

# Neural Networks

Evgeny Burnaev

Skoltech, Moscow, Russia



1 Motivation

2 Perceptron

3 Neural Networks

4 Backpropagation

## 1 Motivation

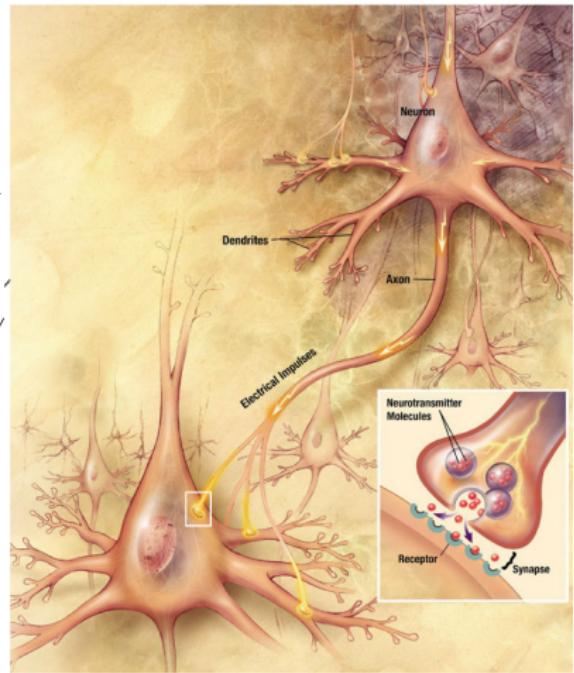
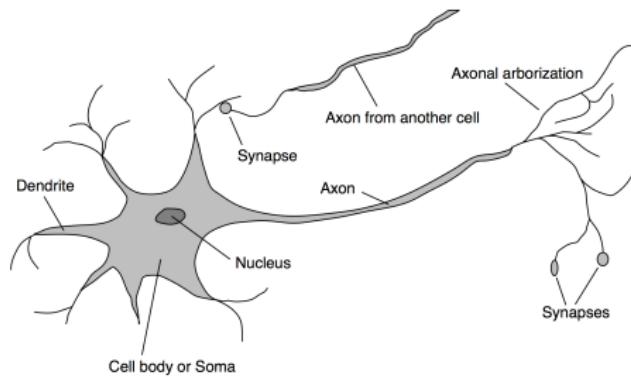
## 2 Perceptron

## 3 Neural Networks

## 4 Backpropagation

- Brain functions (thoughts) 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

# Neurons II



- 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 don't understand how this neural structure leads to what we perceive as "consciousness" or "thought"

# The “one learning algorithm” hypothesis

- There is some evidence that the human brain uses essentially the same algorithm to understand many different input modalities
- Example: Ferret experiments, in which the “input” for vision was plugged into auditory part of brain, and the auditory cortex learns to “see”

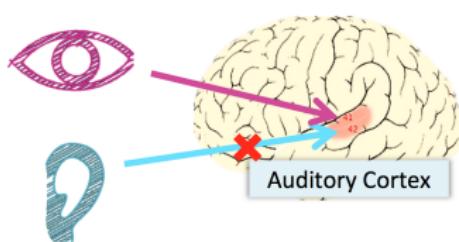


Figure – Auditory cortex learns to see

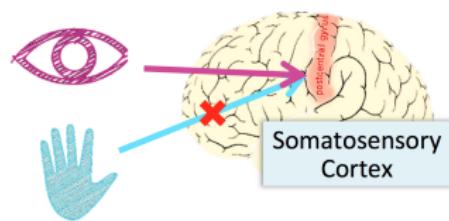


Figure – Somatosensory cortex learns to see



Figure – Seeing with your tongue



Figure – Haptic belt: direction sense

- Computers are way faster than neurons, e.g.: cycle time is  $10^{-9}$  sec for a computer vs.  $10^{-3}$  sec for a human brain
- But there are a lot more neurons ( $10^{11}$  neurons,  $10^{14}$  synapses) than we can reasonably model in modern digital computers, and they all fire in parallel (bandwidth is  $10^9$  bits/sec for a computer vs.  $10^{14}$  bits/sec for a human brain)
- Neural networks are designed to be massively parallel
- The brain is effectively a billion times faster

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

1 Motivation

2 Perceptron

3 Neural Networks

4 Backpropagation

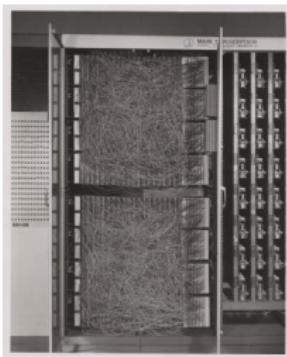
Cornell Aeronautical Laboratory



**Figure –** Rosenblatt & Mark I Perceptron: the first machine that could “learn” to recognize and identify optical patterns

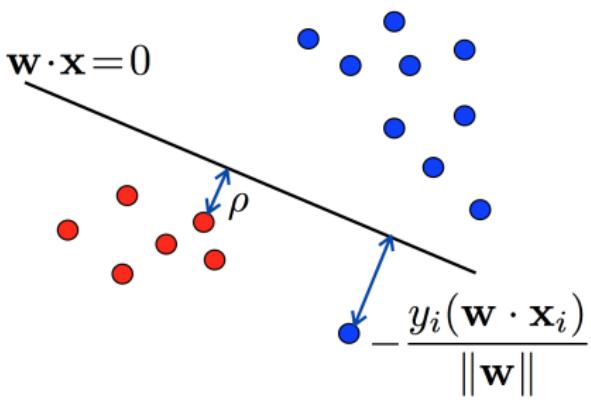
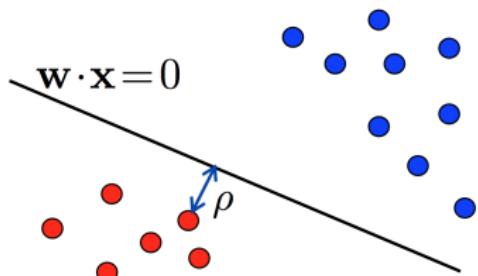
- Invented by F. Rosenblatt in 1957 in an attempt to understand human memory, learning and cognitive processes
- The first neural network model by computation, with a remarkable learning algorithm
- Became the foundation of pattern recognition research

One of the earliest and most influential neural networks: an important milestone in AI



- The Mark I Perceptron machine was connected to a camera with  $20 \times 20$  cadmium sulfide photocells to produce a 400-pixel image. The main visible feature is a patchboard that allowed experimentation with different combinations of input features. To the right of that are arrays of potentiometers, implementing the adaptive weights
- The New York Times wrote that it was "the embryo of an electronic computer that [the Navy] expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence". In short, artificial intelligence

- Margin and errors



Perceptron( $\mathbf{w}_0$ )

1.  $\mathbf{w}_1 \leftarrow \mathbf{w}_0$       (typically  $\mathbf{w}_0 = 0$ )
2. **for**  $t \leftarrow 1$  **to**  $T$  **do**
3.     Receive( $\mathbf{x}_t$ )
4.      $\hat{y}_t \leftarrow \text{sgn}(\mathbf{w}_t \cdot \mathbf{x}_t)$
5.     Receive( $y_t$ )
6.     **if** ( $\hat{y}_t \neq y_t$ ) **then**
7.          $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + y_t \mathbf{x}_t$     (or  $\leftarrow \mathbf{w}_t + \eta y_t \mathbf{x}_t$ ,  $\eta > 0$ )
8.         **else**  $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t$
9. **return**  $\mathbf{w}_{T+1}$

- **Objective function:** convex but not differentiable

$$F(\mathbf{w}) = \frac{1}{T} \sum_{t=1}^T \max(0, -y_t(\mathbf{w} \cdot \mathbf{x}_t)) = \mathbb{E}_{\mathbf{x} \sim \hat{D}}[f(\mathbf{w}, \mathbf{x})]$$

with  $f(\mathbf{w}, \mathbf{x}) = \max(0, -y(\mathbf{w} \cdot \mathbf{x}))$

- **Stochastic gradient:** for each  $\mathbf{x}_t$ , the update is

$$\mathbf{w}_{t+1} \leftarrow \begin{cases} \mathbf{w}_t - \eta \nabla_{\mathbf{w}} f(\mathbf{w}_t, \mathbf{x}_t) & \text{if differentiable,} \\ \mathbf{w}_t & \text{otherwise,} \end{cases}$$

where  $\eta$  is a learning rate parameter

- Here:

$$\mathbf{w}_{t+1} \leftarrow \begin{cases} \mathbf{w}_t + \eta y_t \mathbf{x}_t & \text{if } y_t(\mathbf{w}_t \cdot \mathbf{x}_t) < 0 \\ \mathbf{w}_t & \text{otherwise} \end{cases}$$

- A margin-based upper bound on the number of mistakes or updates made by the Perceptron algorithm when processing a sequence of  $T$  points that can be linearly separated by a hyperplane with margin  $\rho > 0$
- **Theorem:**
  - Let  $\mathbf{x}_1, \dots, \mathbf{x}_T \in \mathbb{R}^d$  be a sequence of  $T$  points with  $\|\mathbf{x}_t\| \leq R$  for all  $t \in [1, T]$ , for some  $R > 0$ .
  - Assume that there exists  $\rho > 0$  and  $\mathbf{v} \in \mathbb{R}^d$  such that for all  $t \in [1, T]$ ,  $\rho \leq \frac{y_t(\mathbf{v} \cdot \mathbf{x}_t)}{\|\mathbf{v}\|}$ .
  - Then, the number of updates (= the number of mistakes) made by the Perceptron algorithm when processing  $\mathbf{x}_1, \dots, \mathbf{x}_T \in \mathbb{R}^d$  is bounded by  $\frac{R^2}{\rho^2}$ .
- **Proof:** Let  $I$  be the set of  $t$ -s at which there is an update and let  $M$  be the total number of updates

- Summing up the assumption inequalities gives:

$$\begin{aligned}
 M\rho &\leq \frac{\mathbf{v} \cdot \sum_{t \in I} y_t \mathbf{x}_t}{\|\mathbf{v}\|} \\
 &= \frac{\mathbf{v} \cdot \sum_{t \in I} (\mathbf{w}_{t+1} - \mathbf{w}_t)}{\|\mathbf{v}\|} \quad (\text{definition of updates}) \\
 &= \frac{\mathbf{v} \cdot \mathbf{w}_{T+1}}{\|\mathbf{v}\|} \\
 &\leq \|\mathbf{w}_{T+1}\| \quad (\text{Cauchy-Schwarz ineq.}) \\
 &= \|\mathbf{w}_{t_m} + y_{t_m} \mathbf{x}_{t_m}\| \quad (t_m \text{ is the largest } t \text{ in } I) \\
 &= \left[ \|\mathbf{w}_{t_m}\|^2 + \|\mathbf{x}_{t_m}\|^2 + 2 \underbrace{y_{t_m} \mathbf{w}_{t_m} \cdot \mathbf{x}_{t_m}}_{\leq 0} \right]^{1/2}
 \end{aligned}$$

$$\begin{aligned} M\rho &\leq \left[ \|\mathbf{w}_{t_m}\|^2 + \|\mathbf{x}_{t_m}\|^2 + 2\underbrace{y_{t_m} \mathbf{w}_{t_m} \cdot \mathbf{x}_{t_m}}_{\leq 0} \right]^{1/2} \\ &\leq [\|\mathbf{w}_{t_m}\|^2 + R^2]^{1/2} \\ &\leq [MR^2]^{1/2} = \sqrt{M}R \text{ (applying the same to previous } t\text{-s in } I) \end{aligned}$$

- bound independent of dimension and tight
- convergence can be slow for small margin, the number of updates made by the algorithm can be in  $\Omega(2^N)$
- among the many variants: voted perceptron algorithm. Predict according to

$$\text{sign} \left( \left( \sum_{t \in I} c_t \mathbf{w}_t \right) \cdot \mathbf{x} \right),$$

where  $c_t$  is the number of iterations  $\mathbf{w}_t$  survives

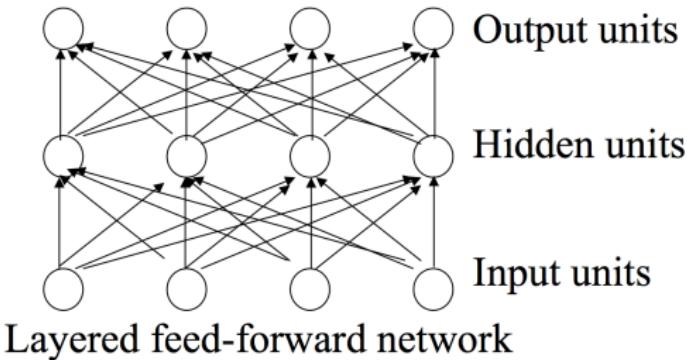
- $\{\mathbf{x}_t : t \in I\}$  are the support vectors for the perceptron algorithm, since  $\mathbf{w}_T = \sum_{t \in I} y_t \mathbf{x}_t$
- non-separable case: does not converge

1 Motivation

2 Perceptron

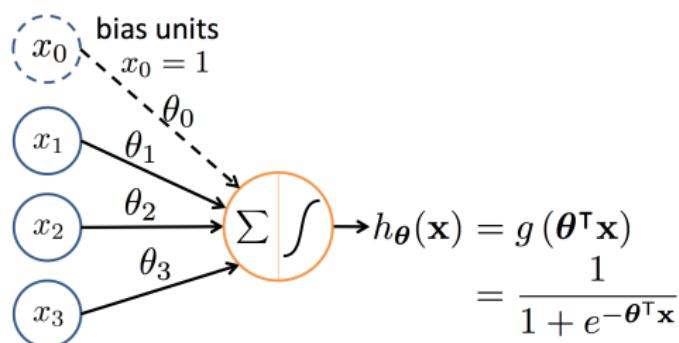
3 Neural Networks

4 Backpropagation

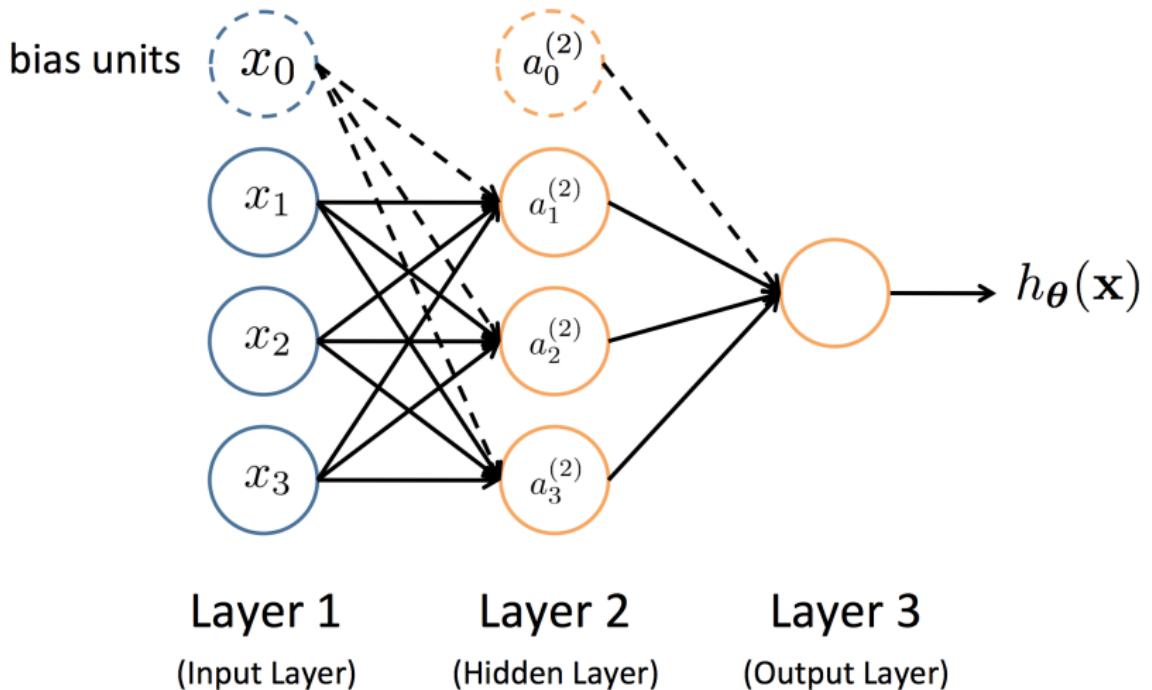


- Neural Networks are made up of **nodes** or **units**, connected by **links**
- Each link has an associated **weight** and activation level
- Each node has an **input function** (typically summing over weighted inputs), an **activation function**, and an **output**

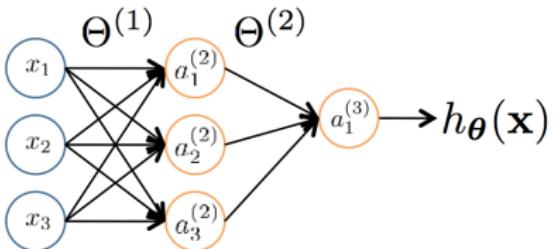
- Let us define  $\mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}$  and  $\boldsymbol{\theta} = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix}$



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



- Input layer units are set by some exterior function, 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 controlling function mapping from layer  $j$  to layer  $j + 1$

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

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

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

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

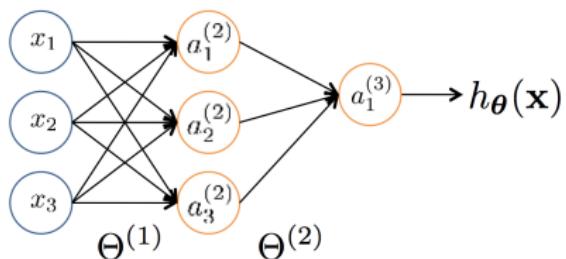
If network has  $s_j$  units in layer  $j$  and  $s_{j+1}$  units in layer  $j + 1$ , then  $\Theta^{(j)}$  has dimension  $s_{j+1} \times (s_j + 1)$ :  $\Theta^{(1)} \in \mathbb{R}^{3 \times 4}$ ,  $\Theta^{(2)} \in \mathbb{R}^{1 \times 4}$

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

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

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

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

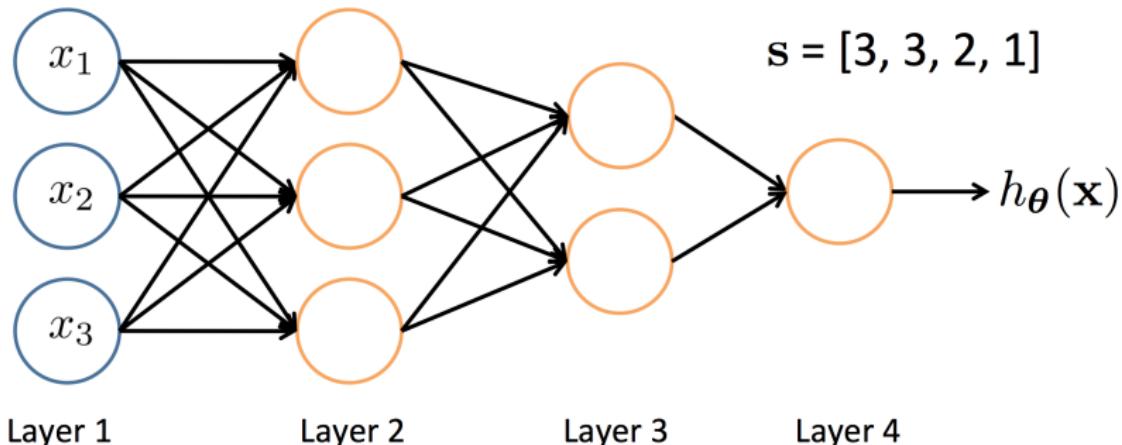


Feed-forward Steps:

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

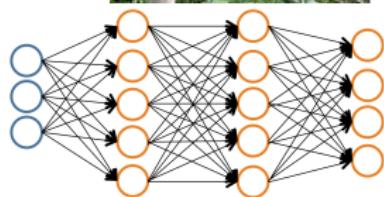
Add  $a_0^{(2)} = 1$

- $\mathbf{z}^{(3)} = \Theta^{(2)}\mathbf{a}^{(2)}$
- $h_\theta(\mathbf{x}) = \mathbf{a}^{(3)} = g(\mathbf{z}^{(3)})$



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

# Multiple Output Units: One-vs-Rest I



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

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

when cat

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

when dog

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

when car

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

when driver



Figure – Cat



Figure – Dog



Figure – Car

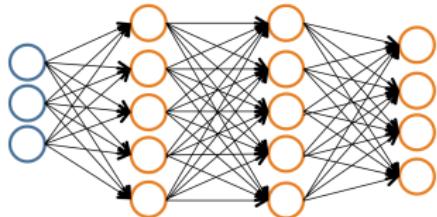


Figure – Driver

- Given  $\{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_m, y_m)\}$

- Convert labels to 1-of- $K$  repr., e.g.  $y_i = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$  when car, etc.

$$\begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$



Given

- $\{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_m, y_m)\}$
- $\mathbf{s} \in (\mathbb{N}_+)^L$  contains number of nodes at each layer,  $s_0 = N$  (number of features)

Multi-class classification ( $K$  classes)

Binary classification:

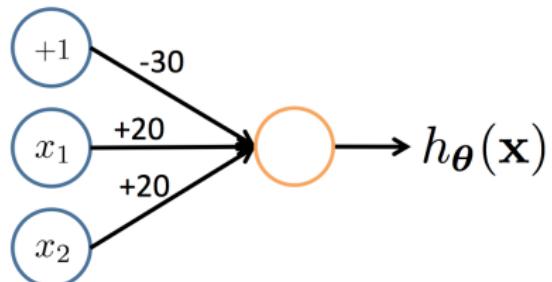
- $y = 0$  or  $1$
- 1 output unit ( $s_{L-1} = 1$ )

- $y \in \mathbb{R}^K$ , e.g.  $y = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$  when car
- $K$  output units ( $s_{L-1} = K$ )

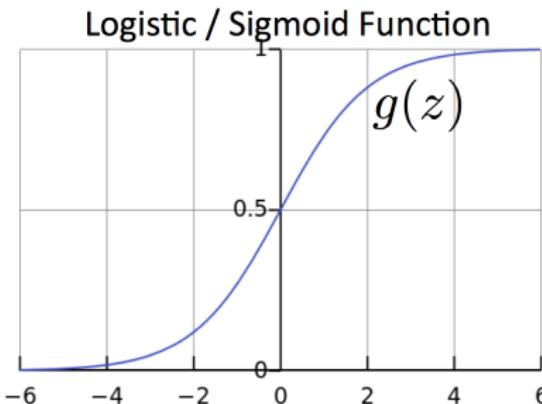
## Simple example: AND

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

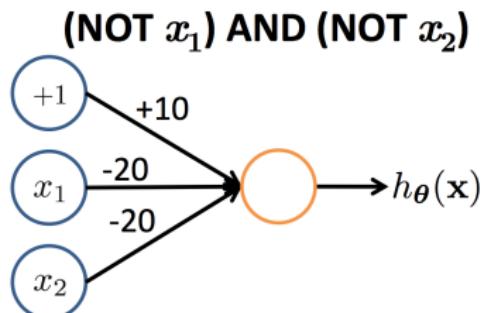
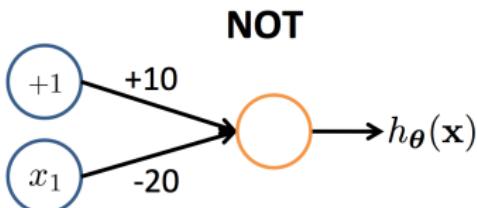
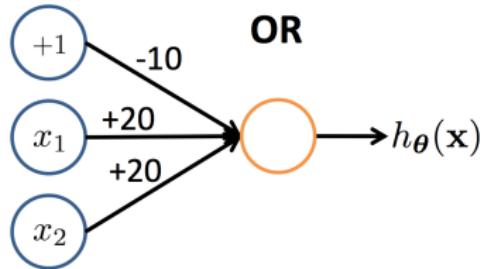
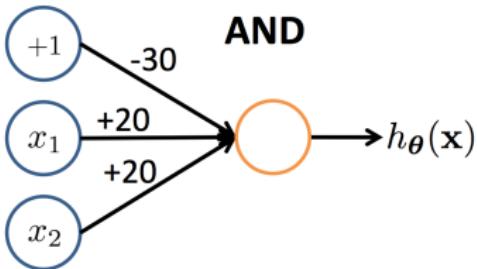
$$y = x_1 \text{ AND } x_2$$



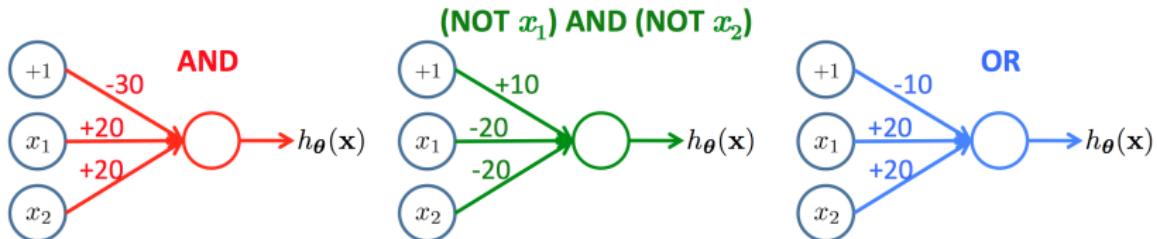
$$h_{\theta}(\mathbf{x}) = g(-30 + 20x_1 + 20x_2)$$



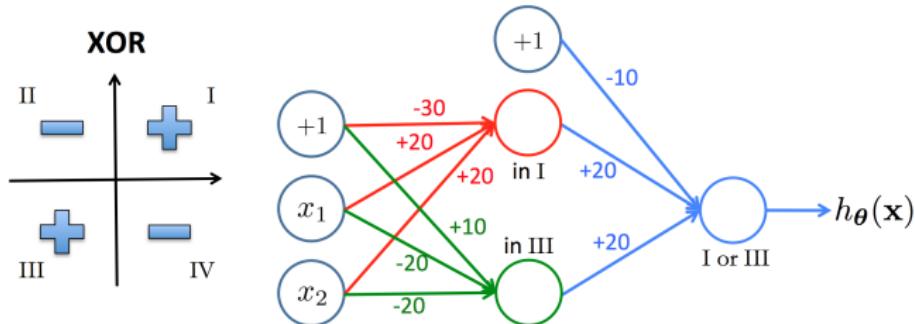
$x_1$	$x_2$	$h_{\theta}(\mathbf{x})$
0	0	$g(-30) \approx 0$
0	1	$g(-10) \approx 0$
1	0	$g(-10) \approx 0$
1	1	$g(10) \approx 1$



Building blocks:



XOR function:

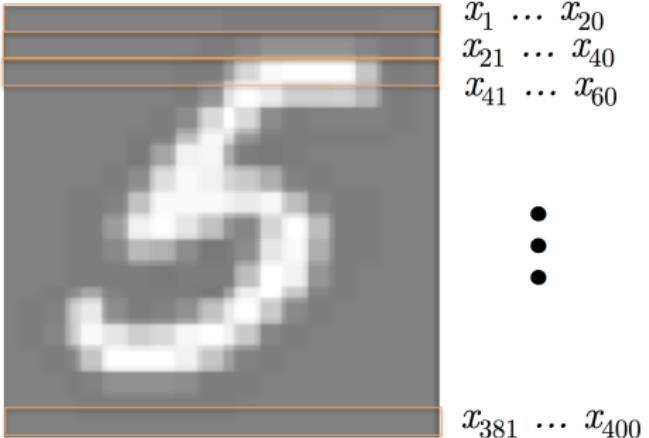


# Representation of Layers I

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	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

20 × 20 pixel images

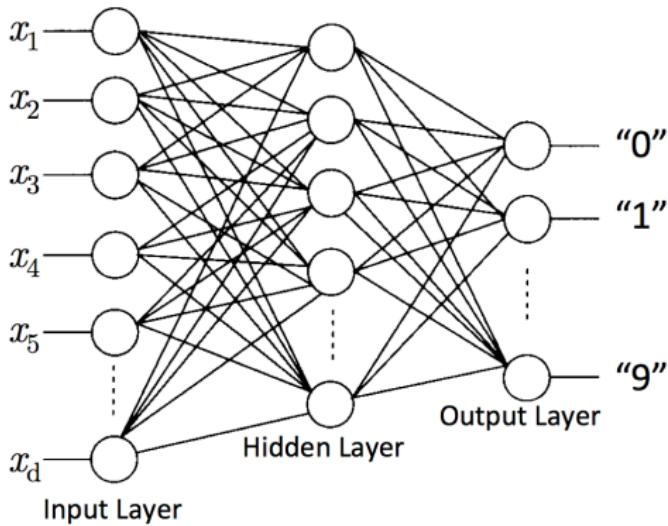
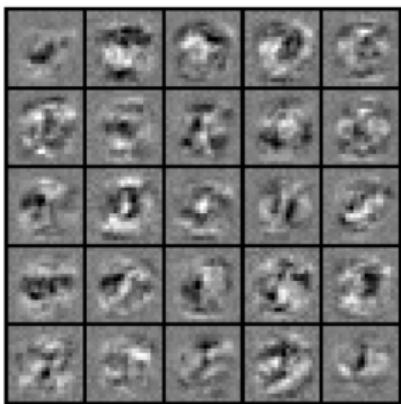
$d = 400$    10 classes



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

## Representation of Layers II

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	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	7	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



Visualization of  
Hidden Layer

1 Motivation

2 Perceptron

3 Neural Networks

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

Logistic Regression for binary {0, 1}-case:

$$\widehat{R}(\theta) = -\frac{1}{m} \sum_{i=1}^m [y_i \log h_\theta(\mathbf{x}_i) + (1 - y_i) \log(1 - h_\theta(\mathbf{x}_i))] + \frac{\lambda}{2m} \sum_{j=1}^N \theta_j^2$$

Neural Network:

$$h_\theta \in \mathbb{R}^K, \quad (h_\theta(\mathbf{x}))_k = k^{\text{th}} \text{ output}$$

$$\begin{aligned}\widehat{R}(\theta) &= -\frac{1}{m} \left[ \sum_{i=1}^m \sum_{k=1}^K y_{ik} \log(h_\theta(\mathbf{x}_i))_k + (1 - y_{ik}) \log(1 - (h_\theta(\mathbf{x}_i))_k) \right] \\ &\quad + \frac{\lambda}{2m} \sum_{l=1}^{L-1} \sum_{i=1}^{s_l-1} \sum_{j=1}^{s_l} \left( \theta_{ji}^{(l)} \right)^2\end{aligned}$$

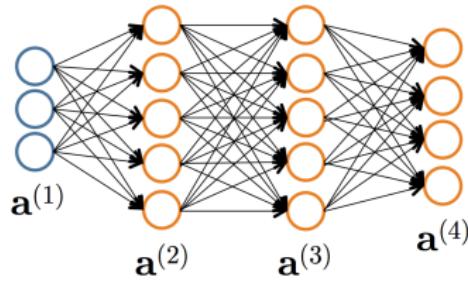
$$\begin{aligned}\widehat{R}(\theta) = & -\frac{1}{m} \left[ \sum_{i=1}^m \sum_{k=1}^K y_{ik} \log(h_\theta(\mathbf{x}_i))_k + (1 - y_{ik}) \log(1 - (h_\theta(\mathbf{x}_i))_k) \right] \\ & + \frac{\lambda}{2m} \sum_{l=1}^{L-1} \sum_{i=1}^{s_l-1} \sum_{j=1}^{s_l} \left( \theta_{ji}^{(l)} \right)^2\end{aligned}$$

- Minimize  $\widehat{R}(\theta)$
- Need code to compute
  - $\widehat{R}(\theta)$
  - $\frac{\partial}{\partial \theta_{ji}^{(l)}} \widehat{R}(\theta)$
- $\widehat{R}(\theta)$  is not convex, so GD on a neural net yields a local optimum, but tends to work well in practice

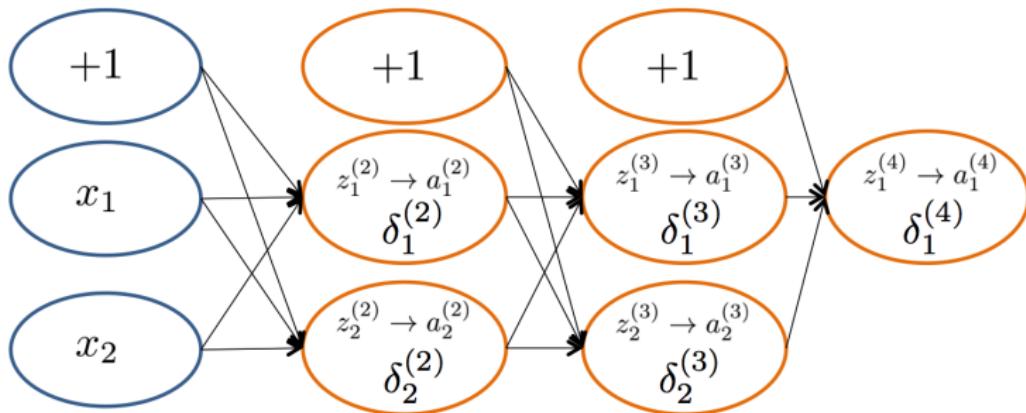
- Given one labeled training instance  $(\mathbf{x}, y)$

## Forward Propagation

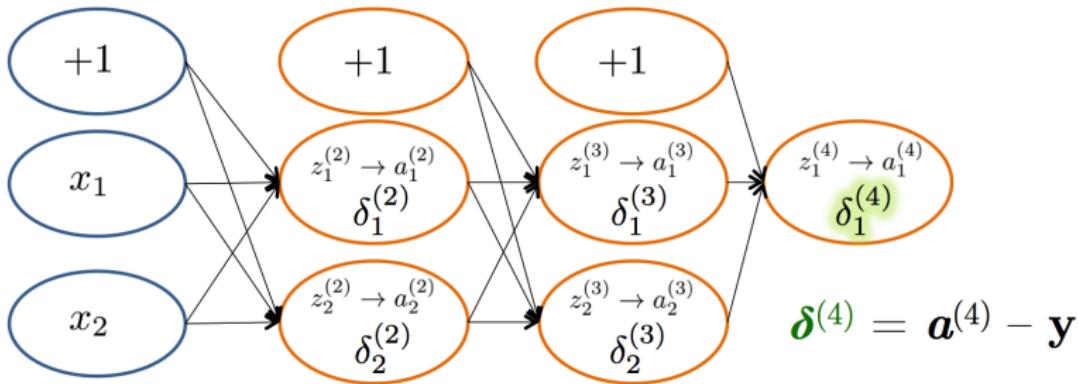
- $\mathbf{a}^{(1)} = \mathbf{x}$
- $\mathbf{z}^{(2)} = \Theta^{(1)}\mathbf{a}^{(1)}$
- $\mathbf{a}^{(2)} = g(\mathbf{z}^{(2)})$  [add  $a_0^{(2)}$ ]
- $\mathbf{z}^{(3)} = \Theta^{(2)}\mathbf{a}^{(2)}$
- $\mathbf{a}^{(3)} = g(\mathbf{z}^{(3)})$  [add  $a_0^{(3)}$ ]
- $\mathbf{z}^{(4)} = \Theta^{(3)}\mathbf{a}^{(3)}$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



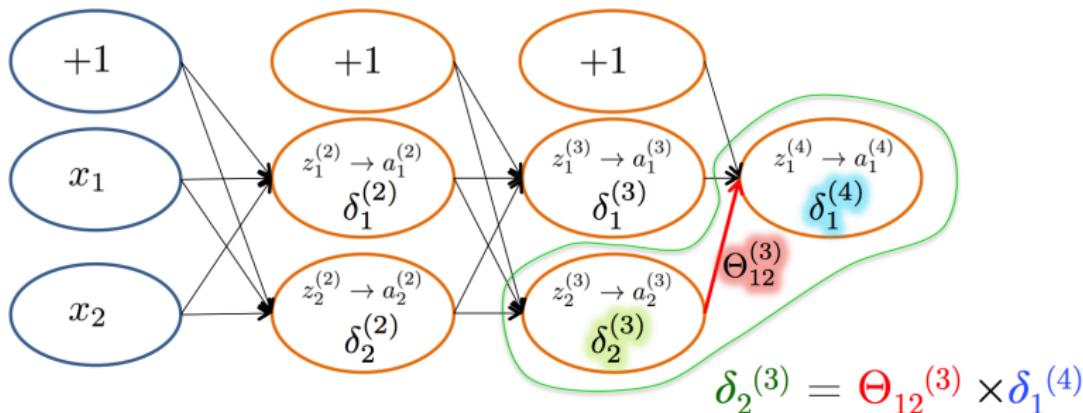
- Each hidden node  $j$  is “responsible” for some fraction of the error  $\delta_j^{(l)}$  in each of the output nodes to which it connects
- $\delta_j^{(l)}$  is divided according to the strength of the connection between hidden node and the output node
- Then, the “blame” is propagated back to provide the error values for the hidden layer



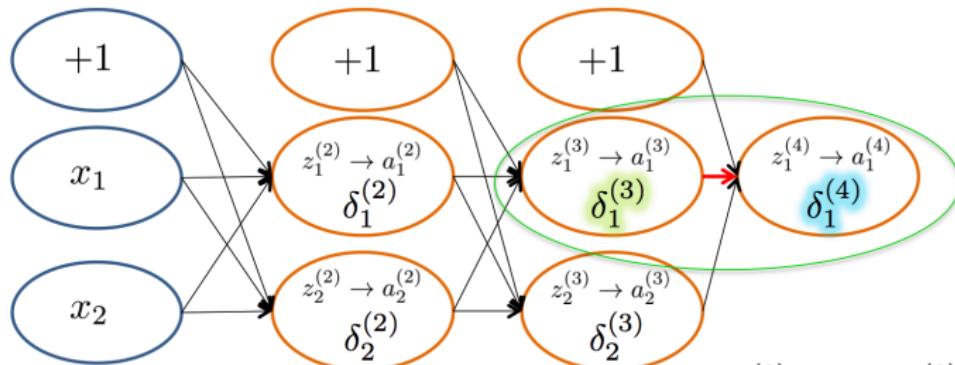
- $\delta_j^{(l)}$  = “error” of node  $j$  in layer  $l$
- Formally,  $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(\mathbf{x}_i)$ ,  
where  $\text{cost}(\mathbf{x}_i) = y_i \log h_\theta(\mathbf{x}_i) + (1 - y_i) \log(1 - h_\theta(\mathbf{x}_i))$



- $\delta_j^{(l)}$  = “error” of node  $j$  in layer  $l$
- Formally,  $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(\mathbf{x}_i)$ ,  
where  $\text{cost}(\mathbf{x}_i) = y_i \log h_\theta(\mathbf{x}_i) + (1 - y_i) \log(1 - h_\theta(\mathbf{x}_i))$



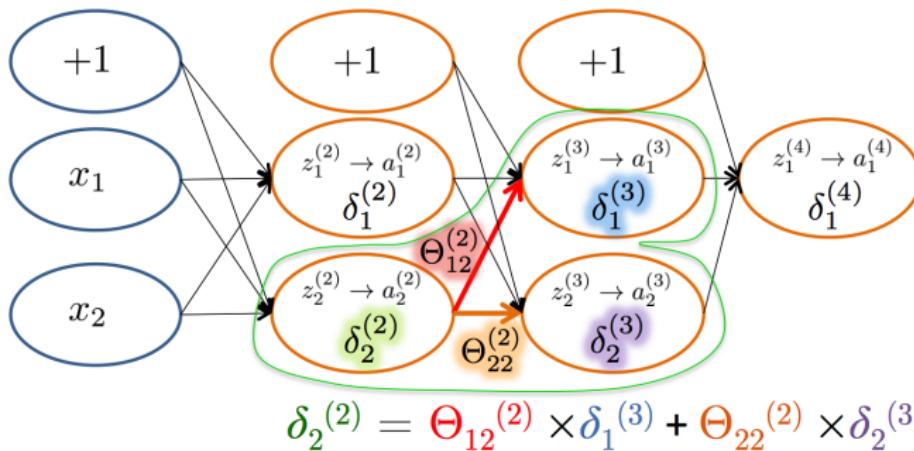
- $\delta_j^{(l)}$  = “error” of node  $j$  in layer  $l$
- Formally,  $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(\mathbf{x}_i)$ ,  
where  $\text{cost}(\mathbf{x}_i) = y_i \log h_\theta(\mathbf{x}_i) + (1 - y_i) \log(1 - h_\theta(\mathbf{x}_i))$



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

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

- $\delta_j^{(l)}$  = “error” of node  $j$  in layer  $l$
- Formally,  $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(\mathbf{x}_i)$ ,  
where  $\text{cost}(\mathbf{x}_i) = y_i \log h_\theta(\mathbf{x}_i) + (1 - y_i) \log(1 - h_\theta(\mathbf{x}_i))$



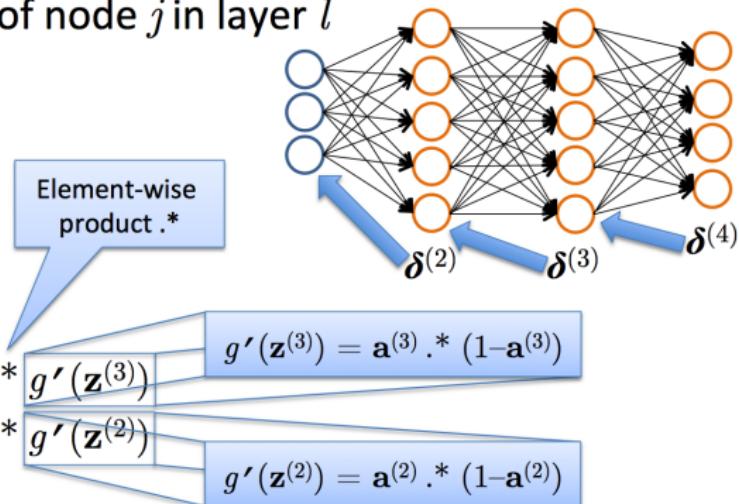
- $\delta_j^{(l)}$  = “error” of node  $j$  in layer  $l$
- Formally,  $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(\mathbf{x}_i)$ ,  
where  $\text{cost}(\mathbf{x}_i) = y_i \log h_\theta(\mathbf{x}_i) + (1 - y_i) \log(1 - h_\theta(\mathbf{x}_i))$

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  $\delta^{(1)}$ )



$$\frac{\partial}{\partial \theta_{ij}^{(l)}} \hat{R}(\theta) = a_j^{(l)} \delta_i^{(l+1)} \quad (\text{ignoring } \lambda; \text{ if } \lambda = 0)$$

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

- $\mathbf{D}^{(l)}$  is the matrix of partial derivatives of  $\widehat{R}(\theta)$
- Note: can vectorize  $\Delta_{ij}^{(l)} = \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$  as  $\Delta^{(l)} = \Delta^{(l)} + \delta^{(l+1)} (\mathbf{a}^{(l)})^T$

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

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

Until weights converge or max #epochs is reached

- Local minima
- Random initialization
- Random division into train and validation sets
- Second order optimization methods
- Early stopping

