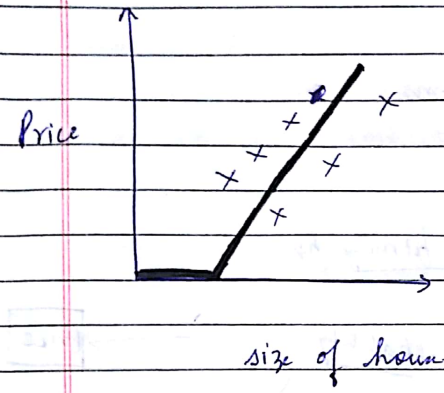
# Deep learning

Date : ..../..../.....

## Housing Price Prediction



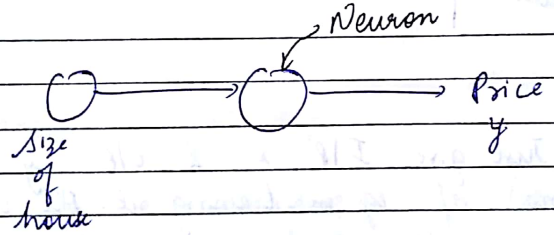

Because price can-not be negative.

We have to predict price as a fun$^n$ of size of the house.

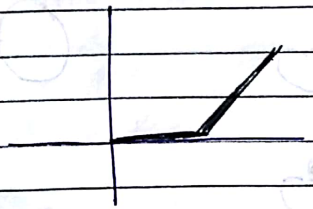
4 Weeks

4hr 16

## Simplest possible neural n/w




Neuron → Inputs size of house, computes the linear fun$^n$.
O/P the price.

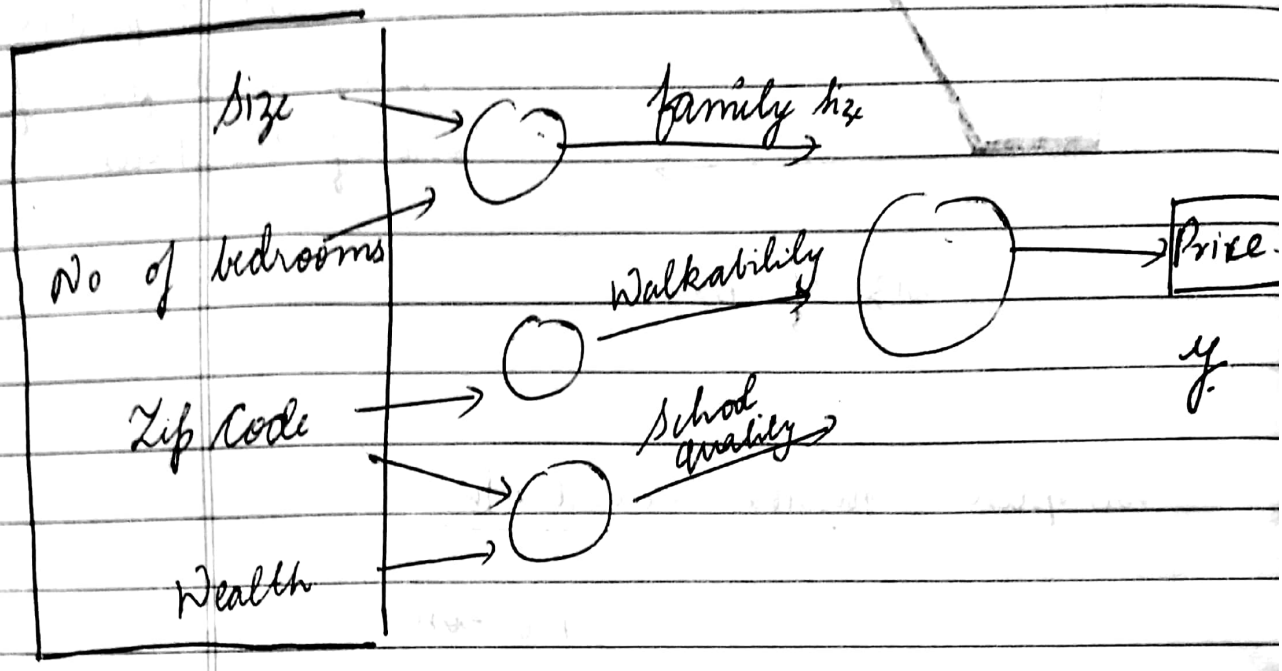


This fun$^n$ ~~lol~~ which goes 0 sometimes and sometimes take off as straight line. This fun$^n$ is called relu. This is rectified linear unit. Rectify means taking a max of zero.

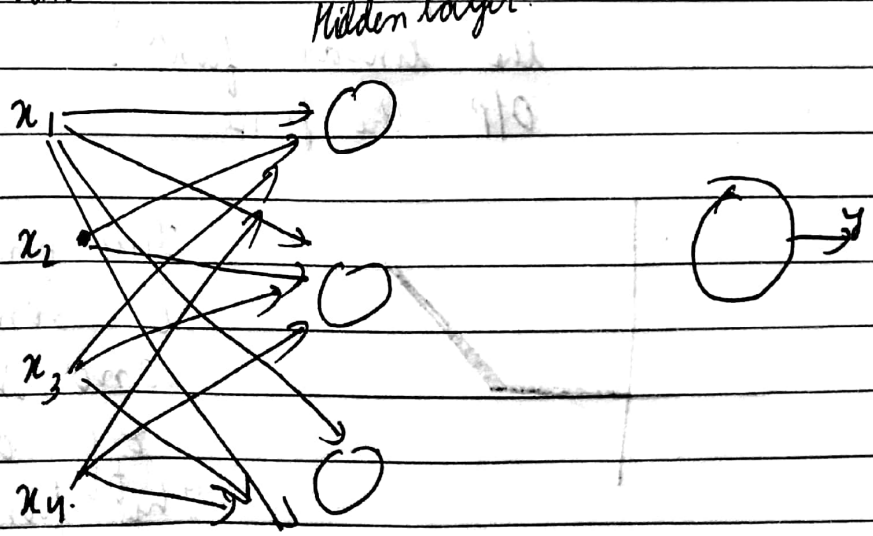
UPCP

Predicting price of house based on several features.

→ Size of house
→ No of bedrooms




Just give I/P x & O/P y for a no of egs in training set. All things in the middle will be figured out on its own



Each hidden unit of hidden layer takes all 4 I/P features. So eg if first node represent family size & it depends only on I/P $x_1$ & $x_2$. Rather we would say neural node to that you decide whatever you want this node to be & we will give you all 4 I/P features & you can compute whatever you want to.

I/P layer & hidden layer in this case is said to be densely connected bcz every I/P feature is connected to every one of the circles of hidden layer.

If we give large training data of x, y, the neural n/w gives good fun$^{n}$s that actually map from x to y.

## Supervised learning

In supervised learning, You have an I/P x. You want a fun$^{n}$ that map it to O/P y.

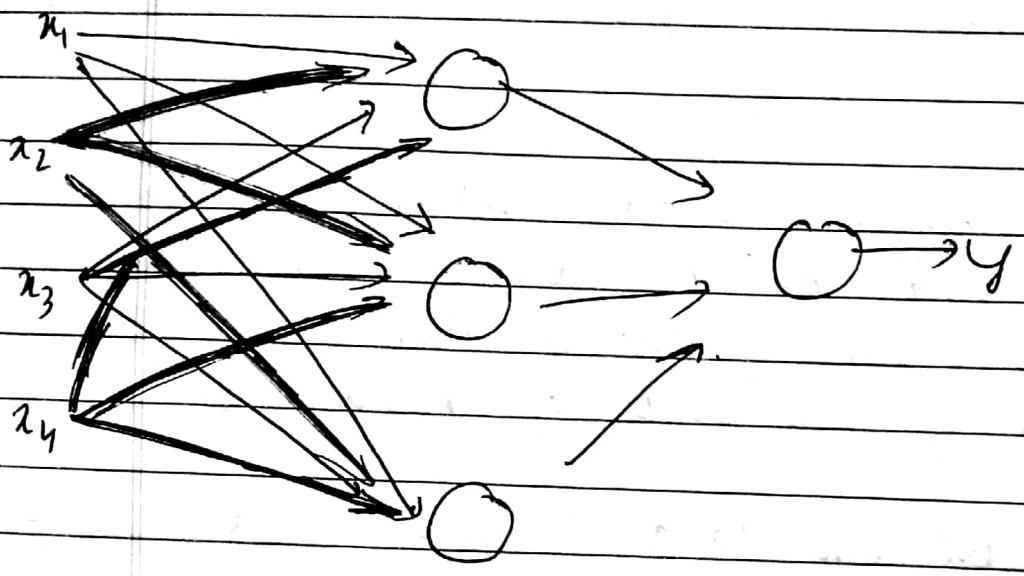
eg →

| | I/P (x) | O/P (y) | App$^{n}$ |
|---|---|---|---|
| → | Home features | Price. | Real Estate |
| → | Audio | text transcript | Speech recog$^{n}$ |
| → | English | Chinese | Machine Tras$^{n}$ |
| → | Image, Radar info | Position of other cars. | Autonomous driving |

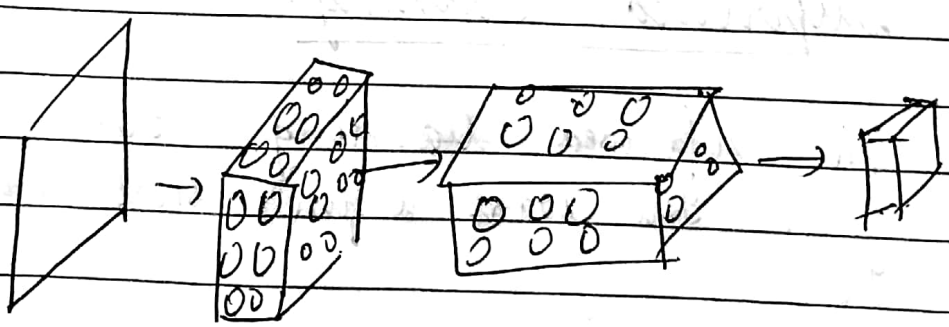
→ For image applications we use CNN
→ For sequence data like audio, we use RNN
→ languages, complex versions of RNN

# Neural Network examples

1. Standard NN




2. Convolutional NN (CNN)



Image

It is used mainly for images.

Date : ../../....

# 3. Recurrent Neural Networks

## Structured data

Means database of data

eg → House price pred$^n$

| Size | bedrooms | ... | Price |
|---|---|---|---|
| 2164 | 3 | | 400 |
| 1600 | 3 | | 330 |
| ⋮ | ⋮ | | ⋮ |
| 3000 | 4 | | 540 |

Here Size & no of bedrooms are known.

eg →
Click on ad. There is info about user such as age, some info of add and then labels Y which we have to predict.

→ In this, each of the features like size, no of bedrooms, age of user has very well defined meaning

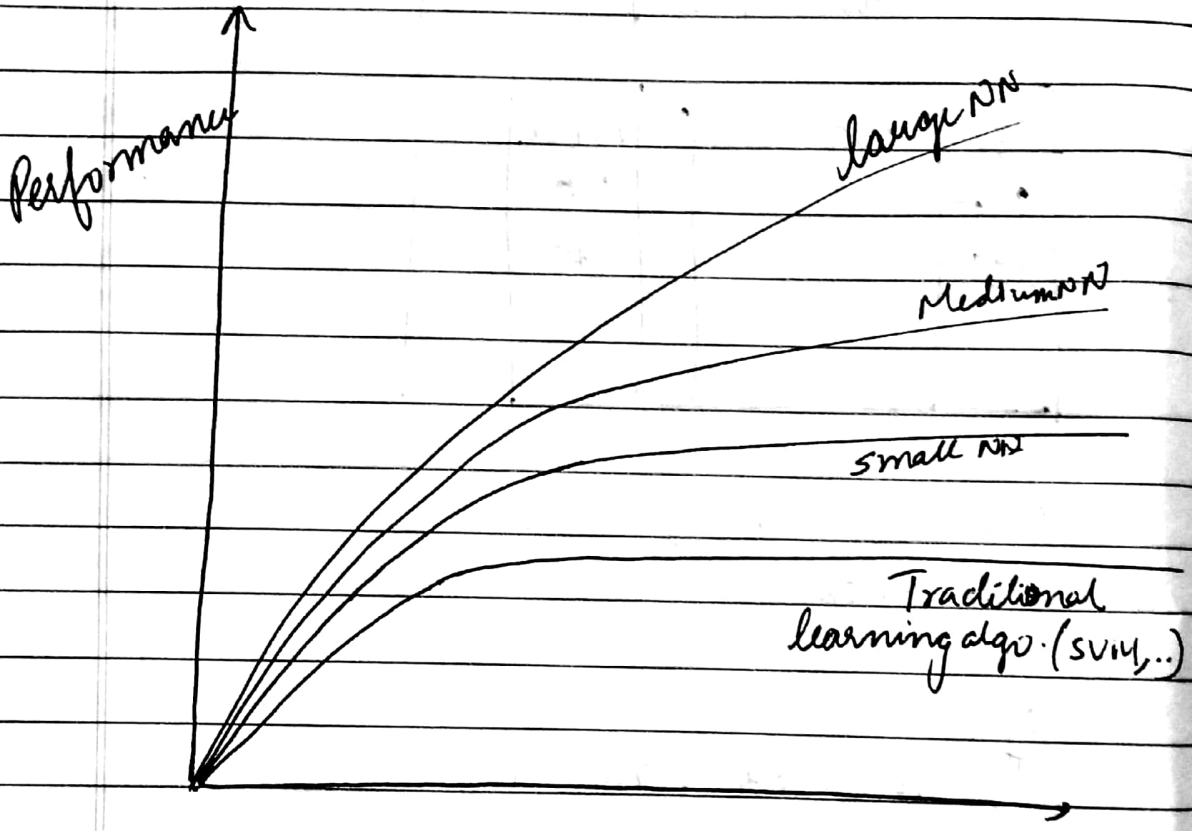
## Unstructured data

→ Refers to things, audio, raw audio or image where you want to recognize what is

Date

might be pixel values in an image or individual words in a piece of text.

Scale drives deep learning progress




If you want to hit high level of performance then you need 2-things first.

→ Train bigger NN

→ Large amt of data.

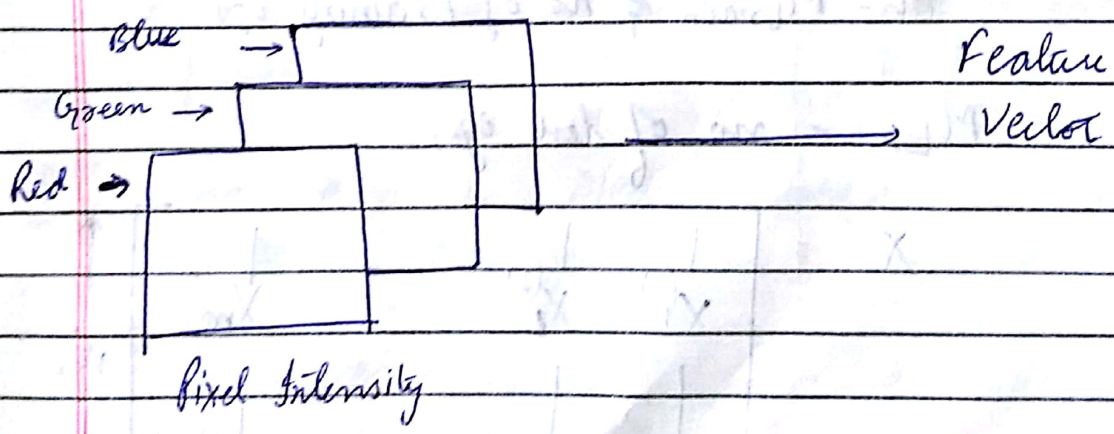
Date : ...../...../.....

# Binary Classifⁿ

eg →
I/P → an image
O/P → Cat (1) or Non-Cat (0)

This is an eg of binary classifⁿ problem. You might have an I/P of an image and want to O/P a label to recognize this image as being either a cat(1) or non cat (0)

Q How an image is stored in comp?
To store an image, comp. supposes 3 separate matrices corresponding to red, green and blue color channels of this image.
So if size of image is 64 × 64 pixels.
Then we have 3 matrices of size 64 × 64 corresponding to red, green and blue ~~comp~~ pixel intensities for your image.




Unrolling pixel values to feature vector:

x = [ red matrices ~~gree~~ all in column.
green

Size of feature vector $x = 64 \times 64 \times 3 = 12288$

$n_x$ represents the dimension of I/P feature vector = 12288

⇒ Single training eg is represented by $(x, y)$

$x \in R^{n_x}$

$y \in \{0, 1\}$

m training egs

$\{(x^{(1)}, y^{(1)}) \rightarrow$ for training eg 1
$(x^{(2)}, y^{(2)}) \rightarrow$
$\vdots$
$(x^{(m)}, y^{(m)})$
$\}$.

$M \simeq M_{train}$ = no of training egs

$M_{test}$ = no of test egs.

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

$n_x$

← no of training egs →

Size of feature vector $x = 64 \times 64 \times 3 = 12288$

$n^{[0]}, n_x$ represents the dimension of I/p feature vector = 12288

⇒ Single training eg is represented by $(x, y)$

$x \in R^{n_x}$

$y \in \{0, 1\}$

$m$ training egs

$\{(x^{(1)}, y^{(1)}) \rightarrow$ for training eg 1
$(x^{(2)}, y^{(2)}) \rightarrow$ for training eg 2
⋮
$(x^{(m)}, y^{(m)})$
$\}$.

$M \simeq M_{train}$ = no of training egs

$M_{test}$ = no of test egs.

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

$\leftarrow$ no of training egs $\rightarrow$

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

Date ../../....

$$Y = \begin{bmatrix} y^{(1)} & y^{(2)} & \cdots & y^{(m)} \end{bmatrix}$$

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

X, Y → All egs (training) are stacked together in columns of a matrix.

## Logistic Regression

It is a learning algorithm used when O/P labels (Y) is a supervised learning prob. i.e the O/P is 0 or 1 so it is a binary class$^n$ prob.

Given an I/P feature vector X corresp-onding to an image that we want to recognize as cat picture or not a cat picture.

Soln An algo to O/P a pred$^n$ called as $\hat{y}$ (y hat)

↓

It is not actual value of O/P. It is just the estimate of Y.

$X \in R^{N_x}$

Parameters of logistic regression are

$w \in R^{N_x}$, $b \in R$ (real no)

UPCP

Date

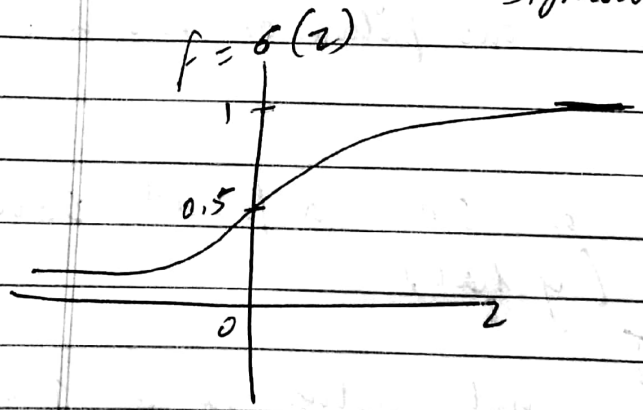
So I/P vector $x$,
Parameters $w, b$ are given.
& we have to generate O/P $\hat{y}$.

One way $\hat{y} = w^T x + b$

Not a good algo for binary classn bcz you want $\hat{y}$ to be the chance that $y=1$ ~~or $\hat{y}$~~.
So $\hat{y}$ vary b/w 0 and 1
but this eqn $\hat{y} = w^T x + b$, Acc to it $\hat{y} > 1$ or can be -ve which dont make sense bcz prob is b/w 0 & 1

logistic regression

O/P
$\hat{y} = \sigma(w^T x + b)$
↓ sigmoid fun^n.




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

$z$ = real no

If $z$ = large
so $\sigma(z) \approx \frac{1}{1+e^{\text{large}}}$
$\simeq \frac{1}{1+0} = 1$

ex

If $z$ = small or large -ve no
$\sigma(z) = \frac{1}{\text{large}} = \frac{1}{\text{large}} = 0$.

UPCP

Date / /

## Logistic regression cost function:

To train the parameters W and B of logistic regression model you need to define cost function.

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

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

To learn parameters of model,
Training set of m examples is given.
So we want to find parameters W & B so that the predictions are close to actual o/p atleast on training data.

So

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

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

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

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

Loss / error function →

Tells how well our algo is.

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

UPCP

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

This loss fun$^n$ is used in logistic reg$^n$.

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

↑ so we want $\log \hat{y}$ to be large. ⇒ $\hat{y}$ to be large.

But $\hat{y}$ is sigmoid fun$^n$ so $\hat{y}$ can't be greater than 1

so $\hat{y} = 1$

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

so want $\log(1-\hat{y})$ to be large

⇒ $1-\hat{y}$ large

⇒ $\hat{y}$ small.

$\hat{y}$ is b/w 0 & 1

so $\hat{y} = 0$

Cost fun$^n$

Measures how well you are doing on entire training data.

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

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

Loss fun$^n$ is applied to single training eg.

Cost fun$^n$ is cost of your parameters

Date : / /

So in training logistic regrn model we try to find parameters like weights such that they minimize the overall cost.

* logistic regr" can be thought of as small NN.

## Gradient Descent

Ans to logistic regression algo:

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

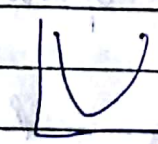
Cost fn

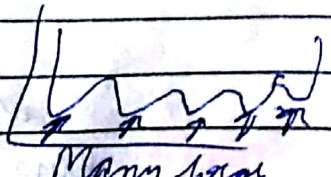
$$J(w,b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

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

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

Fun" J is a convex fun" so it will have 1 minima. This is the reason we squared the error. Now we will have 1 local minima. This is the reason we took this fun" for logistic regression.





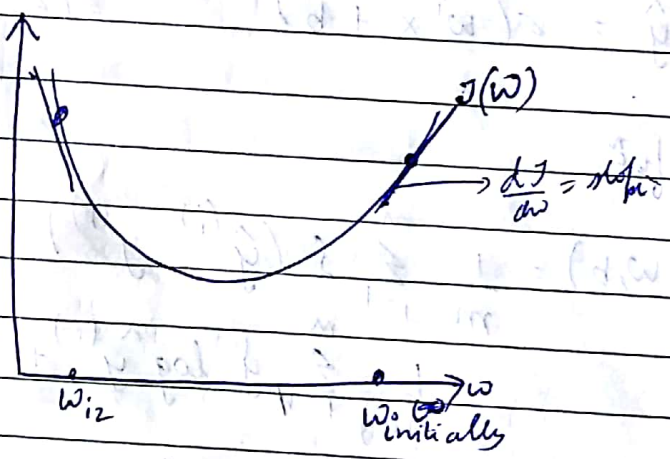
Many local minima in non

Date : ..../...../.....

→ To find good values for parameters, we first initialize them with some random values.

→ What Gradient descent does?
It starts at initial point & takes step in direction of steepest downhill.

eg. Let $J(w)$ be the fun$^n$ you want to minimize & you want to




Ignoring $\beta$ for now for easiness. Imagine it as a 1-D plot.

Updating w repeatedly

Repeat {

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

}

$\alpha \rightarrow$ learning rate & it controls how big each step we take on each iter$^n$.

Date : ...../...../.....

$\frac{dJ(w)}{dw}$ → derivative. This is basically update or change you want to do to the parameter w.

In code we will use dw to represent $\frac{dJ(w)}{dw}$

→ At $w_1$, when we take derivative of J wrt w it is +ve. Thus $w := w - \alpha \frac{dJ(w)}{dw}$

↓ +ve

So w is decremented.

& we end up taking a step to the left & so this algo slowly decreases the parameter

Case 2: If $w_{12}$ is in beginning then slope here will be -ve.

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

↓ -ve

So $w = w +$ something.

→ So whether we initialize w with minor mistake, ~~it with~~ this gradient descent will move you towards global minimum.

---

Since in logistic regression, both b and w are parameters so we do this for both w & b

$J(w, b)$

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

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

UPCP

Date : ..../..../....

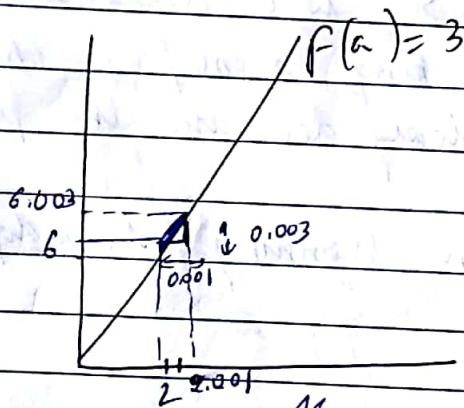
$\frac{dJ(w,b)}{dw}$ is also written as $\frac{\partial J(w,b)}{\partial w}$

In code use

dw for $\frac{\partial J(w,b)}{\partial w}$

db for $\frac{\partial J(w,b)}{\partial b}$

# Derivatives




If $a = 2$ $f(a) = 6$

$a = 2.001$ $f(a) = 6.003$

On moving a by 0.001 on right, f(a) goes 0.003 on upper side i.e 3 times a moves on right

⇒ The slope or derivative of f(a) at $a = 2$ is 3.

$$\text{slope} = \frac{\text{height}}{\text{width}} = \frac{0.003}{0.001}$$

Date : ...../...../.....

$a = 5 \quad f(a) = 15$

$a = 5.001 \quad f(a) = 15.003$

So when we bump a to the right by 0.001, f(a) goes up by 0.003. i.e 3 times a moves on right

$\Rightarrow$ slope at $a = 5$ is 3

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

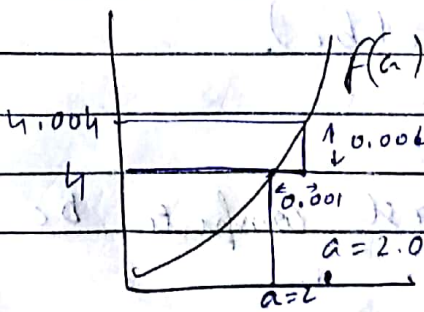
slope of function f(a) when we move by variable by tiny little amt.

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

When we ~~nudge~~ change the value of a by infinitesimally small amount, then f(a) goes three times the small amt a has been changed.

Since for all values of a, when we change a by small amt, f(a) changes by 3 times. Hence slope of fun$^n$ is same everywhere.

Eg




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

$a = 2.001 \quad f(a) = 4.004001$

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

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

UPCP

Date : ..../...../.....

In this eg, slope is diff for diff values of a.

$a = 5$ $\quad f(a) = 25$

$a = 5.001$ $\quad f(a) = 25.010$

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

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

* $\frac{d}{dF} f(a)$ $\quad f(a) = a^3$

$= 3a^2$

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

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

## Computation Graph

$J(a, b, c)$

J be fun$^n$ of 3 variables a, b & c

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

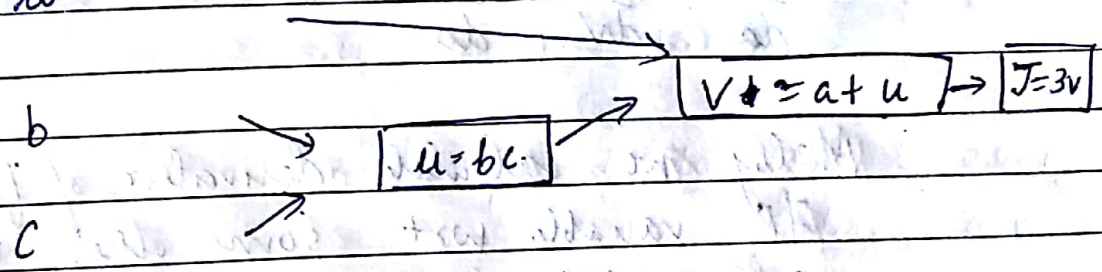
3 steps.

1.) $u = bc$ First compute bc

2.) $v = a + u$

3.) $J = 3v$.

Date : ...../...../.....




eg → a = 5 V = 11 J = 33.
b = 3 u = 6
c = 2.

Derivatives with computation graph.

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

~~J = 3V~~

To calculate the derivative of final variable J wrt v, we have gone backwards 1 step.

$\frac{dJ}{da} =$

i.e if we change value of a by small amounts, how does the value of J changes.

a = 5 → 5.001
V = 11 → 11.001
J = 33 → 33.003

$\frac{dJ}{da} = 3$ Since change in J is three times that of a.

So change in a creates change in V which creates change in J

UPCP

Date : ..../..../.....

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

Mostly we calculate derivative of final o/p variable wrt some other variable denoted by dvar.

Final o/p var → J

other variables → a, b, c, u, v in prev eg

In software what name to give to these variables can be $\frac{d\text{FinalOutputVar}}{d\text{Var}}$ or dJdvar or dvar

How to compute derivatives to find gradient descent for logistic regression

Using computation graph, we will do it.

logistic regression as follows:

Predicⁿ $\hat{y}$ is computed as follows.

$$z = w^T x + b$$

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

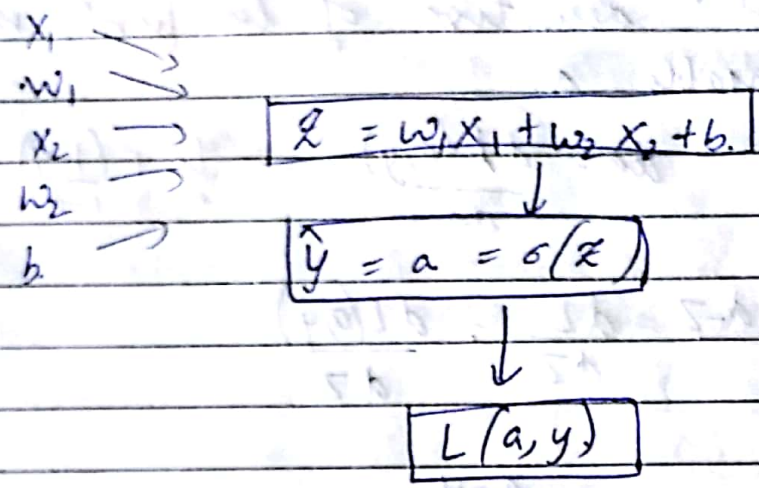
$$\mathcal{L}(a, y) = -(y \log(a) + (1-y) \log(1-a))$$

↙ Loss with 1 eg

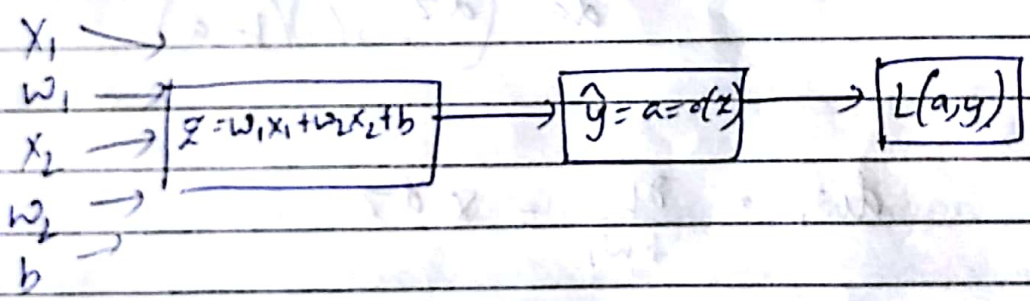
Date : / /

Consider we have only 2 features $x_1$ & $x_2$ for this example

$w_1$, $w_2$, $b$




Comp$^n$ graph




In logistic regression we want to modify the parameters $w$ and $b$ in order to reduce/minimize loss.

These are steps of forward prop$^n$ on how to compute loss of 1 example

training.

Now how to go backwards to compute the derivatives.

UPCP

Date : ...../...../.....

What we want to do is to compute derivative wrt this loss.

When going backwards, first thing is to find derivative of loss fun$^n$ wrt the variable a.

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

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

$$= a - y$$

$$= \frac{dL}{da} \times \left(\frac{da}{dz}\right) \rightarrow (a)(1-a)$$

$$dw_1 = \frac{\partial L}{\partial w_1} = x_1 dz.$$

$$dw_2 = x_2 \partial z$$

~~$dw_3 = d\cdot$~~

$$db = dz.$$

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

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

$$b = b - \alpha db$$

UPCP

Date : ..../...../.....

## Gradient descent on m examples.

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

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

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

$dw_1^{(i)}$ → $(x^{(i)}, y^{(i)})$ for each eg.

### Algo

1) Initialize $J=0$, $dw_1 = 0$; $dw_2 = 0$; $db = 0$

Use a for loop over training set and compute derivatives wrt each eg & add them up

for $i = 1$ to $m$

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

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

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

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

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

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

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

] for 2 features

UPCP

$J /= m$

$dw_1 /= m$

$db /= m$

] for computing avg.

Date : ..../...../.....

So after algo we get

$$dw_1 = \frac{\partial J}{\partial w_1}$$

So after this perform 1 step of grad descent.

$w_1 := w_1 - \alpha dw_1$

$w_2$

$b :$

# Vectorization

Date : ..../..../.....

Vectorizⁿ is basically for getting rid of explicit for loops in your code.

In Deep learning, large dataset is used for algo to give good results.

But we need to have our codes run faster.

What is vectorization ?

$$z = w^T x + b$$

$$w = \begin{bmatrix} | \\ | \end{bmatrix} \qquad x = \begin{bmatrix} | \\ | \end{bmatrix} \qquad w \in R^{N_x} \quad x \in R^{N_x}$$

If we have non vectorizⁿ implemenⁿ, we do:

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

This non vectorized impⁿ will be very slow.

In contrast vectorized implementation, just compute w transpose x directly.

```
z = np.dot(w, x) + b   // compute w^T x
```

Non vectorized implemen takes about 300 times longer than vectorized version.

Date : ..../...../.....

eg:

$u = Av$

vector matrix vector

// Non-vectorized

$u_i = \sum_j A_{ij} v_j$

```
u = np.zeros((n, 1))

for i ...
    for j ...
        u[i] += a[i][j] * v[j]
```

// Vectorized.

```
u = np.dot(A, v)
```

eg 2: You need to apply exponential operation on every element of a matrix/vector.

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

This is non vectorized implⁿ.

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

UPCP

Date : ..../..../.....

Vectorized imp$^n$ 1.

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

↑ vector ↑ vector

//Numpy has many vector value func$^n$

np.log(v) → Computes elemnet wise log
np.abs(v) → ——— absolute
np.maximum(v,0) → ——— maximum
v**2 → ——— square
1/v → ——— inverse

Logistic regression derivatives without vectorizn

$J=0, dw1=0, dw2=0, db=0$

for i=1 to m:

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

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

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

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

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

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

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

~~J=0~~

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

But if there were many ~~I/P~~ I/P features then we would have used for loop to find $dw_1, dw_2, dw_3 \ldots$

UPCP

Date : ..../..../....

for $j = 1 \ldots n_x$

$dw_j \mathrel{+}= \_\_\_\_$.

So instead of initializing each $dw_1, dw_2 \ldots$ to 0, we will make a vector $dw$

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

So now instead of for loop write

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

& later instead of $dw_1 = dw_1/m$ we write

$$dw = dw/m$$

## Vectorizing logistic regression

We will talk about how to vectorize the implemen$^n$ of logistic regression, so that we can process the entire training set that is implement a single iter$^n$ of gradient descent wrt an entire training set without even using a single for loop.

Forward prop$^n$ step of logistic regres$^n$. So if there are $m$ training egs then to make pred$^n$ on first eg we need to find:

→ $z^{(1)} = w^T x^{(1)} + b$.

→ $a^{(1)} = \sigma(z^{(1)})$

→

Date : ...../...../.....

Now to make pred$^n$ on 2nd training eg, we need to compute:

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

Similarly for all training eqs, predictions is done in this manner.

→ So in order to do predic$^n$s or carry out forward prop$^n$ steps on our M training eqs, we can do without explicitly using for loop. But how?

Soln

→ Firstly, X is a matrix we defined to be our training I/Ps stacked together in diff columns

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

X is a $(n_x, m)$ dimensional matrix.

→ Now computing $z^{(1)}, z^{(2)}, z^{(3)}$ in 1 line of code. in 1 step.

Construct a $(1, m)$ matrix which is a row vector.

$$\begin{bmatrix} z^{(1)} & z^{(2)} & z^{(3)} & \cdots & z^{(m)} \end{bmatrix}$$

& it can be enpressed as

$$= w^T X + \begin{bmatrix} b & b & \cdots & b \end{bmatrix}$$

↓ 1 × M vector or M Matrix

UPCP

Date : ..../..../.....

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

↳ Row Vector.

$$= \begin{bmatrix} w^T x^{(1)} & w^T x^{(2)} & w^T x^{(3)} & \cdots & w^T x^{(m)} \end{bmatrix}$$

$$+ \begin{bmatrix} b & b & \cdots \end{bmatrix}$$

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

$z^{(1)}$ $z^{(2)}$

So as we obtained X by taking all training eg and then stacking them horizontally.

$$Z = \begin{bmatrix} z^{(1)} & z^{(2)} & z^{(3)} & \cdots & z^{(m)} \end{bmatrix}$$

Capital Z — small z

Command :

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

w transpose

In python, it assumes x to be a real no or 1X1 matrix

Date : ...../...../.....

But when we add the vector $(w^T x)$ to real no b, it automatically takes this real no b and expands it out to this (1 x M) row vector.

→ Calculating A1, A2 .... AM all at the same step.

Just as X i.e. stacking all Training egs. in 1 matrix

$$A = [a^{(1)} \quad a^{(2)} \quad a^{(3)} \quad \dots \quad a^{(m)}]$$

$$= \sigma(Z)$$

↓

This sigmoid fun$^n$ input Z and O/Ps A.

## Vectorizⁿ logistic Regression's Gradient computation

Here we use vectorⁿ to perform gradient descent for all M training egs. all at 1 time.

→ For gradient descent we computed dz1 for first example

$dz^{(1)} = a^{(1)} - y^{(1)}$ // training eg 1

$dz^{(2)} = a^{(2)} - y^{(2)}$ // training eg 2

and so on

Define new variable $dz$.

$$dz = [dz^{(1)} \; dz^{(2)} \; dz^{(3)} \; .. \; dz^{(m)}]$$

1 x m matrix or $m^t$ dimensional vector

$$A = [a^{(1)} \; a^{(2)} \; \times \; a^{(m)}]$$

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

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

Here instead of initializing
We already removed for loop by
but we have 2nd for loop
over training egs.
we did $dw = 0$
but we still have to loop over
~~2m egs~~ training egs where
$dw \; += x^{(1)} dz^{(1)}$ for eg 1.
$dw \; += x^{(2)} dz^{(2)}$ ... 2

m times

And then $dw /= m$

Similarly for db

Date : ...../...../.....

$db = 0$

$db += dz^{(1)}$

$db += dz^{(2)}$

.

.

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

So vectorizing it.

$$db = \frac{1}{m} \sum_{i=1}^{m} dz^{(i)}$$ // $dz$ are in row vector.

so in python.

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

↳ sum of all elements.

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

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

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

In python

$dw = \frac{1}{m} \cdots$

UPCP

Date : ...../...../.....

# Implementing logistic regression

## 1. Non vectorized implementation

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

↓ cost. ↓ partial derivatives wrt cost.

for $i = 1$ to $m$:

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

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

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

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

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

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

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

$J = J/m$, $dw_1 = dw_1/m$, $dw_2 = dw_2/m$

$db = db/m$

## 2. Vectorized implementation

$$Z = w^T X + b$$

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

$$A = \sigma(Z)$$

$$dZ = A - y$$

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

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

UPCP

Date : ...../...../.....

$w = w - \alpha dw$
$b = b - \alpha db$

Using above code we implemented single iter$^{n}$ of gradient descent for logistic regression.

But for many iterations of implementing gradient descent we need for loop

## Broadcasting in Python

Technique to run codes in python faster.

eg: In matrix ; no of calories, protein, fats in 100g of different food is given.

| | Apples | beef | eggs | Potatoes |
|---|---|---|---|---|
| Carb | 56.0 | 0.0 | 4.4 | 68.0 |
| Protein | 1.2 | 104.0 | 52.0 | 8.0 |
| fats | 1.8 | 135.0 | 99.0 | 0.9 |

Goal:
Calculate %ag of carb, protein, fats from each of the four foods

In apples; (56 + 1.2 + 1.8) calories is there.
= 59

$\frac{56}{59} = 94.9\%$ % of calories in apples is from carb

UPCP

Date : ...../...../.....

For this calculation

We need to add each of 4 columns to get total no of calories in 100g of apple, beef & then divide throughout the matrix.

Can we do this without for loop.

Soln:

→ A = (3, 4) // 3X4 Matrix

→ Using 1 line of python, sum column & we will get 4 elements. Using it we get total no of cal in 100g of this food.

→ 2nd line of python line, we divide each of 4 columns by their corresponding sum

Code

```
Import numpy as np
A = np.array([[56.0, 0.0, 4.4, 68.0],
              [1.2, 104.0, 52.0, 8.0],
              [1.8, 135.0, 99.0, 0.9]])
print (A)
```

Date : ..../...../.....

cal = A.sum(axis=0)

↓ means sum vertically.

print(cal)

[59, 239, 155.4 76.9]

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

print(percentage).

↓ python broadcasting

here A is 3X4 matrix

cal is 1X4 matrix.

Here cal is already cal is 1X4 matrix.

So need for reshaping.

So in python if you are not sure of dimensions of matrix, use reshape command.

reshape is O(1) oper$^n$.

A → (3 X 4) Matrix

cal → (1 X 4) —

How can we divide (3X4) Matrix by (1X4) matrix

Date : ...../...../.....

# eg: of broadcasting

1. $\begin{bmatrix}1\\2\\3\\4\end{bmatrix} + 100$

When we add 1×4 Matrix to a number, python will take this no & auto expand to 4×1 matrix.

So it becomes

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

Here broadcasting occurs

2. $\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix} + \begin{bmatrix}100 & 200 & 300\end{bmatrix}$

2×3 Matrix (m×n Matrix)

1×n matrix (general)

Python copies the matrix on right m times to ~~top~~ make it (m×n) matrix. This is also broadcasting.

$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix} + \begin{bmatrix}100 & 200 & 300\\100 & 200 & 300\end{bmatrix}$$

$$= \begin{bmatrix}101 & 202 & 303\\104 & 205 & 306\end{bmatrix}$$

UPCP

Date : ...../...../.....

3. $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 100 \\ 200 \end{bmatrix}$

$m \times n$ matrix

$m \times 1$ matrix

Here copy $m \times 1$ matrix $n$ times horizontally.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 100 & 100 \\ 200 & 200 \end{bmatrix}$$

$$= \begin{bmatrix} \end{bmatrix}$$

## General principle of broadcasting in python

If you have (m, n) matrix and you add or subtract or multiply or divide ~~them~~ with a (1, n) matrix then it will copy m times into (m, n) matrix & then apply oper$^n$ element wise.

If conversely you took (m, n) ——— with (m, 1) ——— n ——— (m, n) ———

$(m, 1) + R$

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

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

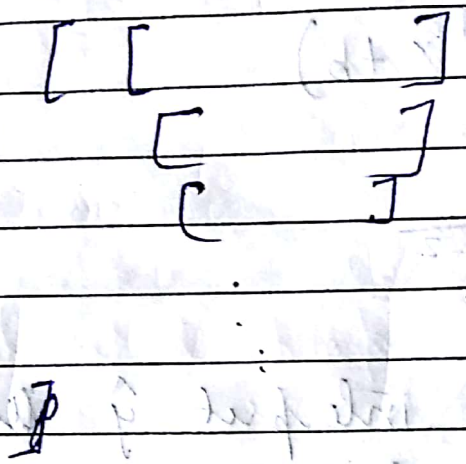
// Creating 5 random variables in python

```
import numpy as np
a = np.random.randn(5)
print(a)
```

[ 0.502951 . . . ]

```
print(a.shape)
(5,)   // Rank 1 array & is neither row nor column vector
print(a.T)   // Here ~~trans~~ no change occurs on doing its transpose
print(np.dot(a, a.T))
   // gives a no.
```

Now transpose gives correct result
In doing this, there will be 2 brackets



However in previous case 1 bracket exists.

## Python / numpy vectors

→ a = np.random.randn(5)
a.shape (5, )
// rank 1 array. It does not give consistent result. Neither row / column vector.

Rather use this :)

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

→ assert (a.shape == (5,1))

→ a.reshape ((5,1)) → if it is a rank / ~~matrix~~ array.
This can eliminate it.

Date : ..../...../.....

# There is a cost fun$^n$ in logistic regression. Justification of using that cost function

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

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

We want to interpret $\hat{y}$ as the prob that $y=1$ given $x$.

So we want our algo to output $\hat{y}$ as the chance that $y=1$ given $x$.
($x$ : set of I/P features)

$\Rightarrow$ If $y=1$, then $p(y|x) = \hat{y}$

$y=0$ ——————— $= 1-\hat{y}$

$y=0$ or $1$ is only possible case as it is binary classification.

On Summarizing.

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

In case 1:

$y=1$.

$$p(y|x) = \hat{y}\,(1-\hat{y})^0$$
$$= \hat{y}.$$

UPCP

In case 2:

$y=0$.

Date : ..../...../.....

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

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

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

Cost on m eqs:

$$p(\text{labels in training set}) = \prod_{i=1}^{m} p(y^{(i)} | x^{(i)})$$

log —

$$(a*b) + (a*c) - (b+c)$$

$$(a*b) + (a*c) - (b+c)$$

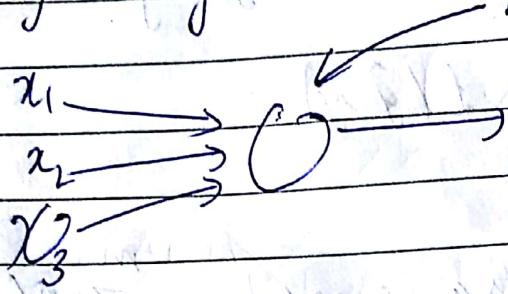
1) np.exp(x) → Works for any np.array x & applies exponential fun$^{n}$ to every coordinate

Overview of Neural Network

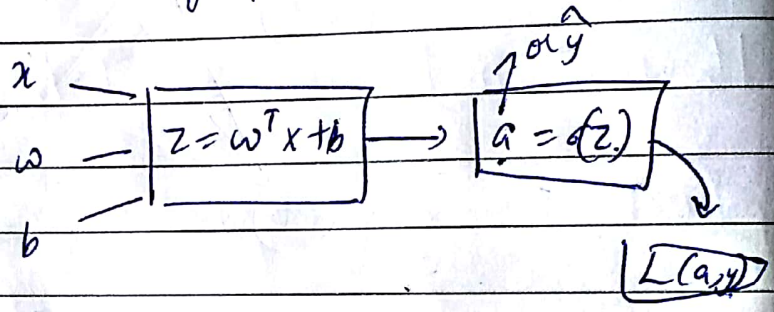
Date : ..../..../....

## What is Neural Network ?

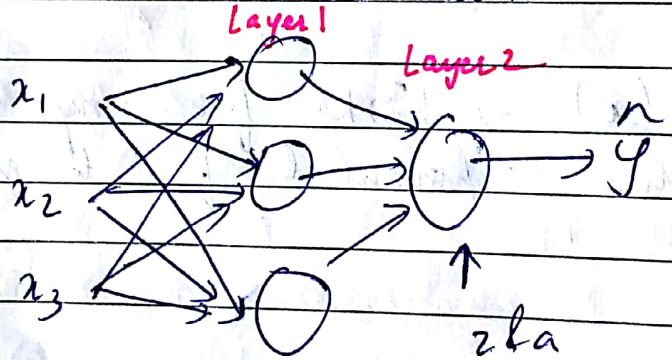
logistic regression




This models corresponds to full computational graph.




Neural Network looks this ;




First we I/P x i.e features and parameters w and b. It helps to

UPCP

$X^{(i)}$ → round brackets for refering to training eg no.

$X^{[i]}$ → refers to layer number. i.e "refering to quantities associated with layer [i]

Date: ...../...../.....

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

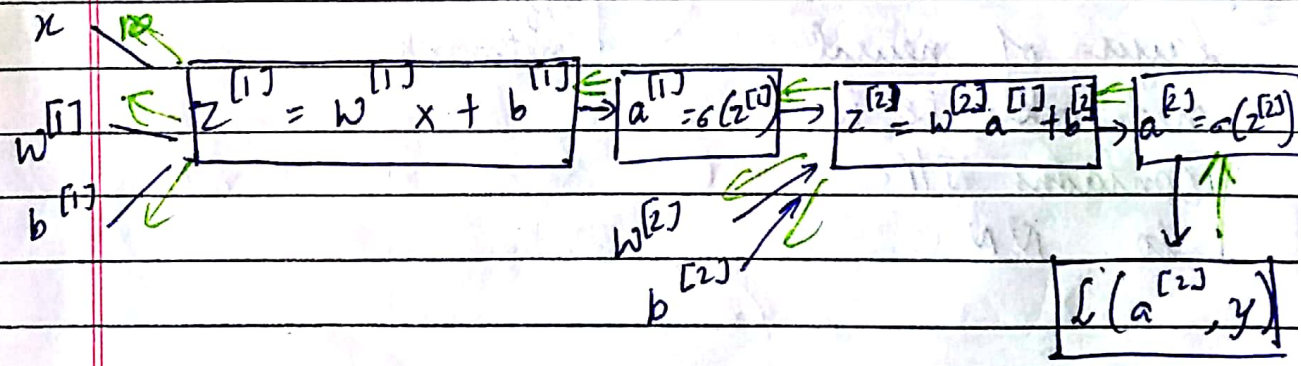
Superscript and square brackets are used to associate with quantities refering to layer 1

[2] → To refer to quantities associated with layer 2.

After computing $z^{[1]}$, Similar to logistic regression, we compute $a^{[1]}$.

Then we compute $z^{[2]}$ using another linear eqⁿ & then compute $a^{[2]}$, $a^{[2]}$ is the final o/p of NN.

Given - backward propⁿ




→ As in logistic regression, we did backward propagation & computed derivatives, gradients. Similarly neural network will end up doing backward calculation.

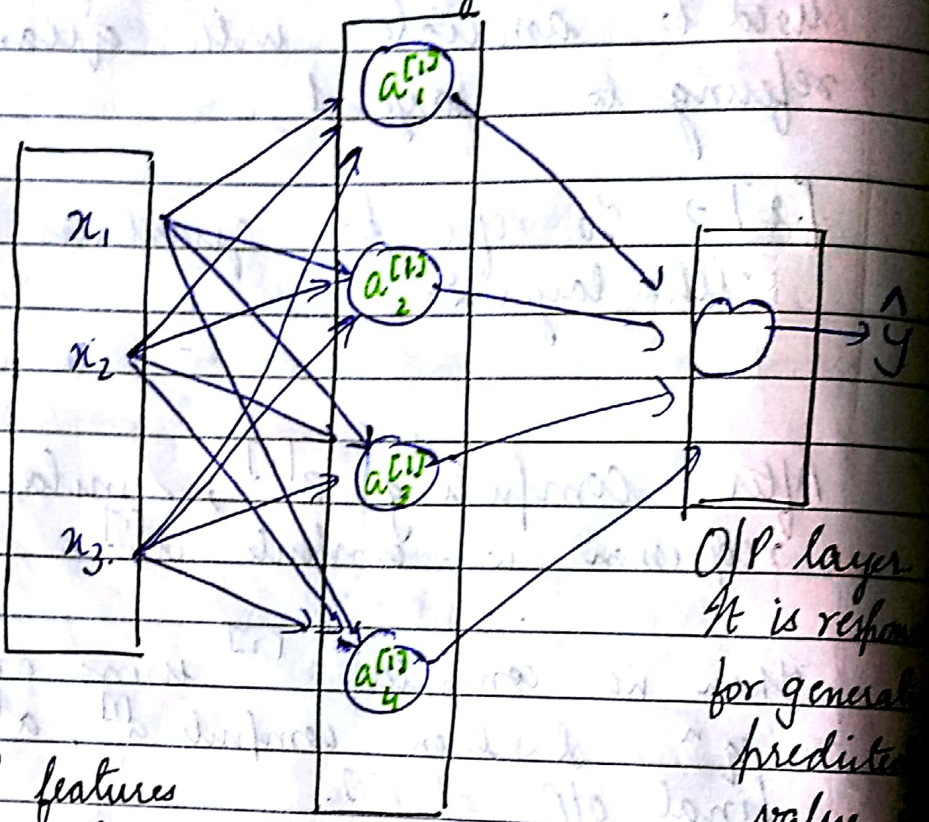
Backward propⁿ occurs from right to left & hence $dw^{[2]}$ $db^{[2]}$ $dw^{[1]}$ $da^{[1]}$

One layer hidden

Date : ..../..../.....

# Neural Network Representation

One hidden layer neural network




I/P features stacked vertically. This is called I/P layer of neural network. i.e it contains I/Ps to NN

Hidden layer of neural network.

O/P layer. It is respo... for genera... predict... value

In supervised learning, the training contains values of x and y i.e I/P and the target o/p y.

The things in the hidden layer are not seen in the training set. Hidden bcz we don't see in training

UPCP

Date : ..../..../.....

$X \rightarrow$ Input feature vector $\rightarrow a^{[0]}$

↑ alternative representation.

a also stands for activations and it refers to the values that different layers of Neural N/w pass on to the subsequent layers.

So I/P layer passes on values X to the hidden layer. Hence we call it activations of I/P layer.

The hidden layer will generate activations & we represent that using ~~a[1]~~ $a^{[1]}$.
Hence the nodes of hidden layer will generate $a_1^{[1]}$, $a_2^{[1]}$, $a_3^{[1]}$, $a_4^{[1]}$ respectively

$$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix} \rightarrow 4 \times 1 \text{ Matrix or 4 dime vector}$$

O/P layer generates value $a^{[2]}$ which is just a real no.
so $\hat{y} = a^{[2]}$

This N/w is called 2 layer Neural N/w bcz when we count layers in NW, we don't count I/P layer. So, hidden layer is layer 1 & O/P layer is layer 2.

Date : ..../..../.....

Hidden layer & O/P layer have parameters associated with it parameters w and b.

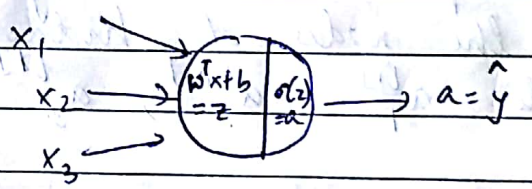
$w^{[1]}$ , $b^{[1]}$ — Parameters of Hidden layer

$w$: $4 \times 3$ , $b$: $4 \times 1$

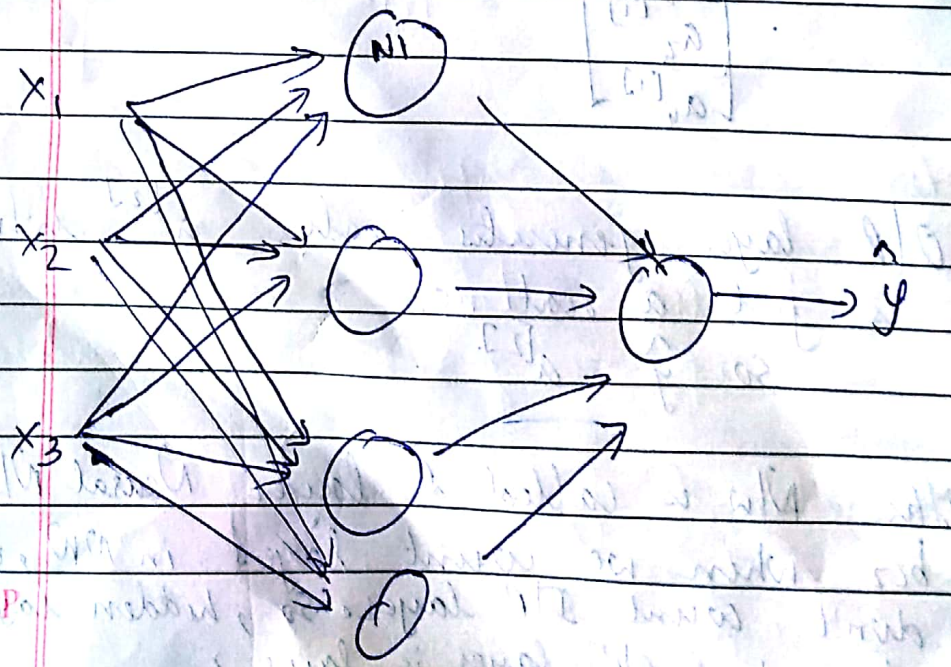
$w^{[2]}$ , $b^{[2]}$ — Parameters of O/P layer

$w$: $(1,4)$ , $b$: $(1,1)$

* In logistic regression, circle represents 2 steps of computation.
  - → Computing z
  - → " activation



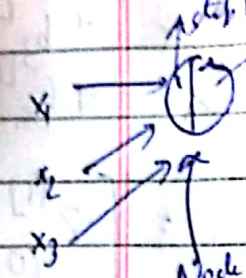

* A Neural Net does this many times



→ Each node of hidden layer does two steps of comp$^n$.

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

2nd step = $a_1^{[1]} = \sigma(z_1^{[1]})$

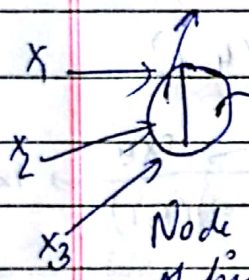



$a_i^{[l]}$ → layer

→ node no in layer.

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

$a_2^{[1]} = \sigma(z_2^{[1]})$.




So

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

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

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

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

If we do this using for loops, it becomes inefficient.

So we vectorize it.

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

$$+ \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix}$$

Given I/P $x \rightarrow (4,3)$ $\quad (3,1)$

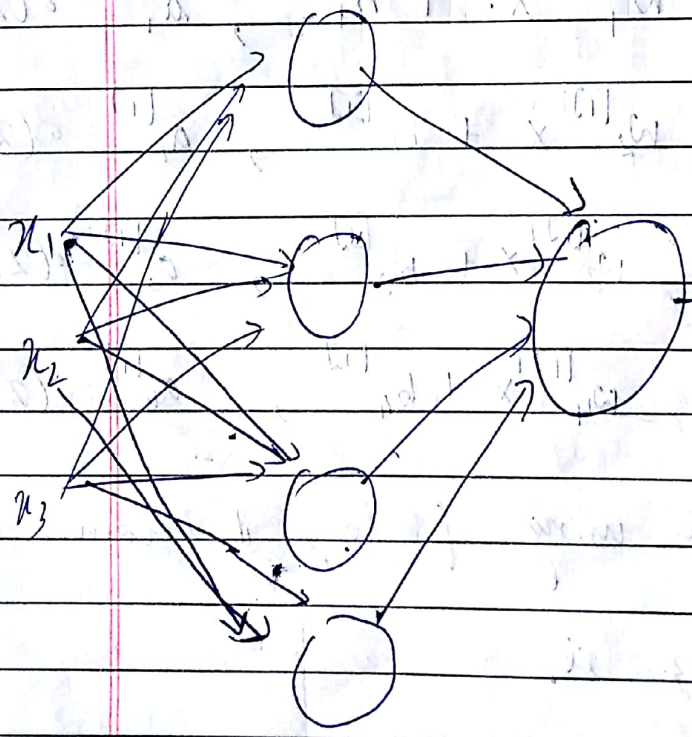
$$\Rightarrow \underset{(4,1)}{z^{[1]}} = W^{[1]} x + b^{[1]} \rightarrow (4,1)$$

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

$$z^{[2]}_{(1,1)} = W^{[2]} a^{[2]} + b^{[2]} \quad (1\times4)\ (4,1) \rightarrow (1\times1)$$

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

$x$ can be represented by $a^{[0]}$




UPCP

Date : ..../..../.....

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

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

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

⋮

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

$a^{[2](i)}$

layer 2, training eg i

→ In case of non vectorized implementation, if we have m training egs then we need to do :

for i = 1 to m :

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

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

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

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

→ Vectorizing to get rid of for loop

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

Training eg stacked in column

Size of matrix X is $n_x \times m$

UPCP

Date : ..../...../.....

Compute $z^{[1]}$

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

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

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

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

So we changed from lower case vector $x$ to capital $X$ matrix by stacking lower case $x$ in diff columns.

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

Corresponds to 1st hidden unit of this training eg.

2nd

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

the horizontal ~~ya~~, or we index for diff training egs.

This vertical index corresponds to different nodes in the neural n/w.

# Justific$^n$ for vectorized Implemen$^n$

for training eg 1 2 3

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

for eg 3 $z^{[1](3)} = w^{[1]}x^{(3)} + b^{[1]}$

Assume $b = 0$.

$w^{[1]} = \begin{bmatrix} — \\ — \\ — \end{bmatrix}$ $\quad w^{[1]}x^{(1)} = \begin{bmatrix} \cdot \\ \cdot \\ \cdot \\ \cdot \end{bmatrix}$ Column vector.

$w^{[1]}x^{(2)} = \begin{bmatrix} \cdot \\ \cdot \\ \cdot \\ \cdot \end{bmatrix}$ Some other column vector

$w^{[1]}x^{(3)} = \begin{bmatrix} \cdot \\ \cdot \\ \cdot \\ \cdot \end{bmatrix}$ Other column vector.

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

All training egs are stacked vertically.

$w^{[1]}x^{(1)}$ $w^{[1]}x^{(2)}$ $w^{[1]}x^{(3)}$

$w^{[1]}X = \begin{bmatrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix}$

CP

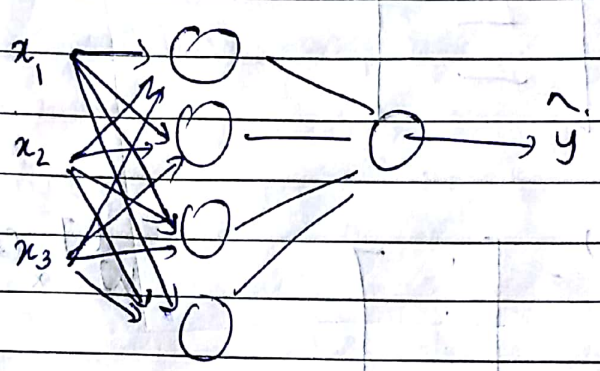
Date : ..../..../.....

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

## Activation Functions

What activation fun$^n$ to use in the neural network.

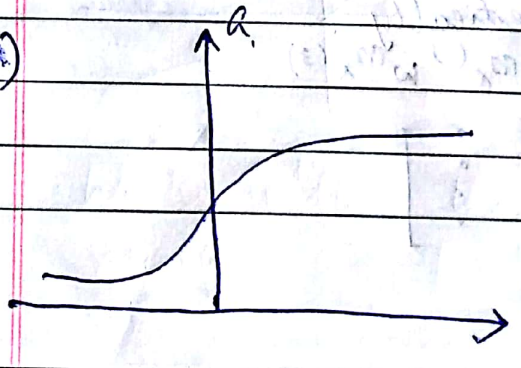
In forward prop$^n$ step of NN, the steps are.




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

In these two steps, we use the sigmoid functions.

1)




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

Date : ..../...../.....

In generic case we can use diff activation fun$^n$.

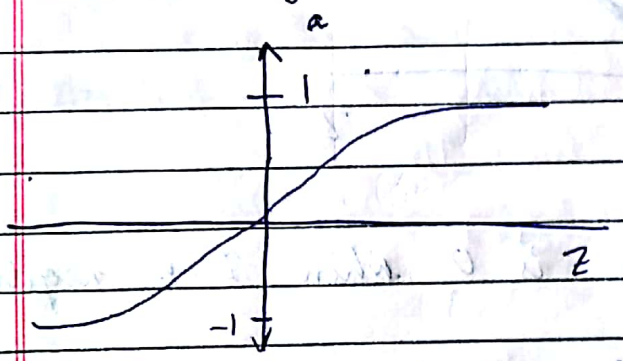
$$a^{[1]} = g(z^{[1]})$$

↓ generic.

where g is any non-linear fun$^n$ but not sigmoid function.

Sigmoid ranges b/w 0 & 1.

2. Tan h i.e hyperbolic tangent
   → is an activa$^n$ fun$^n$. It is works better than sigmoid.
   → It goes b/w +1 & -1.




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

If we use tanh ~~for~~ as activation fun$^n$ in hidden layers than it works better than sigmoid fun$^n$.

→ In O/P, we prefer using sigmoid fun$^n$ because we want O/P b/w 0 & 1. If O/P is b/w -1 and 1, it doesn't make sense. So in case of binary classifi$^n$ prob we use activ$^n$ fun$^n$ of sigmoid.

UPCP

Date : ..../..../.....

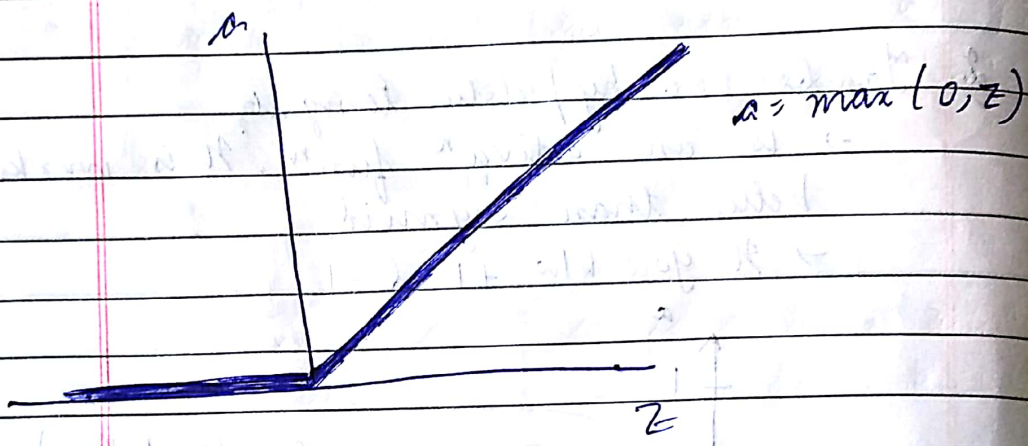
→ We can have diff activ$^{n}$ fun$^{ns}$ for diff layers.

$g^{[1]}(z^{[1]})$ → refers to activation fun$^{n}$ for layer 1.

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

↳ Refers to activ$^{n}$ fun$^{n}$ for layer 2.

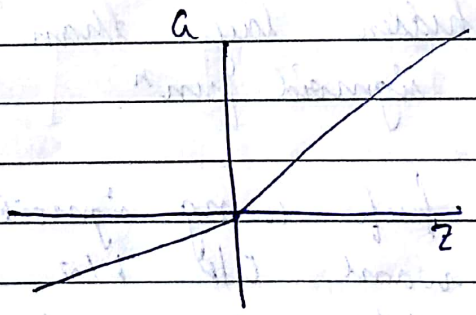
3.) RELU → Rectified linear unit.




disadv. →

→ derivative is 0 when z is negative

4.) Leaky RELU → Another version of RELU.



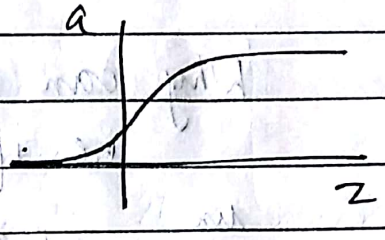

Works better than RELU but not used mostly.

UPCP

RELU and leaky RELU has advantage. For most values of z, the derivative or slope of actin fun$^n$ is very diff from 0. and so our Neural network will learn very faster than when we use tan h or sigmoid activa$^n$ fun$^n$.

## Pros and cons of different activation fun$^n$s

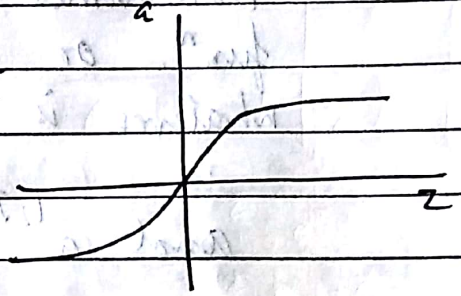
1. Sigmoid : $a = \frac{1}{1+e^{-z}}$




→ Never used. ~~except~~ we do

→ Use only in O/P layer when doing binary classification.

→ tan h : $a = \frac{e^z - e^{-z}}{e^z + e^{-z}}$

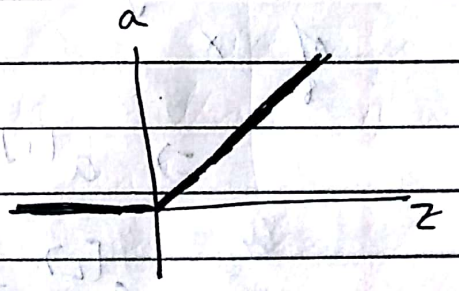



→ Better than sigmoid fun$^n$

→ RELU $a =$

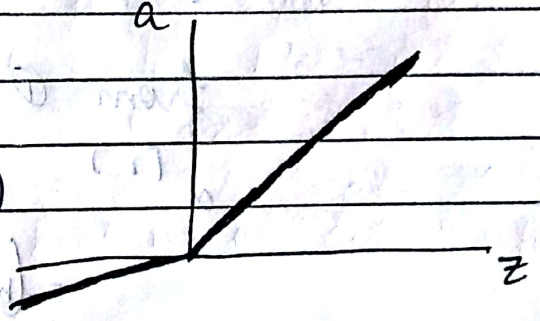
→ Mostly used by default.

$a = \max(0, x)$




→ Leaky RELU.

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




UPCP

# Why do we need non-linear activ$^n$ fun.$^n$?

→ For a Neural N/W, the forward prop$^n$ eq$^n$s are:

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

Why can we not get rid of $a^{[1]} = g^{[1]}(z^{[1]})$ i.e if we can remove activa$^n$ fun$^n$ & use

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

Sometimes also called as linear activ$^n$ fun$^n$. or identity activ$^n$ fun$^n$. bcz it o/ps whatever is the I/P

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

Here $\hat{y}$ or o/p is just a linear fun$^n$ of $x$

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

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

from (1) & (2)

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

UPCP

If we use linear activ$^n$ fun$^n$ then NN o/p a linear fun$^n$ of I/p.

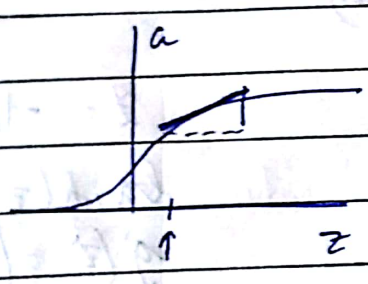
→ Linear activ$^n$ fun$^n$ is useless.

→ It is used only if we are doing M/c learning on a ~~linear~~ regression prob.

## Derivatives of Activation fun$^{ns}$.

For backpropagation steps, we need to compute derivatives / slope of activ$^{ns}$ fun$^n$

Sigmoid activ$^n$ fun$^n$




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

slope of fun$^n$

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

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

$$= g(z)\left(1 - g(z)\right)$$

$z = 10$.

$g(z) = 1$

$\frac{d}{d(z)} g(z) \approx 1(1-1)$

$\approx 0$.

$z = -10$

$g(z) = 0$.

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

$\approx 0$.

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

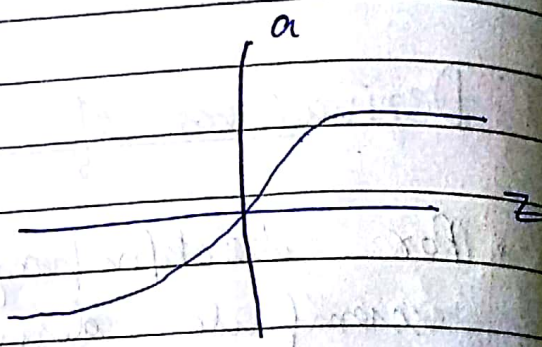
$$a = g(z) = \frac{1}{1+e^{-z}}$$

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

→ tanh.

$$g(z) = \tanh(z)$$

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




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

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

$z = 10$

$\tanh(z) \approx 1$

$g'(z) = 0$.

$z = -10$

$\tanh(z) \approx -1$

$g'(z) = (1 - (-1)^2)$

$= 0$.

$z = 0$.

$\tanh(z) \approx 0$

$g'(z) = 1$

$$a = g(z)$$

$$g'(z) = (-a)^2$$

Lecture - 2

# Wireless Commⁿ

Date : 22/01/18

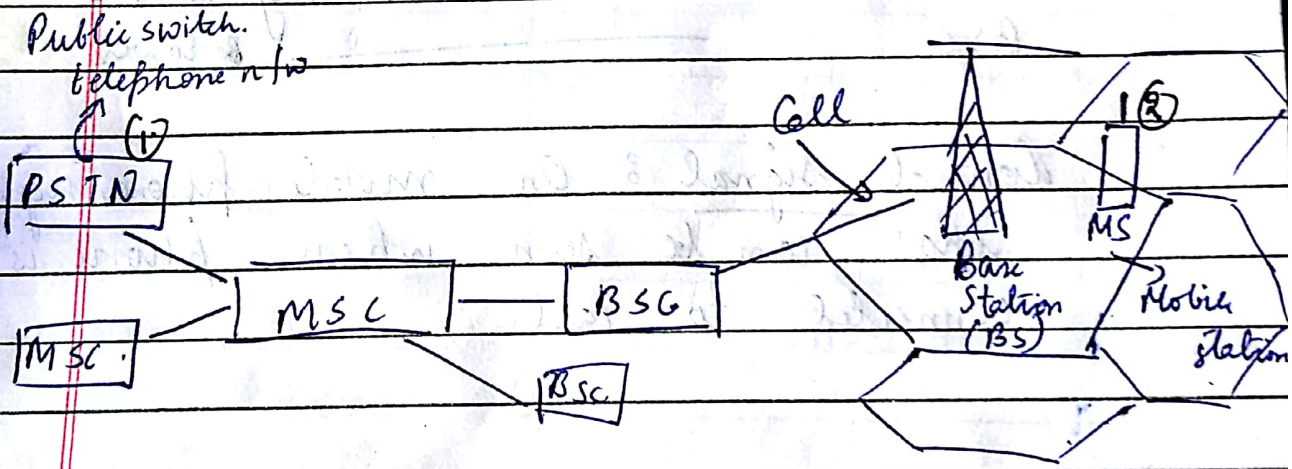
Cordless phone
Pagers
Mobile phone.

## Few terminologies

→ MSC → Mobile switching centre

→ BSC → Base station control.

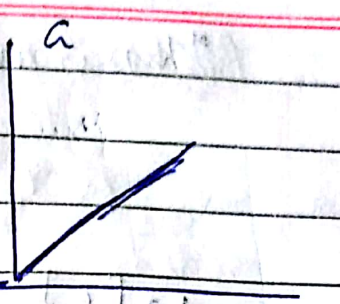
Ericson company ke devices Mohali BSNL exchang में लगे हैं।


Mobile → Mobile Station
Tower → Base station
Area jisme tower signals deta hai → Cell
Subscriber → jiska ph hai

Date : ..../...../.... ..

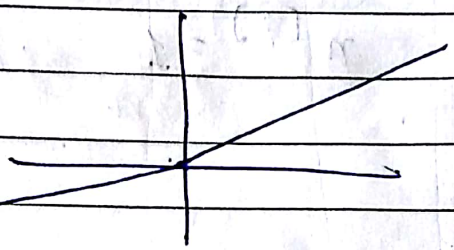
→ RELU:




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

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

→ Leaky ReLU



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

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

## Gradient descend for neural networks

How to compute grad descent for NN wi
1 hidden layer.

NN with 1 hidden layer:

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

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

↓ no of Input features

↓ no of hidden units

↓ no of o/p units.

Date : ...../...../.....

Then matrix $w^{[1]} = n^{[1]} \times n^{[0]}$

$b^{[1]} = n^{[1]} \times 1$

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

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

$n^{[2]} = 1$ · All the examples seen till now.

for NN: Cost fun$^n$: $J(w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]})$

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

In case of binary class$^n$, loss fun$^n$ is same that was in logistic regression,

$L \rightarrow$ is loss when NN predicts $\hat{y}$.

To train the parameters of our algo, we need to perform gradient descent.

→ For training parameters of NN, it is important to initialize the parameters randomly rather than to all 0's.

→ Then each loop of grad descent

UPCP

Date : ...../...../.....

would compute predictions $\hat{y}$.

→ Then find derivatives of cost fun$^n$ wrt variable $dw^{[1]}$, $b^{[1]}$ .... & so on.

→ Finally gradient descent update.

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

→ learning rate

Gradient descent :

→ Repeat {

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

$$dw^{[1]} = \frac{\partial J}{\partial w^{[1]}}, \quad db^{[1]} = \frac{\partial J}{\partial b^{[1]}} \ldots$$

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

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

:

Formulas for computing derivatives.

→ Forward prop$^n$.

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

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

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

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

↓ for binary class$^n$.

Date : ..../..../.....

→ Backward propagation:

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

$$y = [y^{(1)} \; . \; . \; . \; y^{(m)}]$$

↓ all eqs stacked horizontally

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

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

↓ python numpy command for summing across 1-D. in this case summing horizontally.

Keepdims → prevents python from producing Rank 1 matrix & so produces vector.

Rank 1 → (5,)

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

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

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

g derivative of activ$^n$ fun$^n$ for hidden layer.

element wise product

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

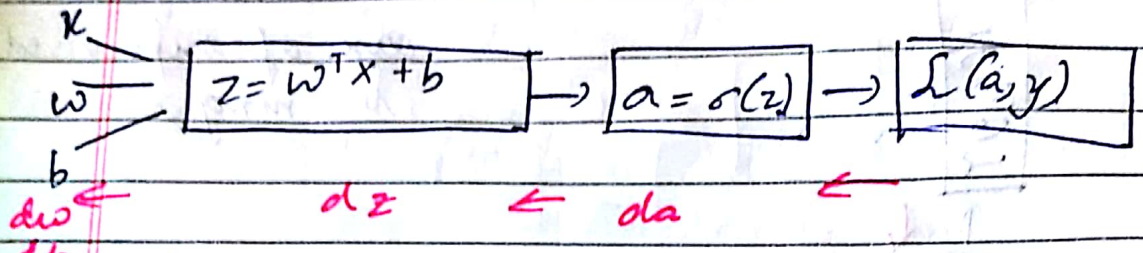
$$db^{[1]} = \frac{1}{m} np.sum(dz^{[1]}, axis=1, keepdims=True)$$

↳ $(n^{[1]}, 1)$

UPCP

# Backpropagation Intuition

In logistic regression,
The forward prob$^n$ is to finding $\hat{y}$ i.e prediction




Backward prop$^n$
finding derivatives.

$$\mathcal{L}(a,y) = \text{loss} = -(y \log a + (1-y) \log(1-a))$$

$$da = \frac{d}{da} \mathcal{L}(a,y) = -\frac{y}{a} + \frac{(1-y)}{1-a}$$

$$dz = da * g'(z) = \frac{d}{da} \mathcal{L}(a,y)$$

$$= \frac{d\mathcal{L}(a,y)}{dz} \times \frac{dz}{dz}$$

$$= da * g'(z)$$

$$g(z) = \sigma(z)$$

$$dw = dz \cdot x$$

$$db = dz$$

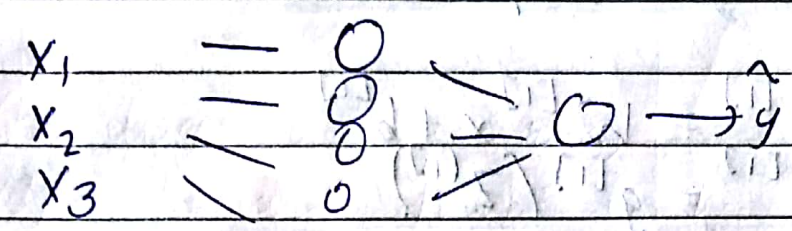
Date : / /

2. In NN we do 2 steps b/c 2 layers are there.

Steps are:

⇒ No need for taking derivative of x in case of supervised learning.

Check dimensions




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

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

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

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

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

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

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

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

## Summary of Gradient descent.

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

$dW^{[2]} = dz^{[2]} a^{[1]T}$

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

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

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

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

# Vectorized Implementation.

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

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

Stacking all eqs ~~that were these~~ horizontally-i

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

Summary

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

$$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T} \quad \text{bcz cost } J(\ldots) = \frac{1}{m} \sum \ldots$$

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

$$dZ^{[1]} = W^{[2]T} dZ^{[2]} * g^{[1]'}(Z^{[1]})$$
$$(n^{[1]}, m) \quad (n^{[1]}, m) \quad (n^{[1]}, m)$$

element wise product

Date : ..../...../.... ..

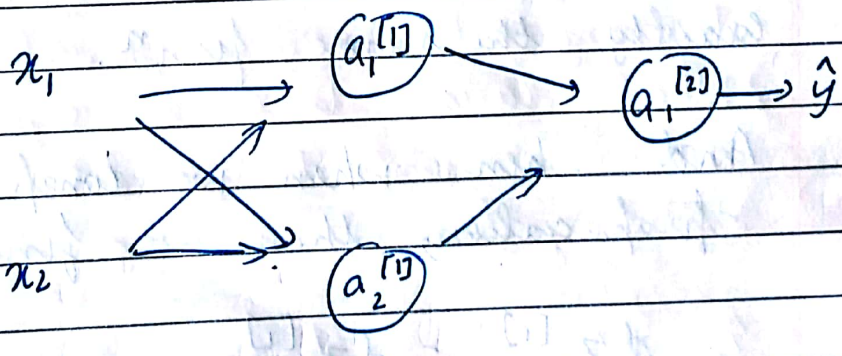
$$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$$

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

→ It was ok to initialize weights to 0 for logistic regression.

But for a NN if we initialize the weights or parameters to all zero and then apply gradient descent then it won't work.

What happens if you initialize the weights to zero?




So if we have 2 I/P features, i.e $n^{[0]} = 2$ and 2 hidden units so $n^{[1]} = 2$.

And so matrix associated with hidden layer, $W^{[1]} = 2 \times 2$

Date : ..../...../.....

Let's say you initialize it to all 0's, so 0 0 0 0, 2×2 matrix.

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

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

Initializing bias to 0 is ok but initializing weights to 0 is just not ok. It is a prob.

Problem with this type of initializ$^n$ is that for any eg you give you have:

$a_1^{[1]} = a_2^{[1]}$. So this activation units will be same bcz both of these hidden units are computing exactly the same fun$^n$.

And then when we compute back-propagation, then we find that

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

$$W^{[2]} = \begin{bmatrix} 0 & 0 \end{bmatrix}$$

So if we initialize the parameters with 0 then for each layer,

UPCP

Date : ..../..../....

all units becomes identical.

dw =

i.e After each iteration of training the 2 hidden units are computing exactly the same function.

So all hidden units become identical & no matter how long you do the training they remains identical.

Solution

Initialize your parameters randomly.

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

↓ generates gaussian random variable

↓ Multiply by very small no so as to initialize with very small values.

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

↓ b does not face symmetry prob. So it is okay to initialize it with zeros.

Why multiplied with 0.01 & not 100 bcz we have to initialize with very small random values.

UPCP

Bcz if we are using a tanh or sigmoid activation function, or just at o/p layer.

If weights are too large, then activ$^n$ fun$^n$, $z^{[1]} = w^{[1]}x + b^{[1]}$

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

so if w ≈ very big

z

~~then a[1] saturated~~ to

so in that case we are more likely to end up in the flat part of tanh or sigmoid function. Where slope is small & gradient descent = slow & learning = slow.

So

if w large then we are likely to end up even at the start of training with very large values of z. Which causes tanh or sigmoid activation fun$^n$ to be saturated, thus slowing down learning.

If we don't have any sigmoid or tanh activ$^n$ functions throughout our neural network. ~~This~~ But if we are doing binary class$^n$ & o/p is a sigmoid fun$^n$, then you don't want initial parameters to be too large

UPCP

So multiplying by 0.01 would be reasonable to try.

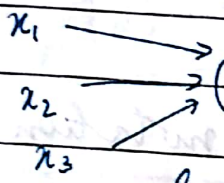
And same for w 2.

→ When we have small NN then 0.01 will work ok.

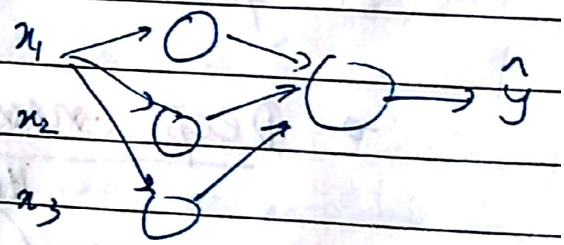
→ But when training a very deep NN, then you might need to pick up a diff constant than 0.01.

## Week 4

### What is a deep Neural Network

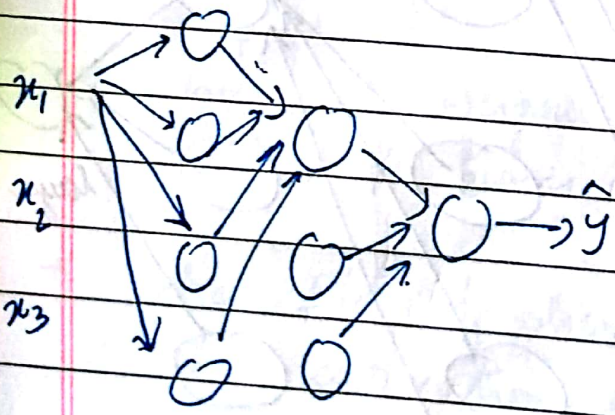
→ 


logistic regression
1 layer NN

→ 


NN 1 hidden layer
so 2 layer NN.

→ NN with 2 hidden layers




3 layer NN