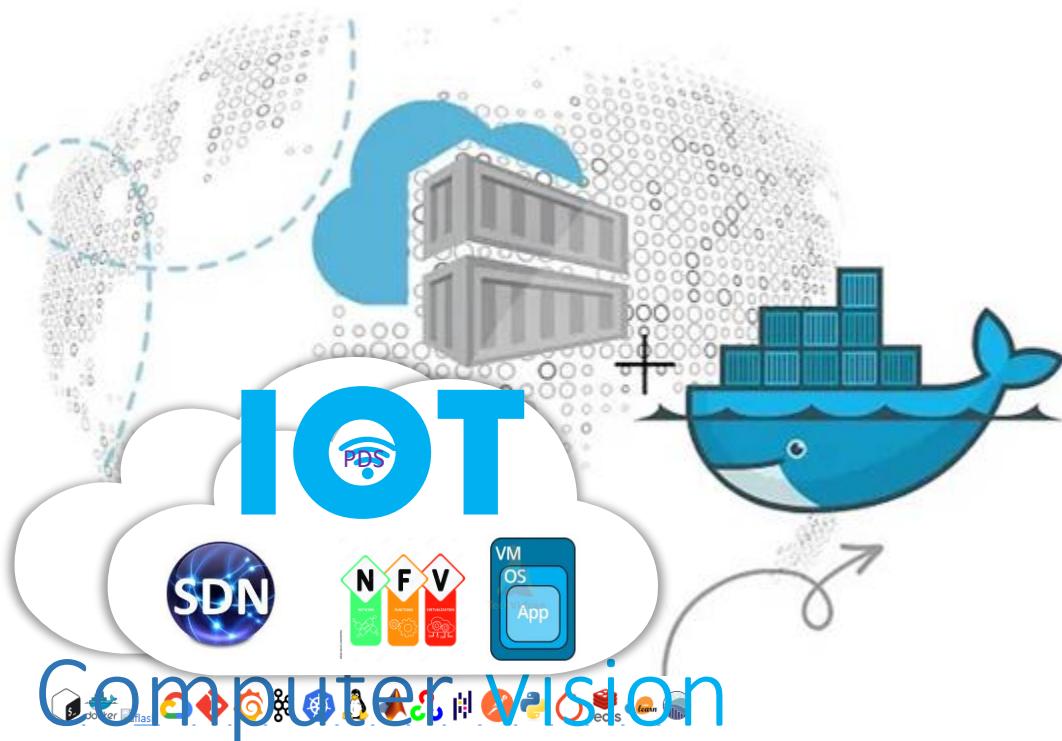
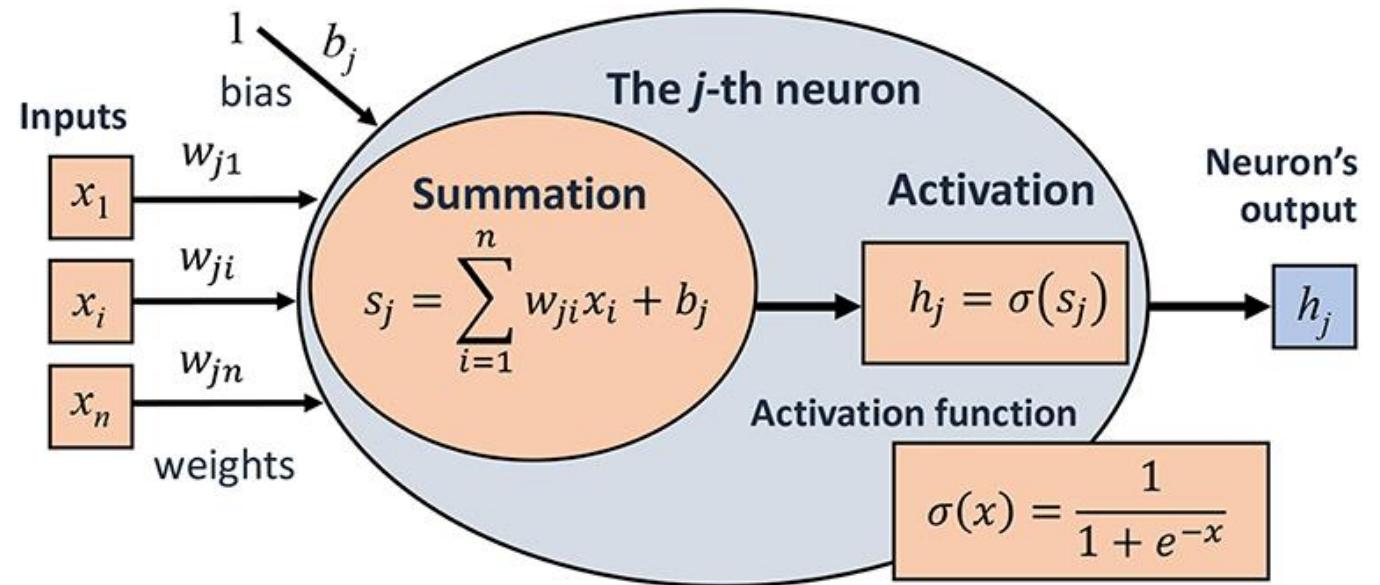
↑
Network of neurons



↑
Network of neurons



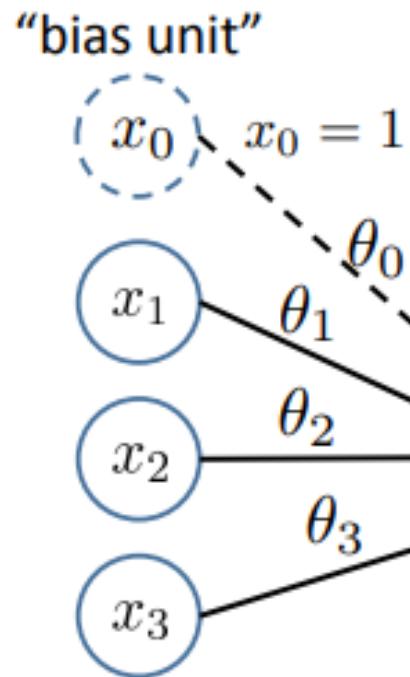
Neural Networks



Neural Networks

- Origins: Algorithms that try to mimic the brain
- 40s and 50s: Hebbian learning and Perceptron
- Perceptrons book in 1969 and the XOR problem
- Very widely used in 80s and early 90s; popularity diminished in late 90s.
- Recent resurgence: technique for many applications
- Artificial neural networks are not nearly as complex or intricate as the actual brain structure

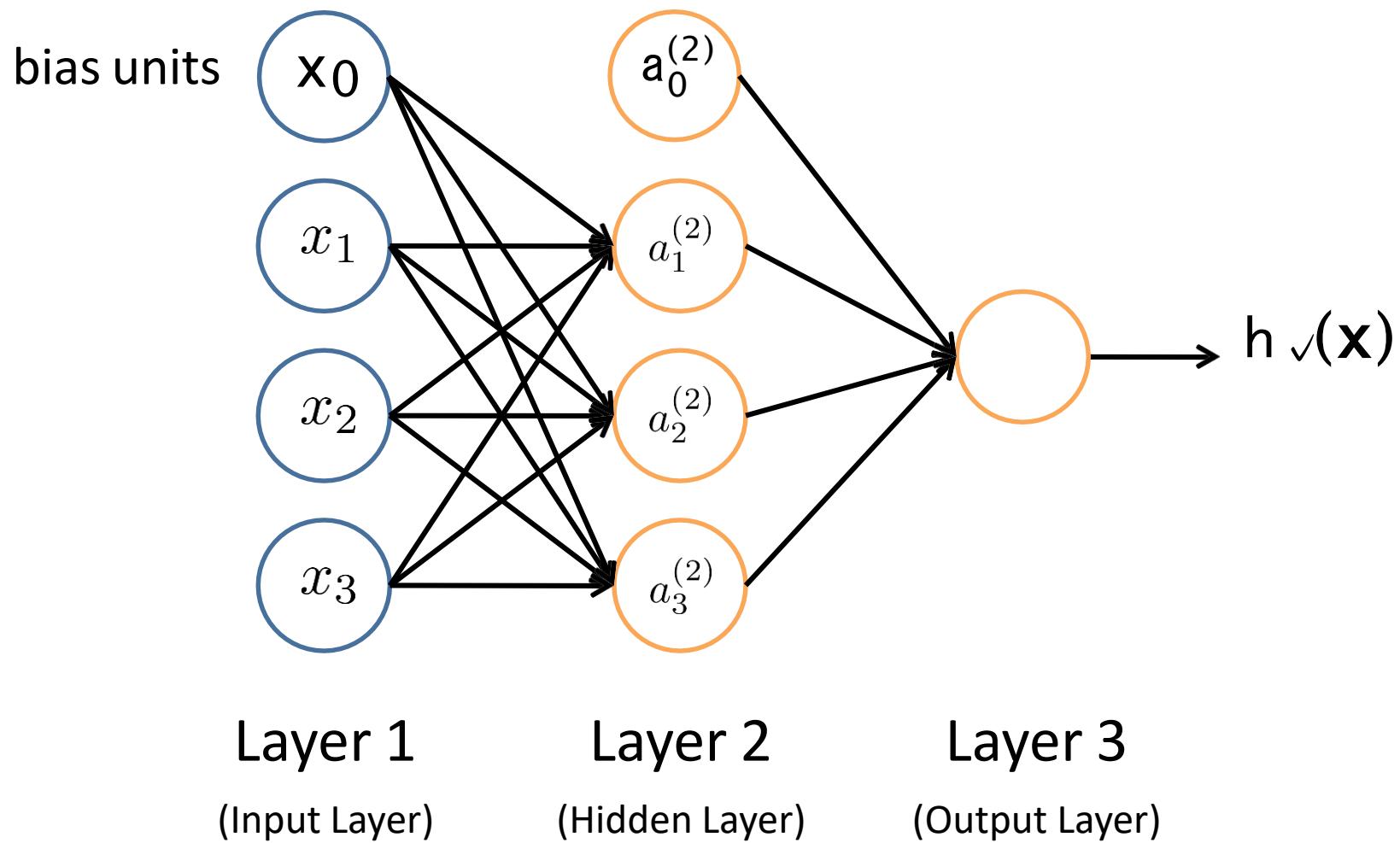
Single Node



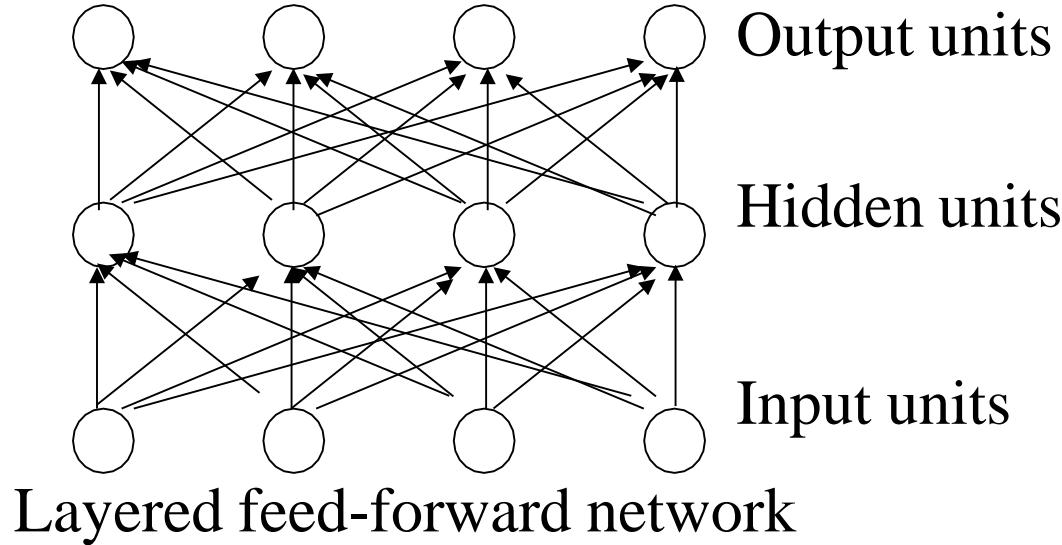
$$\mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad \boldsymbol{\theta} = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix}$$
$$h_{\boldsymbol{\theta}}(\mathbf{x}) = g(\boldsymbol{\theta}^T \mathbf{x}) = \frac{1}{1 + e^{-\boldsymbol{\theta}^T \mathbf{x}}}$$

Sigmoid (logistic) activation function: $g(z) = \frac{1}{1 + e^{-z}}$

Neural Network



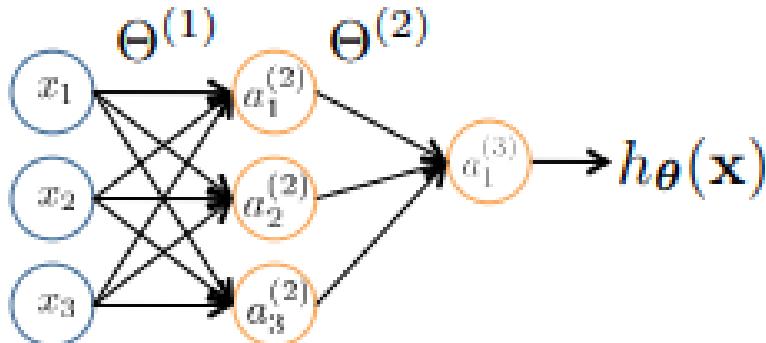
Neural networks Terminology



- Neural networks are made up of **nodes** or **units**, connected by **links**
- Each link has an associated **weight** and **activation level**
- Each node has an **input function** (typically summing over weighted inputs), an **activation function**, and an **output**

Feed-Forward Process

- Input layer units are set by external data, which causes their output links to be **activated** at the specified level
- Working forward through the network, the **input function** of each unit is applied to compute the input value
 - Usually this is just the weighted sum of the activation on the links feeding into this node
- The **activation function** transforms this input function into a final value
 - Typically this is a **nonlinear** function, often a **sigmoid** function corresponding to the “threshold” of that node



$a_i^{(j)}$ = “activation” of unit i in layer j

$\Theta^{(j)}$ = weight matrix stores parameters
from layer j to layer $j + 1$

$$a_1^{(2)} = g(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3)$$

$$a_2^{(2)} = g(\Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3)$$

$$a_3^{(2)} = g(\Theta_{30}^{(1)}x_0 + \Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2 + \Theta_{33}^{(1)}x_3)$$

$$h_\Theta(x) = a_1^{(3)} = g(\Theta_{10}^{(2)}a_0^{(2)} + \Theta_{11}^{(2)}a_1^{(2)} + \Theta_{12}^{(2)}a_2^{(2)} + \Theta_{13}^{(2)}a_3^{(2)})$$

If network has s_j units in layer j and s_{j+1} units in layer $j+1$,
then $\Theta^{(j)}$ has dimension $s_{j+1} \times (s_j + 1)$.

$$\Theta^{(1)} \in \mathbb{R}^{3 \times 4} \quad \Theta^{(2)} \in \mathbb{R}^{1 \times 4}$$

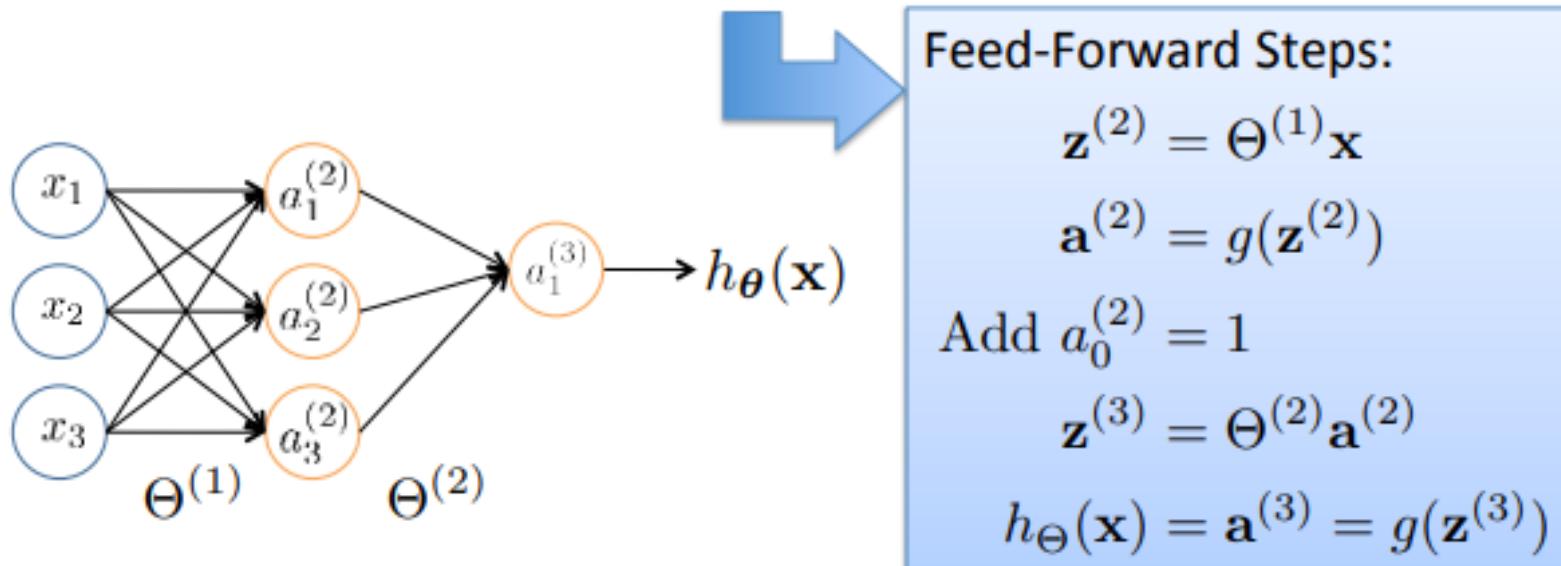
Vectorization

$$a_1^{(2)} = g \left(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3 \right) = g \left(z_1^{(2)} \right)$$

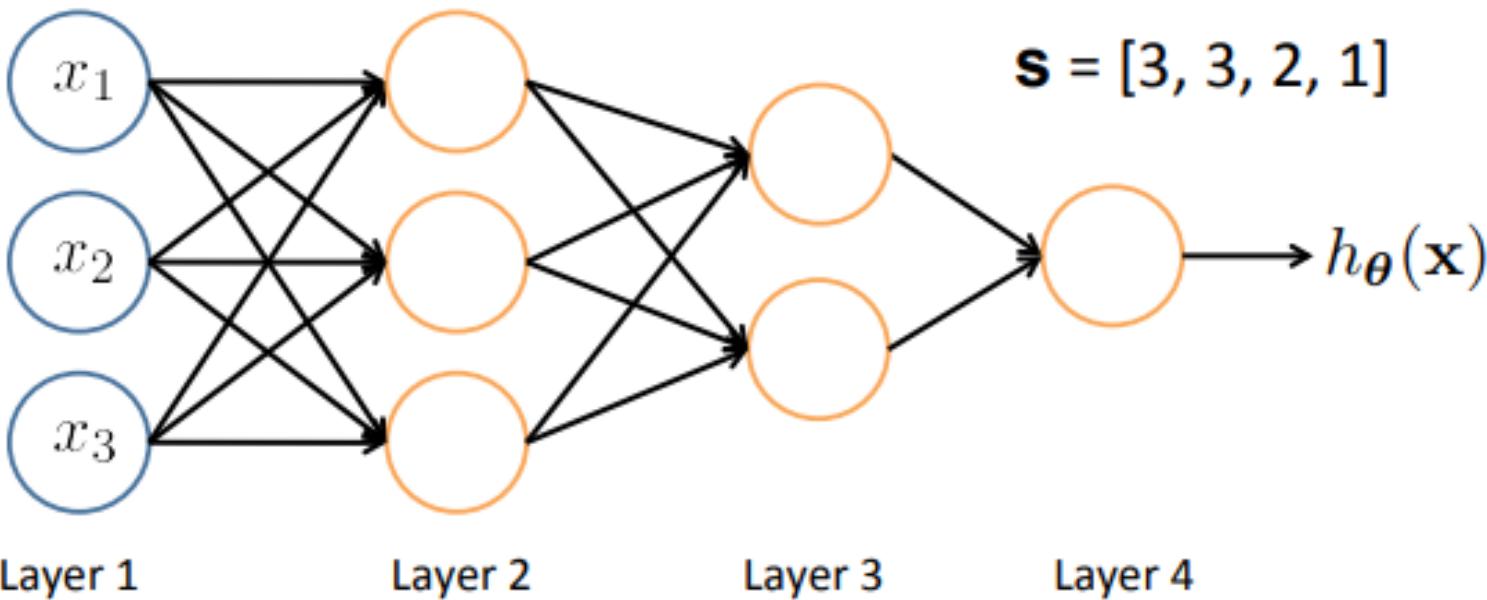
$$a_2^{(2)} = g \left(\Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3 \right) = g \left(z_2^{(2)} \right)$$

$$a_3^{(2)} = g \left(\Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3 \right) = g \left(z_3^{(2)} \right)$$

$$h_{\Theta}(\mathbf{x}) = g \left(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)} \right) = g \left(z_1^{(3)} \right)$$



Other Network Architectures



L denotes the number of layers

$\mathbf{s} \in \mathbb{N}^{+L}$ contains the numbers of nodes at each layer

- Not counting bias units
- Typically, $s_0 = d$ (# input features) and $s_{L-1} = K$ (# classes)

Multiple Output Units: One vs Rest



Pedestrian



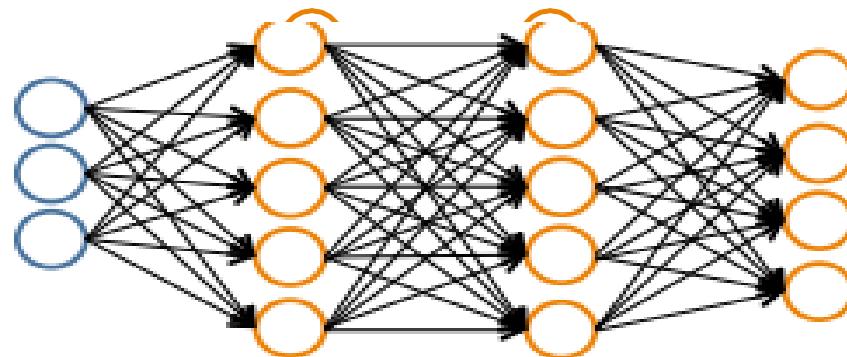
Car



Motorcycle



Truck



$$h_{\Theta}(\mathbf{x}) \in \mathbb{R}^K$$

We want:

$$h_{\Theta}(\mathbf{x}) \approx \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

when pedestrian

$$h_{\Theta}(\mathbf{x}) \approx \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

when car

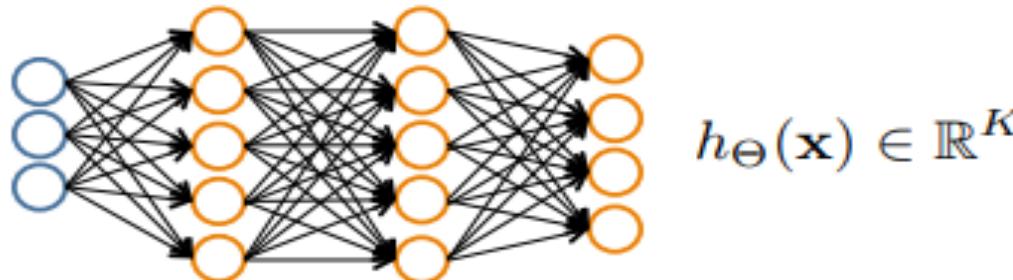
$$h_{\Theta}(\mathbf{x}) \approx \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

when motorcycle

$$h_{\Theta}(\mathbf{x}) \approx \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

when truck

Multiple Output Units: One vs Rest



We want:

$$h_{\Theta}(\mathbf{x}) \approx \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad \text{when pedestrian}$$

$$h_{\Theta}(\mathbf{x}) \approx \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad \text{when car}$$

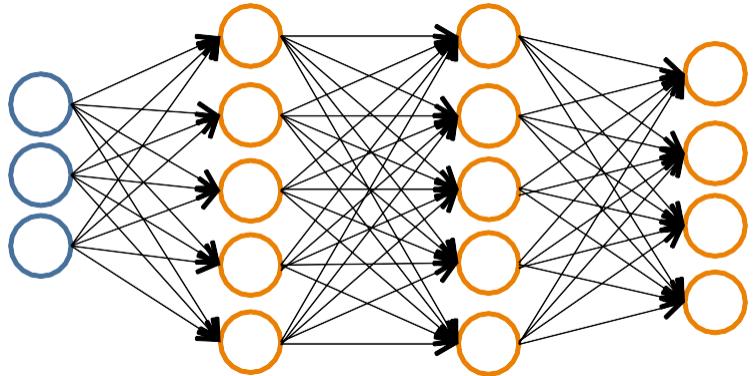
$$h_{\Theta}(\mathbf{x}) \approx \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \quad \text{when motorcycle}$$

$$h_{\Theta}(\mathbf{x}) \approx \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \quad \text{when truck}$$

- Given $\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_n, y_n)\}$
- Must convert labels to 1-of- K representation

– e.g., $y_i = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$ when motorcycle, $y_i = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$ when car, etc.

Neural Network Classification



Given:

$$\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_n, y_n)\}$$

$\mathbf{s} \in \mathbb{N}^{+L}$ contains # nodes at each layer
– $s_0 = d$ (# features)

Binary classification

$$y = 0 \text{ or } 1$$

1 output unit ($s_{L-1} = 1$)

Multi class classification (K classes)

$$\mathbf{y} \in \mathbb{R}^K \quad \text{e.g. } \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

pedestrian car motorcycle truck

K output units ($s_{L-1} = K$)

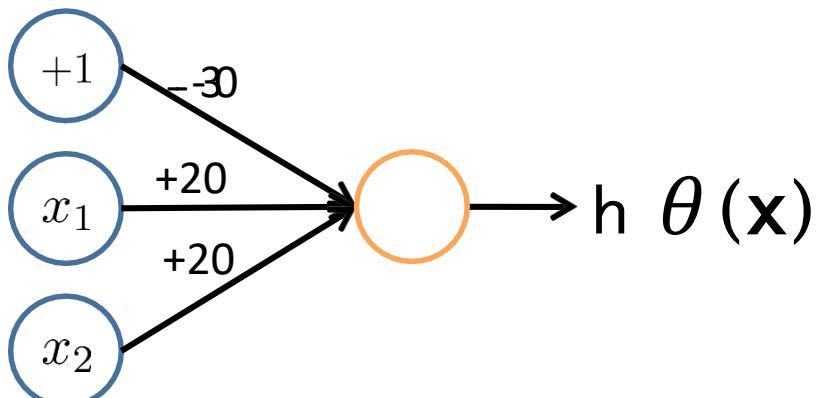
Understanding Representations

Representing Boolean Functions

Simple example: AND

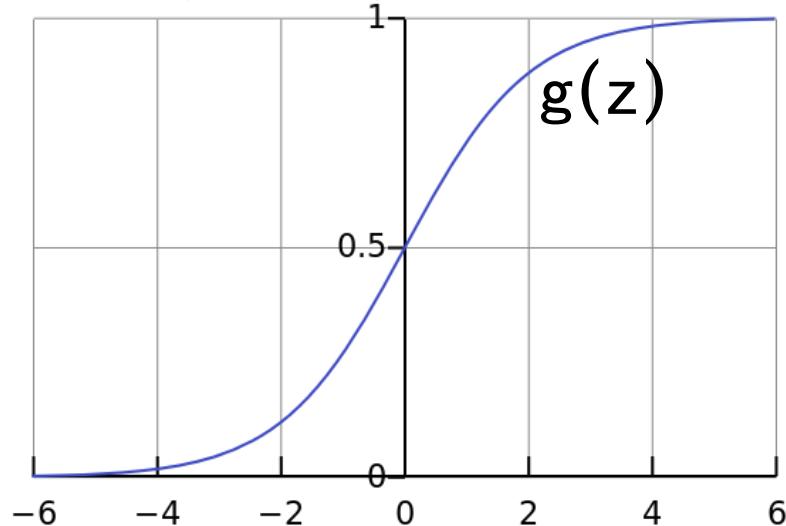
$$x_1, x_2 \in \{0, 1\}$$

$$y = x_1 \text{ AND } x_2$$



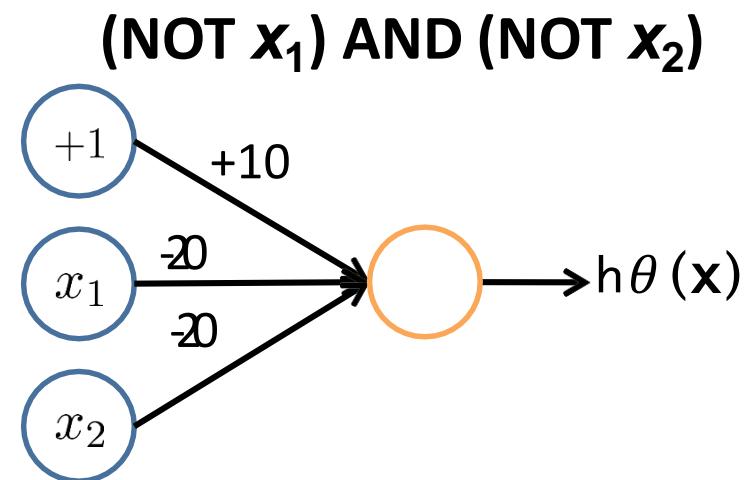
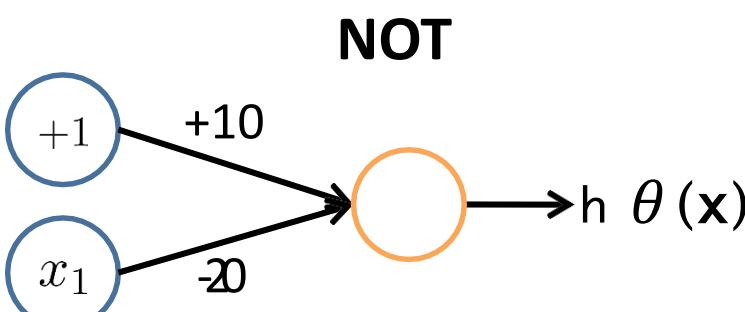
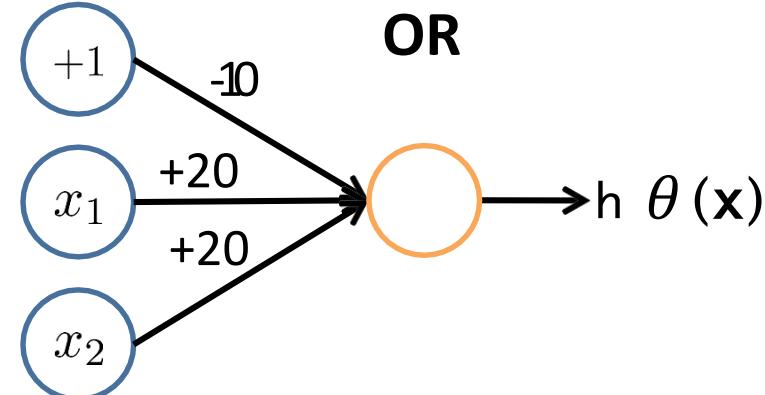
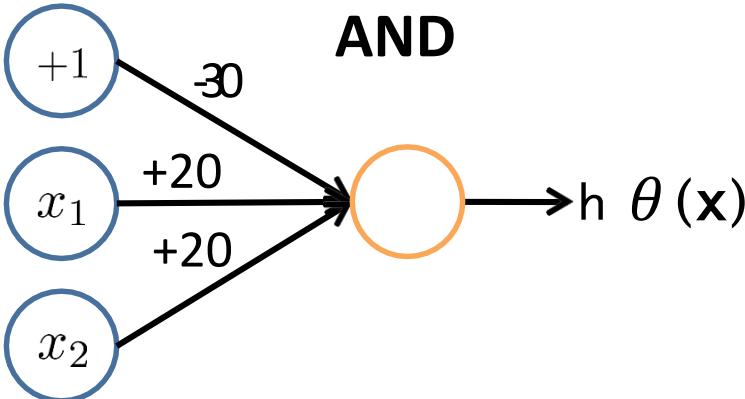
$$h_\theta(\mathbf{x}) = g(-30 + 20x_1 + 20x_2)$$

Logistic / Sigmoid Function

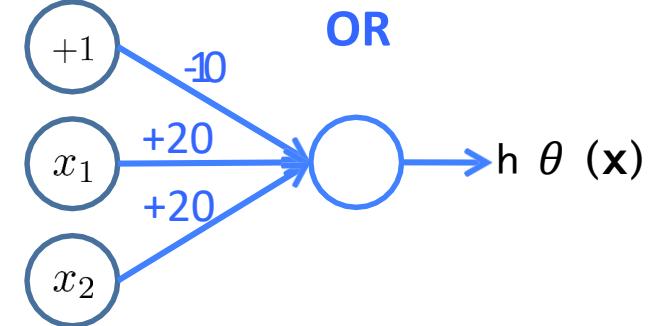
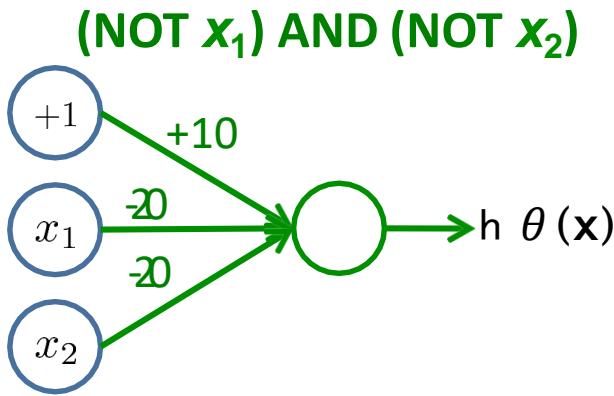
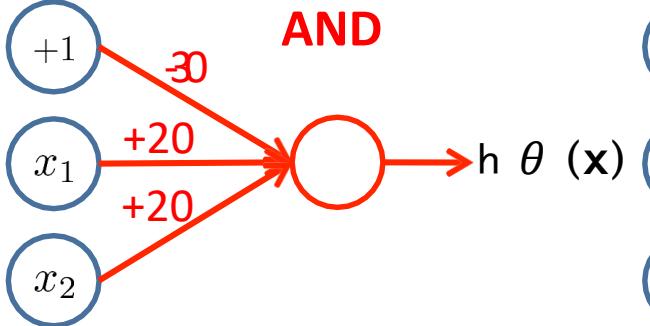


x_1	x_2	$h_\theta(\mathbf{x})$
0	0	$g(-30) \approx 0$
0	1	$g(-10) \approx 0$
1	0	$g(-10) \approx 0$
1	1	$g(10) \approx 1$

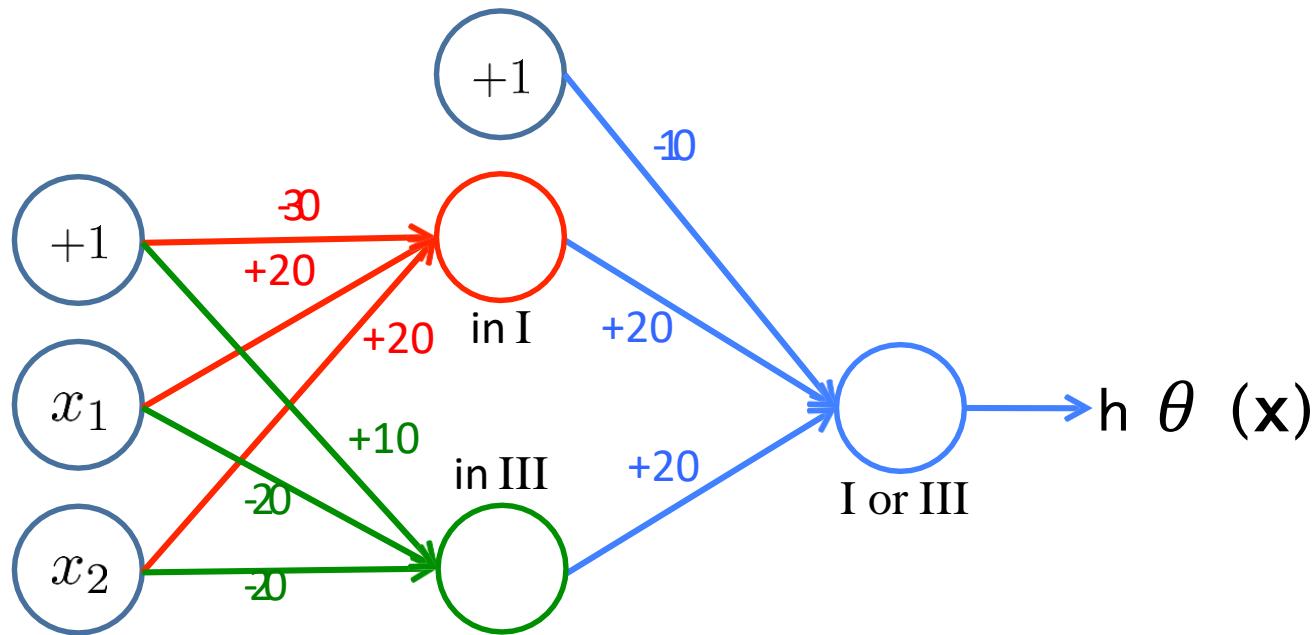
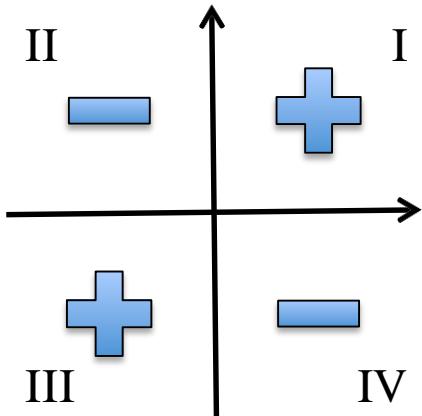
Representing Boolean Functions



Combining Representations to Create Non Linier Function



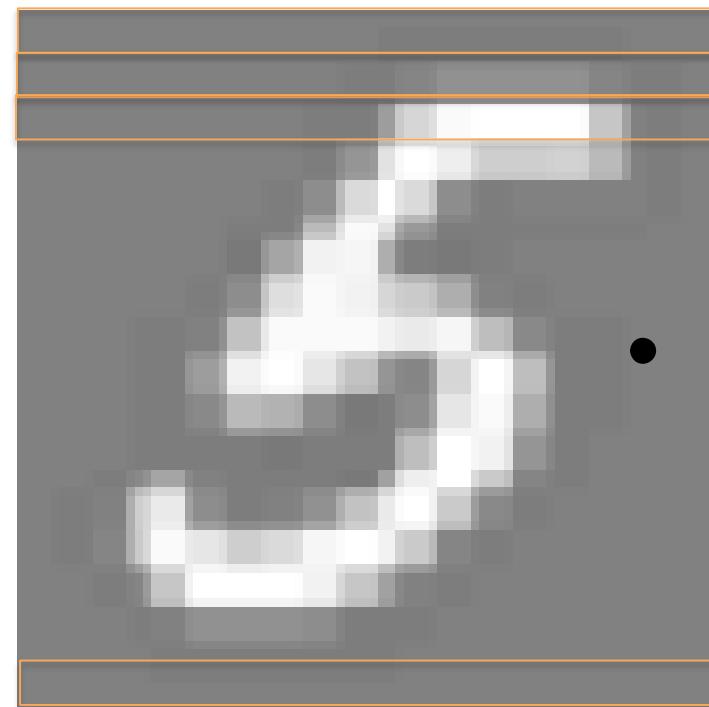
NOT XOR



Layering Representations



20×20 pixel images
 $d = 400$ 10 classes

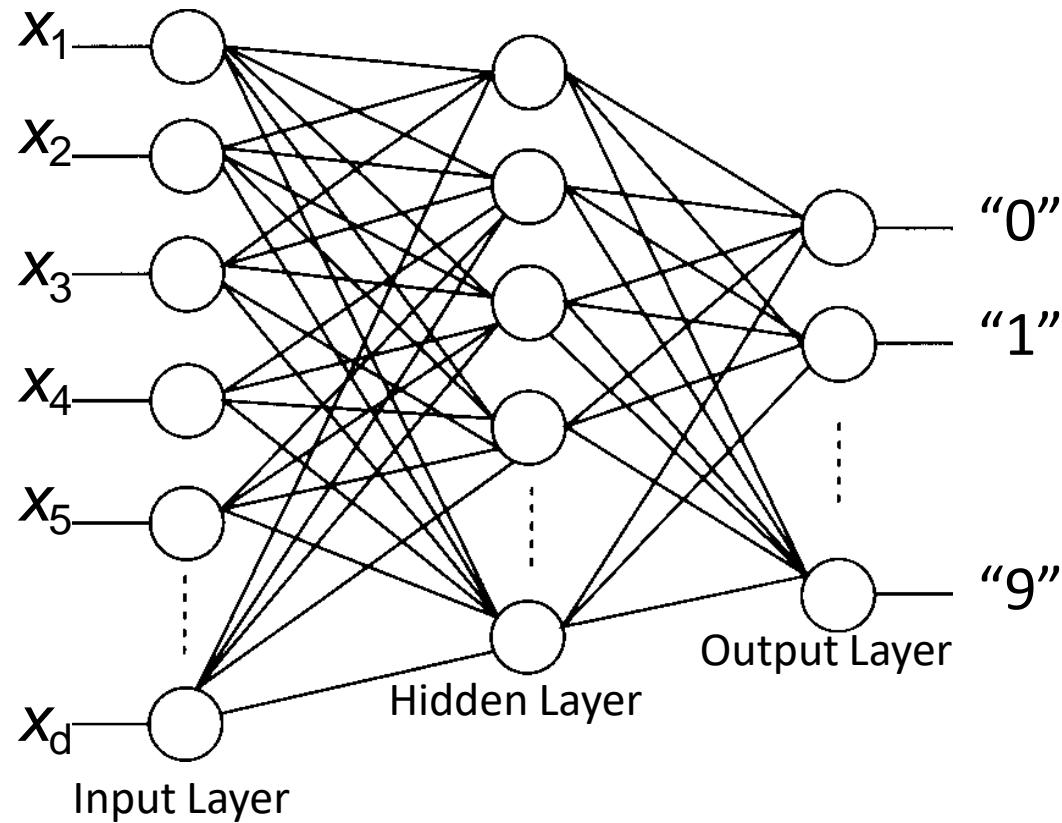


$x_1 \dots x_{20}$
 $x_{21} \dots x_{40}$
 $x_{41} \dots x_{60}$
⋮
 $x_{381} \dots x_{400}$

Each image is “unrolled” into a vector \mathbf{x} of pixel intensities

Layering Representations

7	9	6	5	8	7	4	4	1	8
0	7	3	3	2	4	8	4	5	1
6	6	3	2	9	2	3	3	2	6
1	3	7	1	5	6	5	2	4	4
7	0	9	8	7	5	8	9	5	4
4	6	6	5	0	2	1	3	6	9
8	5	1	8	9	3	8	7	3	6
1	0	2	8	2	3	0	5	1	5
6	7	8	2	5	3	9	7	0	0
7	9	3	9	8	5	7	2	9	8



Neural Network Learning

Perceptron Learning Rule

$$\theta \leftarrow \theta + \alpha(y - h(\mathbf{x}))\mathbf{x}$$

Intuitive rule:

- If output is correct, don't change the weights
- If output is low ($h(\mathbf{x}) = 0, y = 1$), increment weights for all the inputs which are 1
- If output is high ($h(\mathbf{x}) = 1, y = 0$), decrement weights for all inputs which are 1

Perceptron Convergence Theorem:

- If there is a set of weights that is consistent with the training data (i.e., the data is linearly separable), the perceptron learning algorithm will converge [Minsky & Papert, 1969]

Batch Perceptron

Given training data $(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^n$

Let $\theta \leftarrow [0, 0, \dots, 0]$

Repeat:

```
    Let  $\mathbf{A} \leftarrow [0, 0, \dots, 0]$ 
    for  $i = 1 \dots n$ , do
        if  $y^{(i)} \mathbf{x}^{(i)} \theta \leq 0$           // prediction for  $i^{th}$  instance is incorrect
             $\mathbf{A} \leftarrow \mathbf{A} + y^{(i)} \mathbf{x}^{(i)}$ 
     $\mathbf{A} \leftarrow \mathbf{A} / n$                   // compute average update
     $\theta \leftarrow \theta + \alpha \Delta$ 
Until  $\|\Delta\|_2 < \epsilon$ 
```

- Simplest case: $\alpha = 1$ and don't normalize, yields the fixed increment perceptron
- Each increment of outer loop is called an **epoch**

Learning in NN: Backpropagation

- Similar to the perceptron learning algorithm, we cycle through our examples
 - If the output of the network is correct, no changes are made
 - If there is an error, weights are adjusted to reduce the error
- We are just performing (stochastic) gradient descent!

Cost Function

Logistic Regression:

$$J(\theta) = -\frac{1}{n} \sum_{i=1}^n [y_i \log h_{\theta}(\mathbf{x}_i) + (1 - y_i) \log (1 - h_{\theta}(\mathbf{x}_i))] + \frac{\lambda}{2n} \sum_{j=1}^d \theta_j^2$$

Neural Network:

$$h_{\Theta} \in \mathbb{R}^K \quad (h_{\Theta}(\mathbf{x}))_i = i^{th} \text{output}$$

$$\begin{aligned} J(\Theta) = & -\frac{1}{n} \left[\sum_{i=1}^n \sum_{k=1}^K y_{ik} \log (h_{\Theta}(\mathbf{x}_i))_k + (1 - y_{ik}) \log (1 - (h_{\Theta}(\mathbf{x}_i))_k) \right] \\ & + \frac{\lambda}{2n} \sum_{l=1}^{L-1} \sum_{i=1}^{s_{l-1}} \sum_{j=1}^{s_l} \left(\Theta_{ji}^{(l)} \right)^2 \end{aligned}$$

k^{th} class: true, predicted
not k^{th} class: true, predicted

Optimizing the Neural Network

$$J(\Theta) = -\frac{1}{n} \left[\sum_{i=1}^n \sum_{k=1}^K y_{ik} \log(h_\Theta(\mathbf{x}_i))_k + (1 - y_{ik}) \log(1 - (h_\Theta(\mathbf{x}_i))_k) \right] + \frac{\lambda}{2n} \sum_{l=1}^{L-1} \sum_{i=1}^{s_{l-1}} \sum_{j=1}^{s_l} (\Theta_{ji}^{(l)})^2$$

Solve via: $\min_{\Theta} J(\Theta)$

Unlike before, $J(\Theta)$ is not convex, so GD on a neural net yields a local optimum

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = a_j^{(l)} \delta_i^{(l+1)} \quad (\text{ignoring } \lambda; \text{ if } \lambda = 0)$$

Forward
Propagation

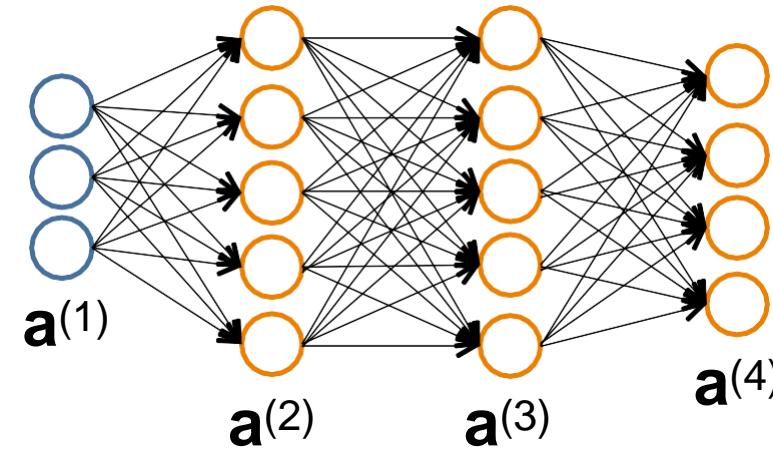
Backpropagation

Forward Propagation

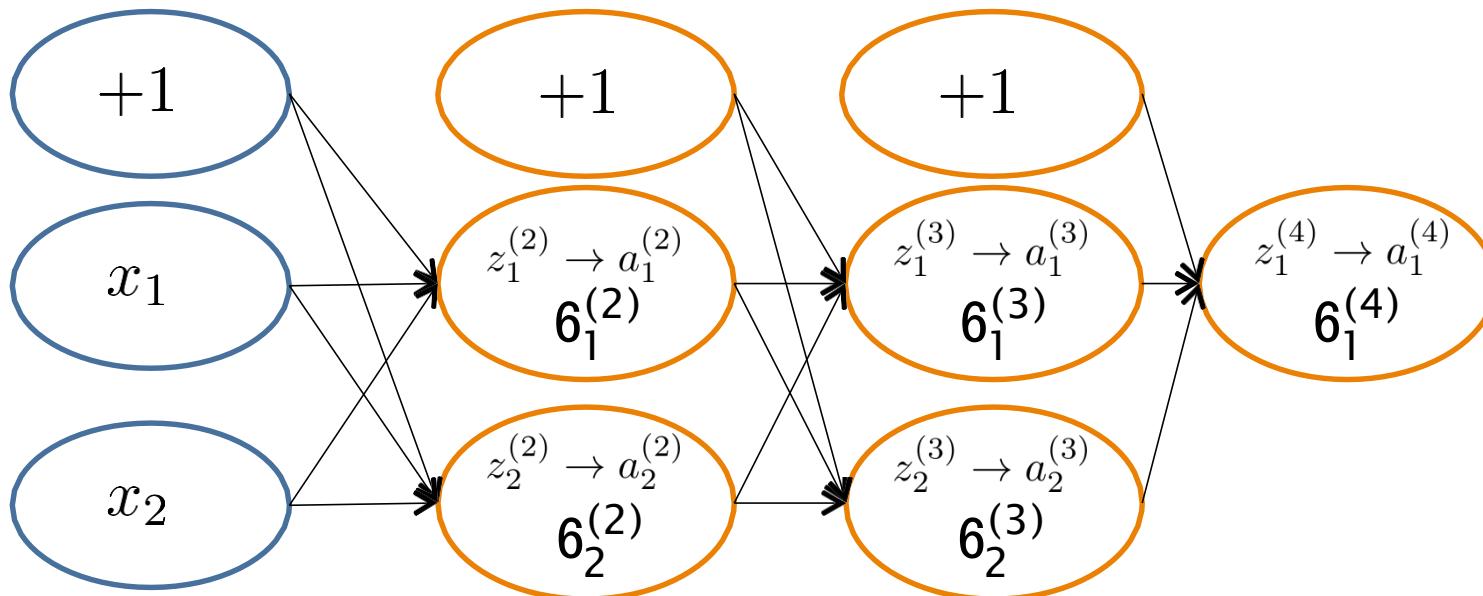
- Given one labeled training instance (\mathbf{x}, y) :

Forward Propagation

- $\mathbf{a}^{(1)} = \mathbf{x}$
- $\mathbf{z}^{(2)} = \Theta^{(1)}\mathbf{a}^{(1)}$
- $\mathbf{a}^{(2)} = g(\mathbf{z}^{(2)})$ [add $a_0^{(2)}$]
- $\mathbf{z}^{(3)} = \Theta^{(2)}\mathbf{a}^{(2)}$
- $\mathbf{a}^{(3)} = g(\mathbf{z}^{(3)})$ [add $a_0^{(3)}$]
- $\mathbf{z}^{(4)} = \Theta^{(3)}\mathbf{a}^{(3)}$
- $\mathbf{a}^{(4)} = h_{\Theta}(\mathbf{x}) = g(\mathbf{z}^{(4)})$



Backpropagation Intuition

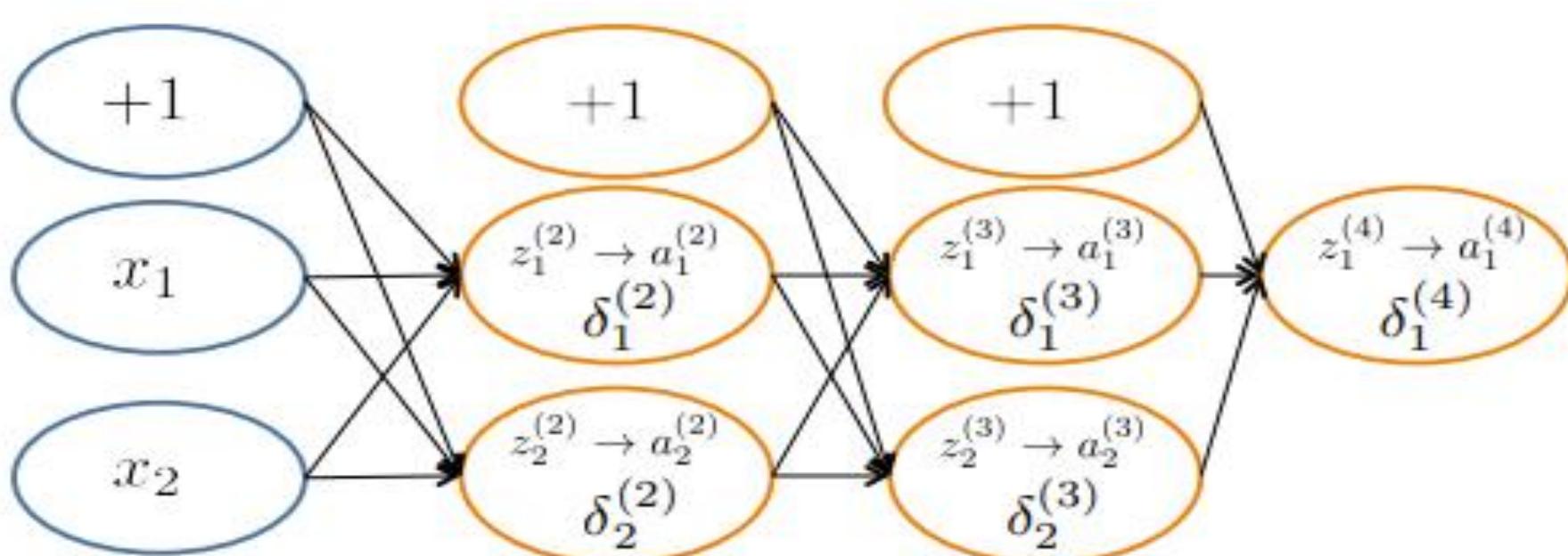


$\delta_j^{(l)}$ = “error” of node j in layer l

Formally, $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(\mathbf{x}_i)$

where $\text{cost}(\mathbf{x}_i) = y_i \log h_{\rightarrow}(\mathbf{x}_i) + (1 - y_i) \log(1 - h_{\rightarrow}(\mathbf{x}_i))$

Backpropagation Intuition

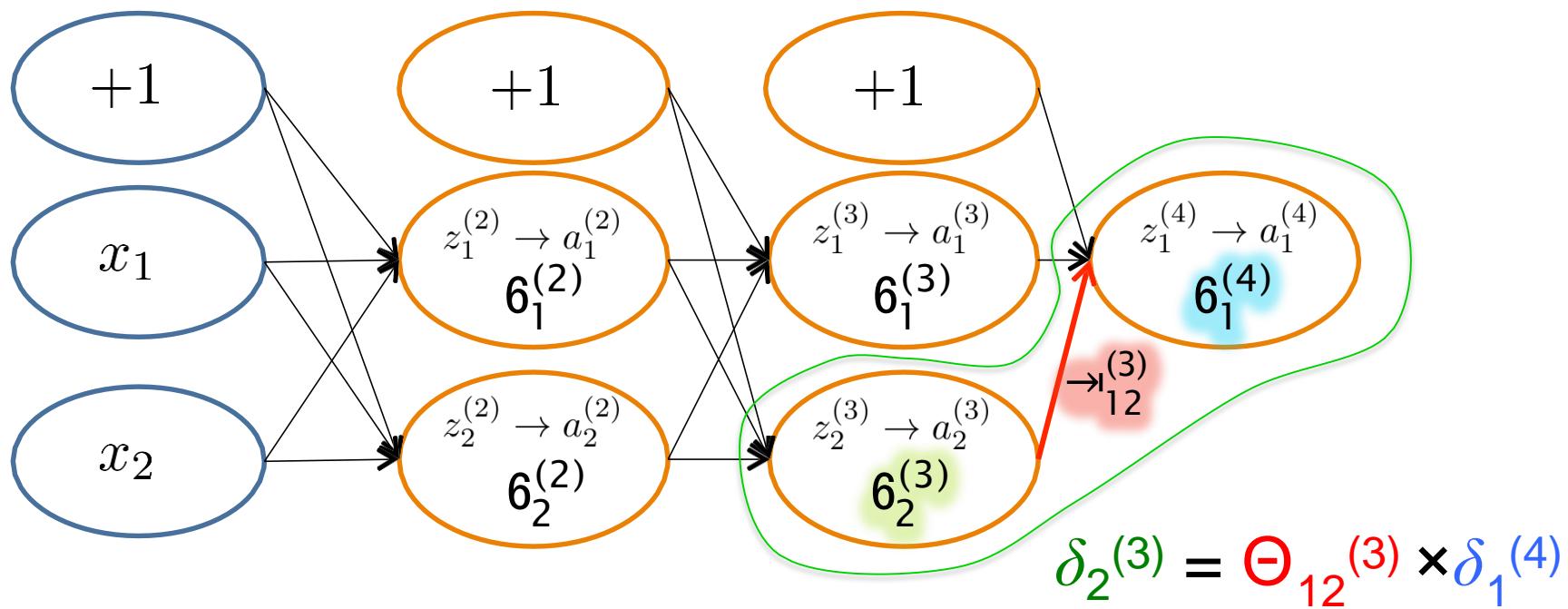


$\delta_j^{(l)}$ = “error” of node j in layer l

Formally, $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(\mathbf{x}_i)$

where $\text{cost}(\mathbf{x}_i) = y_i \log h_{\Theta}(\mathbf{x}_i) + (1 - y_i) \log(1 - h_{\Theta}(\mathbf{x}_i))$

Backpropagation Intuition

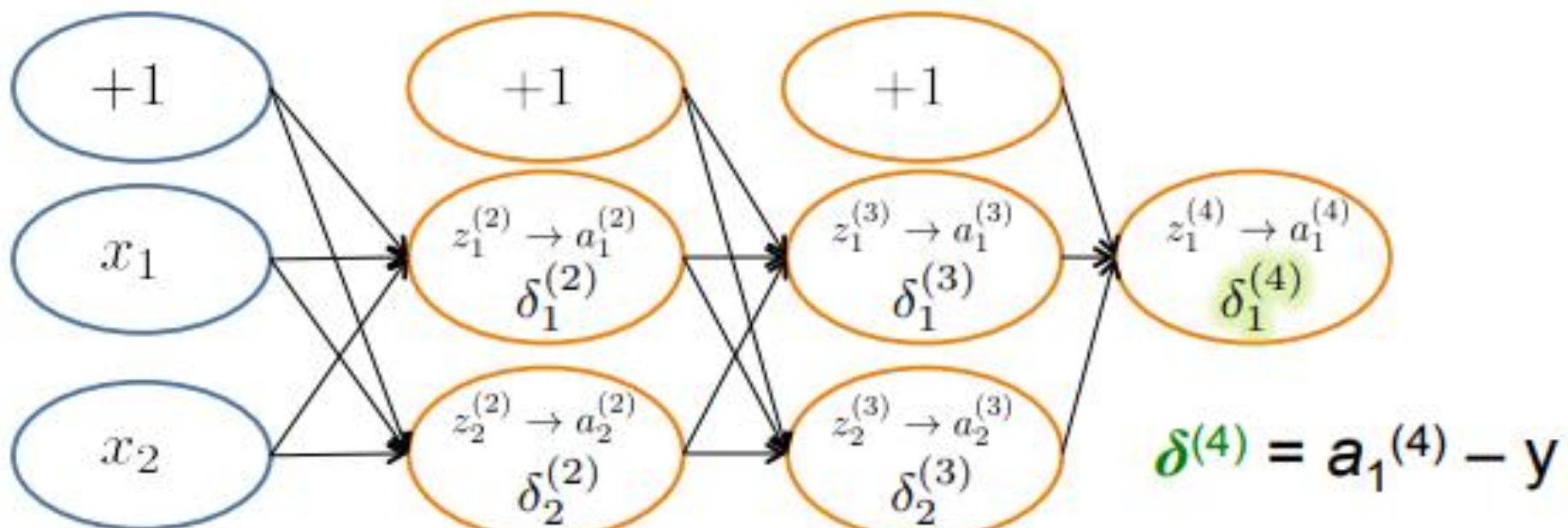


$\delta_j^{(l)}$ = “error” of node j in layer l

Formally, $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(\mathbf{x}_i)$

where $\text{cost}(\mathbf{x}_i) = y_i \log h_{\rightarrow}(\mathbf{x}_i) + (1 - y_i) \log(1 - h_{\rightarrow}(\mathbf{x}_i))$

Backpropagation Intuition

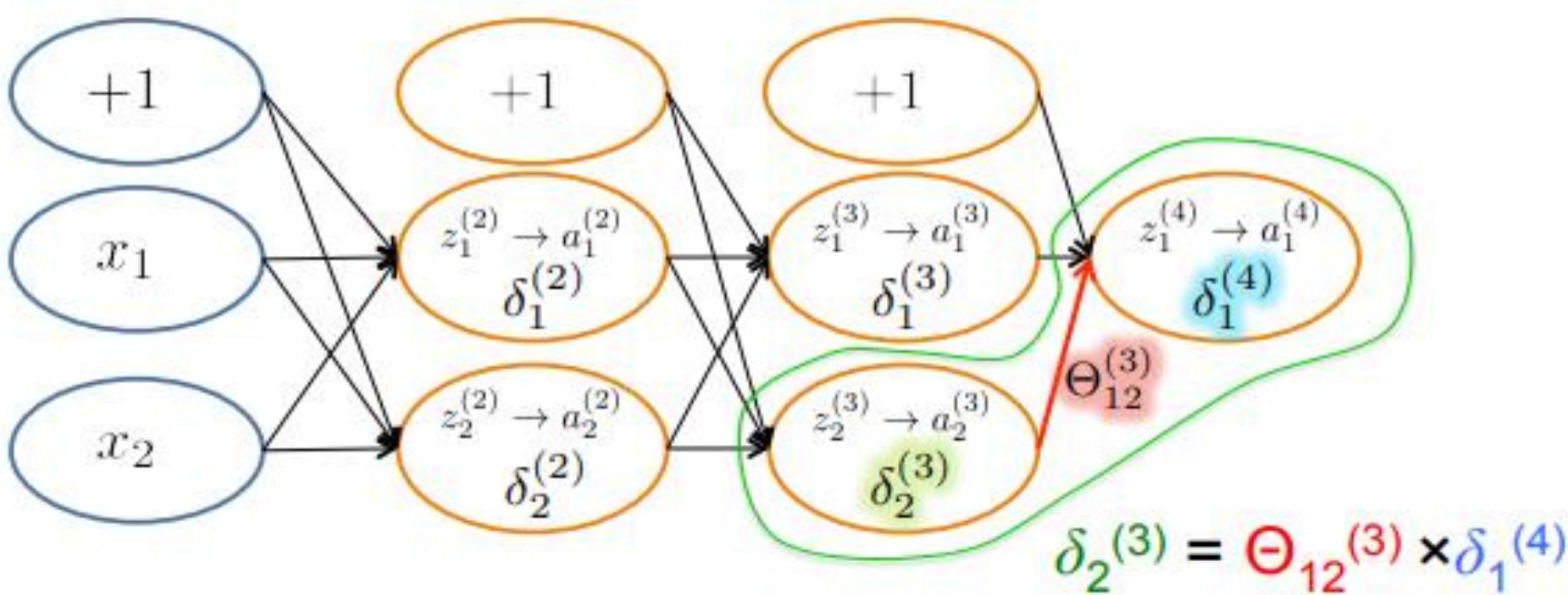


$\delta_j^{(l)}$ = “error” of node j in layer l

Formally, $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(\mathbf{x}_i)$

where $\text{cost}(\mathbf{x}_i) = y_i \log h_{\Theta}(\mathbf{x}_i) + (1 - y_i) \log(1 - h_{\Theta}(\mathbf{x}_i))$

Backpropagation Intuition



$\delta_j^{(l)}$ = “error” of node j in layer l

Formally, $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(\mathbf{x}_i)$

where $\text{cost}(\mathbf{x}_i) = y_i \log h_\Theta(\mathbf{x}_i) + (1 - y_i) \log(1 - h_\Theta(\mathbf{x}_i))$

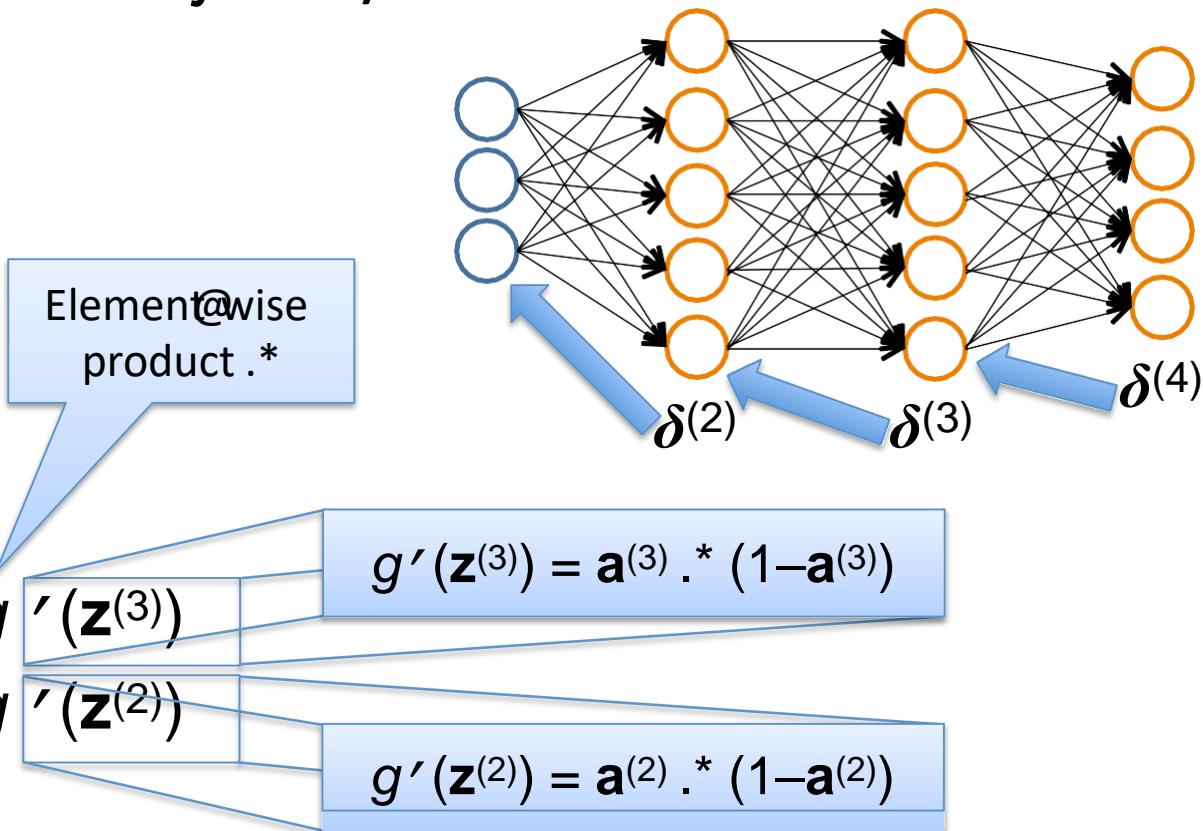
Backpropagation: Gradient Computation

Let $\delta_{j(l)}$ = “error” of node j in layer l

(#layers $L = 4$)

Backpropagation

- $\delta^{(4)} = \mathbf{a}^{(4)} - \mathbf{y}$
- $\delta^{(3)} = (\Theta^{(3)})^T \delta^{(4)} . * g'(\mathbf{z}^{(3)})$
- $\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} . * g'(\mathbf{z}^{(2)})$
- (No $\delta^{(1)}$)



Backpropagation

Set $\Delta_{ij}^{(l)} = 0 \quad \forall i, j$

(Used to accumulate gradient)

For each training instance (\mathbf{x}_i, y_i) :

Set $\mathbf{a}^{(1)} = \mathbf{x}_i$

Compute $\{\mathbf{a}^{(2)}, \dots, \mathbf{a}^{(L)}\}$ via forward propagation

Compute $\mathbf{6}^{(L)} = \mathbf{a}^{(L)} - y_i$

Compute errors $\{\mathbf{6}^{(L-1)}, \dots, \mathbf{6}^{(2)}\}$

Compute gradients $\Delta_{ij}^{(l)} = \Delta_{ij}^{(l)} + \mathbf{a}_j^{(l)} \delta_i^{(l+1)}$

Compute avg regularized gradient $D_{ij}^{(l)} = \begin{cases} \frac{1}{n} \Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)} & \text{if } j \neq 0 \\ \frac{1}{n} \Delta_{ij}^{(l)} & \text{otherwise} \end{cases}$

$D^{(l)}$ is the matrix of partial derivatives of $J(\Theta)$

Training a Neural Network via Gradient Descent with Backprop

Given: training set $\{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)\}$

Initialize all $\Theta^{(l)}$ randomly (NOT to 0!)

Loop // each iteration is called an epoch

Set $\Delta_{ij}^{(l)} = 0 \quad \forall l, i, j$ (Used to accumulate gradient)

For each training instance (\mathbf{x}_i, y_i) :

Set $\mathbf{a}^{(1)} = \mathbf{x}_i$

Compute $\{\mathbf{a}^{(2)}, \dots, \mathbf{a}^{(L)}\}$ via forward propagation

Compute $\delta^{(L)} = \mathbf{a}^{(L)} - y_i$

Compute errors $\{\delta^{(L-1)}, \dots, \delta^{(2)}\}$

Compute gradients $\Delta_{ij}^{(l)} = \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$

Compute avg regularized gradient $D_{ij}^{(l)} = \begin{cases} \frac{1}{n} \Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)} & \text{if } j \neq 0 \\ \frac{1}{n} \Delta_{ij}^{(l)} & \text{otherwise} \end{cases}$

Update weights via gradient step $\Theta_{ij}^{(l)} = \Theta_{ij}^{(l)} - \alpha D_{ij}^{(l)}$

Until weights converge or max #epochs is reached

Several Practical Tricks

Initialization

- Problem is highly non@convex, and heuristics exist to start training (at the least, randomize initial weights)

Optimization tricks

- Momentum@based methods
- Decaying step size
- Dropout to avoid co@adaptation / overfitting

Minibatch

- Use more than a single point to estimate gradient

...

Neural Networks vs Deep Learning?

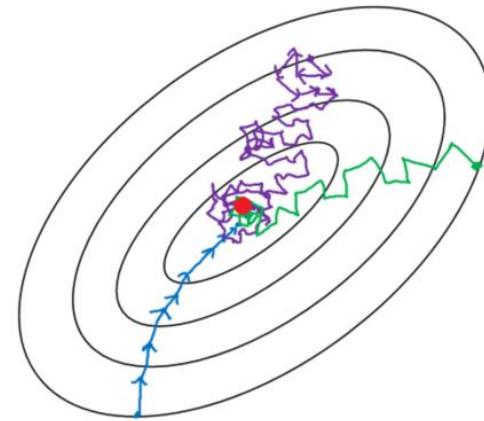
DNN are big neural networks

- Depth: often ~5 layers (but some have 20+)
 - Typically not fully connected!
- Width: hidden nodes per layer in the thousands
- Parameters: millions to billions

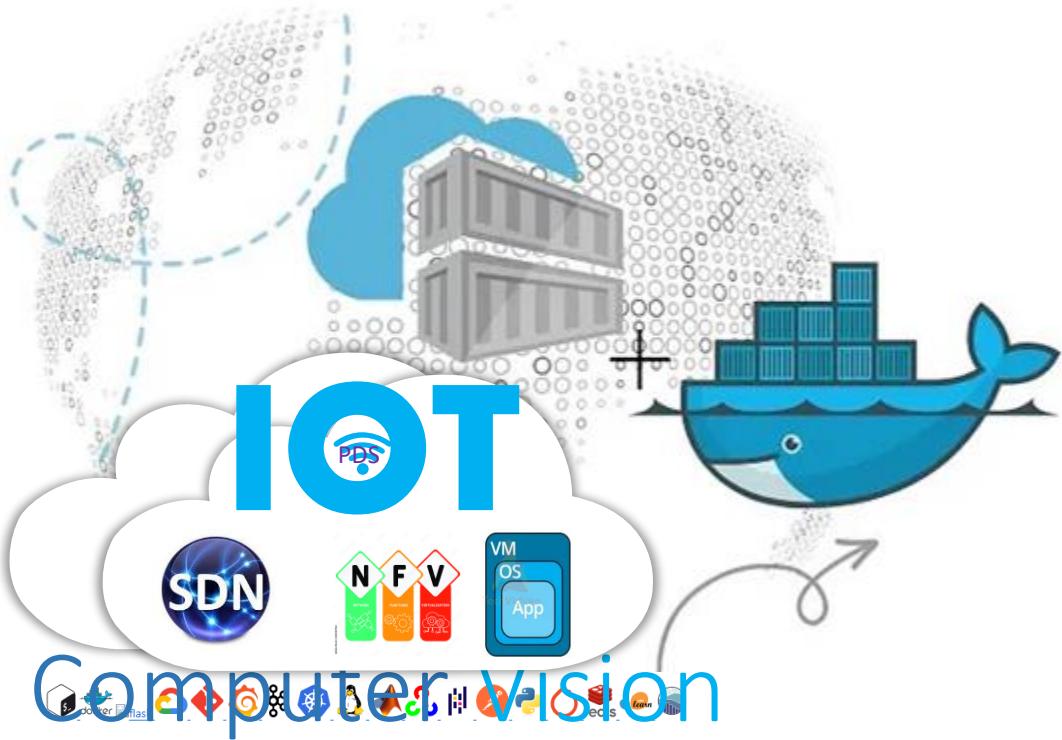
Algorithms / Computing

- New algorithms (pre@training, layer@wise training, dropout, etc.)
- Heavy computing requirements (GPUs are essential)

Mini Batch SGD



- Batch gradient descent
- Mini-batch gradient Descent
- Stochastic gradient descent



Mini-batch SGD

Loop:

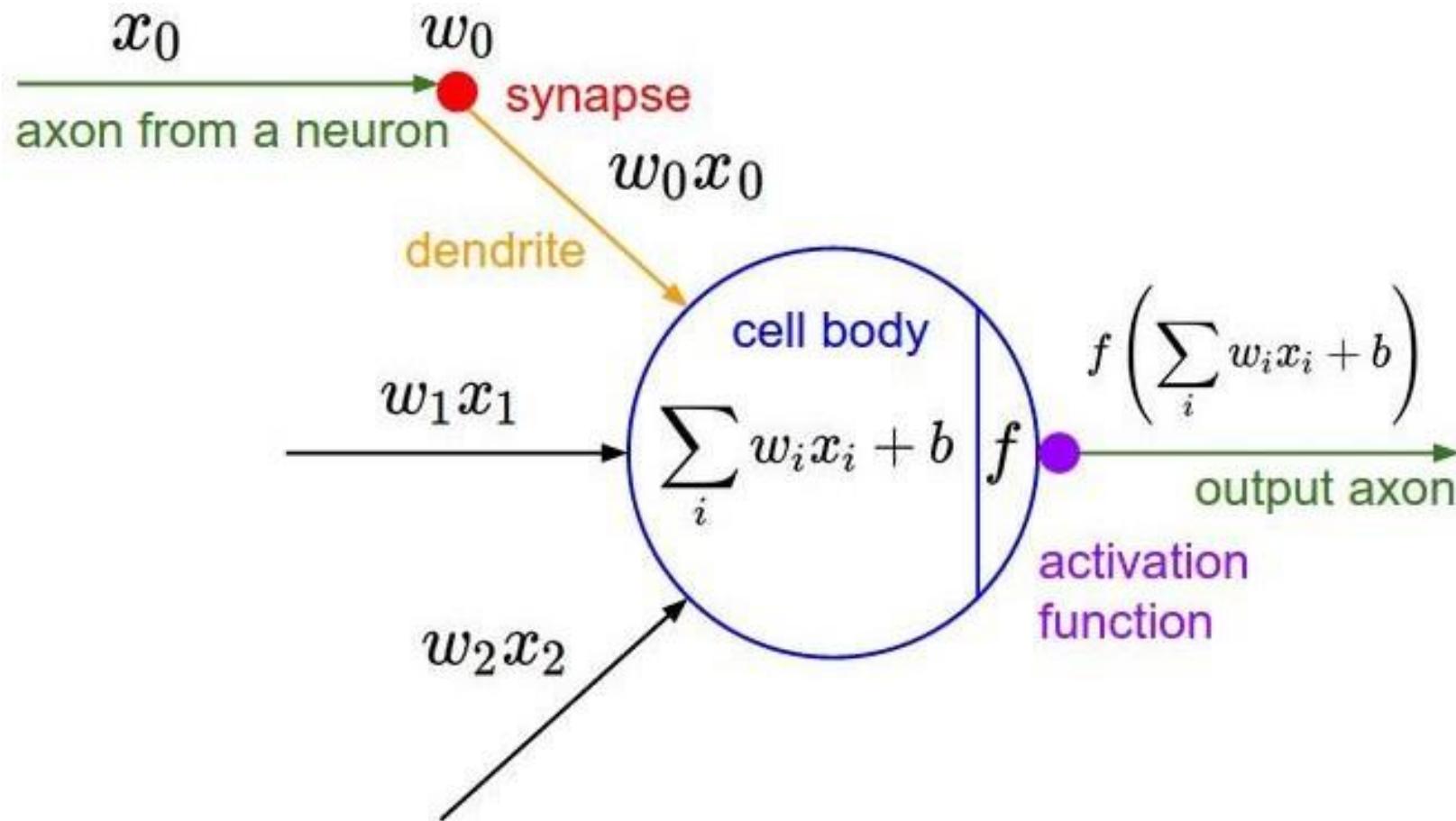
1. **Sample** a batch of data
2. **Forward** prop it through the graph (network), get loss
3. **Backprop** to calculate the gradients
4. **Update** the parameters using the gradient

Training Neural Networks

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

Activation Functions

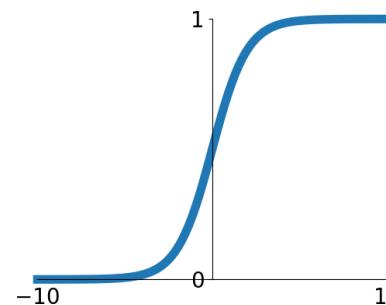
Activation Functions



Activation Functions

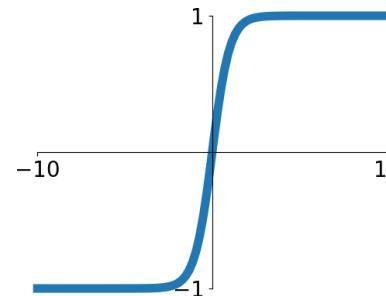
Sigmoid

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



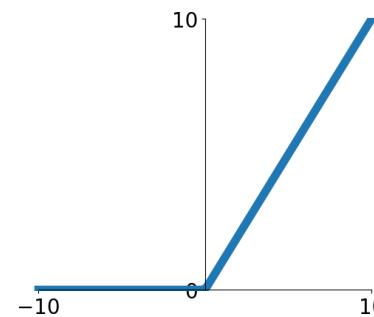
tanh

$$\tanh(x)$$

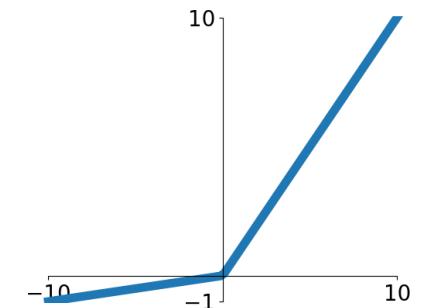


ReLU

$$\max(0, x)$$



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

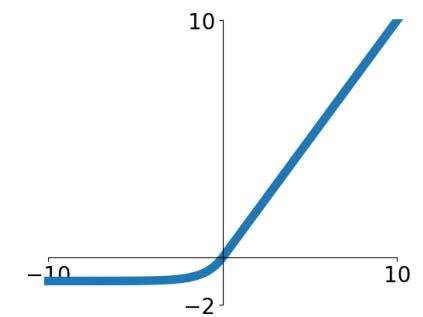


Maxout

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

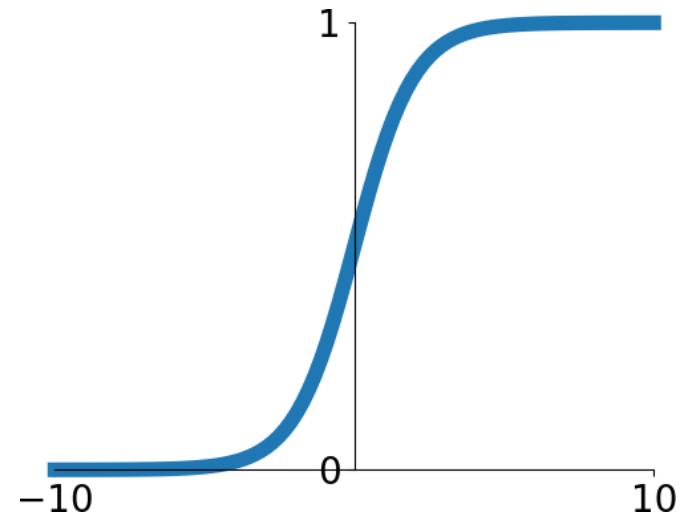
ELU

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



Activation Functions

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

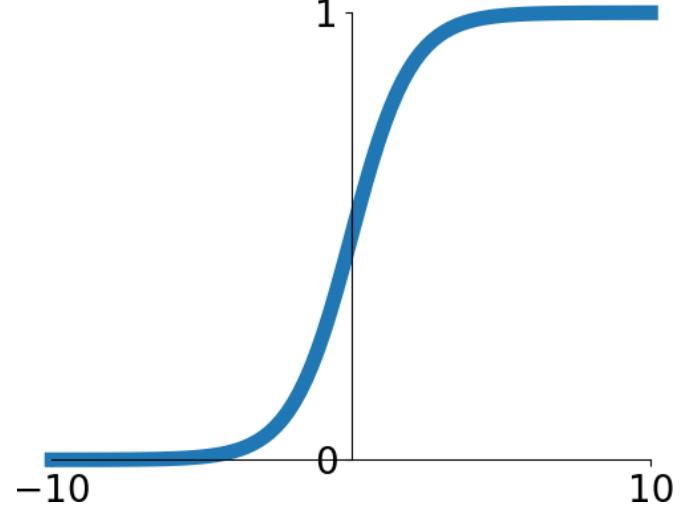


- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

Sigmoid

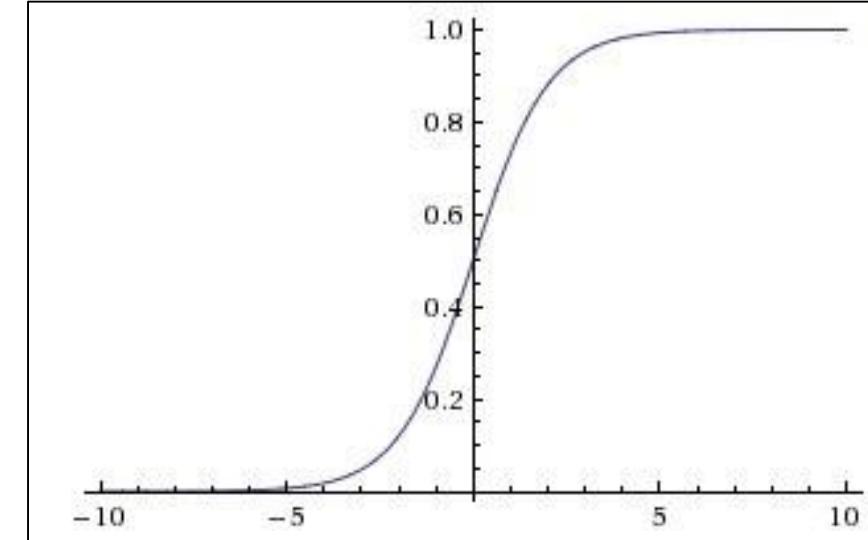
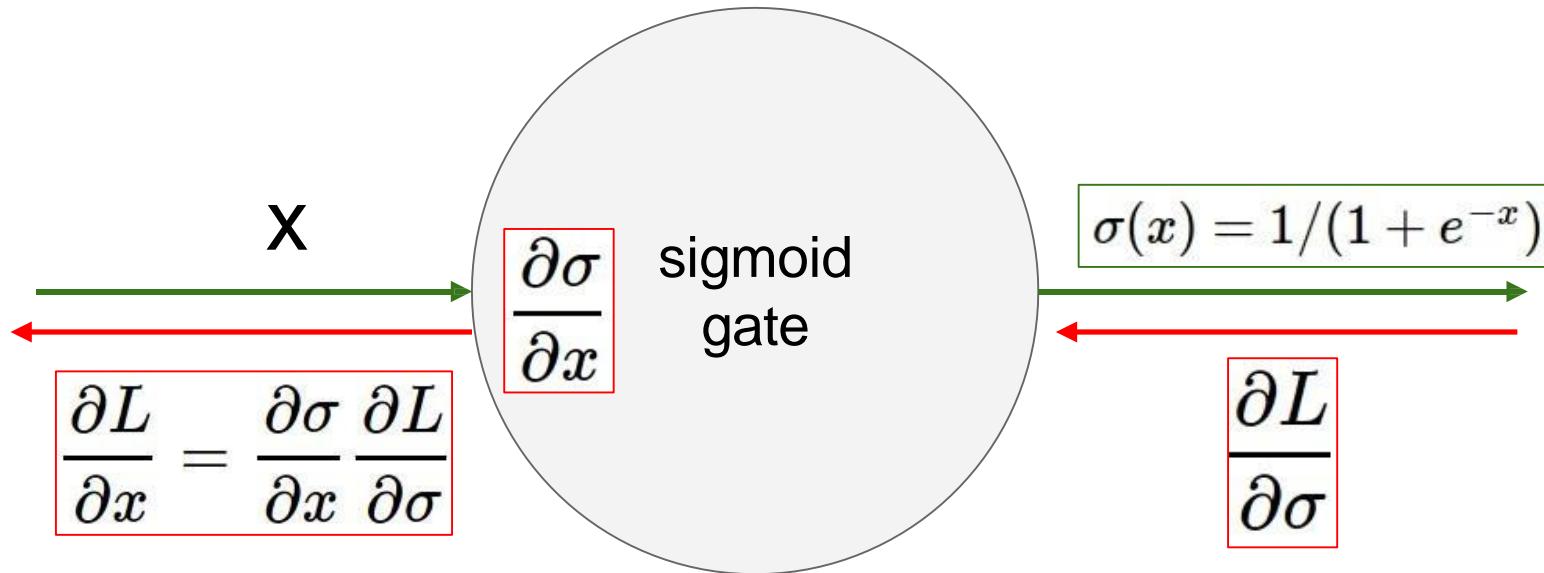
$$\sigma(x) = 1/(1 + e^{-x})$$

Activation Functions

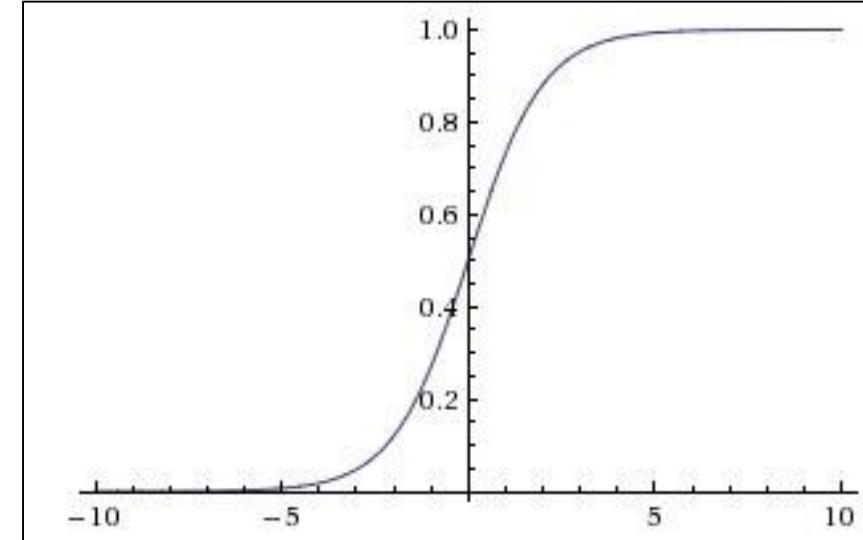
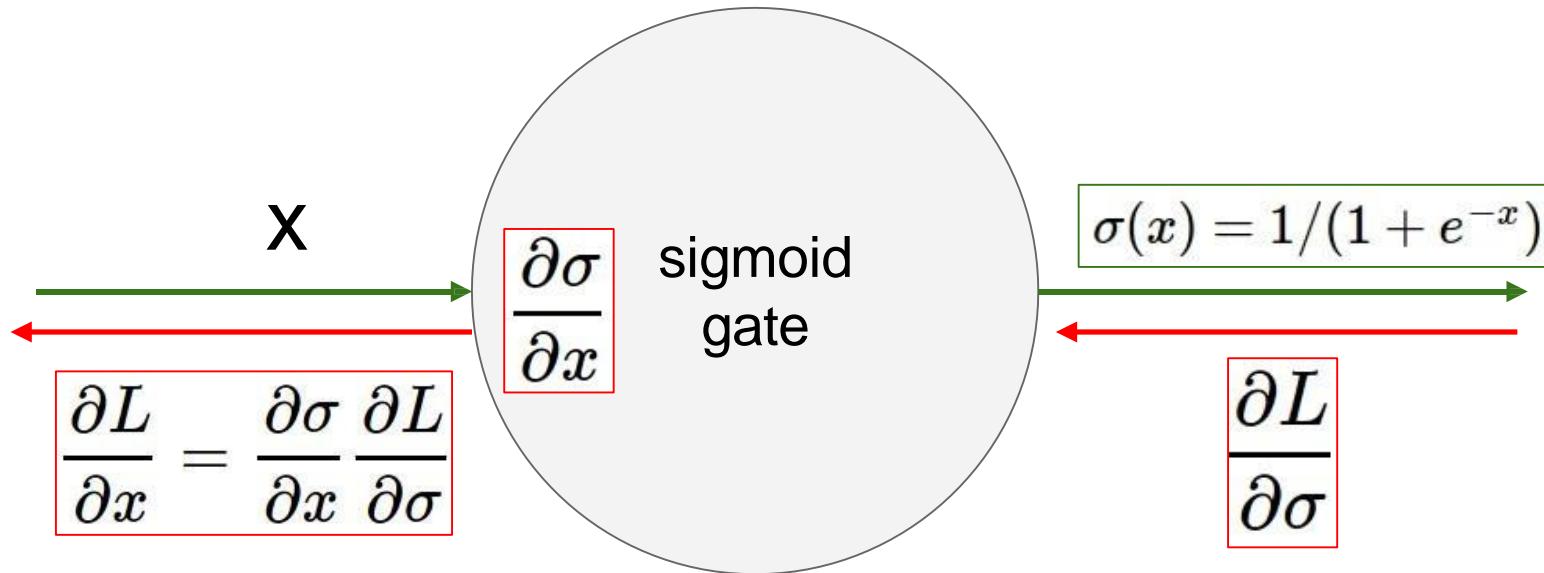


Sigmoid

- 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

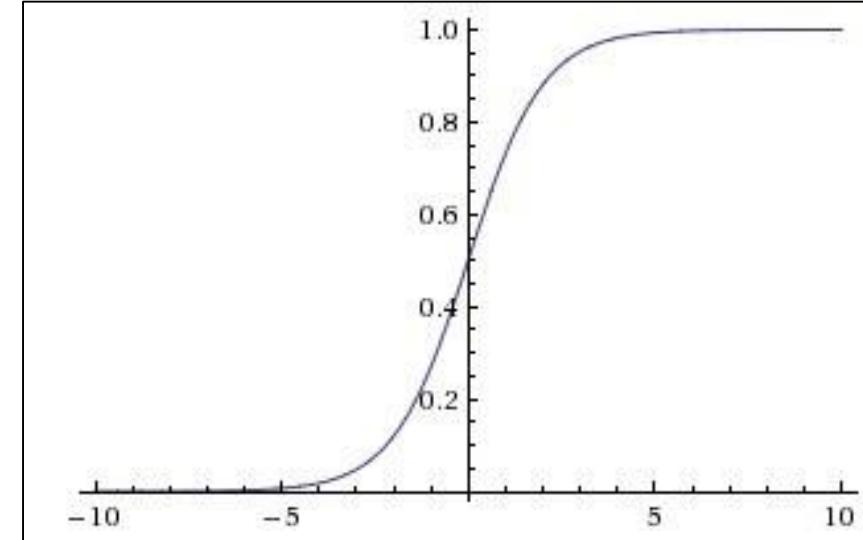
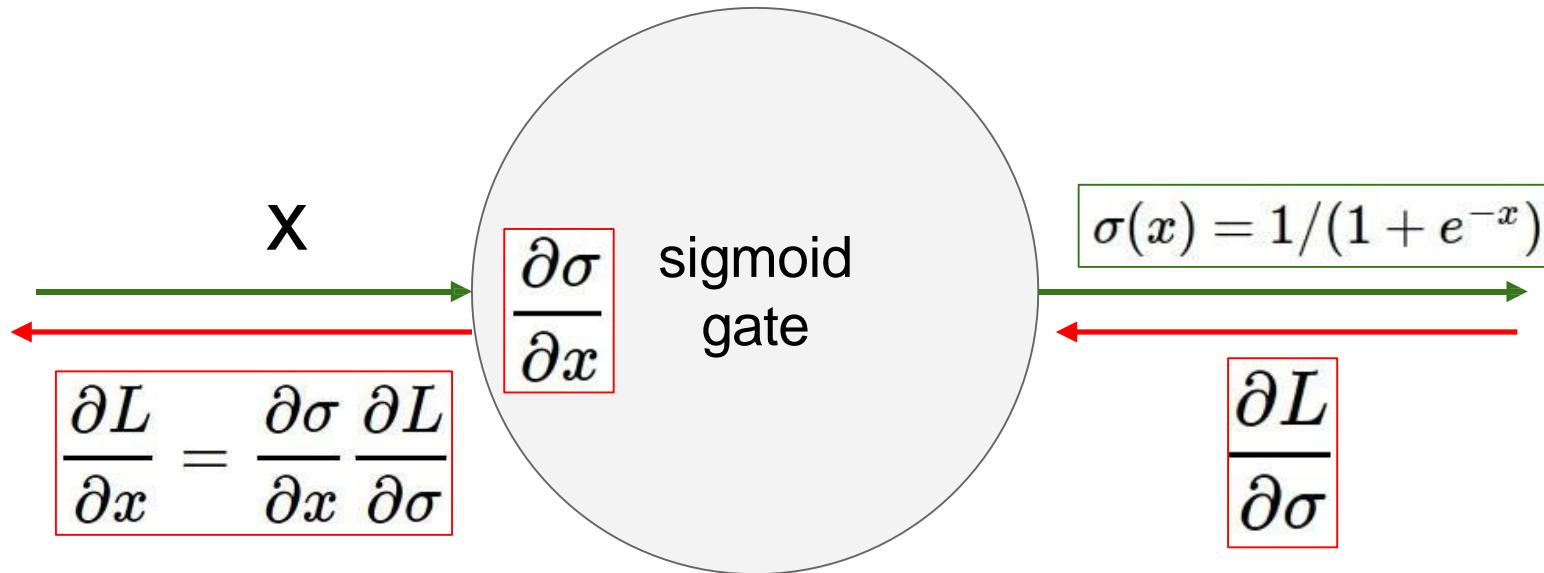


$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1 - \sigma(x))$$



What happens when $x = -10$?

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

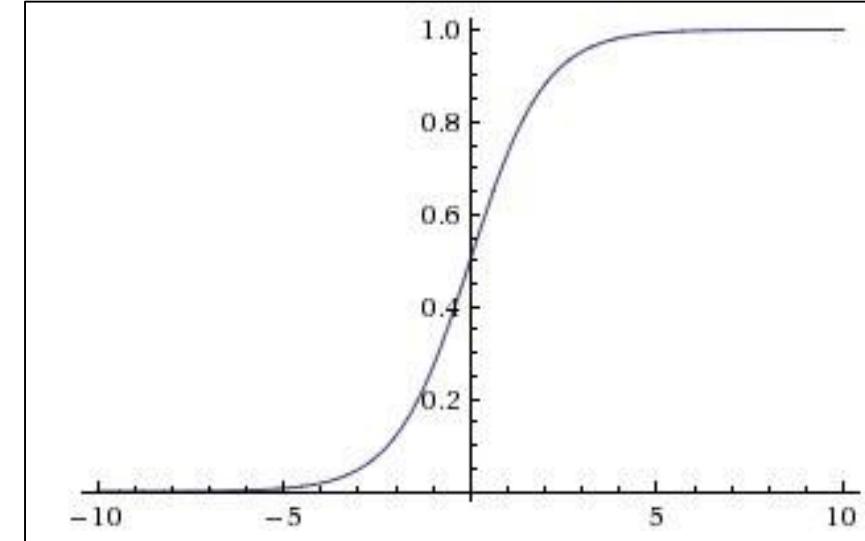
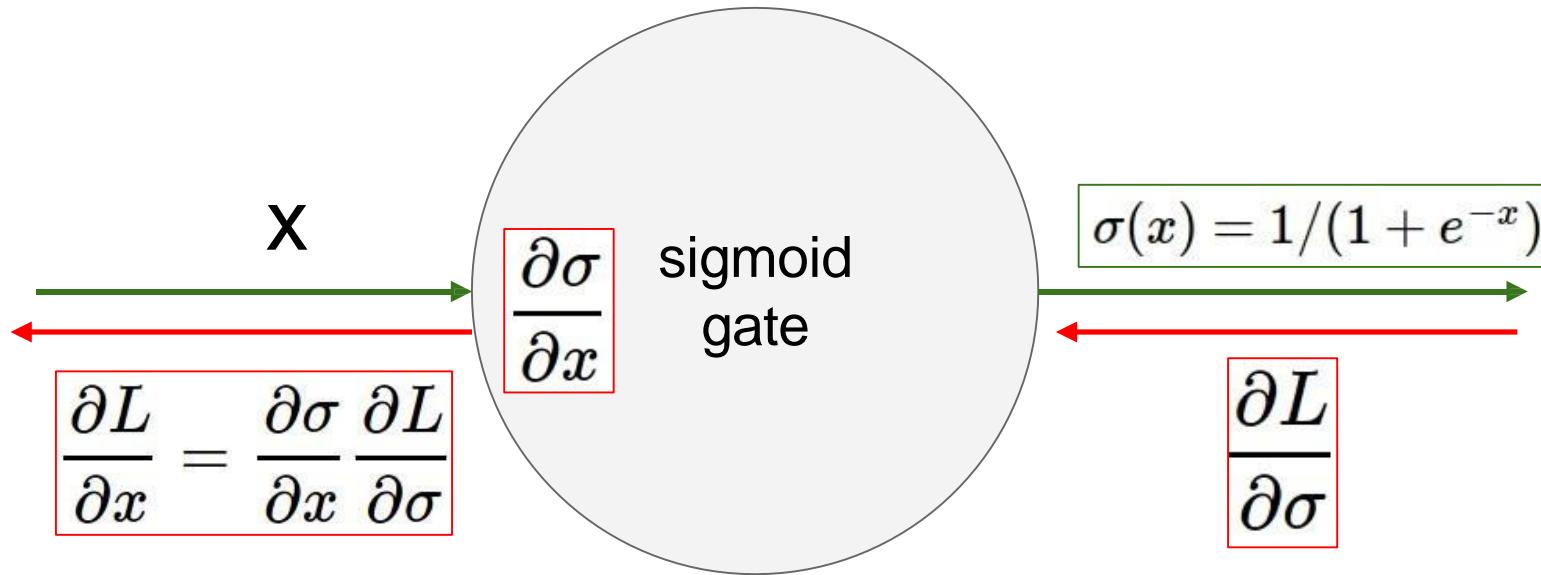


What happens when $x = -10$?

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

$$\sigma(x) = \sim 0$$

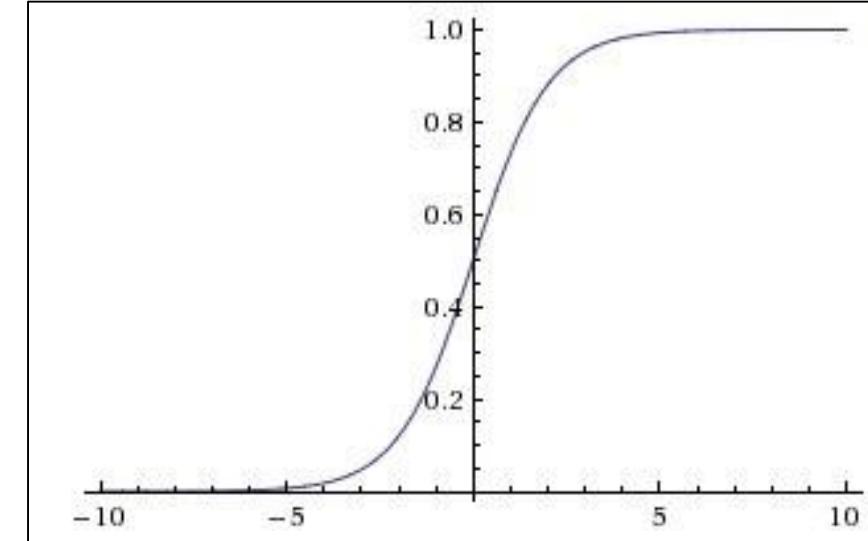
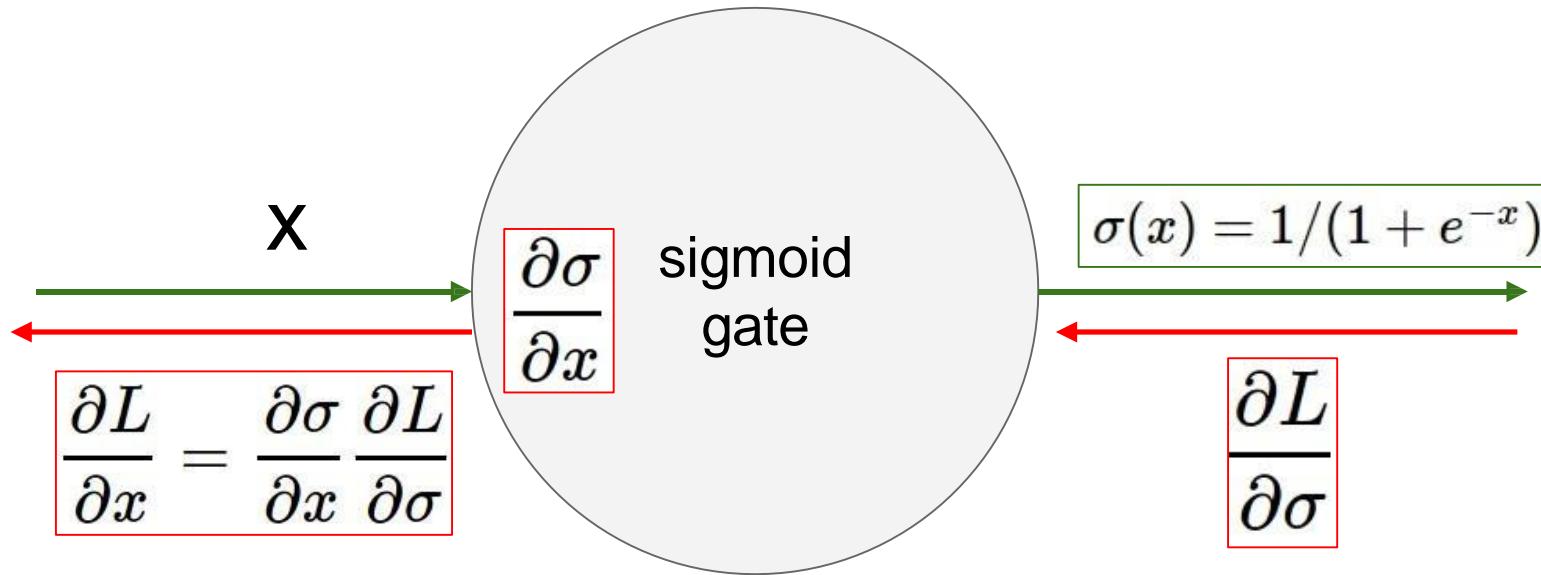
$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x) (1 - \sigma(x)) = 0(1 - 0) = 0$$



What happens when $x = -10$?

What happens when $x = 0$?

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



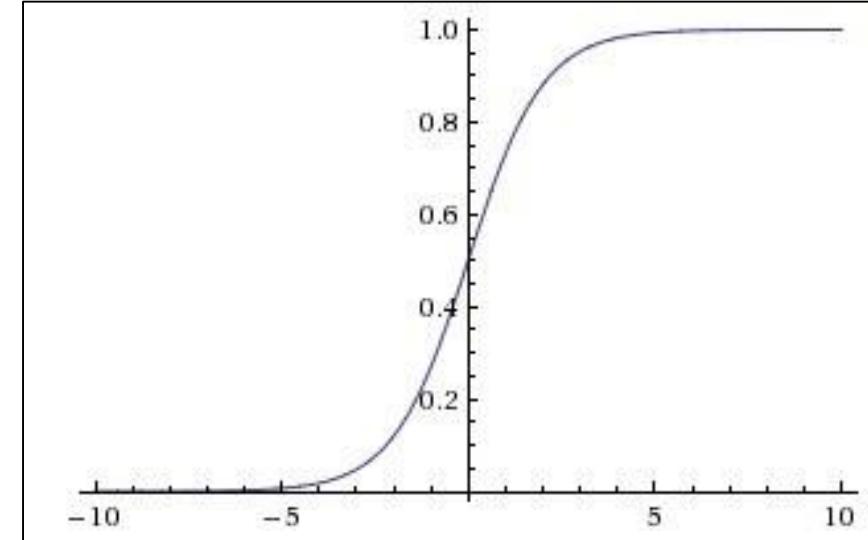
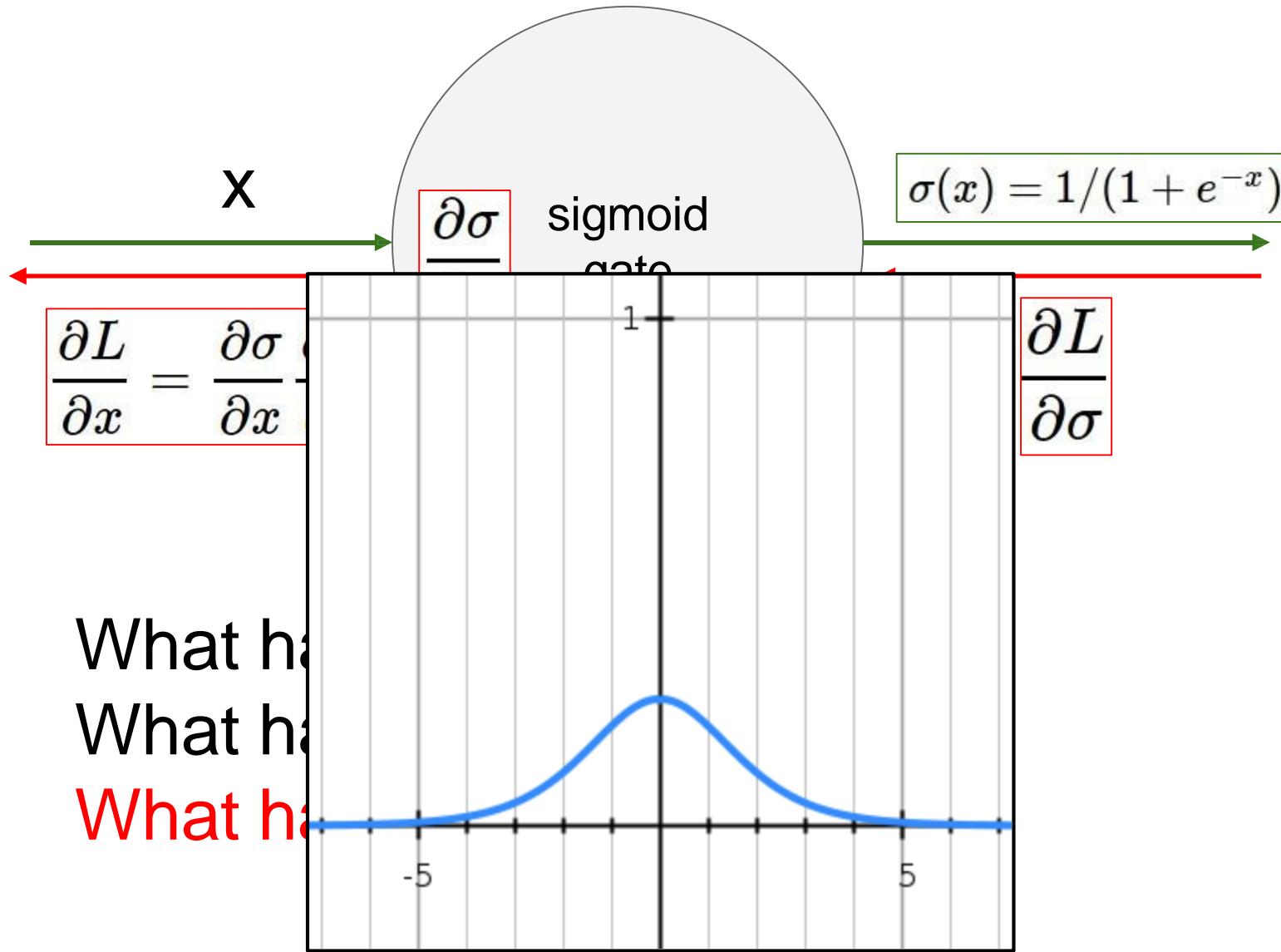
What happens when $x = -10$?

What happens when $x = 0$?

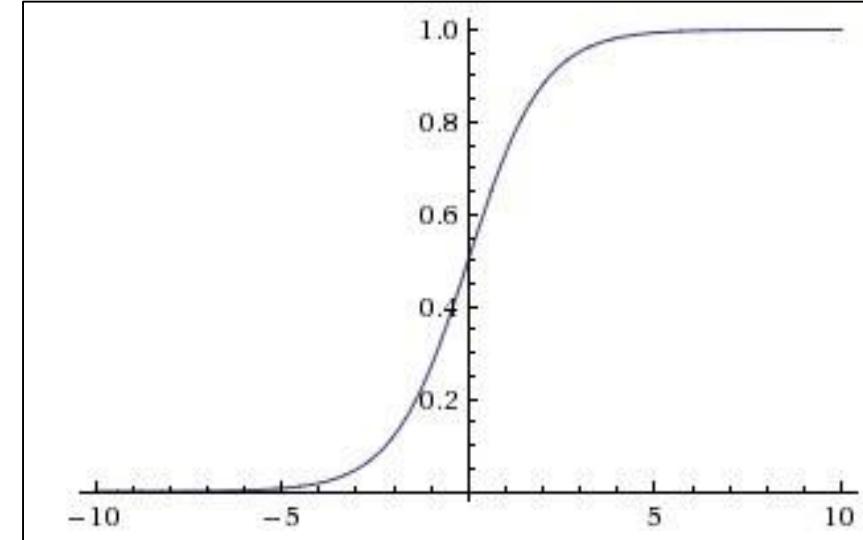
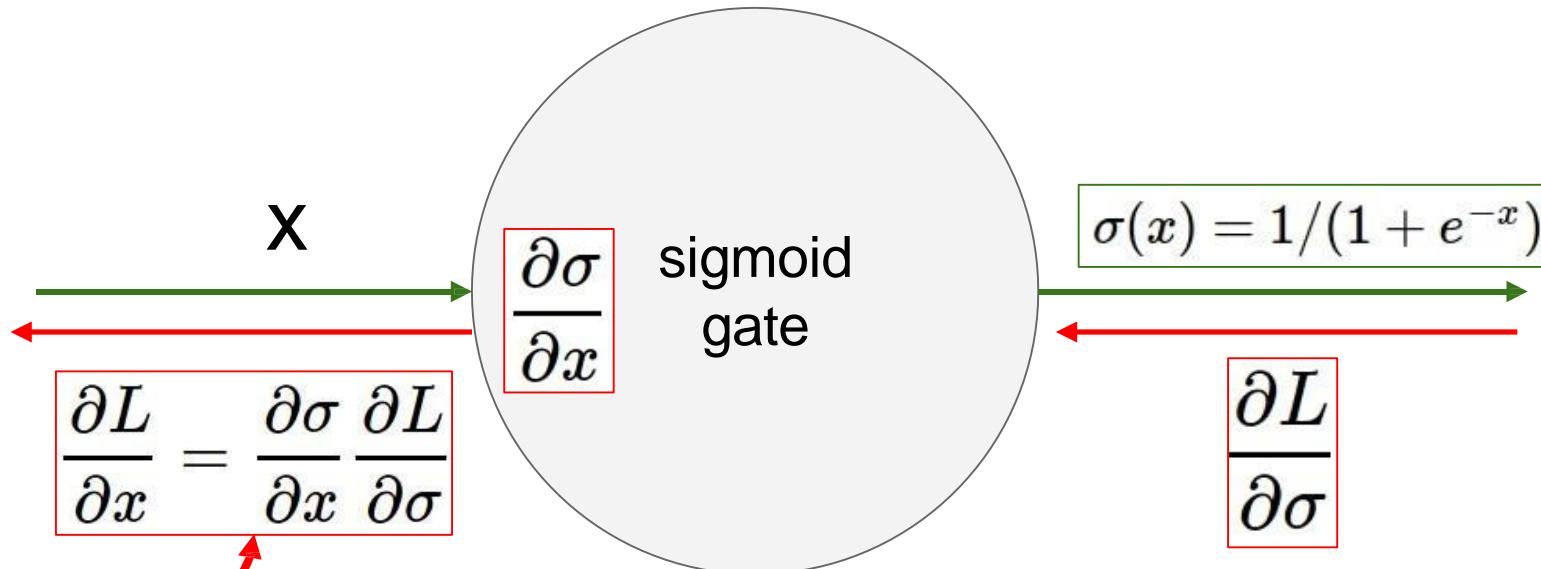
What happens when $x = 10$?

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

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



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

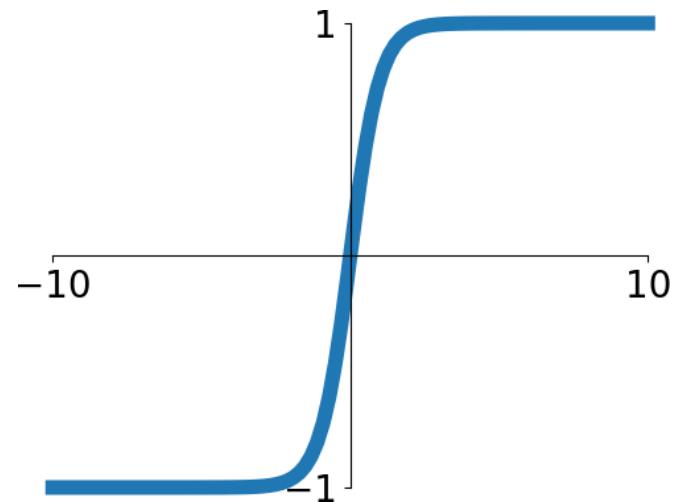


Why is this a problem?

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

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

Activation Functions

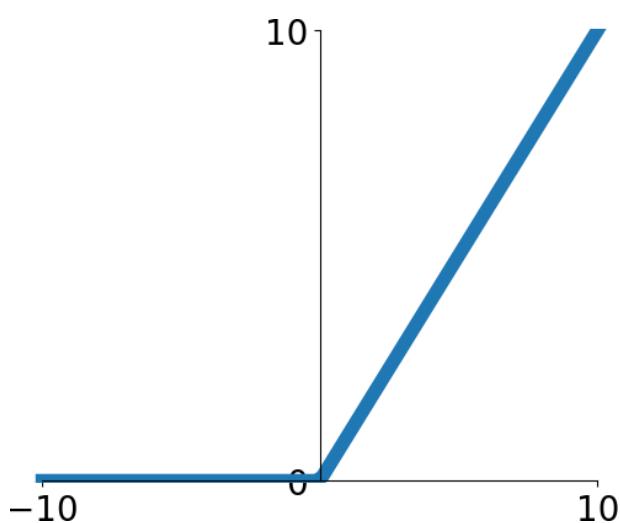


$\tanh(x)$

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

[LeCun et al., 1991]

Activation Functions

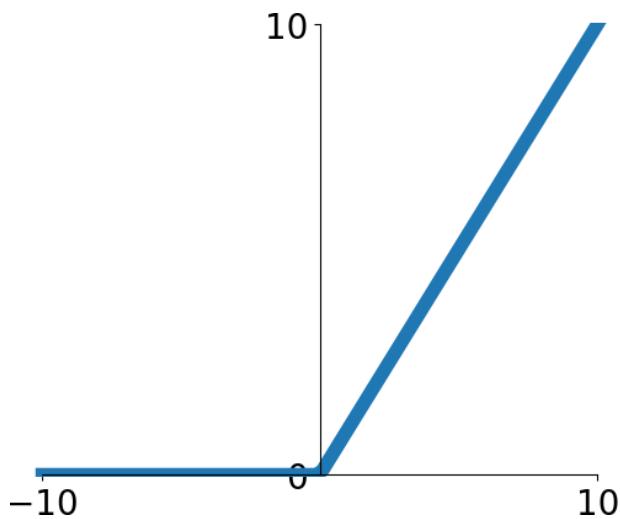


- Computes $f(x) = \max(0, x)$
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)

ReLU
(Rectified Linear Unit)

[Krizhevsky et al., 2012]

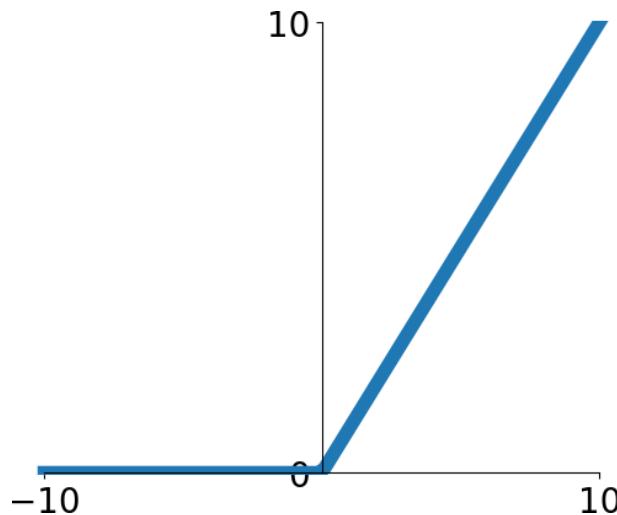
Activation Functions



- 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

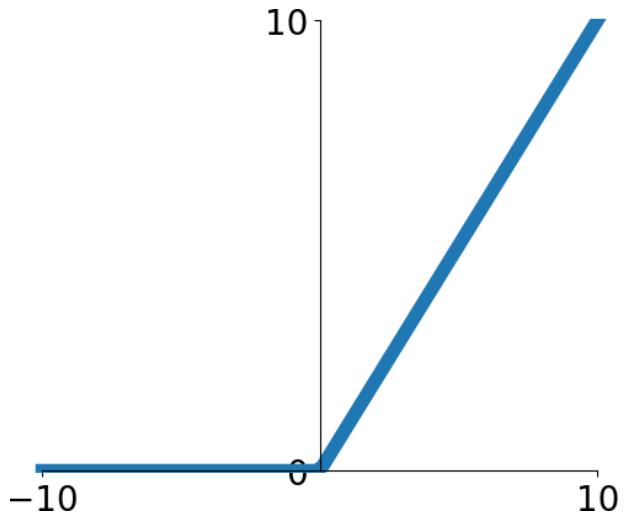
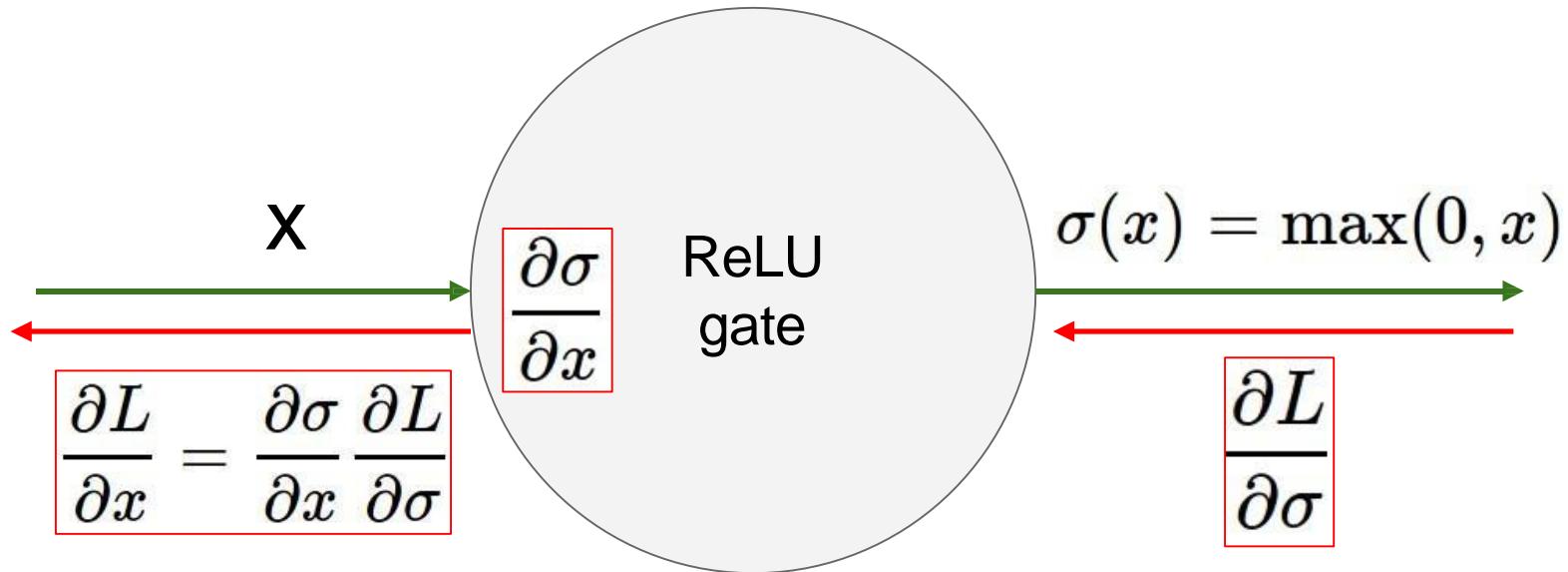
ReLU
(Rectified Linear Unit)

Activation Functions



ReLU
(Rectified Linear Unit)

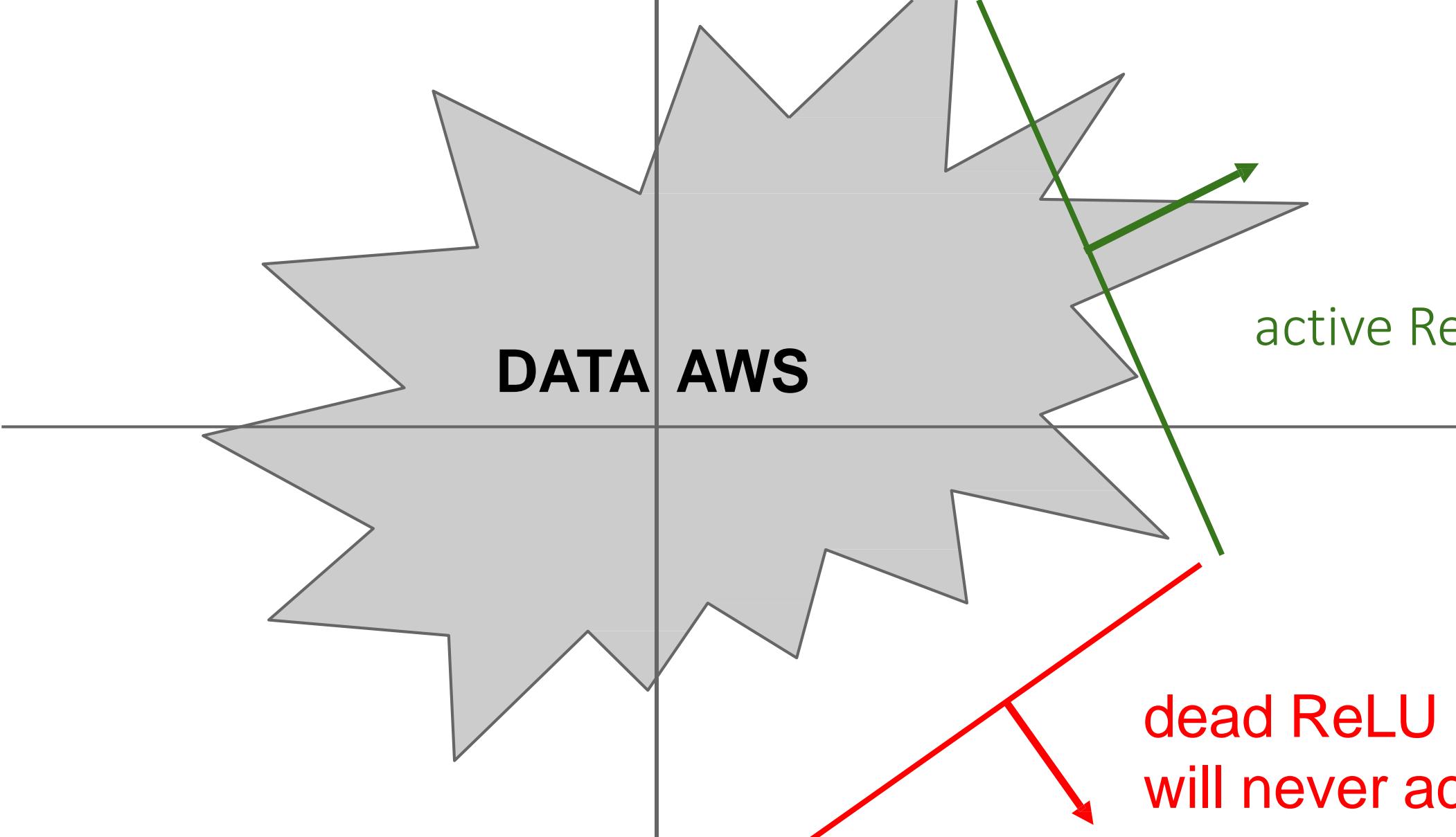
- Computes $f(x) = \max(0,x)$
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- Not zero-centered output
- An annoyance:
hint: what is the gradient when $x < 0$?



What happens when $x = -10$?

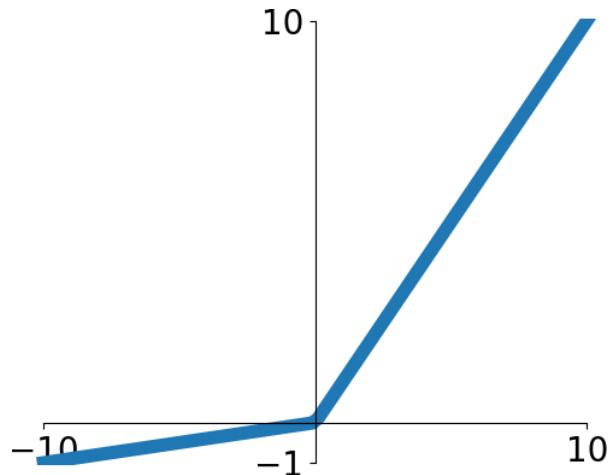
What happens when $x = 0$?

What happens when $x = 10$?



Activation Functions

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



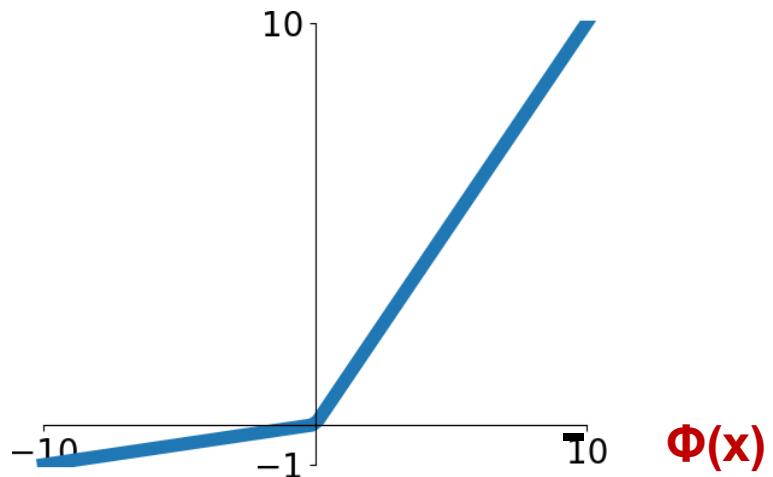
- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- **will not “die”.**

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)

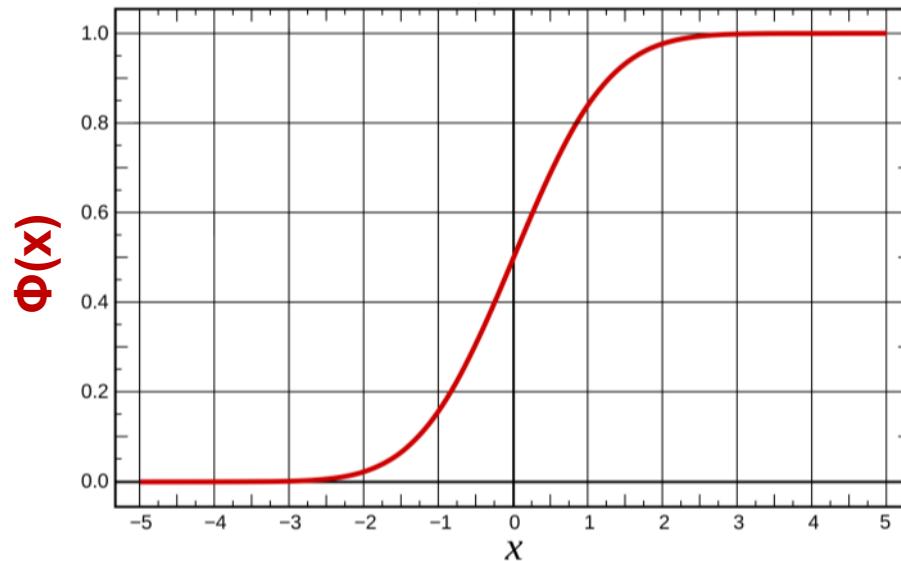
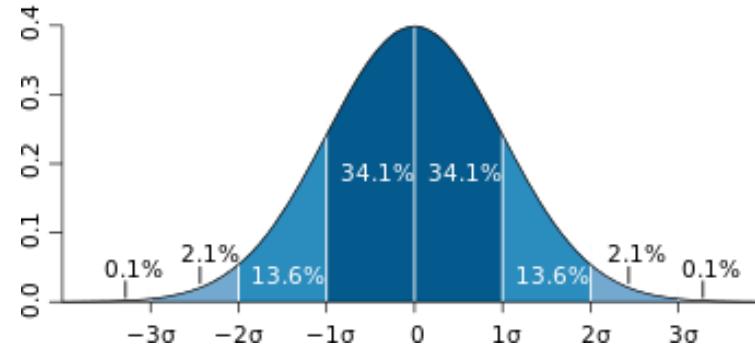
$$f(x) = \max(\alpha x, x)$$

backprop into α (parameter)

Activation Functions

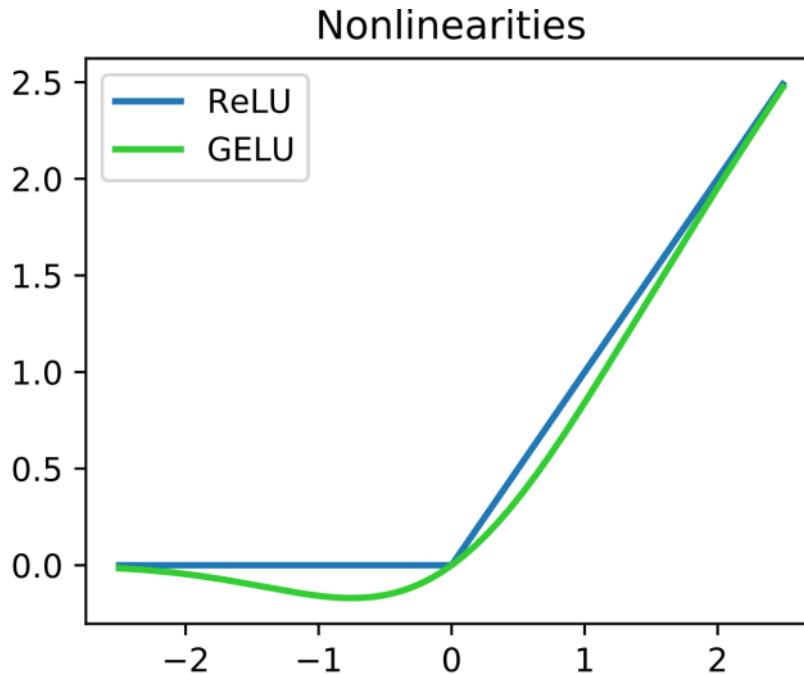
[Hendrycks et al., 2016]

- Computes $f(x) = x * \Phi(x)$



Sources:
https://en.wikipedia.org/wiki/Normal_distribution,
https://en.m.wikipedia.org/wiki/File:Cumulative_distribution_function_for_normal_distribution_mean_0_and_sd_1.png

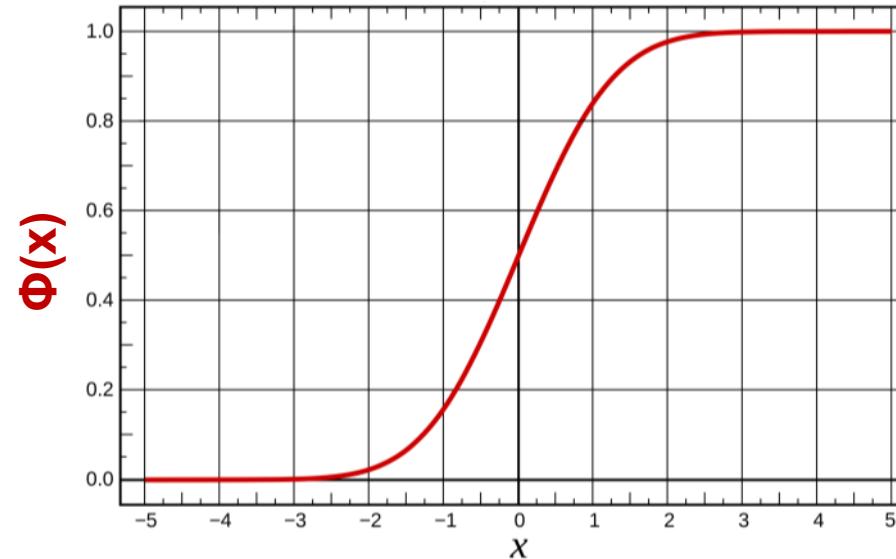
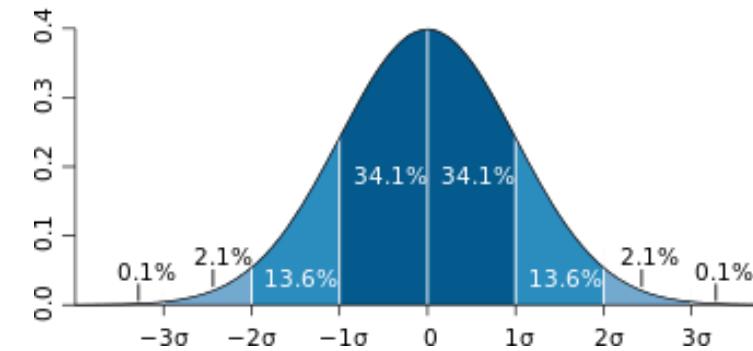
Activation Functions



GELU
(Gaussian Error
Linear Unit)

[Hendrycks et al., 2016]

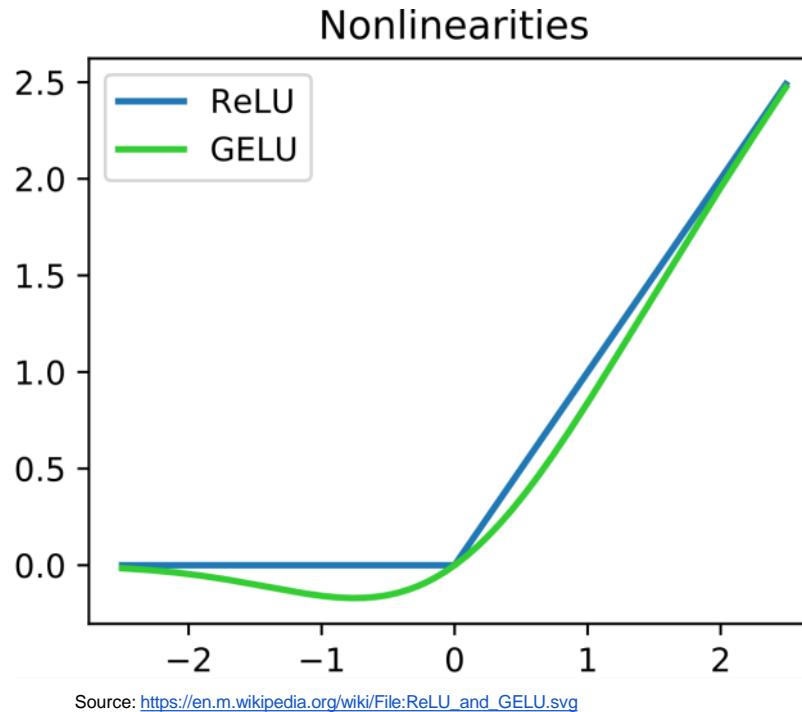
- Computes $f(x) = x^* \Phi(x)$



Sources:
https://en.wikipedia.org/wiki/Normal_distribution,
https://en.m.wikipedia.org/wiki/File:Cumulative_distribution_function_for_normal_distribution_mean_0_and_sd_1.png

Activation Functions

[Hendrycks et al., 2016]



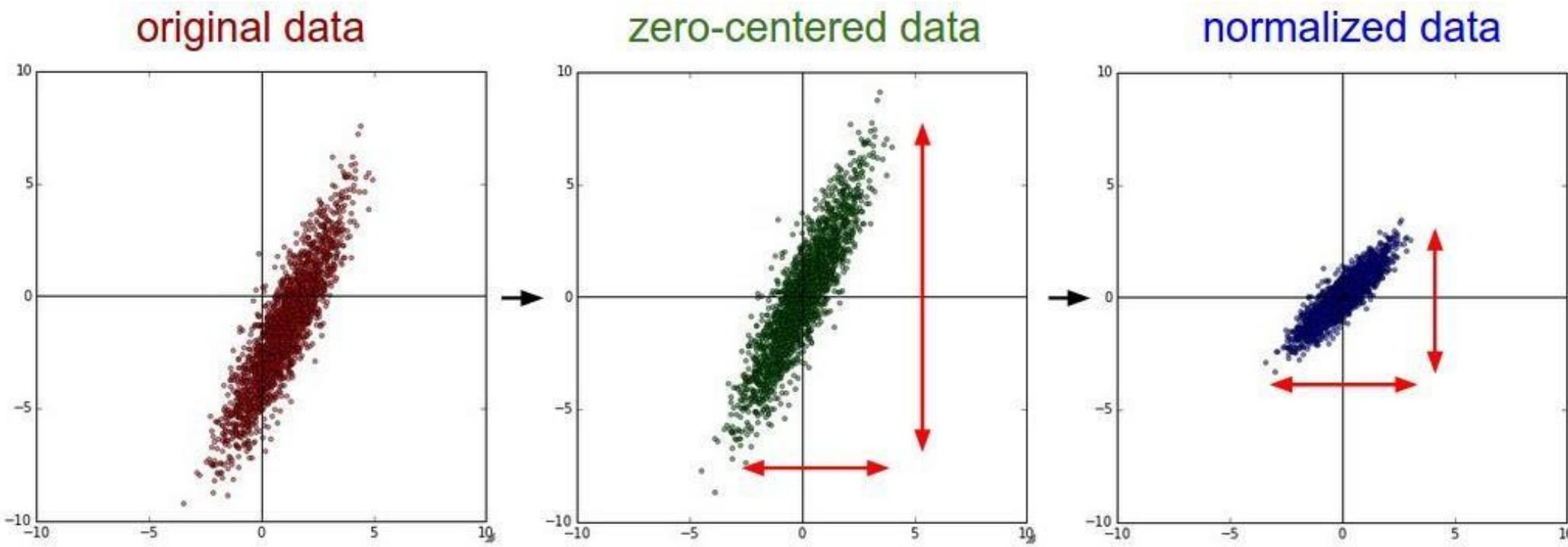
GELU
(Gaussian Error
Linear Unit)

- Computes $f(x) = x^* \Phi(x)$
- Very nice behavior around 0
- Smoothness facilitates training in practice
- Higher computational cost than ReLU
- Large negative values can still have gradient $\rightarrow 0$

TLDR: In practice:

- Use ReLU. Be careful with your learning rates
- Try out Leaky ReLU / PReLU / GELU
 - To squeeze out some marginal gains
- Don't use sigmoid or tanh

Data Preprocessing



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

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

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

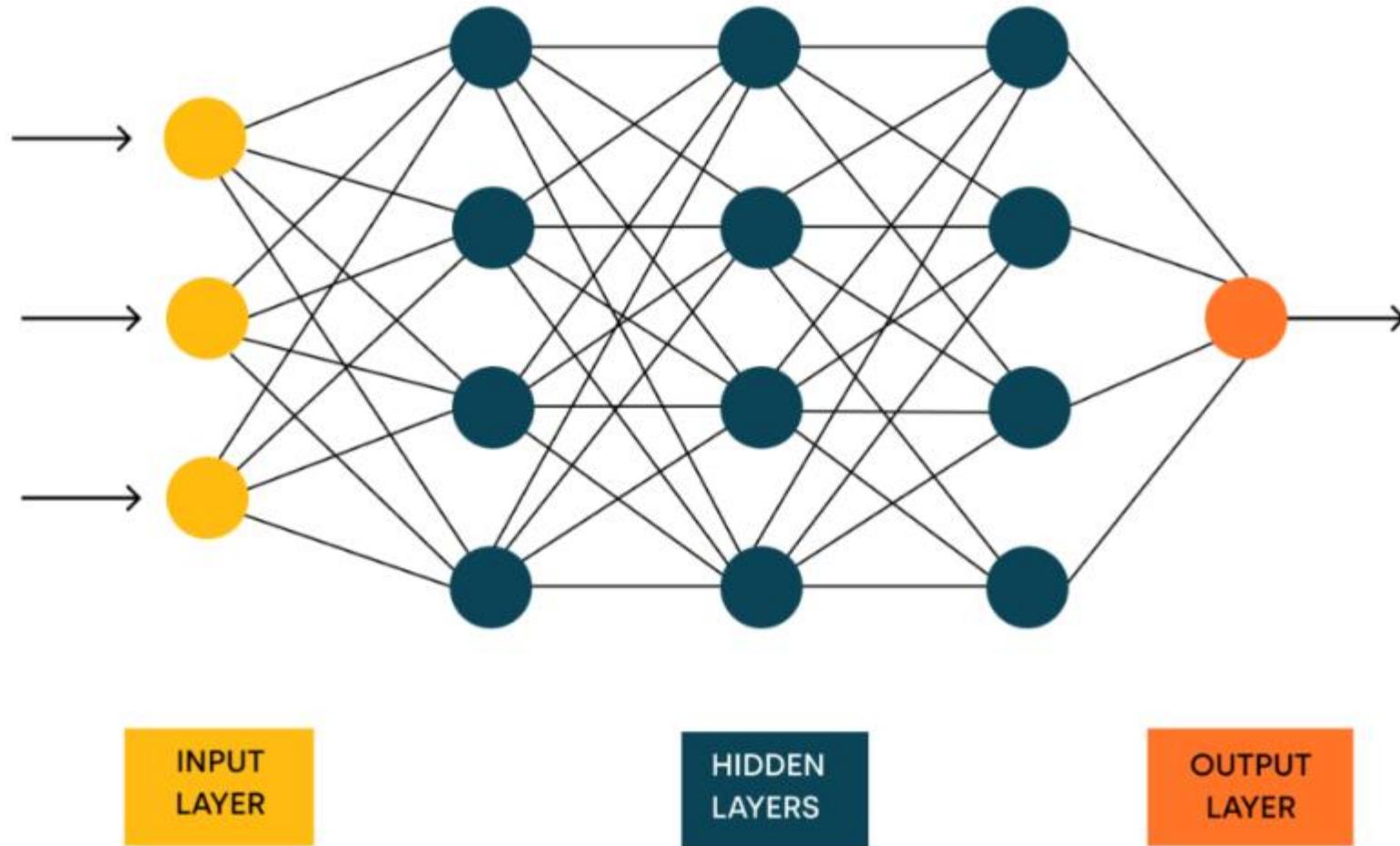
TLDR: In practice for Images: center only

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 and beyond)
(mean along each channel = 3 numbers)

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?

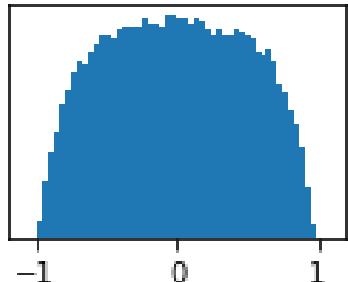
Weight Initialization: Activation statistics

```
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)
```

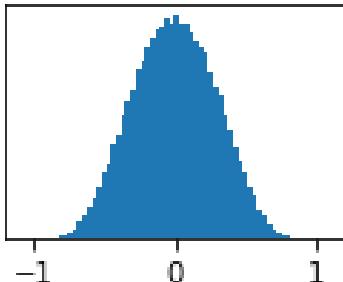
All activations tend to zero for deeper network layers

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

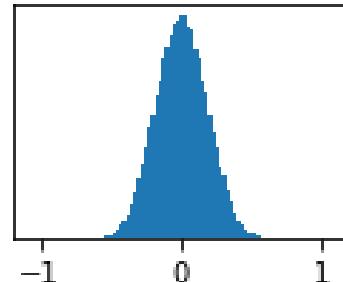
Layer 1
mean=-0.00
std=0.49



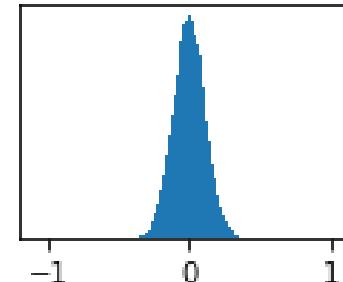
Layer 2
mean=0.00
std=0.29



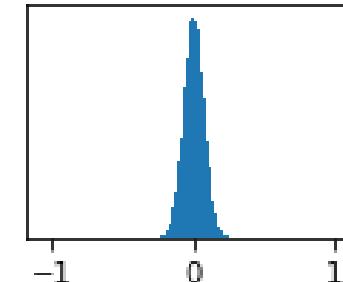
Layer 3
mean=0.00
std=0.18



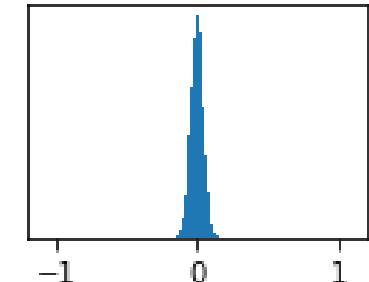
Layer 4
mean=-0.00
std=0.11



Layer 5
mean=-0.00
std=0.07



Layer 6
mean=0.00
std=0.05



Weight Initialization: Activation statistics

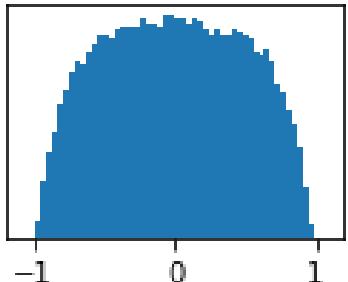
```
dims = [4096] * 7      Forward pass for a 6-layer  
hs = []                  net with hidden size 4096  
x = np.random.randn(16, dims[0])  
for Din, Dout in zip(dims[:-1], dims[1:]):  
    W = 0.01 * np.random.randn(Din, Dout)  
    x = np.tanh(x.dot(W))  
    hs.append(x)
```

All activations tend to zero for deeper network layers

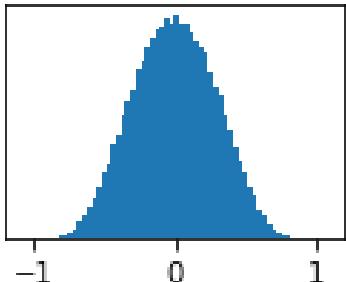
Q: What do the gradients dL/dW look like?

A: All zero, no learning =(

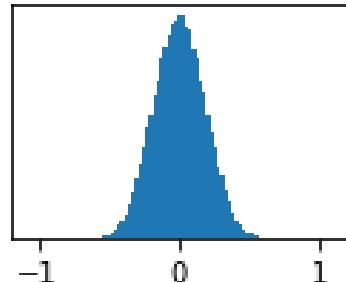
Layer 1
mean=-0.00
std=0.49



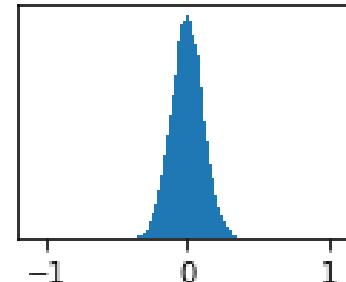
Layer 2
mean=0.00
std=0.29



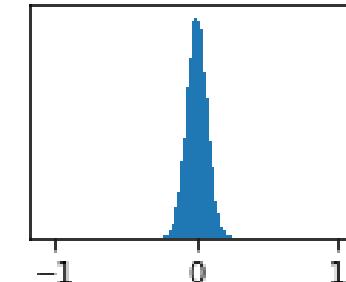
Layer 3
mean=0.00
std=0.18



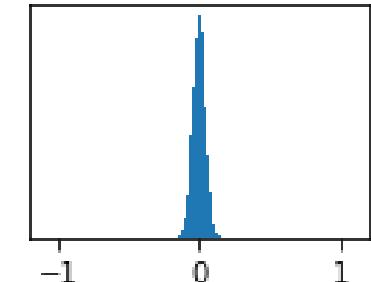
Layer 4
mean=-0.00
std=0.11



Layer 5
mean=-0.00
std=0.07



Layer 6
mean=0.00
std=0.05



Weight Initialization: Activation statistics

```
dims = [4096] * 7      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)
```

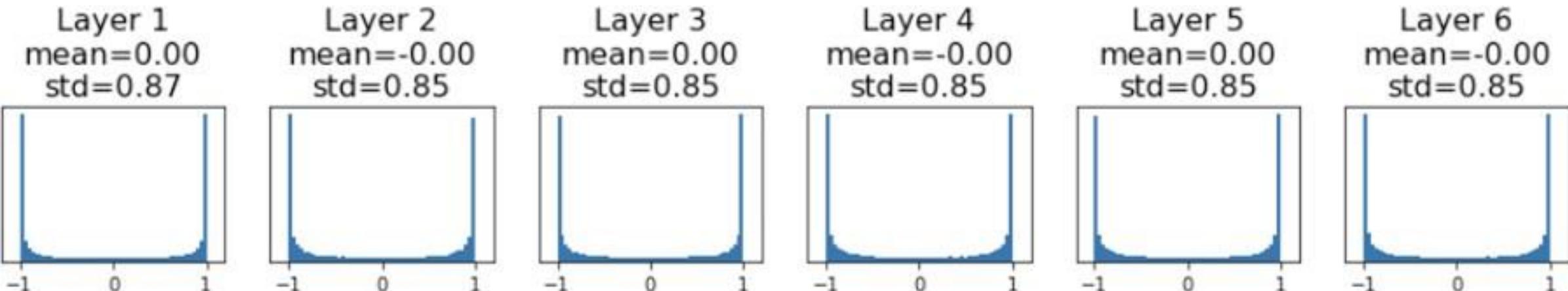
What will happen to the activations for the last layer?

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?



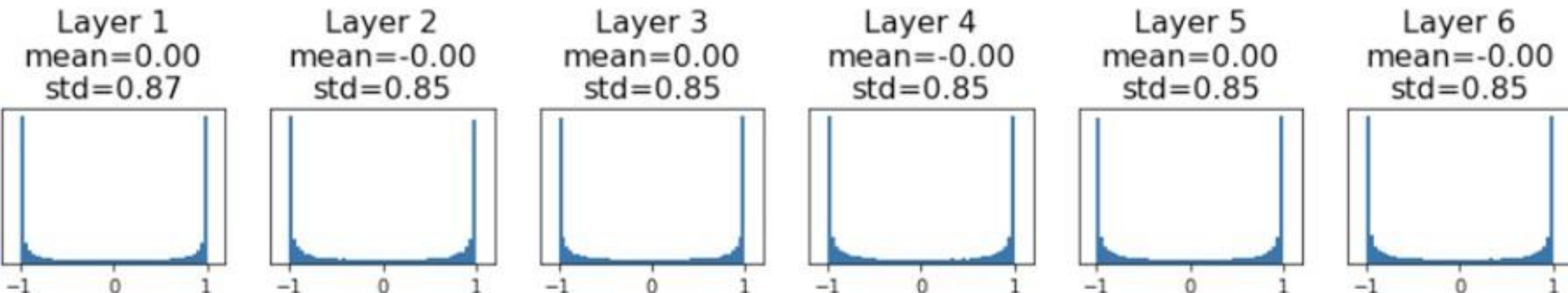
Weight Initialization: Activation statistics

```
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

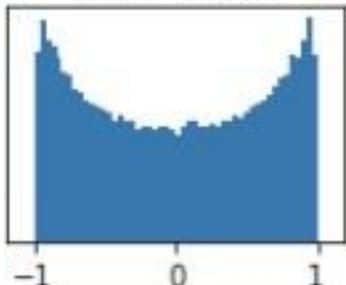
Weight Initialization: “Xavier” Initialization

```
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)
```

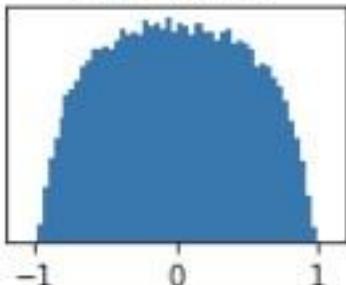
“Xavier” initialization:
 $\text{std} = 1/\sqrt{\text{Din}}$

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

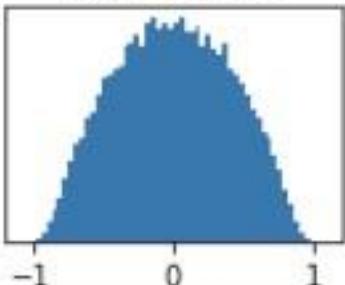
Layer 1
mean=-0.00
std=0.63



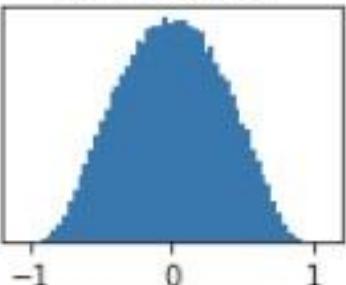
Layer 2
mean=-0.00
std=0.49



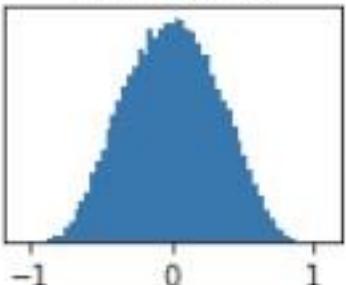
Layer 3
mean=0.00
std=0.41



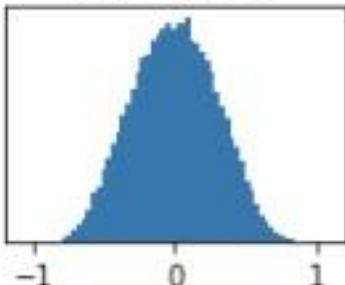
Layer 4
mean=0.00
std=0.36



Layer 5
mean=0.00
std=0.32



Layer 6
mean=-0.00
std=0.30



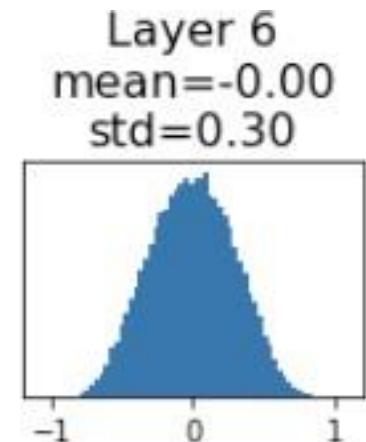
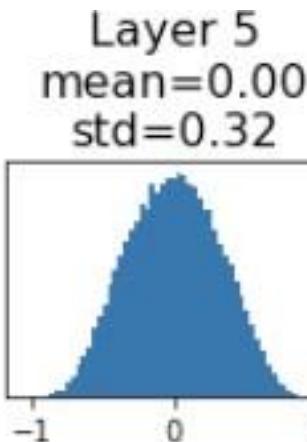
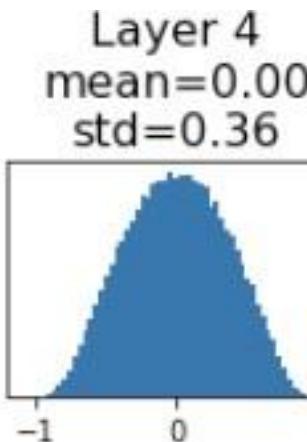
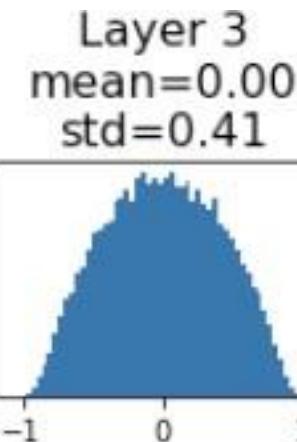
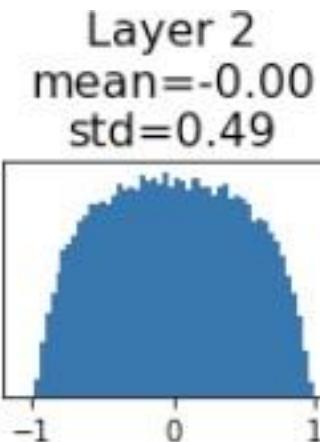
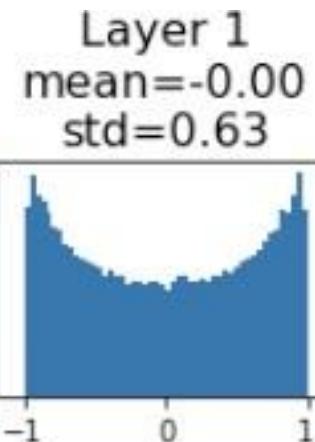
Glorot and Bengio, “Understanding the difficulty of training deep feedforward neural networks”, AISTAT 2010

Weight Initialization: “Xavier” Initialization

```
dims = [4096] * 7          "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)
```

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

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



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

Weight Initialization: What about ReLU?

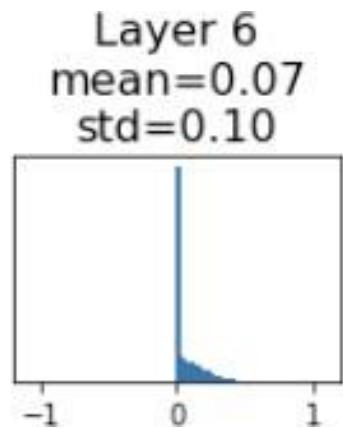
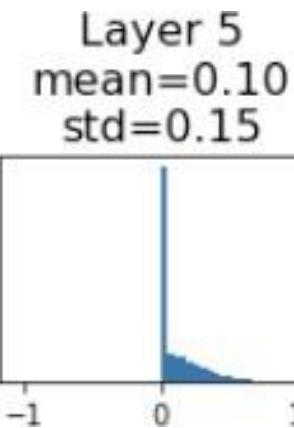
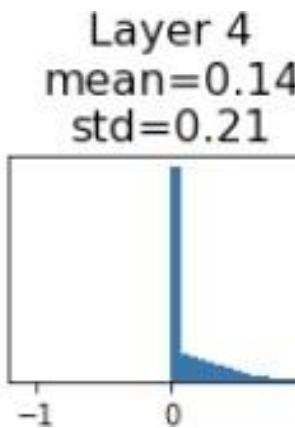
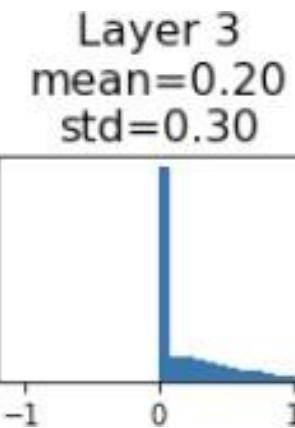
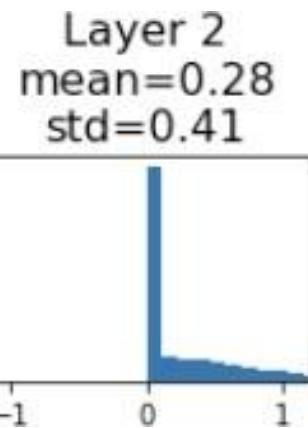
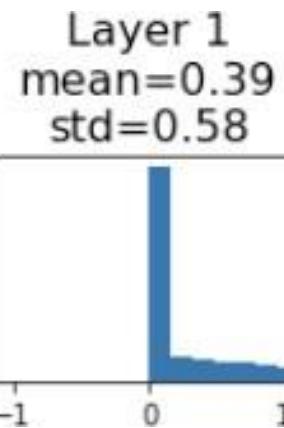
```
dims = [4096] * 7      Change from tanh to ReLU
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

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

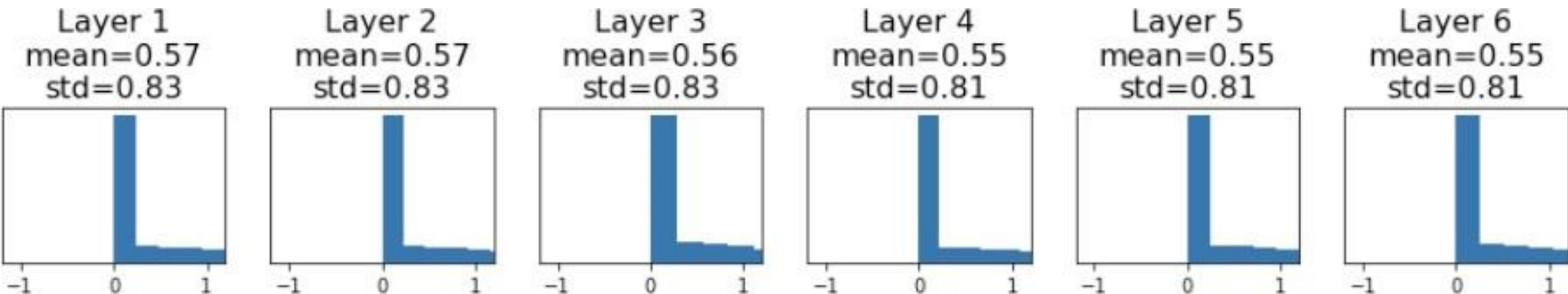
Activations collapse to zero again, no learning =(



Weight Initialization: Kaiming / MSRA Initialization

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

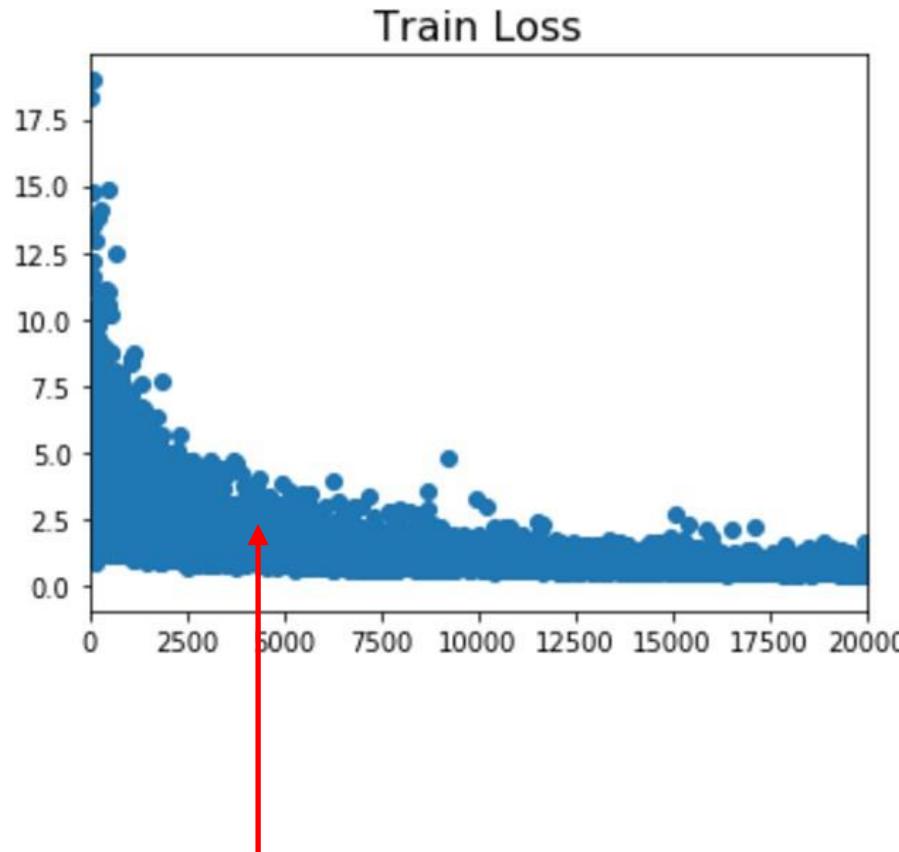
“Just right”: Activations are nicely scaled for all layers!



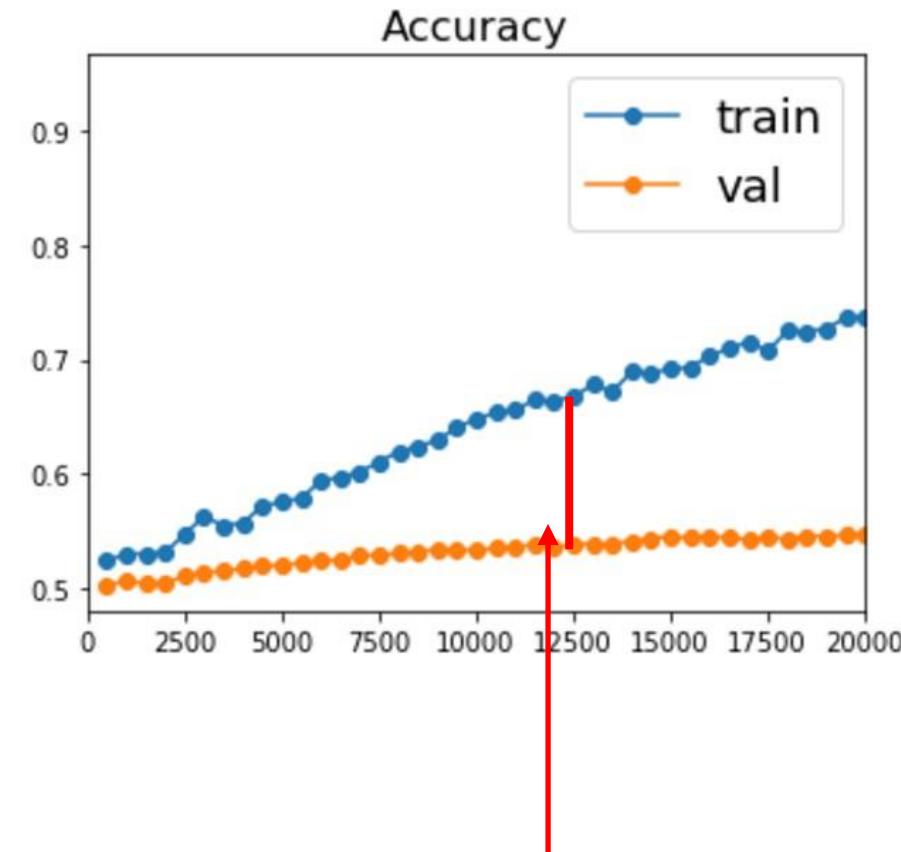
He et al, “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification”, ICCV 2015

Training vs. Testing Error

Beyond Training Error

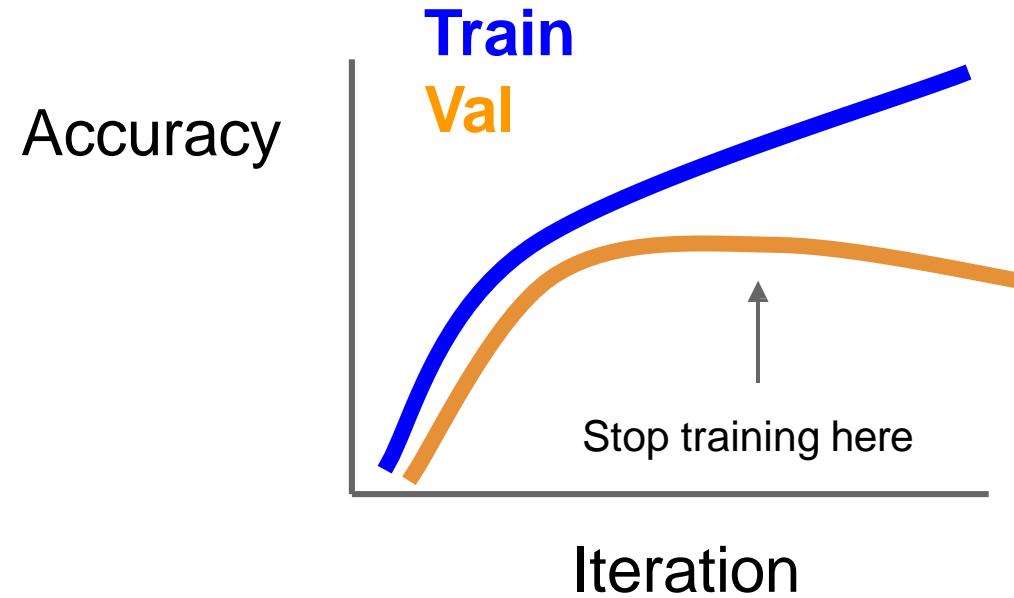
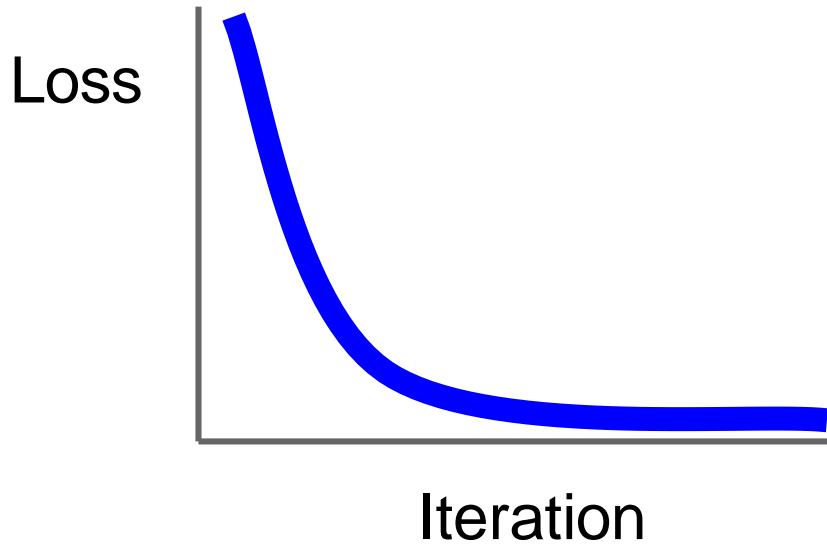


Better optimization algorithms help reduce training loss



But we really care about error on new data - how to reduce the gap?

Early Stopping: Always do this



Stop training the model when accuracy on the validation set decreases
Or train for a long time, but always keep track of the model snapshot
that worked best on val

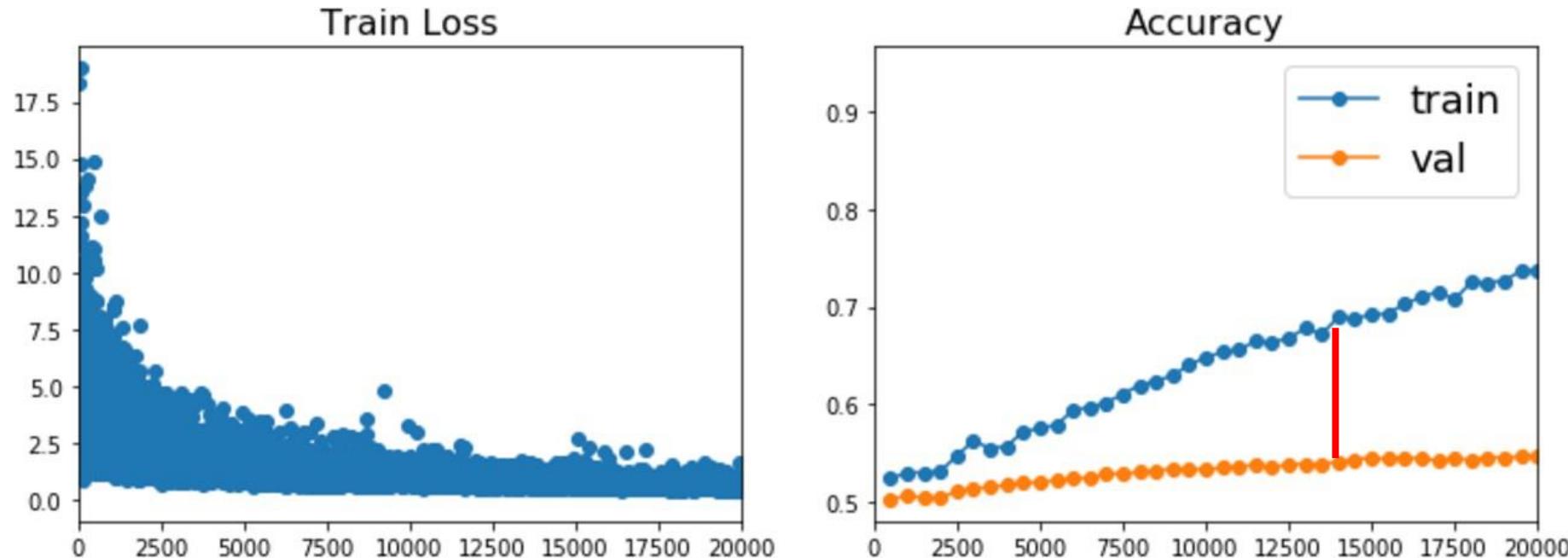
Model Ensembles

1. Train multiple independent models
2. At test time average their results

(Take average of predicted probability distributions, then choose argmax)

Enjoy 2% extra performance

How to improve single-model performance?



Regularization

Regularization: Add term to loss

$$L = \frac{1}{N} \sum_{i=1}^N \sum_{j \neq y_i} \max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1) + \boxed{\lambda R(W)}$$

In common use:

L2 regularization

$$R(W) = \sum_k \sum_l W_{k,l}^2 \quad (\text{Weight decay})$$

L1 regularization

$$R(W) = \sum_k \sum_l |W_{k,l}|$$

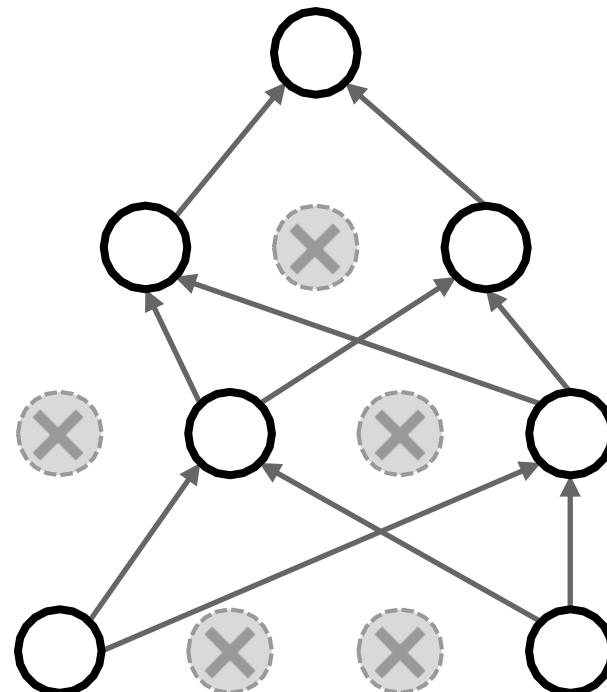
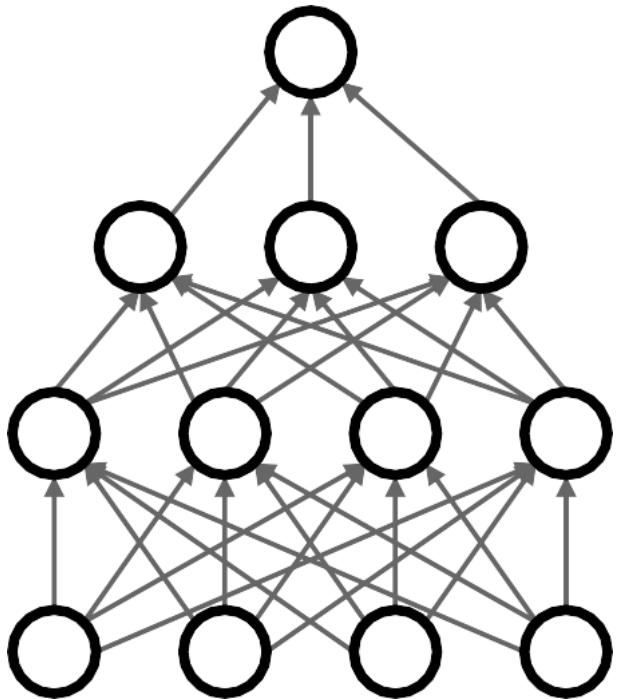
Elastic net (L1 + L2)

$$R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$$

Regularization: Dropout

In each forward pass, randomly set some neurons to zero

Probability of dropping is a hyperparameter; 0.5 is common



Srivastava et al, "Dropout: A simple way to prevent neural networks from overfitting", JMLR 2014

Regularization: Dropout

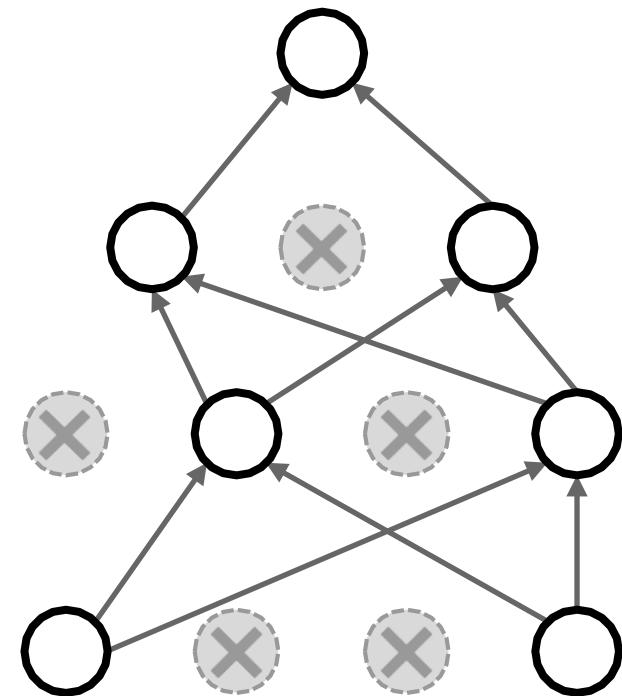
```
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
    """ X contains the data """

    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = np.random.rand(*H1.shape) < p # first dropout mask
    H1 *= U1 # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = np.random.rand(*H2.shape) < p # second dropout mask
    H2 *= U2 # drop!
    out = np.dot(W3, H2) + b3

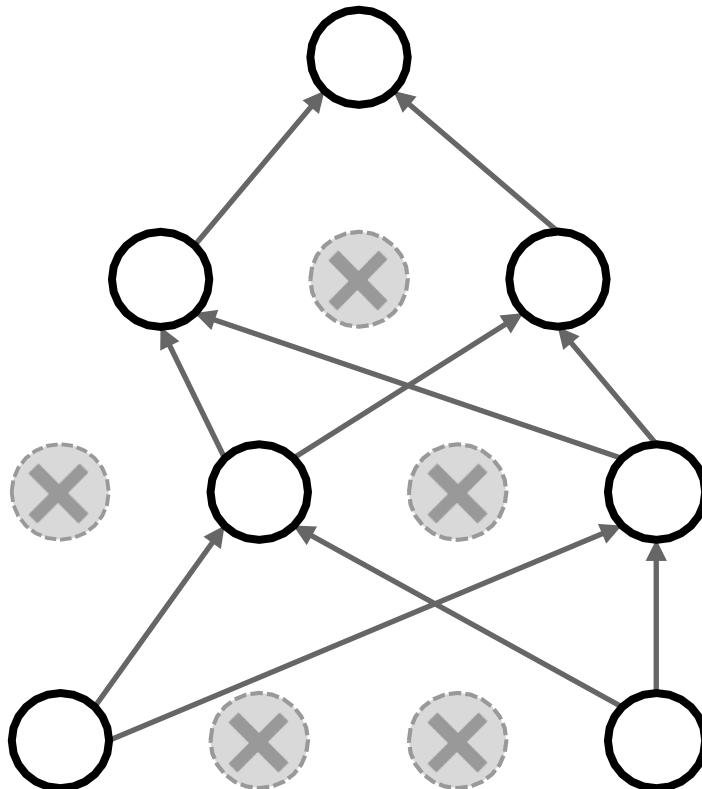
    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)
```

Example forward pass with a 3-layer network using dropout



Regularization: Dropout

How can this possibly be a good idea?

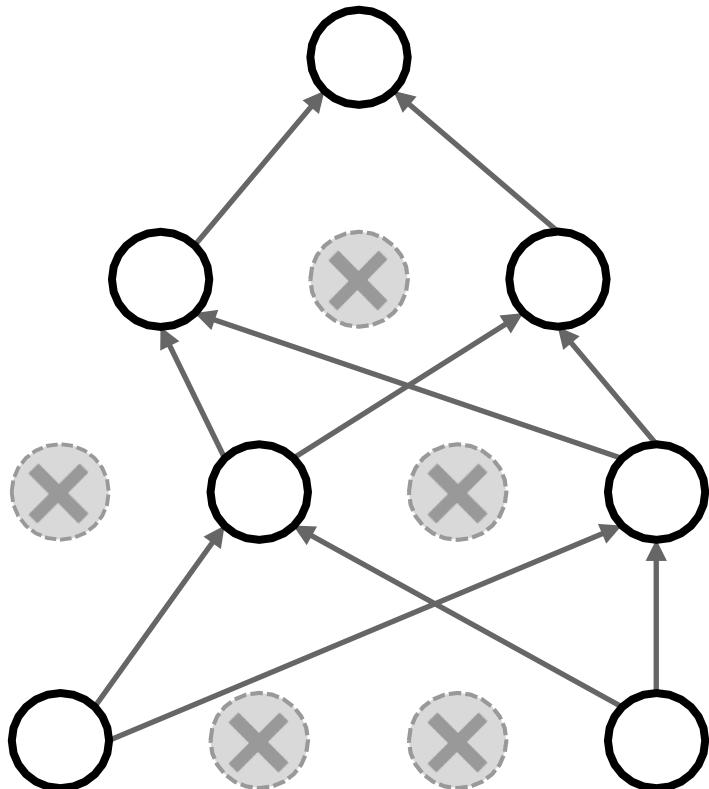


Forces the network to have a redundant representation;
Prevents co-adaptation of features



Regularization: Dropout

How can this possibly be a good idea?



Another interpretation:

Dropout is training a large **ensemble** of models (that share parameters).

Each binary mask is one model

An FC layer with 4096 units has $2^{4096} \sim 10^{1233}$ possible masks!
Only $\sim 10^{82}$ atoms in the universe...

Dropout: Test time

Dropout makes our output random!

$$\boxed{y} = f_W(\boxed{x}, \boxed{z})$$

Output
(label) Input
(image)

Random
mask

Want to “average out” the randomness at test-time

$$y = f(x) = E_z [f(x, z)] = \int p(z) f(x, z) dz$$

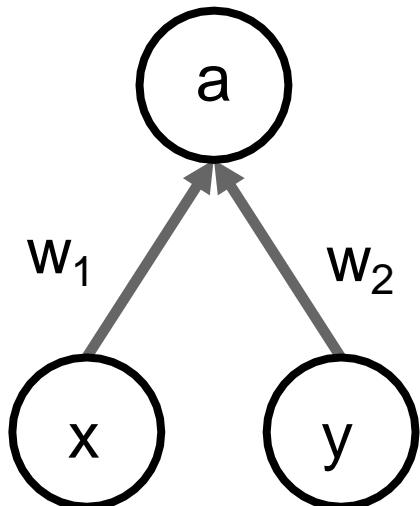
But this integral seems hard ...

Dropout: Test time

Want to approximate
the integral

$$y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)dz$$

Consider a single neuron.

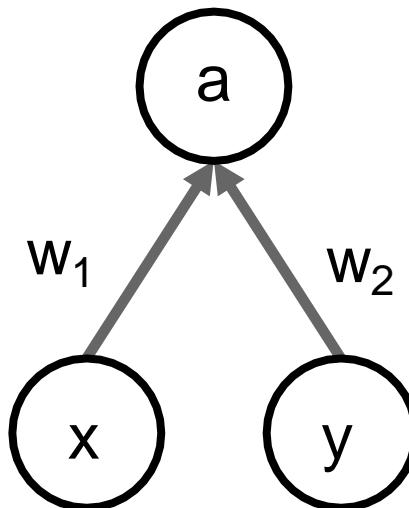


Dropout: Test time

Want to approximate
the integral

$$y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)dz$$

Consider a single neuron.



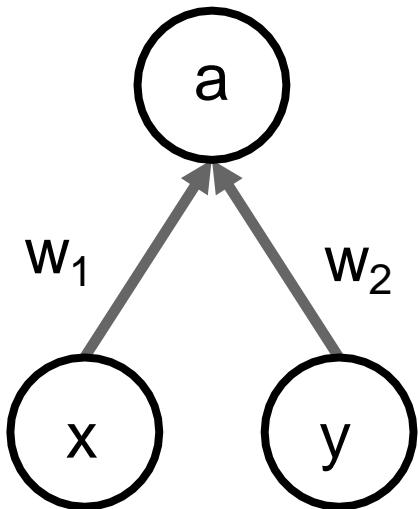
At test time we have: $E[a] = w_1x + w_2y$

Dropout: Test time

Want to approximate
the integral

$$y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)dz$$

Consider a single neuron.



At test time we have: $E[a] = w_1x + w_2y$

During training we have:

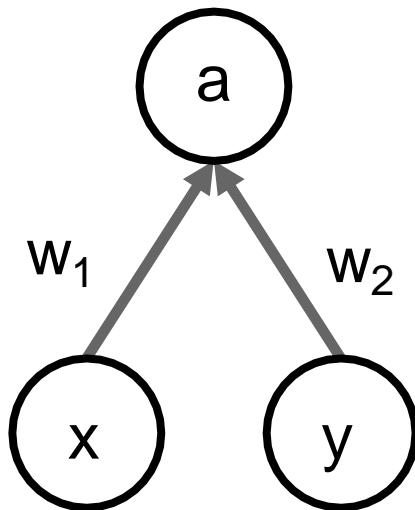
$$\begin{aligned} E[a] &= \frac{1}{4}(w_1x + w_2y) + \frac{1}{4}(w_1x + 0y) \\ &\quad + \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2y) \\ &= \frac{1}{2}(w_1x + w_2y) \end{aligned}$$

Dropout: Test time

Want to approximate
the integral

$$y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)dz$$

Consider a single neuron.



At test time we have: $E[a] = w_1x + w_2y$

During training we have:

$$\begin{aligned} E[a] &= \frac{1}{4}(w_1x + w_2y) + \frac{1}{4}(w_1x + 0y) \\ &\quad + \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2y) \\ &= \frac{1}{2}(w_1x + w_2y) \end{aligned}$$

At test time, multiply
by dropout probability

Dropout: Test time

```
def predict(X):
    # ensembled forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
    H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
    out = np.dot(W3, H2) + b3
```

At test time all neurons are active always

=> We must scale the activations so that for each neuron:

output at test time = expected output at training time

Dropout Summary

```
""" Vanilla Dropout: Not recommended implementation (see notes below) """
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
    """ X contains the data """

    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = np.random.rand(*H1.shape) < p # first dropout mask
    H1 *= U1 # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = np.random.rand(*H2.shape) < p # second dropout mask
    H2 *= U2 # drop!
    out = np.dot(W3, H2) + b3

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)

def predict(X):
    # ensembled forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
    H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
    out = np.dot(W3, H2) + b3
```

drop in train time

scale at test time

More common: “Inverted dropout”

```
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
    H1 *= U1 # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
    H2 *= U2 # drop!
    out = np.dot(W3, H2) + b3

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)

def predict(X):
    # ensembled forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    out = np.dot(W3, H2) + b3
```

test time is unchanged!



Regularization: A common pattern

Training: Add some kind of randomness

$$y = f_W(x, z)$$

Testing: Average out randomness (sometimes approximate)

$$y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)dz$$

Regularization: A common pattern

Training: Add some kind of randomness

$$y = f_W(x, z)$$

Testing: Average out randomness (sometimes approximate)

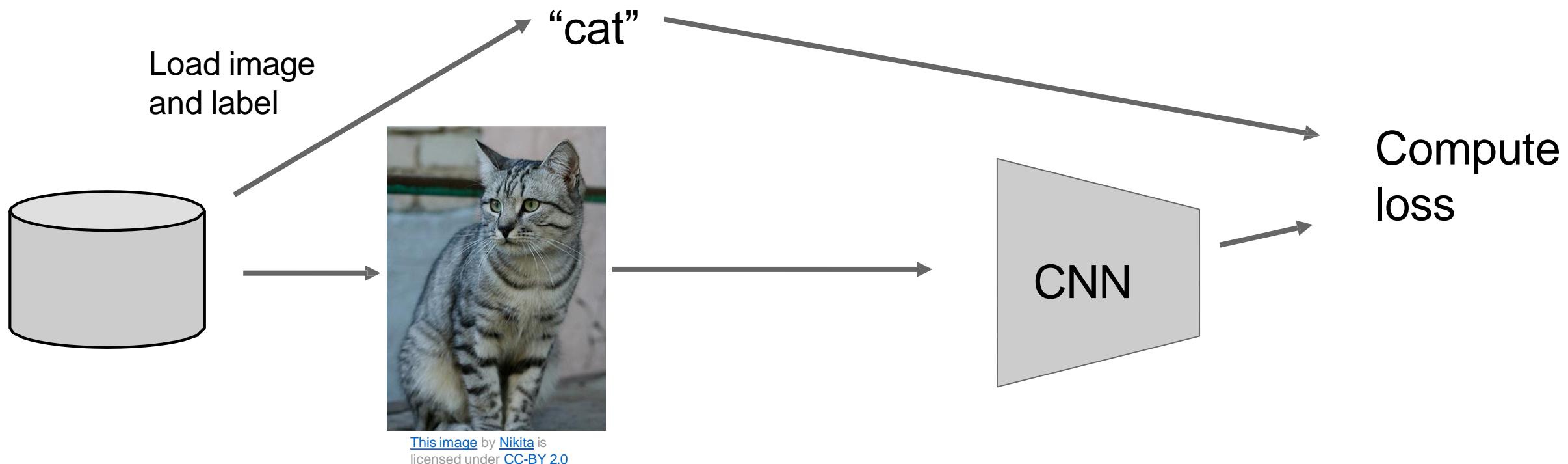
$$y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)dz$$

Example: Batch Normalization

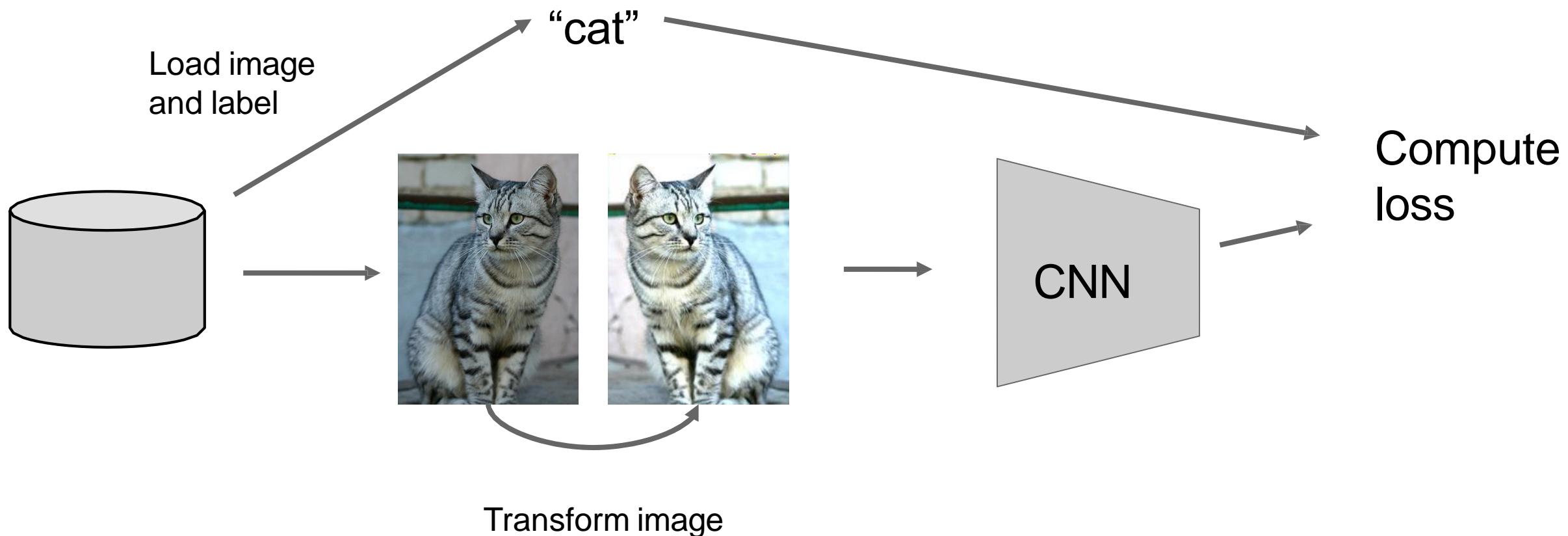
Training: Normalize using stats from random minibatches

Testing: Use fixed stats to normalize

Regularization: Data Augmentation



Regularization: Data Augmentation



Data Augmentation

Horizontal Flips



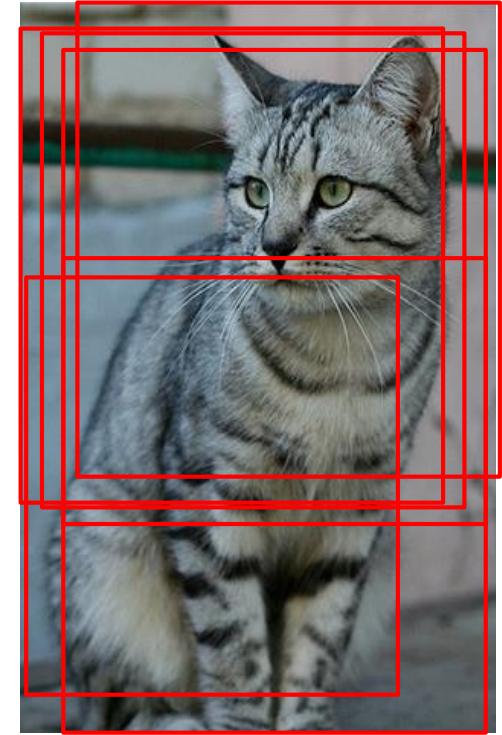
Data Augmentation

Random crops and scales

Training: sample random crops / scales

ResNet:

1. Pick random L in range [256, 480]
2. Resize training image, short side = L
3. Sample random 224×224 patch



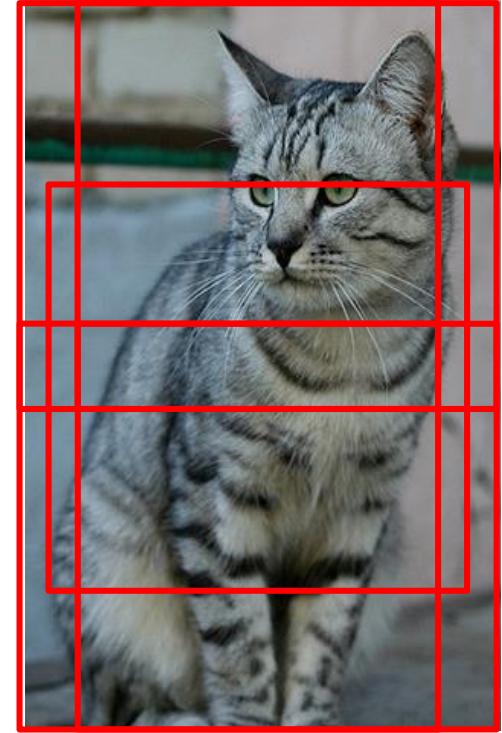
Data Augmentation

Random crops and scales

Training: sample random crops / scales

ResNet:

1. Pick random L in range [256, 480]
2. Resize training image, short side = L
3. Sample random 224×224 patch



Testing: average a fixed set of crops

ResNet:

1. Resize image at 5 scales: {224, 256, 384, 480, 640}
2. For each size, use 10 224×224 crops: 4 corners + center, + flips

Data Augmentation

Color Jitter

Simple: Randomize
contrast and brightness



Data Augmentation

Get creative for your problem!

Examples of data augmentations:

- translation
- rotation
- stretching
- shearing,
- lens distortions, ... (go crazy)

Automatic Data Augmentation

	Original	Sub-policy 1	Sub-policy 2	Sub-policy 3	Sub-policy 4	Sub-policy 5
Batch 1						
Batch 2						
Batch 3						
ShearX, 0.9, 7 Invert, 0.2, 3		ShearY, 0.7, 6 Solarize, 0.4, 8		ShearX, 0.9, 4 AutoContrast, 0.8, 3		Invert, 0.9, 3 Equalize, 0.6, 3
						ShearY, 0.8, 5 AutoContrast, 0.7, 3

Cubuk et al., "AutoAugment: Learning Augmentation Strategies from Data", CVPR 2019

Regularization: Cutout

Training: Set random image regions to zero

Testing: Use full image

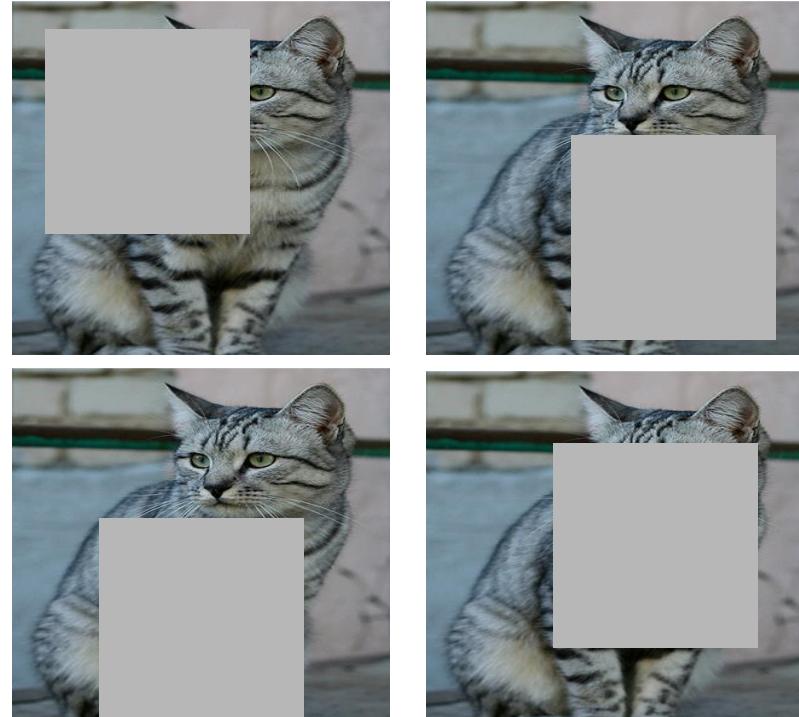
Examples:

Dropout

Batch Normalization

Data Augmentation

Cutout / Random Crop



Works very well for small datasets like CIFAR,
less common for large datasets like ImageNet

DeVries and Taylor, "Improved Regularization of
Convolutional Neural Networks with Cutout", arXiv 2017

Regularization - In practice

Training: Add random noise

Testing: Marginalize over the noise

Examples:

Dropout

Batch Normalization

Data Augmentation

Cutout / Random Crop

- Consider dropout for large fully-connected layers
- Batch normalization and data augmentation almost always a good idea
- Try cutout especially for small classification datasets

Choosing Hyperparameters

(without tons of GPUs)

Choosing Hyperparameters

Step 1: Check initial loss

Turn off weight decay, sanity check loss at initialization
e.g. $\log(C)$ for softmax with C classes

Random guessing $\rightarrow 1/C$ probability for each class
Softmax Loss $\rightarrow -\log(1/C) = \log(C)$

Choosing Hyperparameters

Step 1: Check initial loss

Step 2: Overfit a small sample

Try to train to 100% training accuracy on a small sample of training data (~5-10 minibatches); fiddle with architecture, learning rate, weight initialization

Loss not going down? LR too low, bad initialization

Loss explodes to Inf or NaN? LR too high, bad initialization

Choosing Hyperparameters

Step 1: Check initial loss

Step 2: Overfit a small sample

Step 3: Find LR that makes loss go down

Use the architecture from the previous step, use all training data, turn on small weight decay, find a learning rate that makes the loss drop significantly within ~100 iterations

Good learning rates to try: 1e-1, 1e-2, 1e-3, 1e-4

Choosing Hyperparameters

Step 1: Check initial loss

Step 2: Overfit a small sample

Step 3: Find LR that makes loss go down

Step 4: Coarse grid, train for ~1-5 epochs

Choose a few values of learning rate and weight decay around what worked from Step 3, train a few models for ~1-5 epochs.

Good weight decay to try: 1e-4, 1e-5, 0

Choosing Hyperparameters

Step 1: Check initial loss

Step 2: Overfit a small sample

Step 3: Find LR that makes loss go down

Step 4: Coarse grid, train for ~1-5 epochs

Step 5: Refine grid, train longer

Pick best models from Step 4, train them for longer (~10-20 epochs) with constant learning rate

Choosing Hyperparameters

Step 1: Check initial loss

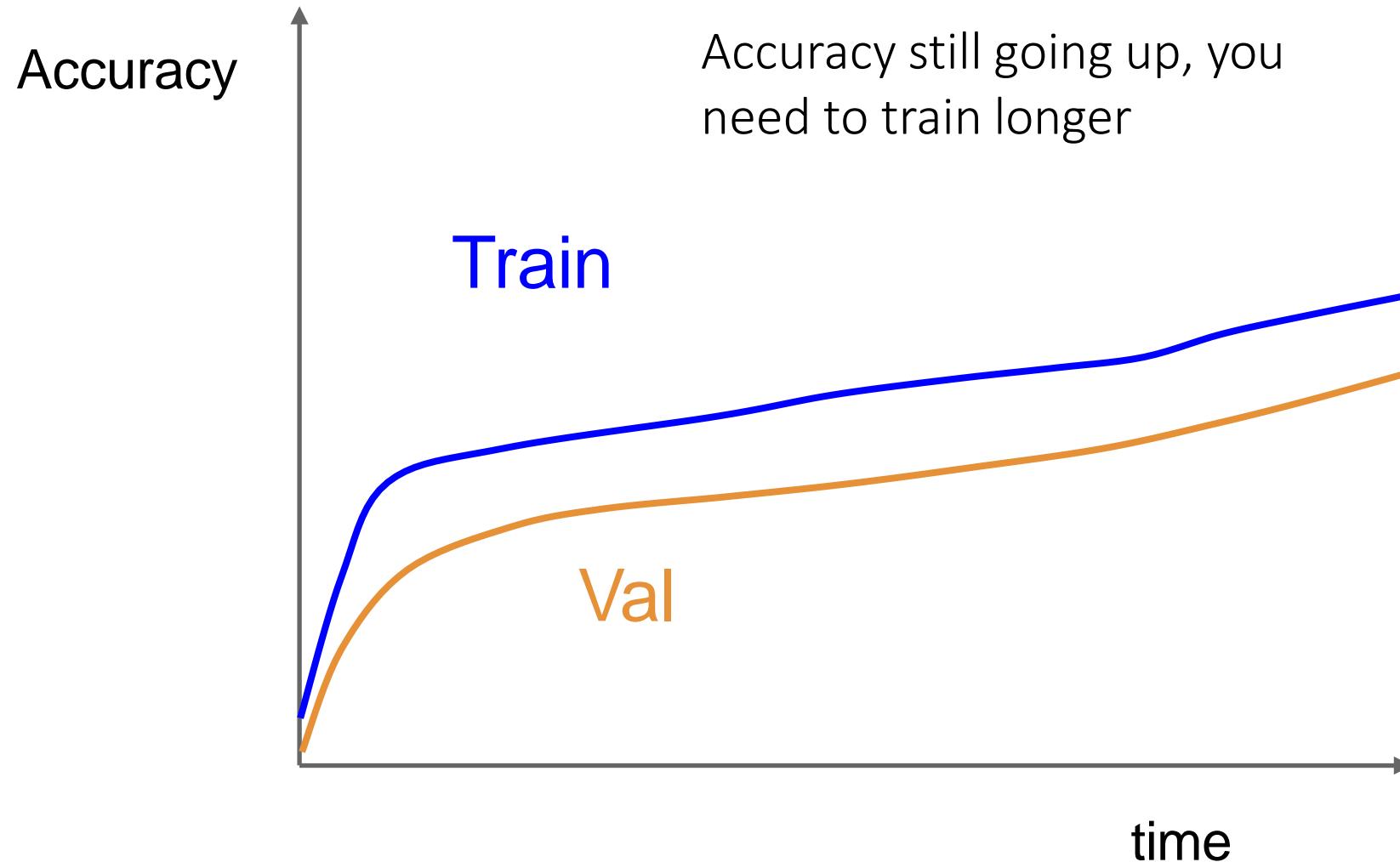
Step 2: Overfit a small sample

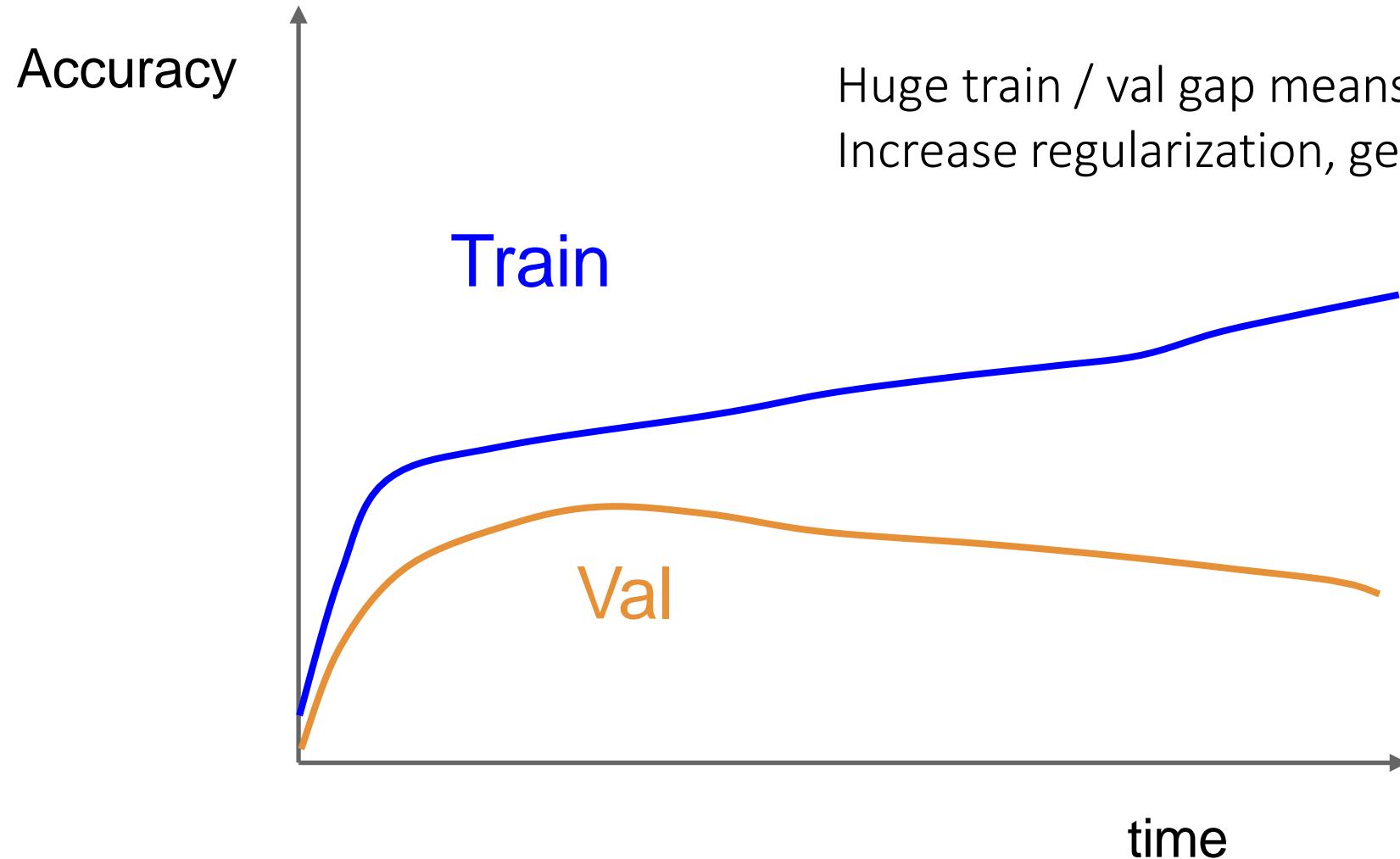
Step 3: Find LR that makes loss go down

Step 4: Coarse grid, train for ~1-5 epochs

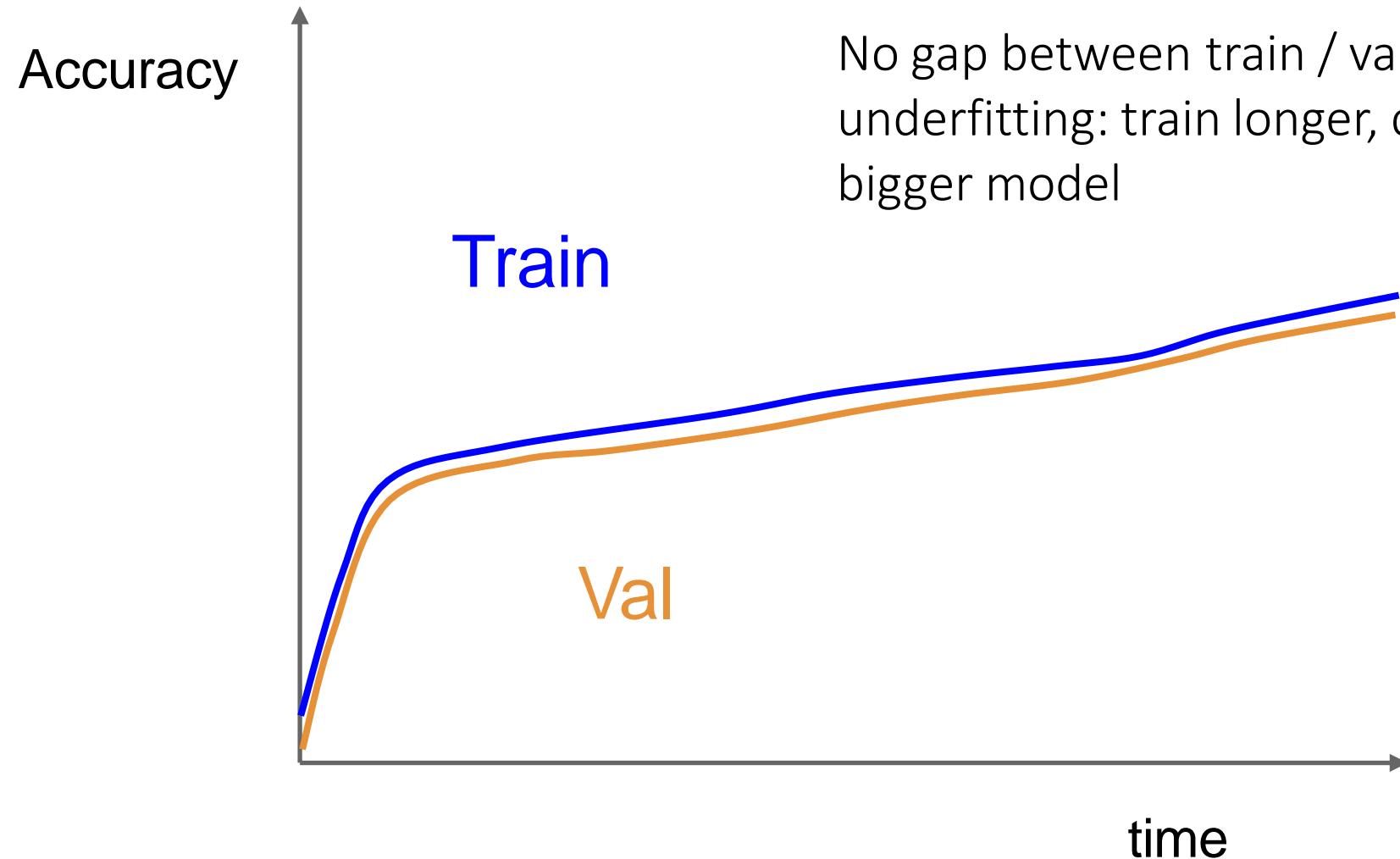
Step 5: Refine grid, train longer

Step 6: Look at loss and accuracy curves





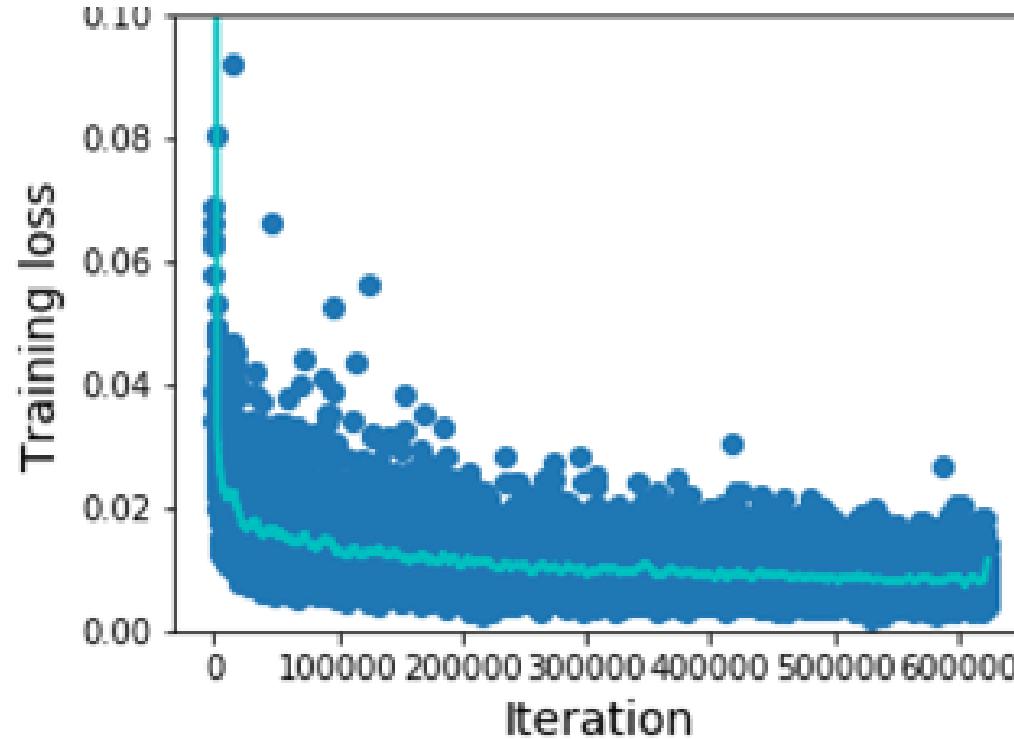
Huge train / val gap means overfitting!
Increase regularization, get more data



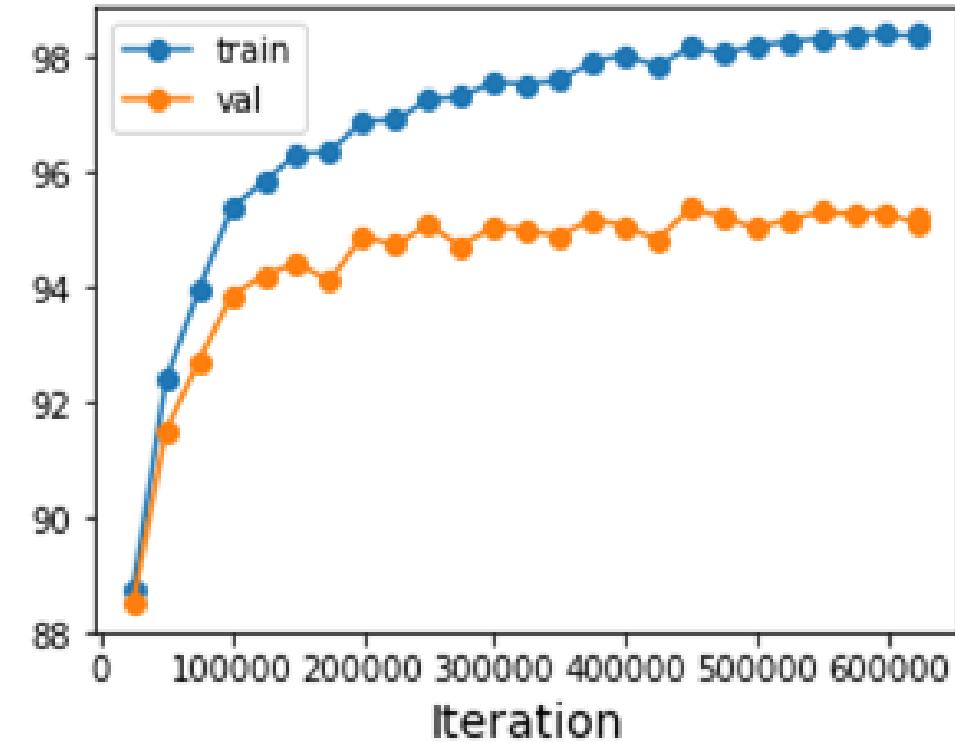
No gap between train / val means underfitting: train longer, can use a bigger model

Look at learning curves!

Training Loss



Train / Val Accuracy

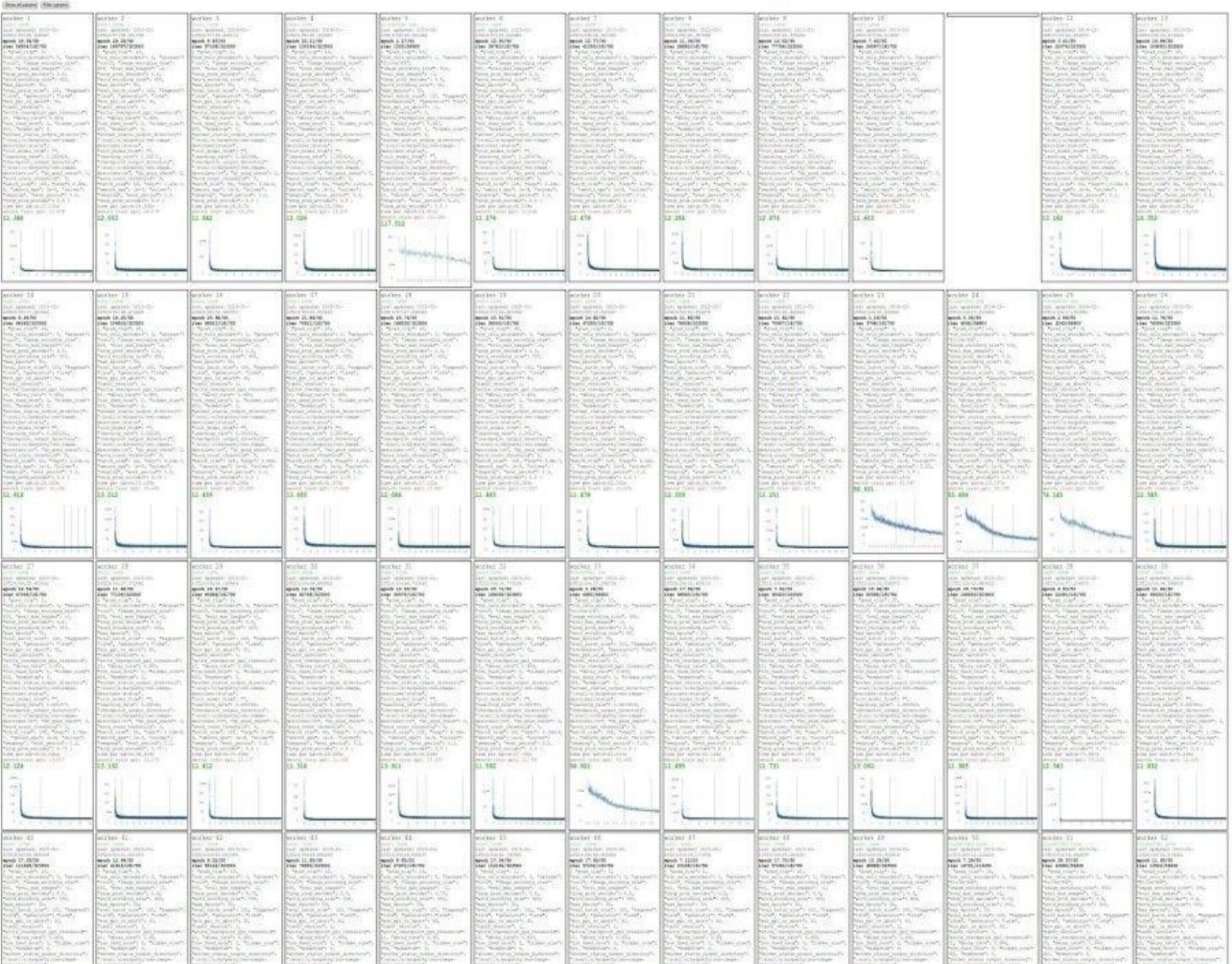


Losses may be noisy, use a scatter plot and also plot moving average to see trends better

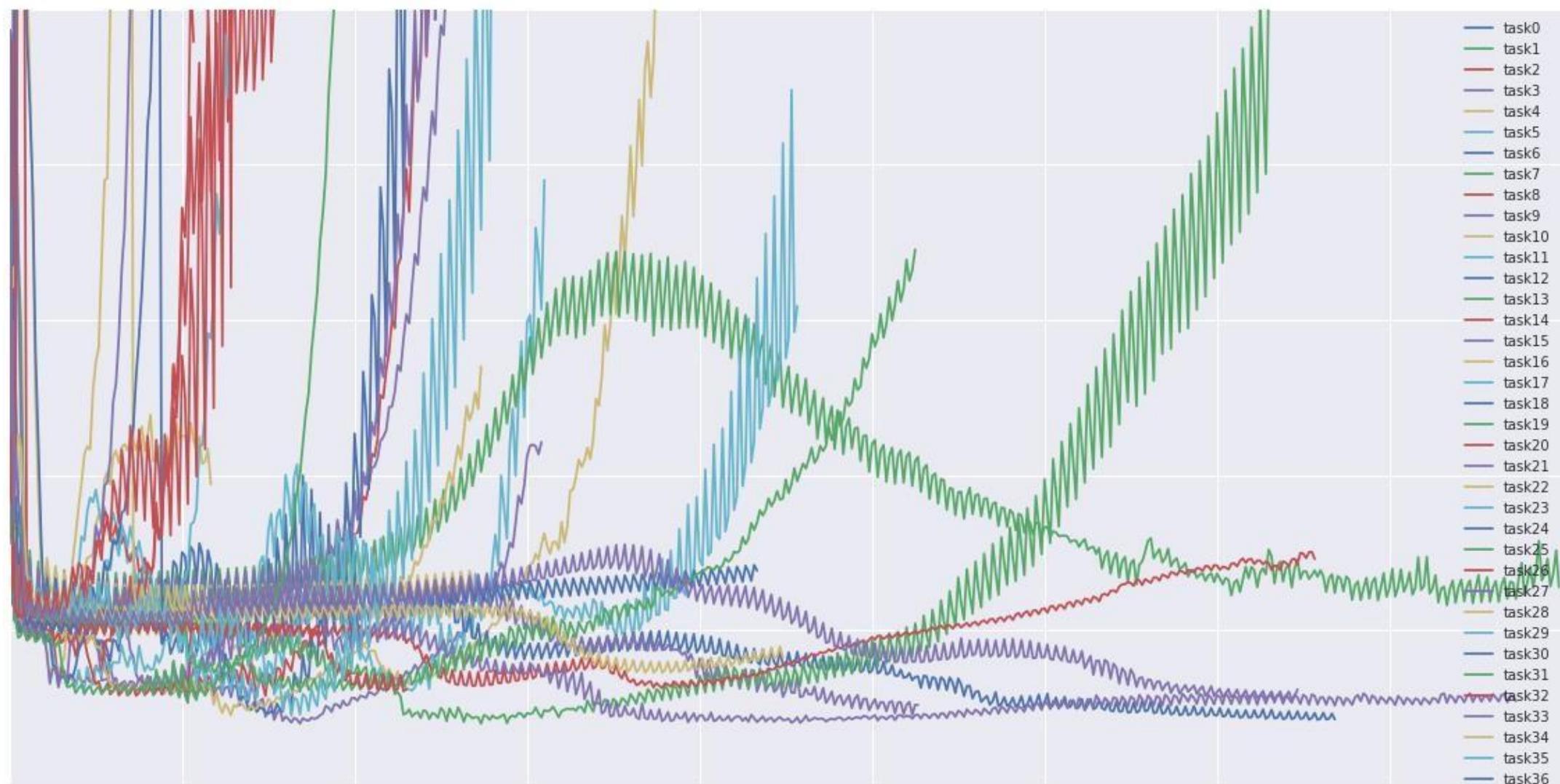
Cross-validation

We develop "command centers" to visualize all our models training with different hyperparameters

check out [weights and biases](#)



You can plot all your loss curves for different hyperparameters on a single plot





Choosing Hyperparameters

Step 1: Check initial loss

Step 2: Overfit a small sample

Step 3: Find LR that makes loss go down

Step 4: Coarse grid, train for ~1-5 epochs

Step 5: Refine grid, train longer

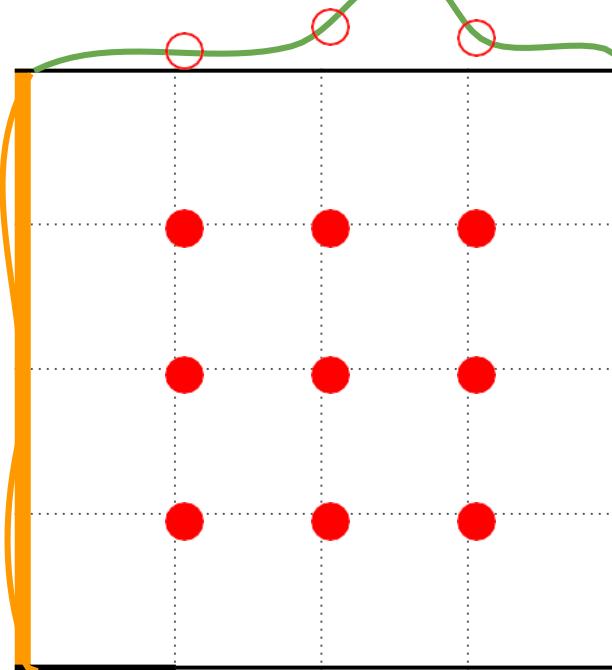
Step 6: Look at loss and accuracy curves

Step 7: GOTO step 5

Random Search vs. Grid Search

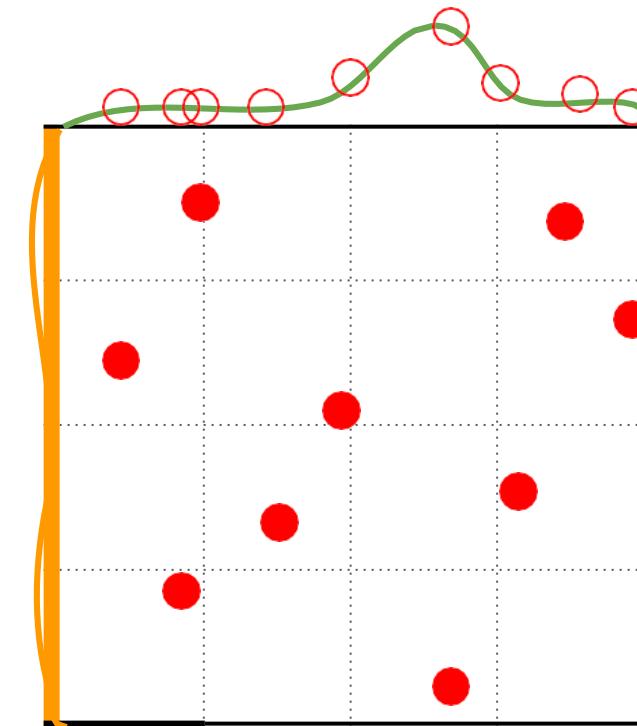
Random Search for Hyper-Parameter Optimization
Bergstra and Bengio, 2012

Grid Layout



Important Parameter

Random Layout



Important Parameter

Unimportant Parameter

Illustration of Bergstra et al., 2012 by Shayne
Longpre, copyright CS231n 2017

We looked in detail at:

TLDRLS

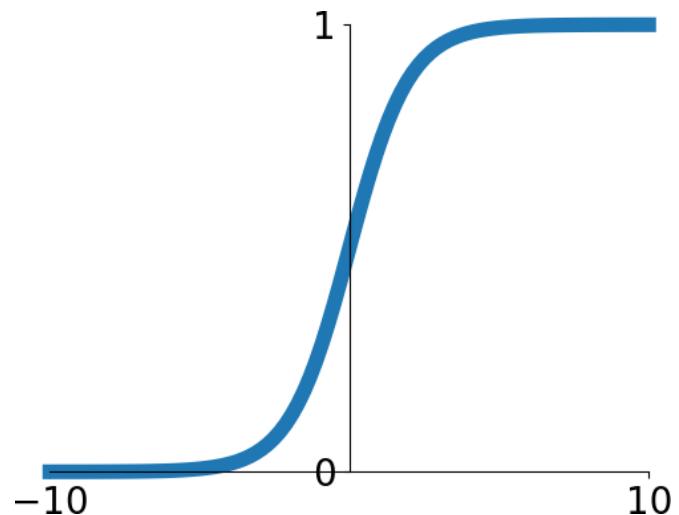
- Activation Functions (use ReLU)
- Data Preprocessing (images: subtract mean)
- Weight Initialization (use Xavier/Kaiming init)
- Batch Normalization (use this!)
- Transfer learning (use this if you can!)

In Lecture: Recap of Content + QA

Appendix – Slides from Previous Years of the Course

Activation Functions

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

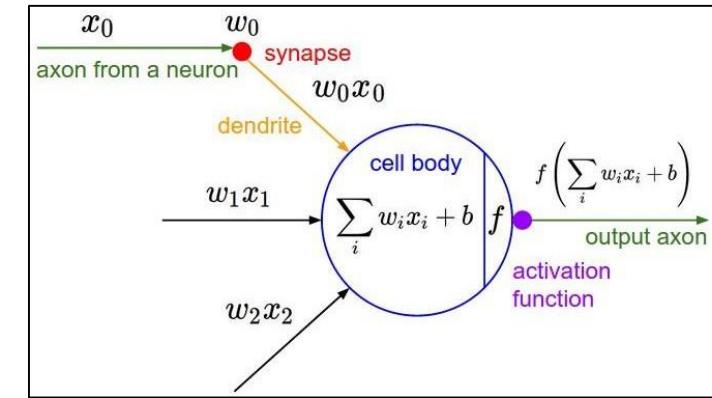


Sigmoid

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron
- 3 problems:
 1. Saturated neurons “kill” the gradients
 2. Sigmoid outputs are not zero-centered

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

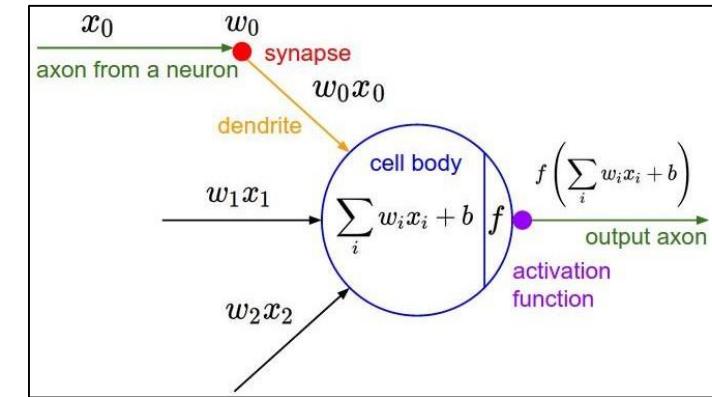
$$f \left(\sum_i w_i x_i + b \right)$$



What can we say about the gradients on w ?

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

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

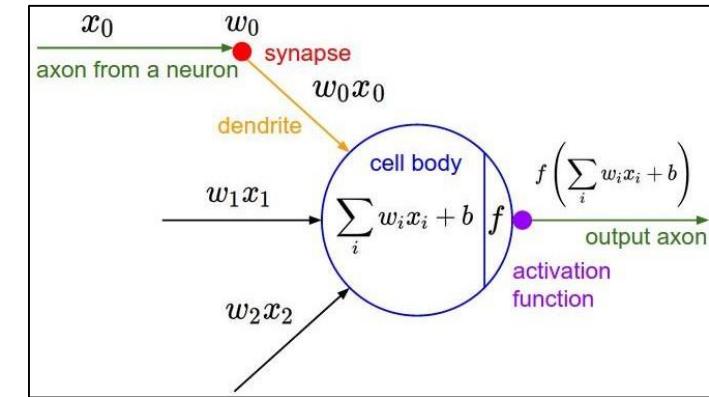


What can we say about the gradients on w ?

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

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

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



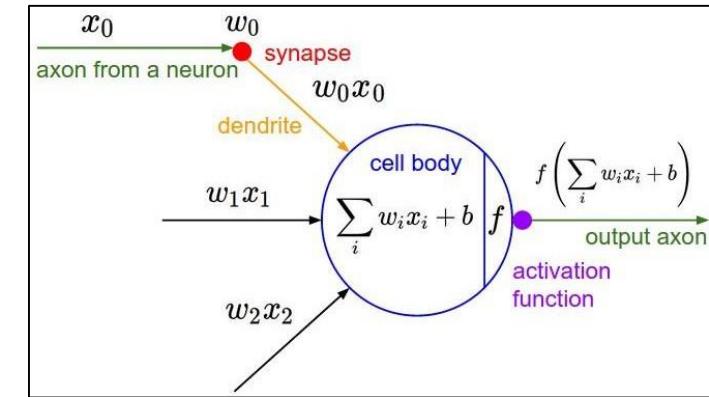
What can we say about the gradients on w ?

We know that 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...

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



What can we say about the gradients on w ?

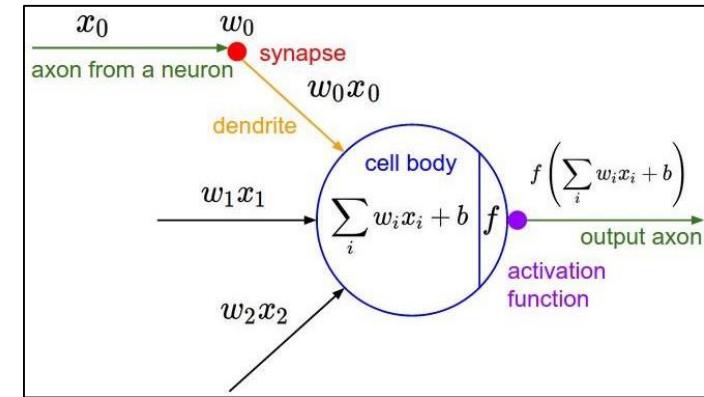
We know that local gradient of sigmoid is always positive

We are assuming x 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...

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



What can we say about the gradients on w ?

We know that local gradient of sigmoid is always positive

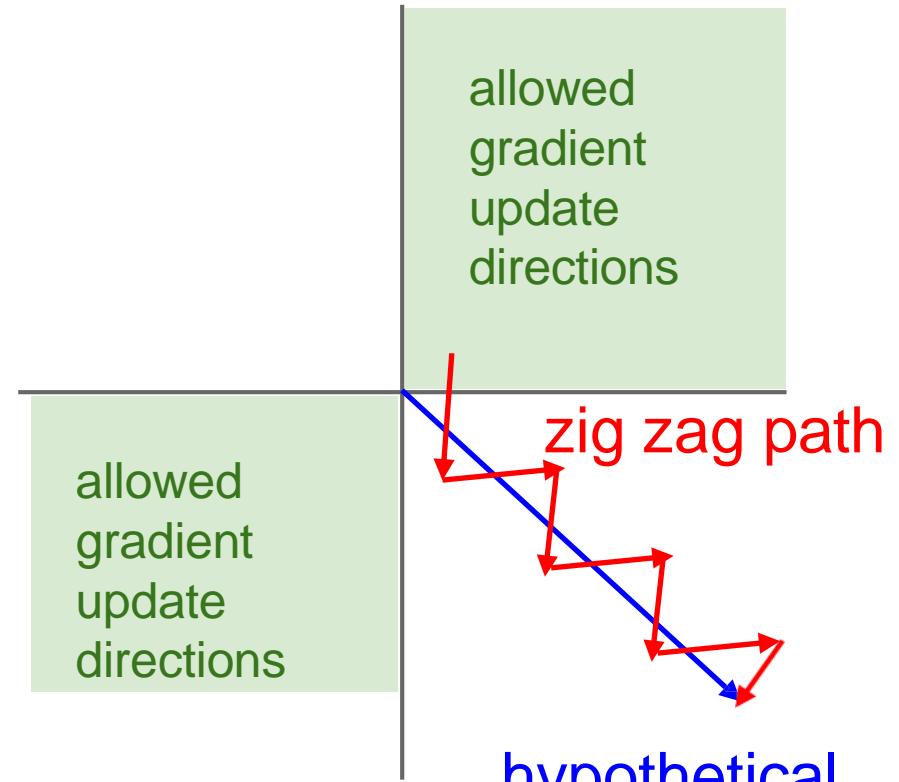
We are assuming x is always positive

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

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

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

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

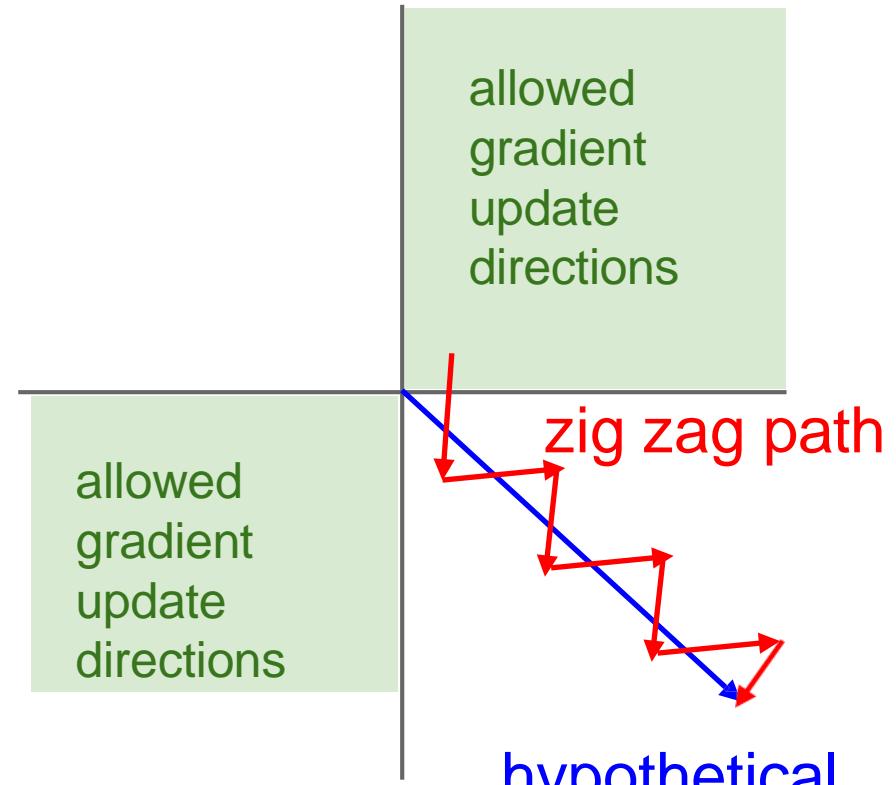


What can we say about the gradients on w ?

Always all positive or all negative :(

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

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



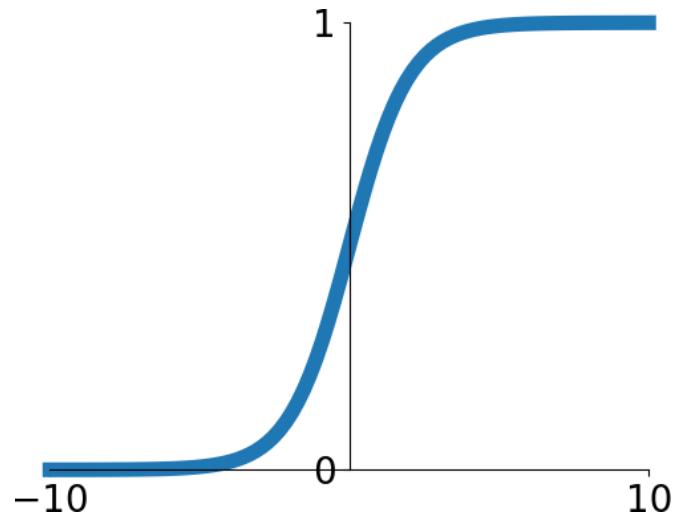
What can we say about the gradients on w ?

Always all positive or all negative :(

(For a single element! Minibatches help)

Activation Functions

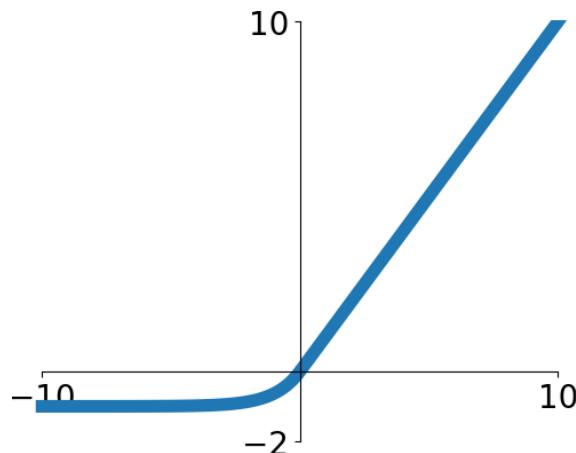
$$\sigma(x) = 1/(1 + e^{-x})$$



Sigmoid

- Squashes numbers to range [0,1]
 - Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron
-
- 3 problems:
 1. Saturated neurons “kill” the gradients
 2. Sigmoid outputs are not zero-centered
 3. $\exp()$ is a bit compute expensive

Exponential Linear Units (ELU)



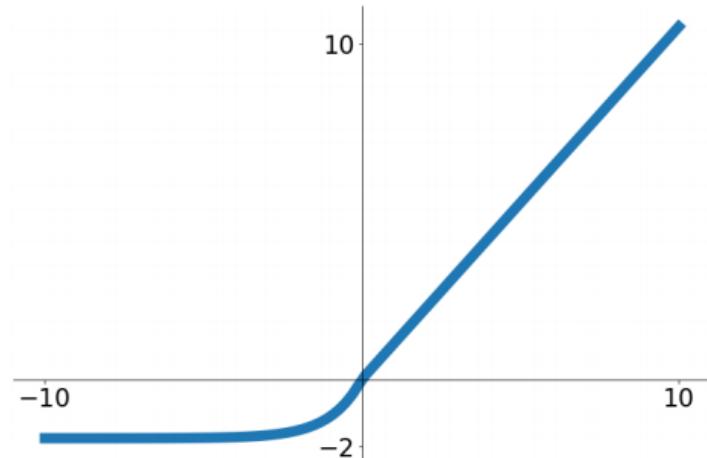
$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha (\exp(x) - 1) & \text{if } x \leq 0 \end{cases}$$

(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

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

$$\alpha = 1.6732632423543772848170429916717$$

$$\lambda = 1.0507009873554804934193349852946$$

Maxout “Neuron”

[Goodfellow et al., 2013]

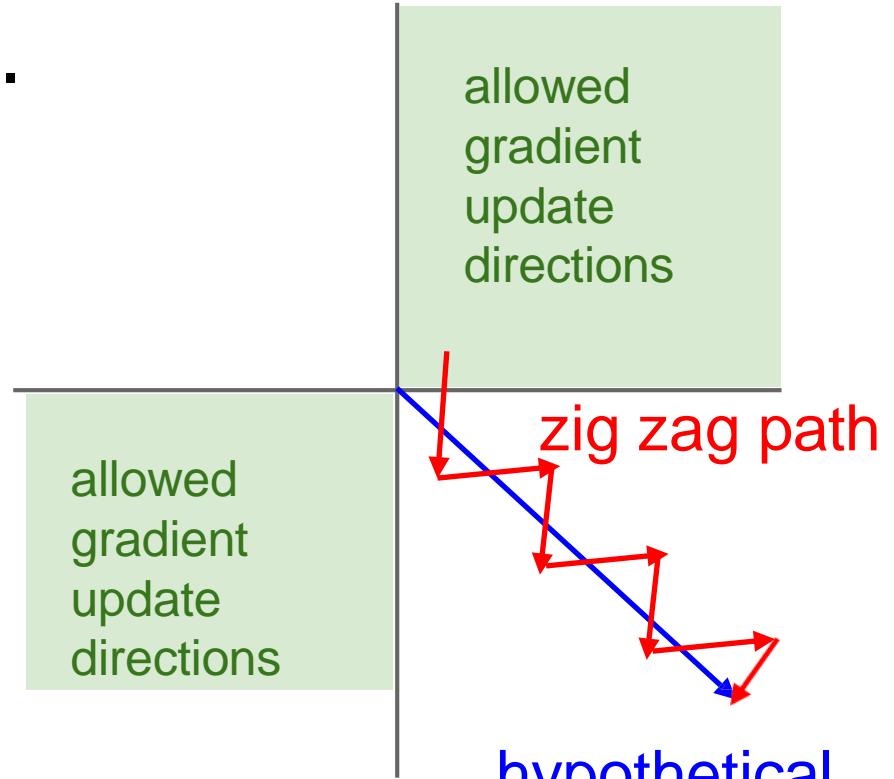
- Does not have the basic form of dot product -> nonlinearity
- Generalizes ReLU and Leaky ReLU
- Linear Regime! Does not saturate! Does not die!

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

Problem: doubles the number of parameters/neuron :(

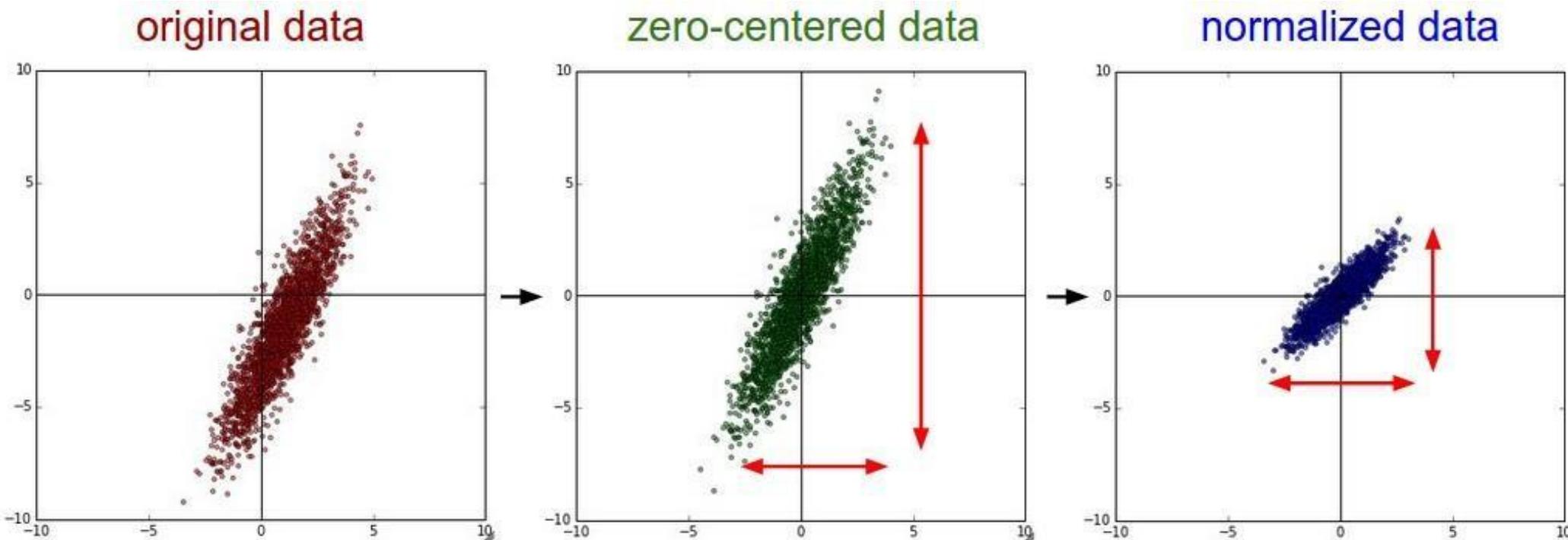
Remember: Consider what happens when
the input to a neuron is always positive...

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



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

Data Preprocessing



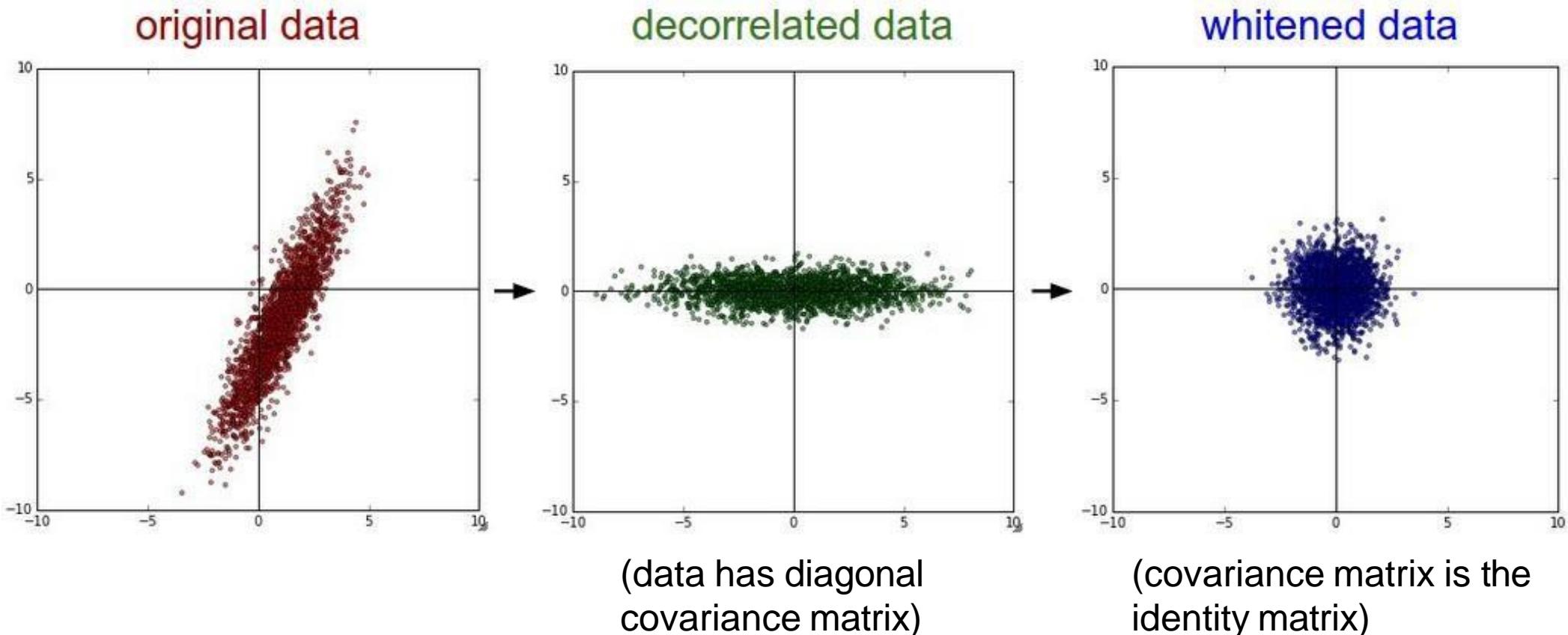
```
X -= np.mean(X, axis = 0)
```

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

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

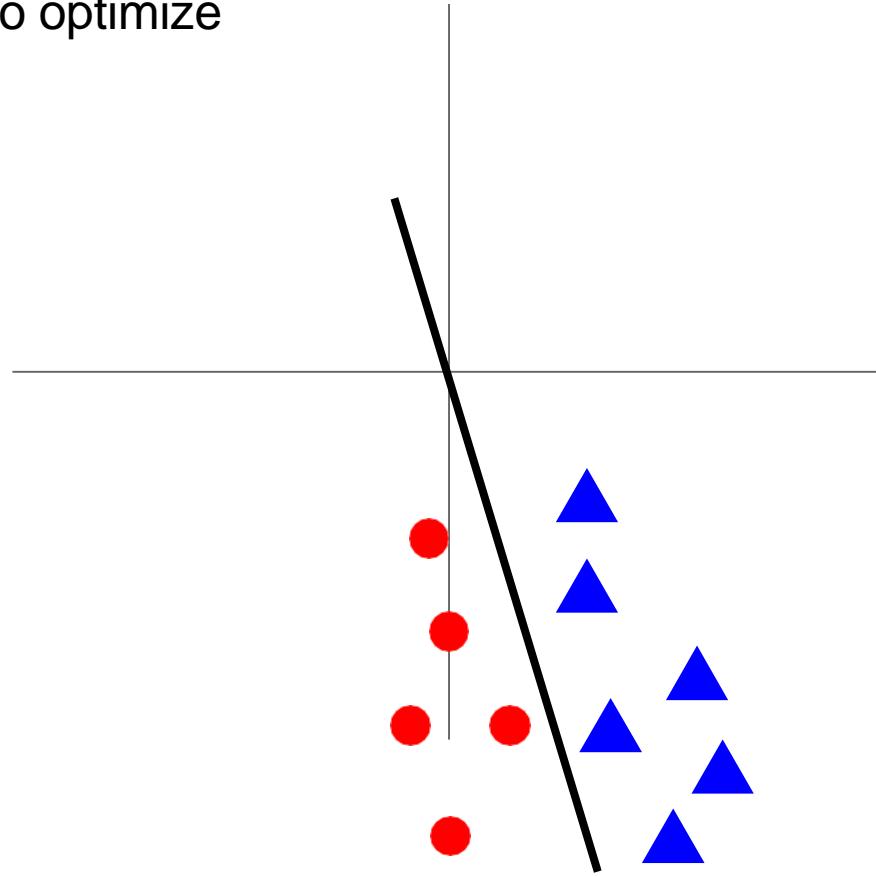
Data Preprocessing

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

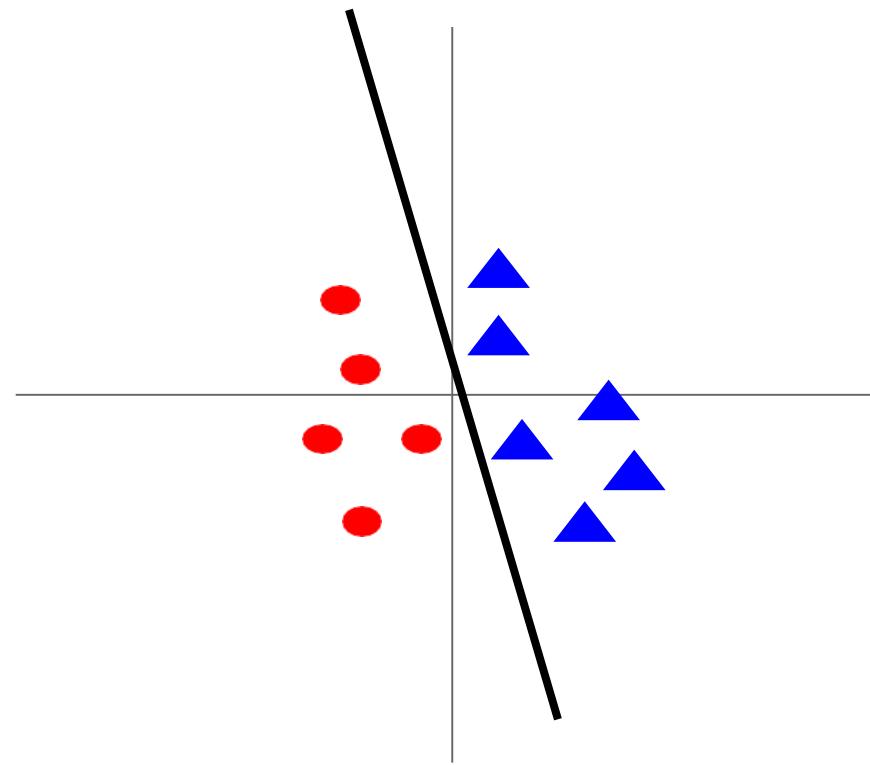


Data Preprocessing

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



Xavier Initialization: Proof of Optimality

```
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)
```

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

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

Let: $y = x_1w_1 + x_2w_2 + \dots + x_{Din}w_{Din}$

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)
```

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

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

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

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

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)
```

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

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

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

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

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

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)
```

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

For conv layers, Din is filter_size² * input_channels

Let: $y = x_1w_1 + x_2w_2 + \dots + x_{Din}w_{Din}$

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

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

$$\text{Var}(y) = \text{Var}(x_1w_1 + x_2w_2 + \dots + x_{Din}w_{Din})$$

[substituting value of y]

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)
```

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

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

Let: $y = x_1w_1 + x_2w_2 + \dots + x_{Din}w_{Din}$

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

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

$$\begin{aligned}\text{Var}(y) &= \text{Var}(x_1w_1 + x_2w_2 + \dots + x_{Din}w_{Din}) \\ &= \text{Din Var}(x_i w_i)\end{aligned}$$

[Assume all x_i, w_i are iid]

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)
```

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

For conv layers, Din is filter_size² * input_channels

Let: $y = x_1w_1 + x_2w_2 + \dots + x_{Din}w_{Din}$

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

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

$$\begin{aligned}\text{Var}(y) &= \text{Var}(x_1w_1 + x_2w_2 + \dots + x_{Din}w_{Din}) \\ &= \text{Din} \text{Var}(x_i w_i) \\ &= \text{Din} \text{Var}(x_i) \text{Var}(w_i)\end{aligned}$$

[Assume all x_i, w_i are zero mean]

Weight Initialization: “Xavier” Initialization

```
dims = [4096] * 7          "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)
```

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

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

Let: $y = x_1w_1 + x_2w_2 + \dots + x_{Din}w_{Din}$

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

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

$$\begin{aligned}\text{Var}(y) &= \text{Var}(x_1w_1 + x_2w_2 + \dots + x_{Din}w_{Din}) \\ &= \text{Din} \text{Var}(x_i w_i) \\ &= \text{Din} \text{Var}(x_i) \text{Var}(w_i)\end{aligned}$$

[Assume all x_i, w_i are iid]

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

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

Data Augmentation

Color Jitter

Simple: Randomize
contrast and brightness



More Complex:

1. Apply PCA to all [R, G, B] pixels in training set
2. Sample a “color offset” along principal component directions
3. Add offset to all pixels of a training image

(As seen in *[Krizhevsky et al. 2012]*, ResNet, etc)

Regularization: A common pattern

Training: Add random noise

Testing: Marginalize over the noise

Examples:

Dropout

Batch Normalization

Data Augmentation

Regularization: DropConnect

Training: Drop connections between neurons (set weights to 0)

Testing: Use all the connections

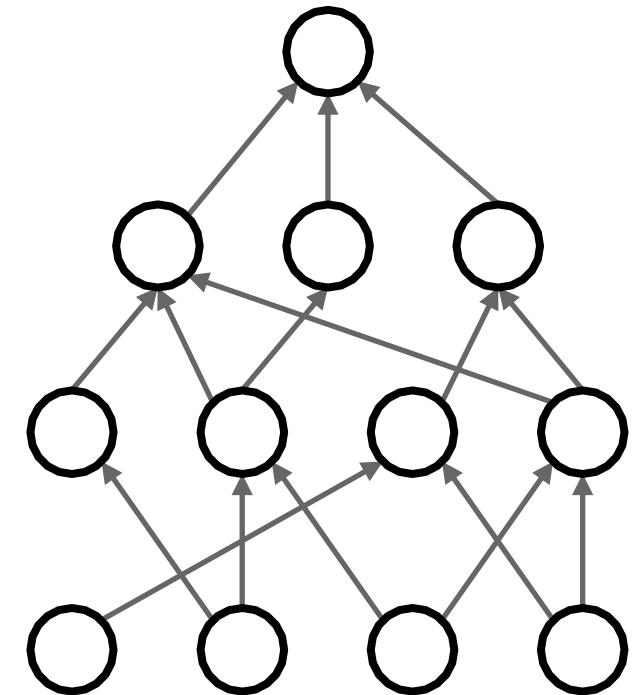
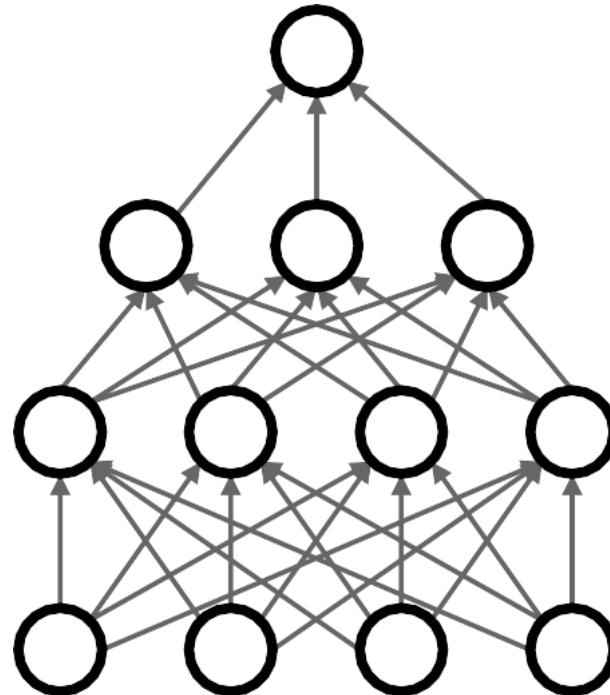
Examples:

Dropout

Batch Normalization

Data Augmentation

DropConnect



Wan et al, "Regularization of Neural Networks using DropConnect", ICML 2013

Regularization: Fractional Pooling

Training: Use randomized pooling regions

Testing: Average predictions from several regions

Examples:

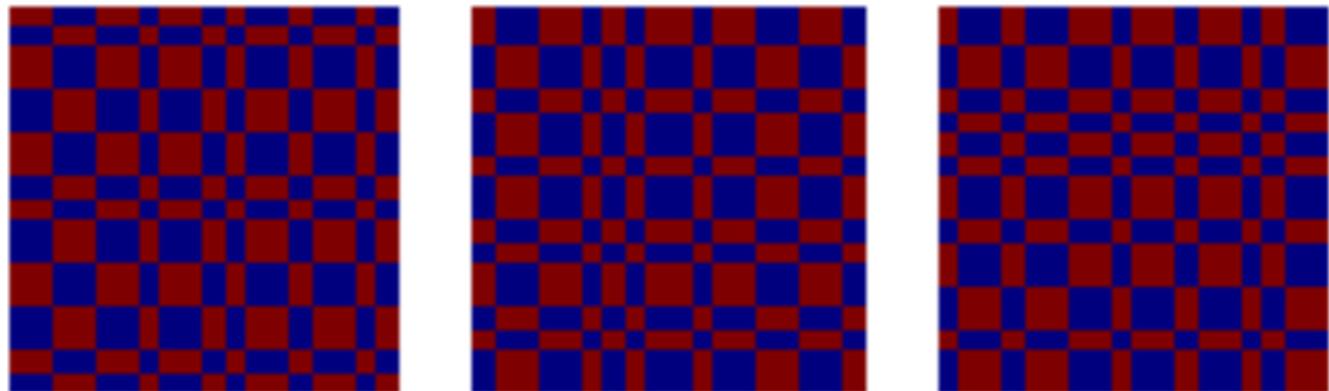
Dropout

Batch Normalization

Data Augmentation

DropConnect

Fractional Max Pooling



Graham, "Fractional Max Pooling", arXiv 2014

Regularization: Stochastic Depth

Training: Skip some layers in the network

Testing: Use all the layer

Examples:

Dropout

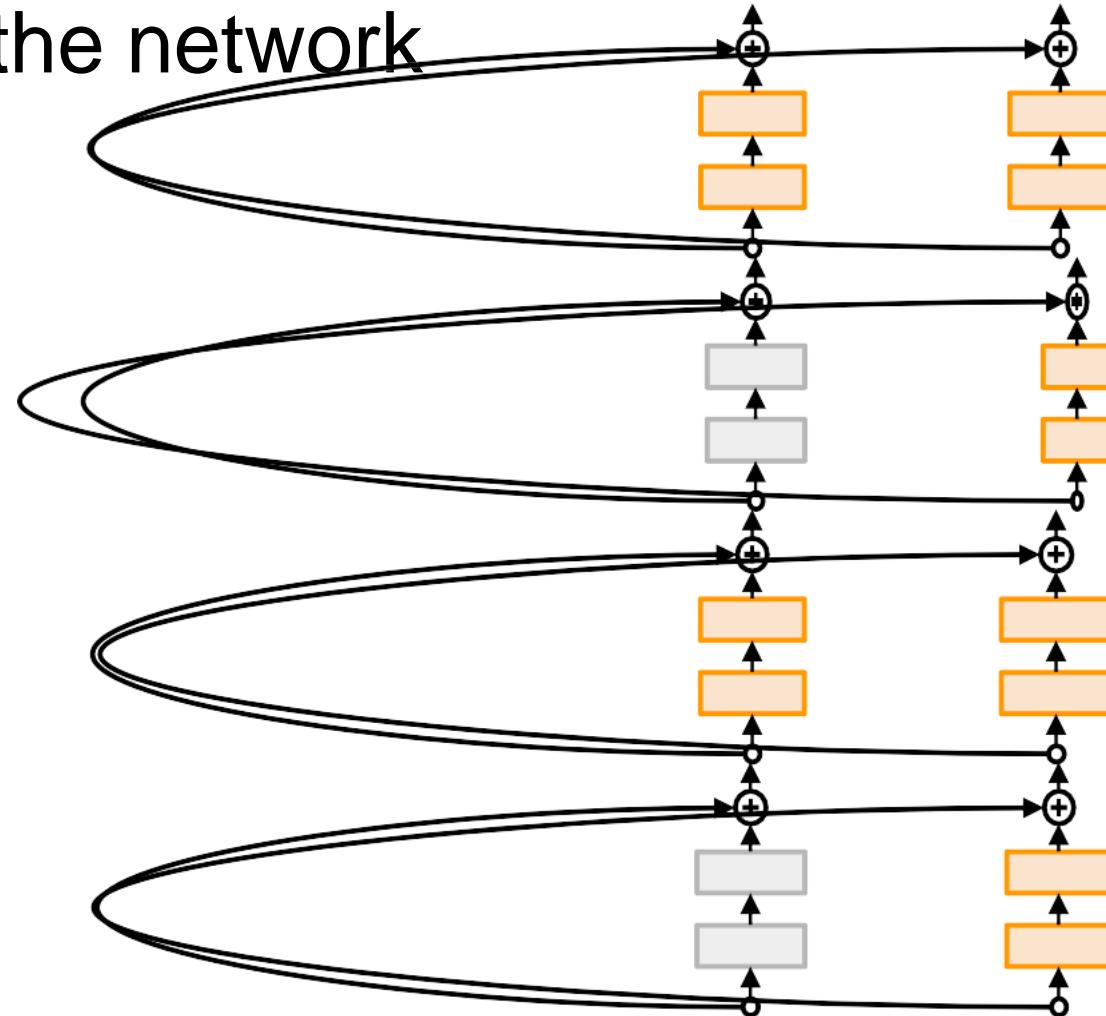
Batch Normalization

Data Augmentation

DropConnect

Fractional Max Pooling

Stochastic Depth



Huang et al, "Deep Networks with Stochastic Depth", ECCV 2016

Regularization: Mixup

Training: Train on random blends of images

Testing: Use original images

Examples:

Dropout

Batch Normalization

Data Augmentation

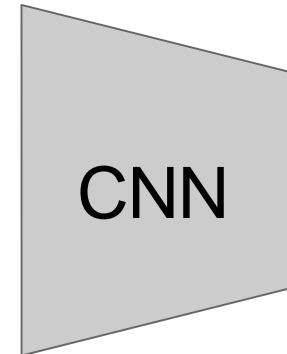
DropConnect

Fractional Max Pooling

Stochastic Depth

Cutout / Random Crop

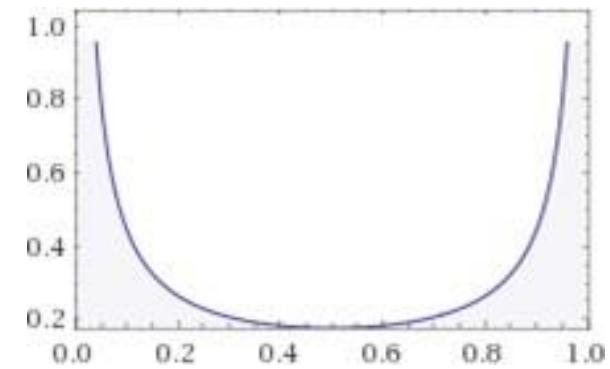
Mixup



Target label:
cat: 0.4
dog: 0.6

Randomly blend the pixels
of pairs of training images,
e.g. 40% cat, 60% dog

Zhang et al, "mixup: Beyond Empirical Risk Minimization", ICLR 2018

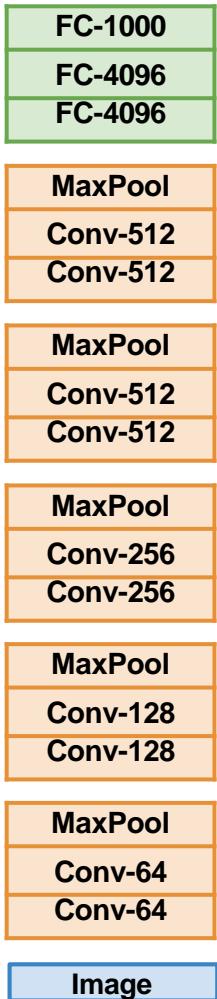


Transfer learning

You need a lot of data if you want to
train/use CNNs?

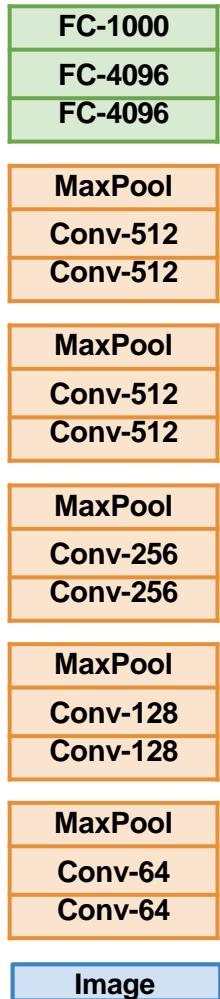
Transfer Learning with CNNs

1. Train on Imagenet

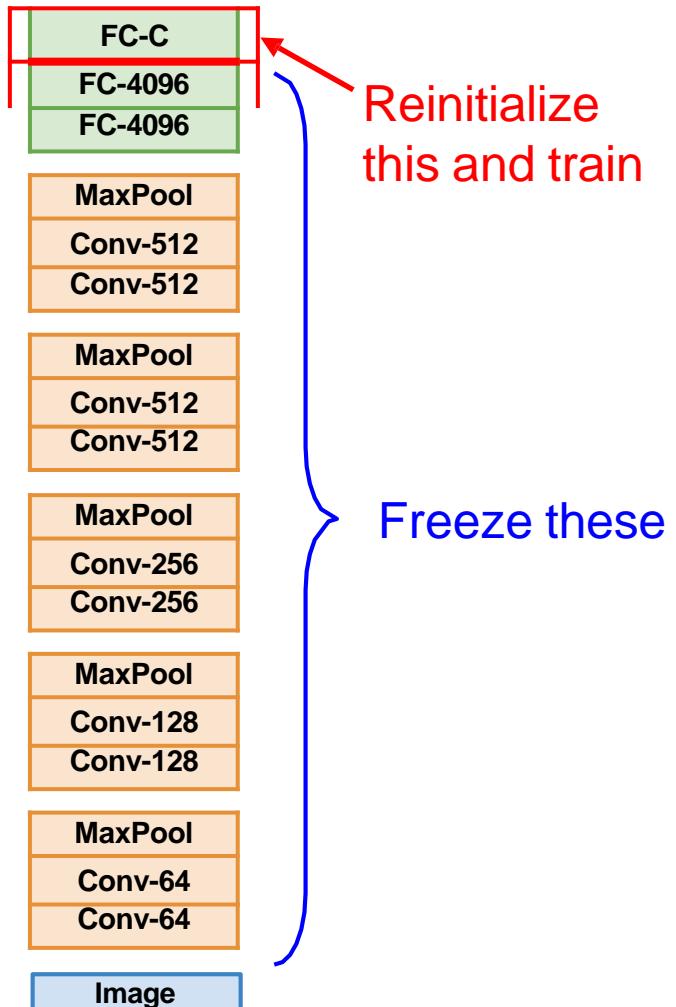


Transfer Learning with CNNs

1. Train on Imagenet

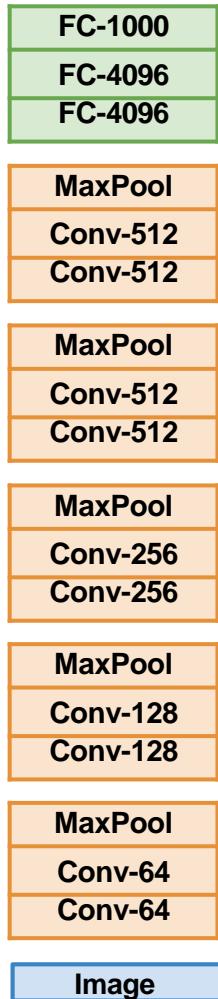


2. Small Dataset (C classes)

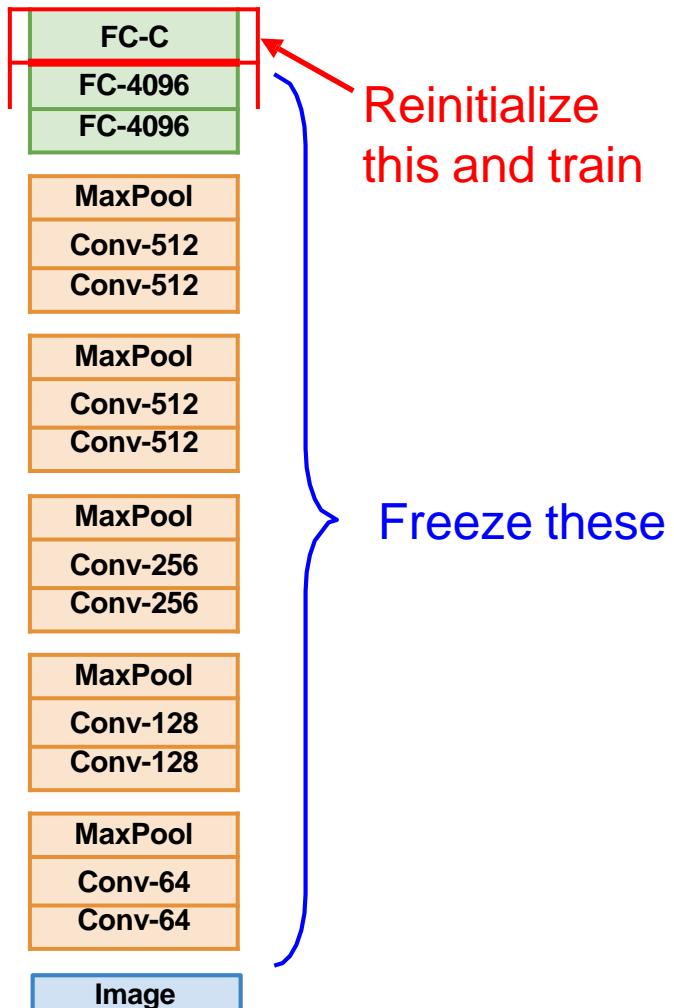


Transfer Learning with CNNs

1. Train on Imagenet

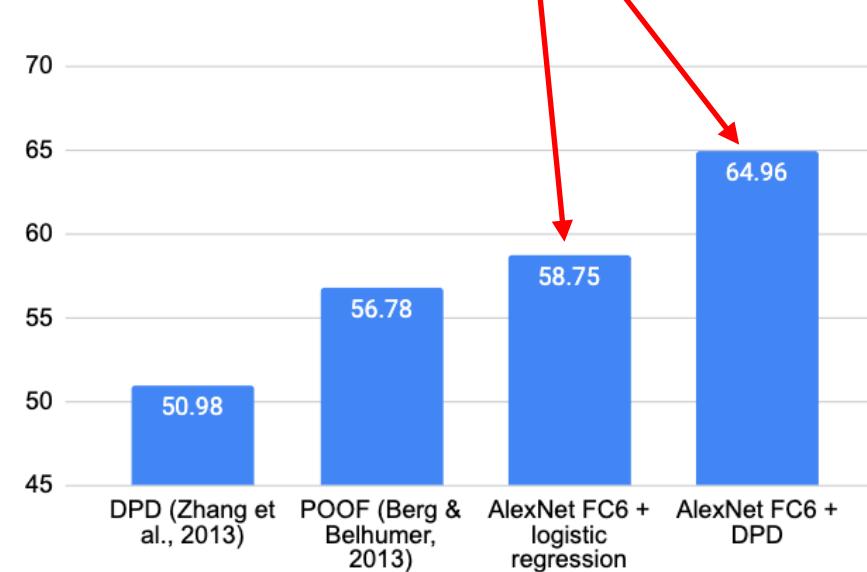


2. Small Dataset (C classes)



Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014
Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014

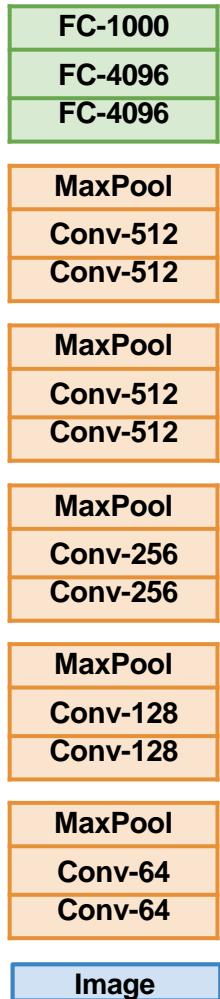
Finetuned from AlexNet



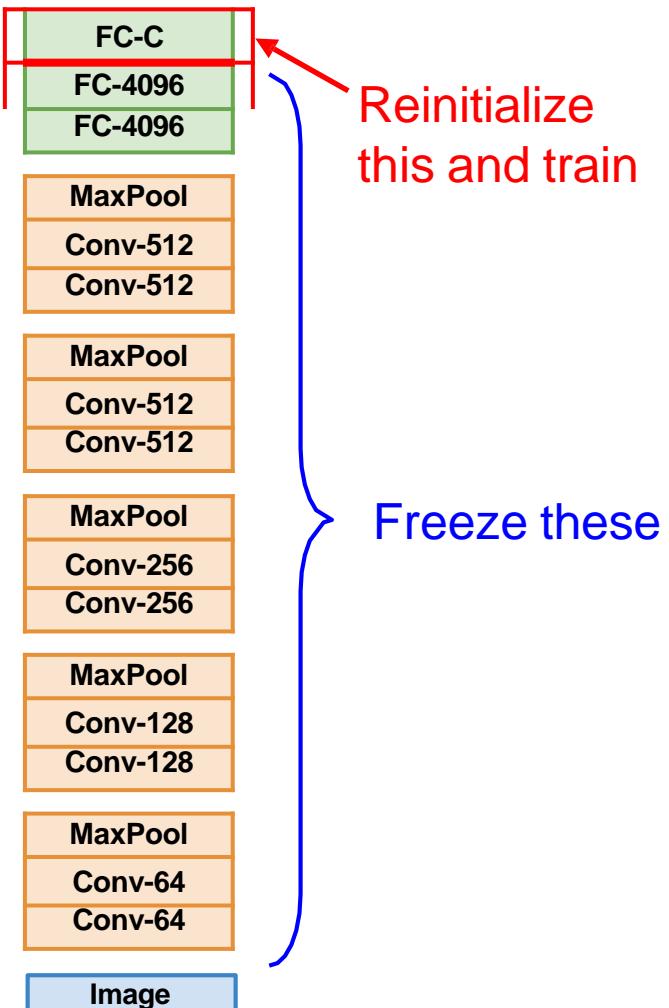
Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014

Transfer Learning with CNNs

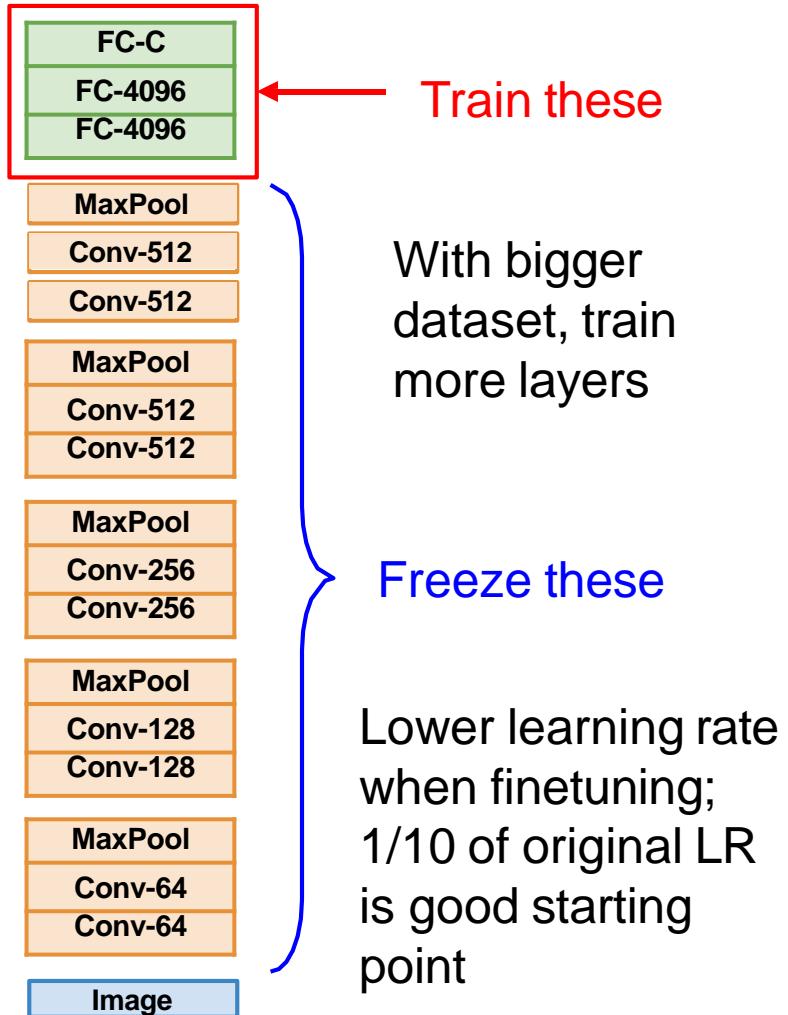
1. Train on Imagenet



2. Small Dataset (C classes)



3. Bigger dataset



Takeaway for your projects and beyond:

Have some dataset of interest but it has < ~1M images?

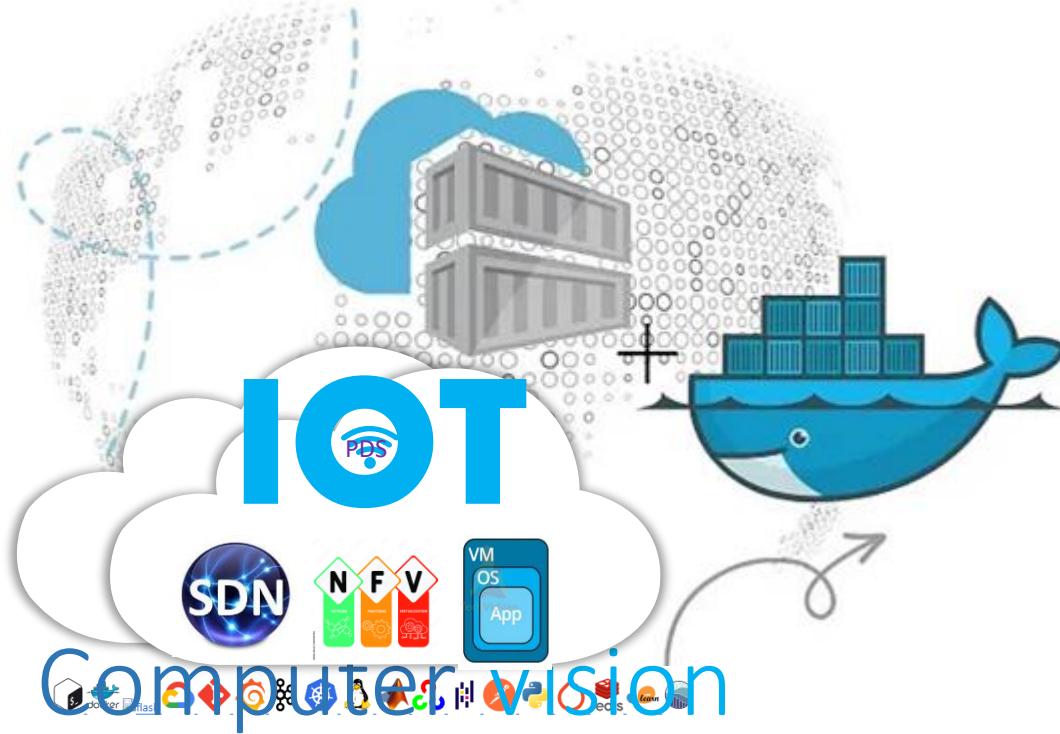
1. Find a very large dataset that has similar data, train a big ConvNet there
2. Transfer learn to your dataset

Deep learning frameworks provide a “Model Zoo” of pretrained models so you don’t need to train your own

TensorFlow: <https://github.com/tensorflow/models>

PyTorch: <https://github.com/pytorch/vision>

- Improve your training error:
 - Optimizers
 - Learning rate schedules
- Improve your test error:
 - Regularization
 - Choosing Hyperparameters



Thank you!
Thank you For being a great class!

https://github.com/amelcharolinesgn2/IoT_simulator-mqtt-NodeRed

<https://www.youtube.com/@AmelOline/videos>

<https://github.com/siagianp>