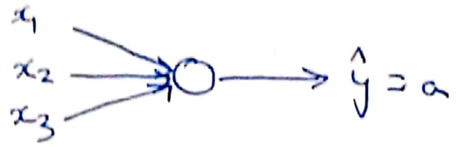
Week 3 : One Hidden Layer NN

Overview

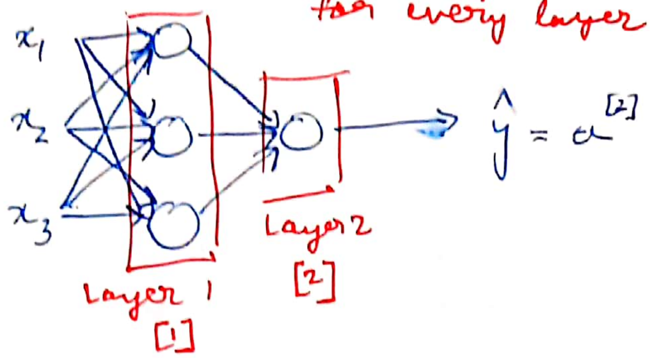
Single Neuron :



x, w, b

$$\underbrace{\boxed{z = w^T x + b} \rightarrow \boxed{a = \sigma(z)}}_{\text{These 2 steps are required for every layer}} \rightarrow \boxed{\mathcal{L}(a,y)}$$

Simple NN :

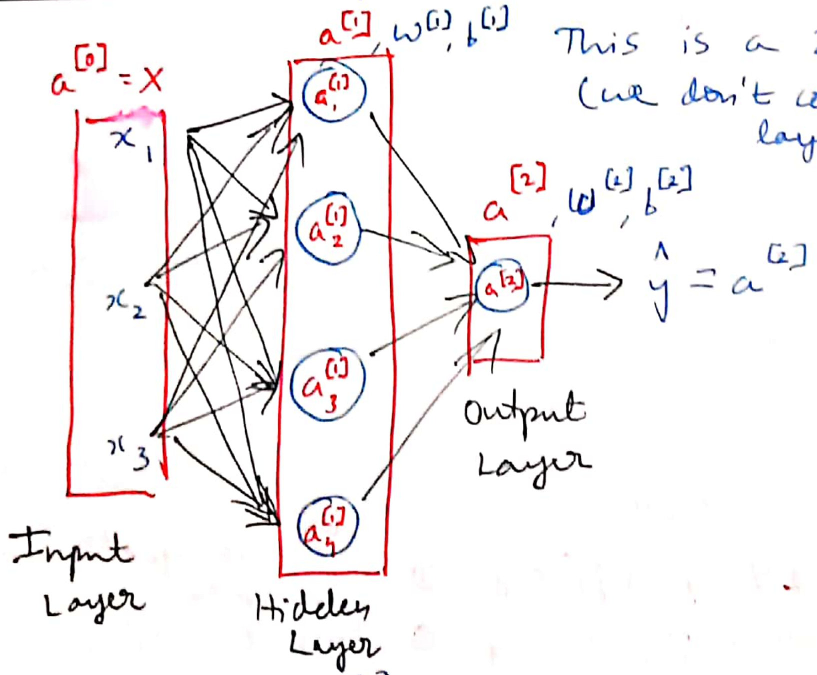


x, $W^{[1]}$, $b^{[1]}$; $W^{[2]}$, $b^{[2]}$

$$\underbrace{\boxed{z^{[1]} = W^{[1]}x + b^{[1]}} \rightarrow \boxed{a^{[1]} = \sigma(z^{[1]})}}_{\text{Layer 1}} \rightarrow \underbrace{\boxed{z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}} \rightarrow \boxed{a^{[2]} = \sigma(z^{[2]})}}_{\text{Layer 2}} \rightarrow \boxed{\mathcal{L}(a^{[2]}, y)}$$

Neural Network Representation

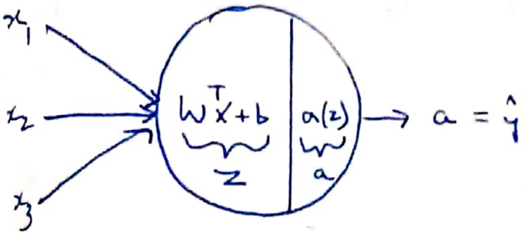
This is a 2 layer NN (we don't count the input layer)



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

Scanned with CamScanner

Computing a Neural Network's Output

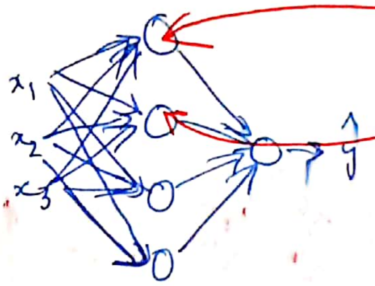


logistic regression represents 2 steps:

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

$\rightarrow a = \sigma(z)$

So a NN does this a lot of times



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

$a_1^{[1]} = \sigma(z_1^{[1]})$ ← node

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

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

$\therefore$ Finding equations for all the nodes,

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

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

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

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

$$z^{[1]} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix} \quad w^{[1]} = \begin{bmatrix} - w_1^{[1]T} - \\ - w_2^{[1]T} - \\ - w_3^{[1]T} - \\ - w_4^{[1]T} - \end{bmatrix} \quad x = a^0 = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad b^{[1]} = \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix} \quad a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix}$$

Given input x,

$z^{[1]} = w^{[1]} \cancel{x}\, a^{[0]} + b^{[1]}$ ← since $x = a^{[0]}$

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

$z^{[2]} = w^{[2]} a^{[1]} + b^{[2]}$ ← we are not doing w^T since in $w^{[2]}$ we have w^T

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

Vectorising across Multiple Examples

In the previous section, we saw how to compute the prediction on a neural network for a single training example. In this section, we will vectorize across multiple training examples.

In looping it would be as follows,

for $i = 1$ to m:

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

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

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

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

where $z^{[1](i)}$

layer

training example i

Consider we stack up these vectors,

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

For intuition

$$x = \begin{bmatrix} x_1^{(1)} & x_1^{(2)} & & x_1^{(m)} \\ x_2^{(1)} & x_2^{(2)} & \cdots & x_2^{(m)} \\ x_3^{(1)} & x_3^{(2)} & & x_3^{(m)} \end{bmatrix}$$

nodes

1 ... 3

Training examples

1 .... m

In the previous section we only considered one training example so took $\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$

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

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

$\therefore$ For multiple examples,

$$Z^{[1]} = W^{[1]}\underbrace{X}_{A^{[0]}} + b^{[1]}$$

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

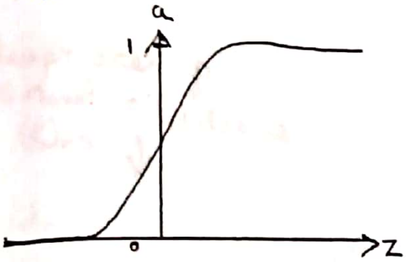
$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$

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

Activation Functions

Till now we have been using the sigmoid function σ. However there are more superior options as well.

Sigmoid



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

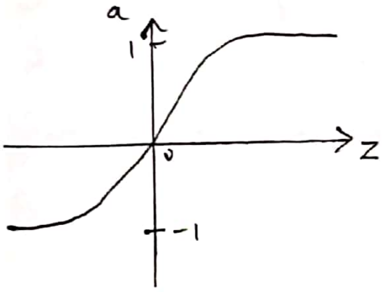
- Gives value between 0 to 1 so only use this for the output ~~function~~ layer

For Output Layer

(If binary classification $y = \{0, 1\}$)

- Worst option to use for hidden layers

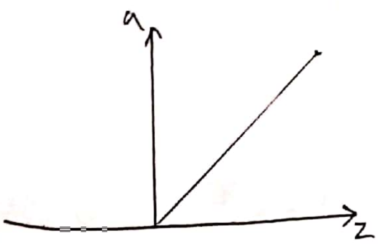
tanh



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

- A better option than sigmoid, but still sucks
- Give values between -1 to 1
- 0 centred
- Don't use lol

ReLU (Rectified Linear Unit)



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

- The perfect function for hidden layers

For hidden layers

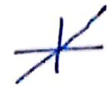
$$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$

Leaky ReLU

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

- If neurons start dying (giving 0 value for everything), then use this

Why can't we use linear activation functions?

Linear function - slope is constant 

Non-linear function - slope varies 

- The activation function has to be a non-linear function (Exa: σ, tanh, ReLU)
- If it is linear, lets say

g here represents an activation function like $\sigma(z)$

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

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

Then the output ~~will be~~ of the NN will be $w'x + b' \Rightarrow (\underbrace{W^{[2]}W^{[1]}}_{w'})x + (\underbrace{W^{[2]}b^{[1]} + b^{[2]}}_{b'})$

Therefore having hidden layers will become pointless cus they won't work

$\therefore$ Only use Non-linear activation functions.

Derivatives of Activation Functions

① Sigmoid Function:

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

since $a = g(z)$

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

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

② Tanh Function:

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

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

③ ReLU Function:

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

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

according to maths undefined if $z = 0$

But in coding, 1 if $z \geq 0$

in software it works

④ Leaky ReLU Function:

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

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

if coding

Gradient Descent for NN

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

Cost Function: $J(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y} - y)$ $\hat{y} = a^{[2]}$

Gradient Descent:

Repeat {

Compute prediction $(\hat{y}^{(i)}, i = 1 \dots m)$

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

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

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

...

}

Formulas for computing derivatives:

Forward propogation:

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

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

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

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

Here for classification $= \sigma(Z^{[2]})$

Backward propogation:

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

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

Make the matrix $(n^{[2]}, 1)$ instead of $(n^{[2]},)$

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

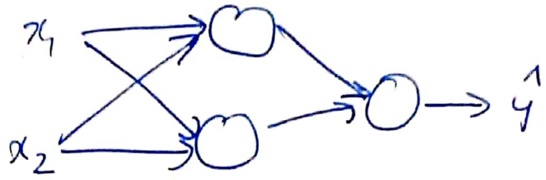
$$dZ^{[1]} = W^{[2]T} dZ^{[2]} * g^{[1]\prime}(Z^{[1]})$$

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

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

Random Initialization:

If you set W to 0 ie.



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

Then $dW = \begin{bmatrix}u & u\\ u & u\end{bmatrix}$ — The changes of all the neurons in the layers will end up being same.

∴ Therefore W values will always be the same and the NN will become symmetrical. Therefore having n no. of nodes won't matter since it'll be equivalent to having 1 node (since all values same).

We use random initializing to break the symmetry:

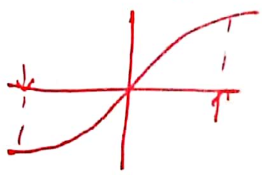
$W^{[1]}$ = np.random.randn((2,2)) * 0.01

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

$W^{[2]}$ = ...

$b^{[2]}$ = 0

This needs to be a small number so that $a^{[1]} = g^{[1]}(z^{[1]})$



The value of $a^{[1]}$ doesn't end up here