Model and Cost function

Not's :

m : no. of training ex.s

x's : input var.s / features

y's : output var.s / ~~features~~ target var. that we are trying to predict

(x, y) : one training ex. / observ^n

$(x^{(i)}, y^{(i)})$: i^{th} training ex. / observ^n

$(x^{(i)}, y^{(i)})$; $i = 1, 2, \ldots, m$: training set

i : index into the training set

X : space of I/P val.s

Y : space of O/P val.s

n : no. of features

Hypothesis : $h_\theta(x) = \theta_0 + \theta_1 x$

θ_i's : para. of model

h maps from x's to y's

Neural Networks

Neural Network is a series of algo.s that endeavours to recognize underlying rel'ships in a set of data through a process that mimics the way the human brain operates.

→ NNs were widely used in 80s and 90s; popularity diminished in late 90s

→ Recent resurgence; state-of-art techniques for many applic's

→ NNs are much more expensive in terms of comput°. One of the reason for their resurgence is that the comput° needed is only recently matched by powerful computers built in last decade.

Need : Algo.s like Lin. Reg. & Log Reg. (simple or polynomial) is not a good way to learn complex non-linear hypotheses whe 'n' is large bc it generates too many features.

2500 x 3 if RGB

ex: 50 x 50 pixel image → 2500 pixels

$n = 2500$

$$x = \begin{bmatrix} \text{pixel 1 intensity} \\ \vdots \\ \text{pixel 2500 intensity} \end{bmatrix}$$

grayscale val. b/w 0–255

$O(n^q) \simeq \frac{n^q}{a}$

n → features
a → polynomial

NNs are a better way to learn complex non-linear hypo., even when n is large

Model Represent^

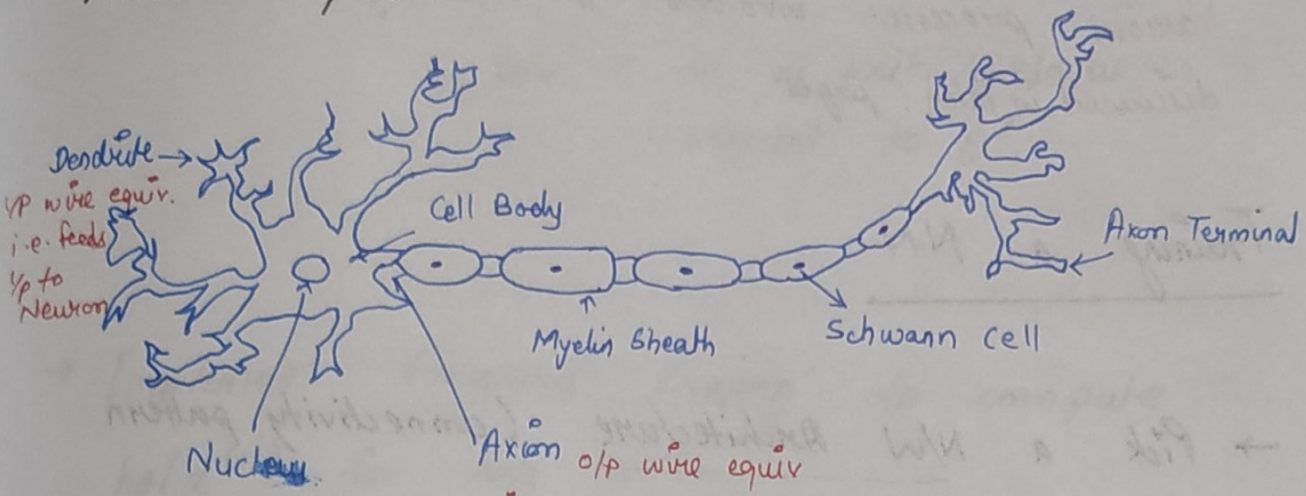


fig. Neuron

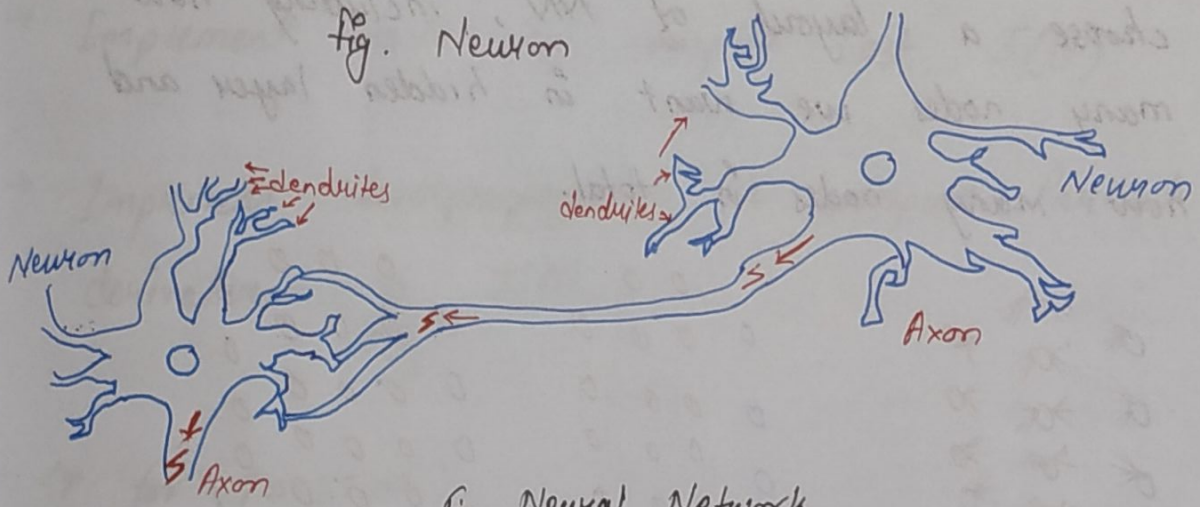


fig. Neural Network

Neurons communicate w/ each other using little pulses of electricity called spikes.

So, a neuron basically sends a spike and the other Neurons basically accept the info., performs some comput^ on it & passes to the next Neuron.

This process is repeated until the desired process is done.

ex: Contrac^ of Muscle.

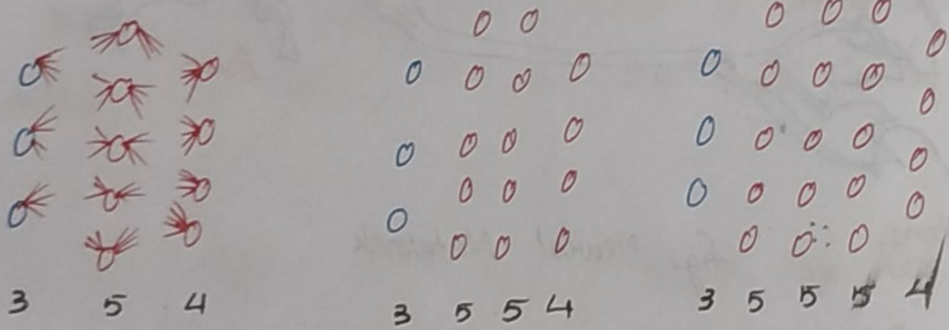
Neural Network Implementⁿ (step-by-step Algo.)

this is an overview to generate an idea of the whole picture.
Various processes involved in this Algo. are discussed in later pages.

Training a NN

→ Pick a N/W Architecture (connectivity pattern b/w nodes)

choose a layout of NN, including how many nodes we want in hidden layers and how many nodes in total.



- no. of i/p units : dimensions of features $x^{(i)}$
- no. of o/p units : no. of classes $y = \{0, 1\}$ or $y = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix}$
- reasonable default : 1 hidden layer, or
  if >1 hidden layer, have same no. of hidden units in every layer (more expensive / accurate)

no. of hidden layers = n * no. of i/p units ; $n \in N$

→ Randomly initialize weights (Θ) in range $[-\epsilon, \epsilon]$

where, $\epsilon = \frac{\sqrt{6}}{\sqrt{L_{in} + L_{out}}}$

L_{in}, L_{out} = no. of units in layers adjacent to Θ^l

→ Implement forward propag^n to compute $h_\theta(x^{(i)})$ for any $x^{(i)}$

→ Implement code to compute Cost $J(\Theta)$

→ Implement Backpropagation to compute Partial derivatives $\frac{\partial}{\partial \Theta_k^{(l)}} J(\Theta)$

/* for $i = 1 : m$

perfor fwd propag^n & back prop using $(x^{(i)}, y^{(i)})$

get activa^s $a^{(l)}$ & delta terms $(\delta^{(l)}, \Delta^{(l)})$

$\delta^{(l)}$ for $l = 2, .., L$

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

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

*/

→ Use Gradient Checking to compare

$\frac{\partial}{\partial \theta_{jk}^{(l)}} J(\Theta)$ computed using backprop v/s

using Numerical estim^ of grad. of $J(\Theta)$.

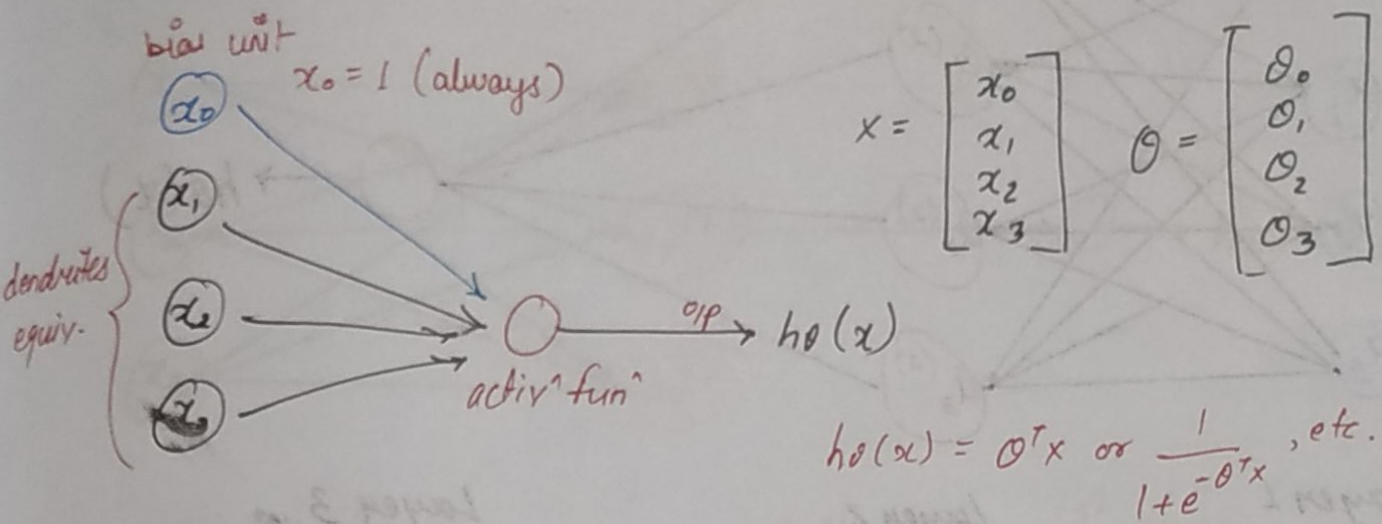
If these val.s are similar, our implement^ is correct & we can proceed.
If not, debug the prev. implement^

Then disable gradient checking code.

→ Use Gradient Descent or any other Advanced optimiz^ method w/ backprop to try to ~~min~~ minimize $J(\Theta)$ as a fun^ of para.s (Θ)

for NN, $J(\Theta)$ is a Non-Convex fun^ and so algo.s like grad. desc. can, theoretically be stuck in the local minima.
But in practice, it is found that it is usually not a great problem and usually algo.s like grad. desc. do a great job at minimizing $J(\Theta)$

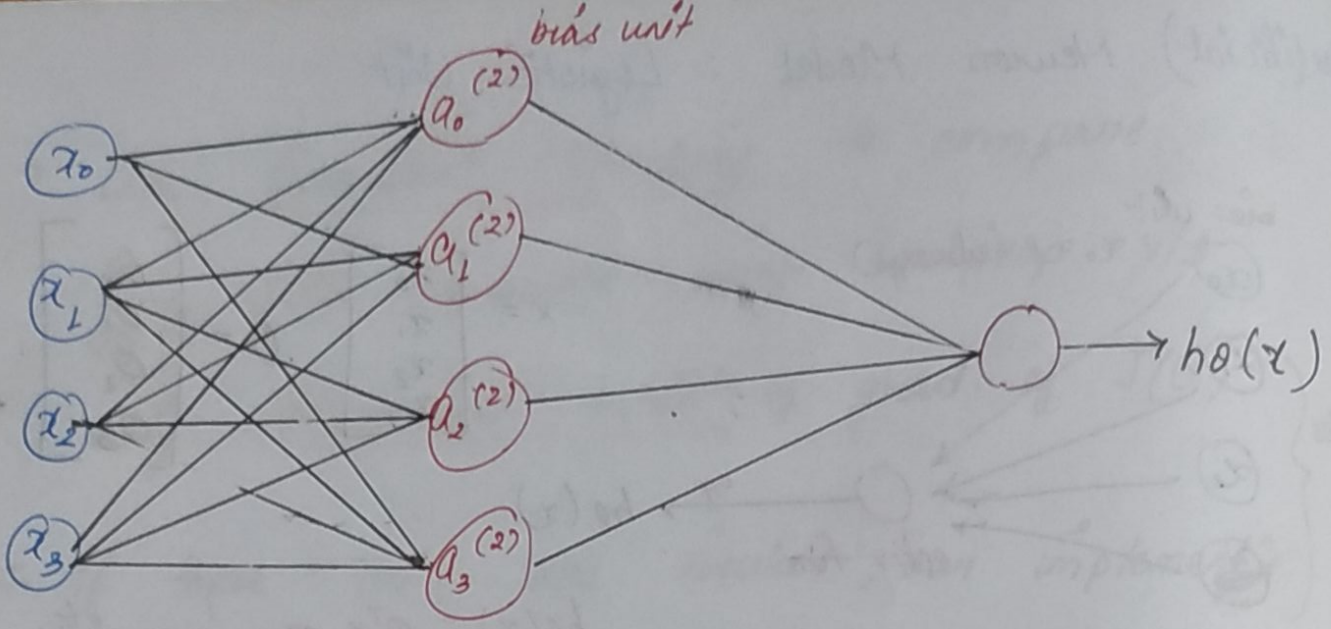
(Artificial) Neuron Model : Logistic Unit



Activ^n fun^n defines how the wt·ed sum of I/p is transformed into an o/p from a node i·e; it delivers an o/p based on I/p s.

The purpose of Activ^n fun^n is to add non-linearity to the N/W.

Activ^n fun^n decides whether or not a neuron should be activated i·e; it decides whether Neuron's I/p to the N/W is imp· to the process of predic^n



Layer 1 — I/p layer

Layer 2 — hidden layer

Layer 3 — O/p layer

$a_i^{(j)}$ → activ^n of unit 'i' in layer 'j'

$\theta^{(j)}$ → matrix of wt.s controlling fun^ mapping from layer 'j' to layer 'j+1'

$$a_1^{(2)} = \sigma\left(\theta_{10}^{(1)} x_0 + \theta_{11}^{(1)} x_1 + \theta_{12}^{(1)} x_2 + \theta_{13}^{(3)} x_3\right)$$

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

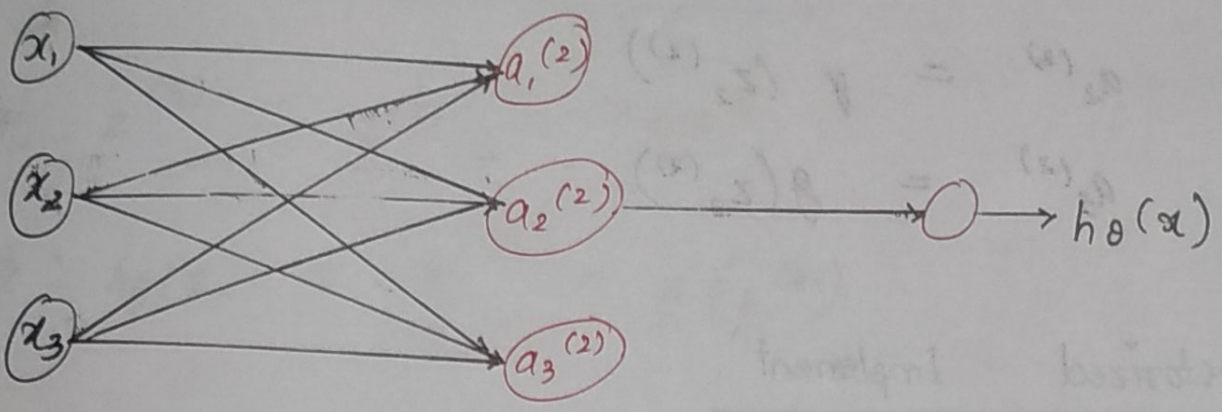
$$a_3^{(2)} = \sigma\left(\theta_{30}^{(1)} x_0 + \theta_{31}^{(1)} x_1 + \theta_{32}^{(1)} x_2 + \theta_{33}^{(1)} x_3\right)$$

(σ: activ^ fun^)

$$h\theta(x) = a_1^{(3)} = \sigma\left(\theta_{10}^{(2)} a_0^{(2)} + \theta_{11}^{(2)} a_1^{(2)} + \theta_{12}^{(2)} a_2^{(2)} + \theta_{13}^{(2)} a_3^{(2)}\right)$$

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

Forward Propagation : Vectorized Implemen^



$a^{(1)} = x$ i.e., x is Activ^ of 1st layer

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

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

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

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

let us define a new var. $z_k^{(j)}$ that encompasses the paras ($\theta^j / a_i^{(j)}$) inside activ^ fun^ $g()$.

$$z_k^{(j)} = \theta_{k,0}^{(j)} x_0 + \theta_{k,1}^{(j)} x_1 + \theta_{k,2}^{(j)} x_2 + \cdots + \theta_{k,n}^{(j)} x_n$$

$a_1^{(2)} = g(z_1^{(2)})$

$a_2^{(2)} = g(z_2^{(2)})$

$a_3^{(2)} = g(z_3^{(2)})$

Vectorized Implementation

$$x = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} \qquad z^{(j)} = \begin{bmatrix} z_1^{(j)} \\ z_2^{(j)} \\ \vdots \\ z_n^{(j)} \end{bmatrix}$$

setting $x = a^{(1)}$, we can rewrite the eq^n as

$$\boxed{\begin{aligned} z^{(j)} &= \Theta^{(j-1)} a^{(j-1)} \\ a^{(j)} &= g(z^{(j)}) \end{aligned}}$$

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

fun^n g is applied el.-wise on $z^{(j)}$

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

$\hookrightarrow \in \mathbb{R}^3$ $\qquad$ $\hookrightarrow \in \mathbb{R}^3$

Now, adding a bias unit ($=1$) to layer j after we have computed $a^{(j)}$ ($a_0^{(j)} = 1$)

$$\Rightarrow \boxed{z^{(j+1)} = \theta^{(j)} a^{(j)}}$$

So, $z^{(3)} = \theta^{(2)} a^{(2)} \in \mathbb{R}^4$

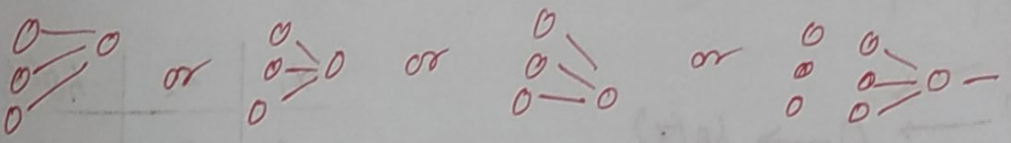
$$\Rightarrow h_\theta(x) = a^{(3)} = g(z^{(3)})$$

We get this final z vec. by multiplying the next θ matx. after $\theta^{(j-1)}$ w/ the val.s of all the activ^n nodes we get

This last theta matx. $\theta^{(j)}$ will have only 1 row wh. is multiplied w/ 1 col. $a^{(j)}$ so that our result is a single val. (Predicted val.)

$$\boxed{h_\theta(x) = a^{(j+1)} = g(z^{(j+1)})}$$

/* At each Neuron at each layer after the I/p layer,



we are doing exactly what we do in other algo.s ($\theta^T x$, $\frac{1}{1+e^{-\theta^T x}}$, etc.) here: $\sigma(z)$

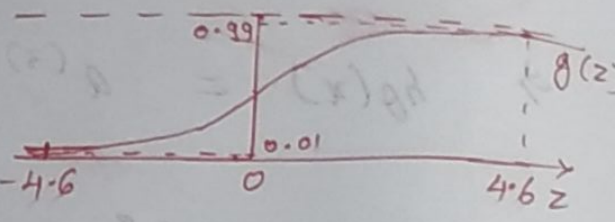
Adding all these intermediate layers in NN.s allows us to more elegantly produce more complex and interesting hypo. */

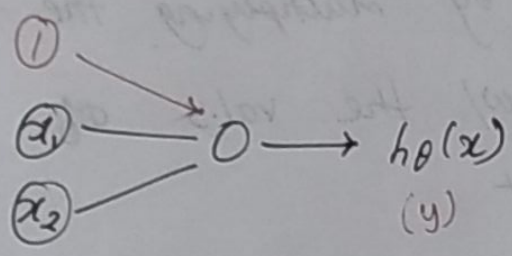
Neural Network example

→ (simple) Logical AND

→ $x_1, x_2 \in \{0, 1\}$

→ $y = x_1$ AND x_2





let, $\theta = [-30 \quad 20 \quad 20]$

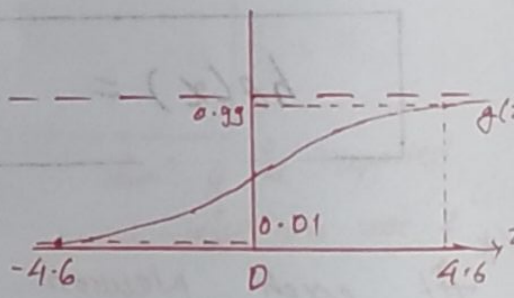
x_1	x_2	$h_\theta(x)$
0	0	$g(-30) \approx 0$
0	1	$g(-10) \approx 0$
1	0	$g(-10) \approx 0$
1	1	$g(10) \approx 1$

Logical AND

$h_\theta(x) = g(\theta^T x) = g(\cancel{-30 + 20x_1 + 20x_2})$ (-30)

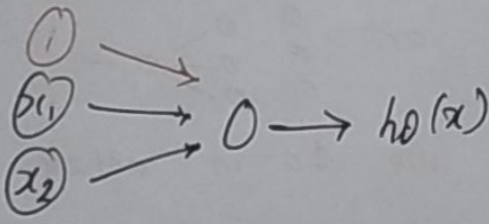
$= g(-30 + 20x_1 + 20x_2)$

(simple) Logical OR



→ $x_1, x_2 \in \{0, 1\}$

→ $y = x_1$ OR x_2



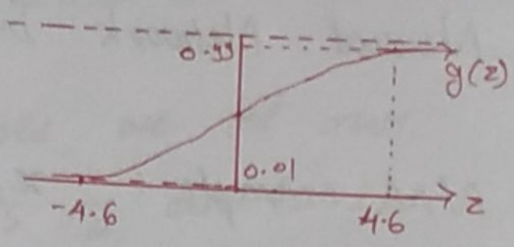
let, $\theta = [-10 \quad 20 \quad 20]$

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

Logical OR

⇒ $h_\theta(x) = g(\theta^T x) = g(-10 + 20x_1 + 20x_2)$

→ (simple) Logical NOT



→ $x_1 \in \{0, 1\}$

→ $y = h_\theta(x) = \text{NOT } x_1$

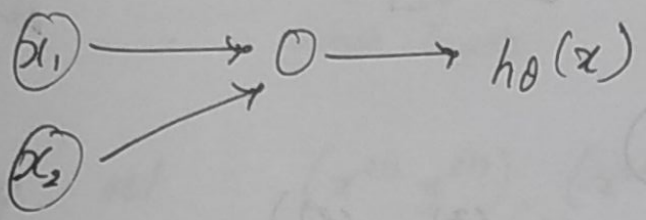
let, $\theta = [10 \quad -20]$

$\Rightarrow h_\theta(x) = g(\theta^T X) = g(10 - 20x_1)$

x_1	$h_\theta(x)$
0	$g(10) \approx 1$
1	$g(-20) \approx 0$

Logical NOT

→ (complex) (NOT x_1) AND (NOT x_2)



let, $\theta = [10 \quad -20 \quad -20]$

$\Rightarrow h_\theta(x) = g(\theta^T X)$

$= g(10 - 20x_1 - 20x_2)$

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

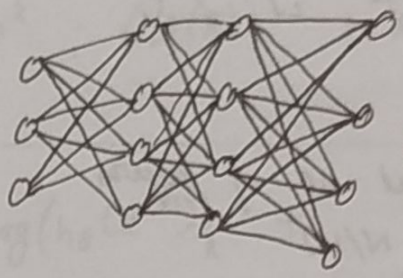
Multiclass Classific^n using Neural Networks

To classify data into multiple classes, we let our hypo. fun^ return a vec. of val.s of dim $(n \times 1)$ where n is the no. of classes.

say, $y \in \{\text{Pedestrian}, \text{Car}, \text{Bike}, \text{Truck}\}$

want,

$h_\theta(x) \approx \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$ when Pedestrian $\quad h_\theta(x) \approx \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$ when Car



$h_\theta(x) \in \mathbb{R}^4$

$h_\theta(x) \approx \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$ when Bike $\quad h_\theta(x) \approx \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$ when Truck

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

where $y \in \left\{ \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \right\}$

So,

> $y, h \in \mathbb{R}^n$
> where, $n \to$ no. of discrete classes

∴ Generalizing

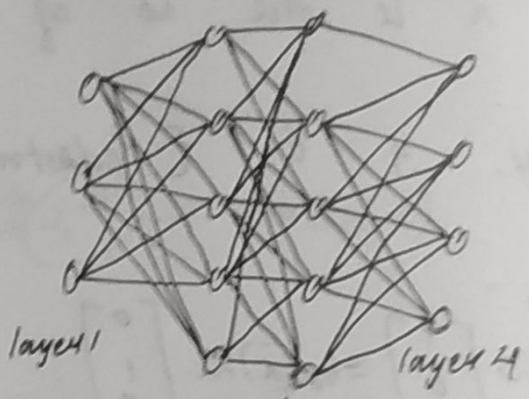
$$\begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} \to \begin{bmatrix} a_0^{(2)} \\ a_1^{(2)} \\ \vdots \\ a_n^{(2)} \end{bmatrix} \to \dots \to \begin{bmatrix} h_\theta(x_1) \\ h_\theta(x_2) \\ \vdots \\ h_\theta(x_n) \end{bmatrix}$$

where $\forall\ h_\theta(x_i) \in \mathbb{R}^n$

Neural Network Learning

NN (Classific^n)

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



L = total no. of layers in N/W

here, $L = 4$

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

here,

$s_L = 3$, $s_2 = 5$, $s_3 = 5$

$s_4 = s_L = 4$

k = no. of units in o/p layer

Binary Classific^n

$y = 0$ or 1 $\{y: 0, 1\}$ i.e., 1 o/p unit

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

$s_L = 1$ $k = 1$

Multi^o class Classific^n (k classes)

$y \in \mathbb{R}^k$ (k dimensional vec. ex: $y \in \left\{\begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\0\\1\end{bmatrix}\right\}$

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

$s_L = k$ $(k \geq 3)$

Cost Function

Log. Reg.

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

Neural Network: $h_\theta(x) \in \mathbb{R}^k$; $(h_\theta(x))_i$ = i^{th} o/p in k-dim. vec.

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

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

↓ → add'al nested summ' that loops through the no. of o/p nodes

no. of cols in θ matx = no. of cols in curr layer (including bias unit)

no. of rows in θ matx = no. of nodes in next layer (excluding bias unit)

NOTE:

- double sum simply adds the log. reg. cost calc.ed for each cell in the o/p layer
- triple sum simply adds up the sq.s of all individual θs in the entire N/W
- the 'i' in triple sum doesn't refer to training ex. 'i'

Backpropagation

"Backpropag" is a technique to minimize Cost in Neural Networks

Goal : $\min_{\Theta} J(\Theta)$

↓

we need to compute : ① $J(\Theta)$

② $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$ | $\Theta_{ij}^{(l)} \in \mathbb{R}$ i.e., a real no.

we calc. the Partial Derivative of Cost fun^ for optimiz^ algo.s like Grad. Desc. using the following Algo :

Backpropagation Algorithm

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

→ set $\Delta_{ij}^{(l)} = 0$ (for all l, i, j) | used to compute $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$

→ for $t = 1$ to m — i.e.; for i^{th} iteⁿ, we'll work w/ $(x^{(i)}, y^{(i)})$

- set $a^{(1)} = x^{(i)}$ — activⁿ
- perform fwd propagⁿ to compute for $l = 2, 3, \ldots, L$
- using $y^{(t)}$, compute $\delta^{(L)} = a^{(L)} - y^{(t)}$
- Compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$ using formula

$$\delta^{(l)} = \underset{np.dot}{\left((\theta^{(l+1)})^T \cdot \delta^{(l+1)}\right)} \underset{\text{element-wise multiplicⁿ}}{.*} \underset{\text{↳ derivative of activⁿ funⁿ}}{g'(z^{(l)})}$$

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

$\delta^{(L)}, \delta^{(L-1)}, \ldots \delta^{(2)}$ → only until layer 2 bc we don't calc. error for I/p layer

for sigmoid funⁿ, we know that,

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

$$\therefore \delta^{(l)} = \left((\theta^{(l)})^T \delta^{(l+1)}\right) .* a^{(l)} .* (1 - a^{(l)})$$

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

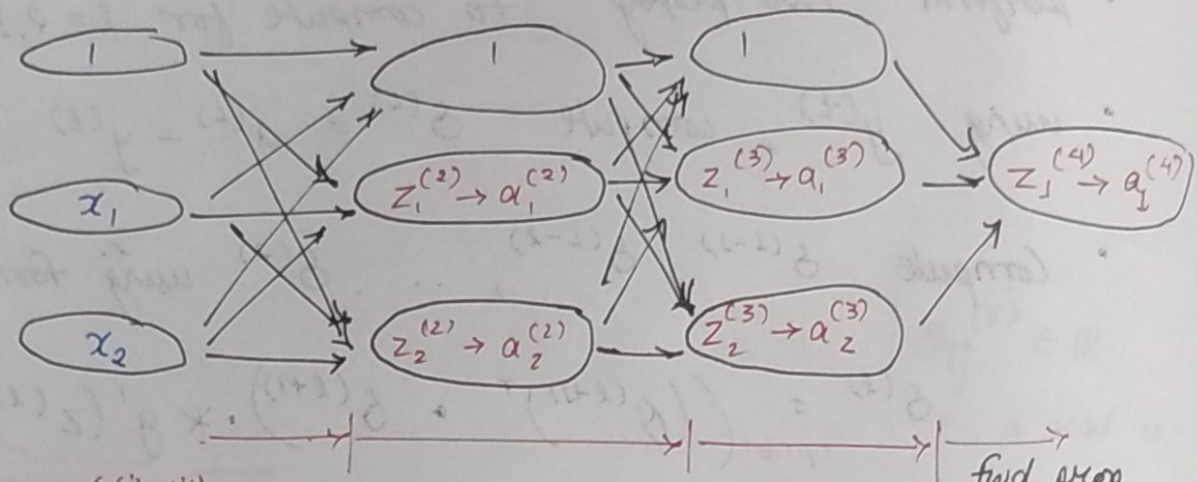
$D_{ij}^{(l)} := \frac{1}{m} \Delta_{ij}^{(l)}$ if $j = 0$ ↳ corresponds to bias term

The matr. 'D' is used as an "accumulator" to add up our val·s as we go along & eventually compute our Partial Derivative. Thus, we get (upon mathematical calc.)

$$\frac{\partial}{\partial \theta_{ij}^{(l)}} = D_{ij}^{(l)}$$

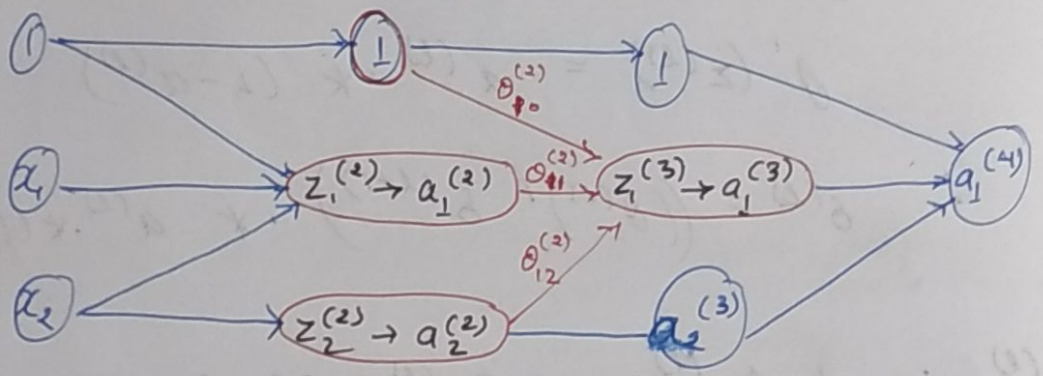
Backpropagation Intuition

Forward propag^



for some $(x^{(i)}, y^{(i)})$

Understanding 1 of the node in 1 of the layers (selected at random)



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

.....

For Intu^n, let us consider a simple non-multiclass classific^n ($k = 1$) tg

Ignoring regulariz^n ($\lambda = 0$), we get

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

$$\text{cost}(t) \approx \left(h_\theta(x^{(t)}) - y^{(t)}\right)^2$$

* Intuitively, Backpropagation computes $\delta_j^{(l)}$ i.e., the "error" of cost of $a_j^{(l)}$

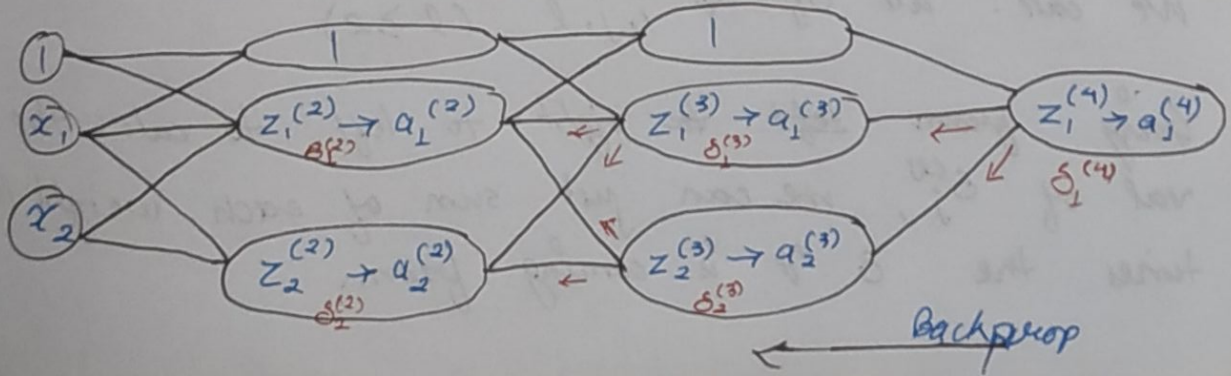
More formally, delta val.s are actually the derivatives of the Cost Fun^n.

$$\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(t) \qquad (\text{for } j \geq 0)$$

where, $\text{cost}(i) = y^{(i)} \log\left(h_\theta(x^{(i)})\right) +$

$$(1 - y^{(i)}) \log\left(1 - h_\theta(x^{(i)})\right)$$

Error, $\delta_j^{(l)} = a_j^{(l)} - y^{(i)}$ (calc.ed val − actual val.)

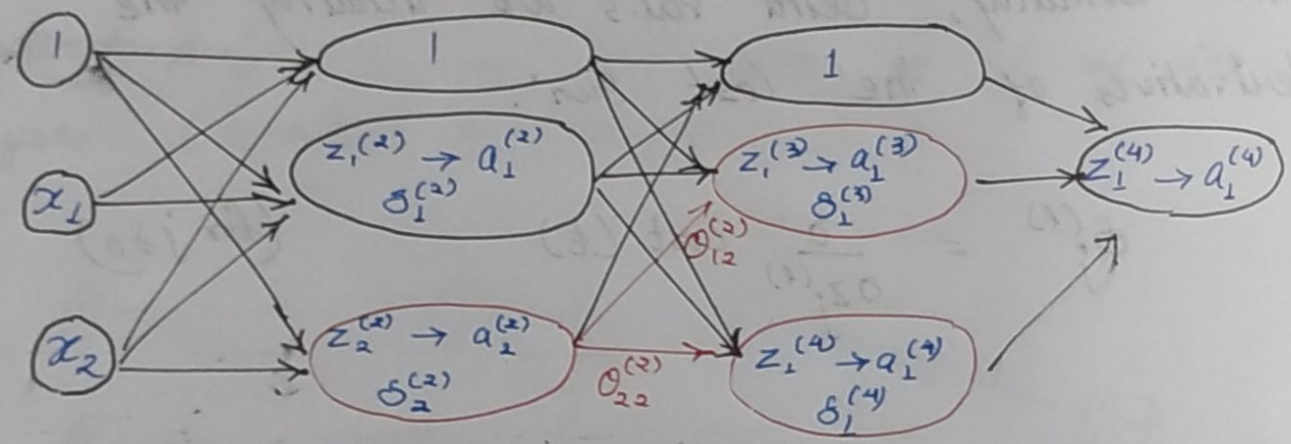


first, we calc. the error for the o/p layer

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

then, we propagate backwards to calc. the error for nodes of prev. layers (until layer 2)

Understanding 1 of the node of 1 of the column, (selected at random)



$$\delta_2^{(2)} = \theta_{12}^{(2)} \times \delta_1^{(3)} + \theta_{22}^{(2)} \times \delta_2^{(4)}$$

Similarly, $\delta_2^{(3)} = \delta_1^{(4)} \times \theta_{12}^{(4)}$

we calc. all $\delta_{ij}^{(l)} \; \forall \; i, j, l \quad (l \geq 2)$

Going from ~~left to~~ right to left, to calc. the val of $\delta_j^{(l)}$, we can just sum of each weight (0), times the δ it is coming from.

Implementation Note: Unrolling Params

w/ NNs, we work w/ sets of matrices:

$\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}, \ldots$ (params / wts) and

$D^{(1)}, D^{(2)}, D^{(3)}, \ldots$ (grads), etc.

We can unroll these matrices into vectors.

$[D]_{m\times 3} = [D]_{m\times 1} + [D]_{m\times 1} + [D]_{m\times 1}$

The advantage of matrix representation of params is that it allows us to take advantage of Vectorized Implementation and is more convenient while performing forward and backpropagation.

The advantage of vector repr. of params is that some advanced optimization algos/methods (in Octave) assume that the params passed to them are "unrolled" into a vector and thus, vector representation of params allows us to use these Advanced Optimization algos/methods.

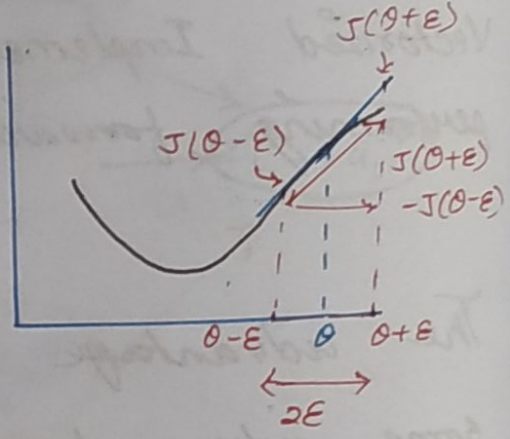
Gradient Checking

Numerical Gradient Checking

Backpropag^ is a tricky algo.
As a result, upon implement^ and use of an optimiz^ algo like grad. desc., it may appear that the para.s are optimizing but the bugs may not eventually minimize $J(\theta)$

So, (Numerical) Gradient Checking is a technique (an expensive technique) to check if implement^ of backpropag^ works as intended.

we can approximate the grad. of our Cost Fun^ w/ :



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

↑ 2 sided difference

or

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

↙ 1 sided difference

w/ multiple theta matrices, $\theta \in \mathbb{R}^n$ (Unrolled / Vectorized),
we can approximate the derivative as

$$\theta = [\theta_1, \theta_2, \ldots, \theta_n]$$

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

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

$$\vdots$$

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

i.e.,

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

a small val. for ε such as $\varepsilon = 10^{-4}$ guarantees that the math works out properly

if ε is too small, we can end up w/ numerical problems

Implement^ :

```
# epsilon = 1e-4
for i from 1 to n:
    thetaPlus    = theta
    thetaPlus(i) = thetaPlus(i) + epsilon
    thetaMinus   = theta
    thetaMinus(i) = thetaMinus(i) - epsilon
    gradApprox(i) = (J(thetaPlus) - J(thetaMinus))
                    / (2*epsilon)
```

Check that gradApprox ≈ DVec

Implement^ Note :

→ Implement backprop to compute DVec

→ Implement grad. checking to compute gradApprox

→ Check if gradApprox and DVec gives similar val.s.
If yes, our implement^ of backprop is correct
If not, debug backprop implement^

→ Turn off grad. checking bc it is expensive in comput^ power & time

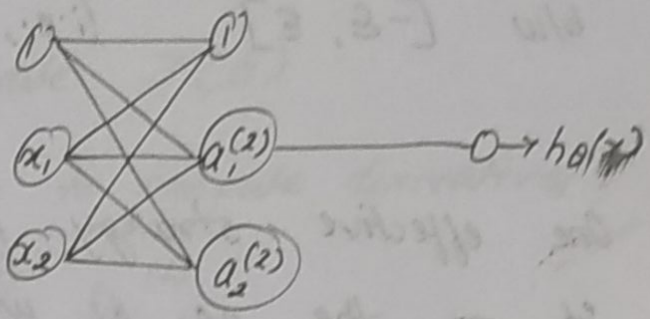
→ Use backprop code for learning i.e., for further optimiz^.

Random Initialization

Need : Zero Initializ^n

Initializing all the wt.s i.e., 0 to 0 does not work. as all nodes will update the same val. repeatedly.

After each update, para.s corresponding to i/ps going into each of 2 hidden units are identical, i.e.;



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

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

$$\theta_{01}^{(1)} = \theta_{02}^{(2)}$$

So no matter how many units in the hidden layer, each of them would compute the same val as they are fed the same i/p's ([0])

Random Initialization : Symmetry Breaking

to avoid the aforementioned problem, we initialize each $\Theta_{ij}^{(l)}$ to a random val. b/w $[-\varepsilon, \varepsilon]$ (i.e., $-\varepsilon \leq \Theta_{ij}^{(l)} \leq \varepsilon$)

One effective strategy for selecting ε is to base it on the no. of units in the N/W.

A good choice is

$$\varepsilon = \frac{\sqrt{6}}{\sqrt{L_{in} + L_{out}}}$$

where, $L_{in} = s_l$ and $L_{out} = s_{l+1}$ are the no. of units in the layer adjacent to Θ_l

Implement :

```
# init_epsilon = some val.
  Theta1     = np.random.rand(row_size, col_size)
               * (2 * init_epsilon) - init_epsilon
  Theta2     =                  ||
  Theta3     =                  ||
```

Neural Network learning (Summary)

→ pick a N/W Architecture

→ randomly initialize wt.s (Θ) in range $[-\epsilon, \epsilon]$

→ implement forward propagation to compute $h_\Theta(x)$

→ implement code to compute $J(\Theta)$

→ implement backpropagation to compute derivatives

$\frac{\partial}{\partial \Theta_k^{(l)}} J(\Theta)$ | $\delta^{(l)}$, $\Delta^{(l)}$, $D_{ij}^{(l)}$

→ use gradient checking to compare gradient computed w/ backpropagation w/ gradient computed w/ numerical estimation

→ if not, debug backpropagation
if yes, disable gradient checking code

→ use Gradient Descent or any other Advanced Optimiz^n Technique w/ Backpropagation to ~~try~~ ~~to~~ min$_\Theta$ $J(\Theta)$

...

→ ~~train~~ ~~it~~