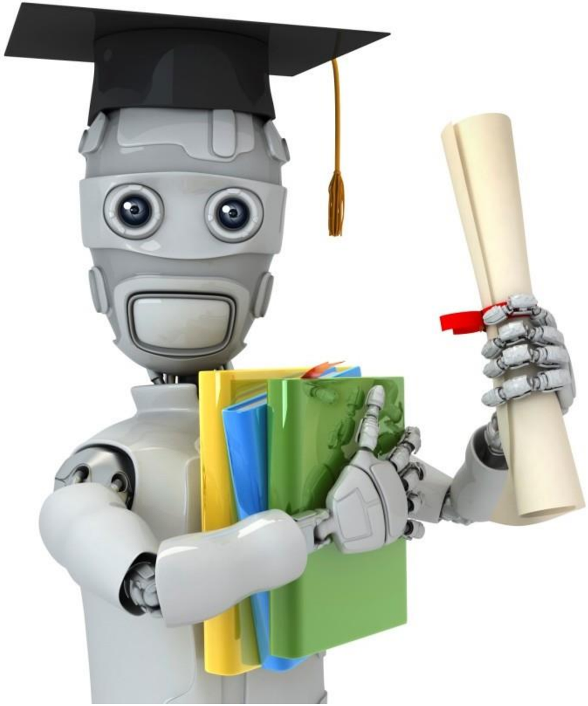
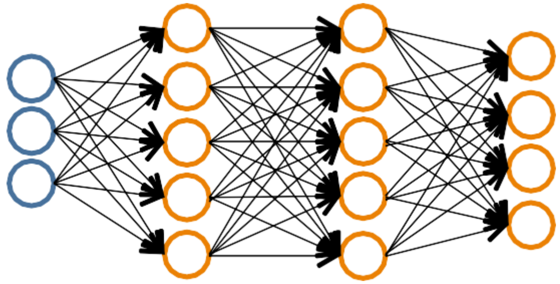


Machine Learning

Neural Networks: Learning

Cost function

Neural Network (Classification)



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

$L =$ total no. of layers in network

$s_l =$ no. of units (not counting bias unit) in layer l

<u>Binary classification</u>

$y = 0 \text{ or } 1$

1 output unit

<u>Multi-class classification</u> (K classes)

$y \in \mathbb{R}^K$ E.g. $\begin{bmatrix}1\\0\\0\\0\end{bmatrix}$, $\begin{bmatrix}0\\1\\0\\0\end{bmatrix}$, $\begin{bmatrix}0\\0\\1\\0\end{bmatrix}$, $\begin{bmatrix}0\\0\\0\\1\end{bmatrix}$

pedestrian car motorcycle truck

K output units

Cost function

Logistic regression:

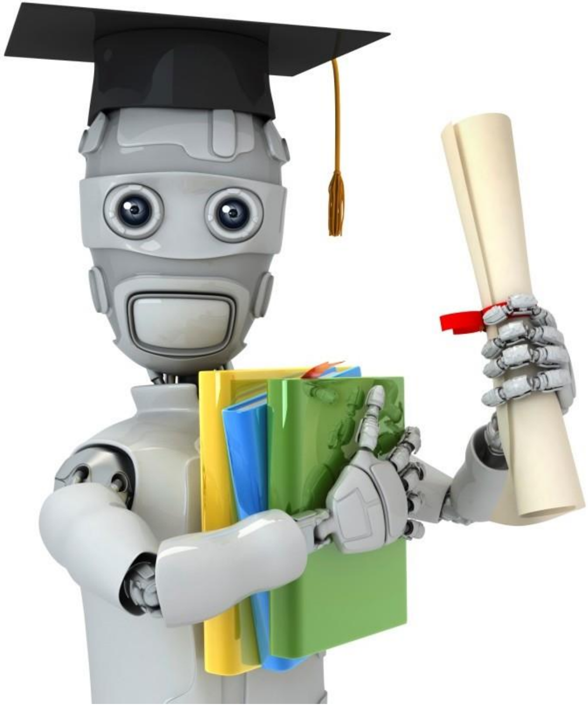
$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

Neural network:

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

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

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



Machine Learning

Neural Networks: Learning

Backpropagation algorithm

Gradient computation

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

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

Need code to compute:

B $J(\Theta)$

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

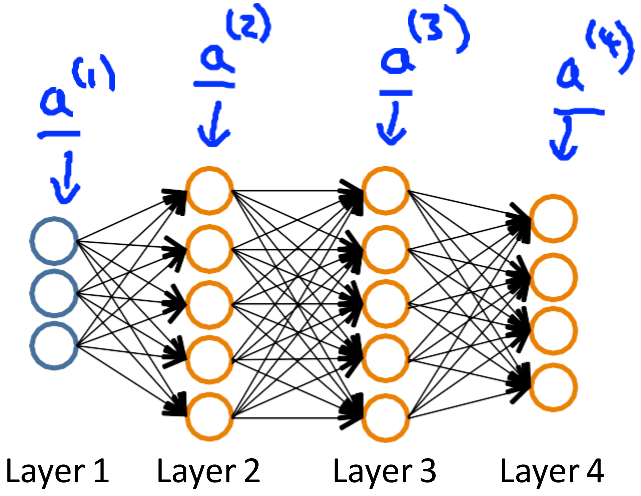
$\Theta_{ij}^{(l)} \in \mathbb{R}$

Gradient computation

Given one training example (x, y):

Forward propagation:

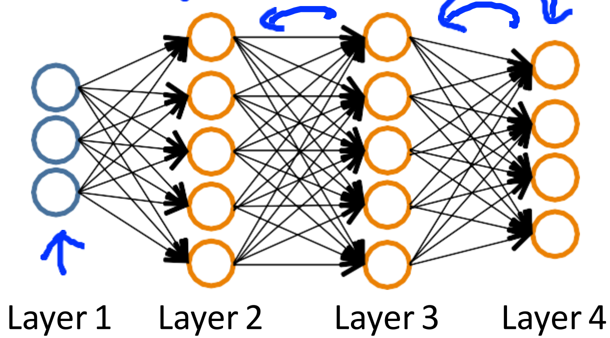
$$a^{(1)} = x$$
$$z^{(2)} = \Theta^{(1)} a^{(1)}$$
$$a^{(2)} = g(z^{(2)}) \quad (\text{add } a_0^{(2)})$$
$$z^{(3)} = \Theta^{(2)} a^{(2)}$$
$$a^{(3)} = g(z^{(3)}) \quad (\text{add } a_0^{(3)})$$
$$z^{(4)} = \Theta^{(3)} a^{(3)}$$
$$a^{(4)} = h_\Theta(x) = g(z^{(4)})$$



Gradient computation: Backpropagation algorithm

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

$\delta^{(2)}$ $a_j^{(l)}$ $\delta^{(3)}$ $\delta^{(4)}$



For each output unit (layer L = 4)

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

$(h_\Theta(x))_j$ $\quad \delta^{(4)} = a^{(4)} - y$

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

$a^{(3)} .* (1 - a^{(3)})$

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

$a^{(2)} .* (1 - a^{(2)})$

(No $\delta^{(1)}$)

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

(ignoring λ; if $\lambda = 0$)

Backpropagation algorithm

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

Set $\triangle_{ij}^{(l)} = 0$ (for all l, i, j). (use to compute $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$)

For $i = 1$ to m ← $(x^{(i)}, y^{(i)})$

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

$\triangle_{ij}^{(l)} := \triangle_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$ $\quad := \triangle^{(l)} + \delta^{(l+1)} (a^{(l)})^T$

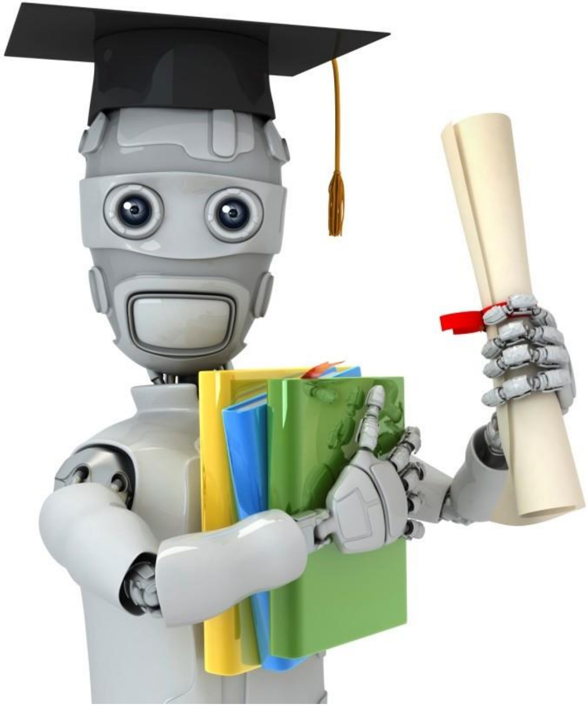
$$D_{ij}^{(l)} := \frac{1}{m} \triangle_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)} \text{ if } j \neq 0$$

$$D_{ij}^{(l)} := \frac{1}{m} \triangle_{ij}^{(l)} \qquad \text{ if } j = 0$$

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

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

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

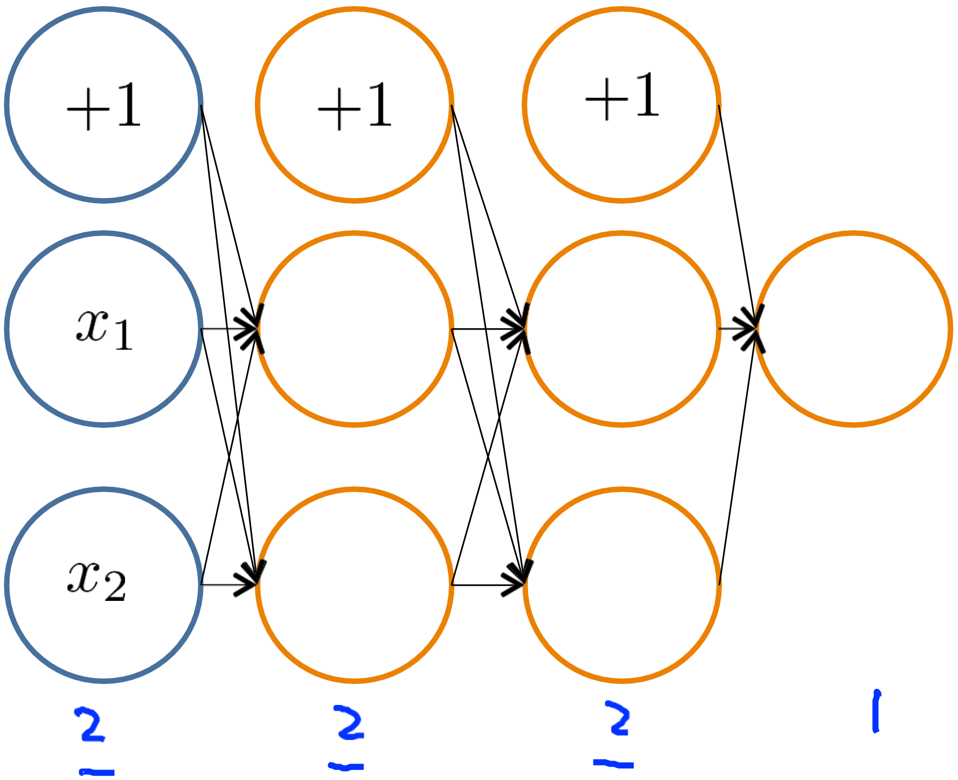


Neural Networks: Learning

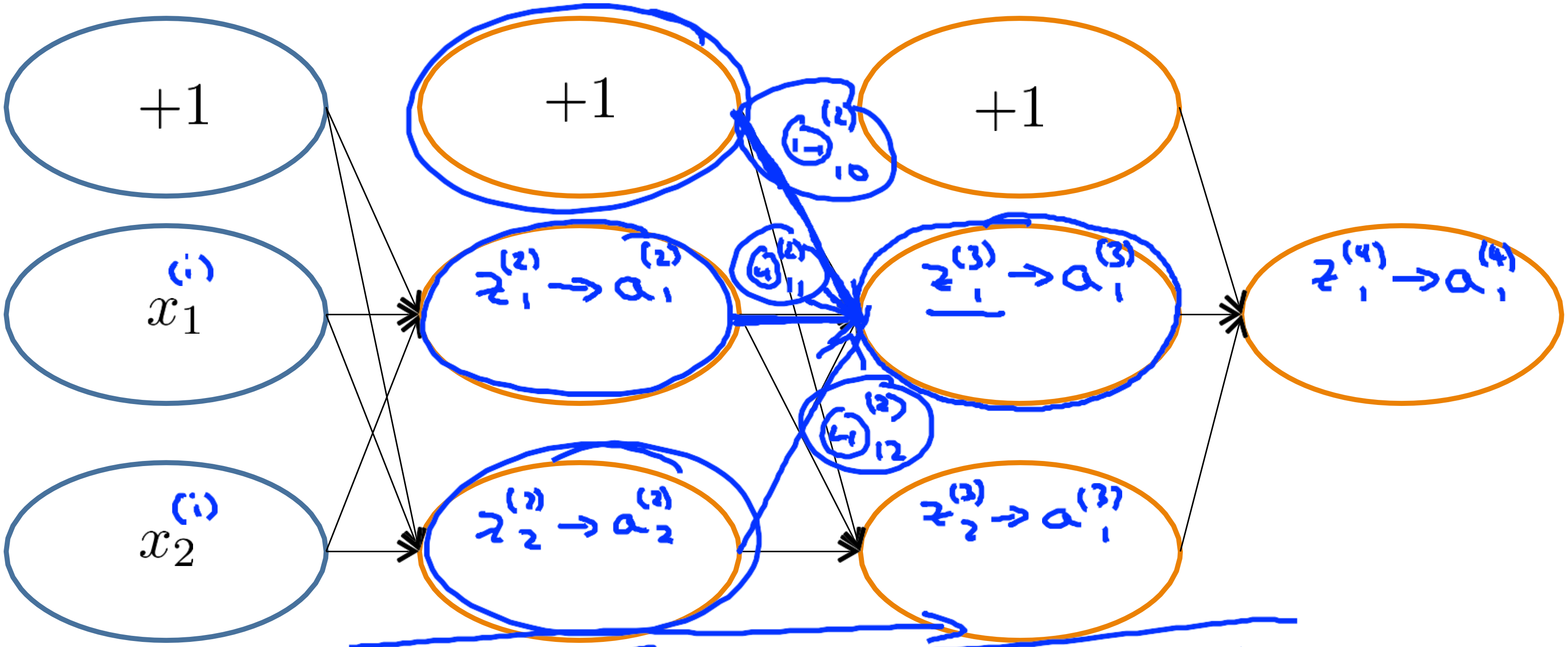
Backpropagation intuition

Machine Learning

Forward Propagation



Forward Propagation



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

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

What is backpropagation doing?

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

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

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

Focusing on a single example $x^{(i)}$, $y^{(i)}$, the case of 1 output unit, and ignoring regularization ($\lambda = 0$),

Note: Mistake on lecture, it is supposed to be 1-h(x).

1-

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

(Think of $\mathrm{cost}(\mathrm{i}) \approx (h_\Theta(x^{(i)}) - y^{(i)})^2$)

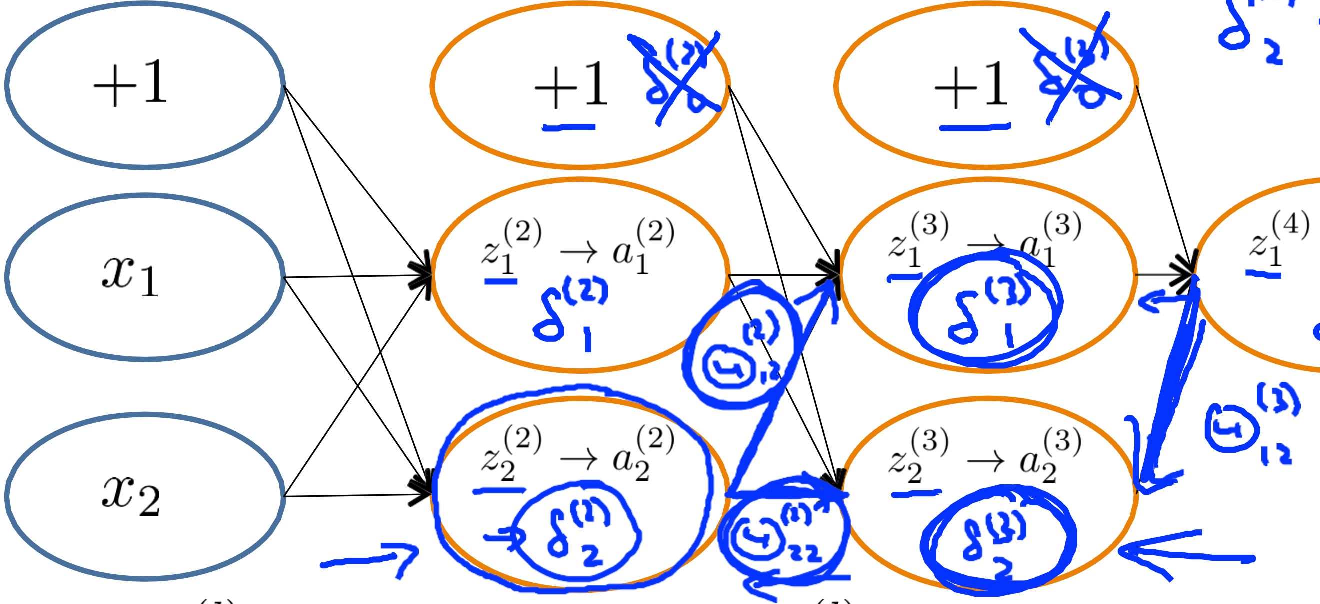
I.e. how well is the network doing on example i?

$y^{(i)}$

Forward Propagation

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

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



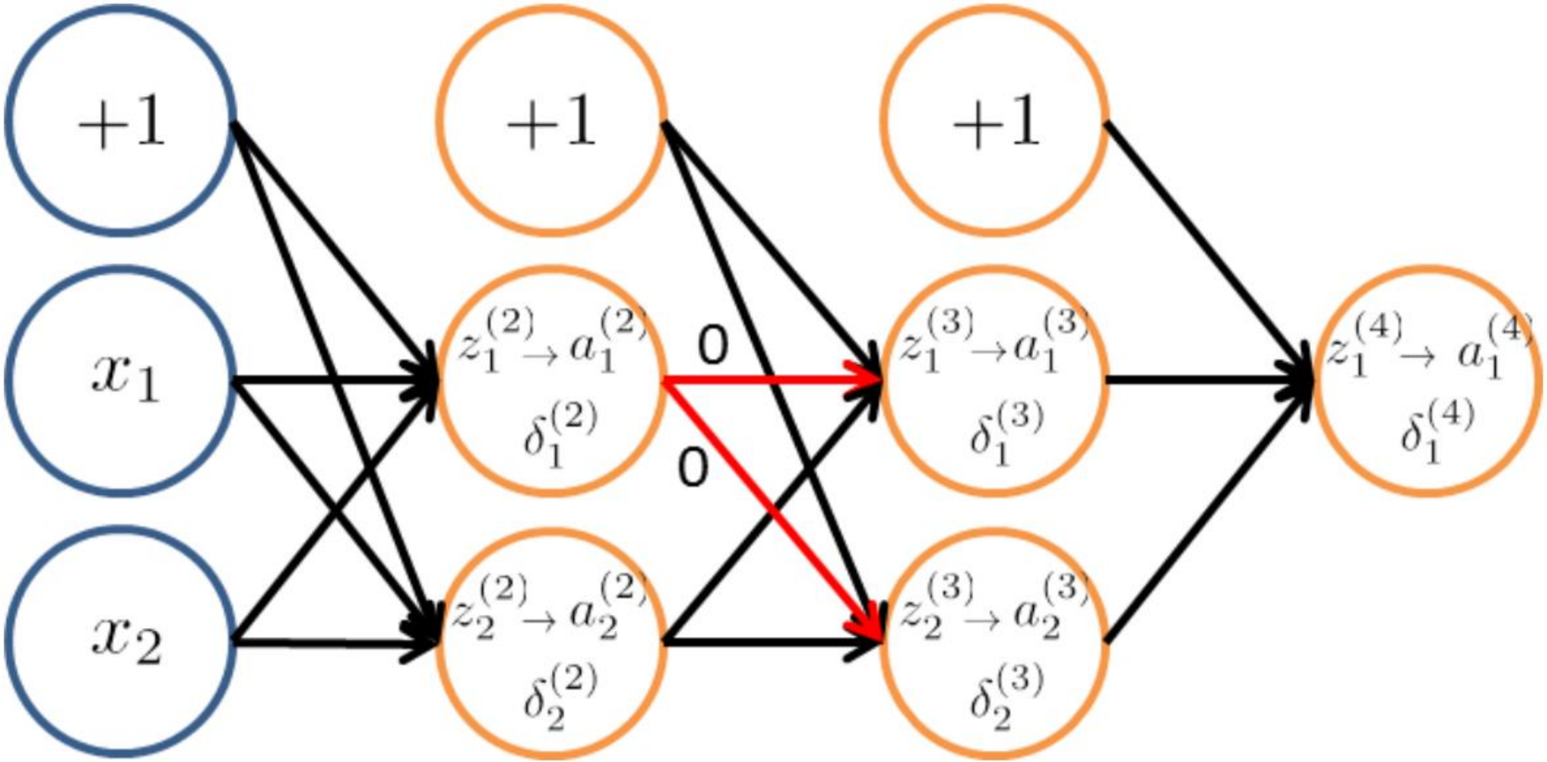
$$\delta_2^{(3)} = \Theta_{12}^{(3)} \cdot \delta_1^{(4)}.$$

$\delta_j^{(l)} =$ "error" of cost for $a_j^{(l)}$ (unit j in layer l).

Formally, $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \mathrm{cost}(\mathrm{i})$ (for $j \geq 0$), where

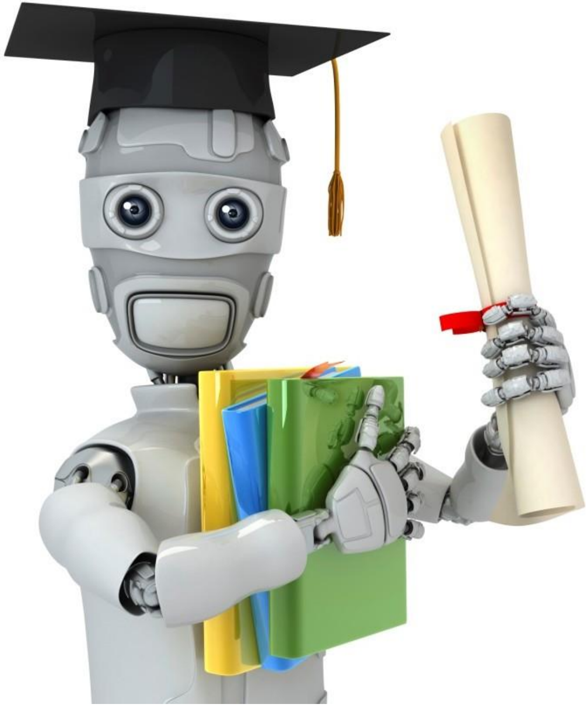
$$\mathrm{cost}(\mathrm{i}) = y^{(i)} \log h_\Theta(x^{(i)}) + (1 - y^{(i)}) \log h_\Theta(x^{(i)})$$

Consider the following neural network:



Suppose both of the weights shown in red ($\Theta_{11}^{(2)}$ and $\Theta_{21}^{(2)}$) are equal to 0. After running backpropagation, what can we say about the value of $\delta_1^{(3)}$?

- $\delta_1^{(3)} > 0$
- $\delta_1^{(3)} = 0$ only if $\delta_1^{(2)} = \delta_2^{(2)} = 0$, but not necessarily otherwise
- $\delta_1^{(3)} \leq 0$ regardless of the values of $\delta_1^{(2)}$ and $\delta_2^{(2)}$
- There is insufficient information to tell



Machine Learning

Neural Networks: Learning

Implementation note: Unrolling parameters

Advanced optimization

```
function [jVal, gradient] = costFunction(theta)
   ...
optTheta = fminunc(@costFunction, initialTheta, options)
```

$\rightarrow \mathbb{R}^{n+1}$ $\rightarrow \mathbb{R}^{n+1}$ (vectors)

Neural Network (L=4):

$\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$ - matrices (`Theta1, Theta2, Theta3`)

$D^{(1)}, D^{(2)}, D^{(3)}$ - matrices (`D1, D2, D3`)

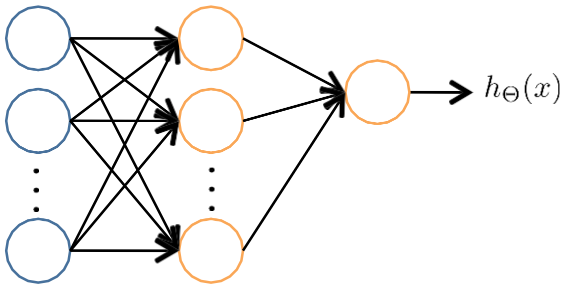
"Unroll" into vectors

Example

$$s_1 = 10, s_2 = 10, s_3 = 1$$

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

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



$\Theta^{(1)}$ $\Theta^{(2)}$ $\Theta^{(3)}$

```
thetaVec = [ Theta1(:); Theta2(:); Theta3(:)];
DVec = [D1(:); D2(:); D3(:)];

Theta1 = reshape(thetaVec(1:110),10,11);
Theta2 = reshape(thetaVec(111:220),10,11);
Theta3 = reshape(thetaVec(221:231),1,11);
```

Learning Algorithm

Have initial parameters $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$.

Unroll to get `initialTheta` to pass to

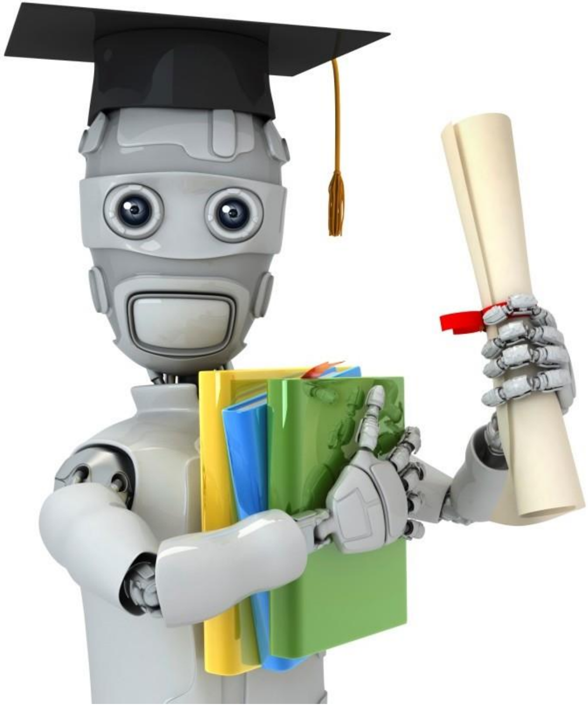
`fminunc(@costFunction, initialTheta, options)`

`function [jval, gradientVec] = costFunction(thetaVec)`

From `thetaVec`, get $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$. reshape

Use forward prop/back prop to compute $D^{(1)}, D^{(2)}, D^{(3)}$ and $J(\Theta)$

unroll $D^{(1)}, D^{(2)}, D^{(3)}$ to get `gradientVec`.

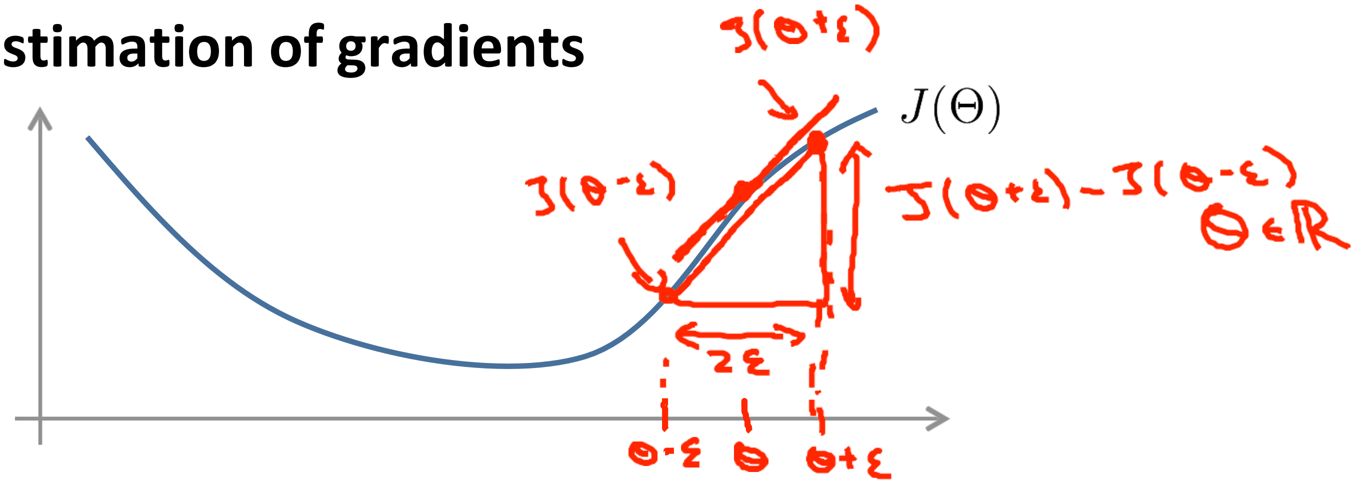


Machine Learning

Neural Networks: Learning

Gradient checking

Numerical estimation of gradients



$$\frac{d}{d\theta} J(\theta) \approx \underbrace{\frac{J(\theta+\varepsilon)-J(\theta-\varepsilon)}{2\varepsilon}} \quad \leftarrow \quad \Bigg| \quad \cancel{\frac{J(\theta+\varepsilon)-J(\theta)}{\varepsilon}}$$

$$\varepsilon = 10^{-4} \leftarrow$$

Implement: `gradApprox = (J(theta + EPSILON) – J(theta – EPSILON)) /(2*EPSILON)`

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 $\frac{d}{d\theta} J(\theta) = 3$).

- 3.0000
- $\boxed{\text{3.0001}}$
- 3.0301
- 6.0002

Parameter vector θ

$\theta \in \mathbb{R}^n$ (E.g. θ is "unrolled" version of $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$)

$$\theta = \theta_1, \theta_2, \theta_3, \ldots, \theta_n$$

$$\frac{\partial}{\partial \theta_1} J(\theta) \approx \frac{J(\theta_1+\epsilon, \theta_2, \theta_3, \ldots, \theta_n) - J(\theta_1-\epsilon, \theta_2, \theta_3, \ldots, \theta_n)}{2\epsilon}$$

$$\frac{\partial}{\partial \theta_2} J(\theta) \approx \frac{J(\theta_1, \theta_2+\epsilon, \theta_3, \ldots, \theta_n) - J(\theta_1, \theta_2-\epsilon, \theta_3, \ldots, \theta_n)}{2\epsilon}$$

$$\frac{\partial}{\partial \theta_n} J(\theta) \approx \frac{J(\theta_1, \theta_2, \theta_3, \ldots, \theta_n+\epsilon) - J(\theta_1, \theta_2, \theta_3, \ldots, \theta_n-\epsilon)}{2\epsilon}$$

$$\begin{bmatrix}\theta_1\\ \theta_2\\ \vdots\\ \theta_i+\epsilon\\ \vdots\\ \theta_n\end{bmatrix} \to \theta_i-\epsilon$$

```
for i = 1:n,
    thetaPlus = theta;
    thetaPlus(i) = thetaPlus(i) + EPSILON;
    thetaMinus = theta;
    thetaMinus(i) = thetaMinus(i) - EPSILON;
    gradApprox(i) = (J(thetaPlus) - J(thetaMinus))
                    /(2*EPSILON);
end;
```

$$\frac{\partial}{\partial\theta_i} J(\theta).$$

Check that `gradApprox` ≈ `DVec`

From backprop.

Implementation Note:

- Implement backprop to compute **DVec** (unrolled $D^{(1)}, D^{(2)}, D^{(3)}$).
- Implement numerical gradient check to compute **gradApprox**.
- Make sure they give similar values.
- Turn off gradient checking. Using backprop code for learning.

DVec

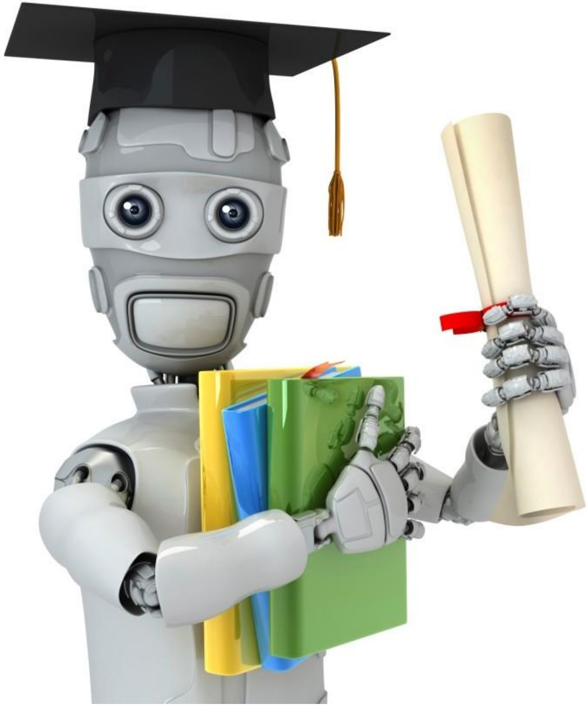
$\delta^{(4)}, \delta^{(3)}, \delta^{(2)}$

Important:

- Be sure to disable your gradient checking code before training your classifier. If you run numerical gradient computation on every iteration of gradient descent (or in the inner loop of `costFunction(…)`)your code will be <u>very</u> slow.

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

- The numerical gradient computation method is much harder to implement.
- The numerical gradient algorithm is very slow.
- Backpropagation does not require setting the parameter EPSILON.
- None of the above.



Machine Learning

Neural Networks: Learning

Random initialization

Initial value of Θ

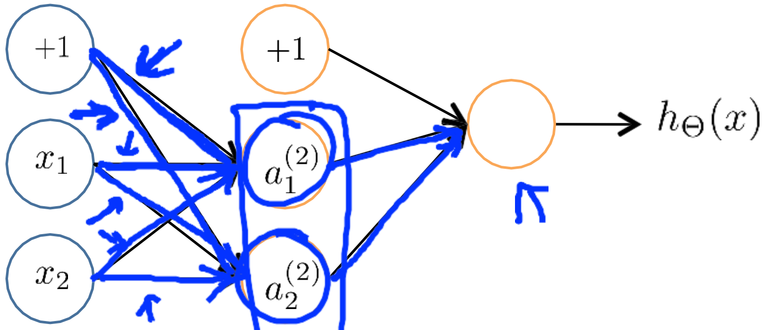
For gradient descent and advanced optimization method, need initial value for Θ.

```
optTheta = fminunc(@costFunction,
               initialTheta, options)
```

Consider gradient descent

Set `initialTheta = zeros(n,1)` ?

Zero initialization



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

$a_1^{(2)} = a_2^{(2)}$. Also $\delta_1^{(2)} = \delta_2^{(2)}$.

$$\frac{\partial}{\partial \Theta_{01}^{(1)}} J(\Theta) = \frac{\partial}{\partial \Theta_{02}^{(1)}} J(\Theta) \qquad \Theta_{01}^{(1)} = \Theta_{02}^{(1)}$$

After each update, parameters corresponding to inputs going into each of two hidden units are identical.

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

Random initialization: Symmetry breaking

Initialize each $\Theta_{ij}^{(l)}$ to a random value in $[-\epsilon, \epsilon]$
(i.e. $-\epsilon \leq \Theta_{ij}^{(l)} \leq \epsilon$)

random 10x11 matrix (betw. 0 and 1)

E.g.

```
Theta1 = rand(10,11)*(2*INIT_EPSILON)
         - INIT_EPSILON;
```

$[-\epsilon, \epsilon]$

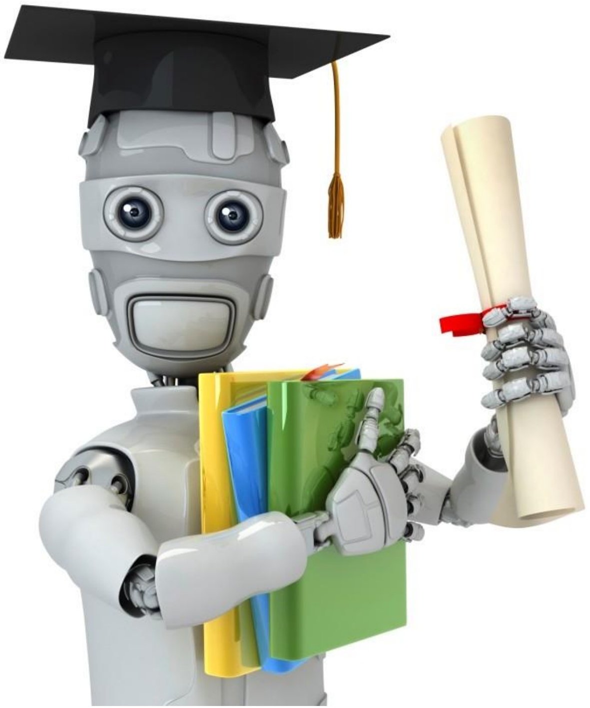
```
Theta2 = rand(1,11)*(2*INIT_EPSILON)
         - INIT_EPSILON;
```

Consider this procedure for initializing the parameters of a neural network:

1. Pick a random number r = rand(1,1) * (2 * INIT_EPSILON) - INIT_EPSILON;
2. Set $\Theta_{ij}^{(l)} = r$ for all i, j, l.

Does this work?

- Yes, because the parameters are chosen randomly.
- Yes, unless we are unlucky and get r=0 (up to numerical precision).
- Maybe, depending on the training set inputs x(i).
- No, because this fails to break symmetry.



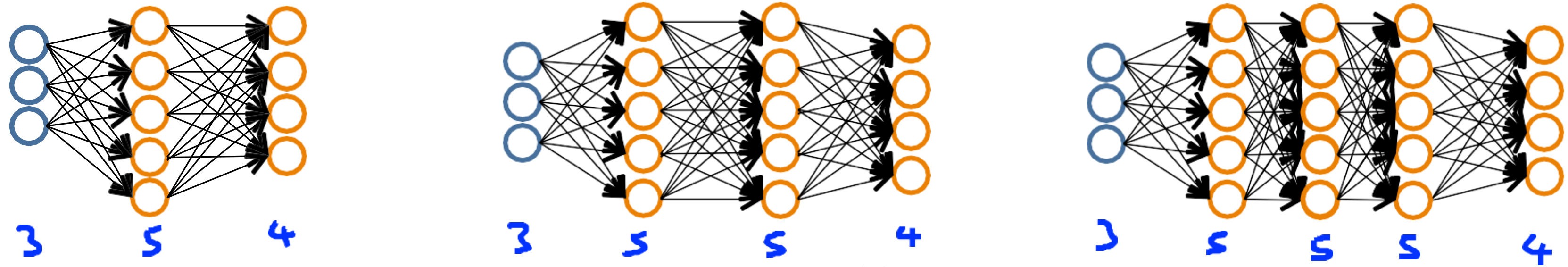
Machine Learning

Neural Networks: Learning

Putting it together

Training a neural network

Pick a network architecture (connectivity pattern between neurons)



No. of input units: Dimension of features $x^{(i)}$

No. output units: Number of classes

Reasonable default: 1 hidden layer, or if >1 hidden layer, have same no. of hidden units in every layer (usually the more the better)

$y \in \{1, 2, 3, \ldots, 10\}$

$\cancel{y = 5}$

$y = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$ or $\begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$ $\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \leftarrow$

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

`for i = 1:m` { $(x^{(1)}, y^{(1)})$ $(x^{(2)}, y^{(2)})$, ... $(x^{(m)}, y^{(m)})$

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

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

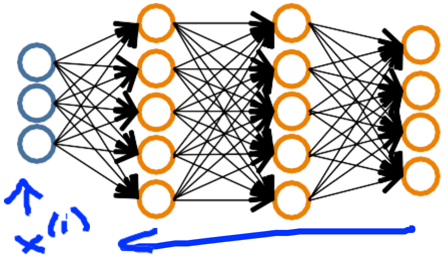
$\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)}(a^{(l)})^T$

....

}

...

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



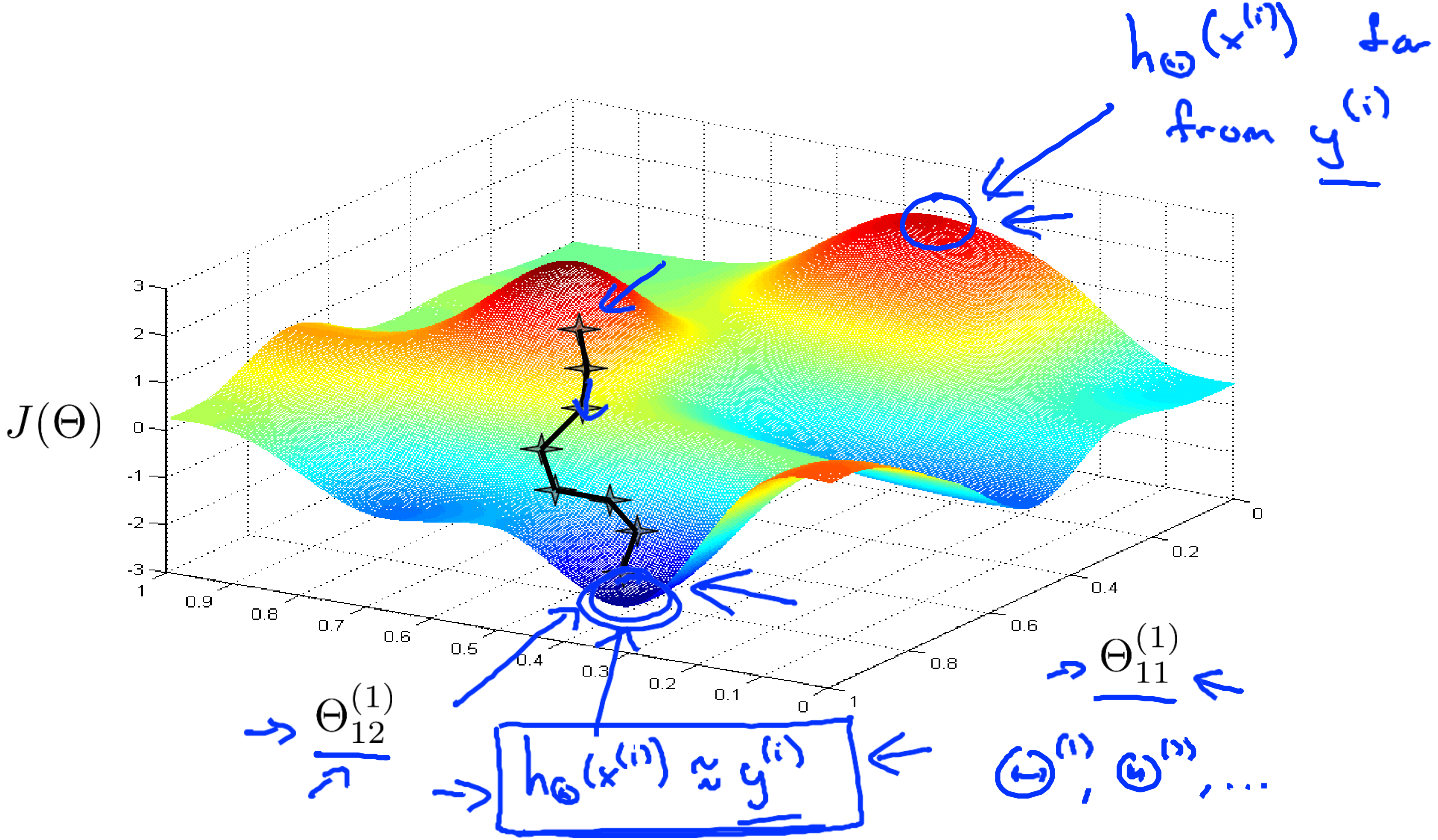
Andrew Ng

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 of $J(\Theta)$.
   Then disable gradient checking code.
6. Use gradient descent or advanced optimization method with backpropagation to try to minimize $J(\Theta)$ as a function of parameters Θ

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

$J(\Theta)$ – non-convex.



$h_\Theta(x^{(i)})$ far
from $y^{(i)}$
$J(\Theta)$
3
2
1
0
-1
-2
-3
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
0.2
0.4
0.6
0.8
1
$\Theta^{(1)}_{11}$
$\Theta^{(1)}_{12}$
$h_\Theta(x^{(i)}) \approx y^{(i)}$
$\Theta^{(1)}, \Theta^{(2)}, \ldots$

Andrew Ng

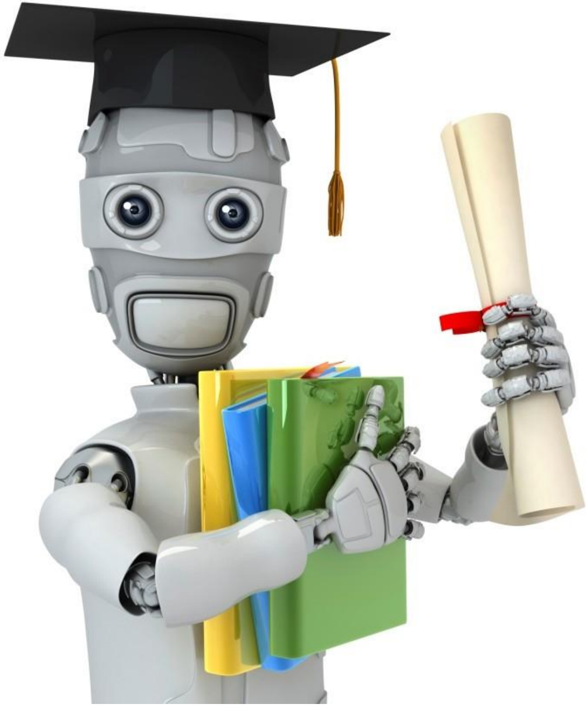
Suppose you are using gradient descent together with backpropagation to try to minimize $J(\Theta)$ as a function of Θ. Which of the following would be a useful step for verifying that the learning algorithm is running correctly?

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

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

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

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



Machine Learning

Neural Networks: Learning

Backpropagation example: Autonomous driving (optional)

[Courtesy of Dean Pomerleau]