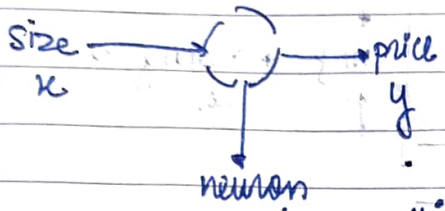
Date

Introduction to neural networks



it just acts as a func

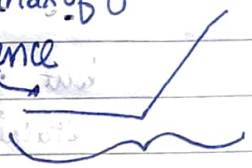
a neural network is built by stacking multiple neurons w/ diff features as i/p to get o/p

ReLu

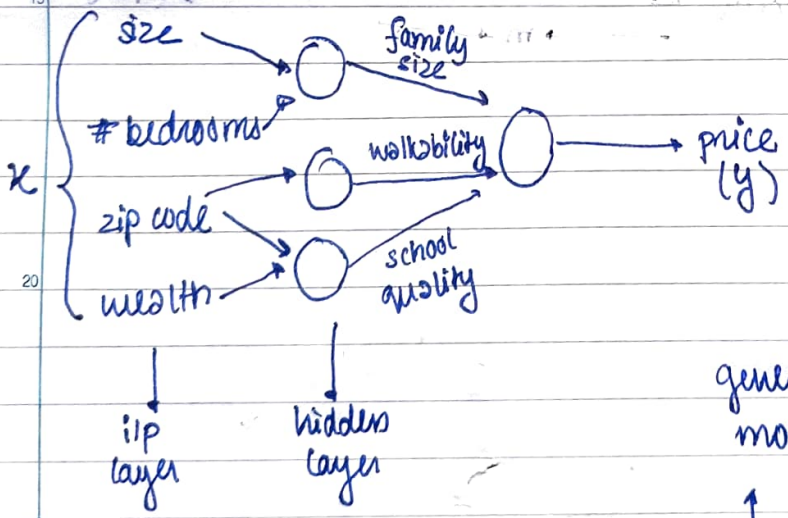
Rectified linear units

mean taking a max of 0

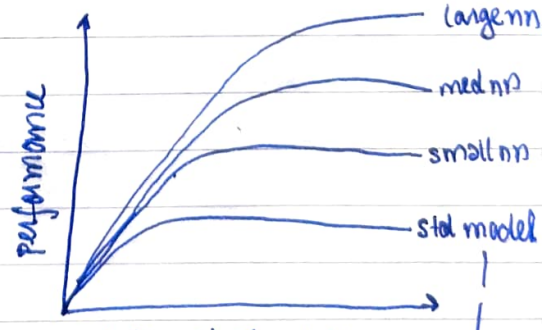
hence



this is what ReLu func looks like



generally w/ traditional models,



like normal supervised model w/o neural network

Date :

Note:

Traditionally we write

$$X = \begin{bmatrix} — & x^1 & — \\ — & x^2 & — \\ & \vdots & \\ — & x^m & — \end{bmatrix}$$

m → training eg no.

n_x → feature no.

$$y = \begin{bmatrix} y^1 \\ y^2 \\ \vdots \\ y^m \end{bmatrix}$$

y → $(m, 1)$

but in neural network if we do horizontal stacking computation becomes easier

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

n_x ↑, → m

X → $X.\text{shape} = (n_x, m)$

$y = [y^1 \; y^2 \cdots y^m]$ → $(1, m)$

Date

Logistic Regression as Neural Network

Generally,

given, $x \in \mathbb{R}^{n_x}$ → n_x dim vector $(x^1, x^2, \ldots x^m)$ → each is a feature

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

& param: $w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$

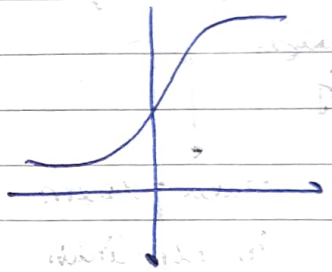
let say we have 64×64 pix img

⇒ x = 64×64 dim vector → 4096

then we know,

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

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



sometime it can also be denoted as,

$x_0 = 1$, $x \in \mathbb{R}^{n_x+1}$, $\hat{y} = P(y=1 \mid x)$

$$= \sigma(\theta^T x)$$

$$\theta = \begin{bmatrix} \theta^0 \\ \theta^1 \\ \vdots \\ \theta^{n_x} \end{bmatrix} \quad \theta^0 \to b = \theta^0, \quad w = \theta^1, \ldots \theta^{n_x}$$

but this makes implementing nn a tad bit more difficult hence we use prev notation keeping w & b seperate

Date : _____

Cost function

for creating loss func we could.

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

→ its a convex loss func

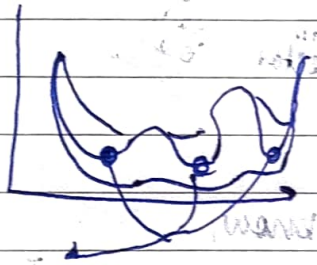
→ but this causes a convex optimization but using it in non-convex models (eg nn) we get a non-convex surface w/ multiple local minima hence creating a non-convex optimization problem

MSE

** used in convex landscape & non-convex model is linear func of its params

but useful only if

causes problem in non-linear param func → problem not because of MSE but cuz of presence of multiple local minima



local minima's

→ MSE is used in non-convex application even if it might cause problem because its simple & reliable and can give pretty good values if we tweak the model a bit

Date :

but MSE not used widely,

we use

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

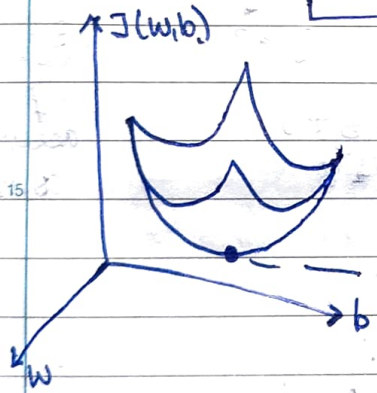
done w/ optimization

the loss func was defined wrt single training eg

cost func measures how well we doing in entire dataset

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

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



J(w,b) is min at this point for which we want the val for w & b

Computation graphs

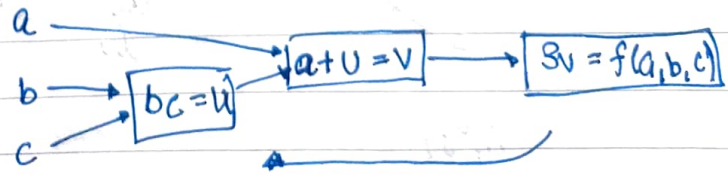
let's say we have,

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

$bc = u$

$a + u = a + bc = v$

$3v = f(a,b,c)$



if we go back we get derivative

Date :

Lets do some back propogation,

$J = 3v$

$\Rightarrow \frac{dJ}{dv} = 3$ } 1 step back

now if we increase 'a' how does that affect J,

$a = 5 \longrightarrow 5.001$

$v = 11 \longrightarrow 11.001$

$\Rightarrow J = 33 \longrightarrow 33.003$ } from here we see $a\uparrow \Rightarrow J\uparrow \times 3$

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

$J = 3v = 3a + 3v$ (assumed cost.)

$\frac{dv}{da} = \frac{da}{da} + \frac{du}{da}$ (if $a\uparrow$ how much $v\uparrow$ by)

$= 1 + 0$

$\Rightarrow \frac{dJ}{da} = 3 + 0 = 3$

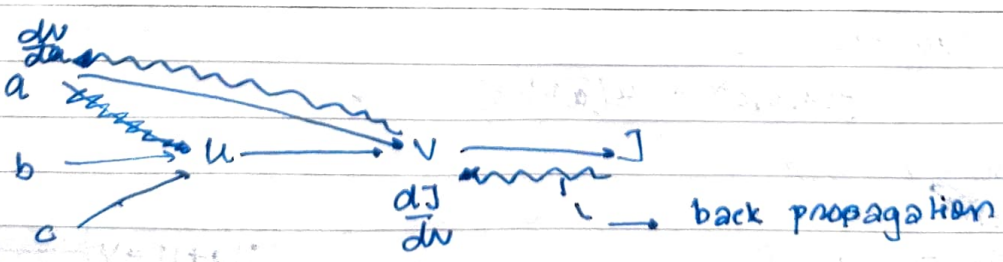
$\frac{dJ}{dv} \cdot \frac{dv}{da}$ ($\frac{dJ}{dv} = 3$, $\frac{dv}{da} = 1+0$)

use d if 'a' affects 'v'

& 'v' affects J

$\Rightarrow a \rightarrow v \rightarrow J$

so till now we have done



Illy,

$\frac{dv}{du} = 1$ & $\frac{dJ}{dv} = 3$ $\Rightarrow \frac{dJ}{du} = 3$

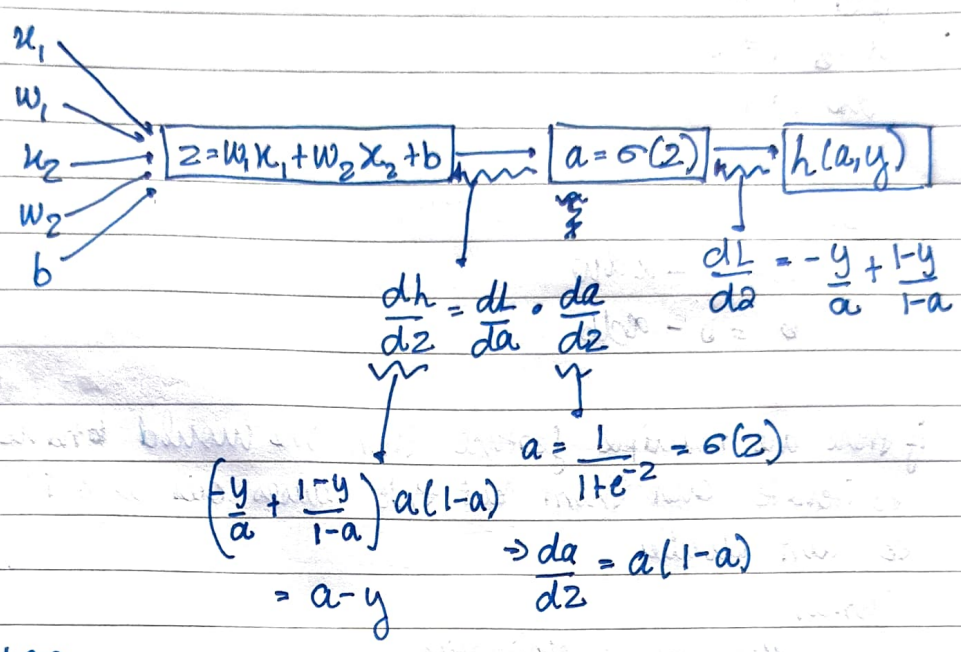
then,

$\frac{dJ}{db} = \frac{dJ}{du} \cdot \frac{du}{db} \Rightarrow \frac{dJ}{db} = 3c$ & $\frac{dJ}{dc} = 3b$

($\frac{du}{db} \rightarrow c$, $\frac{dJ}{du} \rightarrow 3$)

Date : _____

Now using derivatives to back propogate on logistic reg,



then

$$\frac{dh}{dw_1} = x_1 \frac{dh}{dz} \qquad \frac{dh}{dz} \cdot \frac{dz}{dw_1}, \quad \frac{dz}{dw_1} = x_1$$

$$= x_1 \frac{dh}{dz}$$

then updates for gradient descent becomes,

$$w_1 = w_1 - \alpha \frac{dh}{dw_1}$$

Gradient for m examples

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

for i in m:

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

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

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

$dz^i = a^i - y^i$

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

$\vdots$

$dw_n \mathrel{+}= x_n^i dz^i \qquad db \mathrel{+}= dz^i$

Date : _____

$J /= m$

$dw_1 /= m$

$dw_2 /= m$

$db /= m$

$\frac{dJ}{dw_1}$ ←

then,

$w = w - \alpha\, dw$

$b = b - \alpha\, db$

if there are multiple feature then we would have to iterate over them but this causes our algorithms to run slowly

hence,

we use vectorization

before for,

$z = w^T x + b$

we did,

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

vectorized format do,

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

Note:

both

GPU, CPU → have SIMD instru.

single ins multi data → basically parallelism

more used by vectorization

⇒ lesser time taken

Date

COST function

lets say,

if $y=1$: $P(y|x) = \hat{y}$

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

} only 2 ($y=0, y=1$) ∵ binary classification

these 2 eqⁿ can be written as

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

at $y=1$: $\hat{y}^{1} \cdot (1-\hat{y})^{(1-1)}$

$= \hat{y}$

so at $y=0$: $\hat{y}^{0} \cdot (1-\hat{y})^{(1-0)}$

$= 1-\hat{y}$

same as before

Now,

taking log

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

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

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

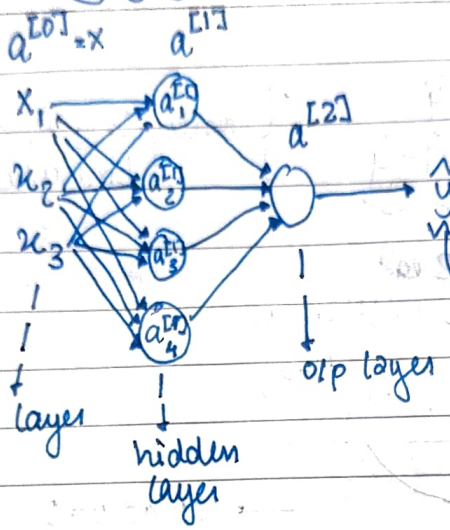
$$\Rightarrow \downarrow L(\hat{y}, y) \longrightarrow \uparrow \log(P(y|x))$$

$a_i^{[l]}$ where, l = layer no. i = node in layer

Date :

Neural Network

in counting layers we don't count i/p layer



$a^{[2]} = \hat{y}$ $a^{[0]} = x$

instead of showing i/p as x we now do,

where,
a = activation func
↓
refers to val the current layer is passing to subsequent layer

here,
$a^{[1]}$ is 4 dim vector
∵ it has 4 diff nodes in its layer

The hidden layer has params attached to it,

$w^{[1]}$ & $b^{[1]}$

$w^{[1]}$ → (4,3) mat
3 ∵ 3 i/p
4 ∵ 4 nodes in the layer

$b^{[1]}$ → (4,1) vect

The o/p layer also has params,

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

$w^{[2]}$ → (1,4)
4 ∵ hidden layer has 4 nodes

$b^{[2]}$ → (1,1)
1 ∵ 1 o/p node

for o/p node the hidden node acts as i/p

Date :

Lets take a look at a single node at the hidden layer

div into two parts

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

doing this for all layers w/ for loop is very inefficient

instead we would

$$\begin{bmatrix} w_1^{[1]} \\ w_2^{[1]} \\ w_3^{[1]} \\ w_4^{[1]} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix}$$

$(4,3)$ $w^{[1]}$ $\qquad$ $b^{[1]}$

$$\therefore \underset{(4,1)}{z^{[1]}} = \underset{(4,3)}{w^{[1]}} \underset{(3,1)}{a^{[0]}} + \underset{(4,1)}{b^{[1]}}$$

in case of multiple training examples

$a^{[2]\,[i]} = \hat{y}^{[i]}$

layer 2 $\qquad$ example no.

for $i = 1$ to m:

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

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

$z^{[1]\,[i]} = \ldots$

$\vdots$

remember training eg were stacked ~~horizontally~~ vert

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

(n_x, m)

Date :

Why stacking training examples vertically is good ??

let see,

$$W^{[1]} = \begin{bmatrix} — \\ — \\ — \end{bmatrix} \quad \& \quad x^{(1)} = \begin{bmatrix} \cdot \\ \cdot \\ \cdot \end{bmatrix}, \quad x^{(2)} = \begin{bmatrix} \cdot \\ \cdot \\ \cdot \end{bmatrix}$$

then according to matrix mul

from

$$W^{[1]} x^{(1)} = \begin{bmatrix} \cdot \\ \cdot \\ \cdot \end{bmatrix} \quad \text{!!!y} \quad W^{[1]} x^{(2)} = \begin{bmatrix} \cdot \\ \cdot \\ \cdot \end{bmatrix}$$

& if we do

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

$$\begin{bmatrix} | & | \\ x^{(1)} & x^{(2)} \\ | & | \end{bmatrix}$$

now if we add $b^{[1]} = \begin{bmatrix} \cdot \\ \cdot \\ \cdot \end{bmatrix}$

to above then python using broadcasting adds $b^{[1]}$ to each col

Hence,

vectorizing in column help makes calc easier & faster

i.e $z^{[1]} = W^{[1]} X + b^{[1]} \rightarrow A^{[0]}$

helps vectorize

```
for i=1 to m:
    z[1](i) = W[1] x(i) + b[1]
```

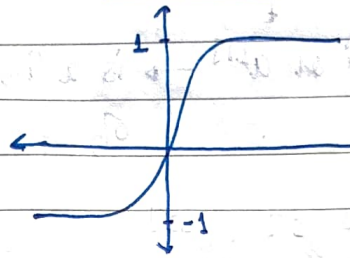
Date :

Activation functions

sigmoid func might not be best

→ tanh func always better

tanh → mainly use for hidden layer



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

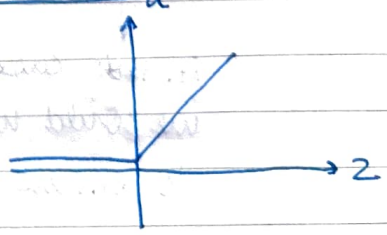
sigmoid → o/p layer to get $\hat{y} \in \{0,1\}$ rather than b/w $\{-1,1\}$ in binary classification

better ∵ data centered → mean about 0 making learning for nxt layer easy

downside of both → z is >> or << then gradient / derivative of slope ≈ 0 → slows down gradient descent

Instead what's very popular is **ReLU**

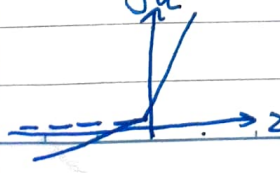
$a = \max(0, z)$



dis-adv ← derivative / slope = 0 when z = -ve

slope = 1 when z = +ve

hence leaky ReLU → $a = \max(0.01z, z)$



Date :

Why do neural network need Non-linear Activation func.

if in forward prop we did

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

$\hat{y}$ or $a^{[1]}$ → is a linear func of i/p

&

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

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

where

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

doing so makes the model no more complex & expressive than a normal logistic func

linear activation func.

can be used in regression problem where

- $y \in \mathbb{R}$ eg. housing prices

here too it's only used in the o/p layer & hidden layers use non-linear funcs

in this case too we could use ReLu to ensure $\hat{y} \geq 0$

Date : _____

Back propagation

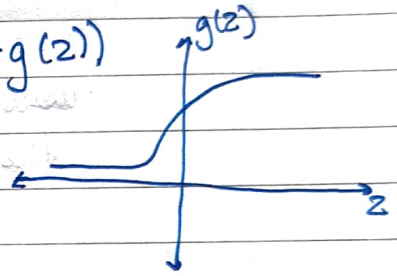
lets say for sigmoid func, $g(z) = \frac{1}{1+e^{-z}}$

we find derivative,

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

i.e

$$g'(z) = g(z)\,(1-g(z))$$



lets do a sanity check

say $z >> 0$

$\Rightarrow g(z) \approx 1$

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

for,

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

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

for ReLU,

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

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

for leaky ReLU ⟶ 0.01 if $z<0$

Date :

in a nn both forward & back propagation happen during training

→ uses i/p to make prediction
⇒ forward prop

→ using loss calc to compute error
↳ diff wrt w & b to get gradient

back propagation

used to optimize the params

uses chain-rule to calc gradients of the non-linear activation funcs

Gradient Descent

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

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

$n^{[0]}$: i/p feature
$n^{[1]}$: hidden units
$n^{[2]}$: o/p units → nodes

then,

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

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

Date :

for neural networks,
it's better to initialize params randomly rather than = 0

∴ gradient descent ⇒ repeat {

compute $\hat{y}^{(i)}$, $i = 1, 2 \ldots, m$

$dw^{[1]} = \frac{dJ}{dw^{[1]}}$

$db^{[1]} = \frac{dJ}{db^{[1]}}$

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

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

$w^{[2]} =$

$b^{[2]} =$

}

till convergence.

Forward

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

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

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

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

Back

$dZ^{[2]} = A^{[2]} - Y$ $= \begin{bmatrix} | & | \\ y^{(1)} & y^{(2)} \\ | & | \end{bmatrix}$ stacked horizontally

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

np.dot

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

to get (n, 1) not (n,)

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

done using np.multiply or * ← element-wise prod

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

Date :

→ for any var x , x & dx have same dims

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

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

$z^{[1]}, dz^{[1]} = (n^{[1]}, 1)$ ∵ → only 1 training eg

incase of 3layered nn,

its all eq^n is in case of a single training eg

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

$dz^{[2]}$ → $\frac{dL}{dz}$

$dW^{[1]} = dz^{[1]} x^T$ → it's basically $a^{[0]T}$

but we do it w/ multiple eg

⇒ need to do vectorization

→ while initializing weights we use a random val instead of 0

eg.

$W^{[1]} = np.random.randint((2,2)) \times 0.01$

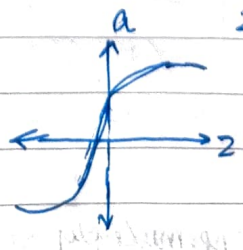
incase of bias it doesn't matt

2×2 matrix → 2 training eg

× 0.01 & not large no.

∵ on calc activation va $z^{[1]}$ turns to be very bi

⇒ we end at flat parts in func causing gradient descent to be very slow



Date :

Deep Neural Network

let L = no. of layers

$n^{[l]}$ = no. of units in layer l

$\Rightarrow n^{[1]} = 5$ - 5 nodes in layer 1

& $n^{[0]} = n_x$ = no. of i/p node

↳ main i/p

$a^{[l]}$ = activation func in layer l = $g^{[l]}(z^{[l]})$

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

Forward Propagation in dnn

same as previously,

general eq^n,

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

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

$A^{[0]} = X$

$z^{[1]} = w^{[1]} x + b^{[1]}$ → np.dot

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

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

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

⋮

to do for all layers we will be using a for loop

→ also while doing propogation forward we must store value of $z^{[l]}$ as cache as it'll be helpfull in backpropagation

↳ as well as $w^{[l]}$, $b^{[l]}$ & $a^{[l-1]}$

Date :

Getting the dimensions right

for first layers in forward prop we have → normal implementation

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

i.e only 1 train eg

part of first layer

(nodes in layer, 1) = (no. of nodes, ip / no. of train eg / nodes in prev layer)

(no. of neuron in this layer, ip to this layer)

⇒ general formula,

$$A^{[l]} = Z^{[l]} = (n^{[l]}, 1)$$
$$W^{[l]} = (n^{[l]}, n^{[l-1]})$$
$$X^{[l]} = (n^{[l]}, 1) = b^{[l]}$$

basically $A^{[0]}$, $l = 0$

vectorized implementation matrix

∵ examples are stacked vertically the dims for Z, X change,

$$Z^{[l]} = (n^{[l]}, m)$$
$$X^{[l]} = (n^{[l]}, m) \rightarrow \text{train eg}$$
$$W^{[l]} = (n^{[l]}, n^{[l-1]})$$
$$b^{[l]} = (n^{[l]}, 1)$$

$(n^{[l]}, m)$ ← to but during '+' it gets duplicated / broadcasted

Date :

in back prop,

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

element-wise prod

$dw^{[l]} = \frac{1}{m} dz^{[l]} \cdot a^{[l-1]T}$ ⟶ vectorized form

dot prod

$db^{[l]} = dz^{[l]}$ ⟶ invectorized form,

$da^{[l-1]} = w^{[l]T} dz^{[l]}$ $\frac{1}{m}$ np.sum(dz[l], axis=1, dims)

↳ substituting $da^{[l-1]}$

$\Rightarrow dz^{[l]} = w^{[l+1]T} dz^{[l+1]} * g^{[l]'}(z^{[l]})$

&

$d\hat{y} = -\frac{y}{a} + \frac{(1-y)}{(1-a)}$ ⟶ ts not vectorized

or

$da^{[l]}$

Parameters & Hyper-parameters

params : $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]} \ldots$

hyperparams : α, #iteration, #L, #hidden unit, choice of activation func

nu. of

params that control W & b

applying dl is very empirical i.e trial & error-ish

idea → code → experiment → idea