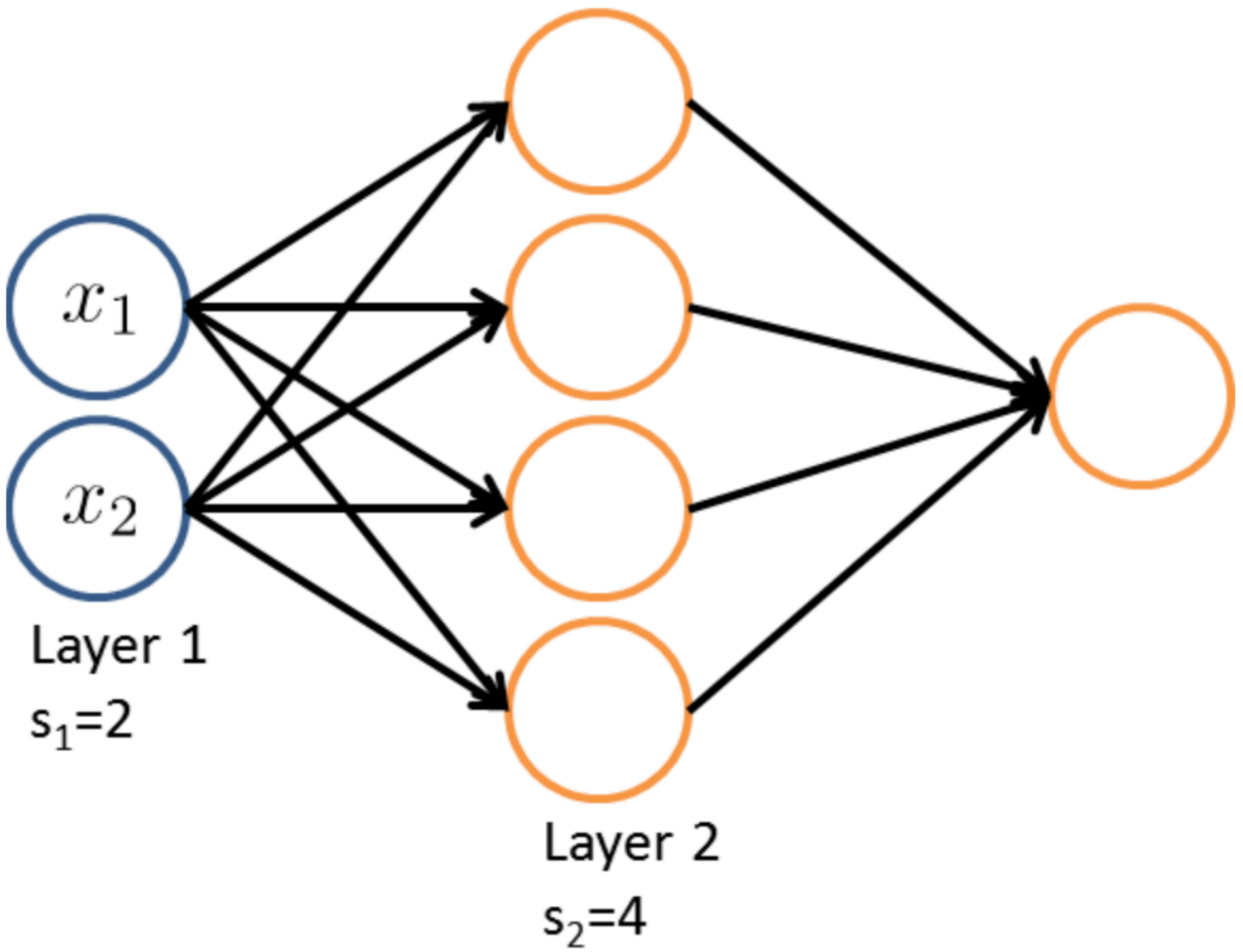
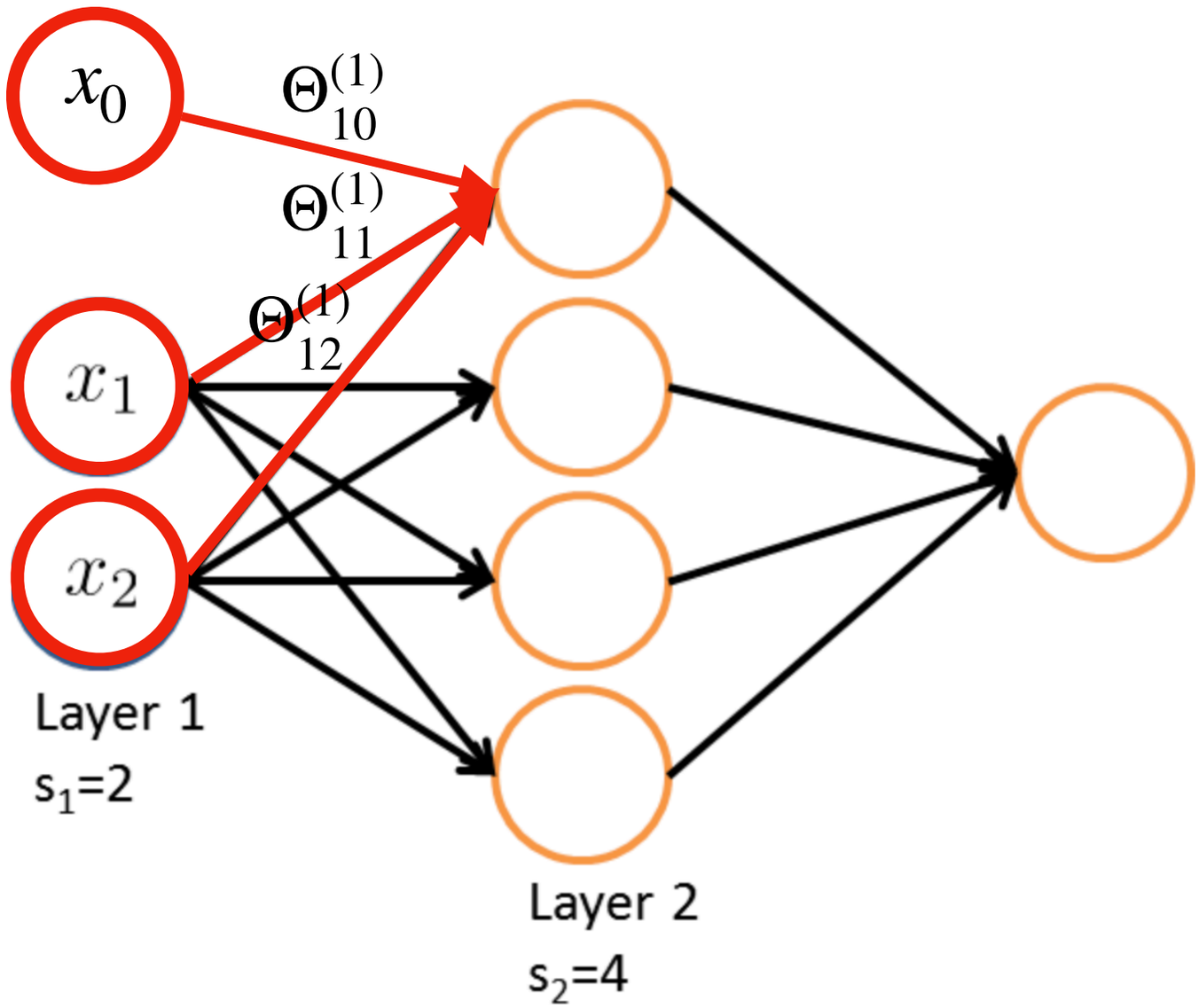
# Neural Networks: Learning

Teeradaj Racharak (เอ็กซ์)
r.teeradaj@gmail.com

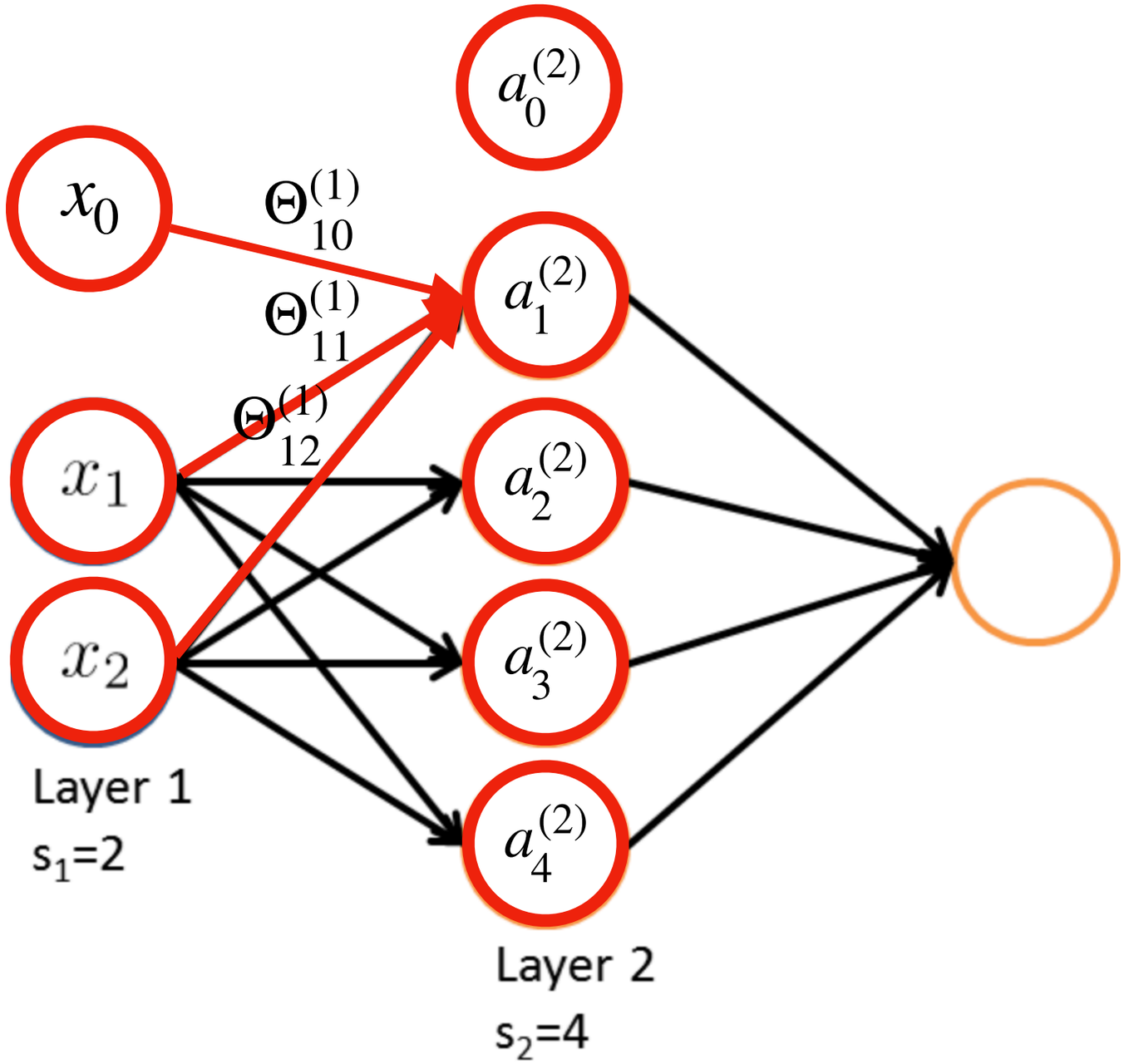
# Feedforward Propagation



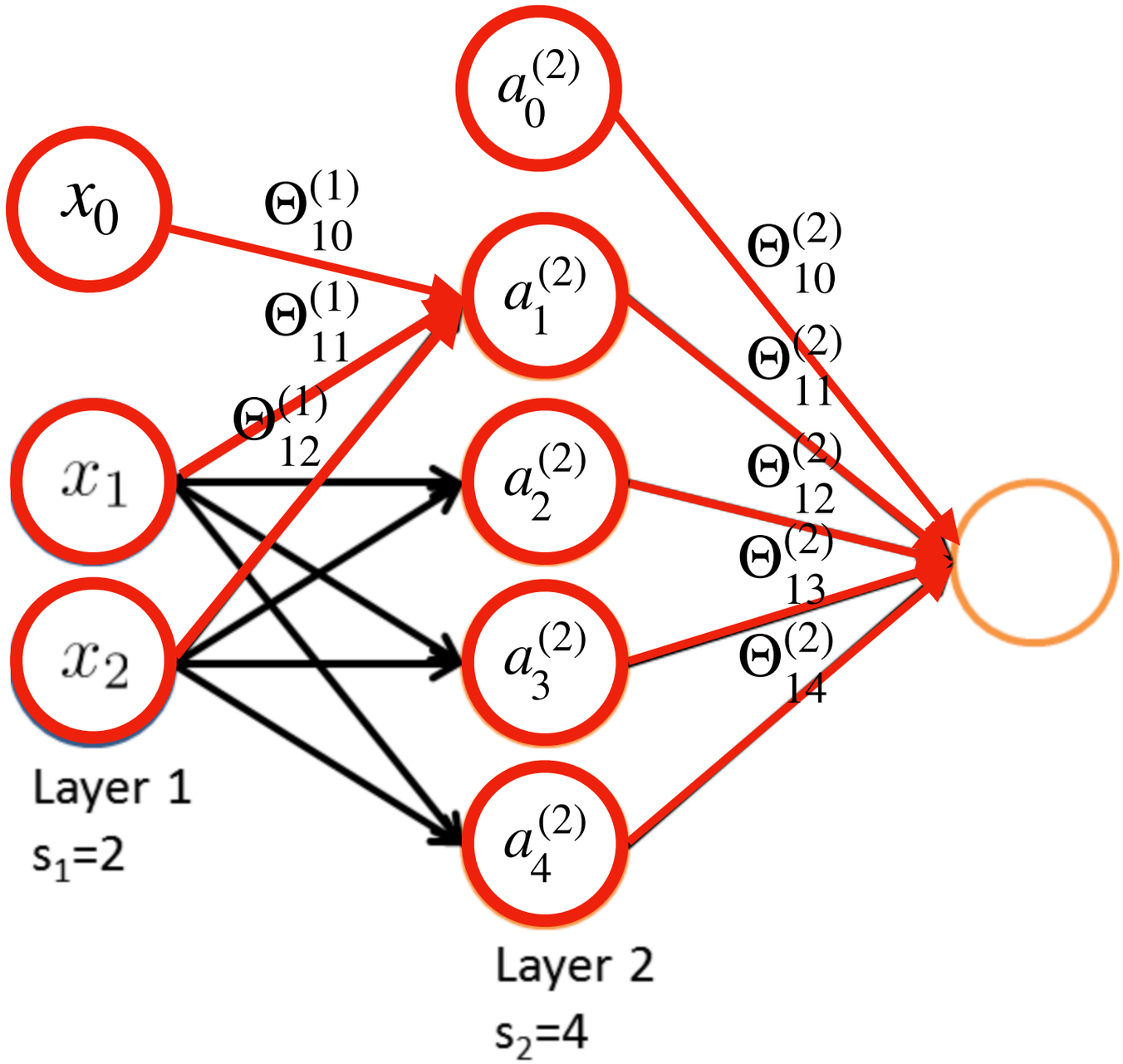
# Feedforward Propagation



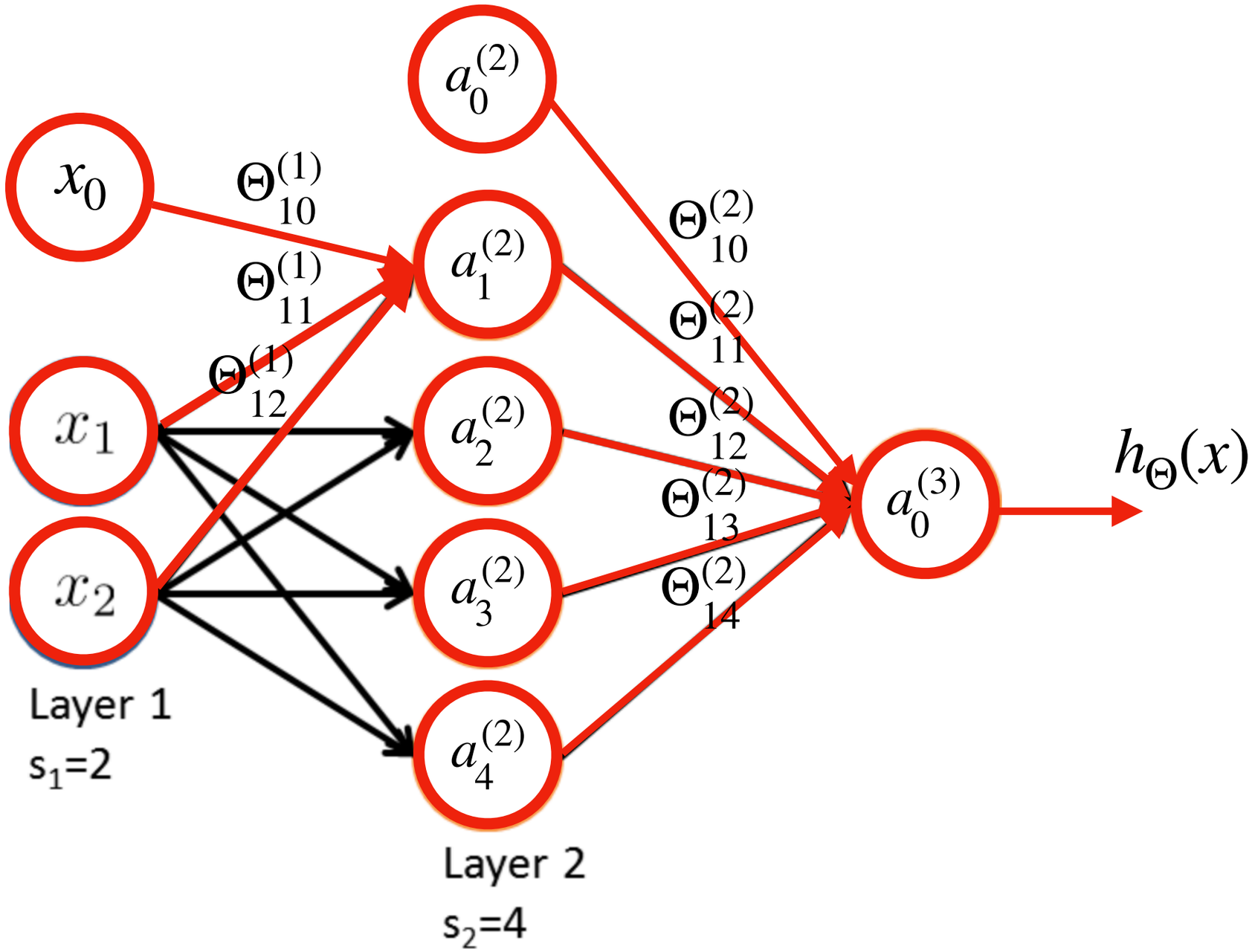
# Feedforward Propagation



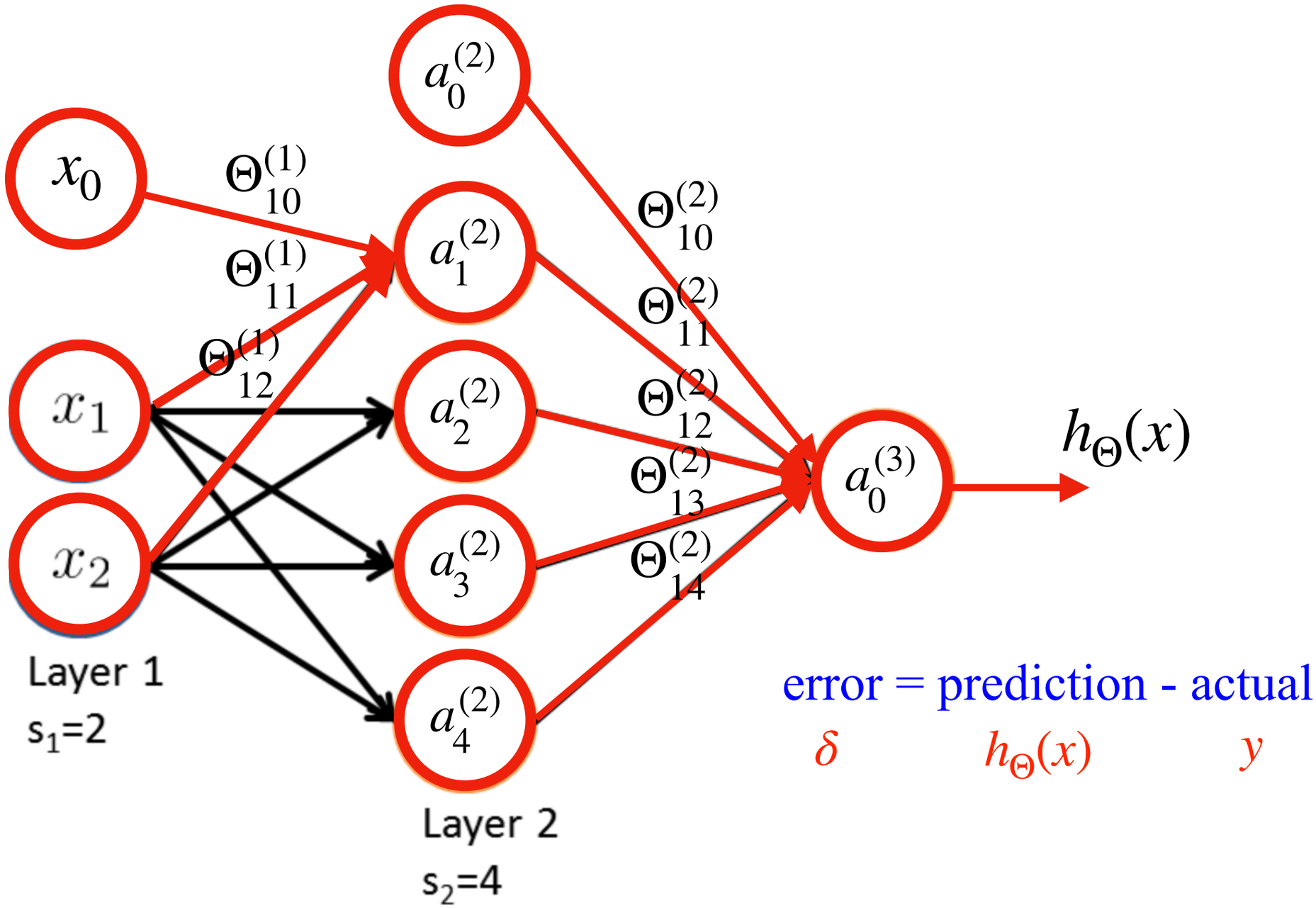
# Feedforward Propagation



# Feedforward Propagation



# Feedforward Propagation



# Learning via Backpropagation

# Backpropagation

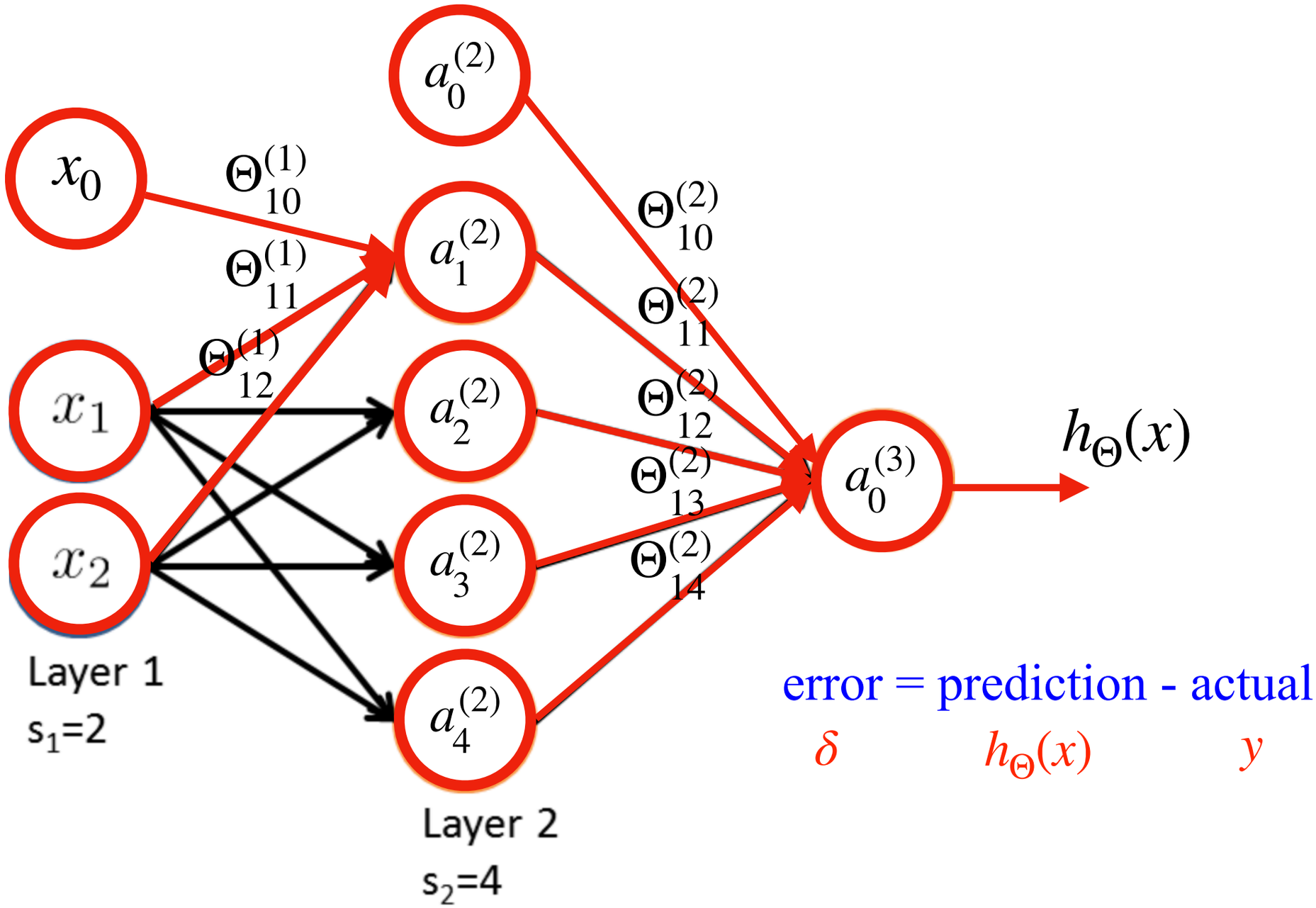
- Backpropagation is shorthand for ‘the backward propagation of errors’ since an error is computed at the output and distributed backwards throughout the network’s layers.

- It was derived by multiple researchers in the early 60’s and first proposed by Werbos to use in neural networks in his 1974 PhD thesis.

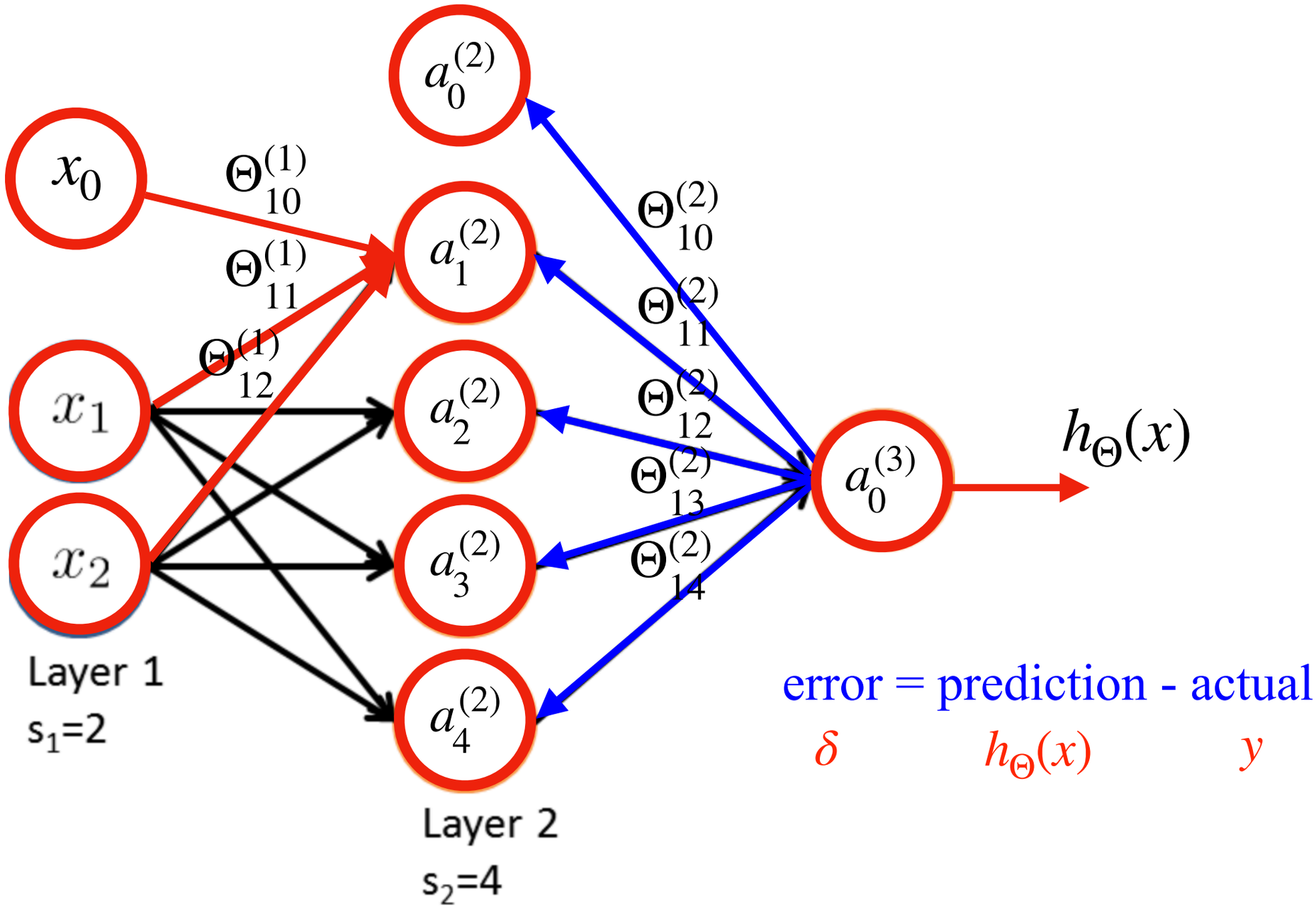
- It is closely related to the *Gauss-Newton* algorithm and is part of continuing research in neural backpropagation.

- In the context of ‘learning’, it is commonly used by the gradient descent optimization algorithm to adjust the weight of neurons by calculating the gradient of the cost function.

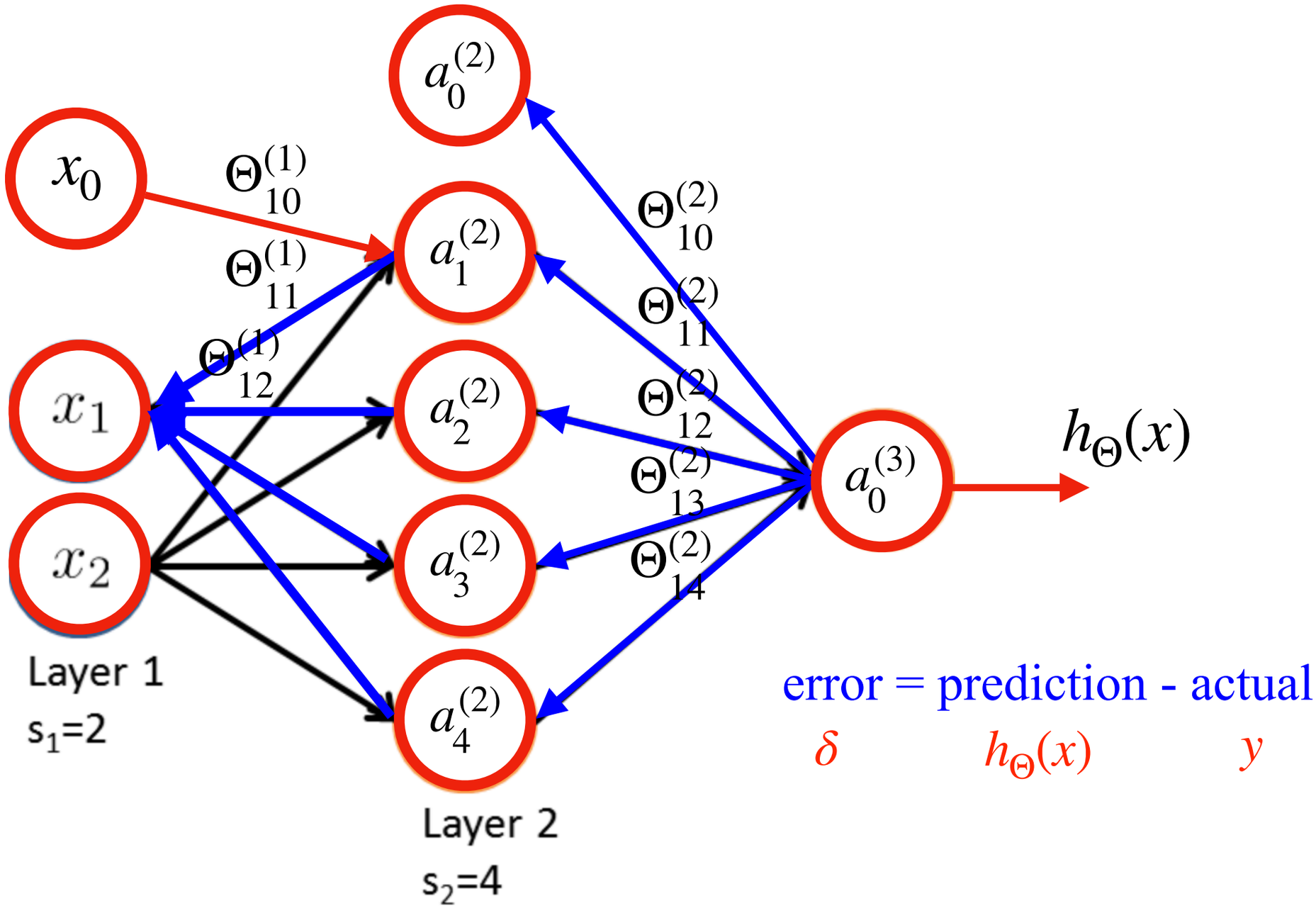
# Backpropagation



# Backpropagation



# Backpropagation



# Question

Suppose you have two training examples $(x^{(1)}, y^{(1)})$ and $(x^{(2)}, y^{(2)})$. Which of the following is correct sequence of operations for computing the gradient? (Below, FP = forward propagation, BP = back propagation).

(i) FP using $x^{(1)}$ followed by FP using $x^{(2)}$. Then, BP using $y^{(1)}$ followed by BP using $y^{(2)}$.

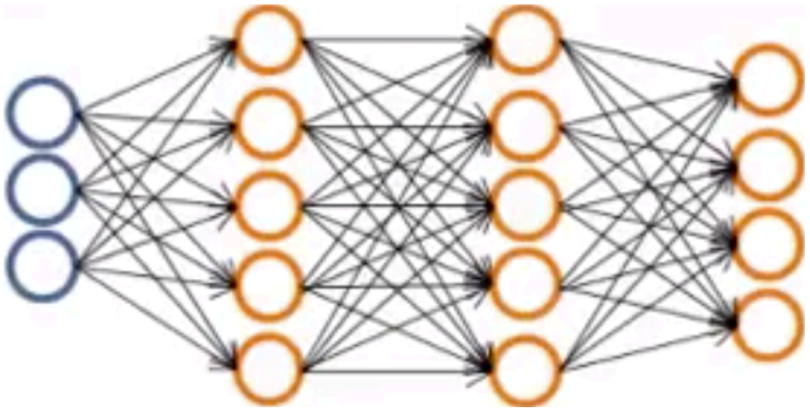
(ii) FP using $x^{(1)}$ followed by BP using $y^{(2)}$. Then, FP using $x^{(2)}$ followed by BP using $y^{(1)}$.

(iii) BP using $y^{(1)}$ followed by FP using $x^{(1)}$. Then, BP using $y^{(2)}$ followed by FP using $x^{(2)}$.

(iv) FP using $x^{(1)}$ followed by BP using $y^{(1)}$. Then, FP using $x^{(2)}$ followed by BP using $y^{(2)}$.

# Running Example

**Neural Network (for Classification)**



Layer 1 Layer 2 Layer 3 Layer 4

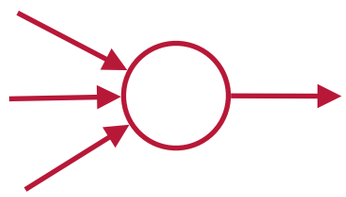
Training set: $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})\}$

$L$ = total no. of layers in a network (*e.g.* $L = 4$)

$s_l$ = # of units (not counting bias unit) in layer $l$

**Type 1: Binary Classification**

$y \in \{0,1\}$ *i.e.* 1 output unit



$$y = \begin{cases} 1 \\ 0 \end{cases}$$

**Type 2: *K*-class Classification**

$y \in \mathbb{R}^K$ *i.e.* $K$ output unit *e.g.*

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

pedestrian car motorcycle truck

# Cost Function

**Logistic Regression:**

$$J(\theta) = -\frac{1}{m}\left[\Sigma_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))\right] + \frac{\lambda}{2m}\Sigma_{j=1}^{n} \theta_j^2$$

**Neural Network for Classification:**

$$J(\Theta) = -\frac{1}{m}\left[\Sigma_{i=1}^{m}\Sigma_{k=1}^{K} y_k^{(i)} \log((h_\Theta(x^{(i)}))_k + (1 - y_k^{(i)}) \log(1 - (h_\Theta(x^{(i)}))_k)\right]$$

$$+\frac{\lambda}{2m}\Sigma_{l=1}^{L-1}\Sigma_{i=1}^{s_l}\Sigma_{j=1}^{s_{l+1}}(\Theta_{ji}^{(l)})^2$$

where $h_\Theta(x) \in \mathbb{R}^K$ such that $(h_\Theta(x))_i = i^{th}$ output

# Training in Neural Network

**Cost Function:**

$$J(\Theta) = -\frac{1}{m}\left[\Sigma_{i=1}^{m}\Sigma_{k=1}^{K} y_k^{(i)} \log(h_\Theta(x^{(i)}))_k + (1 - y_k^{(i)})\log(1 - (h_\Theta(x^{(i)}))_k)\right]$$

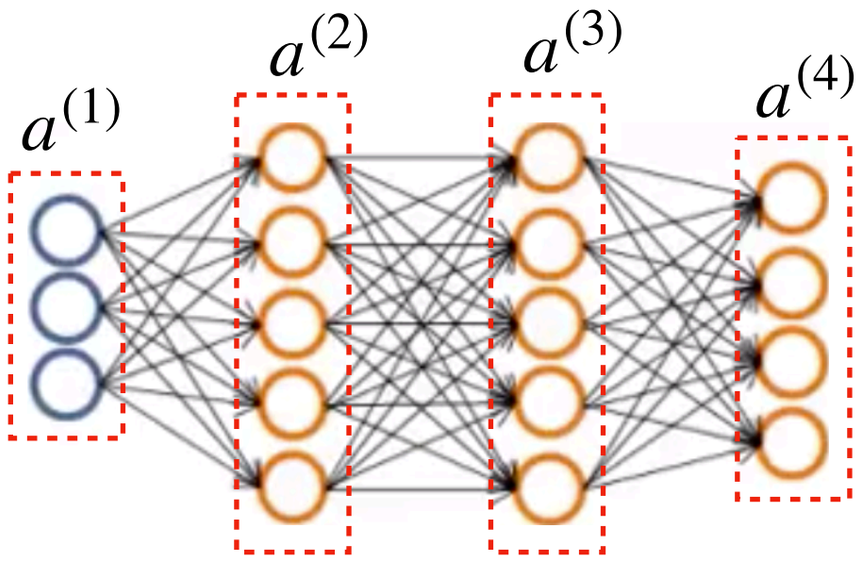
$$+\frac{\lambda}{2m}\Sigma_{l=1}^{L-1}\Sigma_{i=1}^{s_l}\Sigma_{j=1}^{s_{l+1}}(\Theta_{ji}^{(l)})^2$$

**Goal:** $\min_{\Theta} J(\Theta)$

**Gradient Computation** needs to compute:

- $J(\Theta)$
- $\frac{\partial J(\Theta)}{\partial \Theta_{ij}^{(l)}}$ where $\Theta_{ij}^{(l)} \in \mathbb{R}$

# First, Apply Feedforward




Given 'one' training example ($\boldsymbol{x}$, $y$):

**Applying forward propagation**

$$a^{(1)} = \boldsymbol{x}$$

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

$$a^{(2)} = g(z^{(2)}) \quad (\text{add } a_0^{(2)})$$

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

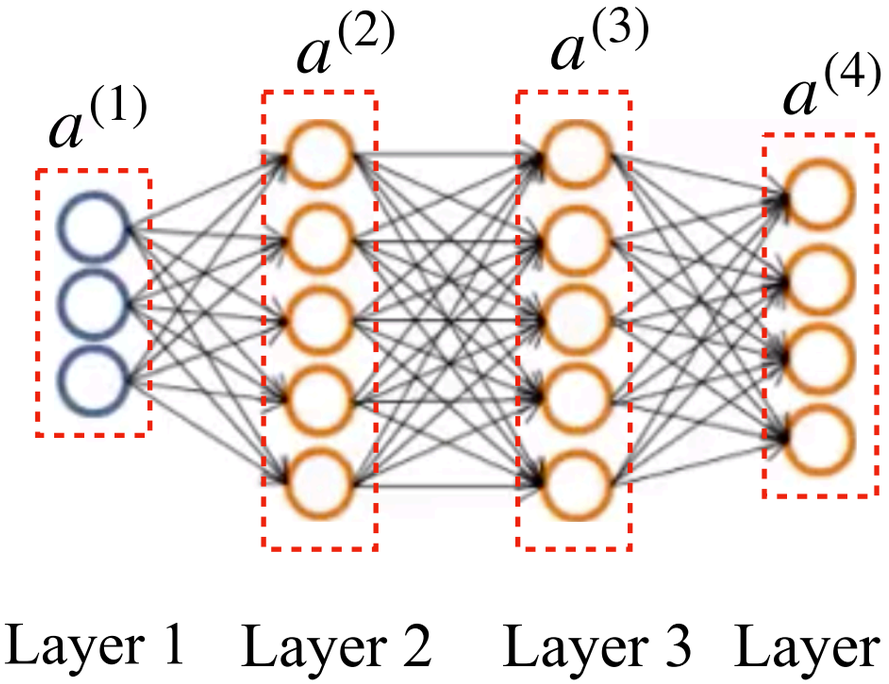
$$a^{(3)} = g(z^{(3)}) \quad (\text{add } a_0^{(3)})$$

$$z^{(4)} = \Theta^{(3)} a^{(3)}$$

$$a^{(4)} = h_\Theta(x) = g(z^{(4)})$$

# Backpropagation (Idea)

**Intuition:** $\delta_j^{(l)}$ represents 'error' of node $j$ in layer $l$
*i.e.* the error we want to capture in $a_j^{(i)}$




To capture that intuition,

For each output unit (layer $L = 4$),

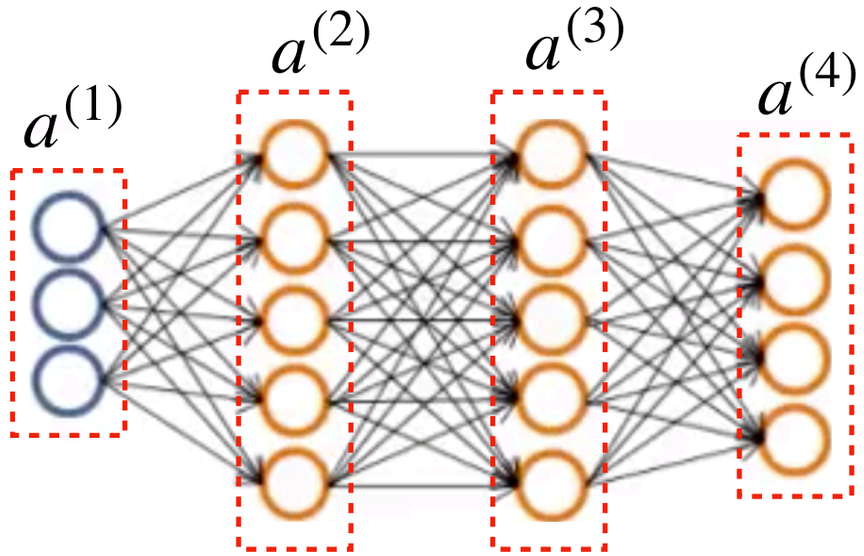
$$\delta_j^{(4)} = a_j^{(4)} - y_i = (h_\Theta(x))_j - y_j$$

or, its vectorized implementation:

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

# Backpropagation (Idea)

**Intuition:** $\delta_j^{(l)}$ represents 'error' of node $j$ in layer $l$
*i.e.* the error we want to capture in $a_j^{(i)}$




To capture that intuition,

For each output unit (layer $L = 4$),

$$\delta_j^{(4)} = a_j^{(4)} - y_i = (h_\Theta(x))_j - y_j$$

or, its vectorized implementation:

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

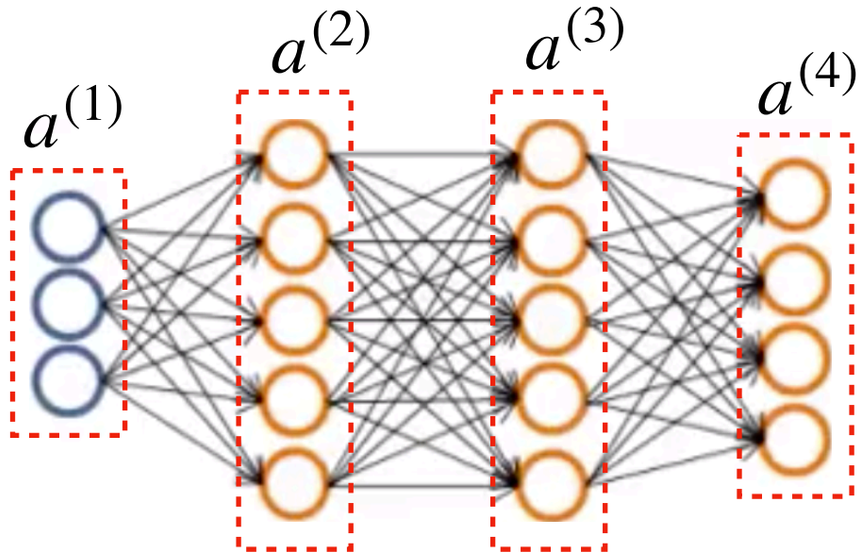
Computing errors in the earlier layers,

$\delta^{(3)} = (\Theta^{(3)})^T \delta^{(4)} .* g'(z^{(3)})$ where $.*$ denotes the element-wise multiplication

$\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} .* g'(z^{(2)})$ and $g'(z^{(i)}) = a^{(i)} .* (1 - a^{(i)})$

# Backpropagation (Idea)

**Intuition:** $\delta_j^{(l)}$ represents 'error' of node $j$ in layer $l$
*i.e.* the error we want to capture in $a_j^{(i)}$




To capture that intuition,

For each output unit (layer $L = 4$),

$$\delta_j^{(4)} = a_j^{(4)} - y_i = (h_\Theta(x))_j - y_j$$

or, its vectorized implementation:

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

Computing errors in the earlier layers,

No $\delta^{(1)}$ since it is the input layer

$$\delta^{(3)} = (\Theta^{(3)})^T \delta^{(4)} .* g'(z^{(3)})$$

$$\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} .* g'(z^{(2)})$$

where $\cdot *$ denotes the element-wise multiplication

and $g'(z^{(i)}) = a^{(i)} .* (1 - a^{(i)})$

# Gradient Computation

**Training set:** $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$

**Set:** $\Delta_{ij}^{(l)} := 0$ (for all $l, i, j$)

For $i = 1$ to $m$:
  Set $a^{(1)} := x^{(i)}$
  Perform forward propagation to compute $a^{(l)}$ for $l = 2,3,\ldots,L$
  Using $y^{(i)}$, compute $\delta^{(L)} = a^{(L)} - y^{(i)}$
  Compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$
  $\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$ (vectorized implementation $\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)}(a^{(l)})^T$)

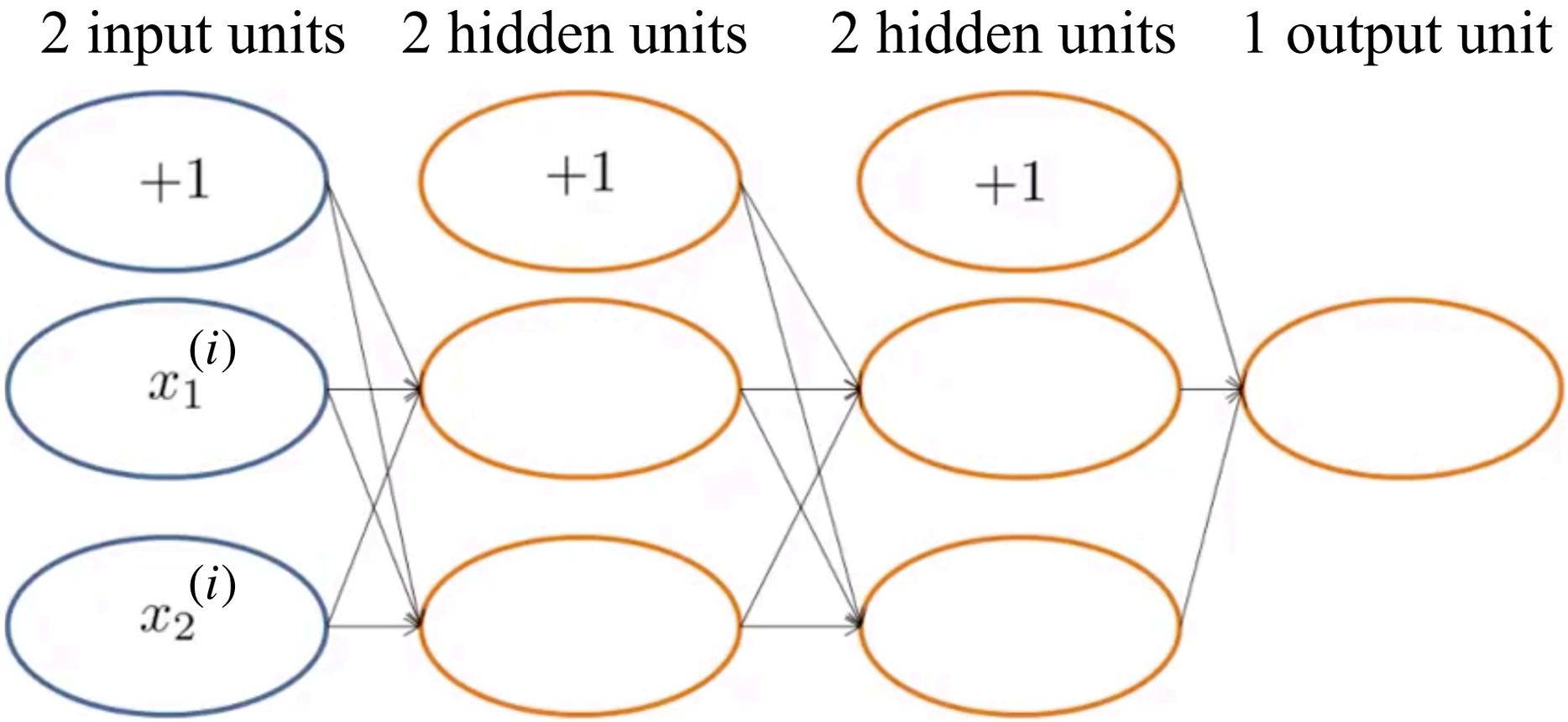
**Finally,** we compute the gradient as follows:

- $D_{ij}^{(l)} := \frac{1}{m}\Delta_{ij}^{(l)} + \lambda\Theta_{ij}^{(l)}$ if $j \neq 0$
- $D_{ij}^{(l)} := \frac{1}{m}\Delta_{ij}^{(l)}$ if $j = 0$

$\left(\textbf{Note: } \frac{\partial J(\Theta)}{\partial \Theta_{ij}^{(l)}} = D_{ij}^{(l)}\right)$

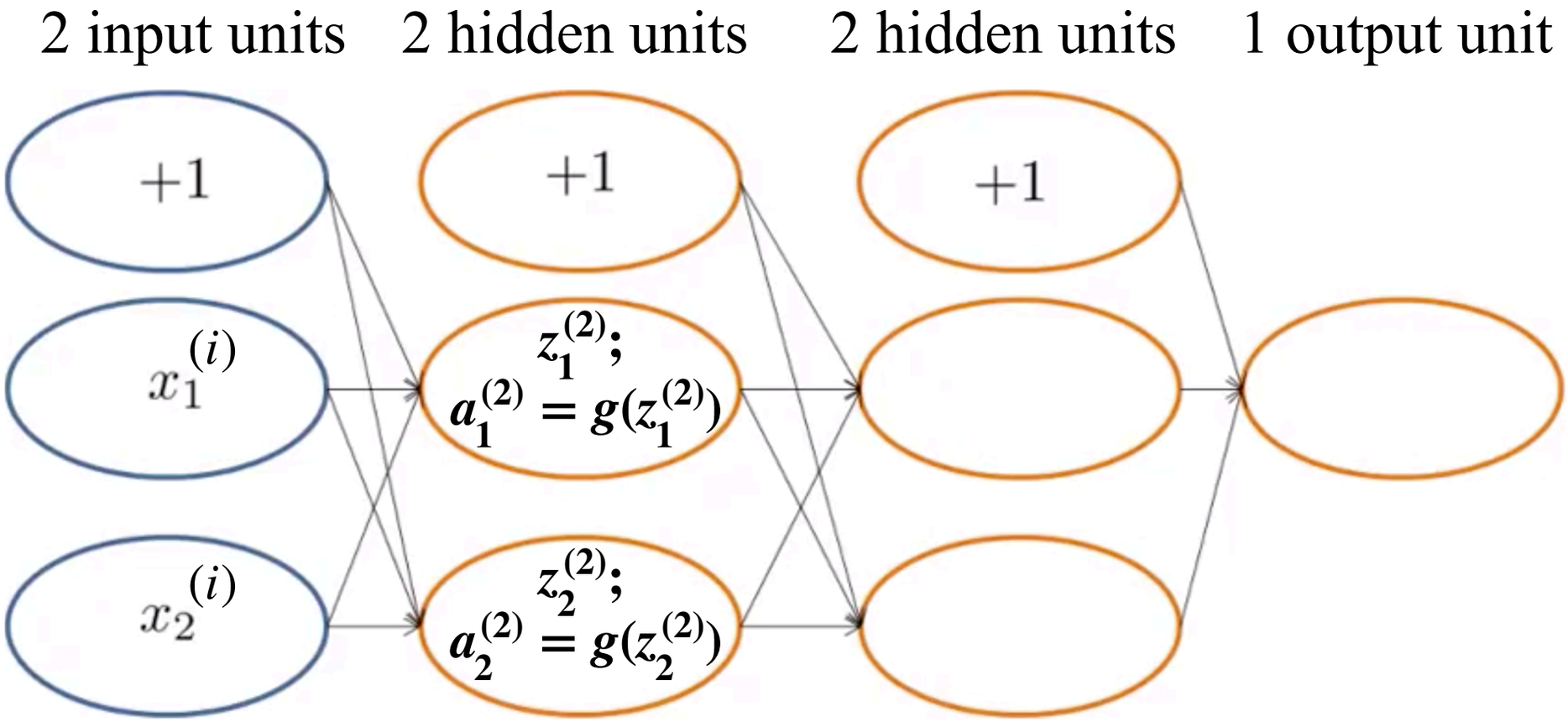
# Feedforward + Backprop

# Forward Propagation



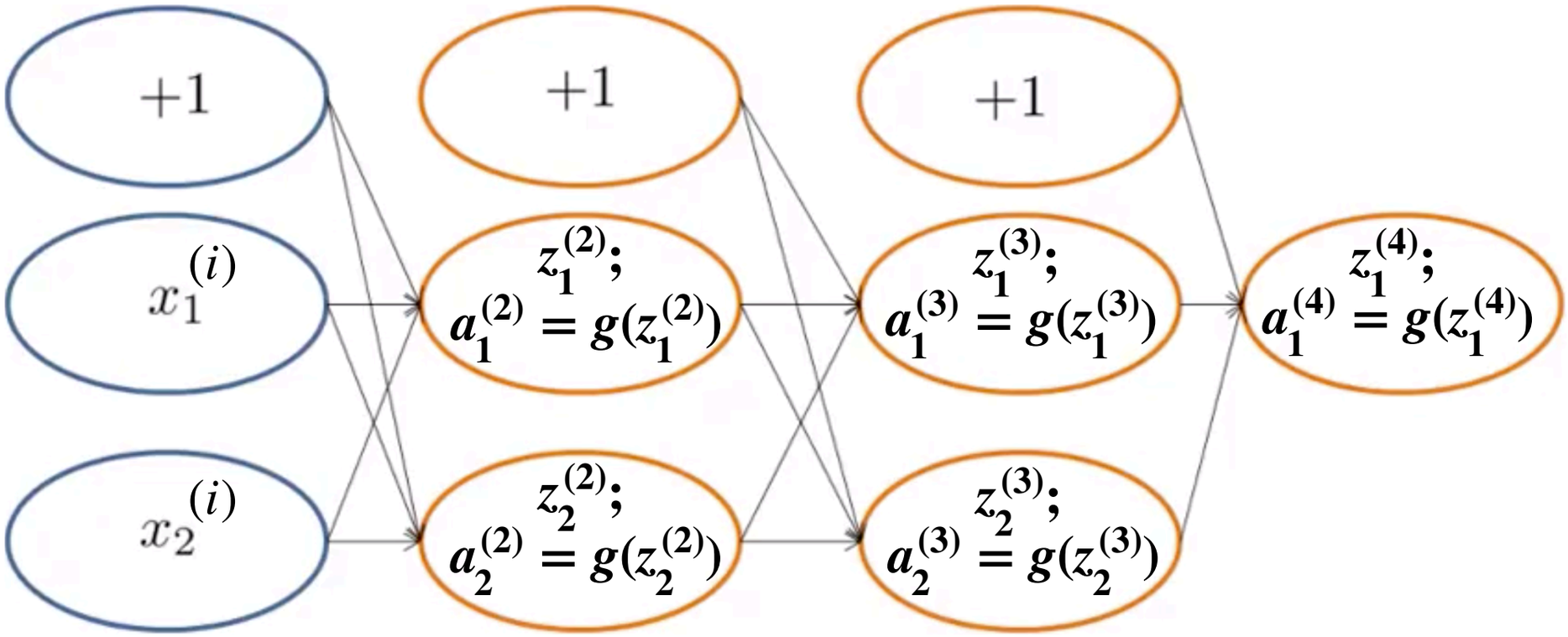

**Training set:** $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$

# Forward Propagation



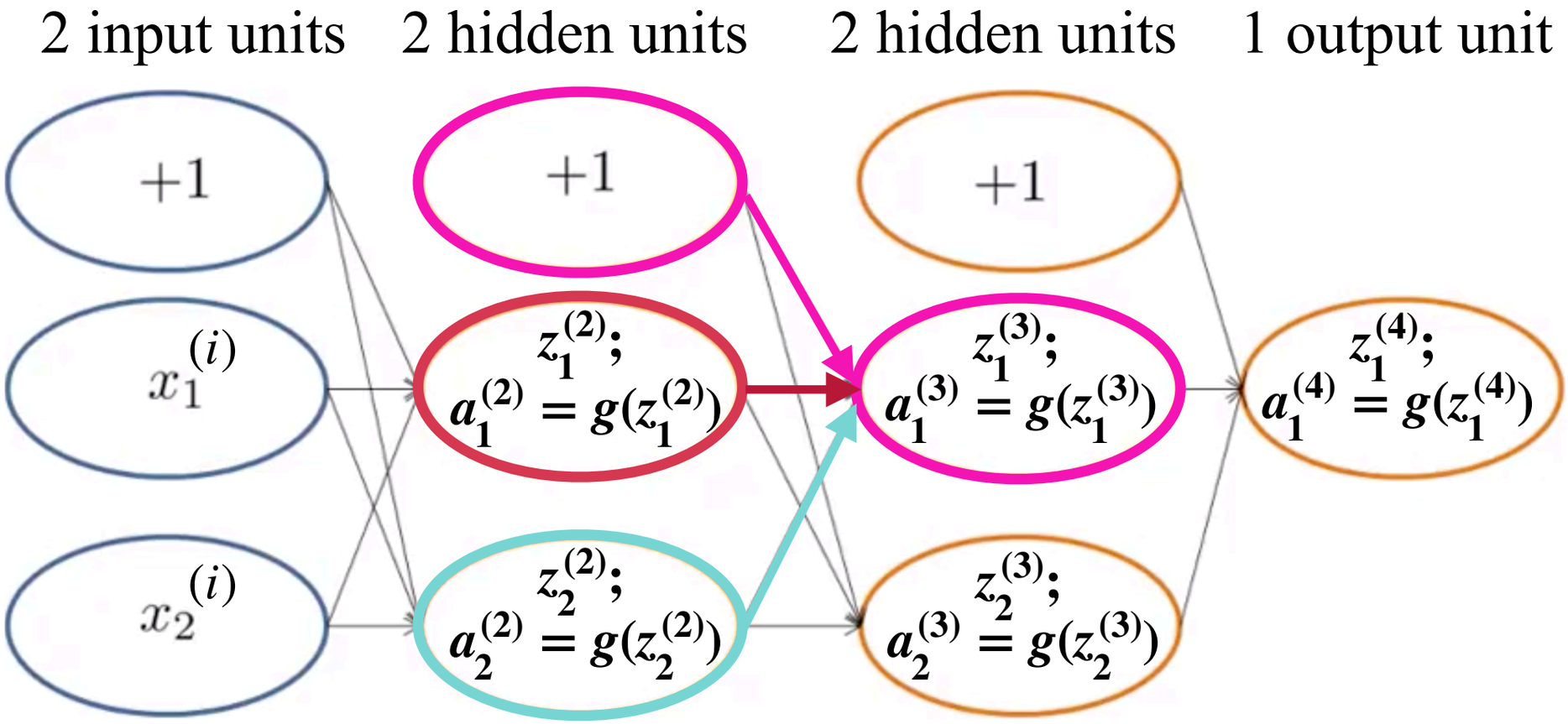

**Training set:** $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$

# Forward Propagation




**Training set:** $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$

# Forward Propagation




$$z_1^{(3)} = \Theta_{10}^{(2)} \times 1 + \Theta_{11}^{(2)} \times a_1^{(2)} + + \Theta_{12}^{(2)} \times a_2^{(2)}$$

# Backpropagation

**To understand what it's doing**, let's consider a case of:

- having 1 output unit
- ignoring regularization *i.e.* $\lambda = 0$
- a single logistic sigmoid output *i.e.* $m = 1$

**Cost function:**

$$J(\Theta) = -\frac{1}{m}\left[\Sigma_{i=1}^{m} y^{(i)} \log(h_\Theta(x^{(i)})) + (1 - y^{(i)})\log(1 - (h_\Theta(x^{(i)}))\right]$$

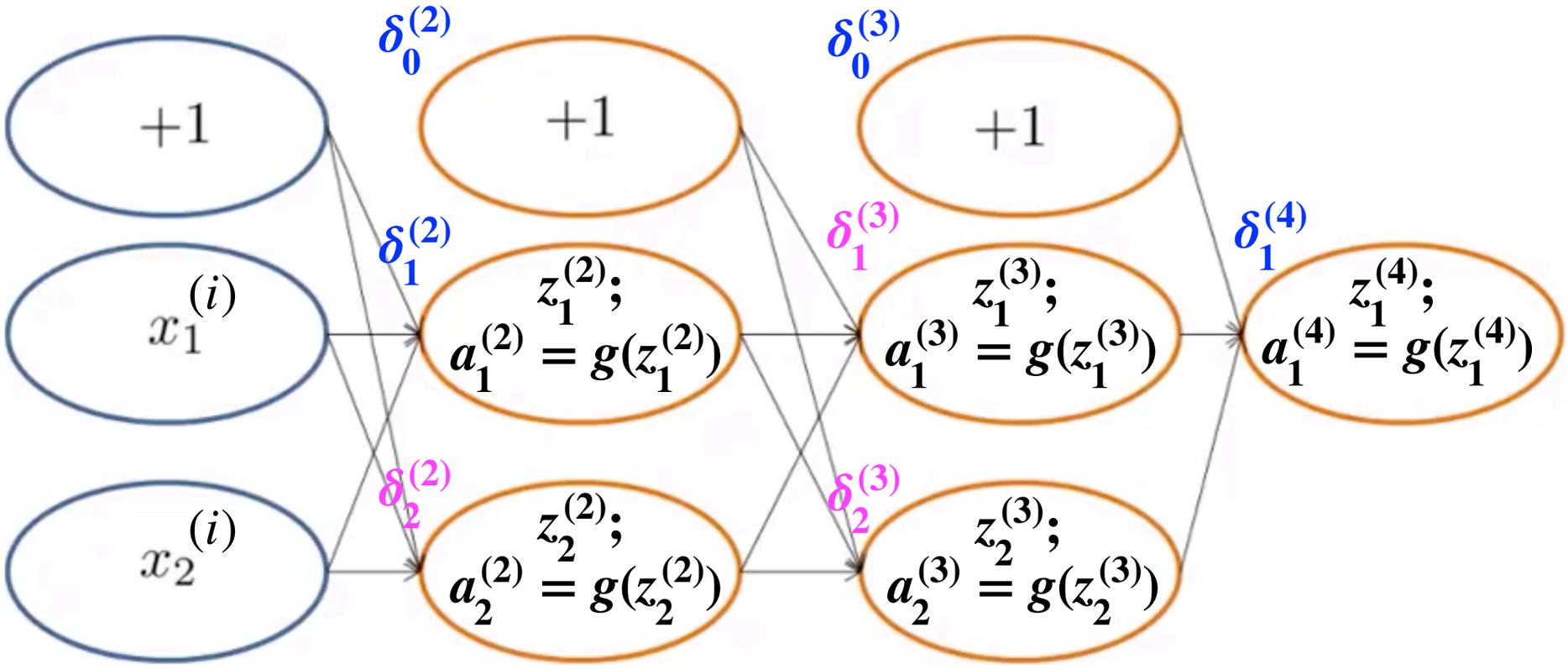
$$+\frac{\lambda}{2m}\Sigma_{l=1}^{L-1}\Sigma_{i=1}^{s_l}\Sigma_{j=1}^{s_{l+1}}(\Theta_{ji}^{(l)})^2$$

$$\Downarrow$$

$$\textbf{cost}(i) = y^{(i)} \log(h_\Theta(x^{(i)})) + (1 - y^{(i)})\log(1 - h_\Theta(x^{(i)}))$$

We can think of $\textbf{cost}(i)$ how well the network is doing on example $i$ ?

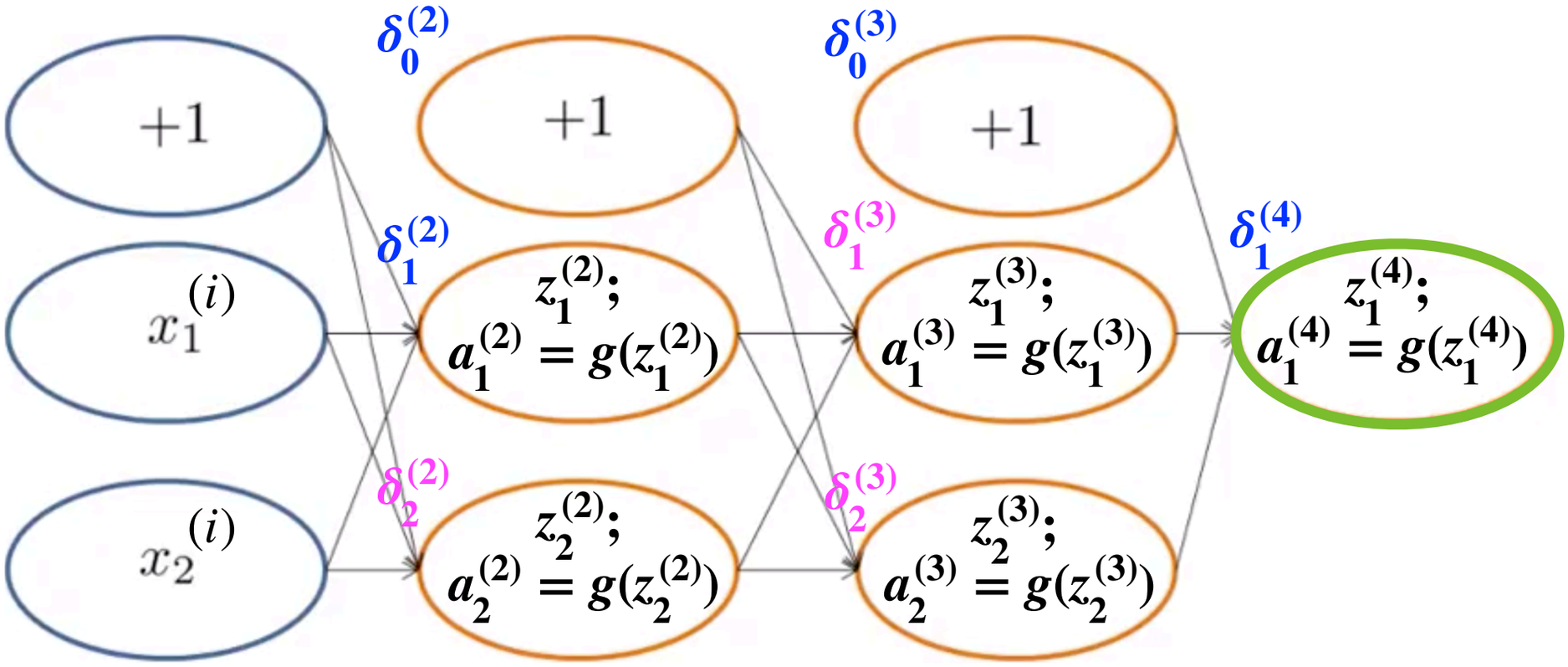
# Backpropagation




$\delta_j^{(l)}$ represents 'error' of cost for $a_j^{(l)}$ (unit $j$ in layer $l$)

**Later,** we show that: $\delta_j^{(i)} = \dfrac{\partial}{\partial z_j^{(l)}}\mathbf{cost}(i)$ for $j \geq 0$

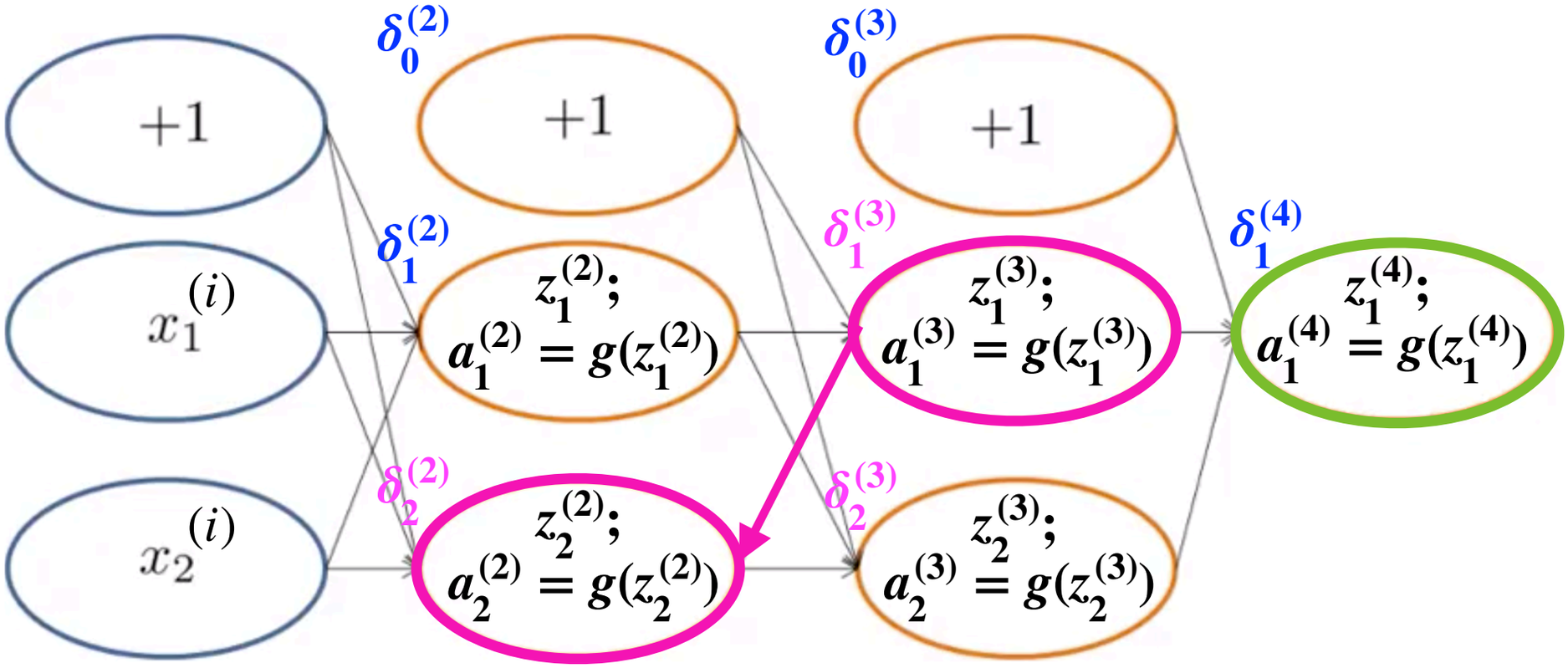
# Backpropagation




$\delta_j^{(l)}$ represents 'error' of cost for $a_j^{(l)}$ (unit $j$ in layer $l$) $\quad \delta_1^{(4)} = a_1^{(4)} - y^{(i)}$

**Later,** we show that: $\delta_j^{(i)} = \dfrac{\partial}{\partial z_j^{(l)}}\mathbf{cost}(i)$ for $j \geq 0$

# Backpropagation



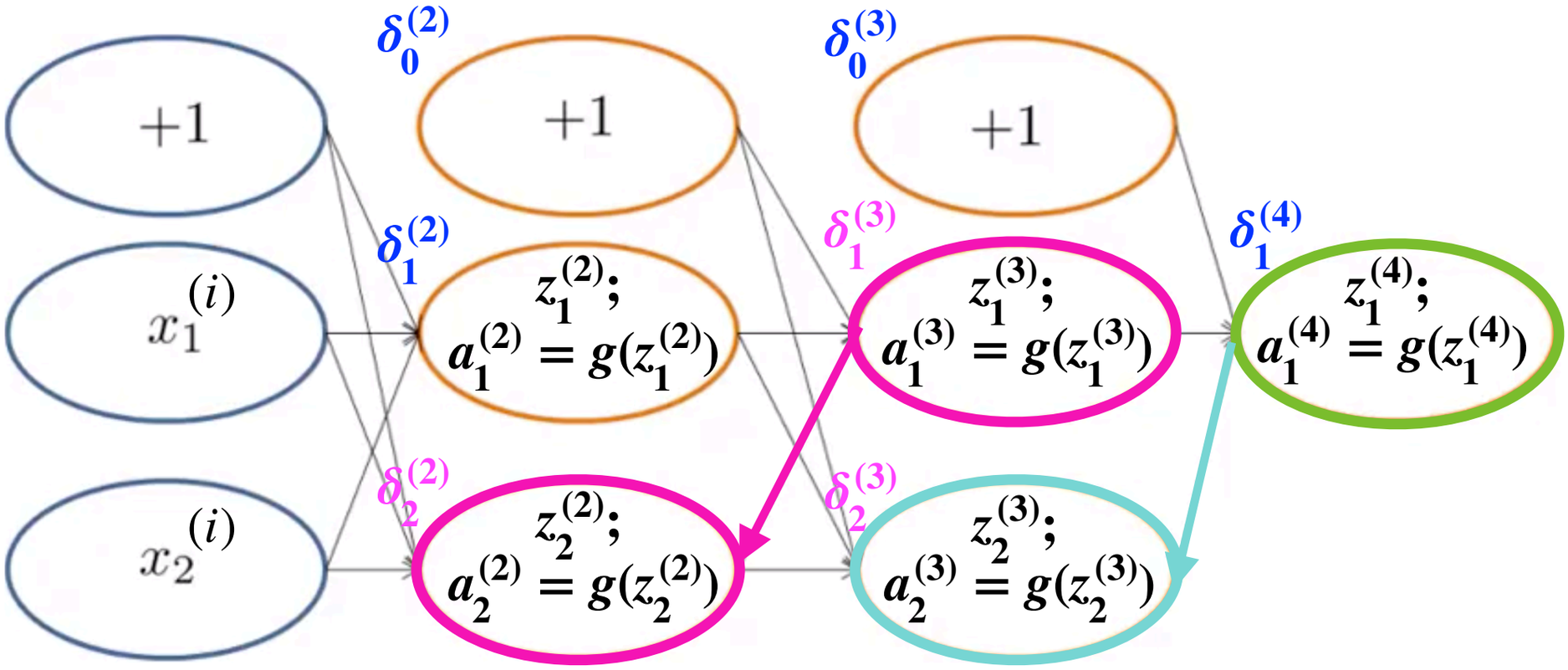

$\delta_j^{(l)}$ represents 'error' of cost for $a_j^{(l)}$ (unit $j$ in layer $l$)

**Later,** we show that: $\delta_j^{(i)} = \dfrac{\partial}{\partial z_j^{(l)}}\mathbf{cost}(i)$ for $j \geq 0$

$$\delta_1^{(4)} = a_1^{(4)} - y^{(i)}$$

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

# Backpropagation




$\delta_j^{(l)}$ represents 'error' of cost for $a_j^{(l)}$ (unit $j$ in layer $l$)

**Later,** we show that: $\delta_j^{(i)} = \dfrac{\partial}{\partial z_j^{(l)}}\mathbf{cost}(i)$ for $j \geq 0$

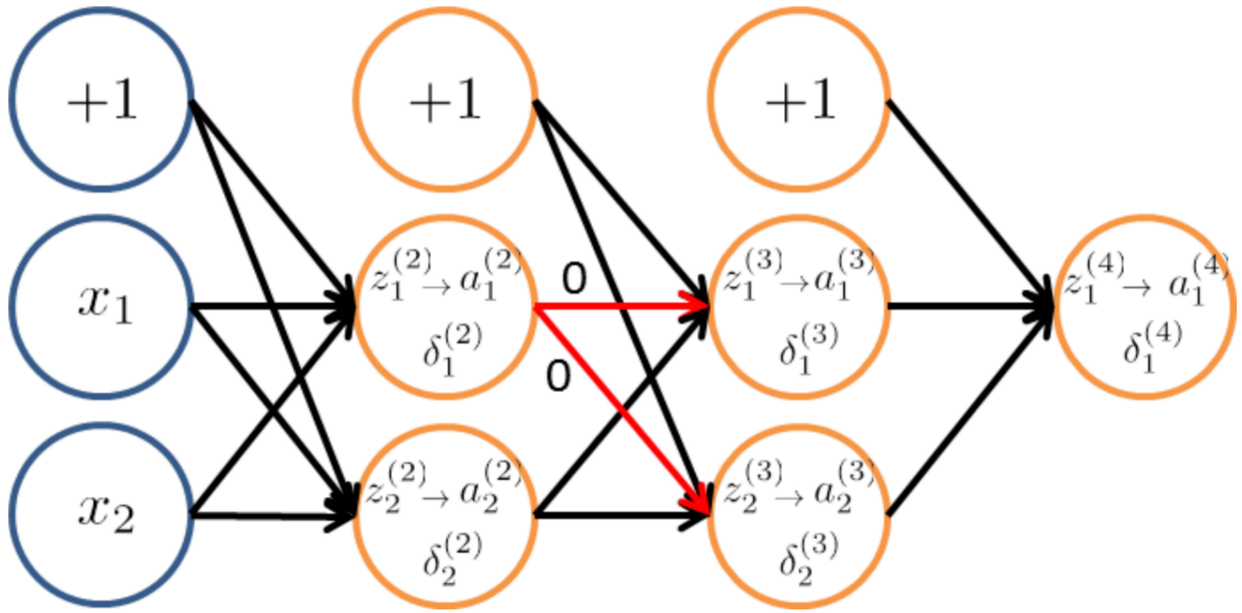
$$\delta_1^{(4)} = a_1^{(4)} - y^{(i)}$$

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

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

# Question

Consider the following neural network. Suppose both of the weights shown in red ($\Theta^2_{11}$ and $\Theta^{(2)}_{21}$) are equal to 0. After running back propagation, what can we say about the value of $\delta^{(3)}_1$?



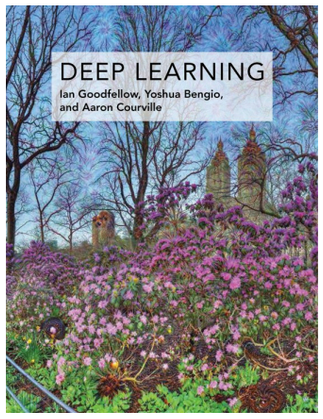

(i) $\delta^{(3)}_1 > 0$

(ii) $\delta^{(3)}_1 = 0$ only if $\delta^{(2)}_1 = \delta^{(2)}_2 = 0$, but not necessarily otherwise

(iii) $\delta^{(3)}_1 \leq 0$ regardless of the values of $\delta^{(2)}_1$ and $\delta^{(2)}_2$

(iv) There is insufficient information to tell

# Backpropagation (w.r.t. mathematical sense)




# Parameter Update

In the case of a single logistic sigmoid at the output layer, after forward propagation, we have a predicted value $\hat{y}$ (or, $h_\theta(x)$).

Neural network parameter update rules are usually derived in terms of backpropagating an error or loss.

If we have an objective function such as maximum likelihood, we can convert to a loss by negating it.

We thus get the log loss function for a network with a single logistic sigmoid output:

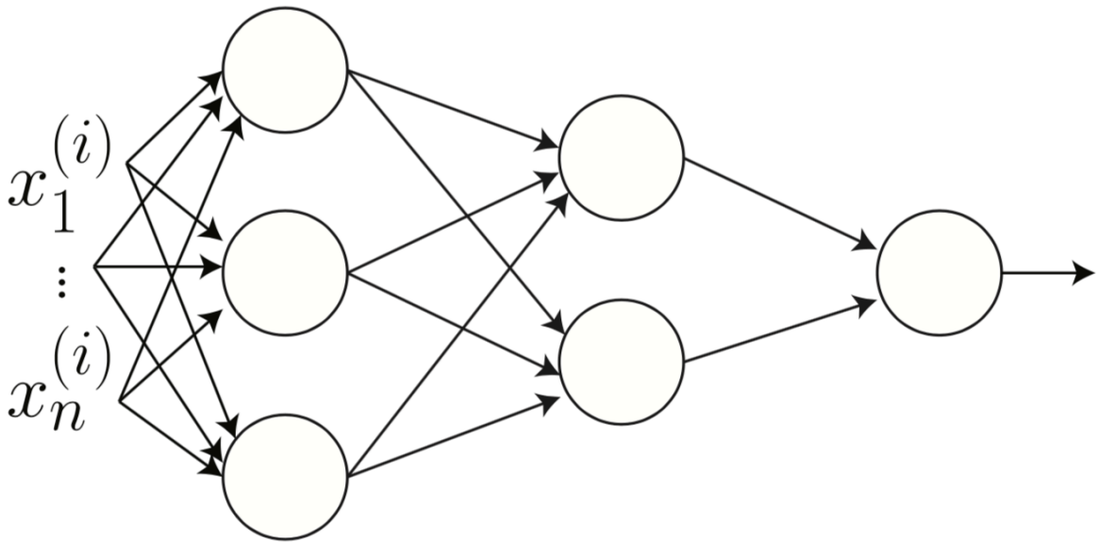
$$\mathscr{L}(\hat{y}, y) = -\left[y \log \hat{y} + (1 - y)\log(1 - \hat{y})\right]$$

Noted that it is easy to do the same for a linear output (Gaussian distribution for $y$) or softmax output (multinomial distribution for $y$).[1]

[1] See Goodfellow et al. (2016) Section 6.2.2.4 for discussion of other output types

# Parameter Update

Now, to update the parameters in layer $l$, we update using gradient descent on the log loss:




$$\mathbf{W}^{(l)} := \mathbf{W}^{(l)} - \alpha \frac{\partial \mathscr{L}}{\partial \mathbf{W}^{(l)}}$$

$$\mathbf{b}_0^{(l)} := \mathbf{b}_0^{(l)} - \alpha \frac{\partial \mathscr{L}}{\partial \mathbf{b}_0^{(l)}}$$

(Let me write $\mathbf{W}^{(l)}$ instead of $\Theta^{(l)}$ and $\mathbf{b}^{(l)}$ instead of $\theta_0^{(l)}$)

# Parameter Update

First, we consider the weights at the output layer. We have:

$$\frac{\partial \mathscr{L}}{\partial \mathbf{W}^{(4)}} = -\frac{\partial}{\partial \mathbf{W}^{(4)}}\big((1-y)\log(1-\hat{y}) + y\log\hat{y}\big)$$

$$= -(1-y)\frac{\partial}{\partial \mathbf{W}^{(4)}}\log\big(1 - g(\mathbf{W}^{(4)}\boldsymbol{a}^{(3)} + \boldsymbol{b}^{(4)})\big)$$

$$-y\frac{\partial}{\partial \boldsymbol{W}^{(4)}}\log\big(g(\boldsymbol{W}^{(4)}\boldsymbol{a}^{(3)} + \boldsymbol{b}^{(4)})\big)$$

$$= \frac{(1-y)g'(\boldsymbol{W}^{(4)}\boldsymbol{a}^{(3)} + \boldsymbol{b}^{(4)})\boldsymbol{a}^{(3)^T}}{1 - g(\boldsymbol{W}^{(4)}\boldsymbol{a}^{(3)} + \boldsymbol{b}^{(4)})} - \frac{yg'(\boldsymbol{W}^{(4)}\boldsymbol{a}^{(3)} + \boldsymbol{b}^{(4)})\boldsymbol{a}^{(3)^T}}{g(\boldsymbol{W}^{(4)}\boldsymbol{a}^{(3)} + \boldsymbol{b}^{(4)})}$$

# Parameter Update

Continuing, we note that for this model, $g(z)$ is the logistic sigmoid.

Let's replace $g(z)$ with $\sigma(z)$ and $g'(z)$ with $\sigma'(z)$ to make this clear.

Recall that $\sigma'(z) = \sigma(z)(1 - \sigma(z))$, allowing us to simplify the expression:

$$\frac{\partial \mathscr{L}}{\partial \boldsymbol{W}^{(4)}} = \dots$$

$$= (1 - y)\sigma(\mathbf{W}^{(4)}\mathbf{a}^{(3)} + \mathbf{b}^{(4)})\mathbf{a}^{(3)^T} - y\Big(1 - \sigma(\mathbf{W}^{(4)}\mathbf{a}^{(3)} + \mathbf{b}^{(4)})\Big)\mathbf{a}^{(3)^T}$$

$$= (1 - y)\mathbf{a}^{(4)}\mathbf{a}^{(3)^T} - y(1 - \mathbf{a}^{(4)})\mathbf{a}^{(3)^T}$$

$$= (\mathbf{a}^{(4)} - y)\mathbf{a}^{(3)^T}$$

(predict - actual) * input.T

# Parameter Update

We will now compute the gradients for $\mathbf{W}^{[3]}$.

Instead of deriving $\partial\mathscr{L}/\partial\mathbf{W}^{[3]}$ directly, we'll use the chain rule *i.e.*

$$\frac{\partial\mathscr{L}}{\partial\mathbf{W}^{[3]}} = \frac{\partial\mathscr{L}}{?}\frac{?}{\partial W^{[3]}}$$

If we look at the forward propagation, we may see that loss $\mathscr{L}$ depends on $a^{[4]}$. Hence,

$$\frac{\partial\mathscr{L}}{\partial\mathbf{W}^{[3]}} = \frac{\partial\mathscr{L}}{\partial a^{[4]}}\frac{\partial a^{[4]}}{?}\frac{?}{\partial W^{[3]}}$$

# Parameter Update

We know that $a^{[4]}$ is directly related to $z^{[4]}$. Therefore,

$$\frac{\partial \mathscr{L}}{\partial \mathbf{W}^{[3]}} = \frac{\partial \mathscr{L}}{\partial a^{[4]}} \frac{\partial a^{[4]}}{\partial z^{[4]}} \frac{\partial z^{[4]}}{?} \frac{?}{\partial W^{[3]}}$$

Moreover, we know that $z^{[4]}$ is directly related to $a^{[3]}$. Then,

$$\frac{\partial \mathscr{L}}{\partial \mathbf{W}^{[3]}} = \frac{\partial \mathscr{L}}{\partial a^{[4]}} \frac{\partial a^{[4]}}{\partial z^{[4]}} \frac{\partial z^{[4]}}{\partial a^{[3]}} \frac{\partial a^{[3]}}{?} \frac{?}{\partial W^{[3]}}$$

Again, $a^{[3]}$ depends on $z^{[3]}$, which directly depends on $\mathbf{W}^{[3]}$. Thus,

$$\frac{\partial \mathscr{L}}{\partial \mathbf{W}^{(3)}} = \frac{\partial \mathscr{L}}{\partial \mathbf{a}^{(4)}} \frac{\partial \mathbf{a}^{(4)}}{\partial \mathbf{z}^{(4)}} \frac{\partial \mathbf{z}^{(4)}}{\partial \mathbf{a}^{(3)}} \frac{\partial \mathbf{a}^{(3)}}{\partial \mathbf{z}^{(3)}} \frac{\partial \mathbf{z}^{(3)}}{\partial \mathbf{W}^{(3)}}$$

# Parameter Update

To evaluate this expression, let's try to reuse what we already have for $\frac{\partial \mathscr{L}}{\partial W^{(4)}}$ first:

$$\frac{\partial \mathscr{L}}{\partial \mathbf{W}^{(4)}} = \frac{\partial \mathscr{L}}{\partial \mathbf{a}^{(4)}} \frac{\partial \mathbf{a}^{(4)}}{\partial \mathbf{z}^{(4)}} \frac{\partial \mathbf{z}^{(4)}}{\partial \mathbf{W}^{(4)}} = (\mathbf{a}^{(4)} - y)\mathbf{a}^{(3)}$$

we can reuse the part

$$\frac{\partial \mathscr{L}}{\partial \mathbf{z}^{(4)}} = \frac{\partial \mathscr{L}}{\partial \mathbf{a}^{(4)}} \frac{\partial \mathbf{a}^{(4)}}{\partial \mathbf{z}^{(4)}} = \mathbf{a}^{(4)} - y$$

For the remaining terms, we have:

$$\frac{\partial \mathbf{z}^{(4)}}{\partial \mathbf{a}^{(3)}} = \mathbf{W}^{(4)} \qquad \frac{\partial \mathbf{a}^{(3)}}{\partial \mathbf{z}^{(3)}} = g'(\mathbf{z}^{(3)}) \qquad \frac{\partial \mathbf{z}^{(3)}}{\partial \mathbf{W}^{(3)}} = \mathbf{a}^{(2)}$$

# Parameter Update

While we have greatly simplified the process, we are not done yet! Because we are computing derivatives in higher dimensions, we must reorder the terms such that the dimensions align.

We note the dimensions of all the terms as follows:

$$\underbrace{\frac{\partial \mathscr{L}}{\partial \mathbf{W}^{(3)}}}_{2\times 3} = \underbrace{(\mathbf{a}^{(4)} - y)}_{1\times 1}\ \underbrace{\mathbf{W}^{(4)}}_{1\times 2}\ \underbrace{g'(\mathbf{z}^{(3)})}_{2\times 1}\ \underbrace{(\mathbf{a}^{(2)})}_{3\times 1}$$

# Parameter Update

Putting the chain rule terms together in an order appropriate for vector calculations, we obtain:

$$\underbrace{\frac{\partial \mathscr{L}}{\partial \mathbf{W}^{(3)}}}_{2\times 3} = \underbrace{\mathbf{W}^{(4)^T}}_{2\times 1} \circ \underbrace{g'(\mathbf{z}^{(3)})}_{2\times 1} \underbrace{(\mathbf{a}^{(4)} - y)}_{1\times 1} \underbrace{\mathbf{a}^{(2)^T}}_{1\times 3}$$

The calculation is also similar for the bias weight, except that we have 1 in place of $\mathbf{a}^{(2)}$.

# Parameter Update

The key to solving the problem efficiently is realizing the term that looks like $\partial \mathscr{L} / \partial \mathbf{z}_i^{(l)}$ has already been calculated and can be reused !

Let's therefore define: $\delta_i^{(l)} = \dfrac{\partial \mathscr{L}}{\partial \mathbf{z}_i^{(l)}}$

We then obtain the backpropagation algorithm for a neural network !

(The remaining gradients are left as an exercise to the audiences)

# Fully-connected Network

Given example $(\boldsymbol{x}, y)$:

Recall that $\mathscr{L}(\hat{y}, y) = -\left[y \log \hat{y} + (1-y)\log(1-\hat{y})\right]$

$\mathbf{a}^{(1)} := \boldsymbol{x}$

for $l = 2 \ldots L$ do

$\mathbf{z}^{(l)} := \mathbf{W}^{(l)^T} \circ \mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}$

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

$$\delta^{(L)} := \frac{\partial \mathscr{L}}{\partial \mathbf{z}^{(L)}}$$

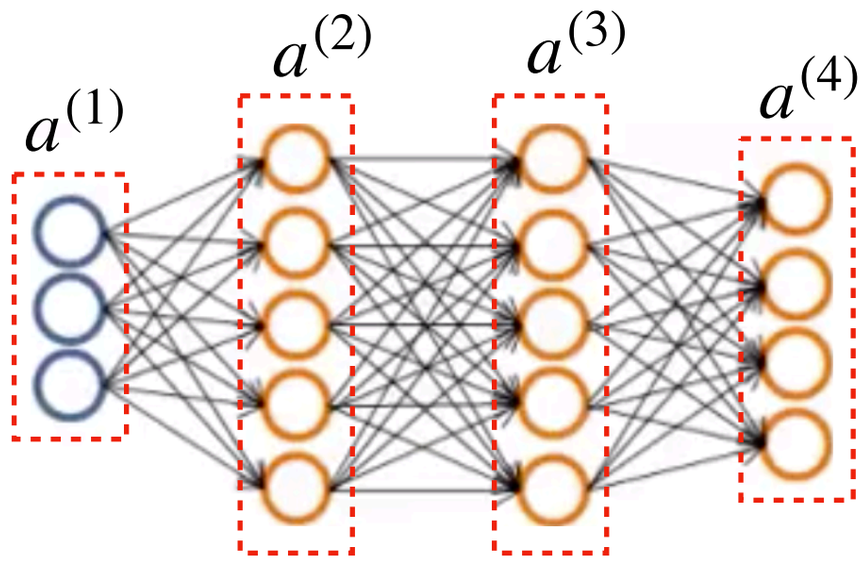
for $l = L \ldots 2$ do

$$\frac{\partial \mathscr{L}}{\partial \mathbf{W}^{(l)}} := \mathbf{a}^{(l-1)} \circ \delta^{(l)^T}$$

$$\frac{\partial \mathscr{L}}{\partial \mathbf{b}^{(l)}} := \delta^{(l)}$$

if $l > 2$ then

$$\delta^{(l-1)} := \left(g'(\mathbf{z}^{(l-1)})\right) \circ \left(\mathbf{W}^{(l)} \circ \delta^{(l)}\right)$$



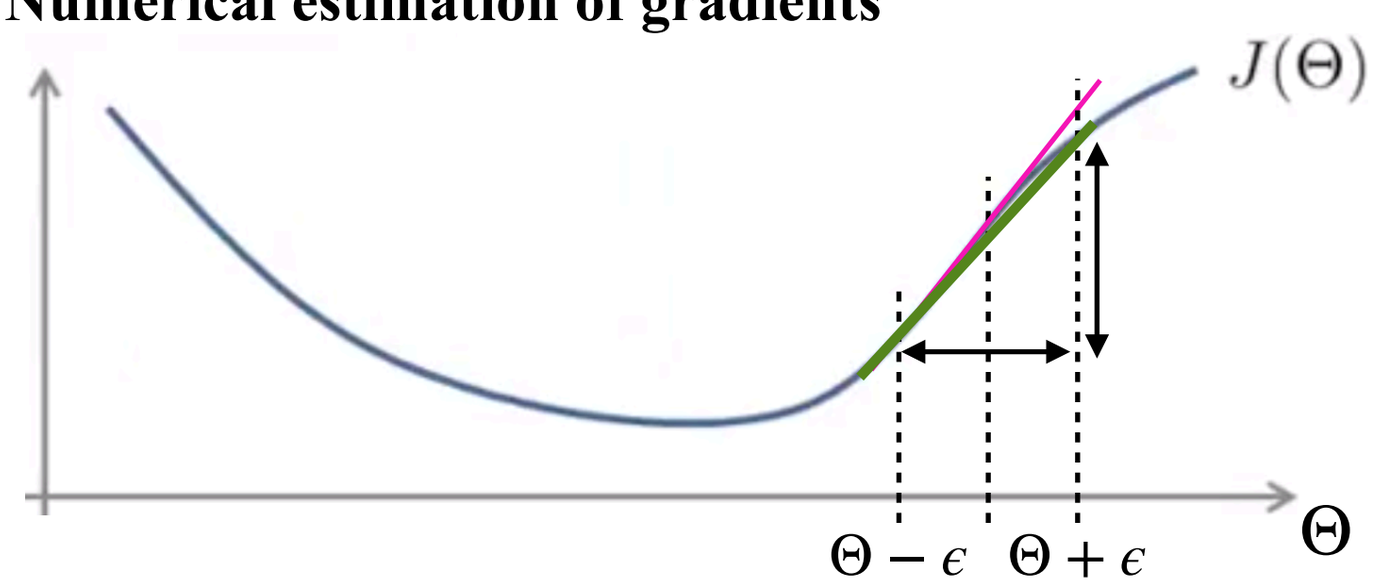

Layer 1 Layer 2 Layer 3 Layer 4

# Neural Networks in Practice

# Gradient Checking

To ensure that our gradient computation (backprop) is implemented correctly by verifying that $\dfrac{J(\Theta+\epsilon)-J(\Theta-\epsilon)}{2\epsilon} \approx$ backprop

**Numerical estimation of gradients**




$$\frac{d}{d\Theta}J(\Theta) \approx \frac{J(\Theta+\epsilon)-J(\Theta-\epsilon)}{2\epsilon}$$

$e.g.\ \ \epsilon = 10^{-4}$

$(\Theta \in \mathbb{R})$

# Question

Let $J(\theta) = \theta^3$. Furthermore, let $\theta = 1$ and $\epsilon = 0.01$. You use the formula:

$$\frac{J(\theta + \epsilon) - J(\theta - \epsilon)}{2\epsilon}$$

to approximate the derivative. What value do you get using this approximation?

(When $\theta = 1$, the true, exact derivative is $\frac{d}{d\theta} J(\theta) = 3$).

(i) 3.0000
(ii) 3.0001
(iii) 3.0301
(iv) 6.0002

# Implementation Guideline

- Implement backprop to compute $\delta^{(4)}, \delta^{(3)}, \delta^{(2)}$
- Implement numerical gradient checking
- See that they give similar values
- Turn off gradient checking and use back prop for learning

✴ Be sure to disable your gradient checking code before training your classifier. If you run numerical gradient computation on every iteration of gradient descent, your code will be ‘very’ slow.

# Question

- What is the main reason that we use the backpropagation algorithm rather than the numerical gradient computation method during learning?

(i) The numerical gradient computation method is much harder to implement.

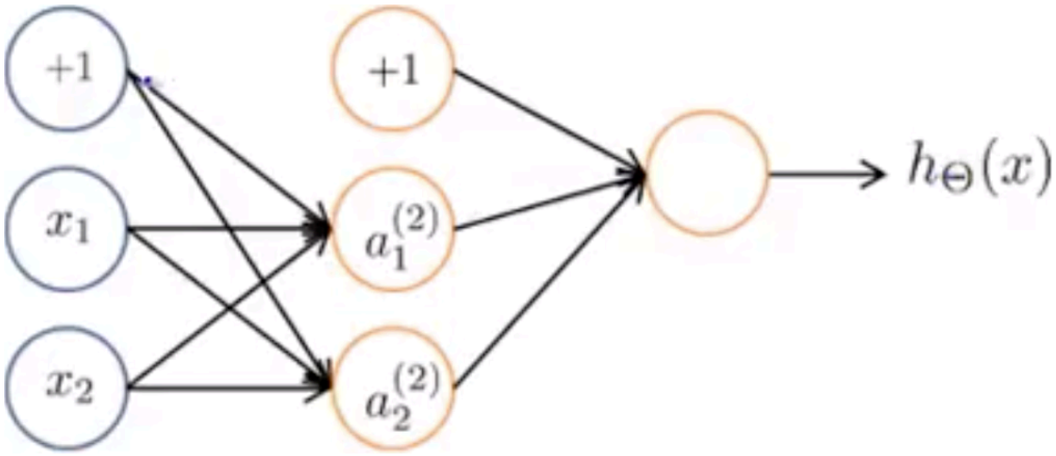
(ii) The numerical gradient algorithm is very slow.

(iii) Backpropagation does not require setting the parameter $\epsilon$

(iv) None of the above.

# Parameter Initialization

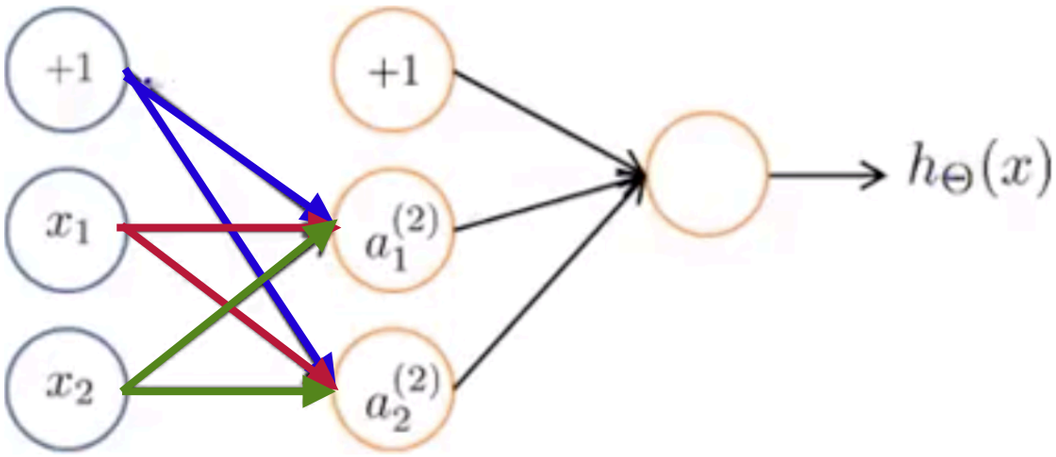
DON'T initialize all parameters to 0 ! (Why should we avoid this?)




$$\Theta_{ij}^{(l)} = 0 \text{ for all } i, j, l$$

# Parameter Initialization

DON'T initialize all parameters to 0 ! (Why should we avoid this?)




$$\Theta_{ij}^{(l)} = 0 \text{ for all } i, j, l$$

$$\left.\begin{aligned} a_1^{(2)} &= \Theta_{10}^{(1)} \times 1 + \Theta_{11}^{(1)} \times x_1 + \Theta_{12}^{(1)} \times x_2 \\ a_2^{(2)} &= \Theta_{20}^{(1)} \times 1 + \Theta_{21}^{(1)} \times x_1 + \Theta_{22}^{(1)} \times x_2 \end{aligned}\right\} \therefore a_1^{(2)} = a_2^{(2)} \text{ and } \delta_1^{(2)} = \delta_2^{(2)}$$

**Thus,** $\dfrac{\partial}{\partial \Theta_{10}^{(1)}} J(\Theta) = \dfrac{\partial}{\partial \Theta_{20}^{(1)}} J(\Theta)$ which implies $\Theta_{10}^{(1)} = \Theta_{20}^{(1)}$

# Parameter Initialization

Initializing to 0 would be a bad idea because the output of each layer would be identical for every unit, and the gradients backpropagated later would also be identical.

Solution: randomly initialize each $\Theta_{ij}^{(l)}$ from $[-\epsilon, \epsilon]$ (this should be small)

*e.g.* close to 0: $\Theta_{jk}^{(i)} \sim \mathcal{N}(0,0.1)$

A better method in practice is called 'Xavier/He' initialization:

$$\Theta_{jk}^{(i)} \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n^{(l)} + n^{(l-1)}}}\right)$$

where $n^{(l)}$ is the number of units in layer $l$. This encourages the variance of the outputs of a layer to be similar to the variance of the inputs.

# Parameter Initialization

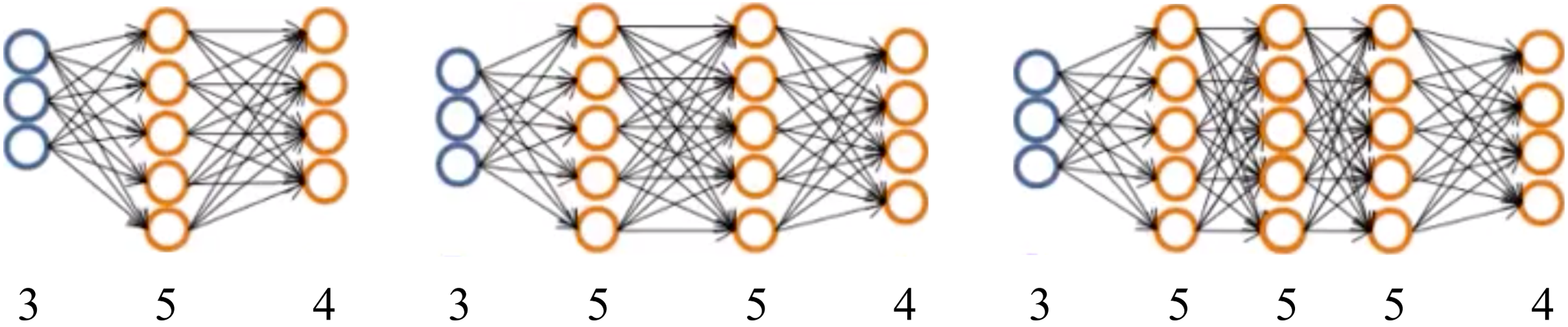
Noted that, for ReLU hidden units, the recommendation for the bias weights is to use a small positive (even constant) initial value.

This ensures that the unit's output is initially positive for most training examples.

# Putting them together

# 1. Pick a network architecture

Network architecture means connectivity pattern between neurons *e.g.*

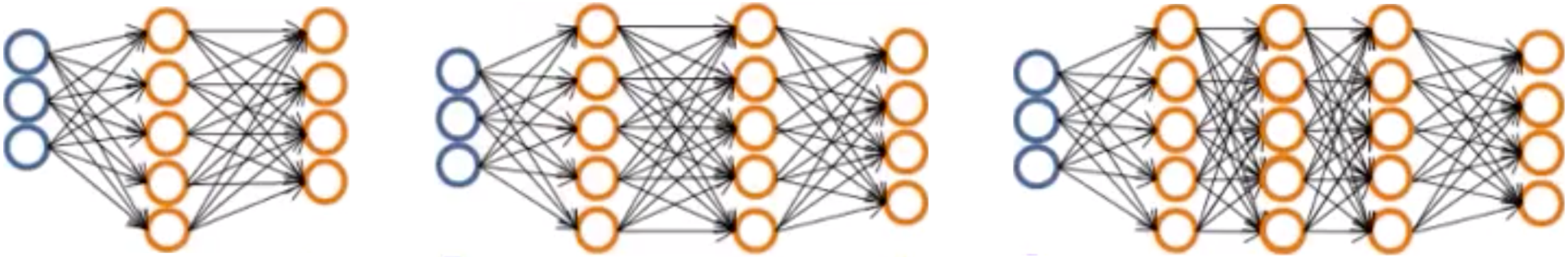



**Architecture choices:**

- how many units in a hidden layer?
- how many hidden layers in a neural network?

# 1. Pick a network architecture

Network architecture means connectivity pattern between neurons *e.g.*




reasonably straightforward
- #input units = dimension of features $x^{(i)}$
- #output units = number of classes (if we are doing a classification problem)

- #hidden layer = 1 (default); otherwise, having the same #hidden units in every layer.
- usually, the more #hidden units, the better (*i.e.* fit to the training dataset).

# 2. Training a neural network

1. Randomly initialize weights
2. Implement forward propagation to get $h_\Theta(x^{(i)})$ for any $x^{(i)}$
3. Implement code to compute cost function $J(\Theta)$
4. Implement backprop to compute partial derivatives $\dfrac{\partial J(\Theta)}{\partial \Theta_{jk}^{(l)}}$

**Forward propagation & Backpropagation:**

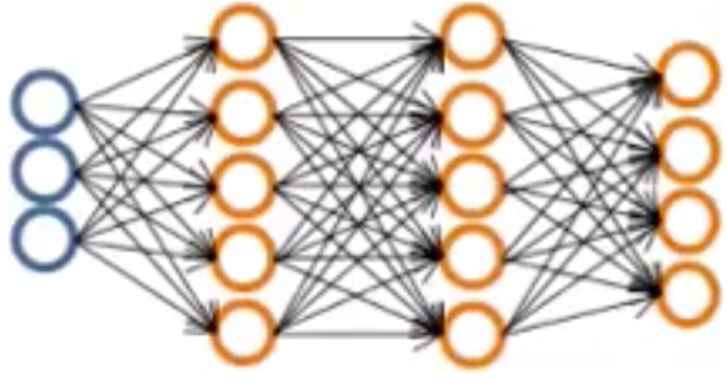
for i = 1 to m:

Perform forward propagation and backpropagation using example $(x^{(i)}, y^{(i)})$

*i.e.* get activations $a^{(l)}$ and delta terms $\delta^{(l)}$ for $l = 2,\ldots,L$

$\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)}(a^{(l)})^T$

(used to compute $\dfrac{\partial}{\partial \Theta_{jk}^{(}l)} J(\Theta)$)

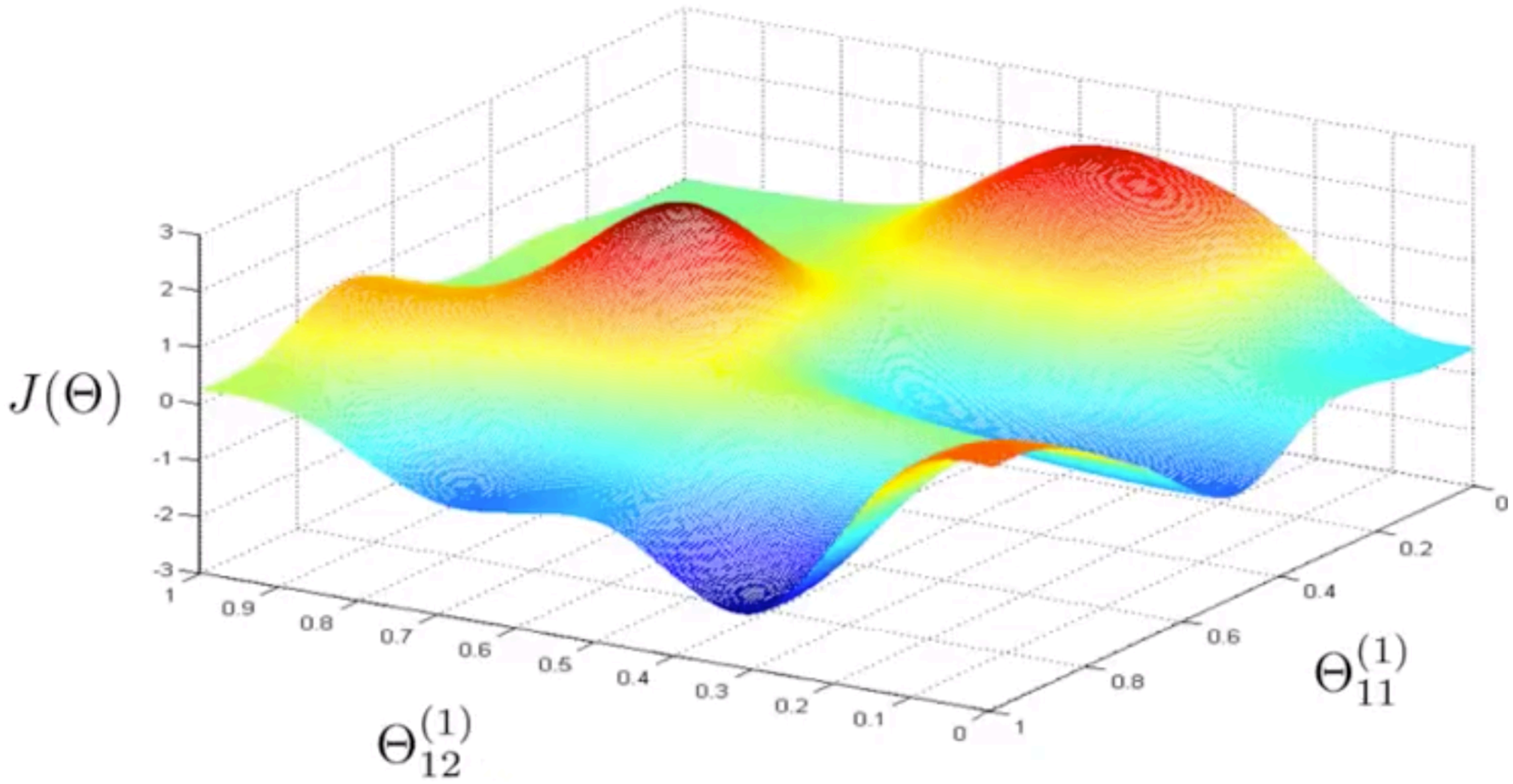


# 2. Training a neural network

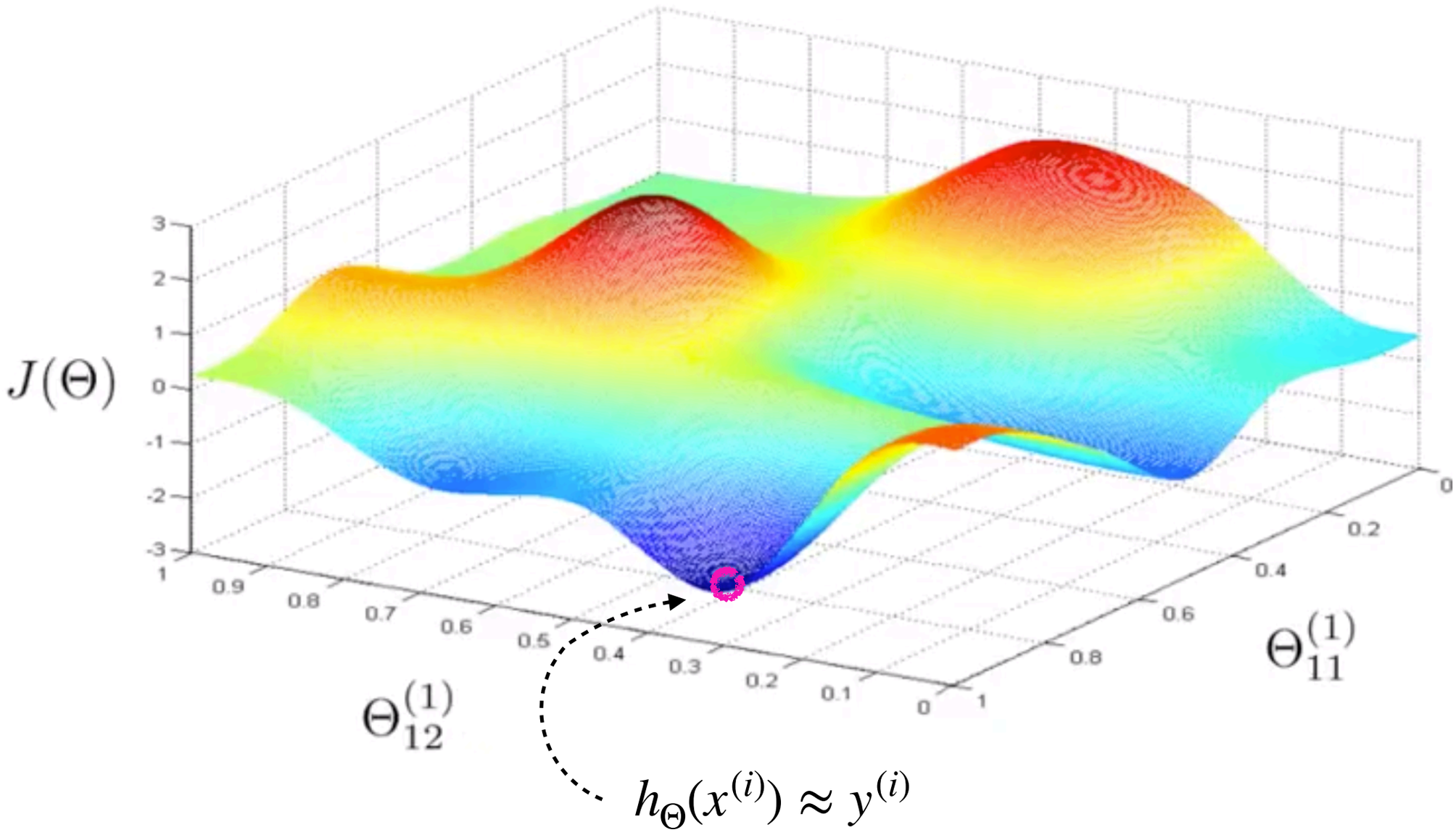
5. Use gradient checking to compare $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$ computed using backpropagation *vs.* using numerical estimate of gradient $J(\Theta)$. Then, disable gradient checking code.
6. Use gradient descent with backpropagation to try to minimize $J(\Theta)$ as a function of parameters $\Theta$

**Remark:** $J(\Theta)$ is non-convex *i.e.* using (batch) gradient descent algorithm can get stuck in a local optima.

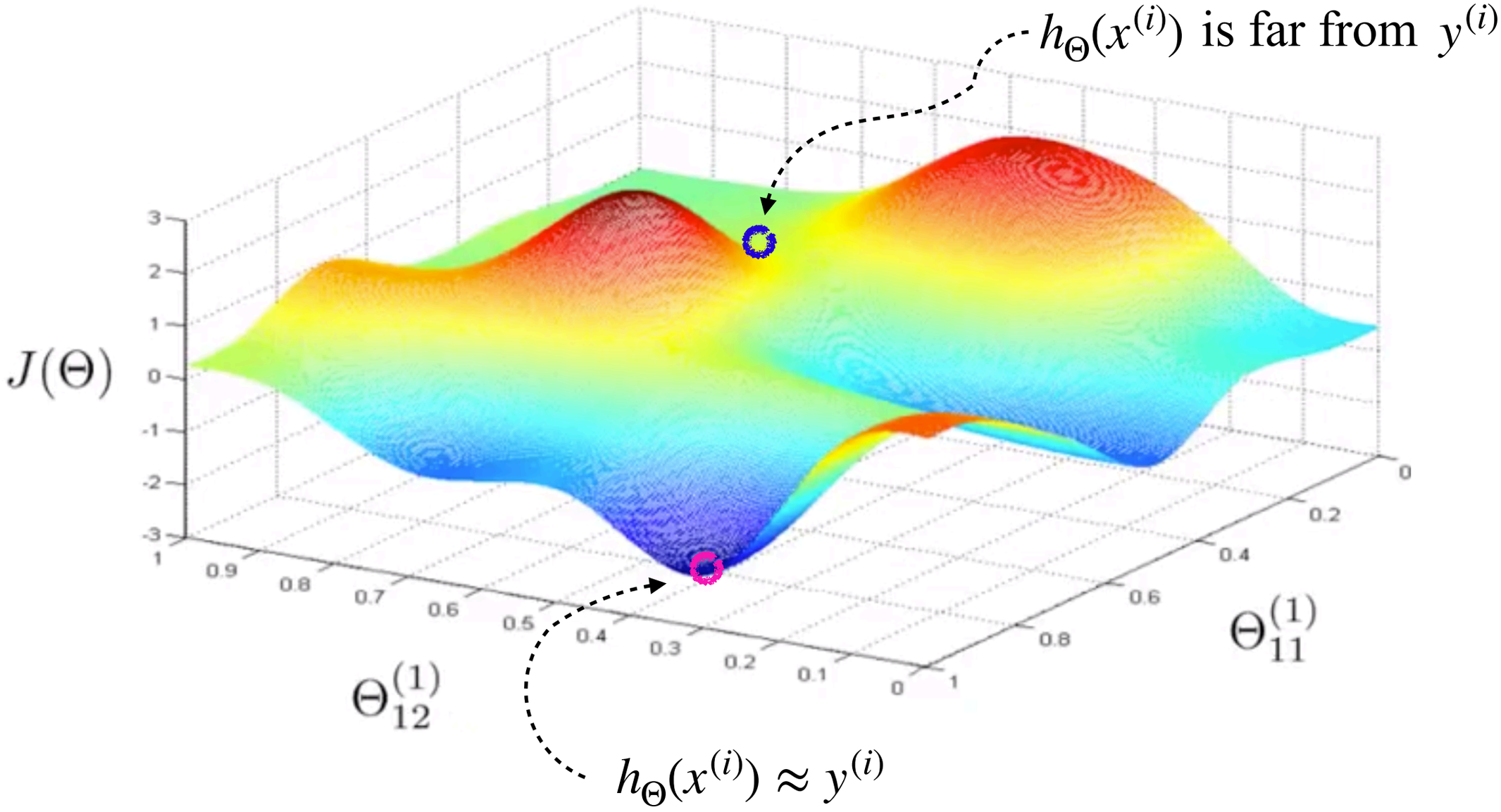
# Backprop with GD



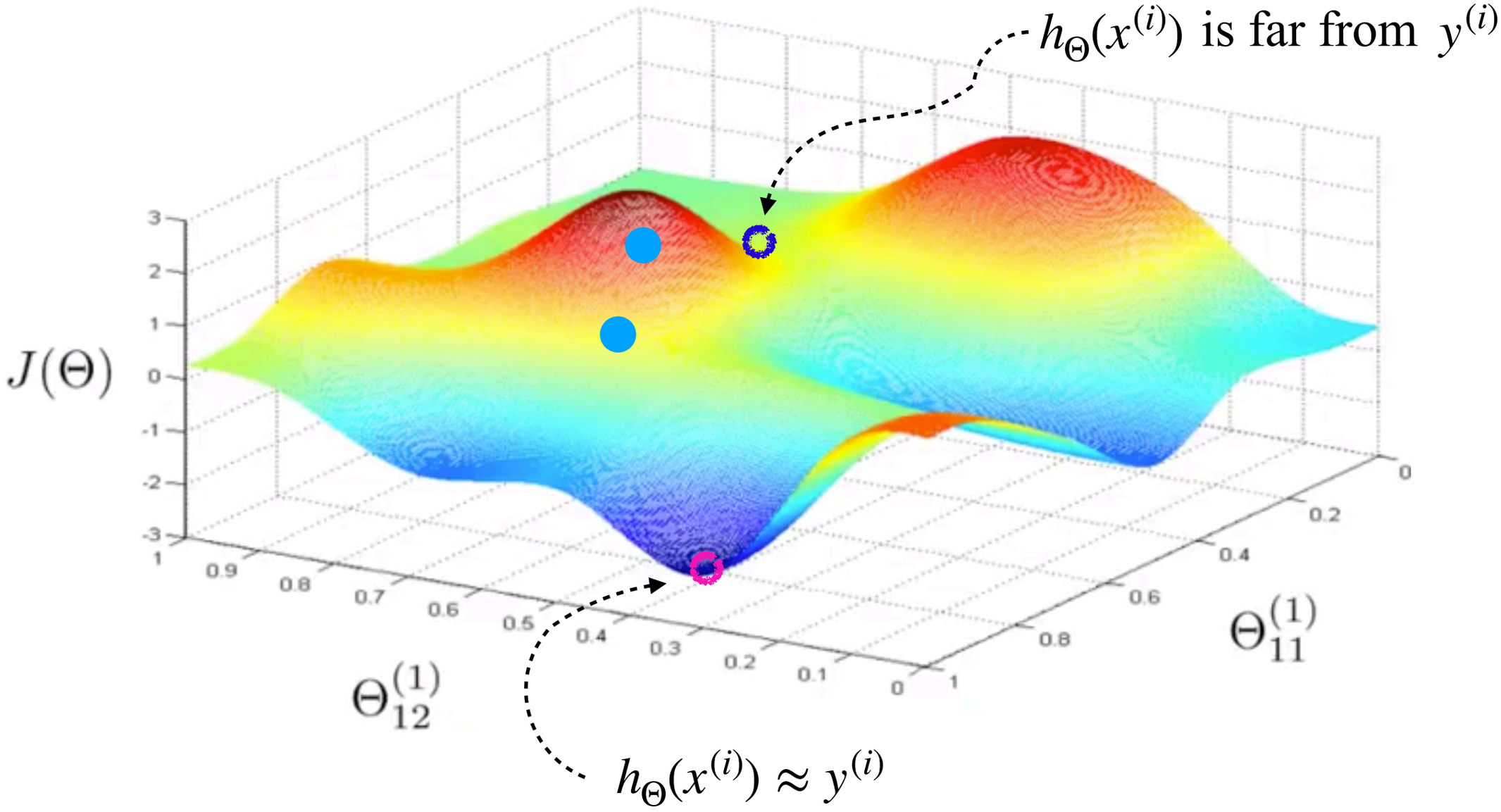
# Backprop with GD



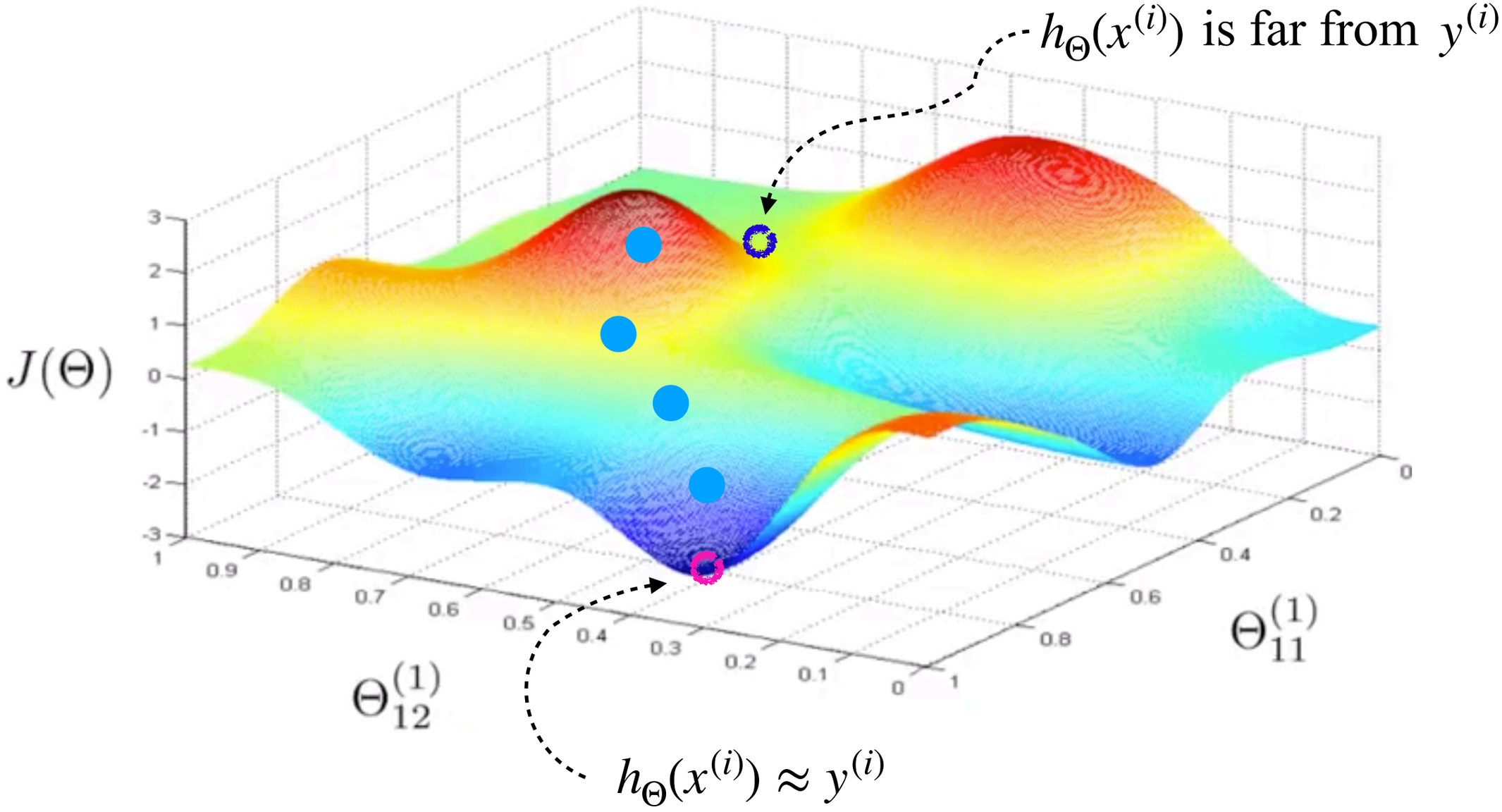
# Backprop with GD



# Backprop with GD



# Backprop with GD



# Question

Suppose you are using gradient descent together with backpropagation to try to minimize $J(\Theta)$. Which of the following would be a useful step for verifying that the learning algorithm is running correctly?

(i) Plot $J(\Theta)$ as a function of $\Theta$, to make sure gradient descent is going downhill

(ii) Plot $J(\Theta)$ as a function of iterations and make sure it is increasing (or at least non-decreasing) with every iteration

(iii) Plot $J(\Theta)$ as a function of iterations and make sure it is decreasing (or at least non-increasing) with every iteration

(iv) Plot $J(\Theta)$ as a function of the number of iterations to make sure the parameter values are improving in classification accuracy.

# Optimization Procedure

# Stochastic *vs.* Batch Revisit !

Thus far, the derivation was for a single $(\boldsymbol{x}, y)$ pair !

What's about batch gradient descent? We would use the rule:

$$\mathbf{W}^{(l)} := \mathbf{W}^{(l)} - \alpha \frac{\partial J}{\partial \mathbf{W}^{(l)}},$$

where $J$ is the cost function:

$$J = \frac{1}{m} \Sigma_{i=1}^{m} \mathscr{L}^{(i)}$$

and $\mathscr{L}^{(i)}$ is the loss for a single example.

Stochastic gradient descent will be more noisy but will usually converge faster than batch gradient descent.

# Mini-batch Revisit !

Since batch gradient descent is more accurate but slower to get moving than stochastic gradient descent, a common compromise is mini-batch gradient descent.

As previously discussed, the mini-batch is a compromise between the accuracy of batch and the speed of stochastic gradient descent.

That is, we split the data set into partitions $B_1, B_2, \ldots$ (or, repeatedly sample uniformly from the full training set) and let:

$$J_i = \frac{1}{|B_i|} \Sigma_{j \in B_i} \mathscr{L}^{(i)}$$

# Momentum

Another common optimization is called momentum.

The problem is that sometimes noisy data make us jump back and forth across valleys in the loss function, leading to slow convergence.

With momentum, we remember the last update to each parameter and use it to calculate a moving average of the gradient over time.

We use the following update rule:

$$\mathbf{V}_{d\mathbf{W}^{(l)}} := \beta \mathbf{V}_{d\mathbf{W}^{(l)}} + (1-\beta)\frac{\partial J}{\partial \mathbf{W}^{(l)}}$$

$$\mathbf{W}^{(l)} := \mathbf{W}^{(l)} - \alpha \mathbf{V}_{d\mathbf{W}^{(l)}}$$

The momentum term $\beta$ will encourage the optimization to accelerate gradually toward the minimum.

# Avoid Overfitting

# Overfitting

Neural networks are universal approximators.

Roughly speaking, given enough training data and sufficient model complexity, backpropagation can learn any function to an arbitrary level of accuracy.

But, when model complexity is too high for the training set size, we observe overfitting: training accuracy is high, but test set accuracy is substantially lower.

Solutions:

1. Decrease model complexity (*e.g.* remove hidden units / layers)
2. Collect more training data
3. Regularization

# Regularization

Let $\mathbf{w}$ denote a single vector containing the set of all parameters in our model (every $w_{ij}^{(l)}$ and $b_i^{(l)}$).

Let $J$ be the model cost function.

L2 regularization adds a new term to the cost function:

$$J_{L2} = J + \frac{\lambda}{2}\|\mathbf{w}\|^2 = J + \frac{\lambda}{2}\mathbf{w}^T\mathbf{w}$$

Why do this?

- $\lambda = 0$ gives us unregularized parameter learning;
- Big $\lambda$ encourages solutions with small $\mathbf{w}$, especially, as many 0 elements as possible.

Forcing less useful parameters to 0 decreases model complexity and will reduce overfitting.

# Regularization

How does the regularizer affect the learning rule?

Previously, we had $\mathbf{w} := \mathbf{w} - \alpha \frac{\partial J}{\partial \mathbf{w}}$

Now, we have $\mathbf{w} := \mathbf{w} - \alpha \frac{\partial J}{\partial \mathbf{w}} - \alpha \frac{\lambda}{2} \frac{\partial \mathbf{w}^T \mathbf{w}}{\partial \mathbf{w}}$

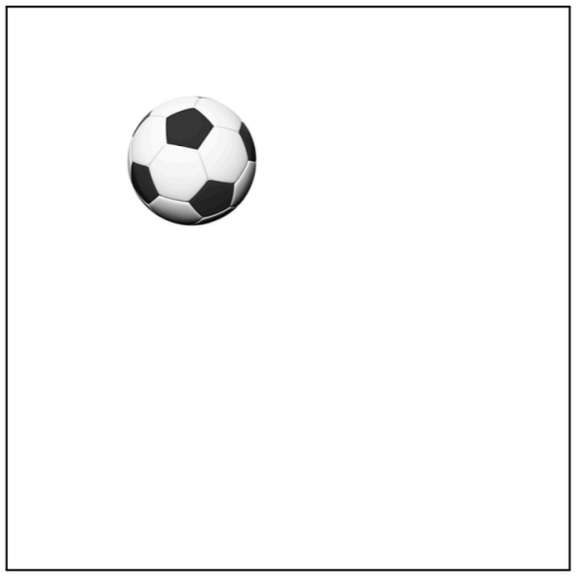
$$\iff (1 - \alpha\lambda)\mathbf{w} - \alpha \frac{\partial J}{\partial \mathbf{w}}$$

Think about what this means ( $\alpha$ and $\lambda$ are both small positive reals).

On each update, we make all the weights' magnitudes a little smaller.
Only the most important weights will survive.

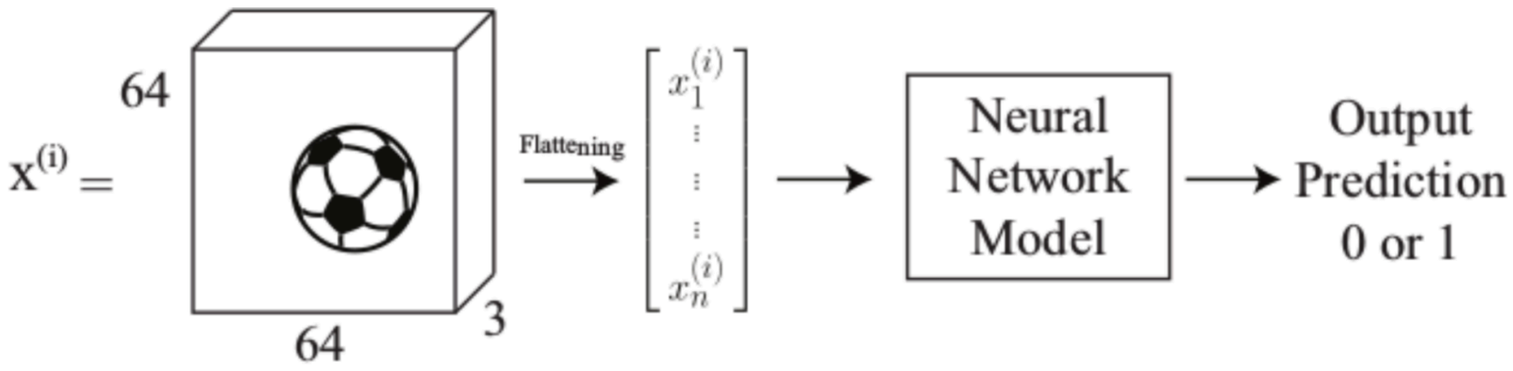
# Parameter Sharing

Let's use an example of classifying an image as containing a soccer ball or not



**Procedure:**

1. scale the image to a standard size, *e.g.* $64 \times 64$
2. flatten the $64 \times 64 \times 3$ elements of the input to a 12,288-element vector and present to our neural network.




(from Ng (2017), CS 229 Lecture note on Deep Learning)

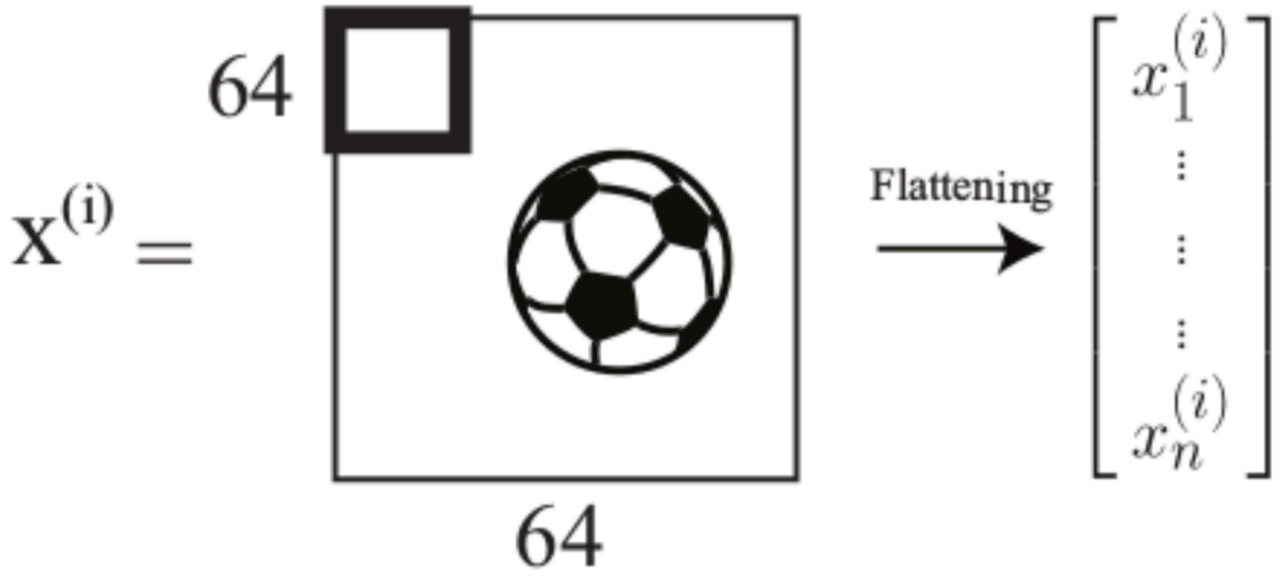
# Parameter Sharing

If we used a single logistic regression model for the $64 \times 64 \times 3 = 12{,}288$ parameters of the model, we would have to train with soccer balls in all possible positions in the image.

Such a model would fail if it encountered a ball in a position it had never seen during training.

One solution is parameter sharing *i.e.* each unit in the hidden layer looks at a different overlapping sub-region of the image, but these units share the same weights.

# Parameter Sharing




(from Ng (2017), CS 229 Lecture note on Deep Learning)

We might draw $\theta$ from $\mathbb{R}^{4\times4\times3}$, meaning we have one weight for each of the R, G, and B pixel intensities in a $4 \times 4$ region.

# Parameter Sharing

To learn $\theta$:

- We might slide the region of interest over each position training image to create many positive 48-element training vectors; or
- We might represent each element in the 'convolution' as a separate unit but tie their weights together during backpropagation.

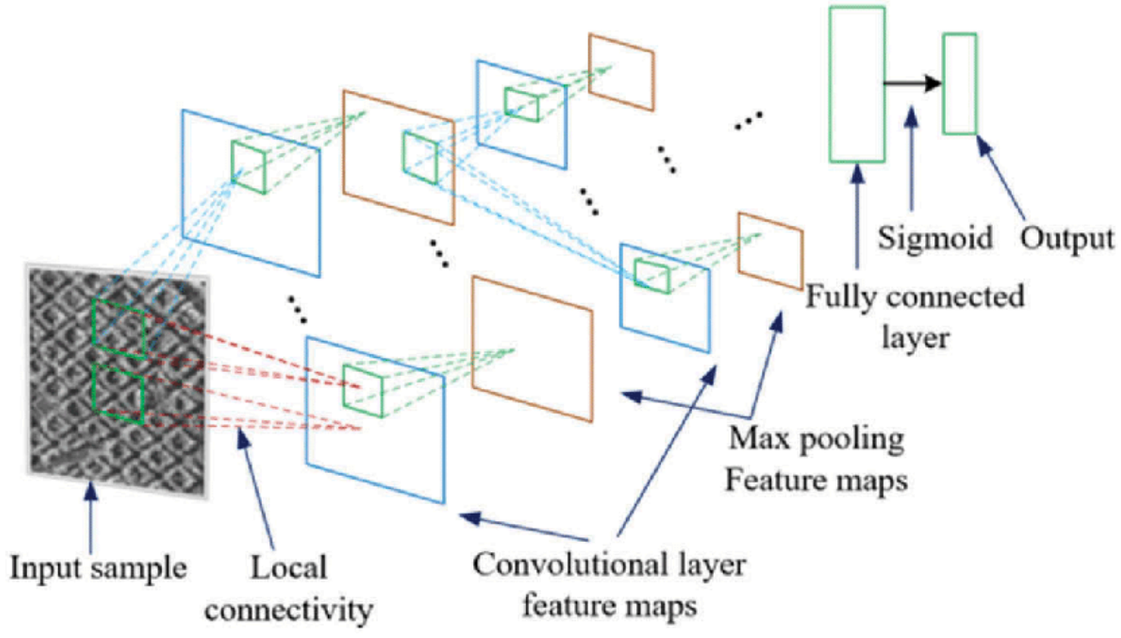
The convolution style of weight sharing is the basic idea of the convolutional neural network (CNN):

- Inspired by Hubel and Weisel's model of the response of neurons in the cat's visual cortex to visual stemuli.
- Inspired by Fukushima's Neocognitron (1988).

# Parameter Sharing

CNNs:

- Applied successfully to handwritten character recognition by LeCun and colleagues (LeNet 5, 1988)
- Won the ImageNet Large Scale Visual Recognition Challenge in 2012 and every ImageNet competition since then.
- Jump started the current round of hype in machine learning and AI !




(from DOI: 10.1080/2150704X.2017.1335906)

# Summary

- We have gone through a very brief introduction to deep learning.
- Hopefully, you get the main ideas and can see how to extend them to real-world problems.
- Further reading:
  - Read up the literature, beginning with AlexNet then moving forward.
  - A nice resource: https://adeshpande3.github.io/ The-9-Deep-Learning-Papers-You-Need-To-Know-About.html
- Later in this class, we will AGAIN come back to another topic of Deep Learning ‘CNNS for image classification’.