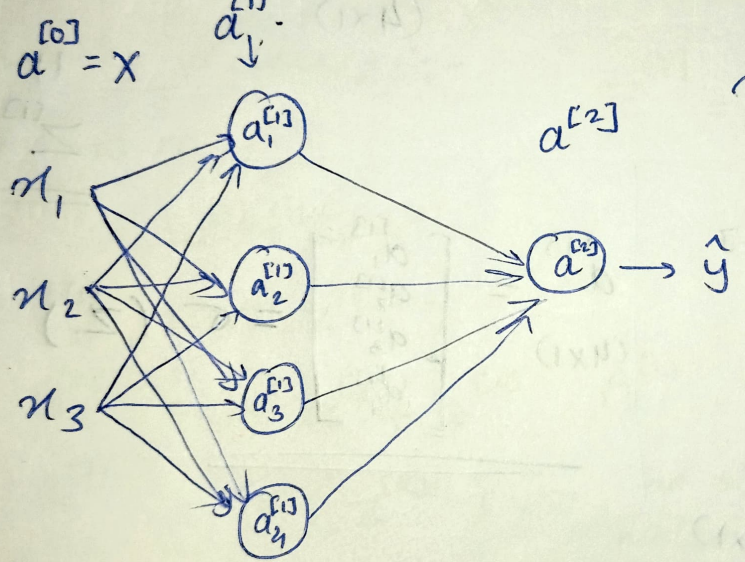
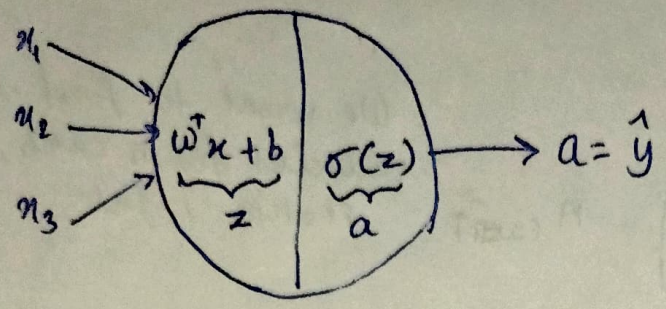
Week 3

☆ Neural Network Representation.



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

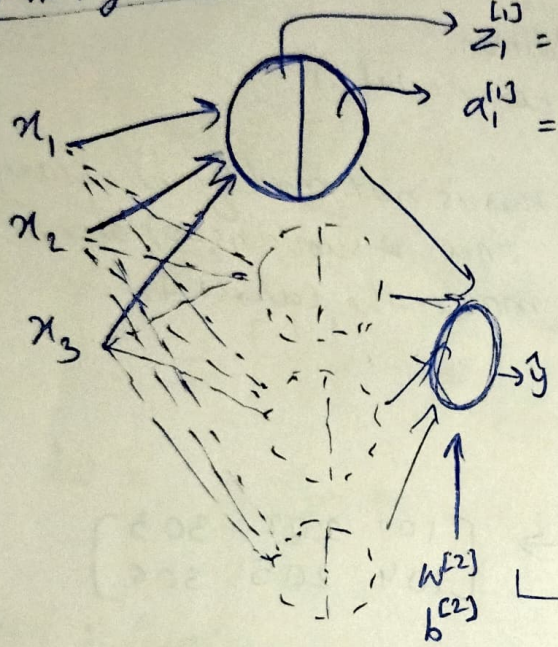
$$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix}$$



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

$$a = \sigma(z)$$

Logistic Regression



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

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

$\Rightarrow a_i^{[l]}$ → layer, → node in layer.

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

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

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

$$\begin{bmatrix} — w_1^{[1]T} — \\ — w_2^{[1]T} — \\ — w_3^{[1]T} — \\ — w_4^{[1]T} — \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix} = \begin{bmatrix} w_1^{[1]T} x + b_1^{[1]} \\ w_2^{[1]T} x + b_2^{[1]} \\ w_3^{[1]T} x + b_3^{[1]} \\ w_4^{[1]T} x + b_4^{[1]} \end{bmatrix}$$

(4×3) (3×1) (4×1) (4×1)

because there are 3 inputs.

$$= z^{[1]}$$

Output layer

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

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

$W^{[2]}$ → (1×4)

$b^{[2]}$ → (1×1)

$z^{[2]} = (1\times1)$

$$a^{[1]}_{(4\times1)} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix} = \sigma(z^{[1]})$$

∴ In summary :- Given Input x:

four lines of code

$\rightarrow z^{[1]}_{(4\times1)} = W^{[1]}_{(4\times3)} a^{[0]}_{(3\times1)} + b^{[1]}_{(4\times1)}$

$\rightarrow a^{[1]}_{(4\times1)} = \sigma(z^{[1]}_{(4\times1)})$

$\rightarrow z^{[2]}_{(1\times1)} = W^{[2]}_{(1\times4)} a^{[1]}_{(4\times1)} + b^{[2]}_{(1\times1)}$

$\rightarrow a^{[2]}_{(1\times1)} = \sigma(z^{[2]}) \longrightarrow \hat{y}$

This computes the result for 1 training example.
∴ $\hat{y}$ here is for the 1st training example.

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

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

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

$\vdots$

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

$a^{[2](m)}$: layer 2; (m): example no.

m training examples

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

no. of features = no. of nodes in input layer

no. of examples

We want to vectorize :-

for $i = 1$ to m:

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

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

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

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

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

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

no. of examples

no. of nodes in the layer.

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

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

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

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

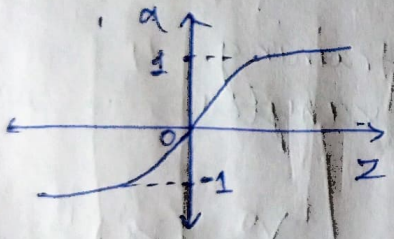
★ Activation functions.

$a = \sigma(z)$

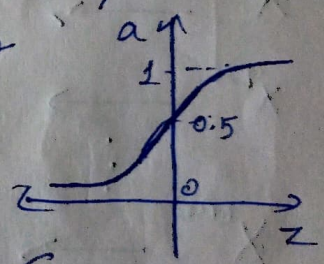
Sigmoid ↙ ↘ this is an activation function → this would be different

In fact, $\tanh(z^{[1]})$ function works better than $\sigma(z^{[1]})$.

↓ for 1st hidden layer or any hidden layer except maybe the ~~last~~ output layer.



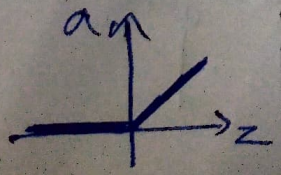
↳ $-1 \leq a \leq 1$



↳ $0 \leq a \leq 1$

→ Problem for both of them is that as the value of z gets really high or really low, the slope tends to zero and the gradient descent ~~&~~ starts getting slower and slower.

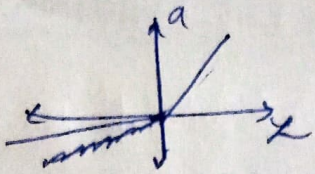
Hence we use ~~Relu~~ ReLU function (Rectified Linear unit)



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

↳ used most commonly.

Leaky ReLU →



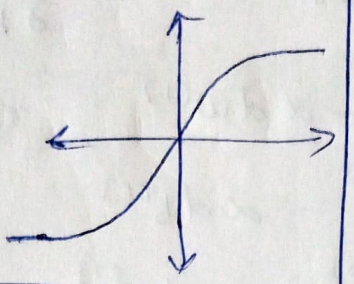
-ve z values do not lead to 'zero' a.

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

* Slopes/derivatives.

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

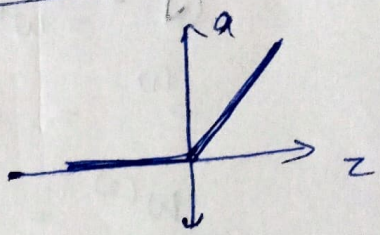
$= \frac{e^z - e^{-z}}{e^z + e^{-z}}$



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

↳ $g'(z) = 1 - a^2$

ReLU

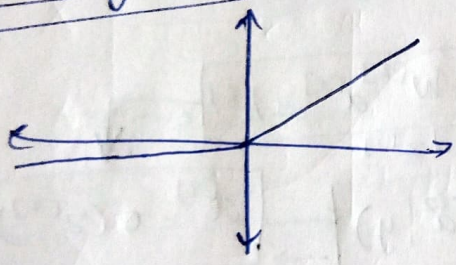


$g(z) = \max(0, z)$

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

↳ $z = 0.000\cdots01$

Leaky ReLU



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

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

* Gradient Descents for neutral networks.

Parameters: $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}$; $n_x = n^{[0]}$, $n^{[1]}$, $n^{[2]} = 1$

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

↑ input features, ↑ hidden units, ↑ output units.

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

↳ $a^{[2]}$

Gradient Descent:

Repeat {

Compute predictions $(\hat{y}^{(1)}, \dots, \hat{y}^{(m)})$

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

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

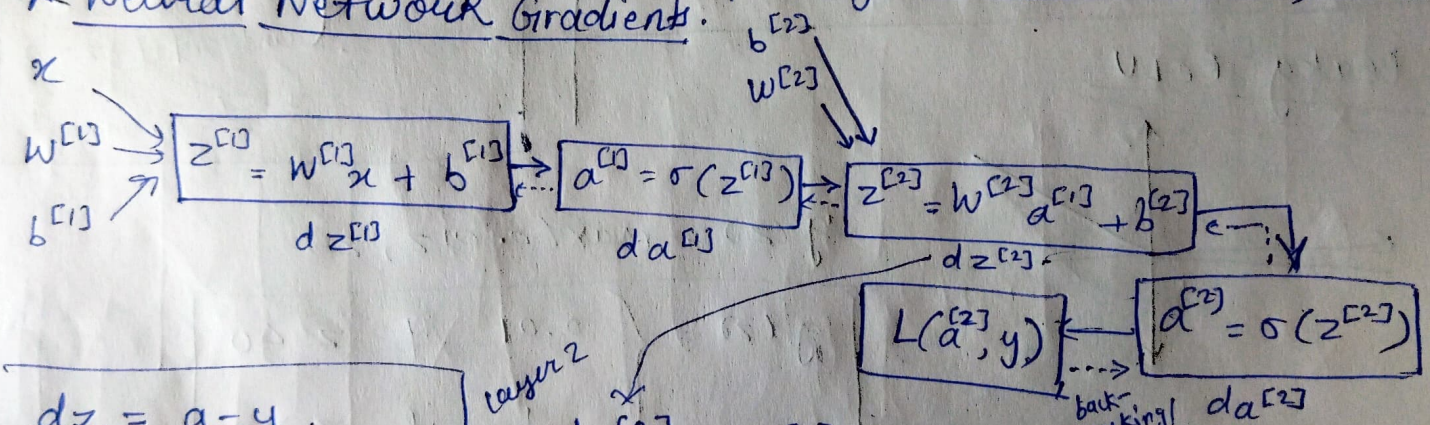
$b^{[1]} = b^{[1]} - \alpha\, db^{[1]}$

$w^{[2]} = w^{[2]} - \alpha\, dw^{[2]}$

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

}

★ Neural Network Gradient. (backpropagation derivation)



$dz = a - y$.

$\frac{dL}{dz} = \frac{\partial L}{\partial a} \cdot \frac{\partial a}{dz}$.

$dz = da \cdot \frac{d\,g(z)}{dz}$

$dz = da \cdot g'(z)$

layer 2

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

$dw^{[2]} = dz^{[2]} \cdot a^{[1]T}$ → similar to "$dw = dz \cdot x$" (replaced by $a^{[1]T}$)

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

layer 1

$dz^{[1]} = w^{[2]T} dz^{[2]} * g^{[1]'}(z^{[1]})$ (?)

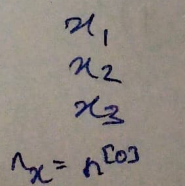
$(n^{[1]} \times 1)$ $(n^{[1]} \times n^{[2]})$ $(n^{[2]} \times 1)$ $(n^{[1]} \times 1)$

element-wise product

$dw^{[1]} = dz^{[1]} \cdot x^T$

$db^{[1]} = dz^{[1]}$ { similar to layer 2 values }

Dimensions.



$w^{[2]} = (n^{[2]} \times n^{[1]})$

$z^{[2]}, dz^{[2]} = (n^{[2]} \times 1)$ { in this case (1,1) }

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

Summary of Gradient Descent :

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

$$dw^{[2]} = dz^{[2]} \cdot a^{[1]T}$$

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

$$dz^{[1]} = w^{[2]T} dz^{[2]} * g^{[1]'}(z^{[1]})$$

$$dw^{[1]} = dz^{[1]} \cdot x^{T}$$

$$db^{[1]} = dz^{[1]}$$

Vectorized implementation (for 'm' no. of examples).

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

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

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

↑ avoids something like (m,) and instead writes (m,1)

$$dZ^{[1]} = \underbrace{W^{[2]T} dZ^{[2]}}_{(n^{[1]}, m)} * \underbrace{g^{[1]'}(Z^{[1]})}_{(n^{[1]}, m)}$$

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

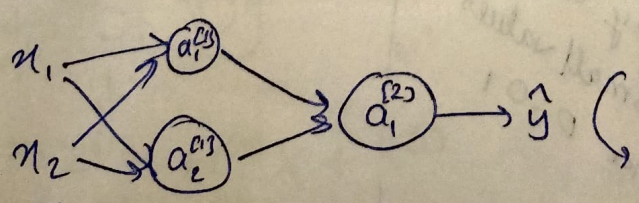
$$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^{T}$$

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

---

* Why we should randomly initialize parameters instead of initializing with zero?

↳



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

$$a_1^{[1]} = a_2^{[1]}$$

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

both nodes will have same values ultimately because they have the same functions

So, both the nodes are exactly symmetrical, they are computing exactly the same thing.

↳ due to this :- $dw = \begin{bmatrix} u & v \\ u & v \end{bmatrix}$ → all rows end up exactly the same.

after every gradient descent : $W^{[1]} = W^{[1]} - \alpha dw$

↳ $W^{[1]} = \begin{bmatrix} — & — \\ — & — \end{bmatrix}$ all rows are exactly same.

So, even after training it many times they will still compute the same thing.

This is why we do random initialization.

Hence,

$$\begin{cases} W^{[1]} = np.random.randn((2,2)) * 0.01 \\ b^{[1]} = np.zero((2,1)) \end{cases}$$

we prefer very small values why?

→ this can remain zero because as long as $W^{[1]}$ is randomized, $dz^{[1]}$ will be unique.

Same as this

$$\begin{cases} W^{[2]} = \dots \\ b^{[2]} = 0 \end{cases}$$

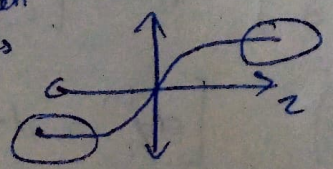
0.01 → we prefer small values.

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

↕ ↓

this will ← if this is big.
be big

(then



at big z values, slope ≈ 0, no/very slow gradient descent.

↑ that's why we multiply by small values like 0.01