WEEK 3

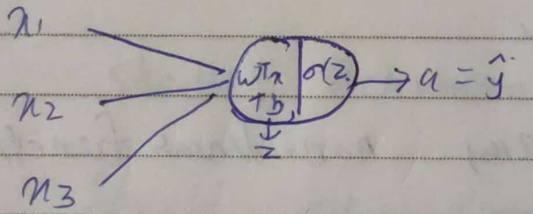
$X^{[i]} \rightarrow$ ~~Row~~ x in i^{th} layer. $[X]^{(i)} \rightarrow$ x in i^{th} training example

Neural network is basically multiple Logistic layers one after other.

Representation of NN

— repetitive, refer ML notes

computing a neural networks output.



every neuron performs 2 computations $\rightarrow z, a$.

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

Vectorizing across multiple training examples

using for loop on m training data

for $i = 1$ to $i = m$: //careful

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

horizontally → different training examples

colown wise → vertically → one neuron/feature of one training example

we combine all training example.

$$X = \begin{bmatrix} \vdots & \vdots & & \vdots \\ x^1 & x^2 & \cdots & x^m \\ \vdots & \vdots & & \vdots \end{bmatrix} \longrightarrow Z \text{ becomes } \begin{bmatrix} \vdots & \vdots & & \vdots \\ z^1 & z^2 & \cdots & z^m \\ \vdots & \vdots & & \vdots \end{bmatrix}$$

$n \times m$ matrix

W remains same across all training examples

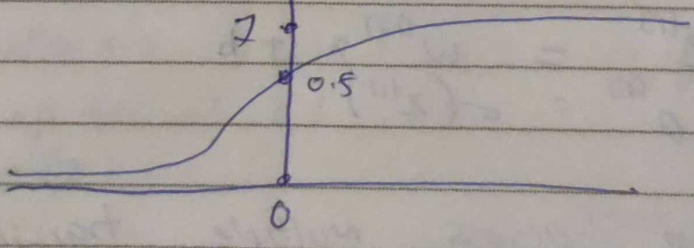
a becomes $a = \begin{bmatrix} \vdots & \vdots & & \vdots \\ a^1 & a^2 & \cdots & a^m \\ \vdots & \vdots & & \vdots \end{bmatrix}$

// same code, just remove for loop !!

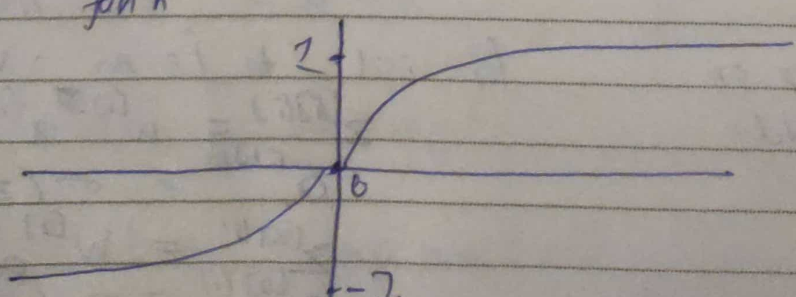
Activation functions
- → our choice.
- → can be any non-linear function.

eg:- sigmoid :- $\sigma(z) = \dfrac{1}{1+e^{-z}}$

b/w 0 and 1
↓
useful in binary classification



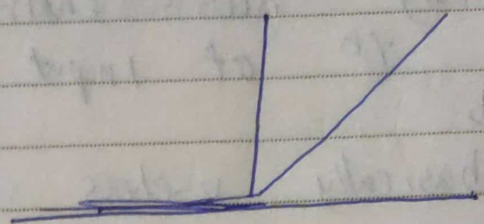
tanh



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

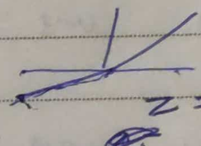
~~Relu~~ ReLU

$z = \max(0, z)$



tanh is superior

- use sigmoid only if you're binary classificat^n
- use ReLU mostly
- ReLU is becoming very common
- disadvantage of ReLU - derivative is 0 for -ve val
- Leaky ReLU: cons it →



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

can change this constant

Why should Activation f^n be Non-Linear?

Why should we even use Activation f^n?

lets not use $\xrightarrow[\text{to}]{\text{equivalent}}$ using a linear activation fn. (with slope 1.)

Our algo becomes:

Given x:

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

then

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

$a^{[2]} \rightarrow$ our output is a linear f^n of x!!

Most problems can't be solved by just a linear f^n

So basically our $\overbrace{\text{entire } f^n}^{\text{entire model}}$ is just a linear f^n of input

↓

its basically useless

If you put multiple layers but they're all linear, the final output will just be still a linear f^n of input

So all the middle layers were useless

So we need to induce non-linearity

[you can use Linear f^n for Regression problems btw] → only in final output layers.

Derivatives of Activation function

$$\frac{d\,\sigma(z)}{dz} = \sigma(z)\left[1-\sigma(z)\right]$$

for large $(z) \longrightarrow \sigma(z) \approx 1$

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

for small (z) $\quad \sigma(z) \approx 0$

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

for $z=0 \rightarrow \sigma(z) = 1/2$

$$\frac{d}{dz}\sigma(z) = \frac{1}{4}$$

$$\frac{d}{dz}\tanh(z) = 1 - (\tanh(z))^2$$

for $z = 10$ $\quad \tanh(z) \approx 1$
derivative ≈ 0

for $z = -10$ $\quad \tanh(z) \approx -1$
derivative ≈ 0

for $z = 0$ $\quad \tanh(z) \approx 0$
derivative ≈ 1

$$\frac{d}{dz}ReLU(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z > 0 \\ \text{undefined} & \text{if } z = 0 \end{cases}$$

$$\frac{d}{dz}(\text{Leaky ReLU}(z)) = \begin{cases} 0.01 & z < 0 \\ 1 & z > 0 \\ \text{undefined} & z = 0 \end{cases}$$

(undefined for $z = 0$ doesn't matter bcz derivative almost never become exactly zero)

Gradient Descent for Neural Networks

example taken is of 1 layer
so basically same as logistic regression

Forward + backward propagation for layer l

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

forward prop

$$z = w \cdot a^{T} + b$$
$$a = \text{sigmoid}(z)$$

using vectorizatⁿ

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

Backward part
Input: $da^{[l]}$

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

basically $d(\text{param}) = g'(z)\, d(\text{for output})$

$$dz^{[l]} = da^{[l]} \times g^{[l]\prime}(z^{[l]})$$
$$dw^{[l]} = dz^{[l]} \cdot a^{[l-1]}$$
$$db^{[l]} = dz^{[l]}$$
$$da^{[l-1]} = W^{[l]T} \cdot dz^{[l]}$$

comes from partial derivations

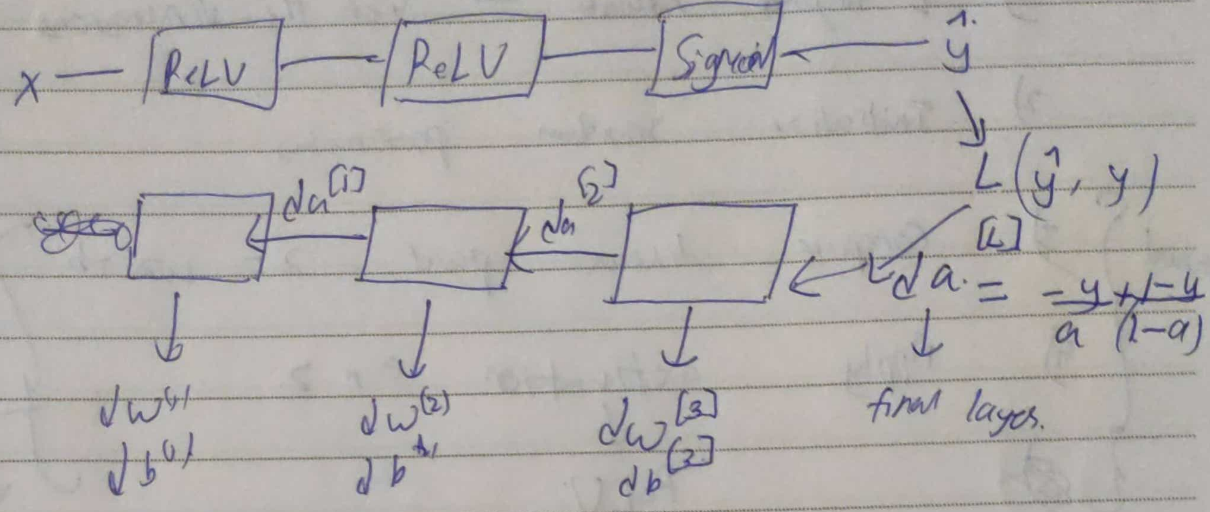
$$dz^{[l]} = W^{[l+1]T} * g^{[l]\prime}(z^{[l]})$$

~~$dz^{[l]} = d$~~

Same for vectorize v.

DOMS

Summarising :-



The eqn's of backward prop can be generalised — for multiple layers. So learn them

Hyperparameters vs parameters → w, b

↳ eg :- learning rate α
- no. of iterations
- no. of hidden layers L
- no. of hidden units
- Choice of activation function

they control the actual parameters w and b

other hyperparameters :- Momentum.
minibatch size.
regularizations ...

1) L layered network — Get the dimensions right.

2) Initialize random parameters

forward

3) Compute linear part $z = wx + b$

4) Apply activation on z.

↓

ReLU

(sigmoid is last

repeat L-1 times

5) compute loss

→ you have the function.

6) complete linear part of backward prop

7) ~~gd~~ combine linear and activation parts

descent

8) Stack backward prop combinations.