ML

Andrew Ng

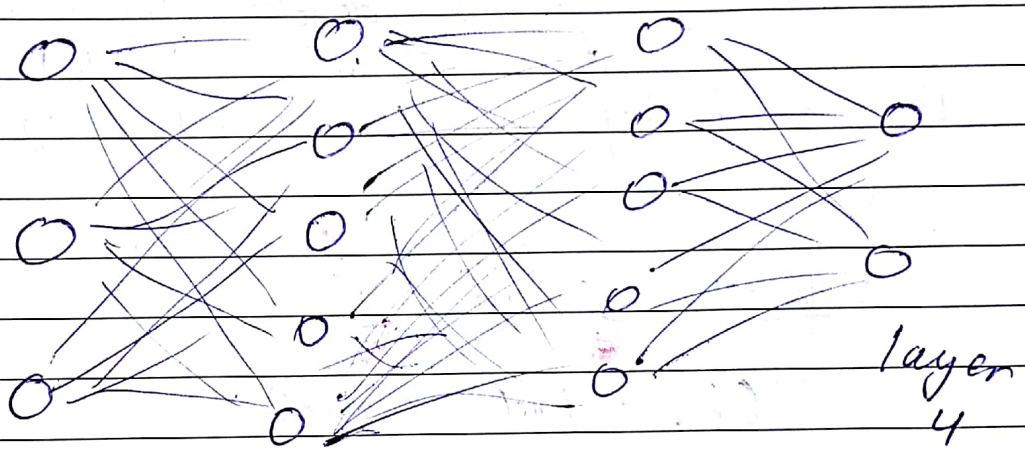
Coursera

Week 5

Date:

Cost function

The focus of neural networks will be predominantly on classification problems



Expressing the entire data set:

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

L = total no. of layers in a network
(in this case $L=4$)

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

Date:

in this case $S_1 = 3$, $S_2 = 5$, $S_3 = 5$, $S_4 = 2$

Binary classification

$y = 0$, $y = 1$ (1 output unit)

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

$S_L = 1$ or $\boxed{k = 1}$ ← k denotes the number of nodes (units) in the output layer

Multi-class classification

(k distinct classes)

$y \in \mathbb{R}^k$ $S_L = k$

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

$$y = h_\theta(x) = \begin{bmatrix}1\\0\\0\\0\end{bmatrix} \text{ or } \begin{bmatrix}0\\1\\0\\0\end{bmatrix} \text{ or } \begin{bmatrix}0\\0\\1\\0\end{bmatrix} \text{ or } \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$$

Date:

Neural Network cost function:

$h_\Theta(x) \in \mathbb{R}^K$, $(h_\Theta(x))_i$ = ith output

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

$$\left. + (1-y_k^{(i)}) \log (1-(h_\Theta(x^{(i)}))_k)\right] +$$

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

cost of a multi-class classification neural network

Explanation:

$$X = \begin{pmatrix} x_1^{(1)} & x_2^{(1)} & x_3^{(1)} & \cdots & x_n^{(1)} \\ x_1^{(2)} & x_2^{(2)} & x_3^{(2)} & \cdots & x_n^{(2)} \\ \vdots & & & & \\ x_1^{(m)} & x_2^{(m)} & x_3^{(m)} & \cdots & x_n^{(m)} \end{pmatrix}$$

Date:

$$Y = \begin{bmatrix} y_1^{(1)} & y_2^{(1)} & \cdots & y_{S_L}^{(1)} \\ y_1^{(2)} & y_2^{(2)} & \cdots & y_{S_L}^{(2)} \\ \vdots & & & \\ y_1^{(m)} & y_2^{(m)} & \cdots & y_{S_L}^{(m)} \end{bmatrix}$$

When outputting the prediction:

$$\begin{bmatrix} h(x^{(1)})_1 & h(x^{(1)})_2 & \cdots & h(x^{(1)})_{S_L} \\ h(x^{(2)})_2 & h(x^{(2)})_2 & \cdots & h(x^{(2)})_{S_L} \\ \vdots & & & \\ h(x^{(m)})_1 & h(x^{(m)})_2 & \cdots & h(x^{(m)})_{S_L} \end{bmatrix}$$

Thus:

$$\sum_{i=1}^{m} \sum_{k=1}^{K} y_k^{(i)} \log\left(h_\theta(x^{(i)})\right)_k$$

means to go through every row of the dataset & cycle through all the outputs (with k) & apply $\log(h_\theta(x^{(i)}))_k$ to the value $y_k^{(i)}$ that is one (cost function for $y = 1$)

Date:

Since there is only one such value then this function is applied once per row.

The remaining 0s in $y^{(i)}$ will then be applied the following function:

$$\sum_{i=1}^{m} \sum_{K=1}^{K} (1 - y_K^{(i)}) \log (1 - (h_\theta(x^{(i)}))_K)$$

(cost function for $y = 0$)

$$\underbrace{\sum_{l=1}^{L-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} (\theta_{ji}^{(l)})^2}_{\text{regularization}}$$

simply implies to use every $\theta_{ji}^{(l)}$ value possible for each layer in the neural network

Note that the values of i & j start at 1, the bias is not included within regularization

Backpropagation algo

Date:

We wish to find $\min_{\Theta} J(\Theta)$

and for that we need:

- $J(\Theta)$

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

Given that there is only 1 training example : (x, y) then forward propagation is executed as such:

$a^{(1)} = x$ ← bias included

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

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

add $a_0^{(2)}$

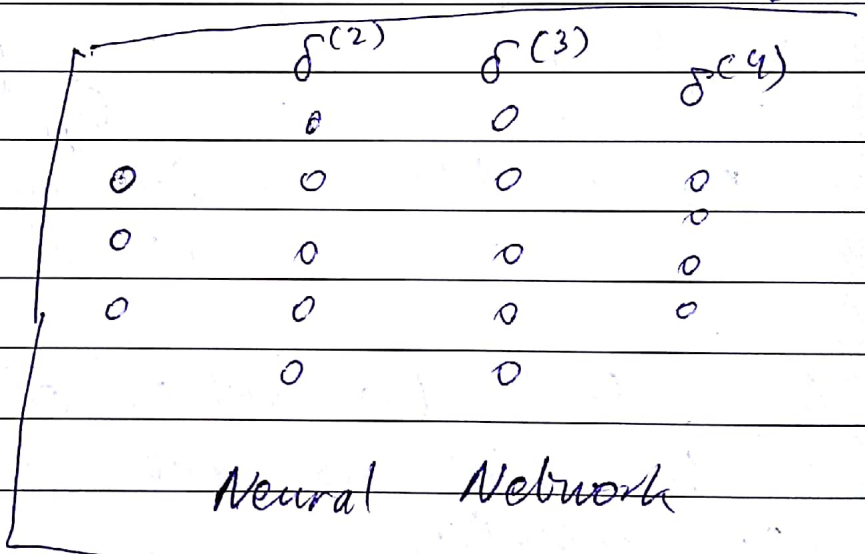
$z^{(3)} = \Theta^{(2)} a^{(2)}$

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

add $a_0^{(3)}$

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

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



Backpropagation with 1 training example

Date:

To compute the derivatives we will use the backpropagation algo.

Intuition: compute $\delta_j^{(l)}$ = error in $a_j^{(l)}$ or error in node j in layer l

Consider this neural network example,

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

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

Vectorised implementation:

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

$\delta^{(4)} \in \mathbb{R}^{s_L}$, $a^{(4)} \in \mathbb{R}^{s_L}$, $y \in \mathbb{R}^{s_L}$

- $\delta^{(3)} = (\Theta^{(3)})^T \delta^{(4)} .* g'(z^{(3)})$
- $\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} .* g'(z^{(2)})$

No $\delta^{(1)}$ as there can be no error associated with inputs (we don't want to change them)

Date:

It can be proved that:

$$g'(z^{(3)}) = a^{(3)} .* (1 - a^{(3)})$$
$$g'(z^{(2)}) = a^{(2)} .* (1 - a^{(2)})$$

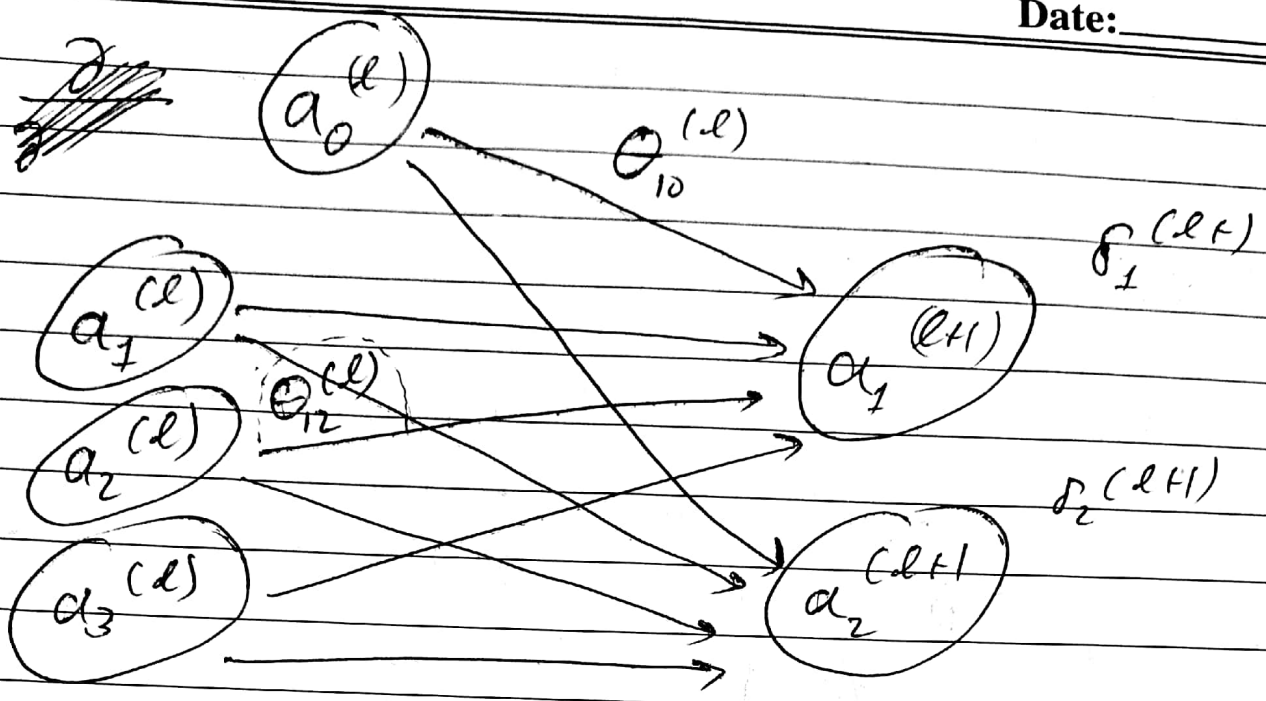
This process is called back propagation because we compute the error of the output layer first ($\delta^{(4)}$ in this case) & then we keep taking steps backwards to compute $\delta^{(3)}$ then $\delta^{(2)}$ & so on

Via complicated mathematical proof it is possible to prove (with the help of $\delta^{(4)}$, $\delta^{(3)}$ & etc) that:

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

(ignoring the regularization term i.e. $\lambda = 0$)

Date:



$$\frac{\partial}{\partial \theta_{12}^{(l)}} J(\Theta) = a_2^{(l)} \delta_1^{(l+1)}$$

$$\frac{\partial}{\partial \theta_{10}^{(l)}} J(\Theta) = a_0^{(l)} \delta_1^{(l+1)} = \delta_1^{(l+1)}$$

1

Note that $\theta_{0j}^{(l)}$ does not exist

thus this is not possible :

$$\frac{\partial}{\partial \theta_{0j}^{(l)}} J(\Theta) = a_j^{(l)} \delta_0^{(l+1)}$$

No connection from $a_j^{(l)}$ to $\delta_0^{(l+1)}$

Date:

Backpropagation algorithm with multiple data points in training set:
$\{(x^{(1)}, y^{(1)}), \ldots (x^{(m)}, y^{(m)})\}$

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

→ This will be used to compute $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$

→ capital δ (pronounced delta)

For $i = 1$ to m:

- ~~Set $a^{(1)} = x^{(i)}$~~
- 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 $\delta^{(L-1)}, \delta^{(L-2)}, \ldots \delta^{(2)}$

  → backpropagation & $\delta^{(1)}$ are non-existant as we do not wish to find the error on the inputs

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

~~1 step~~

on each instance of input, every $\Delta_{ij}^{(l)}$ is updated

1 update per cycle for unique i, j, l & for every unique i, j, l the values are accumulated from i to m

different i's

$\Delta_{ij}^{(l)}$ can not exist as $\theta_{ij}^{(l)}$ is not calculated, ~~bias already has the value of 1~~

Vectorised form of equation:

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

$\Delta^{(l)}$: $s_{l+1} \times (s_l + 1)$

$\delta^{(l+1)}$: $(s_{l+1} \times 1)$

$(a^{(l)})^T$: $(1 \times s_l + 1)$

$s_{l+1} \times (s_l + 1)$

This is the case because of how Θ is :

$$\Theta = \begin{bmatrix} \theta_{10} & \theta_{12} \dots \\ \theta_{20} & \theta_{21} \dots \\ \theta_{30} & \theta_{31} \dots \end{bmatrix}$$

$$\begin{bmatrix} \delta_1^{(l+1)} \\ \vdots \\ \delta_{s_{l+1}}^{(l+1)} \end{bmatrix} \begin{bmatrix} a_0^{(l)} & a_1^{(l)} & \dots & a_{s_l}^{(l)} \end{bmatrix}$$

possible because:

$\frac{\partial}{\partial \theta_{10}^{(l)}} = a_0^{(l)} \delta_1^{(l+1)}$ is possible

Victory

Page No.

~~clear~~ apply average & add regularization

Date:

After the loop, execute:

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

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

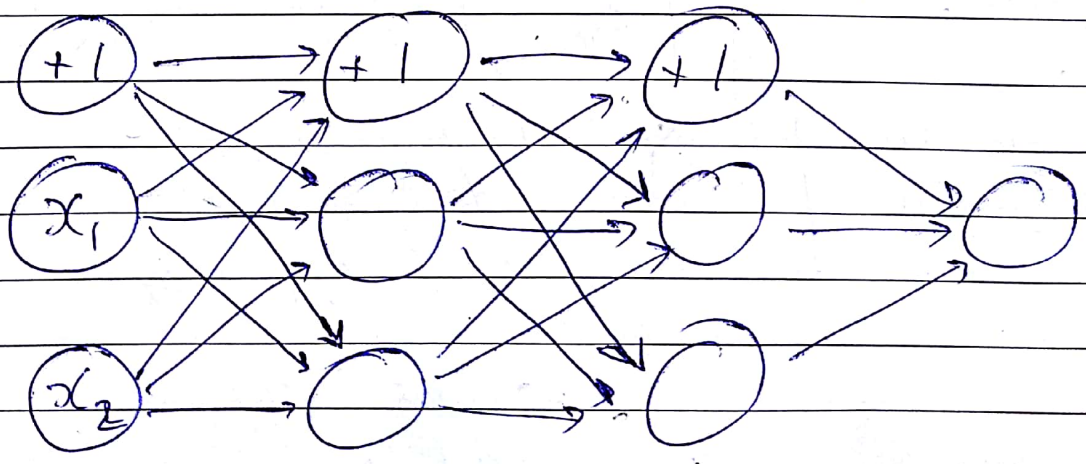
No regularization in bias term

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

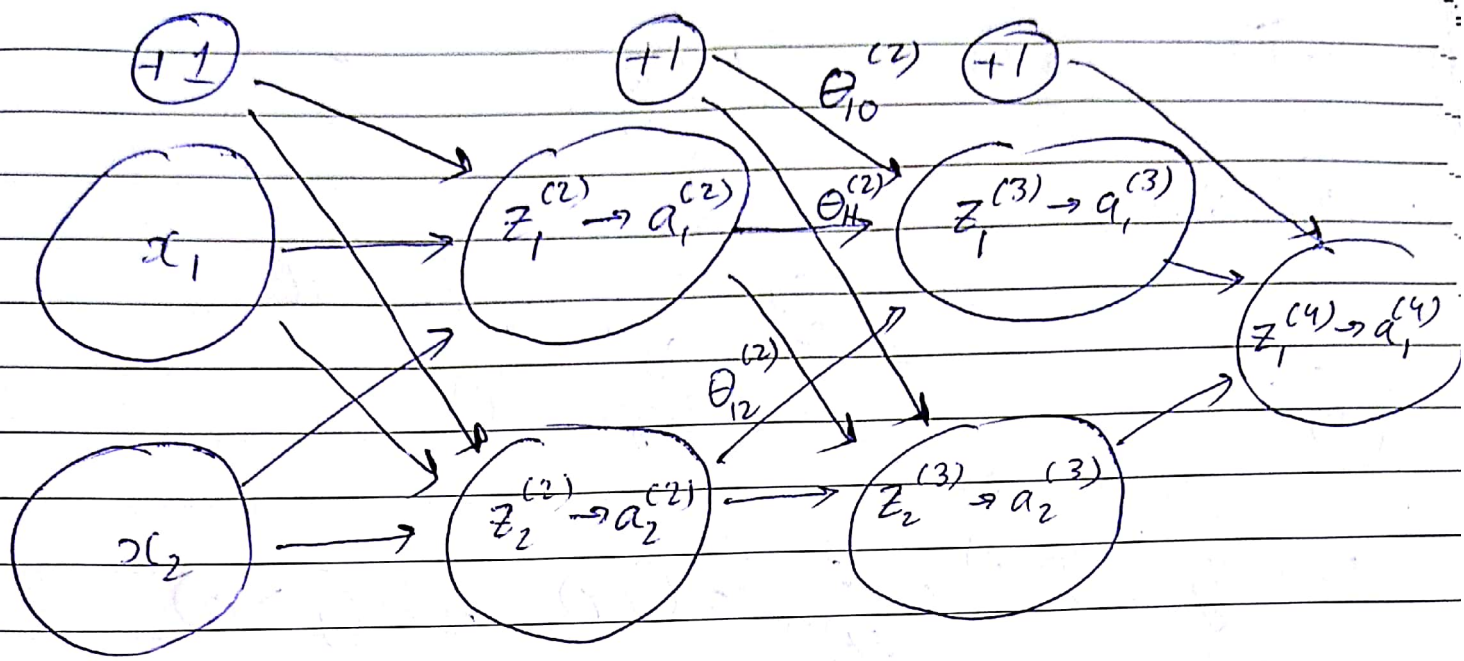
→ with complex mathematical proof this can be proved

Back propagation Intuition

Use the following example to illustrate intuition



Date:



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

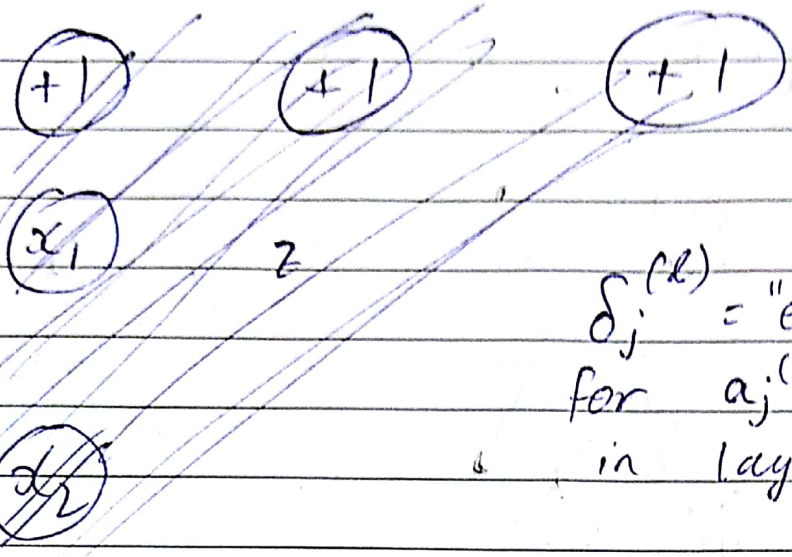
→ Computations flow in this direction for forward propagation

← Computations flow in this direction for backward propagation

~~Backward~~ Back propagation can be considered the inverse operation of forward propagation

Page No.

Date:



$\delta_j^{(l)}$ = "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:

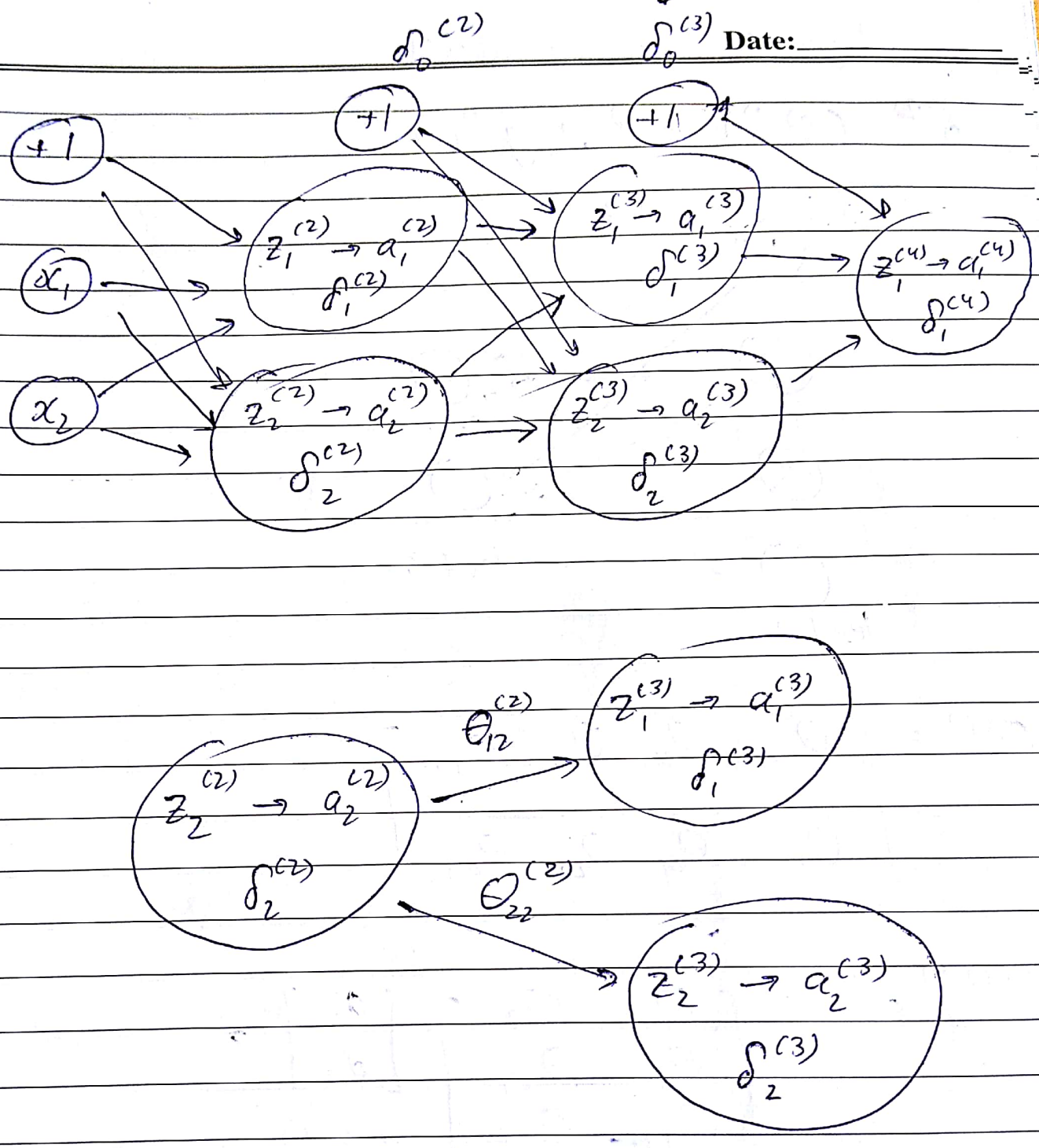
$$\text{cost}(i) = y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log h_\theta(x^{(i)})$$

$\frac{\partial \text{cost}(i)}{\partial z_j^{(l)}}$ → what is the minute change in cost if a minute change in $z_j^{(l)}$ occurs. A minute change in $z_j^{(l)}$ causes for the $h_\theta(x^{(i)})$ to change in value

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

{ not needed – calculated & discarded

Date:

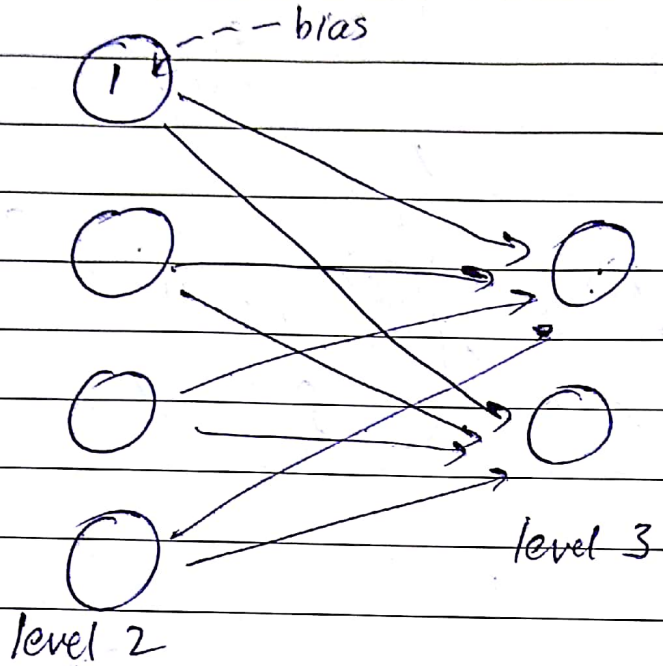


$$\delta_2^{(2)} = \theta_{12}^{(2)} \delta_1^{(3)} + \theta_{22}^{(2)} \delta_2^{(3)}$$

How $\delta_0^{(2)}$ are calculated :

Date:

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



$$\Theta^{(2)} = \begin{bmatrix} \Theta_{10} & \Theta_{11} & \Theta_{12} & \Theta_{13} \\ \Theta_{20} & \Theta_{21} & \Theta_{22} & \Theta_{23} \end{bmatrix}$$

$$(\Theta^{(2)})^T \delta^{(3)} = \begin{bmatrix} \Theta_{10} & \Theta_{20} \\ \Theta_{11} & \Theta_{21} \\ \Theta_{12} & \Theta_{22} \\ \Theta_{13} & \Theta_{23} \end{bmatrix} \begin{bmatrix} \delta_1 \\ \delta_2 \end{bmatrix} =$$

(4×2) $\quad$ (2×1)

Date:

$$= \begin{bmatrix} \Theta_{10}\delta_4 + \Theta_{20}\delta_3 \\ \Theta_{11}\delta_4 + \Theta_{21}\delta_3 \\ \Theta_{12}\delta_4 + \Theta_{22}\delta_3 \\ \Theta_{13}\delta_4 + \Theta_{23}\delta_3 \end{bmatrix}$$

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

does not exist

$$= \begin{bmatrix} \Theta_{10}\delta_4 + \Theta_{20}\delta_3 \\ \Theta_{11}\delta_4 + \Theta_{21}\delta_3 \\ \Theta_{12}\delta_4 + \Theta_{22}\delta_3 \\ \Theta_{13}\delta_4 + \Theta_{23}\delta_3 \end{bmatrix} .* g \begin{bmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \end{bmatrix} = \begin{bmatrix} \delta_0^{(2)} \\ \delta_1^{(2)} \\ \delta_2^{(2)} \\ \delta_3^{(2)} \end{bmatrix}$$

Unrolling parameters

Changing matrices to vectors

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

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

default behaviour is that these are vectors

Date:

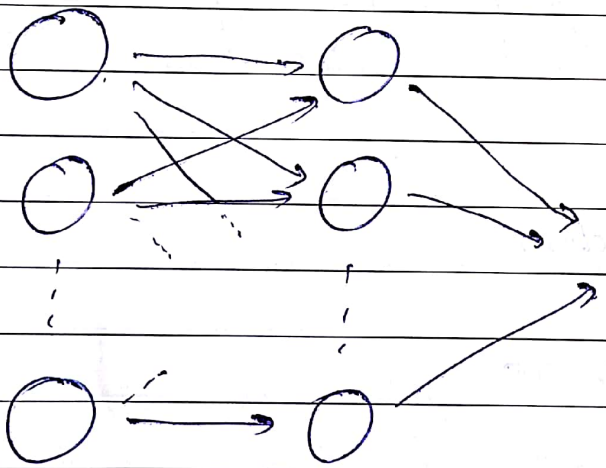
However with a 4 layer Neural Network this is the case:

- $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$ → matrices
- $D^{(1)}, D^{(2)}, D^{(3)}$ → matrices

We must "unroll" these matrices into vectors so that they are in a format suitable for costFunction & fminunc

Example:

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



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

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

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

Seems like mistake is made here: $\Theta^{(3)}$ will not exist

Date:

In octave:

thetaVec = [Theta1(:); Theta2(:); Theta3(:)];

unroll a matrix into a vector

places all the unrolled fragments together

DVec = [D1(:); D2(:); D3(:)];

Going back from vector to matrix representations

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

reshape into a 1×11 matrix

Final learning algorithm

- Have initial parameters $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$
- Unroll to get initialTheta to pass to

fminunc(@costFunction, initialTheta, options)

Date:

function [jVal, gradientVec] = costFunction (thetaVec)

→ From thetaVec, get $\Theta^{(1)}$, $\Theta^{(2)}$, $\Theta^{(3)}$ via reshape
→ use forward prop/back prop to compute $D^{(1)}$, $D^{(2)}$, $D^{(3)}$ & $J(\Theta)$
→ Unroll $D^{(1)}$, $D^{(2)}$, $D^{(3)}$ to get gradientVec

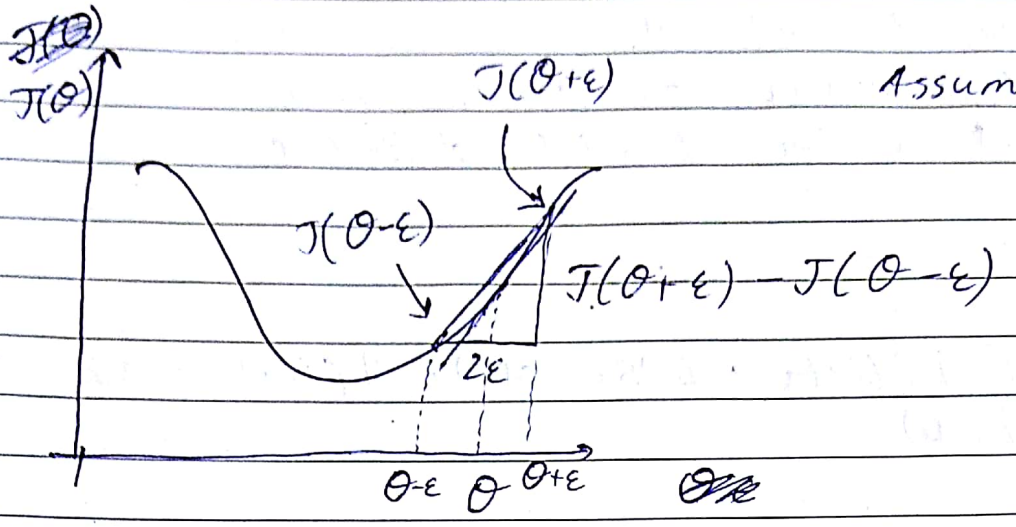
Gradient checking

Back propagation is a complex algorithm & hence subtle bugs can make their way into the ~~int~~ implementation. The cost might be decreasing on every iteration of an optimisation with the bug as well but you might end up with a ~~not~~ neural network with a higher level of error if compared to the bug free implementation

↳ gradient checking eliminates this problem
↳ it is smart to implement gradient checking in other algorithms as well

Gradient checking is used to verify that the actual gradient of the cost function is being calculated

Date:



Assume $\theta \in \mathbb{R}$

We can find an approximation to $\frac{d}{d\theta} J(\theta)$ by connecting a line between $J(\theta+\varepsilon)$ & $J(\theta-\varepsilon)$ & finding out its gradient

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

ε is usually a small value $\approx 10^{-4}$

2-sided difference estimate:

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

1-sided difference estimate:

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

Date:

We use the 2-sided difference for our gradient approximation as it is more accurate than the 1-sided difference

In Octave:

```
gradAprox = (J(theta + EPSILON) - J(theta-EPSILON))
/(2 * EPSILON)
```

Previous example considered a single value θ or $\theta \in \mathbb{R}$. For $\theta \in \mathbb{R}^n$ (θ can be unrolled version of $\theta^{(1)}, \theta^{(2)}, \theta^{(3)}$)

$$\theta = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_n \end{bmatrix}$$

$$\frac{\partial}{\partial \theta_1} J(\theta) \approx \frac{J(\theta_1 + \varepsilon_1, \theta_2, \theta_3 \dots \theta_n) - J(\theta_1 - \varepsilon, \theta_2, \dots, \theta_n)}{2\varepsilon}$$

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

& etc

Date:

In Octave :

for i = 1:n :
① thetaPlus = theta
② thetaPlus(i) = thetaPlus(i) + EPSILON
③ thetaMinus = theta;
④ thetaMinus(i) = thetaMinus(i) - EPSILON
⑤ gradApprox(i) = (J(thetaPlus) - J(thetaMinus)) / (2 * EPSILON)
end;

①-②:
Copy the theta vector & add epsilon to the element of the current iteration:

ThetaPlus is refreshed with theta on each iteration so previous epsilon is removed

$$\begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_i + \varepsilon \\ \vdots \\ \theta_n \end{bmatrix}$$

③-④ : Copy the theta vector & subtract epsilon from the element of the current iteration.

$$\begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_i - \varepsilon \\ \vdots \\ \theta_n \end{bmatrix}$$

Victory

Page No

Date:

Once all gradApprox values are computed check that gradApprox ≈ DVec ← (derivatives from back prop)

If the difference between gradApprox & DVec is in decimal places then this increases the confidence that backprop computed correctly.

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
- Use backprop for learning

↳ This is done because computing the gradient with gradiat checking is computationally expensive as compared to using backprop

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

Victory

Page No.

Random Initialization

Initial value of Θ

For gradient descent & advanced optimisation method, need ~~so~~ initial value of Θ

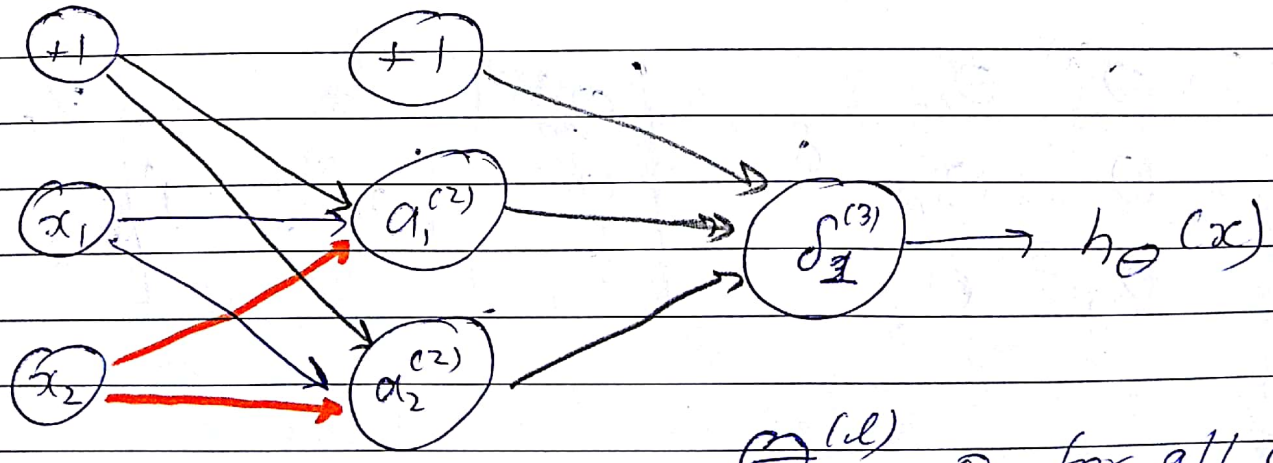
optTheta = ~~fmin~~fminunc(@costFunction, initialTheta, options)

Consider gradient descent

Set initialTheta = zeros(n,1) ?

↳ worked okay in logistic regression but does not work with neural networks

Suppose all weights are initialised to 0 :



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

Date:

All the lines in pencil ~~& lines in pe~~ are the same, all the lines in pen are the same & all in marker are the same hence: $a_1^{(2)} = a_2^{(2)}$

$$\delta^{(l)} = (\Theta^{(l)})^T \delta^{(l+1)} .* g'(z^{(l)})$$

Thus $\delta_1^{(2)} = \delta_2^{(2)}$

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

$$\Theta^{(1)} = \begin{bmatrix} \Theta_{10} & \Theta_{11} & \Theta_{12} \\ \Theta_{20} & \Theta_{21} & \Theta_{22} \end{bmatrix}$$

$$\Theta^{(2)} = \begin{bmatrix} \Theta_{10} & \Theta_{11} & \Theta_{12} \end{bmatrix}$$

$\delta^{(2)} = \Theta^{(2)}$ ~~Θ_{10} Θ_{20} Θ_{11} Θ_{21} Θ_{12} Θ_{22}~~

$$\delta^{(2)} = \begin{bmatrix} \Theta_{10}^{(2)} \\ \Theta_{11}^{(2)} \\ \Theta_{12}^{(2)} \end{bmatrix} \left[\delta_1^{(3)}\right] .* g' \begin{bmatrix} z_0^{(2)} \\ z_1^{(2)} \\ z_2^{(2)} \end{bmatrix}$$

Date:

$$\delta^{(2)} = \begin{bmatrix} \theta_{10}^{(2)} \delta_1^{(3)} \times g''(z_0^{(2)}) \\ \theta_{11}^{(2)} \delta_1^{(3)} \times g'(z_1^{(2)}) \\ \theta_{12}^{(2)} \delta_1^{(3)} \times g'(z_1^{(2)}) \end{bmatrix}$$

all $z^{(2)}$ elements are equal as ~~weights~~ each node in layer 2 has the same exact weights & all elements are equal in $\theta^{(2)}$ as well hence all elements in $\delta^{(2)}$ are equal:

$$\boxed{\delta_1^{(2)} = \delta_2^{(2)}}$$

Thus $\frac{\partial}{\partial \theta_{10}^{(1)}} J(\theta) = \frac{\partial}{\partial \theta_{20}^{(1)}} J(\theta)$

because $\frac{\partial}{\partial \theta_{10}^{(1)}} J(\theta) = a_0^{(1)} \delta_1^{(2)} =$

$= \frac{\partial}{\partial \theta_{20}^{(1)}} J(\theta) = a_0^{(1)} \delta_2^{(2)}$

since $\delta_1^{(2)} = \delta_2^{(2)}$

And when θ_{10} & θ_{20} are updated then they will be equal since initial value, α & derivative are equal ~~for~~ ~~both~~ in θ_{10} & θ_{20}

↳ Thus all the lines of the same colour remain the same value; but will be non-zero

After each update, parameters corresponding to inputs going into each of 2 hidden units are identical

i.e. $a_1^{(2)} = a_2^{(2)}$ since the weights are the same & applied to the same input values after 1 interation

↳ As gradient descent is continued the coloured lines will always remain the same thus each node in ~~the~~ an ~~inp~~ hidden layer computes the exact same function

↳ The layer 3 node sees the same feature from all its inputs

Random initialization is used to get around this problem

has nothing to do with ε in gradient checking

Date:

Random initialization :

Symmetry breaking

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

E.g. → random 10×11 matrix (between 0 & 1)

Theta1 = rand(10, 11) * (2 * INIT_EPSILON) − INIT_EPSILON

↳ in $[-\varepsilon, \varepsilon]$

Symmetry is set not only when all weights are 0 but also when ~~any~~ all weights are equal, we need to break the symmetry by setting random variables

Date:

Putting it together

Training a neural network

Pick a network architecture (connectivity pattern between neurons) - # of ~~hidden~~ layers & # of nodes in these layers

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

No. of output units: Number of classes

$y \in \{1, 2, \ldots 10\}$ where if $y = 1$

then $y = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$ } 10 elements total

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

↳ however having more hidden units makes training more expensive

Date:

Having more nodes in hidden layers than input layers is usually better

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 $i = 1:m$ { iterate over data rows } in $(x^{(i)}, y^{(i)})$:

| Perform forward propagation & backpropagation using example $(x^{(i)}, y^{(i)})$
| (Get activations $a^{(l)}$ & delta terms $\delta^{(l)}$ for $l = 2, \ldots, L$)
|
| $\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} (a^{(l)})^T$

Compute $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$ with regularization & etc

Date:

5) Use gradient checking to compare $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$ computed using backprop

vs using numerical estimate of gradient of $J(\Theta)$

→ Then disable gradient checking code

6) Use gradient ~~checking~~ descent or advanced optimisation method with backpropagation to try to minimize $J(\Theta)$ as a function of parameters Θ

$J(\Theta)$ is non-convex for neural networks & is susceptible to local minima