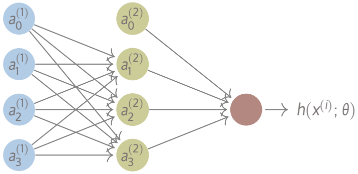
Neural Network Cost Function and Backpropagation

Lecture 9 - DAMLF | ML1

hmi human+machine INTELLIGENCE
Frankfurt School

Review: Forward Propagation



Equations to plug in:

$$z^{(2)} = \begin{bmatrix} z_1^{(2)} \\ z_2^{(2)} \\ z_3^{(2)} \end{bmatrix} \qquad a^{(1)} = \begin{bmatrix} a_0^{(1)} \\ a_1^{(1)} \\ a_2^{(1)} \\ a_3^{(1)} \end{bmatrix}$$

$$a_1^{(2)} = g\left(\Theta_{10}^{(1)} a_0^{(1)} + \Theta_{11}^{(1)} a_1^{(1)} + \Theta_{12}^{(1)} a_2^{(1)} + \Theta_{13}^{(1)} a_3^{(1)}\right)$$

$$a_2^{(2)} = g\left(\Theta_{20}^{(1)} a_0^{(1)} + \Theta_{21}^{(1)} a_1^{(1)} + \Theta_{22}^{(1)} a_2^{(1)} + \Theta_{23}^{(1)} a_3^{(1)}\right)$$

$$a_3^{(2)} = g\left(\Theta_{30}^{(1)} a_0^{(1)} + \Theta_{31}^{(1)} a_1^{(1)} + \Theta_{32}^{(1)} a_2^{(1)} + \Theta_{33}^{(1)} a_3^{(1)}\right)$$

$$h(x^{(i)};\theta) = g\left(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)}\right)$$

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

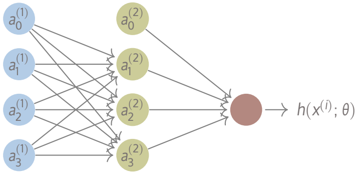
$$a^{(2)} = g\left(z^{(2)}\right)$$

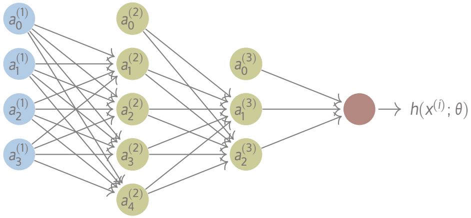
$$z^{(3)} = \Theta^{(2)} a^{(2)}$$

$$a^{(3)} = h(x^{(i)};\theta) = g\left(z^{(3)}\right)$$

$a_0^{(2)} = 1 = a_0^{(1)}$ # bias units

Review: Neural Network Architecture





Multi-class Classification with Neural Networks

One-vs-all Networks with Multiple Outputs

fish

amphibians

birds

reptiles

mammals

One-vs-all Networks with Multiple Outputs

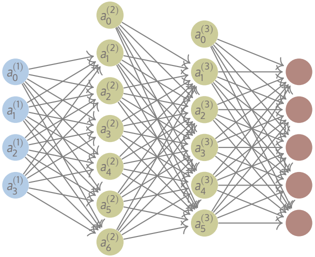
fish

amphibians

birds

reptiles

mammals



$h(x^{(i)}; \theta) \in \mathbb{R}^5$

One-vs-all Networks with Multiple Outputs

fish

$$h(x^{(i)};\theta) = \begin{bmatrix}1\\0\\0\\0\\0\end{bmatrix}$$

amphibians

$$h(x^{(i)};\theta) = \begin{bmatrix}0\\1\\0\\0\\0\end{bmatrix}$$

birds

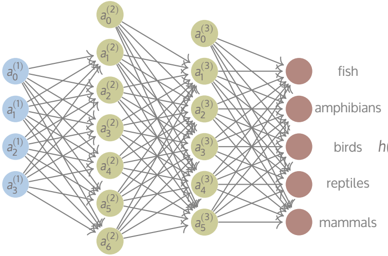
$$h(x^{(i)};\theta) = \begin{bmatrix}0\\0\\1\\0\\0\end{bmatrix}$$

reptiles

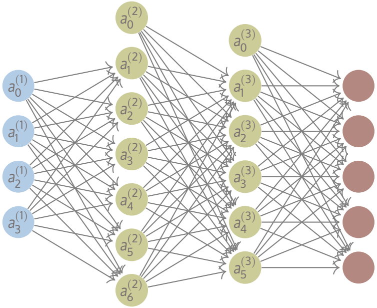
$$h(x^{(i)};\theta) = \begin{bmatrix}0\\0\\0\\1\\0\end{bmatrix}$$

mammals

$$h(x^{(i)};\theta) = \begin{bmatrix}0\\0\\0\\0\\1\end{bmatrix}$$



$h(x^{(i)};\theta) \in \mathbb{R}^5$

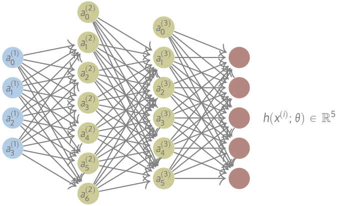


Training Set: $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})$, where each $y^{(i)}$ is **one-hot encoded** as

$$y^{(i)} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \text{ or } \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \text{ or } \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \text{ or } \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \text{ or } \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

Goal: $h(x^{(i)}; \theta) \approx y^{(i)}$

Multi-class Classification Notation

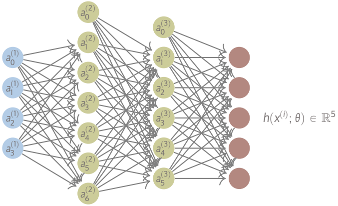


Training set: $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})\}$

Notation:

m	=	Number of training examples
L	=	Number of layers in the network

Multi-class Classification Notation

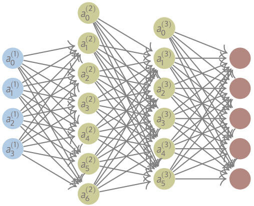


Training set: $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})\}$

Notation:

m	=	Number of training examples
L	=	Number of layers in the network
$a^{(1)}$	=	Input variables / Input layer
$a^{(2)}, \ldots, a^{(L-1)}$	=	Hidden layers
$a^{(L)}$	=	Output layer
s_l	=	Number of units (**not** counting the bias unit) in layer l

Multi-class Classification Notation



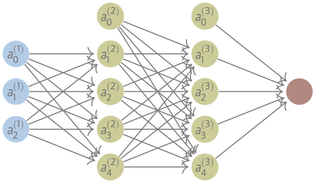
Training set: $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})\}$

Example:

$$
\begin{aligned}
m &= \text{Number of training examples} \\
L &= 4 \\
s_1 &= 3 \\
s_2 &= 6 \\
s_3 &= 5 \\
s_4 &= 5
\end{aligned}
$$

Binary & Multi-class Classification

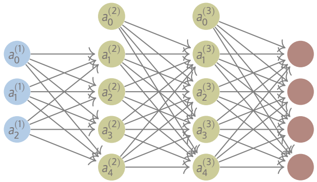
Binary Classification



1 output unit: $h(x^{(i)}; \theta) \in \mathbb{R}$

$y = 1$ or $y = 0$

Multi-class Classification (K-classes)

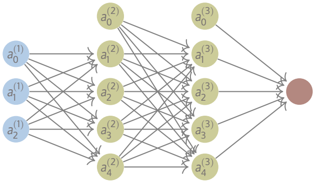


K output units: $h(x^{(i)}; \theta) \in \mathbb{R}^K$

$$y^{(i)} = \begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\0\\1\end{bmatrix}, \text{ for } K = 4.$$

Binary & Multi-class Classification

Binary Classification

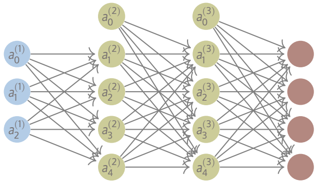


1 output unit: $h(x^{(i)}; \theta) \in \mathbb{R}$

$y = 1$ or $y = 0$

$s_L = 1 \quad (\text{and } K = 1)$

Multi-class Classification (K-classes)



K output units: $h(x^{(i)}; \theta) \in \mathbb{R}^K$

$$y^{(i)} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \text{ for } K = 4.$$

$s_L = K \quad (\text{and } K > 2)$

Cost Function for Logistic Regression

Review: Logistic Regression Cost Function

Regularized Cost Function:

$$J(\Theta) = -\frac{1}{m}\left(\sum_{i=1}^{m} y^{(i)} \log h(x^{(i)};\theta) + (1 - y^{(i)}) \log\left(1 - h(x^{(i)};\theta)\right)\right) + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

Cost Function for Neural Networks

Logistic Regression Cost Function for $h(x^{(i)}; \Theta) \in \mathbb{R}$:

$$J(\Theta) = -\frac{1}{m}\left(\sum_{i=1}^{m} y^{(i)} \log h(x^{(i)}; \theta) + (1 - y^{(i)}) \log\left(1 - h(x^{(i)}; \theta)\right)\right) + \frac{\lambda}{2m}\sum_{j=1}^{n} \theta_j^2$$

$$J(\Theta) = -\frac{1}{m}\left(\quad \text{Cost Function} \quad \right) + \text{Regularization Term}$$

Cost Function for Neural Networks

Logistic Regression Cost Function for $h(x^{(i)}; \Theta) \in \mathbb{R}$:

$$J(\Theta) = -\frac{1}{m}\left(\sum_{i=1}^{m} y^{(i)} \log h(x^{(i)}; \theta) + (1 - y^{(i)}) \log \left(1 - h(x^{(i)}; \theta)\right)\right) + \frac{\lambda}{2m}\sum_{j=1}^{n} \theta_j^2$$

$$J(\Theta) = -\frac{1}{m}\left(\quad \text{Cost Function} \quad \right) + \text{Regularization Term}$$

Neural Network Cost Function for $h(x^{(i)}; \theta) \in \mathbb{R}^K$

$$J(\Theta) = -\frac{1}{m}\left(\text{K Cost Functions} \quad \right) + \text{L-1 layers of Regularization Terms}$$

Cost Function for Neural Networks

Neural Network Cost Function for $h(x^{(i)}; \Theta) \in \mathbb{R}^K$

$$J(\Theta) = -\frac{1}{m}\left(\text{K Cost Functions} \quad \right) \quad + \quad \text{L-1 layers of Regularization Terms}$$

Cost Function for Neural Networks

Neural Network Cost Function for $h(x^{(i)}; \Theta) \in \mathbb{R}^K$

$$J(\Theta) = -\frac{1}{m}\,(\text{ K Cost Functions }) \quad + \quad \text{L-1 layers of Regularization Terms}$$

$$J(\Theta) = -\frac{1}{m}\left(\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(h(x^{(i)}); \Theta)_k + (1 - y_k^{(i)}) \log\left(1 - h(x^{(i)}; \Theta)\right)_k\right) + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(\Theta_{ij}^{(l)}\right)^2$$

Cost Function for Neural Networks

Neural Network Cost Function for $h(x^{(i)}; \Theta) \in \mathbb{R}^K$

$$J(\Theta) = -\frac{1}{m}\left(\text{ K Cost Functions } \right) + \text{ L-1 layers of Regularization Terms}$$

$$J(\Theta) = -\frac{1}{m}\left(\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(h(x^{(i)}); \Theta)_k + (1 - y_k^{(i)}) \log\left(1 - h(x^{(i)}; \Theta)\right)_k\right) + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(\Theta_{ij}^{(l)}\right)^2$$

where $h(x^{(i)}; \Theta)_k$ is the kth output node of the $\mathbb{R}^K$ output vector, and

$$\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(\Theta_{ij}^{(l)}\right)^2 = \left\|\Theta^{(l)}\right\|_F^2$$

is the Frobenius norm, i.e., *the sum of squared elements of a matrix.*

Note that $\Theta_{i,0}$ is not included in the regularization terms.

Backpropagation for Minimizing the Cost Function of Neural Networks

Gradient Computation

Cost Function:

$$J(\Theta) = -\frac{1}{m}\left(\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(h_\Theta(x^{(i)}))_k + (1 - y_k^{(i)}) \log\left(1 - h(x^{(i)};\theta)\right)_k\right) + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(\Theta_{ij}^{(l)}\right)^2$$

Optimization Objective:

$$\min_{\Theta} J(\Theta)$$

Gradient Computation

Cost Function:

$$J(\Theta) = -\frac{1}{m}\left(\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(h_\Theta(x^{(i)}))_k + (1 - y_k^{(i)}) \log\left(1 - h(x^{(i)};\theta)\right)_k\right) + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(\Theta_{ij}^{(l)}\right)^2$$

Optimization Objective:

$$\min_{\Theta} J(\Theta)$$

2-step Strategy:

Compute	Dimension	How?
$J(\Theta)$	$\lvert\Theta_{ij}^{l}\rvert = \sum_{j=1}^{L-1} s_{j+1} \times s_j + 1$	**Forward Propagation**
$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$	$\Theta_{ij}^{l} \in \mathbb{R}$	

Gradient Computation

Cost Function:

$$J(\Theta) = -\frac{1}{m}\left(\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(h_\Theta(x^{(i)}))_k + (1 - y_k^{(i)}) \log\left(1 - h(x^{(i)};\theta)\right)_k\right) + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(\Theta_{ij}^{(l)}\right)^2$$

Optimization Objective:

$$\min_{\Theta} J(\Theta)$$

2-step Strategy:

Compute	Dimension	How?
$J(\Theta)$	$\lvert\Theta_{ij}^{l}\rvert = \sum_{j=1}^{L-1} s_{j+1} \times s_j + 1$	**Forward Propagation**
$\frac{\partial}{\partial\Theta_{ij}^{(l)}} J(\Theta)$	$\Theta_{ij}^{l} \in \mathbb{R}$	**Back Propagation**

Gradient Computation

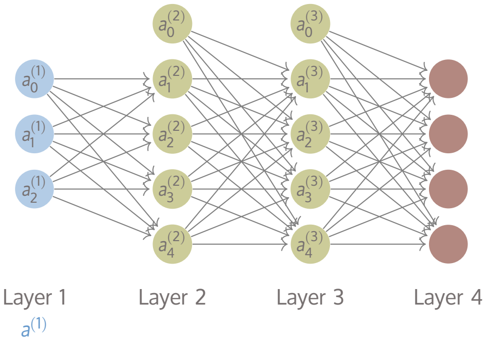
To understand how back propagation works to compute the partial derivative terms $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$, let's focus on a simple example involving just **one** training example, (x, y).

Gradient Computation

Given (x, y):

1. Forward Propagation

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

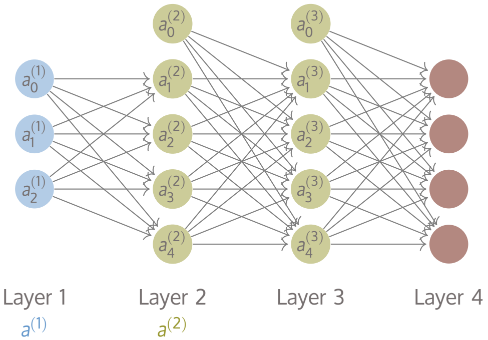


Gradient Computation

Given (x, y):

1. Forward Propagation

$$
\begin{aligned}
a^{(1)} &= x \\
z^{(2)} &= \Theta^{(1)} a^{(1)} \\
a^{(2)} &= g(z^{(2)})
\end{aligned}
$$

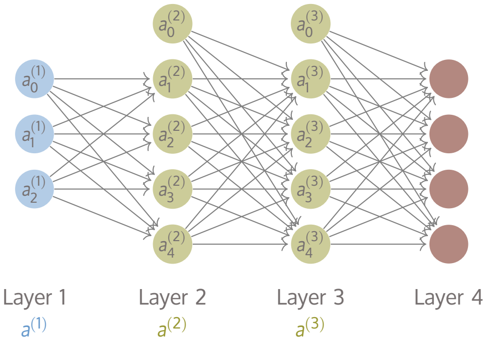


Gradient Computation

Given (x, y):

1. Forward Propagation

$$
\begin{aligned}
a^{(1)} &= x \\
z^{(2)} &= \Theta^{(1)} a^{(1)} \\
a^{(2)} &= g(z^{(2)}) \\
z^{(3)} &= \Theta^{(2)} a^{(2)} \\
a^{(3)} &= g(z^{(3)})
\end{aligned}
$$

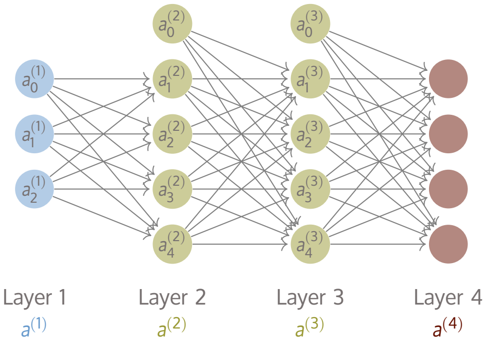


Gradient Computation

Given (x, y):

1. Forward Propagation

$$
\begin{aligned}
a^{(1)} &= x \\
z^{(2)} &= \Theta^{(1)} a^{(1)} \\
a^{(2)} &= g(z^{(2)}) \\
z^{(3)} &= \Theta^{(2)} a^{(2)} \\
a^{(3)} &= g(z^{(3)}) \\
z^{(4)} &= \Theta^{(3)} a^{(3)} \\
a^{(4)} &= g(z^{(4)}) = h(x^{(i)}; \theta)
\end{aligned}
$$



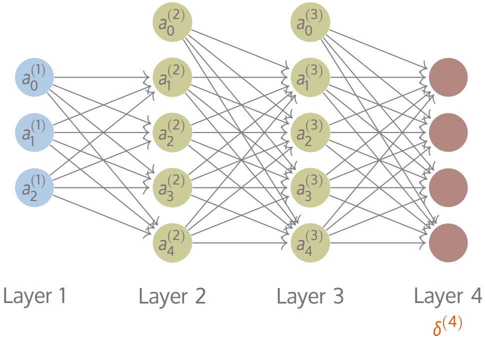
Forward propagation computes the activation values for all neurons in the network.

Gradient Computation

Given (x, y):

2. Back Propagation

Idea: Compute the '**error**' $\delta_j^{(l)}$ of node j in layer l



$$\delta_j^{(4)} = a_j^{(4)} - y_j$$

Gradient Computation

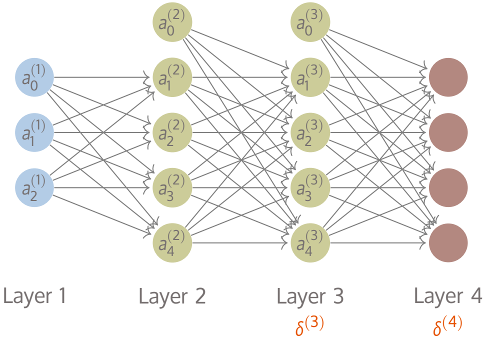
Given (x, y):

2. Back Propagation

Idea: Compute the '**error**' $\delta_j^{(l)}$ of node j in layer l

$$\delta_j^{(3)} = \text{compute errors for layer 3 nodes}$$

$$\delta_j^{(4)} = a_j^{(4)} - y_j$$



Gradient Computation

Given (x, y):

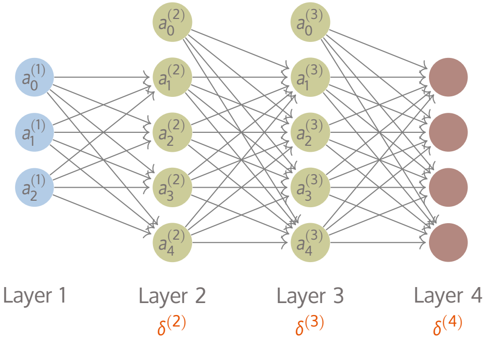
2. Back Propagation

Idea: Compute the '**error**' $\delta_j^{(l)}$ of node j in layer l

$\delta_j^{(2)}$ = compute errors for layer 2 nodes

$\delta_j^{(3)}$ = compute errors for layer 3 nodes

$$\delta_j^{(4)} = a_j^{(4)} - y_j$$



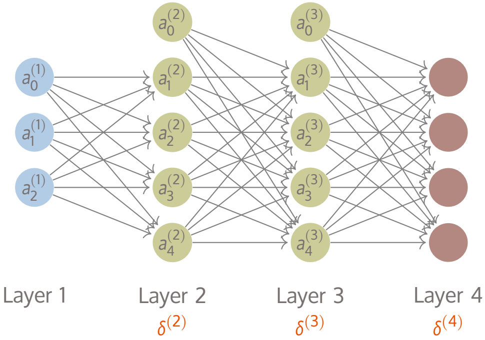
Gradient Computation

Given (x, y):

2. Back Propagation

Idea: Compute the '**error**' $\delta_j^{(l)}$ of node j in layer l

$$\begin{aligned} & \text{Input } a^{(1)} \text{ is without error} \\ \delta_j^{(2)} &= \text{compute errors for layer 2 nodes} \\ \delta_j^{(3)} &= \text{compute errors for layer 3 nodes} \\ \delta_j^{(4)} &= a_j^{(4)} - y_j \end{aligned}$$



Backprop computes the error associated with each activation in the network.

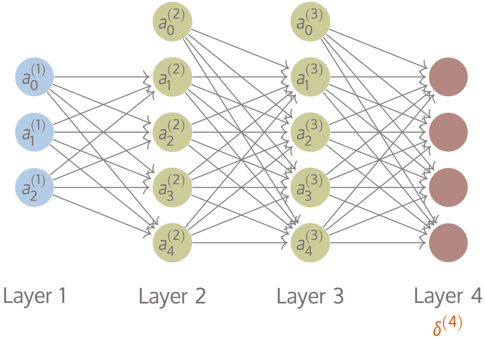
Gradient Computation

Given (x, y):

2. Back Propagation

Example: Compute the '**error**' $\delta_j^{(4)}$:

$$\delta_j^{(4)} = a_j^{(4)} - y_j$$



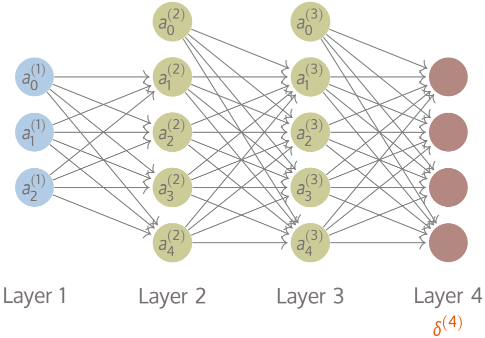
Gradient Computation

Given (x, y):

2. Back Propagation

Example: Compute the '**error**' $\delta_j^{(4)}$:

$$\begin{aligned}\delta_j^{(4)} &= a_j^{(4)} - y_j \\ &= h(x^{(i)}; \theta)_j - y_j \\ &= j^{th} \text{ prediction } - y_j\end{aligned}$$



Gradient Computation

Given (x, y):

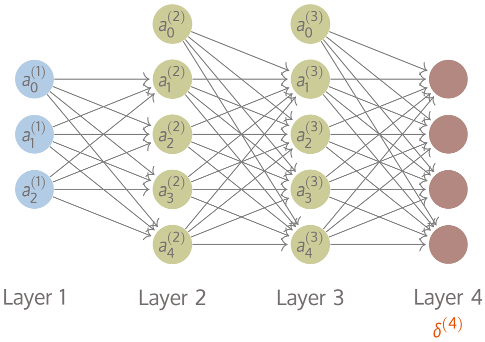
2. Back Propagation

Example: Compute the '**error**' $\delta_j^{(4)}$:

$$\begin{aligned} \delta_j^{(4)} &= a_j^{(4)} - y_j \\ &= h(x^{(i)}; \theta)_j - y_j \\ &= j^{th} \text{ prediction } - y_j \end{aligned}$$

Vectorization:

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



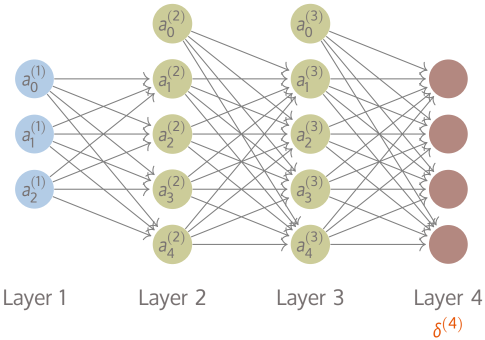
Gradient Computation

Given (x, y):

2. Back Propagation

Vectorization: Compute all '**error**' $\delta^{(l)}$:

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



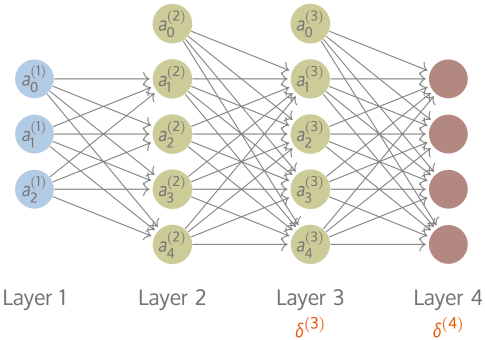
Gradient Computation

Given (x, y):

2. Back Propagation

Vectorization: Compute all '**error**' $\delta^{(l)}$:

$$\delta^{(4)} = a^{(4)} - y$$
$$\delta^{(3)} = (\Theta^{(3)})^T \delta^{(4)} .* [g'(z^{(3)})]$$



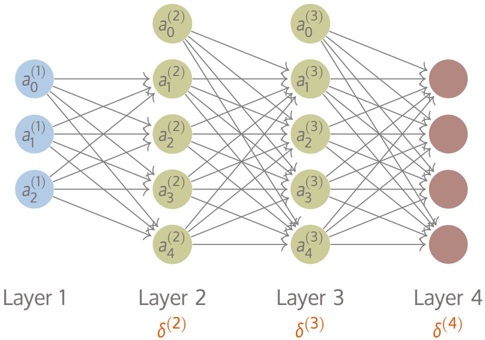
Gradient Computation

Given (x, y):

2. Back Propagation

Vectorization: Compute all '**error**' $\delta^{(l)}$:

$$\delta^{(4)} = a^{(4)} - y$$
$$\delta^{(3)} = (\Theta^{(3)})^T \delta^{(4)} \,.\!*\, [g'(z^{(3)})]$$
$$\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} \,.\!*\, [g'(z^{(2)})]$$



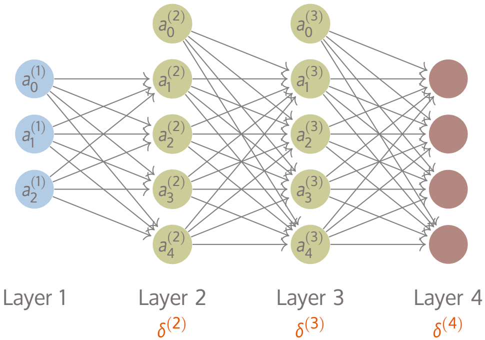
Gradient Computation

Given (x, y):

2. Back Propagation

Vectorization: Compute all '**error**' $\delta^{(l)}$:

$$\delta^{(4)} = a^{(4)} - y$$
$$\delta^{(3)} = (\Theta^{(3)})^T \delta^{(4)} .* [g'(z^{(3)})]$$
$$\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} .* [g'(z^{(2)})]$$



$g'(z)$ is the derivative of the activation function $g(\cdot)$ evaluated at the input values given by z.

$.*$ refers to dot-product multiplication.

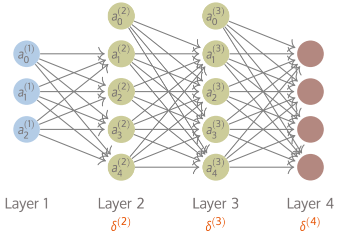
Gradient Computation

Given (x, y):

2. Back Propagation

Vectorization: Compute all '**error**' $\delta^{(l)}$:

$$\delta^{(4)} = a^{(4)} - y$$
$$\delta^{(3)} = (\Theta^{(3)})^T \delta^{(4)} \mathbin{.*} [a^{(3)} \mathbin{.*} (1 - a^{(3)})]$$
$$\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} \mathbin{.*} [a^{(2)} \mathbin{.*} (1 - a^{(2)})]$$



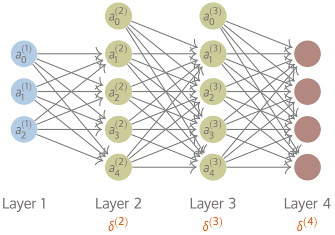
Gradient Computation

Given (x, y):

2. Back Propagation

Vectorization: Compute all '**error**' $\delta^{(l)}$:

$$\delta^{(4)} = a^{(4)} - y$$
$$\delta^{(3)} = (\Theta^{(3)})^T \delta^{(4)} \;.*\; [a^{(3)} \;.*\; (1 - a^{(3)})]$$
$$\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} \;.*\; [a^{(2)} \;.*\; (1 - a^{(2)})]$$



The proof is complicated, but it can be shown that

$$(\Theta^{(l)})^T \delta^{(l+1)} \;.*\; [a^{(l)} \;.*\; (1 - a^{(l)})]$$

is **equivalent** to simply calculating

$$a^{(l)} \delta^{(l+1)}$$

Gradient Computation

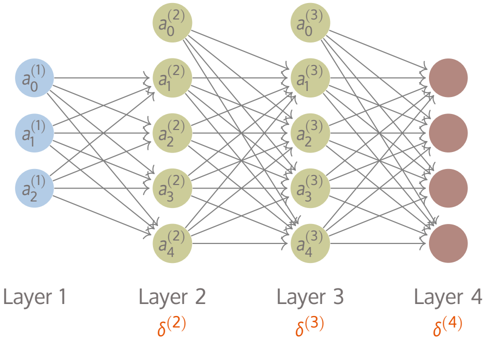
Given (x, y):

2. Back Propagation

Upshot:

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = a_j^{(l)} \delta_i^{(l+1)}$$

(if we ignore λ or simply set $\lambda = 0$)



So much for back propagation for one training example (x, y).

So much for back propagation for one training example (x, y).

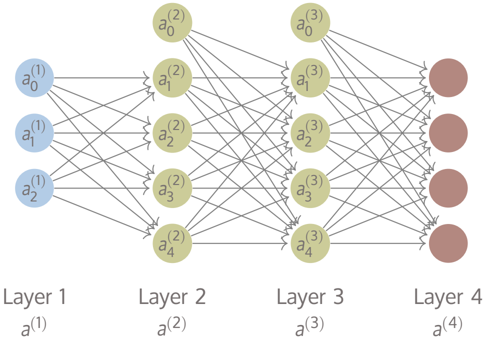
What about lots of training examples?

Gradient Computation

Given $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)}\}$:

1. Forward Propagation

$$
\begin{aligned}
a^{(1)} &= x \\
z^{(2)} &= \Theta^{(1)} a^{(1)} \\
a^{(2)} &= g(z^{(2)}) \\
z^{(3)} &= \Theta^{(2)} a^{(2)} \\
a^{(3)} &= g(z^{(3)}) \\
z^{(4)} &= \Theta^{(3)} a^{(3)} \\
a^{(4)} &= g(z^{(4)}) = h(x^{(i)}; \theta)
\end{aligned}
$$



Back Propagation Algorithm

Algorithm:
Backpropagation

» Given $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$
» Set $\Delta^{l}_{i,j} = 0$

For $i = 1 : m$
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 $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$
$\Delta^{(l)}_{i,j} := \Delta^{(l)}_{ij} + a^{(l)}_{j} \delta^{(l+1)}_{i}$
end

Back Propagation Algorithm

Algorithm:
Backpropagation

» Given $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$
» Set $\Delta^{l}_{i,j} = 0$

For $i = 1 : m$

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 $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$

$\Delta^{(l)}_{i,j} := \Delta^{(l)}_{ij} + a^{(l)}_{j} \delta^{(l+1)}_{i}$ # **Vectorized:** $\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} (a^{(l)})^T$

end

Back Propagation Algorithm

Algorithm:
Backpropagation

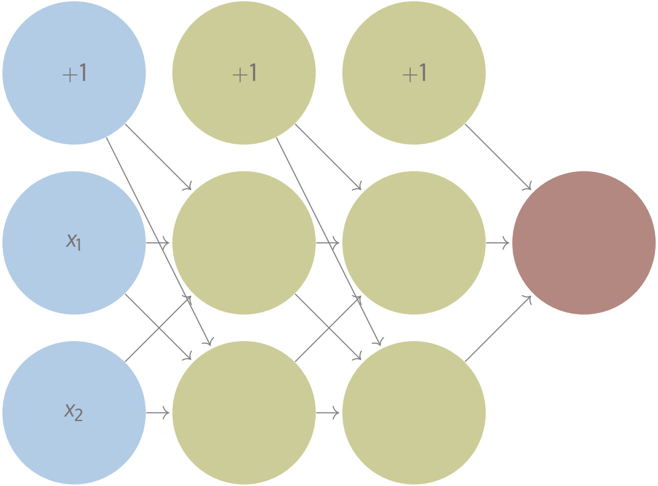
» Given $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)}\}$
» Set $\Delta^{l}_{i,j} = 0$
For $i = 1 : m$
Set $a^{(1)} = x^{(i)}$
Perform forward propagation to compute $a^{(l)}$ for $l = 2, 3, \ldots, L$
Using $y^{(i)}$, compute $\delta^{(L)} = a^{(L)} - y^{(i)}$
Compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$
$\Delta^{(l)}_{i,j} := \Delta^{(l)}_{ij} + a^{(l)}_{j}\delta^{(l+1)}_{i}$ # **Vectorized:** $\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)}(a^{(l)})^{T}$
end
» Compute partial derivatives $\frac{\partial}{\partial\Theta^{(l)}_{ij}} J(\Theta) = D^{(l)}_{ij}$, by

$$D^{(l)}_{ij} := \frac{1}{m}\Delta^{(l)}_{ij} + \lambda\Theta^{(l)}_{ij} \quad \text{if } j \neq 0$$

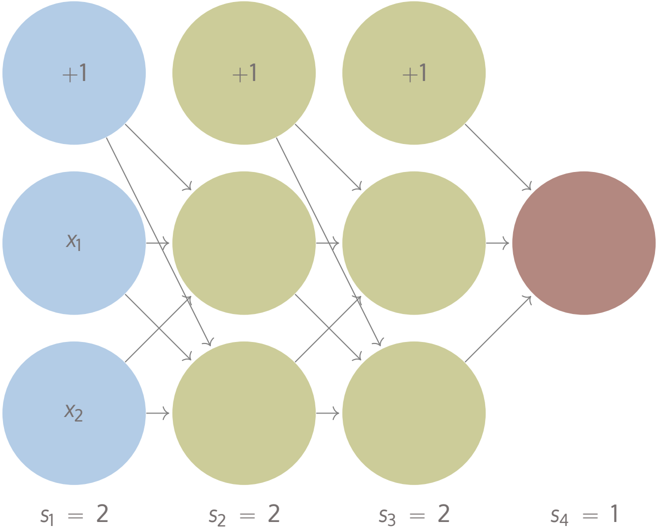
$$D^{(l)}_{ij} := \frac{1}{m}\Delta^{(l)}_{ij} \quad \text{if } j = 0$$

Backpropagation Intuition

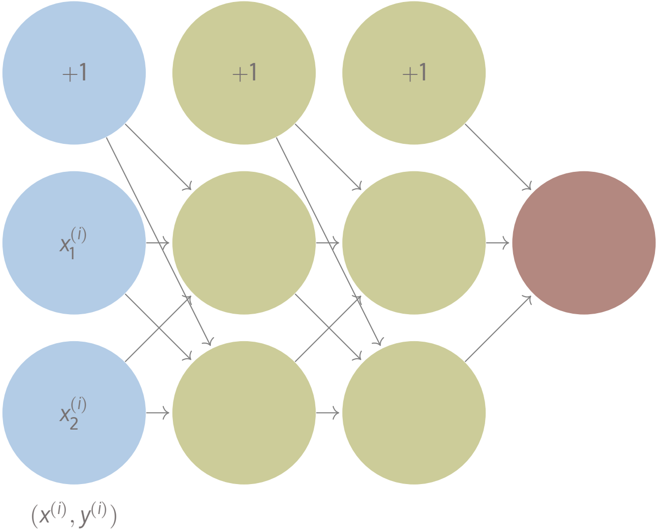
Forward Propagation



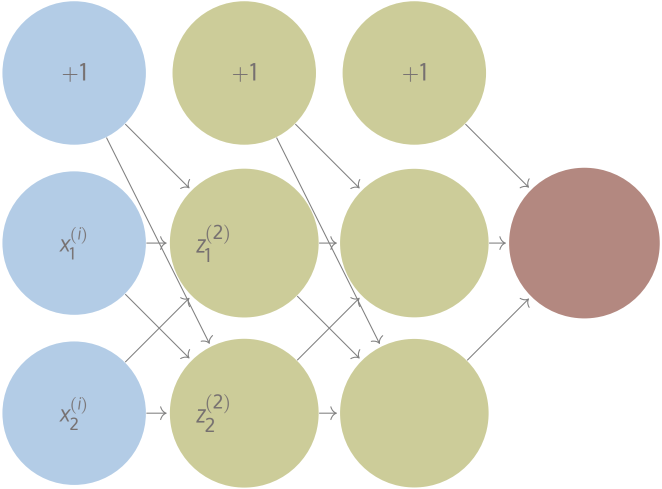
Forward Propagation



Forward Propagation

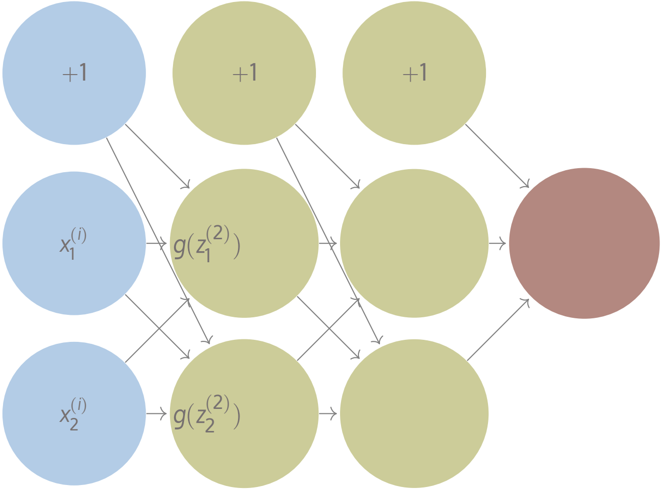


Forward Propagation



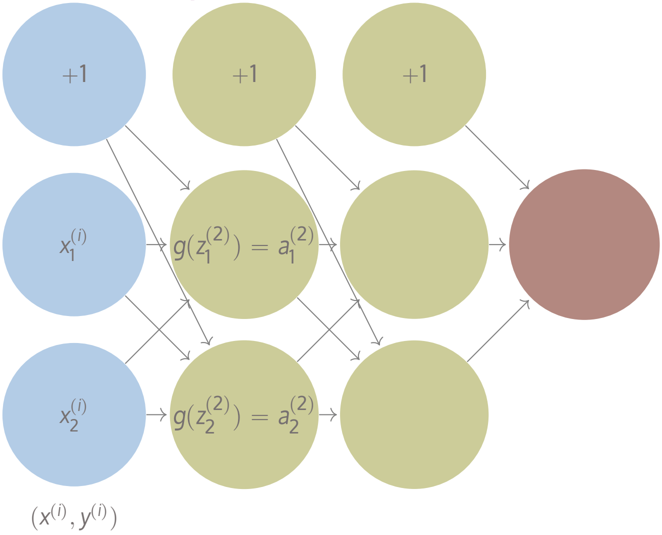
$(x^{(i)}, y^{(i)})$

Forward Propagation

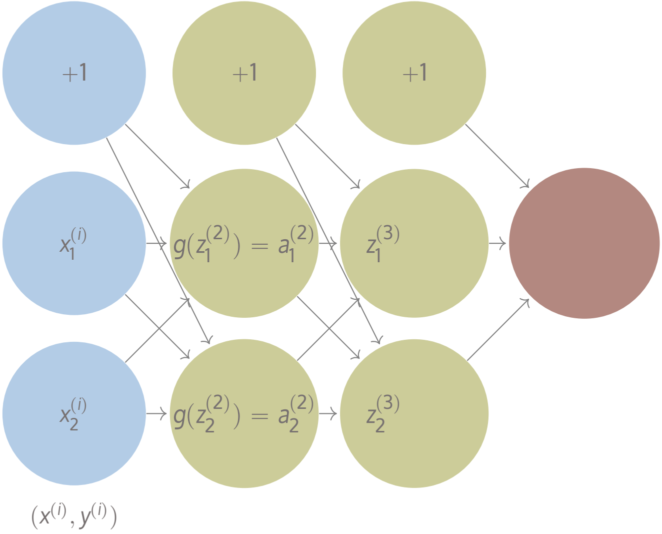


$(x^{(i)}, y^{(i)})$

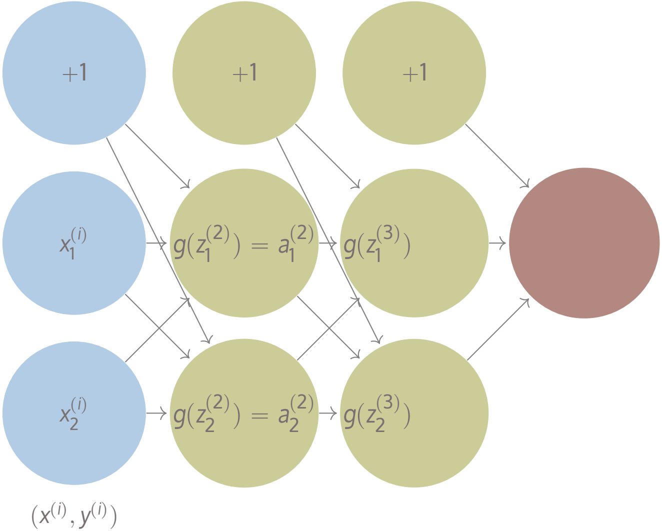
Forward Propagation



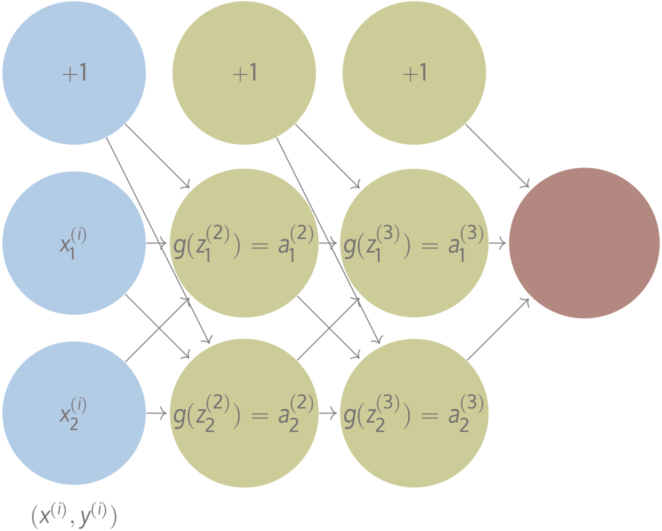
Forward Propagation



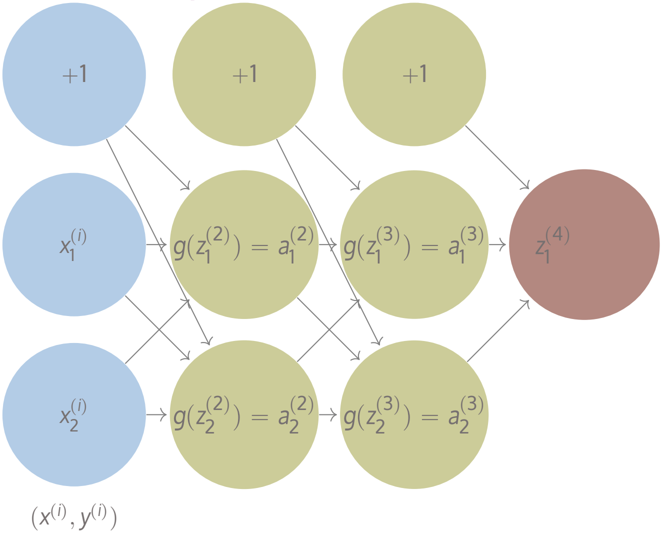
Forward Propagation



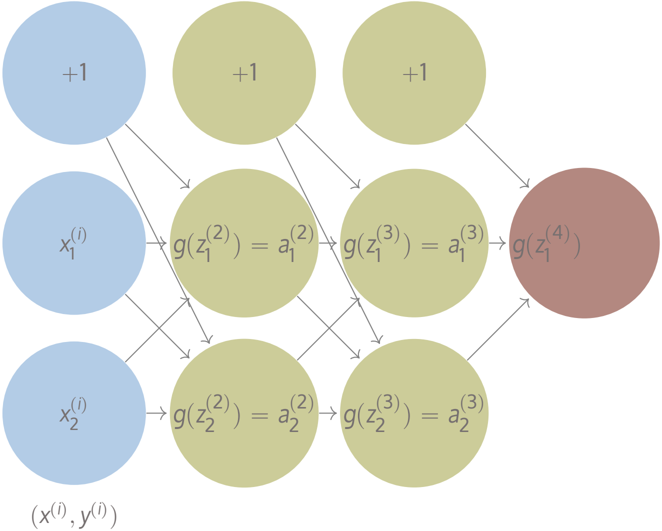
Forward Propagation



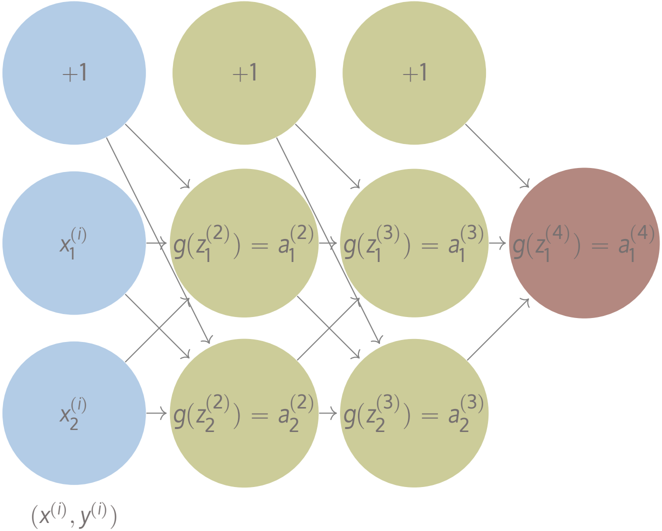
Forward Propagation



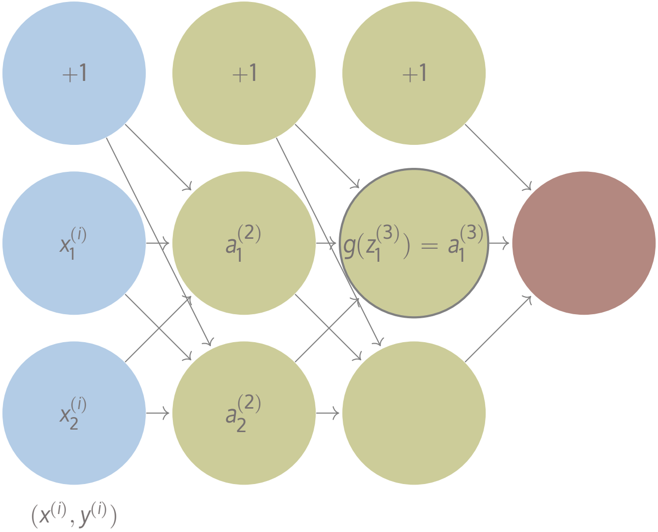
Forward Propagation



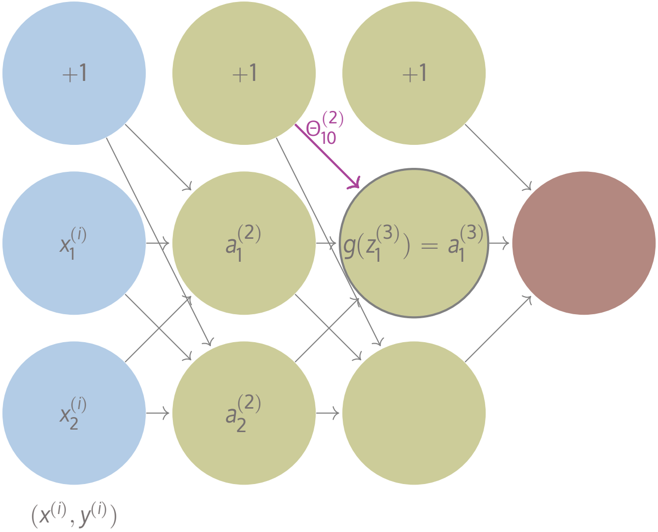
Forward Propagation



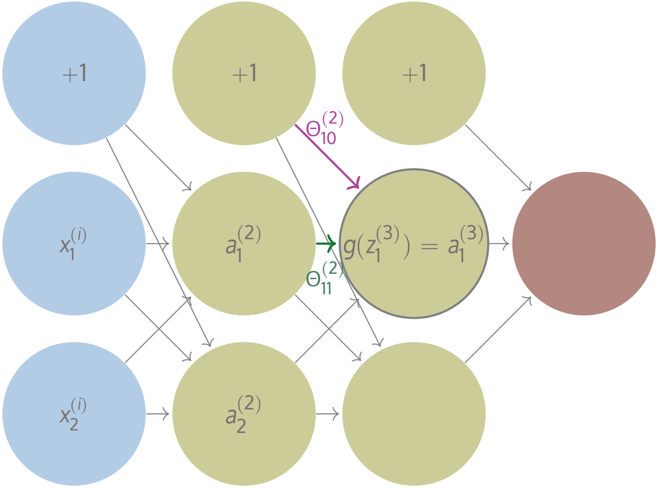
Forward Propagation



Forward Propagation

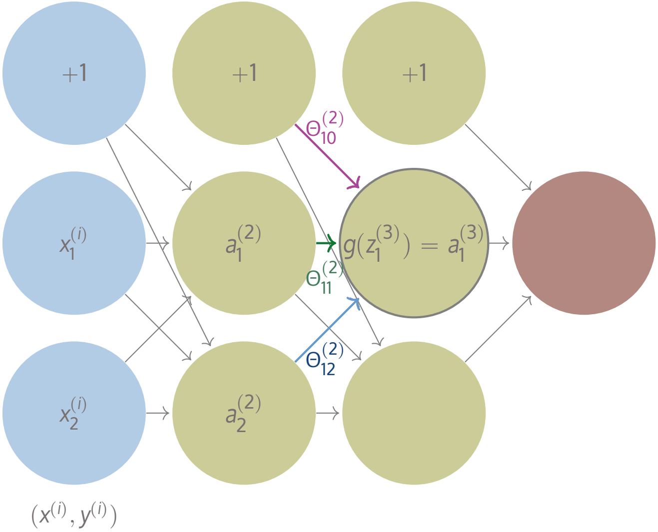


Forward Propagation

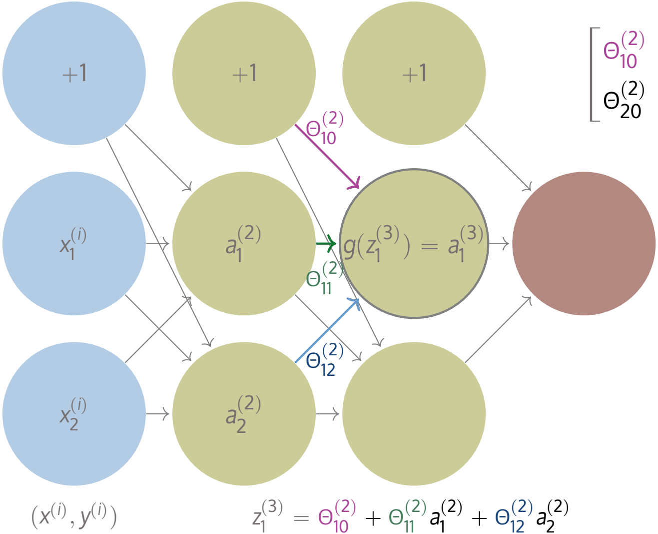


$(x^{(i)}, y^{(i)})$

Forward Propagation



Forward Propagation



$$\begin{bmatrix} \Theta_{10}^{(2)} & \Theta_{11}^{(2)} & \Theta_{12}^{(2)} \\ \Theta_{20}^{(2)} & \Theta_{21}^{(2)} & \Theta_{22}^{(2)} \end{bmatrix}$$

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

Forward propagation flows from *Left-to-Right*

Back propagation flows from *Right-to-Left*

What back propagation is doing

Neural Network Cost Function for $h(x^{(i)}; \Theta) \in \mathbb{R}^K$

$$J(\Theta) = -\frac{1}{m}\left(\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(h(x^{(i)}); \Theta)_k + (1 - y_k^{(i)}) \log\left(1 - h(x^{(i)}; \Theta)\right)_k\right) + \frac{\lambda}{2m}\left(\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}} \Theta_{ij}^{(l)}\right)^2$$

What back propagation is doing

Neural Network Cost Function for $h(x^{(i)}; \Theta) \in \mathbb{R}^K$

$$J(\Theta) = -\frac{1}{m}\left(\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(h(x^{(i)}); \Theta)_k + (1 - y_k^{(i)}) \log\left(1 - h(x^{(i)}; \Theta)\right)_k\right) + \frac{\lambda}{2m}\left(\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}} \Theta_{ij}^{(l)}\right)^2$$

...without regularization:

$$J(\Theta) = -\frac{1}{m}\left(\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(h(x^{(i)}); \Theta)_k + (1 - y_k^{(i)}) \log\left(1 - h(x^{(i)}; \Theta)\right)_k\right)$$

What back propagation is doing

Neural Network Cost Function for $h(x^{(i)}; \Theta) \in \mathbb{R}^K$

$$J(\Theta) = -\frac{1}{m}\left(\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(h(x^{(i)}); \Theta)_k + (1 - y_k^{(i)}) \log\left(1 - h(x^{(i)}; \Theta)\right)_k\right) + \frac{\lambda}{2m}\left(\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}} \Theta_{ij}^{(l)}\right)^2$$

...without regularization:

$$J(\Theta) = -\frac{1}{m}\left(\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(h(x^{(i)}); \Theta)_k + (1 - y_k^{(i)}) \log\left(1 - h(x^{(i)}; \Theta)\right)_k\right)$$

...for only one output unit ($K = 1$):

$$J(\Theta) = -\frac{1}{m}\left(\sum_{i=1}^{m} y^{(i)} \log(h(x^{(i)}); \Theta) + (1 - y^{(i)}) \log\left(1 - h(x^{(i)}; \Theta)\right)\right)$$

What back propagation is doing

Neural Network Cost Function for $h(x^{(i)}; \Theta) \in \mathbb{R}^K$

$$J(\Theta) = -\frac{1}{m}\left(\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(h(x^{(i)}); \Theta)_k + (1 - y_k^{(i)}) \log\left(1 - h(x^{(i)}; \Theta)\right)_k\right) + \frac{\lambda}{2m}\left(\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}} \Theta_{ij}^{(l)}\right)^2$$

...without regularization:

$$J(\Theta) = -\frac{1}{m}\left(\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(h(x^{(i)}); \Theta)_k + (1 - y_k^{(i)}) \log\left(1 - h(x^{(i)}; \Theta)\right)_k\right)$$

...for only one output unit ($K = 1$):

$$J(\Theta) = -\frac{1}{m}\left(\sum_{i=1}^{m} y^{(i)} \log(h(x^{(i)}); \Theta) + (1 - y^{(i)}) \log\left(1 - h(x^{(i)}; \Theta)\right)\right)$$

...for one particular training example: $(x^{(i)}, y^{(i)})$

What back propagation is doing

...for one particular training example: $(x^{(i)}, y^{(i)})$

$$J(\Theta) = -\frac{1}{m}\left(\sum_{i=1}^{m} y^{(i)} \log(h(x^{(i)}); \Theta) + (1 - y^{(i)}) \log\left(1 - h(x^{(i)}; \Theta)\right)\right)$$

What back propagation is doing

...for one particular training example: $(x^{(i)}, y^{(i)})$

$$J(\Theta) = -\frac{1}{m}\left(\sum_{i=1}^{m} y^{(i)} \log(h(x^{(i)}); \Theta) + (1 - y^{(i)}) \log\left(1 - h(x^{(i)}; \Theta)\right)\right)$$

...which calculates the cost associated with the ith training example:

$$\mathrm{cost}(i) = y^{(i)} \log(h(x^{(i)}); \Theta) + (1 - y^{(i)}) \log\left(1 - h(x^{(i)}; \Theta)\right)$$

What back propagation is doing

...for one particular training example: $(x^{(i)}, y^{(i)})$

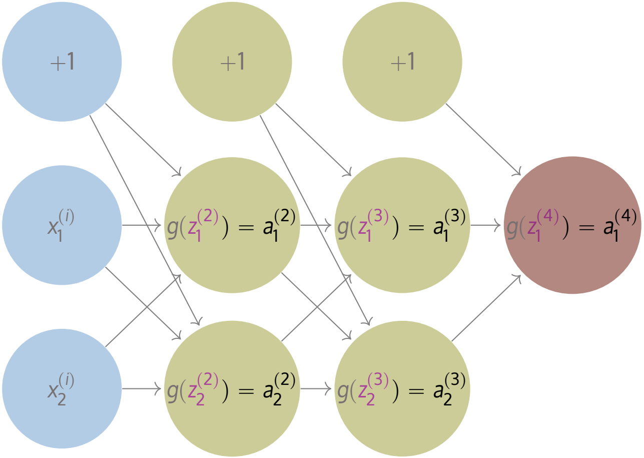
$$J(\Theta) = -\frac{1}{m}\left(\sum_{i=1}^{m} y^{(i)} \log(h(x^{(i)}); \Theta) + (1 - y^{(i)}) \log\left(1 - h(x^{(i)}; \Theta)\right)\right)$$

...which calculates the cost associated with the ith training example:

$$\text{cost}(i) = y^{(i)} \log(h(x^{(i)}); \Theta) + (1 - y^{(i)}) \log\left(1 - h(x^{(i)}; \Theta)\right)$$

Question: *How well is the network doing on example i?*

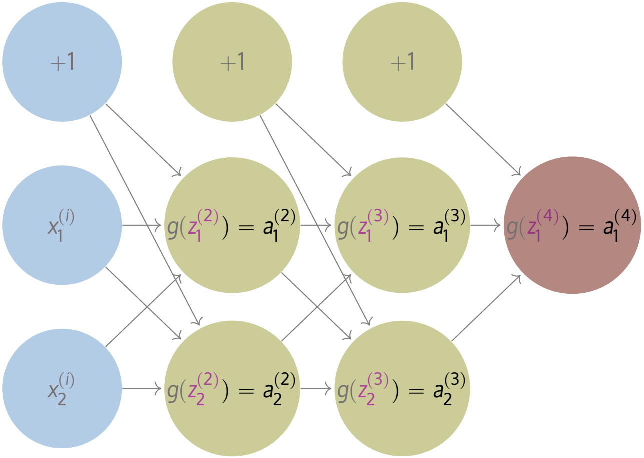
Backward Propagation



$\delta_j^{(l)}$ = informally, the error of **cost** for activation $a_j^{(l)}$ of node j in layer l.

formally, $\delta_j^{(l)}$ is the partial derivative with respect to input $z_j^{(l)}$ of **cost** of training example i:

Backward Propagation

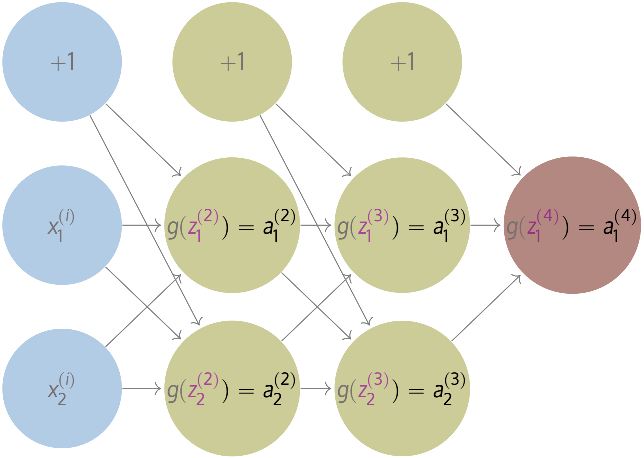


$\delta_j^{(l)}$ = informally, the error of **cost** for activation $a_j^{(l)}$ of node j in layer l.

formally, $\delta_j^{(l)}$ is the partial derivative with respect to input $z_j^{(l)}$ of **cost** of training example i:

$$\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \textbf{cost}(i), \quad \text{where } \textbf{cost}(i) = y^{(i)} \log h(x^{(i)};\theta) + (1 - y^{(i)}) \log(1 - h(x^{(i)};\theta))$$

Backward Propagation



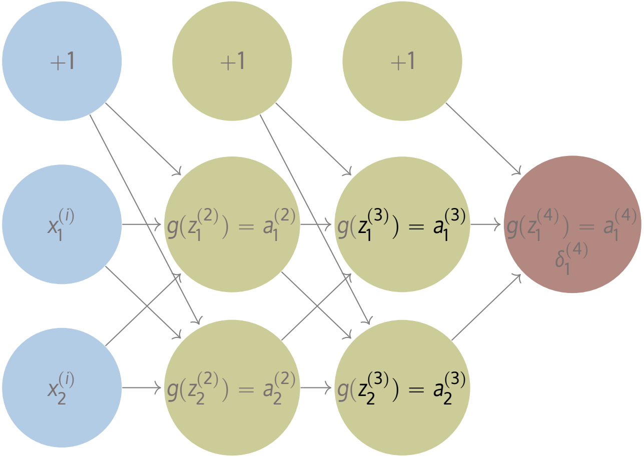
$\delta_j^{(l)}$ = informally, the error of **cost** for activation $a_j^{(l)}$ of node *j* in layer *l*.

formally, $\delta_j^{(l)}$ is the partial derivative with respect to input $z_j^{(l)}$ of **cost** of training example *i*:

$$\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \textbf{cost}(i), \quad \text{where } \textbf{cost}(i) = y^{(i)} \log h(x^{(i)}; \theta) + (1 - y^{(i)}) \log(1 - h(x^{(i)}; \theta))$$

Intuitively, if we go into the network and change these $z_j^{(l)}$ values, that will change the **cost**.

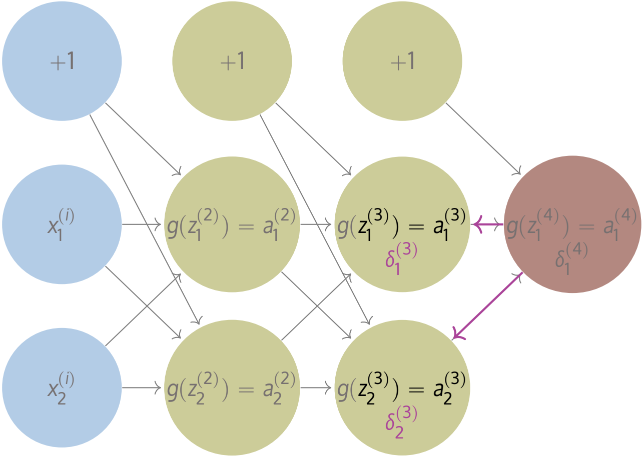
Backward Propagation



$\delta_j^{(l)}$ = the 'error' for $a_j^{(l)}$

$\delta_1^{(4)} = a_1^{(4)} - y^{(i)}$ # Difference between the actual value of $y^{(i)}$ and the predicted value of $y^{(i)}$

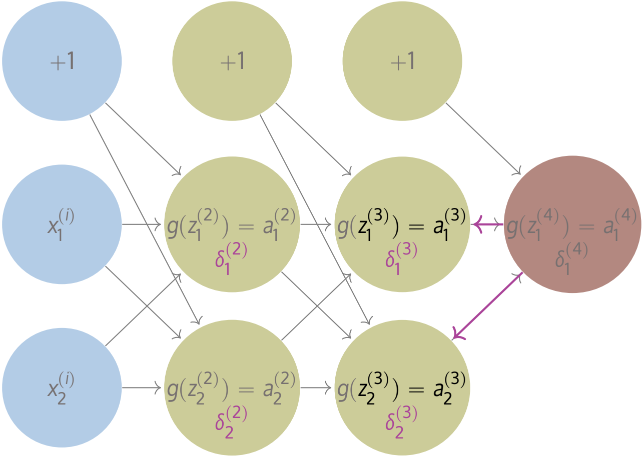
Backward Propagation



$\delta_j^{(l)}$ = the 'error' for $a_j^{(l)}$

$\delta_1^{(4)} = a_1^{(4)} - y^{(i)}$ # Difference between the actual value of $y^{(i)}$ and the predicted value of $y^{(i)}$

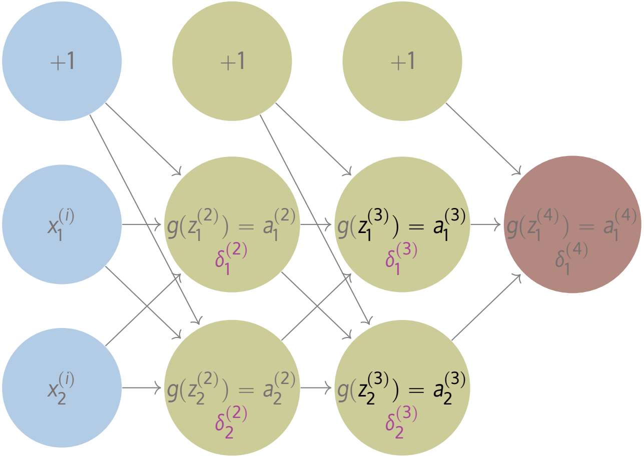
Backward Propagation



$\delta_j^{(l)} =$ the 'error' for $a_j^{(l)}$

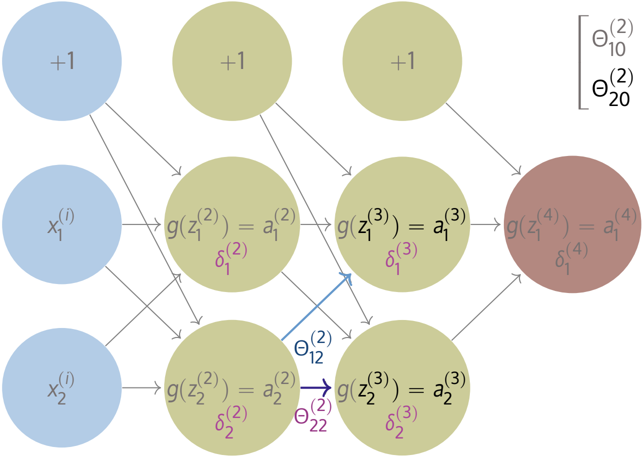
$\delta_1^{(4)} = a_1^{(4)} - y^{(i)}$ # Difference between the actual value of $y^{(i)}$ and the predicted value of $y^{(i)}$

Backward Propagation



Let's look at how to calculate $\delta_2^{(2)}$.

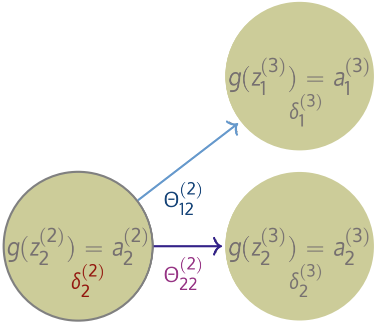
Backward Propagation



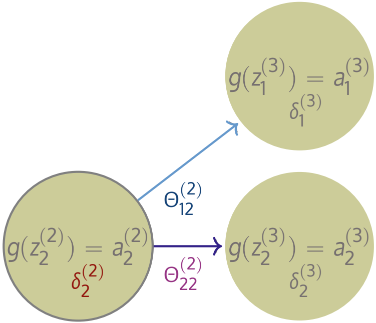
$$\begin{bmatrix} \Theta_{10}^{(2)} & \Theta_{11}^{(2)} & \Theta_{12}^{(2)} \\ \Theta_{20}^{(2)} & \Theta_{21}^{(2)} & \Theta_{22}^{(2)} \end{bmatrix}$$

Let's look at how to calculate $\delta_2^{(2)}$.

Back Propagation



Back Propagation



How to Compute $\delta_2^{(2)}$:

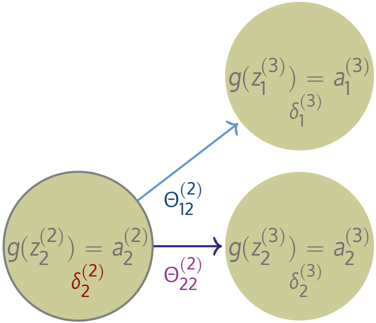
$$(\Theta^{(l)})^T \delta^{(l+1)} .* [a^{(l)} .* (1 - a^{(l)})]$$

$$\delta_2^{(2)} = \Theta_{12}^{(2)} \delta_1^{(3)} .* (1 - a_1^{(2)})] + \Theta_{22}^{(2)} \delta_2^{(3)} .* (1 - a_2^{(2)})]$$

$$\Theta^{(2)} = \begin{bmatrix} \Theta_{10}^{(2)} & \Theta_{11}^{(2)} & \Theta_{12}^{(2)} \\ \Theta_{20}^{(2)} & \Theta_{21}^{(2)} & \Theta_{22}^{(2)} \end{bmatrix}$$

where $\Theta^{(2)} \in \mathbb{R}^{2\times 3}$

Back Propagation



How to Compute $\delta_2^{(2)}$:

$$(\Theta^{(l)})^T \delta^{(l+1)} \;.*\; [a^{(l)} \;.*\; (1 - a^{(l)})]$$

$$\delta_2^{(2)} = \Theta_{12}^{(2)} \delta_1^{(3)} \;.*\; (1 - a_1^{(2)})] + \Theta_{22}^{(2)} \delta_2^{(3)} \;.*\; (1 - a_2^{(2)})]$$

$$\Theta^{(2)} = \begin{bmatrix} \Theta_{10}^{(2)} & \Theta_{11}^{(2)} & \Theta_{12}^{(2)} \\ \Theta_{20}^{(2)} & \Theta_{21}^{(2)} & \Theta_{22}^{(2)} \end{bmatrix}$$

where $\Theta^{(2)} \in \mathbb{R}^{2\times 3}$

Informally, $\delta_2^{(2)}$ is the weighted sum of the errors $\delta_1^{(3)}$ and $\delta_2^{(3)}$, where the weights $\Theta_{12}^{(2)}$ and $\Theta_{22}^{(2)}$ are the corresponding edge strengths.

Some Implementation Details*

Implementation Details

1. Unrolling (or unstacking) Matrices
2. Random Initialization of Parameter Matrices

Unroll and Reshape

Parameter Matrices Dimensions

$\Theta^{(1)} \in \mathbb{R}^{10\times 11}$ $\quad \Theta^{(2)} \in \mathbb{R}^{10\times 11}$ $\quad \Theta^{(3)} \in \mathbb{R}^{1\times 11}$

Derivative Matrices Dimensions

$D^{(1)} \in \mathbb{R}^{10\times 11}$ $\quad D^{(2)} \in \mathbb{R}^{10\times 11}$ $\quad D^{(3)} \in \mathbb{R}^{1\times 11}$

Unroll and Reshape

Parameter Matrices Dimensions

$\Theta^{(1)} \in \mathbb{R}^{10\times 11}$ $\quad \Theta^{(2)} \in \mathbb{R}^{10\times 11}$ $\quad \Theta^{(3)} \in \mathbb{R}^{1\times 11}$

Derivative Matrices Dimensions

$D^{(1)} \in \mathbb{R}^{10\times 11}$ $\quad D^{(2)} \in \mathbb{R}^{10\times 11}$ $\quad D^{(3)} \in \mathbb{R}^{1\times 11}$

Python

```
# Format training set
X_train = X_train.reshape(60000, 784)
X_train = X_train.astype('float32')
X_train = X_train/255

# Format test set
X_test = X_test.reshape(10000, 784)
X_test = X_test.astype('float32')
X_test = X_test/255

print("Training matrix shape", X_train.shape)
print("Testing matrix shape", X_test.shape)
```

Each of the 60,000 training examples in the MNST dataset is a 28x28 matrix.
`X_*.reshape(60000, 784)` takes 60,000 examples of `X_*` and reshapes into a 784-dimension vector.

Example

Network Dimensions

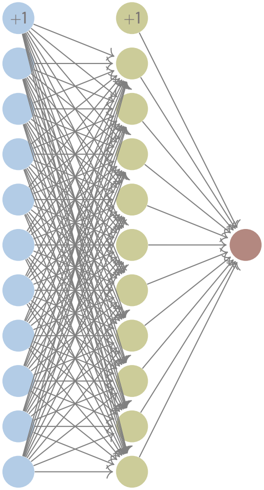
$$s_1 = s_2 = 10; \;\; s_3 = 1$$

Parameter Matrices Dimensions

$$\Theta^{(1)} \in \mathbb{R}^{10\times 11}$$
$$\Theta^{(2)} \in \mathbb{R}^{10\times 11}$$
$$\Theta^{(3)} \in \mathbb{R}^{1\times 11}$$

Derivative Matrices Dimensions

$$D^{(1)} \in \mathbb{R}^{10\times 11}$$
$$D^{(2)} \in \mathbb{R}^{10\times 11}$$
$$D^{(3)} \in \mathbb{R}^{1\times 11}$$



Random Initialization

Initializing Theta

Initial value of Θ: :

To implement either gradient descent or advanced optimization methods, such as `bfgs`, you need to specify initial values for Θ.

Initializing Theta

Initial value of Θ: :

To implement either gradient descent or advanced optimization methods, such as `bfgs`, you need to specify initial values for Θ.

Recall that for logistic regression, we simply initialized theta to a vector of zeros.

Why? Because $g(z) = 1$ if $z \geqslant 0$ and $g(z) = 0$ if $z < 0$

Initializing Theta

Initial value of Θ: :

To implement either gradient descent or advanced optimization methods, such as `bfgs`, you need to specify initial values for Θ.

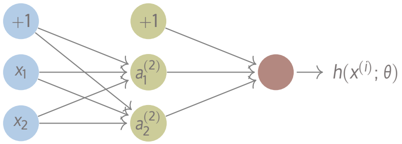
Recall that for logistic regression, we simply initialized theta to a vector of zeros.

Why? Because $g(z) = 1$ if $z \geqslant 0$ and $g(z) = 0$ if $z < 0$

`initialTheta = zeros(n,1)` does **not** work for neural networks.

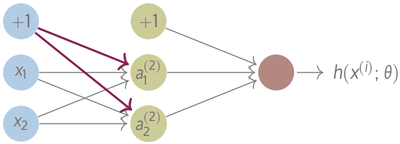
Why not?

Zero initialization of **Θ**



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

Zero initialization of Θ

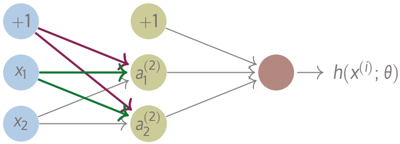


Suppose $\Theta_{ij}^{(l)} = 0$, for all i, j, l..

So:

$$\Theta_{10}^{(1)} = \Theta_{20}^{(1)}$$

Zero initialization of Θ



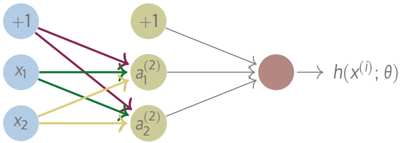
Suppose $\Theta_{ij}^{(l)} = 0$, for all i, j, l..

So:

$$\Theta_{10}^{(1)} = \Theta_{20}^{(1)}$$

$$\Theta_{11}^{(1)} = \Theta_{21}^{(1)}$$

Zero initialization of Θ



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

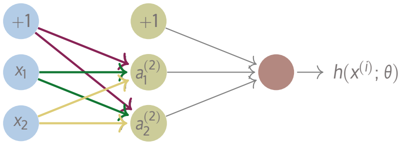
So:

$$\Theta_{10}^{(1)} = \Theta_{20}^{(1)}$$

$$\Theta_{11}^{(1)} = \Theta_{21}^{(1)}$$

$$\Theta_{12}^{(1)} = \Theta_{22}^{(1)}$$

Zero initialization of Θ



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

So:

$$\Theta_{10}^{(1)} = \Theta_{20}^{(1)}$$

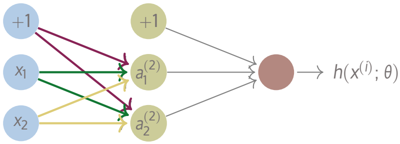
$$\Theta_{11}^{(1)} = \Theta_{21}^{(1)}$$

$$\Theta_{12}^{(1)} = \Theta_{22}^{(1)}$$

Hence,

$$a_1^{(2)} = a_2^{(2)}$$

Zero initialization of Θ



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

So:

$$\Theta_{10}^{(1)} = \Theta_{20}^{(1)}$$

$$\Theta_{11}^{(1)} = \Theta_{21}^{(1)}$$

$$\Theta_{12}^{(1)} = \Theta_{22}^{(1)}$$

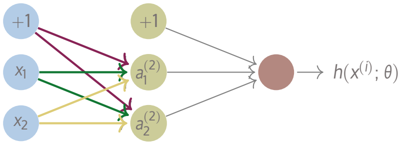
Hence,

$$a_1^{(2)} = a_2^{(2)}$$

and

$$\delta_1^{(2)} = \delta_2^{(2)}$$

Zero initialization of Θ



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

So:

$$\Theta_{10}^{(1)} = \Theta_{20}^{(1)}$$

$$\Theta_{11}^{(1)} = \Theta_{21}^{(1)}$$

$$\Theta_{12}^{(1)} = \Theta_{22}^{(1)}$$

Hence,

$$a_1^{(2)} = a_2^{(2)}$$

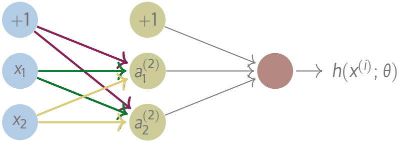
and

$$\delta_1^{(2)} = \delta_2^{(2)}$$

Furthermore,

$$\frac{\partial}{\partial \Theta_{10}^{(1)}} J(\Theta) = \frac{\partial}{\partial \Theta_{20}^{(1)}} J(\Theta), \text{ so } \Theta_{10}^{(1)} = \Theta_{20}^{(1)}$$

Zero initialization of Θ



Upshot: *Since* $\delta_1^{(l)} = \delta_2^{(l)} = 0$, *after each update, the parameters associated with each input that go into each of the two hidden units are identical.*

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

So:

$$\Theta_{10}^{(1)} = \Theta_{20}^{(1)}$$

$$\Theta_{11}^{(1)} = \Theta_{21}^{(1)}$$

$$\Theta_{12}^{(1)} = \Theta_{22}^{(1)}$$

Hence,

$$a_1^{(2)} = a_2^{(2)}$$

and

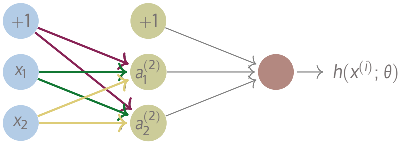
$$\delta_1^{(2)} = \delta_2^{(2)}$$

Furthermore,

$$\frac{\partial}{\partial \Theta_{10}^{(1)}} J(\Theta) = \frac{\partial}{\partial \Theta_{20}^{(1)}} J(\Theta), \text{ so } \Theta_{10}^{(1)} = \Theta_{20}^{(1)}$$

$$\frac{\partial}{\partial \Theta_{11}^{(1)}} J(\Theta) = \frac{\partial}{\partial \Theta_{21}^{(1)}} J(\Theta), \text{ so } \Theta_{11}^{(1)} = \Theta_{21}^{(1)}$$

Zero initialization of Θ



Upshot: *Since* $\delta_1^{(l)} = \delta_2^{(l)} = 0$*, after each update, the parameters associated with each input that go into each of the two hidden units are identical.*

So, the two hidden units are still computing the same function as the input:

$$a_1^{(2)} = a_2^{(2)}$$

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

So:

$$\Theta_{10}^{(1)} = \Theta_{20}^{(1)}$$

$$\Theta_{11}^{(1)} = \Theta_{21}^{(1)}$$

$$\Theta_{12}^{(1)} = \Theta_{22}^{(1)}$$

Hence,

$$a_1^{(2)} = a_2^{(2)}$$

and

$$\delta_1^{(2)} = \delta_2^{(2)}$$

Furthermore,

$$\frac{\partial}{\partial \Theta_{10}^{(1)}} J(\Theta) = \frac{\partial}{\partial \Theta_{20}^{(1)}} J(\Theta), \text{ so } \Theta_{10}^{(1)} = \Theta_{20}^{(1)}$$

$$\frac{\partial}{\partial \Theta_{11}^{(1)}} J(\Theta) = \frac{\partial}{\partial \Theta_{21}^{(1)}} J(\Theta), \text{ so } \Theta_{11}^{(1)} = \Theta_{21}^{(1)}$$

$$\frac{\partial}{\partial \Theta_{12}^{(1)}} J(\Theta) = \frac{\partial}{\partial \Theta_{22}^{(1)}} J(\Theta), \text{ so } \Theta_{12}^{(1)} = \Theta_{22}^{(1)}$$

Random Initialization of Θ

To get around the previous problem with **zero-initialization**, **random initialization** of Θ is a technique that assigns each value $\Theta_{ij}^{(l)}$ a random scalar value in some range, $[-\varepsilon, \varepsilon]$.[1]

[1]Note that this use of the variable ε is entirely different than its use in numerical gradient checking.

Random Initialization of Θ

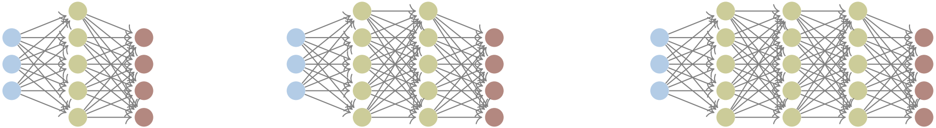
To get around the **problem of symmetric weights**, initialize each $\Theta_{ij}^{(l)}$ to a random small values in $[-\varepsilon, \varepsilon]$ close to zero, that is:

$$-\varepsilon \leqslant \Theta_{ij}^{(l)} \leqslant \varepsilon$$

Neural Networks: The Big Picture

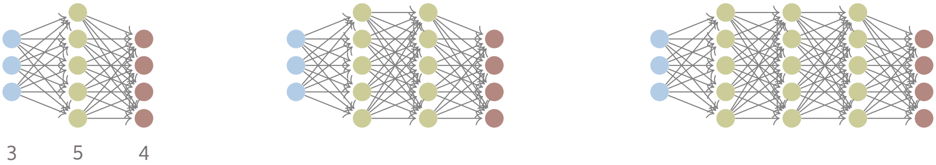
Training a Neural Network

Pick a network architecture (i.e., the connectivity pattern between neurons)



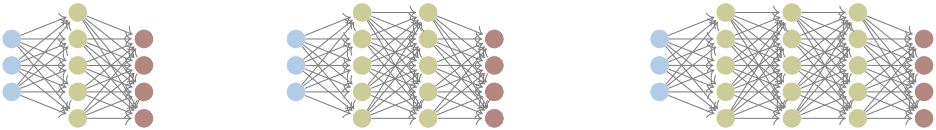
Training a Neural Network

Pick a network architecture (i.e., the connectivity pattern between neurons)



Training a Neural Network

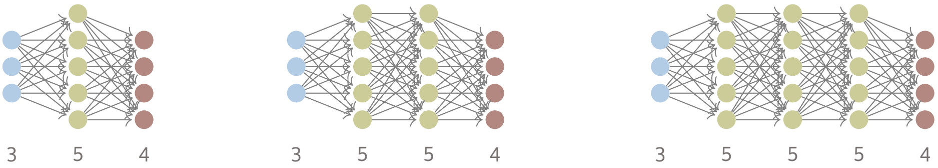
Pick a network architecture (i.e., the connectivity pattern between neurons)



3 5 4 | 3 5 5 4

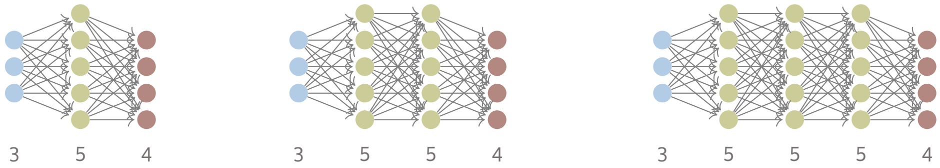
Training a Neural Network

Pick a network architecture (i.e., the connectivity pattern between neurons)



Training a Neural Network

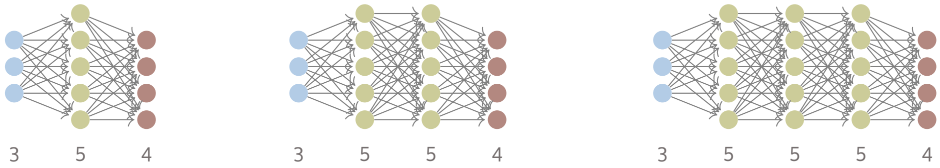
Pick a network architecture (i.e., the connectivity pattern between neurons)



Number of input units:

Training a Neural Network

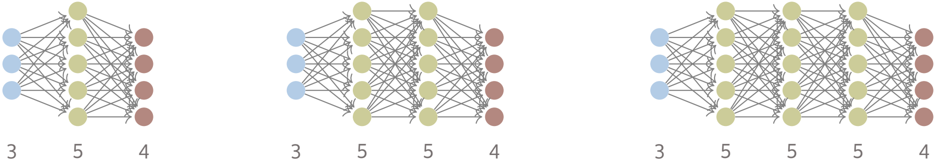
Pick a network architecture (i.e., the connectivity pattern between neurons)



Number of input units: *dimension of features:* $x^{(i)}$

Training a Neural Network

Pick a network architecture (i.e., the connectivity pattern between neurons)

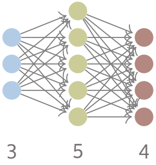


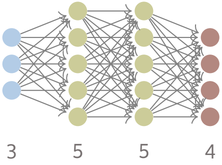
Number of input units: *dimension of features:* $x^{(i)}$

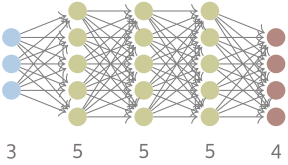
Number of output units:

Training a Neural Network

Pick a network architecture (i.e., the connectivity pattern between neurons)



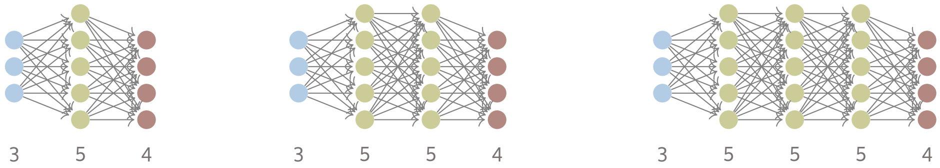




Number of input units:	*dimension of features:*	$x^{(i)}$
Number of output units:	*number of classes:*	K is a $\mathbb{R}^K$ vector

Training a Neural Network

Pick a network architecture (i.e., the connectivity pattern between neurons)



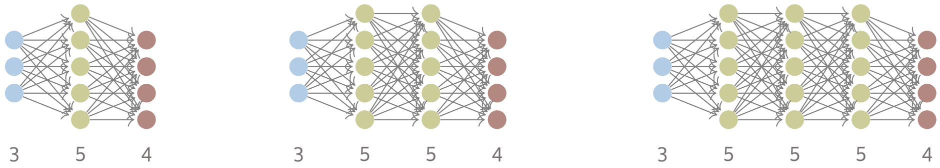
Number of input units: *dimension of features:* $x^{(i)}$

Number of output units: *number of classes:* K is a $\mathbb{R}^K$ vector

Num. of hidden layers:

Training a Neural Network

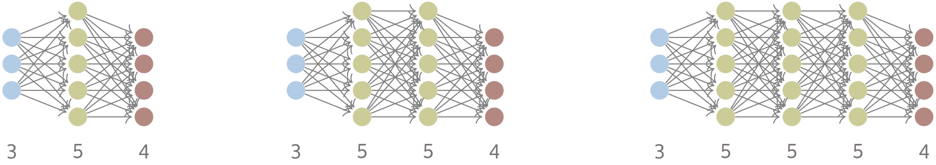
Pick a network architecture (i.e., the connectivity pattern between neurons)



Number of input units:	*dimension of features:*	$x^{(i)}$
Number of output units:	*number of classes:*	K is a $\mathbb{R}^K$ vector
Num. of hidden layers:	*default to try:*	1

Training a Neural Network

Pick a network architecture (i.e., the connectivity pattern between neurons)



Number of input units: *dimension of features:* $x^{(i)}$

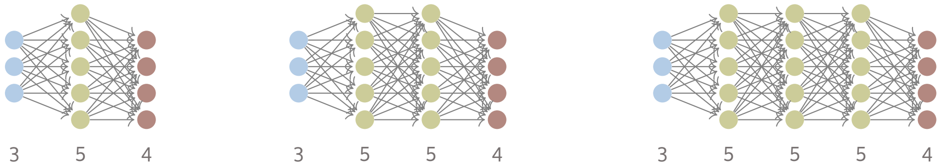
Number of output units: *number of classes:* K is a $\mathbb{R}^K$ vector

Num. of hidden layers: *default to try:* 1

Num. hidden units: *if $>$ 1 hidden layers* *all layers same Number*

Training a Neural Network

Pick a network architecture (i.e., the connectivity pattern between neurons)



Number of input units:	*dimension of features:*	$x^{(i)}$
Number of output units:	*number of classes:*	K is a $\mathbb{R}^K$ vector
Num. of hidden layers:	*default to try:*	1
Num. hidden units:	*if $>$ 1 hidden layers*	*all layers same Number*
	more hidden units better	*but more units, more computation*

Training a Neural Network

0. Pick a network architecture
1. Randomly initialize weights
2. Implement forward propagation to compute $h(x^{(i)}; \theta)$ for any $x^{(i)}$.
3. Implement code to compute the cost function $J(\Theta)$
4. Implement back propagation to compute partial derivatives $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$

Training a Neural Network

0. Pick a network architecture
1. Randomly initialize weights
2. Implement forward propagation to compute $h(x^{(i)}; \theta)$ for any $x^{(i)}$.
3. Implement code to compute the cost function $J(\Theta)$
4. Implement back propagation to compute partial derivatives $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$

```
for i = 1:m {
```

for each training example $(x^{(i)}, y^{(i)})$:

execute forward prop to get activations $a^{(l)}$

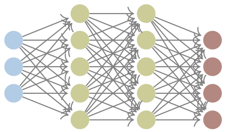
and back prop to get delta terms $\delta^{(l)}$

(for $l = 2, 3, \ldots, L$)

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

```
}
```

compute $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$



Training a Neural Network

0. Pick a network architecture
1. Randomly initialize weights
2. Implement forward propagation to compute $h(x^{(i)}; \theta)$ for any $x^{(i)}$.
3. Implement code to compute the cost function $J(\Theta)$
4. Implement back propagation to compute partial derivatives $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$

for i = 1:m {

for each training example $(x^{(i)}, y^{(i)})$:

execute forward prop to get activations $a^{(l)}$

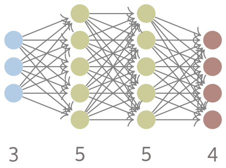
and back prop to get delta terms $\delta^{(l)}$

(for $l = 2, 3, \ldots, L$)

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

}

compute $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$



Training a Neural Network

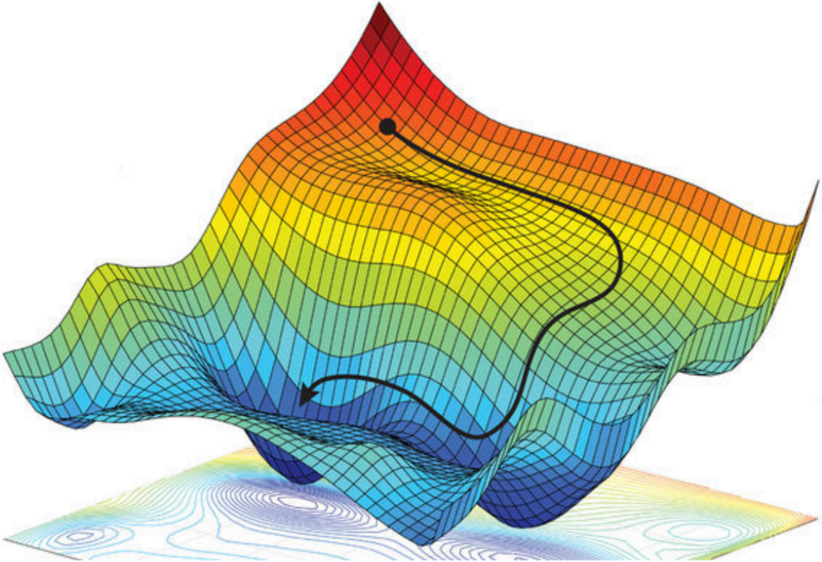
5. Use gradient checking to compare $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$ computed by back propagation and by numerical estimation of the gradient of $J(\Theta)$.

*Then **disable** gradient checking.*

6. Use gradient descent or an advanced optimization method with back propagation to attempt to minimize the cost function $J(\Theta)$ as a function of parameters Θ.

Cost Function

The cost function $J(\Theta)$ is **non-convex**.



Cost Function

The cost function $J(\Theta)$ is **non-convex**.

Question: *Why does gradient-descent work at all in neural networks despite non-convexity?*

Cost Function

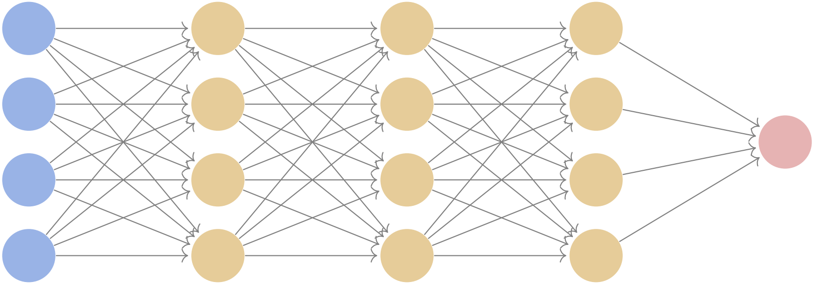
The cost function $J(\Theta)$ is **non-convex**.

Question: *Why does gradient-descent work at all in neural networks despite non-convexity?*

A partial answer is **over-provisioning**: since there are usually many hidden units, there are many different ways that a neural network can approximate the desired input-output relationship and you only need to find one (Carmon and Duchi 2016).

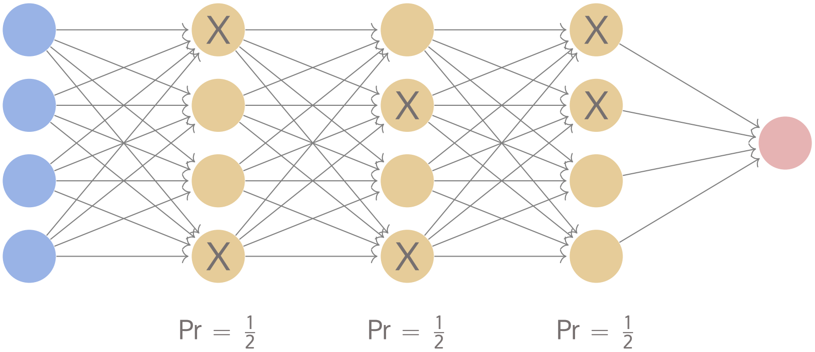
Coda: Dropout Regularization

Dropout Intuition



$\Pr = \frac{1}{2}$ $\Pr = \frac{1}{2}$ $\Pr = \frac{1}{2}$

Dropout Intuition



Dropout

Dropout temporarily removes nodes from a network.

Actually, what is temporarily "removed" are the links going in and out of randomly selected nodes.

Dropout simulates **sparse activation** from a given layer, which in effect reduces the model capacity of the network.

Notes:

- Dropout is used to address overfitting.
- The probability is the probability of a unit in a given layer dropping out.
- Nodes are drawn randomly on each pass.
- So, in practice, different nodes will be dropped in different passes.
- Dropout not used at test time. Otherwise, the predictions would be random. (Keras does this automatically).
- Dropout effectively spreads out weights; ensures that the network does not rely on any single feature.
- By spreading out weights, this effectively reduces the squared norm, $\|\cdot\|_F^2$.
- Probability for keeping units can be varied by layer.
- Downside: the cost function $J(\cdot)$ is no longer well-defined.
- To plot $J(\cdot)$, to check that it is monotonically decreasing, turn off dropout.

References

Carmon, Y. and J. Duchi (2016).
Gradient descent efficiently finds the cubic-regularized non-convex Newton step.
In *Workshop on Nonconvex Optimization for Machine Learning (NIPS 2016)*.

Nielsen, M. (2019).
Neural Networks and Deep Learning.
Determination Press.