

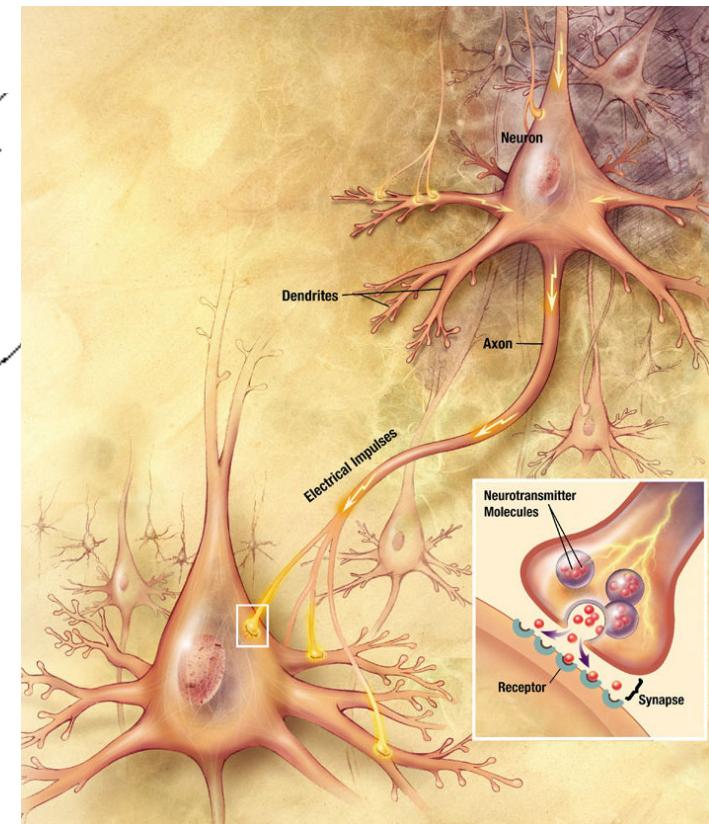
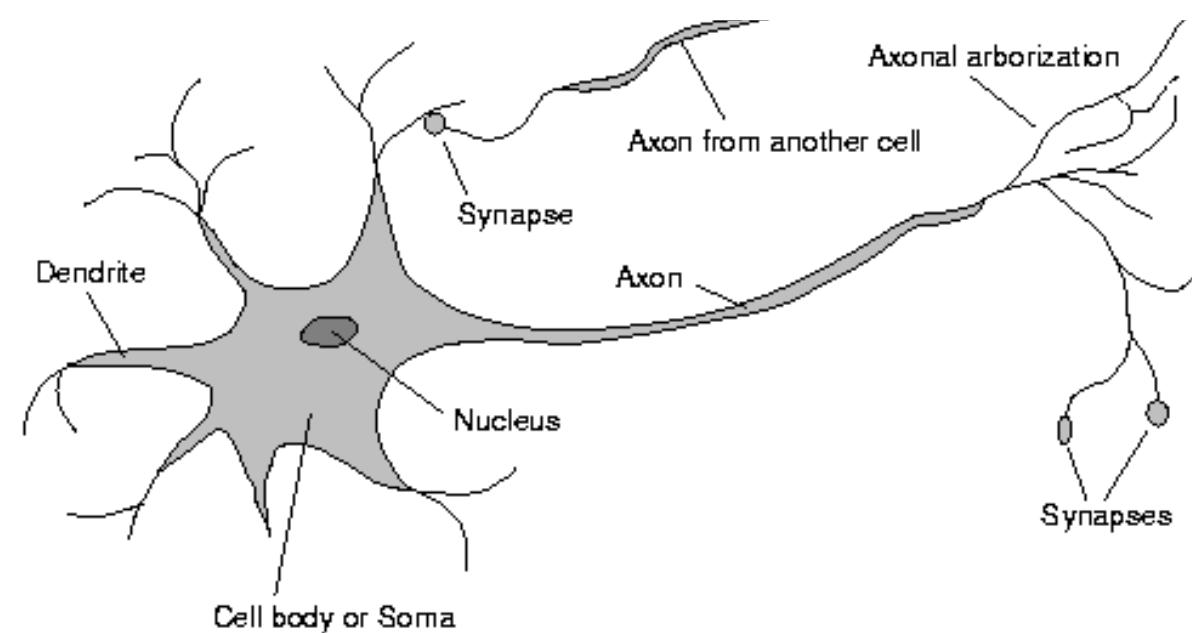


Neural Networks

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!

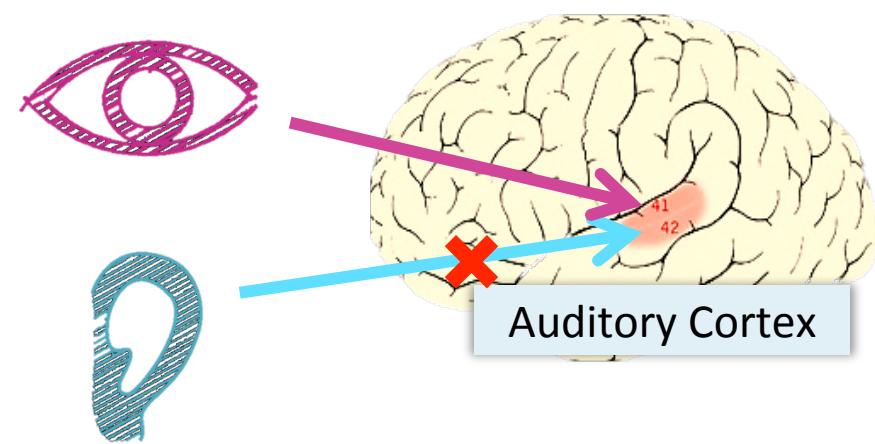
Biology of a Neuron



Brain Structure

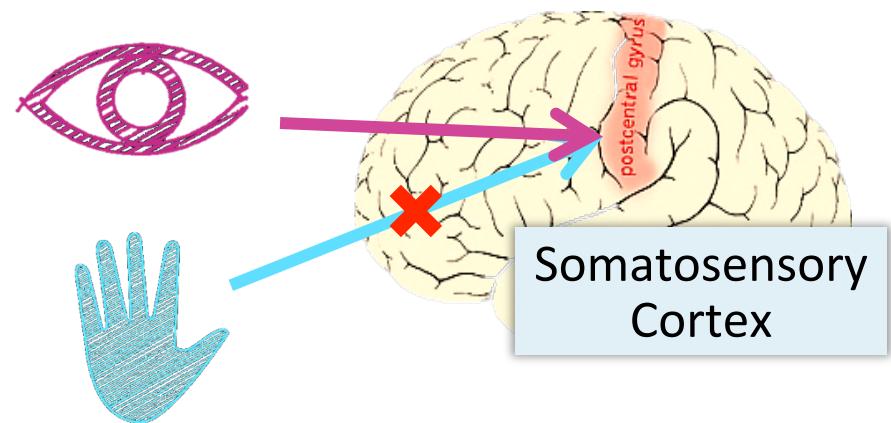
- Different areas of the brain have different functions
 - Some areas seem to have the same function in all humans (e.g., Broca's region for motor speech); the overall layout is generally consistent
 - Some areas are more plastic, and vary in their function; also, the lower-level structure and function 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)
- We *really* don't understand how this neural structure leads to what we perceive as “consciousness” or “thought”

The “One Learning Algorithm” Hypothesis



Auditory cortex learns to see

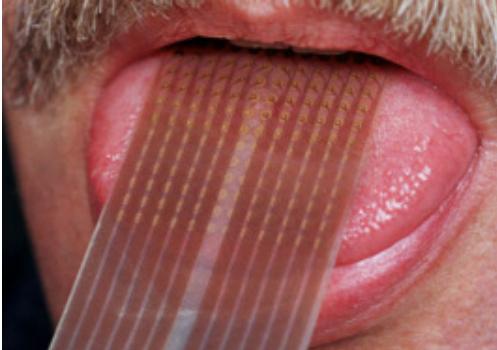
[Roe et al., 1992]



Somatosensory cortex
learns to see

[Metin & Frost, 1989]

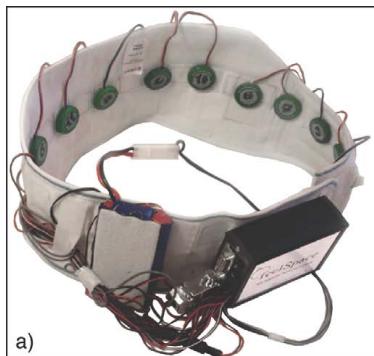
Sensor Representations in the Brain



Seeing with your tongue



Human echolocation (sonar)



Haptic belt: Direction sense



Implanting a 3rd eye

[BrainPort; Welsh & Blasch, 1997; Nagel et al., 2005; Constantine-Paton & Law, 2009]

Comparison of computing power

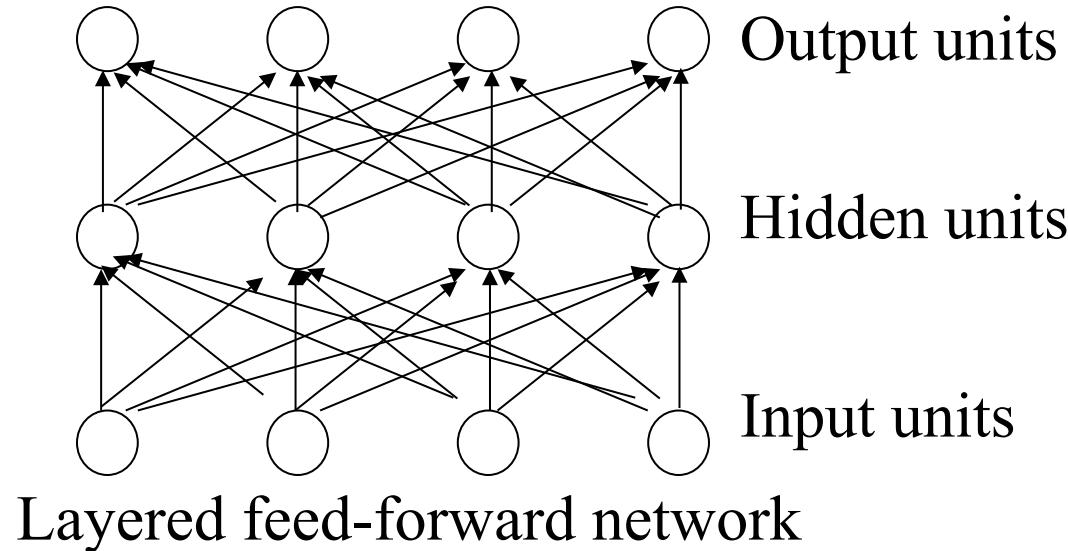
INFORMATION CIRCA 2012	Computer	Human Brain
Computation Units	10-core Xeon: 10^9 Gates	10^{11} Neurons
Storage Units	10^9 bits RAM, 10^{12} bits disk	10^{11} neurons, 10^{14} synapses
Cycle time	10^{-9} sec	10^{-3} sec
Bandwidth	10^9 bits/sec	10^{14} bits/sec

- 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

Neural Networks

- 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

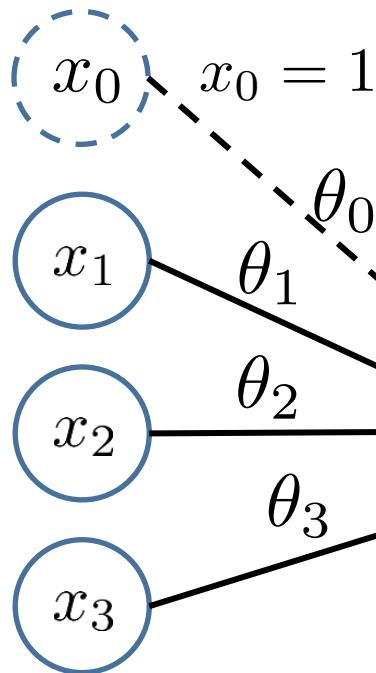
Neural networks



- 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**

Neuron Model: Logistic Unit

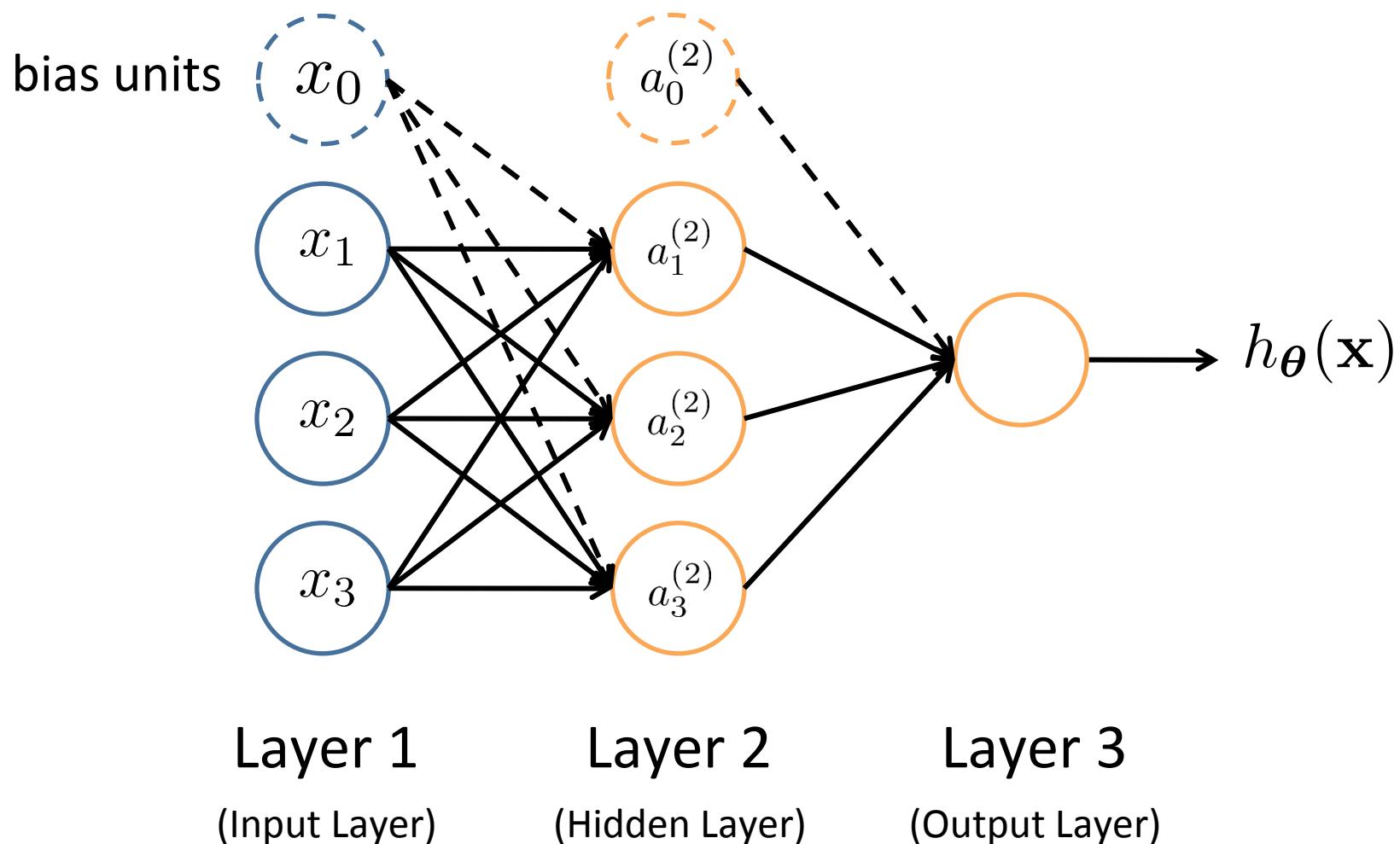
“bias unit”



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

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

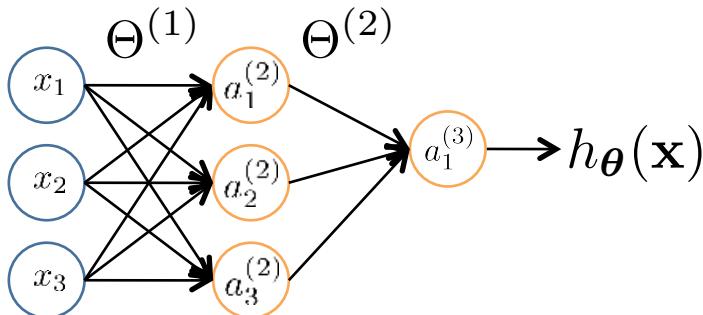
Neural Network



Feed-Forward Process

- 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
- 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 $s_{j+1} \times (s_j + 1)$.

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

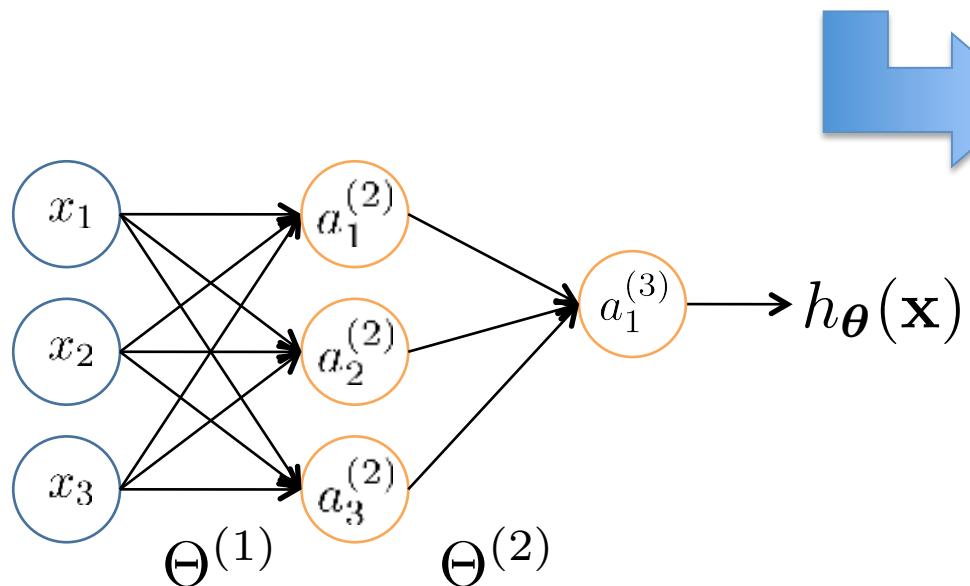
Vectorization

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

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

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

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



Feed-Forward Steps:

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

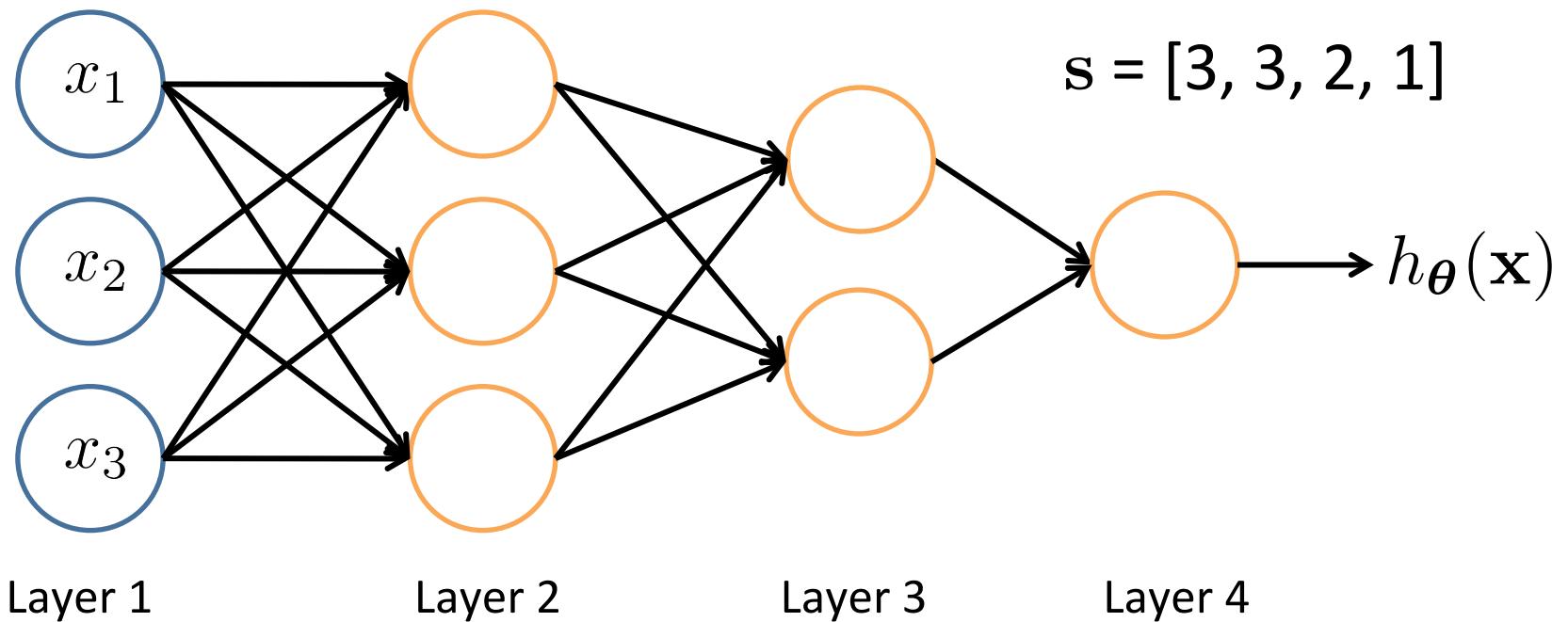
$$\mathbf{a}^{(2)} = g(\mathbf{z}^{(2)})$$

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

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

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

Other Network Architectures



L denotes the number of layers

$s \in \mathbb{N}^+^L$ 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



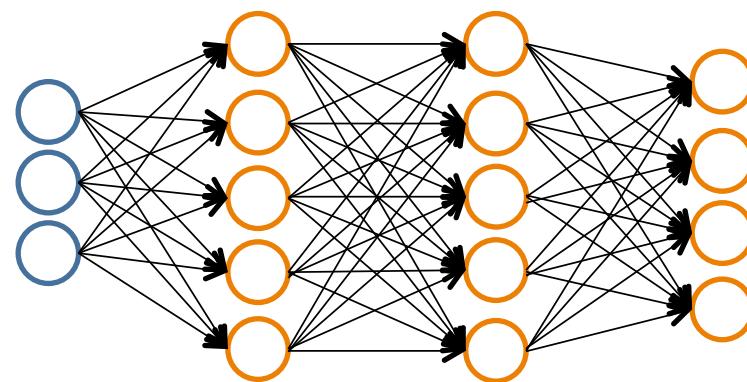
Car



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

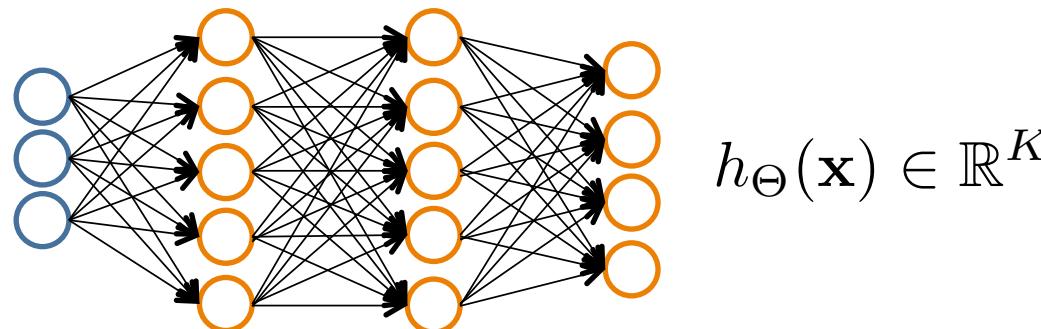
$$h_{\Theta}(\mathbf{x}) \approx \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

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:

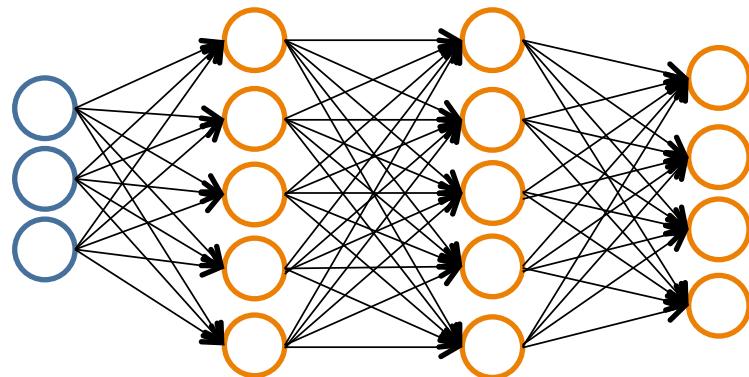
$$h_{\Theta}(\mathbf{x}) \approx \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad h_{\Theta}(\mathbf{x}) \approx \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad h_{\Theta}(\mathbf{x}) \approx \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \quad h_{\Theta}(\mathbf{x}) \approx \begin{bmatrix}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

when pedestrian when car when motorcycle 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 ($s_{L-1} = K$)

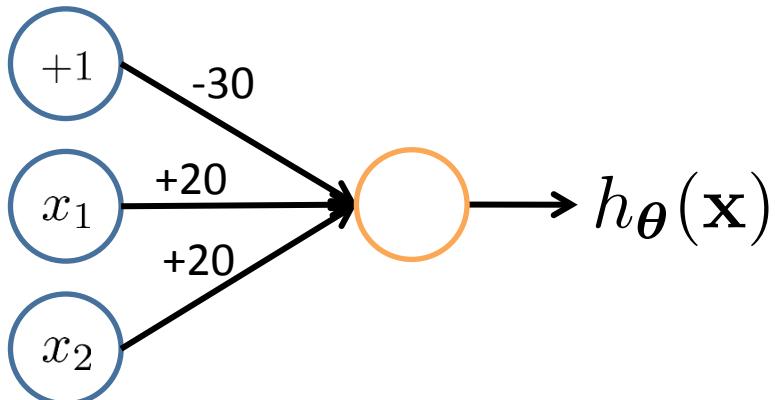
Understanding Representations

Representing Boolean Functions

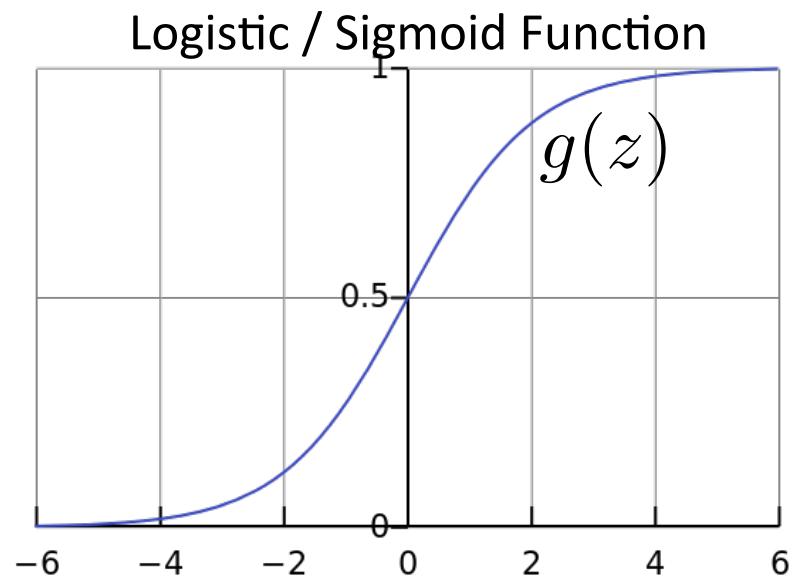
Simple example: AND

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

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

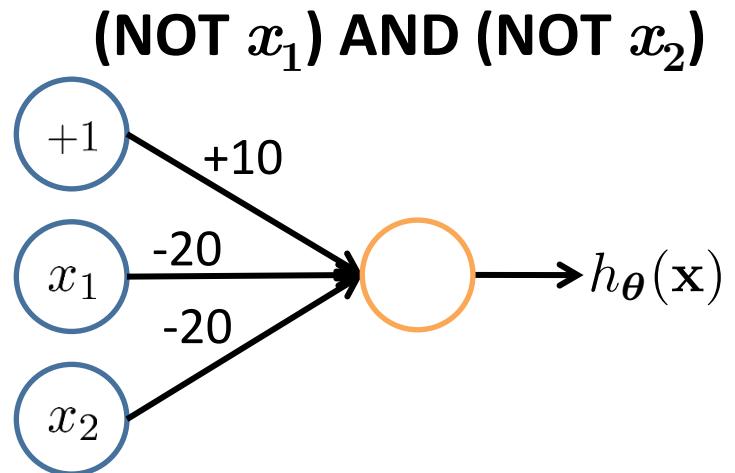
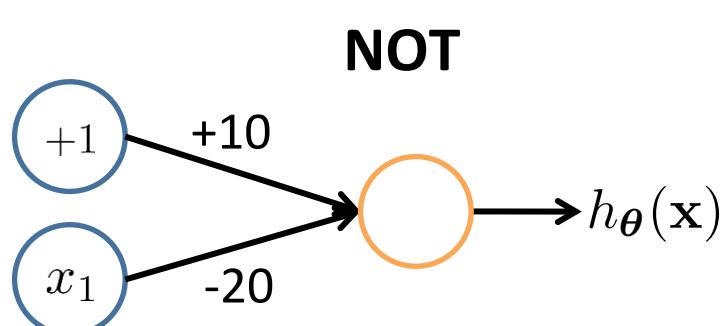
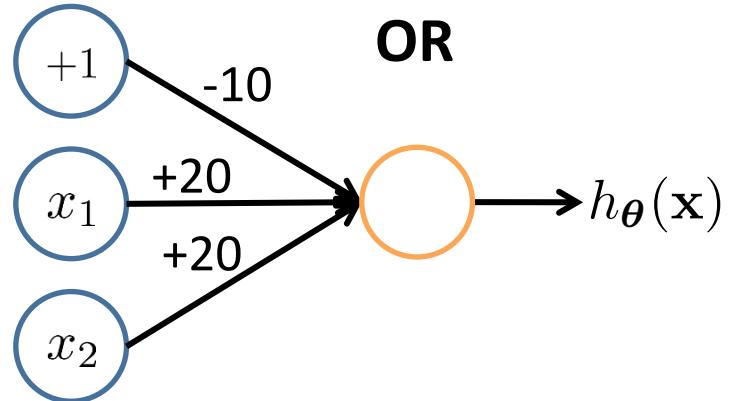
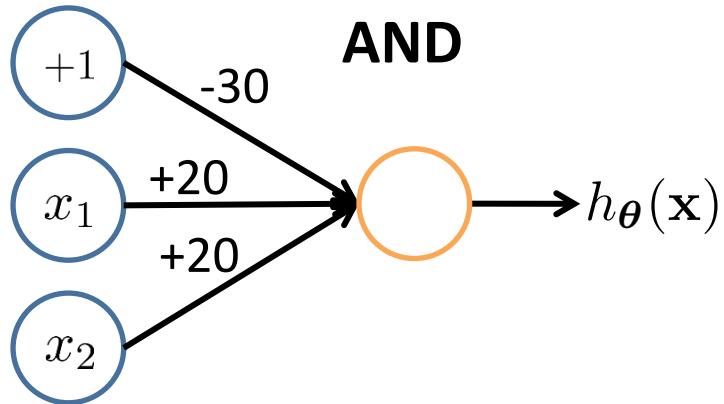


$$h_{\theta}(\mathbf{x}) = g(-30 + 20x_1 + 20x_2)$$

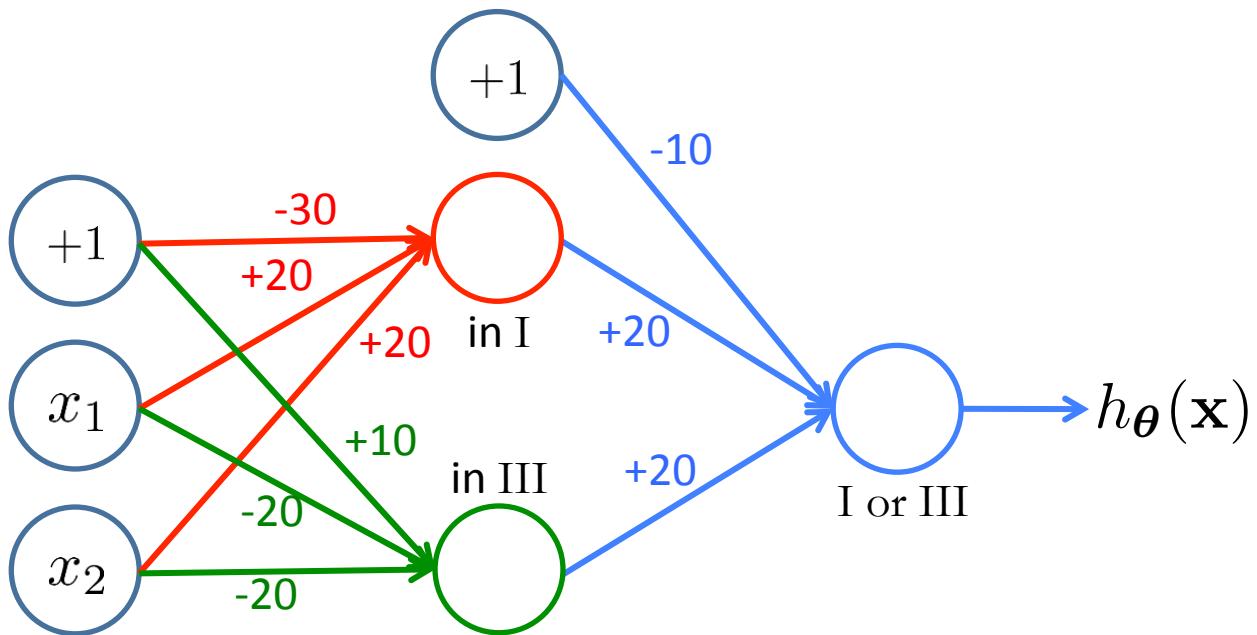
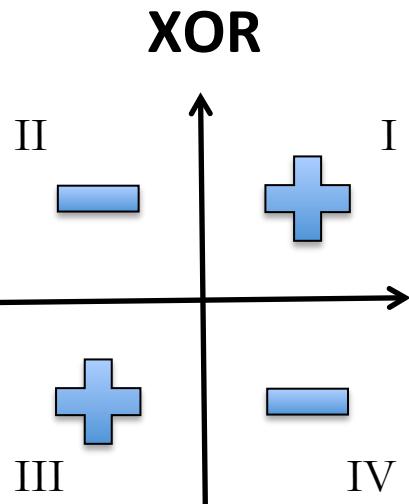
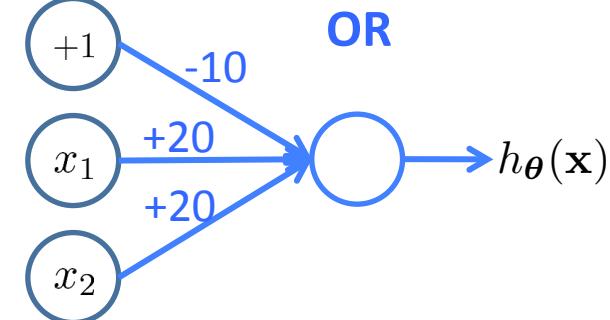
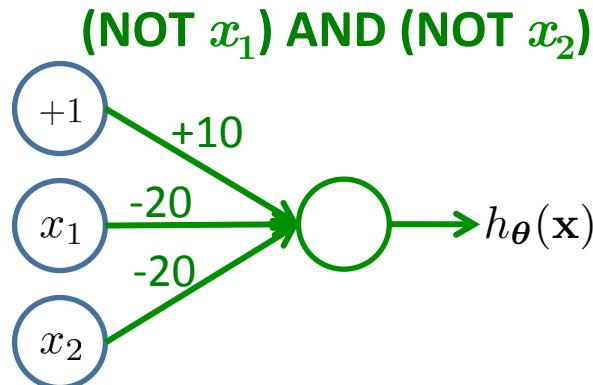
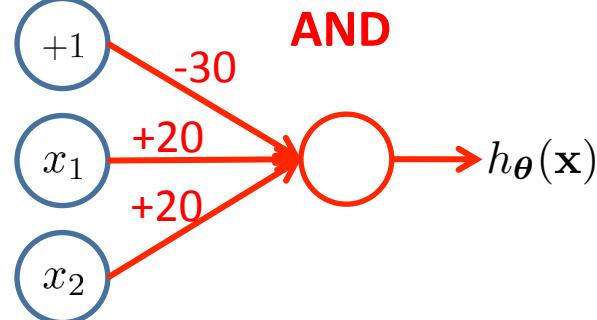


x_1	x_2	$h_{\theta}(\mathbf{x})$
0	0	$g(-30) \approx 0$
0	1	$g(-10) \approx 0$
1	0	$g(-10) \approx 0$
1	1	$g(10) \approx 1$

Representing Boolean Functions



Combining Representations to Create Non-Linear Functions

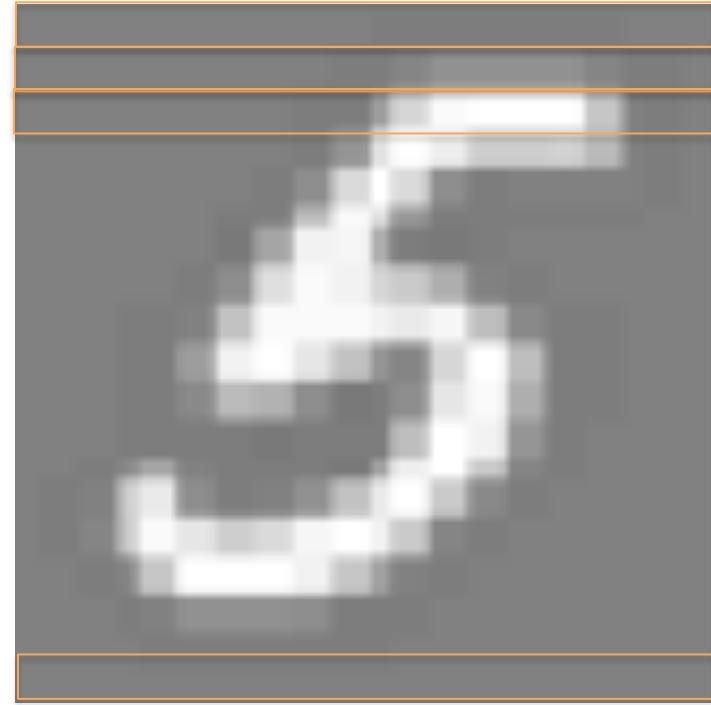


Layering Representations



20×20 pixel images

$d = 400$ 10 classes

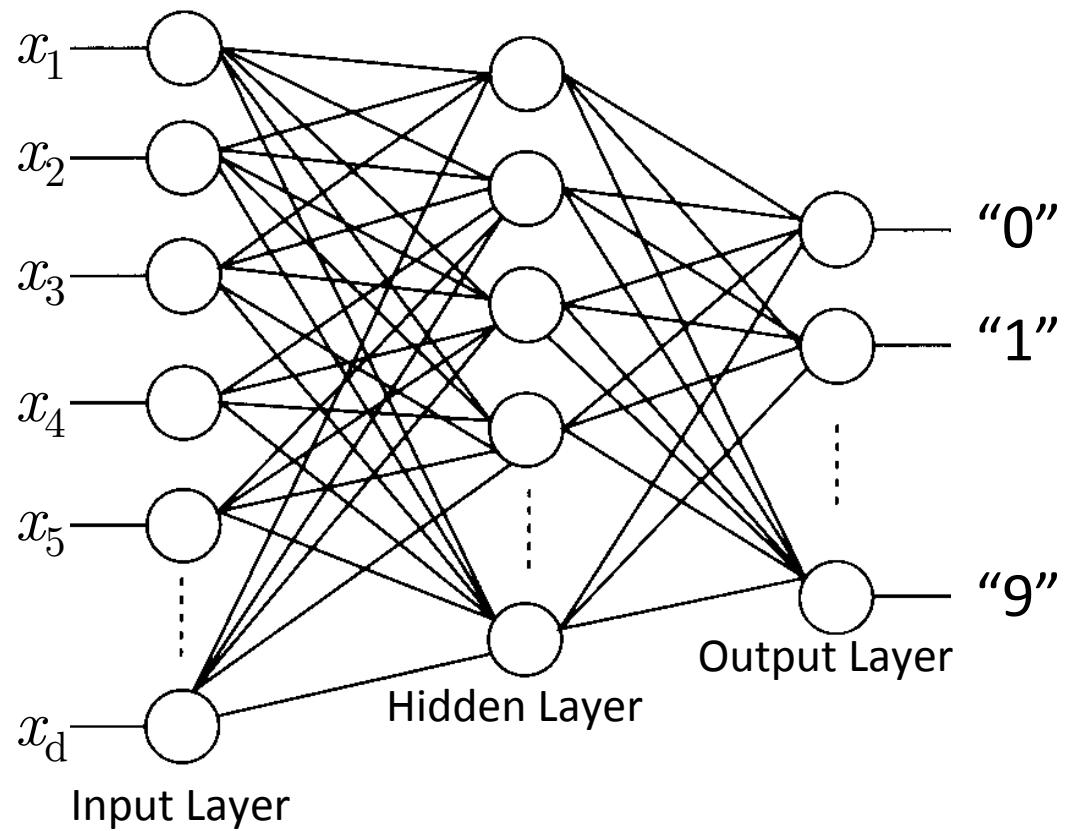
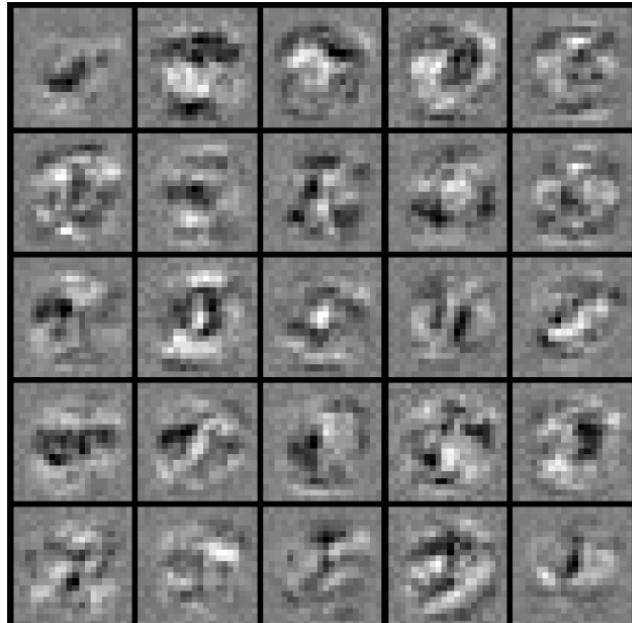


$x_1 \dots x_{20}$
 $x_{21} \dots x_{40}$
 $x_{41} \dots x_{60}$
⋮
 $x_{381} \dots x_{400}$

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

Layering Representations

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



Visualization of
Hidden Layer

LeNet 5 Demonstration: <http://yann.lecun.com/exdb/lenet/>



Neural Network Learning

Perceptron Learning Rule

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

Equivalent to the intuitive rules:

- 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 $\Delta \leftarrow [0, 0, \dots, 0]$

 for $i = 1 \dots n$, do

 if $y^{(i)} \mathbf{x}^{(i)} \theta \leq 0$ // prediction for i^{th} instance is incorrect

$\Delta \leftarrow \Delta + y^{(i)} \mathbf{x}^{(i)}$

$\Delta \leftarrow \Delta / n$ // 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
- The trick is to assess the blame for the error and divide it among the contributing weights

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:

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

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

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

Optimizing the Neural Network

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

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

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

- But, tends to work well in practice

Need code to compute:

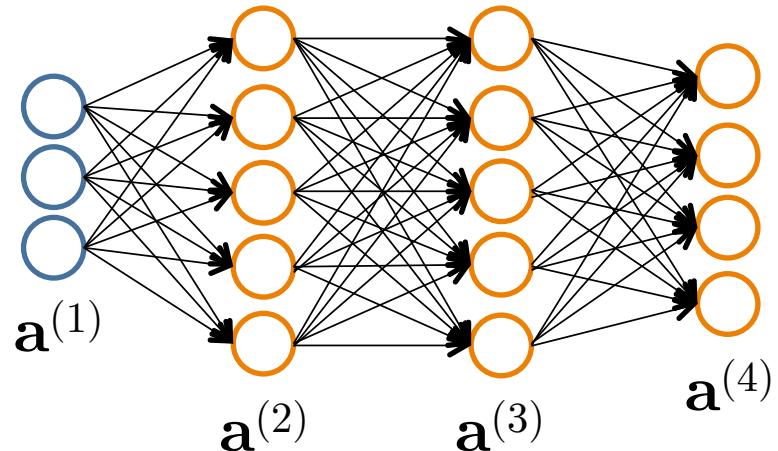
- $J(\Theta)$
- $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$

Forward Propagation

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

Forward Propagation

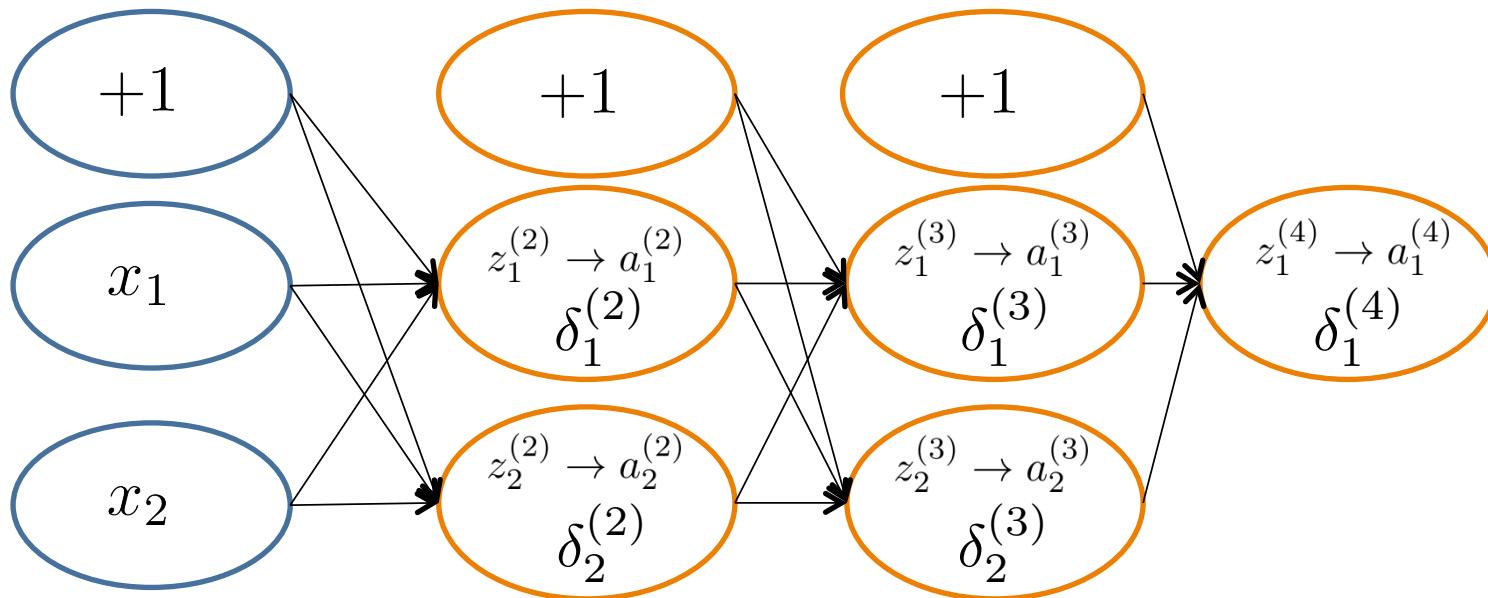
- $\mathbf{a}^{(1)} = \mathbf{x}$
- $\mathbf{z}^{(2)} = \Theta^{(1)}\mathbf{a}^{(1)}$
- $\mathbf{a}^{(2)} = g(\mathbf{z}^{(2)})$ [add $a_0^{(2)}$]
- $\mathbf{z}^{(3)} = \Theta^{(2)}\mathbf{a}^{(2)}$
- $\mathbf{a}^{(3)} = g(\mathbf{z}^{(3)})$ [add $a_0^{(3)}$]
- $\mathbf{z}^{(4)} = \Theta^{(3)}\mathbf{a}^{(3)}$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



Backpropagation Intuition

- 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

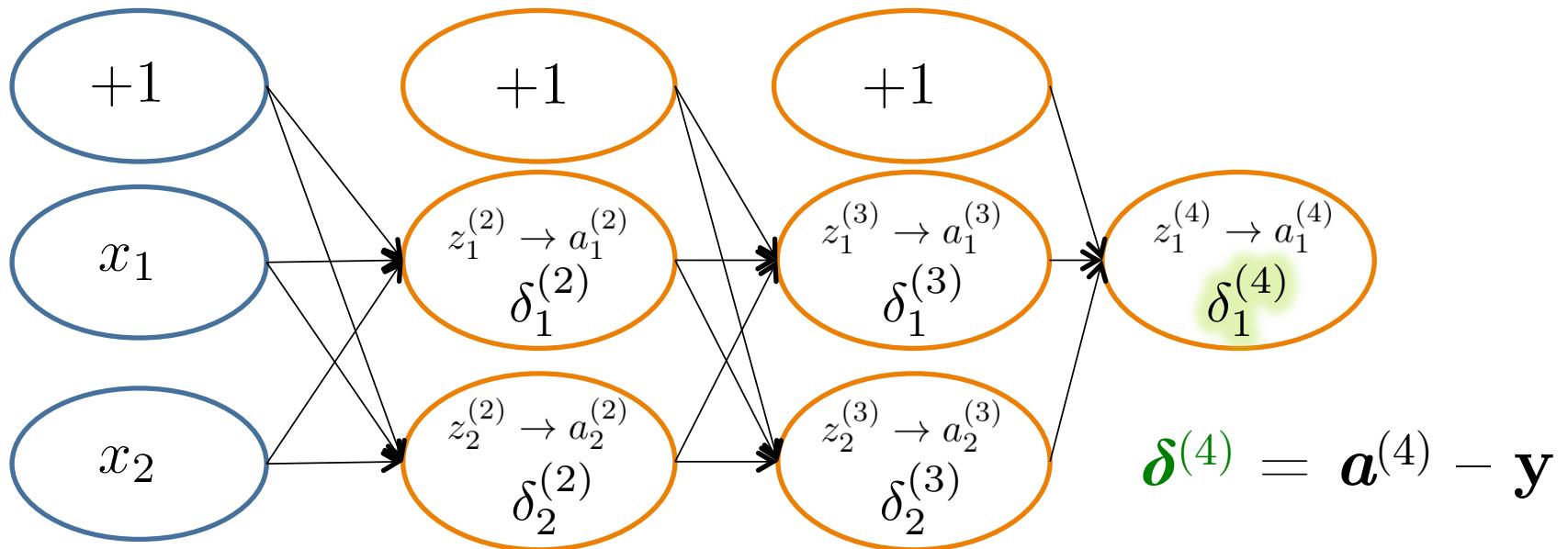


$\delta_j^{(l)}$ = “error” of node j in layer l

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

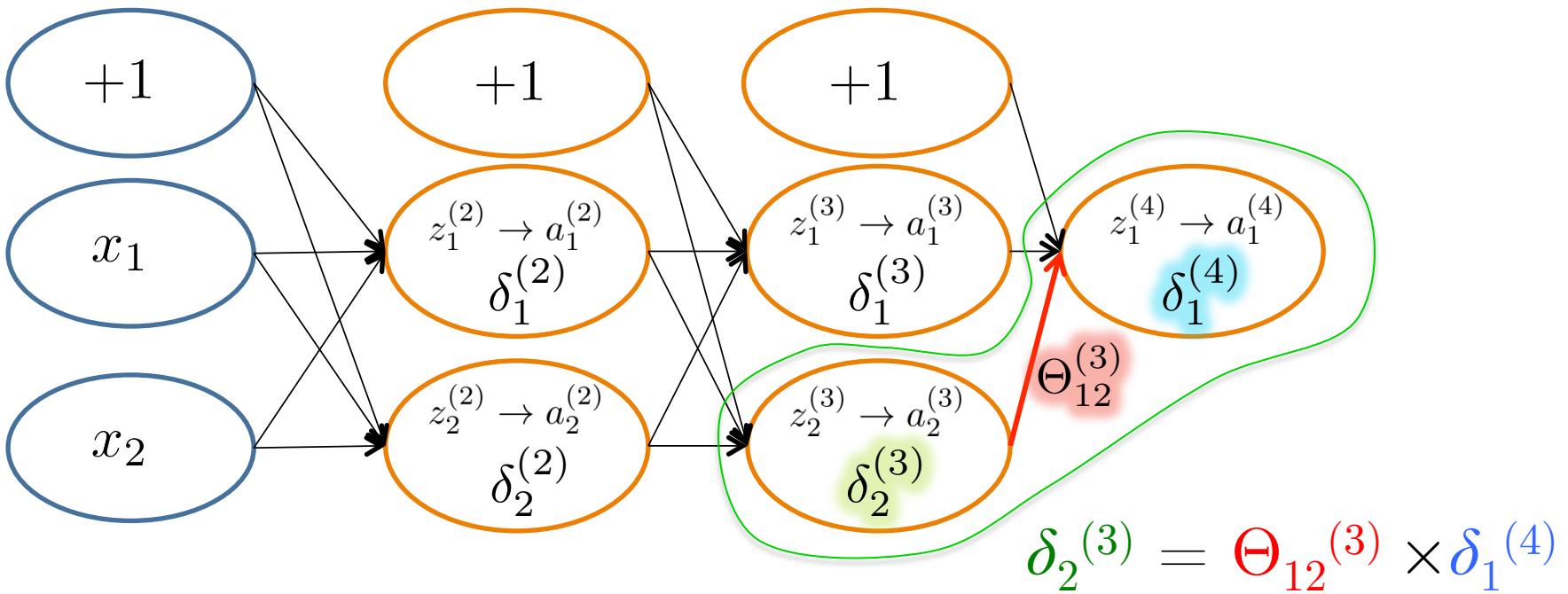


$\delta_j^{(l)}$ = “error” of node j in layer l

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 Intuition

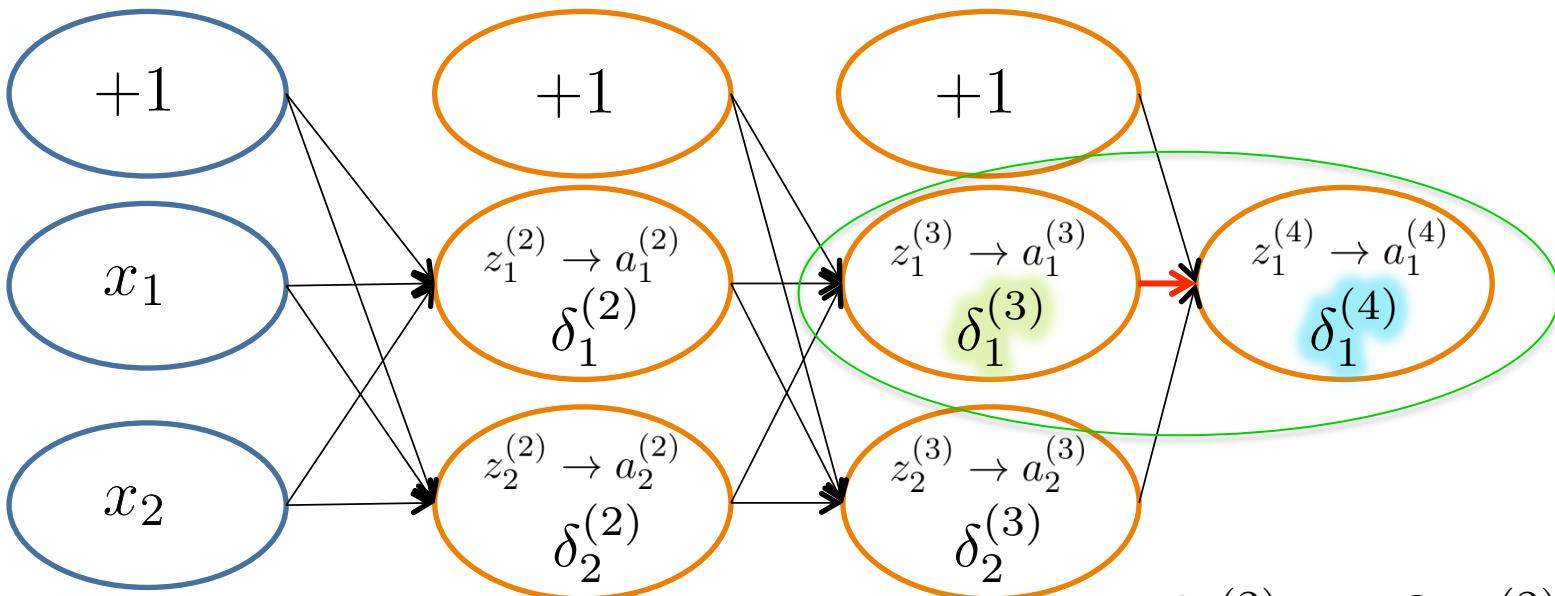


$\delta_j^{(l)}$ = “error” of node j in layer l

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



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

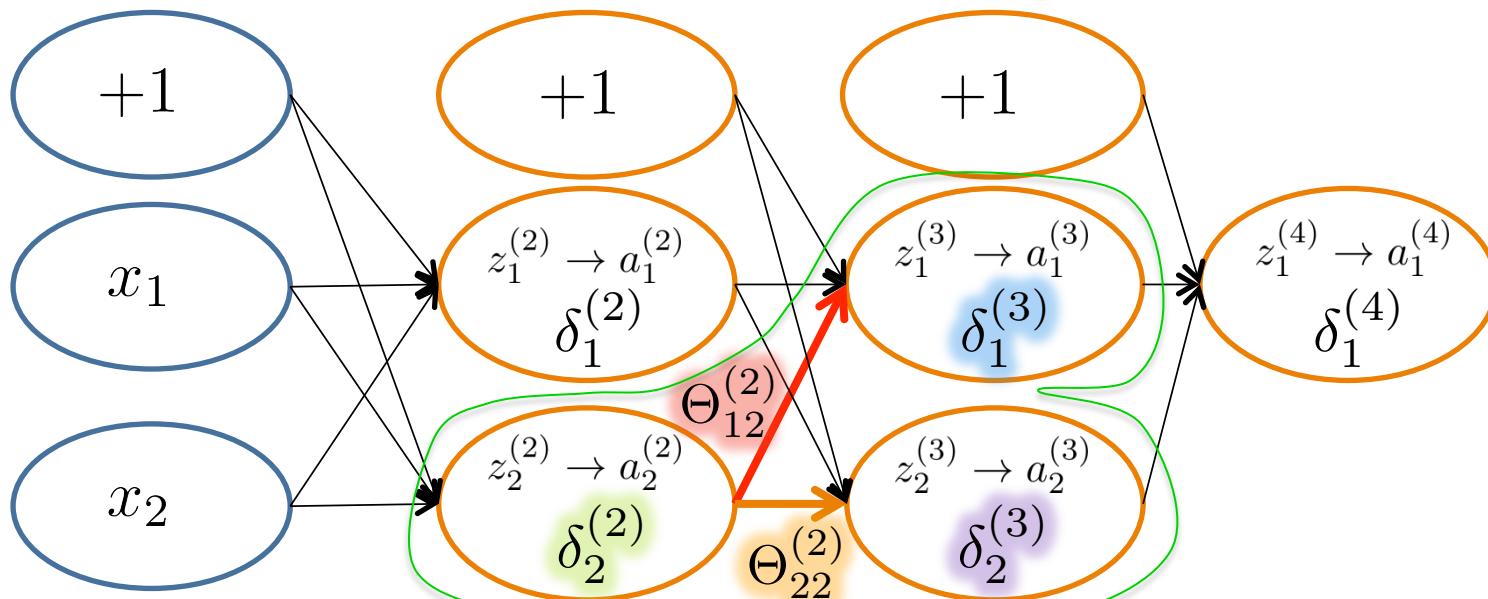
$$\delta_1^{(3)} = \Theta_{11}^{(3)} \times \delta_1^{(4)}$$

$\delta_j^{(l)}$ = “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



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

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

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

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

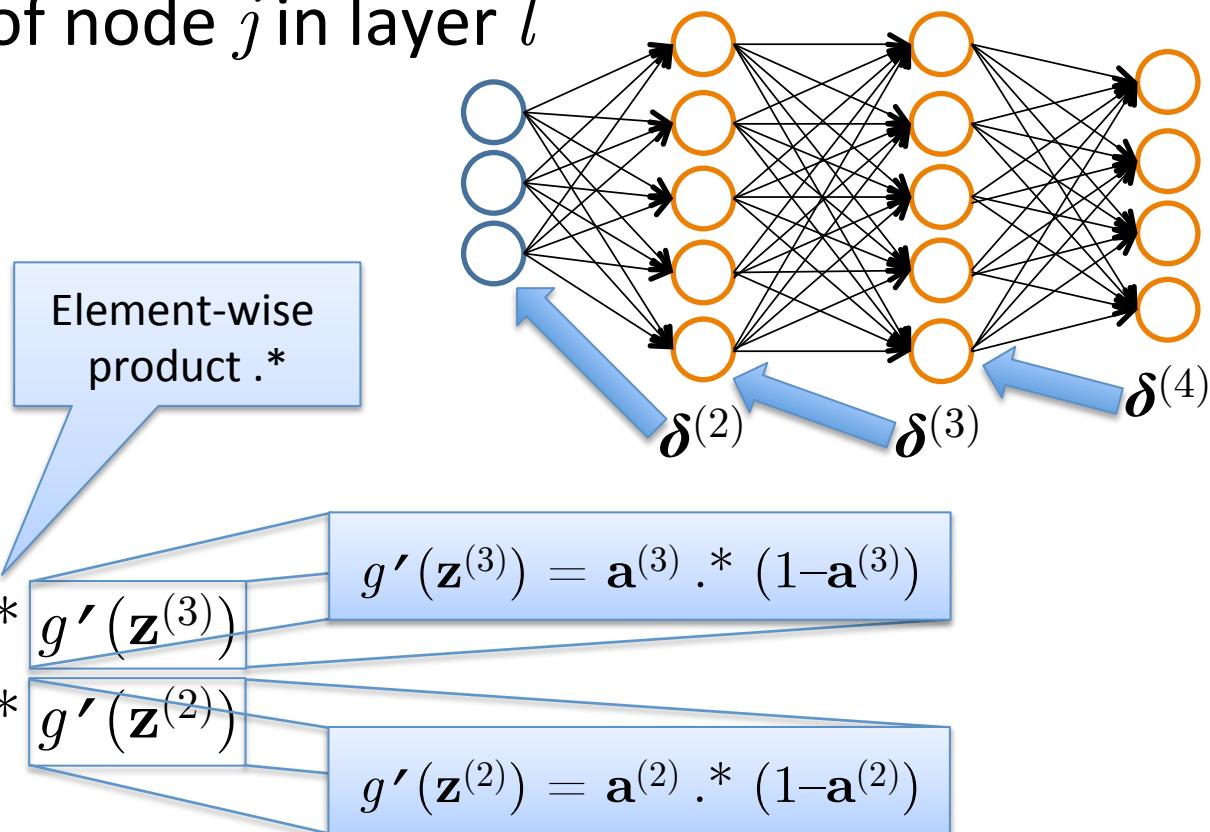
Backpropagation: Gradient Computation

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

(#layers $L = 4$)

Backpropagation

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



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

Backpropagation

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

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

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 $\Delta^{(l)} = \Delta^{(l)} + \boldsymbol{\delta}^{(l+1)} \mathbf{a}^{(l)\top}$

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

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

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

Backprop Issues

“Backprop is the cockroach of machine learning. It’s ugly, and annoying, but you just can’t get rid of it.”

-Geoff Hinton

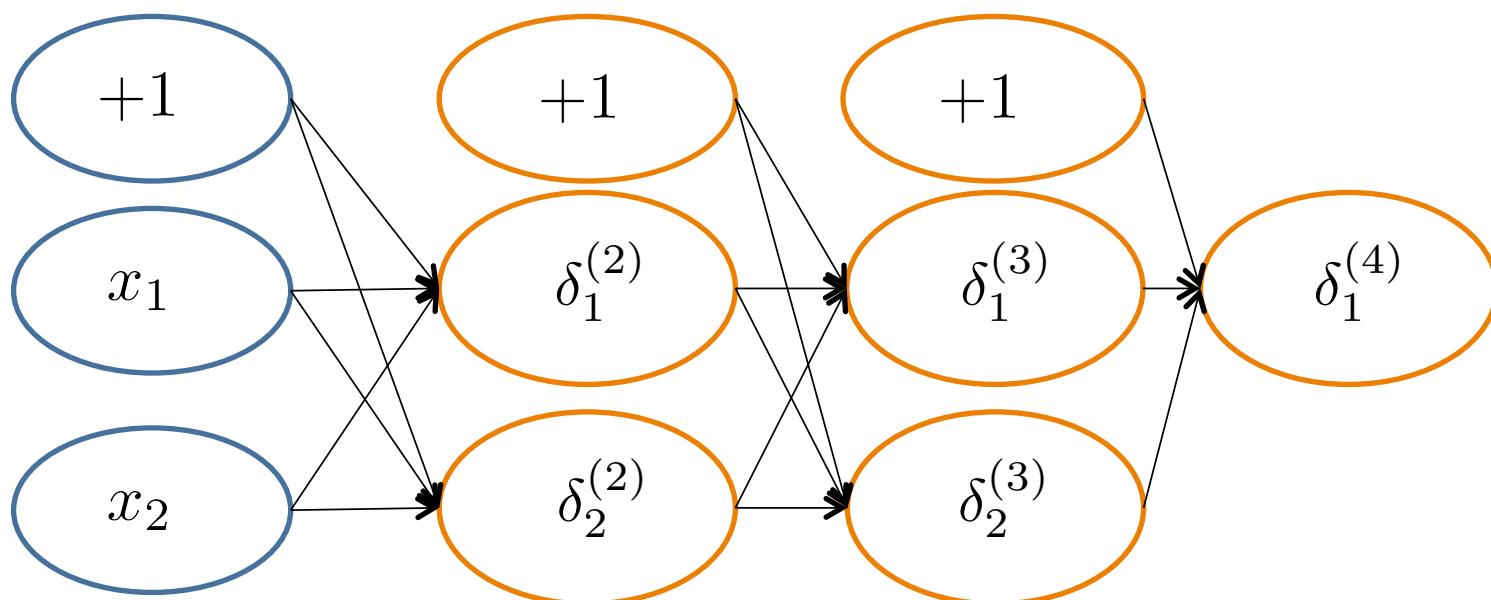
Problems:

- black box
- local minima

Implementation Details

Random Initialization

- Important to randomize initial weight matrices
- Can't have uniform initial weights, as in logistic regression
 - Otherwise, all updates will be identical & the net won't learn



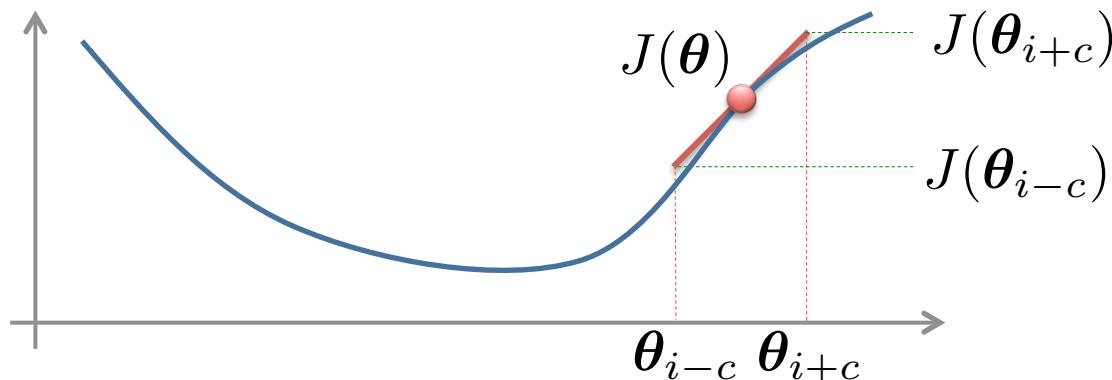
Implementation Details

- For convenience, compress all parameters into θ
 - “unroll” $\Theta^{(1)}, \Theta^{(2)}, \dots, \Theta^{(L-1)}$ into one long vector θ
 - E.g., if $\Theta^{(1)}$ is 10×10 , then the first 100 entries of θ contain the value in $\Theta^{(1)}$
 - Use the `reshape` command to recover the original matrices
 - E.g., if $\Theta^{(1)}$ is 10×10 , then

```
theta1 = reshape(theta[0:100], (10, 10))
```
- Each step, check to make sure that $J(\theta)$ decreases
- Implement a gradient-checking procedure to ensure that the gradient is correct...

Gradient Checking

Idea: estimate gradient numerically to verify implementation, then turn off gradient checking



$$\frac{\partial}{\partial \theta_i} J(\theta) \approx \frac{J(\theta_{i+c}) - J(\theta_{i-c})}{2c} \quad c \approx 1E-4$$

$$\theta_{i+c} = [\theta_1, \theta_2, \dots, \theta_{i-1}, \theta_i + c, \theta_{i+1}, \dots]$$

Change ONLY the i^{th} entry in θ , increasing (or decreasing) it by c

Gradient Checking

$\theta \in \mathbb{R}^m$ θ is an “unrolled” version of $\Theta^{(1)}, \Theta^{(2)}, \dots$

$$\theta = [\theta_1, \theta_2, \theta_3, \dots, \theta_m]$$

Put in vector called `gradApprox`

$$\frac{\partial}{\partial \theta_1} J(\theta) \approx \frac{J([\theta_1 + c, \theta_2, \theta_3, \dots, \theta_m]) - J([\theta_1 - c, \theta_2, \theta_3, \dots, \theta_m])}{2c}$$
$$\frac{\partial}{\partial \theta_2} J(\theta) \approx \frac{J([\theta_1, \theta_2 + c, \theta_3, \dots, \theta_m]) - J([\theta_1, \theta_2 - c, \theta_3, \dots, \theta_m])}{2c}$$
$$\vdots$$
$$\frac{\partial}{\partial \theta_m} J(\theta) \approx \frac{J([\theta_1, \theta_2, \theta_3, \dots, \theta_m + c]) - J([\theta_1, \theta_2, \theta_3, \dots, \theta_m - c])}{2c}$$

Check that the approximate numerical gradient matches the entries in the D matrices

Implementation Steps

- Implement backprop to compute DVec
 - DVec is the unrolled $\{D^{(1)}, D^{(2)}, \dots\}$ matrices
- Implement numerical gradient checking to compute gradApprox
- Make sure DVec has similar values to gradApprox
- Turn off gradient checking. Using backprop code for learning.

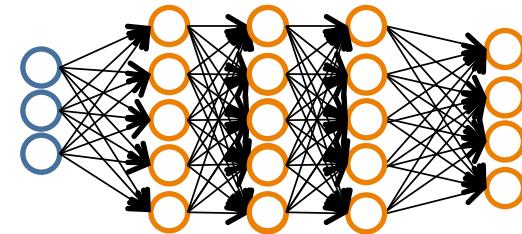
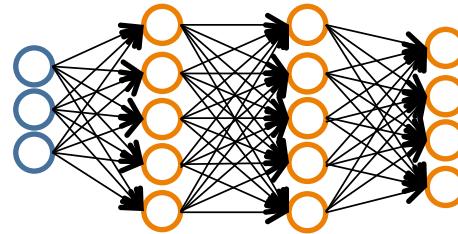
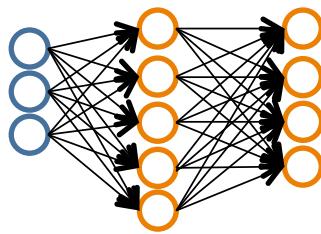
Important: Be sure to disable your gradient checking code before training your classifier.

- If you run the numerical gradient computation on every iteration of gradient descent, your code will be very slow

Putting It All Together

Training a Neural Network

Pick a network architecture (connectivity pattern between nodes)



- # input units = # of features in dataset
- # output units = # classes

Reasonable default: 1 hidden layer

- or if >1 hidden layer, have same # hidden units in every layer (usually the more the better)

Training a Neural Network

1. Randomly initialize weights
2. Implement forward propagation to get $h_{\Theta}(\mathbf{x}_i)$ for any instance \mathbf{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