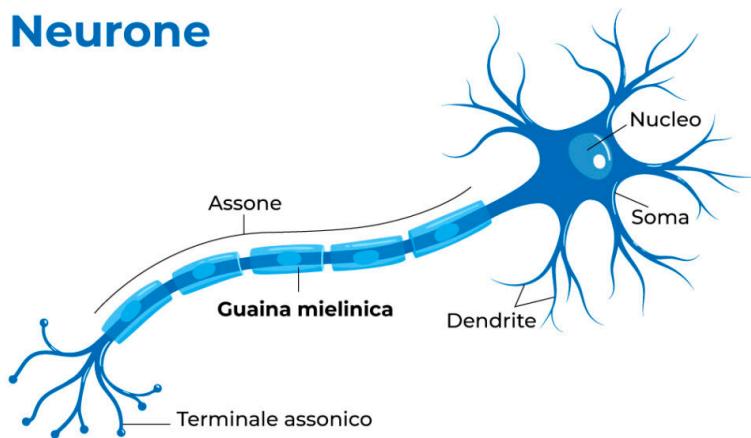


NEURONE BIOLOGICO

Neurone



Un neurone biologico è costituito da tre principali componenti:

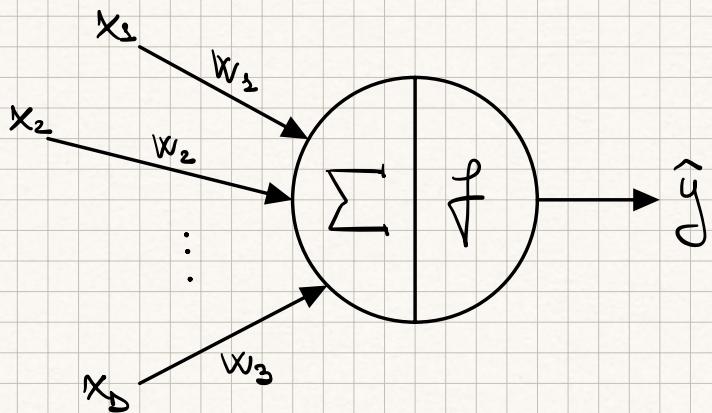
- 1) **SOMA**: Corpo cellulare
- 2) **ASSONE**: Uscita unica che si divide in più ram
- 3) **DENDRITO**: Entrata da cui il neurone riceve segnali

L'idea del neurone biologico è che, se gli arriva un segnale abbastanza forte, esso si attiva. Altrimenti il neurone rimane in uno stato di riposo.

Dato che ci sono più ingressi il neurone fa una somma pesata degli ingressi.

NEURONI ARTIFICIALI

Il neurone artificiale cerca di imitare quello biologico, anche se in modo semplificato



Dove con \sum si indica la **PRE-ATTIVAZIONE**, che è una somma pesata:

$$\sum_{i=1}^m w_i x_i$$

Mentre con $f(\cdot)$ si indica una **FUNZIONE DI ATTIVAZIONE** non lineare, tale per cui l'uscita diventa:

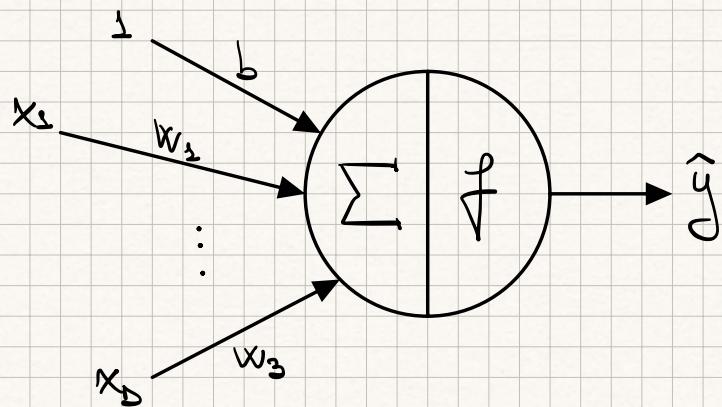
$$\hat{y} = f\left(\sum_{i=1}^m w_i x_i\right)$$

Se il peso di un ingresso è positivo, il neurone viene eccitato, altrimenti si ha un effetto inibitorio.

L'output \hat{y} può essere un numero reale o discreto dentro un intervallo.

BIAS

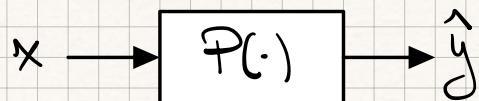
Di base c'è un peso collegato a un ingresso sempre unitario. Quindi c'è sempre:



$$\hat{y} = f \left(\sum_{i=0}^m w_i x_i \right) = f \left(\sum_{i=1}^m w_i x_i + b \right)$$

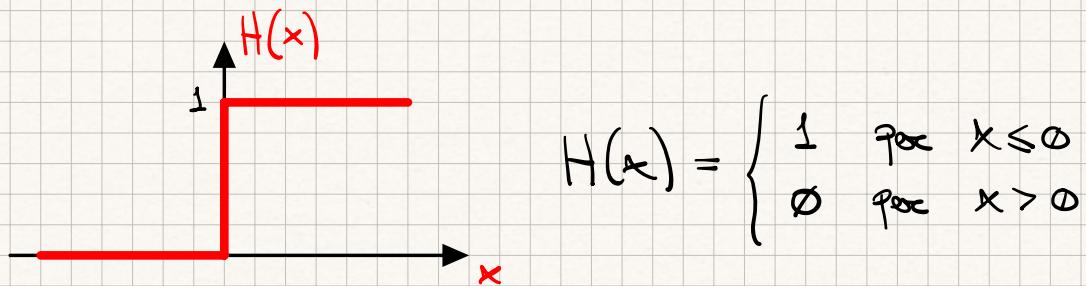
PERCEPTRON

Nel 1958 Rosenblatt introdusse il concetto di **percezione** che è un singolo neurone con una funzione di attivazione



Dove $x \in \mathbb{R}^D$ e $\hat{y} \in \{0, 1\}$.

Tipicamente la funzione di attivazione è la funzione **gradino o Heaviside**:



Parametri

In un neurone abbiamo quindi dei parametri da settare, il vettore $w \in \mathbb{R}^D$ e $b \in \mathbb{R}$. Il nostro percepitrone P diventa:

$$P(\vec{x}, \vec{w}, b) = H\left(\sum_{i=1}^D w_i x_i + b\right)$$

Possiamo scrivere la pre-attivazione come

$$\sum_{i=0}^{D+1} w_i x_i \quad \text{con } x_i \in X_e$$

DELTA RULE

È un regola di correzione dell'errore di una rete neurale. Questa regola dice che il peso al passo successivo è pari a:

$$W_{\text{new}} = W_{\text{old}} - \eta (\hat{y} - y) x$$

Dove \hat{y} è la predizione, y è l'output attuale (la label del dataset) e x è l'ingresso. Mentre η è il learning rate.

Note: Il singolo percezzone è bravo nei problemi di classificazione binaria.

CODICE MATLAB: Training del perceptron con la delta rule

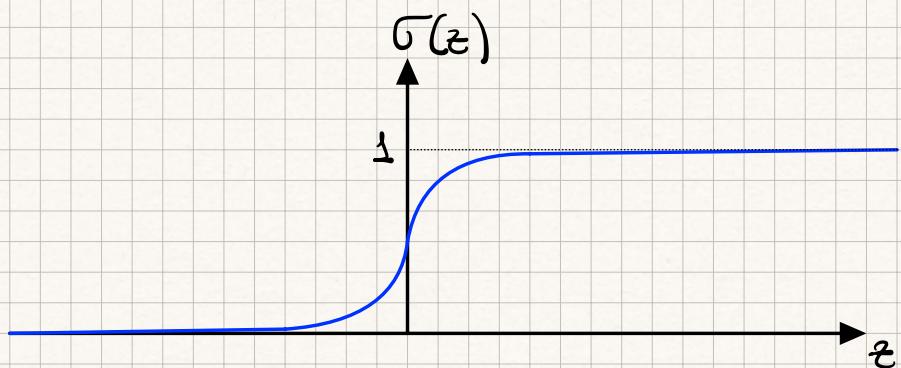
```
LAB_03 > perception_train.m
1 function w = perception_train(Xtrain, ytrain, w_init, eta)
2 % Train of the perceptron, using the delta rule.
3 % See Probabilistic Machine Learning: An Introduction", by Kevin
4 % P. Murphy, Chapter 10 ("Logistic Regression").
5
6 w = w_init;
7 norm2 = 1e5; tol=1e-4;
8
9 while (norm2 > tol)
10     w_old = w;
11     for i=1:size(Xtrain,1)
12         y_pred = heaviside(w'*Xtrain(i,:));
13         r = y_pred - ytrain(i);      % r can only be 0 or 1
14         w = w - eta*r*Xtrain(i,:);
15     end
16     norm2 = norm(w_old - w); % norm 2 of the difference of the two vectors
17 end
18
19 end
20
21 function a = heaviside(x)
22     if x > 0
23         a = 1;
24     else
25         a = 0;
26     end
27 end
```

MULTILAYER PERCEPTRON

L'idea è usare più perceptron per riuscire a classificare più di una classe.

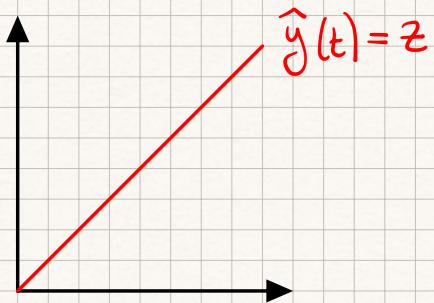
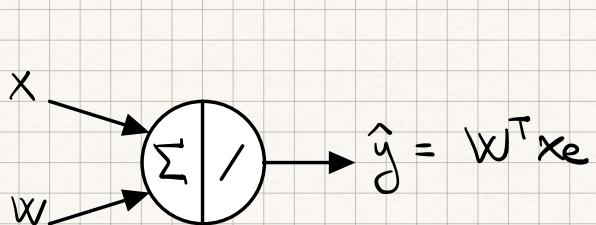
Sostituiamo a ogni neurone una funzione di attivazione differenziabile, come la **sigmoid** ($\sigma(\cdot)$):

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

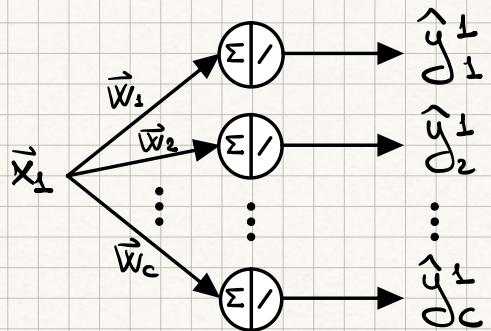


Esempio funzione lineare

Se usiamo un neurone con funzione d'attivazione lineare



5 se usiamo più neuroni?



In questo caso abbiamo un **single layer perceptron** in grado di risolvere una regressione di tipo multi-linear regression con più output.

Abbiamo $X \in \mathbb{R}^{P \times D}$ e $Y \in \mathbb{R}^{P \times C}$, questo perché X e Y sono così fatte:

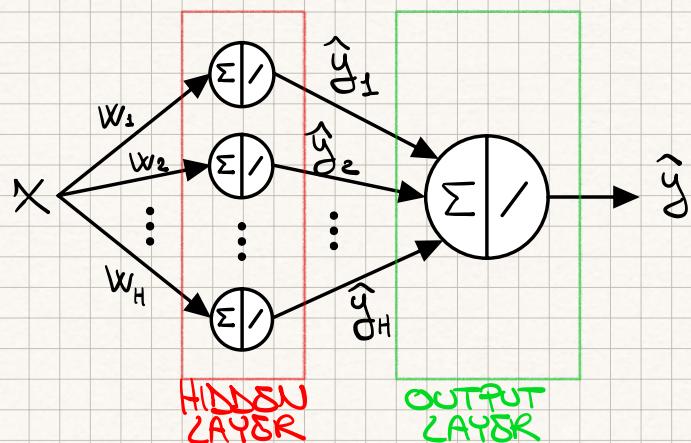
$$X = \begin{bmatrix} \leftarrow x_1 \rightarrow \\ \leftarrow x_2 \rightarrow \\ \vdots \\ \leftarrow x_p \rightarrow \end{bmatrix} \quad P$$

$$Y = \begin{bmatrix} y_1 = [\hat{y}_1^1 \dots \hat{y}_1^C] \\ \leftarrow y_2 \rightarrow \\ \vdots \\ \leftarrow y_p \rightarrow \end{bmatrix} \quad P$$

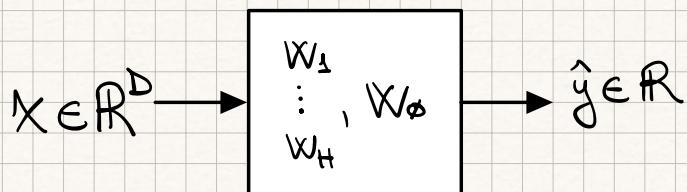
Dove l'uscita y_1 è il vettore $[\hat{y}_1^1 \dots \hat{y}_1^C]$ dato dal solo ingresso x_1 . La composizione di tutti gli ingressi mi dà la matrice Y .

Se le **colonne** di Y sono indipendenti fra di loro posso usare L_1, \dots, L_C loss diverse per impostare $w_1 \dots w_c$

Aggiungiamo ora un livello di output, utile ad aggregare le uscite del livello 1:



Anche il singolo neurone d'uscita avrà la sua matrice dei pesi W_ϕ , che insieme ai pesi dei livelli nascosti precedenti formano il modello:



Dove $W_1, \dots, W_H \in \mathbb{R}^{D \times 1}$, mentre $W_\phi \in \mathbb{R}^{H+1}$.

In formula possiamo dire che:

$$z_1^1 = \sum_{i=0}^D w_i^1 \cdot x_i$$

$$z_2 = \sum_{i=0}^H w_{\phi i} \cdot \hat{y}_i$$

Per trovare \hat{z}_2 bisogna estendere $\hat{\vec{y}}$:

$$\hat{\vec{y}}_e = \begin{bmatrix} \frac{1}{\hat{y}_1} \\ \hat{y}_1 \\ \vdots \\ \hat{y}_H \end{bmatrix}$$

Alla fine la nostra uscita del modello dipenderà da:

$$\hat{\vec{y}} = \vec{F}(\vec{x}, \vec{w}_1, \dots, \vec{w}_H, \vec{w}_\phi)$$

Raggruppiamo i pesi dei livelli nascosti

Il passo successivo è organizzare tutti i pesi degli hidden layers in un'unica matrice W_1 :

$$W_1 = \begin{bmatrix} \leftarrow W_1^T \rightarrow \\ \vdots \\ \leftarrow W_H^T \rightarrow \end{bmatrix} \in \mathbb{R}^{H \times (D+1)}$$

Da qui possiamo dire che $\hat{z}_1 = W_1 \cdot x_e$, ovvero la creazione di tutti i neuroni dell'hidden layer 1. Possiamo moltiplicare perché le dimensioni tornano:

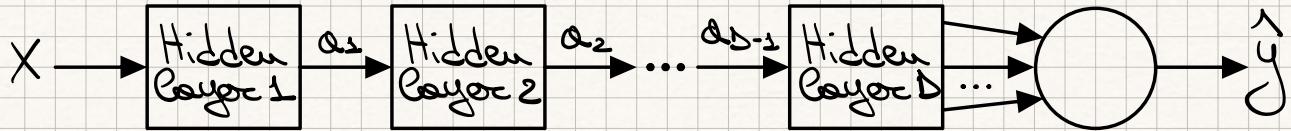
$$W_1 \in \mathbb{R}^{H \times (D+1)}, \quad x_e \in \mathbb{R}^{(D+1) \times 1}, \quad \hat{z}_1 \in \mathbb{R}^{H \times 1}$$

Usando la funzione di attivazione lineare:

$$Q_i = \text{lin}(z_i) = z_i \quad \xrightarrow{\text{perché lineare}}$$

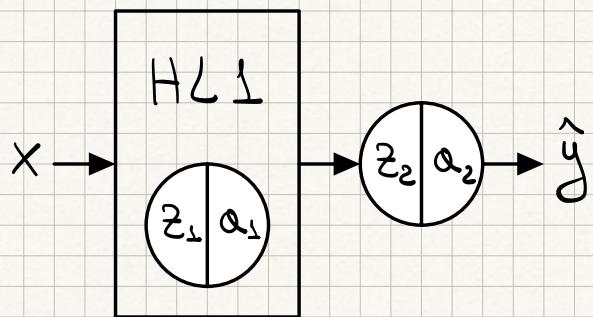
Dove Q_i è l'attivazione del livello $i+1$: $Q_{i+1} \equiv \hat{y}_i$

In generale:



SINGOLO HIDDEN LAYER

Torniamo al singolo hidden layer per capire cosa succede matematicamente



In calcoli:

$$Z_1 = W_1 \cdot X$$

$$Q_1 = Z_1 \quad // \text{livello lineare}$$

$$Z_2 = W_2 Q_1 \quad // \text{Non si può forzare}$$

$$Q_2 = W_2 \cdot Z_2$$

$$\hat{y} = Q_2$$

Non va bene perché non tornano le dimensioni degli elementi che moltiplichiamo. In particolare:

$$W_2 \in \mathbb{R}^{H+1}, \quad Q_1 \in \mathbb{R}^H$$

bisogna ricordare il bias in W_2 :

$$z_2 = \sum_{i=1}^H w_{2i} \cdot q_{1i} + b_2$$

Bisogna quindi **estendere** q_i ogni volta che viene fatto il prodotto scalare con una matrice dei pesi. Di base, l'estensione viene fatta così:

$$\delta_{\text{extended}}(q_i) = \begin{bmatrix} 1 \\ z_{-q_1} \\ \vdots \\ z_{-q_H} \end{bmatrix}$$

In equazioni diventano:

$$z_{-q_0} = X$$

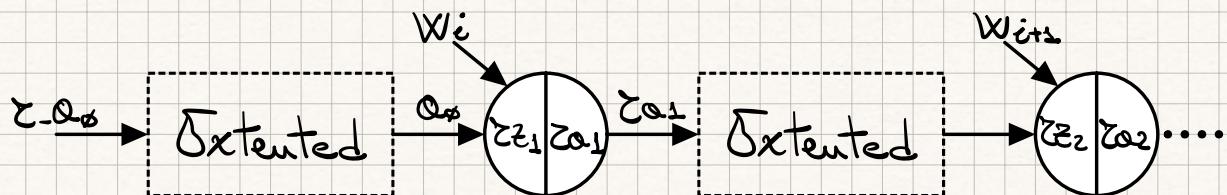
$$q_0 = \delta_{\text{extended}}(z_{-q_0})$$

$$z_{-z_1} = W_1 \cdot q_0$$

$$z_{-q_1} = \text{linc}(z_{-z_1})$$

$$q_1 = \delta_{\text{extended}}(z_{-q_1})$$

$$z_{-z_2} = W_2 \cdot q_1$$



Alla fine la nostra uscita sarà: (Se l'ultimo livello ha attivazione lineare)

$$\hat{y} = W_2 \cdot q_1 = W_2 \cdot \delta_{\text{xt}}(z_{-q_1})$$

Se facciamo tutti i conti, si può ridurre l'intera rete
a una sola matrice dei pesi:

$$\hat{y} = W_2' W_1' x$$

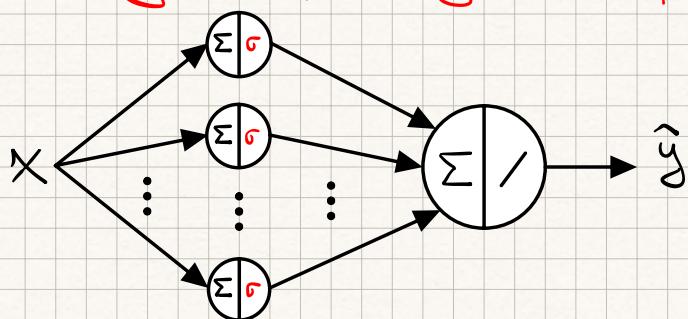
Note: Una W per ogni layer.

Se usiamo solo funzioni lineari, abbiamo trasformazioni
lineari, e possiamo collassare $W_1 \dots W_n$ in una sola
 W .

LIVELLO NON LINEARE

Fin ora abbiamo visto come si comporta un MLP con solo trasformazioni lineari. Cosa succede se introduciamo livelli non lineari?

Questo è un **single output regression problem (SORP)**:



All' posto del livello lineare, mettiamo un sigmoid. Così facendo le equazioni diventano:

$$\tau \cdot Q_0 = X$$

$$Q_0 = \text{Sigmoid}(\tau \cdot Q_0)$$

$$\tau \cdot z_1 = W_1 \cdot Q_0$$

$$\tau \cdot Q_1 = \sigma(\tau \cdot z_1)$$

$$Q_1 = \text{Sigmoid}(\tau \cdot Q_1)$$

$$\tau Q_2 = \tau \cdot z_2 = W_2 \cdot Q_1$$

Se il problema che vogliamo risolvere è per sua natura non lineare, allora questa introduzione può aiutarci.

Perché è importante il livello non lineare?

- 1) Altrimenti servirebbero molti più layer alla rete.
- 2) Se la relazione fra input e output è non lineare, così riusciamo a catturarla meglio.

FORWARD PASS

È il modo in cui la rete, preso un ingresso x , restituisce il risultato \hat{y} . Possiamo vederlo come una funzione f :

$$\hat{y} = f(x, w_1 \dots w_H)$$

Esempio

Alleniamo la rete su un MINI-BATCH del dataset grande B

$$X = \begin{bmatrix} x_1 \\ \vdots \\ x_B \end{bmatrix} \in \mathbb{R}^{B \times D} \quad \text{con } B \gg D$$

Da qui l'ingresso è:

$$\Sigma A_\phi = X^\top$$

$$A_\phi = \text{Ext}(\Sigma A_\phi) = \begin{bmatrix} 1 & \dots & 1 \\ x_1 & \dots & x_B \end{bmatrix} \in \mathbb{R}^{(D+1) \times B}$$

Riceviamo Σz_1 che ora è una matrice:

$$\Sigma z_1 = W_1 \cdot A_\phi \in \mathbb{R}^{H \times B}$$

$$\Sigma A_1 = \sigma(\Sigma z_1)$$

$$A_1 = \text{Ext}(\Sigma A_1) \in \mathbb{R}^{(H+1) \times B}$$

$$\Sigma z_2 = W_2 \cdot A_1$$

Oscrizio Matlab

Scrivere un MLP con il MSE e funzione di attivazione lineare.

```
function [rA2,A1,A0,rZ1] = MLP_MSELIN_forward(X, W1, W2)
    % Compute the forward step

    rA0 = X'; % rA0 is the "reduced A0" and it coincides with the transpose of the tall input matrix (nObs x nInput)
    A0 = MLP_extend(rA0); % A0 = E(rA0). It is the "extended" version of rA0, obtained by it by adding a row of ones as its new first row
    rZ1 = W1*A0; % rZ1 = \sum(W1,A0). It is the pre-activation at layer 1 (the hidden one)
    rA1 = MLP_sigmoid(rZ1); % rA1 = \sigma(rZ1). It is the output of the first layer (the hidden one)
    A1 = MLP_extend(rA1); % A1 = E(rA1). It is the extended version of rA1
    rZ2 = W2*A1; % rZ2 = \sum(W2,A1). It is the pre-activation at layer 2 (the output one)
    rA2 = rZ2; % rA2 is the output of the second layer (the output one). It is reduced, i.e. unextended.

end

function sig = MLP_sigmoid(z)
    sig = 1./(1+exp(-z));

function [X_new] = MLP_extend(X)
    % Extend matrix X by adding the bias (a row of ones as the new first row)
    X_new = ones(size(X,1)+1, size(X,2));
    X_new(2:end,:) = X;
end
```