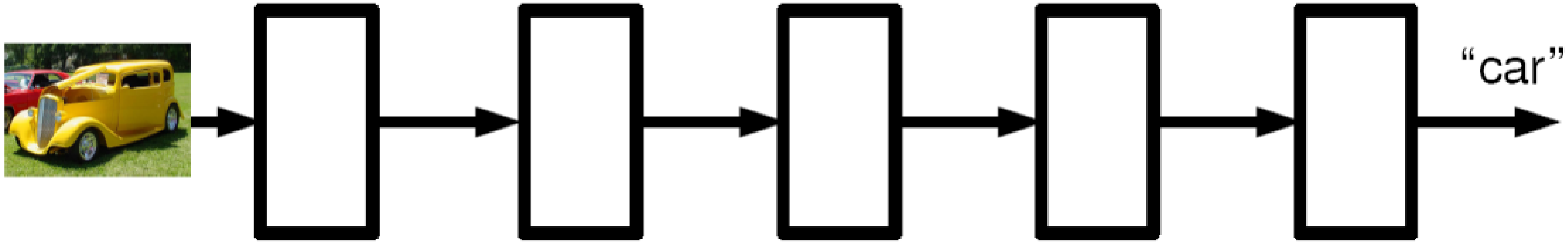
DEEP LEARNING

Artificial Neural Networks

Deep Learning (DL)

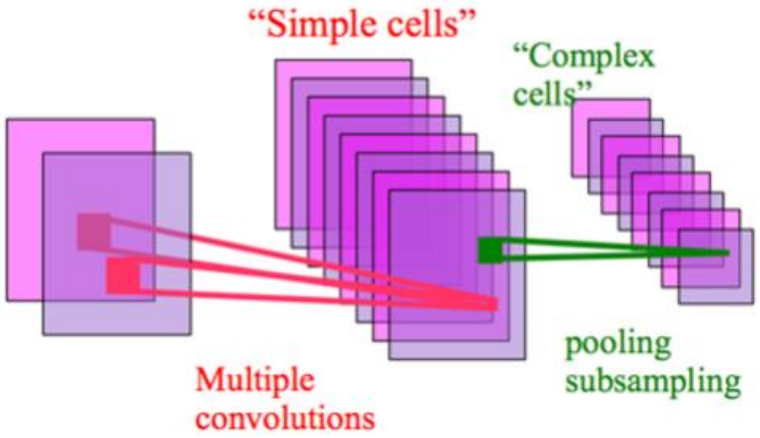
- "DL is part of a broader family of machine learning methods based on artificial neural networks with representation learning" – WIKI



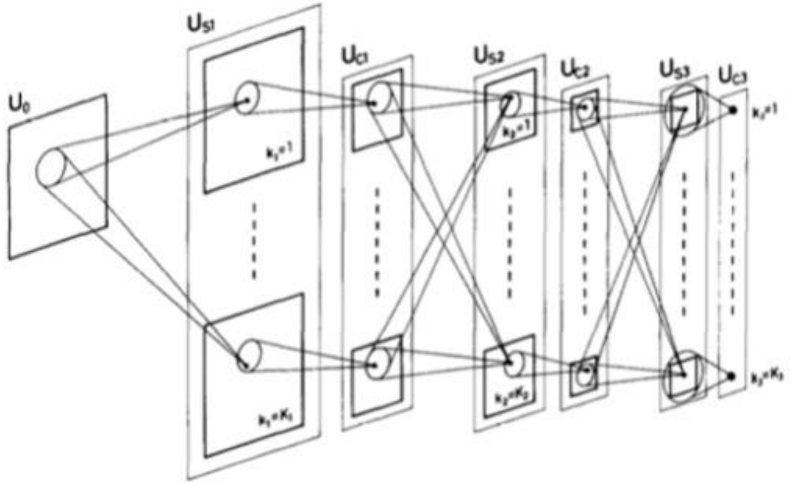
- Cascade of non-linear transformations
- End to end learning
- General framework (any hierarchical model is deep)

Deep Learning

- Before 2012
  - Hubel & Wiesel ('60s) Simple & Complex cells architecture:



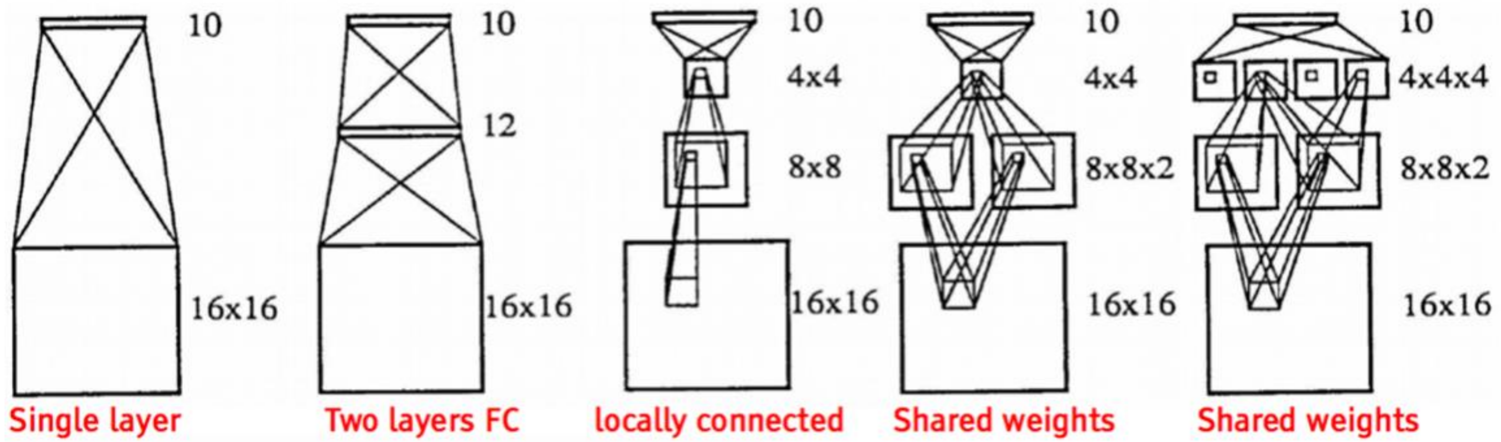
  - Fukushima's Neocognitron ('70s):



Figures from Yann LeCun's CVPR 2015 plenary

Deep Learning

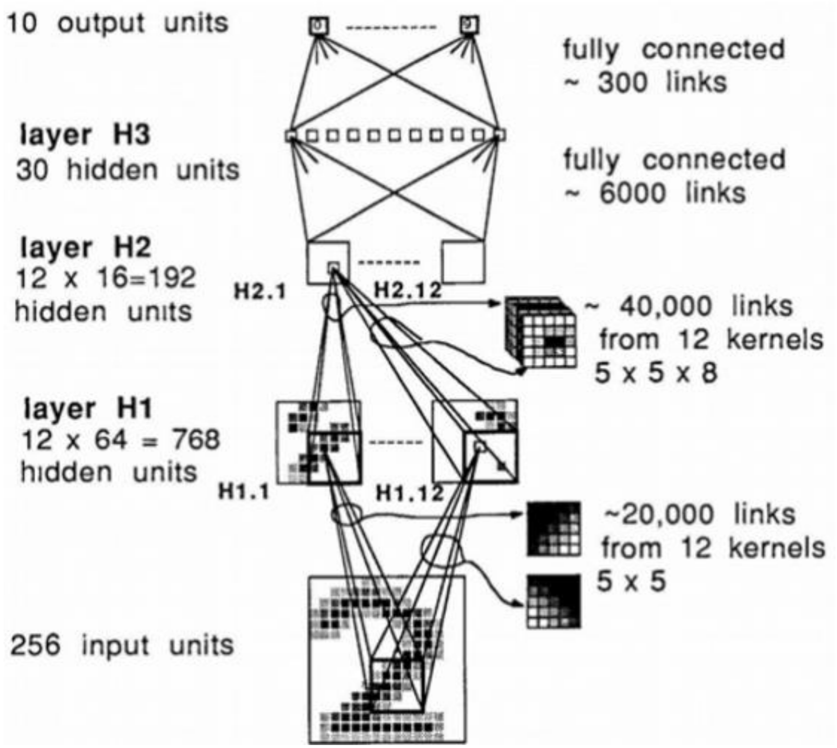
- Before 2012
  - Yann LeCun’s Early ConvNets (’80s):
  - Used for character recognition
  - Trained with back propagation



Figures from Yann LeCun’s CVPR 2015 plenary

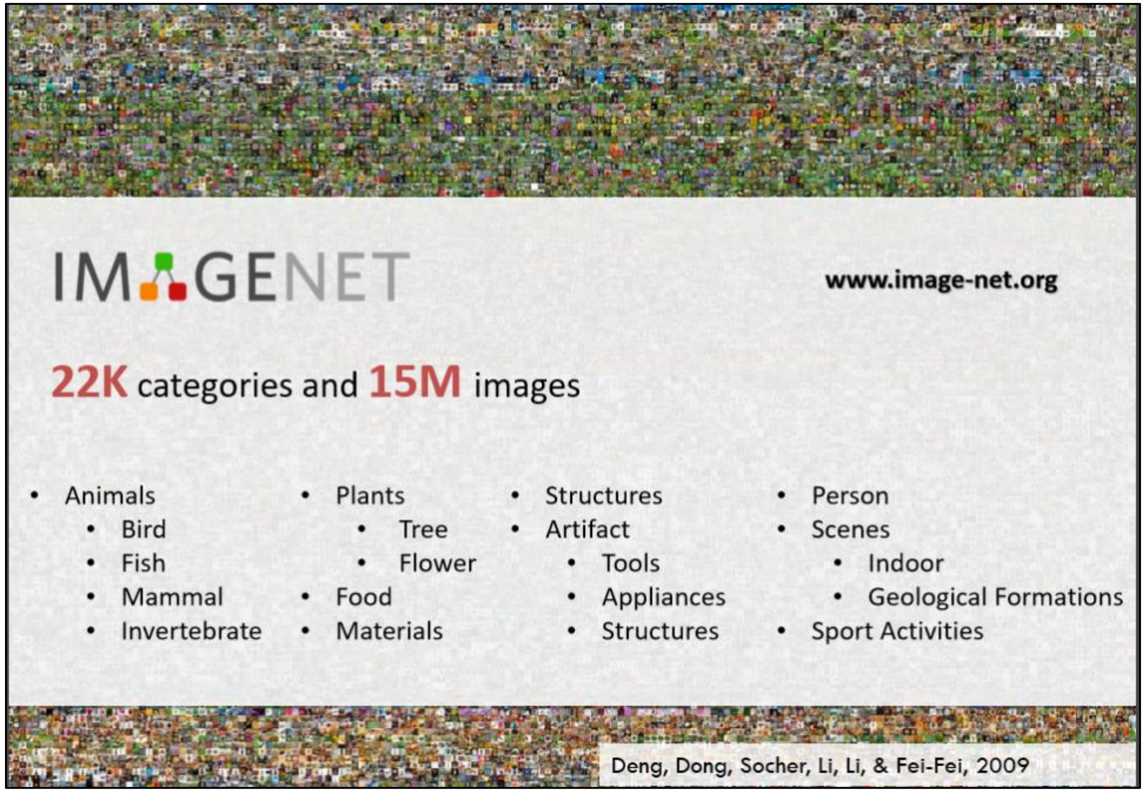
Deep Learning

- Before 2012
  - Competitive performance, but not widely used
  - State-of-the-art in handwritten pattern recognition (LeCun et al. ’89, Ciresan et. al. ’07, etc.]



Figures from Yann LeCun’s CVPR 2015 plenary

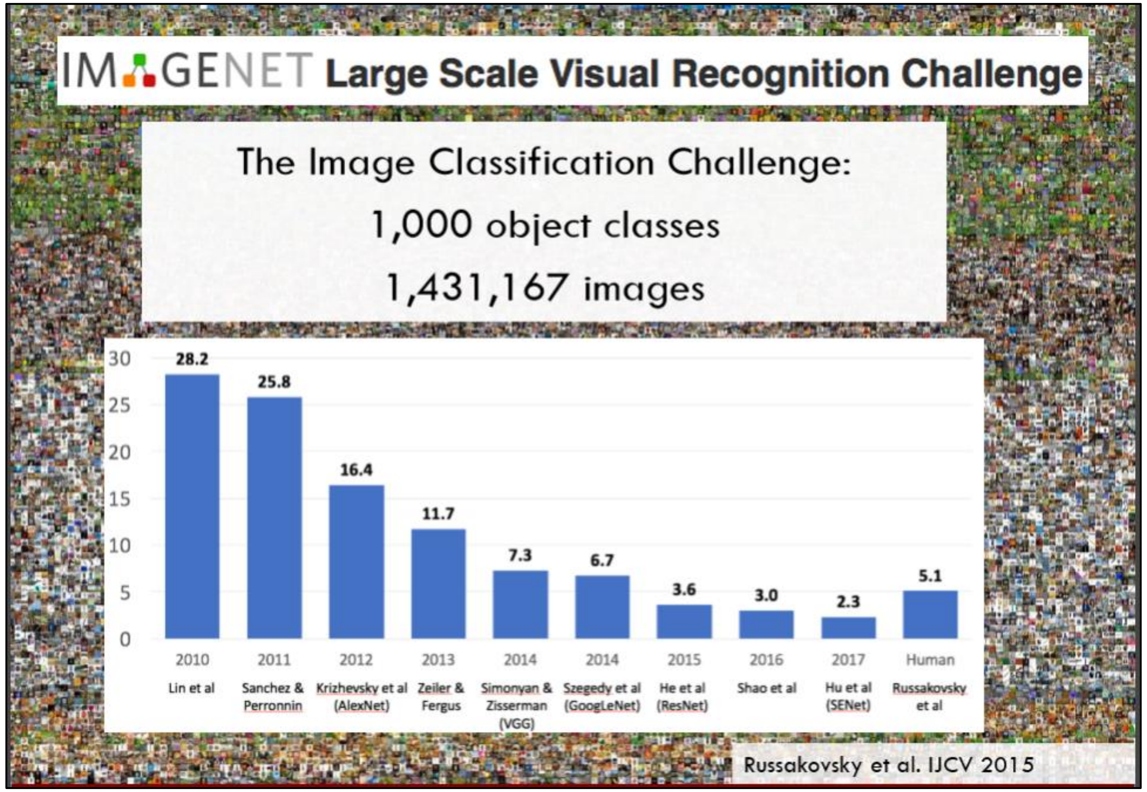
Deep Learning: New Era (1/2)



ImageNet Dataset

Figures from Fei-Fei Li's slides, CVPR 2015 plenary

Deep Learning: New Era (2/2)



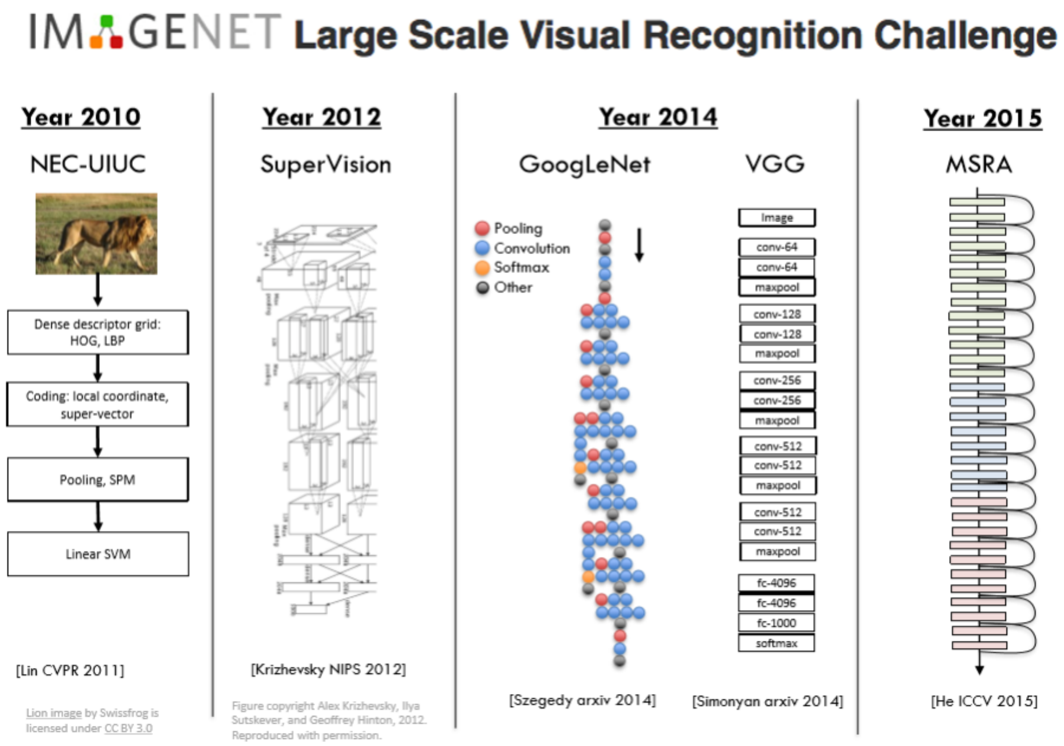
ImageNet Classification Challenge

Figures from Fei-Fei Li's slides, CVPR 2015 plenary

Deep Learning

What makes this result possible?

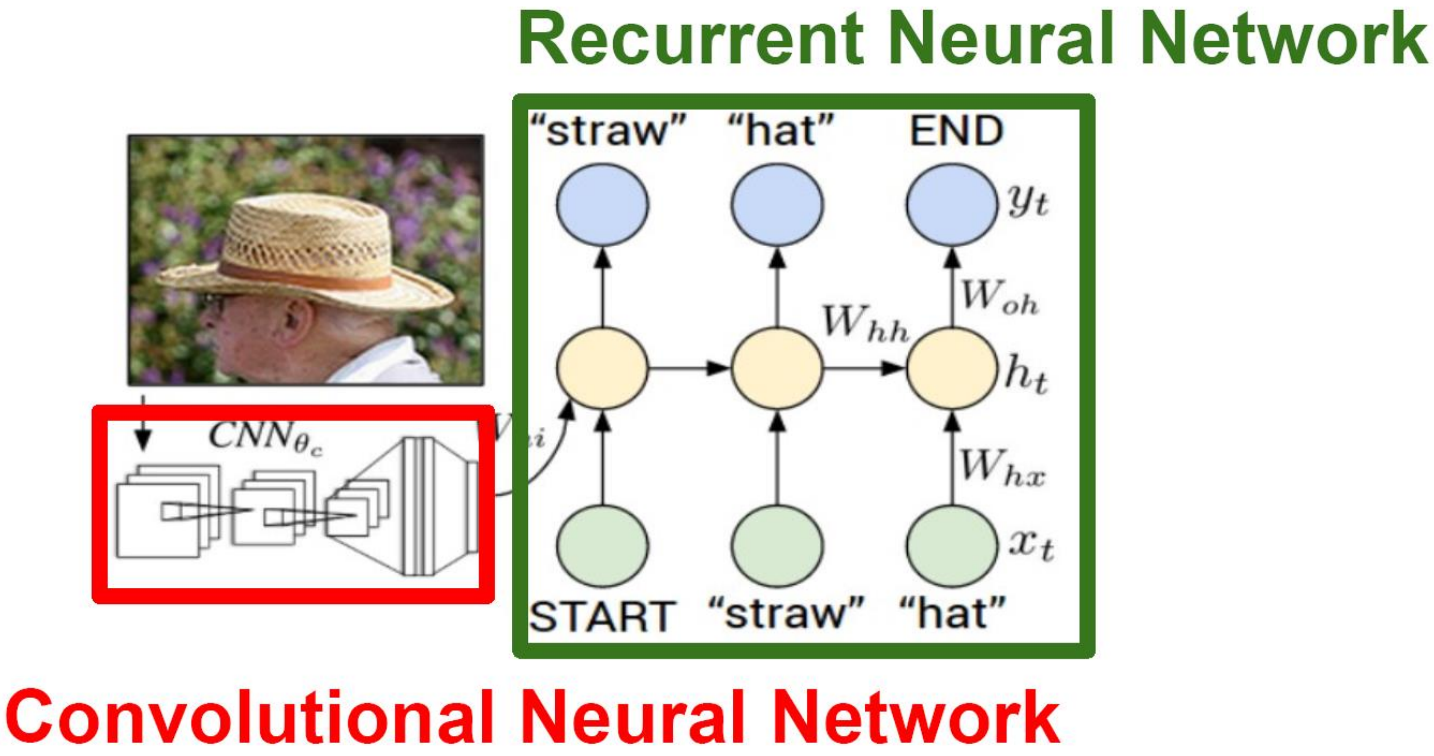
- More data to train
  - 1.4 million, high-resolution training samples
  - 1000 object categories
- Better Hardware (GPU)
- Better algorithm (e.g., Dropout, batch normalization)



Figures from Fei-Fei Li's slides

Deep Learning

- Brilliant performance in different applications and dataset



MSR 2014

Deep Learning: Generative Models

- Train CNN in an adversarial way

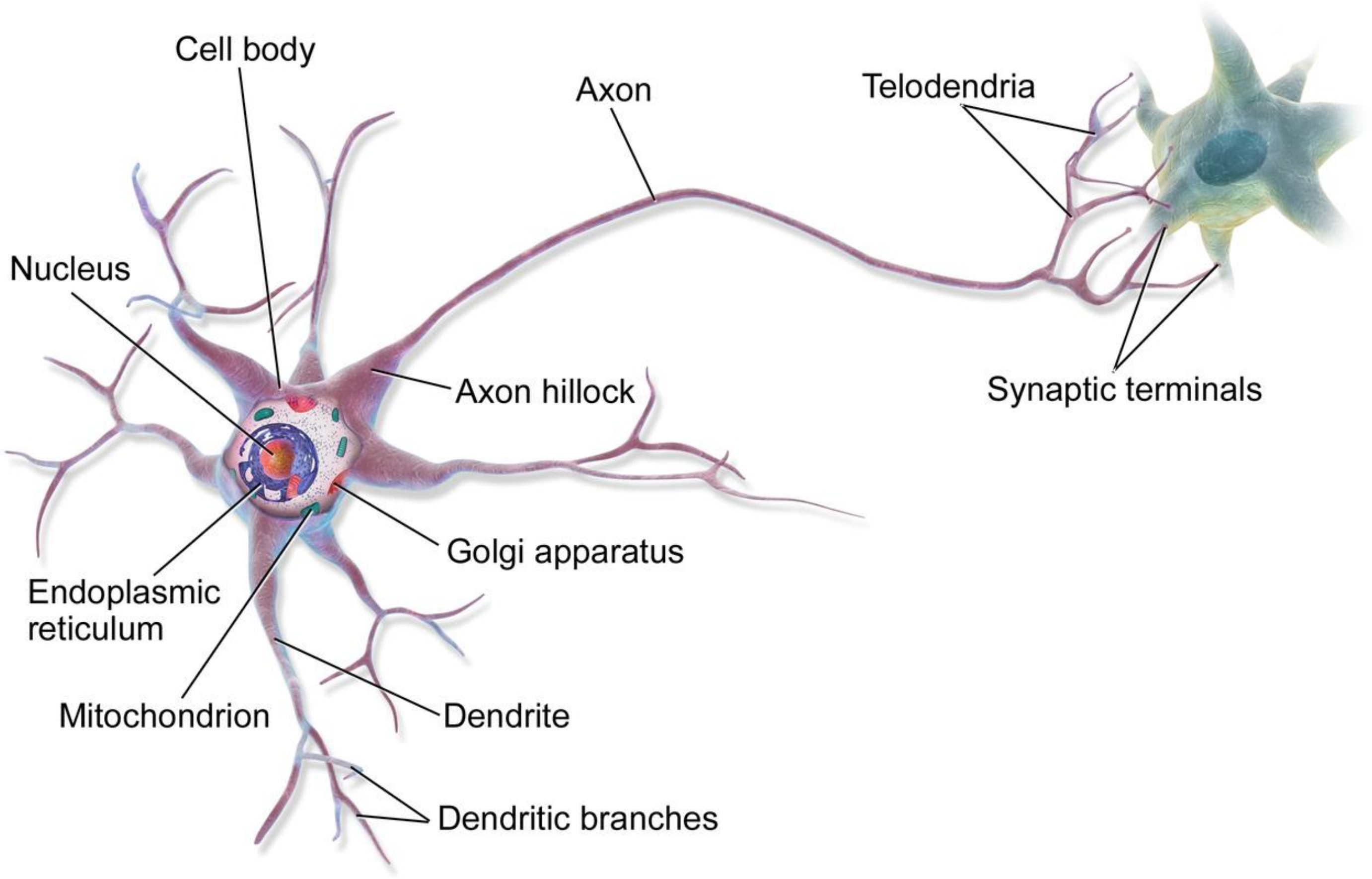
Style-GAN
Karrans et. al., 2020

Neural Networks

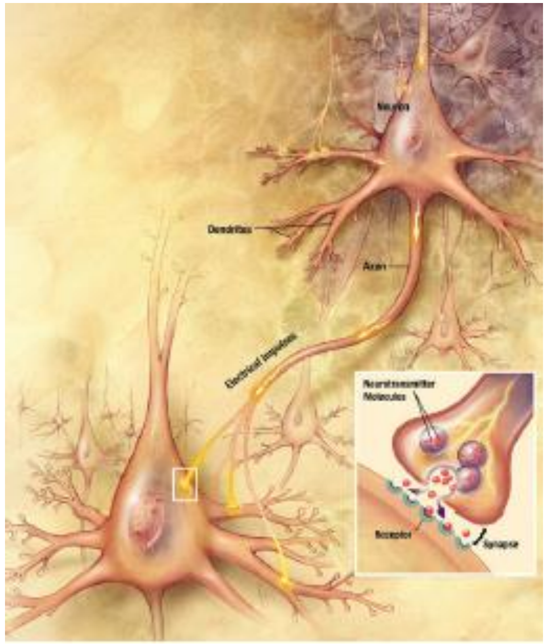
Neural Function

- Brain function (thought) occurs as the result of the firing of **neurons**
- Neurons connect to each other through **synapses**, which propagate **action potential** (electrical impulses) by releasing **neurotransmitters**
  - Synapses can be excitatory (potential-increasing) or inhibitory (potential-decreasing), and have varying activation thresholds
  - Learning occurs as a result of the synapses’ plasticity: They exhibit long-term changes in connection strength
- There are about 10^{11} neurons and about 10^{14} synapses in the human brain!

Biology of A Neuron



Source: ["Blausen 0657 - Multipolar neuron - English labels "]() by [Bruce Blaus](), license: [CC BY]().



Source: National Institutes of Health

Brain Structure

- Different areas of the brain have different functions
  - Some areas seem to have the same function in all humans (e.g., Broca’s region for motor speech); the overall layout is generally consistent
  - Some areas are more plastic, and vary in their function; also, the lower-level structure and function vary greatly
- We don’t know how different functions are “assigned” or acquired
  - Partly the result of the physical layout / connection to inputs (sensors) and outputs (effectors)
  - Partly the result of experience (learning)

Comparison of Computing Power

INFORMATION CIRCA 2012	Computer	Human Brain
Computation Units	10-core Xeon: 10^9 Gates	10^{11} Neurons
Storage Units	10^9 bits RAM, 10^{12} bits disk	10^{11} neurons, 10^{14} synapses
Cycle time	10^{-9} sec	10^{-3} sec
Bandwidth	10^9 bits/sec	10^{14} bits/sec

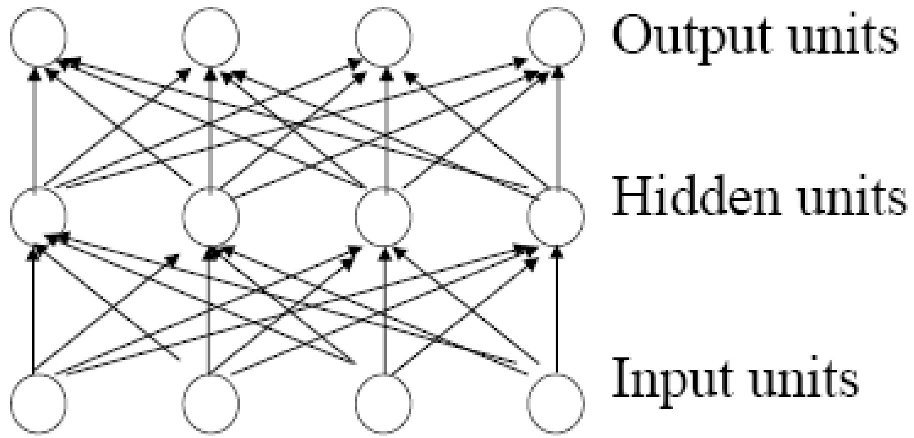
- Neural networks are made up of **nodes** or **units**, connected by **links**
- Each link has an associated **weight** and **activation level**
- Each node has an **input function** (typically summing over weighted inputs), an **activation function**, and an **output**

Neural Networks (1/2)

- Origins: Algorithms that try to mimic the brain.
- Very widely used in 80s and early 90s; popularity diminished in late 90s.
- Recent resurgence: State-of-the-art technique for many applications
- Artificial neural networks are not nearly as complex or intricate as the actual brain structure

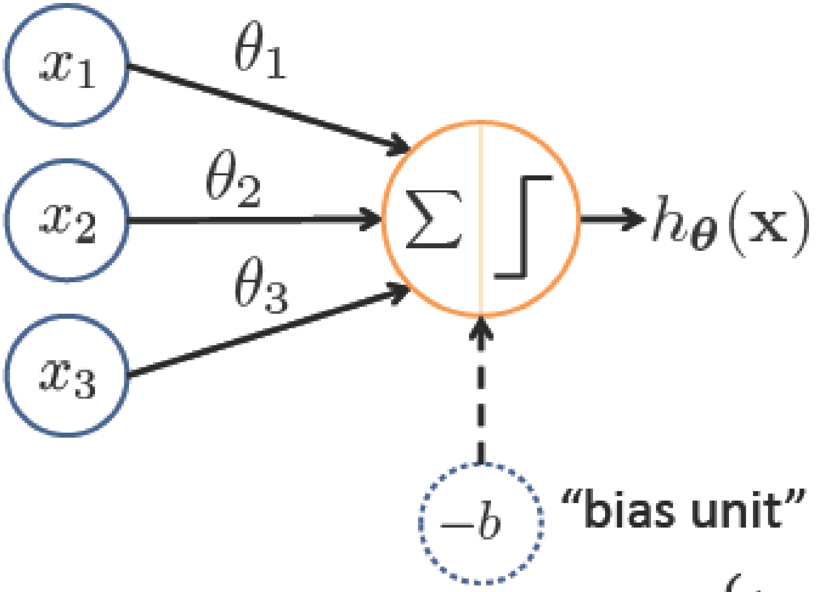
Neural Networks (2/2)

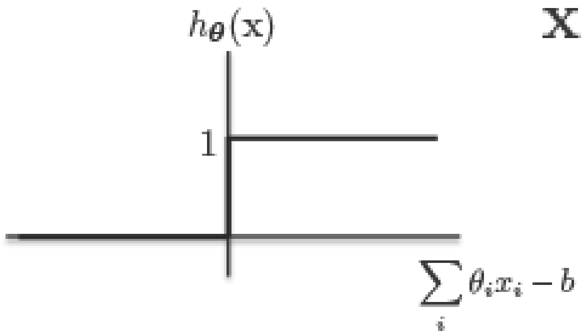
- Computers are way faster than neurons…
- But there are a lot more neurons than we can reasonably model in modern digital computers, and they all fire in parallel
- Neural networks are designed to be massively parallel
- The brain is effectively a billion times faster



Layered feed-forward network

Neuron Model: Threshold Unit





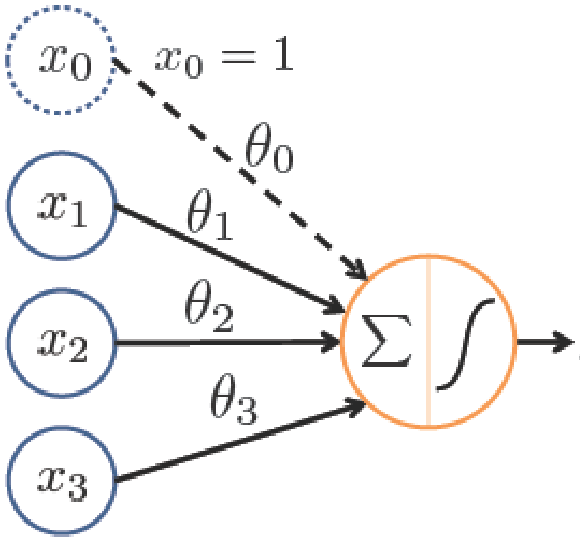
$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad \boldsymbol{\theta} = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix}$$

Activate if weighted sum of inputs is larger than "threshold"

Step activation function: $h_{\boldsymbol{\theta}}(\mathbf{x}) = \begin{cases} 1 & \text{if} \quad \sum_i \theta_i x_i \geq b \\ 0 & \text{otherwise} \end{cases}$

Neuron Model: Logistic Unit

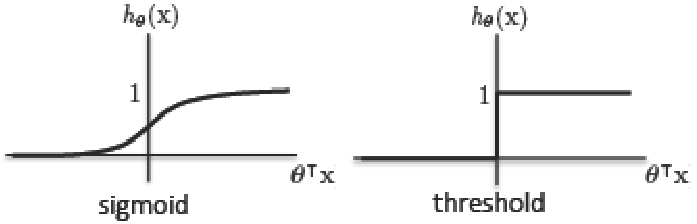
“bias unit”



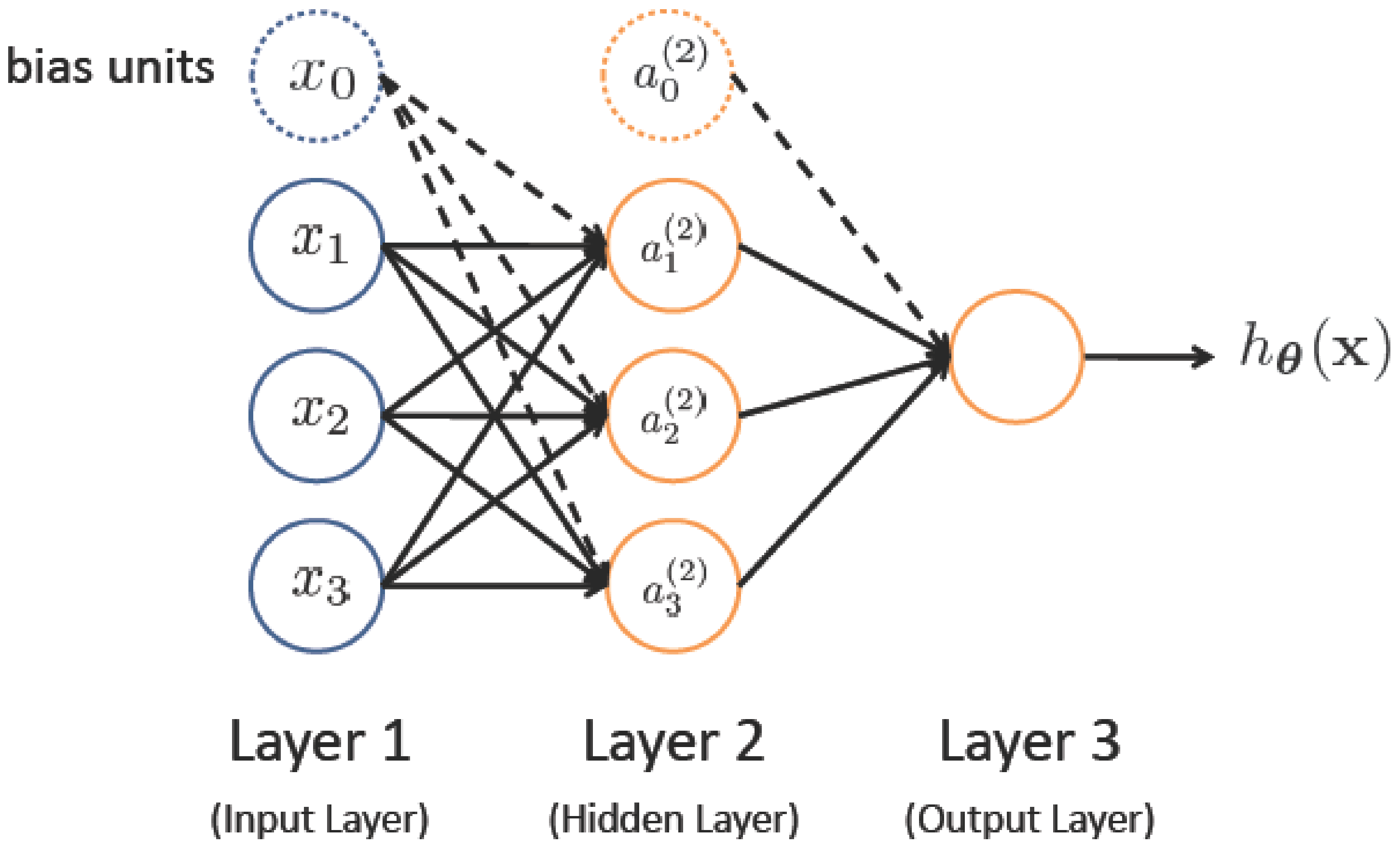
$$\mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \qquad \boldsymbol{\theta} = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix}$$

$$h_{\boldsymbol{\theta}}(\mathbf{x}) = g\left(\boldsymbol{\theta}^\intercal \mathbf{x}\right) = \frac{1}{1 + e^{-\boldsymbol{\theta}^\intercal \mathbf{x}}}$$

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



Neural Network



Feed-forward Process (1/2)

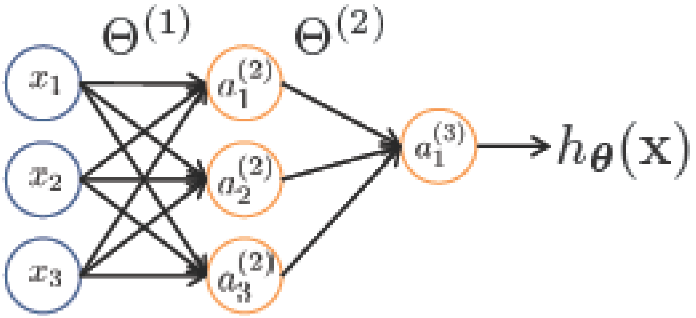
- Input layer units are set by some exterior function (think of these as **sensors**), which causes their output links to be **activated** at the specified level

- 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

Feed-forward Process (2/2)

- 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

Neural Network



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

$\Theta^{(j)} =$ weight matrix controlling function mapping 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 $s_{j+1} \times (s_j+1)$.

$$\Theta^{(1)} \in \mathbb{R}^{3\times4} \qquad \Theta^{(2)} \in \mathbb{R}^{1\times4}$$

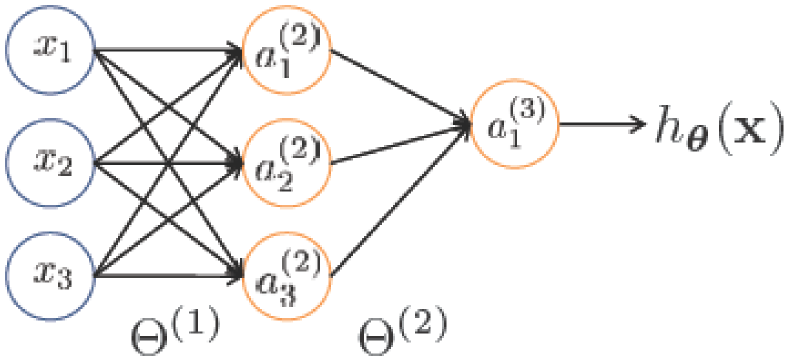
Vectorization

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

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

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

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



Feed-Forward Steps:

$$\mathbf{z}^{(2)} = \Theta^{(1)}\mathbf{x}$$

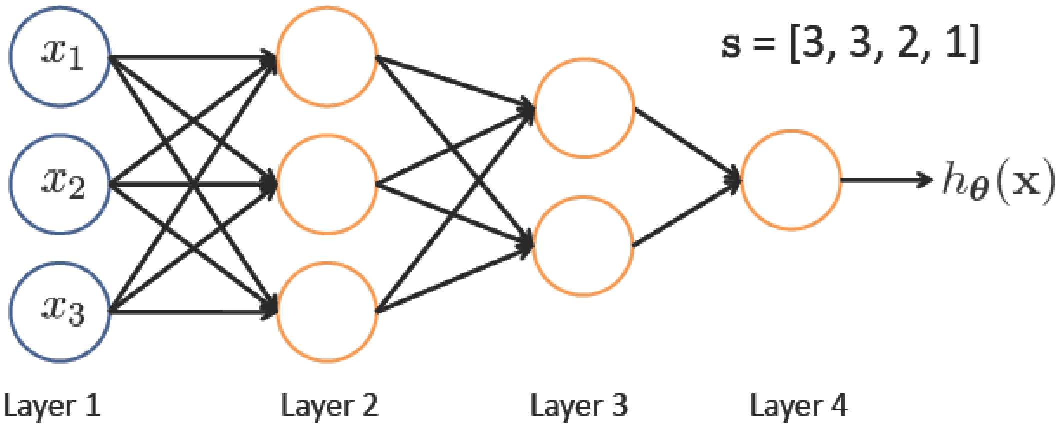
$$\mathbf{a}^{(2)} = g(\mathbf{z}^{(2)})$$

$$\text{Add } a_0^{(2)} = 1$$

$$\mathbf{z}^{(3)} = \Theta^{(2)}\mathbf{a}^{(2)}$$

$$h_\Theta(\mathbf{x}) = \mathbf{a}^{(3)} = g(\mathbf{z}^{(3)})$$

Other Network Architectures



- L denotes the number of layers
- $\mathrm{s} \in \mathbb{N}^{+L}$ contains the numbers of nodes at each layer
  - Not counting bias units
  - Typically, $s_0 = d$ (# input features) and $s_{L\text{-}1} = K$ (# classes)

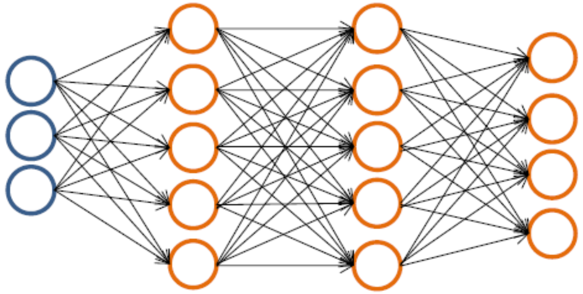
Multiple Output Units: One-vs-Rest (1/2)

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

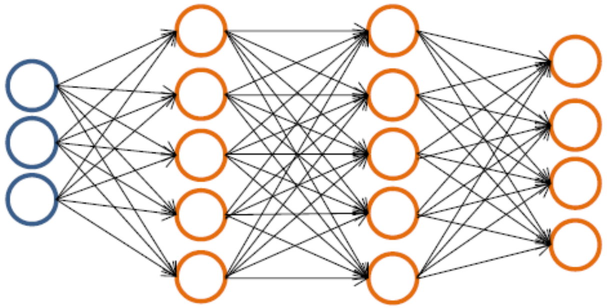
$$h_\Theta(\mathbf{x}) \approx \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

when motorcycle

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

when truck

Multiple Output Units: One-vs-Rest (2/2)



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

We want:

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

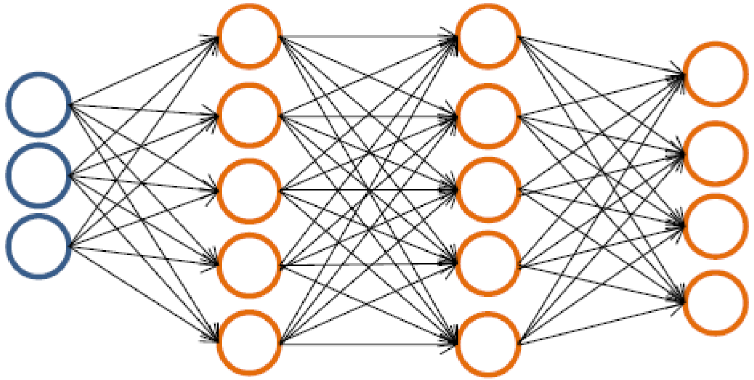
$h_\Theta(\mathbf{x}) \approx \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$ when car

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

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

- Given $\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)\}$
- Must convert labels to 1-of-K representation
  - e.g., $\mathbf{y}_i = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$ when motorcycle, $\mathbf{y}_i = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$ when car, etc.

Neural Network Classification (1/3)



Given:

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

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

- $s_0 = d$ (# features)

<u>Binary classification</u>

y = 0 or 1

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

<u>Multi-class classification</u> (K classes)

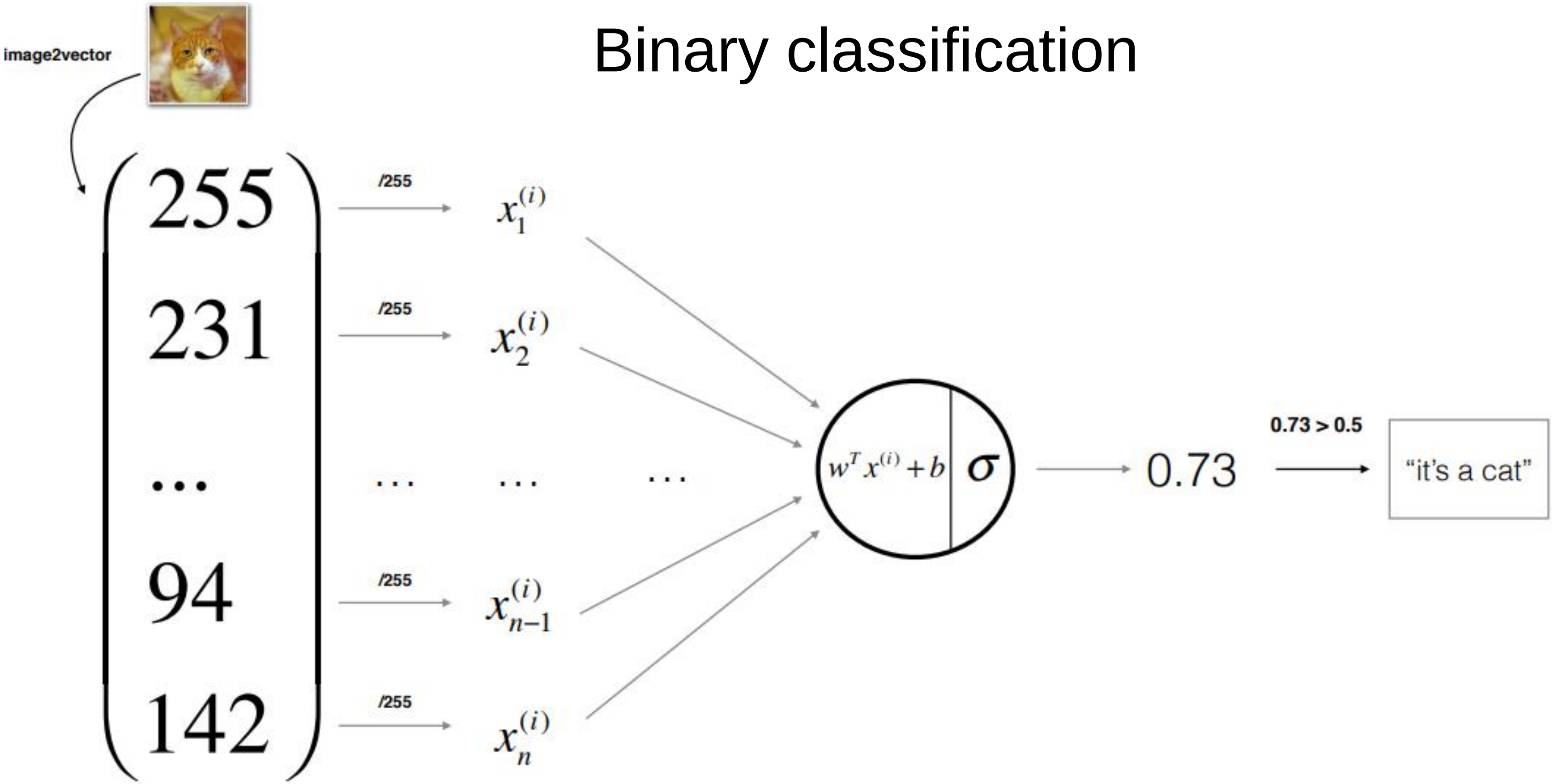
$\mathbf{y} \in \mathbb{R}^K$ e.g. $\begin{bmatrix}1\\0\\0\\0\end{bmatrix}$, $\begin{bmatrix}0\\1\\0\\0\end{bmatrix}$, $\begin{bmatrix}0\\0\\1\\0\end{bmatrix}$, $\begin{bmatrix}0\\0\\0\\1\end{bmatrix}$

pedestrian car motorcycle truck

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

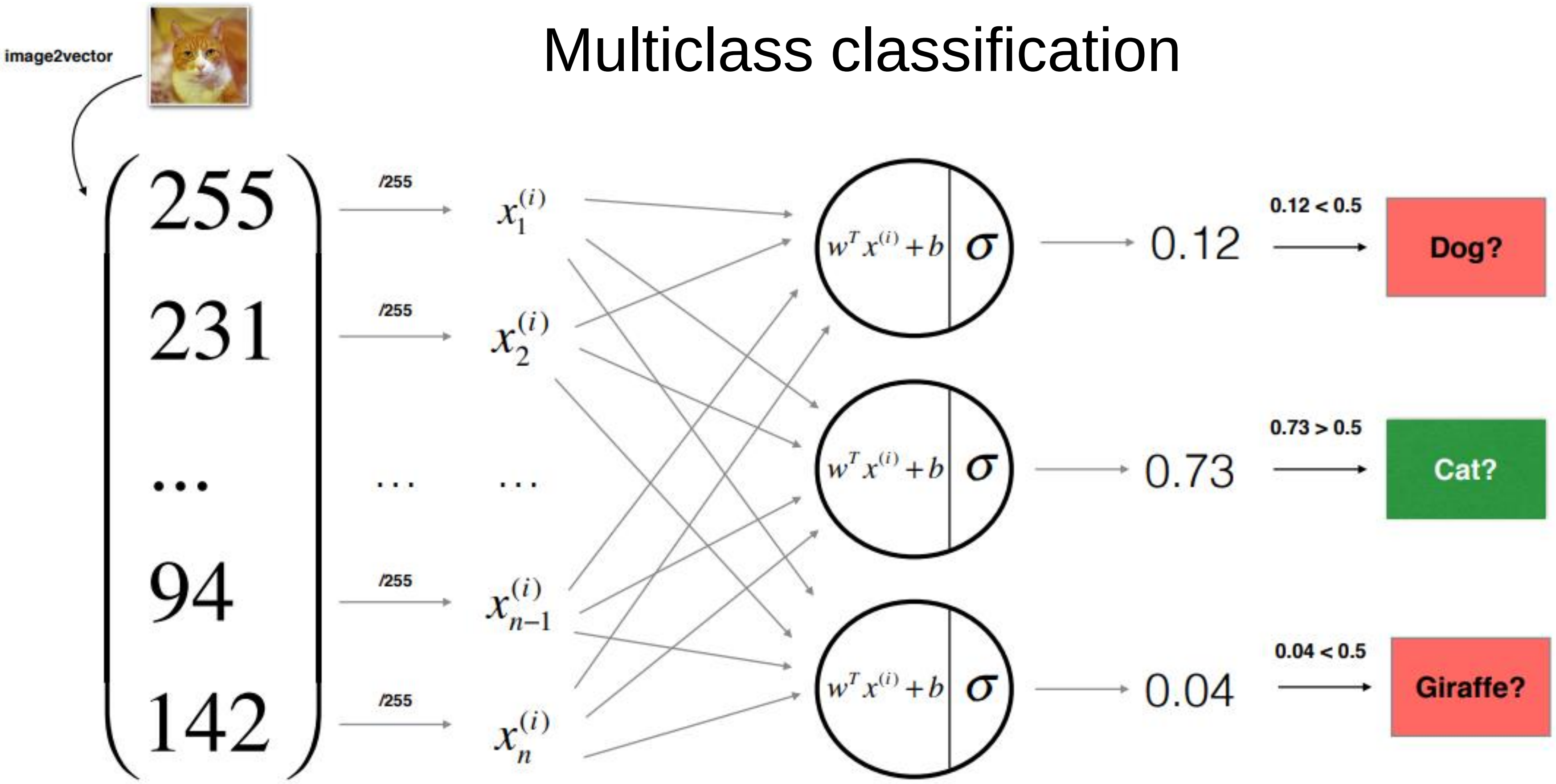
Neural Network Classification (2/3)

Binary classification



Neural Network Classification (3/3)

Multiclass classification



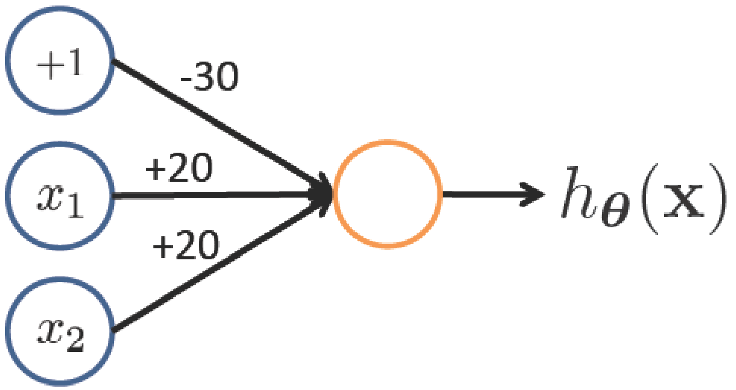
Understanding Representations

Representing Boolean Functions (1/2)

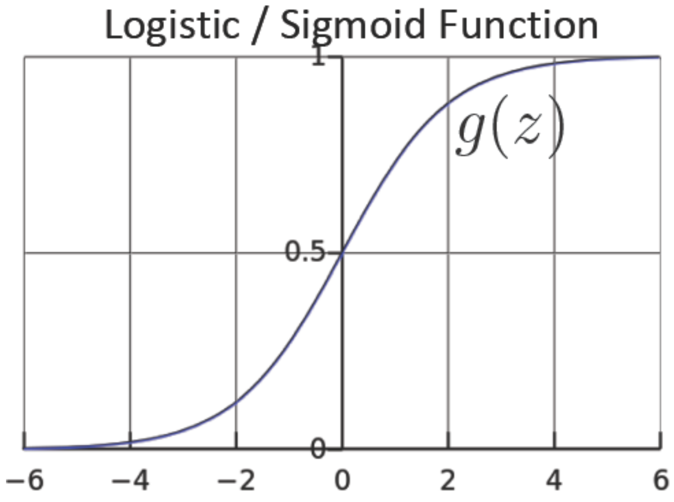
Simple example: AND

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

$y = x_1 \text{ AND } x_2$

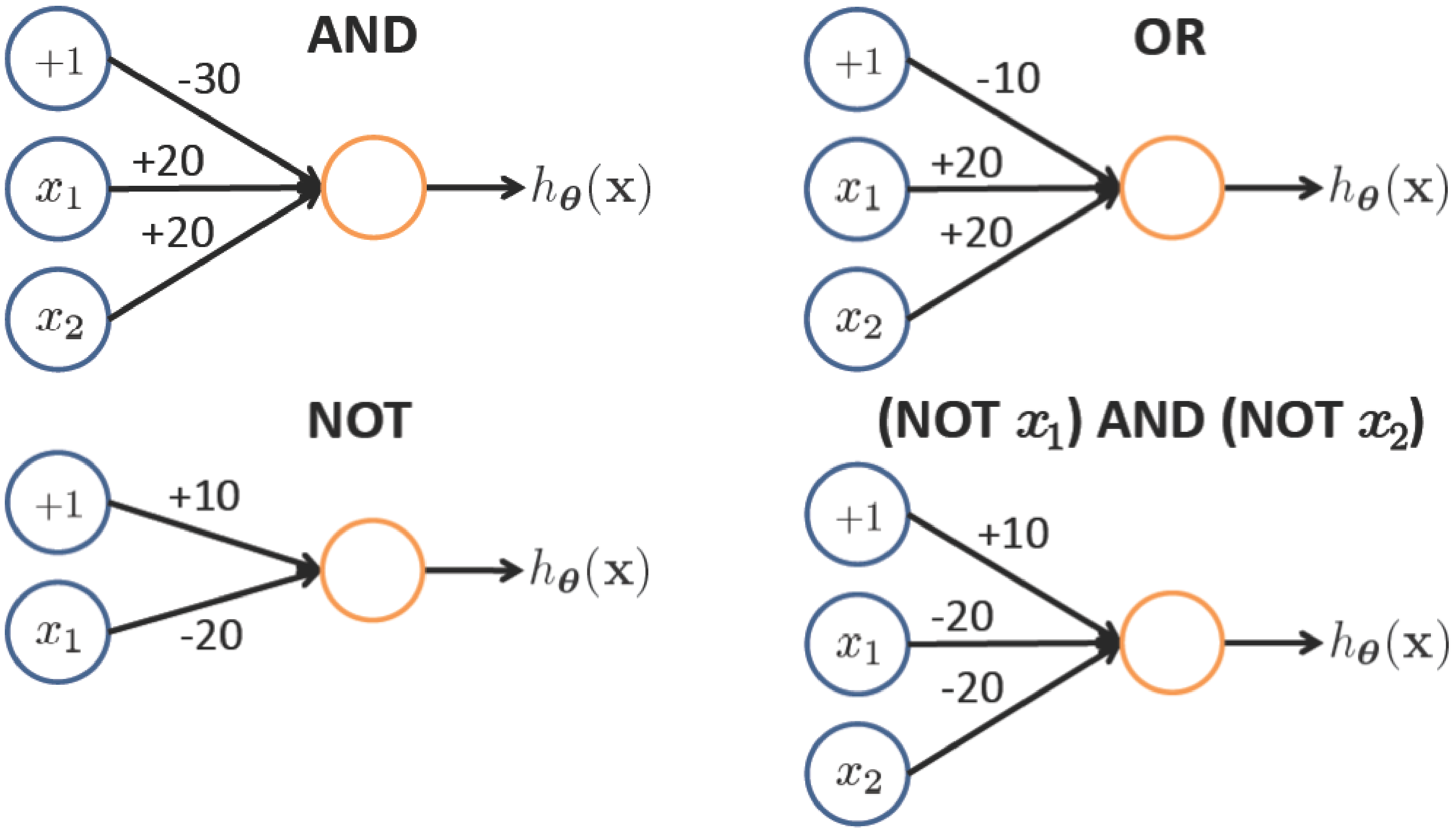


$$h_\Theta(\mathbf{x}) = g(-30 + 20x_1 + 20x_2)$$

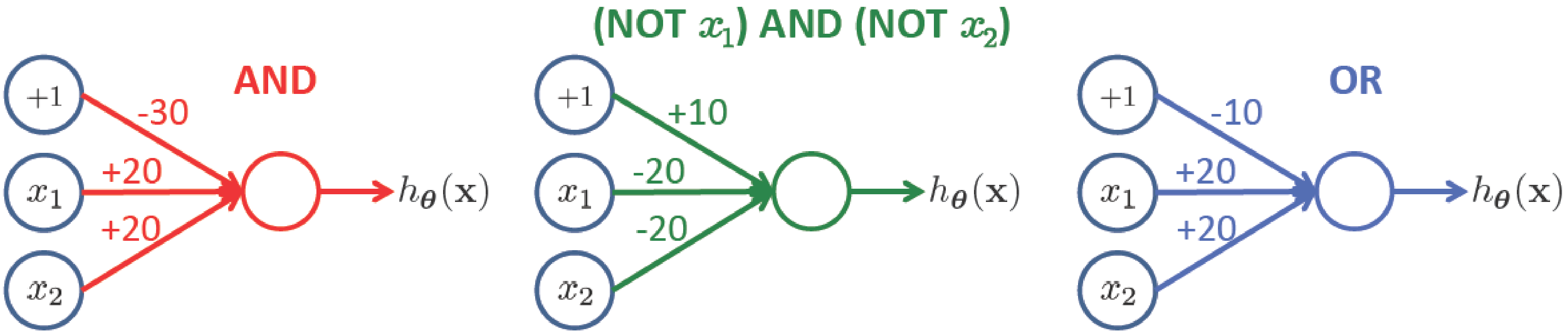


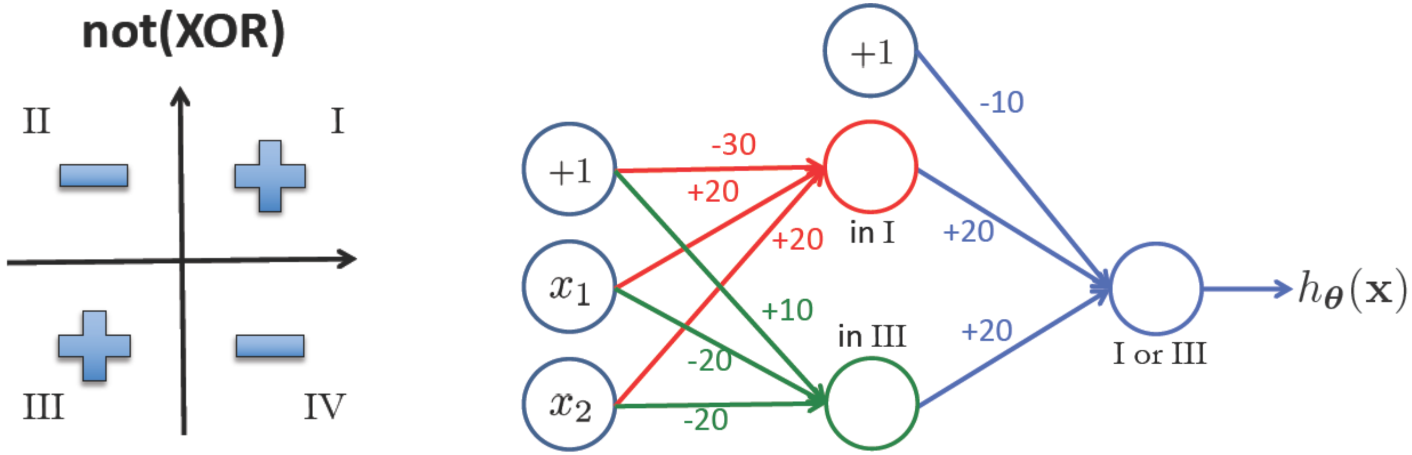
x_1	x_2	$h_\Theta(\mathbf{x})$
0	0	$g(-30) \approx 0$
0	1	$g(-10) \approx 0$
1	0	$g(-10) \approx 0$
1	1	$g(10) \approx 1$

Representing Boolean Functions (2/2)



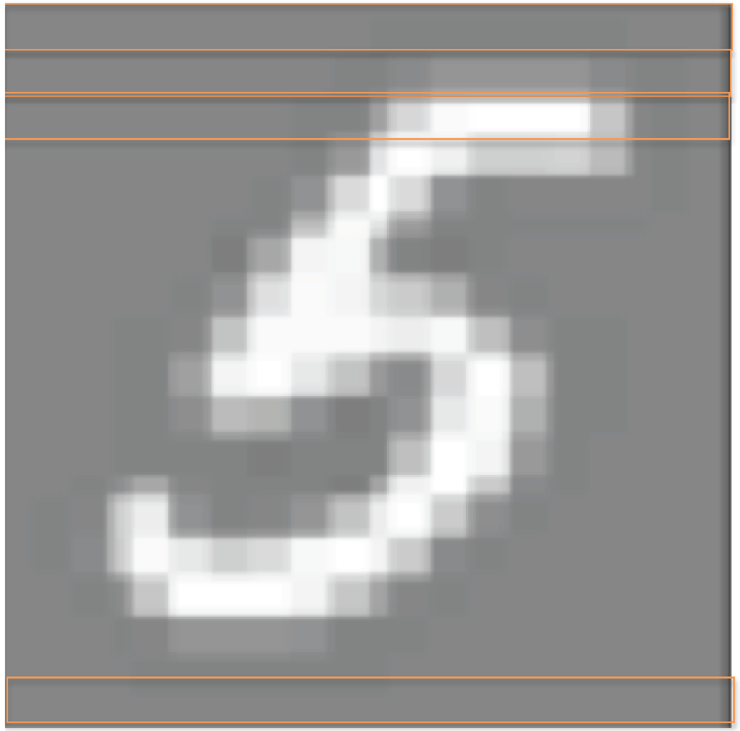
Combining Representations





Layering Representations (1/2)

20x20 pixel images
d = 400, 10 classes

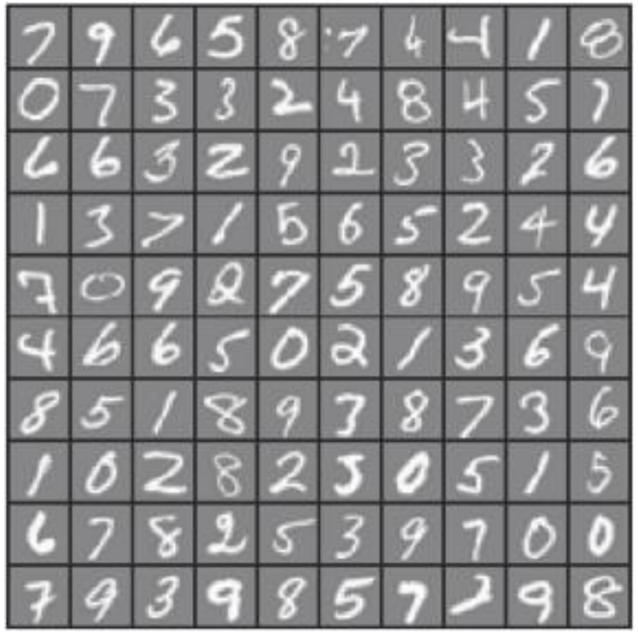


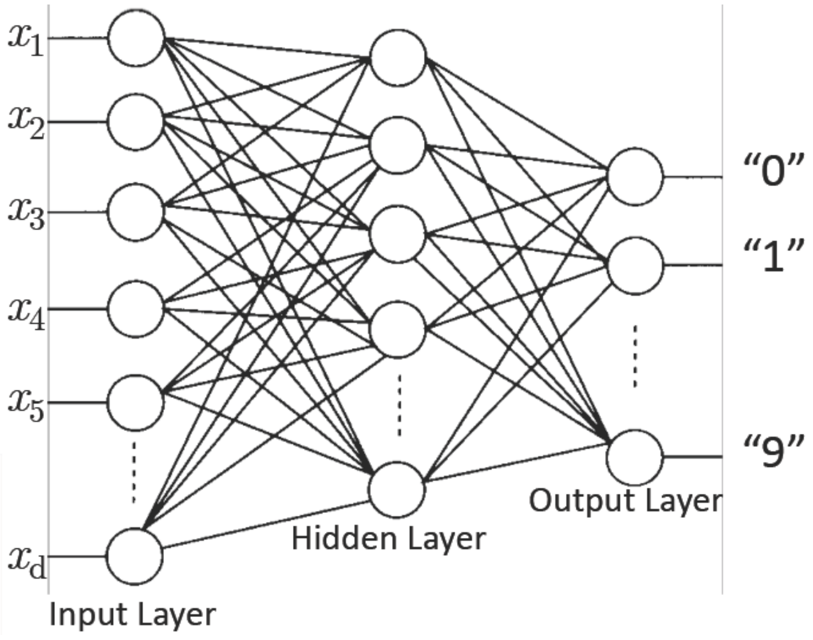
$x_1 \; \dots \; x_{20}$
$x_{21} \; \dots \; x_{40}$
$x_{41} \; \dots \; x_{60}$

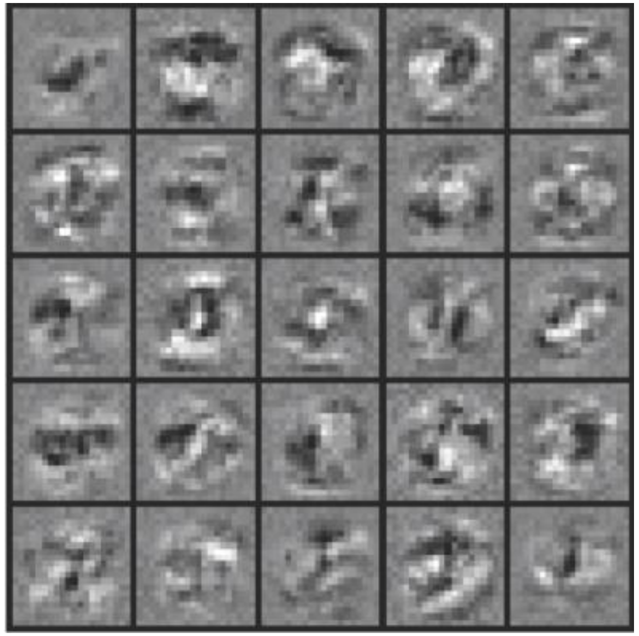
$\vdots$

$x_{381} \; \dots \; x_{400}$

Layering Representations (2/2)







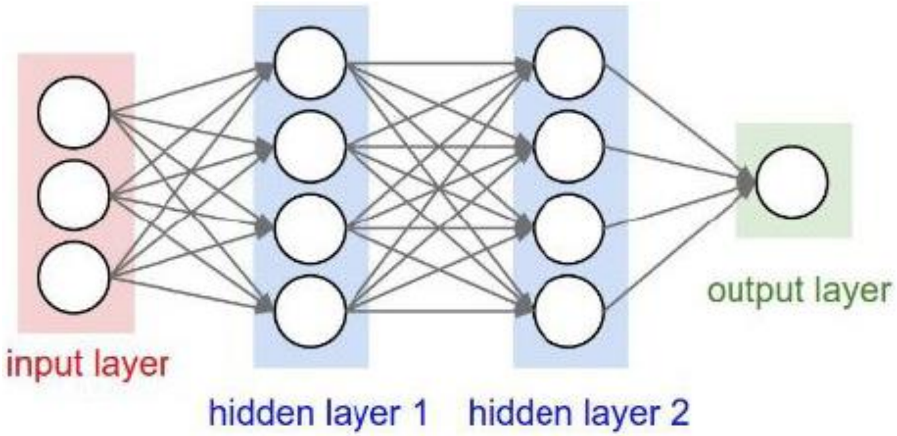
Visualization of Hidden Layer

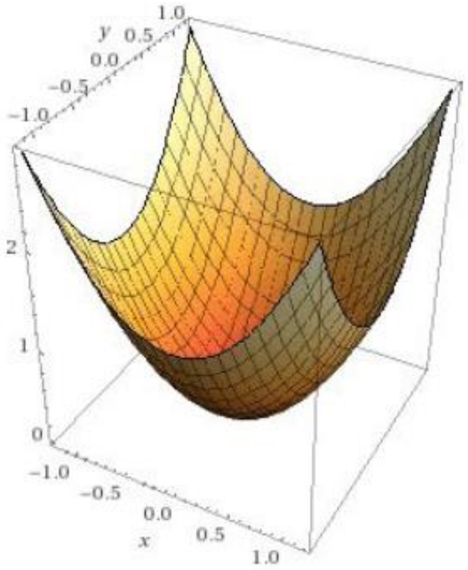
LeNet Demonstration

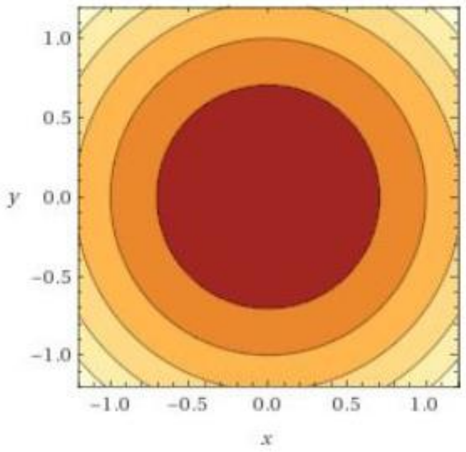
Neural Network Learning

Motivation

- Recall: Optimization objective is to minimize loss
- Goal: how should we tweak the parameters to decrease the loss slightly?







Problem Statement

- Given a function ***f*** with respect to inputs ***x***, labels ***y***, and parameters compute the gradient of ***Loss*** with respect to θ

$$\mathrm{Loss} = f(x,y;\,\theta)$$

Backpropagation

- An algorithm for computing the gradient of a **compound** function as a series of **local, intermediate gradients**

$$\mathrm{Loss} = ((\sigma(xW_1 + b_1)W_2 + b_2) - y)^2$$

- Identify intermediate functions (forward prop)
- Compute local gradients
- Combine with upstream error signal to get full gradient

Modularity (1/2)

- Simple example
  - Compound function $f(x, y, z) = (x + y)z$
  - Intermediate variables $q = x + y$
  - Forward propagation $f = qz$

Modularity (2/2)

- Neural network example
  - Compound function $$Loss = ((\sigma(xW_1 + b_1)W_2 + b_2) - y)^2$$
  - Intermediate variables $$h_1 = xW_1 + b_1$$ $$z_1 = \sigma(h_1)$$ $$z_2 = z_1W_2 + b_2$$
  - Forward propagation $$Loss = (z_2 - y)^2$$

Forward and Backward

Intermediate **Variables**
(forward propagation)

$$h_1 = xW_1 + b_1$$

$$z_1 = \sigma(h_1)$$

$$z_2 = z_1 W_2 + b_2$$

$$Loss = (z_2 - y)^2$$

Intermediate **Gradients**
(backward propagation)

$$\frac{\partial h_1}{\partial x} = W_1^T$$

$$\frac{\partial z_1}{\partial h_1} = \sigma'(h_1) = z_1 \circ (1 - z_1)$$

$$\frac{\partial z_2}{\partial z_1} = W_2^\top$$

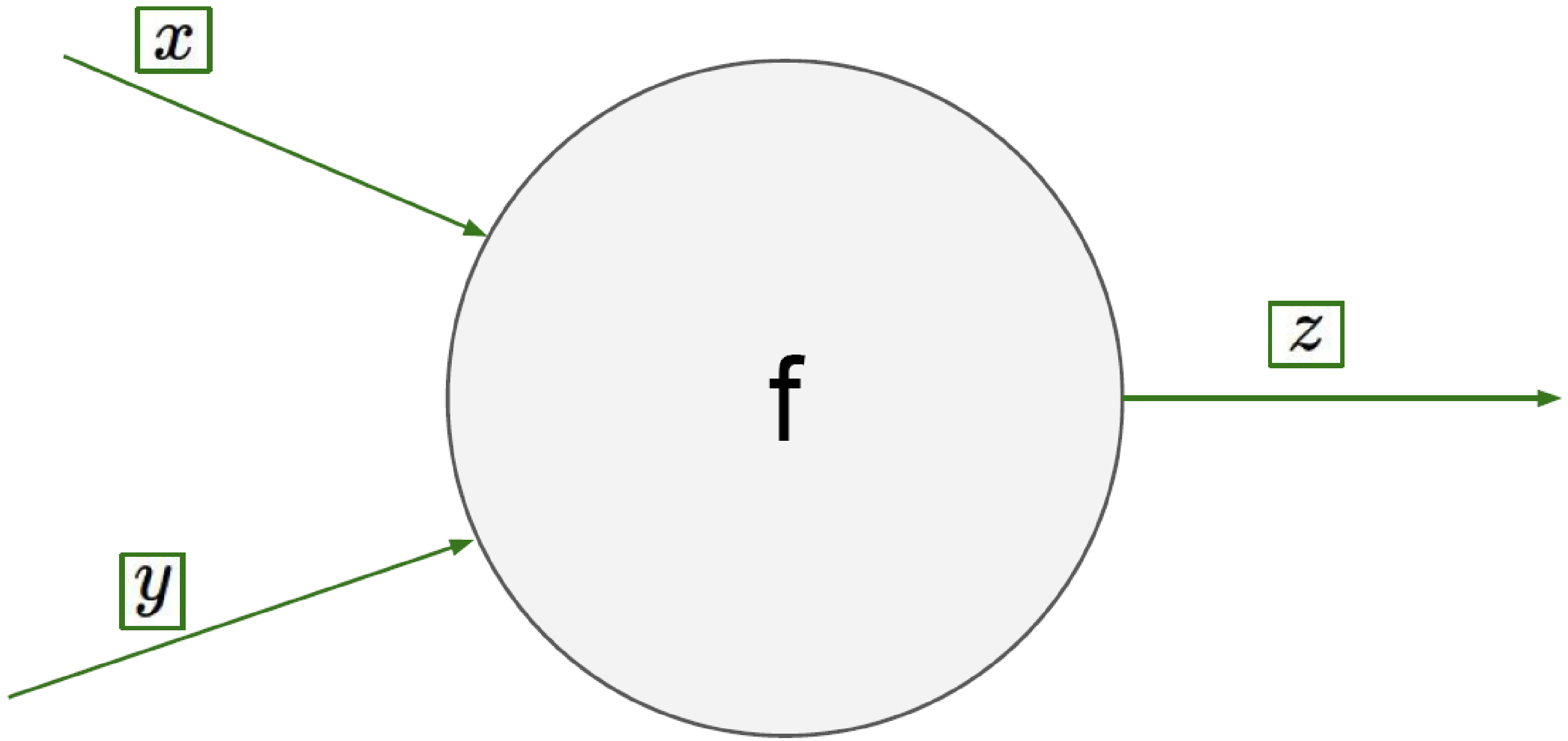
$$\frac{\partial Loss}{\partial z_2} = 2(z_2 - y)$$

Chain Rule (1/9)

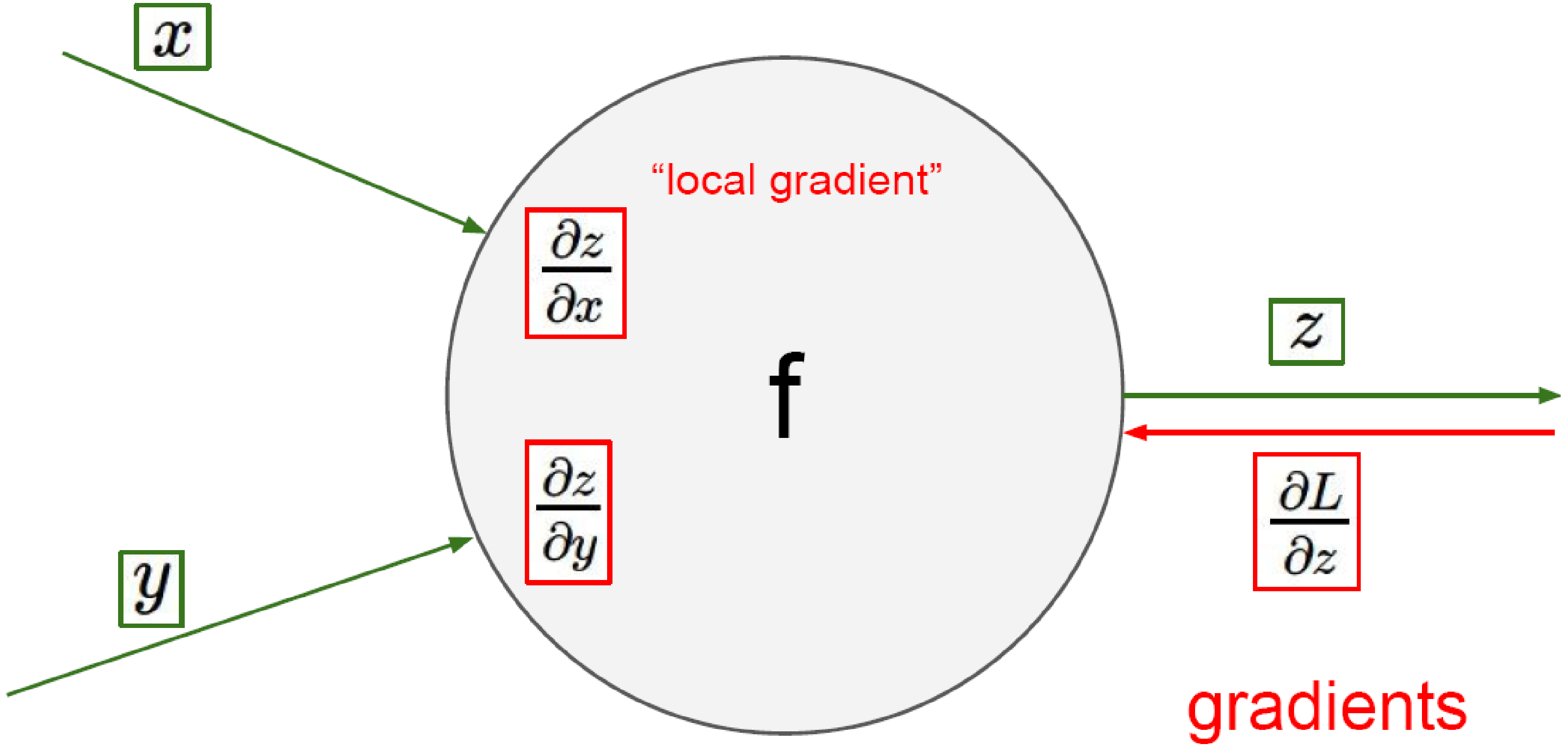
- Key chain rule intuition
  - Expresses the derivative of the composition of two differentiable functions f and g in terms of the derivatives of f and g
  - Slopes multiply

$$\frac{d((f \circ g)(x))}{dx} = \frac{d(f(g(x)))}{d(g(x))}\frac{d(g(x))}{dx}$$

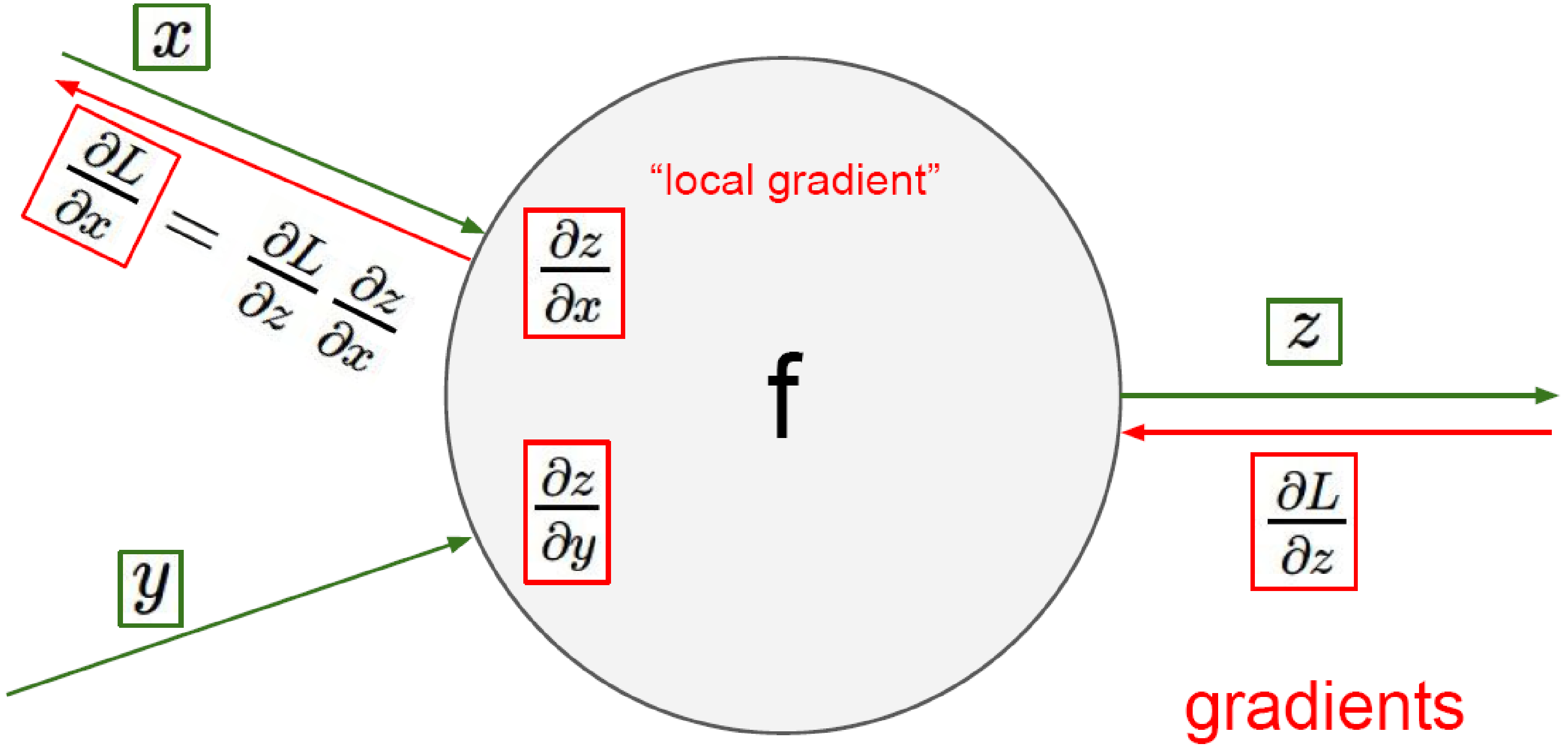
Chain Rule (2/9)



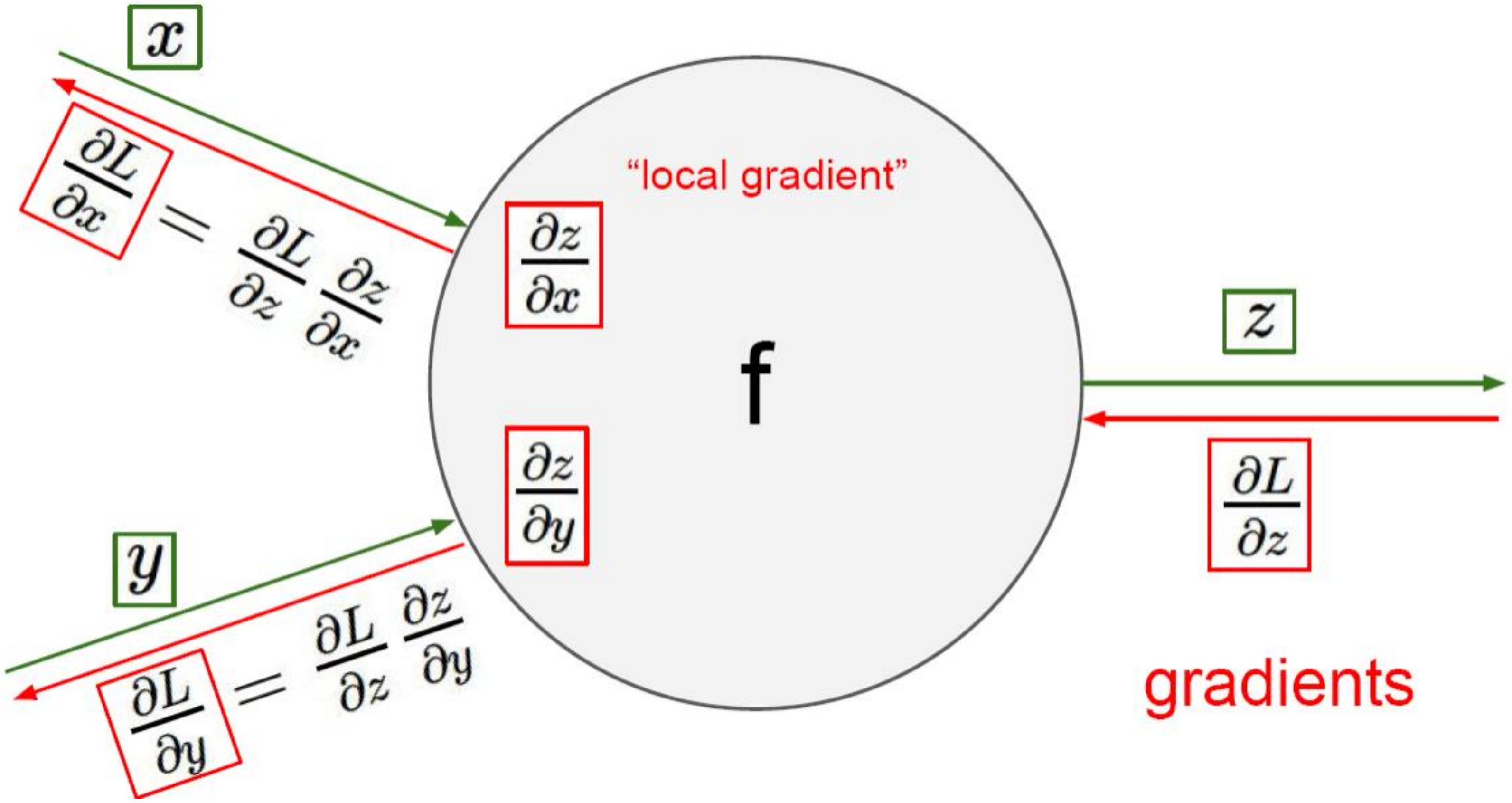
Chain Rule (3/9)



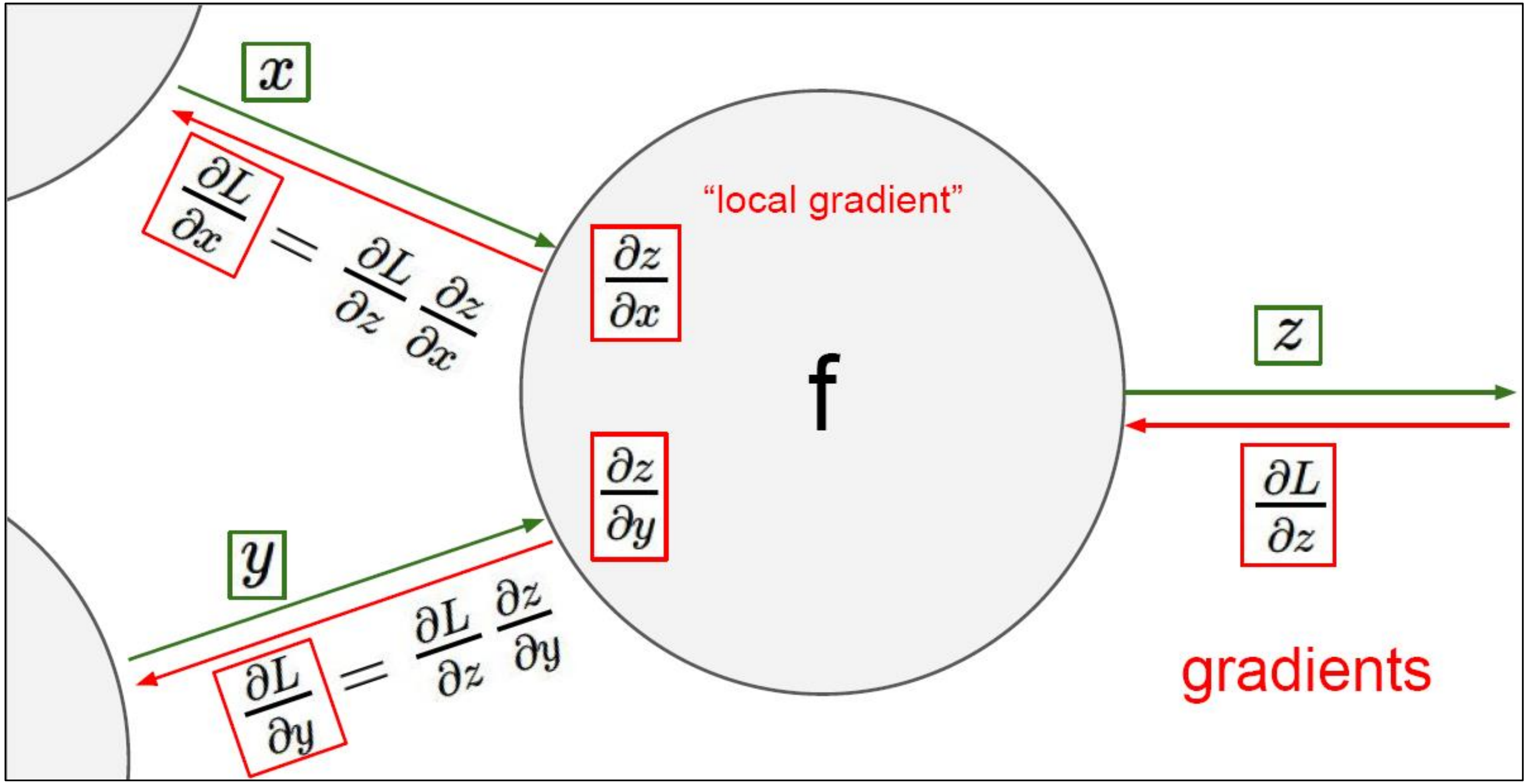
Chain Rule (4/9)



Chain Rule (5/9)

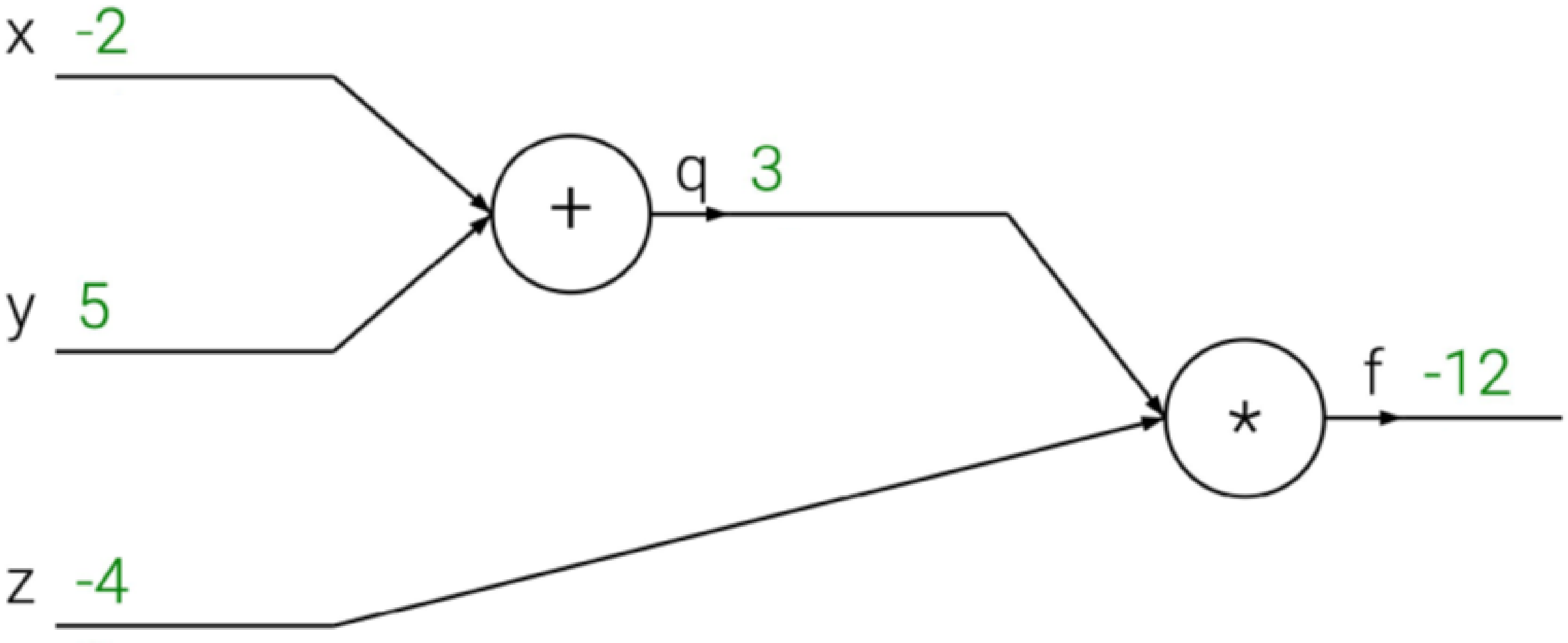


Chain Rule (6/9)



Chain Rule (7/9)

- Circuit intuition



Chain Rule (8/9)

- Circuit intuition

$$f(x, y, z) = (x + y)z$$

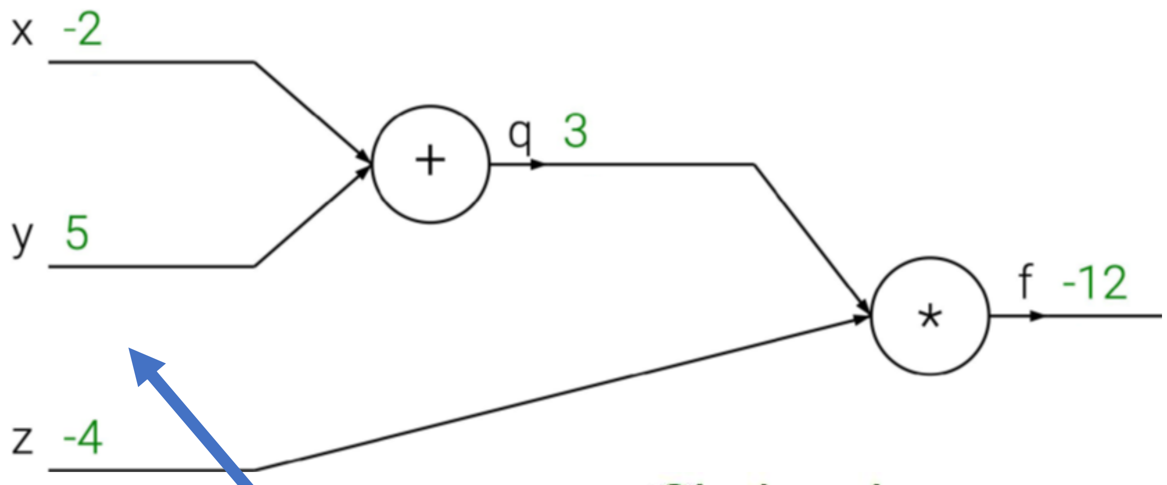
e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

Partial derivative of q with respect to y



$$\frac{\partial f}{\partial y}$$

Chain rule:

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial y}$$

Chain Rule (9/9)

- Circuit intuition

$$f(x, y, z) = (x + y)z$$

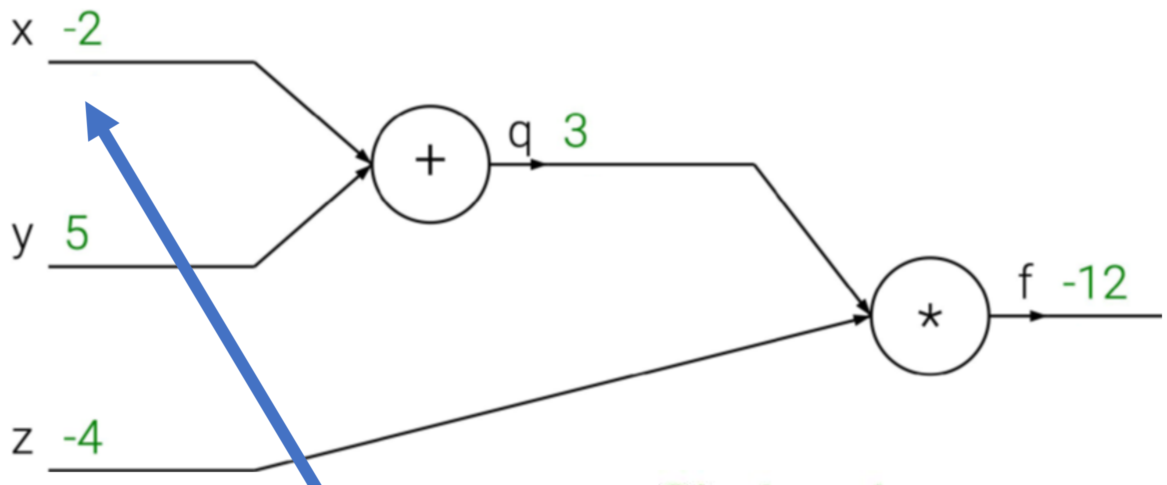
e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

Partial derivative of q with respect to y



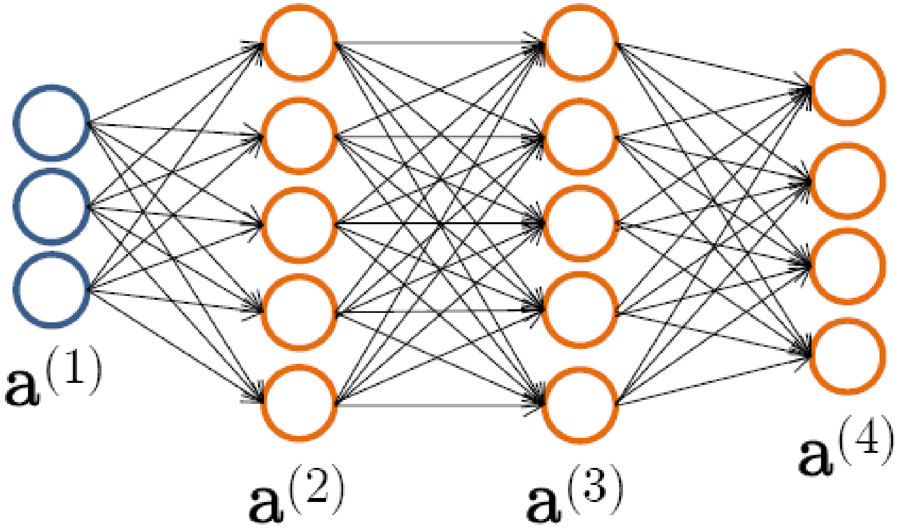
$$\frac{\partial f}{\partial x}$$

Chain rule:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x}$$

Forward Propagation

- Given one labeled training instance (**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 $\mathrm{a}_0{}^{(2)}$]
  - $\mathbf{z}^{(3)} = \Theta^{(2)}\mathbf{a}^{(2)}$
  - $\mathbf{a}^{(3)} = g(\mathbf{z}^{(3)})$ [add $\mathrm{a}_0{}^{(3)}$]
  - $\mathbf{z}^{(4)} = \Theta^{(3)}\mathbf{a}^{(3)}$
  - $\mathbf{a}^{(4)} = \mathrm{h}_\Theta(\mathbf{x}) = g(\mathbf{z}^{(4)})$

Learning in NN: Backpropagation

- 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
- The trick is to assess the blame for the error and divide it among the contributing weights

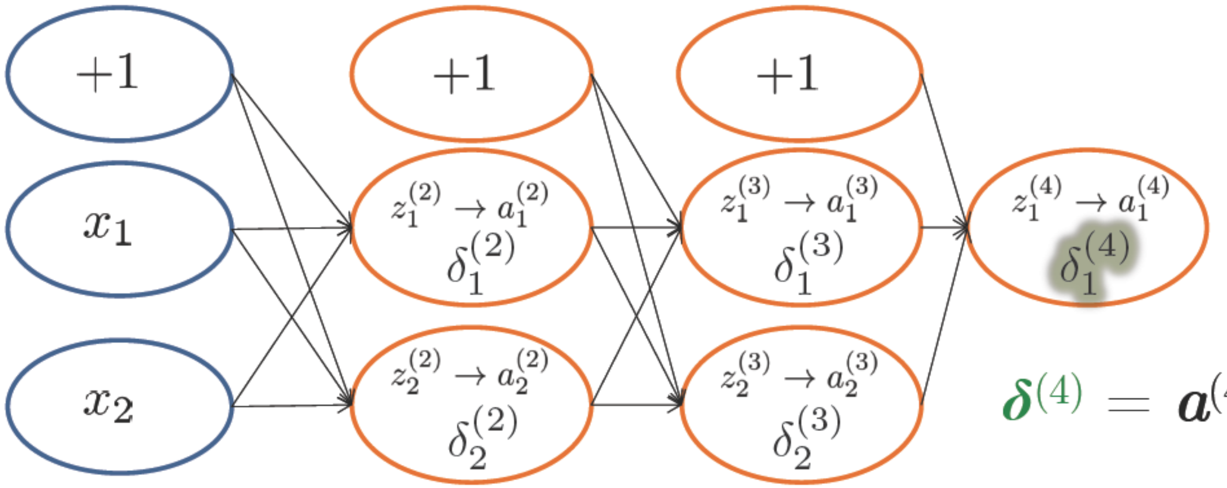
Backpropagation Intuition (1/5)

- Each hidden node j is responsible for some fraction of the error $\delta j^{(l)}$ in each of the output nodes to which it connects

- $\delta j^{(l)}$ is divided according to the strength of the connection between hidden node and the output node

- Then, the “blame” is propagated back to provide the error values for the hidden layer

Backpropagation Intuition (2/5)

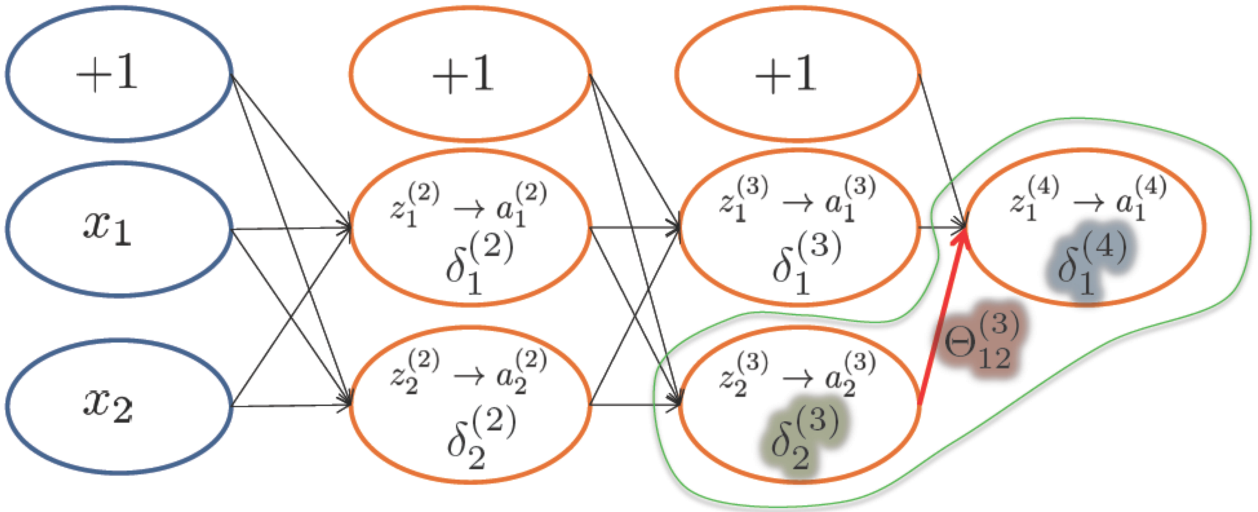


$\boldsymbol{\delta}^{(4)} = \boldsymbol{a}^{(4)} - \mathbf{y}$

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

Formally, $\delta_j^{(l)} = \dfrac{\partial}{\partial z_j^{(l)}} \text{cost}(\mathbf{x}_i) \quad$ 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 (3/5)

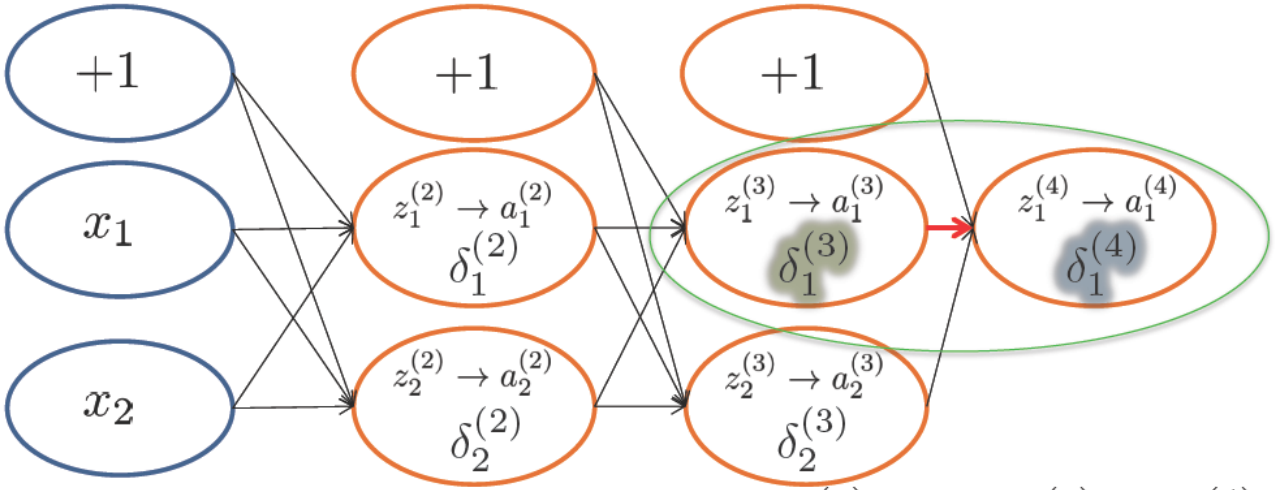


$$\delta_2{}^{(3)} = \Theta_{12}{}^{(3)} \times \delta_1{}^{(4)} \times g_2{}'^{(3)}$$

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

Formally, $\delta_j^{(l)} = \dfrac{\partial}{\partial z_j^{(l)}} \mathrm{cost}(\mathbf{x}_i)$ where $\mathrm{cost}(\mathbf{x}_i) = y_i \log h_\Theta(\mathbf{x}_i) + (1 - y_i) \log(1 - h_\Theta(\mathbf{x}_i))$

Backpropagation Intuition (4/5)



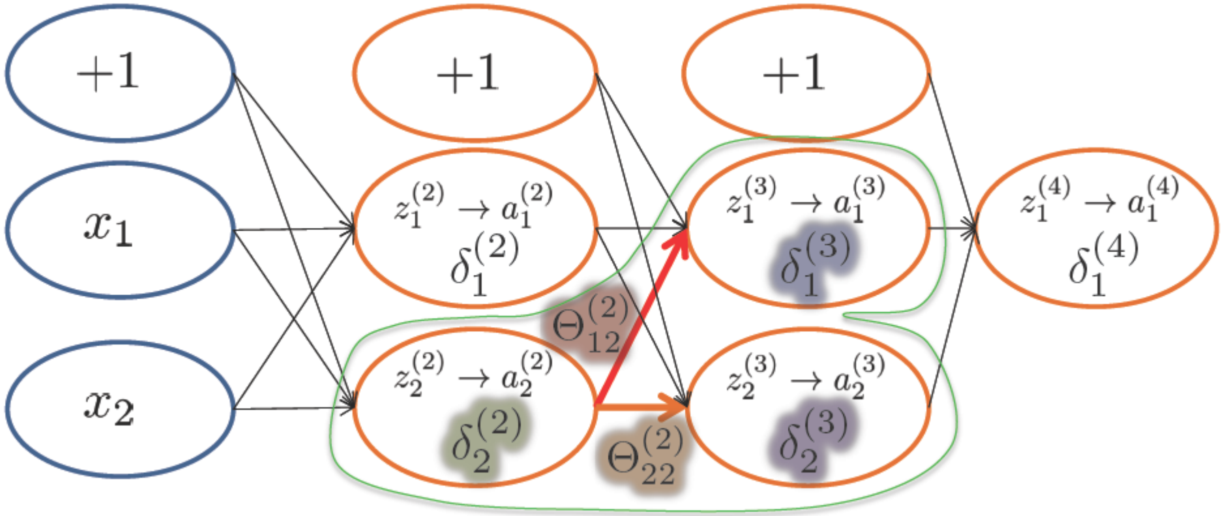
$$\delta_2^{(3)} = \Theta_{12}^{(3)} \times \delta_1^{(4)} \times g_2'^{(3)}$$

$$\delta_1^{(3)} = \Theta_{11}^{(3)} \times \delta_1^{(4)} \times g_1'^{(3)}$$

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

Formally, $\delta_j^{(l)} = \dfrac{\partial}{\partial z_j^{(l)}} \text{cost}(\mathbf{x}_i)$ 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 (5/5)



$$\delta_2^{(2)} = [\Theta_{12}^{(2)} \times \delta_1^{(3)} + \Theta_{22}^{(2)} \times \delta_2^{(3)}] \times g_2{}'^{(2)}$$

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

Formally, $\delta_j^{(l)} = \dfrac{\partial}{\partial z_j^{(l)}} \mathrm{cost}(\mathbf{x}_i) \quad \text{where } \mathrm{cost}(\mathbf{x}_i) = y_i \log h_\Theta(\mathbf{x}_i) + (1 - y_i) \log(1 - h_\Theta(\mathbf{x}_i))$

Backpropagation: Gradient Computation

- Let $\delta_j^{(l)}$ = "error" of node j in layer l

  (#layers $L = 4$)

Element-wise product .*

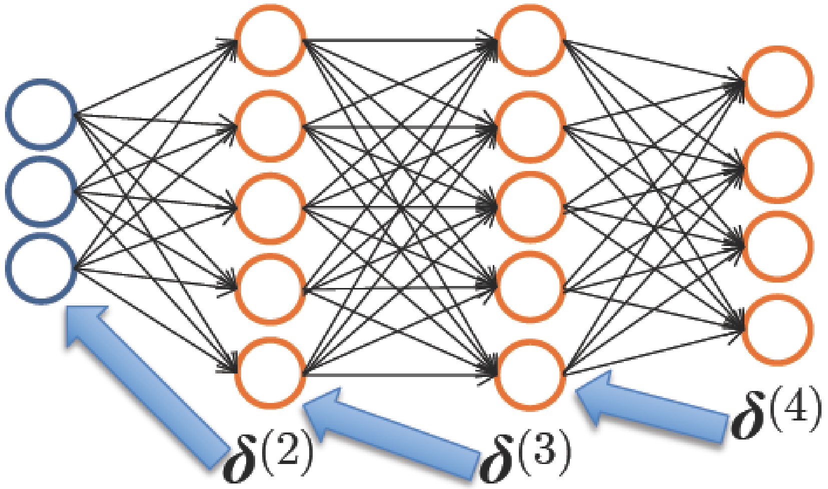
Backpropagation

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

$g'(\mathbf{z}^{(3)}) = \mathbf{a}^{(3)} \;.\!*\; (1 - \mathbf{a}^{(3)})$

$g'(\mathbf{z}^{(2)}) = \mathbf{a}^{(2)} \;.\!*\; (1 - \mathbf{a}^{(2)})$

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



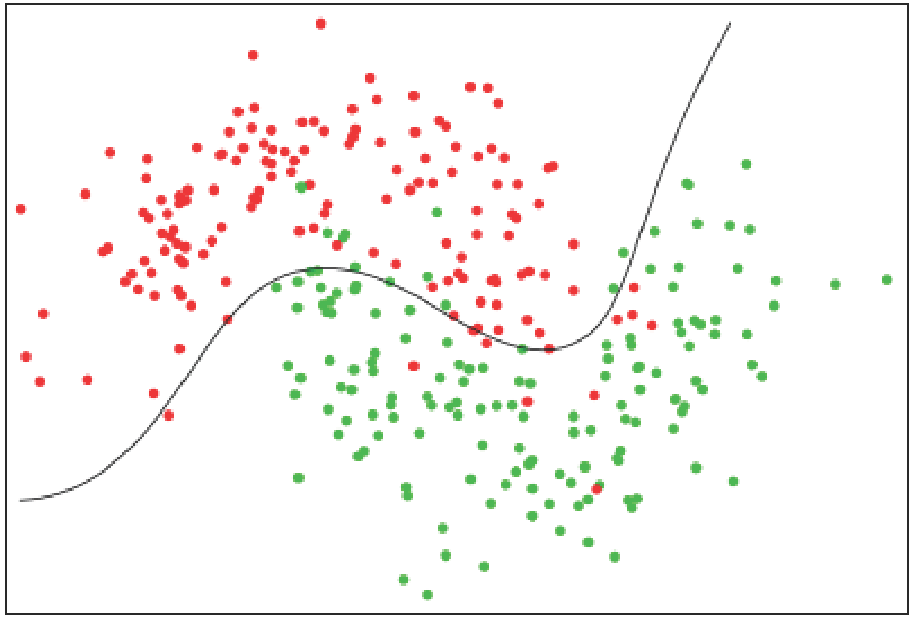
Example Implementation

- For the sake of better understanding, we’re not using vectorized form!
  - Initialize a network
  - Forward Propagate
  - Backward Propagate
- Let’s use the vectorized form now

Non-linear Decision Boundary

- Can apply basis function expansion to features, same as with linear regression

$$\boldsymbol{x} = \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix} \rightarrow \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_1 x_2 \\ x_1^2 \\ x_2^2 \\ x_1^2 x_2 \\ x_1 x_2^2 \\ \vdots \end{bmatrix}$$



Logistic Regression

- Given $\left\{ \left(\boldsymbol{x}^{(1)}, y^{(1)} \right), \left(\boldsymbol{x}^{(2)}, y^{(2)} \right), \ldots, \left(\boldsymbol{x}^{(n)}, y^{(n)} \right) \right\}$

  where $\boldsymbol{x}^{(i)} \in \mathbb{R}^d, \ y^{(i)} \in \{0, 1\}$

- Model:

$$h_{\boldsymbol{\theta}}(\boldsymbol{x}) = g\left(\boldsymbol{\theta}^{\mathsf{T}} \boldsymbol{x}\right)$$

$$g(z) = \frac{1}{1 + e^{-z}}$$

$$\boldsymbol{\theta} = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_d \end{bmatrix}$$

$$\boldsymbol{x}^{\mathsf{T}} = \begin{bmatrix} 1 & x_1 & \ldots & x_d \end{bmatrix}$$

Cost function

- Logistic regression

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

- Neural network

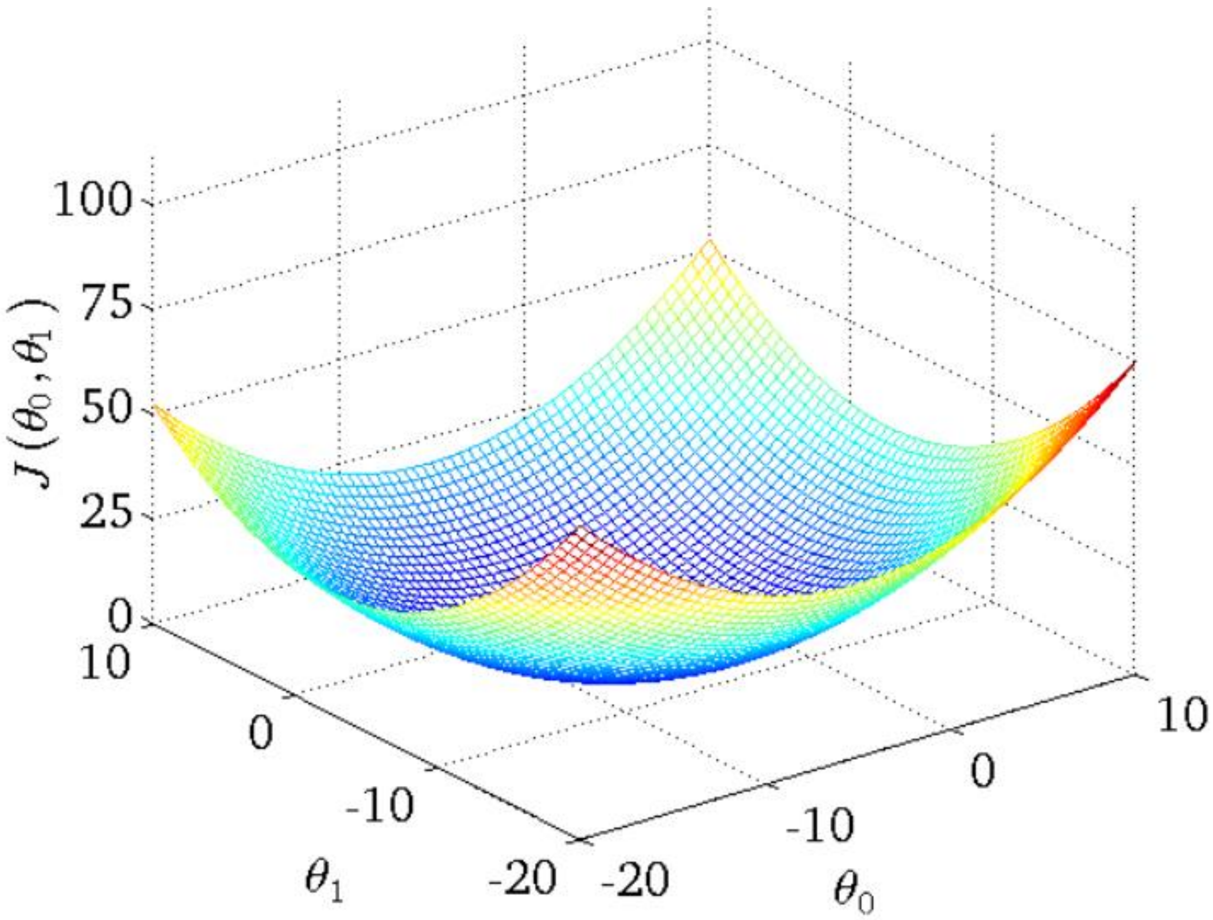
$$h_\Theta \in \mathbb{R}^K \qquad (h_\Theta(\mathbf{x}))_i = i^{th}\text{output}$$

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

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

K is the number of output units
L is the total number of layers in the network, and
sl is the number of units in layer l

Intuition Behind Cost Function



Optimizing the Neural Network

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

Solve via: $\min J(\Theta)$

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

- But, tends to work well in practice

Need code to compute: $J(\Theta)$

$$\frac{\partial}{\partial\Theta_{ij}^{(l)}}J(\Theta)$$

Gradient Descent for Logistic Regression

$$J_{\text{reg}}(\boldsymbol{\theta}) = -\sum_{i=1}^{n}\left[y^{(i)}\log h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) + \left(1-y^{(i)}\right)\log\left(1-h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)})\right)\right] + \frac{\lambda}{2}\|\boldsymbol{\theta}_{[1:d]}\|_2^2$$

Want $\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$

- Initialize $\boldsymbol{\theta}$
- Repeat until convergence (simultaneous update for $j = 0 \ldots d$)

$$\theta_0 \leftarrow \theta_0 - \alpha \sum_{i=1}^{n}\left(h_{\boldsymbol{\theta}}\left(\boldsymbol{x}^{(i)}\right) - y^{(i)}\right)$$

$$\theta_j \leftarrow \theta_j - \alpha \left[\sum_{i=1}^{n}\left(h_{\boldsymbol{\theta}}\left(\boldsymbol{x}^{(i)}\right) - y^{(i)}\right) x_j^{(i)} + \lambda\theta_j\right]$$

Example Implementation

- Gradient Descent (GD)

Backpropagation

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

For each training instance $(\mathbf{x}_k, y_k)$:

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

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

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

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

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

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

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

Note: Can vectorize $\Delta_{ij}^{(l)} = \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$ as $\boldsymbol{\Delta}^{(l)} = \boldsymbol{\Delta}^{(l)} + \boldsymbol{\delta}^{(l+1)} \mathbf{a}^{(l)\mathsf{T}}$

Training a Neural Network (1/2)

- Gradient descent with Backprop

Given: training set $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$
Initialize all $\Theta^{(l)}$ randomly (NOT to 0!)
Loop // each iteration is called an epoch
    Set $\Delta_{i,j}^{(l)} = 0 \quad \forall l, i, j$ (Used to accumulate gradient)
    For each training instance $(\mathbf{x}_k, y_k)$:
        Set $\mathbf{a}^{(1)} = \mathbf{x}_k$
        Compute $\{\mathbf{a}^{(2)}, \ldots, \mathbf{a}^{(L)}\}$ via forward propagation
        Compute $\boldsymbol{\delta}^{(L)} = \mathbf{a}^{(L)} - y_k$
        Compute errors $\{\boldsymbol{\delta}^{(L-1)}, \ldots, \boldsymbol{\delta}^{(2)}\}$
        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

Backpropagation

Training a Neural Network (2/2)

- Mini-batch gradient descent with Backprop

Given: training set $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$
Initialize all $\Theta^{(l)}$ randomly (NOT to 0!)
Loop // each iteration is called an epoch
    Loop // each iteration is a mini-batch
        Set $\Delta_{i,j}^{(l)} = 0 \quad \forall l, i, j$ (Used to accumulate gradient)
        Sample m training instances $\mathcal{X} = \{(\mathbf{x}'_1, y'_1), \ldots, (\mathbf{x}'_m, y'_m)\}$ without replacement
        For each instance in $\mathcal{X}$, $(\mathbf{x}_k, y_k)$:
            Set $\mathbf{a}^{(1)} = \mathbf{x}_k$
            Compute $\{\mathbf{a}^{(2)}, \ldots, \mathbf{a}^{(L)}\}$ via forward propagation
            Compute $\boldsymbol{\delta}^{(L)} = \mathbf{a}^{(L)} - y_k$
            Compute errors $\{\boldsymbol{\delta}^{(L-1)}, \ldots, \boldsymbol{\delta}^{(2)}\}$
            Compute gradients $\Delta_{ij}^{(l)} = \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$

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

        Update weights via gradient step $\Theta_{ij}^{(l)} = \Theta_{ij}^{(l)} - \alpha D_{ij}^{(l)}$
    Until all training instances are seen
Until weights converge or max #epochs is reached

Backpropagation

Summary

- Neural nets will be very large: impractical to write down gradient formula by hand for all parameters
- **backpropagation** = recursive application of the chain rule along a computational graph to compute the gradients of all inputs/parameters/intermediates
- Implementations maintain a graph structure, where the nodes implement the **forward**() / **backward**() API
- **forward:** compute result of an operation and save any intermediates needed for gradient computation in memory
- **backward:** apply the chain rule to compute the gradient of the loss function with respect to the inputs

Training a Neural Network

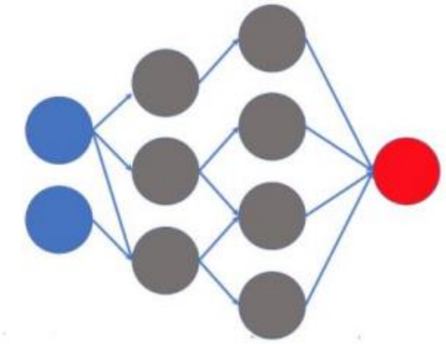
1. Randomly initialize weights
2. Implement forward propagation to get $h\Theta(\mathrm{x}i)$ for any instance $\mathrm{x}i$
3. Implement code to compute cost function $J(\Theta)$
4. Implement backprop to compute partial derivatives $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$
5. Use gradient checking to compare $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$ computed using backpropagation vs. the numerical gradient estimate.
   - Then, disable gradient checking code

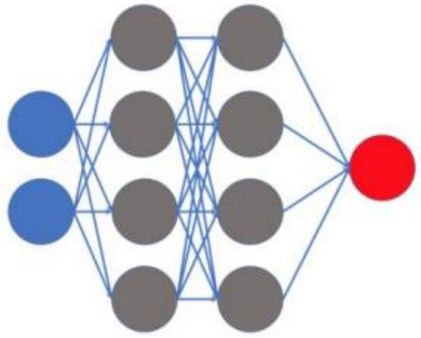
6. Use gradient descent with backprop to fit the network

Multilayer Perceptron

Multilayer Perceptron

- A multilayer perceptron is a special case of a feedforward neural network, where every layer is a fully connected layer
- In some definitions, the number of nodes in each layer is the same
- In many definitions the activation function across hidden layers is the dame


Feed-forward NN


Multilayer perceptron

More Example Implementations

- Make Predictions
- Multi-class Classification