# Machine Learning
# DSECL ZG565

Dr. Monali Mavani

BITS Pilani
Pilani Campus

These slides are prepared by the instructor, with grateful acknowledgement of Tom Mitchell, Andrew Ng< Prof. Chetana, Prof. Vimal and many others who made their course materials freely available online.

# Session Content

- Perceptron (Chapter 4 Tom Mitchell)
- Neural Network Architecture (Andrew Ng Notes and Chapter 4 Tom Mitchell)
- Back propagation Algorithm (Andrew Ng Notes)

# Neural Networks

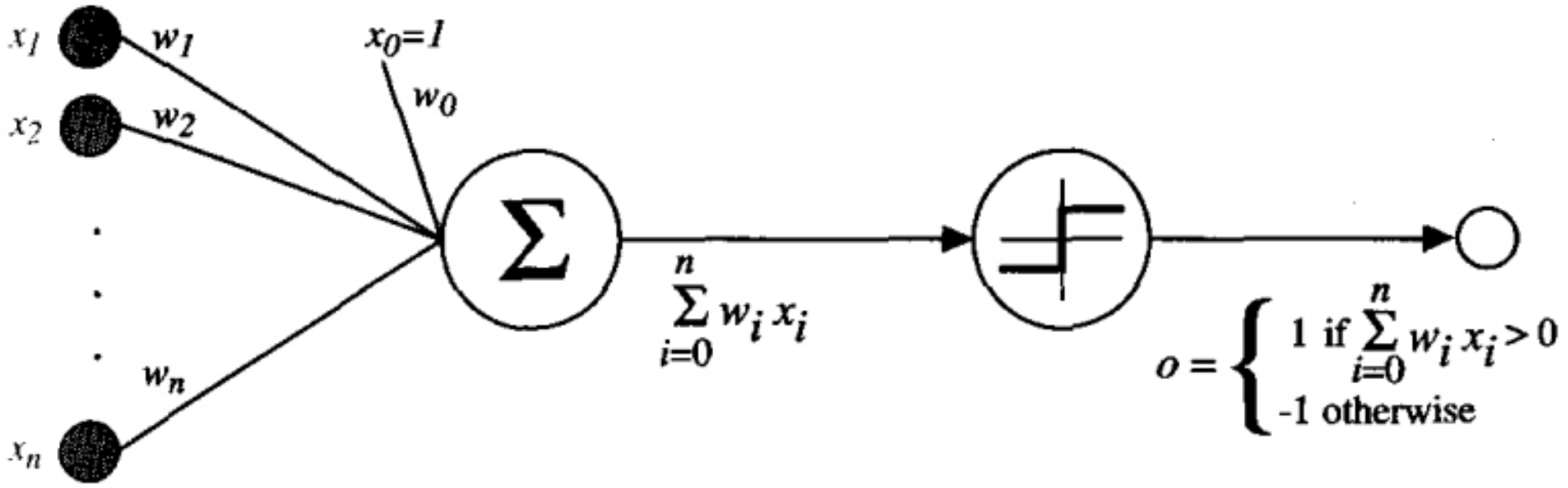
- Origins: Algorithms that try to mimic the brain.
- Very widely used in 80s and early 90s; popularity diminished in late 90s.
- Recent resurgence: State-of-the-art technique for many applications
- Artificial neural networks are not nearly as complex or intricate as the actual brain structure

Based on slide by Andrew Ng

# When to use Neural Network

- Input is high-dimensional discrete or real-valued (e.g. raw sensor input)
- Output is discrete or real valued
- Output is a vector of values
- Possibly noisy data
- Form of target function is unknown
- Human readability of result is unimportant

# Perceptron

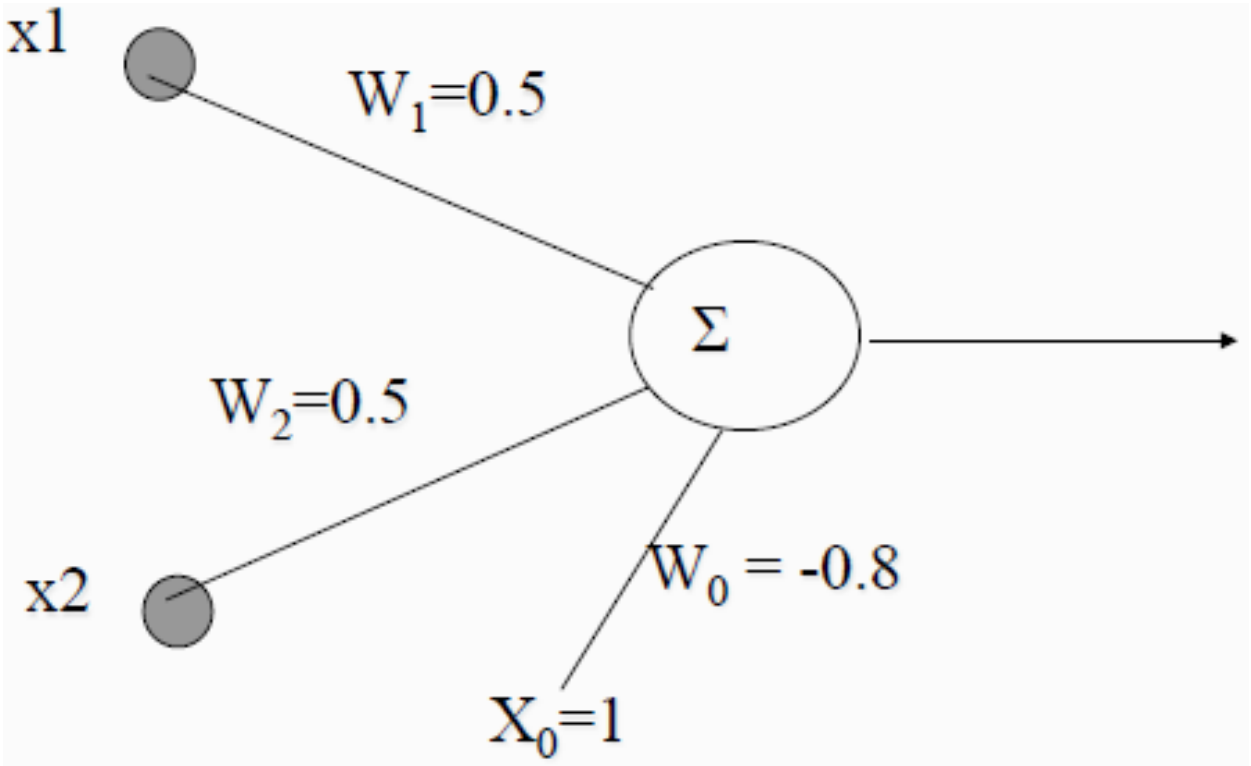



$$o(\vec{x}) = sgn(\vec{w} \cdot \vec{x})$$

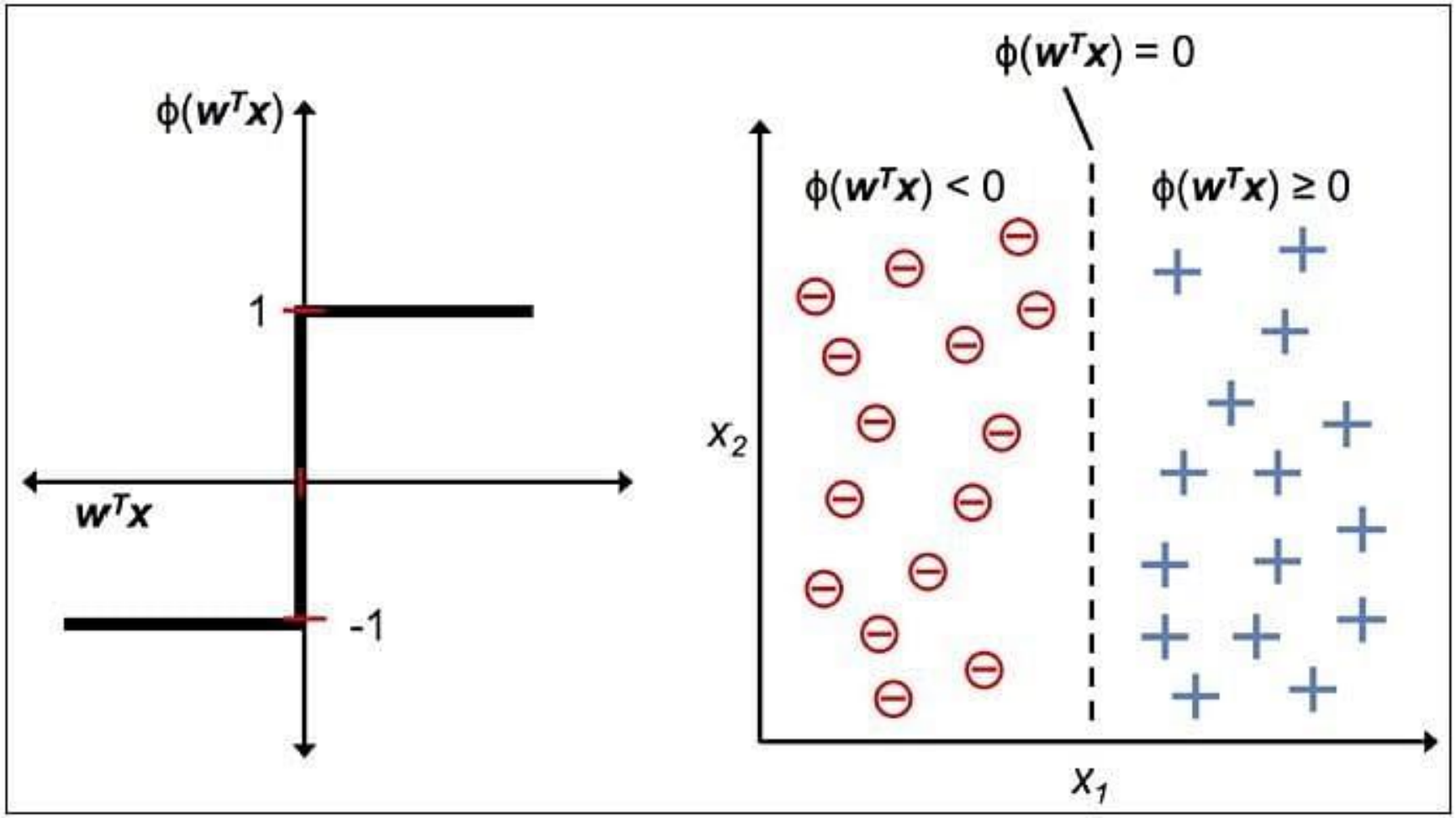
$$sgn(y) = \begin{cases} 1 \text{ if } y > 0 \\ -1 \text{ otherwise} \end{cases}$$

- A perceptron is s unit that takes a vector of real-valued inputs, calculates a linear combination of these inputs, then outputs a 1 if the result is greater than some threshold and -1 otherwise
- The quantity $(-w_0)$ is a threshold that the weighted combination of inputs $w_1x_1 + \ldots + w_nx_n$ must surpass in order for the perceptron to output a 1.

# AND Network



# Perceptron Decision Surface

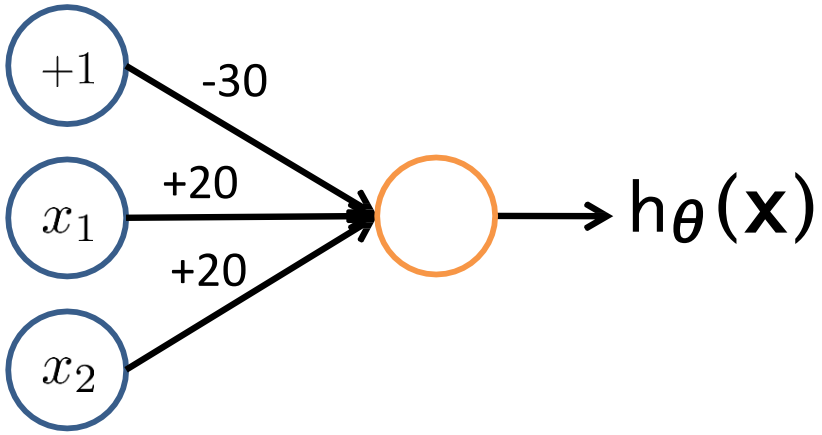


# Representing Boolean Functions

**Simple example: AND**

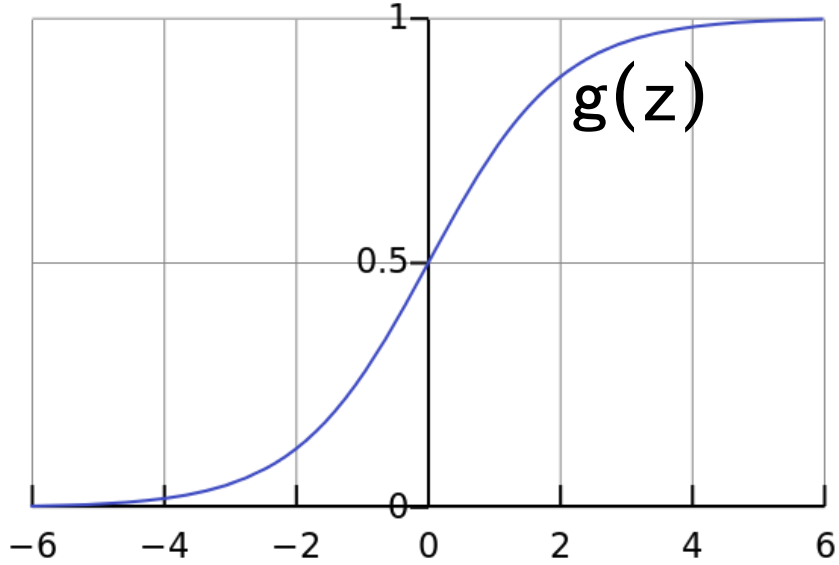
$x_1, x_2 \in \{0, 1\}$

$y = x_1 \text{ AND } x_2$




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

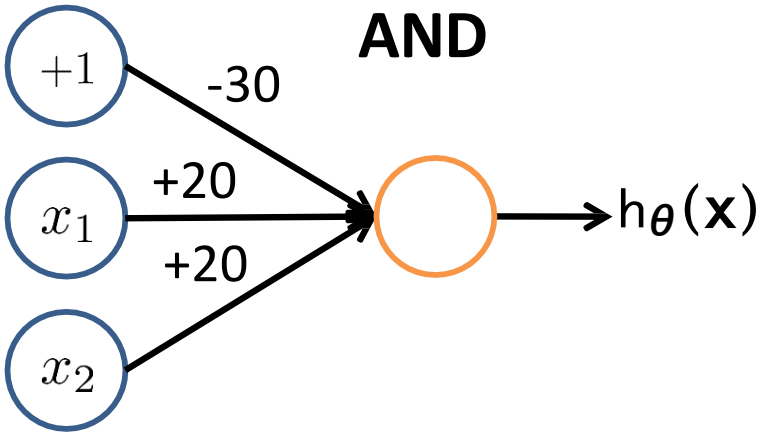
Logistic / Sigmoid Function

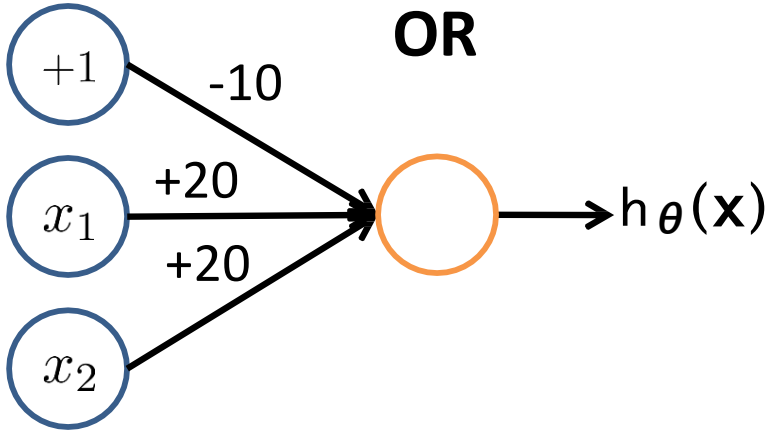


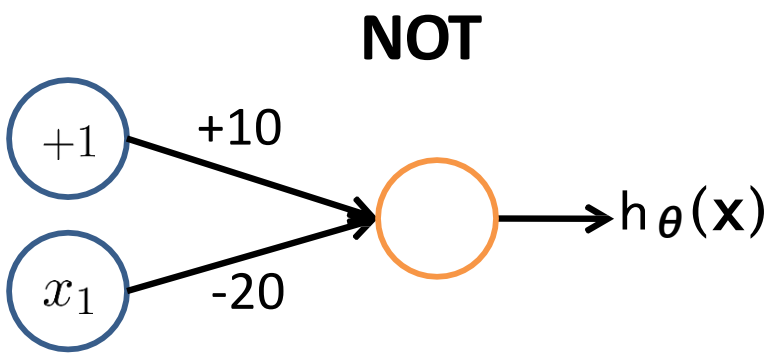

| $x_1$ | $x_2$ | $h_\Theta(\mathbf{x})$ |
|---|---|---|
| 0 | 0 | $g(-30) \approx 0$ |
| 0 | 1 | $g(-10) \approx 0$ |
| 1 | 0 | $g(-10) \approx 0$ |
| 1 | 1 | $g(10) \approx 1$ |

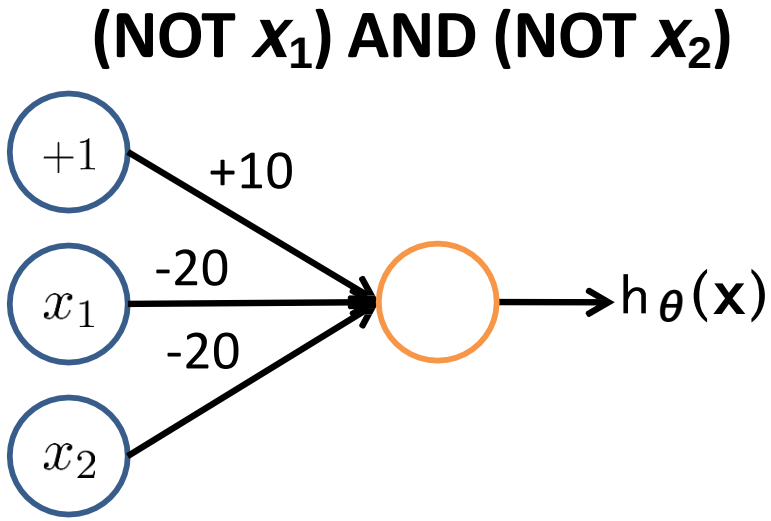
Based on slide and example by Andrew Ng

# Representing Boolean Functions



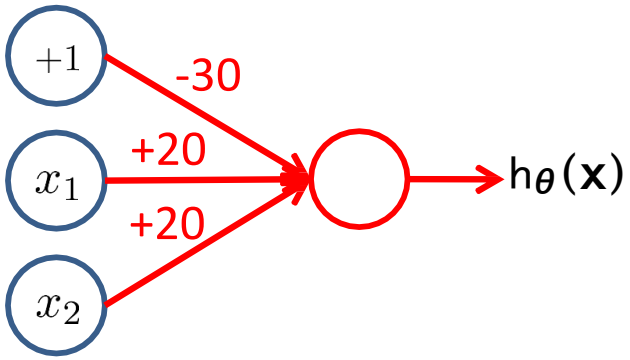



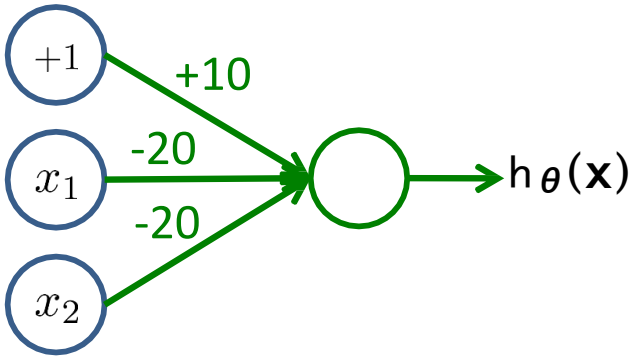
# Combining Representations to Create Non-Linear Functions



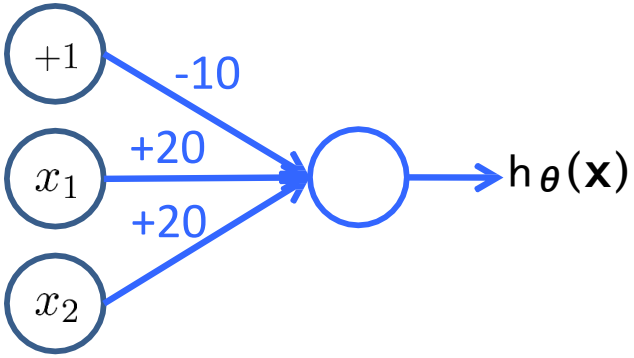
AND




(NOT $x_1$) AND (NOT $x_2$)




OR






Based on example by Andrew Ng

# Training a Perceptron

. • **Method 1: Perceptron training rule**

• **Method 2: Delta rule : LMS and gradient descent.**

# Method 1: perceptron training rule

- weight update incrementally after each individual training sample

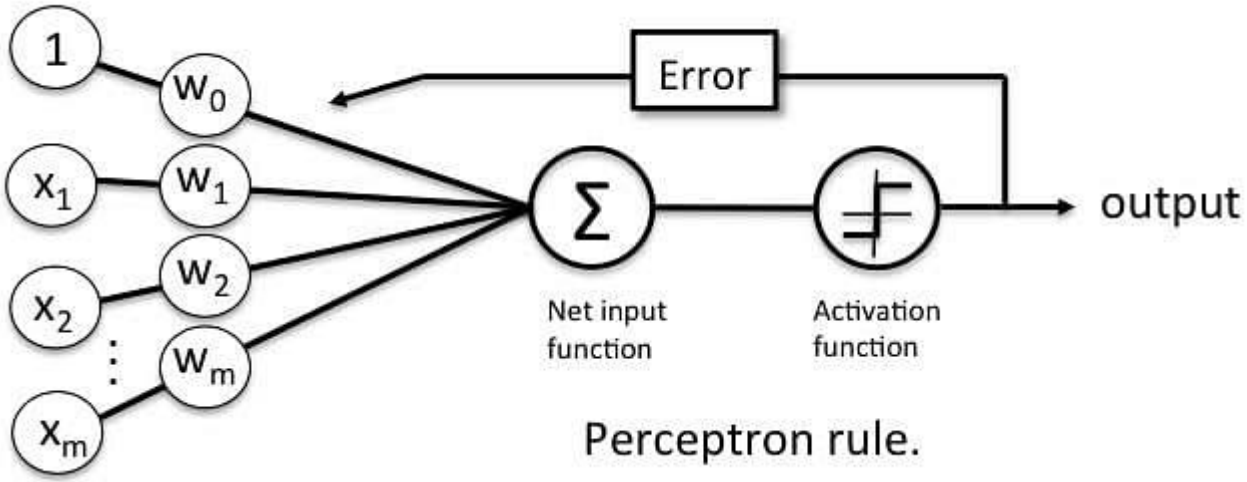



Figure Credit: https://sebastianraschka.com/Articles/2015_singlelayer_neurons.html#artificial-neurons-and-the-mcculloch-pitts-model

# Method 1: perceptron training rule



$$o(x_1, \ldots, x_n) = \begin{cases} 1, & \text{if } \omega_o + \omega_1 x_1 + \cdots + \omega_n x_n > 0 \\ -1, & \text{else} \end{cases}$$

$$o(\vec{x}) = sgn(\vec{w} \cdot \vec{x})$$

$$sgn(y) = \begin{cases} 1 \text{ if } y > 0 \\ -1 \text{ otherwise} \end{cases}$$

$$w_i \leftarrow w_i + \Delta w_i$$

$$\Delta w_i = \eta(t - o)x_i$$

Do until converge
  For each $x$ in $D$
    For each $\omega_i$
      $\omega_i \leftarrow \omega_i + \Delta\omega_i$

- $t$: target value
- $o$: current prediction
- $\eta$: learning rate. small value.

# Method 2: Delta rule : LMS and gradient descent.



- Also known as Adaptive Linear Neuron (adaline), Widrow-Hoff rule

- training an unthresholded perceptron; that is, a linear unit for which the output o is given by

$$o(\vec{x}) = \vec{w} \cdot \vec{x}$$

- We can define a cost function and use gradient descent optimization to update the weights.

Figure Credit: https://sebastianraschka.com/Articles/2015_singlelayer_neurons.html#artificial-neurons-and-the-mcculloch-pitts-model

# Method 2: Delta rule : LMS and gradient descent.

- Linear unit: $o(\vec{x}) = \vec{w} \cdot \vec{x}$

$$o_l(\boldsymbol{x}) = \omega_o + \omega_1 x_1 + \cdots + \omega_n x_n$$

- Define Error: $E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$ ← Least squares method.

- Gradient Decent:

$$\nabla E(\vec{w}) \equiv \left[ \frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots, \frac{\partial E}{\partial w_n} \right]$$

Do until converge
  For each $d$ in $D$
    For each $\omega_i$
      $\omega_i \leftarrow \omega_i + \Delta\omega_i$

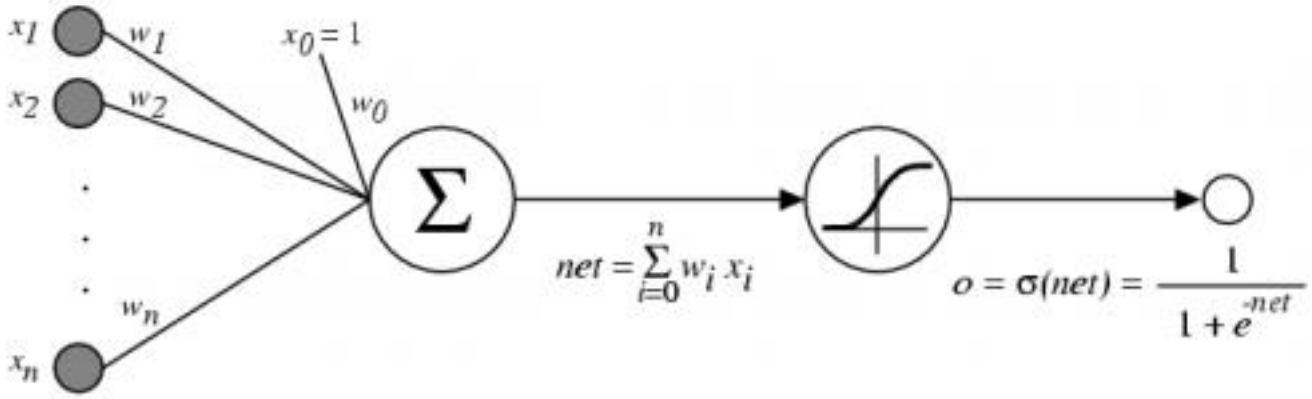
$$\vec{w} \leftarrow \vec{w} + \Delta\vec{w}$$

$$\Delta\vec{w} = -\eta \nabla E(\vec{w})$$

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

# MULTILAYER NETWORKS AND THE BACKPROPAGATION ALGORITHM

# Perceptron: Sigmoid Function




$\sigma(x)$ is the sigmoid function

$$\frac{1}{1+e^{-x}}$$

$$\begin{aligned} g'(z) &= \frac{d}{dz}\frac{1}{1+e^{-z}} \\ &= \frac{1}{(1+e^{-z})^2}(e^{-z}) \\ &= \frac{1}{(1+e^{-z})} \cdot \left(1 - \frac{1}{(1+e^{-z})}\right) \\ &= g(z)(1-g(z)). \end{aligned}$$

Nice property: $\frac{d\sigma(x)}{dx} = \sigma(x)(1-\sigma(x))$

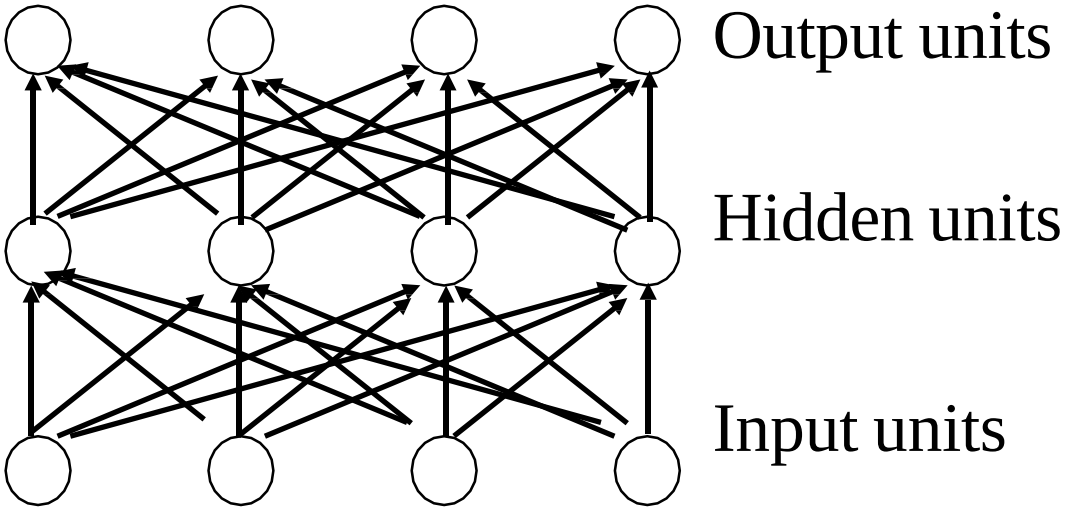
We can derive gradient decent rules to train

- One sigmoid unit
- *Multilayer networks* of sigmoid units → Backpropagation

# Multilayer network

- Single perceptrons can only express linear decision surfaces.
- In contrast, the kind of multilayer networks learned by the FORWARD and BACKPROPAGATION algorithm are capable of expressing a rich variety of nonlinear decision surfaces

# 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**

Based on slide by T. Finin, M. desJardins, L Getoor, R. Par

# Hidden Units

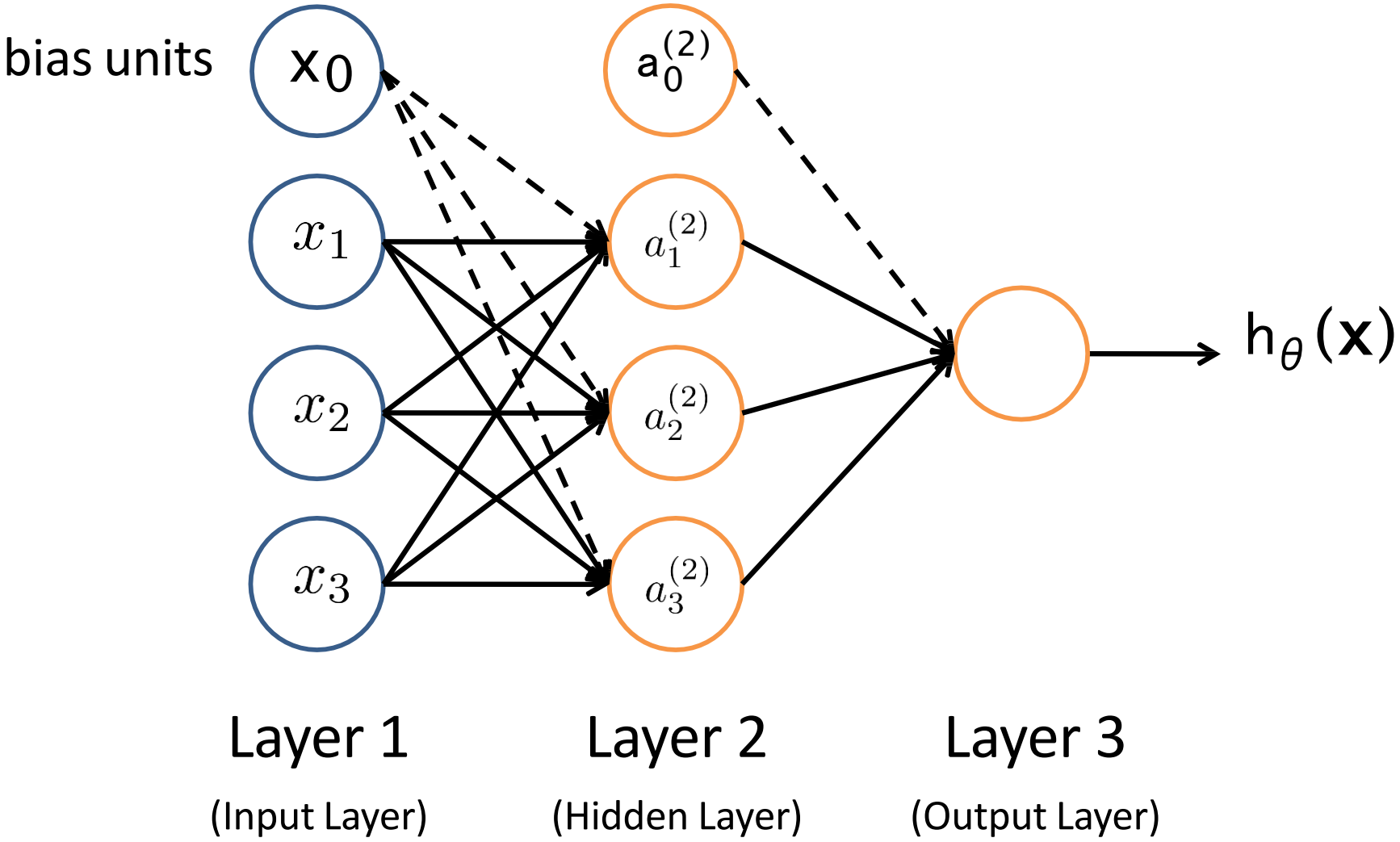
- Hidden units are nodes that are situated between the input nodes and the output nodes.

- Given too many hidden units, a neural net will simply memorize the input patterns.

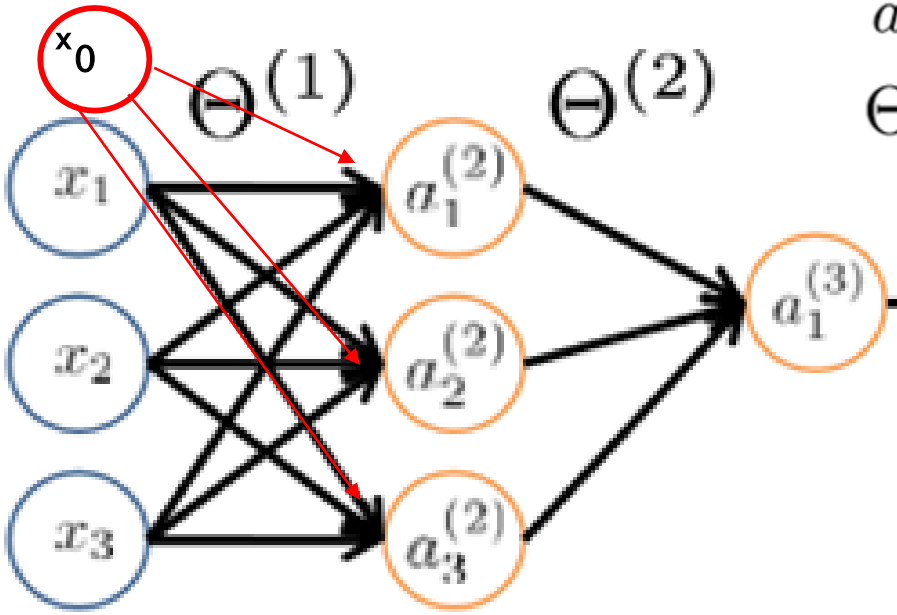
- Given too few hidden units, the network may not be able to represent all the necessary representations.

# Example forward pass – simple NN

# Neural Network



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

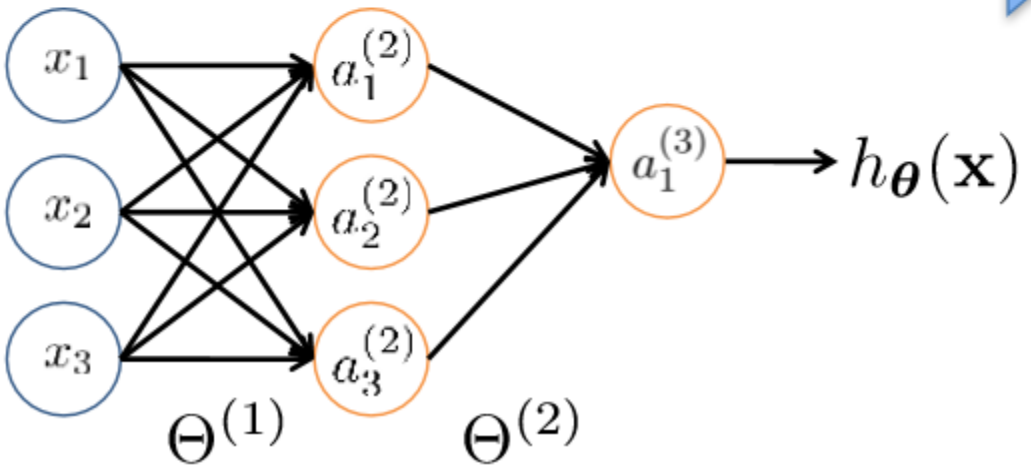
# Vectorization

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

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

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

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




Feed-Forward Steps:

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

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

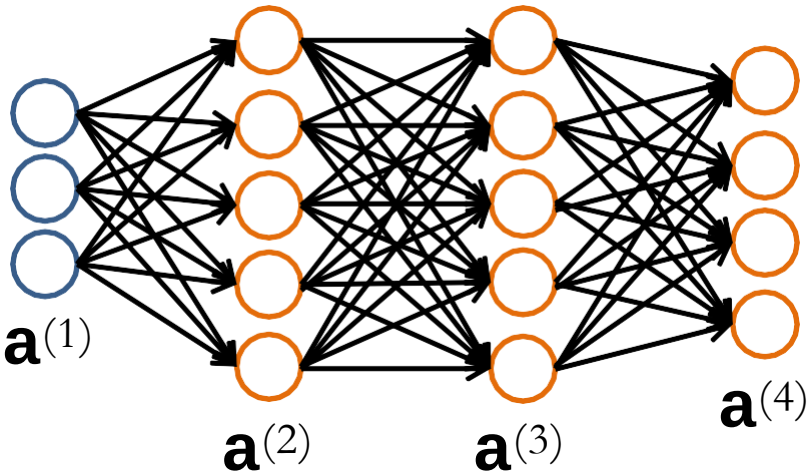
Add $a_0^{(2)} = 1$

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

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

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

Based on slide by Andrew Ng

# Therefore ANN is specified by



an architecture: a set of neurons and links connecting neurons. Each link has a weight,

a neuron model: the information processing unit of the NN,

a learning algorithm: used for training the NN by modifying the weights in order to solve the particular learning task correctly on the set of training examples.

# Cost Function

(9.1 NN video of Andrew Ng)

## 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:

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

$$J(\Theta) = -\frac{1}{n}\left[\sum_{i=1}^{n}\sum_{k=1}^{K} y_{ik}\log(h_\Theta(\mathbf{x}_i))_k + (1-y_{ik})\log\left(1-(h_\Theta(\mathbf{x}_i))_k\right)\right]$$

$$+\frac{\lambda}{2n}\sum_{l=1}^{L-1}\sum_{i=1}^{s_{l-1}}\sum_{j=1}^{s_l}\left(\Theta_{ji}^{(l)}\right)^2$$

$k^{\text{th}}$ class: true, predicted
not $k^{\text{th}}$ class: true, predicted

# Optimizing the Neural Network

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

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

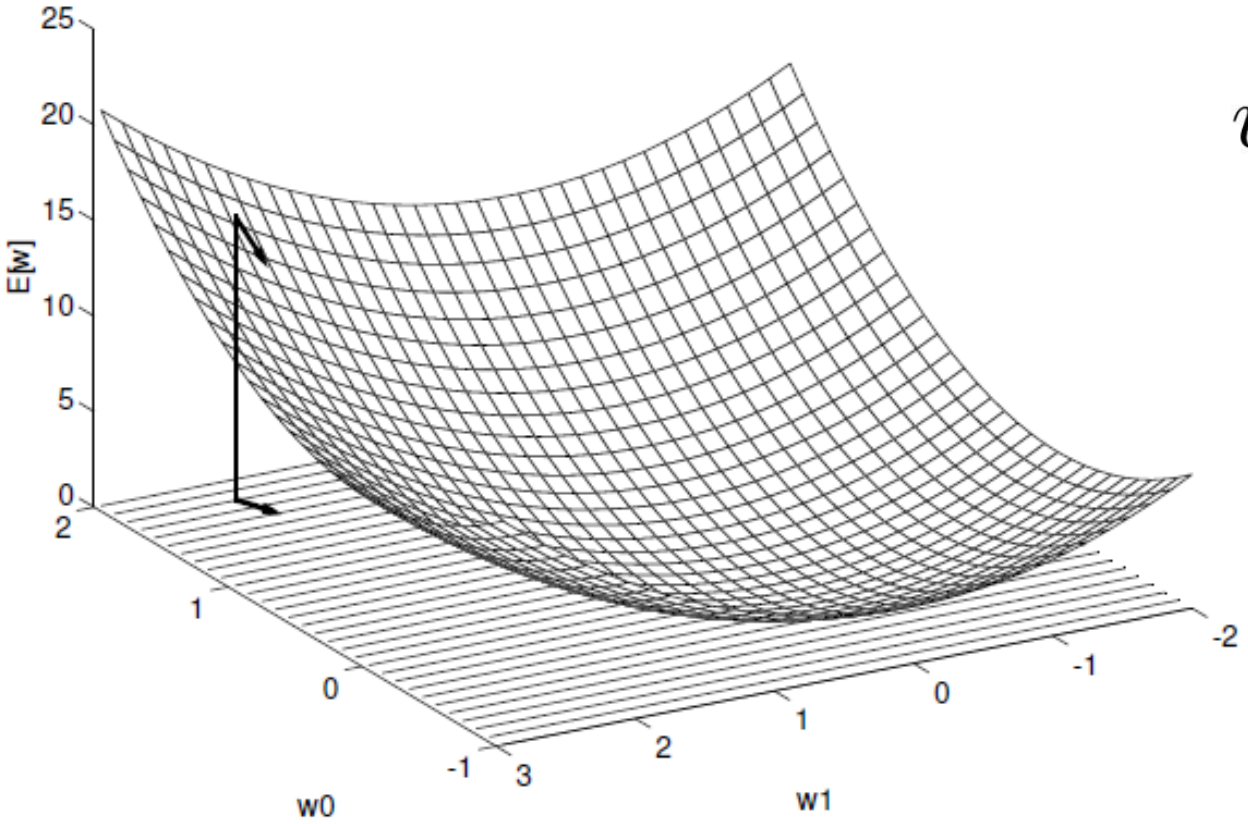
$J(\Theta)$ is not convex, so GD on a neural net yields a local optimum

- But, tends to work well in practice

Need code to compute:

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

# Gradient Descent




$$w^{[l]} = w^{[l]} - \alpha \frac{\partial C}{\partial w^{[l]}}$$

$$b^{[l]} = b^{[l]} - \alpha \frac{\partial C}{\partial b^{[l]}}$$

# 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

Based on slide by T. Finin, M. desJardins, L Getoor, R. Par

# Chain rule



## Chain Rule of Differentiation (reminder)

- The rate of change of a function of a function is the multiple of the derivatives of those functions.

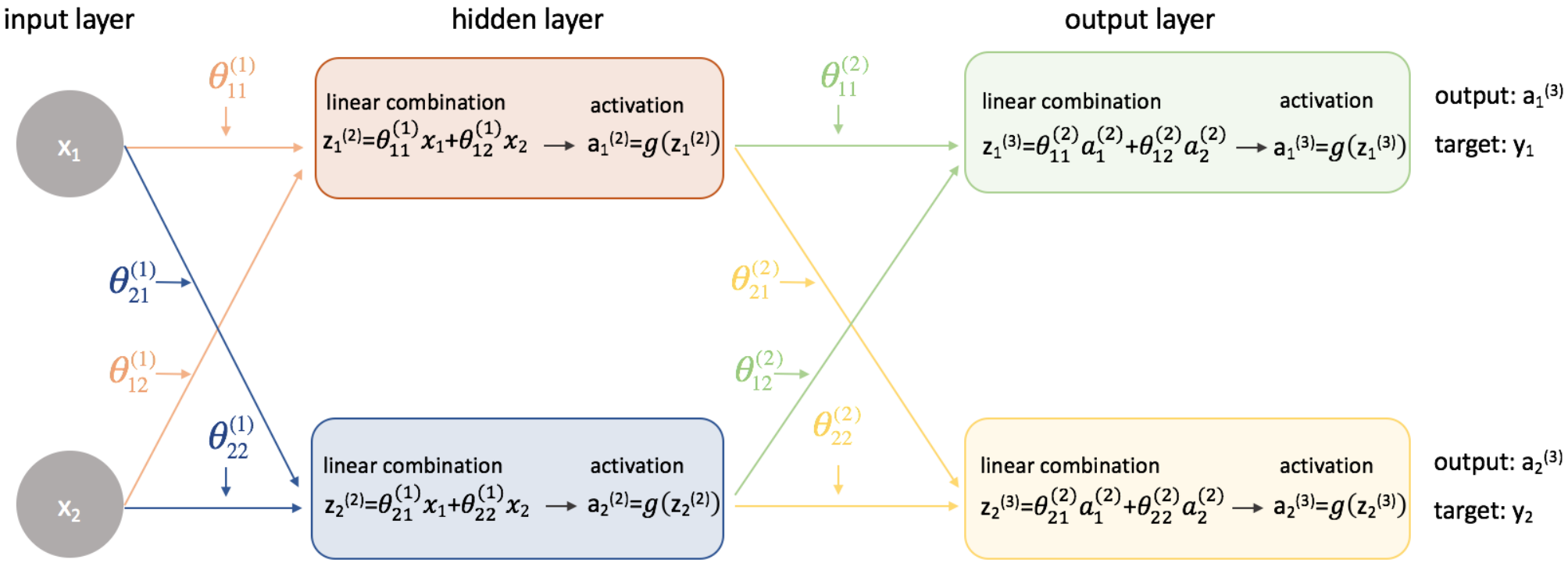
- You have probably learned this in school (nothing new here)

$$\frac{\partial}{\partial x} f(g(x)) = g'(x) \cdot f'(g)$$

$$\frac{\partial}{\partial x} f(g(h(i(j(k(x)))))) = \frac{\partial k}{\partial x}\frac{\partial j}{\partial k}\frac{\partial i}{\partial j}\frac{\partial h}{\partial i}\frac{\partial g}{\partial h}\frac{\partial f}{\partial g}$$

# Backprop equation simple layer

# Example – back prop equations for 3 layer NN



# Theta (1) parameters

$$\frac{\partial J(\theta)}{\partial \theta_{11}^{(1)}} = \left(\frac{\partial J(\theta)}{\partial a_1^{(3)}}\right)\left(\frac{\partial a_1^{(3)}}{\partial z_1^{(3)}}\right)\left(\frac{\partial z_1^{(3)}}{\partial a_1^{(2)}}\right)\left(\frac{\partial a_1^{(2)}}{\partial z_1^{(2)}}\right)\left(\frac{\partial z_1^{(2)}}{\partial \theta_{11}^{(1)}}\right) + \left(\frac{\partial J(\theta)}{\partial a_2^{(3)}}\right)\left(\frac{\partial a_2^{(3)}}{\partial z_2^{(3)}}\right)\left(\frac{\partial z_2^{(3)}}{\partial a_1^{(2)}}\right)\left(\frac{\partial a_1^{(2)}}{\partial z_1^{(2)}}\right)\left(\frac{\partial z_1^{(2)}}{\partial \theta_{11}^{(1)}}\right)$$

$$\frac{\partial J(\theta)}{\partial \theta_{12}^{(1)}} = \left(\frac{\partial J(\theta)}{\partial a_1^{(3)}}\right)\left(\frac{\partial a_1^{(3)}}{\partial z_1^{(3)}}\right)\left(\frac{\partial z_1^{(3)}}{\partial a_1^{(2)}}\right)\left(\frac{\partial a_1^{(2)}}{\partial z_1^{(2)}}\right)\left(\frac{\partial z_1^{(2)}}{\partial \theta_{12}^{(1)}}\right) + \left(\frac{\partial J(\theta)}{\partial a_2^{(3)}}\right)\left(\frac{\partial a_2^{(3)}}{\partial z_2^{(3)}}\right)\left(\frac{\partial z_2^{(3)}}{\partial a_1^{(2)}}\right)\left(\frac{\partial a_1^{(2)}}{\partial z_1^{(2)}}\right)\left(\frac{\partial z_1^{(2)}}{\partial \theta_{12}^{(1)}}\right)$$

$$\frac{\partial J(\theta)}{\partial \theta_{21}^{(1)}} = \left(\frac{\partial J(\theta)}{\partial a_1^{(3)}}\right)\left(\frac{\partial a_1^{(3)}}{\partial z_1^{(3)}}\right)\left(\frac{\partial z_1^{(3)}}{\partial a_2^{(2)}}\right)\left(\frac{\partial a_2^{(2)}}{\partial z_2^{(2)}}\right)\left(\frac{\partial z_2^{(2)}}{\partial \theta_{21}^{(1)}}\right) + \left(\frac{\partial J(\theta)}{\partial a_2^{(3)}}\right)\left(\frac{\partial a_2^{(3)}}{\partial z_2^{(3)}}\right)\left(\frac{\partial z_2^{(3)}}{\partial a_2^{(2)}}\right)\left(\frac{\partial a_2^{(2)}}{\partial z_2^{(2)}}\right)\left(\frac{\partial z_2^{(2)}}{\partial \theta_{21}^{(1)}}\right)$$

$$\frac{\partial J(\theta)}{\partial \theta_{22}^{(1)}} = \left(\frac{\partial J(\theta)}{\partial a_1^{(3)}}\right)\left(\frac{\partial a_1^{(3)}}{\partial z_1^{(3)}}\right)\left(\frac{\partial z_1^{(3)}}{\partial a_2^{(2)}}\right)\left(\frac{\partial a_2^{(2)}}{\partial z_2^{(2)}}\right)\left(\frac{\partial z_2^{(2)}}{\partial \theta_{22}^{(1)}}\right) + \left(\frac{\partial J(\theta)}{\partial a_2^{(3)}}\right)\left(\frac{\partial a_2^{(3)}}{\partial z_2^{(3)}}\right)\left(\frac{\partial z_2^{(3)}}{\partial a_2^{(2)}}\right)\left(\frac{\partial a_2^{(2)}}{\partial z_2^{(2)}}\right)\left(\frac{\partial z_2^{(2)}}{\partial \theta_{22}^{(1)}}\right)$$

# Theta (2) parameters

$$\frac{\partial J(\theta)}{\partial \theta_{11}^{(2)}} = \left(\frac{\partial J(\theta)}{\partial a_1^{(3)}}\right)\left(\frac{\partial a_1^{(3)}}{\partial z_1^{(3)}}\right)\left(\frac{\partial z_1^{(3)}}{\partial \theta_{11}^{(2)}}\right)$$

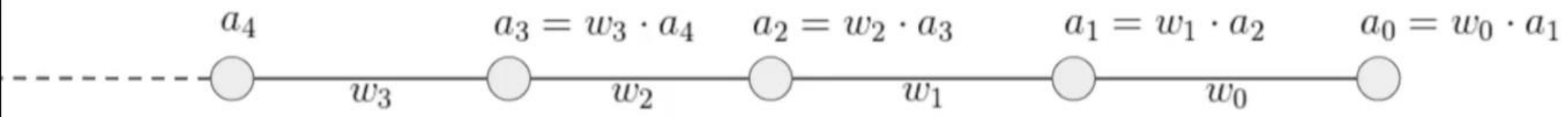
$$\frac{\partial J(\theta)}{\partial \theta_{12}^{(2)}} = \left(\frac{\partial J(\theta)}{\partial a_1^{(3)}}\right)\left(\frac{\partial a_1^{(3)}}{\partial z_1^{(3)}}\right)\left(\frac{\partial z_1^{(3)}}{\partial \theta_{12}^{(2)}}\right)$$

$$\frac{\partial J(\theta)}{\partial \theta_{21}^{(2)}} = \left(\frac{\partial J(\theta)}{\partial a_2^{(3)}}\right)\left(\frac{\partial a_2^{(3)}}{\partial z_2^{(3)}}\right)\left(\frac{\partial z_2^{(3)}}{\partial \theta_{21}^{(2)}}\right)$$

$$\frac{\partial J(\theta)}{\partial \theta_{22}^{(2)}} = \left(\frac{\partial J(\theta)}{\partial a_2^{(3)}}\right)\left(\frac{\partial a_2^{(3)}}{\partial z_2^{(3)}}\right)\left(\frac{\partial z_2^{(3)}}{\partial \theta_{22}^{(2)}}\right)$$

# Backpropogation

## Backpropagation Generalized to several layers




$$-r\frac{\partial a_0}{\partial w_0}\frac{\partial C}{\partial a_0}$$

$$-r\frac{\partial a_1}{\partial w_1}\frac{\partial a_0}{\partial a_1}\frac{\partial C}{\partial a_0}$$

$$C = (a_0 - y)^2$$

$$-r\frac{\partial a_2}{\partial w_2}\frac{\partial a_1}{\partial a_2}\frac{\partial a_0}{\partial a_1}\frac{\partial C}{\partial a_0}$$

For example, we adjust w3 as follows:

$$-r\frac{\partial a_3}{\partial w_3}\frac{\partial a_2}{\partial a_3}\frac{\partial a_1}{\partial a_2}\frac{\partial a_0}{\partial a_1}\frac{\partial C}{\partial a_0}$$

$$w_3' = w_3 - r \cdot a_4 \cdot w_2 \cdot w_1 \cdot w_0 \cdot 2(a_0 - y)$$

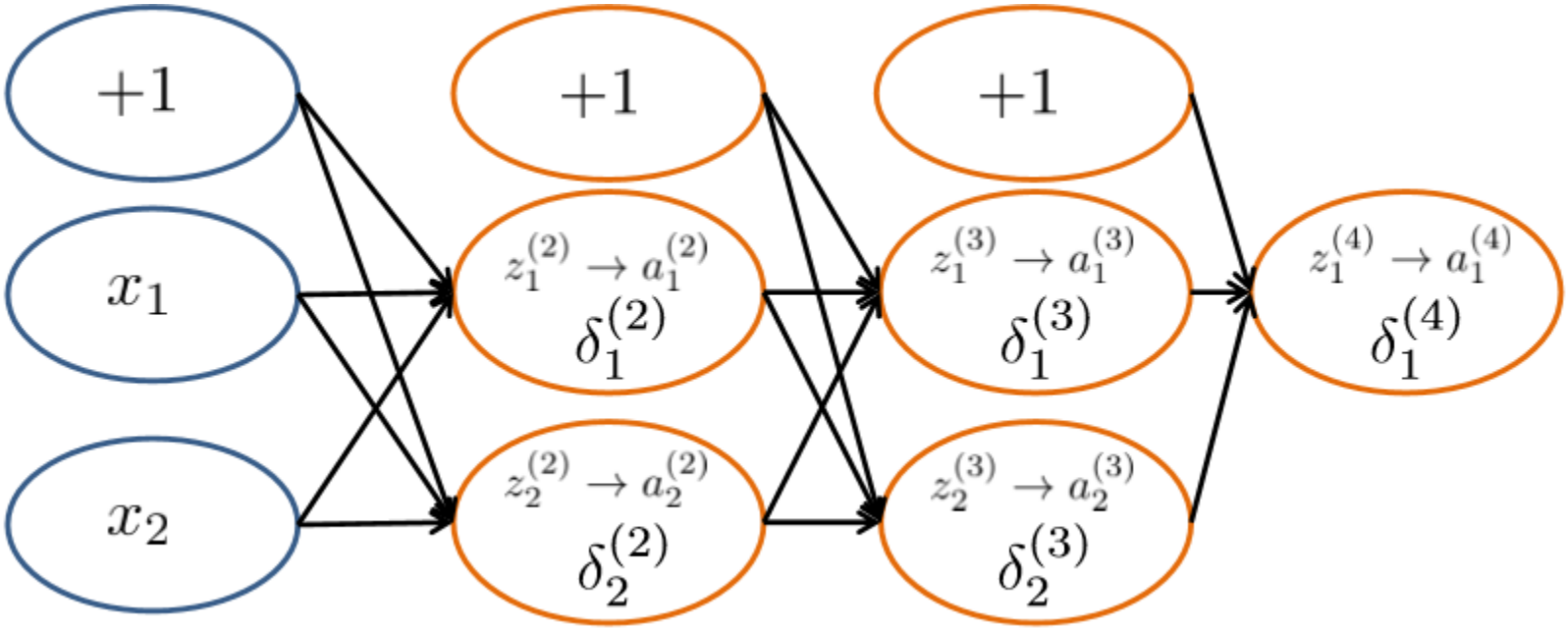
# Backpropagation Intuition

- 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

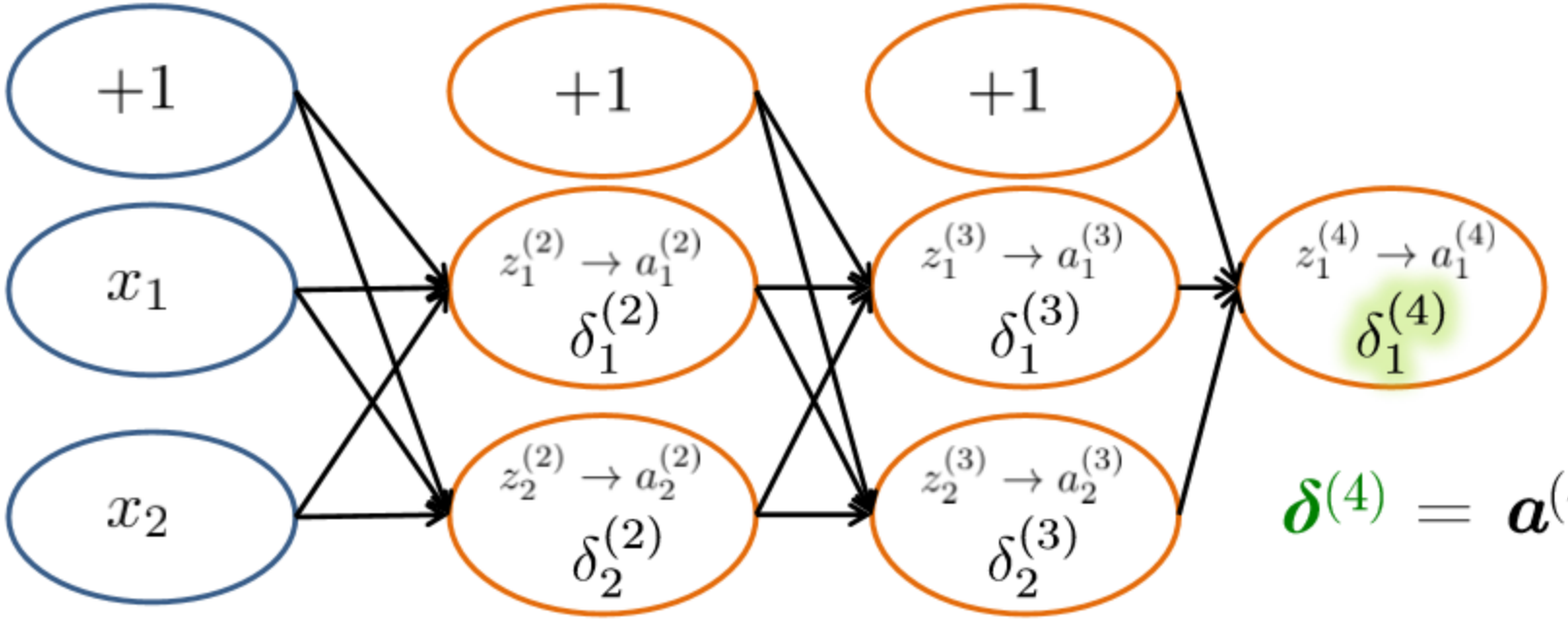



$\delta_j^{(l)}$ = "error" of node $j$ in layer $l$

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

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




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

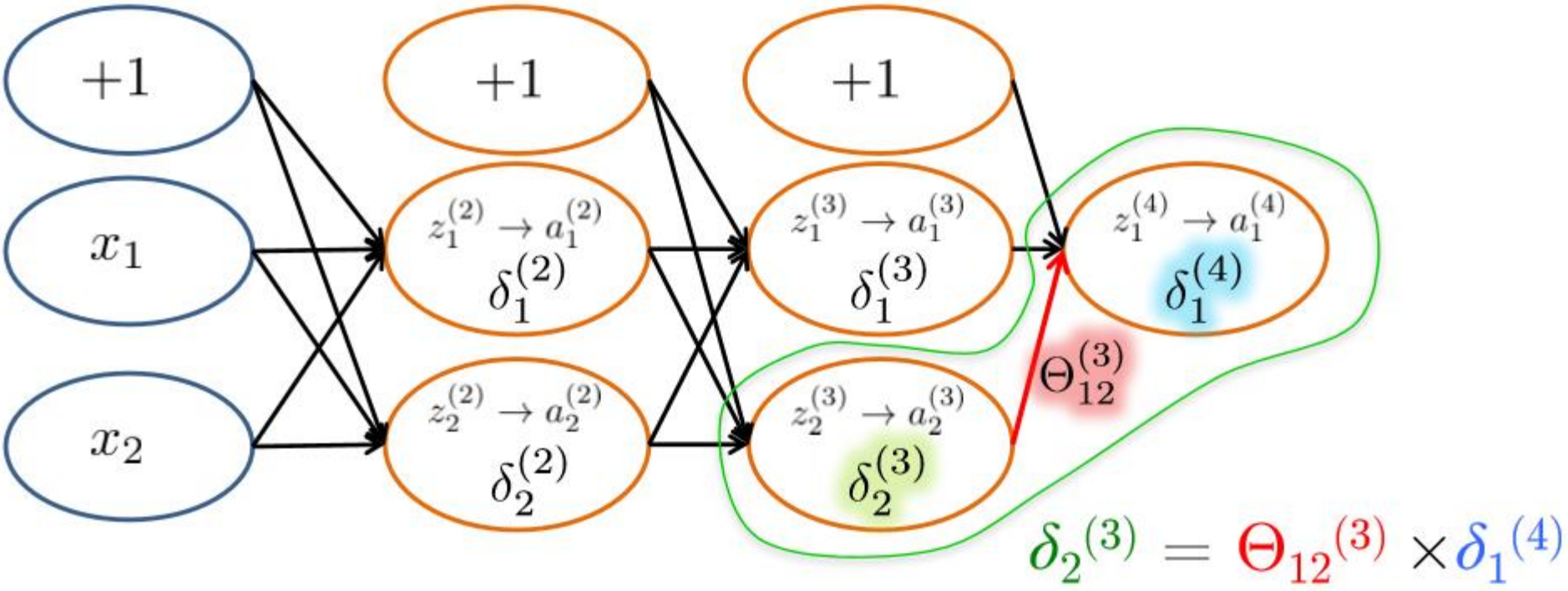
$\delta_j^{(l)}$ = "error" of node $j$ in layer $l$

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

# Derivation last layer error

# Other than last layer error derivation

# Backpropagation Intuition

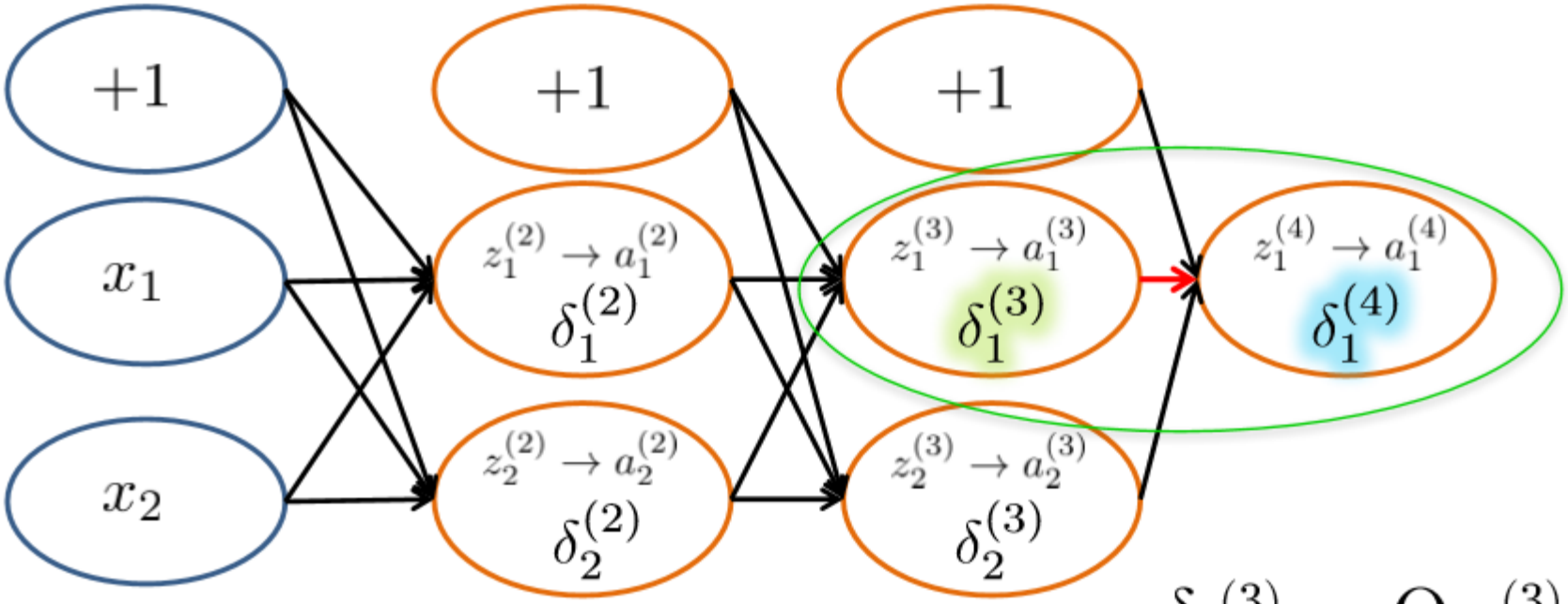



$\delta_j^{(l)}$ = "error" of node $j$ in layer $l$

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




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

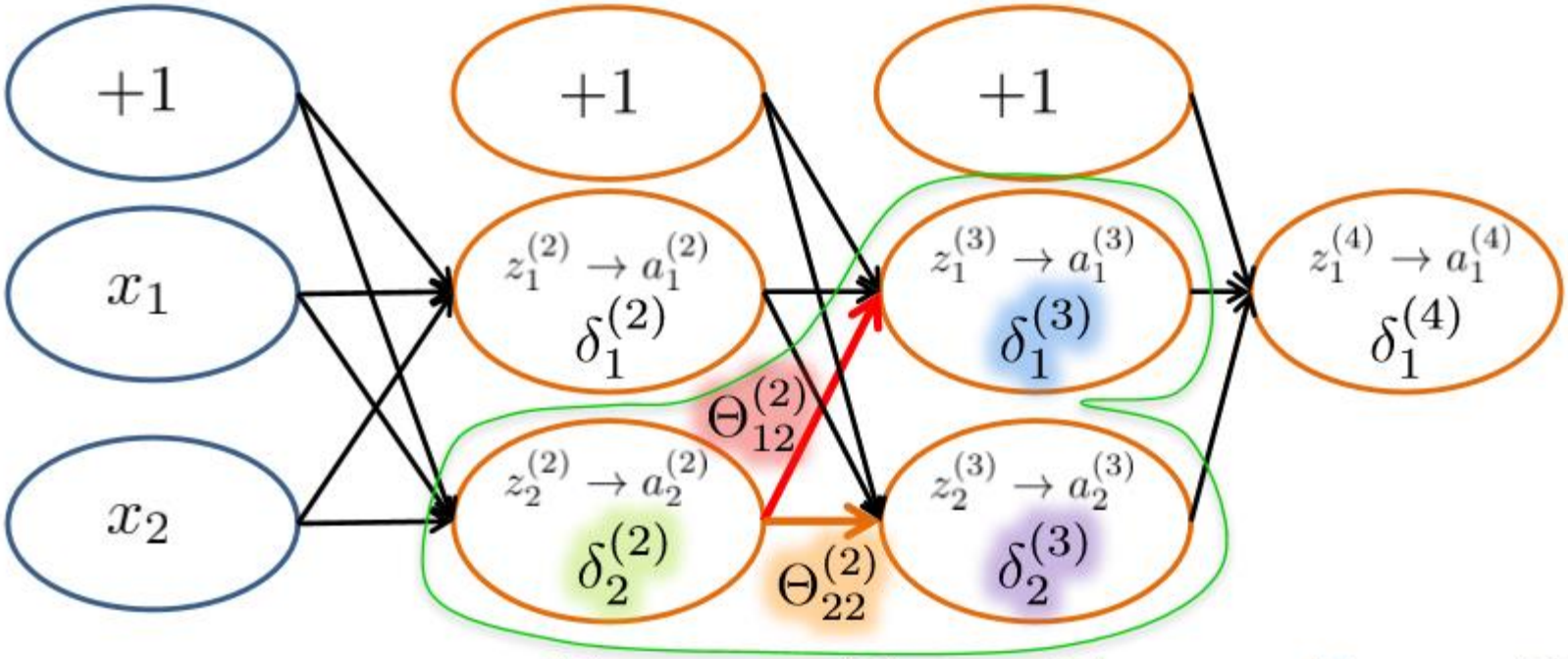
$$\delta_1^{(3)} = \Theta_{11}^{(3)} \times \delta_1^{(4)}$$

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

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

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




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

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

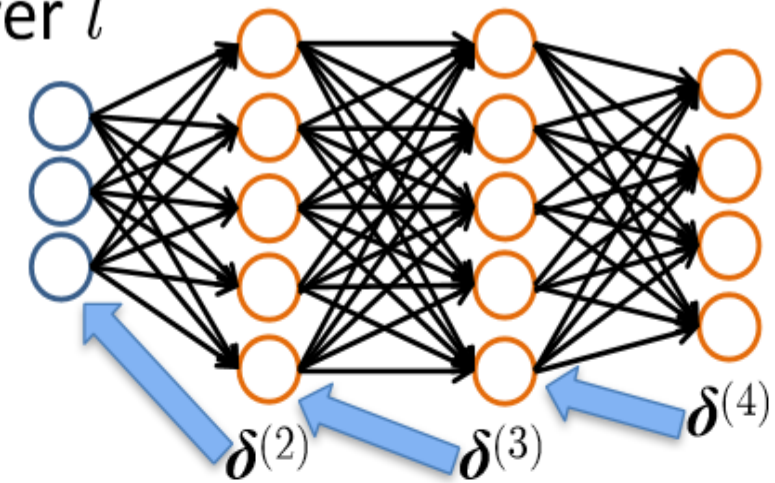
Formally, $\delta_j^{(l)} = \dfrac{\partial}{\partial z_j^{(l)}} \text{cost}(\mathbf{x}_i)$

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

# Cost function using error term

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

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

# Backpropagation + GD

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

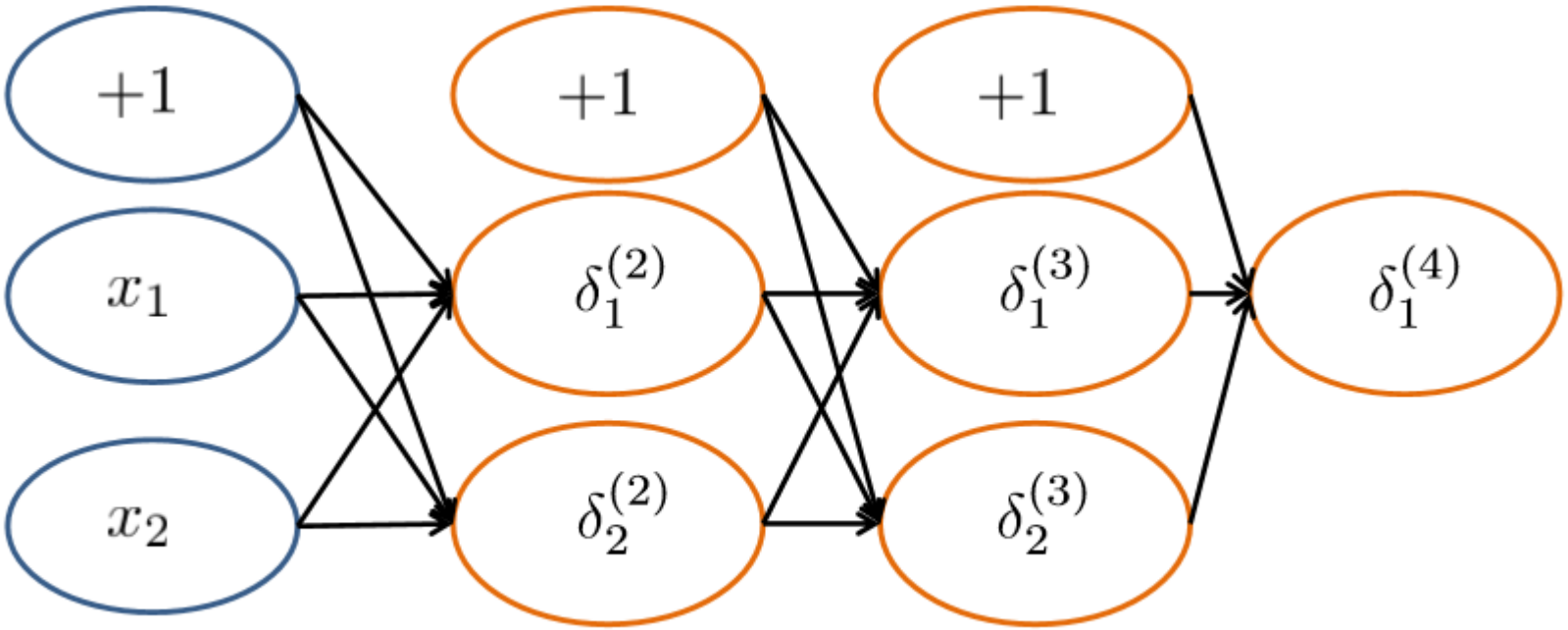
Update weights via gradient step $\Theta_{ij}^{(l)} = \Theta_{ij}^{(l)} - \alpha D_{ij}^{(l)}$

Until weights converge or max #epochs is reached

Backpropagation

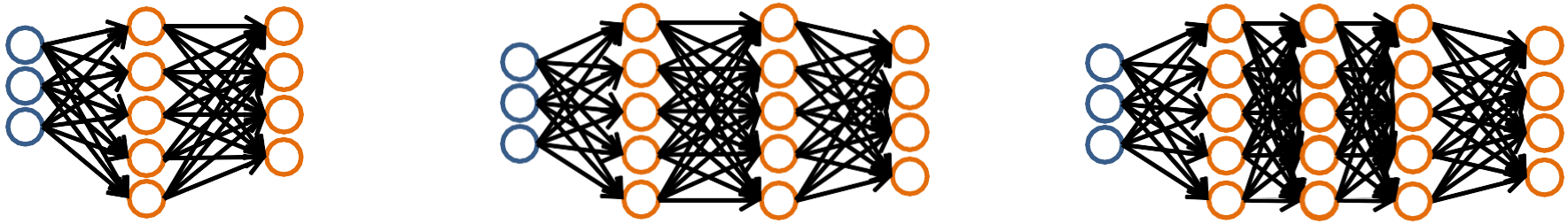
# Random Initialization

- 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



# 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)

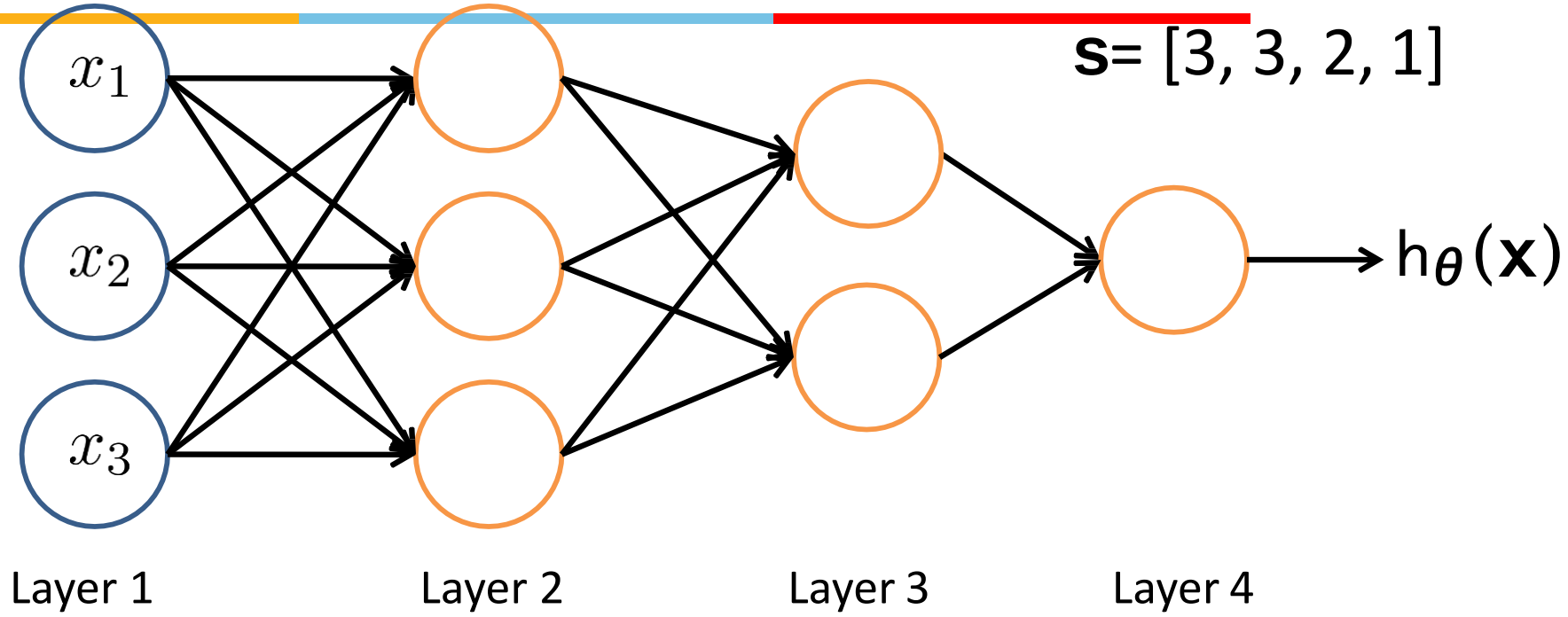
Based on slide by Andrew Ng

# 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

Based on slide by Andrew Ng

# Other Network Architectures




$L$ denotes the number of layers

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

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

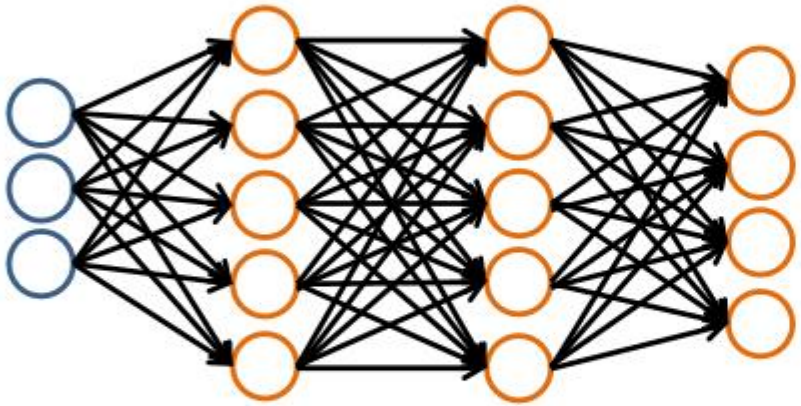
# Multiple Output Units: One-vs-Rest

Pedestrian

Car

Motorcycle

Truck



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

We want:

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

when pedestrian

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

when car

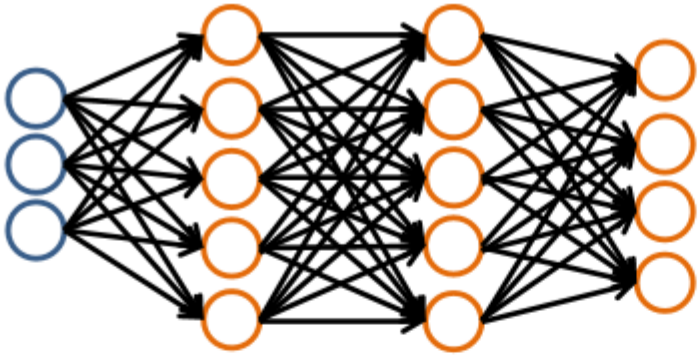
$$h_\Theta(\mathbf{x}) \approx \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

when motorcycle

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

when truck

# Multiple Output Units: One-vs-Rest



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

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

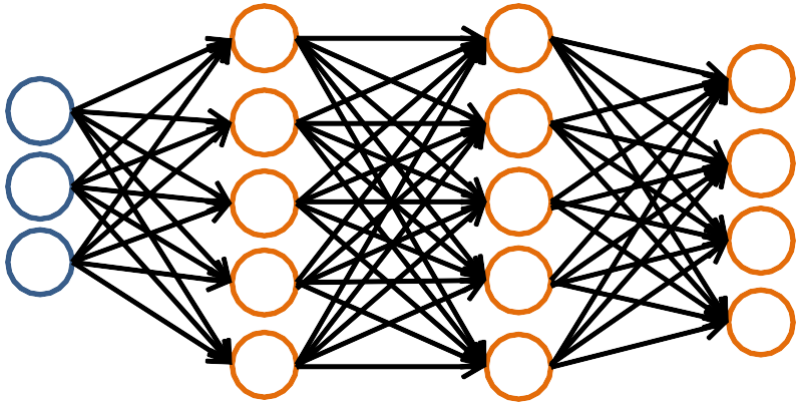
$$h_\Theta(\mathbf{x}) \approx \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$ when motorcycle

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

Based on slide by Andrew Ng

# Neural Network Classification



**Given:**

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

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

- $s_0 = d$ (# features)

Binary classification

$y$ = 0 or 1

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

Multi-class classification ($K$ classes)

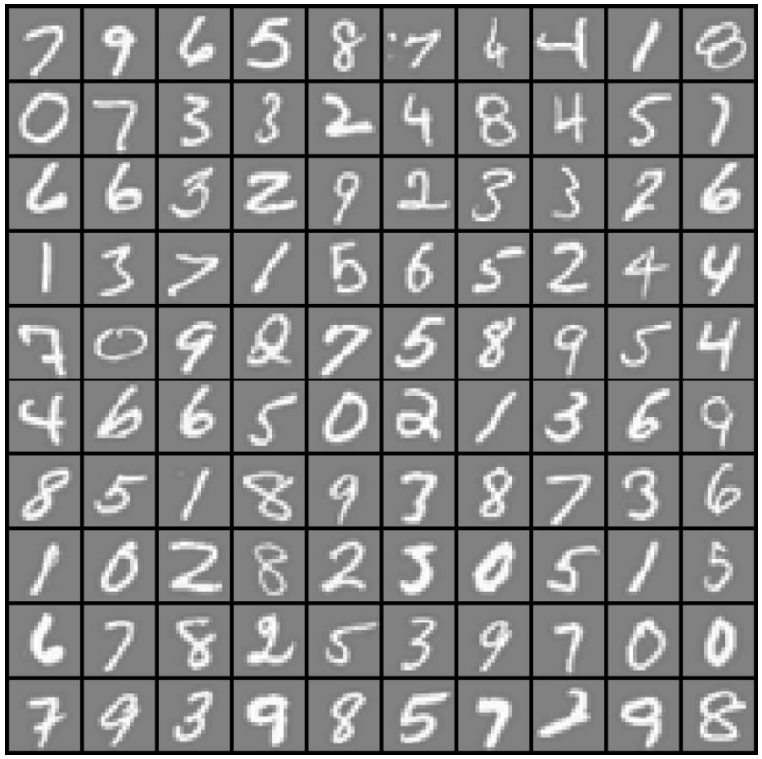
$\mathbf{y} \in \mathrm{R}^K$ e.g. $\begin{bmatrix}1\\0\\0\\0\end{bmatrix}$, $\begin{bmatrix}0\\1\\0\\0\end{bmatrix}$, $\begin{bmatrix}0\\0\\1\\0\end{bmatrix}$, $\begin{bmatrix}0\\0\\0\\1\end{bmatrix}$

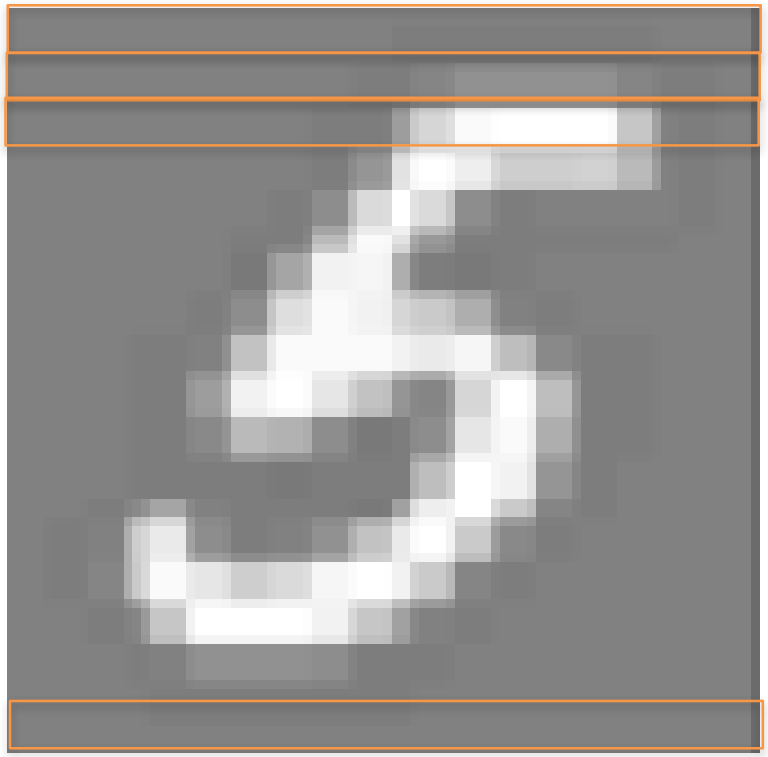
pedestrian car motorcycle truck

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

Slide by Andrew Ng

# Layering Representations





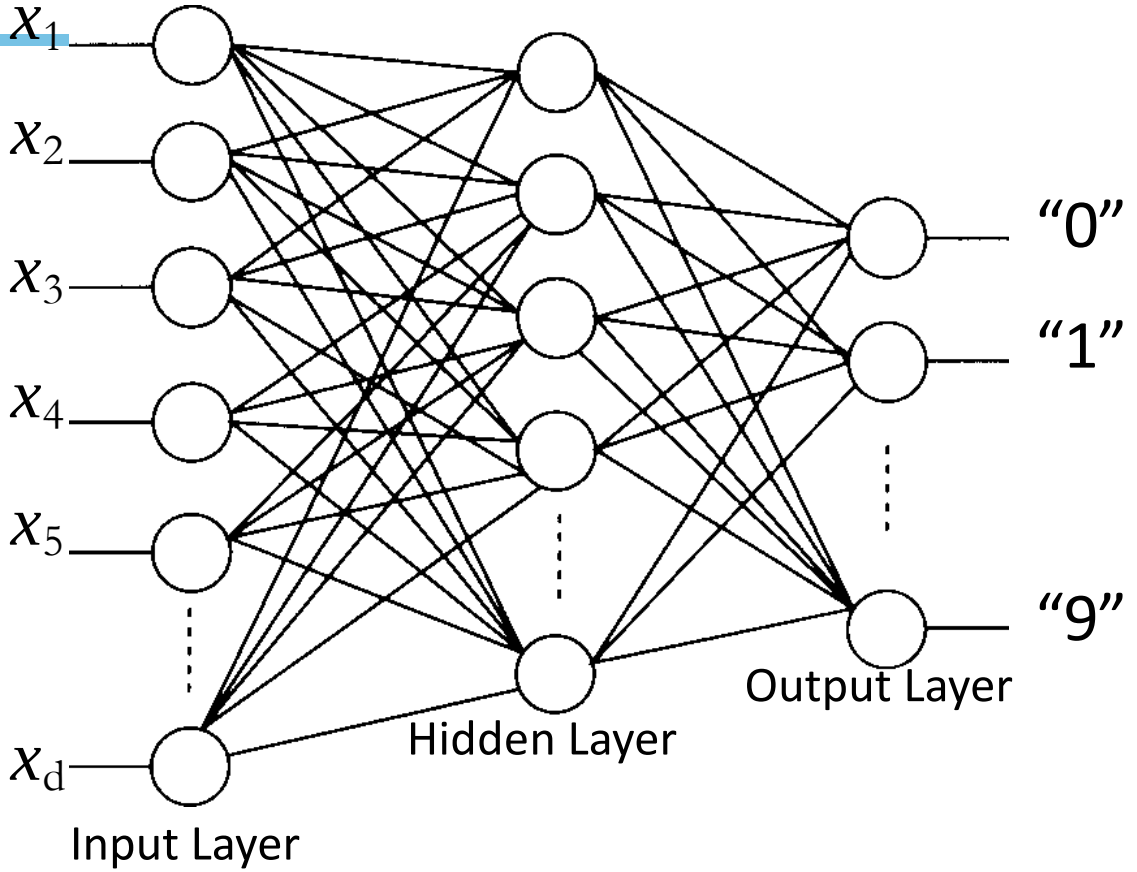
$x_1 \dots x_{20}$
$x_{21} \dots x_{40}$
$x_{41} \dots x_{60}$

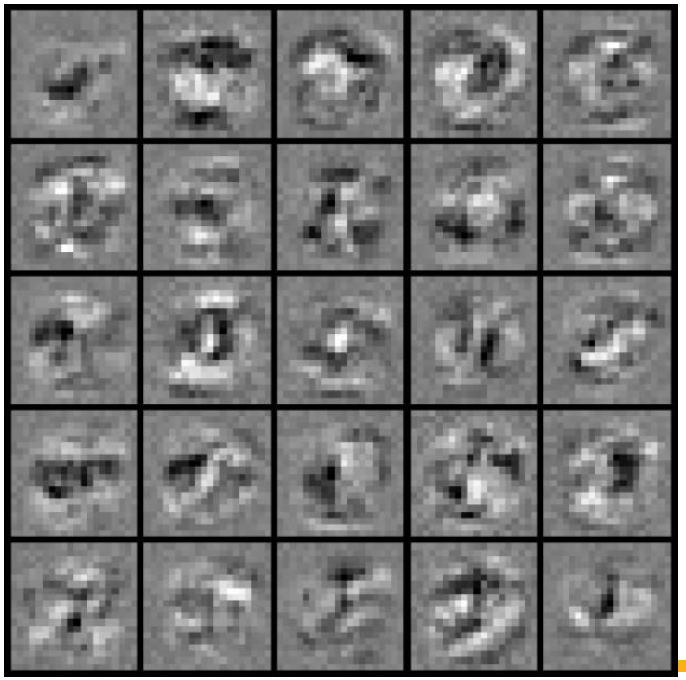
$x_{381} \dots x_{400}$

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

Each image is “unrolled” into a vector **x** of pixel intensities

# Layering Representations

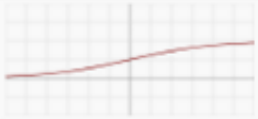
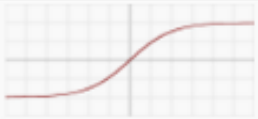
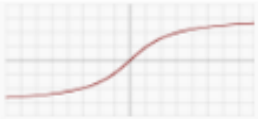


Visualization of Hidden Layer

# Activation functions



| Name | Plot | Equation | Derivative |
|---|---|---|---|
| Identity |  | $f(x) = x$ | $f'(x) = 1$ |
| Binary step |  | $f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$ | $f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$ |
| Logistic (a.k.a Soft step) |  | $f(x) = \frac{1}{1+e^{-x}}$ | $f'(x) = f(x)(1-f(x))$ |
| TanH |  | $f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$ | $f'(x) = 1 - f(x)^2$ |
| ArcTan |  | $f(x) = \tan^{-1}(x)$ | $f'(x) = \frac{1}{x^2+1}$ |
| Rectified Linear Unit (ReLU) |  | $f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$ | $f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$ |
| Parameteric Rectified Linear Unit (PReLU) [2] |  | $f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$ | $f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$ |
| Exponential Linear Unit (ELU) [3] |  | $f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$ | $f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$ |
| SoftPlus |  | $f(x) = \log_e(1+e^x)$ | $f'(x) = \frac{1}{1+e^{-x}}$ |

# Early Stopping

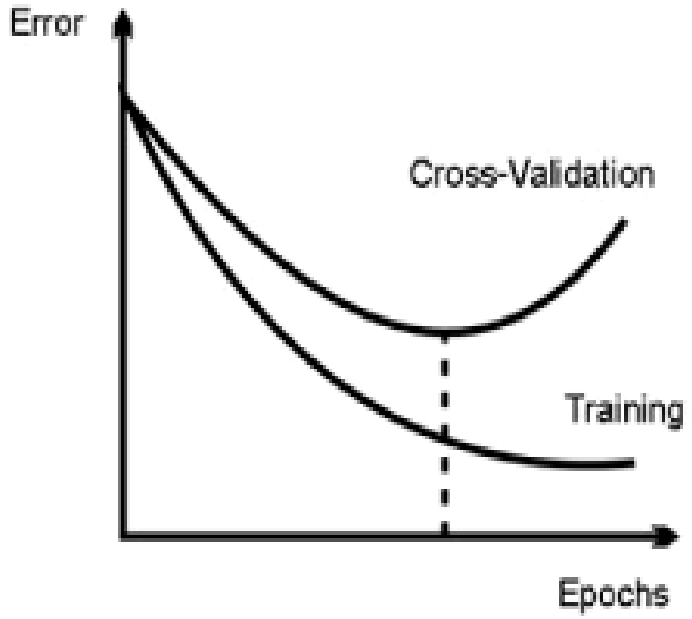
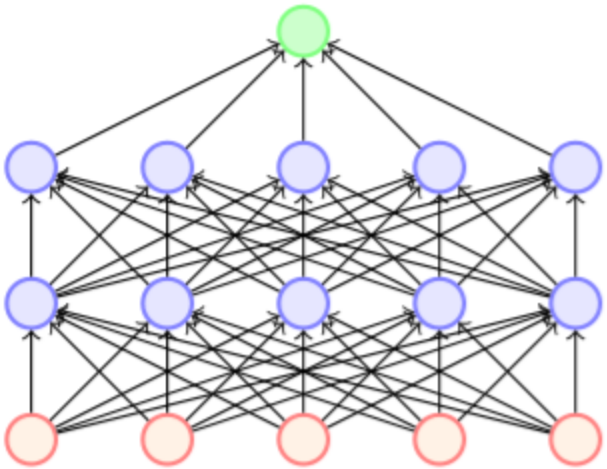


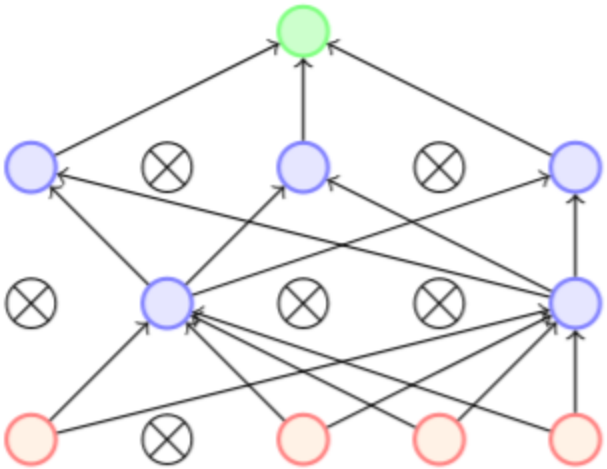

**Figure 2:** Profiles for training and cross-validation errors.

- If we let a complex model train long enough on a given data set it can eventually learn the data exactly.
- Given data that isn't represented in the training set, the model will perform poorly (overfitting).
- **How is the sweet spot for training located?**
- When the error on the training set begins to deviate from the error on the validation set,
- a threshold can be set to determine the early stopping condition and the ideal number of epochs to train.

# Dropout

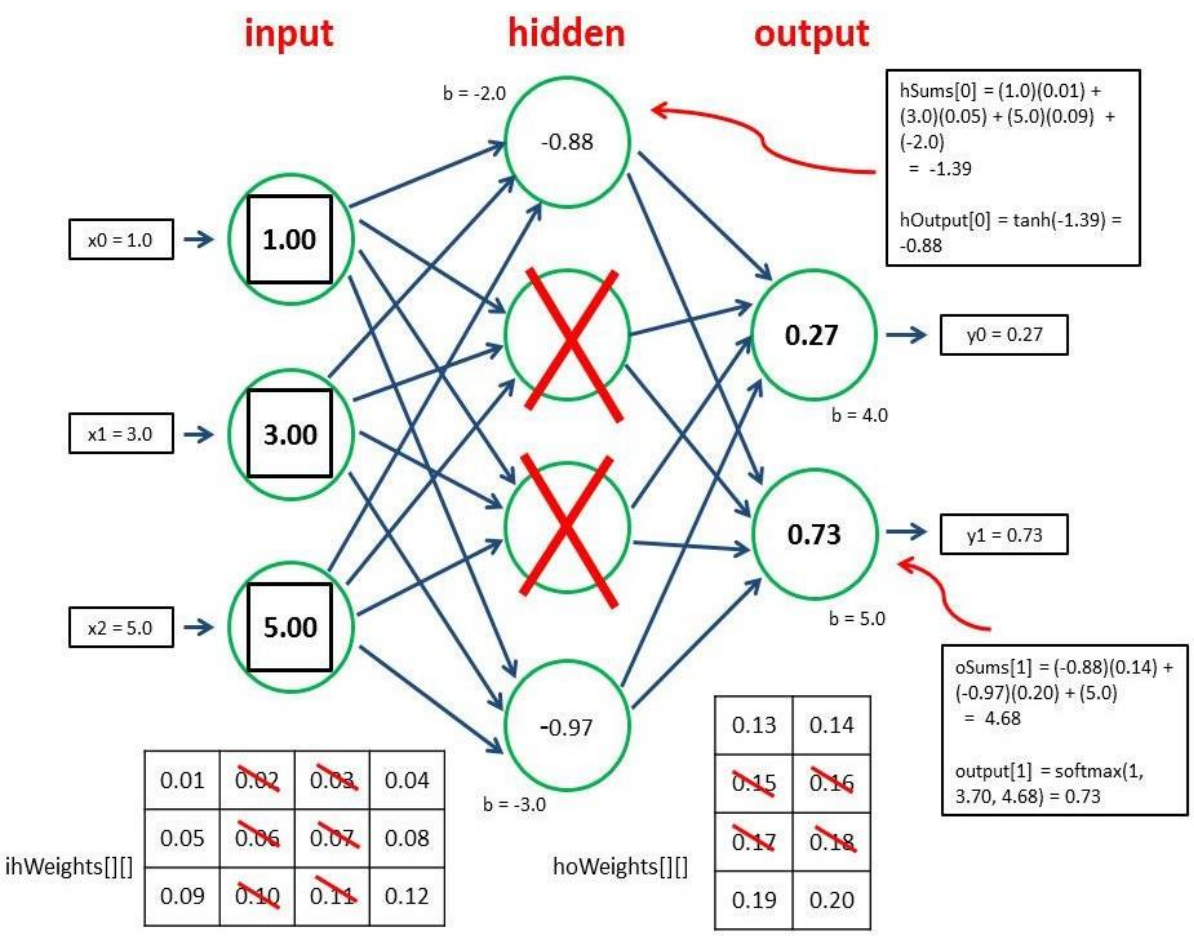






- Dropout refers to dropping out units
- Temporarily remove a node and all its incoming/outgoing connections resulting in a thinned network
- Each node is retained with a fixed probability (typically $p = 0.5$) for hidden nodes and $p = 0.8$ for visible nodes

# Dropouts



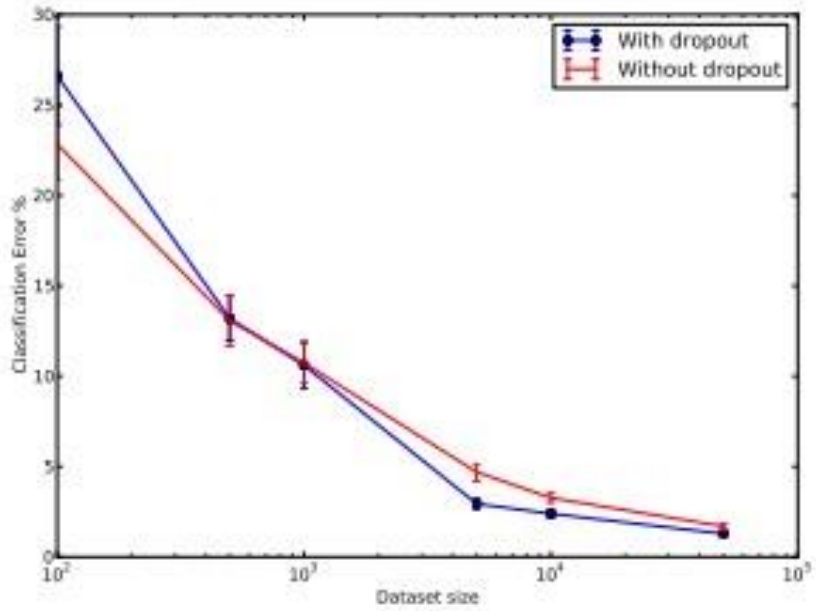




Figure 10: Effect of varying data set size.

Andrew Ng videos on neural network
https://www.youtube.com/watch?v=EVeqrPGfuCY&list=PLLssT5z_DsK-h9vYZkQkYNWcItqhlRJLN&index=45

Autonomous driving using neural network
https://www.youtube.com/watch?v=ppFyPUx9RIU&list=PLLssT5z_DsK-h9vYZkQkYNWcItqhlRJLN&index=57

https://medium.com/binaryandmore/beginners-guide-to-deriving-and-implementing-backpropagation-e3c1a5a1e536

Gradient checking
http://ufldl.stanford.edu/tutorial/supervised/DebuggingGradientChecking/

# Derivative of sigmoid activation function

$$\sigma(z^{[L]}) = \frac{1}{1+e^{-z^{[L]}}}$$

$$\sigma'(z^{[L]}) = \frac{d}{dz^{[L]}}\left(\frac{1}{1+e^{-z^{[L]}}}\right)$$

$$= -\frac{1}{(1+e^{-z^{[L]}})^2}.\frac{d}{dz^{[L]}}(1+e^{-z^{[L]}})$$

$$= -\frac{e^{-z^{[L]}}}{(1+e^{-z^{[L]}})^2}.\frac{d}{dz^{[L]}}(-z^{[L]})$$

$$= \frac{e^{-z^{[L]}}}{(1+e^{-z^{[L]}})^2}$$

$$= \frac{e^{-z^{[L]}}+1-1}{(1+e^{-z^{[L]}})^2}$$

$$= \frac{1}{1+e^{-z^{[L]}}}.\left(\frac{e^{-z^{[L]}}+1-1}{1+e^{-z^{[L]}}}\right)$$

$$= \frac{1}{1+e^{-z^{[L]}}}.\left(\frac{1+e^{-z^{[L]}}}{1+e^{-z^{[L]}}} - \frac{1}{1+e^{-z^{[L]}}}\right)$$

$$= \frac{1}{1+e^{-z^{[L]}}}.\left(1 - \frac{1}{1+e^{-z^{[L]}}}\right)$$

$$= \sigma(z^{[L]}).(1-\sigma(z^{[L]}))$$

$$\sigma'(z^{[L]}) = \sigma(z^{[L]}).(1-\sigma(z^{[L]}))$$

$$\therefore \quad \sigma'(z^{[L]}) = a^{[L]}.(1-a^{[L]}) \quad \ldots 2$$