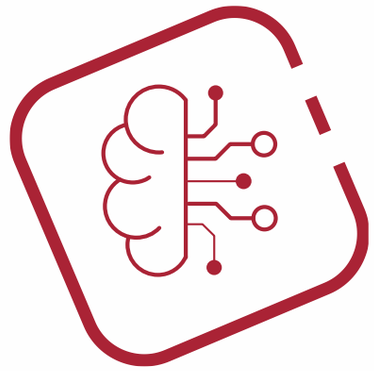
ITSAR ISTITUTO TECNICO SUPERIORE
ANGELO RIZZOLI

Deep Learning

Alice Castelnuovo



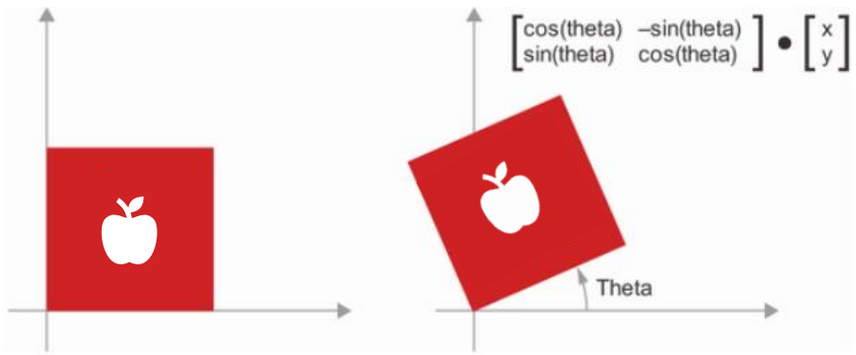
Trasformazioni affini

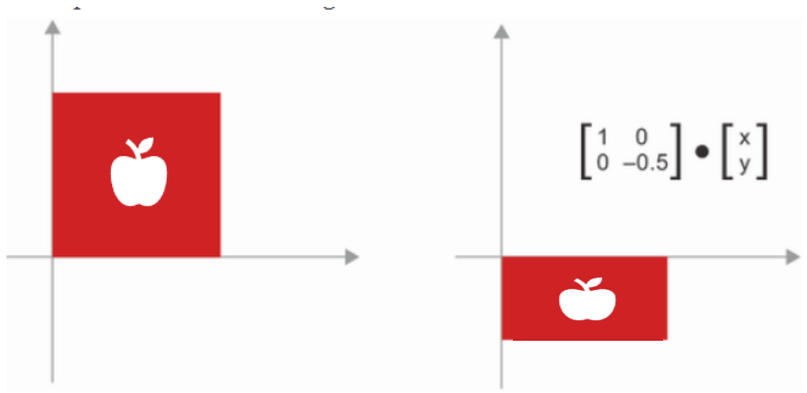
Abbiamo visto che ogni neurone, preso in input un vettore di dati X, trasforma il vettore secondo questa regola:

$$Z = W \cdot X + b$$

Ottenendo così un nuovo vettore Z.
Ogni trasformazione ottenuta moltiplicando un vettore per una matrice (**trasformazione lineare**) e aggiungendo un termine bias (**traslazione**) prende il nome di **trasformazione affine.**

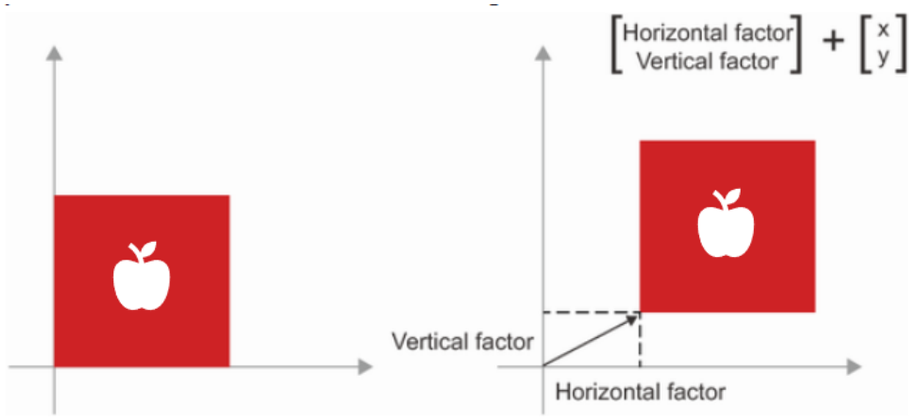
A seconda di come definisco la matrice W le trasformazioni che si applicano al vettore di partenza saranno diverse:





Trasformazioni affini

Un esempio di traslazione:



Esercizio

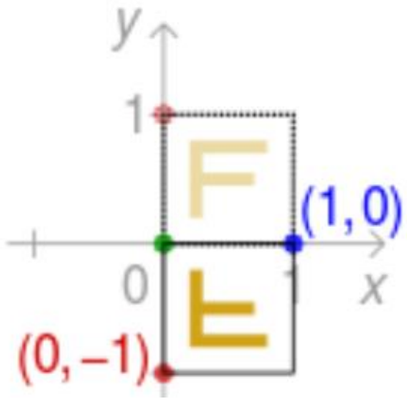
Associa ad ogni trasformazione affine la corrispondente matrice W. Un modo semplice è capire dove viene mappato il punto [1,1].

1. $W = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$

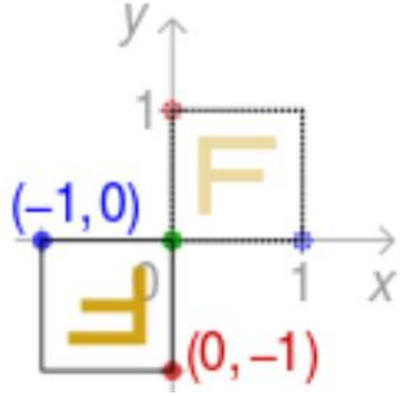
2. $W = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$

3. $W = \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}$

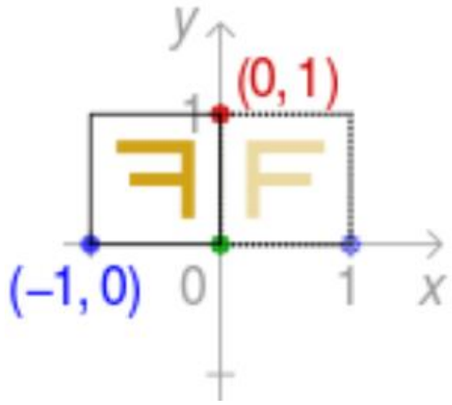
A.



B.



C.



Activation function…why?

Supponiamo di avere una ANN con due layer e un valore in input x. Allora la trasformazione applicata a x sarà la seguente:

$$\sigma^{[2]}(w^{[2]}\left(\sigma^{[1]}\left(w^{[1]}x + b^{[1]}\right)\right) + b^{[2]})$$

Cosa accadrebbe se non avessimo funzioni di attivazione?

Abbiamo visto che trasformazioni affini di trasformazioni affini generano trasformazioni affini:

$$f(x) = 3x + 1$$
$$g(x) = -x + 3$$

$$g(f(x)) = -(3x + 1) + 3 = -3x + 2$$

$g(f(x))$ continua ad essere una trasformazione affine.

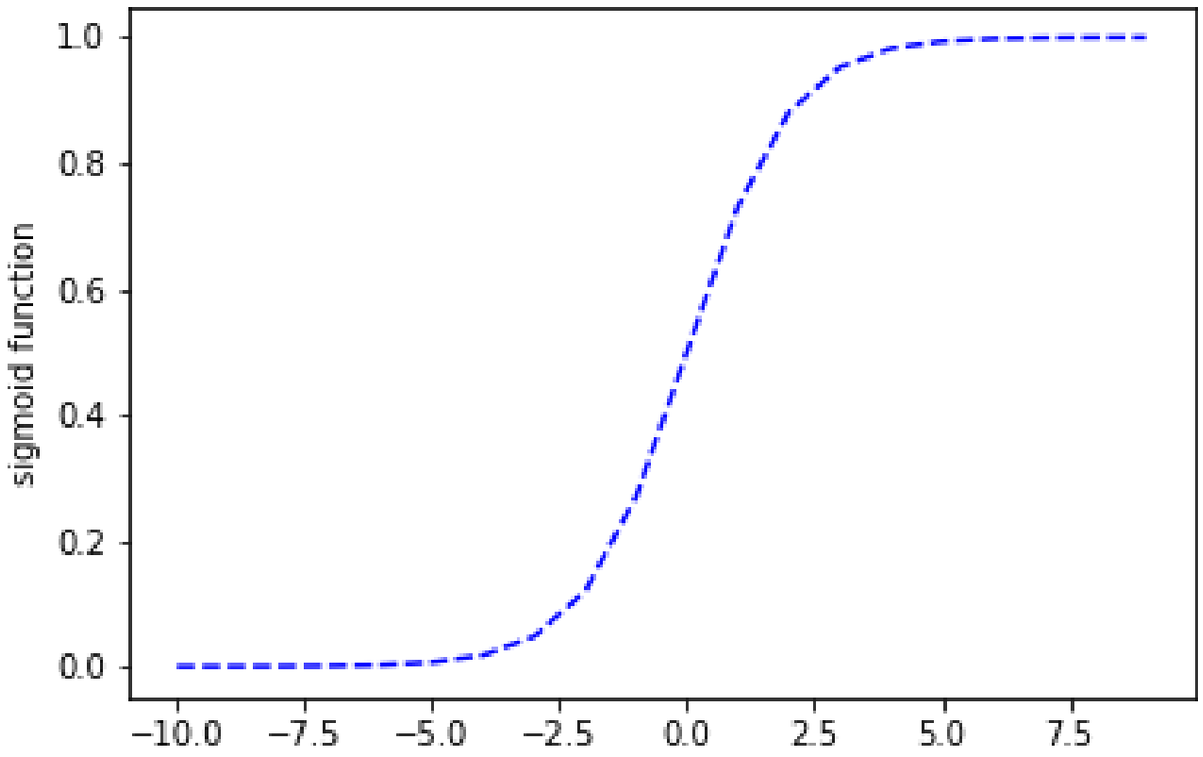
Avere come funzione di attivazione delle funzioni affini (o non avere del tutto delle funzioni di attivazione) rende inutile avere tanti hidden layer. Questo perché tutta la catena di trasformazioni può essere ricondotta ad un solo layer.

$$w^{[2]}\left(w^{[1]}x + b^{[1]}\right) + b^{[2]}) \rightarrow \tilde{w}x + \tilde{b}$$

Per questo motivo le funzioni di attivazione degli hidden layers sono sempre scelte **non lineari**.

Activation Functions

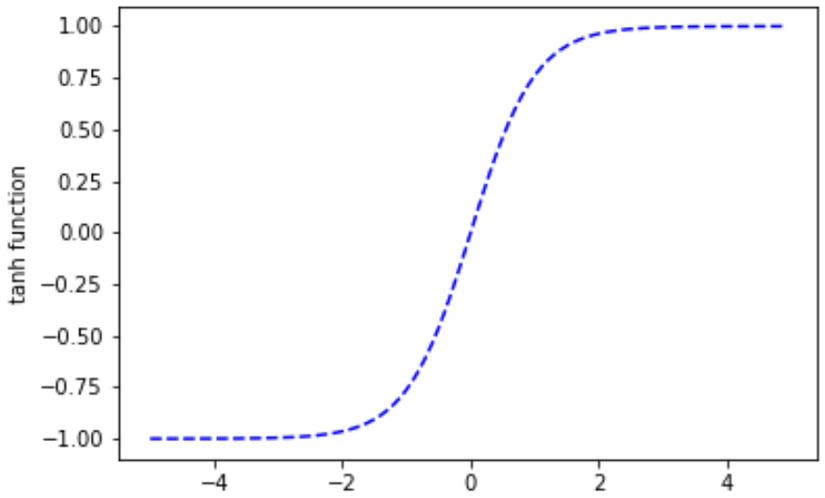
Finora, le uniche funzioni di attivazione che abbiamo visto sono state la sigmoide e la heaviside:



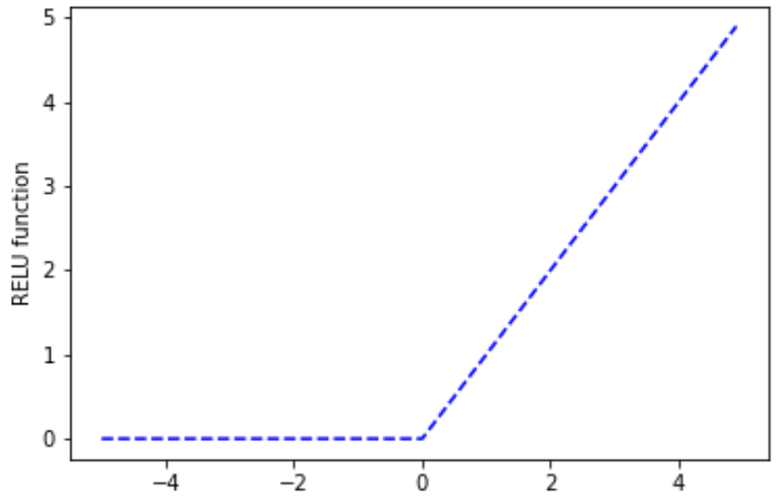
$$f(z) = \frac{1}{1 + e^{-z}}$$

Activation Functions

In realtà ci sono altre funzioni che vengono utilizzate:



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



$$RELU(z) = max(0, z)$$

Spesso, negli hidden layers si sceglie di utilizzare la RELU, questo perché:

1. La derivata è facile da calcolare;
2. Il valore in output non è limitato;

Esercizio

Utilizzando l'ambiente di simulazione di TensorFlow (https://playground.tensorflow.org/) scegli il dataset circolare e implementa una rete con tre hidden layer scegliendo come numero di neuroni 4,2,2.

1. Confronta il risultato che ottieni scegliendo come funzione di attivazione la RELU e la lineare.
2. Scegli come funzione di attivazione la lineare e prova a variare il numero di hidden layer. Cosa noti?

Esercizio

Date le funzioni di attivazione e le loro derivate, fai un grafico per le funzioni di attivazione e uno per le corrispettive derivate:

Funzioni	Derivate
$\boldsymbol{\sigma(z) = \frac{1}{1 + e^{-z}}}$	$\boldsymbol{\sigma'(z) = \sigma(z)(1 - \sigma(z))}$
$\boldsymbol{tanh(z) = 2\sigma(2z) - 1}$	$\boldsymbol{tanh'(z) = 1 - \big(tanh(z)\big)^2}$
$\boldsymbol{RELU(z) = max(0, z)}$	$\boldsymbol{RELU'(z) = \begin{cases} 0\ if\ z < 0 \\ 1\ if \geq 0 \end{cases}}$

Hint: per implementare la RELU puoi usare la funzione maximum di numpy. Mentre per implementare la derivata della RELU puoi utilizzare la funzione np.heaviside.

Activation Functions

Negli hidden layer è importante scegliere delle funzioni di attivazione che non siano lineari.
Tipicamente si tende a scegliere la RELU o delle sue varianti.

Ma qual è invece la logica da utilizzare quando parliamo di output layer?

- Se il problema che stiamo cercando di risolvere è un problema di classificazione binaria allora tipicamente si utilizza la funzione **sigmoide**;

- Se il problema che stiamo cercando di risolvere è un *problema di classificazione multiclasse* allora si usa la funzione **softmax** (generalizza la logistic regression) e un'unità di output per ogni classe;

- Se stiamo risolvendo un problema di regressione ($y \in \mathbb{R}$) allora ha senso usare una funzione di attivazione lineare;

- Se stiamo risolvendo un problema di regressione ($y \in \mathbb{R}^+$) allora si può utilizzare la **RELU**;

Back-propagation Algorithm

Abbiamo visto che il back-propagation algorithm è composto principalmente da due parti:

Forward propagation → partendo dai dati, si calcola l'output finale applicando le trasformazioni lineari e la funzione di attivazione per ogni layer della rete;

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

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

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

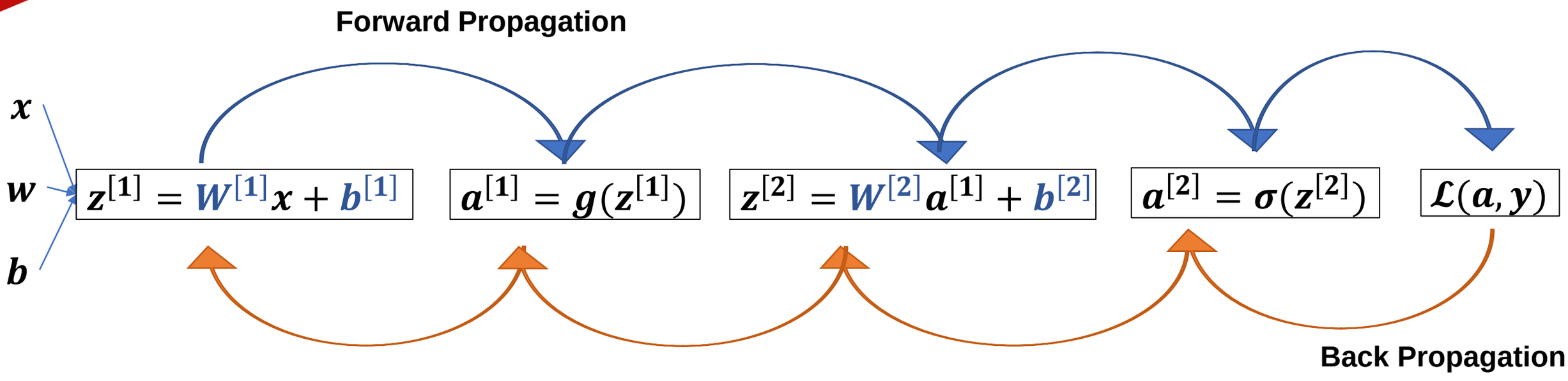
$$\hat{y} = A^{[2]} = \sigma(Z^{[2]})$$

Back propagation → percorrendo a ritroso la rete si calcolano le derivate parziali che serviranno per aggiornare i parametri;

$$\frac{d\mathcal{L}}{dW^{[2]}} \qquad \frac{d\mathcal{L}}{dW^{[1]}}$$

$$\frac{d\mathcal{L}}{db^{[2]}} \qquad \frac{d\mathcal{L}}{db^{[1]}}$$

Back propagation step



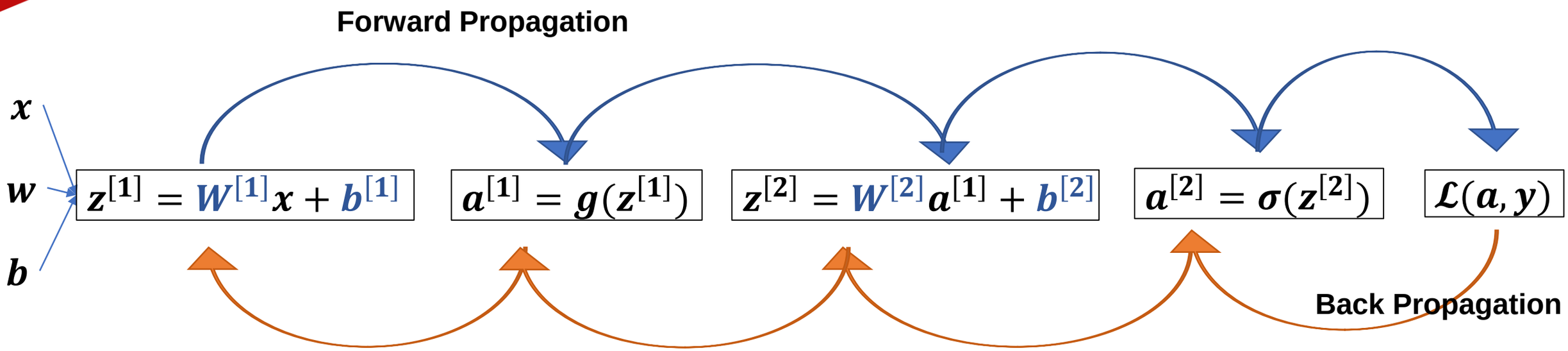
Scopo:
Vogliamo calcolare le derivate parziali così da poter aggiornare i parametri ad ogni passaggio.
Ci servono:

$$\frac{d\mathcal{L}}{dW^{[2]}} \qquad \frac{d\mathcal{L}}{dW^{[1]}}$$

$$\frac{d\mathcal{L}}{db^{[2]}} \qquad \frac{d\mathcal{L}}{db^{[1]}}$$

Per calcolarli, usiamo la chain rule.

Back propagation step



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

$$da^{[2]} = -\frac{y}{a^{[2]}} + \frac{1-y}{1-a^{[2]}}$$

Il secondo layer si comporta esattamente come il caso della Logistic Regression.

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

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

Back propagation step

Forward Propagation

x

w

b

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

Back Propagation

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

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

$$da^{[2]} = -\frac{y}{a^{[2]}} + \frac{1-y}{1-a^{[2]}}$$

$$z^{[2]} = w_1^{[2]}g(z_1^{[1]}) + w_2^{[2]}g(z_2^{[1]}) + w_3^{[2]}g(z_3^{[1]}) + w_4^{[2]}g(z_4^{[1]}) + b^{[2]}$$

$$\frac{d\mathcal{L}}{dz^{[1]}} = \underbrace{\frac{d\mathcal{L}}{da^{[2]}}\frac{da^{[2]}}{dz^{[2]}}}_{dz^{[2]}}\underbrace{\frac{dz^{[2]}}{da^{[1]}}\frac{da^{[1]}}{dz^{[1]}}}_{\frac{dz^{[2]}}{dz^{[1]}}}$$

?

$dz^{[2]}$ **(1,1)**

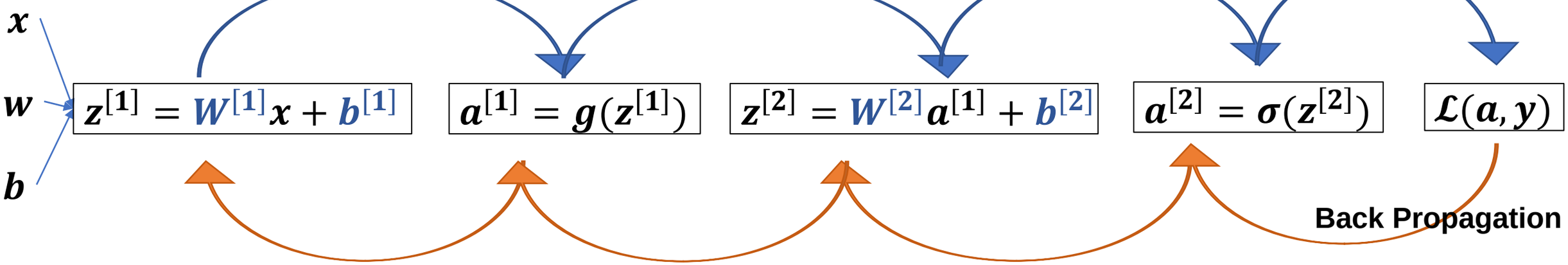
$\frac{dz^{[2]}}{dz^{[1]}}$ **(4,1)**

$$\frac{dz^{[2]}}{dz^{[1]}} = \begin{pmatrix} w_1^{[2]}g'(z_1^{[1]}) \\ w_2^{[2]}g'(z_2^{[1]}) \\ w_3^{[2]}g'(z_3^{[1]}) \\ w_4^{[2]}g'(z_4^{[1]}) \end{pmatrix} = W^{[2]T} * g'(z^{[1]})$$

(4,1) * **(4,1)**

Back propagation step

Forward Propagation



$$dz^{[1]} = dz^{[2]}W^{[2]T} * g'(z^{[1]}) \qquad da^{[1]} = W^{[2]T}dz^{[2]} \qquad dz^{[2]} = a^{[2]} - y$$

$$da^{[2]} = -\frac{y}{a^{[2]}} + \frac{1-y}{1-a^{[2]}}$$

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

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

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

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

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

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

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

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

Forward propagation

vectorization

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

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

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

$$dZ^{[1]} = \left(W^{[2]T} dZ^{[2]}\right) * g'(z^{[1]})$$

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

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

Back propagation

Check sulle dimensioni

$$Z^{[1]} = W^{[1]}X + b^{[1]} \qquad (n_h, m)$$

$$A^{[1]} = g^{[1]}(Z^{[1]}) \qquad (n_h, m)$$

$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]} \qquad (n_y, m)$$

$$\hat{y} = A^{[2]} = \sigma(Z^{[2]}) \qquad (n_y, m)$$

$$dZ^{[2]} = A^{[2]} - y \qquad (n_y, m)$$

$$\boldsymbol{dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}} \qquad (n_y, n_h)$$

$$\boldsymbol{db^{[2]} = \frac{1}{m} np.sum(dZ^{[2]} \dots)} \qquad (n_y, 1)$$

$$dZ^{[1]} = W^{[2]T} dZ^{[2]} * g'(z^{[1]}) \qquad (n_h, m)$$

$$\boldsymbol{dW^{[1]} = \frac{1}{m} dZ^{[1]} A^{[0]T}} \qquad (n_h, n_x)$$

$$\boldsymbol{db^{[1]} = \frac{1}{m} np.sum(dZ^{[1]} \dots)} \qquad (n_h, 1)$$

Dove, nel nostro caso:

$n_y = 1$

$n_h = 4$

$n_x = 2$

Backpropagation Algorithm

Supponiamo di avere un dataset X composto da n features e m istanze.

1. **Forward Propagation:** per ogni elemento del dataset calcoliamo l'output $\hat{y}$:

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

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

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

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

2. **Backward Propagation:** Una volta che abbiamo l'output $\hat{y}$ e tutte le computazioni intermedie, calcoliamo le derivate per ogni parametro:

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

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

…

3. **Gradient Descent Step:** Una volta che abbiamo a disposizione le derivate, aggiorniamo i parametri del modello utilizzando il metodo del gradiente:

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

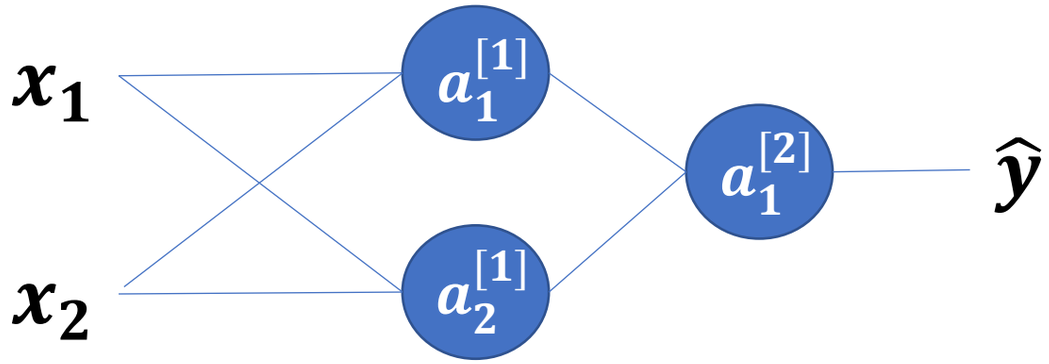
$$W^{[2]} = \mathrm{W}^{[2]} - \lambda dW^{[2]}$$

…

4. Si ripetono i punti precedenti fino a quando non è stato soddisfatto un certo criterio;

Inizializzare i parametri

Ci rimane da capire come inizializzare i parametri per poter procedere con il primo step.



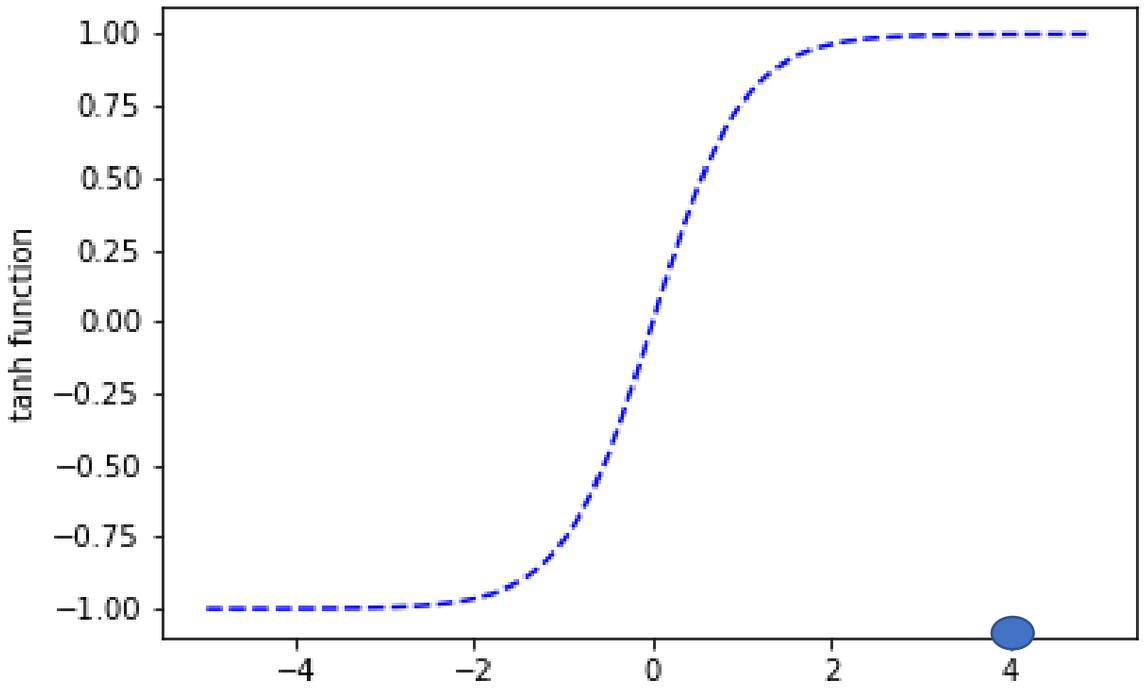
Supponiamo di inizializzare i pesi $W^{[1]}$ e $W^{[2]}$ con gli stessi valori. Allora $a_1^{[1]}$ avrà esattamente lo stesso valore di $a_2^{[1]}$, e quindi $a_1^{[2]}$ riceverà in input gli stessi valori. Nello step di backpropagation, essendo che ogni nodo contribuisce in egual misura al calcolo di $\hat{y}$, le derivate saranno le stesse e quindi i parametri saranno aggiornati della stessa quantità.

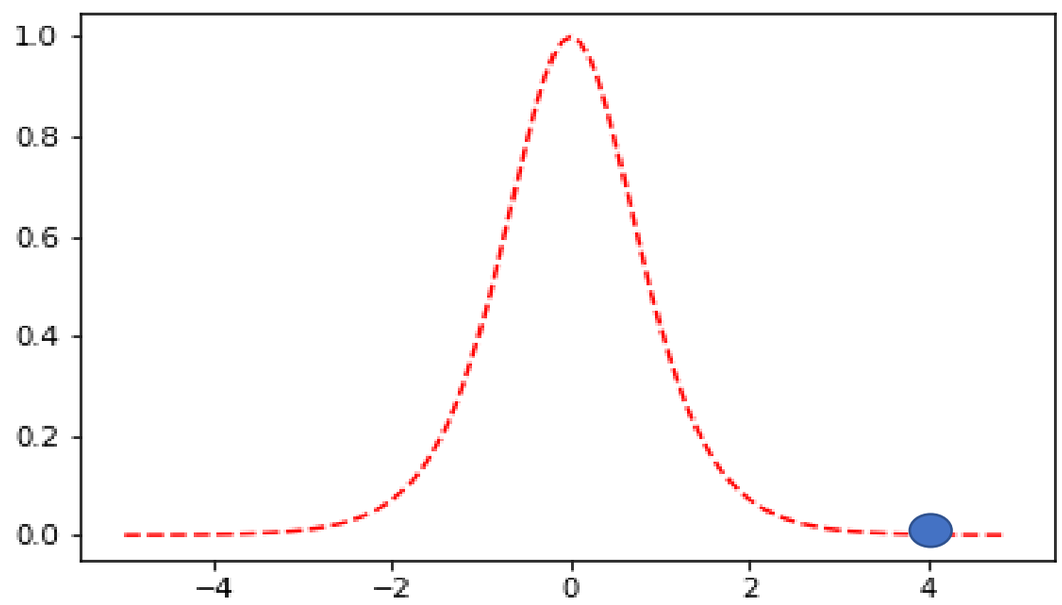
Se i parametri del modello vengono inizializzati con gli stessi valori allora avere un layer con 100 neuroni porterà allo stesso risultato che avere un layer con un solo neurone.

Break the simmetry: è importante inizializzare i parametri in maniera casuale;

Inizializzare i parametri

Supponiamo di avere come funzione di attivazione degli hidden layer la tanh.

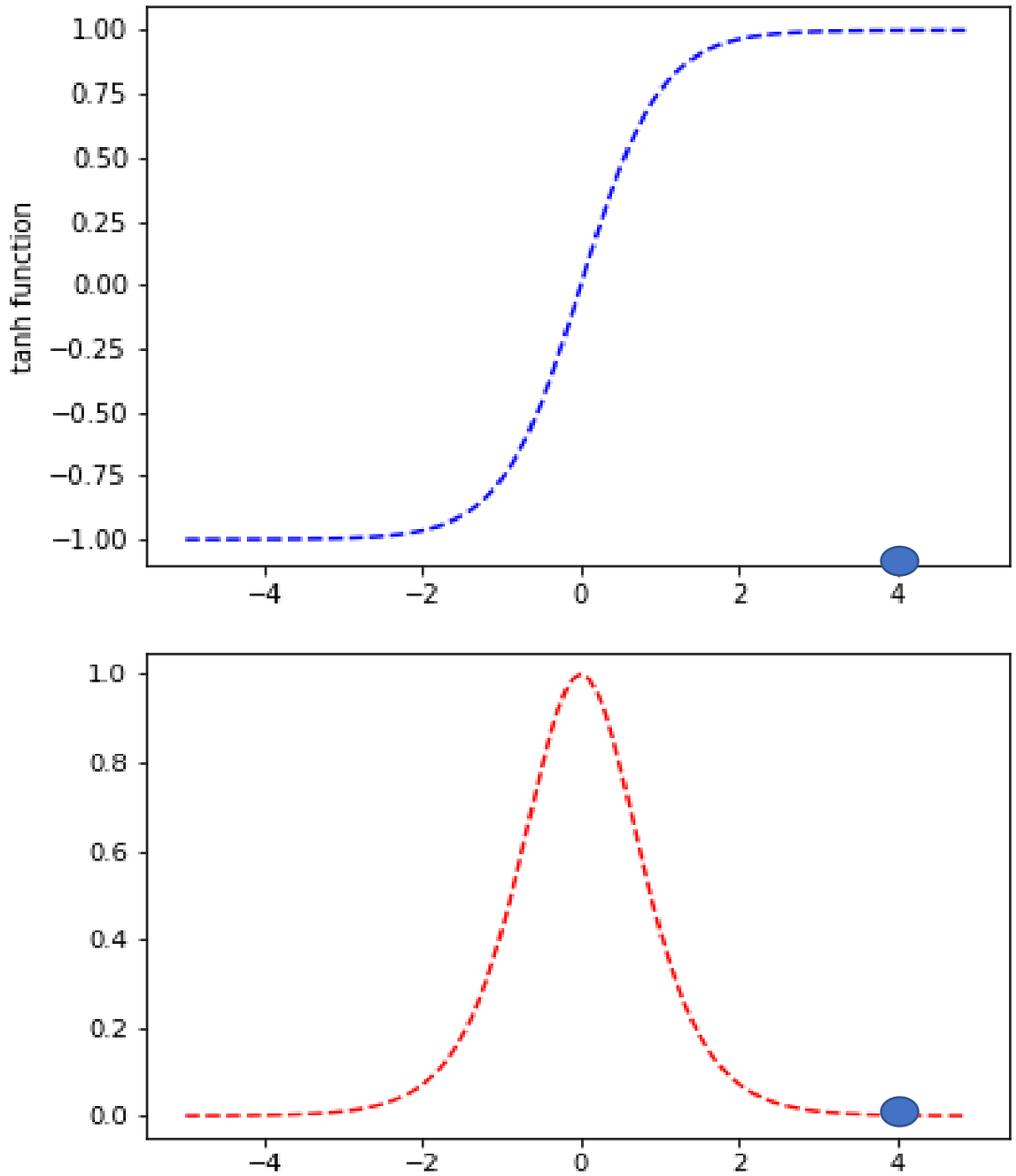




Se inizializziamo i parametri con dei valori lontani da zero, il valore delle derivate sarà praticamente nullo.
Se consideriamo una rete con tanti hidden layer, durante la fase di back-propagation le derivate dei primi layer verranno calcolate facendo il prodotto delle derivate dei layer a destra (chain rule). Se ognuna di queste derivate è piccola, facendo il prodotto si otterranno delle derivate sempre più piccole.

In questo caso l'algoritmo potrebbe metterci molto a convergere o non covergere mai.
Questo fenomeno prende il nome di **Vanishing Gradient Problem** ed è stato uno dei motivi per cui il Deep Learning fu parzialmente abbandonato all'inizio degli anni 2000.

Inizializzare i parametri



Quando si utilizza come funzione di attivazione la tangente iperbolica un buon modo per inizializzare i parametri è il seguente:

Estrarre da una distribuzione normale con valore atteso 0 e varianza $\sigma^2 = \frac{1}{fan_{avg}}$

Dove $fan_{avg} = \frac{\text{fan}_{\text{in}}+\text{fan}_{\text{out}}}{2}$

fan_{in} → numero di neuroni in input;
fan_{out} → numero di neuroni in output;

Inizializzare i parametri

A seconda della funzione di attivazione scelta, la varianza da utilizzare cambia secondo questa regola:

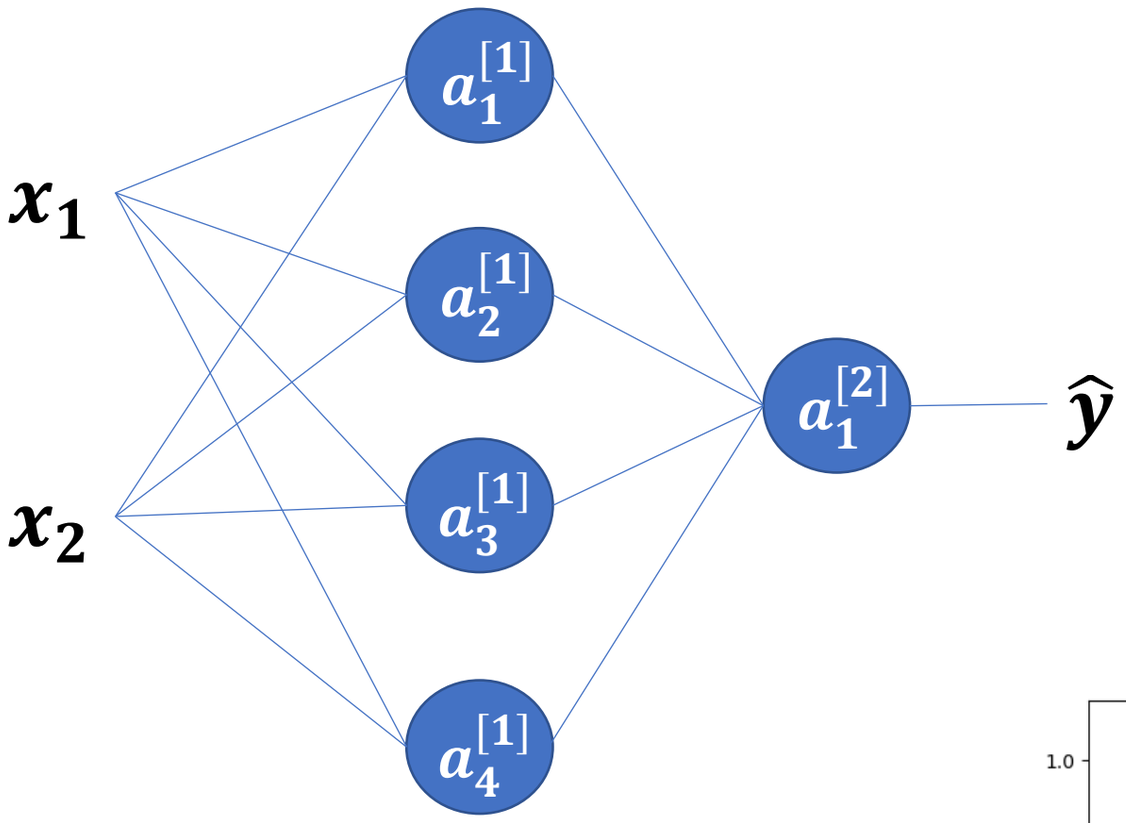
Nome	Funzione di attivazione	$\sigma^2(\mathbf{Normal})$
Glorot	Tanh, sigmoide, softmax	$1/fan_{\text{avg}}$
He	RELU e varianti	$2/fan_{in}$

Esercizio

Supponi che $fan_{in} = 4$ e $fan_{out} = 4$, l'obiettivo è generare dei dati estratti da una distribuzione normale con varianza: $1/fan_{\text{avg}}$

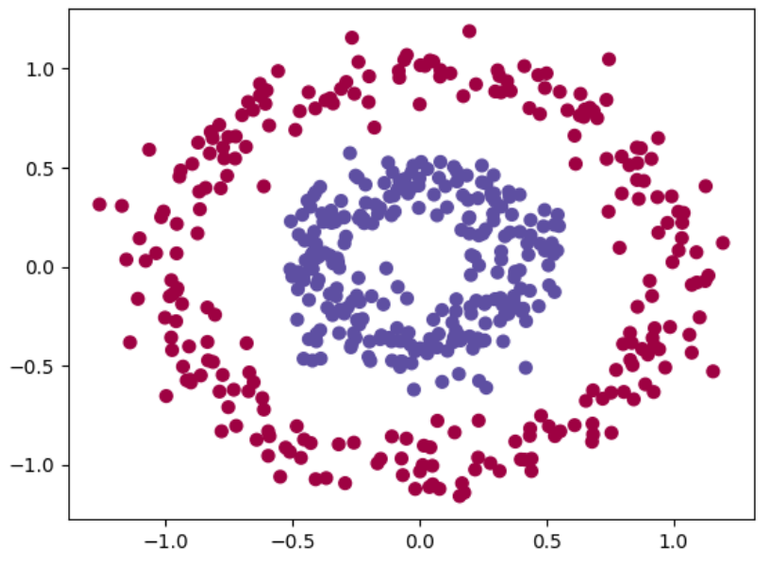
- Generare dei dati distribuiti normalmente: Z = np.random.randn(1000,1) ;
- Fai un istogramma dei dati ottenuti (plt.hist())
- Calcola la varianza;
- Prova a moltiplicare i dati per 2 e calcola di nuovo la varianza;
- Trova la regola da applicare ai dati di partenza in modo tale che abbiano varianza pari a $1/fan_{\text{avg}}$;
- Confronta l'istogramma dei dati iniziali con quello ottenuto dalla formula di Glorot;

Artificial Neural Network



Vogliamo implementare una ANN con la seguente struttura:

1. Un hidden layer composto da 4 neuroni;
2. Funzione di attivazione per $a^{[1]}$ → tanh
3. Funzione di attivazione per $a^{[2]}$ → σ



Inizializzare i parametri

Supponiamo di voler implementare una ANN con un solo hidden layer. Allora sarà necessario inizializzare i parametri del modello:

$b^{[1]} \rightarrow (n_h, 1)$
$W^{[1]} \rightarrow (n_h, n_x)$
$b^{[2]} \rightarrow (n_y, 1)$
$W^{[2]} \rightarrow (n_y, n_h)$

Inizializziamo i parametri utilizzando la regola di Glorot.

Scrivi una classe ANN con costruttore che si occupi di inizializzare i parametri.

```
class ANN:

    def __init__(self, n_x, n_h, n_y):
        np.random.seed(2)

        self.n_x = n_x
        self.n_h = n_h
        self.n_y = n_y

        self.W1 = …
        self.b1 = …
        self.W2 = …
        self.b2 = …
```

Forward Propagation

Aggiungi alla classe ANN un metodo

```
def forward_propagation(self, X):
```

Che calcoli i valori e restituisca in output il seguente dizionario:

```
{"Z1": Z1,
      "A1": A1,
      "Z2": Z2,
      "A2": A2}
```

$$Z^{[1]} = W^{[1]}X + b^{[1]} \qquad (n_h, m)$$

$$A^{[1]} = g^{[1]}(Z^{[1]}) \qquad (n_h, m)$$

$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]} \qquad (n_y, m)$$

$$\hat{y} = A^{[2]} = \sigma(Z^{[2]}) \qquad (n_y, m)$$

Funzione di costo

Aggiungi alla classe ANN un metodo che dati in input Y e A2, calcoli il valore della cost function:

$$J = -\frac{1}{m}\sum_{i=0}^{m}(y^{(i)}\log\left(a^{[2](i)}\right) + \left(1 - y^{(i)}\right)\log\left(1 - a^{[2](i)}\right))$$

```
def compute_cost(self, Y, A2):
```

Back propagation

Aggiungi alla classe ANN un metodo che calcoli la back_propagation e aggiorna i parametri:

$$dZ^{[2]} = A^{[2]} - y \qquad (n_y, m)$$

$$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T} \qquad (n_y, n_h)$$

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

$$dZ^{[1]} = W^{[2]T} dZ^{[2]} * g'(z^{[1]}) \qquad (n_h, m)$$

$$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T \qquad (n_h, n_x)$$

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

```
back_propagation(self, X, Y, output, lambd)
```

Dove $g'(z^{[1]})$ è la derivata della tanh (activation.tanh_dev(A1)) e la regola per aggiornare i parametri è la seguente:

$$W_1 = W_1 - \text{lambd} * dW1 \qquad b_2 = b_2 - \text{lambd} * db2$$

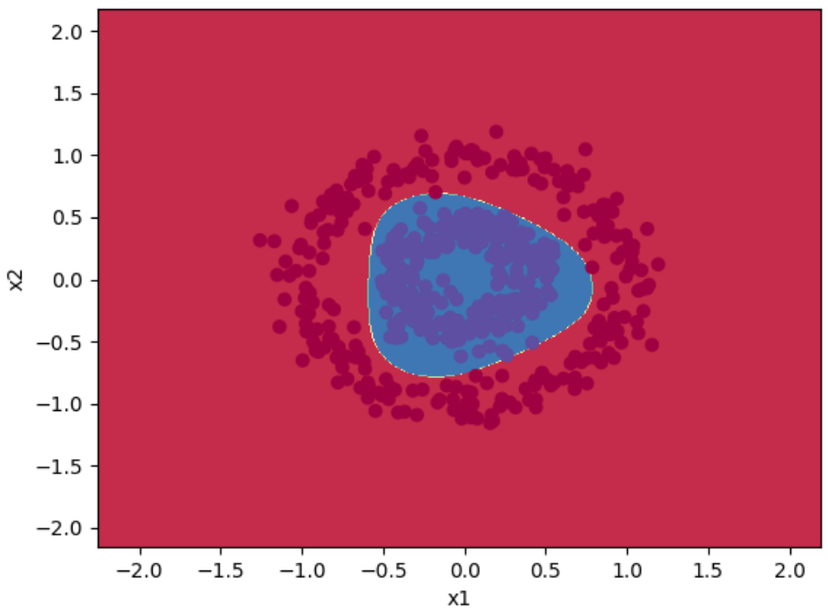
$$b_1 = b_1 - \text{lambd} * db1 \qquad W_2 = W_2 - \text{lambd} * dW2$$

Fit

Sfruttando gli step che abbiamo fatto finora, scrivi un metodo fit che calcoli i parametri del modello e un metodo predict che, dato in input X, calcoli $a^{[2]}$ e restituisca $a^{[2]} > 0.5$.
Nel metodo fit calcola la funzione di costo all'aumentare del numero di iterazioni e salva l'immagine ottenuta.

```
fit(self, X, Y, n_iter=10000, lambd=0.1)
```

```
predict(self, X)
```



Accuracy

Alla fine, scrivi nel main un metodo che calcoli l’accuracy ottenuta dopo aver fatto training del modello.