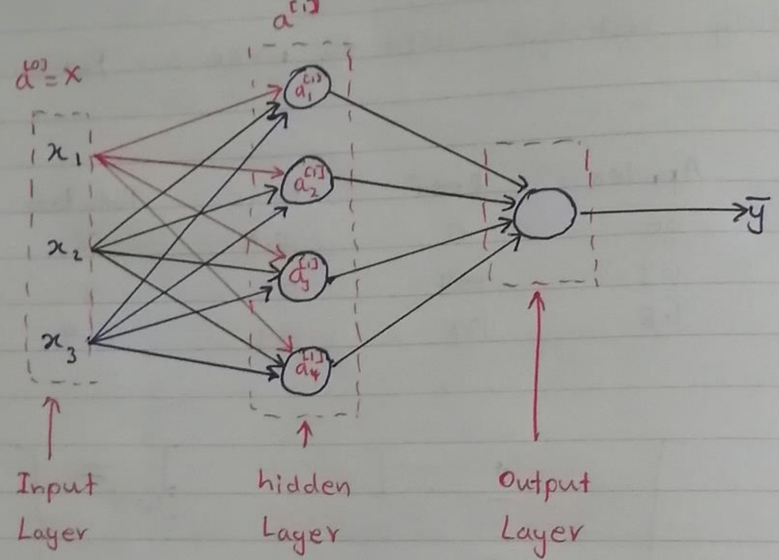
Week 3

Shallow Neural Networks

Neural Networks Overview

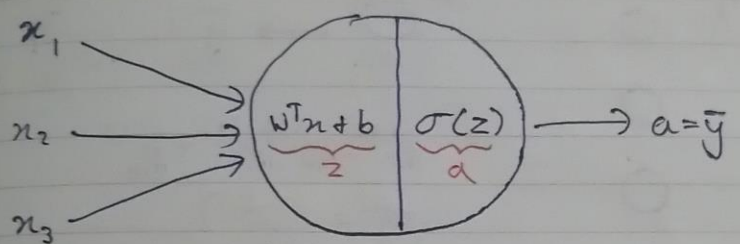


* When we count number of layers in neural network, we don't count input layer. Therefor, above network has only 2 Layers (hidden layer and Output layer)

* In the ~~traini~~ training set, the true values for nodes in hidden layer are not observed. It means you don't see what should be in the training set. that is the meaning of hidden layer.

Computing a Neural Network's Output

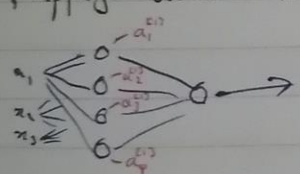
* Let's look in to one node of hidden layer and see what happed in that node



$z = w^Tx+b$

$a = \sigma(z)$

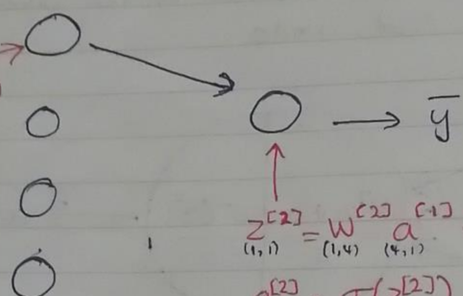
* Now, apply above theory into ~~to~~ our previous nural network



$$z_1^{[1]} = w_1^{[1]T}x + b_1^{[1]} \quad , \quad a_1^{[1]} = \sigma(z_1^{[1]})$$
$$z_2^{[1]} = w_2^{[1]T}x + b_2^{[1]} \quad , \quad a_2^{[1]} = \sigma(z_2^{[1]})$$
$$z_3^{[1]} = w_3^{[1]T}x + b_3^{[1]} \quad , \quad a_3^{[1]} = \sigma(z_3^{[1]})$$
$$z_4^{[1]} = w_4^{[1]T}x + b_4^{[1]} \quad , \quad a_4^{[1]} = \sigma(z_4^{[1]})$$

$$\underbrace{\begin{bmatrix} - w_1^{[1]T} - \\ - w_2^{[1]T} - \\ - w_3^{[1]T} - \\ - w_4^{[1]T} - \end{bmatrix}}_{(4\times 3)} * \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}}_{(3\times 1)} + \underbrace{\begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix}}_{(4\times 1)} = \begin{bmatrix} w_1^{[1]T}x + b_1^{[1]} \\ w_2^{[1]T}x + b_2^{[1]} \\ w_3^{[1]T}x + b_3^{[1]} \\ w_4^{[1]T}x + b_4^{[1]} \end{bmatrix} = \underbrace{\begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix}}_{(4\times 1)}$$

* After ~~abov~~ ~~above~~ that we are going to discuss what happen ~~to~~ in output layer.



$\underset{(4,1)}{z^{[1]}} = \underset{(4,3)}{W^{[1]}} \underset{(3,1)}{x} + \underset{(4,1)}{b^{[1]}}$

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

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

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

Vectorizing Across Multipd Examples

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

$x^{(1)} \longrightarrow a^{[2](1)} = \bar{y}^{(1)}$

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

$\vdots$

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

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

Layer 2

for i=1 to m:

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

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

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

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

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

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

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

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

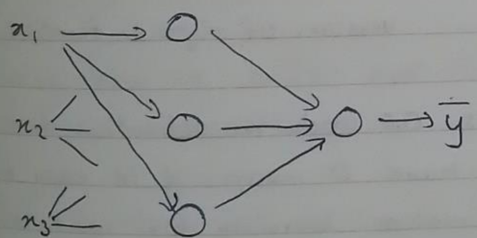
$$X = \begin{bmatrix} | & | & & | \\ x^{[1]} & x^{[2]} & \cdots & x^{[m]} \\ | & | & & | \end{bmatrix}_{(n_x, m)} \qquad Z = \begin{bmatrix} | & | & & | \\ z^{[1](1)} & z^{[1](2)} & \cdots & z^{[1](m)} \\ | & | & & | \end{bmatrix}$$

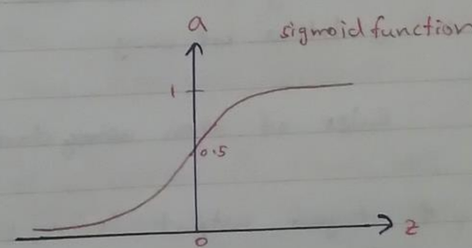
$$A^{[1]} = \begin{bmatrix} | & | & & | \\ a^{[1](1)} & a^{[1](2)} & \cdots & a^{[1](m)} \\ | & | & & | \end{bmatrix}$$

$\longleftrightarrow$ Total examples

$\updownarrow$ hidden unites

Activation Function





Given x:

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

$$\left.\begin{array}{l} \to a^{[1]} = \sigma(z^{[1]}) \\ \quad z^{[2]} = W^{[2]} a^{[1]} + b^{[2]} \\ \to a^{[2]} = \sigma(z^{[2]}) \end{array}\right.$$

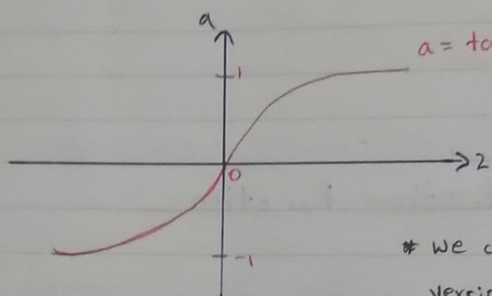
sigmoid function ~~activation function~~

* But in geraral cases we can use different funtion called "g" of z . g could be a non linear function that may not be the sigmoid function

$a^{[1]} = g(z^{[1]})$
$a^{[2]} = g(z^{[2]})$

* Activation function that almost always better than the sigmoid function. (tan h function or hyperbolic tangh function)

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



* we can tell this graph is lifted version of sigmoid function.

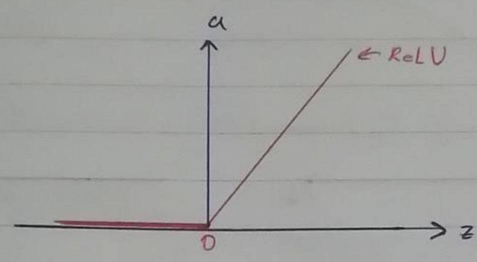
* In sigmoid function's, mean is 0.5 and in activation function's mean is 0. If your data have 0 mean it is easy to centering your data and it makes learning for nenf layer a little bit easy.

Rules of when ~~using~~ choosing activation function

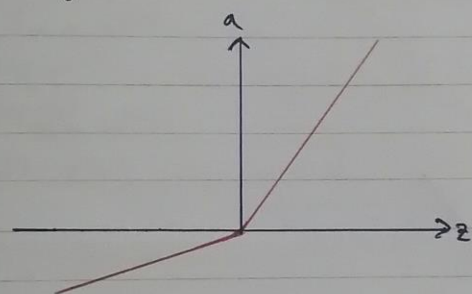
* if your output is 0, 1 ~~or if yo~~ (if you using binary classification) then the sigmoid activation function is the most suitable choice for the output layer.

* If you are not sure what to use for hidden layer, Use the ReLU activation function

* But one disadvantage of ReLU is that the derivative is equal to 0.



* When z is a negative we can use "Leaky ReLU".



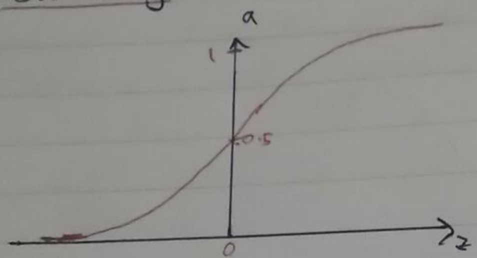
* Leaky Relu is better ~~the~~ than Relu. But we didn't use Leaky as much in practice.

* But advantage of ReLU and Leaky ReLU is that for a lot of the space of z, the derivative of the activation function, the slope of the activation function is very different from 0.

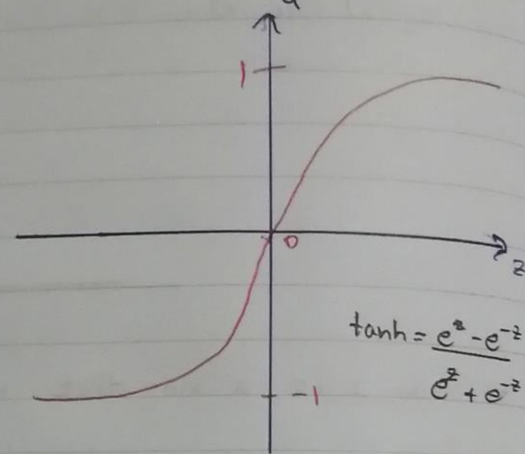
* So, ~~I~~ in practice using the ReLU ~~at~~ activation function, ~~with~~ your neural network will often learn faster than when using tanh or the sigmoid activation function.

* tanh is the best to use

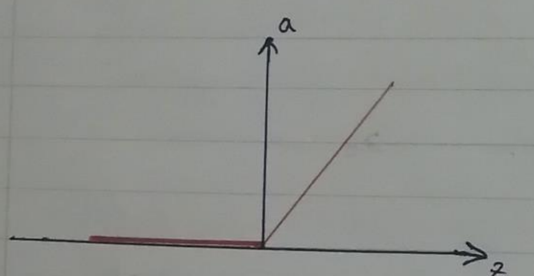
Summary



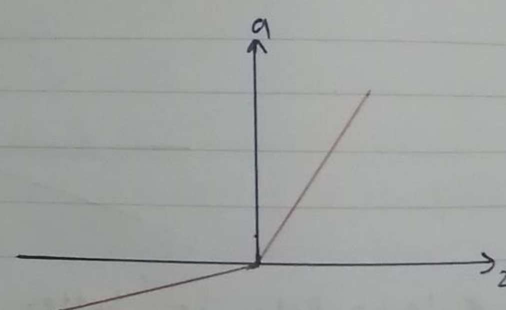
sigmoid $\rightarrow a = \dfrac{1}{1+e^{-z}}$



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

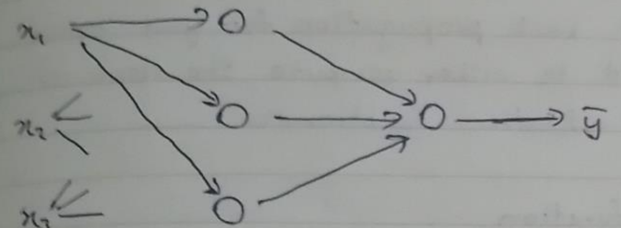


ReLU $\rightarrow a = \max(0, z)$



Leaky ReLU $\rightarrow a = \max(0.01z, z)$

Why do we need Non-Linear Activation Functions?



Given x: Assume $g(z) = z$

for hidden layer
$$z^{[1]} = W^{[1]}x + b^{[1]}$$
$$a^{[1]} = g^{[1]}(z^{[1]}) \longrightarrow \therefore a^{[1]} = z^{[1]}$$

for output layer
$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$
$$a^{[2]} = g^{[2]}(z^{[2]}) \longrightarrow \therefore a^{[2]} = z^{[2]}$$

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

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

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

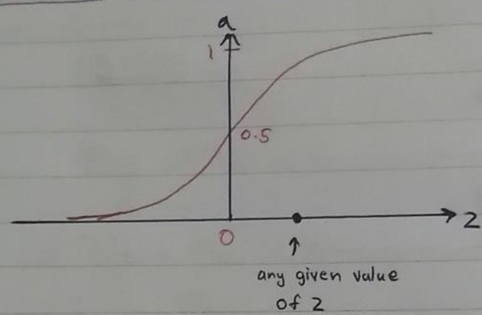
$$a^{[2]} = W'x + b'$$ ← This means neural network outputting linear function of input.

* In above ~~prove~~ calculation we can see if we don't have ~~[illegible]~~ activation function then no matter how many layers our neural network because all layers doing ~~linear~~ is just computing linear activation function. So you might not have any hidden layers.

Derivatives of Activation Function

* When you implement back propagation for your neural network, you need to either compute the slope or derivative of the activation functions.

Sigmoid activation function



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

* On above, given value of z will have some slope or derivative
* ~~Slo~~ We can define that slope as

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

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

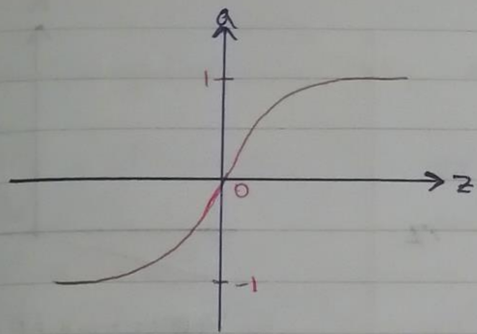
$$= g(z)\,(1 - g(z))$$

if z is very large ($z=10$) then $g(z) \approx 1$ and $\frac{d}{dz} g(z) \approx 0$

if z is negative ($z=-10$) then $g(z) \approx 0$ and $\frac{d}{dz} g(z) \approx 0$

if z is equal to 0 then $g(z) = \frac{1}{2}$ (0.5) and $\frac{d}{dz} g(z) = \frac{1}{2}\left(1 - \frac{1}{2}\right) = \frac{1}{4}$

<u>Tanh activation function</u>



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

* we can define slope as

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

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

if ~~z=0~~ $z = 10$, then $\tanh(z) \approx 1$

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

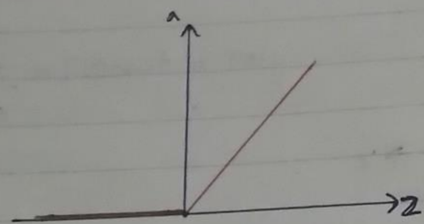
if $z = -10$ then $\tanh(z) \approx -1$

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

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

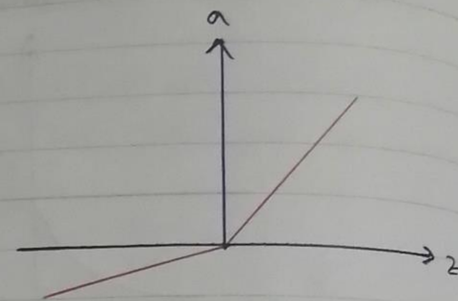
$$\frac{d}{dz} g(z) = 1$$

ReLU and Leaky ReLU



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

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



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

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

Gradient Descent for Neural Networks

parameters :- $\underset{(n^{[1]}, n^{[2]})}{w^{[1]}}, \underset{(n^{[1]}, 1)}{b^{[1]}}, \underset{(n^{[2]}, n^{[1]})}{w^{[2]}}, \underset{(n^{[2]}, 1)}{b^{[2]}}$

$n_x = \underset{\text{input features}}{n^{[0]}}, \underset{\text{hidden units}}{n^{[1]}}, \underset{\text{output units}}{n^{[2]} = 1}$

cost function :- $J(w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}) = \frac{1}{m} \sum_{i=1}^{m} L(\underset{a^{[2]}}{\bar{y}}, y)$

* To train parameters, of out Algorithm, we have to perform gradient descent.

Formulas for computing derivatives

Forward Propagation

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

Backward Propagation

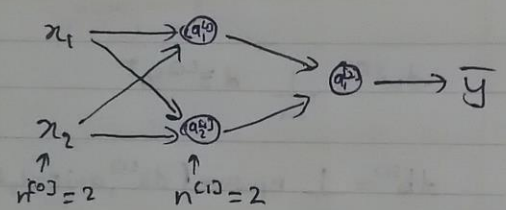
$$dz^{[2]} = A^{[2]} - Y \qquad Y = [y^{[1]}, y^{[2]} \dots y^{[m]}]$$
$$dw^{[2]} = \frac{1}{m} dz^{[2]} A^{[1]T}$$
$$db^{[2]} = \frac{1}{m} \text{np.sum}(dz^{[2]}, \text{axis}=1, \text{keepdims}=\text{True})$$
$$dz^{[1]} = \underbrace{W^{[2]T} dz^{[2]}}_{(n^{[1]}, m)} \underset{\substack{\uparrow \\ \text{elementwise} \\ \text{Product}}}{*} \underbrace{g^{[1]\prime}(z^{[1]})}_{(n^{[1]}, m)}$$
$$dw^{[1]} = \frac{1}{m} dz^{[1]} X^T$$
$$db^{[1]} = \frac{1}{m} \text{np.sum}(dz^{[1]}, \text{axis}=1, \text{keepdims}=\text{True})$$

Randon Initialization

* When you change your neural network, it is important to initialize the weights randomdy.

* For Logistic regression, It is okey to initialize the weights to 0. But for a neural network, initialize weights and parameters to 0 and then applied gradient descent doesn't work much.

What happens ~~to~~ if you initialize weights to zero?



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

* Initializing bias (b) to 0 is actually okey but intializing w to 0 is a problem.

* If you initialized w to 0 then $a_1^{[1]} = a_2^{[1]}$. It mean both of hidden nodes in ~~hidden~~ hidden Layers are computing exactly same function

* And then, when you compute backpropagation, $dz_1^{[1]} = dz_2^{[1]}$. it is also computing same function