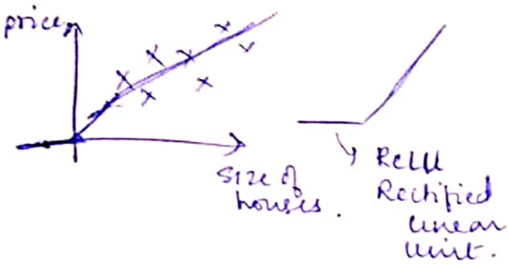
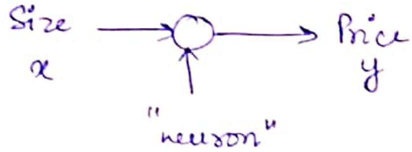
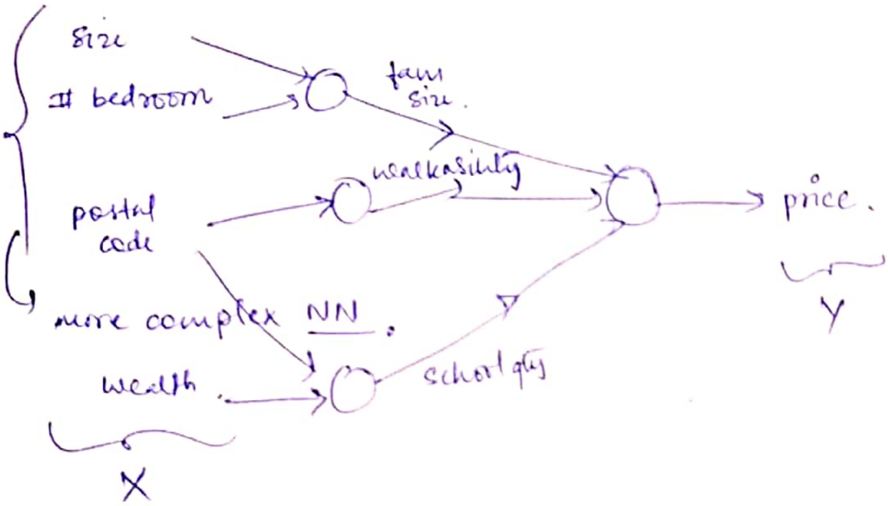
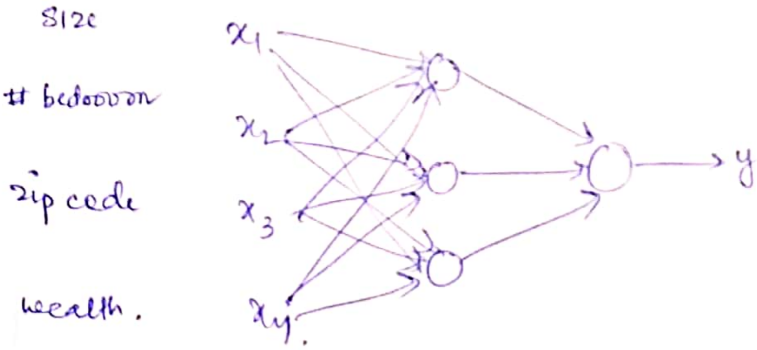
I. Neural N/w & Deep Learning:

* What is a NN?





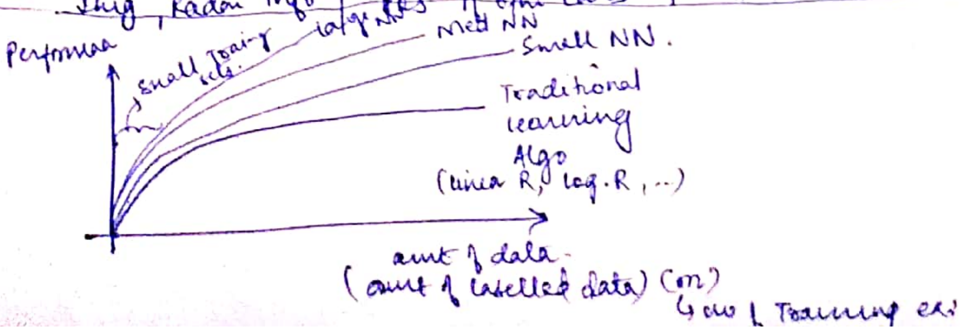




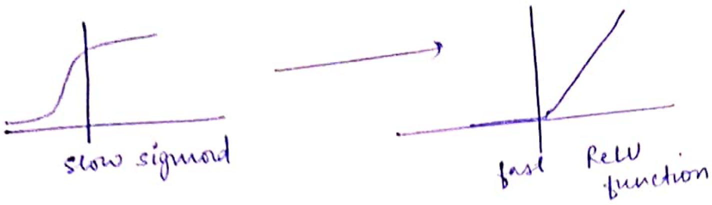
* Supervised learning with Neural N/w:

Ex:

Input (x)	o/p (y)	Applicⁿ
Ad, user info	Click on ad? (0/1)	Online Advert → Standard NN
Image	Obj (1, ..., 1000)	Photo tagging → CNN
Audio	Text transcript	speech recog → RNN
⋮	⋮	⋮
Img, Radar info	Posⁿ of other cars	Auto drive → Hybrid/complex:



DL:



⊕ logistic Regression as Neural N/w :

* Binary classification :

unroll pixel values into a feature vector.

$$x = \begin{bmatrix} 255 \\ 231 \\ \vdots \\ 255 \\ 134 \\ \vdots \end{bmatrix}$$

if img is 64x64 . & 3 vectors.

$\therefore$ $n_x = 64 \times 64 \times 3 = 12288$

$x \rightarrow y$.

Notation :

(X, y) : single T.S. $\in \mathbb{R}^{n_x}$, & $y \in \{0, 1\}$

m training example.

$m = m_{train}$ $\quad m_{test}$ = # test examples.

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix} \quad \updownarrow n_x, \quad \overleftrightarrow{m}$$

$X \in \mathbb{R}^{n_x \times m}$

$Y = [y^{(1)}, y^{(2)} \; --- \; y^{(m)}]$

$Y \in \mathbb{R}^{1 \times m}$

* Logistic Regression

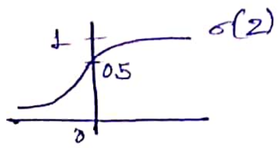
X, want $\hat{y} = P(y=1 | x)$

$X \in \mathbb{R}^{n_x}$

params $w \in \mathbb{R}^{n_x}$

$b \in \mathbb{R}$

Output $\hat{y} = \sigma(w^T x + b)$



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

if z large, $\sigma(z) \approx 1$

z small < 0, $\sigma(z) \approx 0$

$x_0 = 1 \quad x \in \mathbb{R}^{n_x+1}$

$\hat{y} = \sigma(\theta^T x)$

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_{n_x} \end{bmatrix} \begin{array}{l} \} \; b \\ \left.\begin{array}{l} \\ \\ \\ \end{array}\right\} \; w \end{array}$$

→ logistic Regression cost funcⁿ:

$\hat{y} = \sigma(w^T x + b) \qquad \sigma(z) = \frac{1}{1+e^{-z}}$

$z^{(i)} = w^T x^{(i)} + b$

given $\{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}$ $\quad \hat{y}^{(i)} \approx y^{(i)}$

$\left.\begin{array}{l} x^{(i)} \\ y^{(i)} \\ z^{(i)} \end{array}\right\} \rightarrow i^{th}$ example.

loss (error funcⁿ): $L(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$

$L(\hat{y}, y) = -(y \log \hat{y} + (1-y)\log(1-\hat{y}))$

$y = 1 \; ; \; L(\hat{y}, y) = -\log \hat{y}$ ← want $\log \hat{y}$ large, want $\hat{y}$ large.

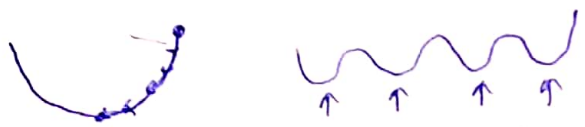
$y = 0 \; ; \; L(\hat{y}, y) = -\log(1-\hat{y})$ ← want $\log(1-\hat{y})$ large, want $\hat{y}$ small.

Cost function (J) for entire T.S.

$$\boxed{J(w, b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log \hat{y}^{(i)} + (1-y^{(i)})\log(1-\hat{y}^{(i)})\right]}$$

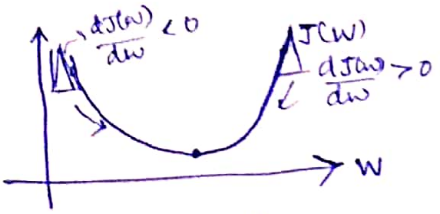
* gradient descent :

want to find w and b that minimise $J(w, b)$



convex funcⁿ

not convex.



Repeat {

$w := w - \alpha \frac{dJ(w)}{dw}$ ← learning rate

}

$\underbrace{\frac{dJ(w)}{dw}}_{\text{"dw"}}$

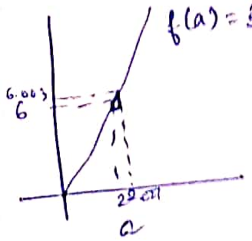
$\Rightarrow w := w - \alpha\, dw$

$J(w, b)$ $\quad$ {

$w := w - \alpha \frac{dJ(w,b)}{dw}$

$b := b - \alpha \frac{dJ(w,b)}{db}$ }

* Derivatives :



$f(a) = 3a$.

$a = 2 \quad f(a) = 6$

$a = 2.001 \quad f(a) = 6.003$.

Slope (derivative) of $f(a)$ at $a = 2$ is '3'.

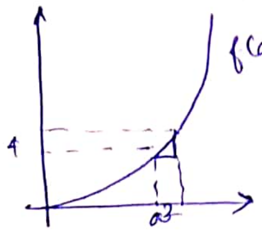
↓

h/w.

$\frac{0.003}{0.001} = 3$

$\frac{df(a)}{da} = 3 = \frac{d}{da} f(a)$.

derivatives, nudging by infinitesimally small amount



$f(a) = a^2$

$a = 2 \quad f(a) = 4$

$a = 2.001 \quad f(a) \approx 4.004$

$\left.\frac{df(a)}{da}\right|_{a=2} = 4$.

* Computation graph

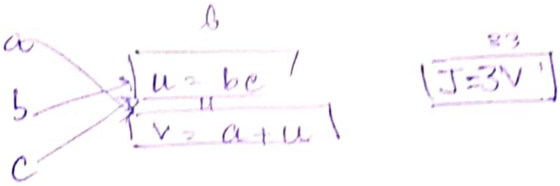
$J(a,b,c) = 3(a+\underbrace{bc}_{u})$

$u = bc$

$u = a + u$

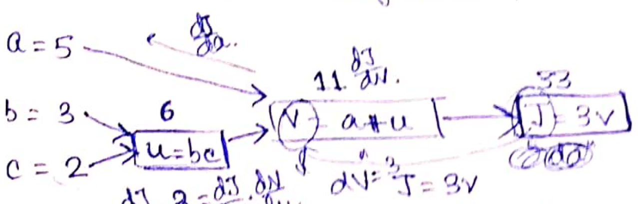
$J = 3v$.

$= 3(5+3\times2) = 33$



one step of (←) prop. on computation graph yields derivative of final o/p variable.

* Derivatives with computation graph :-



V = 11 → 11.001

J = 33 → 33.003.

← Back propagation

a: 5 → 5.001

v = 11 → 11.001

J = 33 → 33.003

$\frac{dJ}{da} = \frac{dJ}{dv} \cdot \frac{dv}{da}$ → chain rule.

$f(a) = 3a$

$\frac{df(a)}{da} = \frac{df}{da} = 3$

$J = 3v$

$\frac{dJ}{dv} = 3$.

$\boxed{\frac{d\,\text{FinalOutputVariable}}{d\,\text{Var}}}$

↳ "dvar"

* logistic regression Grad. Descent.

$z = w^T x + b.$

$\hat{y} = a = \sigma(z)$

$L(a,y) = -(y(\log \hat{y}) + (1-y)\log(1-\hat{y}))$

$$\begin{array}{l} x_1 \\ w_1 \\ x_2 \\ w_2 \\ b \end{array} \rightarrow \boxed{z = w_1x_1 + w_2x_2 + b} \rightarrow \boxed{a = \sigma(z)} \rightarrow \boxed{L(a,y)}$$

"dz" $= \frac{dL}{dz} = \frac{dL(a,y)}{dz}$

$= a - y.$

$= \frac{dL}{da} \cdot \frac{da}{dz}$

"da" $= \frac{dL(a,y)}{da}.$

$= -\frac{y}{a} + \frac{1-y}{1-a}$

$\frac{dL}{dw_1} =$ "dw_1" $= x_1 dz$

"dw_2" $= x_2 dz$

"db" $=$ "dz"

$w_1 := w_1 - \alpha dw_1$

$w_2 := w_2 - \alpha dw_2$

$b := b - \alpha db.$

** for 'm' training examples :

$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L(a^{(i)}, y)$

$a^{(i)} = \hat{y}^{(i)} = \sigma(z^{(i)}) = \sigma(w^T x^{(i)} + b)$

$\frac{\partial}{\partial w_1} J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \underbrace{\frac{\partial}{\partial w_1} L(a^{(i)}, y^{(i)})}_{dw_1^{(i)} \,-\, (x^{(i)}, y^{(i)})}$

$J = 0$; $dw_1 = 0$; $dw_2 = 0$; $db = 0$.

for $i = 1$ to m

$z^{(i)} = w^T x^{(i)} + b$

$a^{(i)} = \sigma(z^{(i)})$

$J + = -[y^{(i)} \log a^{(i)} + (1-y^{(i)}) \log(1-a^{(i)})]$

$dz^{(i)} = a^{(i)} - y^{(i)}$

$dw_1 += x_1^{(i)} dz^{(i)}$

$dw_2 += x_2^{(i)} dz^{(i)}$

$db += dz^{(i)}$

Ans

$J /= m$

$dw_1 /= m$

$dw_2 /= m$

$db /= m$

sol^n from for loops less eff.

→ vectorisation is sol^n.

* vectorisation :

$z = w^T x + b$ $\quad w = \begin{bmatrix} \vdots \\ \vdots \end{bmatrix} \quad x = \begin{bmatrix} \vdots \\ \vdots \end{bmatrix}$

$w \in \mathbb{R}^{n_x}$

$x \in \mathbb{R}^{n_x}$

vectorised impl^n :

z = np.dot (w, x) + b.

{GPU, CPU} SIMD - single inst. multiple data.

→ helps in parallelism.

- avoid for loops wherever possible.

$u = AV$.

$u_i = \sum_j A_{ij} V_j$

u = np.zeros ((n, 1))
for i = ...
for j = ...
} long time.

vect. version

u = np.dot (A, v)

$V = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$.

non vectorised.

```
u = np.zeros ((n,1))
for i in range (n):
    u(i) = math.exp (v[i])
```

vectorised.

```
import numpy as np.
u = np.exp(v)
np.log (v)
np.abs (v)
np.max (v, 0)
⋮
```

vectorised imp of ~~logistic~~ logistic regression.

$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ | & | & & | \end{bmatrix}$ (n_x, m) $\mathbb{R}^{n_x \times m}$

$Z = [z^{(1)}, z^{(2)}, \dots, z^{(m)}] = w^T X + \underbrace{[b \ b \ b \dots b]}_{1 \times m}$

$= \begin{bmatrix} w^T x^{(1)} + b & w^T x^{(2)} + b & w^T x^{(3)} + b & \dots \end{bmatrix}$

Z = np.dot (w.T, X) + b $\quad$ "Broadcasting"

$\hookrightarrow (1,1) \ \mathbb{R}$

$A = [a^{(1)}, a^{(2)}, \dots, a^{(m)}] = \sigma(Z)$

Vectorizing logistic Regrⁿ gradient o/p :

$dz^{(1)} = a^{(1)} - y^{(1)}$ $dz^{(2)} = a^{(2)} - y^{(2)}$ . . . . .

$dZ = [dz^{(1)}\ dz^{(2)} \dots dz^{(m)}]$

$A = [a^{(1)}, \dots, a^{(m)}]$ $Y = [y^{(1)} \dots y^{(m)}]$

$dZ = A - Y$

$dw = 0$
$dw \mathrel{+}= x^{(1)} dz^{(1)}$
$dw \mathrel{+}= x^{(2)} dz^{(2)}$
$\vdots$
$dw /= m$

$db = 0$
$db \mathrel{+}= dz^{(1)}$
$db \mathrel{+}= dz^{(2)}$
$\vdots$
$db \mathrel{+}= dz^{(m)}$
$db /= m$

$db = \frac{1}{m} \sum_{i=1}^{m} dz^{(i)}$

$= \frac{1}{m}$ (np.sum(dz))

$dw = \frac{1}{m} X dz^T$

$= \frac{1}{m} \begin{bmatrix} | & | & \\ x^{(1)} & x^{(2)} & \dots \\ | & | & \end{bmatrix} \begin{bmatrix} dz^{(1)} \\ \vdots \\ dz^{(m)} \end{bmatrix}$

$= \frac{1}{m} [x^{(1)} dz^{(1)} + \dots + x^{(m)} dz^{(m)}]_{n \times 1}$

vectorised: logistic regression.

$Z = w^T X + b$
$=$ np.dot(w.T, X) + b.
$A = \sigma(Z)$
$dz = A - Y$
$dw = \frac{1}{m} X dz^T$
$db = \frac{1}{m}$ np.sum(dz)

* Broadcasting in python.

Ex:

	Apples	Beef	Eggs	Potatoes
Carb	56.0	0.0	4.4	68.0
Protein	1.2	104.0	52.0	8.0
Fats	1.8	135.0	99.0	0.9

$= A_{(3,4)}$..

% calories : without explicit for loop.   axis = 1 : horizontal sum

cal = A.sum(axis = 0) ⟶ $\boxed{\text{vertical sum}}$.

percentage = 100 * A / cal.reshape(1, 4)

↓ python broadcasting

↓ divide by 1×4 matrix.

$(3,4)/(1,4)$.

Ex:

$\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} + 100 \equiv \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} + \begin{bmatrix} 100 \\ 100 \\ 100 \\ 100 \end{bmatrix} = \begin{bmatrix} 101 \\ 102 \\ 103 \\ 104 \end{bmatrix}$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 100 & 200 & 300 \\ 100 & 200 & 300 \end{bmatrix} = \begin{bmatrix} 101 & 202 & 303 \\ 104 & 205 & 306 \end{bmatrix}$$

(m,n) $\quad (1,m) \rightarrow m,n)$

General principle.

$(m,n) \;\; \begin{matrix} + \\ * \\ / \end{matrix} \;\; \begin{matrix} (1,n) \rightsquigarrow (m,n). \\ (m,1) \rightsquigarrow (m,n) \end{matrix}$

→ Tip : To avoid bugs, don't use datastructures.

a= np.random. randn (5).

a. shape = (5,) ⟶ can fine bugs.

"rank 1 array"

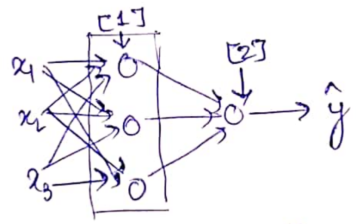
instead
(1, 5) or (5, 1)

or

assert (a.shape == (5,1))

a=a.reshape((5,1))

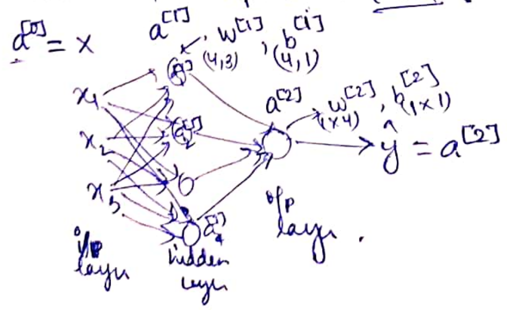
Shallow Neural N/w :



$z^{[1]} = w^{[1]} x + b^{[1]}$; $a^{[1]} = \sigma(z^{[1]})$

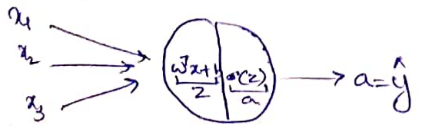
$z^{[2]} = w^{[2]} a^{[1]} + b^{[2]}$; $a^{[2]} = \sigma(z^{[2]})$

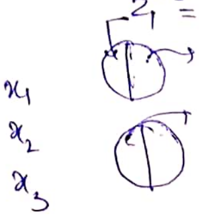
* Neural N/w representation :



2 layer NN:
- don't count hidden layer.

* Computing a neural N/w's output.





$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]}$

$a_1^{[1]} = \sigma(z_1^{[1]})$

$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]}$, $a_2^{[1]} = \sigma(z_2^{[1]})$

⋮

$$\begin{bmatrix} — w_1^{[1]T} — \\ — w_2^{[1]T} — \\ — w_3^{[1]T} — \\ — w_4^{[1]T} — \end{bmatrix}_{(4,3)} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix}$$

$W^{[1]}$

$$a^{[1]} = \sigma(z^{[1]})$$

given i/p 'x',

$z^{[1]} = W^{[1]} a^{[0]} + b^{[1]}$

$a^{[1]} = \sigma(z^{[1]})$

next layer.

$z^{[2]} = W^{[2]} a^{[1]} + b^{[2]}$

$a^{[2]} = \sigma(z^{[2]})$

$X \longrightarrow a^{[2]} = \hat{y}$.

$x^{(1)} \longrightarrow \hat{y}^{(1)} = a^{[2](1)}$

$x^{(2)} \longrightarrow \hat{y}^{(2)} = a^{[2](2)}$

$\vdots$

$x^{(m)} \longrightarrow \hat{y}^{(m)} = a^{[2](m)}$

$a^{[2],(i)}$ ↓ layer 2, ↳ example i

for i=1 to m

$z^{[1](i)} = W^{[1]} x^{(i)} + b^{[1]}$

$a^{[1](i)} = \sigma(z^{[1](i)})$

$z^{[2](i)} = W^{[2]} a^{[1](i)} + b^{[2]}$

$a^{[2](i)} = \sigma(z^{[2](i)})$

vectorizaⁿ :

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix} \quad (n_x \times m)$$

$$Z^{[1]} = \begin{bmatrix} | & | & & | \\ z^{[1](1)} & z^{[1](2)} & \cdots & z^{[1](m)} \\ | & | & & | \end{bmatrix}$$

$$A^{[1]} = \begin{bmatrix} | & | & & | \\ a^{[1](1)} & a^{[1](2)} & \cdots & a^{[1](m)} \\ | & | & & | \end{bmatrix}$$

training ex →

↓ Hidden units.

$Z^{[1]} = W^{[1]} X + b^{[1]}$

$A^{[1]} = \sigma(Z^{[1]})$

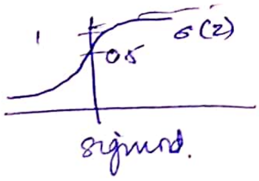
$Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$

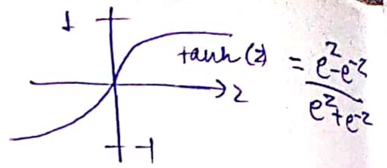
$A^{[2]} = \sigma(Z^{[2]})$

*Activation func^n:

$\sigma(z^{[1]}) \hookrightarrow g(z^{[1]})$

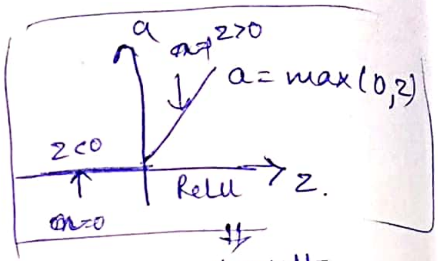
$\sigma(z^{[2]}) \hookrightarrow g(z^{[2]})$





$y \in \{0, 1\}$
↓
use sigmoid

activation func can be diff for diff layers.



* why nonlinear Activation func^n?

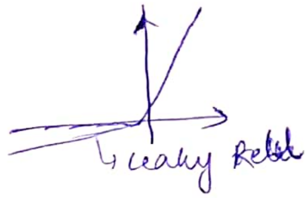
- if not, then

$a^{[1]} = z^{[1]} = w^{[1]}x + b$

$a^{[2]} = z^{[2]} = w^{[2]}a^{[1]} + b^{[2]}$

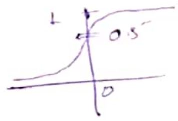
∴ linear implementation only.

∴ might not have any hidden layer.



*Derivatives of Activation func^n.

→ $g(z) = \frac{1}{1+e^{-z}}$



$\frac{d}{dz} g(z) = \frac{1}{1+e^{-z}}\left(1 - \frac{1}{1+e^{-z}}\right)$

$= g(z)\,(1 - g(z))$.

$z \approx 10$

$g(z) \approx 1$.

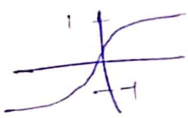
$g'(z) = \frac{d}{dz} g(z) \approx 0$.

$z \approx -10$

$g(z) \approx 0$

$\frac{d}{dz} g(z) \approx 0$

→ $g(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$.



$\frac{d}{dz} g(z) = 1 - (\tanh(z))^2$.

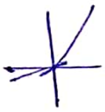
→ ReLU & leaky ReLU..

$g(z) = \max(0, z)$

$g'(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$

technically undefined at zero.

$g(z) = \max(0.01, z)$



$g'(z) = \begin{cases} 0.01 & z < 0 \\ 1 & z \geq 0 \end{cases}$

Scanned by CamScanner

* Gradient descent for neural N/w :

Params : $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}$

$w^{[1]}: (n^{[1]}, n^{[0]})$, $b^{[1]}: (n^{[1]}, 1)$, $w^{[2]}: (n^{[2]}, n^{[1]})$, $b^{[2]}: (n^{[2]}, 1)$.

$n_x = n^{[0]}, \; n^{[1]}, \; n^{[2]} = 1$

cost func : $J(w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}) = \frac{1}{m} \sum_{i=1}^{n} L(\underset{\uparrow}{\hat{y}}, y)$

$\hat{y} = a^{[2]}$.

Grad. descent

Repeat {

Compute pred $(\hat{y}^{(i)}, \ldots \; i=1, \ldots m)$

$dw^{[1]} = \frac{dJ}{dw^{[1]}}$

$db^{[1]} = \frac{dJ}{db^{[1]}}$

$w^{[1]} := w^{[1]} - \alpha \, dw^{[1]}$

$b^{[1]} := b^{[1]} - \alpha \, db^{[1]}$

Formulas for computing derivatives.

forward prop :

$Z^{[1]} = W^{[1]}X + b^{[1]}$

$A^{[1]} = g^{[1]}(Z^{[1]})$

$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$

$A^{[2]} = g^{[2]}(Z^{[2]}) =$

Back propagn :

$dz^{[2]} = A^{[2]} - Y$

$dw^{[2]} = \frac{1}{m} dz^{[2]} A^{[1]T}$

$db^{[2]} = \frac{1}{m}$ np.sum $(dz^{[2]}$, axis=1 , keepdims=True)

$dz^{[1]} = w^{[2]T} dz^{[2]} * g^{[1]'}(z^{[1]})$

$dw^{[1]} = \frac{1}{m} dz^{[1]} X^T$

$db^{[1]} = \frac{1}{m}$ np.sum $(dz^{[1]}$, axis=1 , keepdims=True

* Random initialisations

initialising w to all zeros is problematic

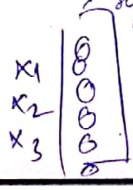
$a_1^{[1]} = a_2^{[1]}$ $\quad dz_1^{[1]} = dz_2^{[1]}$.

$\therefore$ 2 hidden units can be completely identical.

$dw = \begin{bmatrix} u & v \\ u & v \end{bmatrix}$ after iterations, all rows equal in $w^{[1]}$

symmetric

symmetric



Sol^n: $w^{[1]} =$ np.random.randn $((2,2)) * 0.01$

$b^{[1]} =$ np.zero((2,1))

we wanna initialise small values.

$w^{[2]} =$ ---

$b^{[2]} = 0$

Deep neural Networks :

$L \rightarrow$ no of layers

$n^{[l]} \rightarrow$ no of units in 'l'.

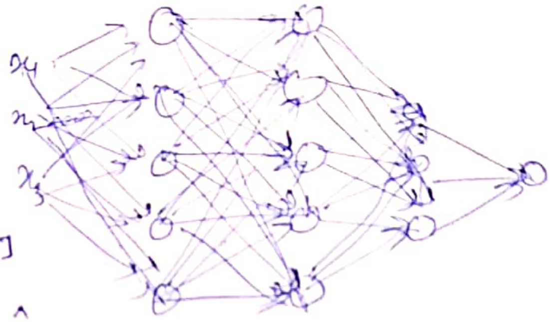
$n^{[1]} = 5$

$n^{[2]} = 5$

$n^{[3]} = 3$

$X = a^{[0]}$

$a^{[L]} = \hat{y}$



4 layers.

* Forward propagation in a deep neural N/W :-

X: $z^{[1]} = w^{[1]}x + b^{[1]}$

$a^{[1]} = g^{[1]}(z^{[1]})$

$z^{[2]} = w^{[2]}a^{[1]} + b^{[2]}$

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

...

$z^{[4]} = w^{[4]}a^{[3]} + b^{[4]}$

$a^{[4]} = g^{[4]}(z^{[4]})$

$z^{[l]} = w^{[l]}a^{[l-1]} + b^{[l]}$

$a^{[l]} = g^{[l]}(z^{[l]})$

vectorised

$Z^{[1]} = W^{[1]}A^{0} + b^{[1]}$

$A^{[1]} = g^{[1]}(Z^{[1]})$

⋮

for deep NN, a for loop has to be there. loop goes from 1 to L

* getting matrix dimensions right.

$L=5$. $n^{[1]} = 3$ $n^{[2]} = 5$ $n^{[3]} = 5$ $n^{[4]} = 2$ $n^{[5]} = 1$

$z^{[1]} = w^{[1]}x + b^{[1]}$

$(3,1)$ $(2,1)$

$(n^{[1]},1)$ $(n^{[0]},1)$

$[\vdots] = [\vdots \vdots][\vdots]$

$w^{[1]} = (n^{[1]}, n^{[0]})$

$w^{(l)} = (n^{[l]}, n^{[l-1]})$

$z^{[2]} = w^{[2]}a^{[1]} + b^{[2]}$

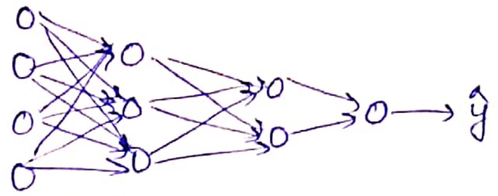
$(5,1)$ $(5,3)$ $(3,1)$ $(5,1)$.

vectorised

$Z^{[1]} = W^{[1]}X + b^{[1]}$

$(n^{[1]},m)$ $(n^{[1]},n^{[0]})$ $(n^{[0]},m)$ $(n^{[1]},1) \rightarrow$ broadcasting here.

* Intuition abt deep representation.



- Face recognition
- Audio → low level audio → Phonemes → words → Sentence.

- Circuit theory & deep learning:
func that compute with small L layer deep NN that shallower NN need more exponential hidden layers.

* Building blocks of deep NN.

Forward layer l : $w^{[l]}, b^{[l]}$

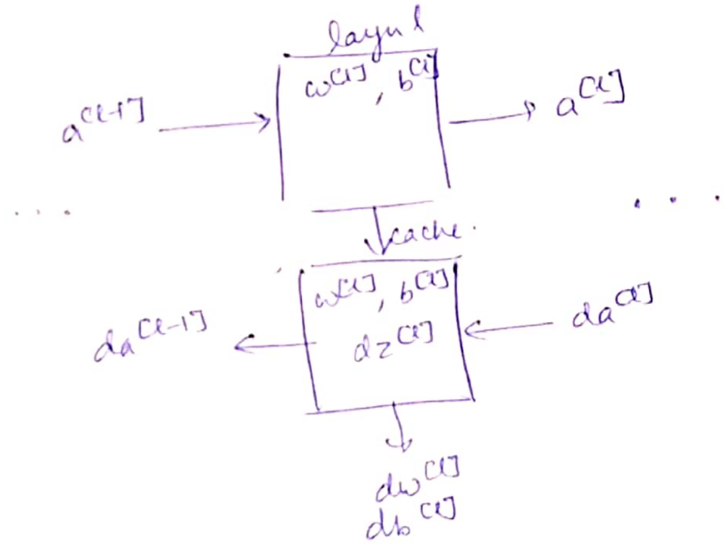
i/p : $a^{[l-1]}$

o/p : $a^{[l]}$

$z^{[l]} = w^{[l]} a^{[l-1]} + b^{[l]}$

$a^{[l]} = g^{[l]}(z^{[l]})$

cache $z^{[l]}$



Backward

$da^{[l]}$ o/p : $da^{[l-1]}$

Cache ($z^{[l]}$ $dw^{[l]}$

$db^{[l]}$

* Fwd & Back Propagation.

· Fwd

→ i/p $a^{[l-1]}$ $w^{[l]}, b^{[l]}$

o/p $a^{[l]}$, cache ($z^{[l]}$)

vectorized :

$Z^{[l]} = W^{[l]} \cdot A^{[l-1]} + b^{[l]}$

$X = A^{[0]} \rightarrow \square \rightarrow \square \rightarrow \square \rightarrow \cdots$

· Back

i/p $da^{[l]}$

o/p $da^{[l-1]}$ $dw^{[l]}, db^{[l]}$

$dz^{[l]} = da^{[l]} * g^{[l]\prime}(z^{[l]})$

$dw^{[l]} = dz^{[l]} \cdot a^{[l-1]}$

$db^{[l]} = dz^{[l]}$

$da^{[l-1]} = w^{[l]T} \cdot dz^{[l]}$

$dz^{[l]} = w^{[l+1]T} dz^{[l+1]} * g^{[l]\prime}(z^{[l]})$

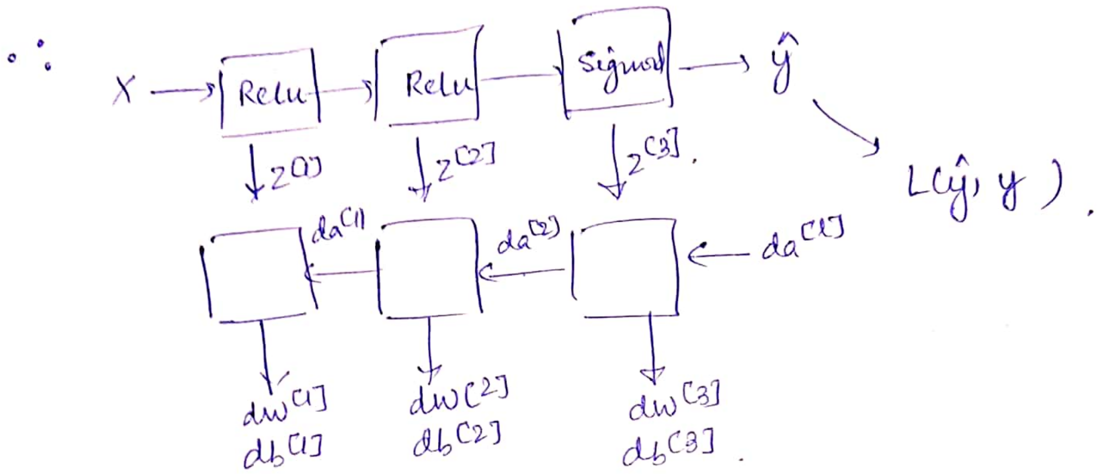
vectorized

$dZ^{[l]} = dA^{[l]} * g^{[l]'}(Z^{[l]})$

$dW^{[l]} = \frac{1}{m} dZ^{[l]} . A^{[l-1]T}$

$db^{[l]} = \frac{1}{m}$ np.sum ($dZ^{[l]}$, axis=1, keepdims = True)

$dA^{[l-1]} = W^{[l]T} . dZ^{[l]}$



* Parameters vs Hyperparameters.

W, b.

Learning rate α

\# iterations

\# hidden layer L

\# hidden unit, $n^{[1]}, n^{[2]}, \ldots$

choice of activation function

* Deep learning & human brain :