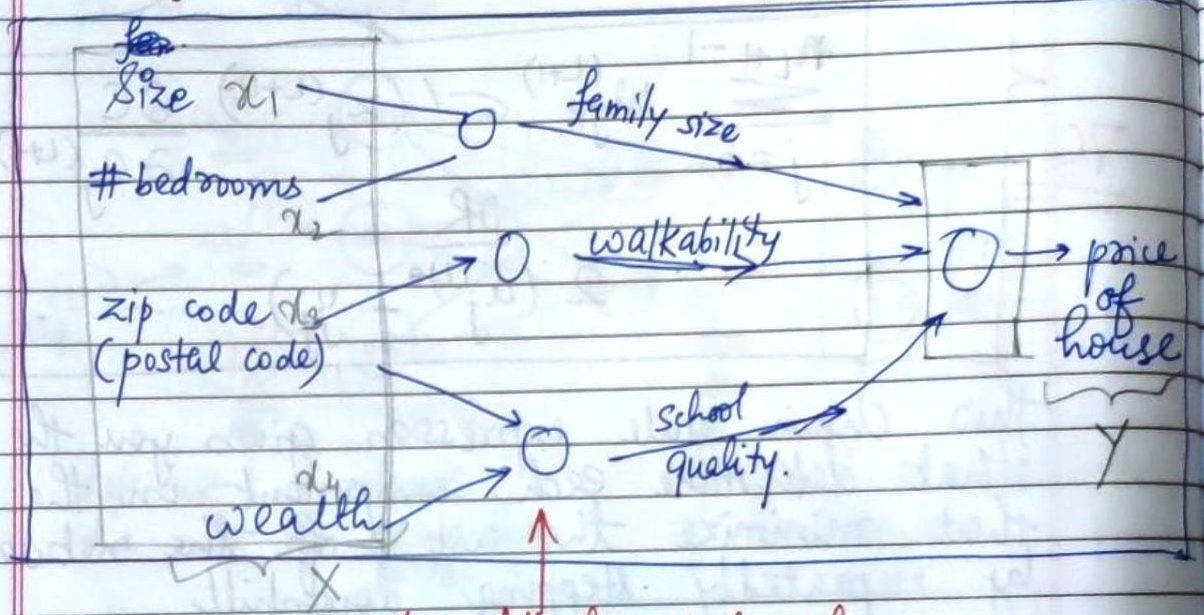
DEEP LEARNING (Coursera)

Housing Price Prediction → Simple eg of neural network

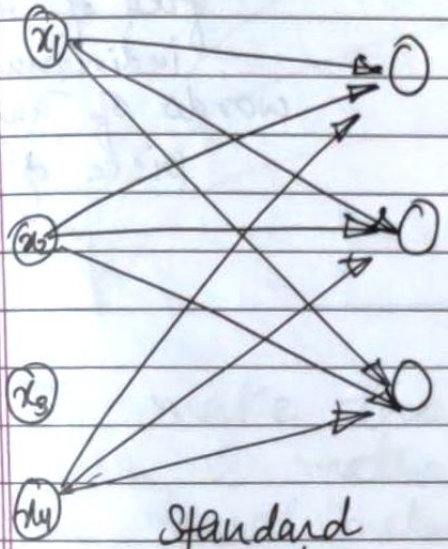


Ideally, All the circles of this middle layer are connected to every other circle of previous layer in a neural network.

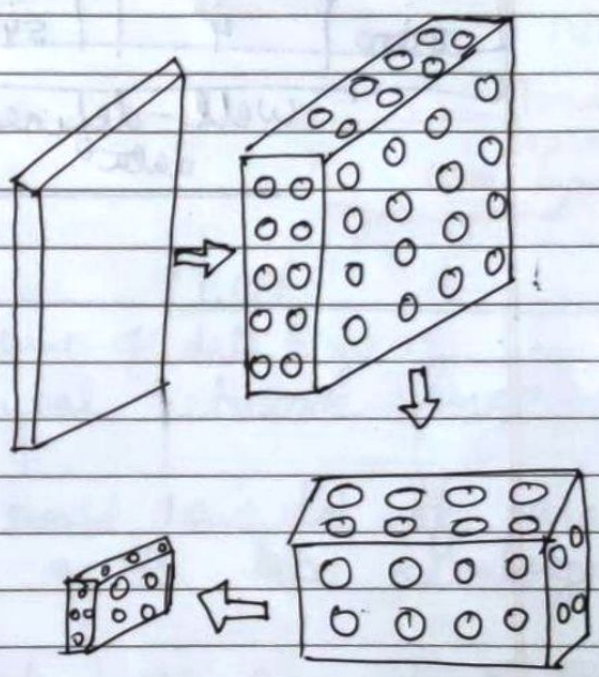
Supervised Learning

Input (x)	Output (y)	Application
(1) Home features	Price	Real Estate (Standard neural network)
(2) Ad, user info	Click on ad? (0/1)	Online Advertising
(3) Image	Object (1, ..., 1000)	Photo tagging (CNN)
(4) Audio	Text transcript	Speech recognition (RNN)
(5) English	Chinese	Machine translation
(6) Image, Radar info	Position of other cars	Autonomous Driving (custom or hybrid neural network)

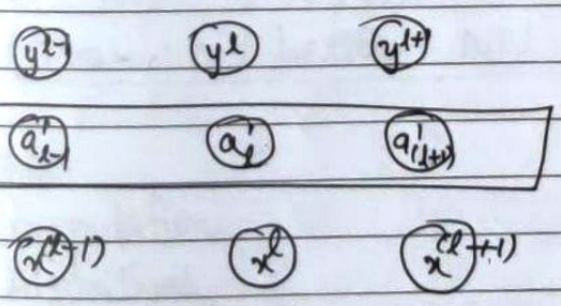
Image Processing applications often use convolutional neural network (CNN).
For Audio applications we use 2- dimensional network (time series), we use (RNN) or Reaurrent neural network. ~~language~~.



Standard NN



Convolutional NN



Recurrent NN

Supervised Learning → UNSTRUCTURED DATA

① ↓ STRUCTURED DATA

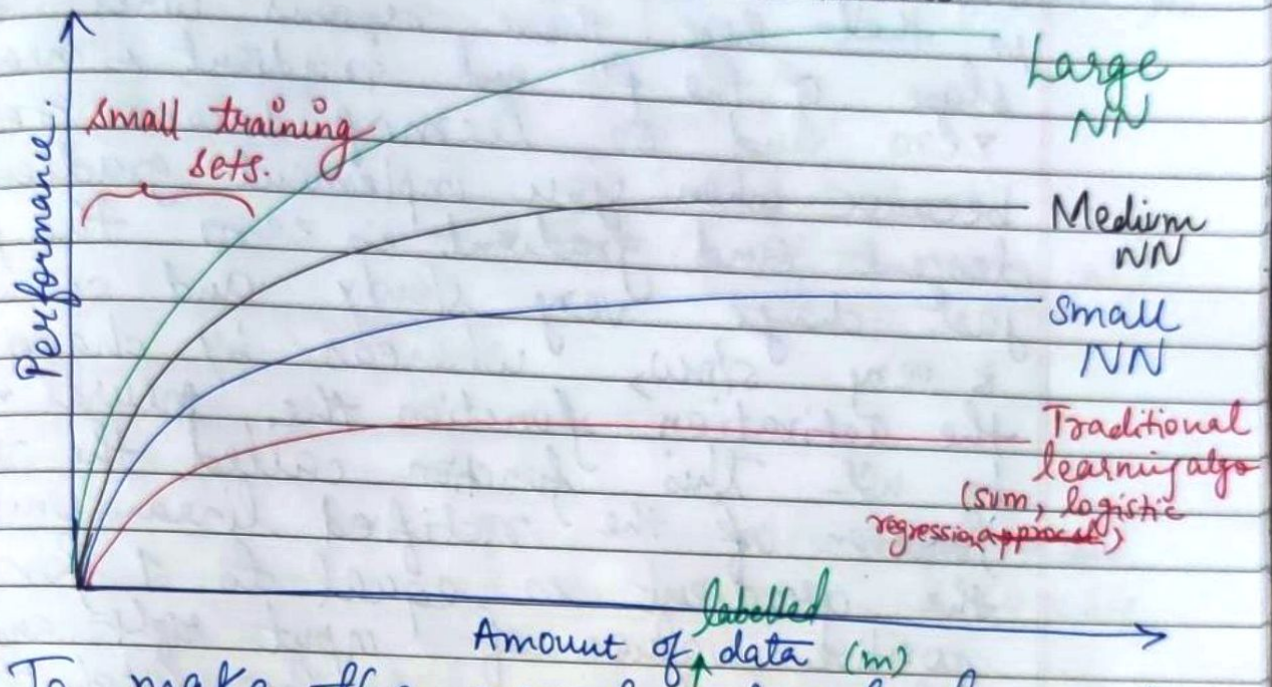
Size	Bedrooms	...	Price
2104	3		400
1600	3		330
⋮	⋮		⋮
3000	4		540

well-defined data

② ↓ AUDIO, IMAGE, TEXT

pixel values of image or freq of audio or individually words of any piece of text

SCALE DRIVES DEEP LEARNING PROCESS.



To make the neural network large there are 2 methods

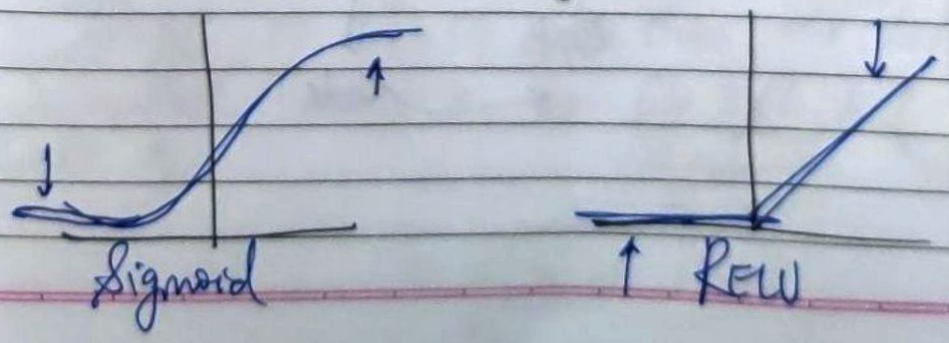
→ The neural network should have lot of parameters lot of connections so as to take advantage of large NN

→ You need to provide huge amount of data on the x-axes

→ (Train a better NN or throw more data at it)

1. Data
2. Computation
3. Algorithms

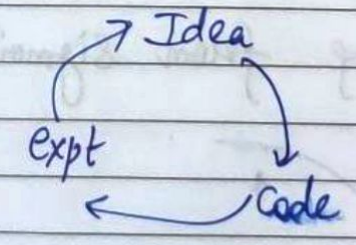
Switching from sigmoid f^n to RELU f^n



One of the problems with sigmoid function is that there these regions where the slope of the fn and gradient is nearly zero and so learning becomes really slow because when you implement gradient descent and gradient is zero the parameters just change very slowly and so learning is very slow, whereas by changing the activation function the neural network to use this function called the value function of the rectified linear unit (relu) the gradient is equal to 1 for all positive values of input right and so the gradient is much less likely to gradually shrink to zero and the gradient here the slope of this line is zero on the left but it turns out that just by switching to the sigmoid fn to the relu fn has made an algorithm called gradient descent work much faster so this is an example of maybe relatively simple algorithm

↓

This leads to fast computation.



when neural network takes long time to train, then it takes long time to go about this cycle.

BINARY CLASSIFICATION

Considering logistic regression as an algorithm for Binary classification

Image as input

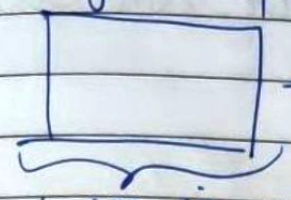


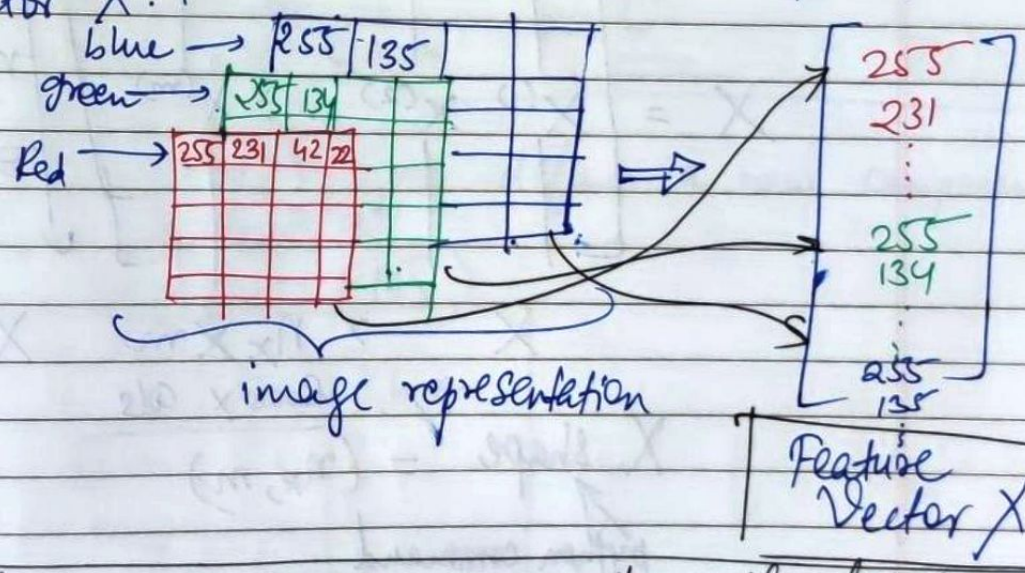
image in computer is represented by 3 seprate matrices correspondig to red green blue color channels. if ur image is 64 x 64 pixels than u will have 3 channels (RGB) with 64 x 64 pixels intensity ~~channels~~ values.

Output (Y)

① 1 (if its a cat)

② 0 (if its not a cat)

To convert all this pixel intensity values to a feature vector we are going to unroll all of these pixel values into a input feature vector X.



So if image is 64 x 64 pixels than the total values in feature Vector X is $\underbrace{64 \times 64}_{\text{dimension of image}} \times 3$ total nos. ← 3 channels

$$n = \boxed{n_x = 12288}$$

So in binary classification, our goal is to input an image represented by a feature vector x and predict the corresponding label y is 1 or 0 (cat or non-cat)

$$\boxed{x \longrightarrow y}$$

NOTATIONS

(1) Training example has dimension (x, y)

$$x \in \mathbb{R}^{n_x}, \quad y \in \{0, 1\}$$

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

$m = m_{train} \Rightarrow$ training example

$m = m_{test} \Rightarrow$ test example.

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

$\Rightarrow$ m columns, n_x rows.

$$X \longrightarrow \underset{\text{Rows} \times \text{Cols}}{n_x \times m} \qquad X \in \mathbb{R}^{n_x \times m}$$

$X.\text{shape} = (n_x, m)$

python command to find shape of matrix

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

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

$Y.\text{shape} = (1, m)$

Logistic Regression

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

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

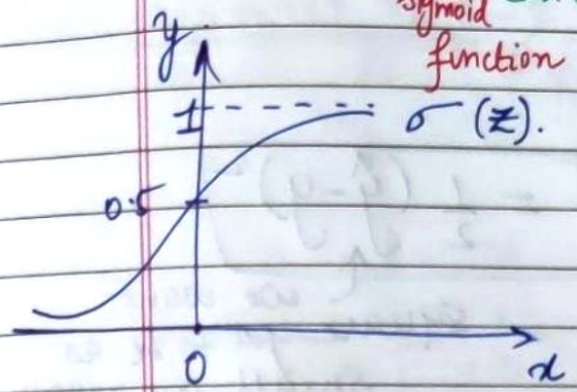
$\boxed{0 \le \hat{y} \le 1}$ ★

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

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

sigmoid function

can be much bigger than 1 or can be negative



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

If z is very large than e^{-z} is 0 so

$$\sigma(z) \approx \frac{1}{1+0} = 1$$

If z is very small or a very large negative no, then e^{-z} is $e^{\infty} \to \infty$

$$\sigma(z) \approx \frac{1}{1+\infty} \approx 0$$

In some notations, we define a new component

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

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

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_{n_x} \end{bmatrix} \begin{matrix} \} b \leftarrow \\ \\ \\ \\ \\ \end{matrix} \quad \} w \leftarrow$$

⇒ we will NOT use this notation, we want w & b as seprate

Logistic Regression Cost Function

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

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

$\hat{y}^{(i)} \approx y^{(i)}$

$i \rightarrow$ i^{th} example

Loss (error) function:

$$\boxed{\mathcal{L}(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2}$$

Difficult to optimise

we want square fn to be as small as possible

we use this loss fn.

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

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

$-\log \hat{y}$ should be as small as possible.

$\log \hat{y}$ should be as large as possible

$\hat{y}$ should be large

(2) If $y = 0$, $\mathcal{L}(\hat{y}, y) = \boxed{-\log(1-\hat{y})}$

$-\log(1-\hat{y})$ should be as small as possible

$\log(1-\hat{y})$ should be large.

$1-\hat{y}$ should be large

$\hat{y}$ should be as small as possible (0)

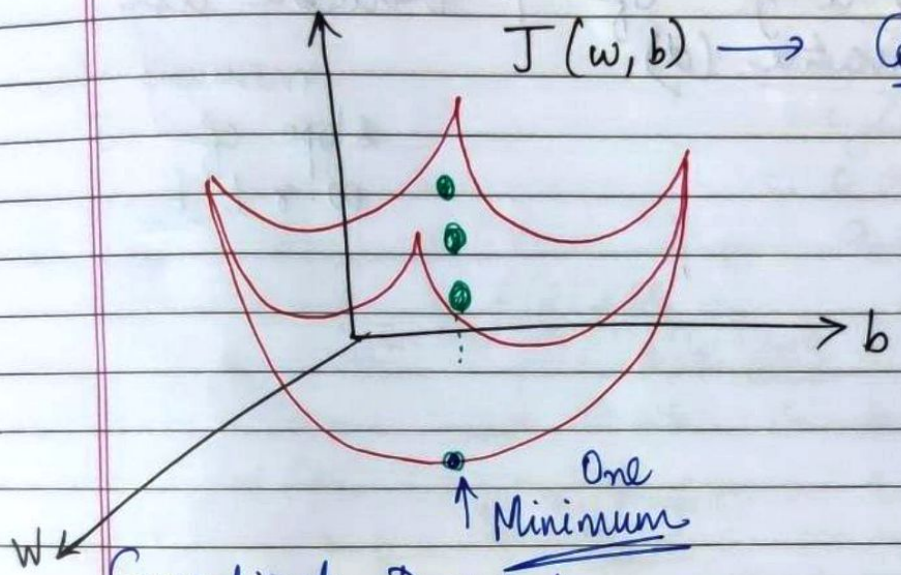
$\boxed{0 < \hat{y} < 1}$

Cost Function

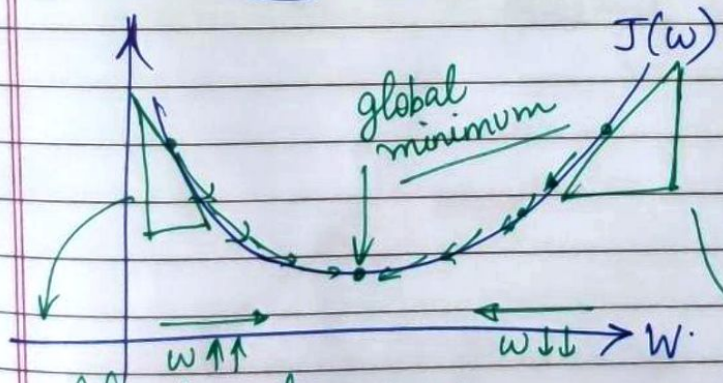
$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

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

Want to find w, b that minimise J(w,b)



Gradient Descent



learning rate

Repeat {

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

}

slope "dw"

$w := w - \alpha dw$

If the slope is -ve, we add the term αdw to the w and the value ↑↑

If slope is +ve, we subtract the term αdw from the w and the value of w ↓↓

In any case, the gradient descent will move towards the global minimum

NOTE

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

measuring the slope of J w.r.t $\underline{w}$

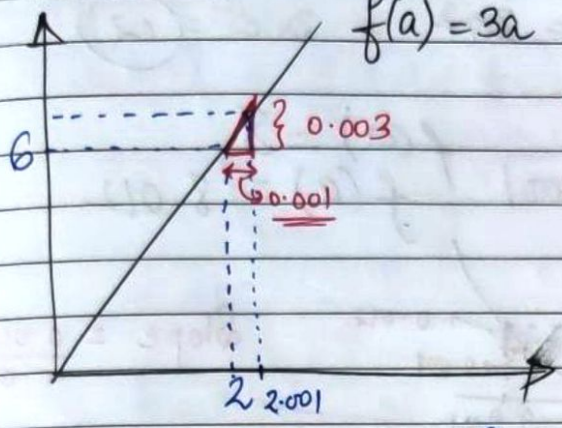
$b := b - \alpha \frac{dJ(w,b)}{dw}$ → $\frac{\partial J(w,b)}{\partial b}$ → db

measuring the slope of J w.r.t $\underline{b}$

If J is a f^n of 2 or more variables, ~~take~~ partial derivative (∂)

If J is a f^n of 1 variable use total derivative (d)

DERIVATIVES.



If u nudge x by amount of 0.001, f(a) gets nudged by 0.003 3 times as of x.

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

$$\text{Slope} = \frac{\text{height of } \Delta}{\text{weight of } \Delta} = \frac{0.003}{0.001} = 3$$

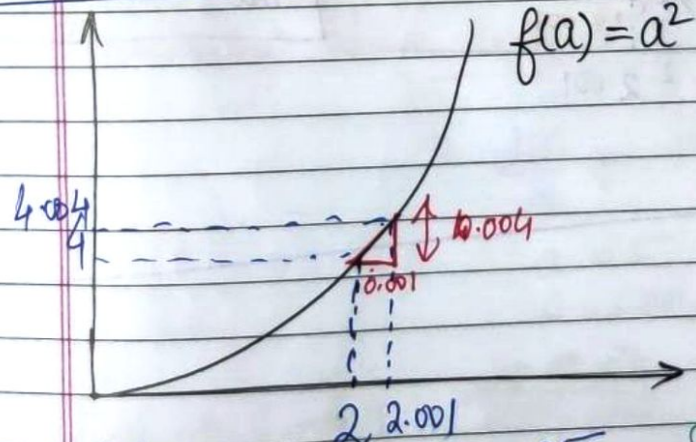
If $a = 5$ $f(a) = 15$

$a = 5.001$ $f(a) = 15.003$.

Slope at $a = 5$ is also 3.

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

Slope is same even when we change the value of a



If $a = 2$ $f(a) = 4$.

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

Slope (derivative) of f(a) at a = 2 is 4

$$\frac{d}{da} f(a) = 4 \text{ when } a = 2$$

If $a = 5$, $f(a) = 25$

If $a = 5.001$, $f(a) = 25.010$

$$\frac{d}{da} f(a) = 10 \text{ when } a = 5$$

Here, Slope is different for all different values of a

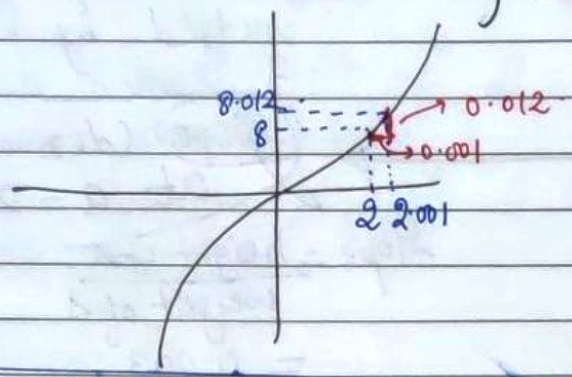
$$\frac{d}{da} f(a) = \frac{d}{da} a^2 = 2a$$

* If $f(a) = a^3$

$$\frac{d}{da} f(a) = 3a^2 = 3 \times 2^2 = 12$$

when $a = 2$, $f(a) = 8$

When $a = 2.001$, $f(a) = 8.012$



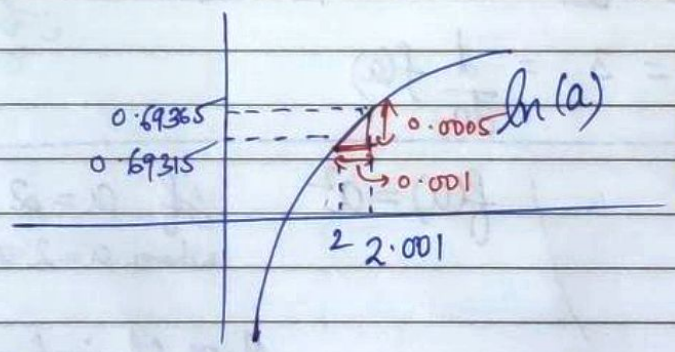
Slope $= \frac{0.012}{0.001} = 12$

* If $f(a) = \log_e(a) = \ln(a)$

$$\frac{d}{da} f(a) = \frac{1}{a} = \frac{1}{2}$$

$\Rightarrow$ when $a = 2$, $f(a) = 0.69315$

$a = 2.001$, $f(a) \approx 0.69365$

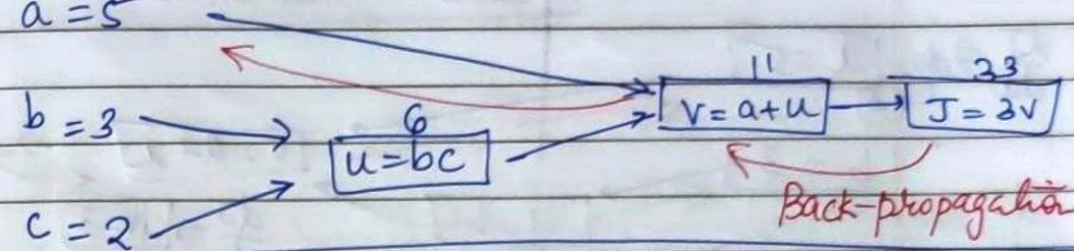


$$\boxed{\frac{d}{da} f(a) = \frac{1}{2}}$$

COMPUTATION GRAPH

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

$\Rightarrow$ ① $u = bc$
$\Rightarrow$ ② $v = a+u$
$\Rightarrow$ ③ $J = 3v$



Here J in this case u need to optimise and in case of logistic regression J is the cost f^n, that we try to minimise

DERIVATES WITH COMPUTATION GRAPH

$\frac{dJ}{dv} = ?$

$J = 3v$

Considering above example. If $v = 11$, $J = 33$
$v = 11.001$, $J = 33.003$.

$$\frac{dJ}{dv} = \text{Slope} = \frac{0.003}{0.001} = 3$$

$\frac{dJ}{da} = ?$

$a = 5$, $v = 11$, $J = 33$
$a = 5.001$, $v = 11.001$, $J = 33.003$

$$\frac{dJ}{da} = \text{Slope} = \frac{0.003}{0.001} = 3$$

$a \rightarrow v \rightarrow J$ OR

$$\boxed{\frac{dJ}{da} = \frac{dJ}{dv} \cdot \frac{dv}{da}} \Rightarrow \text{chain Rule}$$

$$\frac{dv}{da} = 1, \quad \frac{dJ}{dv} = 3 \qquad \therefore \frac{dJ}{da} = 3 \cdot 1 = 3$$

The derivates of final output variable with respect to various intermediate variables it is known as "dvar"

$$\hookrightarrow \boxed{\frac{d\,(\text{final Output Var})}{d\,(\text{var})}} \Rightarrow \underline{\text{dvar}}$$

BACK – PROPAGATION

$\frac{dJ}{da} \Rightarrow da \rightarrow 3$ $\boxed{a = 5}$

$\boxed{b = 3}$

$\frac{dJ}{db} \Rightarrow db \rightarrow 6$

$\boxed{c = 2}$

$\frac{dJ}{dc} \Rightarrow \boxed{dc = 9}$

$\boxed{u = bc}$

$du = 3$

$\boxed{v = a + u}$ $\rightarrow$ $\boxed{J = 3v}$

$\frac{dJ}{dv} \Rightarrow \boxed{dv = 3}$

$$\frac{dJ}{du} = 3 = \boxed{\underbrace{\frac{dJ}{dv}}_{3} \cdot \underbrace{\frac{dv}{du}}_{1}}$$

$u = 6$ $\quad u \Rightarrow 6.001$

$v = 11$ $\quad v \Rightarrow 11.001$

$J = 33$ $\quad J \Rightarrow \underline{33.003}$

$\star$ $\frac{dJ}{db} = ?$

$$\frac{dJ}{db} = \underbrace{\frac{dJ}{du}}_{3} \cdot \underbrace{\frac{du}{db}}_{2}$$

$b = 3 \rightarrow 3.001$

$u = b.c = 6 \rightarrow 6.002$

$\frac{du}{db} = 2$

$J = 33.006$

$\frac{dJ}{db} = 6$

$\boxed{\frac{dJ}{db} = 6}$

$\frac{dJ}{dc} = ?$

$$\frac{dJ}{dc} = \underbrace{\frac{dJ}{du}}_{3} \cdot \underbrace{\frac{du}{dc}}_{3}$$

$c = 2 \rightarrow 2.001$

$u = b.c = 6 \rightarrow 6.003$

$\frac{du}{dc} = 3$

$J = 33.009$

$\frac{dJ}{dc} = 9$

$\boxed{\frac{dJ}{dc} = 9}$

Logistic Regression Recap.

$z = w^T x + b.$

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

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

$a \rightarrow$ output of Logistic regression

$y \rightarrow$ ground truth label.

x_1, w_1, x_2, w_2, b $\rightarrow$ $\boxed{z = w_1x_1 + w_2x_2 + b} \Rightarrow \boxed{\hat{y} = a = \sigma(z)} \rightarrow \boxed{L(a,y)}$

$$\text{"}dz\text{"} = \frac{dL(a,y)}{dz} = \frac{a-y}{\underbrace{dL(a,y)}_{da}} \cdot \frac{da}{dz} \rightarrow a(1-a)$$

da, $\frac{d}{dz} g(z)$

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

$\rightarrow da \cdot g'(z)$

$dz(x_1)$

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial w_1}$$

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial w_2} \Rightarrow dz\, x_2$$

$$\frac{\partial L}{\partial w_1} = \text{"}dw_1\text{"} = x_1\, dz \qquad dw_2 = x_2\, dz \qquad db = dz$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial b}$$

$dz(1)$

$$w_1 := w_1 - \alpha\, dw_1$$
$$w_2 := w_2 - \alpha\, dw_2$$
$$b := b - \alpha\, db$$

We have seen this to apply for 1 training example, we have to apply gradient descent of "m" training examples

PROOF :- $a = \sigma(z)$ $\qquad \frac{da}{dz} = \sigma'(z) = \frac{a^2(1-\sigma(z))}{\sigma(z)} = a(1-a)$

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

$$\sigma'(z) = \frac{e^{-z}}{(1+e^{-z})^2} = \sigma^2(z) \cdot e^{-z}$$

$$a^2 e^{-z} \rightarrow \frac{1}{\sigma(z)} - 1$$

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

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

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

Input data $(x^{(i)}, y^{(i)})$

$J = 0;\ dw_1 = 0,\ dw_2 = 0;\ db = 0$

for $i = 1$ to m

$$\begin{cases} z^{(i)} = w^T x^{(i)} + b \\ a^{(i)} = \sigma(z^{(i)}) \end{cases}$$

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

If there are n features u have to write $dw_1, dw_2, \ldots, dw_n$

$$\begin{cases} dz^{(i)} = a^{(i)} - y^{(i)} \\ dw_1 += x_1^{(i)} dz^{(i)} \\ dw_2 += x_2^{(i)} dz^{(i)} \\ db += dz^{(i)} \end{cases}$$

$n = 2 \Rightarrow$ two features

for loop ended

indentation

$J /= m;$

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

for $j = 1$ to n

$dw_1 = \frac{\partial J}{\partial w_1}$ $\qquad dw_2 = \frac{\partial J}{\partial w_2}$

$dw_j += x_j^{(i)} dz^{(i)}$

$w_1 := w_1 - \alpha\, dw_1$

$w_2 := w_2 - \alpha\, dw_2$

$b := b - \alpha\, db$

There are 2 weaknesses with calculation

1. We have used 2 for loops.
   i) first for loop for m training examples calculation of cost fn
   ii) Second for loop for all the features

NOTE: When u ~~are~~ are implementing deep learning algorithms you find that having explicit for loop in your code makes your algorithm run less efficiency and so in the deep learning error would move to bigger and bigger data sets. and so being able to implement your algorithms without using explicit for loops is really important and will help you to scale to much bigger data sets so it turns out that there are set of techniques called vectorisation techniques that allow you to get rid of this explicit for loops in code Vectorisation makes the code very efficient, without using even a single for loop.

VECTORISATION

What is Vectorisation?

In Logistic Regression fn

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

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

Non Vectorised implementation

```
z=0
for i in range (nx):
    z += w[i]*x[i]
z += b
```

Vectorised implementation

$$z = \underbrace{np.dot(w.T, x)}_{w^T x} + b$$

This find $w^T x$ directly.

Practical representation in Jupyter notebook.

```
a = np.random.rand(1000000)
b = np.random.rand(1000000)

tic = time.time()          ← time at start of taking dot prod
c = np.dot(a,b)            ← Dot prod of 2 vectors a,b
toc = time.time()          ← time at end of taking dot prod
print(c)                   ← output of dot prod of a,b
print("vectorised version:" + str(1000*(toc-tic)) + "ms")
                           ↑ time reqd for taking dot prod in millisec
c=0
tic = time.time()          ← time before for loop execution
for i in range(1000000)    ← For loop
    c += a[i]*b[i]         ← multiplicatn of 2 matrices
toc = time.time()          ← time after execution of for loop
print(c)                   ← output of for loop
print("For Loop:" + str(1000*(toc-tic)) + "ms")
                           ↑ time reqd for execution of for loop in millisec
```

Output

250286.989866 — same output

Vectorised version: 1.50275... ms

— " —

For Loop: 474.2957... ms — Huge Difference

So, Vectorisation is much faster! So, we use it for deep learning algorithm

GPU → graphics processor unit.

Computations of GPU's are remarkably good at these ~~sim~~ calculations and are better than CPU's

GPU & CPU → both have vectorisatn instruction

→ SIMD → single instruction multiple data

If you use simD instructions in functions like np.dot-fn or other fn that don't require explicitly declaring a for loop it enables python numpy to take much better advantage of parallelism to do computations much faster for both CPU & GPU

THUMB RULE → So, avoid using explicit declaration of for loop.

Non-Vectorised method.

$u = Av$

$u_i = \sum_j A_{ij} V_j$

$u_{i\times 1} = A_{i\times j} \cdot V_{j\times 1}$

```
u = np.zeros((n,1))
for i ....
    for j ....
        u[i] += A[i][j] * v[j]
```

Vectorised method ⟶ u = np.dot(A,v)

VECTORS AND MATRIX VALUED FUNCTIONS.

You need to apply the exponential operation on every element of a matrix/vector.

$$v = \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} \longrightarrow u = \begin{bmatrix} e^{v_1} \\ e^{v_2} \\ \vdots \\ e^{v_n} \end{bmatrix}$$

Non-Vectorised method

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

Vectorised method

```
import numpy as np.
u = np.exp(v)
```

-5	-4	-3	-2	-1

↓ np.abs()

5	4	3	2	1

np. log(v) → To calculate log of every element of matrix

np. abs(v) → To calculate absolute value of every element of matrix.

np. maximum(v, 0) → Compares the corresponding elements of two arrays and selects the maximum of those 2 elements and gives output as an array of maximum elements.

eg.
Input array1: [2, 8, 125]
Input array2: [3, 3, 15]

Output array after selecting maximum: [3, 8, 125]

VECTORISING LOGISTIC REGRESSION

Step 1:- Reducing one for loop → make dw a vector

$J=0,\ dw_1=0,\ dw_2=0,\ db=0$ $\quad dw = np.zeros[(n_x,1)]$

for $i=1$ to m:

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

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

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

$dz^{(i)} = a^{(i)}[1-a^{(i)}]$

~~for $j=1$ to n_x:~~

~~$dw_j += x_j^{(i)} dz^{(i)}$~~ → $dw += x^{(i)} dz^{(i)}$

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

$J = J/m$;

~~$dw_1 = dw_1/m,\ dw_2 = dw_2/m,$~~ ~~$db$~~ $db = db/m$

$dw /= m$

for m training examples you need to perform this operations 'm' times

$z^{(1)} = w^T x^{(1)} + b \qquad z^{(2)} = w^T x^{(2)} + b \qquad z^{(3)} = w^T x^{(3)} + b$

$a^{(1)} = \sigma(z^{(1)}) \qquad a^{(2)} = \sigma(z^{(2)}) \qquad a^{(3)} = \sigma(z^{(3)})$

Goal → To perform this operation w/o for loop ..... m times

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

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

$$\underbrace{[\quad]}_{w^T} \begin{bmatrix} | & | & | & & | \\ x^{(1)} & x^{(2)} & x^{(3)} & \cdots & x^{(m)} \\ | & | & | & & | \end{bmatrix}$$

$$[\underbrace{w^T x^{(1)} + b}_{z^{(1)}} \quad \underbrace{w^T x^{(2)} + b}_{z^{(2)}} \quad + w^T x^{(3)} + b \cdots \quad \underbrace{w^T x^{m} + b}_{z^{(m)}}]_{1\times m}$$

BROADCASTING IN PYTHON ⟹ $\boxed{Z = np.dot(W.T, X) + b}$

python automatically takes 1xm matrix

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

Vectorizing Logistic Regression Gradient Computation

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

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

$\underline{\underline{1 \times m}}$

$A = [a^{(1)} \; \ldots \; a^{(m)}]$ $\quad$ $Y = [y^{(1)} \; \ldots \; y^{(m)}]$

$$\boxed{dZ = A - Y = [a^{(1)} - y^{(1)} \quad a^{(2)} - y^{(2)} \; \ldots]}$$

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

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

$$\boxed{db = \frac{1}{m} \, np.sum(\underline{\underline{dZ}})}$$

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

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

$$= \frac{1}{m} \left[x^{(1)} dz^{(1)} + \ldots + x^{(m)} dz^{(m)} \right]$$

for iter in range( ) : no of iteration of gradient descent

$Z = W^T X + b$

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

$A = \sigma(Z)$

$dZ = A - Y$

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

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

$w = w - \alpha dw$ → learning rate

$b = b - \alpha db$

this will work without for loop if only calculate 1 iteration of gradient descent

BROADCASTING IN PYTHON

Example

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

$= A_{(3,4)}$

Calculate % of Carb, Proteins, Fat. Can you do that without use of explicit for loop?

In: cal = A.sum(axis = 0)
print(cal)

← sum of all column elements

Note → if u want sum of all row elements then use axis = 1

Out: [59 239 155.4 76.9]

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

3x4 matrix

0 ↓ 1 →

rebundant, but safe to use it. the 1x4 matrix gets converted to 3x4 by copying the values of it below

```
[[94.91525424   0.          2.83140283   88.42652...]
 [2.03389...    43.514644...  33.4620...   10.403120...]
 [3.0508476...  56.4853...    63.706...    1.17035...]]
```

Percentages are printed

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

python will auto expand it to 4x1 vector.

2] $\underset{(m,n)}{\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix}} * \begin{bmatrix}100 & 200 & 300\end{bmatrix}$

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

$\xrightarrow{\text{Converted}} \begin{bmatrix}100 & 200 & 300\\100 & 200 & 300\end{bmatrix} = \begin{bmatrix}101 & 202 & 303\\104 & 205 & 306\end{bmatrix}$

3] $\underset{(m,n)}{\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix}} * \begin{bmatrix}100\\200\end{bmatrix} = \begin{bmatrix}101 & 102 & 103\\204 & 205 & 206\end{bmatrix}$

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

$\xrightarrow{\text{Converted}} \begin{bmatrix}100 & 100 & 100\\200 & 200 & 200\end{bmatrix}$

(m,n) matrix

$+$

$-$

$*$ $(1,n) \rightsquigarrow (m,n)$ → copy m times

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

→ A note on python / numpy vectors

→ a = np.random.randn(5)
a.shape = (5,) ← neither a row vector nor a column vector
"Rank 1 array"

→ When u remove transpose of matrix a print (a.T) it prints the same matrix as a.

→ When u multiply a, & a.T then, u get a no, not a matrix.

↳ Don't use this data structures with rank 1 array or shape is (n,) while coding in neural Network!

a=np.random.randn(5, 1) → a.shape = (5,1)
↳ column vector.

a = np.random.randn(1,5) → a.shape = (1,5)
↳ row vector

use this

If u are not aware of dimension of your vector use assertion statement

assert statement checks whether the condition is true, if its true prog continues to execute if its false prog stops executing with an assertion error (plus) an optional message.

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

to reshape an array

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

Use row & column vectors instead of rank one arrays. Also use assertion & reshape to adjust the dimensions of array

WORLD STAR™
Date :
Page :

OPTIONAL

Explaination of Logistic regression cost function

Why do we use that cost function for logistic regression?

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

Interpret- $\hat{y} = P(y=1/x)$

$$\begin{cases} \text{If } y=1: & P(y|x) = \hat{y} \\ \text{If } y=0: & P(y|x) = 1-\hat{y} \end{cases}$$

This both results interprets by expression

$$\boxed{P(y|x) = \hat{y}^{y} (1-\hat{y})^{(1-y)}}$$

① If $y=1$: $P(y|x) = \hat{y}^{1}(1-\hat{y})^{0}$
$= \hat{y}(1)$

② If $y=0$: $P(y|x) = \hat{y}^{(0)}(1-\hat{y})^{(1-0)}$
$= (1-\hat{y})$

$$\log P(y|x) = \log \hat{y}^{y}(1-\hat{y})^{(1-y)} = y \log \hat{y} + (1-y)\log(1-\hat{y})$$

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

We want to minimise the loss fn

When u are training learning algorithm you want to make probabilities large whereas here we went to minimise the loss fn so minimising the loss corresponds to maximise the mark of probability

$$\log P(\text{labels in training set}) = \log \prod_{i=1}^{m} P(y^{(i)}/x^{(i)})$$

$$\log P(\text{—''—}) = \sum_{i=1}^{m} \underbrace{\log(P(y^{(i)}|x^{(i)}))}_{-L(\hat{y}^{(i)}, y^{(i)})}$$

$\log P(-\,\text{''}\,-)$

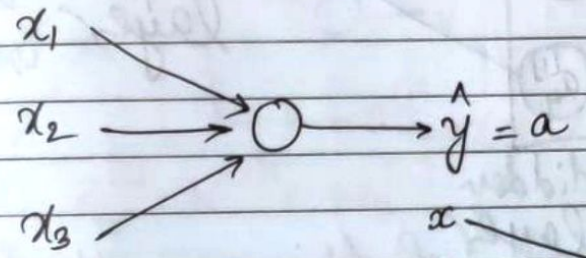
~~Cost function~~ $= -\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})$

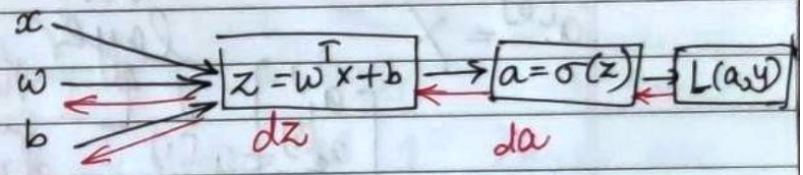
For cost fn, we remove the -ve sign to minimise it, and divide by a scaling factor m

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

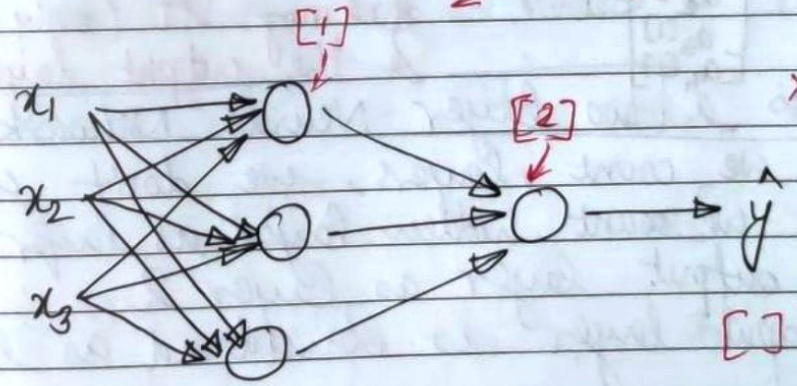
extra scaling factor for convience to get values scaled

→ What is a Neural Network?





$(1-\hat{y})$

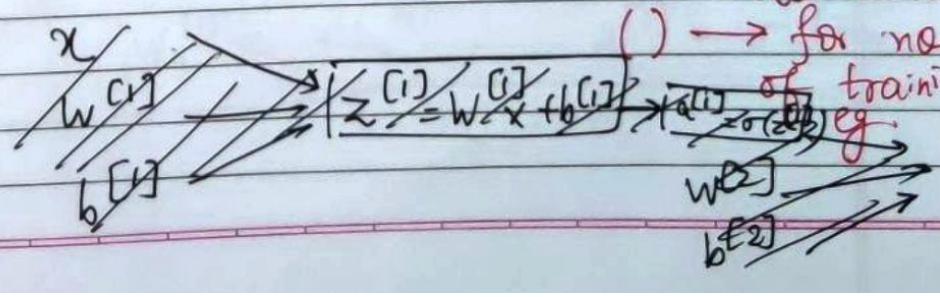


$x^{(i)} \rightarrow$ for training example

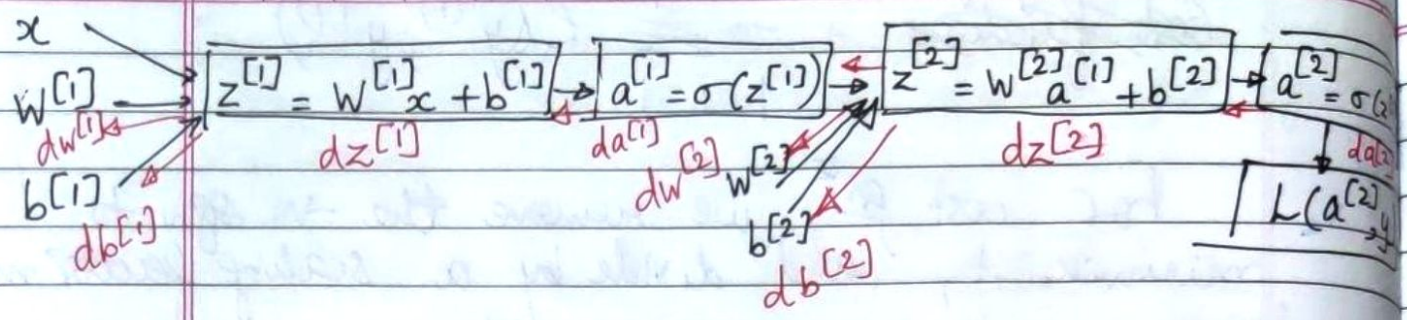
NOTE:

[ ] → represents different layers

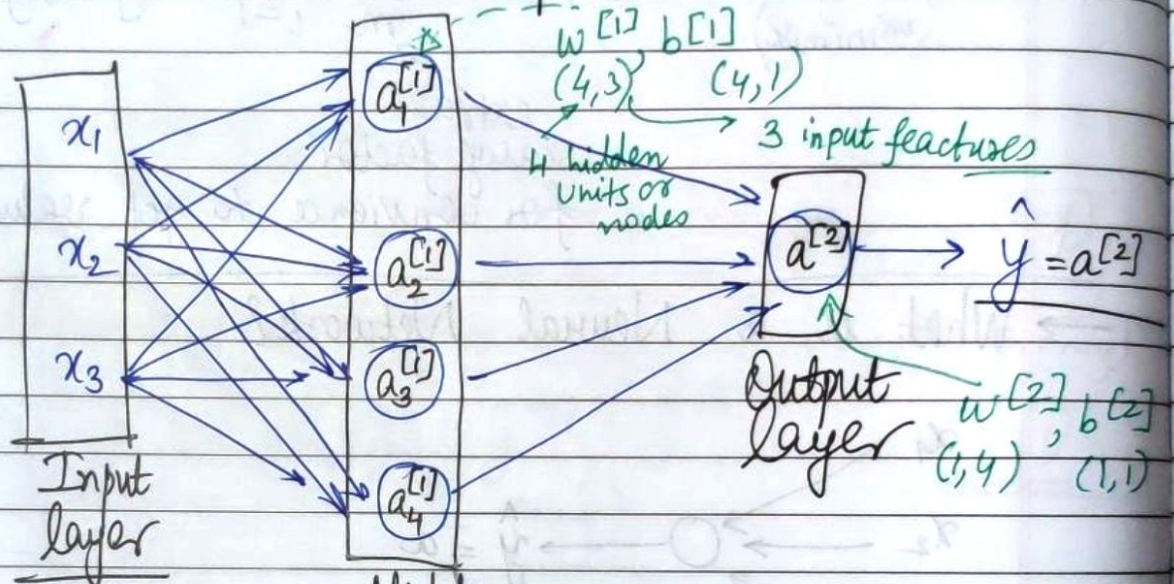
( ) → for no of training eg.



FLOWCHART.



(I) Neural Network Representation



$a^{[0]} = X$

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

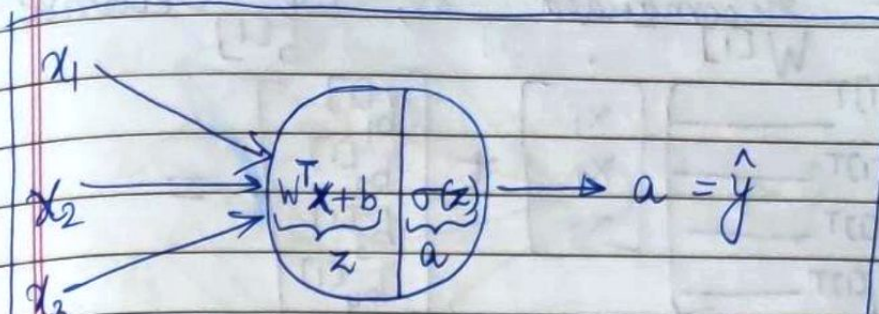
The things in the hidden layer cannot be seen in the training set (only the input & the output layers is visible)

This is a $\boxed{\text{two layer Neural Network}}$

When we count layers, we don't count input layer, we count hidden layer as layer 1 and your output layer as layer 2

(Input layer is considered as layer 0)

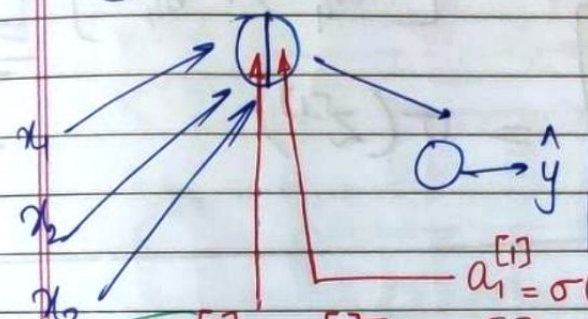
⇒ COMPUTING A NEURAL NETWORK'S OUTPUT



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

$a_i^{[l]}$ — $[l]$ ← layer, i ← node in layer.

Considering 1st node of 1st layer in (I)

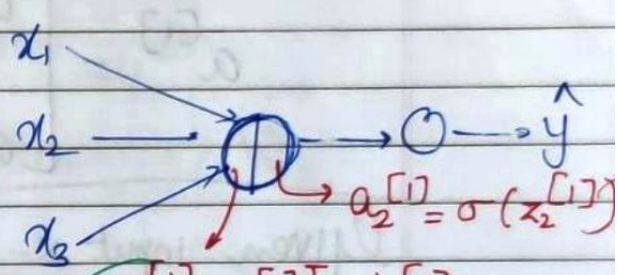


$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]}$

1st layer

1st node in 1st layer

Considering 2nd node of 1st layer in (I)



$z_2^{[1]} = w_2^{[1]T} + b_2^{[1]}$

1st layer

2nd node of 1st layer.

So eq^n's for all 4 nodes of 1st layer are

$$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]}, \quad a_1^{[1]} = \sigma(z_1^{[1]})$$

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

$$z_3^{[1]} = w_3^{[1]T} x + b_3^{[1]}, \quad a_3^{[1]} = \sigma(z_3^{[1]})$$

$$z_4^{[1]} = w_4^{[1]T} x + b_4^{[1]}, \quad a_4^{[1]} = \sigma(z_4^{[1]})$$

Using for loops to do those transformations is not recommended so, we vectorise

$$\underbrace{\begin{bmatrix} \text{---} & w_1^{[1]T} & \text{---} \\ \text{---} & w_2^{[1]T} & \text{---} \\ \text{---} & w_3^{[1]T} & \text{---} \\ \text{---} & w_4^{[1]T} & \text{---} \end{bmatrix}}_{\underline{\underline{4\times 3}}} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \underbrace{\begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix}}_{} =$$

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

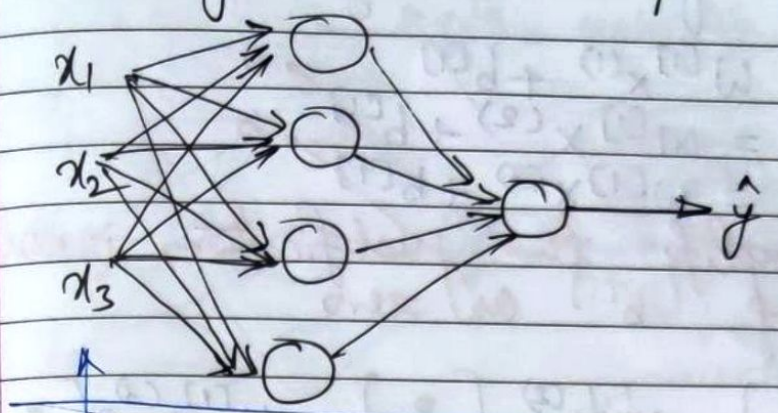
$$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ \vdots \\ a_4^{[1]} \end{bmatrix} = \sigma(z^{[1]})$$

Given input x:

for neural network (I)

$$\begin{cases} \underset{(4,1)}{z^{[1]}} = \underset{(4,3)}{W^{[1]}} \underset{(3,1)}{a^{[0]}} + \underset{(4,1)}{b^{[1]}} \\ \underset{(4,1)}{a^{[1]}} = \sigma(\underset{(4,1)}{z^{[1]}}) \\ \underset{(1,1)}{z^{[2]}} = \underset{(1,4)}{W^{[2]}} \underset{(4,1)}{a^{[1]}} + \underset{(1,1)}{b^{[2]}} \\ \underset{(1,1)}{a^{[2]}} = \sigma(\underset{(1,1)}{z^{[2]}}) \end{cases}$$

Vectorising accross multiple examples.



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

$$x^{(1)} \longrightarrow a^{[2](1)} = \hat{y}^{(1)}$$
$$x^{(2)} \longrightarrow a^{[2](2)} = \hat{y}^{(2)}$$
$$x^{(3)} \longrightarrow a^{[2](3)} = \hat{y}^{(3)}$$
$$\vdots \qquad\qquad \vdots$$
$$x^{(m)} \longrightarrow a^{[2](m)} = \hat{y}^{(m)}$$

NOTE $a^{[l](i)}$ ← example i, ← layer l.

Non Vectorised

for $i = 1$ to m

$$z^{[1](i)} = W^{[1]} x^{(i)} + b^{[1]} \Rightarrow W^{[1]} A^{[0](i)} + b^{[1]}$$
$$a^{[1](i)} = \sigma(z^{[1](i)})$$
$$z^{[2](i)} = W^{[2]} a^{[1](i)} + b^{[2]}$$
$$a^{[2](i)} = \sigma(z^{[2](i)})$$

$X \Rightarrow$ input vector $= A^{[0]}$

$$X = \begin{bmatrix} | & | & | & & | \\ x^{(1)} & x^{(2)} & x^{(3)} & \cdots & x^{(m)} \\ | & | & | & & | \end{bmatrix}_{(n_x, m)}$$

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

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

activat^n of first hidden unit of first training example

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

activat^n of first hidden unit on m training eg.

hidden units

activat^n of 2nd hidden unit of first training eg.

training example

Explaination for Vectorised Implementation.

$$Z^{[1](1)} = W^{[1]}x^{(1)} + \cancel{b^{[1]}} \to 0$$
$$Z^{[1](2)} = W^{[1]}x^{(2)} + \cancel{b^{[1]}} \to 0$$
$$Z^{[1](3)} = W^{[1]}x^{(3)} + \cancel{b^{[1]}} \to 0$$

To simplify this justification, ignore b assuming $b^{[1]}$ as zero

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

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

$$W^{[1]}X^{(i)} = Z^{[1](i)}$$

FINAL EXPRESSION after broadcasting

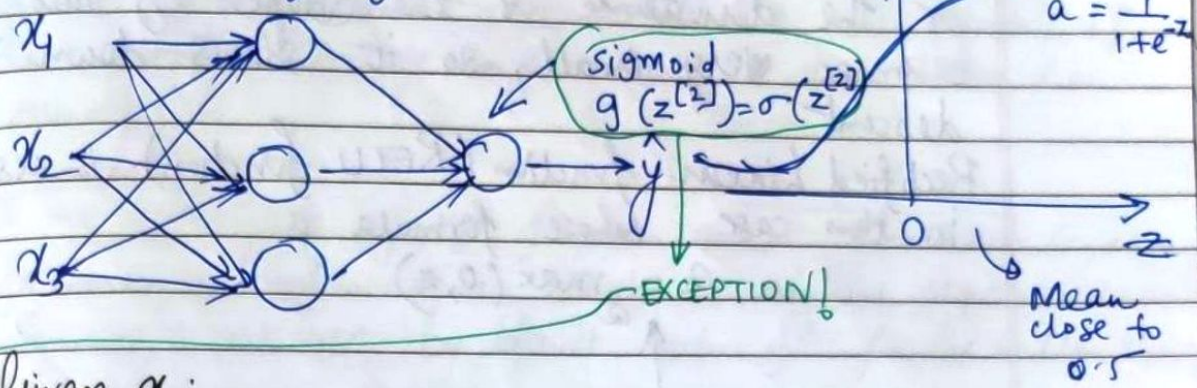
$$Z^{[1](i)} = W^{[1]}X^{(i)} + b^{[1]}$$

with python broadcasting we add $b^{[1]}$ individually to each of the column of the matrix

If u replace sigmoid f^n to tanh function in the hidden layer it almost always works better bcos tanh f^n range is b/t -1 & 1 the mean of activation is almost zero.

ACTIVATION FUNCTIONS

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

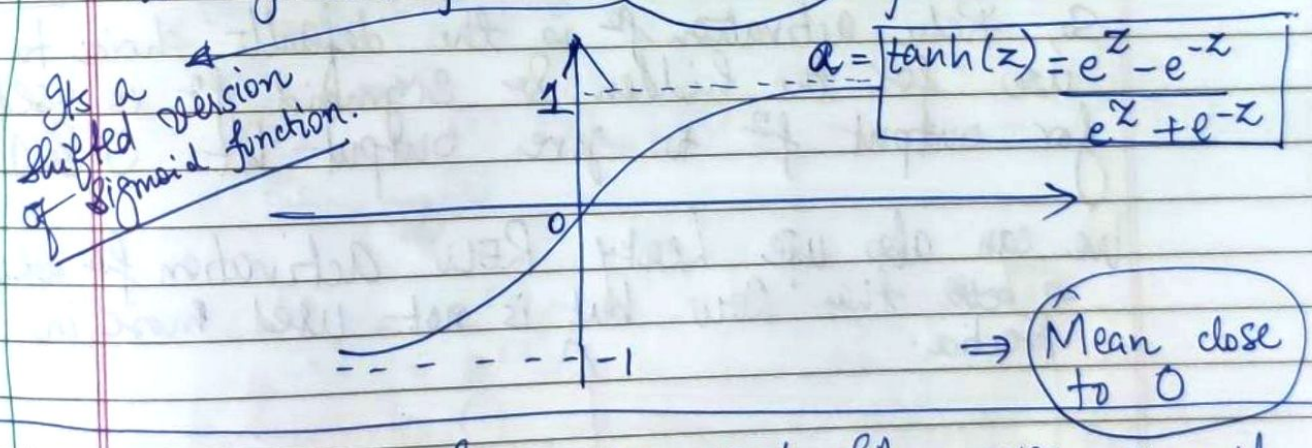


Given x:

$$z^{[1]} = W^{[1]}x + b^{[1]}$$
$$a^{[1]} = \sigma(z^{[1]})$$ replaced by $g(z^{[1]})$
$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$
$$a^{[2]} = \sigma(z^{[2]})$$ replaced by $g(z^{[2]})$

Here g is a non-linear f^n that may not be the sigmoid f^n.

Activation f^n almost always works better than the sigmoid f^n ⇒ tanh$x$ function.

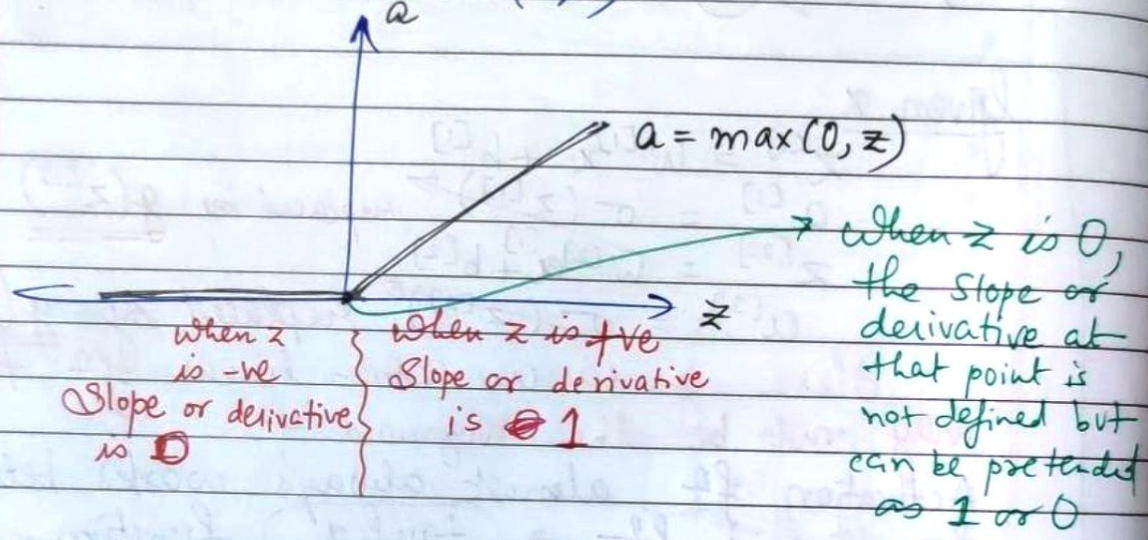


Almost we ~~have~~ use tanh f^n over sigmoid function because it superior but the exception is output layer if it is b/t 0 and 1 so it makes sense for $\hat{y}$ to be b/t 0 & 1 rather than b/t -1 & 1.

One of the drawback of sigmoid or tanh f^n is if z is very large than the gradient or the derivative or the slope of this function becomes very small, so it slows down gradient descent

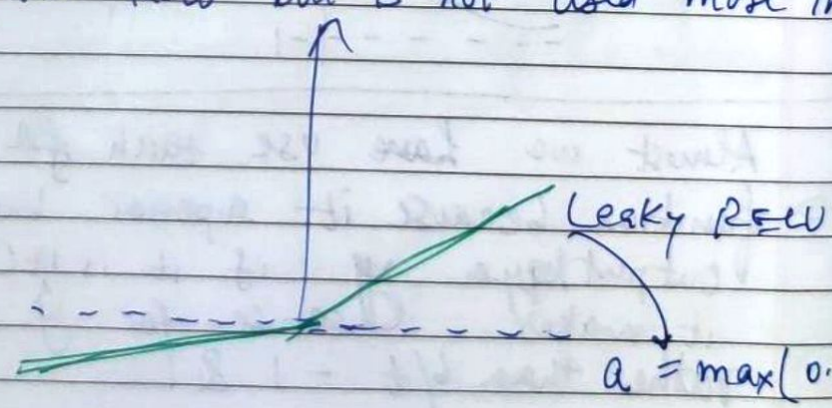
Rectified Linear function (RELU function) is used in this case whose formula is

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



So, Relu activation f^n is the default choice to use for the hidden & sigmoid f^n is used for output f^n to give output b/t 0 & 1.

We can also use Leaky RELU activation f^n which is better than RELU but is not used more in practice.



The main advantage of using RELU f^n over tanh or sigmoid function, is the value of slope will be almost different from 0, which will run faster, if the output is close to 0 than, it slows down the running.

→ Conclusion

So we never use sigmoid function except the output layer for binary classification as tanh is always superior than sigmoid, we use by default relu f^n for the hidden layer

→ Why do you need a non-linear Activation functions?

If we use linear activat^n function $(g(z) = z)$ ⇒ (linear activation function or identity activation function) which outputs whatever we input.

Given x :-

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

$$a^{[1]} = z^{[1]}$$

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

$$a^{[2]} = z^{[2]}$$

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

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

Substitute (1) in (2)

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

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

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

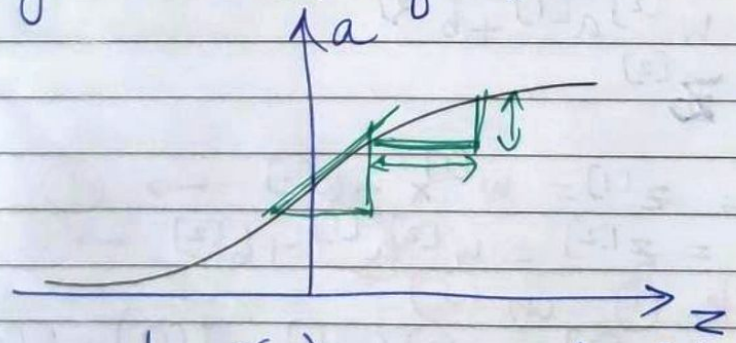
So, to use linear f^n in hidden layer is useless koz the composition of 2 linear f^n is itself a linear f^n

So unless u have non-linearity, you are not computing more interesting functions even as u go deeper into the network, there is just one place where u use linear activation function G of z equal to z ($g(z) = z$) and that's when we do machine learning on regression problem so if y is a real no, for ~~for eg~~ :- if you are trying to predict housing prices ($y \in \mathbb{R}$) anywhere between \$0 to \$1,000,000 so that ($\hat{y} \in \mathbb{R}$) ~~here~~ but that the hidden layer should not use any activation f^n ~~that~~ they can use RELU, tanh or Leaky Relu.

So, one place u use it only the linear activat^n f^n only in output layer. but other than that using linear activat^n f^n in hidden layer except some special circumstances is not allowed/rare.

DERIVATES OF ACTIVATION FUNCTION

I. Sigmoid activation function.



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

$$\frac{d}{dz} g(z) = \text{slope of } g(x) \text{ at } z = \frac{(e^{-z}+1)-1}{(1+e^{-z})^2}$$

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

$$= g(z)(1 - g(z)) = a(1-a)$$

$$\boxed{g'(z) = a(1-a)}$$

i) If $z \to$ large $z = 10$ $\quad g(z) = \frac{1}{1+0} = 1$

$e^{-z} = 0$

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

$1(1-1)$

ii) If $z \to$ small (-ve) $z = -10$ $\quad g(z) = \frac{1}{1+\infty} = 0$

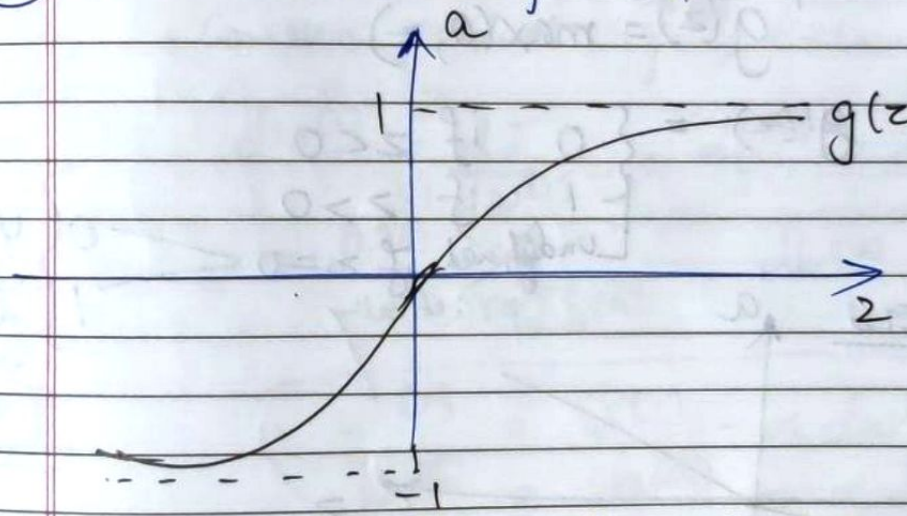
$e^{-z} = \infty$

$\frac{d}{dz} g(z) = 1(1-0) = 1$

iii) If $z = 0$ $\quad g(z) = \frac{1}{1+e^0} = \frac{1}{2}$

$\frac{d}{dz} g(z) = \frac{1}{2}\left(1 - \frac{1}{2}\right) = 1/4$

II Tanh Activation function



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

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

$= 1 - (\tanh(z))^2 = 1 - [g(z)]^2$

$$\frac{d}{dz}\left(\frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}\right) = \frac{(e^{z} + e^{-z})(e^{z} + e^{-z}) - (e^{z} - e^{-z})(e^{z} - e^{-z})}{(e^{z} + e^{-z})^2}$$

$$= \cancel{\frac{-4e^{z}e^{-z}}{(e^{z} - e^{-z})^2}}$$

$$= 1 - \left[\frac{(e^{z} - e^{-z})}{(e^{z} + e^{-z})}\right]^2 = 1 - [\tanh(z)]^2$$

$$\boxed{a = g'(z) = 1 - a^2}$$

i) If $z = 10$, (large) $\tanh(z) \approx 1$ $\left(\frac{\infty}{\infty}\right)$.

$g'(z) \approx 0$

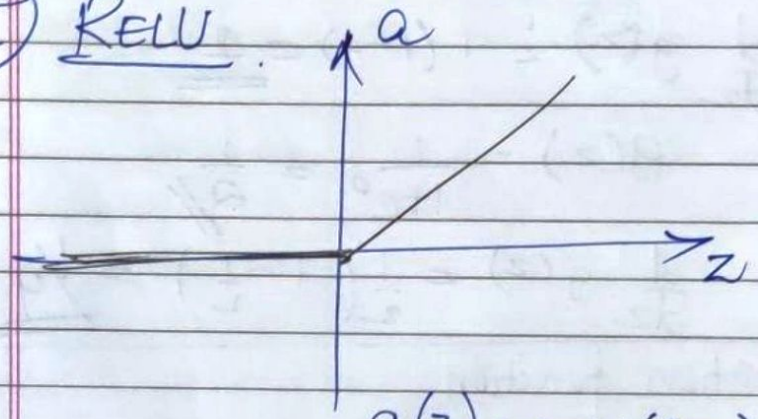
ii) If $z = -10$ (-ve) (small) $\tanh z \approx -1$ $\left(\frac{-\infty}{\infty}\right)$.

$g'(z) \approx 0$

iii) If $z = 0$, $\tanh z = 0$

$g'(z) = 1$

III) RELU

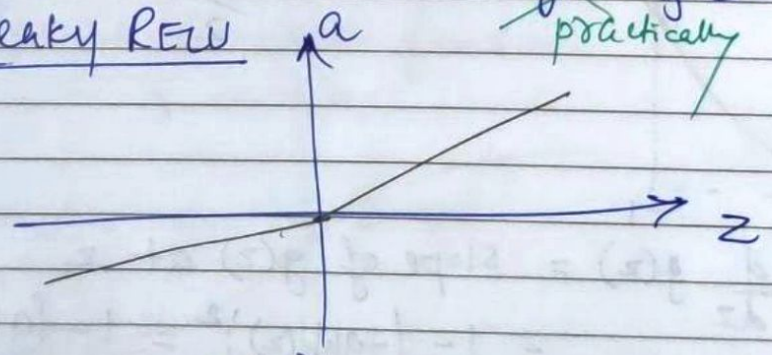


$$g(z) = \max(0, z)$$

$$g'(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z > 0 \\ \cancel{\text{undefined}} \text{ practically} & \text{if } z = 0 \end{cases}$$

0, 1 } you can decide any one of them

IV) Leaky RELU



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

$$g'(z) = \begin{cases} 0.01, & \text{if } z < 0 \\ 1, & \text{if } z > 0 \\ \cancel{\text{undefined}}, & \text{if } z = 0 \end{cases}$$

0.01, 1 } you can decide any one of them

GRADIENT DESCENT IN NEURAL NETWORKS

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

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

Dimensions

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

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

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

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

Cost function: $J(w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}, y)$

$a^{[2]}$

Gradient descent:-

Repeat {

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

$$dw^{[1]} = \frac{dJ}{dw^{[1]}}, \quad db^{[1]} = \frac{dJ}{db^{[1]}}, \ldots$$

$$dw^{[2]} = \frac{\partial J}{\partial w^{[2]}}, \quad db^{[2]} = \frac{\partial J}{\partial b^{[2]}}$$

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

$$b^{[2]} = b^{[2]} - \alpha\, db^{[1]}$$

Formulas for computing derivates

Forward propagation

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

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

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

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

for binary classificat^n output f^n is sigmoid f^n

Backpropagation :-

① $dZ^{[2]} = A^{[2]} - Y$

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

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

$\cancel{db^{[2]} = \frac{1}{m} dZ^{[2]} A}$

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

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

③ $db^{[2]} = \frac{1}{m}$ np.sum($dZ^{[2]}$, axis = 1, keepdims = True)

to take sum of element → sum horizontally (row)

it allows python to keep a check that it does not return a rank 1 matrix like this $(n,)$

④ $\underbrace{dZ^{[1]}}_{(n^{[1]}, m)} = \underbrace{W^{[2]T} dZ^{[2]}}_{(n^{[1]}, m)} * \underbrace{g^{[1]\prime}(Z^{[1]})}_{(n^{[1]}, m)}$

elementwise product

⑤ $dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$

⑥ $\underbrace{db^{[1]}}_{(n^{[1]}, 1)} = \frac{1}{m}$ np.sum($dZ^{[1]}$, axis=1, keepdims = True)

BACK PROPAGATION INTUITION

$$x \to \boxed{z^{[1]} = W^{[1]}x + b^{[1]}} \to \boxed{a^{[1]} = \sigma(z^{[1]})} \to \boxed{z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}} \to \boxed{a^{[2]} = \sigma(z^{[2]})} \to \boxed{L(a^{[2]}, y)}$$

$W^{[1]}$, $b^{[1]}$, $W^{[2]}$, $b^{[2]}$

$dW^{[1]}$, $db^{[1]}$, $dz^{[1]}$, $da^{[1]}$, $dW^{[2]}$, $db^{[2]}$, $dz^{[2]}$, $da^{[2]}$

$$dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]\prime}(z^{[1]})$$

element wise product

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

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

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

$$\underset{(n^{[1]},1)}{dz^{[1]}} = \underset{(n^{[1]},n^{[2]})}{W^{[2]T}} \underset{(n^{[2]},1)}{dz^{[2]}} * \underset{(n^{[1]},1)}{g^{[1]\prime}(z^{[1]})}$$

$$dz^{[2]} = a^{[2]} - y.$$

$$dW^{[2]} = dz^{[2]} a^{[1]T} \Rightarrow \text{"}dW\text{"} = dz \cdot x$$

$$db^{[2]} = dz^{[2]}$$

→ Note :- Always check the ~~eliminel~~ dimensions of matrices like $W^{[2]}$, $z^{[2]}$, $dz^{[2]}$, $z^{[1]}$, $dz^{[1]}$, $a^{[1]}$, $a^{[2]}$ then that will eliminate most of the bugs.

$$dW^{[1]} = dz^{[1]} \cdot x^T$$

$$db^{[1]} = dz^{[1]}$$

$$\boxed{x^T = a^{[0]T}}$$

6 Eq^n's (Most imp)

1. $dz^{[2]} = a^{[2]} - y.$
2. $dW^{[2]} = dz^{[2]} a^{[1]T}$
3. $db^{[2]} = dz^{[2]}$
4. $dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]\prime}(z^{[1]})$
5. $dW^{[1]} = dz^{[1]} x^T$
6. $db^{[1]} = dz^{[1]}$

Vectorised implementation

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

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

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

$$dZ^{[1]} = W^{[2]T} dZ^{[2]} * g^{[1]\prime}(Z^{[1]})$$

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

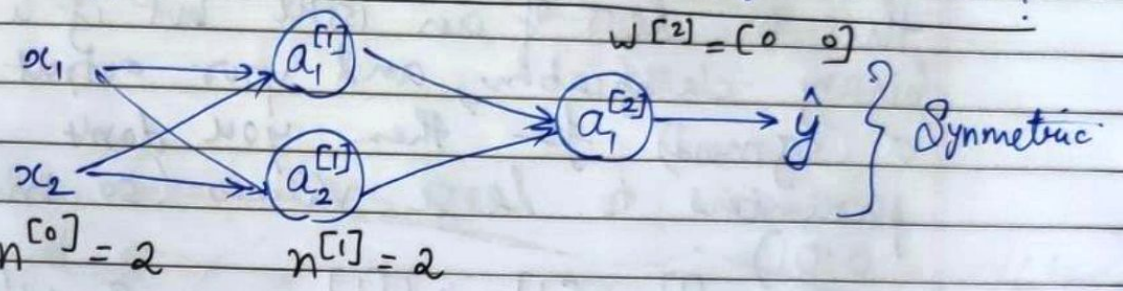
$$db^{[1]} = \frac{1}{m} \text{np.sum}(dZ^{[1]}, \text{axis}=1, \text{keepdims}=\text{True})$$

WORLD STAR™ Date: Page:

RANDOM INITIALISATION

For Logistic Regression, it was ok to initialise the weights to zero, but for neural network if u initialise the weights to zero & apply gradient descent than it won't work.

→ What happens if you initialise weights to zero?



$$W^{[1]} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \qquad b^{[1]} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

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

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

exactly the same

$$dW = \begin{bmatrix} u & v \\ u & v \end{bmatrix} \qquad W^{[1]} = W^{[1]} - \alpha dW$$

$$\begin{bmatrix} - & - & - & - \\ - & - & - & - \end{bmatrix}$$

2 Rows exactly same

So, there is no meaning of running gradient descent for 2 training examples, for large neural network also same argument continues and there is no point of computing gradient descent

The solution to this is INITIALISE THE PARAMETER RANDOMLY

random initialisat

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

generally we multiply by 0.01 to initialise it with very small random values.

same goes for W_2, b_2

You can initialise b → $b^{[1]} = np.zero((2,1))$ with zero, as W is initialised by random values

Why multiplying by 0.01?

If W is too large, you are more likely to end up even at very start of training with very large values of Z which causes your tanh or a sigmoid activation to be saturated or slowing down learning if you don't have any sigmoid or tanh activation f^n throughout your neural network this is less of an issue but if u are doing binary classification, and your output unit is σ (sigmoid) f^n then you don't initialise parameters to large values (so multiplying by 0.01)

① $z^{[1]} = W^{[1]}x + b^{[1]}$

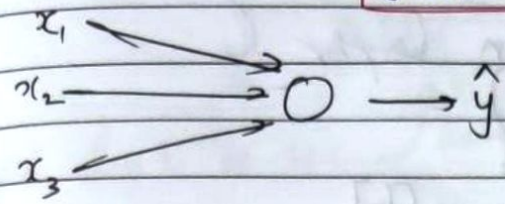
② $a^{[1]} = g^{[1]}(z^{[1]})$

So $W^{[1]}$ should be initialized by small values.

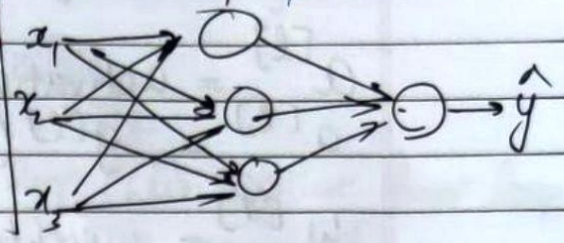
```
W[1] = np.random.randn((2,2)) * 0.01
b[1] = np.zeros((2,1))
W[2] = np.random.randn((1,2)) * 0.01
b[2] = ~~np.zeros~~ 0
```

DEEP L- LAYER NEURAL NETWORK

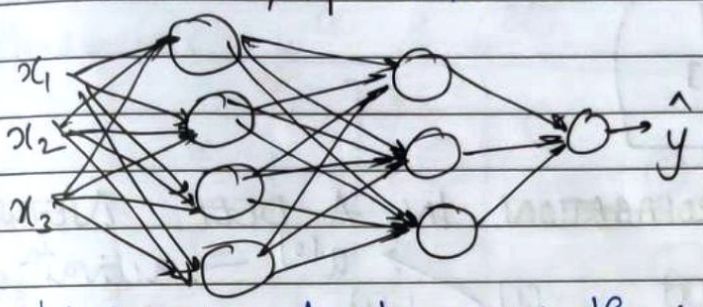
Logistic Regression / 1 layer NN (SHALLOW)



1 hidden layer / 2 layer NN



2 hidden layer | 3 layer NN



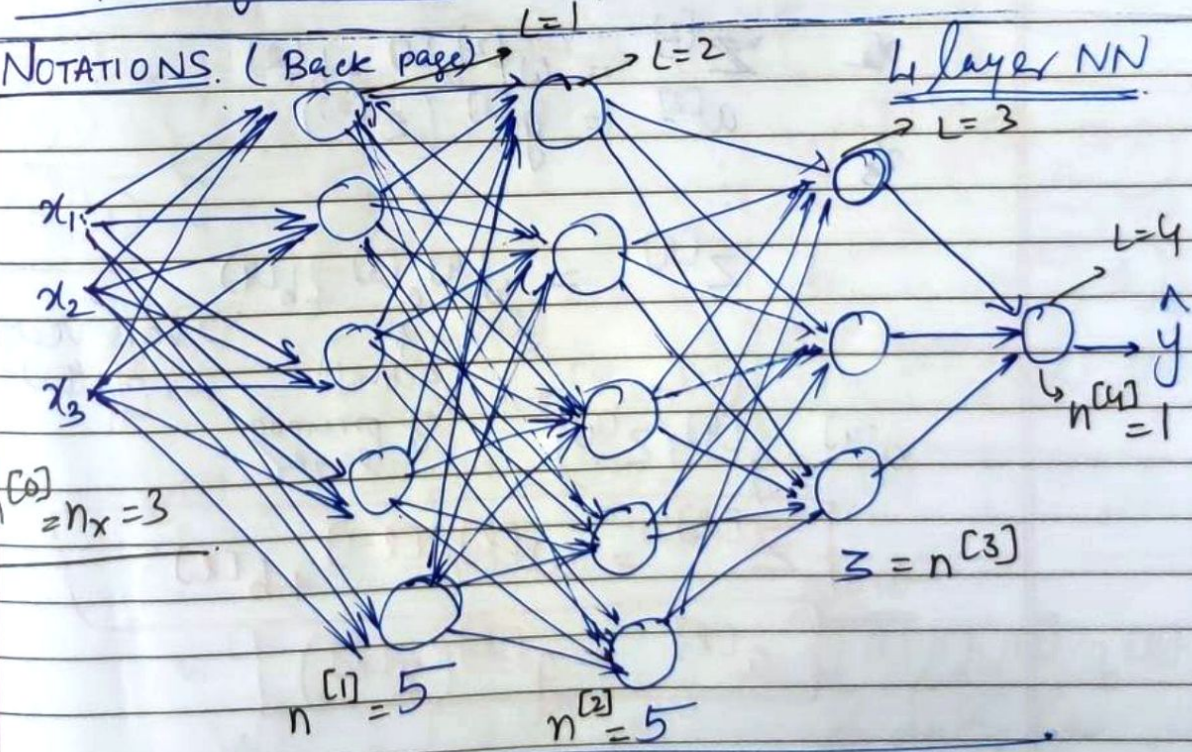
Similarly.
5 hidden layer / 6 layer NN

(DEEP)

Note:- we don't count the input layer to calculate layers of NN (we only count O/p & hidden layer)

NOTATIONS. (Back page)

4 layer NN



$L = 4$ (No of layers)

$n^{(l)}$ = (no of nodes or no of units in layer l)

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

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

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

$b^{[2]} \rightarrow$ to compute $z^{[L]}$

input $\rightarrow X = a^{[0]}$

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

$\rightarrow$ FORWARD PROPAGATION IN A DEEP NETWORK

X : $z^{[1]} = w^{[1]} X + b^{[1]}$ $\rightarrow a^{[0]} \rightarrow$ activatn of layer 0.

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

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

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

$\vdots$

$z^{[4]} = w^{[4]} a^{[3]} + b^{[4]}$ $\rightarrow$ bais vector of that layer

$a^{[3]}$: activatn of previous layer

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

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

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

Vectorized

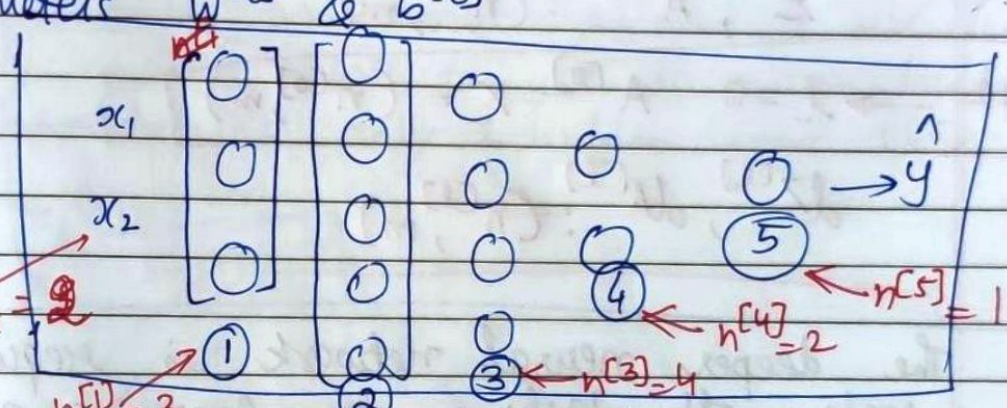
$$Z^{[1]} = W^{[1]}A^{[0]} + b^{[1]} \qquad A^{[1]} = g^{[1]}(Z^{[1]})$$
$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[1]} \qquad A^{[2]} = g^{[2]}(Z^{[2]})$$

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

for $l = 1$ to 4: using for loop (here explicit ~~for~~ loop is required, no option remaining).

⟹ GETTING DIMENSIONS OF MATRIX CORRECT

Parameters $W^{[l]}$ & $b^{[l]}$



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

$(3,1) = (3,2)(2,1) + (3,1)$ $(n^{[1]},1) = (n^{[1]},n^{[0]})(n^{[0]},1) + (n^{[1]},1)$

$$W^{[1]} : (n^{[1]}, n^{[0]})$$
$$W^{[2]} : (5,3) \Rightarrow (n^{[2]}, n^{[1]})$$

$$Z^{[2]} = W^{[2]} \cdot a^{[1]} + b^{[2]}$$

$(5,1) = (5,3)(3,1) + (5,1)$

$$W^{[3]} = (4,5)$$
$$W^{[4]} = (2,4)$$
$$W^{[5]} = (1,2)$$

$$\boxed{W^{[l]} : (n^{[l]}, n^{[l-1]})}$$
$$\boxed{b^{[l]} : (n^{[l]}, 1)}$$
$$\boxed{dW^{[l]} : (n^{[l]}, n^{[l-1]})}$$
$$\boxed{db^{[l]} : (n^{[l]}, 1)}$$

NOTE

$W^{[l]}$ & $dW^{[l]}$, $b^{[l]}$ & $db^{[l]}$ has same dimensions

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

$Z^{[1]} = W^{[1]} X + b^{[1]} \rightarrow (n^{[1]}, m)$

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

m training examples.

If 1 training eg then

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

NOTE:-

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

$Z^{[l]}, A^{[l]} : (n^{[l]}, m)$

$\boxed{\text{except } l \rightarrow l = 0 \quad A^{[0]} = X = (n^{[0]}, m)}$

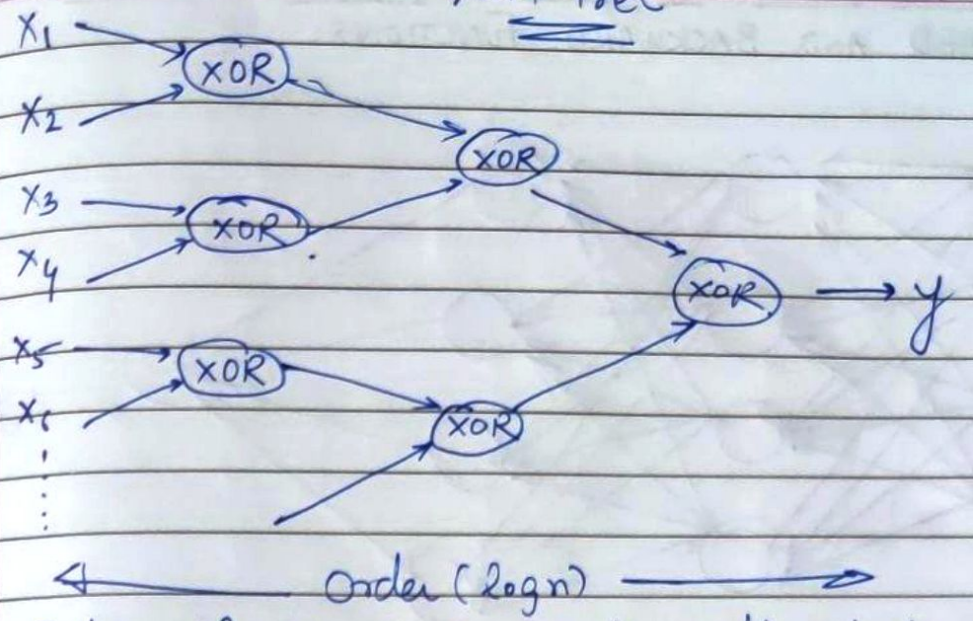
$dZ^{[l]}, dA^{[l]} : (n^{[l]}, m)$

The deeper neural network is required to perform the tasks on deep learning effectively as it divides the tasks into hidden layers and the task can be done more efficiently

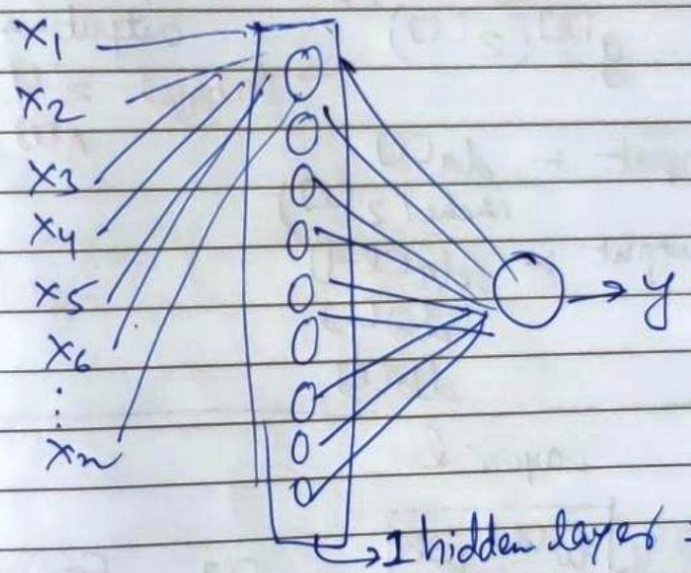
CIRCUIT THEORY AND DEEP LEARNING.

There are functions you can compute with a "small" L-Layer deep neural network that shallower networks require exponentially more hidden units to compute.

XOR Tree



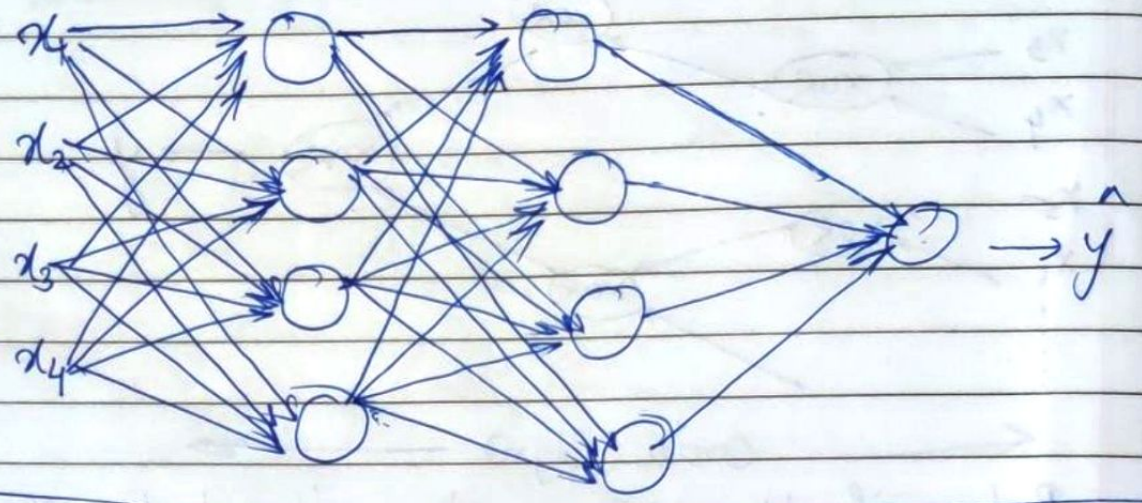
But if u are not allowed to use deeper NN, u don't have to use so many hidden layer u have to use only 1 hidden layer then



→ 1 hidden layer ⇒ exponentially large to compute all 2^n computations.

BUILDING BLOCKS OF DEEP NEURAL NETWORKS

FORWARD AND BACKWARD FUNCTIONS.



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

→ Forward

$$Z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}$$ input → $a^{[l-1]}$

$$a^{[l]} = g^{[l]}(Z^{[l]})$$ output → $a^{[l]}$, cache($Z^{[l]}$)

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

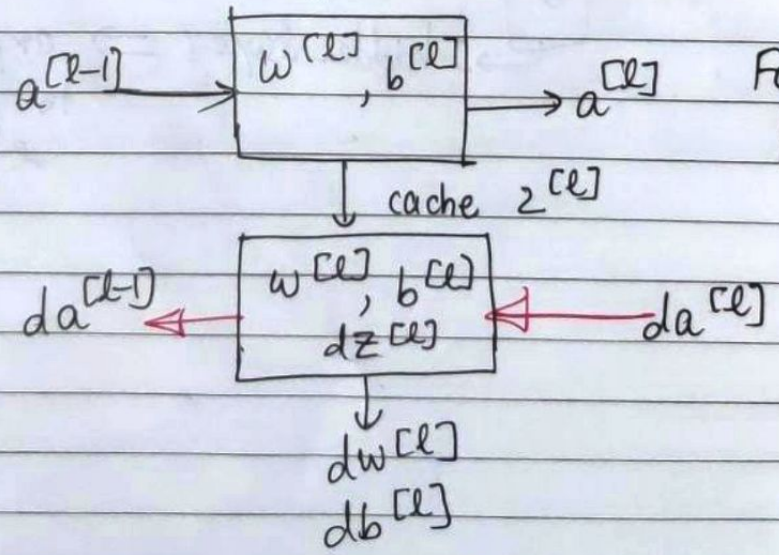
$A^{[l]} = g^{[l]}(Z^{[l]})$

→ Backward

Input :- $da^{[l]}$
cache($Z^{[l]}$)

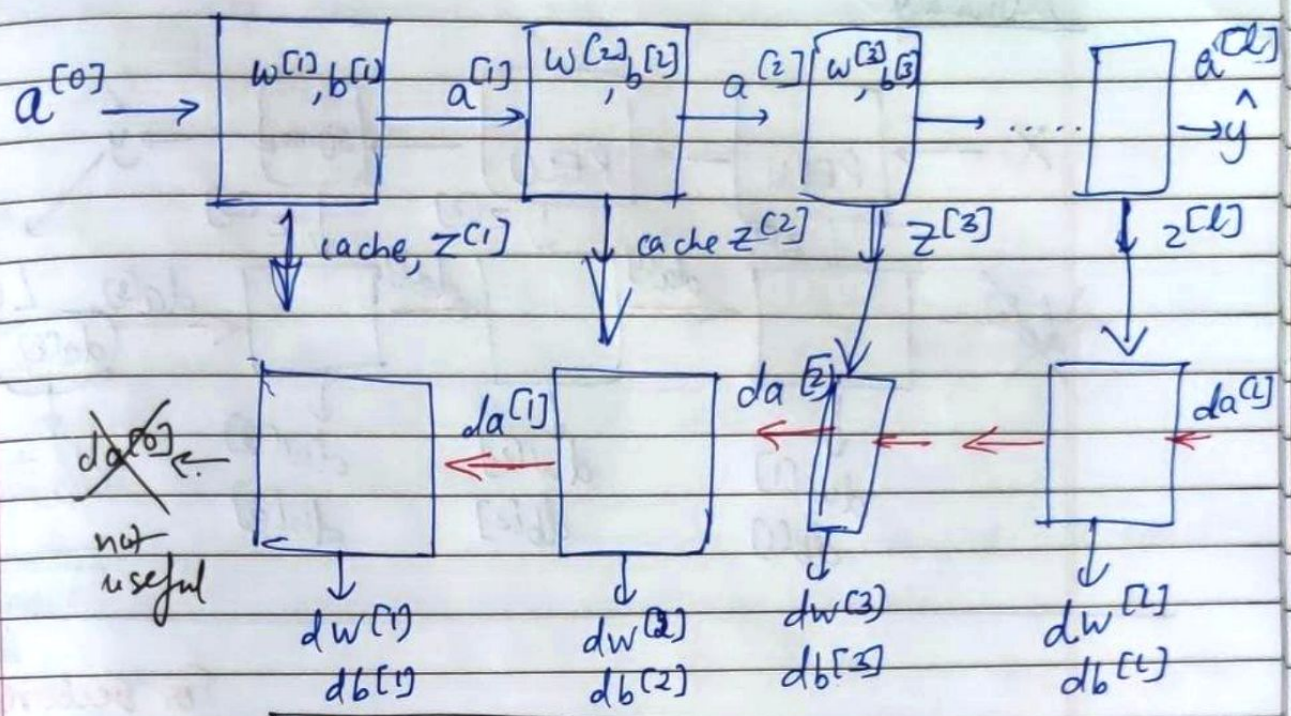
output :- $da^{[l-1]}$
$dW^{[l]}$
$db^{[l]}$

Layer l.



Forward Propagation

Backward propagation



$$\boxed{\begin{aligned} w^{[l]} &:= w^{[l]} - \alpha dw^{[l]} \\ b^{[l]} &:= b^{[l]} - \alpha db^{[l]} \end{aligned}}$$

Formulas for forward propagation is written ~~ahead~~ on previous page. Formulas for backward propagation is as follows:

$$\left.\begin{aligned} dz^{[l]} &= da^{[l]} * g^{[l]\prime}(z^{[l]}) \\ dw^{[l]} &= dz^{[l]} \cdot a^{[l-1]} \\ db^{[l]} &= dz^{[l]} \\ da^{[l-1]} &= w^{[l]T} \cdot dz^{[l]} \end{aligned}\right\}$$

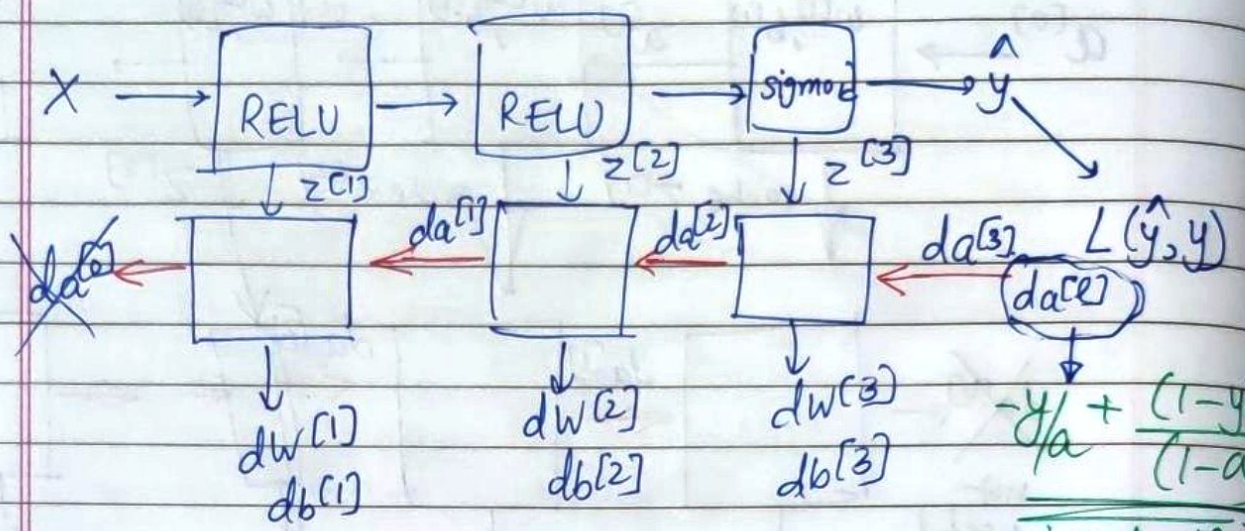
Substitute in this!

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

Vectorized

1) $dZ^{[l]} = dA^{[l]} * g^{[l]\prime}(Z^{[l]})$
2) $dW^{[l]} = \frac{1}{m} dZ^{[l]} A^{[l-1]T}$
3) $db^{[l]} = \frac{1}{m}$ np.sum($dZ^{[l]}$, axis = 1, keepdims = True)
4) $dA^{[l-1]} = W^{[l]T} \cdot dZ^{[l]}$

Summary:



For vectorised initialize backprop by:

$$dA^{[l]} = \left(-\frac{y^{(1)}}{a^{(1)}} + \frac{(1-y^{(1)})}{(1-a^{(1)})}\right) \cdots \frac{-y^{m}}{a^{m}} + \left(\frac{1-y^{(m)}}{1-a^{(m)}}\right)$$

PARAMETERS V/S HYPERPARAMETERS

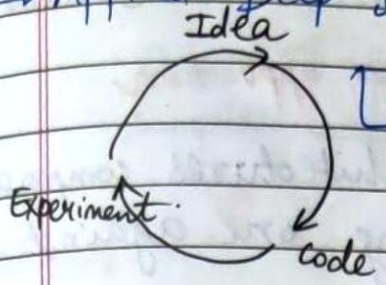
→ What are Hyperparameters?

Parameters: $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, W^{[3]}, b^{[3]} \ldots$

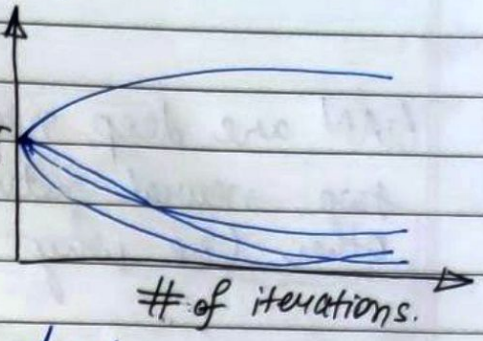
HYPERPARAMETERS:
- Learning Rate α
- # Iterations
- # hidden layers L
- # hidden units $n^{[1]}, n^{[2]}, \ldots$
- choice of activation function

others
↳ Momentum, Mini batch size, regularization, ...

→ Applied Deep learning is a very empirical Process.



$\alpha =$



You just need to keep trying a lot, and find the best value of hyperparameters

Also, CPU's & GPU's keep on changing, so exact values of hyperparameters can't be removed