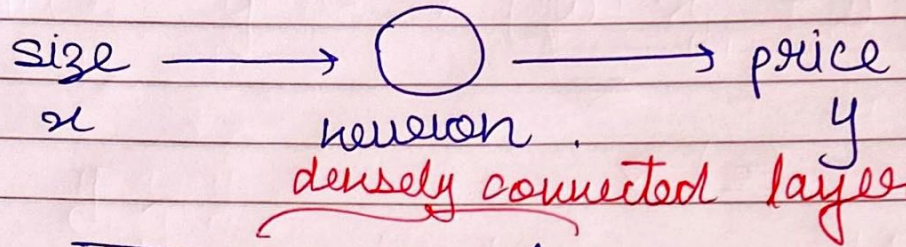
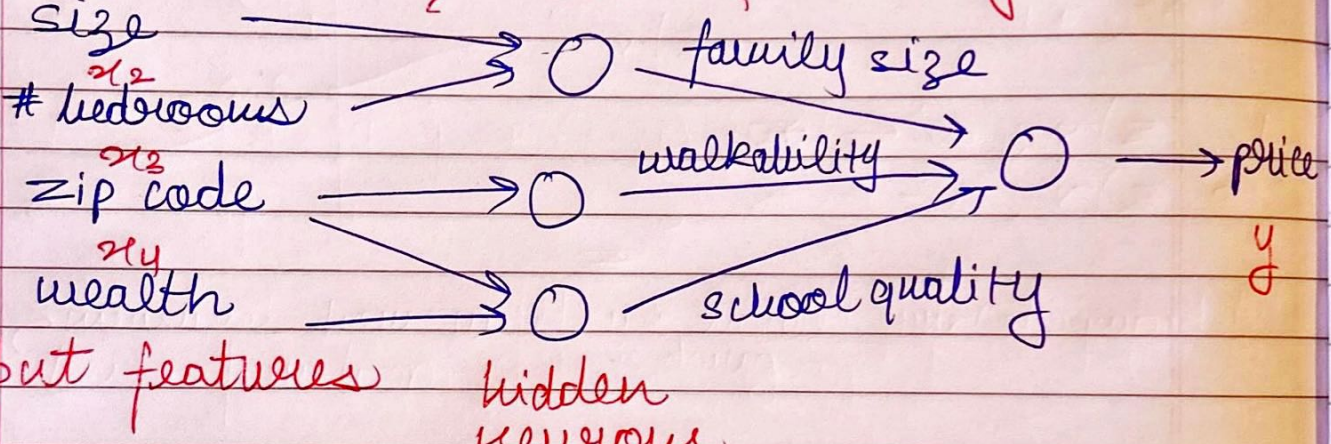
19/7/24

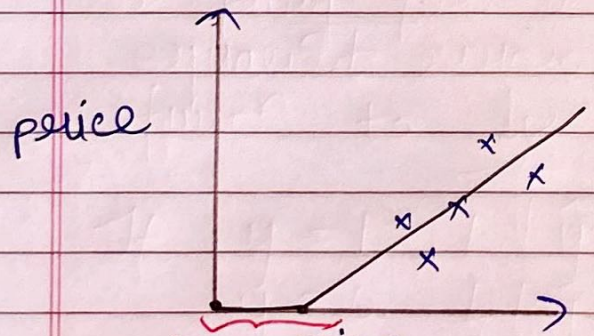
DEEP LEARNING

* What is a neural network?
Stacking together many neurons gives us a neural network.



densely connected layer.





Relu function
(Rectified Linear Unit)

∵ price cannot be negative

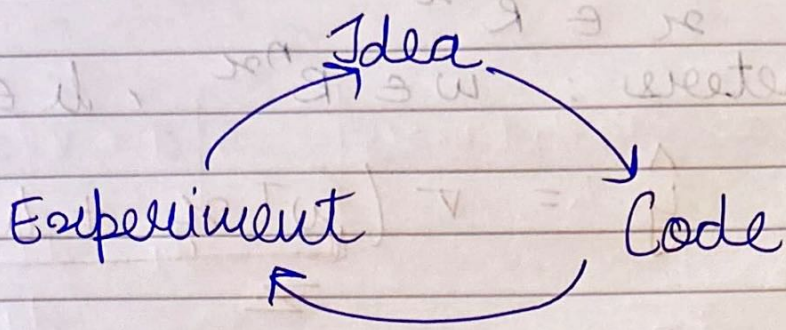
* Common Neural Networks

CNN → images

RNN → sequential data

Hybrid / Custom → advanced models.

Structured Data	Unstructured Data
Proper data set, every feature is defined	Images, audio, random texts.



as building an effective NN is an iterative process.

* BINARY CLASSIFICATION:
input image : 64×64.

feature vector : matrix $X = \begin{bmatrix} 255 \\ 231 \\ \vdots \\ 255 \\ 231 \\ \vdots \end{bmatrix} = 64 \times 64 \times 3$ (∵ RGB)

$n_x = 12288$ (here)

$x \longrightarrow y$

image of cat ⊙/1.

m = no. of training examples

Basic Notations

* $X \in R^{n_x \times m}$ ($n_x \times m$ dimensional matrix)

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

$\leftarrow m \rightarrow$

* $Y \in R^{1 \times m} = [y^{(1)} \; y^{(2)} \; \ldots \; y^{(m)}]$

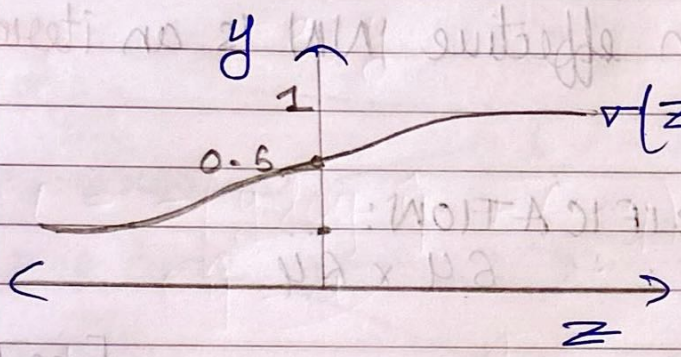
* LOGISTIC REGRESSION:

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

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

Parameters : $w \in R^{n_x}$, $b \in R$.

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



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

$z \to \infty$; $\sigma(z) = 1$.

$z \to -\infty$; $\sigma(z) \approx 0$.

* Loss function : → measures how well our model func^n (sigmoid here) is doing

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

→ opting for a sigmoid function over squared error here as we want the loss func^n to be convex.

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

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

* Cost Function : → measures how well parameters w, b are doing

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

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

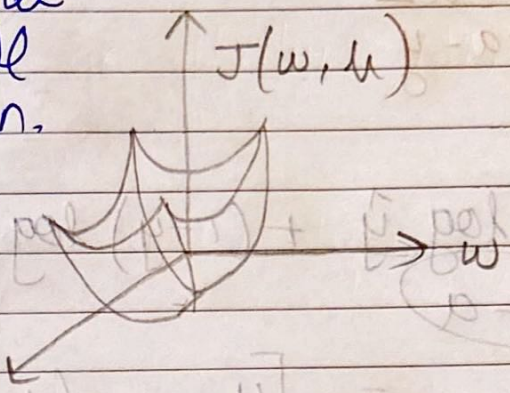
GRADIENT DESCENT:

→ want to find w, b that minimizes $J(w, b)$

$$w = w - \alpha \frac{\partial J(w,b)}{\partial w}$$

$$b = b - \alpha \frac{\partial J(w,b)}{\partial b}$$

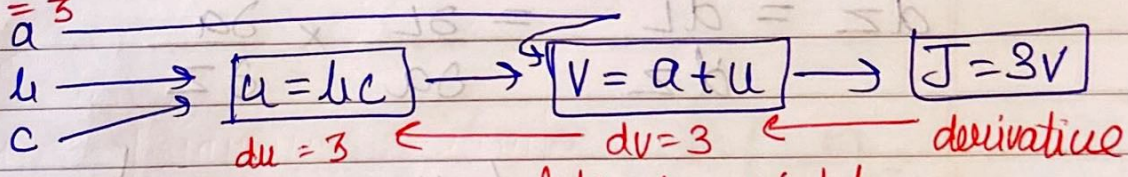
With any initial value, you will reach the min. (path maybe diff.).



Why gradient descent?
→ it is a convex func°. Thus, has one minima only (global)

COMPUTATION GRAPH:

$J(a, b, c) = 3(a + bc)$

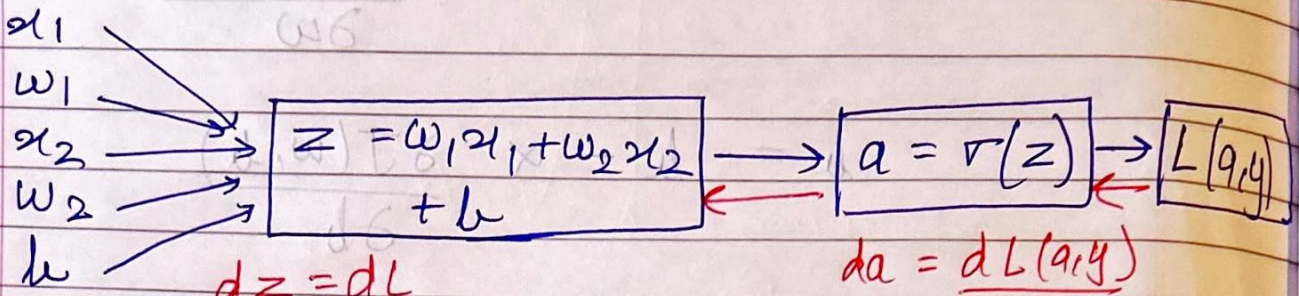


We observe, through a left to right pass we computed the value of J.

To compute derivatives, we move from right to left, i.e., in technical terms use backpropagation to calculate derivative.

★ dvar in code ≡ J (here).
in code, $\frac{dJ}{da} = da = 3$ (here)

Gradient Descent for Logistic Regression (using computation graph)



$dz = \frac{dL}{dz} = a - y$

$da = \frac{dL(a,y)}{da} = \frac{-y}{a} + \frac{1-y}{1-a}$

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

$\hat{y} = a$

$da = \frac{\partial L}{\partial a} = -\left[\frac{y}{a} + \frac{-(1-y)}{(1-a)}\right] = -\frac{y}{a} + \frac{(1-y)}{(1-a)}$

$dz = \frac{dL}{dz} = \frac{\partial L}{\partial a} \times \frac{\partial a}{\partial z}$

$= \left(\frac{-y}{a} + \frac{1-y}{(1-a)}\right) \frac{\partial}{\partial z} \sigma(z)$

$\frac{\partial}{\partial z} \frac{1}{1+e^{-z}}$

$= \left(\frac{-y}{a} + \frac{1-y}{(1-a)}\right) \times$

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

$a = \frac{1}{1+e^{-z}} \Rightarrow e^{-z} = \frac{1}{a} - 1$

$$\frac{\partial a}{\partial z} = a^2 \frac{(1-a)}{a} = a(1-a).$$

$$\therefore dz = \frac{dL}{dz} = \frac{\left(-\frac{y}{a} + \log + a - a\log\right) a(1-a)}{a(1-a)}$$

$$dz = a - y$$

* Logistic Regression on m examples
  - using for loop

for i:1 → m

$z^{(i)} =$

$a^{(i)} =$

$J_t =$

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

$dw_1 \mathrel{+}=$

$dw_2 \mathrel{+}=$

$db \mathrel{+}=$

] considering only w_1 & w_2

$J /= m$

$dw_1 /= m$; $dw_2 /= m$; $db /= m$

~~Usi~~ (greater datasets are involved)

As we advance in DL, for loops can be time-consuming, thus we use vectorization.

* VECTORIZATION

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

$w^T x$

* Neural Network Programming Guideline.
- Whenever possible, avoid explicit for loops.
- Use built in functions of Numpy & Python, makes the code more efficient.

* Vectorizing Logistic Regression :

$$X \text{ (training set matrix)} = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ | & | & & | \end{bmatrix} \Rightarrow X \in \mathbb{R}^{n_x \times m}$$

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

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

w.T: transpose of w; b: real number

Broadcasting in Python.

Gradient Descent Pseudo Code.

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

We can use a for loop here IF we want to implement Gradient Descent multiple times.

* BROADCASTING IN PYTHON:

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

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

$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix} + \begin{bmatrix}100\\200\end{bmatrix} \begin{matrix}100 & 100\\200 & 200\end{matrix} = \begin{bmatrix}101 & 102 & 103\\204 & 205 & 206\end{bmatrix}$$

can be +/-/*/ /

General Principle: $\frac{+}{*}$ $(1, n) \rightsquigarrow (m, n)$

$/$ $(m, 1) \rightsquigarrow (m, n)$

- main form of broadcasting in neural network programming when which is useful in

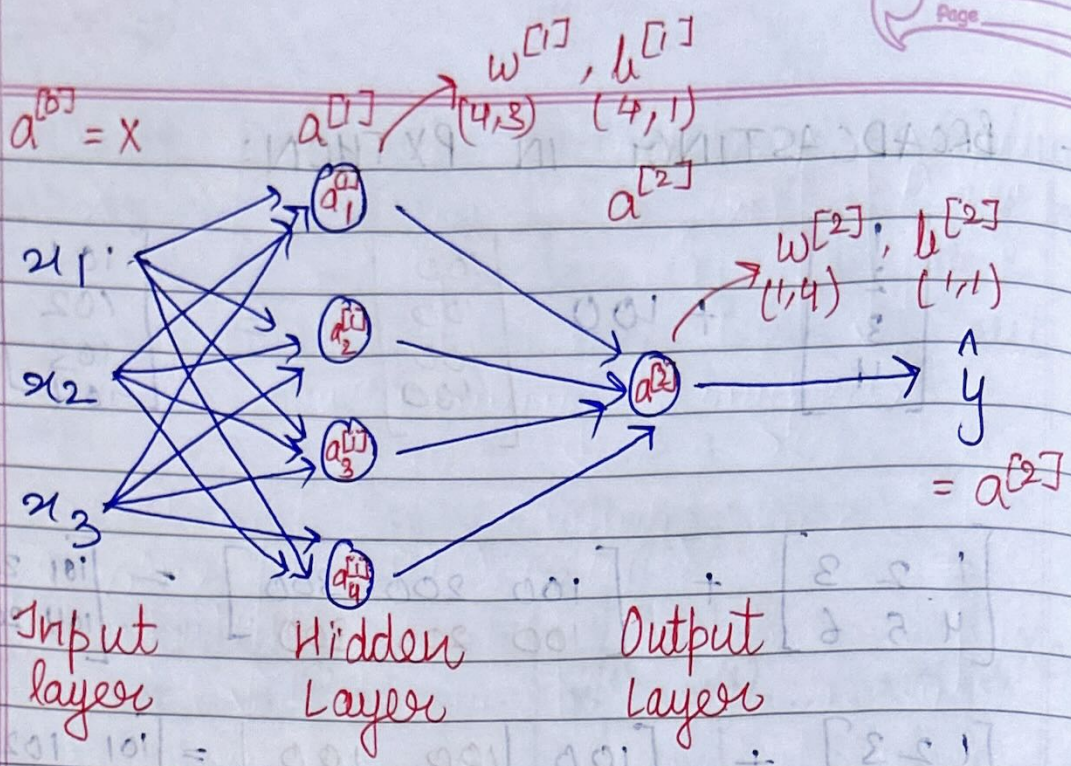
Tip: with python/numpy vectors don't use 1D array, rather use:

a = np.random.randn(5,1)

here a.shape = (5,1) & NOT (5)

* Neural Network Representation:

- The things in the hidden layers are not visible to the training set. (i.e., the value of y is not visible to the NN while computing y)



- 2 layer NN (considered conventionally, as input layer does not count as an official layer).

$$a^{[1]}_{(n_x,\,1)} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix}$$

for each feature (x_1, x_2, x_3) the value of the parameters (w_1, w_2, w_3, w_4) is different. Thus, value of z is also different

first parameter of first training example

second

$$\begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \\ w_{41} & w_{42} & w_{43} \end{bmatrix}_{4\times 3} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}_{3\times 1}$$

↳ 4th parameter of 2nd training eg

Note: $z = wx + b$ ($z_n \propto w_n$ not x)

$$\begin{bmatrix} w_{11}x_1 + w_{12}x_2 + w_{13}x_3 \\ w_{21}x_1 + w_{22}x_2 + w_{23}x_3 \\ w_{31}x_1 + w_{32}x_2 + w_{33}x_3 \\ w_{41}x_1 + w_{42}x_2 + w_{43}x_3 \end{bmatrix} = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{bmatrix}$$

4th parameter of 1st i/p of 2nd + ... + 4th of mth

$$+ \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$

nth parameter of mth training example × mth x.

$[w_{11} \quad w_{12} \quad w_{13} \quad w_{14}]$

$$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]}, \quad a_1^{[1]} = \sigma(z_1^{[1]})$$
$$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]}, \quad a_2^{[1]} = \sigma(z_2^{[1]})$$
$$z_4^{[1]} = w_4^{[1]T} x + b_4^{[1]}, \quad a_4^{[1]} = \sigma(z_4^{[1]})$$

$W^{[1]}$ (4,3) $(n_x, 4)$

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

(4,1) $(n_x, 1)$ $b^{[1]}$ (4,1)

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

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

* Vectorizing across multiple examples:

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

$$z^{[1]} = W^{[1]} X + b^{[1]}$$
$$A^{[1]} = \sigma(z^{[1]})$$
$$z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$$
$$A^{[2]} = \sigma(z^{[2]})$$

} Forward Propagation

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

n features

m training examples →

$z^{[l]}$ = nth value of corresponding m

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

$A^{[l]}$ = n_{hl} hidden units

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

iterating all training examples (m) horizontally →

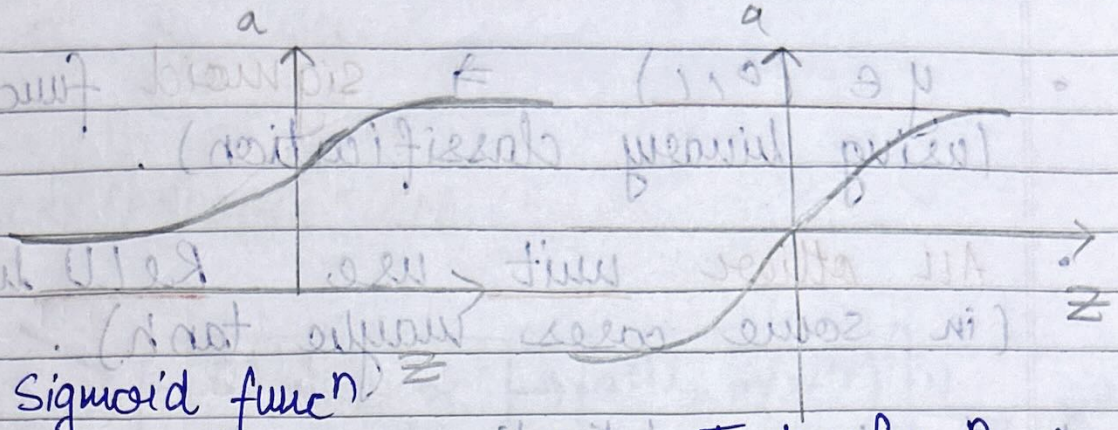
Eg: $a_1^{[l](1)}$ ⇒ first hidden unit of the first training example

$a_2^{[l](1)}$ ⇒ second hidden unit of the first training example.

$a_1^{[l](2)}$ ⇒ first hidden unit of the second training example.

★ $a_n^{[l](i)}$ ⇒ n^{th} hidden unit of the i^{th} training example.

ACTIVATION FUNCTIONS :



Sigmoid func^n

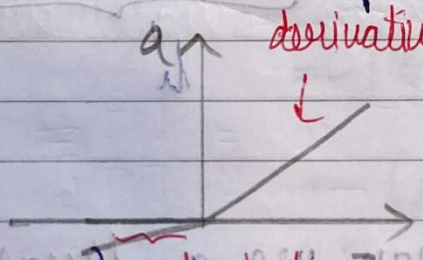
$$a = \frac{1}{1+e^{-z}}$$

Tanh func^n

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

tanh >> sigmoid **except** for when $y \in (0,1)$.

- The activation functions of different layers can be different
- Disadvantage : if ~~$y \to \infty$~~, $z \to \infty$ or $z \to -\infty$, the gradient/ derivative becomes very slow, thus affecting gradient descent.
- Thus another function that's used is: ReLU (Rectified Linear Unit) func^n :



$a = \max(0, z)$

thus derivative is 0 or 1.

Preferably, leaky ReLU > ReLU

(∵ z is −ve).

→ leaky ReLU $a = \max(0.01z, z)$.

* Rules to select Activation Functions:

- $y \in (0,1)$ ⇒ sigmoid func^n (using binary classification).
- All other unit use ReLU by default (in some cases maybe tanh).

* Linear Activation Function is almost futile in a hidden NN, as we're proceeding in the NN but not performing any complex func^n.

* Given x:

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

$$a^{[2]} = z^{[2]} = W^{[2]} a^{[1]} + b^{[2]}$$
$$= W^{[2]} \left(W^{[1]} x + b^{[1]} \right) + b^{[2]}$$
$$= \underbrace{\left(W^{[2]} W^{[1]}\right)}_{W'} x + \underbrace{\left[W^{[2]} b^{[1]} + b^{[2]}\right]}_{b'}$$
$$= W' x + b'$$

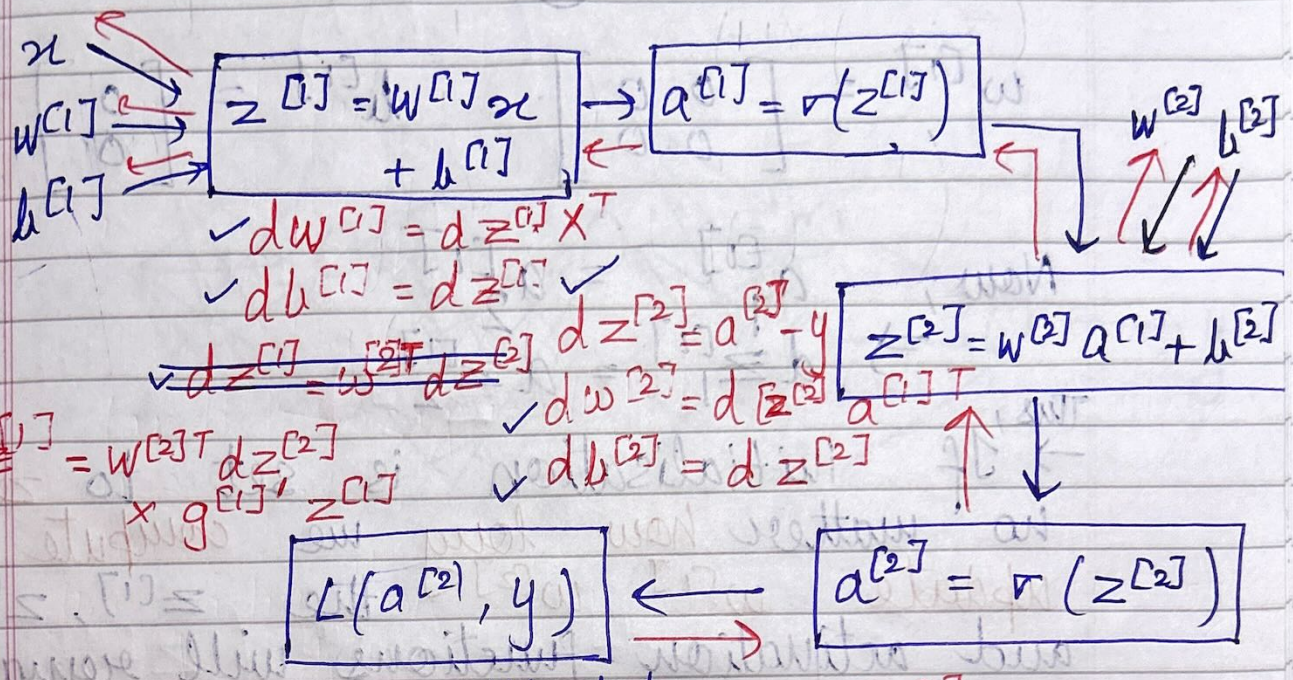
- The only time we can use a linear activation func^n is in the output layer.
Eg: in the housing price prediction eg.

$n \times 1$

$1 \times n$

- Non-Linear Activation func^n is a critical part of NN.

* BACK PROPAGATION :



n_x : input features, $n^{[0]}$, $n^{[1]}$, $n^{[2]}$ → activation func^n parameters.

$da^{[2]}$

Dimensions :

$W^{[2]} : (n^{[2]}, n^{[1]})$

$z^{[2]}, dz^{[2]} : (n^{[2]}, 1)$

$z^{[1]}, dz^{[1]} : (n^{[1]}, 1)$

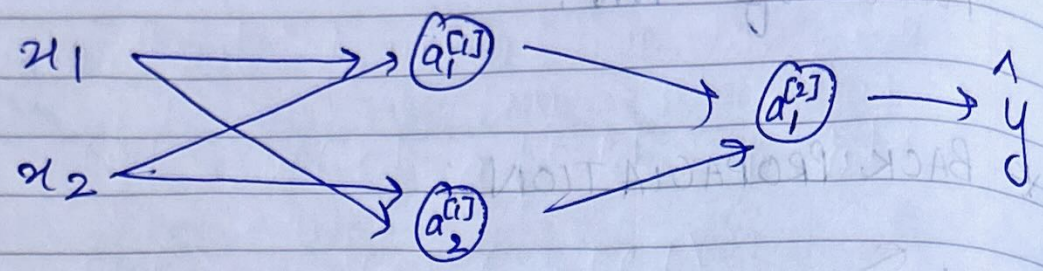
$dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]'}(z^{[1]})$

$= (n^{[1]}, n^{[2]}) \times (n^{[2]}, 1) * (n^{[1]}, 1)$

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

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

* INITIALISATION:



$$w^{[1]} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \quad b^{[1]} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Now, $a_1^{[1]} = a_2^{[1]}$

$\Rightarrow dz_1^{[1]} = dz_2^{[1]}$

Thus,

→ If initialisation is set to zero, no matter how long we compute & update $w^{[1]}, w^{[2]}$ the $z^{[1]}, z^{[2]}$ and activation functions will remain symmetric (give same output).

- As long w is not initialised to 0, we can initialise $b^{[1]}$ to 0, as it does not have symmetry issue.

- We generally randomly initialise the parameters to very small numbers

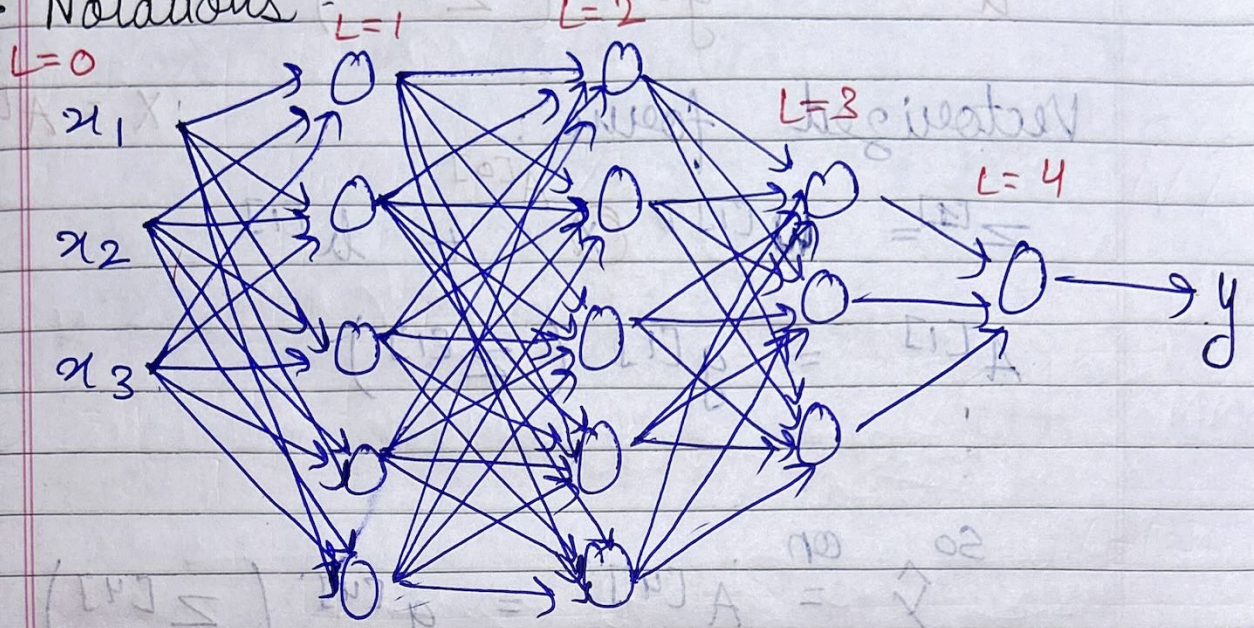
* DEEP NEURAL NETWORK.

0 hidden layer; 2 layer NN; 3 layer NN, 5 layer NN

shallow NN — deep NN

Eg: Logistic Regression.

- Notations:



L = 4 (layers)

$n^{[l]}$ = units in layer l.

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

$n^{[0]} = 3$

$= n^{[L]}$

$a^{[l]}$ = activations in layer l

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

$w^{[l]}$ = weights for $z^{[l]}$

* Forward Propagation in DNN:

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

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

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

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

and so on.

Generalised formula:

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

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

Vectorized form:

$X = A^{[0]}$

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

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

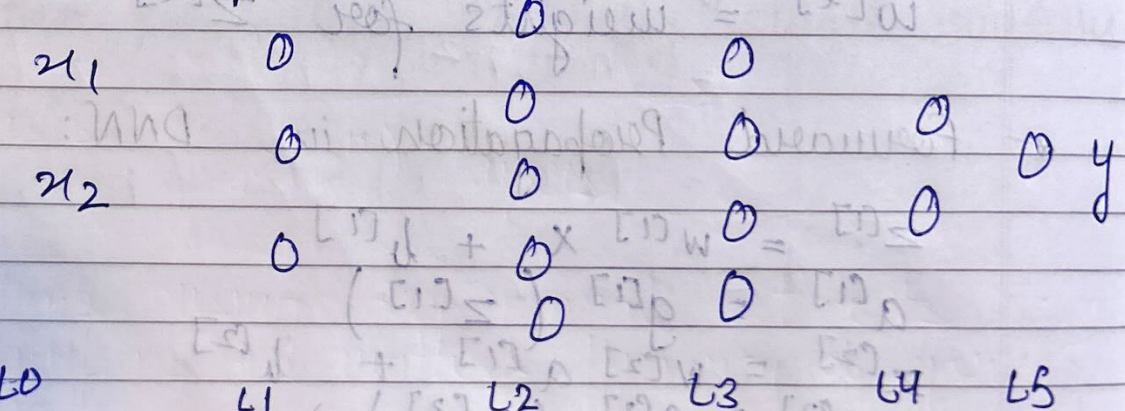
⋮

So on

$$\hat{y} = A^{[4]} = g^{[4]}(z^{[4]})$$

For computing the activation of layers from 1 to n^{th} layer we need to use an explicit for loop.

* Matrix Dimensions:



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

$(3,1) \quad (3,2) \quad (2,1)$ ignore (for now)

$(n^{[1]}, 1) \quad (n^{[2]}, n^{[0]}) \quad (n^{[0]}, 1)$

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

$$z^{[2]} \Rightarrow (n^{[2]}, 1)$$

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

$(5,1) \quad (5,3) \quad (3,1)$

$(n^{[2]}, 1) \quad (n^{[2]}, n^{[1]}) \quad (n^{[1]}, 1)$

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

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

On implementing vectorization:

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

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

* Intuition on Deep Representation:

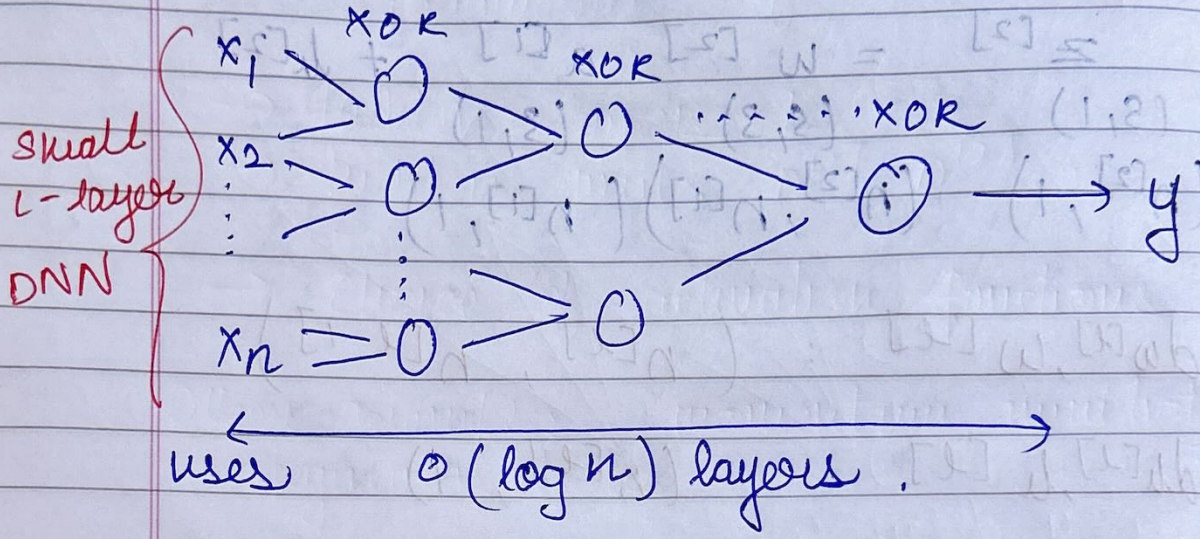
Eg: Face recognition, audio classification.

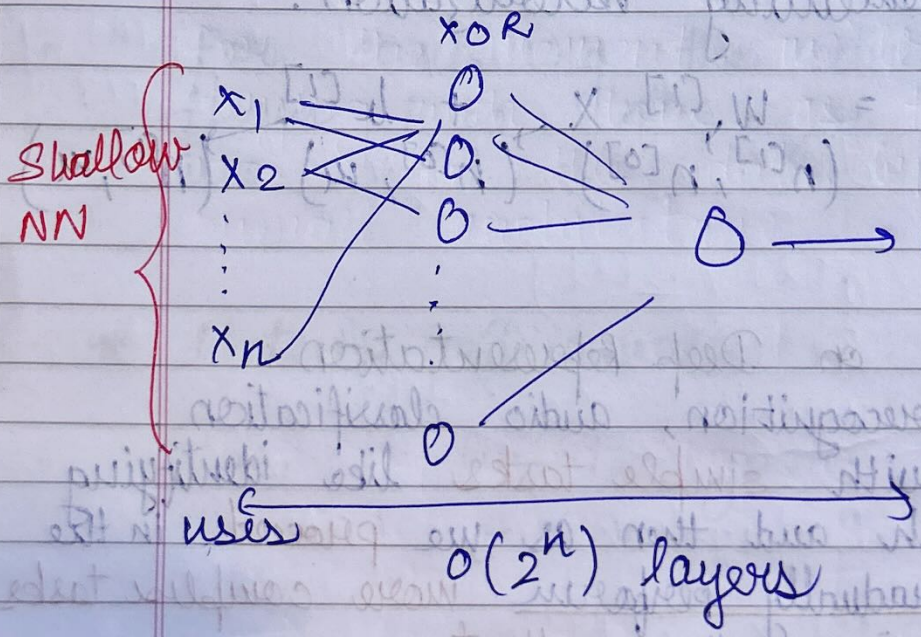
We begin with simple tasks like identifying edges, pitch and then as we proceed in the NN we gradually perform more complex tasks to determine the output

* Circuit Theory:

There are functions you can compute with 'small' L-layer DNN that shallower networks require exponentially more hidden units to compute.

Eg: $y = x_1 \text{ XOR } x_2 \text{ XOR } x_3 \text{ XOR } \ldots \text{ XOR } x_n$





Thus, using deep neural networks really simplies the math & efficiency of your model.

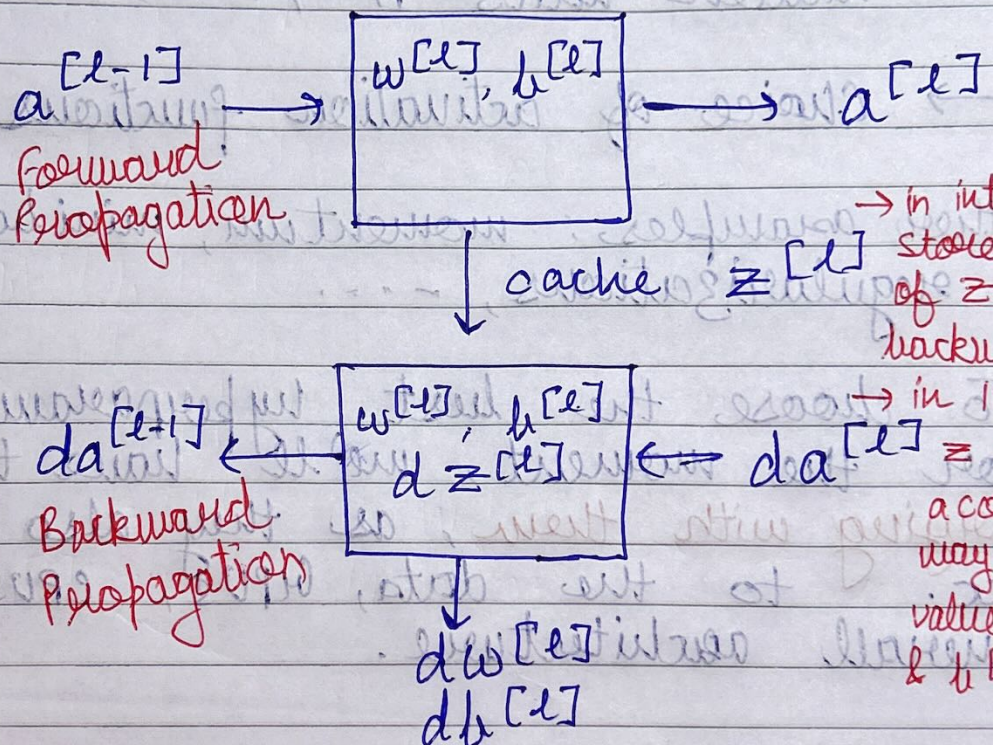
* Forward & Backward Func^n:

Layer l : $w^{[l]}$, $b^{[l]}$

Forward : Input $a^{[l-1]}$, output $a^{[l]}$
cache $z^{[l]}$

Backward : Input $da^{[l]}$, output $da^{[l-1]}$
cache ($z^{[l]}$) $dw^{[l]}$
$db^{[l]}$

LAYER L



→ in intuition to store the value of z for backward prop.
→ in IMPLEMENTATION z serves as a convinient way to copy the values of $w^{[l]}$ & $b^{[l]}$

* Backward Prop. :

Input : $da^{[l]}$

Output : $da^{[l-1]}$, $dW^{[l]}$, $db^{[l]}$

$dz^{[l]} = da^{[l]} * g^{[l]'}(z^{[l]})$ (input, from cache)

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

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

$dz^{[l]} = w^{[l+1]T} dz^{[l+1]} * g^{[l]'}(z^{[l]})$ ($da^{[l]}$, from cache)

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

What are hyperparameters ?

→ actual parameters that control the model
Parameters : $W^{[1]}$, $b^{[1]}$, $W^{[2]}$, $b^{[2]}$, ...
→ hyper parameters which are key in the model & impact the parameters
Hyperparameters :

→ learning rate α
→ # iterations
→ # parameters of $W^{[2]}$

→ hidden layers L
→ hidden units $n^{[1]}$, $n^{[2]}$, ...

→ Choice of activation functions.

Other examples : momentum, mini batch size, regularizations, ...

To choose the best hyperparameters for the moment, we'll have to keep trying with them, as they also change acc. to the data, GPUs, CPUs & overall architecture.

After course 1 :

→ implement logistic Reg
→ activation fn
→ gradient descent
→ backprop

* Implementing Logistic Regression:

Input features:

x_1, x_2, x_3, x_4.

Loan-ID, gender, married, dependents, education, self-employed, applicant-income, coapplicant Income, LoanAmount, Loan Amount term, Credit History, Property Area,

y: Loan Status.