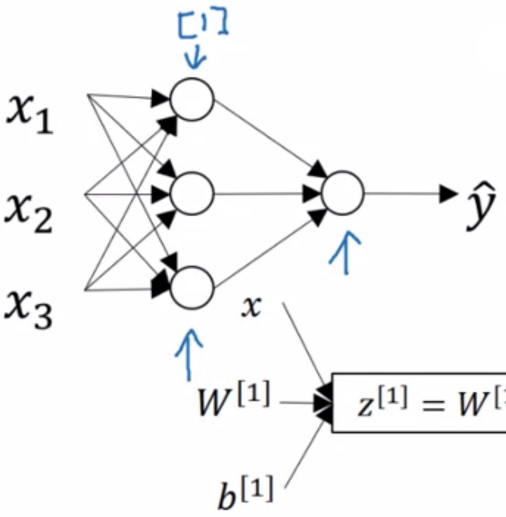
Neural Network

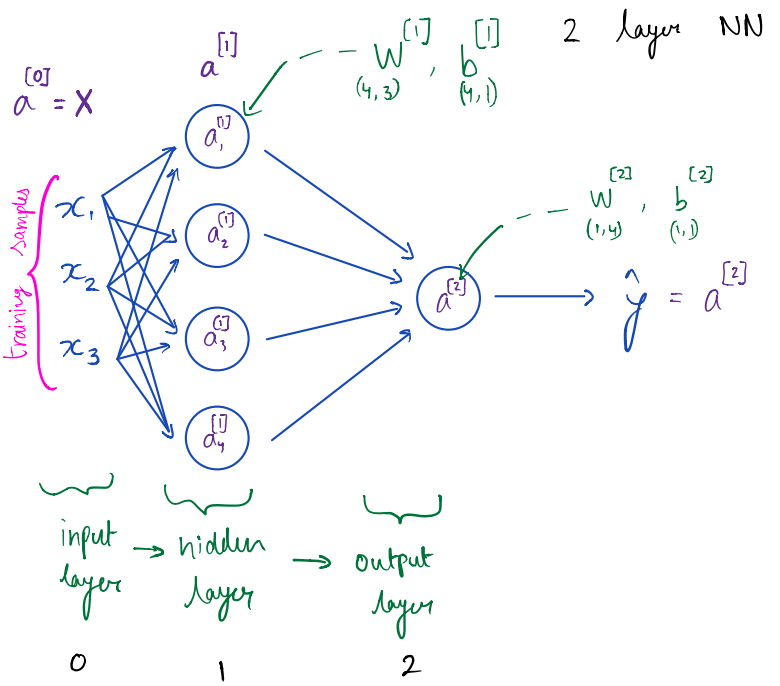


index for the NN layer

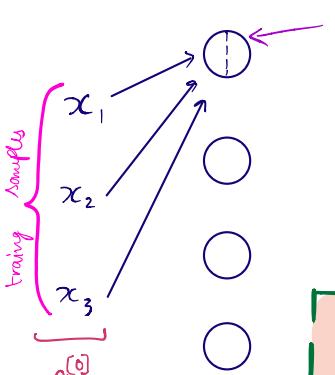
$z^{[1]} = W^{[1]}x + b^{[1]} \rightarrow a^{[1]} = \sigma(z^{[1]} \rightarrow z^{[2]} = W^{[2]}a^{[1]} + b^{[2]} \rightarrow a^{[2]} \rightarrow \mathcal{L}(a^{[2]}, y)$

$W^{[2]}$, $a^{[2]}$

Notation



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



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

$a_i^{[l]}$ ← l = layer, i ← node in layer ← notation

Vectorized:

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

$z^{[1]} = w^{[1]} x + b^{[1]}$ — $(4,1) = (4,3)(3,1) + (4,1)$

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

$\longrightarrow$

$z^{[2]} = w^{[2]} x + b^{[2]}$ — $(1,1) = (1,4)(4,1) + (1,1)$

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

$X = \begin{bmatrix} | & | & & | \\ x_1 & x_2 & \cdots & x_m \\ | & | & & | \end{bmatrix}$ (n rows, m columns) $\longrightarrow$ * training samples *

Vectorized implementation across multiple examples

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

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

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

$\vdots$

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

$a^{[2](i)}$ — layer 2, example i

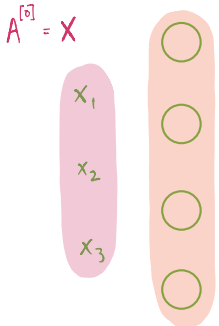
$A^{[1]} = \sigma(w^{[1]} x + b^{[1]})$

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

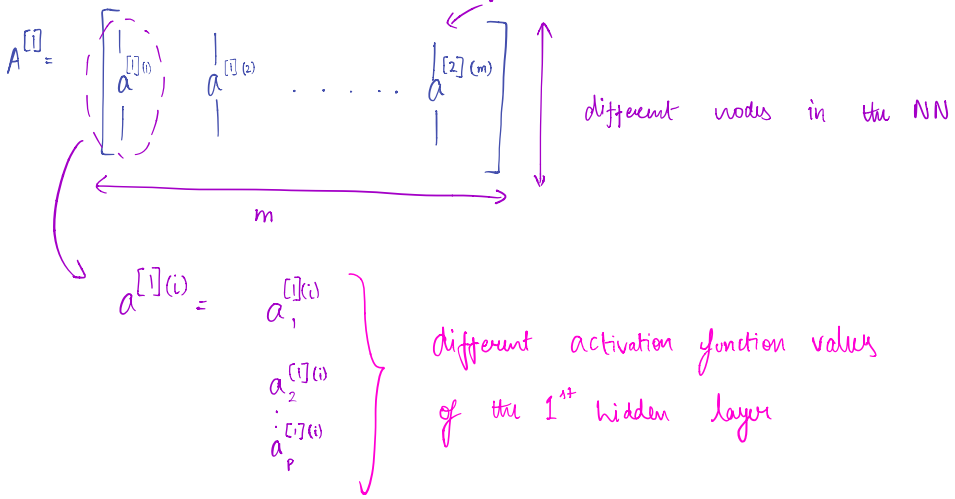
$A^{[1]} = \sigma(z^{[1]})$

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

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



$A^{[2]} = \sigma(w^{[2]} A^{[1]} + b^{[2]}) = \hat{y}$



Activation Functions

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

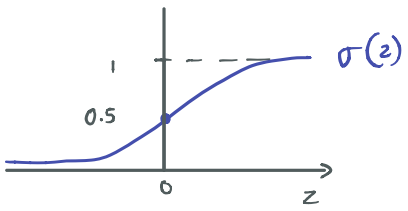
$$A^{[1]} = \sigma(z^{[1]}) \quad g^{[1]}(z^{[1]})$$

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

$$A^{[2]} = \sigma(z^{[2]}) \quad g^{[2]}(z^{[1]})$$

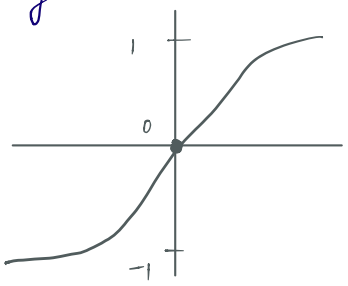
Sigmoid σ

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

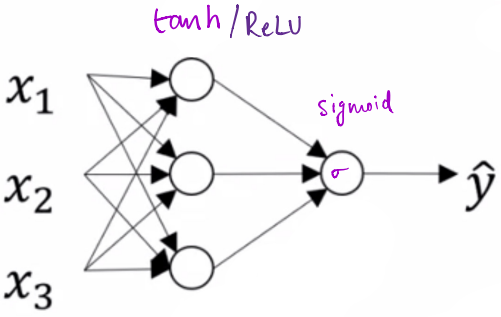


tanh g

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

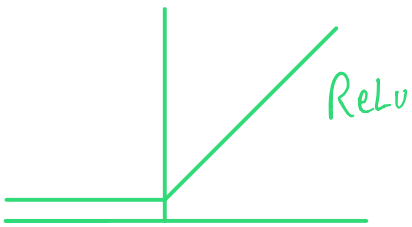


taking mean leads to closer to 0 than 0.5



- downside → z is very large dz is small → learning problems → slows down learning

ReLU



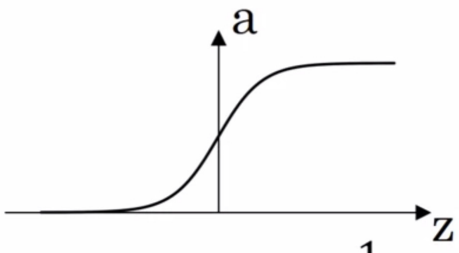
$$a = \max(0, z)$$

→ derivative of $z > 0$ is positive

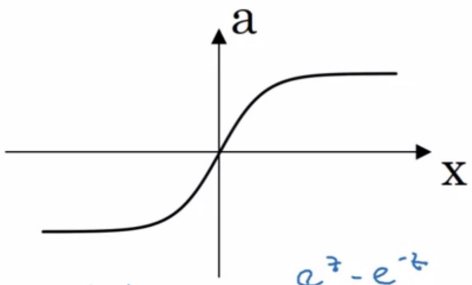
→ derivative of $z < 0$ is 0 → downside

→ slope is positive so learning is fast

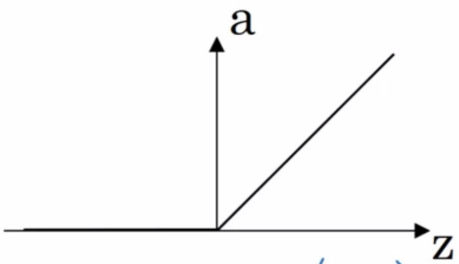
→ usually use this



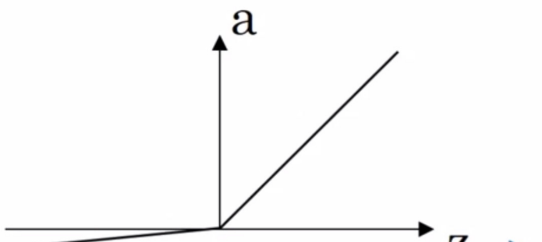
sigmoid: $a = \frac{1}{1+e^{-z}}$



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



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



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

Need for non linear Activation function

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

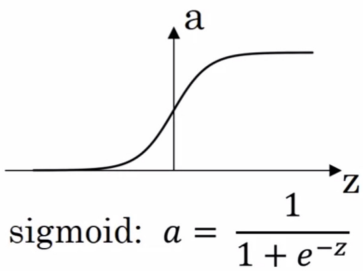
$A^{[1]} = \sigma(z^{[1]}) \quad z^{[1]}$

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

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

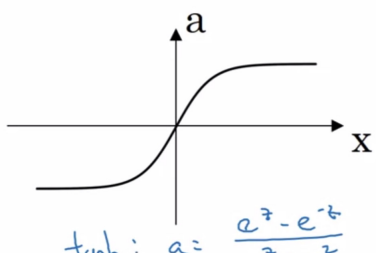
outputs a linear function if there is no activation function = no hidden layers = linear regression

* only okay in regression problems *



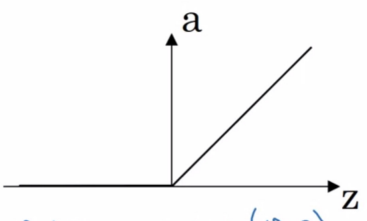
sigmoid: $a = \frac{1}{1+e^{-z}}$

g is large → σ = 1
g is small → σ = 0



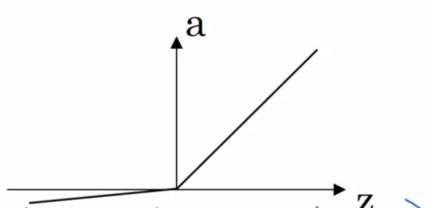
tanh: $a = \frac{e^{z}-e^{-z}}{e^{z}+e^{z}}$

g is large → tanh = 1
g is small → tanh = -1



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

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



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

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

Gradient Descent for Neural Networks

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

$n_x = n^{[0]}$ (input features), $n^{[1]}$ (hidden units), $n^{[2]} = 1$ (output units)

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

Gradient Descent:

Repeat {

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

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

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

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

$w^{[2]} = \ldots$

$b^{[2]} = \ldots$

}

Forward Propagation:

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

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

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

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

Back Propagation:

$$dZ^{[2]} = A^{[2]} - Y \qquad Y = [y^{(1)}\ y^{(2)} \ldots y^{(m)}]$$

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

$db^{[2]} = \frac{1}{m}$ np.sum$(dZ^{[2]}$, axis = 1 (horizontal sum), keepdims = T (prevents rank 1 arrays $(n,)$))

$dZ^{[2]} = \underbrace{W^{[2]T} dZ^{[2]}}_{(n^{[1]}, m)} * \underbrace{g^{[1]\prime}(Z^{[1]})}_{(n^{[1]}, m)}$ (* = element-wise product)

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

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

Summary of Gradient Descent

$$dz^{[2]} = a^{[2]} - y$$
$$dw^{[2]} = dz^{[2]} a^{[1]T}$$
$$db^{[2]} = dz^{[2]}$$
$$dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]'}(z^{[1]})$$
$$dw^{[1]} = dz^{[1]} x^{T}$$
$$db^{[1]} = dz^{[1]}$$

$$dZ^{[2]} = A^{[2]} - Y$$
$$dw^{[2]} = \frac{1}{m} dz^{[2]} A^{[1]T}$$
$$db^{[2]} = \frac{1}{m} np.sum(dz^{[2]}, axis=1, keepdims=T)$$

axis=1: horizontal sum; keepdims=T: prevents rank 1 arrays $(n,)$

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

$*$: element-wise product

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

Random Initialization

- Setting W to a zero matrix will lead to the same activation function in the given hidden layer.

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

- During backpropagation

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

- This leads to two or more units in the hidden layers to be the same.
- If all of the units are same, it will lead to the same activation function across all hidden layers. This means that there won't be any point of using hidden layers and is just simple linear regression.

Solution to this problem is random initialization

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

→ don't want weights to be too large ⇒ gradient descent is slow

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

$W^{[2]} = \ldots$

$b^{[2]} = 0$

Programming Assignment

Reminder: The general methodology to build a Neural Network is to:

1. Define the neural network structure (# of input units, # of hidden units, etc).
2. Initialize the model's parameters
3. Loop:
    - Implement forward propagation
    - Compute loss
    - Implement backward propagation to get the gradients
    - Update parameters (gradient descent)