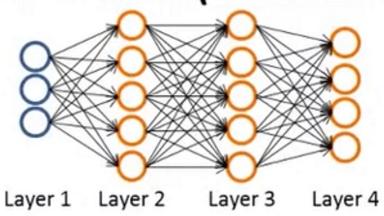
Week 5

- ## Cost Function and Backpropagation:

a. 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 $L=4$

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

$s_1=3 \quad s_2=5 \quad s_3=5 \quad s_4=s_L=4.$

Binary classification

$y=0$ or 1. $h_\Theta(x) \in \mathbb{R}$

1 output layer. $s_L=1$

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.

$h_\Theta(x) \in \mathbb{R}^K \quad s_L=K \ (K \geq 3)$.

Neural network:

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

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

sum of K outputs.

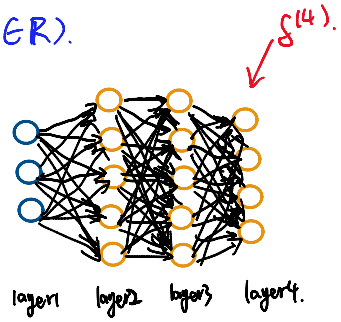
$$+\frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{ji}^{(l)})^2$$

don't sum $i=0$.

layer $1 \to$ layer $L-1$.

b. Backpropagation Algorithm:

We need to compute: $J(\Theta)$, $\frac{\partial}{\partial w_{ij}^{(l)}} J(\Theta)$. ($w_{ij} \in \mathbb{R}$).



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

$a_j^{(l)}$ = "activation" of node j in layer l.

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

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

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

element-wise.

$g'(z^{(3)}) \to a^{(3)} .* (1-a^{(3)})$.

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

$g'(z^{(2)}) \to a^{(2)} .* (1-a^{(2)})$.

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

$$\frac{\partial}{\partial w_{ij}^{(l)}} J(\Theta) = a_j^{(l)} \delta_i^{(l+1)}$$ (ignoring λ or $\lambda=0$).

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

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

$\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$ ← $\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} (a^{(l)})^T$.

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

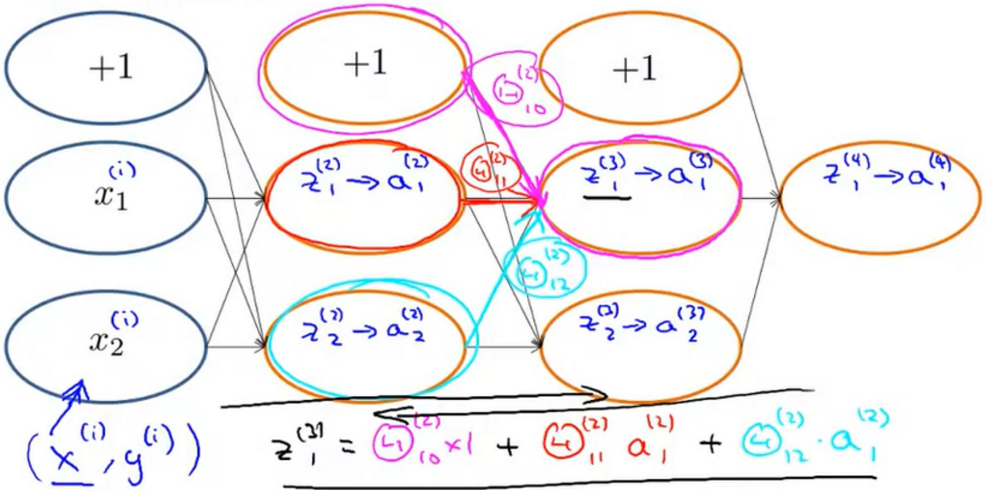
$D_{ij}^{(l)} := \frac{1}{m} \Delta_{ij}^{(l)}$ if $j = 0$

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

c. Backpropagation Intuition:

Look at the Forward propogation first.

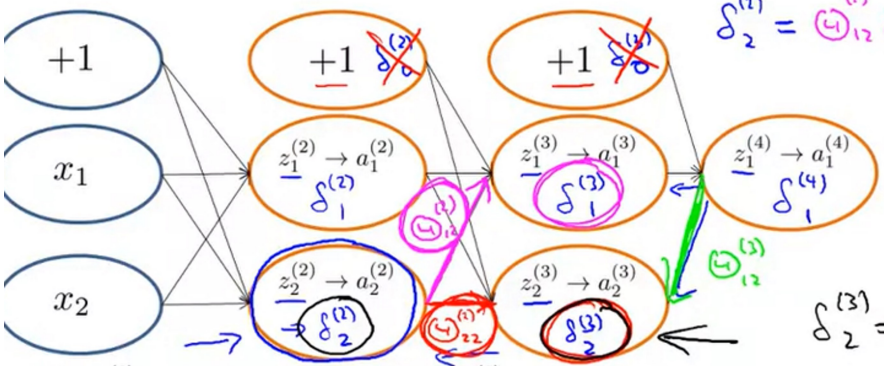
Forward Propagation



Forward Propagation

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

$\delta_2^{(2)} = \Theta_{12}^{(1)} \delta_1^{(3)} + \Theta_{22}^{(2)} \cdot \delta_2^{(3)}$



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

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

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

- Backpropagation in Practice:
  a. Implementation Note: Unrolling Parameters:

Advanced optimization

```
function [jVal, gradient] = costFunction(theta)
  ...
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`)

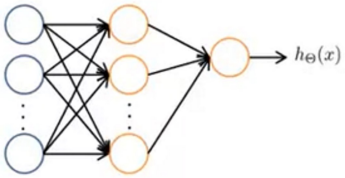
"Unroll" into vectors

Example

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

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

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




```
→ 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 $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$.

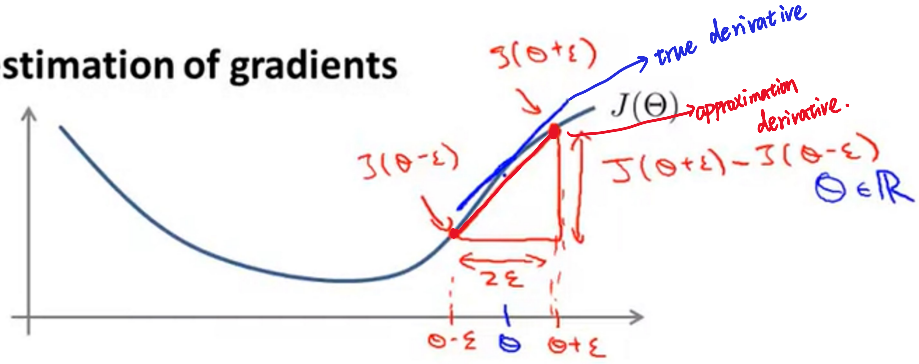
→ Use forward prop/back prop to compute $D^{(1)}, D^{(2)}, D^{(3)}$ and $J(\Theta)$.

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

  b. Gradient Checking:

Through gradient descent, it looks like J of theta is decreasing, but you might wind up with a neural network that has a higher level of error.

Numerical estimation of gradients



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

$\varepsilon = 10^{-4}$ { too small : difficult to compute. too big : bad result.

vector θ: $\theta \in \mathbb{R}^n$

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

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

$$\vdots$$

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

Check approximation $\approx$ DVec → from backpropogation.

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.

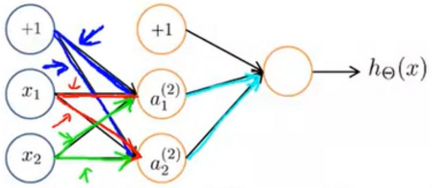
very slow. (→ DVec $\delta^{(4)}, \delta^{(3)}, \delta^{(2)}$ efficient.

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 very slow.

c. Random Initialization:

Zero initialization : if we initial with 0



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

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

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

After each update, parameters corresponding to

inputs going into each of two hidden units are identical.

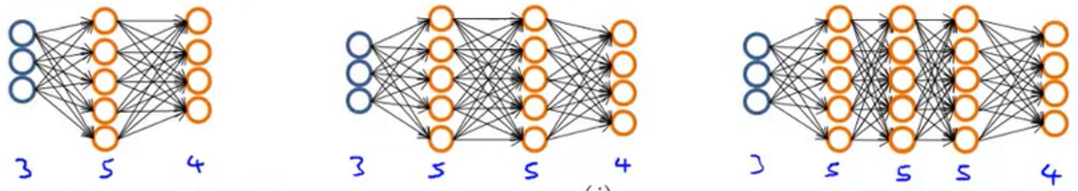
Random initialization:Symmetry breaking

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

i.e. $-\varepsilon \le \Theta_{ij}^{(l)} \le \varepsilon$.

d. Putting It Together:

① Pick a suitable neural network:



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)

$y \in \{1, 2, 3, \dots, 10\}$ ~~$y = 5$~~ transfor y into one-hot vector. $y = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$ or $\begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$ $\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$ ←

② randomly initialize weights

③ implement forwardpropogation to get $h_\Theta(x^{(i)})$ for any $x^{(i)}$.

④ implement code to compute cost function $J(\Theta)$.

⑤ implement backpropogation to compute partial derivatives $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$

⑥ use gradient checking to compare $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$ computed using backpropogation vs. using estimate of gradient of $J(\Theta)$.

⑦ use gradient descent or advanced optimization method with backpropogation to try to minimize $J(\Theta)$ as a function of parameters Θ.