WEEK-5

Neural Networks: Representation & Learning — Week 5:

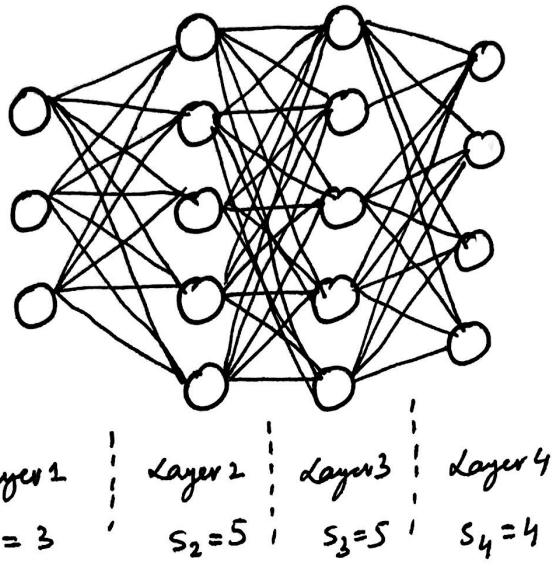
Cost function & Back Propagation

- fitting the parameters for a training set.

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

L = total no. of layers in network

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



$L = 4$

① Binary classification

$y \in \{0, 1\}$

$\Rightarrow 0 \rightarrow h_\theta(x)$

1 o/p unit

$s_L = 1$

$K = 1$

② Multi-class classification $K \geq 3$

$y \in \mathbb{R}^K$. $\begin{bmatrix} \\ \\ \end{bmatrix}_{K \times 1}$

K output units.

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

$s_L = K$

Cost- function for Neural Network :

→ Logistic regression

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

→ Neural Network :

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

K-dimensional vector

$$\underset{NN}{J(\theta)} = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{K=1}^{K} y_K^i \log(h_\theta(x^i))_K + (1-y_K^i)\log(1-(h_\theta(x^i))_K\right.$$

$$\left. + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{S_l}\sum_{j=1}^{S_{l+1}}\left(\theta_{ji}^{l}\right)^2\right]$$

Back Propagation Algorithm

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

Need to compute → ① $J(\theta)$

→ ② $\frac{\partial J(\theta)}{\partial \theta_{ij}^{l}}$

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

⇒ Basically partialy derivative of all θ

Ex: Gradient computation: one training example (x, y)

$L = 4$

⇒ 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)

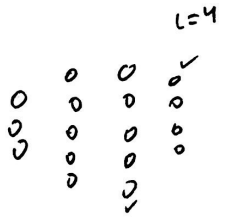
$z^4 = \theta^3 (a^3)$

$a^4 = g(z^4)$

Intuition: δ_j^l = "error" of node j in layer l

⇒ for each output unit ($L=4$)

$\delta_j^4 = a_j^4 - y_j$ → hypothesis $(h_\theta(x))_j$



⇒ $\delta^{(3)} = (\theta^{(3)})^T \delta^{(4)} .* g'(z^{(3)})$ → $a^{(3)} .* (1 - a^{(3)})$

⇒ $\delta^{(2)} = (\theta^{(2)})^T \delta^{(3)} .* g'(z^{(2)})$ → $a^{(2)} .* (1 - a^{(2)})$

$$\boxed{\frac{\partial J(\theta)}{\partial \theta_{ij}^{(l)}} = a_j^{(l)} \delta_i^{(l+1)}} \Rightarrow \text{(ignoring } \lambda; \text{ if } \lambda = 0)$$

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

Back propagation Algorithm

① Set $\Delta_{ij}^{(l)} = 0 \quad \forall\, l, i, j$

will be used to compute $\frac{\partial}{\partial \theta_{ij}^{(l)}} J(\theta)$

loop:

② for $i = 1$ to m,

→ set $a^1 = x^i$

→ perform forward propagation to compute $a^{(l)}$ for $l = 2, 3, \dots L$

→ using y^i, compute $\delta^L = a^L - y^i$

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

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

end loop:

③
$$\begin{cases} D_{ij}^{(l)} = \frac{1}{m} \Delta_{ij}^{l} + \frac{\lambda}{m} \Theta_{ij}^{(l)} & \text{if } j \neq 0 \\ D_{ij}^{l} = \frac{1}{m} \Delta_{ij}^{l} & \text{if } j = 0 \end{cases} \qquad \frac{\partial J(\theta)}{\partial \theta_{ij}^{l}} = D_{ij}^{l}$$

Backpropagation Intuition

→ what is backpropagation doing?

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

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

→ focussing on a single example x^i, y^i, the case of 1 output unit ($\& \lambda = 0$) ($\& K = 1$)

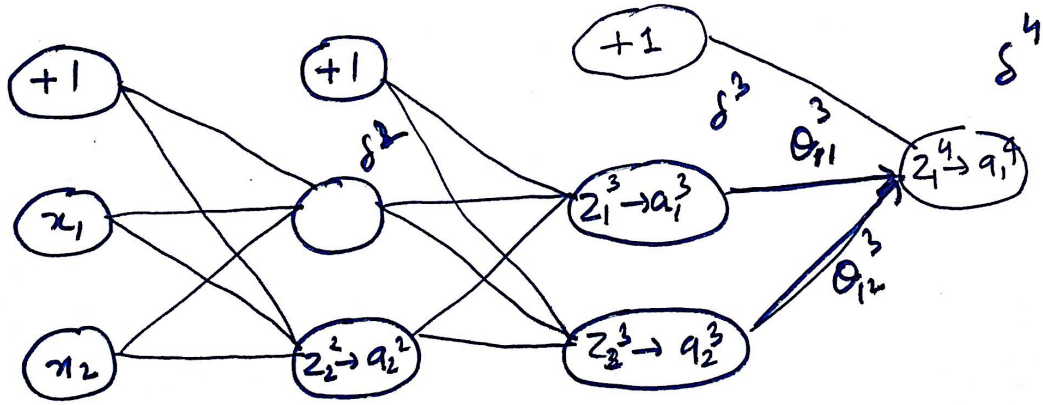
→ δ_j^l = "error" of cost for a_j^l (unit j in layer l).

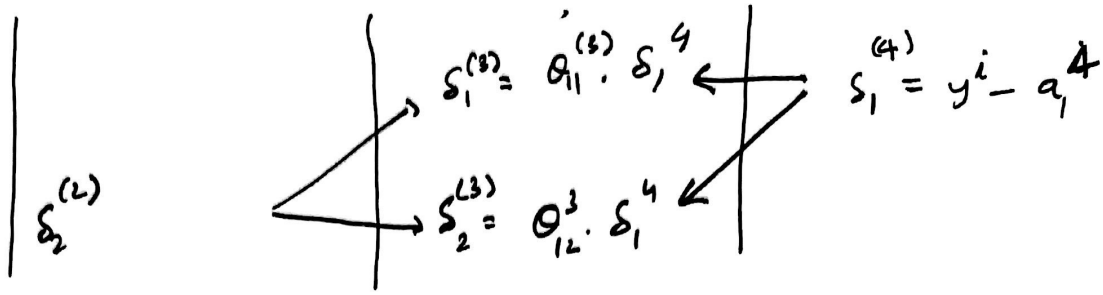
[formally, $\delta_j^l = \frac{\partial}{\partial z_j^l}$ cost(i) for $j \geq 0$]

↓

calculating the error, so it can change the (z)/activation unit value

→





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

Implementation Note: Unrolling parameters.

→ when using advanced optimization, fminunc uses initial theta as vector

function [jval, grad] = costFunction([theta])

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

...

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 vectors"

Example:

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

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

$D^1 \in \mathbb{R}^{10\times 11}$

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

This sets and unrolls all matrices into a long vector.

# Theta1 = reshape(thetaVec(1:110), 10, 11);
Theta2 = reshape(thetaVec(111:220), 10, 11);
Theta3 = reshape(thetaVec(121:330), 10, 11);

Learning Algorithm:

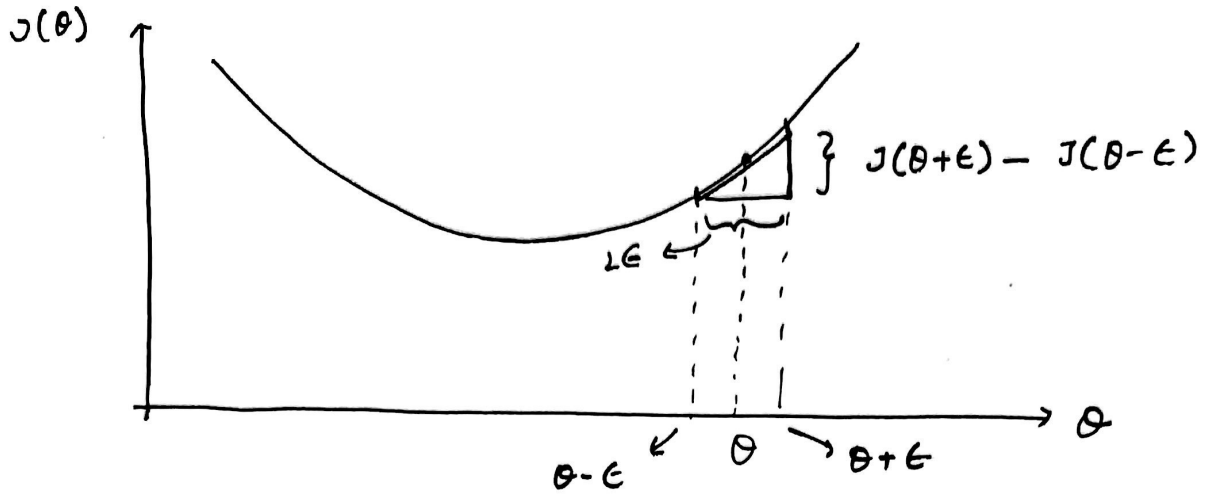
↳ ① Have initial parameters $\Theta^1, \Theta^2, \Theta^3$
② unroll to get ~~initial parameters~~ initialTheta
③ fminunc(@costfn, initialTheta, options)

↳ function [jval, gradientVec] = costfunction(thetaVec)

↳ from thetaVec get $\Theta^1, \Theta^2, \Theta^3$

↳ Use forward prop / back prop to compute D1, D2, D3. and J(Θ)

Gradient checking



$$\therefore \frac{\partial J(\theta)}{\partial \theta} = \frac{J(\theta+\epsilon) - J(\theta-\epsilon)}{2\epsilon}$$ } gradient / slope (Two-sided difference)

$\epsilon \approx 10^{-4}$ } set epsilon

Implement in Octave : gradApprox = (J(theta + Approx) − J(theta − Approx)) / (2 * Approx)

Parameter vector $\theta = [\theta_1, \theta_2, \theta_3 \dots \theta_n] \in \mathbb{R}^n$

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

⋮

```
for i = 1:n,
thetaminus , thetaplus = theta ;
        thetaPlus(i) = thetaplus(i) + ε
        thetaminus(i) = thetaminus − ε
        gradapprox(i) = (J(thetaplus(i) − J(thetaminus(i))
                        / 2ε
end;
```

Now, check

$$\text{gradApprox}(i) \approx \text{DVec}$$

↓ from backpropagation

●● Impletation Note:

① Implement backprop to compute DVec (unrolled D1, D2, D3)

② Implement numerical gradient check to compute gradApprox.

③ Make sure they give similar values

** ④ Turn off gradient checking. Using backprop code for learning.

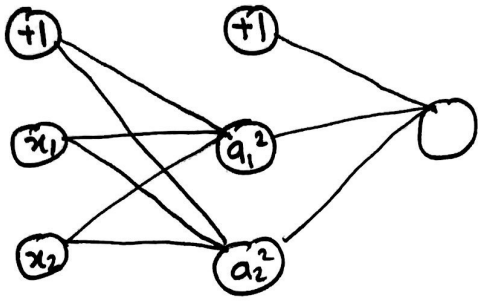
Because: If you run numerical gradient computation on every iteration of gradient descent, your code will be very slow.

Random Initialization: initial value of θ

[optTheta = fminunc(@costfunction, initialtheta, options)]

* if initialtheta = zeros(n, 1) ↘ works well for logistic regression]
  ↘ does not work well in Neural Network] *

Case 1: zero initialization



if $\theta_{ij}^{1} = 0$

$a_1^2 = a_2^2$, Also, $\delta_1^2 = \delta_2^2$

$\theta_{01}^{(1)} = \theta_{02}^{(1)}$

$\frac{\partial J(\theta)}{\partial \theta_{01}^{(1)}} = \frac{\partial J(\theta)}{\partial \theta_{02}^{(1)}}$

*** if activation units are identical, then it results in redundantly complex network producing same o/p

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

Case 2: Random initialization : [Symmetry Breaking] ***

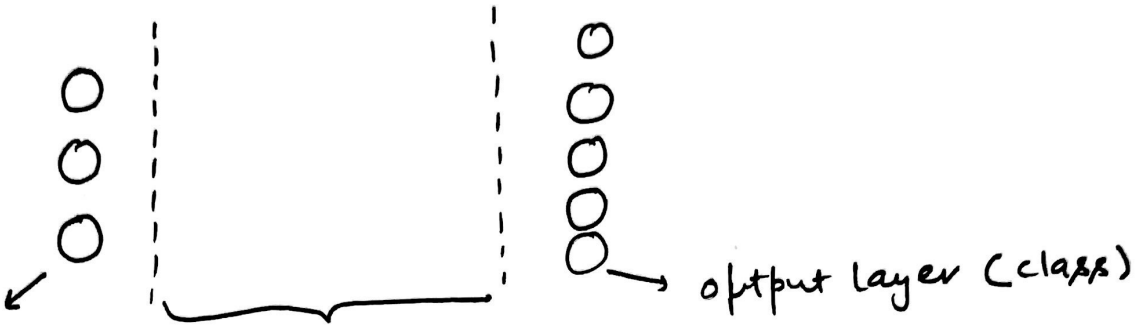
→ initialize each θ_{ij}^{l} to a random value in $[-\epsilon, \epsilon]$

{ eg. Theta 1 = rand(10, 11) * (2 * Init_Epsilon) − init_epsilon }

Summary: Putting it together

→ Training a Neural Network

① Pick a Network architectures (connectivity pattern between neurons)



Reasonable default:

1 hidden layer / > 1 hidden layer, have same no. of hidden units. in every layer [(usually the more the better) → computationally expensive]

→ Steps:

1. Randomly initialize weights.
2. Implement FP to get $h_\Theta(x)^i$ for any x^i
3. Implement code to compute cost fn. $J(\Theta)$
4. Implement Back-propagation (BP) to compute partial derivaties $\frac{\partial J(\Theta)}{\partial \Theta^l_{ij}}$
5. Use gradient checking to compare $\frac{\partial}{\partial \Theta^l_{jk}} J(\Theta)$ computed using BP vs. using numerical estimate
6. Use Grad descent or advanced optimization method with BP to try to minimize $J(\Theta)$ as a fn. of parameters Θ