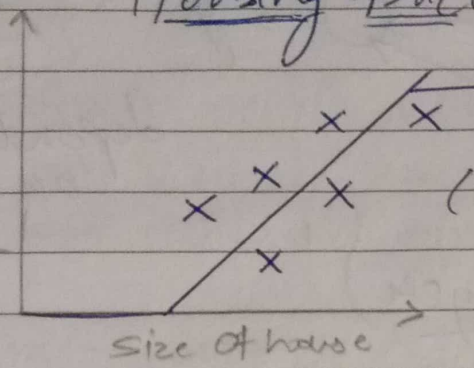
Neural Networks and Deep learning

(Deep Learning · AI Specialization - Coursera)

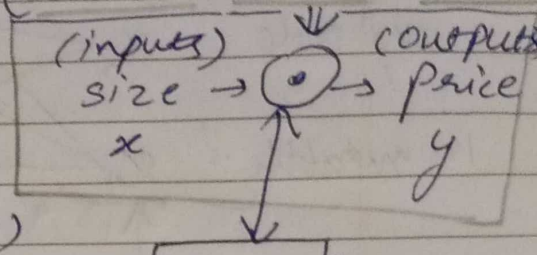
What is neural network ?

⇒ Deep learning refers to train Neural Networks

Housing Price Prediction (little neural network)



This is a function (i.e smallest neural network)



"single Neuron" in neural network

(as price wont be -ve) (we continue the line to 0)

Which implements the func

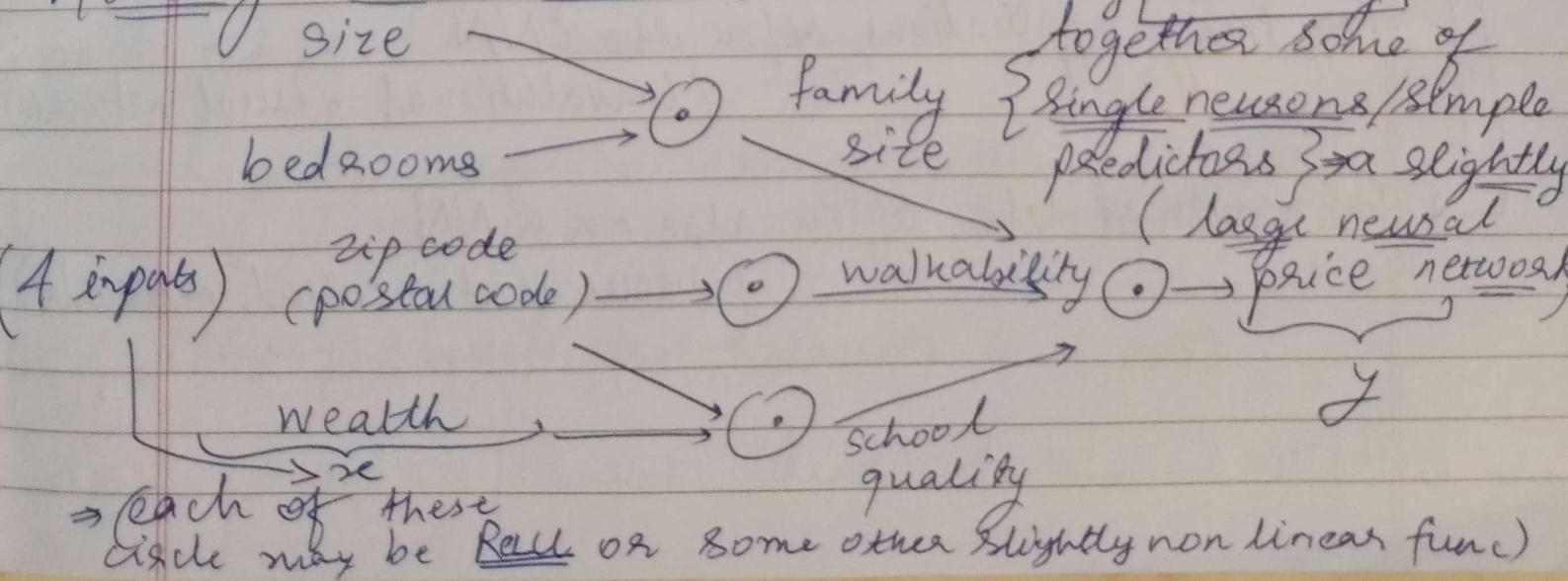
Neuron → inputs the size, computes the linear funct, outputs estimated price (max 0)

⇒ function like this tending to 0 & then following straight line is called (ReLU)

{ "ReLU" → Rectified linear Unit } Rectify means taking max of 0 due to which such shape of func is obtained.

In bigger Neural network many such neurons are stacked together.

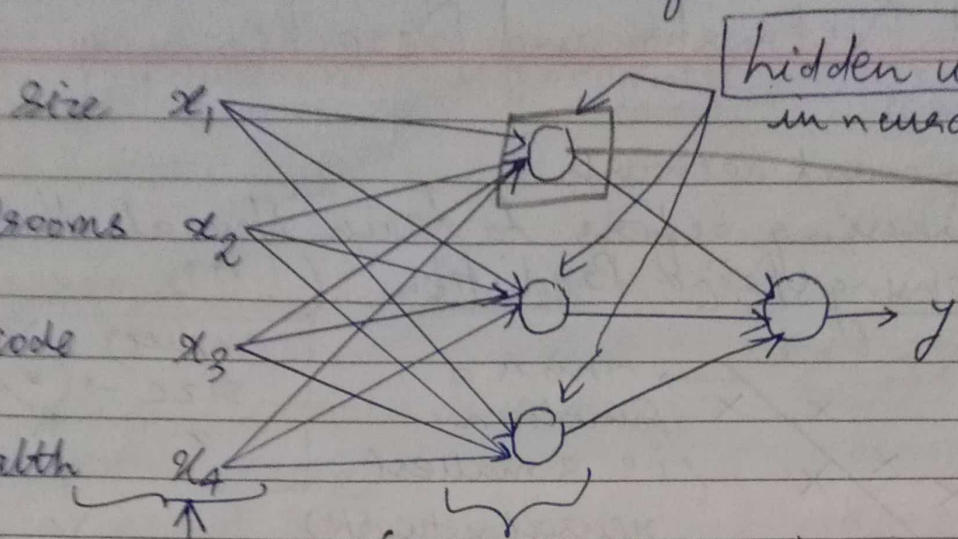
Housing Price Prediction -



so by [stacking] together some of {single neurons/simple predictors} → a slightly (large neural network)

⇒ Each of these circle may be ReLU or some other slightly non linear func)

things occuring in middle will be figured by the neural networks itself



Every is interconnected with

Supervised learning with Neural Networks -

Eg

Input (x)	Output (y)	Application
Audio	Text transcript	Speech recognition
English	Chinese	Machine translation
Ad, user info	click on ad? (0/1)	Online Advertising

For image applications, often use CNN (Convolutional neural network)

For sequenced data, often use an RNN (recurrent neural network)

Structured Data | Unstructured Data

⇒ Databases of data (difficult) ⇒ audio, images, text

* computers struggle to sense unstructured data compared to structured data *

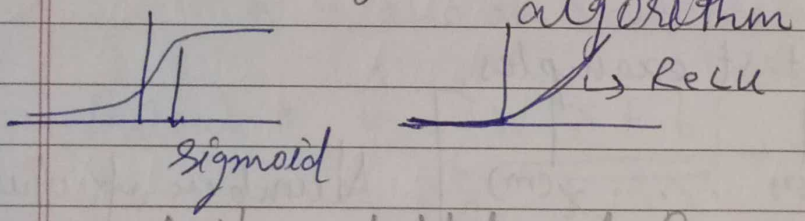
Why is Deep learning taking off?

⇒

Scale is driving DL
- ↳ size of neural network
- ↳ amount of data (labelled data)

{ m → size of training sets (number of training examples) }

Sigmoid function to ReLU func
→ gradient descent works fast this algorithm



Basics of Neural Network Programming

framework for NN

Build deep neural network (NN)

forward pass / forward propagation step

backward pass / backward propagation step

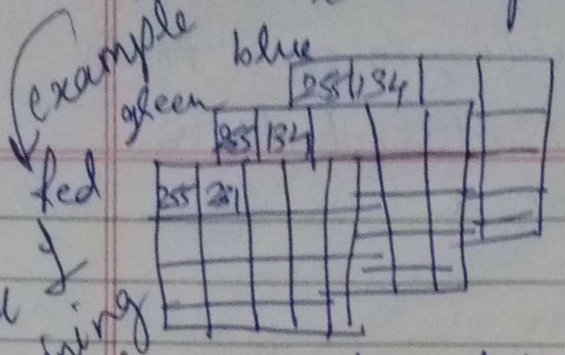
→ Logistic regression is an algorithm for binary classification

Binary classification → (classify in 2 grps)

[input image] ⟶ 1 (cat) vs 0 (non-cat)

output

$n = n_x$: represents dimension of input feature vector



64 × 64 pixel value for each matrix
we will transform 3 into feature vector whose dimension will then be $n_x = 12288$

Corresponding label is (x) $x = \begin{bmatrix} 255 \\ 231 \\ \vdots \\ 255 \\ 134 \\ \vdots \\ 255 \\ 134 \end{bmatrix}$

y 1 or 0

determine

Notation:

(x, y) $x \in \mathbb{R}^{n_x}$

n_x dimensional feature

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

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

(M)

$M = M_{train}$ ⇒ number of train examples

M_{test} ⇒ number of test examples

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

No. of rows

Number of columns matrix X will have 'm' where M is no. of training examples

column

↳ Training set inputs

n_x ← m →

Height of this matrix is n_x

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

dimensional matrix

X.shape ⇒ python command for (finding Shape) of "matrix" that this is a n_x, m

output labels y → its easy to stack in column

$Y = [\, y^{(1)} \; y^{(2)} \; \ldots \; y^{(m)} \,] \rightarrow$ no. of rows = 1

$Y \in \mathbb{R}^{1 \times m}$ $\quad$ Y.shape = (1, m)

↳ dimensional matrix

Logistic Regression

given x $\quad$ o/p a prediction $\hat{y}$ ⇒ estimate of y (y hat)

$x \in \mathbb{R}^{n_x}$ $\quad$ $\hat{y} = P(y=1 \mid x)$

x dimensional vector $\quad$ ↳ probability of chance that y is equal to 1 given input features x

$\hat{y}$ tells what is chance this is a cat pic

Parameters of logistic Regression:-

$W \in \mathbb{R}^{n_x}$ $\quad$ $b \in \mathbb{R}$

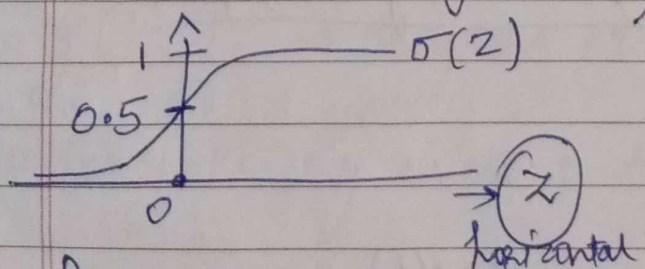
W is also an X dimensional vector

Output : $\hat{y} = \{ W^T x + b \}$ → linear function of X

↳ $0 \leq \hat{y} \leq 1$

This quantity can even be greater than one or even negative so that's not correct.

Therefore: $\hat{y} = \sigma(W^T x + b)$ $\quad$ sigmoid func



where here sigmoid(z) ⇒ σ(z)

$Z = W^T X + b$

formula for sigmoid fun

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

where z is $\mathbb{R}$

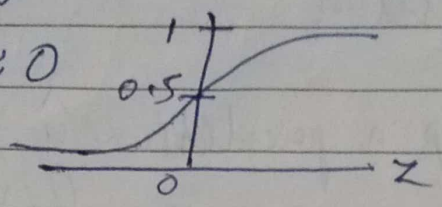
If z is large; $1+e^{-z} \approx 1$

$\therefore \sigma(z) \approx 1$

If z is very small / very large -ve number.

that relates when z is very large -ve then $\sigma(z) \approx 0$

so this is $\sigma(z) = \frac{1}{1+e^{-z}} = \frac{1}{1+\text{Big num}} \approx 0$



keep the parameters W and b separate

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

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

$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_{n_x} \end{bmatrix}$ $\theta_0 \leftarrow b$, $\theta_1, \theta_2, \ldots, \theta_{n_x} \leftarrow W$

But donot follow this approach

---

to train, W & b of logistic regression parameters

need define a cost function

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

$\Rightarrow$ given $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$, want $\hat{y}^{(i)} \approx y^{(i)}$

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

$\hat{y}^{(i)} = \frac{1}{1+e^{-z^{(i)}}}$

$x^{(i)}, y^{(i)}, z^{(i)}$ } i^{th} example

Loss(error) function → is restricted to single training example (applied)

measure how well our algorithm is doing.

$L(\hat{y}, y)$ may be Square error $(\hat{y}-y)^2$

(loss) fun → true label or $\frac{1}{2}$ of square error $\frac{1}{2}(\hat{y}-y)^2$

→ algorithm outputs

→ measure how good output $\hat{y}$ is when true label is y

required to define to (need)

But loss function used in logistic regression is as follows:

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

Why this is preferred?

If $y=1$ $L(\hat{y}, y) = \boxed{-\log \hat{y}}$ ← we want loss func as small as possible

$\log \hat{y}$ → as large as possible

$\hat{y}$ → want large $\hat{y} = \sigma(z)$ it can never be bigger than one $\hat{y} \not> 1$

So for $y=1$ $\hat{y}$ will be close to 1

If $y=0$ $L(\hat{y}, y) = \boxed{-\log(1-\hat{y})}$ ← as small as possible

$\log(1-\hat{y})$ as large as possible ∴ want $\hat{y}$ as small ∵ $\hat{y}$ lies between 0 & 1

So for $y=0$ $\hat{y}$ as close to zero as possible

In general $y=1$ $\hat{y}$ is as large as possible

$y=0$ $\hat{y}$ is as small as poss

true labels

Cost function -

(J)
of parameter W and b

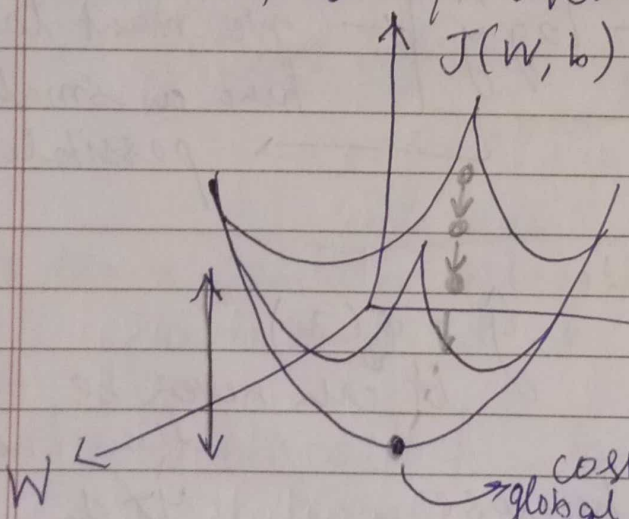
$$J(W,b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log\hat{y}^{(i)} + (1-y^{(i)})\log(1-\hat{y}^{(i)})\right]$$

Gradient Descent -

↳ algorithm to train parameter W on training set

Cost function measures how well are the parameters W and b are doing on training set

want to find W, b that minimize $J(W,b)$



height of surface represents value of J, b at certain pt

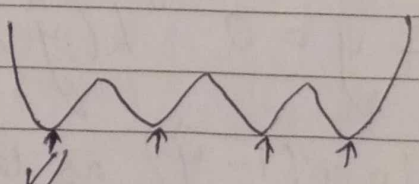
b corresponds to minimum of cost fun (J)

cost fun J is convex fun

global optimum

initialize W and b by some initial value

for logistic regression any initialization works

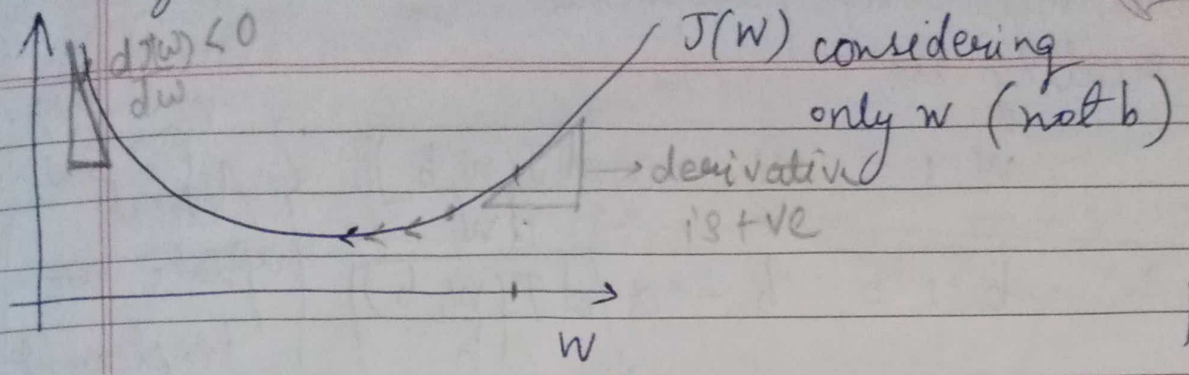


non-convex lots of different local optima

gradient descent starts at that initial pt and takes step in the steepest downhill direction.

(this is 1 iteration of gradient descent)

This iterations continue until it converges to global optimum or close to global optimum.



Gradient descent algorithm repeats and this gets repeated until algorithm converges.

Repeat {

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

}

learning rate

derivative

learning rate : controls how big a step we take on each iteration in gradient descent algorithm.

: update of the change want to make to parameters w

to represent derivative term in code ⇒ 'dw'

$w := w - \alpha\, dw$

gradient descent update

∴ derivative is +ve

$w := w - \alpha\, dw$

Subtracting from w

so w will move towards left

⇒ gradient descent algorithm slowly decrease parameter

-ve slope/derivative

so thereby w will increase and move towards right

So basically from wherever u start you will try to move towards centre (global minimum)

In previous examples considering cost function $J(w)$ where w is only parameter

But in reality its $J(w,b)$

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

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

this notation basically $\frac{\partial J(w,b)}{\partial w}$ (partial derivative)

Actual update we implement

'dw' (update w)

If J is a func of 1 variable we use 'd'

Since J is a func of 2 parameters

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

'db' (update b)

This basically represents how much fun $J(w,b)$ slopes in b direction

for a straight line, slope of the line remains same (derivative of function is constant i.e same)

1. derivative of a func just means slope of fun
2. slope can be diff at diff pts on function.

Computation Graph - (computes output)

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

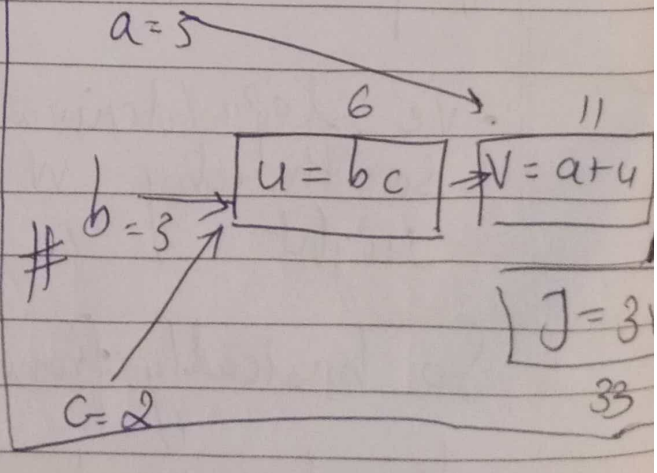
$u = bc$

$v = a + u$

$J = 3v$

for computing derivatives, we will use right to left pass

Computation graph :-



(left to right pass)

forward pass/forward propagation step to compute output

backward pass/backward propagation step to compute derivatives (Right to left pass) (gradients / derivatives of output)

$\frac{dJ}{dv} = ?$ $\quad J = 3v$

$v = 11 \rightarrow 11.001$

$= 3$ $\quad J = 33 \xrightarrow{\text{Final}} 33.003$

Derivatives with Computation Graph

if we want derivative of output variable (compute) we are doing one step of backpropagation → one step backwards in graph

$a = 5 \rightarrow 5.001$

$v = 11 \rightarrow 11.001$

$J = 33 \rightarrow 33.003$

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

(3·1) $\frac{dJ}{da}$

$a = 5$

$dv = 3$

11 $\quad$ 33

$b = 3$ $\quad$ 6 $\quad$ $u = bc$ $\quad$ $du = 3$ $\quad$ $v = a + u$ $\quad$ $\frac{dJ}{dv}$ $\quad$ $J = 3v$

$c = 2$

chain rule

while computing derivatives

$\frac{dv}{da} = 1$

dFinalOutputVar / dVar → Intermediate var

$\frac{dJ}{dv} = 3$ $\quad dv = 3$ ← representation format

$\frac{dJ}{da} = 3 \quad da = 3$

$\frac{dJ}{du} = 3 \quad du = 3$

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

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

$\frac{dJ}{dv} \cdot \frac{dv}{da} = 3 \cdot 1$

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

$\frac{dJ}{dvar}$ $\quad$ dvar

we represent this in the code as "dvar"

$dv = \left(\frac{dJ}{dv}\right)$

$u = 6 \rightarrow 6.001$

$v = 11 \rightarrow 11.001$

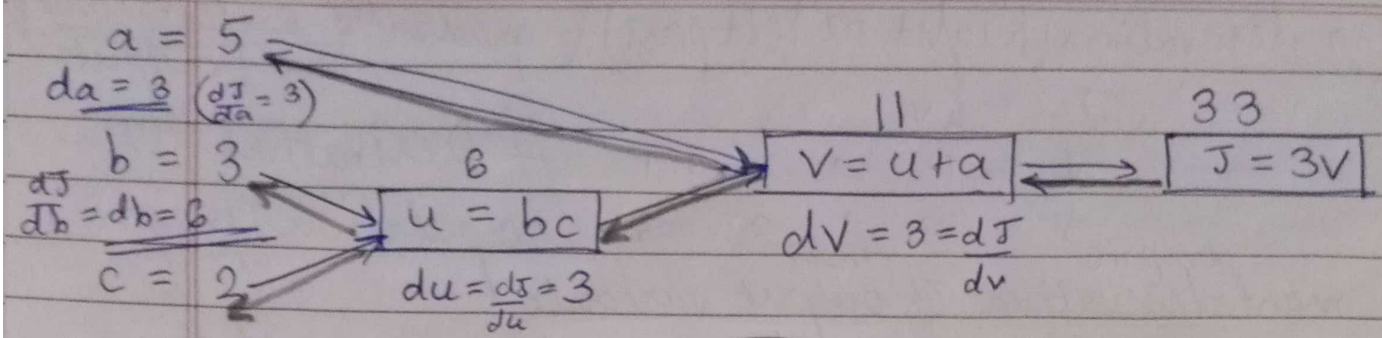
$J = 33 \rightarrow 33.003$

$\frac{dJ}{db} = \frac{dJ}{du} \cdot \frac{du}{db}$

$b = 3 \rightarrow 3.001$

$u = 6 \rightarrow 6.002$

dvar: derivative of final op variable wrt to intermediate variables.



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

$3 = \frac{dJ}{du} = \frac{dJ}{dV} \cdot \frac{dV}{du} = 3 \cdot 1$

$db = \frac{dJ}{db} = 6 = \frac{dJ}{du} \cdot \frac{du}{db} = 3 \cdot 2$

$u = 6 \rightarrow 6.001$
$V = 11 \rightarrow 11.001$
$J = 33 \rightarrow 33.003$

$b = 3 \rightarrow 3.001$
$u = 6 \rightarrow 6.002$
$V = 11 \rightarrow 11.002$
$J = 33 \rightarrow 33.006$

$c = 2 \rightarrow 2.001$
$u = 6 \rightarrow 6.003$
$V = 11 \rightarrow 11.003$
$J = 33 \rightarrow 33.009$

$dc = \frac{dJ}{dc} = 9 = \frac{dJ}{du} \cdot \frac{du}{dc} = 3 \cdot 3$

Logistic Regression Gradient Descent -

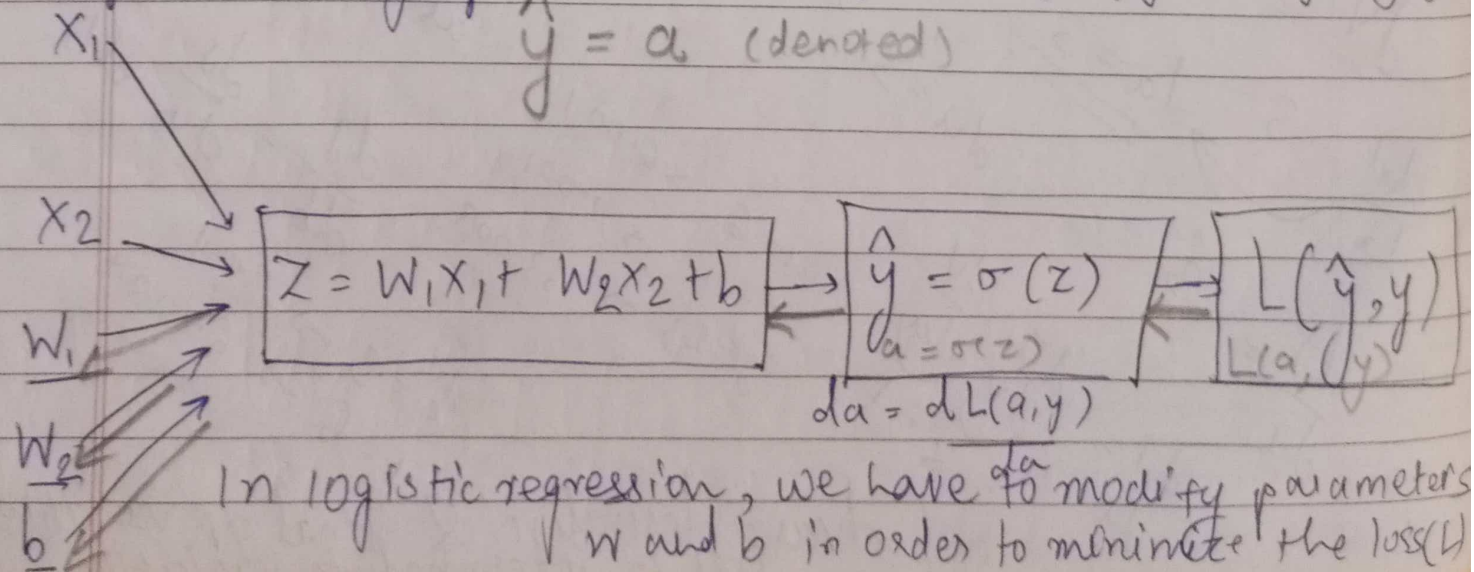
for logistic regression $z = w^T x + b$

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

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

Computational Graph -

$\hat{y} = a$ (denoted)



In logistic regression, we have to modify parameters w and b in order to minimize the loss(L)

This is entirely for one training example

Now computing for derivatives (backward propagation step)

$$da = \frac{dL(a,y)}{da} = d\left[-\{y\log(a) + (1-y)\log(1-a)\}\right]$$

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

$$dz = \frac{dL(a,y)}{dz} = d\left[-(y\log a + (1-y)\log(1-a))\right]$$

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

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

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

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

$$dz = d\left[-\{-y\log(1+e^{-z}) + (1-y)(-z - \log(1+e^{-z}))\}\right]$$

$$= d\left[y\log(1+e^{-z}) + z + \log(1+e^{-z}) - yz - y\log(1+e^{-z})\right]$$

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

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

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

$$\frac{dL}{dw_1} = dw_1 = x_1 \cdot dz \qquad \frac{dL}{db} = db = dz$$

$$\frac{dL}{dw_2} = dw_2 = x_2 \cdot dz$$

updating values:

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

(α : learning rate)

Now, we will check for m training examples how this works-

Gradient Descent on m Examples - (Training eg)

$$J(w,b) = \frac{1}{m}\sum_{i=0}^{m} L(a^{(i)}, y^{(i)}) \qquad (x^{(i)}, y^{(i)})$$

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

$$dw_1^{(i)},\ dw_2^{(i)},\ db^{(i)}$$

derivative wrt to w_1 of overall cost function is also average of derivatives wrt to w_1 of individual loss terms.

$$\frac{\partial}{\partial w_1} J(w,b) = \frac{1}{m}\sum_{i=0}^{m} \frac{\partial}{\partial w_1} L(a^{(i)}, y^{(i)})$$

$$\times \left[dw_1^{(i)} - x^{(i)} y^{(i)} \right] \times$$

$\Rightarrow$ algorithm for m training examples -

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

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

for i = 1 to m

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

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

for j = 1 .. n_x

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

$n_x = 2$

$$\begin{cases} dw_1\ += x_1^{(i)} dz^{(i)} \\ dw_2\ += x_2^{(i)} dz^{(i)} \end{cases} \rightarrow dw\ += x^{(i)} dz^{(i)}$$

considering $n_x = 2$ (two features (w_1, w_2) i.e only 2 input features)

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

} end of for loop

$$J /= m$$

dw /= m $\begin{cases} dw_1 /= m \\ dw_2 /= m \end{cases}$

$$db /= m$$

} averages over no. of training egs.

dw_1, dw_2, db are used as accumulators so they do not have a superscript i over it.
whereas $dz^{(i)}$ is for the respective training set example i'e calculated.

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

also updated values -

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

This is over a single step of gradient descent

You have to repeat this process over multiple times for gradient descent algorithm to implement.

We have to write 2 for loops for this :
(gradient descent for single step)

→ 1st for loop ←
2nd for loop ← for all the n features over here
(we have considered 2 features here)
($n = 2$ & $n_x = 2$)

Now basically writing such for loops makes the code inefficient for bigger datasets.
In Deep-learning algorithm (getting rid of for loops)
Therefore, Vectorization Techniques are used

PYTHON & VECTORIZATION

Vectorization art of getting rid of explicit ~~out~~ for loops in your code.

What is vectorization -

$$Z = W^T X + b \qquad W = \begin{bmatrix} \cdot \\ \cdot \\ \cdot \end{bmatrix} \quad X = \begin{bmatrix} \cdot \\ \cdot \\ \cdot \end{bmatrix}$$

non-vectorized implementation -

```
z = 0
for i in range(n-x)
    z += w(i) * x(i)
z += b
```

$W \in R^{n_x}$ $X \in R^{n_x}$ dimensional vectors

vectorized (in python on numpy)

$Z = \underbrace{np.dot(w, x)}_{W^T \cdot X} + b$

So for deep learning we prefer to use GPU rather than CPU but using vectorised approach we can even perform them on CPU

#

These computations involve [SIMD] instructions (single instructions multiple data)

so as GPU's are good at SIMD instructions than CPU's. (but CPU's are not too bad)

$u = Av$ → matrix A, vector v

$u_i = \sum_j A_{ij} v_j$

$u = np.dot(A, v)$ → Vectorised

non-Vectorised

```
u = np.zeros((n, 1))
for i . . . .   ← 1st for loop
    for j . . . .   ← 2nd for loop
        u_i += A[i][j] * V[j]
```

Vectors & matrix valued functions -

Say you need to apply exponential operations on every element of a matrix / vector.

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

non - vectorised

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

This makes the vector u set to zero

Vectorised

```
→ u = np.exp(v)
```

library numpy consists of vector valued functions -

```
np.log(v)        element wise log
np.abs(v)        element wise absolute value
np.maximum(v,0)  take max
v**2             square of every element
1/v → v^-1       inverse
```

Vectorising logistic Regression -

4 propagation step -

$X \Rightarrow$ Training inputs matrix $= \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}$ (n_x, m) $\mathbb{R}^{n_x \times m}$

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

M dimensional row vector (m : training eg)

W^T is a row vector

$$Z = [z^{(1)}\ z^{(2)} \dots z^{(m)}] = [W^T \text{ (1×n)}] \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ | & | & & | \end{bmatrix}_{n \times m} + [b\ b\ \dots\ b]_{1 \times m}$$

$$= [(W^T x^{(1)} + b) \quad (W^T x^{(2)} + b) \quad \dots \quad W^T x^{(m)} + b]$$

$(W^T x^{(1)} + b) \rightarrow z_1$, $(W^T x^{(2)} + b) \rightarrow z_2$

Capital Z consists of all z (lower case) in horizontal stacking

Capital X consists of all x (lower case) in horizontal stacking

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

(W.T: transpose; np.dot(W.T, X): vector; b: real no.)

confusion :-

b : real no. / (1×1 matrix)

but when we add this vector to this real no. python automatically takes this real no b & expands it out into 1×m row vector.

This is called Broadcasting in Python.

$$A = [a^{(1)}\ a^{(2)} \dots a^{(m)}] = \sigma(Z)$$ → for sigmoid fun use np.exp()

capital A consists of all a (lower case) in horizontal stacking.

↳ This is for m training examples

Vectorising Logistic Regression's Gradient Output -

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

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

1 × m row vector

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

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

(2) → $dZ = A - Y = [a^{(1)} - y^{(1)}\ \ a^{(2)} - y^{(2)}\ \cdots\ a^{(m)} - y^{(m)}]$

$dw = 0$ $\qquad$ $db = 0$

to eliminate for loop required for this purpose

$dw\ += x^{(1)} dz^{(1)}$ $\qquad$ $db\ += dz^{(1)}$

$dw\ += x^{(2)} dz^{(2)}$ $\qquad$ $db\ += dz^{(2)}$

$\vdots$ $\qquad$ $\vdots$

$dw /= m$ $\qquad$ $db /= m$

$\{dw /= m\}$

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

four step propagation

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

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

$b := b - \alpha db$

$w := w - \alpha dw$

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

$$= \frac{1}{m} \left[x^{(1)} dz^{(1)} + x^{(2)} dz^{(2)} + \dots + x^{(m)} dz^{(m)} \right]$$

$n \times 1$ ~~row vector~~

Getting rid of main for loop – multiple for iterations of gradient descent we need a for loop for that

for single iteration of gradient descent

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

$$A = \sigma(Z)$$

$$dZ = A - Y$$

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

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

$w := w - \alpha dw$ } update
$b := b - \alpha db$ }

Broadcasting in Python -

Calories from Carbs, Proteins, fats in 100g of different foods -

	Apples	Beef	Eggs	Potatoes
Carbs	56.0	0.0	4.4	68.0
Proteins	1.2	104.0	52.0	8.0
Fats	1.8	135.0	99.0	0.9

$= A_{3\times4}$

calories from

Calculate % of carbs, proteins, fats { without using for loop }

To create a matrix, we use numpy library

eg- A = np.array([[2, 1, 3, 4],
[5, 6, 7, 8]])

o/p: [[2 1 3 4]
[5 6 7 8]]

cal = A.sum(axis = 0) # To sum vertically

print (cal)

o/p : [7 7 10 12]

{axis 0 sum vertically} → axis 1 {sum horizontally}

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

print (percentage)

#redundant

reshape: while writing python codes if not sure what are dimensions of matrix.

o/p [[28.55 14.28 30 33.33]
[71.42 85.55 70 66.66]]

eg of Python broadcasting -

take a matrix A (2x4) & divide it by (1x4) matrix

Operation of (2x4) matrix divide by (1x4) matrix

Broadcasting —

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

↑ This is considered as

Broadcasting works with both row vectors & column vectors

Broadcasting done

(2) $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}_{2\times3} + \begin{bmatrix} 100 & 200 & 300 \\ 100 & 200 & 300 \end{bmatrix}_{1\times3 \text{ or } 2\times3} = \begin{bmatrix} 101 & 202 & 303 \\ 104 & 205 & 306 \end{bmatrix}$

(3) $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}_{2\times3} + \begin{bmatrix} 100 \\ 200 \end{bmatrix}_{2\times1} \begin{matrix} 100 & 100 \\ 200 & 200 \end{matrix}_{2\times3} = \begin{bmatrix} 101 & 102 & 103 \\ 204 & 205 & 206 \end{bmatrix}$

General Matrix :-
Principle Broadcasting

$(m,n) \quad \overset{+}{-}{*}{/} \quad (1,n) \rightsquigarrow (m,n)$

→ matrix

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

$[1 \quad 2 \quad 3] + 100 = [101 \quad 102 \quad 103]$

don't use

```
a = np.random.randn(5)
a.shape = (5,)
```
"This is a rank 1 array"
It is neither a row vector nor a column vector

} Don't use this in your code

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

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

logistic regression cost function -

If $y=1$: $p(y|x) = \hat{y}$ ①

$y=0$: $p(y|x) = 1-\hat{y}$ ②

Considering :- $p(y|x) = \hat{y}^{y}(1-\hat{y})^{(1-y)}$
both ① & ②

Since logarithmic function is strictly monotonically increasing function; & we have to maximized $\log p(y|x)$

$\uparrow \log[p(y|x)] = y\log\hat{y} + (1-y)\log(1-\hat{y})$

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

we minimize loss function

Statistics Principle :- Maximum Likelihood Estimation

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

To maximise this;

$\log p(\text{labels} \ldots) = \log \prod_{i=1}^{m} p(y^{(i)}|x^{(i)})$

log of products terms is $\uparrow$ log individual terms equal to sum of $\downarrow$ of

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

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

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

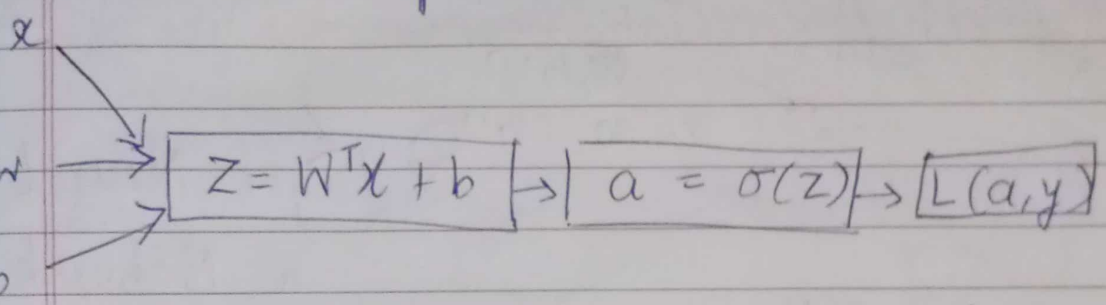
Cost function: $J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})$

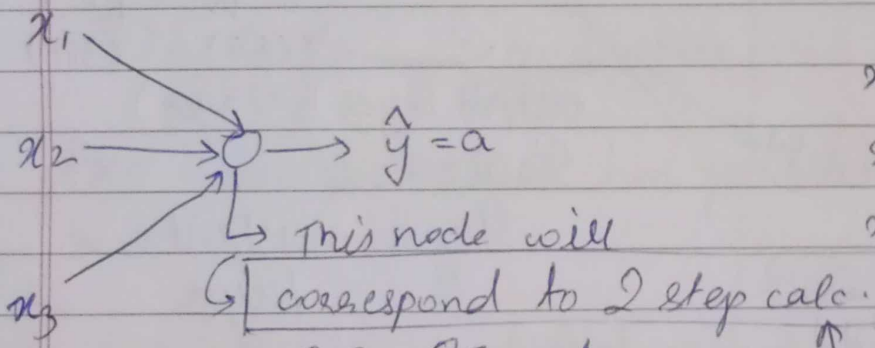
(minimise)

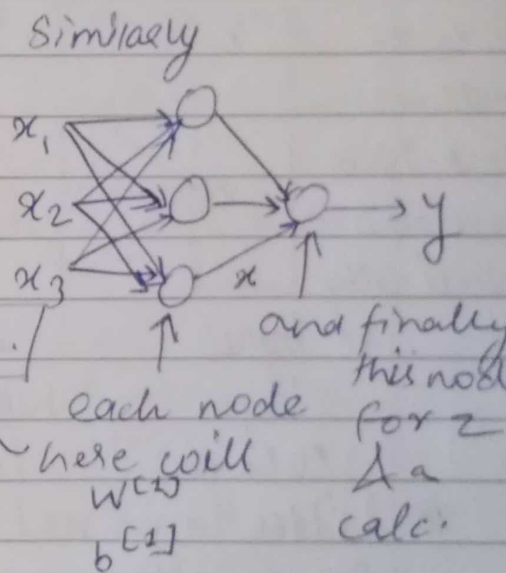
Why we minimise cost function?

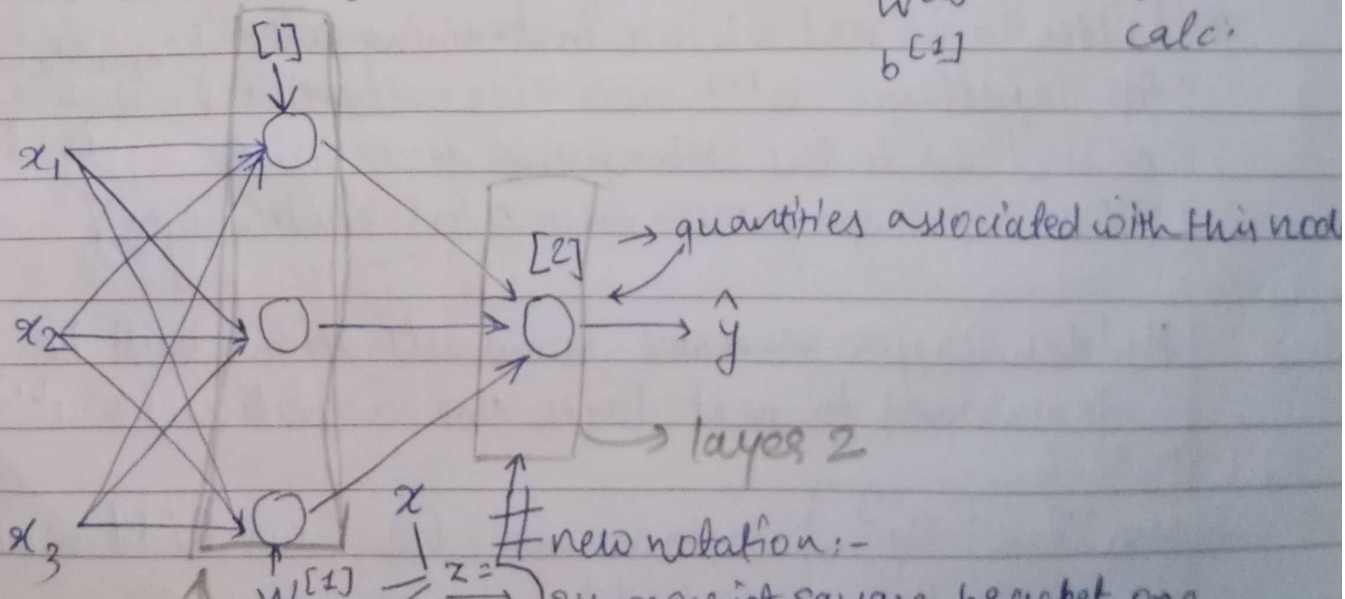
What is a Neural Network?

and how to implement









new notation:-

superscript square bracket one that represents quantities associated with this stack of nodes, its called a layer.

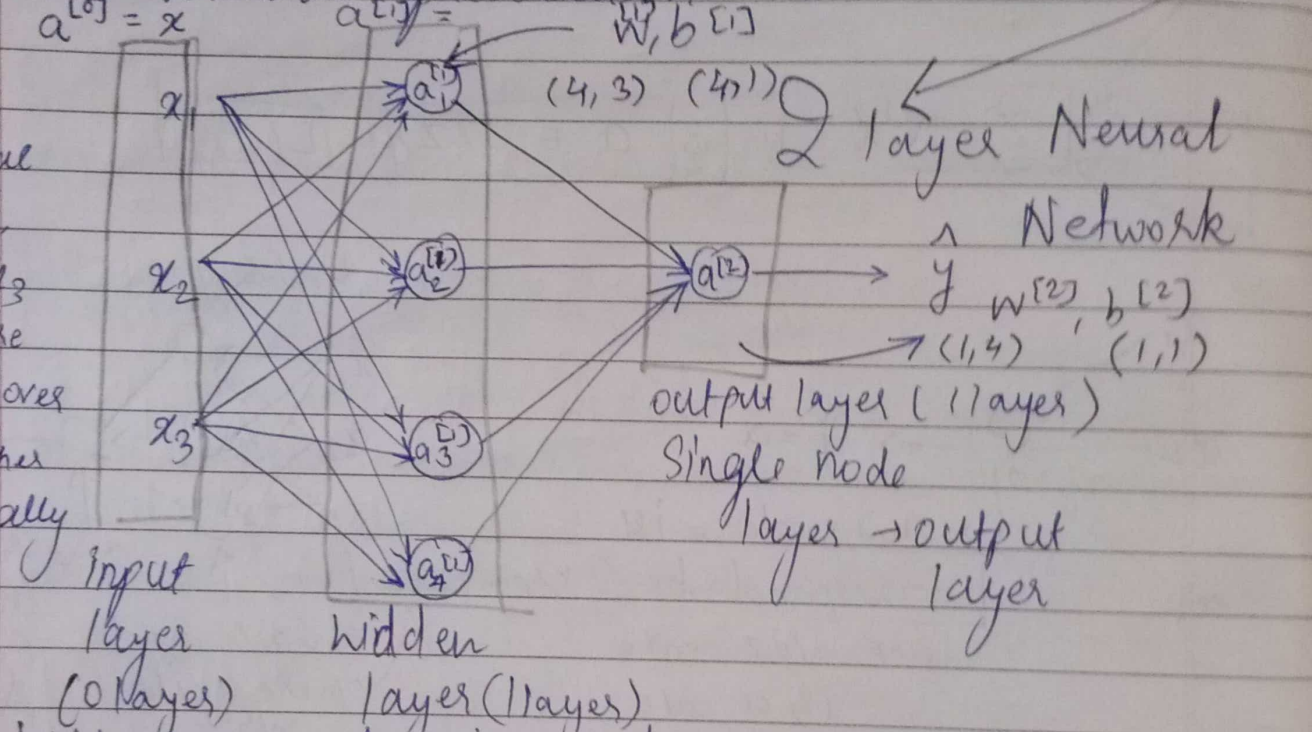
$x \xrightarrow[b^{[1]}]{w^{[1]}} z^{[1]} = w^{[1]}x + b^{[1]} \rightarrow a^{[1]} = \sigma(z^{[1]}) \xrightarrow{w^{[2]}} z^{[2]} = w^{[2]}a^{[1]} + b^{[2]} \rightarrow$

→ $a^{[2]} = \sigma(z^{[2]}) \longrightarrow L(a^{[2]}, y)$

'z' calculation followed by 'a' calculation.

To compute derivatives - to get backward calculations in a similar manner

Neural Network Representation -



* hidden layers not visible in training sets.
for input layer $a^{[0]}$ acts like activations of input layer which gets passed on to the subsequent next layer
(Here i/p layer passes value x to hidden layer)

Hidden layer generates some activations will be transferred to next layer the activation is $a^{[1]}$

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

So these are the four hidden units in hidden layer.

output layer will generate some activation $a^{[2]}$ which will be passed on $\hat{y}$ so $\hat{y} = a^{[2]}$

So, this network is refered as to "2 layer neural network"
We don't consider input layer only considering hidden & output layer. [1 hidden + 1 o/p]

We mention input layer as a zero & count the rest layers (hidden + output) as per notation convention.

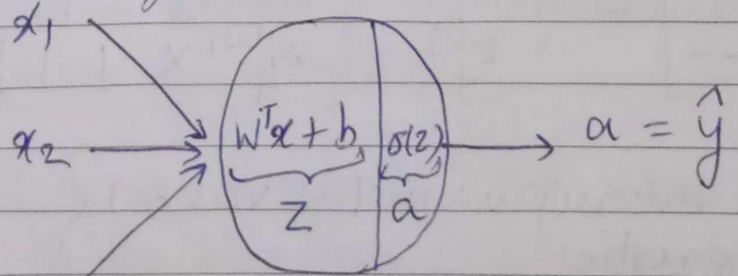
Imp — Hidden layers & output layer will have parameters associated with it.

Hidden layer has parameters w & b associated with it.

Also some o/p layers has parameters w & b associated with it.

Computing a Neural Network's Output -

In Logistic Regression -



$z = W^T x + b$

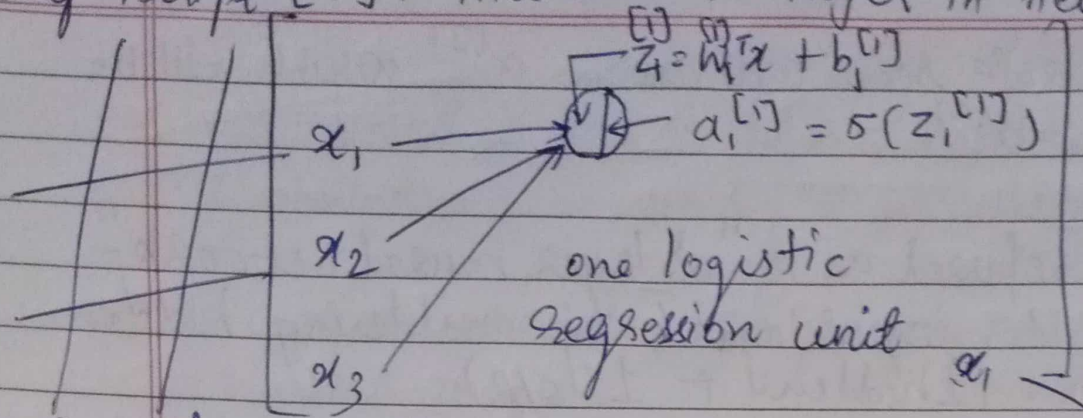
$a = \sigma(z)$

This 2 step computation of z & a is repeated many times in neural network.

Let's consider the first hidden node.

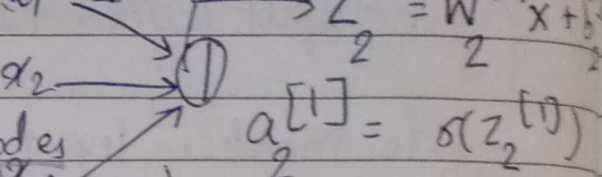
subscript 1 : indicates 1st node in that corresponding layer

superscript [1] : indicates 1st layer in neural network



i.e $a_i^{[l]}$ → layer
i → node in layer

#



Second node in hidden layer

Similarly, we write z & a calc for all nodes

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

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

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

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

Vectorising this

to calculate z & a as activations

consider; $W^{[1]}$ (4×3)

$$z^{[1]}_{(4,1)} = \begin{bmatrix} — & w_1^{[1]T} & — \\ — & w_2^{[1]T} & — \\ — & w_3^{[1]T} & — \\ — & w_4^{[1]T} & — \end{bmatrix}_{(4\times3)} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}_{(3,1)} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix}_{(4,1)} = \begin{bmatrix} w_1^{[1]T} x + b_1^{[1]} \\ w_2^{[1]T} x + b_2^{[1]} \\ w_3^{[1]T} x + b_3^{[1]} \\ w_4^{[1]T} x + b_4^{[1]} \end{bmatrix} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix}$$

input features (nx)

Stacking 4 w vectors (parameters vectors)
i.e we have 4 logistic regression units

we will run all these values stacked vertically

we stacked all z values for each node of that layer

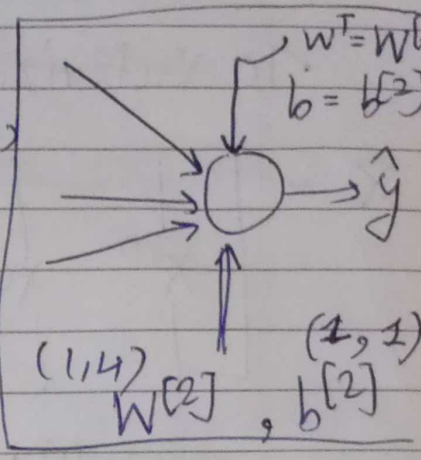
$$\underline{a}^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix} = \sigma(\underline{z}^{[1]})$$

for hidden layer;

$x = a^{[0]}$ (3,1)

$z^{[1]}_{(4,1)} = W^{[1]}_{(4,3)} a^{[0]}_{(3,1)} + b^{[1]}_{(4,1)}$

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



for output layers, we have parameters W & b as well

here; $x = a^{[1]}$

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

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

} o/p layer

Vectorised version

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

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

} hidden layer

for 2 layer neural network similarly

Vectorising across multiple training examples for neural networks -

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

for m training egs; repeat above step over m egs.

$x^{(1)} \longrightarrow a^{[2](1)} = \hat{y}^{(1)}$ (i) → refers to training eg i

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

$\vdots$

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

$a^{[2](i)}$ → training example i

→ layer 2

← training examples

↕ different units (i.e hidden units of layer)

for $i = 1$ to m

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

To Vectorize it.

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

explained later

$$\begin{aligned} Z^{[1]} &= W^{[1]} X + b^{[1]} \\ A^{[1]} &= \sigma(Z^{[1]}) = g^{[1]}(Z^{[1]}) \\ Z^{[2]} &= W^{[2]} A^{[1]} + b^{[2]} \\ A^{[2]} &= \sigma(Z^{[2]}) \end{aligned}$$

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

(column vector stacking)

So $X^{(i)}$ elements / examples are stacked vertically in a column manner one after another to get X

Similarly doing this for $Z^{[1]}$, and $A^{[1]}$, $Z^{[2]}$ & $A^{[2]}$

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

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

scanning from left to right refers to the scanning of all training examples

scanning from top to bottom consists of various nodes present in that layer (corresponding layer nodes)

different units in layer

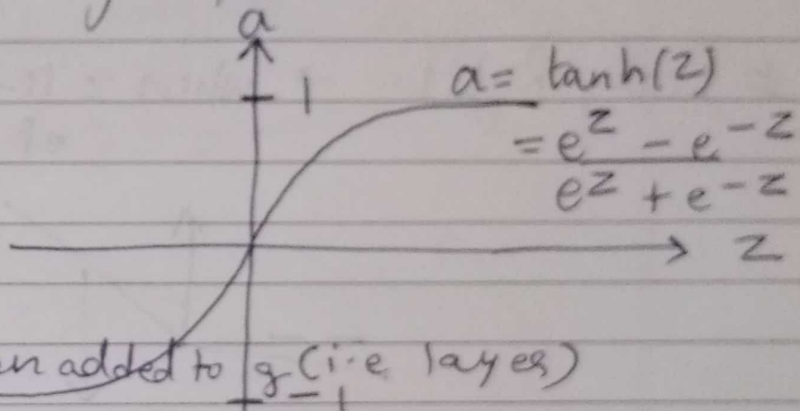
Why we use this activation func (hyperbolic tangent func)
→ It causes the mean of our data to come close to zero (as it ranges betw −1 & 1) (than mean 0.5)
→ Due to this centering of data takes place

Activation functions
Till now we were using sigmoid activation function, but there are still different choices available of activation function

If a function is a non-linear function & not a sigmoid function.

Activation function working better than sigmoid func is Tangent function / Hyperbolic tangent function.

$a = \tanh(z)$
(goes between −1 & +1)



let $a^{[1]} = g^{[1]}(z^{[1]})$
$a^{[2]} = g^{[2]}(z^{[2]})$
↳ superscript even added to g (i.e layers)

But there is one exception, for output layer we have to use the sigmoid activation function as it ranges btwn 0 & 1 the value of $\hat{y}$ easy to use. ∵ $y \in (0,1)$, ∴ $\hat{y} \in (0,1)$ for binary classification used as a output layer

∴ Here $g(z^{[2]}) = \sigma(z^{[2]})$

If z is very large or is very small, the slope of function is close to zero (very small)
This
⟹ (slow downs gradient descent)

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

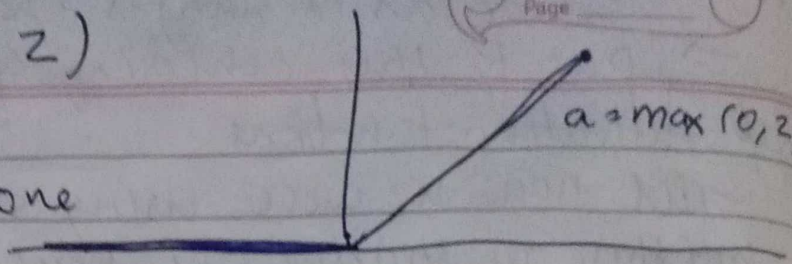
$= \frac{(e^{2z} + 2 + e^{-2z}) - (e^{2z} + e^{-2z} - 2)}{(e^z + e^{-z})^2}$

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

$= \left[\frac{\quad}{(e^z + e^{-z})^2}\right.$

Sol for this → Rectified Linear Unit (ReLU function)

$a = \max(0, z)$

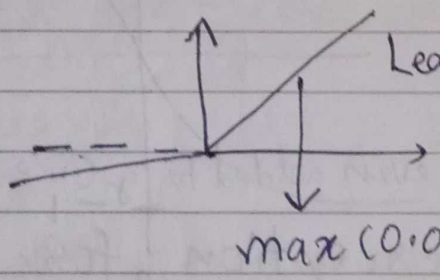


if z is +ve
slope (derivative) is one
i.e da = 1

if z is -ve
slope (derivative) is zero
i.e da = 0

ReLU activation function is the default choice now-a-days.

Leaky ReLU → when z is -ve its a slight slope instead of zero.

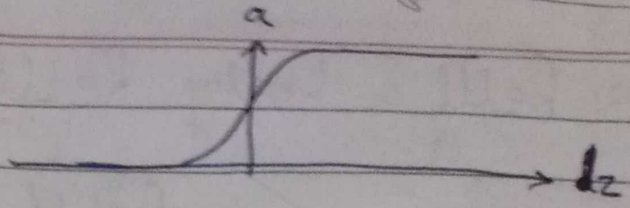


→ Choose Activation function that Bests Suit Your Applications ←

If we use linear (identity activation function) (i.e $a^{[1]} = z^{[1]}$) activation function; it won't matter the presence of hidden layer then it behaves like logistic regression. It is basically calculating i/p to give us o/p using linear activation function and there is no factor of non-linearity in that case

Derivatives of Activation Functions -

Sigmoid activation function -



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

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

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

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

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

we know $a = g(z)$

$z = 10$, $g(z) \approx 1$

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

$z = -10$ $g(z) \approx 0$

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

$z = 0$ $g(z) \approx 1/2$

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

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

Tanh activation function -

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

$\frac{dg(z)}{dz} = g'(z)$ = slope of $g(z)$ at z {Special page}

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

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

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

$z = 10$ $\tanh(z) \approx 1$

$g'(z) \approx 0$

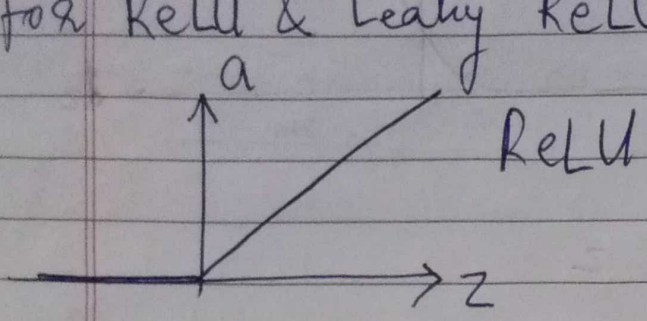
$z = -10$ $\tanh(z) \approx -1$

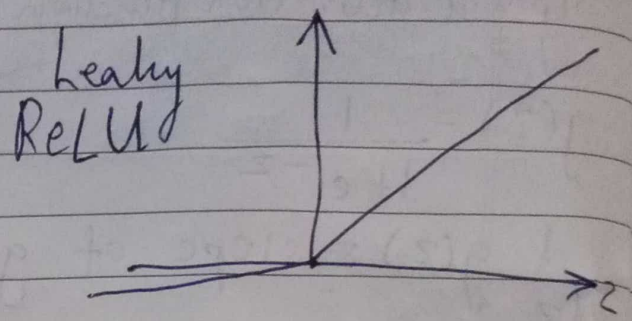
$g'(z) \approx 0$

$z = 0$ $\tanh(z) = 0$

$g'(z) = 1$

for ReLU & Leaky ReLU





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

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

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

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

technically this will be treated as 0.00000...

Gradient descents for neural networks – (1 hidden layer)

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

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

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

input units, hidden units, output units

Cost function: $J(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]})$

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

$\hat{y} \rightarrow a^{[2]}$

Gradient Descent:

Repeat {

np.sum → Python numpy command for summing across 1 dimension of matrix

role of keepdims : Its prevents Python from outputting 1 d array (rank 1 array)

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

Compute derivatives $dW^{[1]} = \frac{\partial J}{\partial W^{[1]}}$, $db^{[1]} = \frac{\partial J}{\partial b^{[1]}}$, ...

Similarly $dW^{[2]}$ & $db^{[2]}$

update values
$W^{[1]} := W^{[1]} - \alpha\, dW^{[1]}$
$b^{[1]} := b^{[1]} - \alpha\, db^{[1]}$
$W^{[2]} := W^{[2]} - \alpha\, dW^{[2]}$
$b^{[2]} := b^{[2]} - \alpha\, db^{[2]}$

This is over one gradient descent iteration and continues until parameters are converging.

---

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 o/p only

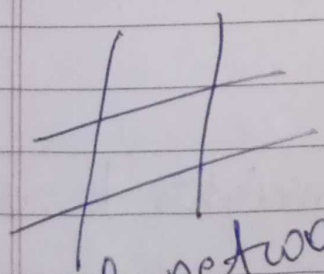
back propagation :-

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

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

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

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



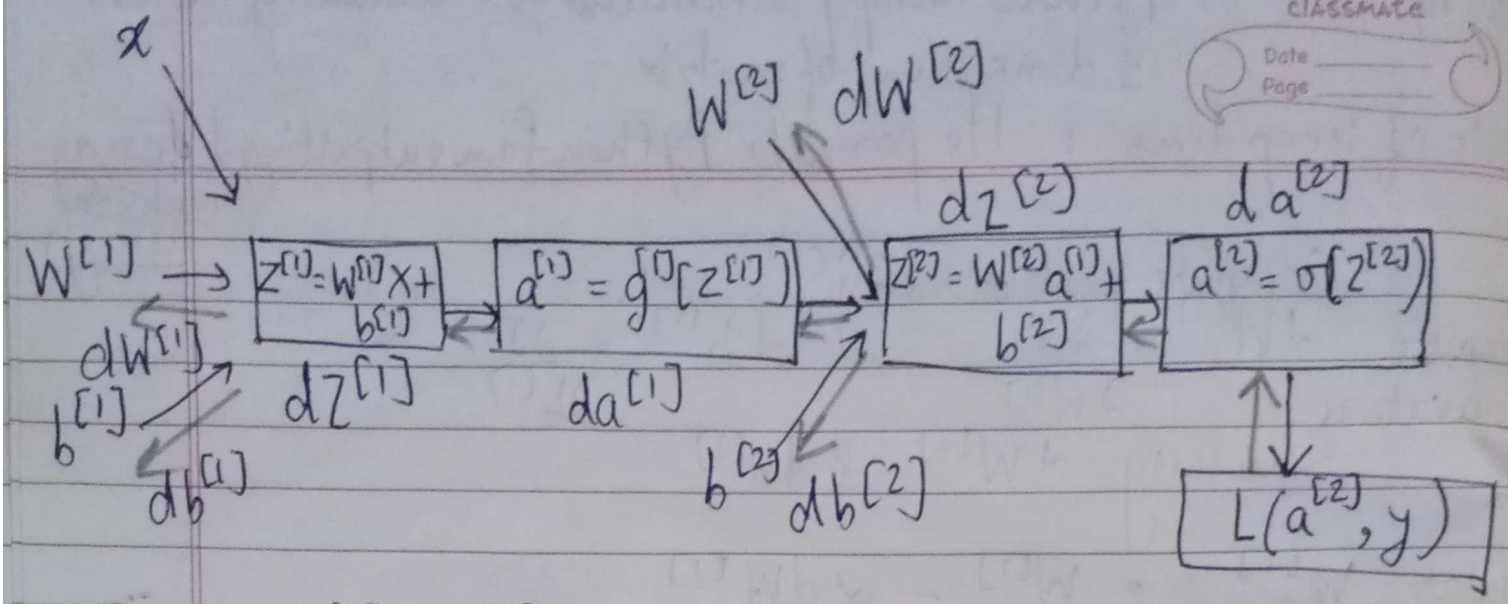
neural network (with 1 hidden layer)

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

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

Here, element wise product of 2 matrices

$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$ ($= dZ^{[1]} A^{[0]T}$) $db^{[1]} = \frac{1}{m}$ np.sum($dZ^{[1]}$, axis=1, keepdims=True)



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

$$dW^{[2]} = dz^{[2]} a^{[1]T} \rightarrow "dW = dZ \cdot X"$$

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

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

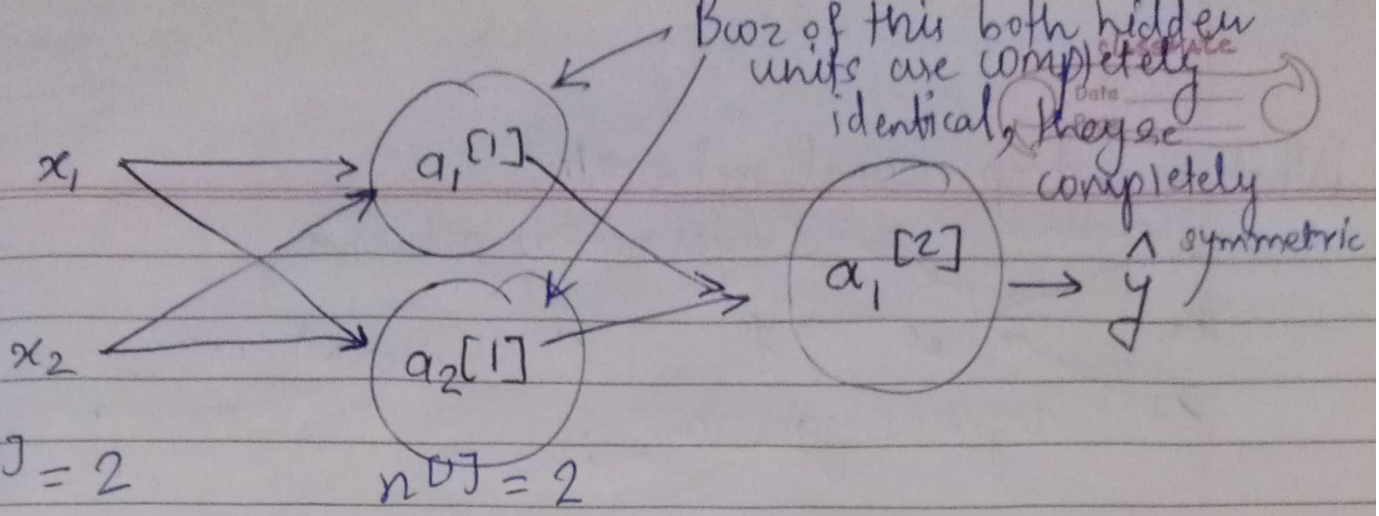
for W & dW, Z & dZ should have same dimensions
i.e Variable & its derivative should have same dimension.

* : element - wise product

Random Initialization -

What happens if you initialize weights to zero?

for logistic regression, we initialized weights with zero which was fine but for a neural network initializing weights to parameters(W) to all zero and applying gradient descent wont work



$$W^{[1]} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \qquad b^{[1]} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \qquad W_0^{[2]} = [0\ 0]$$

↓ Problem

↳ this is fine

$a_1^{[1]} = a_2^{[1]}$ → It would be same

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

how

So many time you run neural network, both hidden units are still computing the same function.

SOl to this :- $W^{[1]} = np.random.randn((2,2)) * 0.01$

(gaussian vars)

initializing to very small values.

? Why only 0.01

np for (no problem) → $b^{[1]} = np.zeros((2,1))$

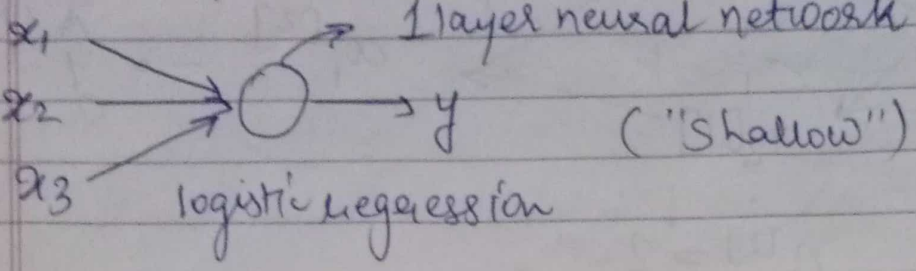
if you take / initialize with larger values $z^{[1]}$ will be very large / very small i.e these will be the parts of the activation function where slope / gradient is very small so gradient descent is very slow

$W^{[2]} = \ldots$

$b^{[2]} = 0$

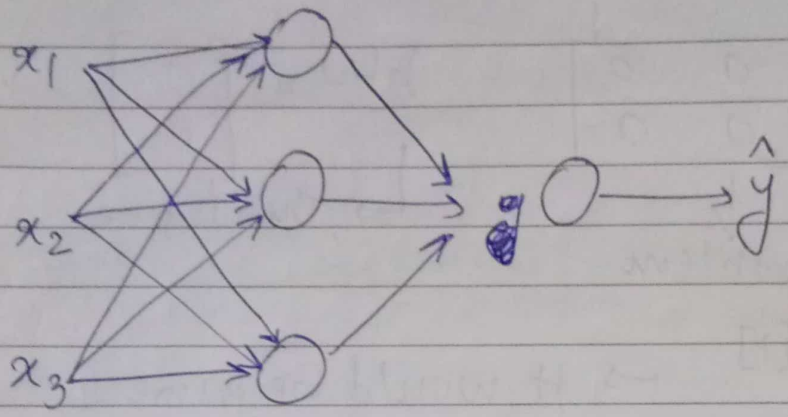
[↑] [↓] [↑] [↓]

What is a deep neural network?



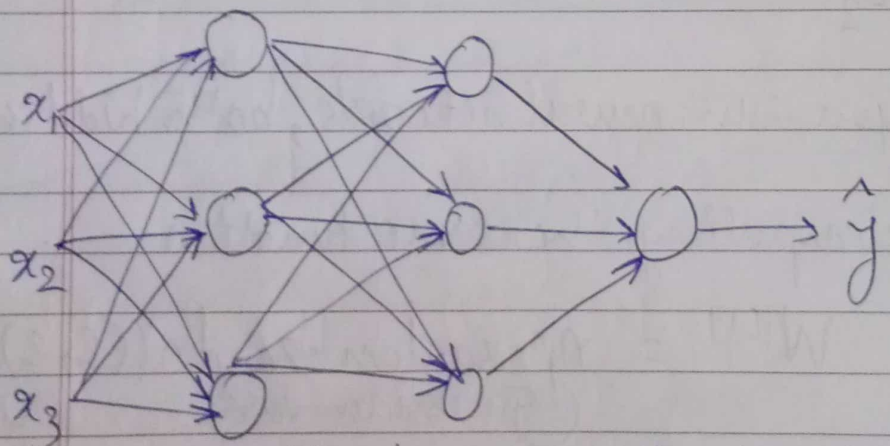
1 layer neural network ("shallow")

logistic regression



2 layer neural network

1 hidden layer neural network



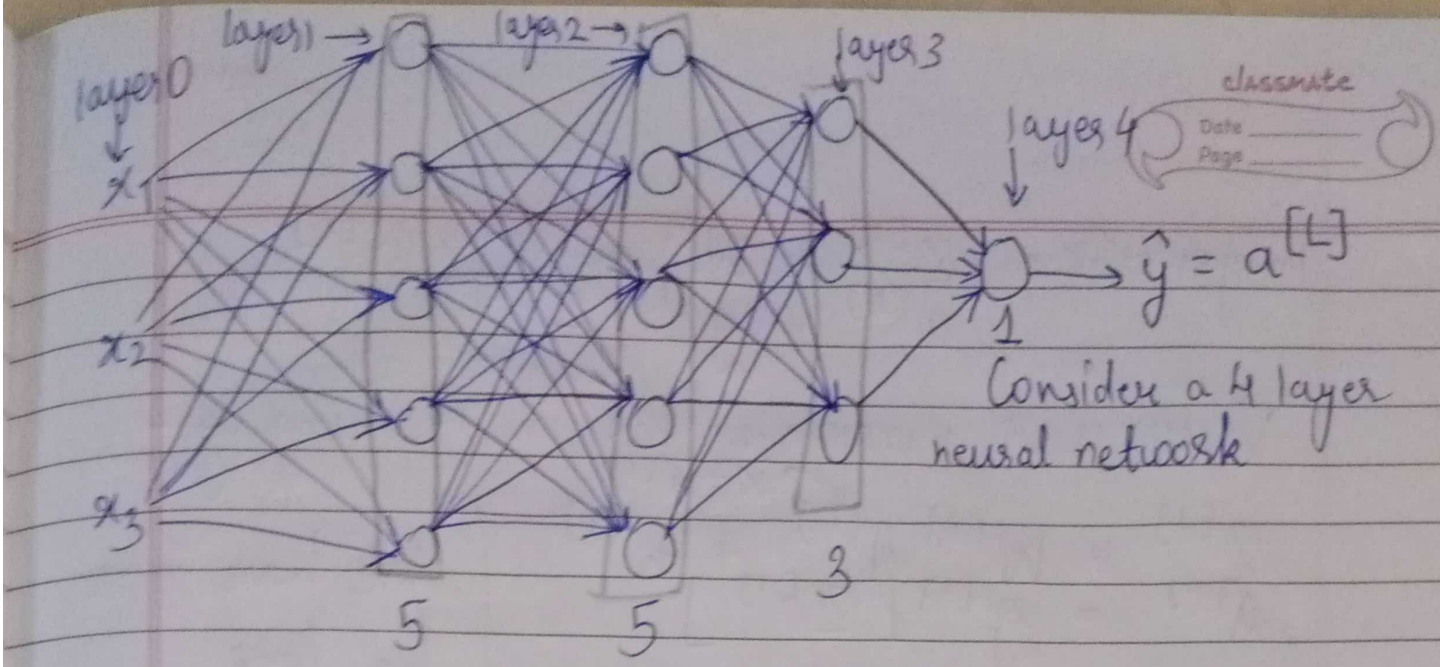
3 layer neural network

2 hidden layer neural network

5 layer neural network - ("deeper")

The degree depends upon the depth of neural network

deep neural network notation -



Consider a 4 layer neural network

L : Number of layers in the network $L = 4$ (# layers)
$n^{[l]}$: Number of nodes/units present in layer l
$a^{[l]}$: Activations in layer l
$n^{[0]} = 3 = n_x$
$n^{[1]} = 5$
$n^{[2]} = 5$
$n^{[3]} = 3$
$n^{[4]} = n^{[L]} = 1$

input features : X
X is called activations of layer zero $a[0] = X$

$a^{[l]} = g^{[l]}(z^{[l]})$ $\quad z^{[l]} \Rightarrow$ we require $W^{[l]}$ (weights & $b^{[l]}$ of parameters)

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

for a single training example-

X : $z^{[1]} = W^{[1]} \underbrace{X}_{a^{[0]}} + b^{[1]} = W^{[1]} a^{[0]} + b^{[1]}$
$a^{[1]} = g^{[1]}(z^{[1]})$
$z^{[2]} = W^{[2]} a^{[1]} + b^{[2]}$
$a^{[2]} = g^{[2]}(z^{[2]})$
:
$z^{[4]} = W^{[4]} a^{[3]} + b^{[4]}$
$a^{[4]} = g^{[4]}(z^{[4]})$

In general: $Z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}$
$a^{[l]} = g^{[l]}(Z^{[l]})$

Vectorised version: for m training eg -

$X = A^{[0]}$

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

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

$Y = g^{[4]}[Z^{[4]}] = A^{[4]}$

This (for loop) needs to get implemented

We have to use for loop for $l = 1, \cdots, L$ (Here $L = 4$)

This for loop can't be avoided

dimensions of matrix $\Rightarrow W$

$W^{[l]} \Rightarrow (n^{[l]}, n^{[l-1]})$

dimensions of matrix $\Rightarrow b$

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

dimension of dW & W should be same.
Similarly for db & b.

dimensions of X, Z, A depends on vectorization (number of training examples m)

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

But $Z^{[l]}, A^{[l]} \Rightarrow (n^{[l]}, m)$

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

for a layer in deep neural network:-

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

forward propagation : input $a^{[l-1]}$, output $a^{[l]}$

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

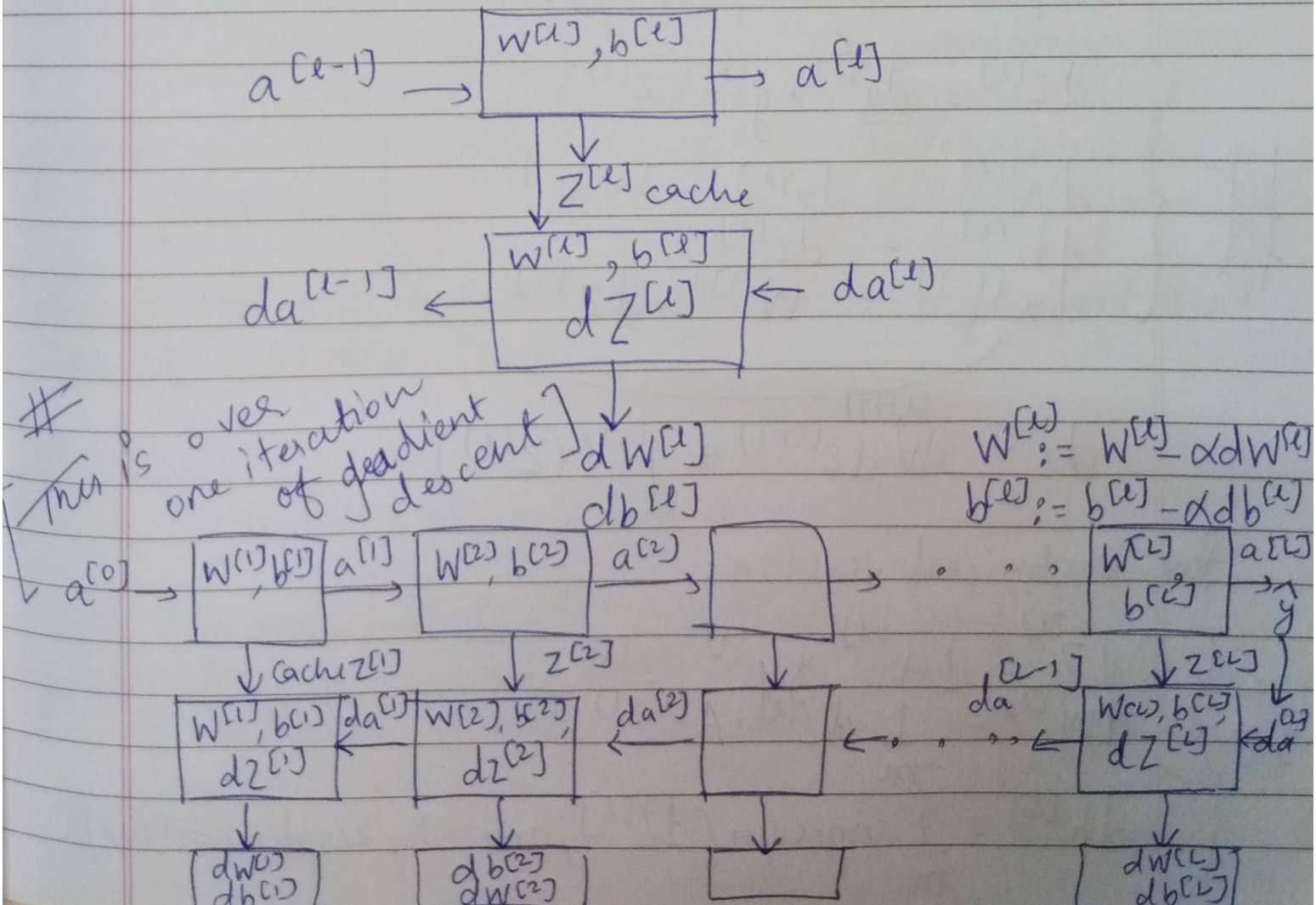
$$a^{[l]} = g^{[l]}(Z^{[l]})$$

cache $Z^{[l]}$ is stored for back propagation (storing)

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

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

layer l



This is over one iteration of gradient descent

forward propagation for layer l

Input $a^{[l-1]}$

Output $a^{[l]}$, cache $(z^{[l]})$ → $W^{[l]}, b^{[l]}$

$$z^{[l]} = W^{[l]} \cdot a^{[l-1]} + b^{[l]}$$
$$a^{[l]} = g^{[l]}(z^{[l]})$$

Vectorized: -

In general

$$Z^{[l]} = W^{[l]} \cdot A^{[l-1]} + b^{[l]}$$
$$A^{[l]} = g^{[l]}(Z^{[l]})$$

backward propagation for layer l

input data - $da^{[l]}$

output - $da^{[l-1]}, dw^{[l]}, db^{[l]}$

$$\begin{cases} dz^{[l]} = \underline{\underline{da^{[l]}}} * g^{[l]\prime}(z^{[l]}) \\ dW^{[l]} = dz^{[l]} \cdot a^{[l-1]T} \\ db^{[l]} = dz^{[l]} \\ da^{[l-1]} = W^{[l]T} \cdot dz^{[l]} \end{cases}$$

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

for vectorized version:

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

$$dA^{[l-1]} = W^{[l]T} \cdot dZ^{[l]}$$

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

#

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

Parameters V/s Hyperparameters

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

Hyperparameters: learning rate α
no of iterations
no of hidden layers
no of hidden units
choice of activation function

These the parameters that control the parameters W & b ultimately. Hence they are called hyperparameters whose values need to be set

Hyperparameters influence the parameters

↓

alot of setting up needs to be done to better implement that neural network

Its like -
* idea → code → experiment *

Best, to be identified directly → research is ongoing hyperparameters