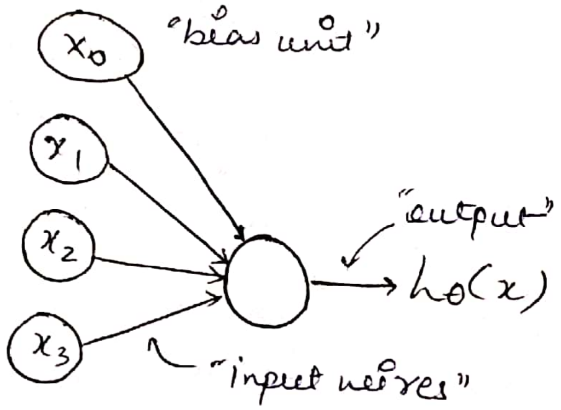
Neural Networks

Non linear Hypothesis

- Adding quadratic and cubic features may be very large in numbers so it will not be effective to use logistic & linear regression

Neurons and the brain

Model Representation (Neuron Model)



$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix}$$

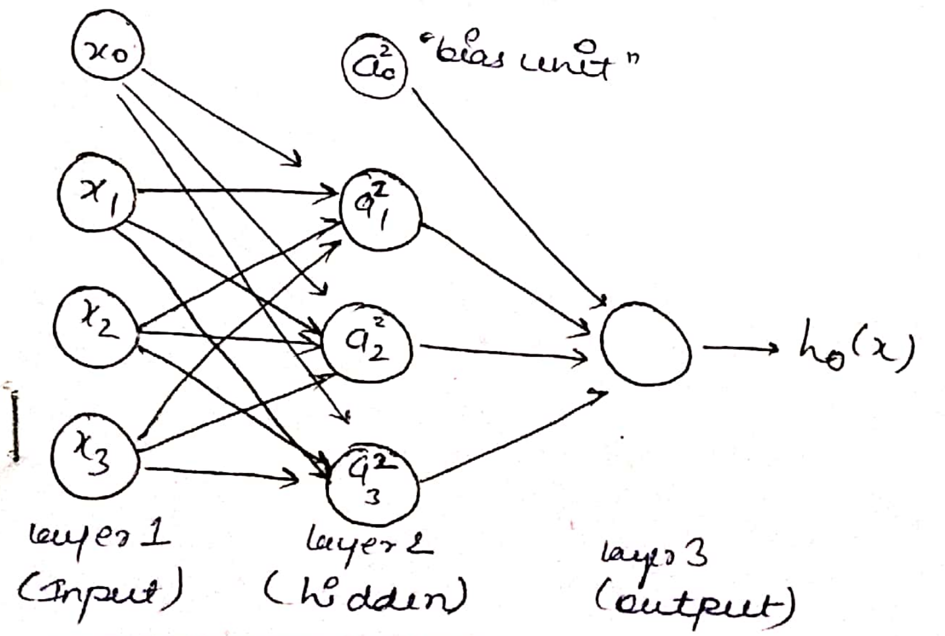
"weights" (parameters)

$$h_\theta(x) = \frac{1}{1+e^{-\theta^T x}}$$

Sigmoid (logistic) activation function

$$g(z) = \frac{1}{1+e^{-z}}$$

Neural Network



- Any layer with no input (X) or output (y) is hidden layer

$a_i^{(j)}$ → "activation" of unit i in layer j

$\theta^{(j)}$ → matrix of weight controlling function mapping from layer j to layer $j+1$.

- If network has s_j units in layer j, s_{j+1} units in layer $j+1$, then $\theta^{(j)}$ will be of dimension $s_{j+1} \times (s_j + 1)$

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

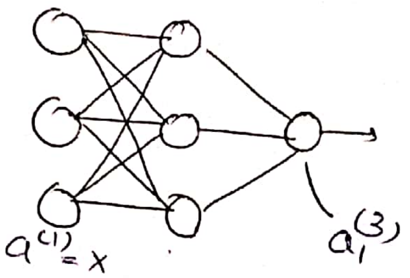
$$a_2^{(2)} = g(\theta_{20}^{(1)} x_0 + \theta_{21}^{(1)} x_1 + \theta_{22}^{(1)} x_2 + \theta_{23}^{(1)} x_3)$$

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

$$h_\theta(x) = a_1^{(3)} = g(\theta_{10}^{(2)} a_0^{(2)} + \theta_{11}^{(2)} a_1^{(2)} + \theta_{12}^{(2)} a_2^{(2)} + \theta_{13}^{(2)} a_3^{(2)})$$

Forward Propagation : Vectorized Implementation

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



$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad z^{(2)} = \begin{bmatrix} z_1^{(2)} \\ z_2^{(2)} \\ z_3^{(2)} \end{bmatrix} \qquad a_1^{(2)} = g(z_1^{(2)})$$

$z^{(2)} = \Theta^{(1)} x \Rightarrow \Theta^{(1)} a^{(1)}$ (defining $a^{(1)} = x$ in input layer)

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

Add $a_0^{(2)} = 1 \rightarrow a^{(2)} \in \mathbb{R}^4$

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

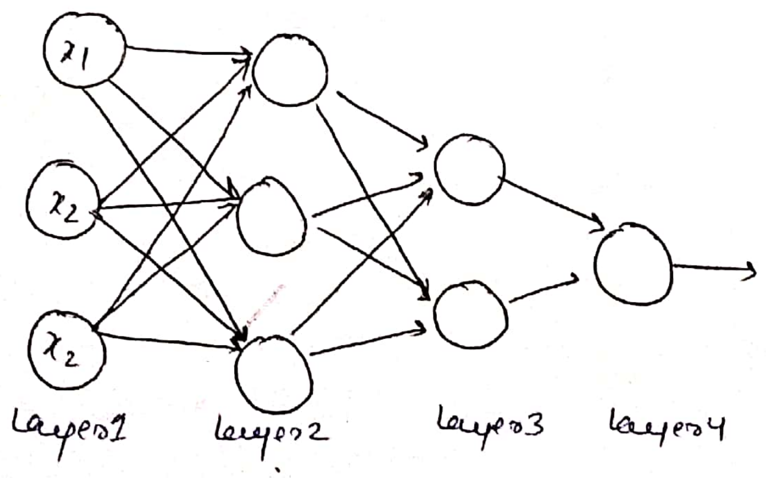
$h_\Theta(x) = a^{(3)} = g(z^{(3)})$

Neural Networks learning its own features

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

rather than using original features its using features a_1 a_2 a_3. They themselves are learned as functions of input. Mapping of function from layer 1 to layer 2 is defined by some other parameters.

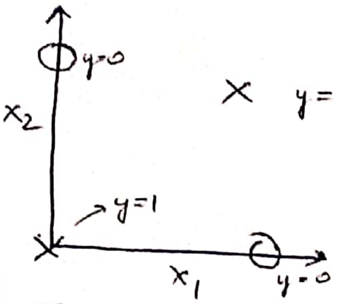
Other network architectures



Neural Examples and Intuitions

Non Linear classification example : XOR/XNOR

x_1, x_2 are binary (0 or 1)

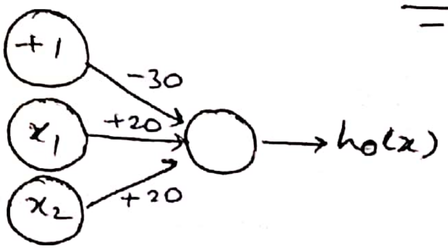


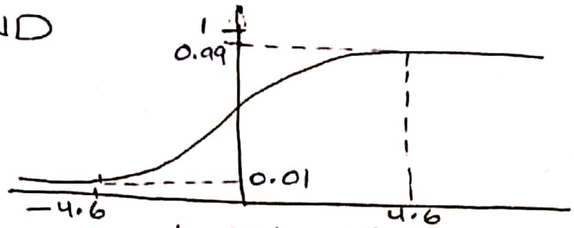
$y = x_1 \text{ XOR } x_2$

Simple example : AND

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

$y = x_1 \text{ AND } x_2$





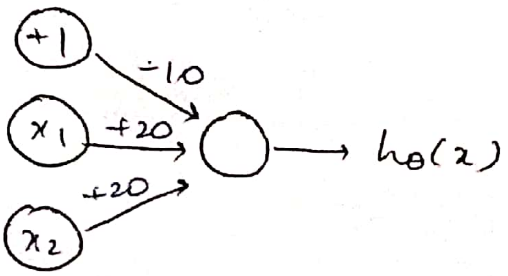
$h_\theta(x) = g(-30 + 20x_1 + 20x_2)$

$\Theta^{(1)}_{10}$ $\Theta^{(1)}_{11}$ $\Theta^{(1)}_{12}$

x_1	x_2	$h_\theta(x)$
0	0	$g(-30) \approx 0$
0	1	$g(-10) \approx 0$
1	0	$g(-10) \approx 0$
1	1	$g(10) \approx 1$

AND

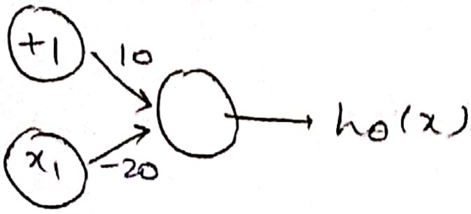
- Simple example : OR



x_1	x_2	$h_\theta(x)$
0	0	$g(-10) \approx 0$
0	1	$g(10) \approx 1$
1	0	$g(10) \approx 1$
1	1	$g(30) \approx 1$

Scanned with CamScanner

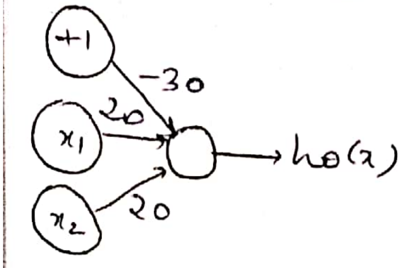
Negation (NOT)



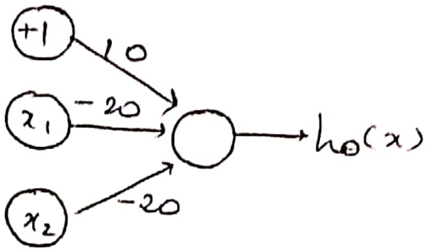
x_1	$h_\theta(x)$
0	$g(10) \approx 1$
1	$g(-10) \approx 0$

$h_\theta(x) = g(10 - 20x_1)$

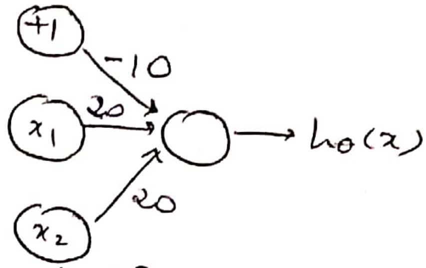
x_1 XNOR x_2



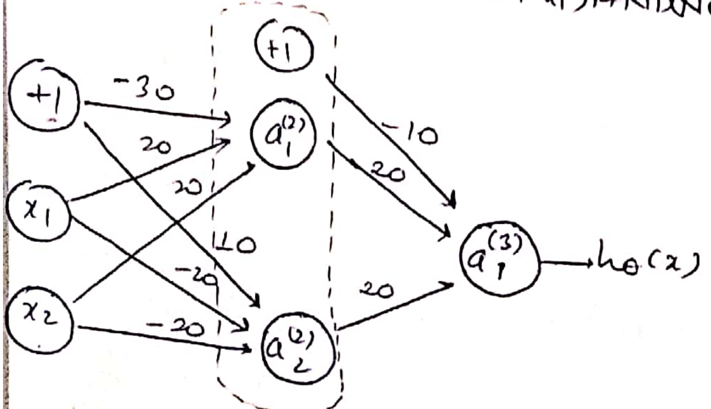
x_1 AND x_2



(NOT x_1) AND (NOT x_2)

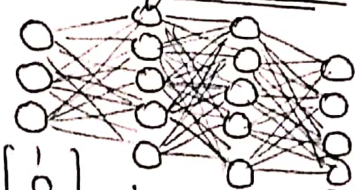


x_1 OR x_2



x_1	x_2	$a_1^{(2)}$	$a_2^{(2)}$	$h_\theta(x)$
0	0	0	1	1
0	1	0	0	0
1	0	0	0	0
1	1	1	0	1

Multiclass classification



$h_\theta(x) \in R^4$

Want $h_\theta(x) \approx \begin{bmatrix}1\\0\\0\\0\end{bmatrix}$ when pedestrian, $h_\theta(x) \approx \begin{bmatrix}0\\1\\0\\0\end{bmatrix}$ when car, $h_\theta(x) \approx \begin{bmatrix}0\\0\\1\\0\end{bmatrix}$ when bike, etc

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

$y^{(i)}$ one of $\begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$ $\rightarrow (x^{(i)}, y^{(i)})$

$\underbrace{h_\theta(x^{(i)}) \approx y^{(i)}}_{R^4}$

Neural Networks Learning

cost function and Backpropagation

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

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

L = total no. of layers

s_l = no. of units (not bias inc.) in layer l

Gradient Computation

Forward Propagation

$a^{(1)} = x$

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

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

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

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

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

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

Back Propagation

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

for each output unit (layer L=4)

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

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

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

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

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

Set $\Delta_{ij}^{(l)} = 0$ (for all l, i, j) ⟶ used to compute $\frac{\partial}{\partial \theta_{ij}}(J(\theta))$

for $i = 1$ to m

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

Perform forward propagation to compute $a^{(l)}$, $l = 2, 3, \ldots l$

using $y^{(i)}$ compute $\delta^{(L)} = a^{(L)} - y^{(i)}$

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

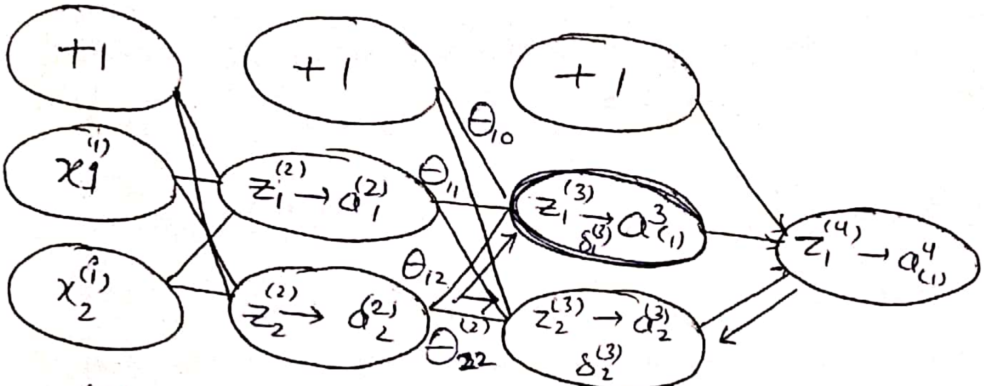
$\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$

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

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

Intuition

Forward Propagation



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

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

$$\delta_2^{(2)} = \theta_{12}\delta_1^{(3)} + \theta_{12}^{(2)}\delta_2^{(3)}$$ → Here we multiply the weights $\theta_{12}^{(2)}$ and $\theta_{22}^{(2)}$ by their respective values of δ found to right of each edge

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

using back propagation

Back Propagation in Practise

Implementation Note : Unrolling Parameters

Neural Network ($L=4$)

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

→ $D^{(1)}, D^{(2)}, D^{(3)}$ - matrices (D_1, D_2, D_3)

"unroll into vectors"

~~Advanced optimization function [jVal, gradient~~

Example

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

$D^{(1)} \in R^{10\times11}$, $D^{(2)} \in R^{10\times11}$, $D^{(3)} \in R^{1\times11}$

$\theta^{(1)} \in R^{10\times11}$, $\theta^{(2)} \in R^{10\times11}$, $\theta^{(3)} \in R^{1\times11}$

Octave code to roll and unroll theta

```
thetaVec = [Theta1(:); Theta2(:); Theta3(:)];
DVec = [D1(:); D2(:); D3(:)];
Theta1 = reshape(thetaVec(1:110),10,11)
Theta2 = reshape(thetaVec(110:220),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)

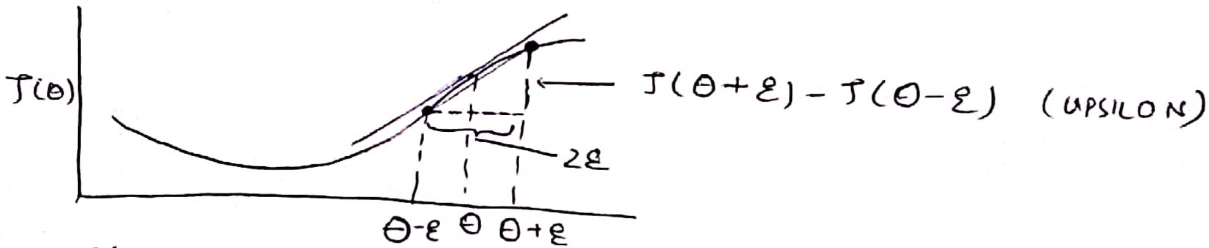
→ from thetaVec, get $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$ (reshape)

→ 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.

Gradient Checking

Gradient checking will assure that backpropagation works as intended



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

[ε should be 10^{-4}]

Octave code

```
epsilon = 1e-4;
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;
```

- Implement backprop to compute DVec (unrolled $D^{(1)}, D^{(2)}, D^{(3)}$)
- Implement numerical gradient ~~to~~ check to compute gradApprox
- Make sure they give similar values
- Turn off gradient checking. use backprop
- Be sure to disable otherwise code will be slow.

andom Initialization

Initializing all theta weights to zero does not work with the neural networks. When we backpropagate, all nodes will update to same value repeatedly. Instead we can randomly initialize our weights for our Θ matrices using the given method

(Symmetry Breaking)

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

(i.e. $-\epsilon \leq \Theta_{ij}^{(l)} \leq \epsilon$)

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

ϵ here is irrelevant to gradient checking.

Putting Together

Pick Network Architecture

No of Input units: Dimension of features

No of Output units: No of classes.

reasonable default: 1 hidden layer

same no. of hidden layer units, if h.l > 1

(usually more the better, expensive tho)

Training a Neural Network

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

for i = 1:m{

Perform fpropagation and backprop using example $(x^{(i)}, y^{(i)})$

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

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

5.) use gradient checking to compare $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$. computed using backprop .vs. using numerical estimate

Then disable gradient checking code.

6.) use gradient descent or advanced optimization method with backpropagation to try to min $(J(\Theta))$ as fun of Θ.