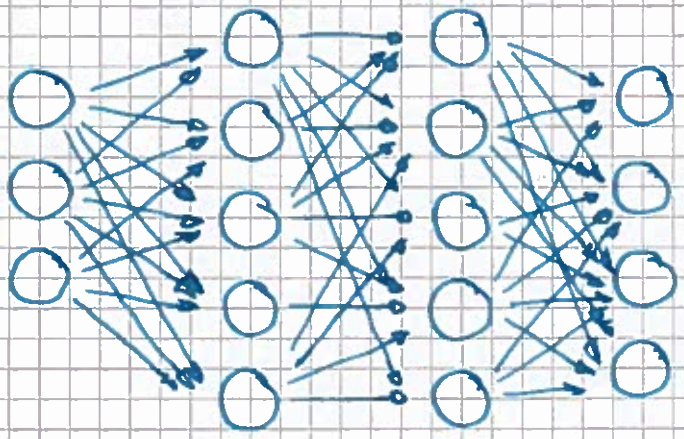
Chapter 9

Neural Networks - Cost function & Back propagation

Neural Network (Classification)



Layer 1 Layer 2 Layer 3 Layer 4

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

L = total # of layers in Network $\Rightarrow L = 4$

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

$s_1 = 3,\ s_2 = 5,\ s_3 = 5,\ s_4 = 4 = s_L$

Binary classification

$y = 0$ or 1 $\Rightarrow$ $\rightarrow$ $h_\Theta(x)$

1 Output unit

$h_\Theta(x) \Rightarrow \mathbb{R}$

$s_L = 1 \Rightarrow K = 1$

# of output units in Binary classification = 1.

Multi-class classification (K classes)

$y \in \mathbb{R}^K$

e.g. $\begin{bmatrix}1\\0\\0\\0\end{bmatrix}$ $\begin{bmatrix}0\\1\\0\\0\end{bmatrix}$ $\begin{bmatrix}0\\0\\1\\0\end{bmatrix}$ $\begin{bmatrix}0\\0\\0\\1\end{bmatrix}$

K output units

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

$s_L = K \quad (K \geq 3)$

If only 2 output classes use Binary classification

Logistic regression:

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

Neural Network:

$h_\theta(x) \in \mathbb{R}^K \Rightarrow (h_\theta(x))_i = i^{th}$ output

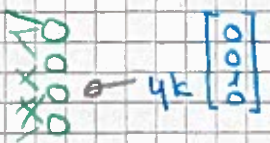
$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log\left(h_\theta(x^{(i)})\right)_k + (1-y_k^{(i)})\cdot\log\left(1-h_\theta(x^{(i)})_k\right)\right]$$

$$+ \underbrace{\frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}\left(\Theta_{ji}^{(l)}\right)^2}_{\text{called weight decay}}$$

K sum of output unit

for neural networks, cost function is a generalization. It generates K outputs $\Rightarrow$ K dimensional vector

$h_\Theta(x)$ is K dim. vector $\Rightarrow h_\Theta(x)_i$ refers to i^{th} value in vector



4 output units in final layer

Gradient Computation:

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

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

use

$$\min_\Theta J(\Theta)$$

Need to compute

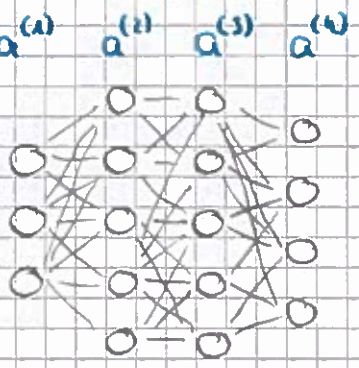
- $J(\Theta)$

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

Given one training example (x, y)

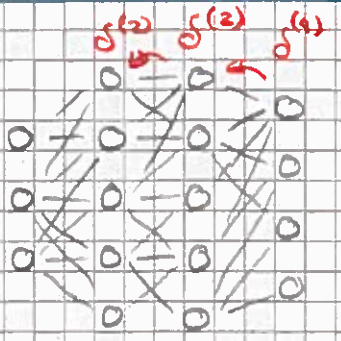
Forward propagation

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



Gradient computation: Backpropagation algorithm

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



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

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

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

dimension of vector is equal to number of output units

$$\delta^{(3)} = (\Theta^{(3)})^T \delta^{(4)} .* \underbrace{g'(z^{(3)})}_{a^{(3)} .* (1 - a^{(3)})}$$

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

(no $\delta^{(1)}$)!

↳ $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = a_j^{(l)} \cdot \delta_i^{(l+1)}$ (ignoring λ; if $\lambda = 0$)

Back propagation algorithm:

training set: $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots (x^{(m)}, y^{(m)})\}$

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

For $i = 1 : m$

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

perform forward propagation $\delta^{(L)} = y^{(i)}$

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

l - layer

j = node in the layer

i = error of affected node in layer

$\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \cdot \delta_i^{(l+1)}$ ⇒ accumulate partial derivative terms

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

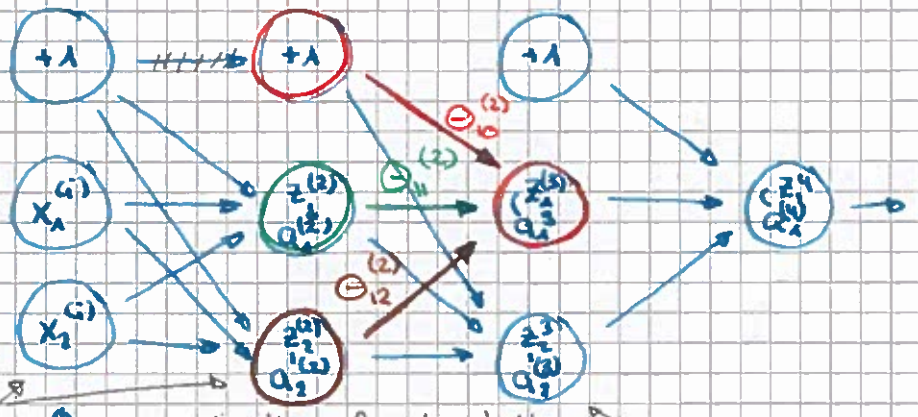
end

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

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

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

Forward propagation:



calculate: $z_1^{(3)} = \underbrace{\Theta_{10}^{(1)} \cdot 1}_{} + \underbrace{\Theta_{11}^{(1)} a_1^{(2)}}_{} + \underbrace{\Theta_{12}^{(1)} a_2^{(2)}}_{}$

What is backpropagation doing?

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \log(h_\Theta(x^{(i)}) - y^{(i)}) + (1 - y^{(i)}) \log(1 - (h_\Theta(x^{(i)}))\right]$$
$$+ \frac{\lambda}{2m} \sum_{l=1}^{L-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_l+1} (\Theta_{ji}^{(l)})^2$$

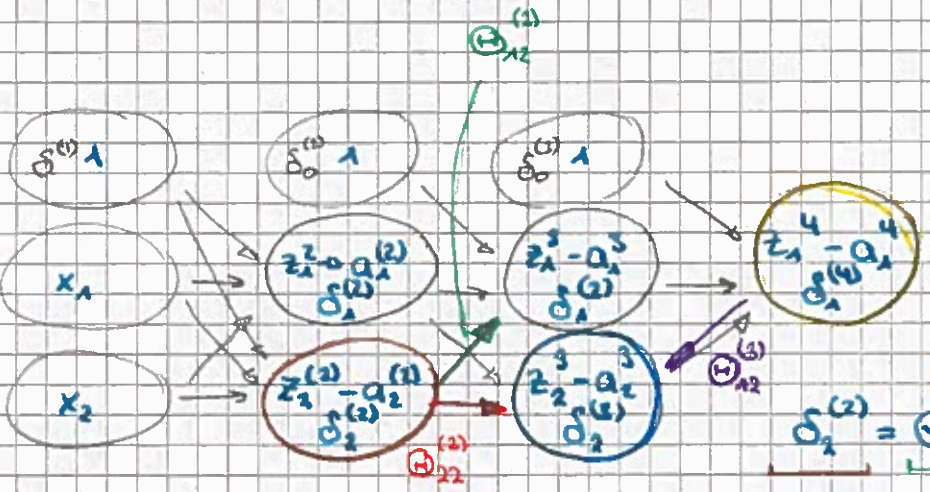
Focus on a single example $x^{(i)}, y^{(i)}$, the case of 1 output unit, and ignoring regularization ($\lambda = 0$),

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

↳ think of $\text{cost}(i) \approx (h_\Theta(x^{(i)} - y^{(i)})^2$

how well is the network doing on example i?

Backward propagation intuition



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

real value — predicted value

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

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

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

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

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

Implementation Note: Unrolling Parameters

Advanced optimization:

```
function [jVal, gradient] = costFunction(theta)
  ....                      % jVal: ℝ vector; gradient: ℝ^{n+1} vectors for Linear Regression
optTheta = fminunc(@costFunction, initialTheta, options)
```

Neural Network (L=4)

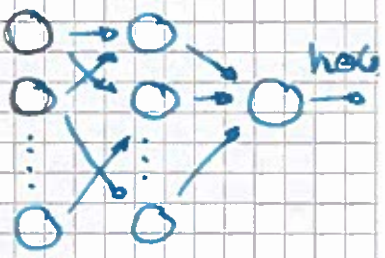
$\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$ - matrices (Theta1, Theta2, Theta3)

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

"Unroll" into vectors

Example:

$s_1 = 10$ (input layer 10 units), $s_2 = 10$ (hidden layer 10 units), $s_3 = 1$ - output layer 1 unit



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

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

In Octave:

Create a vector:

```
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);
```

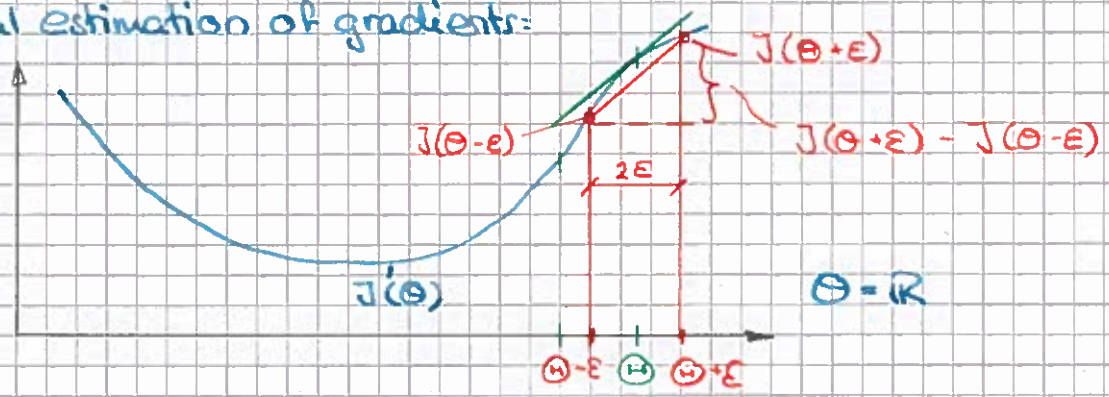
Learning Algorithm

- 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 to get matrices
Use forward propagation/backward prop. to compute
$D^{(1)}, D^{(2)}, D^{(3)}$ and $J(\Theta)$
Unroll $D^{(1)}, D^{(2)}, D^{(3)}$ to get gradientVec

Numerical estimation of gradients:



$$\frac{\partial}{\partial\theta} J(\theta) \approx \frac{J(\theta+\varepsilon) - J(\theta-\varepsilon)}{2\varepsilon}$$

$\varepsilon = 10^{-4}$

2 sided difference

more accurate - use this

$$\frac{J(\theta+\varepsilon) - J(\theta)}{\varepsilon}$$

1 sided difference

Implementation:

$$\text{gradApprox} = \frac{J(\text{theta} + \text{EPSILON}) - J(\text{theta} - \text{EPSILON})}{2 \cdot \text{EPSILON}}$$

Parameters Vector:

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

$\theta = \theta_1, \theta_2 \dots \theta_n$

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

$$\frac{\partial}{\partial\theta_2} J(\theta) = \frac{J(\theta_1, \theta_2+\varepsilon, \theta_3 \dots \theta_n) - J(\theta_1, \theta_2-\varepsilon, \theta_3 \dots \theta_n)}{2\varepsilon}$$

$$\vdots$$

$$\frac{\partial}{\partial\theta_n} J(\theta) = \frac{J(\theta_1, \theta_2, \theta_3 \dots \theta_n+\varepsilon) - J(\theta_1, \theta_2, \theta_3 \dots \theta_n-\varepsilon)}{2\varepsilon}$$

Octave implementation:

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

$\theta = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_{i+\epsilon} \\ \vdots \\ \theta_n \end{bmatrix} \rightarrow \theta_{i+\epsilon}$

gradApprox(i) $\approx \frac{\partial}{\partial \theta_i} J(\theta)$

Check that gradApprox $\approx$ DVec

gradApprox: numerically computed; DVec: backwards propagation

Implementation note:

- Implement back prop 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. Use back prop code for learning
  - ↳ DVec computationally efficient

Important:

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

Initial Value of Θ

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

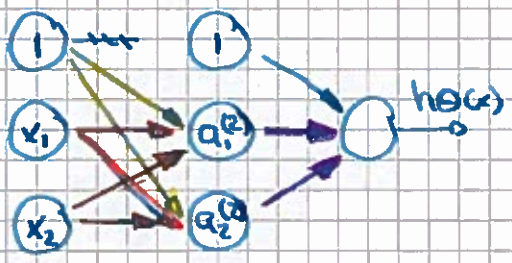
```
optTheta = fminunc(@costFunction, initialTheta, options)
```

Consider gradient descent

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

Zero initialization

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



elements in same color will have the same value

We end up for all our traing samples

$a_1^{(2)} = a_2^{(2)}$, also $\delta_1^{(2)} = \delta_2^{(2)}$

$\Rightarrow$ same applies for the partial derivatives.

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

Even after the first iteration

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

Random initialization: Symmetry breaking

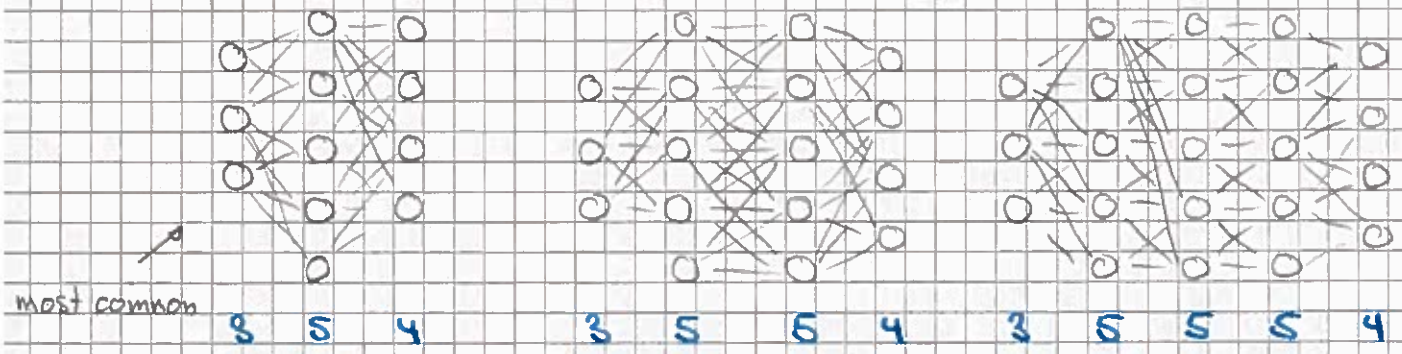
Initialize each $\Theta_{ij}^{(l)}$ to a random value $[-\varepsilon, \varepsilon]$ (e.g. $-\varepsilon \leq \Theta_{ij}^{(l)} \leq \varepsilon$)

→ rand 10×11 Matrix (between 0 and 1)

```
Theta1 = rand(10,11) * (2 * INIT_EPSILON) - INIT_EPSILON;
                                  [-e,e]
Theta2 = rand(1,11) * (2 * INIT_EPSILON) - INIT_EPSILON;
```

Training a neural network:

Pick the network architecture (connectivity pattern between neurons)



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

No. of output units: Number of classes

Reasonable default: 1 hidden layer, or if >1 hidden layer, have same number of hidden layer units in every layer (usually the more the better)

⇒ multi class classification (e.g $y \in \{1,2,3,4 \dots 10\}$)

$$y_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \quad y_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \quad y_5 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

e.g. # features = 120 / output 0 - 9

input units = 120

hidden layer 2-3 * 120 units

output units 10

Training a neural network:

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

for $i = 1:m$ {
Perform forward propagation and backprop using example $(x^{(i)}, y^{(i)})$
(Get activations $a^{(l)}$ and delta terms $\delta^{(l)}$ for $l = 2, \dots L$).
$\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} (a^{(l)})^T$
}

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

5) Use gradient checking to compute $\frac{\partial}{\partial \Theta_{ijk}^{(l)}} J(\Theta)$ 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 backprop to try to minimize $J(\Theta)$ as a function of parameters Θ

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

$J(\Theta)$ - non convex → can get to local minimum which is not a problem.