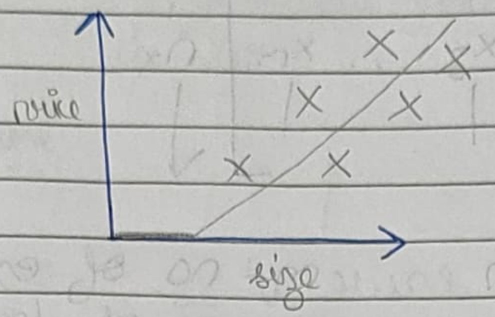
Course I

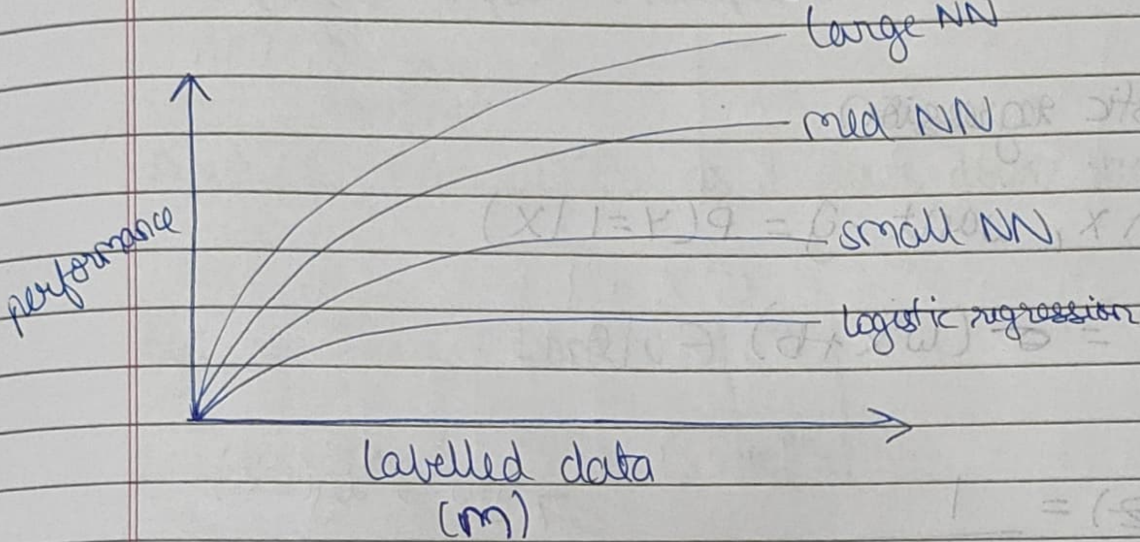
ReLU
Rectified Linear Unit



size → ○ → price
neuron

Supervised Learning

Input, Output pair given in training data.



ReLU makes learning faster than sigmoid

∵ the gradient in sigmoid tend to 0 hence lowering rate of learning

Hence, ReLU >> sigmoid as activation function of NN

$$X = \begin{array}{c} \longleftarrow m \longrightarrow \\ \begin{bmatrix} | & | & | & & | \\ x_1 & x_2 & x_3 & \cdots & x_m \\ | & | & | & & | \end{bmatrix} \end{array} \updownarrow n_x$$

x_i = feature vector

m = no of example
n = no of features.

$$Y = \underbrace{[\,y_1 \;\; y_2 \; \ldots \; y_m\,]}_{\longleftarrow m \longrightarrow} \updownarrow 1 \qquad y_i = \{0, 1\}$$

★ logistic regression

given x, want $\hat{y} = P(y=1 \mid x)$

$$\hat{y} = \sigma(w^T x + b) \in [0,1]$$

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

$y=0 \rightarrow \log(1-\hat{y})$

$y=1 \rightarrow \log(\hat{y})$

loss $= y \log \hat{y} + (1-y)\log(1-\hat{y})$ (individual case)

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

avg total loss

$$= \frac{-1}{m}\sum_{i=1}^{m}\left[y^i \log \hat{y}^i + (1-y^i)\log(1-\hat{y}^i)\right]$$

dvar → d (Final Output Var)
⇓ d (var)
in code
variable name

vectorize the code as it runs 300 times faster then non vectorize code.

SIMD: parallel computation
↳ single input multiple data.

AVOID explicit FOR LOOPS as it slows down the code

Eg: $v = \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix}$ want $u = \begin{bmatrix} e^{v_1} \\ \vdots \\ e^{v_n} \end{bmatrix}$

Non vector

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

→ this explicit for loop is removed and code is made simpler and faster

Vector

```
u = np.exp(v)
```

xplore the numpy built in function ~~library~~ lib.

$dw = np.zeros((n_x, 1))$

$dw \mathrel{+}= x^i dz^i$

$dw \mathrel{/}= m$

these changes help remove one of the explicit for loop.

☆ $z^1 = w^T x^1 + b$ $\quad a^1 = \sigma(z^1)$

$z^2 = w^T x^2 + b$ $\quad a^2 = \sigma(z^2)$

$z^3 = w^T x^3 + b$ $\quad a^3 = \sigma(z^3)$

$$X = \begin{bmatrix} | & | & | & & | \\ x_1 & x_2 & x_3 & \cdots & x_m \\ | & | & | & & | \end{bmatrix}$$

$X_{n_x \times m}$ matrix

$$Z = \underbrace{[z^1, z^2 \;\cdots\; z^m]}_{m} = w^T X + \underbrace{[b \; b \; \cdots \; b]}_{m}$$

$$= [w^T x^1 \quad w^T x^2 \;\cdots\; w^T x^m]$$

$$+ [b \;\; b \; \cdots \; b]$$

$$= \underbrace{[w^T x^1 + b \quad w^T x^2 + b \;\cdots\; w^T x^m + b]}_{1 \times m}$$

Code: $Z = np.dot(w.T, X) + b$

real number

called broadcasting { python converts to it to $1 \times m$ matrix

$A = [a^1\ a^2 \dots a^m] = \sigma(Z)$ → in assignment

★ Vectorizing Logistic Regression

$dz^i = a^i - y^i$

$dZ = [dz^1\ dz^2 \dots dz^m]$ 1×m matrix

$A = [a^1\ a^2 \dots a^m]$

$Y = [y^1\ y^2 \dots y^m]$

$\boxed{dZ = A - Y} = [a^1 - y^1\ \ a^2 - y^2 \dots a^m - y^m]$

1 step mai calculated.

np.sum(dZ) = sum of all elements.

Using vector for all operations, removes the need for explicit for loops

Also, it make the code simpler, increase readability and faster and efficient.

* Vectorized implementation.

$Z = np.dot(w.T, X) + b$

$A = \sigma(Z)$

$dZ = A - Y$

$dW = \frac{1}{m} X \, dZ^T$

$db = \frac{1}{m} \; np.sum(dZ)$

put this in a for loop to calculate multiple iteration

Broadcasting.

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

broadcasting

$(m,n) \; \frac{+}{\times}, \; (1,n) = (m,n) + (m,n) = (m,n)$

copy row 1, m times

a= np.random.radn(5)

a.shape = (5,) → rank 1 vector } X do not use this

a= np.random.randn(5,1) → USE THIS

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

log P(label in trainingset) $= \log \prod_{i=1}^{m} P(y^i|x^i)$

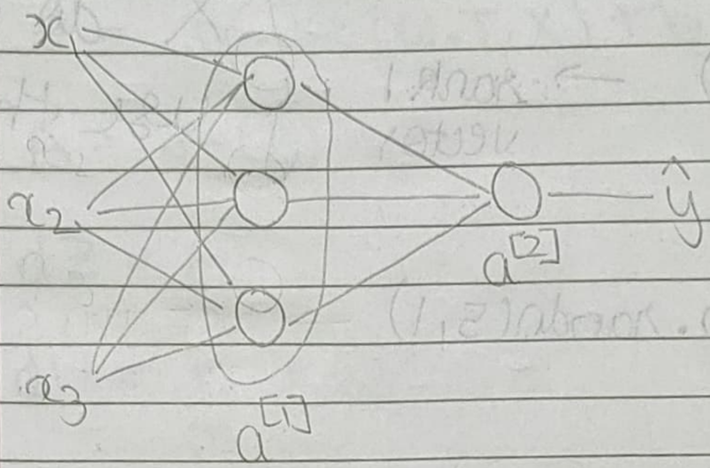
$$= \sum_{i=1}^{m} \log P(y^i|x^i)$$

$$= -\sum_{i=1}^{m} L(\hat{y}^i, y^i)$$

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

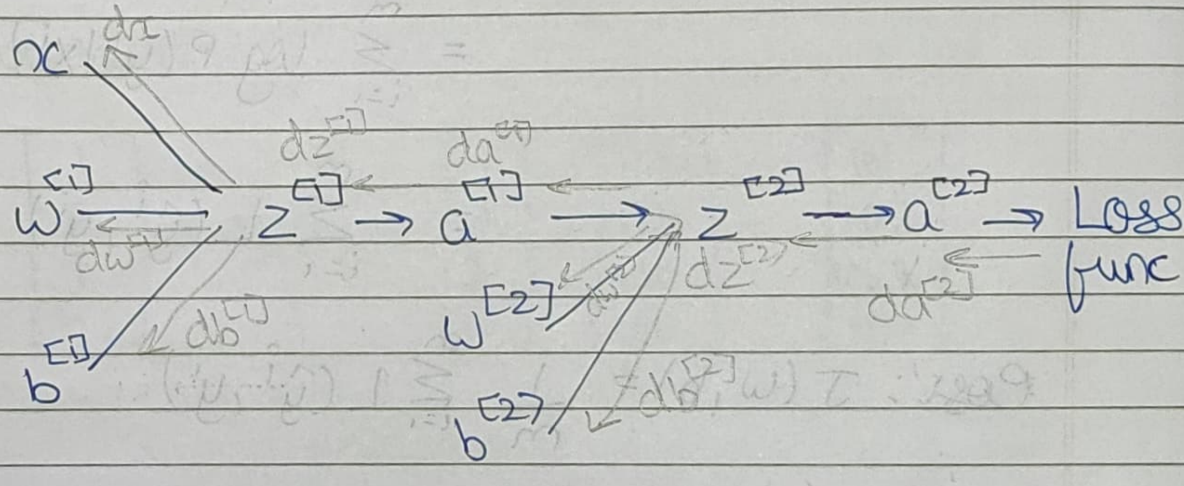
The training set is assumed to be i.i.d
↓
independent and indentically distributed.

Overview

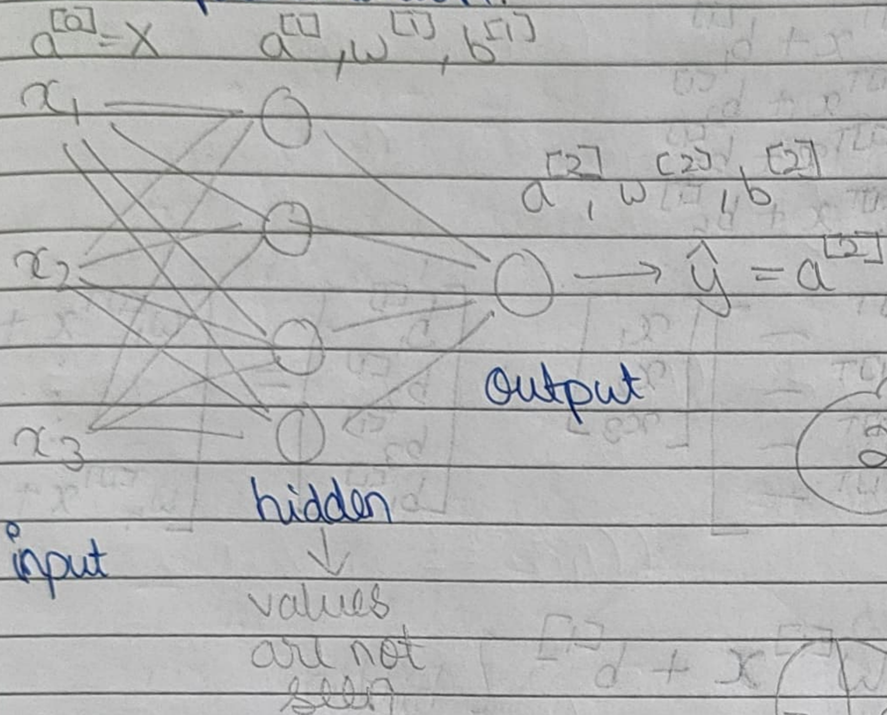


[1] → layer 1
[2] → layer 2

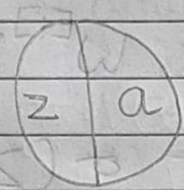
(1) → example 1
(2) → example 2

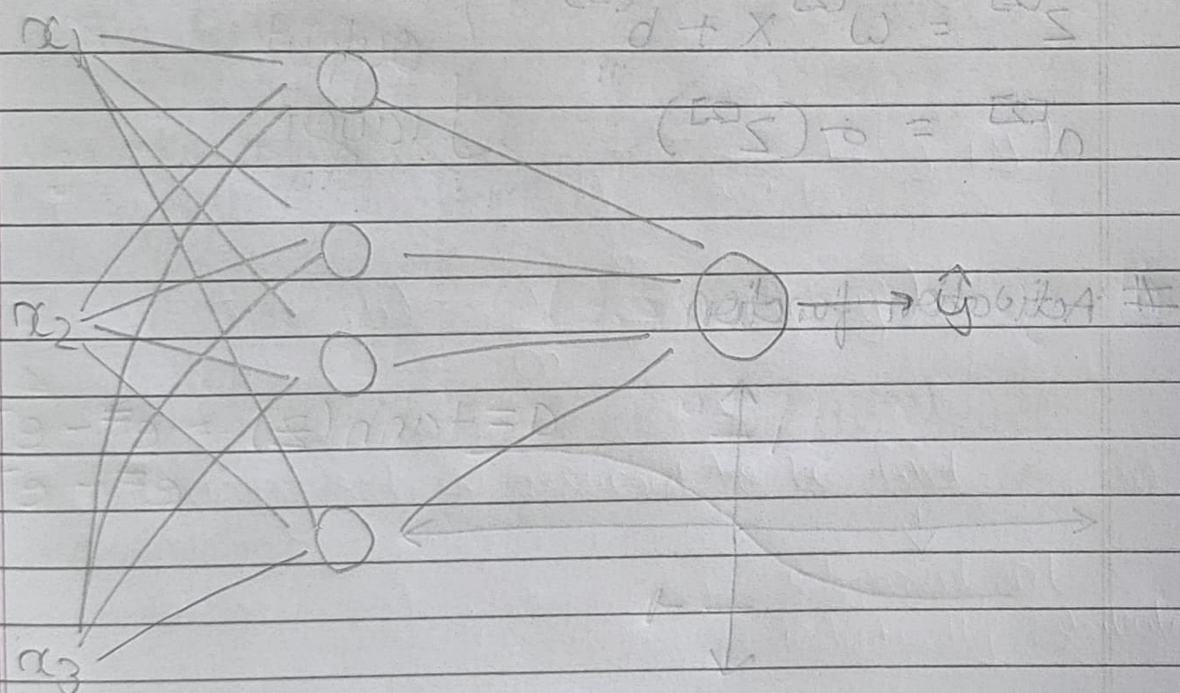


☆ NN Representation.



2 layer NN





$$z_i^{[l]} = w_i^{[l]T} x + b_i^{[l]}$$

$$a_i^{[l]} = \sigma(z_i^{[l]})$$

l = layer i = number

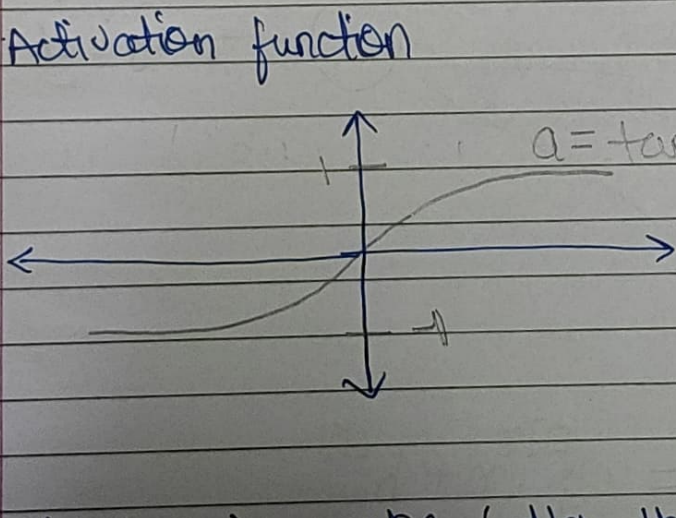
$$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]}$$
$$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]}$$
$$z_3^{[1]} = w_3^{[1]T} x + b_3^{[1]}$$
$$z_4^{[1]} = w_4^{[1]T} x + b_4^{[1]}$$

$$Z^{[1]} = \begin{bmatrix} - & w_1^{[1]T} & - \\ - & w_2^{[1]T} & - \\ - & w_3^{[1]T} & - \\ - & w_4^{[1]T} & - \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix} = \begin{bmatrix} w_1^{[1]T} x + b_1^{[1]} \\ \vdots \\ \vdots \\ w_4^{[1]T} x + b_4^{[1]} \end{bmatrix}$$

$$\left.\begin{array}{l} z^{[1]} = W^{[1]} x + b^{[1]} \\ a^{[1]} = \sigma(z^{[1]}) \end{array}\right\} \text{hidden layer}$$

$$\left.\begin{array}{l} z^{[2]} = W^{[2]} x + b^{[2]} \\ a^{[2]} = \sigma(z^{[2]}) \end{array}\right\} \text{output layer}$$

Activation function



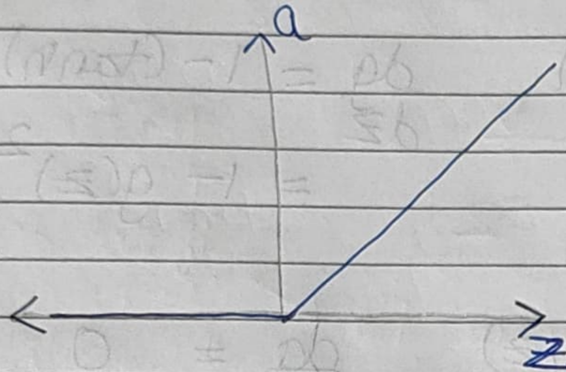
$$a = \tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

$\tanh(z)$ works better than $\sigma(z)$

sigmoid for binary classification for output layer

tanh gives derivative as 1 for large values and hence better than sigmoid but still not good enough

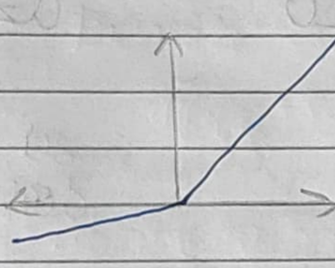
$\therefore$ ReLU = max(z,0) is used



$a = \max(0, z)$

derivative
if $z > 0 \Rightarrow 1$
if $z < 0 \Rightarrow 0$

Leaky ReLU



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

better than ReLU but doesnt make huge difference

hidden layer mai linear use nhi karte because uska output linear hi dega so its redundant

☆ Derivative of activation function.

a] $g(z) = \dfrac{1}{1+e^{-z}}$ $\qquad \dfrac{dg}{dz} = \dfrac{+e^{-z}}{(1+e^{-z})^2}$

$= \sigma(z)$ $\qquad = \dfrac{1}{1+e^{-z}}\left(1 - \dfrac{1}{1+e^{-z}}\right)$

$\qquad = g(z)(1-g(z))$

b] $g(z) = \tanh(z)$ $\qquad \dfrac{dg}{dz} = 1-(\tanh)^2$

$\qquad = 1 - g(z)^2$

c] $g(z) = \max(0, z)$ $\qquad \dfrac{dg}{dz} = \begin{cases} 0 & z<0 \\ 1 & z>0 \\ \text{undefined} & z=0 \end{cases}$

ReLU

set to 1 in code
doesnt affect anything

d] $g(z) = \max(0.01z, z)$ $\qquad \dfrac{dg}{dz} = \begin{cases} 1 & z \geq 0 \\ 0.01 & z<0 \end{cases}$

leaky ReLU

classmate
Date
Page

★ gradient descent for nural networks:

Forward:

$$Z^{[1]} = W^{[1]}X + b^{[1]}$$
$$A^{[1]} = g^{[1]}(Z^{[1]})$$

$$Z^{[2]} = W^{[2]}X + b^{[2]}$$
$$A^{[2]} = g^{[2]}(Z^{[2]})$$

Back:

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

$$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}$$

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

$$dZ^{[1]} = W^{[2]T} dZ^{[2]} * g^{[1]'}(Z^{[1]})$$

$$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$$

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

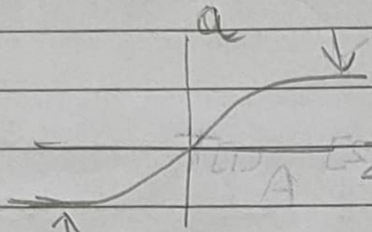
* Random initialization is better than 0

bcz all nodes will perform ~~same~~ same calc and doesn't make sense.

so random initialization.

Eg:

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



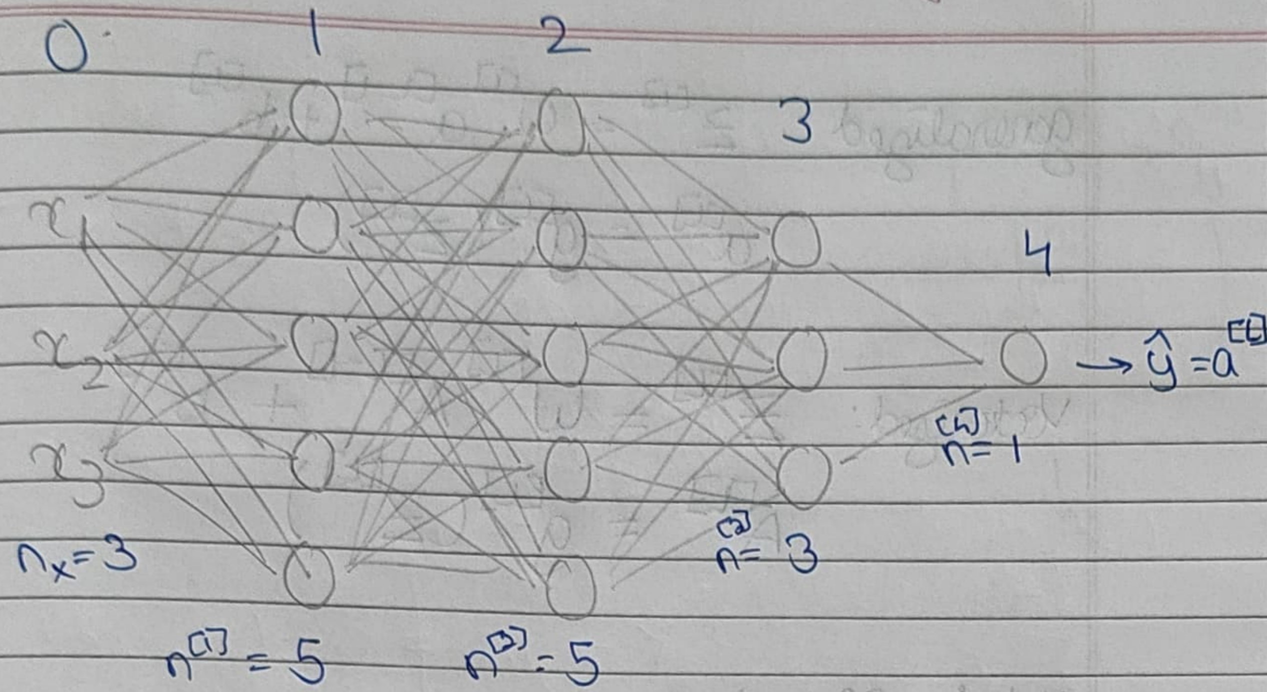
to minimize the parameters bcz if not done then learning slows as the gradient reaches 1

issue when using tanh or sigmoid

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

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

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



$L = 4$ (# layers)

$n^{[l]}$ = nodes in layer l

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

$x = a^{[0]}$

★ Forward Propogation for example x

$x: \quad z^{[1]} = w^{[1]} x + b^{[1]} = w^{[1]} a^{[0]} + b^{[1]}$

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

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

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

$\vdots$

$z^{[4]} = w^{[4]} a^{[3]} + b^{[3]}, \quad a^{[4]} = g^{[4]}(z^{[4]}) = \hat{y}$

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

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

Vectorized: $Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]}$

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

★ Matrix Dimensions

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

$$dZ^{[l]} = Z^{[l]} = (n^{[l]}, m)$$

$$dA^{[l]} = a^{[l]} = (n^{[l]}, m)$$

$$b^{[l]} = (n^{[l]}, 1) \xrightarrow{\text{broadcasting}} (n^{[l]}, m)$$

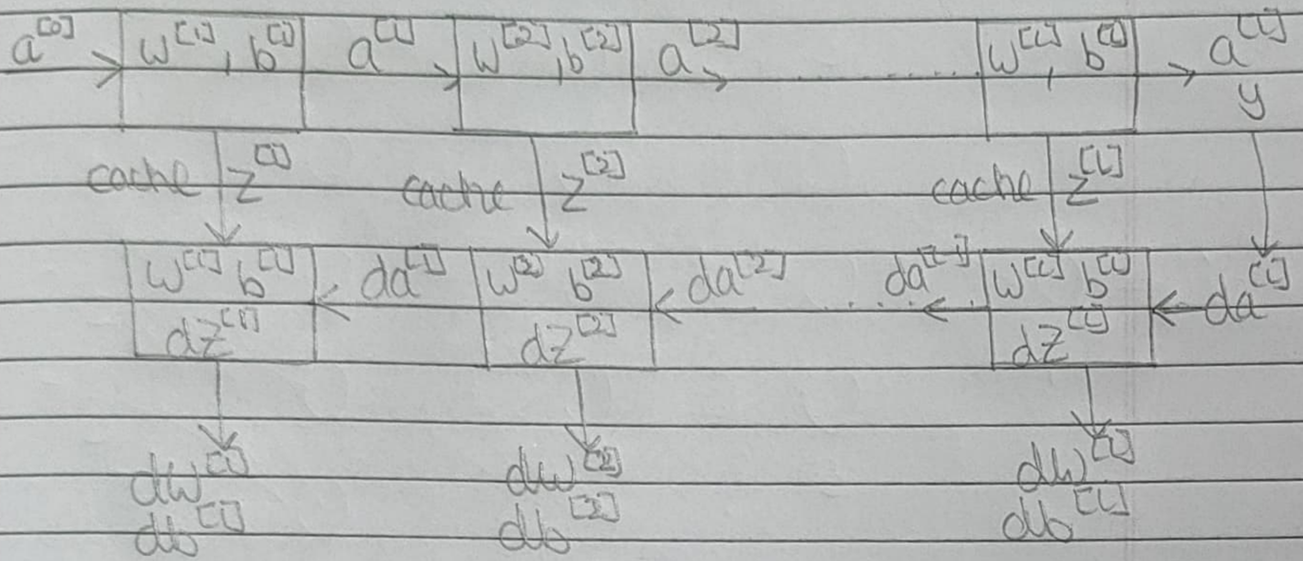
$$dW^{[l]} = (n^{[l]}, n^{[l-1]})$$

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

generalized form

vectorized implementation

★ Deep learning starts with simpler function and slowly making it more and more complex and hence deeper NN start performing better and better than the shallower ones.



$$W^{[l]} := W^{[l]} - \alpha \, dW^{[l]}$$
$$b^{[l]} := b^{[l]} - \alpha \, db^{[l]}$$

☆ Hyperparameters : α (learning rate)
no of iterations
no of hidden layers
no of nodes in a layer
activation functions