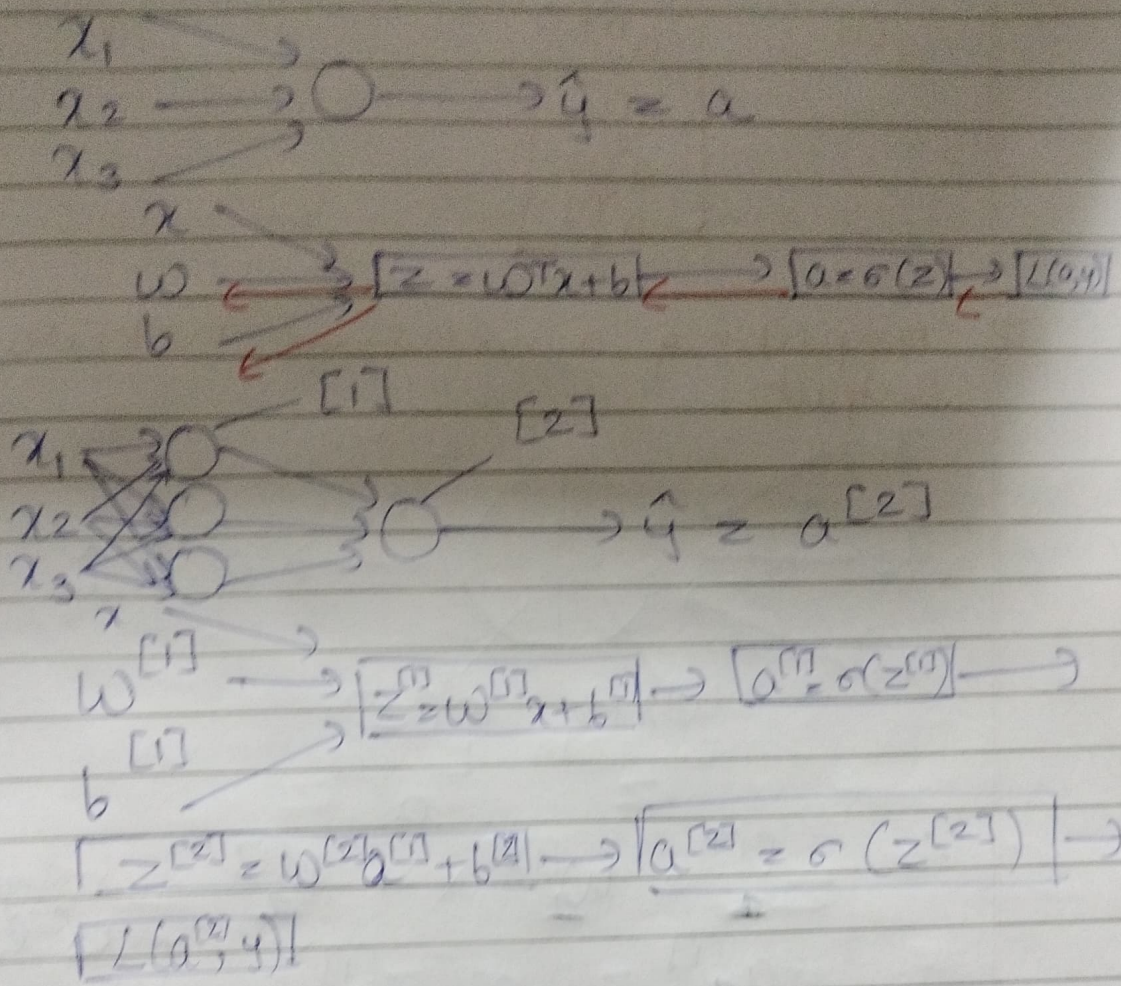
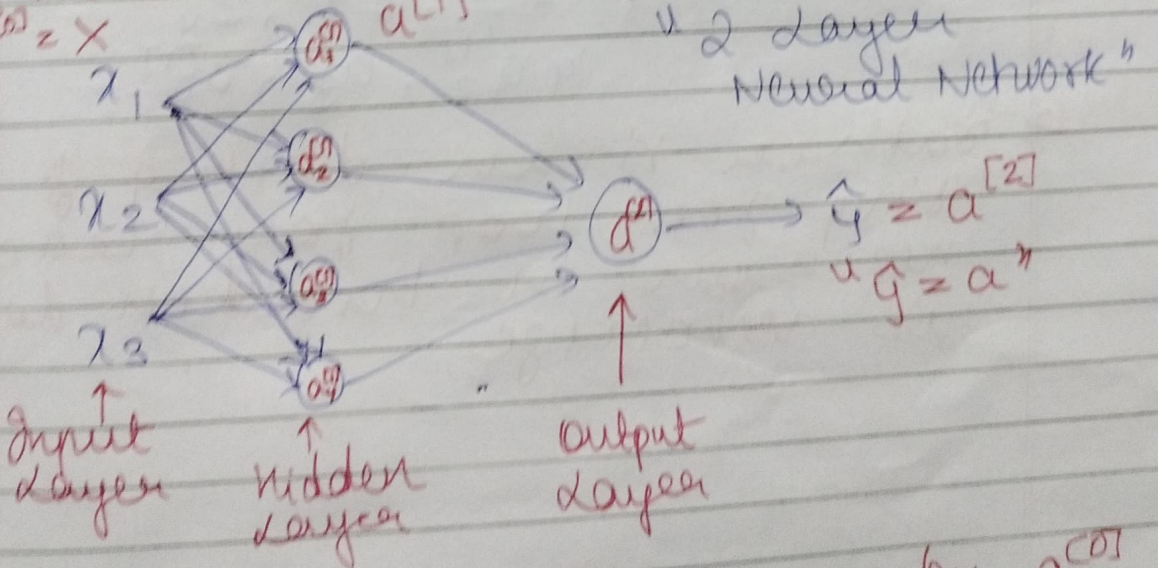
Shallow neural Network



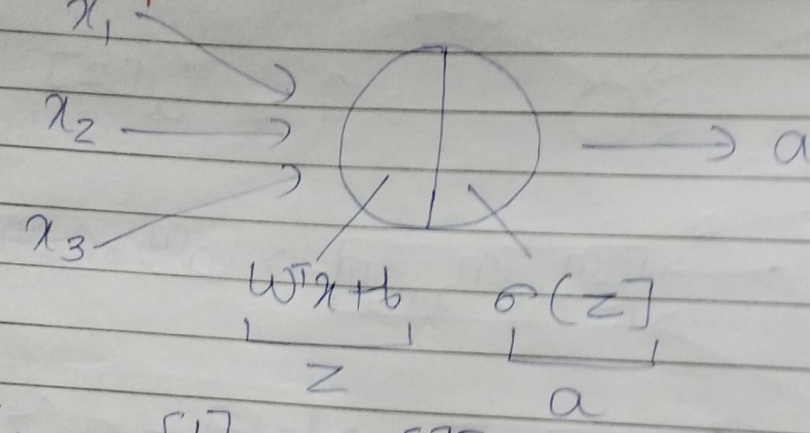
Neural Network representation



activations of input layer by $a^{[0]}$
it hidden layer $a^{[1]}$

$$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix}$$

Computing a neural network's output



$\therefore z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]}$ } similarly $b^{[2]}, b^{[3]}, b^{[4]}$ & $w^{[2]}, w^{[3]}, w^{[4]}$

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

similarly $z_2^{[1]}, z_3^{[1]}, z_4^{[1]}$

& $a_2^{[1]}, a_3^{[1]}, a_4^{[1]}$

→ rather than doing it using for loop for each neuron in hidden layer we vectorize

$W^{[1]}$ — $(w^{[1]T})$

$$\begin{bmatrix} \cdots & w_1^{[1]} & \cdots \\ \cdots & w_2^{[1]} & \cdots \\ \cdots & w_3^{[1]} & \cdots \\ \cdots & w_4^{[1]} & \cdots \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix}$$

$(4,3)$ $\quad (3\times1)$

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

$$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix} = \sigma(z^{[1]})$$

Given input x:

$\rightarrow$ $z^{[1]}_{(4,1)} = W^{[1]}_{(4,3)} x_{(3,1)} + b^{[1]}_{(4,1)}$

$\rightarrow$ $a^{[1]}_{(4,1)} = \sigma(z^{[1]}_{(4,1)})$

$\rightarrow$ $z^{[2]}_{(1,1)} = W^{[2]}_{(1,4)} a^{[1]}_{(4,1)} + b^{[2]}_{(1,1)}$

$\rightarrow$ $a^{[2]}_{(1,1)} = \sigma(z^{[2]}_{(1,1)})$

Vectorizing across multiple examples

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

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

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

$\vdots$

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

$a^{[2](i)}$ — training eg. i

a — layer no.

∴ for i = 1 to m

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

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

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

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

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

as seen earlier

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

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

no of neurons in hidden layer

m (training egs)

PAGE No.
DATE / /

Justification for vectorized implementation

$$z^{[1](1)} = w^{[1]} x^{(1)} + b^{[1]} \to 0$$

$$z^{[1](2)} = w^{[1]} x^{(2)} + b^{[1]} \to 0$$

$$z^{[1](3)} = w^{[1]} x^{(3)} + b^{[1]} \to 0$$

$$W^{[1]} = \begin{bmatrix} — \\ — \\ — \\ — \end{bmatrix} \qquad W^{[1]} x^{(1)} = \begin{bmatrix} \cdot \\ \cdot \\ \cdot \\ \cdot \end{bmatrix}$$

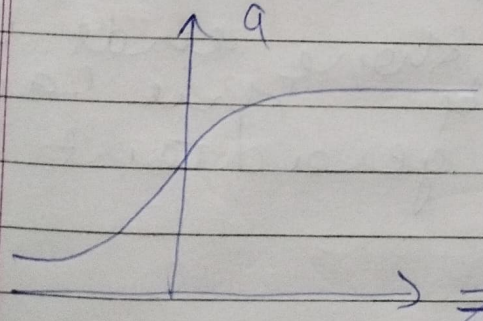
$$W^{[1]} x^{(2)} = \begin{bmatrix} \cdot \\ \cdot \\ \cdot \\ \cdot \end{bmatrix} \qquad W^{[1]} x^{(3)} = \begin{bmatrix} \cdot \\ \cdot \\ \cdot \\ \cdot \end{bmatrix}$$

$$W^{[1]} \begin{bmatrix} | & | & | \\ x^{(1)} & x^{(2)} & x^{(3)} \\ | & | & | \end{bmatrix} = \begin{bmatrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix}$$

$$= \begin{bmatrix} | & | & | \\ z^{[1](1)} & z^{[1](2)} & z^{[1](3)} \\ | & | & | \end{bmatrix} = Z^{[1]}$$

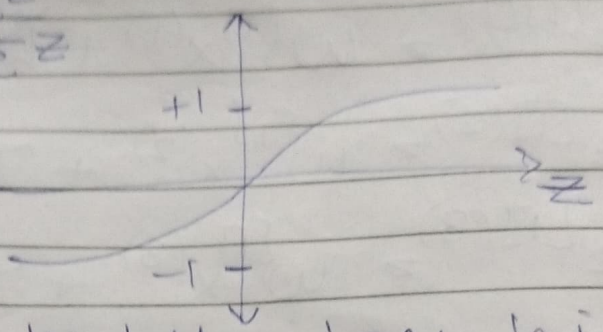
Activation function

→ So far we have used sigmoid function



⇒ A function that works better than sigmoid is tanh(z) = g(z)

$$= \frac{e^z - e^{-z}}{e^z + e^{-z}}$$



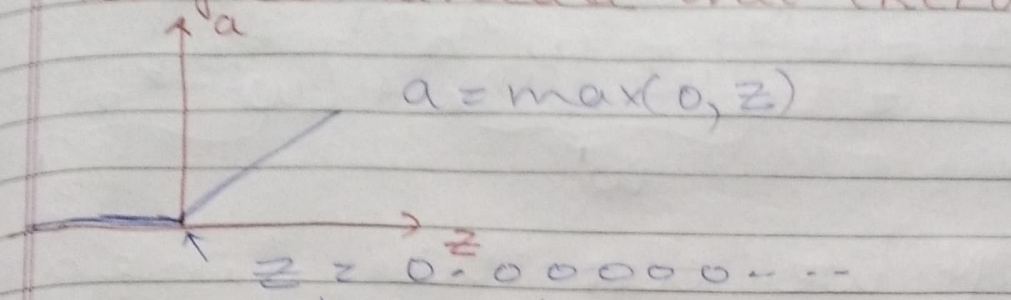
This works better for hidden units than σ(z) as values btw +1 & -1 the mean of activations that come out of hidden layer are closer to 0 mean. This makes learning for next layer a little bit easier

⇒ tanh(z) not used for O/p layer instead sigmoid used as O/p must be btw 0 & 1

⇒ Sometimes activations are diff for diff layers eg $g^{[1]}(z)$, $g^{[2]}(z)$ etc

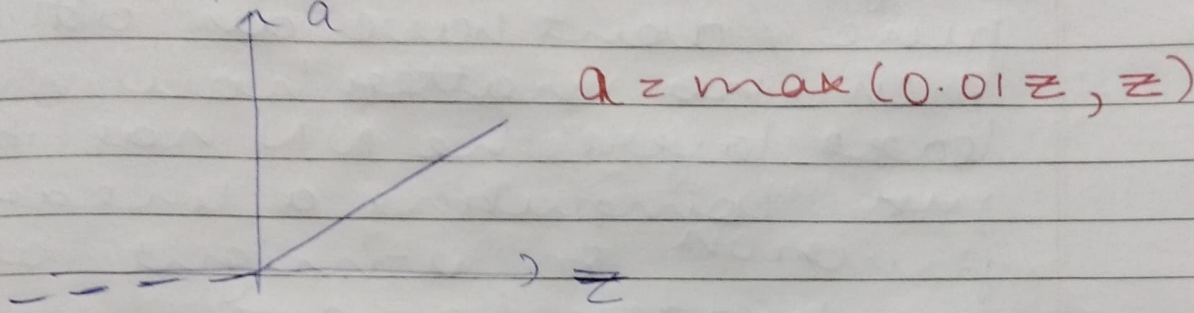
⇒ downside of both sigmoid & tanh func is that if z is very large or small gradient of derivative of slope becomes very small & slope ends up close to 0 & this can slow down gradient descent.

⇒ ∴ A more popular choice is rectified linear Unit (ReLU)



→ one disadvantage is that when z < 0 derivative is ~~zero~~

→ ∴ another version called leaky ReLU ~~was~~ used



Sigmoid — Almost never use only use in O/P layer for binary classification

tanhx — used in hidden layers in pref to σ

ReLU — when not sure what to use

Why do we need activation functions

⇒ If no activation functions

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

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

$= (w^{[2]}w^{[1]})x + (w^{[2]}b^{[1]} + b^{[2]})$

$= w'$ $\qquad b'$

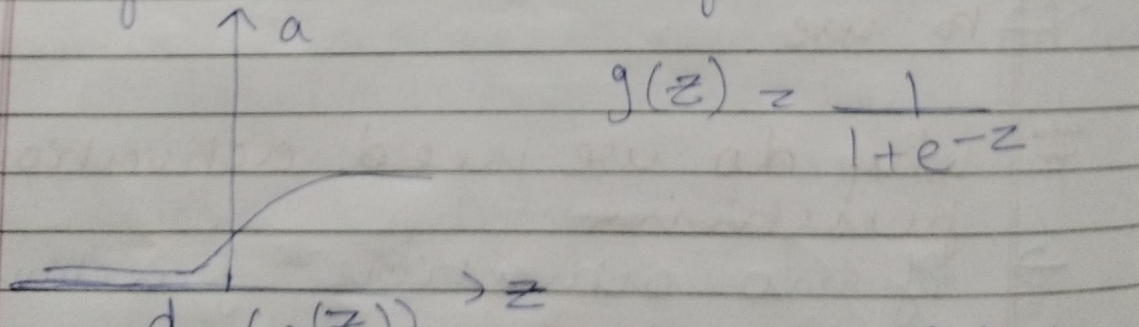
→ If we use this linear activation function the network is just outputting a linear func of input

→ If u use a linear activation func / don't have an activation func then no matter how many layers the NN has it all just computing a linear activation function so u might as have no hidden layers

→ Model is no more expressive than standard logistic regression without any hidden layer

Derivatives of activation functions

1) Sigmoid activation function

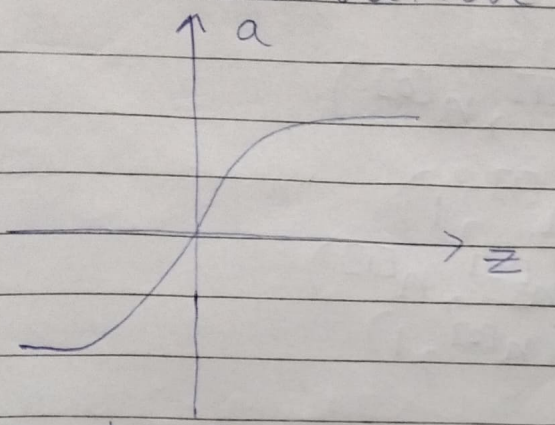


$\frac{d}{dz}(g(z))$ = slope of $g(x)$ at z

$$= \frac{e^{-z}}{(1+e^{-z})^2} = g(z)(1-g(z))$$

$$= a(1-a)$$

2) Tanh activation function



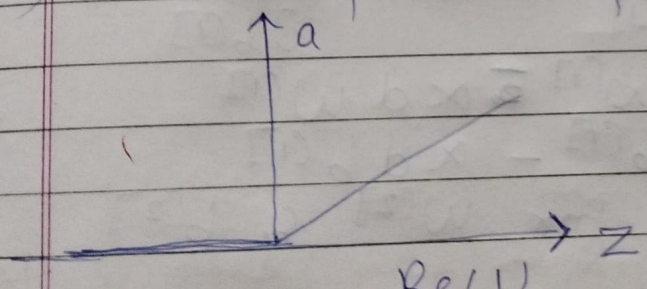
$$g(z) = \tanh(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$$

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

$\therefore a = g(z)$

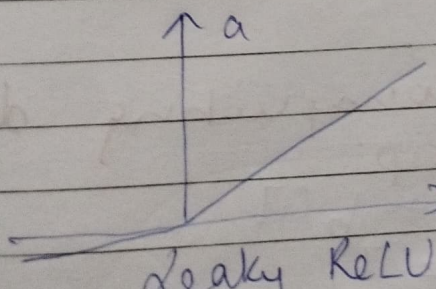
$\therefore g'(z) = 1-a^2$

3) ReLU & Leaky ReLU



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

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



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

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

Gradient descent for neural network with 1 hidden layer

→ Parameters $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}$

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

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

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

$w^{[2]} = (n^{[2]}, n^{[1]})$

$b^{[2]} = (n^{[2]}, 1)$

→ cost function: $J(w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]})$

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

$\uparrow a^{[2]}$

→ Gradient descent:

compute predictions $\hat{y}^{(i)}$ for $i = 1$ to m

Repeat until parameters look like they are converging {

$dw^{[1]} = \frac{dJ}{dw^{[1]}}$ $\quad db^{[1]} = \frac{dJ}{\partial b^{[1]}}$

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

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

Similarly for $w^{[2]}$ & $b^{[2]}$ }

Formula for computing derivatives

i) forward prop:

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

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

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

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

PAGE NO.
DATE / /

back propagation

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

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

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

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

$(n^{[2]}, 1)$ ← prevents rank 1 matrix

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

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

* ← element wise product

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

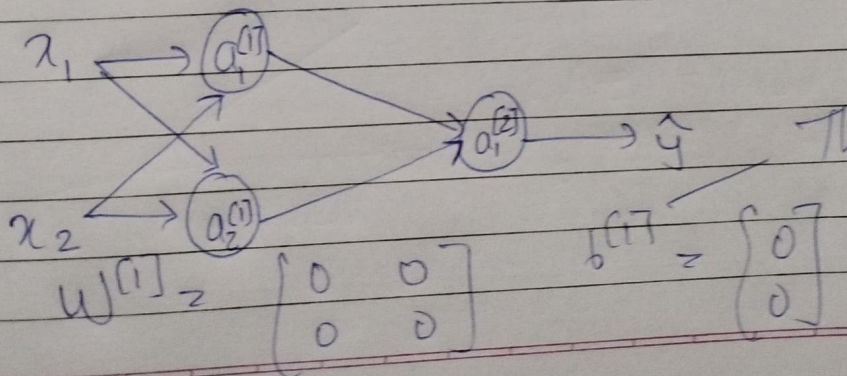
$$db^{[1]} = \frac{1}{m} np.sum(dz^{[1]}, axis=1, keepdims=True)$$

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

$dz^{[2]} = (n^{[2]}, m)$

$dz^{[1]} = (n^{[1]}, m)$

Random Initialization



→ problem with initializing w to 0 is that
$a_1^{[1]}$ & $a_1^{[2]}$ will be equal

& in back prop
$dz_1^{[1]} = dz_2^{[1]}$

→ We want diff layers to compute diff functions ∴ we initialize randomly

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

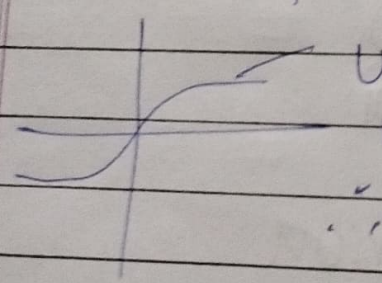
$b^{[1]} = np.zeros((2,1))$

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

$b^{[2]} = 0$

we want to initialize wt to very small val

→ if w is very big, z will be very big :-



very likely to end up at the flat part
∴ slope is very small