Neural Networks

Evgeny Burnaev

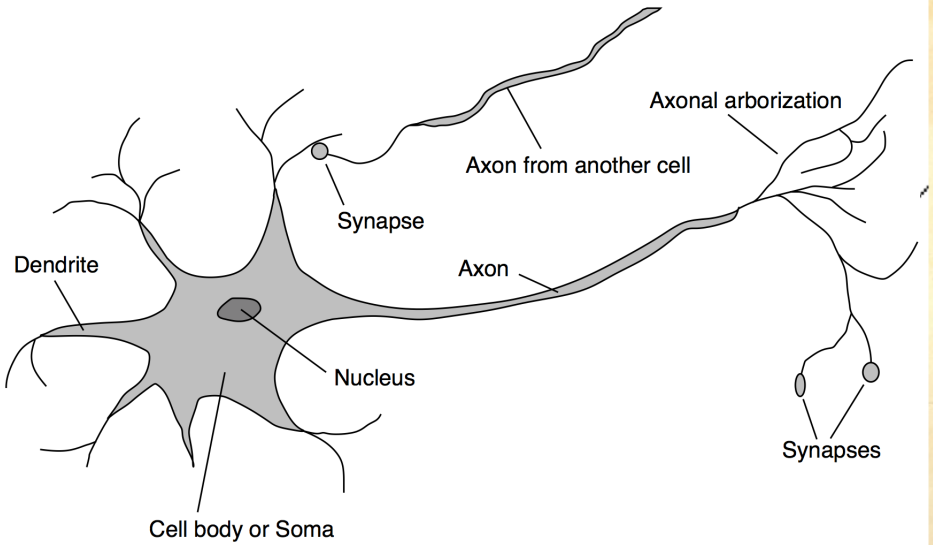
Skoltech, Moscow, Russia

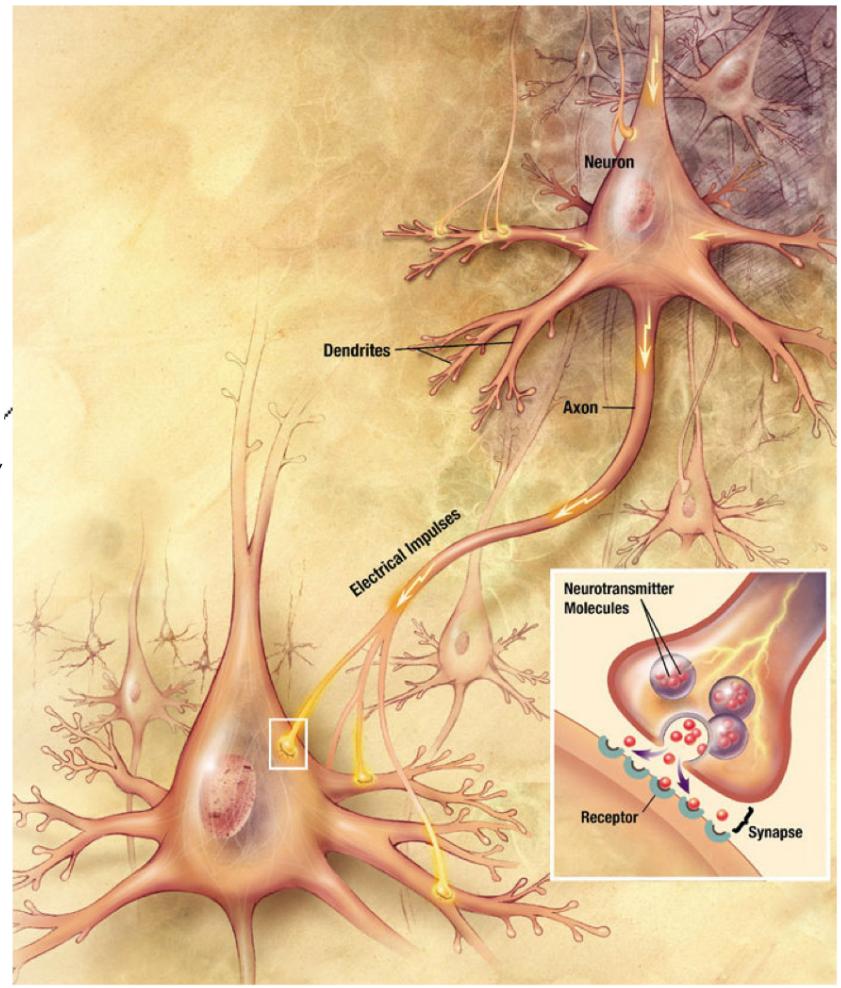
Outline

Neurons I

- 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 tresholds**
  - 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


Axonal arborization
Axon from another cell
Synapse
Dendrite
Axon
Nucleus
Synapses
Cell body or Soma


Neuron
Dendrites
Axon
Electrical Impulses
Neurotransmitter Molecules
Receptor
Synapse

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 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"

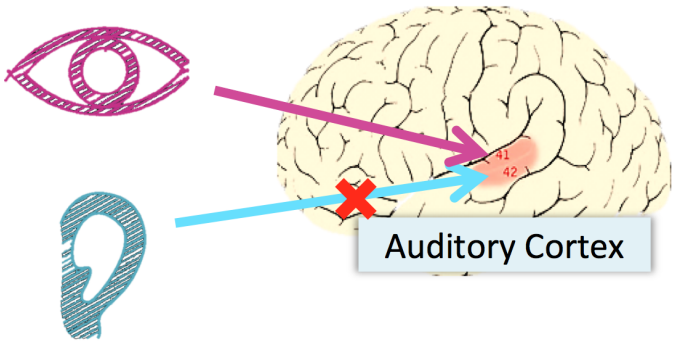


Figure : Auditory cortex learns to see

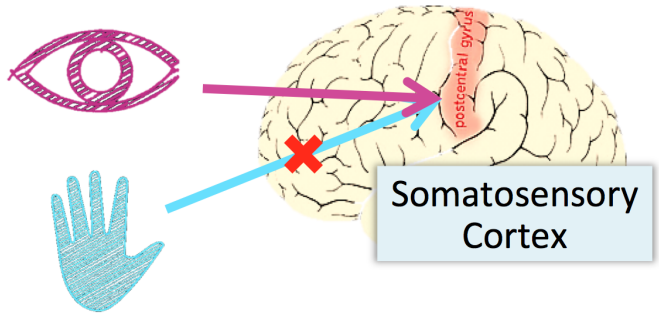


Figure : Somatosensory cortex learns to see

Sensor Representations in the Brain

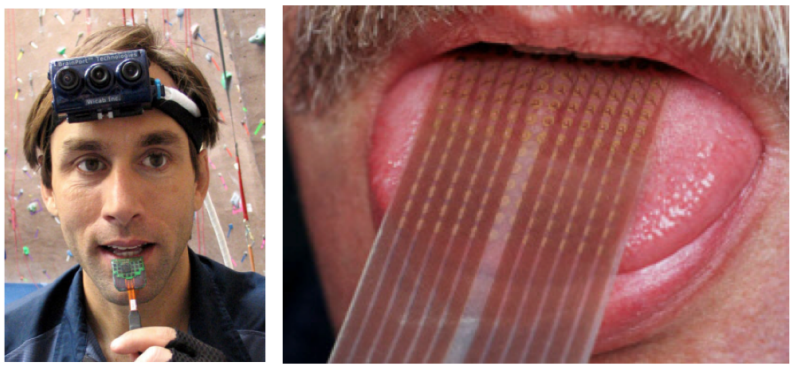


Figure : Seeing with your tongue

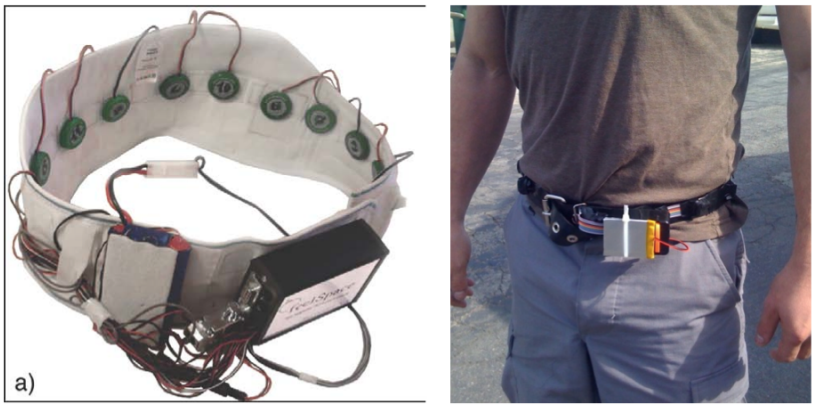


Figure : Haptic belt: direction sense

Comparison of computing power

- 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

Neural Networks

- 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

Rosenblatt and his Perceptron I

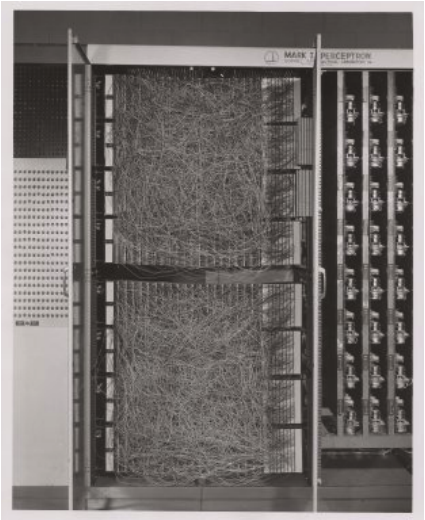
Cornell Aeronautical Laboratory

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

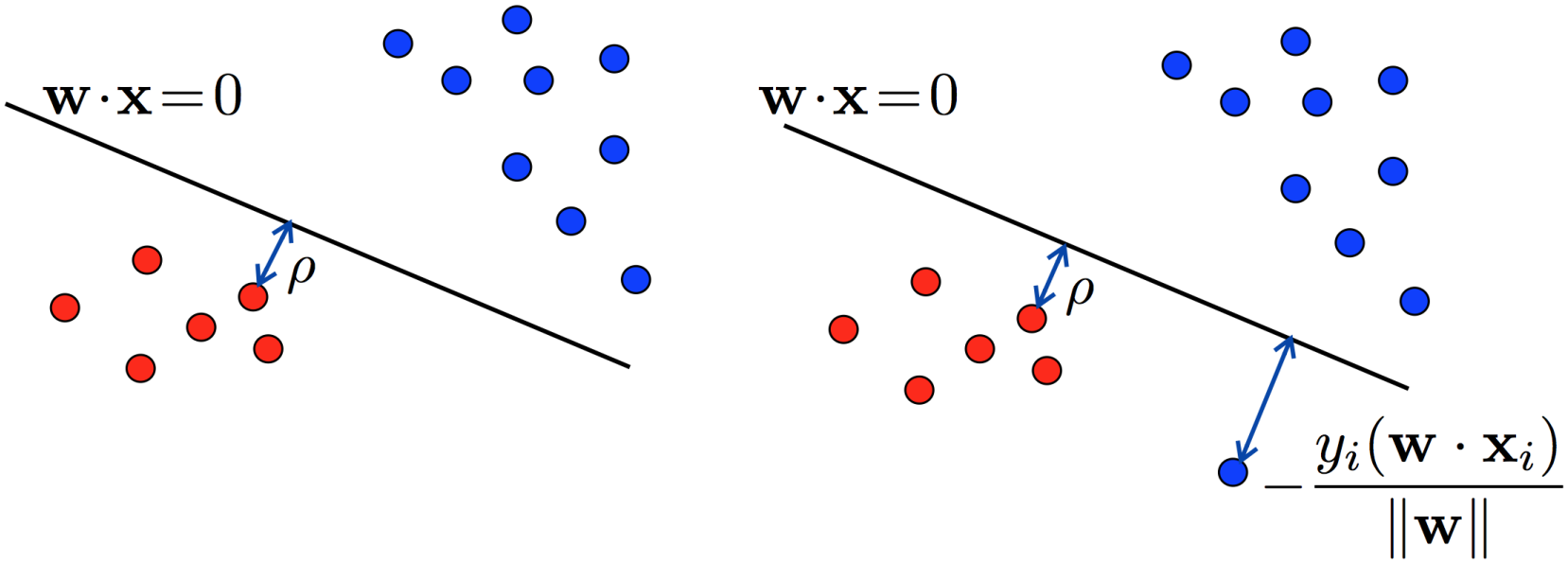
Rosenblatt and his Perceptron II



- The Mark I Perceptron machine was connected to a camera with 20×20 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

Separating Hyperplane

- Margin and errors



Perceptron Algorithm

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** $\mathbf{w}_{T+1}$

Perceptron = Stochastic Gradient Descent

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

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

  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 η is a learning rate parameter
- Here:

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

Perceptron Algorithm — Bound I

The following theorem gives 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 $\mathbf{x}_1, \ldots, \mathbf{x}_T \in \mathbb{R}^N$ 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 $\mathbf{v} \in \mathbb{R}^N$ 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 $\mathbf{x}_1, \ldots, \mathbf{x}_T \in \mathbb{R}^N$ 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

Perceptron Algorithm — Bound II

- Summing up the assumption inequalities gives:

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

Perceptron Algorithm — Bound III

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

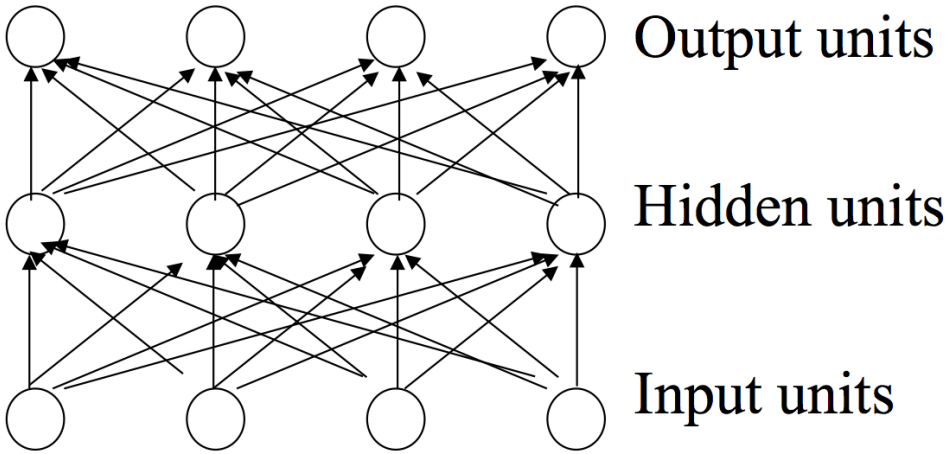
Comments

- 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

Neural Networks

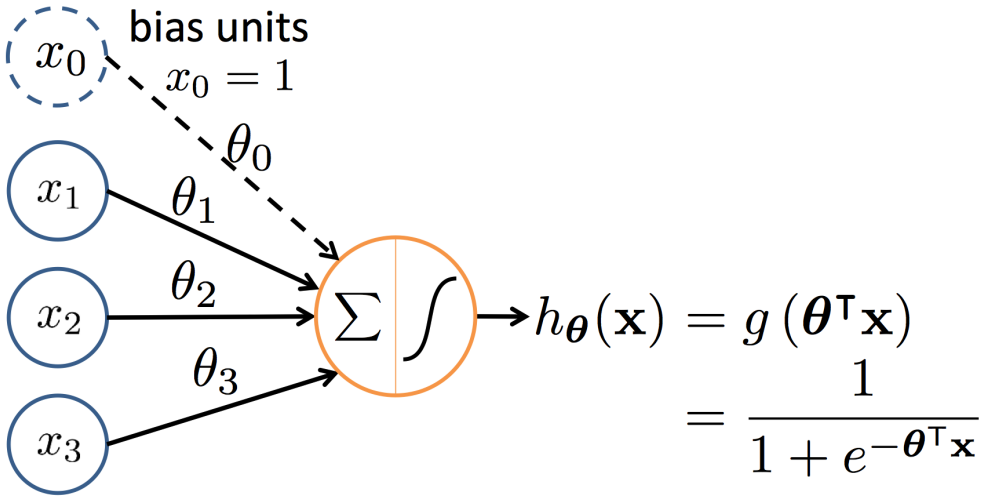


Layered feed-forward network

- 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**

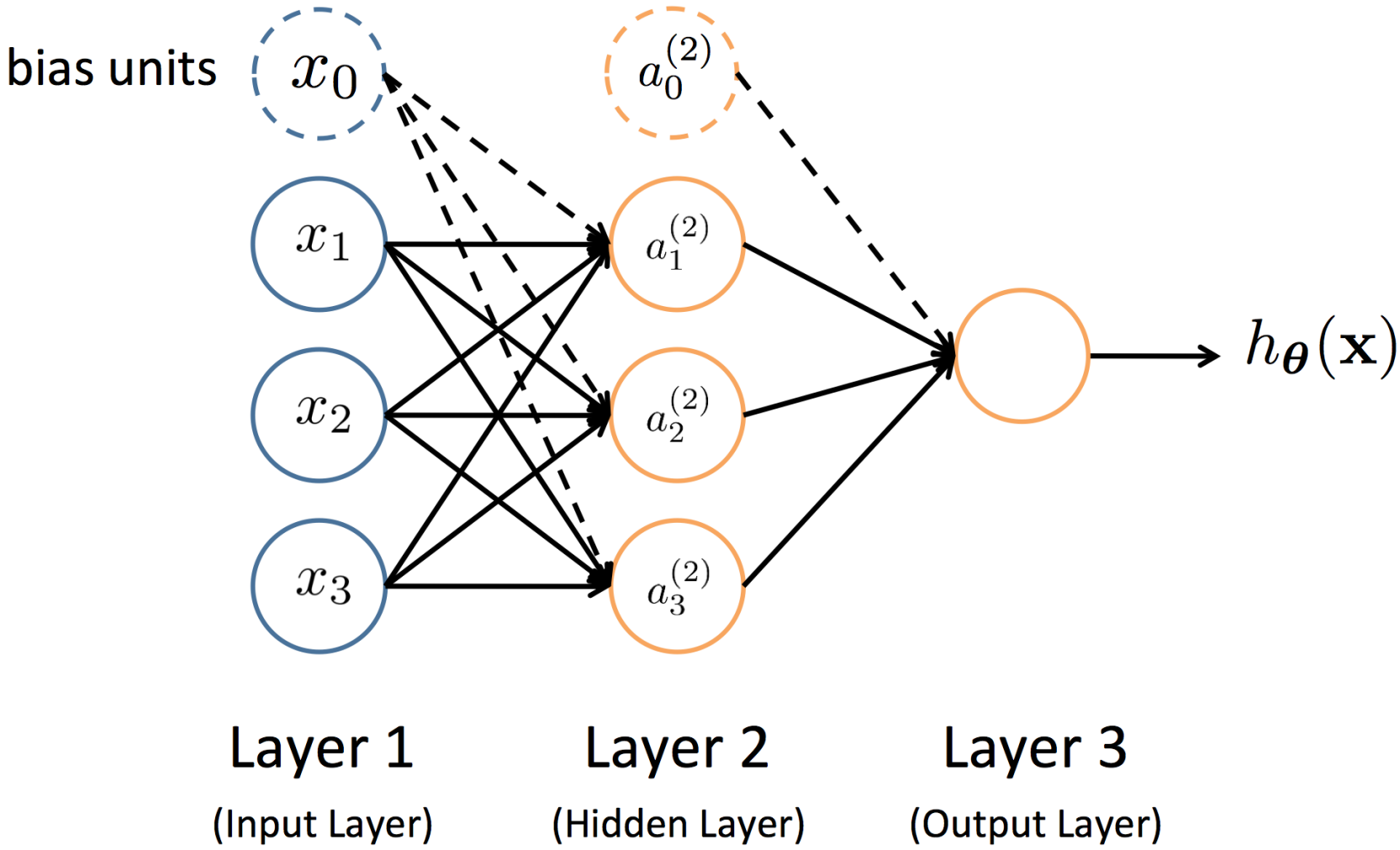
Neural Networks: Logistic (Sigmoid) Unit

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



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

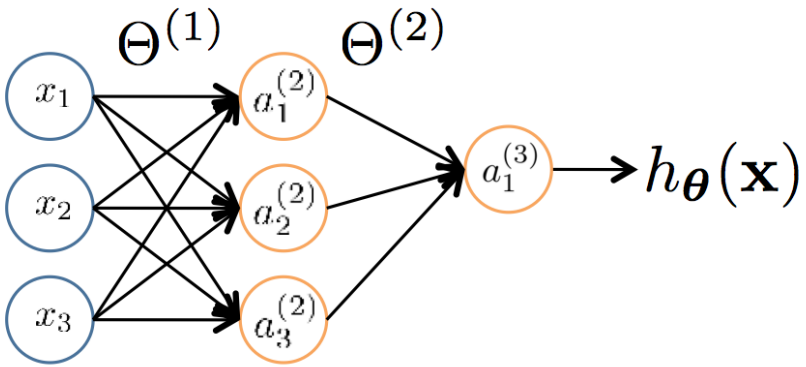
Multi-Layered Neural Network



Feed-Forward Process

- 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 betwork, 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: Equations



- $a_i^{(j)}$ = "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 $s_{j+1} \times (s_j + 1)$: $\Theta^{(1)} \in \mathbb{R}^{3\times 4}$, $\Theta^{(2)} \in \mathbb{R}^{1\times 4}$

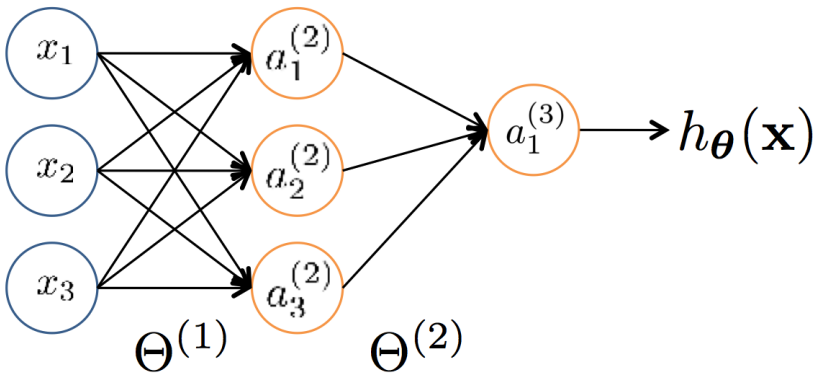
Neural Network: Vector Format

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

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

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

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



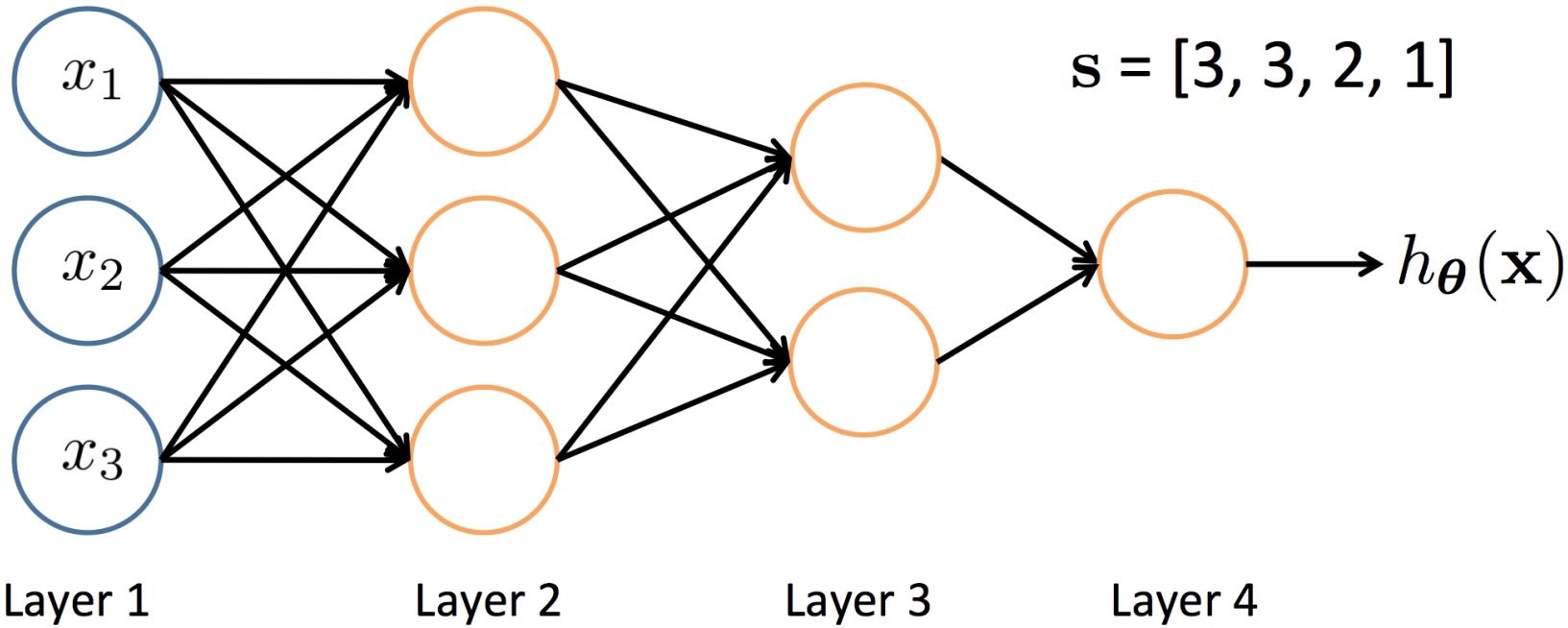
Feed-forward Steps:

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

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

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

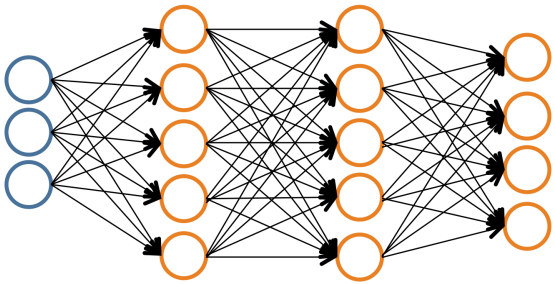
Other Neural Network Architectures



- 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

Multiple Output Network: One-vs-Rest II

Figure : Cat

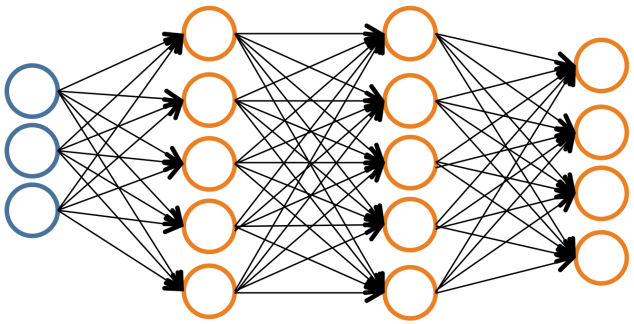
Figure : Dog

Figure : Car

Figure : Driver

- Given $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)\}$
- Convert labels to 1-of-K repr., e.g. $y_i = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$ when car, etc.

Neural Network Classification



Binary classification:

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

Given

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

Multi-class classification (K classes)

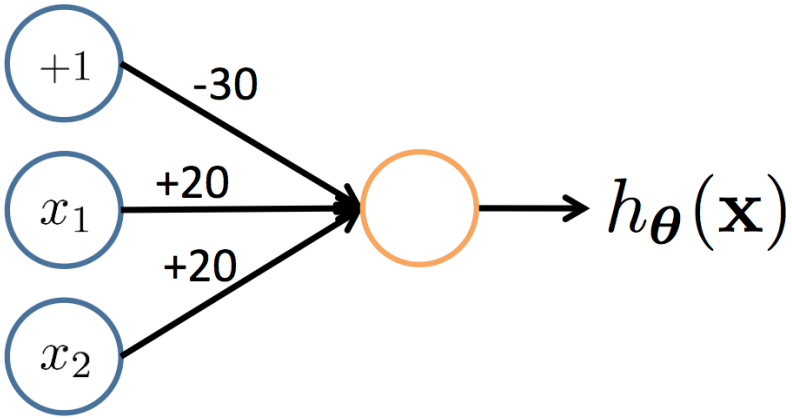
- $y \in \mathbb{R}^K$, e.g. $y = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$ when car
- K output units ($s_{L-1} = K$)

Representing Boolean Functions I

Simple example: AND

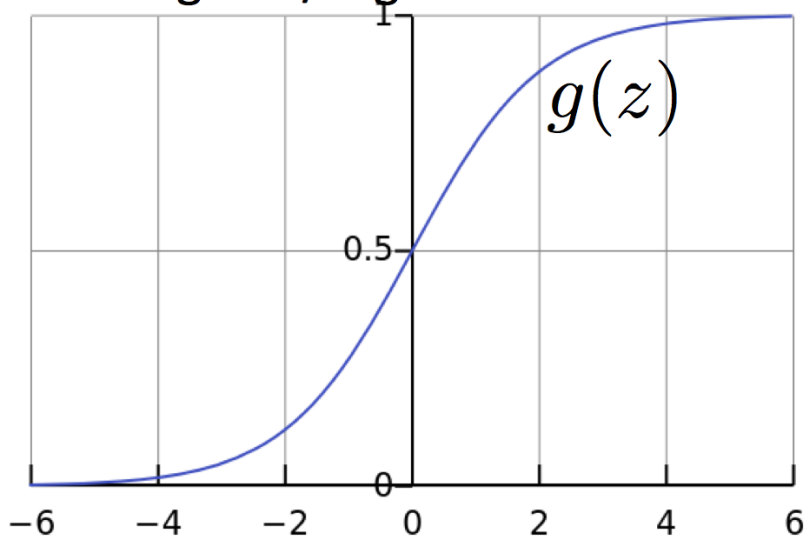
$x_1, x_2 \in \{0, 1\}$

$y = x_1$ AND x_2



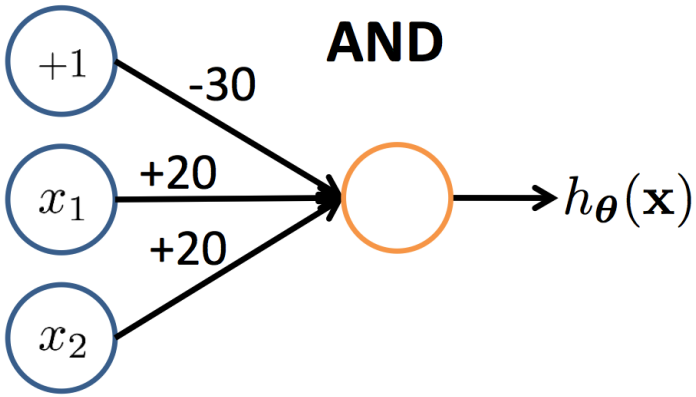
$$\mathrm{h}_\Theta(\mathbf{x}) = g(-30 + 20x_1 + 20x_2)$$

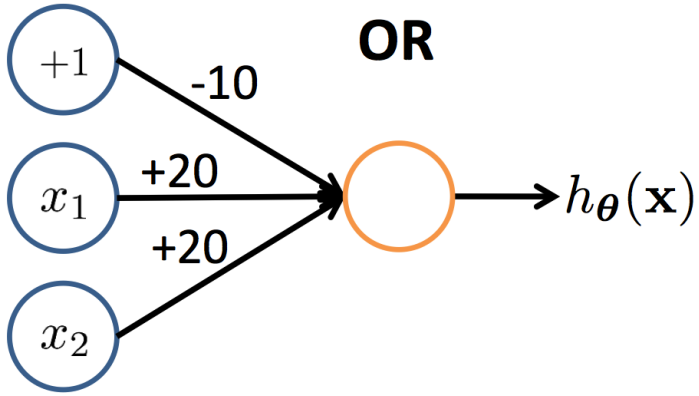
Logistic / Sigmoid Function

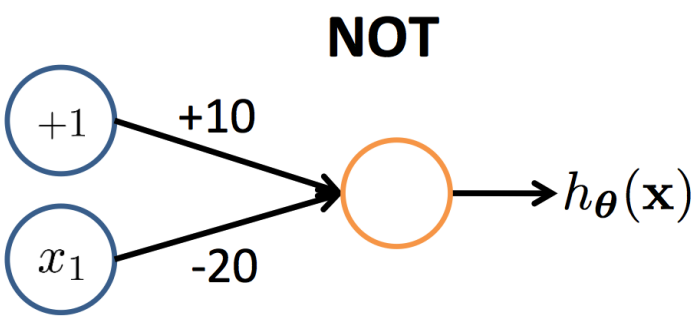


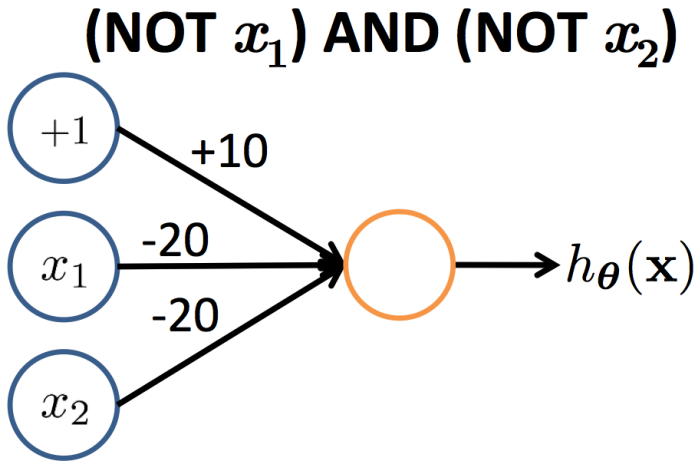
x_1	x_2	$\mathrm{h}_\Theta(\mathbf{x})$
0	0	$g(-30) \approx 0$
0	1	$g(-10) \approx 0$
1	0	$g(-10) \approx 0$
1	1	$g(10) \approx 1$

Representing Boolean Functions II



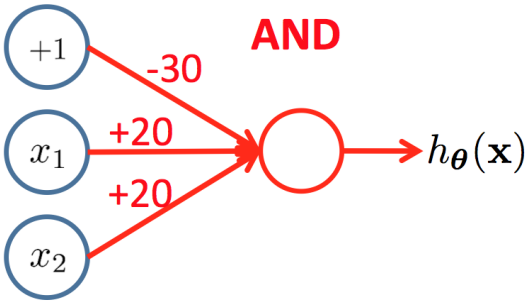


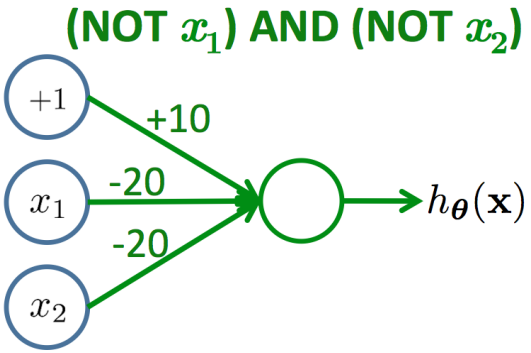


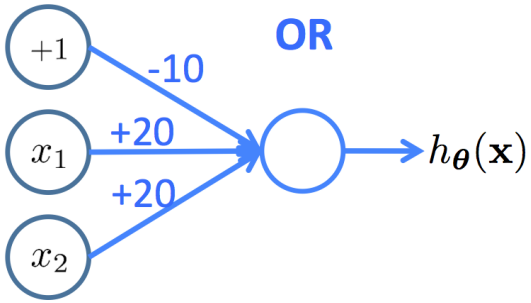


Combining Representations for Non-linearity

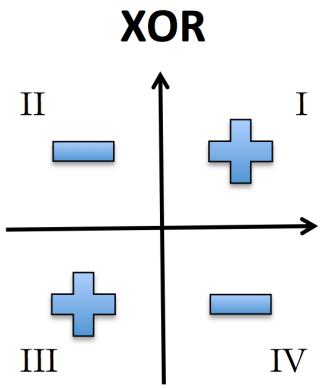
Building blocks:

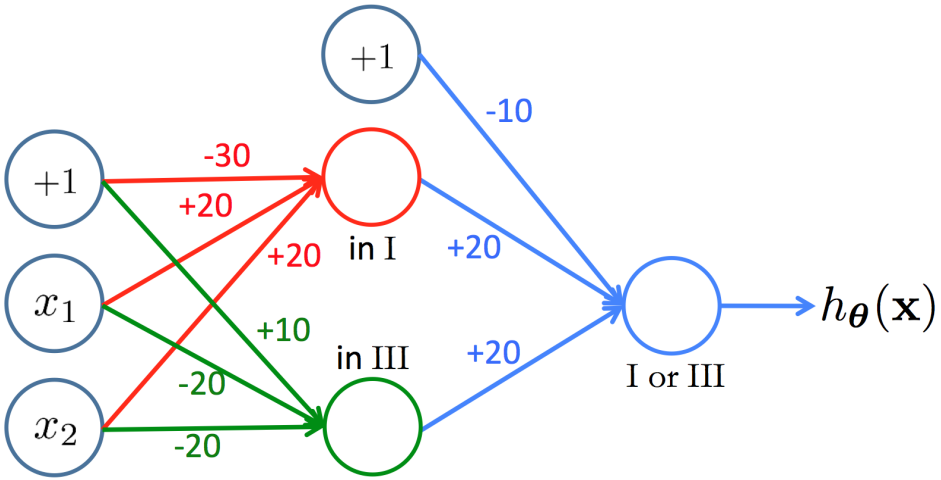




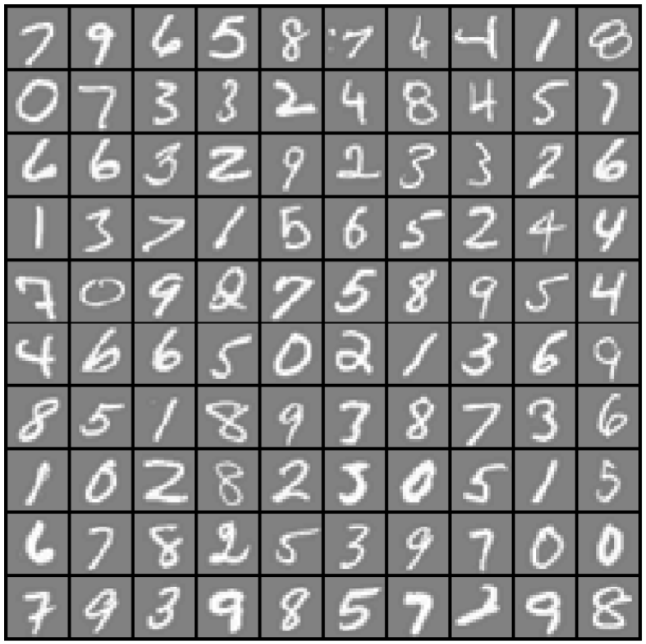


XOR function:



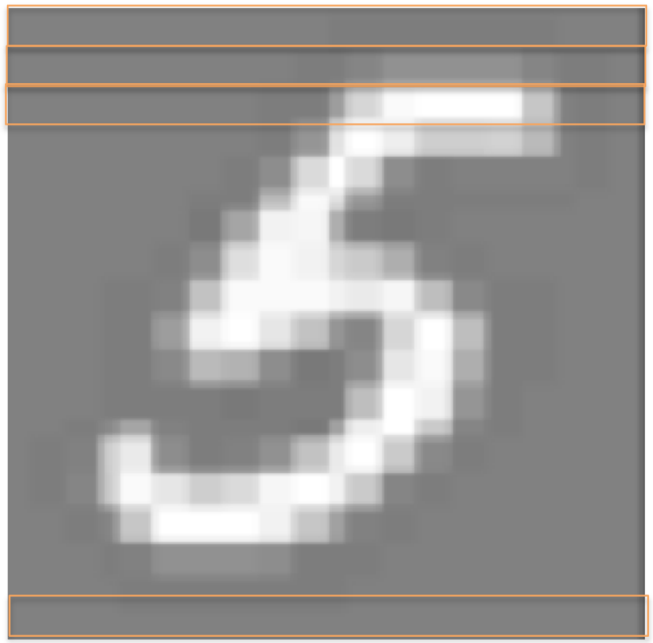


Representation of Layers I



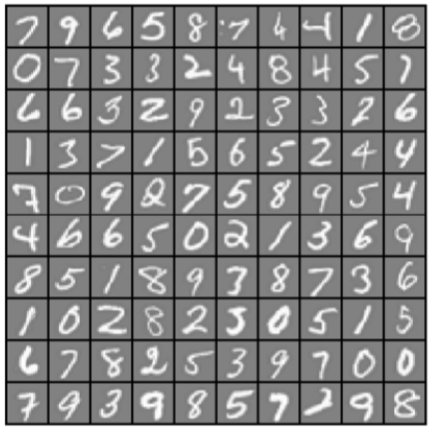
20 × 20 pixel images

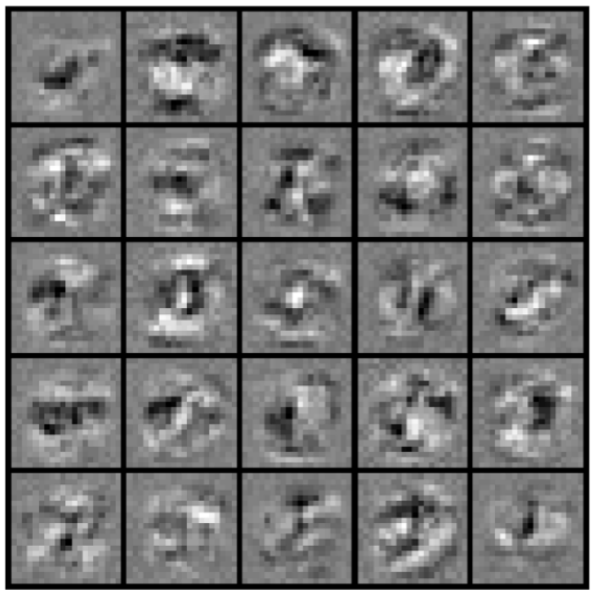
$d = 400$ 10 classes

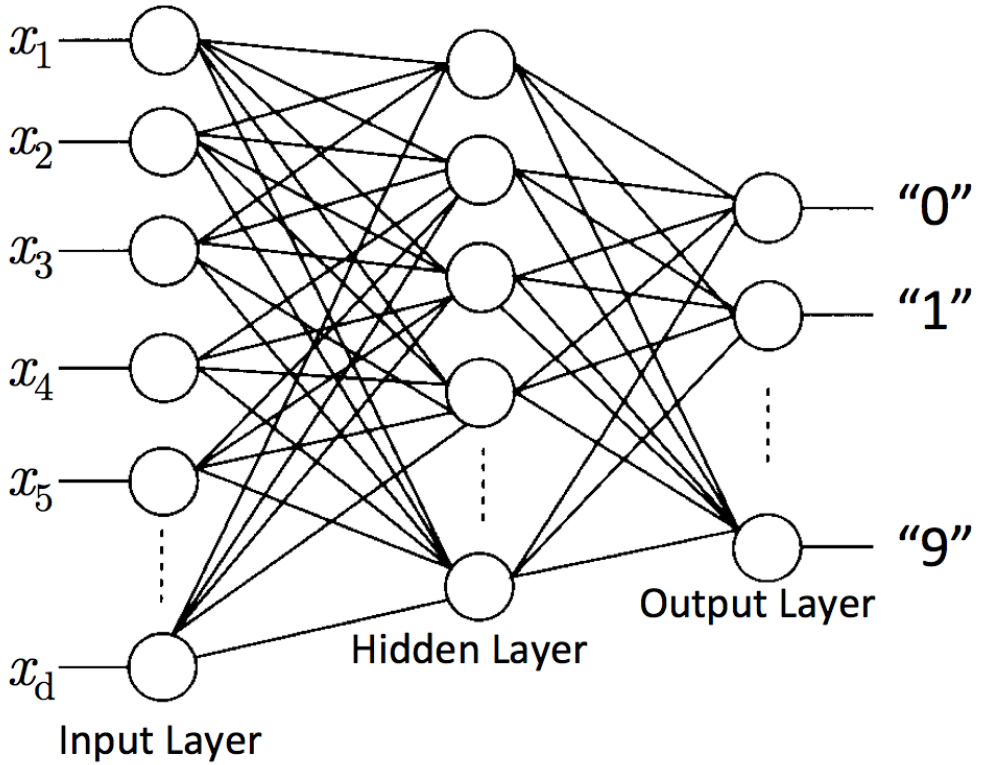


Each image is "unrolled" into a vector $\mathbf{x}$ of pixel intensities

Representation of Layers II







Visualization of Hidden Layer

Learning in NN: Backpropagation

- Similar to the perceptron learning algotihm, 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

Cost Function: Log-Loss

Logistic Regression:

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

Neural Network:

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

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

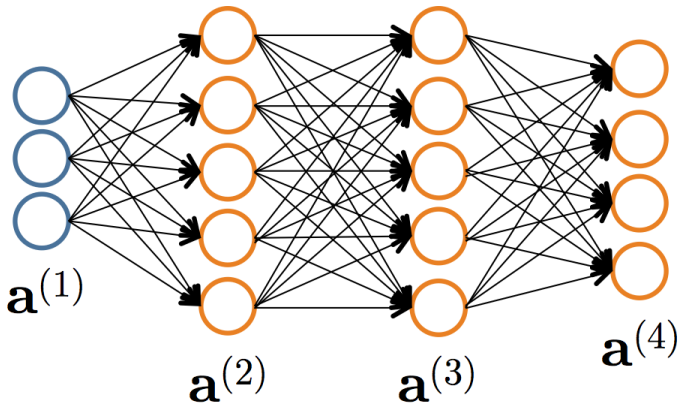
Optimizing the Neural Network

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

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

Forward Propagation

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



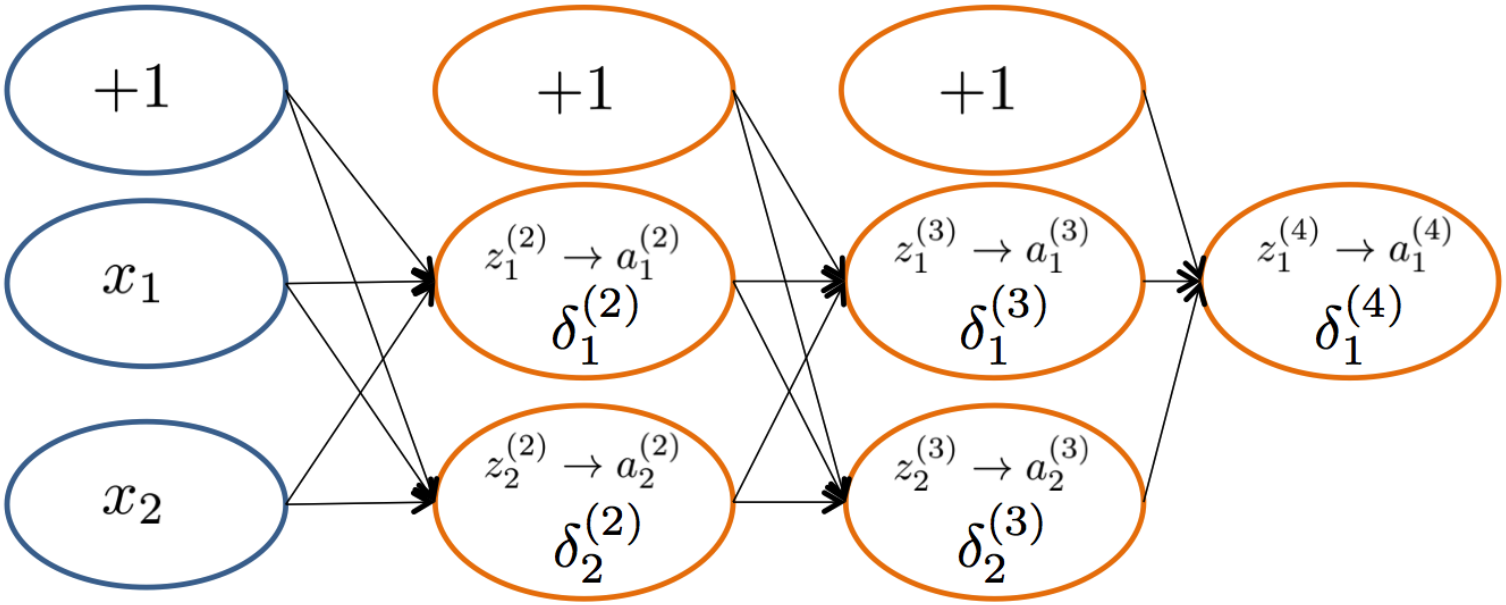
Forward Propagation

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

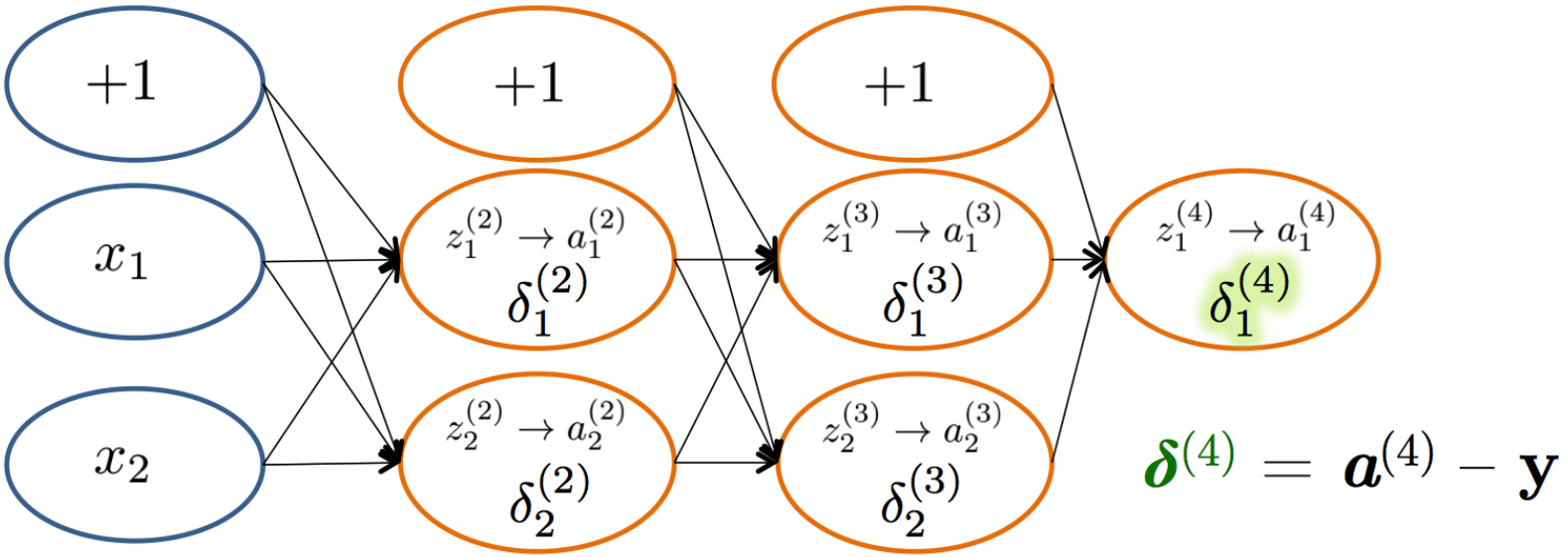
- Each hidden node j is "responsible" for some fraction of the error $\delta_j^{(l)}$ in each of the output nodes to which it connects
- $\delta_j^{(l)}$ is divided according to the strength of the connection between hidden node and the output node
- Then, the "blame" is propagated back to provide the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

Backpropagation Intuition II



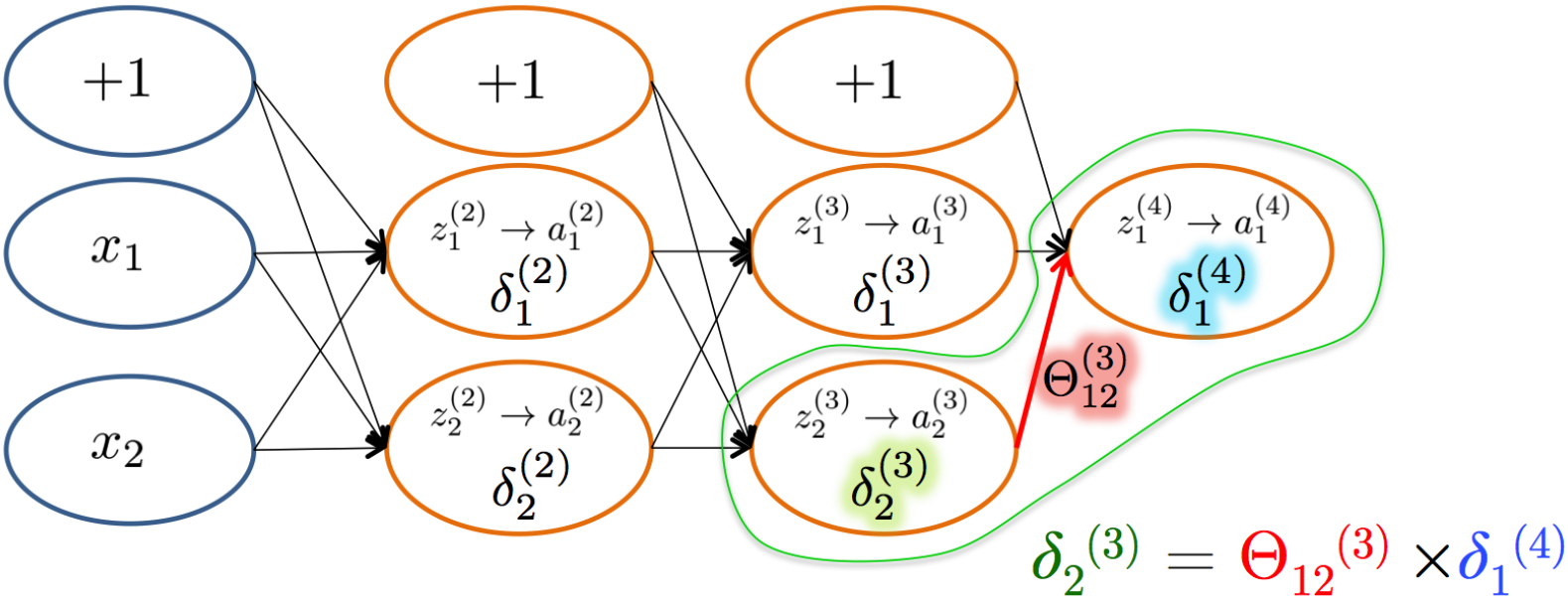
- $\delta_j^{(l)} =$ "error" of node j in layer l
- Formally, $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(\mathbf{x}_i)$,
  where $\text{cost}(\mathbf{x}_i) = y_i \log h_\theta(\mathbf{x}_i) + (1 - y_i) \log(1 - h_\theta(\mathbf{x}_i))$

Backpropagation Intuition III



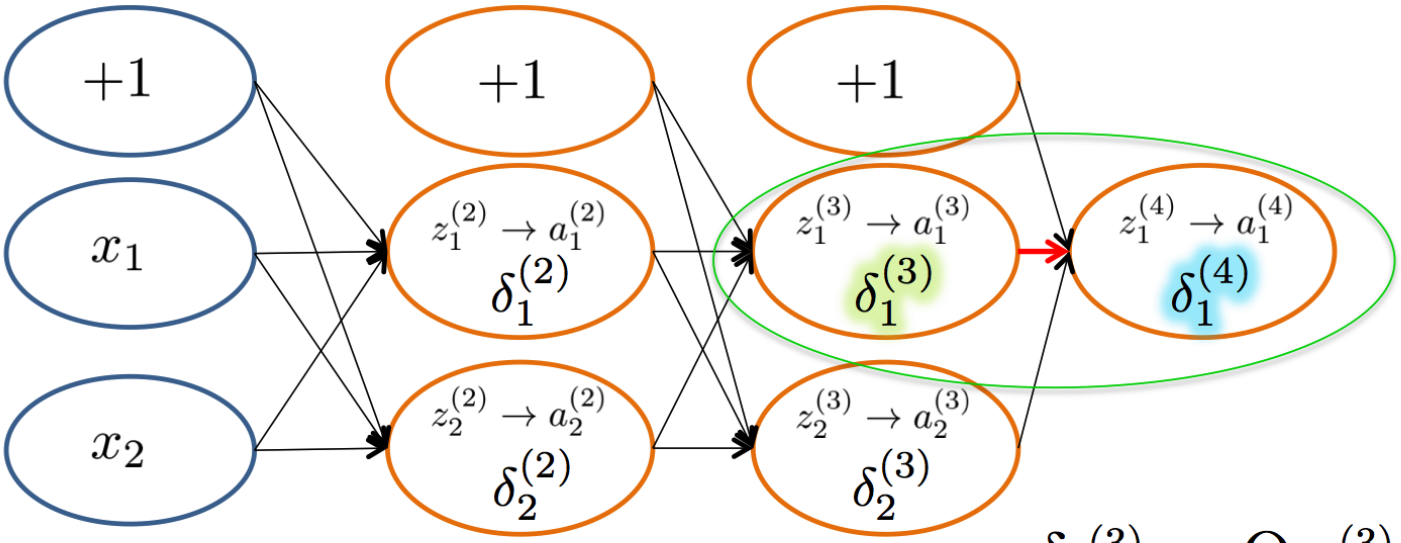
- $\delta_j^{(l)}$ = "error" of node j in layer l
- Formally, $\delta_j^{(l)} = \dfrac{\partial}{\partial z_j^{(l)}} \mathrm{cost}(\mathbf{x}_i)$,
  where $\mathrm{cost}(\mathbf{x}_i) = y_i \log h_\theta(\mathbf{x}_i) + (1 - y_i) \log(1 - h_\theta(\mathbf{x}_i))$

Backpropagation Intuition IV



- $\delta_j^{(l)}$ = "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 V

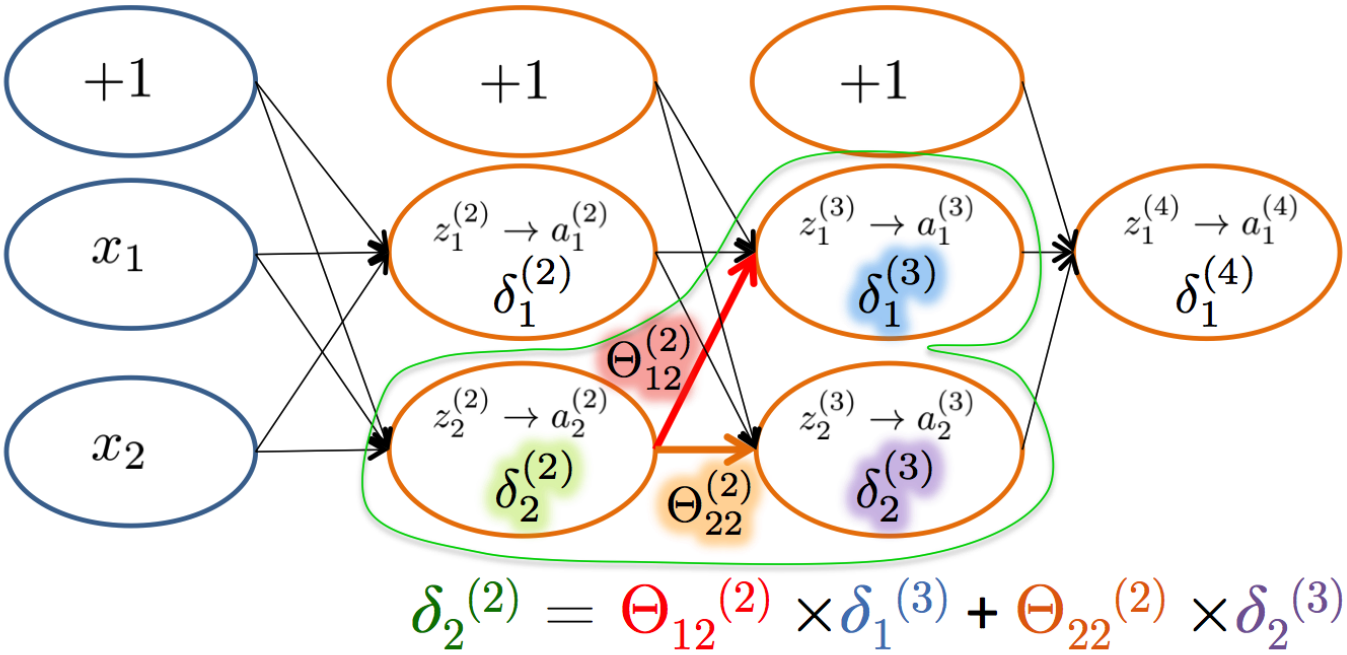


$$\delta_2{}^{(3)} = \Theta_{12}{}^{(3)} \times \delta_1{}^{(4)}$$

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

- $\delta_j^{(l)}$ = "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 VI

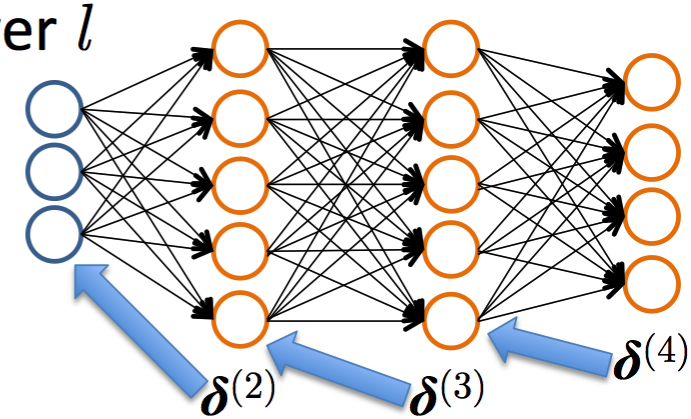


- $\delta_j^{(l)}$ = "error" of node j in layer l
- Formally, $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(\mathbf{x}_i)$,
  where $\text{cost}(\mathbf{x}_i) = y_i \log h_\theta(\mathbf{x}_i) + (1 - y_i) \log(1 - h_\theta(\mathbf{x}_i))$

Backpropagation: Gradient Computation

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

(#layers $L = 4$)



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 $\boldsymbol{\delta}^{(1)}$)

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

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

$$\frac{\partial}{\partial \theta_{ij}^{(l)}} \hat{R}(\theta) = a_j^{(l)} \delta_i^{(l+1)}$$

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

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

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

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

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

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

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

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

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

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

Backprop issues

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