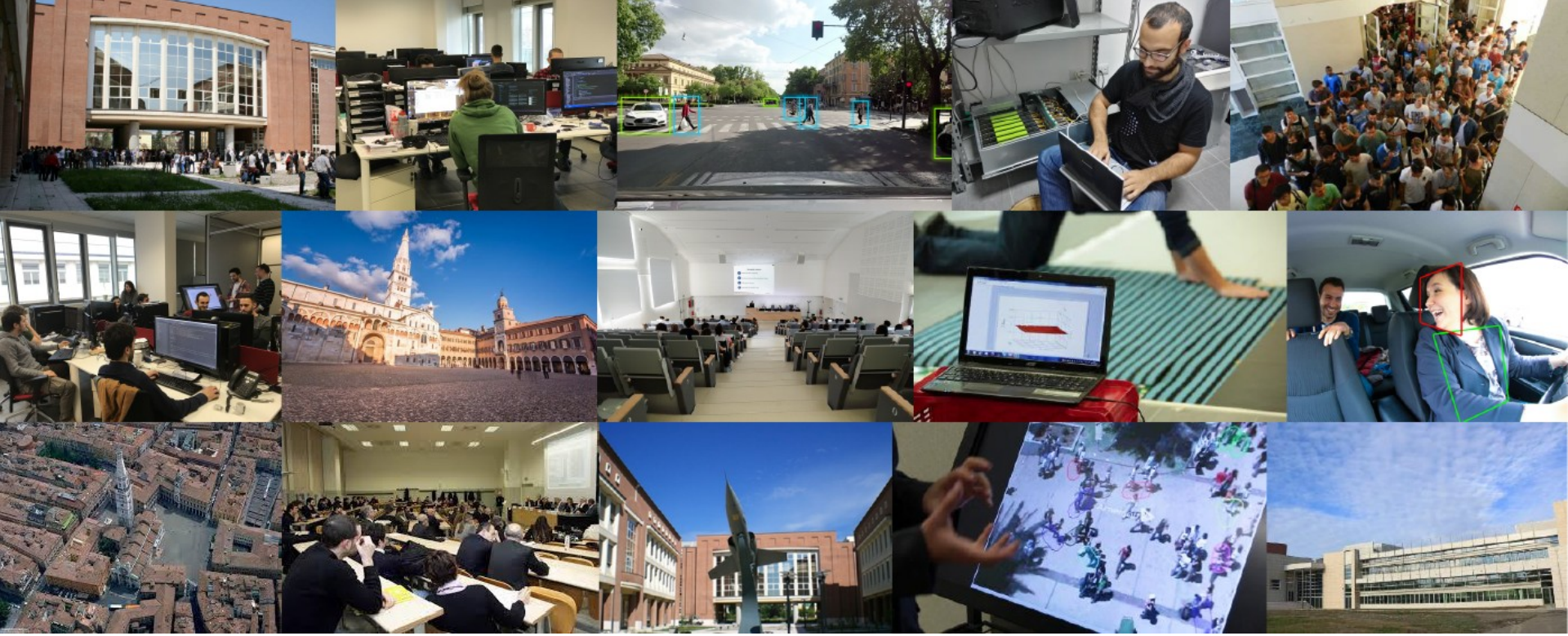


# MACHINE LEARNING
## - Reti Neurali -

AImage Lab

UNIMORE UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

# Notazione

| | | |
|---|---|---|
| $x_j$, j = 1, …, n | → | features |
| $\mathbf{x} = [x_0, \ldots, x_n]$ | → | feature vector ($x_0$=1 per comodità notazionale) |
| $\mathbf{t} = [t_1, \ldots, t_k]$ | → | *vettore* di ground-truth per le variabili target |
| $(\mathbf{x}^{(i)}, \mathbf{t}^{(i)})$, i = 1, …, N | → | training sample |
| **w, v** | → | vettori di parametri |
| $\mathbf{w}^T$ | → | vettore **w** trasposto |
| W, V | → | matrici di parametri |
| $\theta = \{W, \boldsymbol{v}\}$ | → | insieme di più vettori e matrici di parametri |
| J() | → | loss function |

# Artificial Neural Networks

AImage Lab
UNIMORE UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

Le Artificial Neural Network (ANN) sono dei modelli *parametrici* ispirati alle reti neurali *biologiche* e possono essere usate in maniera molto flessibile sia come modelli discriminativi che generativi per task discriminativi (e.g., regressione e classificazione) e per task generativi (e anche per task unsupervised)

In questo corso vedremo il caso particolare (ma fondamentale) di ANN discriminative applicate a problemi discriminativi

Nella trattazione che faremo studieremo i vari modelli di ANN in linea generale e poi vedremo che gli stessi modelli, con piccole variazioni, possono essere usati sia per la regressione che per la classificazione

Le ANN sono importanti per vari motivi, tra cui:

- La loro generalità li rende un framework capace di unificare vari aspetti del ML
- Sono un metodo molto potente per risolvere problemi di ML
- Sono alla base dell'IA moderna ("Deep Learning")

# Le tre «vite» delle Artificial Neural Networks

AImage Lab
UNIMORE UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

Le ANN hanno vissuto fasi alterne di «popolarità» e «decadenza»:

- 1957, prima proposta: Percettrone [Rosenblatt]
- 1969, limiti del Percettrone [Minsky-Papert]. Prima “morte” delle ANN
- 1986-1987, Backpropagation [Hinton e altri autori indipendenti]. Rinascita delle ANN
- Metà anni ‘90: le SVM (e poi altri metodi, tra cui le Random Forest) soppiantano le ANN perchè sembrano essere più efficaci, sopratutto con dataset di training piccoli. Seconda “morte” delle ANN
- 2006-2012, nascita del Deep Learning, grazie a nuovi algoritmi, hardware parallelo e alla disponibilità dei “Big Data”. Seconda “rinascita” delle ANN

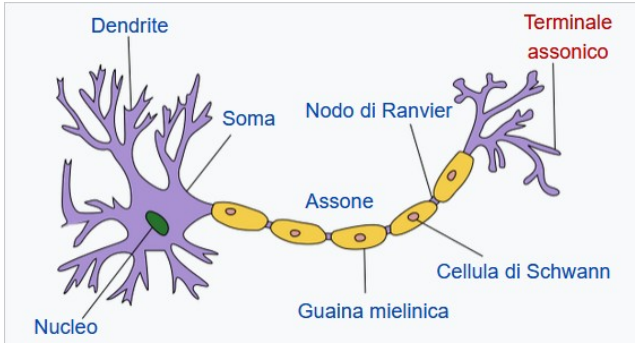
Iniziamo ad esplorare le ANN partendo dalla metafora biologica a cui si ispirano, ovvero i neuroni del cervello animale

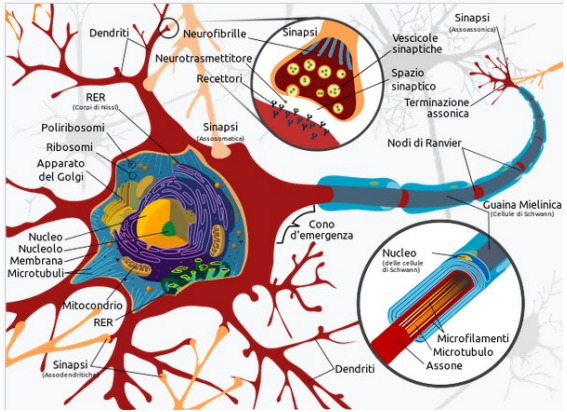
# Neurone biologico

AImage Lab
UNIMORE UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

I componenti principali di un neurone biologico sono:

- Il nucleo, in cui c'è una membrana che, polarizzandosi e depolarizzandosi, genera degli impulsi elettrici
- L'assone, che è una specie di "coda" con cui gli impulsi uscenti dal nucleo vengono trasmessi ad altri neuroni
- I dendriti, che hanno diramazioni simili a un albero, e che ricevono segnali in entrata dagli assoni ad essi collegati
- Le sinapsi, che collegano assoni e dendriti facendo passare il segnale elettrico attraverso neurotrasmettitori. La quantità di questi neurotrasmettitori è modificabile nel tempo e questa modifica è alla base della memoria
- Nel nucleo gli impulsi elettrici provenienti dai neuroni connessi vengono sommati. Se superano una certa soglia fissa la membrana si polarizza ed emette un segnale che sarà diffuso dall'assone



# Artificial Neural Networks

AImage Lab
UNIMORE UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

Una ANN è un modello matematico che imita (fino ad un certo punto…) il funzionamento del cervello biologico

Senza perdita di generalità, una ANN è un grafo in cui:

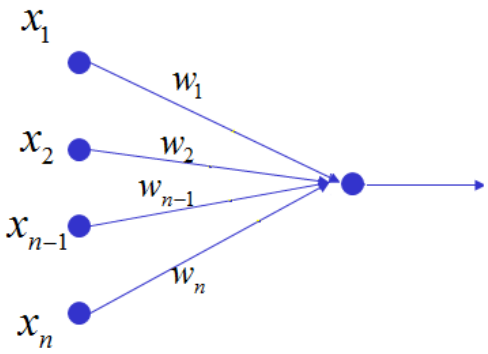
- I nodi (o “unit”), con le loro “funzioni di attivazione” associate imitano il comportamento della soglia di attivazione del nucleo dei neuroni biologici
- Gli archi imitano le connessioni tra neuroni (assone-dendriti)
- I pesi (parametri) associati ad ogni arco imitano le sinapsi che regolano la trasmissione del segnale cerebrale e sono modificabili durante il training

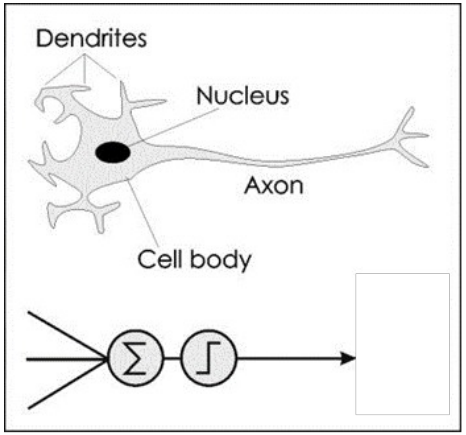
# La rete neurale più semplice: il Percettrone

AImage Lab
UNIMORE UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

Il Percettrone (Rosenblatt 1957) può essere visto come il blocco costitutivo di ANN dall'architettura più complessa e simula il comportamento di un singolo neurone biologico, con i suoi ingressi multipli e la sua uscita dipendente da una soglia di attivazione

Strutturalmente, il Percettrone è costituito da *n* nodi di ingresso, uno per ogni feature, e un solo nodo d'uscita. Gli *n* nodi di input sono collegati col nodo di output e un peso è associato ad ogni arco

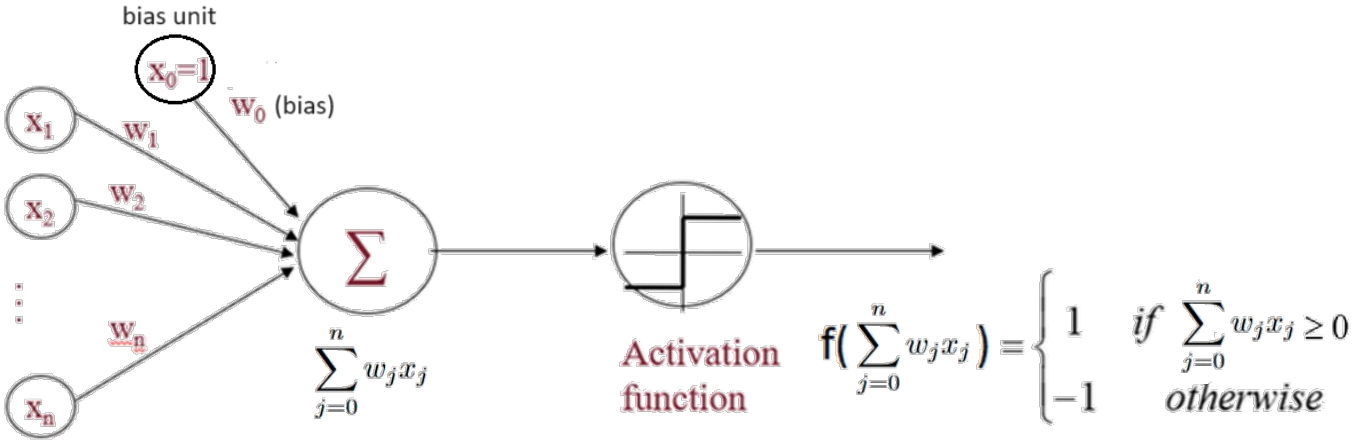


# Percettrone

Il modello parametrico del Percettrone è: $\boldsymbol{w}^T \boldsymbol{x}$, dove:
$\boldsymbol{w} = [w_0, w_1, \ldots, w_n]$ e $\boldsymbol{x} = [1, x_1, \ldots, x_n]$

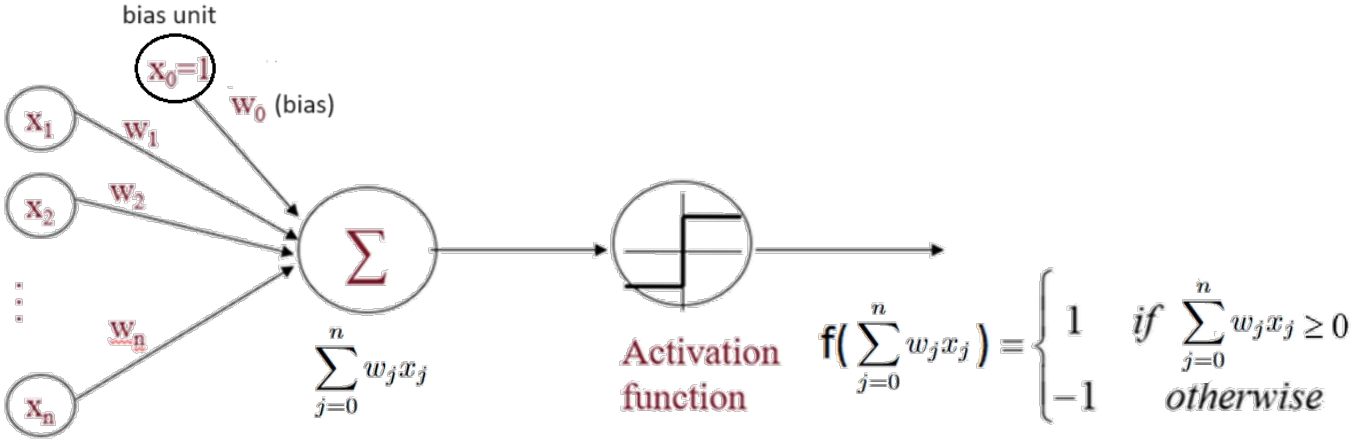
La parte parametrica, quindi, è un modello lineare



# Percettrone

La feature fittizia $x_0 = 1$, svolge lo stesso ruolo dell' «intercept term» nella terminologia della Linear Regression ed è associata ad un $n+1$-esimo nodo detto «bias unit»

La costante $w_0$ che è associata al nodo collegante il bias unit con l'output unit è detta «bias»

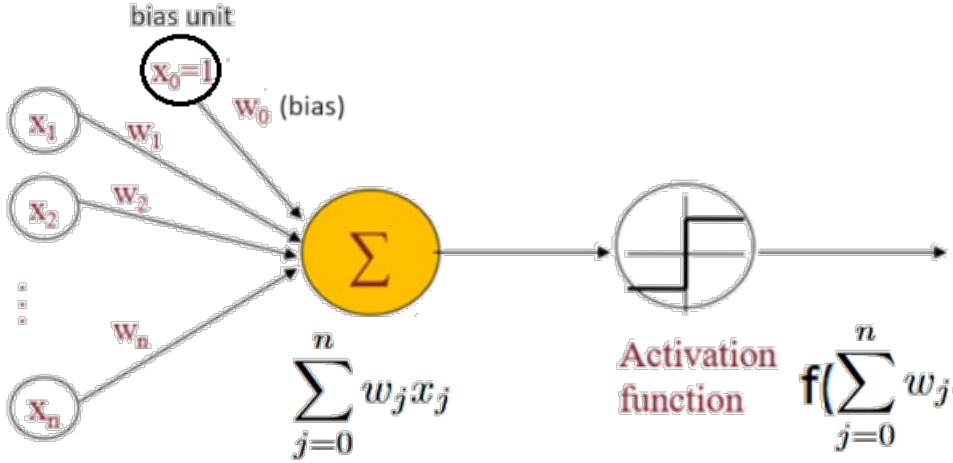


# Input function

La «funzione di input» (*input function*) è una combinazione lineare delle feature $x_1, \ldots, x_n$ tramite i *pesi* (ovvero i parametri):

$$w_0 + w_1 x_1 + \ldots + w_n x_n = \sum_{j=0}^{n} w_j x_j = \boldsymbol{w}^T \boldsymbol{x}$$

L'input function, quindi, è esattamente uguale alla funzione ipotesi della Linear Regression o alla score function di un classificatore lineare (e.g., nella Logistic/Softmax Regression o nelle SVM senza kernel)



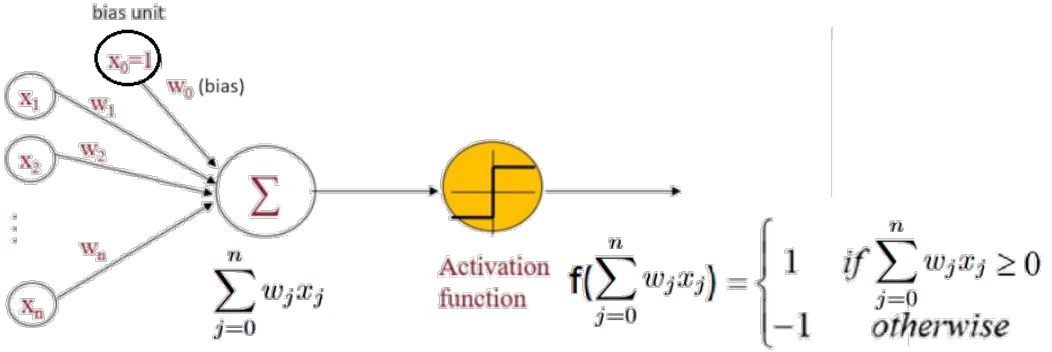

$$f\left(\sum_{j=0}^{n} w_j x_j\right) = \begin{cases} 1 & \text{if } \sum_{j=0}^{n} w_j x_j \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

# Activation function

La funzione (non lineare) *f(),* detta *activation function* (o “nonlinearità”) ha lo scopo di trasformare il valore della funzione di input in maniera non lineare e la sua importanza sarà chiara in seguito

Storicamente il Percettrone era visto come un classificatore binario e la funzione di attivazione corrispondeva alla decision rule di un classificatore lineare (v. figura sotto)

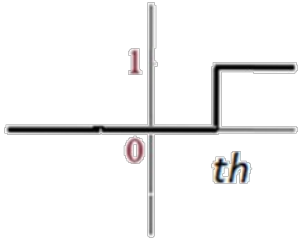
Activation function più “moderne” sono in genere totalmente o parzialmente differenziabili e il loro output è detto *activation*



# Tipiche funzioni di attivazione

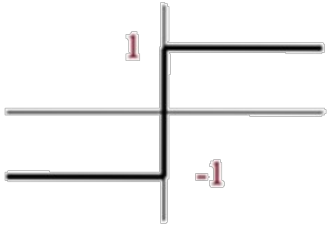
- Step function :

$$step_{th}(x) = \begin{cases} 1 & x > th \\ 0 & otherwise \end{cases}$$



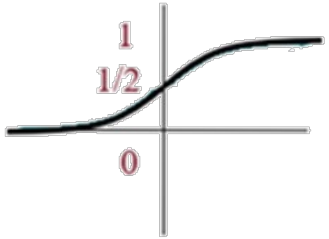

- Sign function :

$$sign(x) = \begin{cases} +1 & x \geq 0 \\ -1 & otherwise \end{cases}$$




- Sigmoid/logistic function (*differenziabile*):

$$sigmoid(x) = \frac{1}{1+e^{-x}}$$

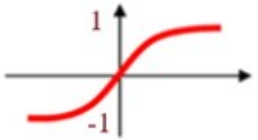


# Tipiche funzioni di attivazione

AImage Lab
UNIMORE UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

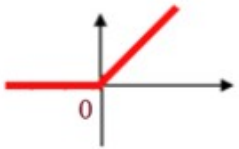
Tangente iperbolica
(differenziabile):

$$tanh(x) = \frac{sinh(x)}{cosh(x)}$$




Rectified Linear Unit
(parzialmente differenziabile):

*ReLU(x) = max (0, x)*



# Analogie tra il Percettrone e altri modelli parametrici lineari



Storicamente la funzione di attivazione del Percettrone era la sign function che, quindi, coincide con la decision rule dei classificatori lineari o delle SVM

Più in generale, noi col termine “Percettrone” intenderemo la struttura vista finora con una nonlinearità qualsiasi associata al nodo di output

È interessante notare che, ad esempio, se usassimo la logistic function, otterremmo lo stesso modello della Logistic Regression

Eliminando del tutto la funzione di attivazione (o usando la “funzione identità”), otteniamo il modello della Linear Regression o, equivalentemente, la Score Function di un classificatore lineare

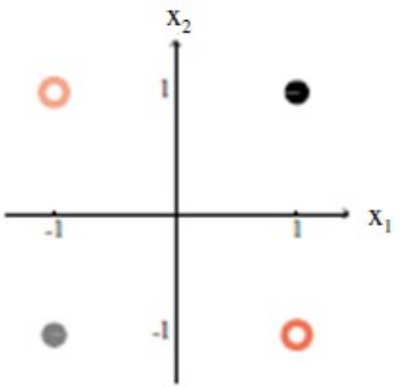
# Limiti del Percettrone

In sostanza, al di là della terminologia peculiare delle ANN e dell'analogia biologica, un Percettrone non è altro che un classificatore lineare

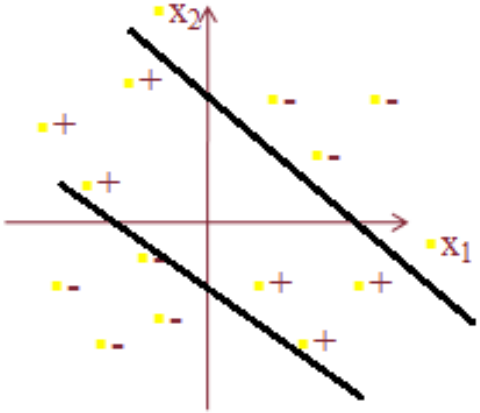
Quindi ha un bordo di decisione che è un iper-piano. E, di conseguenza, non può classificare classi linearmente non separabili (farebbe "underfitting")

# Limiti del Percettrone

Questa cosa fu dimostrata per la prima volta nel 1969 da Minsky e Papert (“Perceptrons: an introduction to computational geometry”), utlizzando il “problema dello XOR”




Funzione XOR




Decision regions non connesse

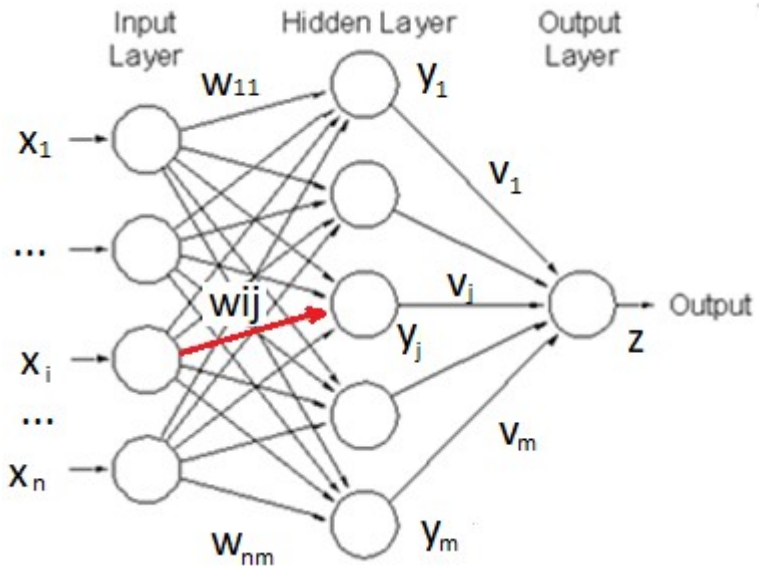
# Limiti del Percettrone

AImage Lab
UNIMORE UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

Abbiamo già incontrato il problema della separazione lineare quando abbiamo studiato le SVM
Nel caso delle SVM abbiamo visto che il problema viene affrontato col “trucco del kernel”
Il kernel, nelle SVM, serve a simulare la creazione di uno spazio delle feature diverse in cui i feature vector delle due classi siano (sperabilmente) linearmente separabili
Abbiamo anche osservato che, usando il kernel nelle SVM, non c’è bisogno di creare esplicitamente la funzione di mapping $\boldsymbol{\phi(x)}$ che dovrebbe trasformare $\boldsymbol{x}$ in un feature vector definito in uno spazio diverso
Lo svantaggio del kernel, però, stà nel fatto che deve essere “scelto a mano”, valori dei suoi iper-parametri compresi
E non c’è nessuna garanzia che un determinato kernel, con determinati valori dei suoi iper-parametri, porti ad una separazione lineare delle classi
Ebbene, le ANN risolvono il problema della non separabilità lineare *costruendo esplicitamente* $\boldsymbol{y} = \boldsymbol{\phi(x)}$
Per l’esattezza, usando le ANN, è possibile rappresentare la funzione $\boldsymbol{\phi(x)}$ con una funzione (analitica) parametrica, quindi *ottimizzabile automaticamente* usando il Gradient Descent
$\boldsymbol{\phi(x)}$ viene realizzato usando “layer” aggiuntivi

# Multilayer Perceptron (MLP)

Un Multilayer Perceptron (MLP) è una ANN composta da una serie di Percettroni
In particolare, esistono 3 “layer“, ognuno composto da un insieme di neuroni (v. figura a lato)
Le connessioni sono possibili solo tra layer consecutivi
Ogni neurone del primo layer (*input layer*) è connesso con tutti i neuroni del secondo layer (*hidden layer*)
A sua volta, ogni neurone dell’hidden layer è connesso con il neurone di output (*output layer*)
Ad ogni connessione è associato un peso (parametro)



# Multilayer Perceptron (MLP)

L'hidden layer implementa la funzione $\boldsymbol{y} = \phi(\boldsymbol{x})$, ovvero la trasformazione del feature space che stiamo cercando di costruire

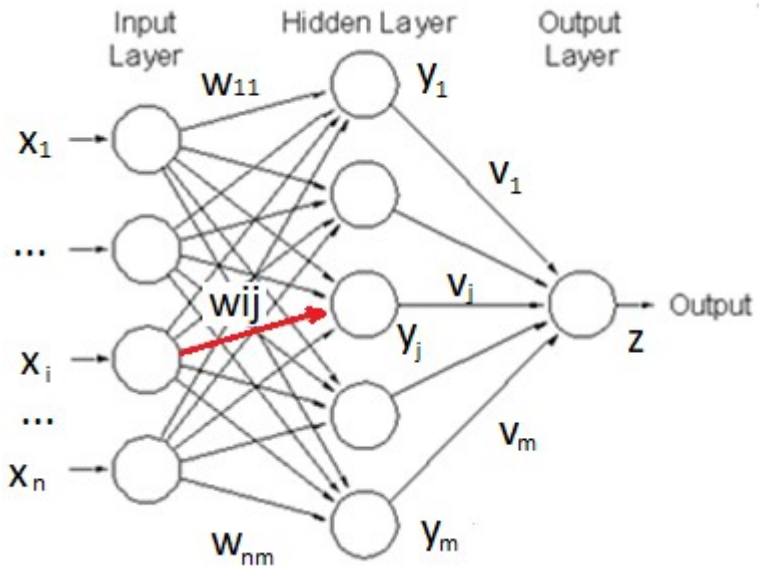
Infatti $\boldsymbol{y} = [y_1, \ldots, y_j, \ldots, y_m]$ è un vettore di $m$ valori reali, dove:

- $y_j$ è l'output (l'activation) del j-esimo neurone dell'hidden layer
- Tale neurone è collegato con tutti i neuroni di input, per cui è anch'esso un Percettrone, nel senso che calcola: $y_j = f(\boldsymbol{w}_j^T \boldsymbol{x})$, dove:

1. $f()$ è la nonlinearità
2. $\boldsymbol{w}_j$ è il vettore dei pesi $\boldsymbol{w}_j = [w_{1j}, w_{2j}, \ldots w_{ij}, \ldots, w_{nj}]$

- Per cui $\boldsymbol{y} = \phi(\boldsymbol{x}) = [f(\boldsymbol{w}_1^T \boldsymbol{x}), \ldots, f(\boldsymbol{w}_j^T \boldsymbol{x}), \ldots, f(\boldsymbol{w}_m^T \boldsymbol{x})]$
- $\phi(\boldsymbol{x})$ è una funzione da $R^n$ ad $R^m$ ed è parametrica
- I suoi parametri sono i vettori $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_m$, che posso "impilare" nella matrice $W$ di dimensioni $m$ X $n$:

$$W = \begin{bmatrix} \boldsymbol{w}_1^T \\ \ldots \\ \boldsymbol{w}_m^T \end{bmatrix}$$



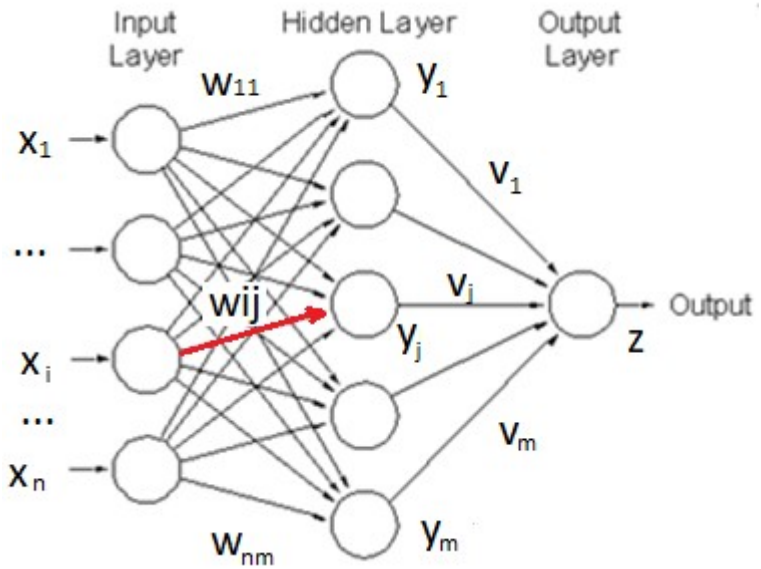
# Multilayer Perceptron (MLP)

*W* è detta la matrice dei pesi delle connessioni “input-to-hidden”

***y***, invece, è detto “activation vector”, in quanto è un vettore di activation

Con un leggero abuso notazionale, considerando *f()* come una funzione *element-wise* applicata ad un vettore, potremmo anche scrivere:

***y*** = *f(W**x**)*



# Multilayer Perceptron (MLP)

***y*** viene quindi dato in input al secondo layer, anch'esso un Percettrone

L'output *z* è uno scalare, ed è ottenuto con la solita regola del Percettrone, ovvero:

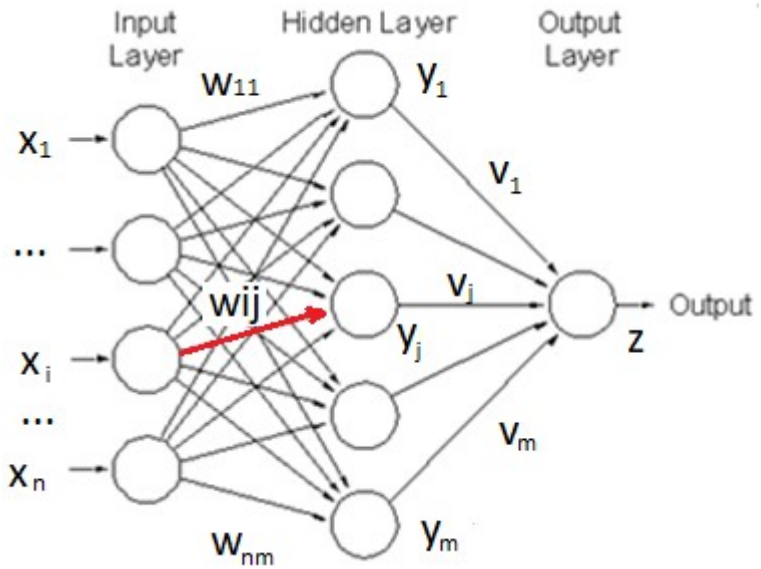
$z = f(\boldsymbol{v}^T \boldsymbol{y})$,

dove $\boldsymbol{v} = [v_1, \ldots, v_m]$ è il vettore dei pesi delle connessioni "hidden-to-output"

La cosa importante da notare è che il Percettrone dell'ultimo layer prende in input ***y*** e non ***x***!

Ovvero, prende in input una versione modificata di ***x***

Tra un pò vedremo perchè ciò è fondamentale per risolvere problemi non-lineari

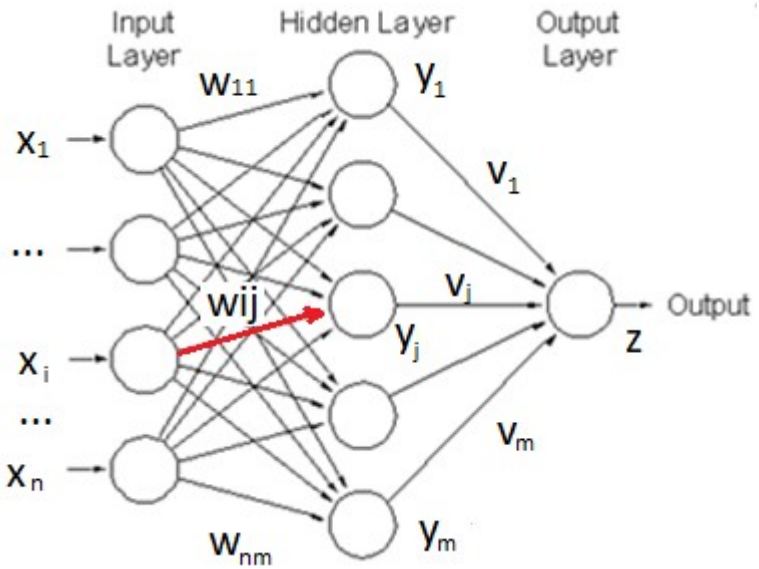


# Multilayer Perceptron (MLP)

Intanto scriviamo la funzione complessiva calcolata da questo MLP:

$z = f(\boldsymbol{v}^T f(W\boldsymbol{x}))$

Si tratta di una funzione “analatica” (composta), i cui parametri complessivi sono $\{W, \boldsymbol{v}\}$



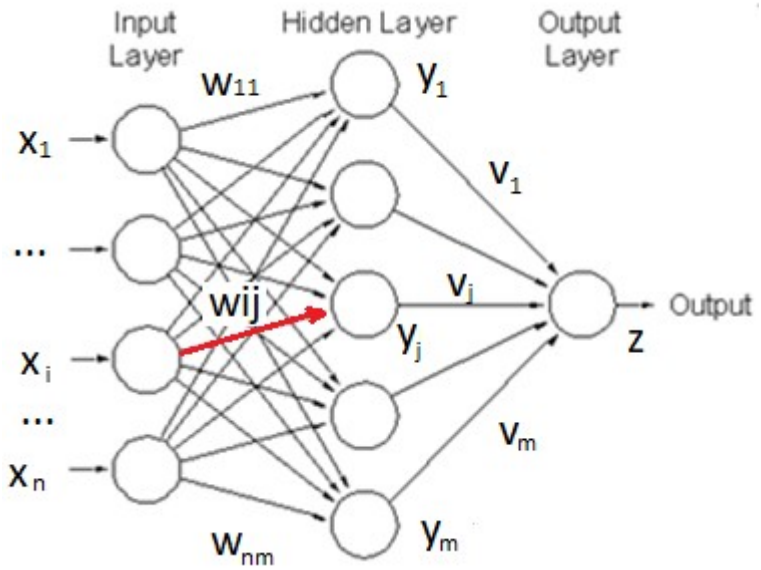
# Multilayer Perceptron (MLP)

Potrei anche usare nonlinearità diverse nei due layer (e.g., *f()* e *g()*), ottenendo:

$z = g(\mathbf{v}^T f(W\mathbf{x}))$

Se *f()* e *g()* sono differenziabili, allora posso usare il Gradient Descent per trovare {*W*,***v***}

Come addestrare un MLP lo vedremo in seguito, per ora concentriamoci sulle *capacità di rappresentazione* dell'MLP, assumendo di poter trovare "magicamente" il valore dei pesi



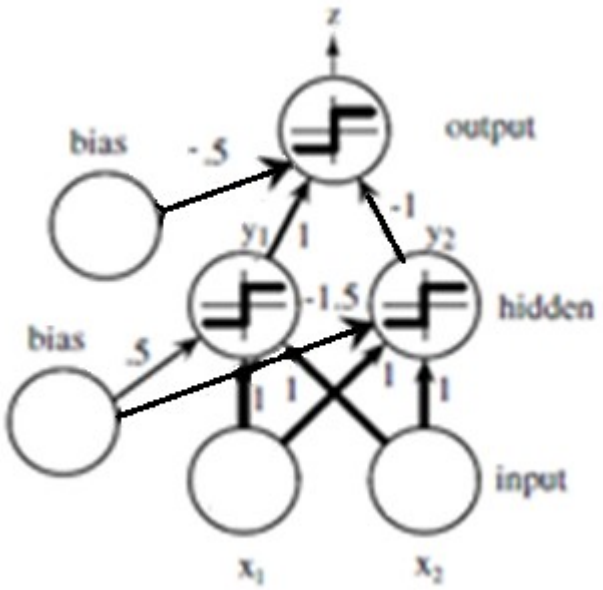
# Una possibile soluzione al problema dello XOR

Nella figura a lato è mostrato un MLP che risolve il problema dello XOR

Il valore dei pesi è stato trovato manualmente perchè lo scopo è dimostrare che esiste una ANN che può risolvere i problemi di classificazione non lineare del Percettrone (per ora non stiamo considerando il training)

Esistono infinite altre possibili combinazioni di pesi e questa scelta è solo un esempio

Per lo stesso motivo, le nonlinearità sono delle sign function (per ora non c'interessa se siano differenziabili o meno)



# Una possibile soluzione al problema dello XOR

Calcoliamo $\boldsymbol{y} = [y_1, y_2] = \phi(\boldsymbol{x}) = [f(\boldsymbol{w}_1^T \boldsymbol{x}), f(\boldsymbol{w}_2^T \boldsymbol{x})]$:

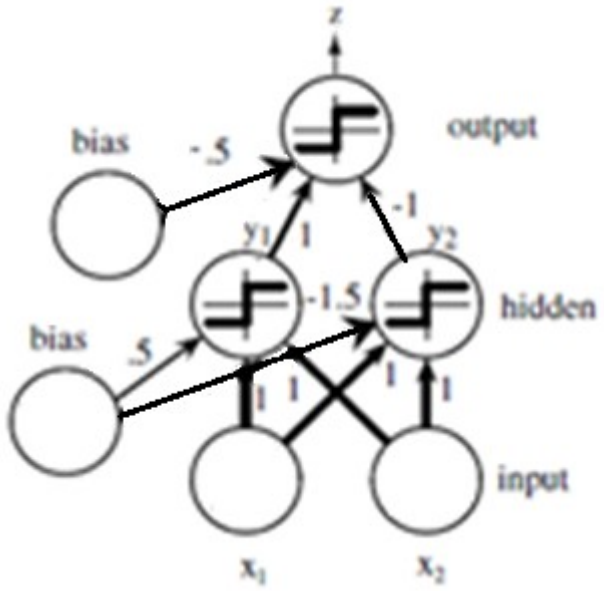
$$y_1 = \begin{cases} 1 & \text{if } x_1 + x_2 + 0.5 \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

$$y_2 = \begin{cases} 1 & \text{if } x_1 + x_2 - 1.5 \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

e $z = f(\boldsymbol{v}^T \boldsymbol{y})$:

$$z = \begin{cases} 1 & \text{if } y_1 - y_2 - 0.5 \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

Ricordiamoci che $f(\boldsymbol{v}^T \boldsymbol{y})$ è definita nello spazio i cui assi sono $y_1$ e $y_2$

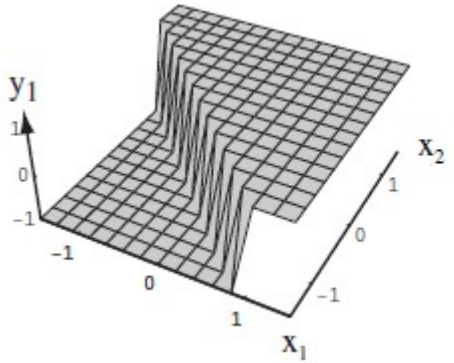


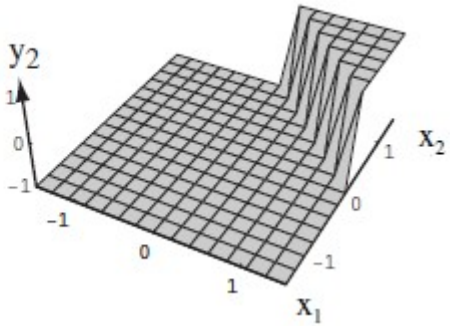
# Una possibile soluzione al problema dello XOR

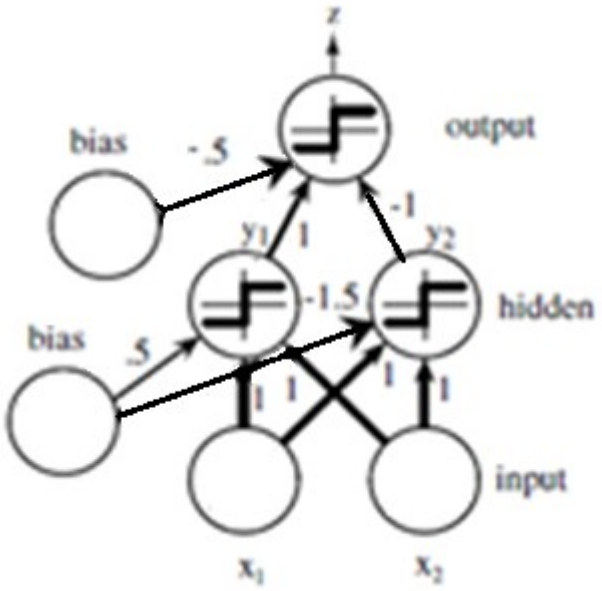
Calcoliamo ora i decision boundary e le decision region sia di $y_1 = f(\boldsymbol{w}_1^T \boldsymbol{x})$ che di $y_2 = f(\boldsymbol{w}_2^T \boldsymbol{x})$ :

- $x_2 = - x_1 - 0.5$ $\quad (y_1)$
- $x_2 = - x_1 + 1.5$ $\quad (y_2)$

I grafici delle funzioni $f(\boldsymbol{w}_1^T \boldsymbol{x})$ ed $f(\boldsymbol{w}_2^T \boldsymbol{x})$ sono mostrati sotto



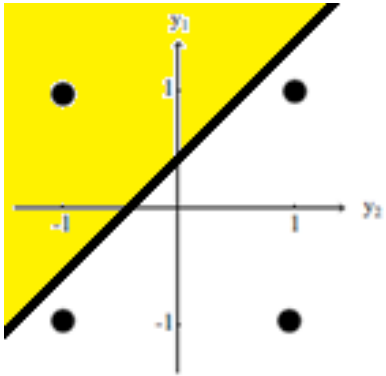
# Una possibile soluzione al problema dello XOR

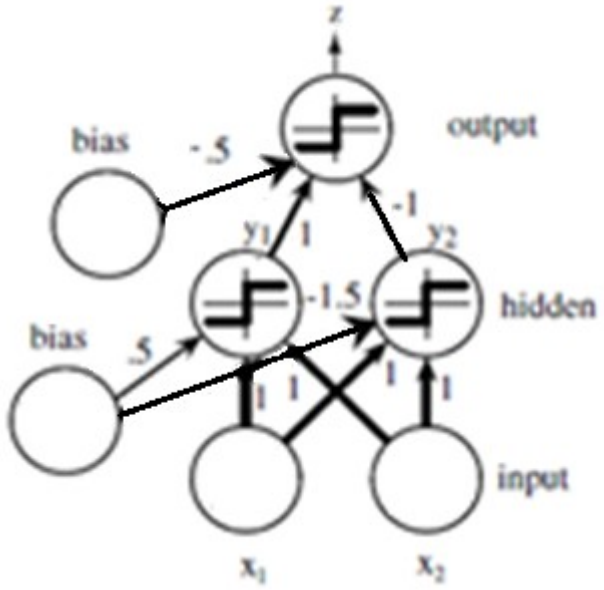
Calcoliamo il decision boundary di $z = f(\mathbf{v}^T \mathbf{y})$:

- $y_1 = y_2 + 0.5$

Il grafico qui sotto mostra le decision region di $f(\mathbf{v}^T \mathbf{y})$ (in giallo quando $z = 1$)

Attenzione: in quest'esempio $y_1$ ed $y_2$ possono assumere solo valori discreti (indicati dai 4 punti neri) a causa delle nonlinearità di tipo binario che abbiamo scelto (sign function)
Se invece l'output delle nonlinearità è continuo, $y_1$ e $y_2$ definiranno uno spazio continuo ($R^2$)

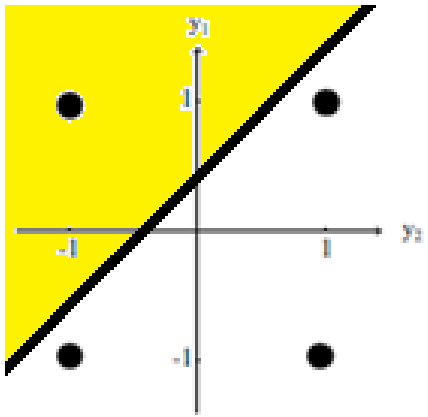


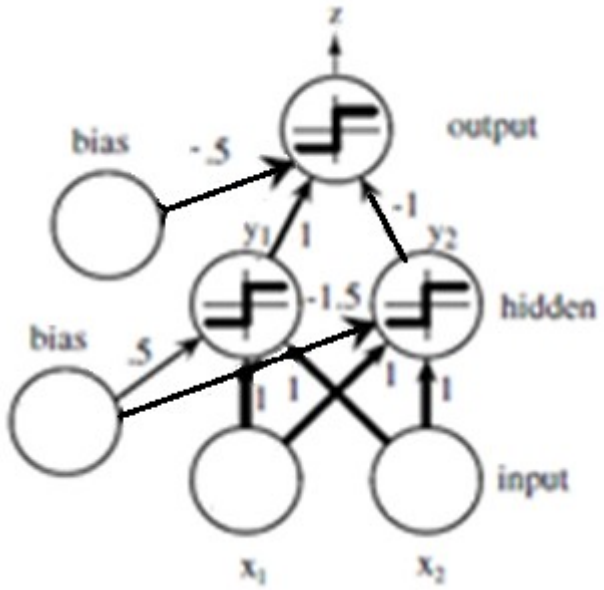



# Una possibile soluzione al problema dello XOR

$$z = \begin{cases} 1 & \text{if } y_1 - y_2 - 0.5 \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

Quindi $z = 1$ se e solo se $y_1 = 1$ *and* $y_2 = -1$



# Una possibile soluzione al problema dello XOR

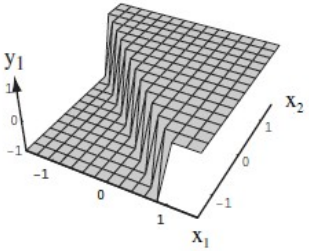
Andiamo adesso a disegnare il grafico di $z = f(\mathbf{v}^T f(W\mathbf{x}))$

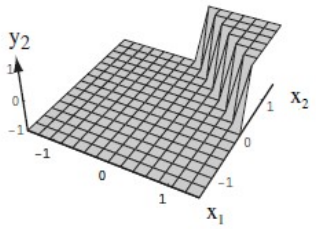
A differenza della funzione $f(\mathbf{v}^T \mathbf{y})$, il cui input sono i valori di $y_1$ ed $y_2$, $f(\mathbf{v}^T f(W\mathbf{x}))$ prende in input le feature iniziali $x_1$ ed $x_2$

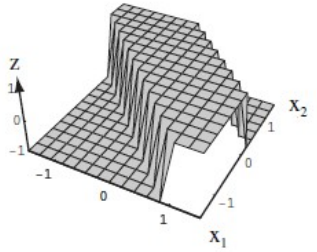
Per costruire il grafico di $f(\mathbf{v}^T f(W\mathbf{x}))$ devo tener a mente che $z = 1$ se e solo se $y_1 = 1$ *and* $y_2 = -1$. Nella figura a destra ciò è ottenuto considerando l'intersezione di questi due eventi

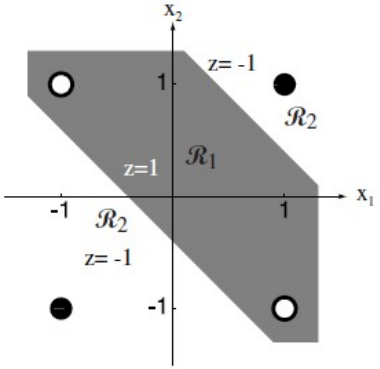
Il grafico di $f(\mathbf{v}^T f(W\mathbf{x}))$ mostra che una delle decision region ($R_2$) di questo MLP non è connessa

Perciò $f(\mathbf{v}^T f(W\mathbf{x}))$ riesce a rappresentare partizioni dello spazio delle feature ($x_1$ ed $x_2$) **non linearmente separabili**!



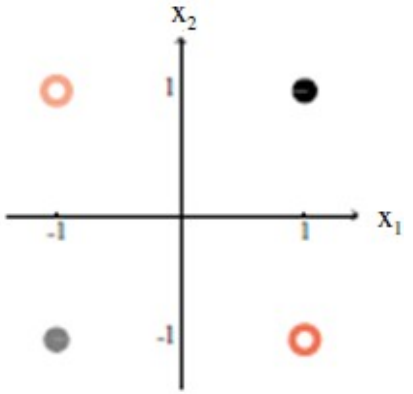
# Trasformazione delle feature

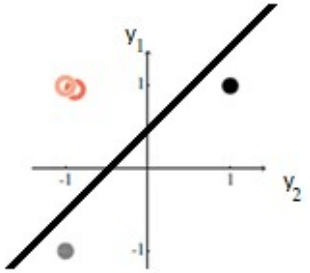
Ciò che ha permesso la separazione delle due classi è stata la trasformazione dello spazio delle feature ($x_1$ e $x_2$) in un **nuovo spazio delle feature** ($y_1$ e $y_2$)

$$y_1 = \begin{cases} 1 & \text{if } x_1 + x_2 + 0.5 \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

$$y_2 = \begin{cases} 1 & \text{if } x_1 + x_2 - 1.5 \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

$$z = \begin{cases} 1 & \text{if } y_1 - y_2 - 0.5 \geq 0 \\ -1 & \text{otherwise} \end{cases}$$




Questa trasformazione è stata operata dall'hidden layer:
$\boldsymbol{y} = [y_1, y_2] = \phi(\boldsymbol{x}) = [f(\boldsymbol{w_1}^T \boldsymbol{x}), f(\boldsymbol{w_2}^T \boldsymbol{x})]$

# Potere espressivo del MLP

AImage Lab
UNIMORE UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

Generaliziamo quello che abbiamo visto finora

**Teorema di approssimazione universale** [Cybenko e altri]: Ogni funzione continua $h(\boldsymbol{x})$ con dominio in un insieme compatto può essere approssimata con un qualsiasi desiderato livello di accuratezza da un MLP con un unico hidden layer, funzioni di attivazione sigmoidali, un appropriato numero (finito) di neuroni nell'hidden layer e un appropriato valore dei pesi

Ciò, in sostanza, significa che un MLP che calcola $g(\boldsymbol{v}^T f(W\boldsymbol{x}))$ può essere usato per approssimare *qualsiasi funzione* $h(\boldsymbol{x})$ da $R^n$ ad $R$ *con un qualsiasi desiderato livello di precisione*

In altre parole, $g(\boldsymbol{v}^T f(W\boldsymbol{x}))$ può approssimare una qualsiasi (ideale) funzione di predizione dei dati e, quindi, per un qualsiasi task, esiste sempre un MLP che può risolverlo *con un sufficiente grado di precisione senza fare underfitting*!

Il teorema può essere esteso a nonlinearità diverse dalle funzioni sigmoidali

# Potere espressivo del MLP: regressione

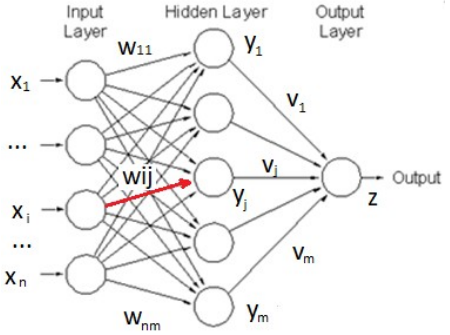
Ad esempio, nel caso della regressione, da un dataset di training *T* contenente coppie di associazioni variabili indipendenti-variabile target (***x***, *z*), lo scopo è trovare una funzione di predizione *z* = *h*(***x***) che rappresenti la relazione tra ***x*** e *z*

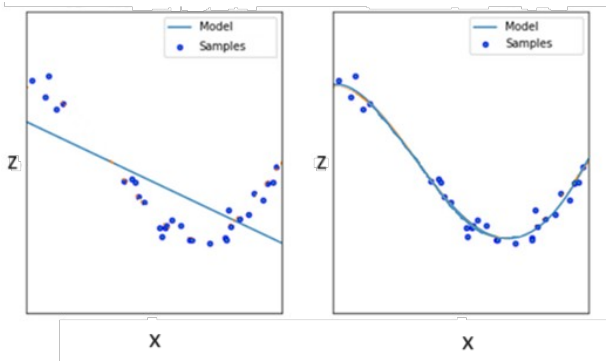
Il teorema di approssimazione universale ci dice che posso usare un MLP per costruire una funzione *z* = *g*(*f*(***x***)) che approssimi questa *h*() ideale, ovvero che approssimi la distribuzione dei dati in *T* con un livello di precisione arbitrario (v. figura)

Ad esempio, potremmo usare la tangente iperbolica per i neuroni dell'hidden layer e la funzione identità (= nessuna funzione di attivazione) per l'output layer:

- *f*(*x*) = *tanh*(*x*)
- *g*(*x*) = *x* (funzione identità)

Quindi *g*(***v***$^T$ *f*(*W****x***)) = ***v***$^T$ *tanh*(*W****x***) (*tanh*() è applicata element-wise)

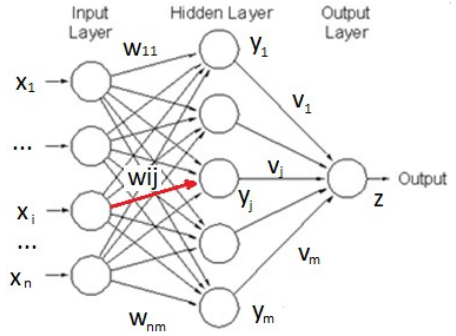


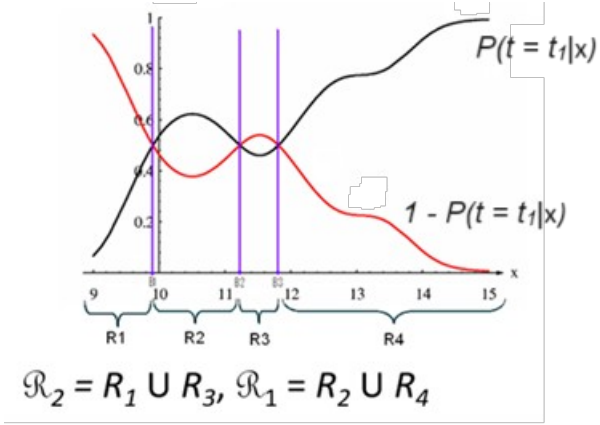



# Potere espressivo del MLP: classificazione binaria

Nel caso della classificazione binaria, senza perdita di generalità, dato *T*, potremmo definire come obiettivo quello di costruire una funzione di predizione $h(\boldsymbol{x})$ che rappresenti la posterior $h(\boldsymbol{x}) = P(t = t_1|\boldsymbol{x})$, dove $t_1$ è la label di una delle due classi e $h(\boldsymbol{x})$ è una funzione che calcola la probabilità che ad un dato feature vector $\boldsymbol{x}$ corrisponda la classe $t_1$

Il terorema di approssimazione universale ci dice che, indipendentemente dalla forma specifica di questa posterior e da come siano disposte le classi nello spazio delle feature, *h()* può essere approssimata da un MLP con un livello di precisione arbitrario



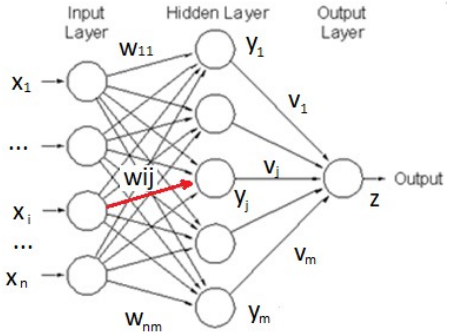
# Potere espressivo del MLP: classificazione binaria

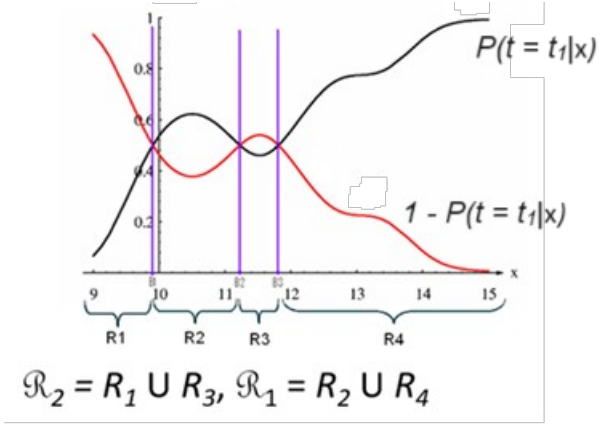
Ad esempio, come nel caso della regressione, potremmo usare *tanh()* per l'hidden layer. Per l'output layer, invece, usiamo la funzione logistica, in modo da avere $g(\mathbf{v}^T f(W\mathbf{x})) = P(t = t_1|\mathbf{x}) \in [0,1]$:

- $f(x) = tanh(x)$
- $g(x) = 1 / (1 + e^{-x})$

Il teorema ci dice che, usando queste nonlinearità, è *possibile* costruire un MLP che *non faccia underfitting* per un task di classificazione binaria

N.B.: passando dal task di regressione a quello di classificazione binaria abbiamo cambiato solo la nonlinearità dell'output layer!



# Potere espressivo del MLP



In base al teorema di approssimazione universale sappiamo quindi che un MLP, è, *potenzialmente*, uno strumento abbastanza potente *per risolvere qualsiasi task di ML senza fare underfitting* (mentre non ci dice niente a proposito dell'overfitting…)

"Potenzialmente" vuol dire che, dato un qualsiasi task, sappiamo che un MLP per risolverlo senza fare underfitting **esiste**, anche se non è detto che siamo in grado di trovarlo…

Il teorema, ad esempio, non ci dice quanti neuroni deve avere l'hidden layer (*m*)

*m* è un iper-parametro il cui valore che deve essere scelto utilizzando un validation set

Fissare *m* non è tuttavia un problema, perchè, di solito, molti valori diversi di *m* portano a risultati soddisfacenti (e.g., usando *m* dello stesso ordine di grandezza di *n*): *m* non è un iper-parametro sensibile

La lacuna più importante del teorema, però, è che non fornisce nessuna garanzia nè alcun metodo operativo per trovare un valore appropriato dei pesi…

Per risolvare il problema dello XOR, i pesi li abbiamo scelti "a mano", cosa ovviamente non fattibile in generale

# Backpropagation



Infatti, nonostante gli MLP fossero già noti negli anni ’60, non era chiaro come addestrarli, ovvero come trovare automaticamente il valore migliore per i pesi $\{W, \boldsymbol{v}\}$

Nel 1986 Hinton e altri autori (in maniera indipendente) proposero di usare l’algoritmo della Backpropagation, il quale permette di implementare il Gradient Descent per addestrare una ANN multi-layer calcolando (in maniera efficiente) l’aggiornamento dei vari pesi

Il concetto base della Backpropagation è considerare $g(\boldsymbol{v}^T f(W\boldsymbol{x}))$ come una funzione *composta* $h(\boldsymbol{x}) = g(\boldsymbol{v}^T f(W\boldsymbol{x}))$ e applicare la regola di derivazione per le funzioni composte (“chain rule” in inglese, da non confondere con l’ omonima regola della probabilità congiunta):

$D[g(f(x))] = g'(f(x))\, f'(x)$

L’algoritmo di Backpropagation consiste nel calcolare queste derivate (parziali) composte per tutti i pesi dell’MLP e farlo in manirea efficiente, evitando duplicazioni di calcoli

Nello specifico, l’efficienza è ottenuta calcolando prima il gradiente relativo ai pesi hidden-to-output ($\boldsymbol{v}$) e poi, “andando indietro”, quello dei pesi input-to-hidden ($W$)

Vediamolo un pò più in dettaglio usando un task di regressione

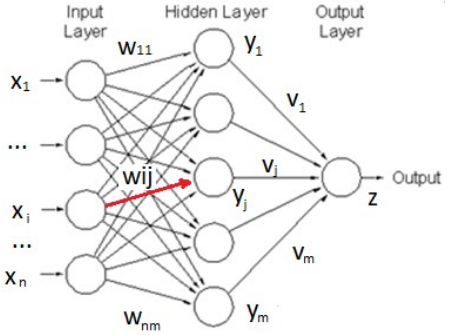
# Training: regressione

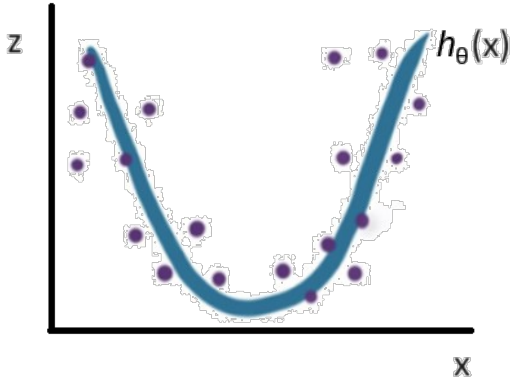
Indichiamo con θ = {*W,**v***} l'insieme complessivo dei parametri che dobbiamo stimare

Per cui possiamo scrivere $h_\theta(\boldsymbol{x})$ per indicare che la funzione complessiva $h(\boldsymbol{x})$ calcolata dall'MLP è un modello parametrico che dipende da θ

Nel caso specifico della regressione avremo:
$h_\theta(\boldsymbol{x}) = g(\boldsymbol{v}^T f(W\boldsymbol{x})) = \boldsymbol{v}^T \tanh(W\boldsymbol{x})$

Supponiamo inoltre di avere un insieme supervisionato di training $T = \{(\boldsymbol{x}^{(1)}, z^{(1)}), \ldots (\boldsymbol{x}^{(i)}, z^{(i)}), \ldots, (\boldsymbol{x}^{(N)}, z^{(N)})\}$, dove $z^{(i)} \in R$ è l'i-esimo valore di ground truth per la variabile target associata al feature vector $\boldsymbol{x}^{(i)} \in R^n$

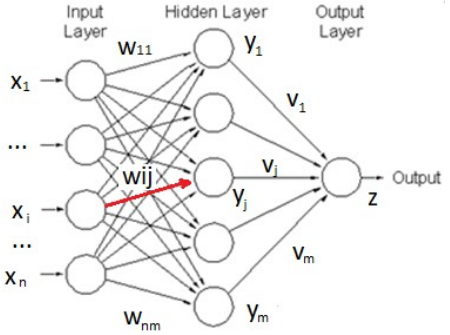


# Training: regressione

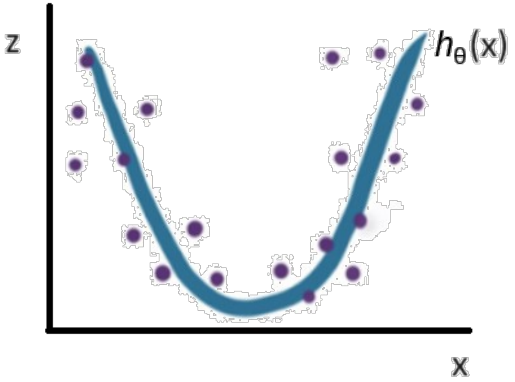
Per prima cosa ci serve una loss function. Useremo la Least Squares («minimi quadrati») già incontrata nella Linear Regression:

$$J(\theta) = \frac{1}{2}\sum_{i=1}^{N}(h_\theta(\boldsymbol{x}^{(i)}) - z^{(i)})^2$$




Il fatto che useremo la stessa loss function già utilizzata per la Linear Regression non deve meravigliare: una loss function è una funzione d'errore relativa ad un determinato task (regressione, in questo caso) che può essere applicata a qualsiasi modello parametrico usato per risolvere quel task

Nel caso della Linear regression il modello usato ($h_\theta(\boldsymbol{x})$) è lineare, per cui la funzione di predizione può esprimere solo relazioni lineari tra le variabili indipendenti e quella dipendente




Nel caso dell'MLP, $h_\theta(\boldsymbol{x})$ è non lineare rispetto ad $\boldsymbol{x}$, e questo fa la differenza

# Training: regressione

AImage Lab
UNIMORE UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

Per addestrare l'MLP si usa il Gradient Descent

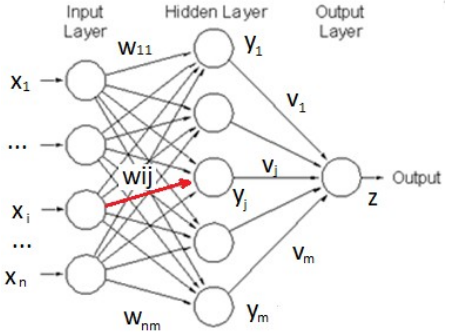
Per cui si procede al solito modo: stabilisco anzitutto se voglio usare la versione "batch" o quella "stochastic"

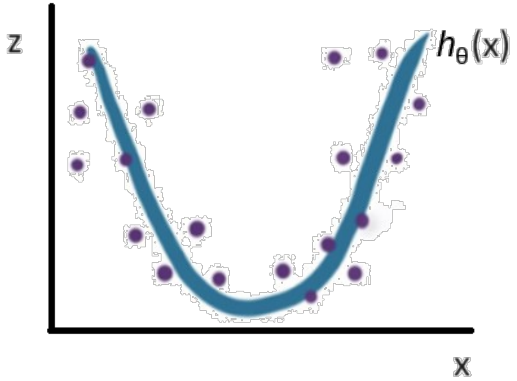
Dopodichè inizializzo i parametri in maniera random, ovvero inizializzo sia *W* che ***v***

Procederò quindi per iterazioni successive aggiornando tutti i pesi

La differenza con il Gradient Descent standard è che ora dobbiamo aggiornare sia *W* che ***v***

Avrò quindi bisogno di calcolare il gradiente rispetto ad ogni parametro in θ = {*W*,***v***}



# Training: regressione

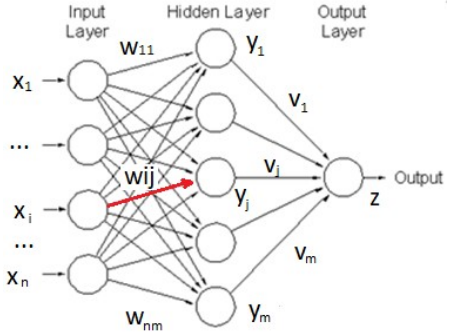
Formalmente, devo calcolare:

$$\frac{\partial}{\partial v_j} J(W, \boldsymbol{v}) \qquad (1)$$

per ogni $v_j \in \boldsymbol{v}$, e:

$$\frac{\partial}{\partial w_{ij}} J(W, \boldsymbol{v}) \qquad (2)$$

per ogni $w_{ij} \in W$



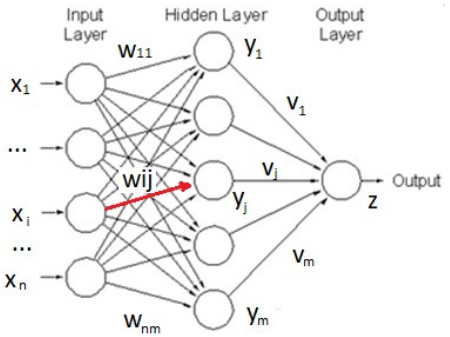

La parte difficile consiste nel fatto che i gradienti calcolati in (2) dipendono (tramite chain rule) dai gradienti calcolati in (1)

Senza entrare nei dettagli tecnici, l'algorimo di Backpropagation permette di effettuare efficientemente questo calcolo iniziando da (1) e poi "propagando all'indietro" (da cui il nome) il valore dei gradienti calcolati in (1) che servono per calcolare (2)

# Backpropagation: regressione

Supponiamo di avere un solo sample ***x***. I passi specifici che la Backpropagation segue per aggiornare i pesi sono i seguenti

1. *Forward pass.* Calcolo le attivazioni per tutti i layer: ***y*** = *f*(*W****x***) e $z = \boldsymbol{v}^T \boldsymbol{y}$
2. *Backward pass.*
   - Calcolo i gradienti in eq. (1) utilizzando ***y*** e la loss function calcolata rispetto al valore *z* (non mi serve altro)
   - Aggiorno i pesi contenuti in ***v***
   - Calcolo il gradiente in eq. (2) utilizzando ***x***, la regola di derivazione per le funzioni composte e il valore dei gradienti dei pesi in ***v*** calcolati precedentemente
   - Aggiorno i pesi contenuti in *W*




$$\frac{\partial}{\partial v_j} J(W, \boldsymbol{v}) \qquad (1)$$

$$\frac{\partial}{\partial w_{ij}} J(W, \boldsymbol{v}) \qquad (2)$$

# Training: classificazione binaria

Nel caso della classificazione binaria lo schema è analogo, ma devo usare una loss function diversa

Userò la Binary Cross Entropy già vista nel caso della Logistic Regression

Come prima, $h_\theta(\boldsymbol{x}) = g(\boldsymbol{v}^T f(W\boldsymbol{x}))$ indica la funzione complessiva calcolata dall'MLP e $\theta = \{W, \boldsymbol{v}\}$

Ricordiamoci che ora $g()$ è la logistic function

$T = \{(\boldsymbol{x}^{(1)}, t^{(1)}), \ldots (\boldsymbol{x}^{(i)}, t^{(i)}), \ldots, (\boldsymbol{x}^{(N)}, t^{(N)})\}$, dove $t^{(i)} \in \{0,1\}$ è l'i-esimo valore di ground truth per il feature vector $\boldsymbol{x}^{(i)} \in R^n$

La loss può essere quindi espressa come segue:

$$J(\theta) = -\sum_{i=1}^{N} t^{(i)} \log(h_\theta(\boldsymbol{x}^{(i)})) + (1 - t^{(i)}) \log(1 - h_\theta(\boldsymbol{x}^{(i)}))$$

Anche in questo caso ciò che fa la differenza rispetto alla Logistic Regression è la parte parametrica della funzione ipotesi (cioè il modello che sto usando, basato su un MLP e non su una funzione lineare)

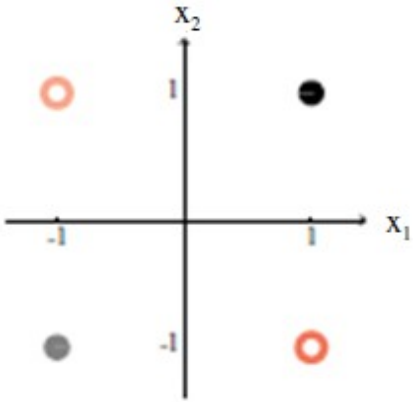
Il resto è identico al caso della regressione

# Esempio di classificazione binaria: il problema dello XOR

Vediamo come il problema dello XOR possa essere risolto mostrando i risultati di un *esperimento reale*, in cui il Gradient Descent è stato usato per trovare i pesi *in maniera automatica*

$T = \{(\boldsymbol{x}^{(1)}, t^{(1)}), \ldots (\boldsymbol{x}^{(i)}, t^{(i)}), \ldots, (\boldsymbol{x}^{(N)}, t^{(N)})\}$, dove:

- Se $\boldsymbol{x}^{(i)} = [x_1, x_2] = [1, 1]$, allora $t^{(i)} = 0$
- Se $\boldsymbol{x}^{(i)} = [x_1, x_2] = [-1, 1]$, allora $t^{(i)} = 1$
- Se $\boldsymbol{x}^{(i)} = [x_1, x_2] = [-1, -1]$, allora $t^{(i)} = 0$
- Se $\boldsymbol{x}^{(i)} = [x_1, x_2] = [1, -1]$, allora $t^{(i)} = 1$

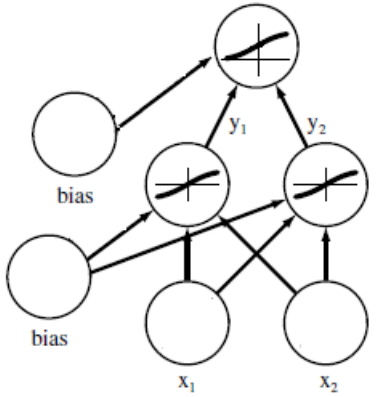


# Esempio di classificazione binaria: il problema dello XOR

A fianco c'è la struttura schematica dell'MLP con le activation function corrispondenti

N.B.: siccome ora la nonlinearità dell'hidden layer è *tanh()*, con codominio continuo, anche il feature space definito da $y_1$ e $y_2$ sarà continuo

I pesi vengono inizializzati in maniera random



# Esempio di classificazione binaria: il problema dello XOR

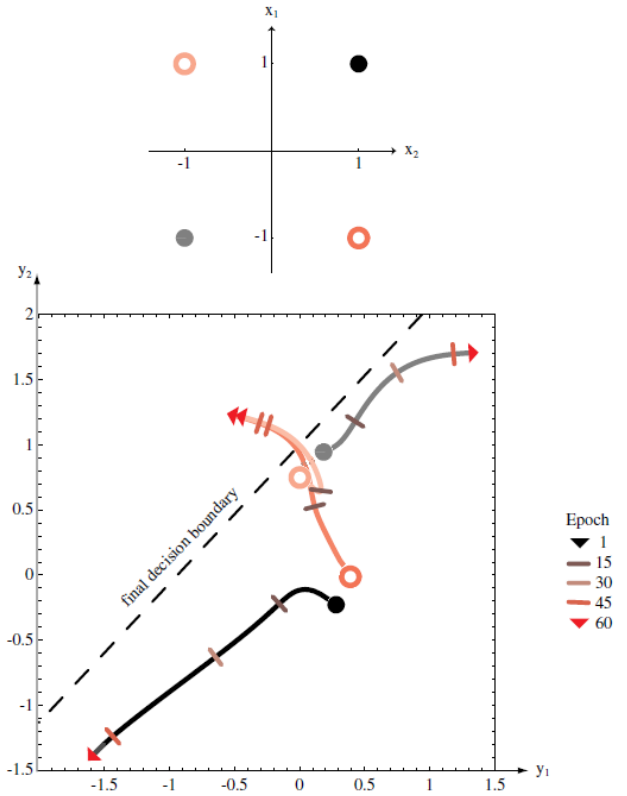
La figura in basso mostra l'evoluzione dei valori delle activation dell'hidden layer, ovvero il valore di $[y_1, y_2] = [f(\boldsymbol{w_1}^T \boldsymbol{x}), f(\boldsymbol{w_2}^T \boldsymbol{x})]$

Più precisamente, "epoca" nel gergo delle ANN vuol dire *N* iterazioni di Gradient Descent (con Backpropagation) su un dataset *T* di *N* elementi

Per cui la figura mostra, epoca dopo epoca, come cambia, per ogni $\boldsymbol{x} = [x_1, x_2]$, il corrispondente valore di $[y_1, y_2]$ al mutare di $\boldsymbol{w_1}$ e $\boldsymbol{w_2}$

Colori diversi delle "traiettorie" corrispondono alla rappresentazione di sample $\boldsymbol{x}$ diversi nel feature space dell'hidden layer

La cosa importante da notare è che il feature space dell'hidden layer, ovvero le rappresentazioni basate su $[f(\boldsymbol{w_1}^T \boldsymbol{x}), f(\boldsymbol{w_2}^T \boldsymbol{x})]$, **cambia nel tempo**



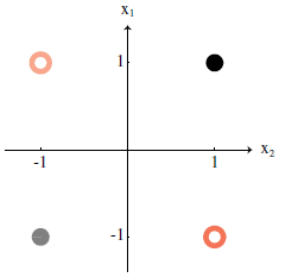
# Esempio di classificazione binaria: il problema dello XOR

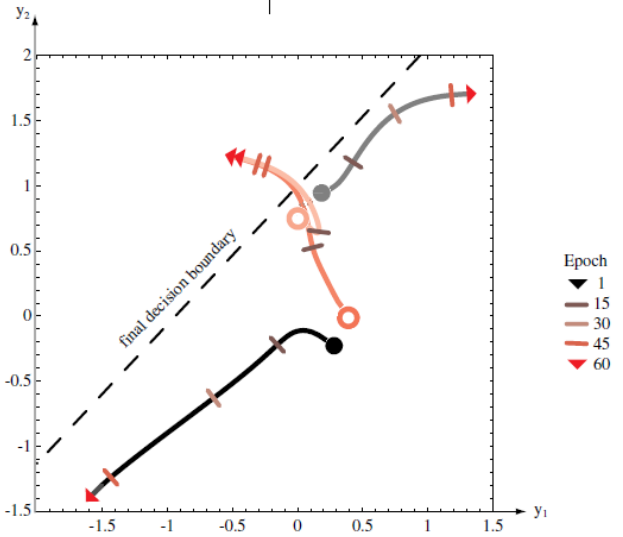
All'inizio le due classi non sono separabili

Alla fine del training lo diventano, come indica il decision boundary corrispondente alla fine dell'ultima epoca (la linea tratteggiata)

Quest'esempio mostra un principio fondamentale delle ANN (e del Deep Learning, come vedremo), ovvero che un feature space ($y_1$, $y_2$) che risulta adeguato per un dato task può essere ottenuto ottimizzando i parametri

Ovvero, può essere appresa automaticamente la trasformazione $\boldsymbol{y} = \phi(\boldsymbol{x})$ che meglio permette di risolvere il task



# Il ruolo delle nonlinearità

AImage Lab
UNIMORE UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

Resta da chiarire qual è il ruolo delle nonlinearità

Abbiamo visto che, cambiando l'activation function nell'output layer (e la loss function) possiamo usare lo stesso MLP sia per un task di regressione che di classificazione binaria

Ciò rende l'MLP una struttura particolarmente flessibile

Ma qual è il ruolo delle activation function nell'hidden layer?

# Il ruolo delle nonlinearità

Se non ci fossero, torneremmo al potere rappresentativo (limitato) del Percettrone. Dimostrazione:

Sappiamo che un MLP calcola $h(\boldsymbol{x}) = g(\boldsymbol{v}^T f(W\boldsymbol{x}))$

Se $f()$ non ci fosse (i.e., $f()$ = funzione identità), allora avremmo $h(\boldsymbol{x}) = g(\boldsymbol{v}^T W\boldsymbol{x})$

Ricordiamoci che $\boldsymbol{v}^T$ è un vettore $1 \times m$, mentre $W$ è una matrice $m \times n$:

$\boldsymbol{v} = [v_1, \ldots, v_m]$, $W = \begin{bmatrix} \boldsymbol{w}_1^T \\ \ldots \\ \boldsymbol{w}_m^T \end{bmatrix}$

Per cui, se $\boldsymbol{r} = \boldsymbol{v}^T W$ ($\boldsymbol{r}$ è il vettore di parametri $1 \times n$ ottenuto moltiplicando $\boldsymbol{v}^T$ e $W$), allora posso riscrivere $h()$ come: $h(\boldsymbol{x}) = g(\boldsymbol{r}^T \boldsymbol{x})$

In altri termini, se non avessi le nonlinearità nel layer hidden, la parte parametrica dell'MLP sarebbe una funzione lineare dell'input (i.e., una sommatoria pesata delle feature ottenuta con i pesi in $\boldsymbol{r}$), esattamente come nel Percettrone, nella Linear Regression, nella Logistic Regression e nelle SVM (senza kernel) e perderemmo il potere espressivo garantito dal teorema di approssimazione universale

# Classificazione multi-classe

AImage Lab
UNIMORE UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

Estendiamo ora l'MLP per trattare problemi con *k* classi

Anzitutto, in maniera analoga a quanto abbiamo visto nel caso della Softmax Regression, esprimiamo la variabile target con un "hot vector":

Se $(\boldsymbol{x}^{(i)}, j) \in T$ e $j \in \{1, \ldots, k\}$ è la label di ground truth, allora $\boldsymbol{t}^{(i)} = [0, 0, \ldots, 1, 0, \ldots, 0]^T$ (con un 1 in j-esima posizione)

Per cui possiamo esprimere $T$ in questo modo:
$T = \{(\boldsymbol{x}^{(1)}, \boldsymbol{t}^{(1)}), \ldots (\boldsymbol{x}^{(i)}, \boldsymbol{t}^{(i)}), \ldots, (\boldsymbol{x}^{(N)}, \boldsymbol{t}^{(N)})\}$

# Classificazione multi-classe

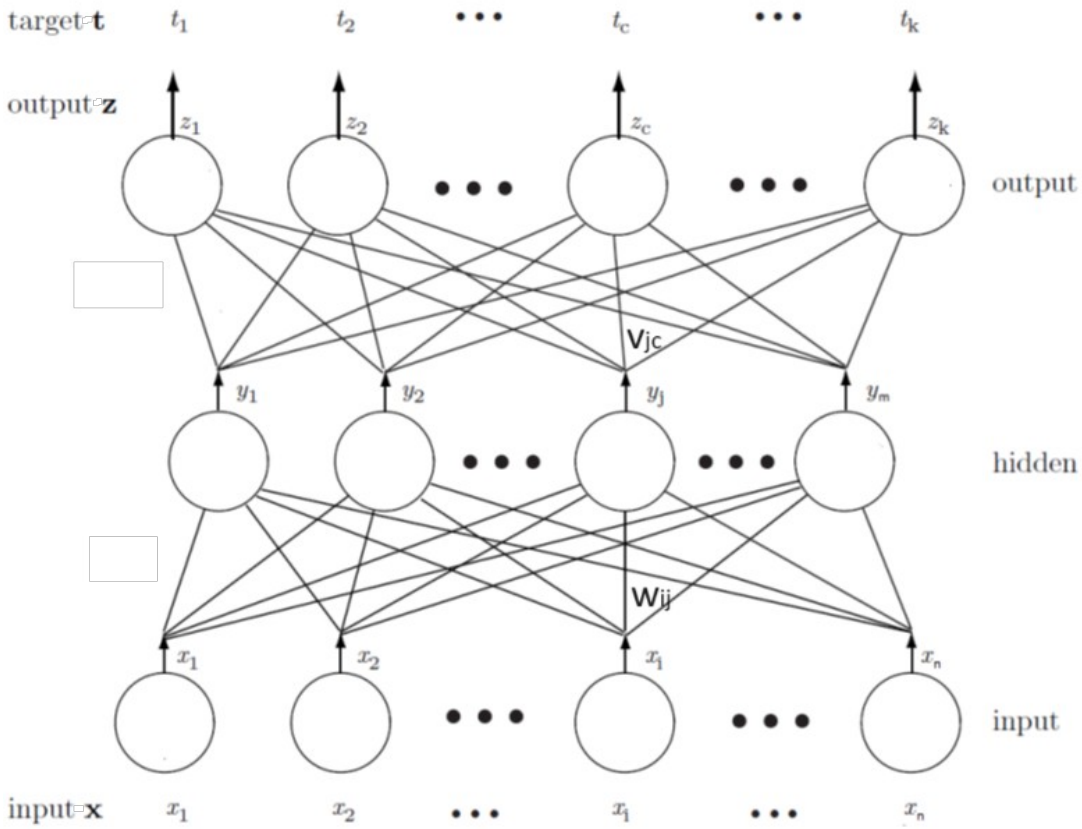
Nuovamente in maniera analoga a quanto abbiamo visto nel caso della Softmax Regression, abbiamo bisogno di *k* output, perchè, per un dato ***x***, dobbiamo stimare, simultaneamente:

- $P(t = 1|\boldsymbol{x})$
- ***...***
- $P(t = k|\boldsymbol{x})$

e poi scegliere la classe corrispondente alla posterior con valore massimo

Quindi un solo neurone di output non basta

# MLP: generalizzazione con *k* neuroni di output



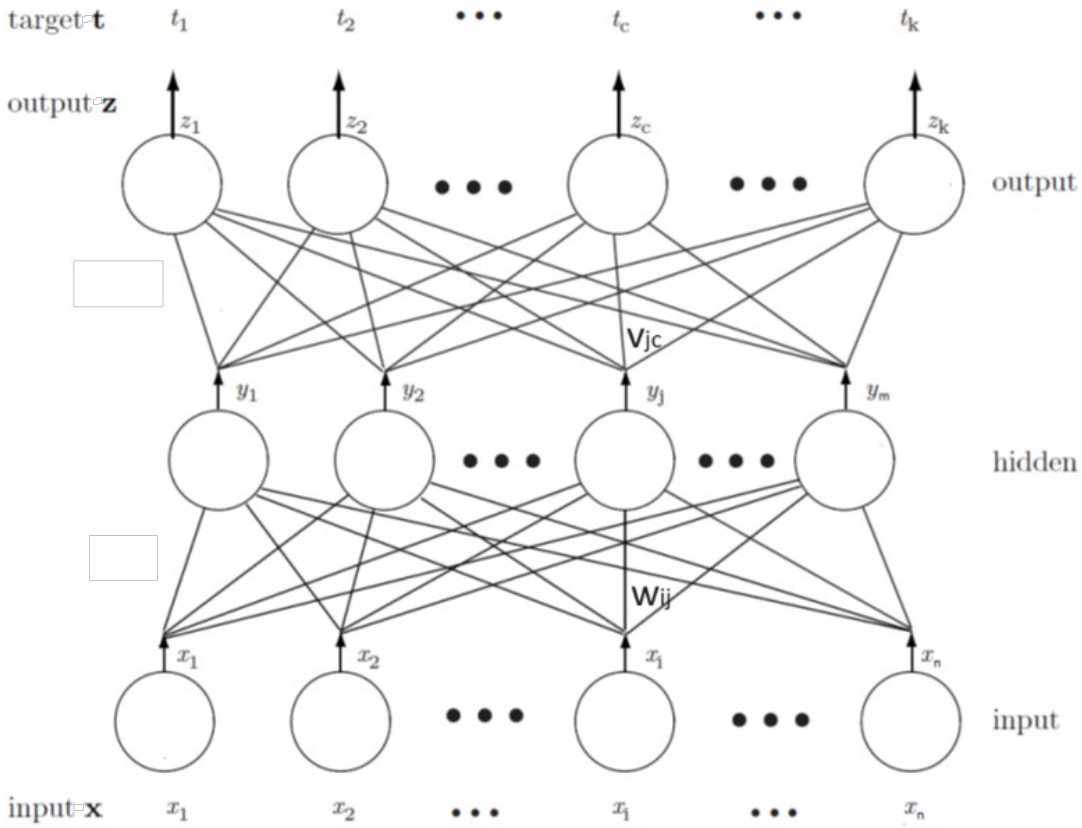

L'MLP a fianco ha *k* neuroni nell'output layer, ognuno dei quali è connesso con tutti i neuroni dell'hidden layer

La funzione che calcola è:
**z** = *h*(**x**), *d*ove ora ***z*** è un vettore!

$\boldsymbol{z} = [z_1, \ldots, z_c, \ldots, z_k]$

- $z_1 = g(\boldsymbol{v}_1^T f(W\boldsymbol{x})) = P(t = 1|\boldsymbol{x})$
- …
- $z_c = g(\boldsymbol{v}_c^T f(W\boldsymbol{x})) = P(t = c|\boldsymbol{x})$
- …
- $z_k = g(\boldsymbol{v}_k^T f(W\boldsymbol{x})) = P(t = k|\boldsymbol{x})$

# MLP: generalizzazione con *k* neuroni di output




- $z_1 = g(\boldsymbol{v}_1^T f(W\boldsymbol{x})) = P(t = 1|\boldsymbol{x})$
- …
- $z_c = g(\boldsymbol{v}_c^T f(W\boldsymbol{x})) = P(t = c|\boldsymbol{x})$
- …
- $z_k = g(\boldsymbol{v}_k^T f(W\boldsymbol{x})) = P(t = k|\boldsymbol{x})$

Per cui ora i parametri dell'MLP sono $\theta = \{W, V\}$, dove:

$$V = \begin{bmatrix} \boldsymbol{v}_1^T \\ \dots \\ \boldsymbol{v}_k^T \end{bmatrix}$$

# MLP: generalizzazione con *k* neuroni di output



Riguardo alla funzione di attivazione dell'ultimo layer, *g()*, usiamo la funzione Softmax:

$$z_c = g(\boldsymbol{v}_c^T f(W\boldsymbol{x})) = g(\boldsymbol{v}_c^T \boldsymbol{y}) = \frac{e^{\boldsymbol{v}_c^T \boldsymbol{y}}}{\sum_{q=1}^{k} e^{\boldsymbol{v}_q^T \boldsymbol{y}}}$$

Per addestrare questo MLP, possiamo usare la Cross Entropy (v. Softmax Regression…) congiuntamente col Gradient Descent implementato mediante Backpropagation

# Flessibilità dell'MLP

AImage Lab
UNIMORE UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

Riassumendo, un MLP può essere usato per:

- task di regressione non lineari se l'output layer ha un solo neurone e non esiste *g()*
- task di classificazione binaria non lineari se l'output layer ha un solo neurone e *g()* è la logistic function
- task di classificazione con *k* classi non lineari se l'output layer ha *k* neuroni e *g()* è la softmax function

Un MLP può anche avere più di 3 layer, ma questo lo vedremo nella prossima lezione, quando parleremo di Deep Learning

# Iper-parametri (topologia)



Quando si progetta il modello di una rete neurale bisogna tener conto che:

- il numero di input e output unit è dato dal problema, così come le funzioni di attivazione dell'output layer
- il numero di hidden layer, le funzioni di attivazione dell'hidden layer e il numero di hidden unit sono iper-parametri (da scegliere usando il validation set)

→ Più sono gli hidden unit, più sono i parametri da ottimizzare
→ Più sono i parametri, maggiore è l'espressività del modello, ma anche maggiore è il rischio di overfitting

Per limitare il rischio di overfitting nel caso delle reti neurali, due possibili strategie di regolarizzazione sono la **weight decay** e l'**early stopping**

# Weight decay

La weight decay consiste in una penalità $L_2$ sui pesi da aggiungere alla loss function (qualsiasi loss function), usando esattamente lo stesso principio della Ridge Regression:

$$\hat{J}(\theta) = J(\theta) + \frac{\lambda}{2}\left(\sum_{j=1}^{m} \boldsymbol{w}_j^T \boldsymbol{w}_j + \sum_{c=1}^{k} \boldsymbol{v}_c^T \boldsymbol{v}_c\right)$$

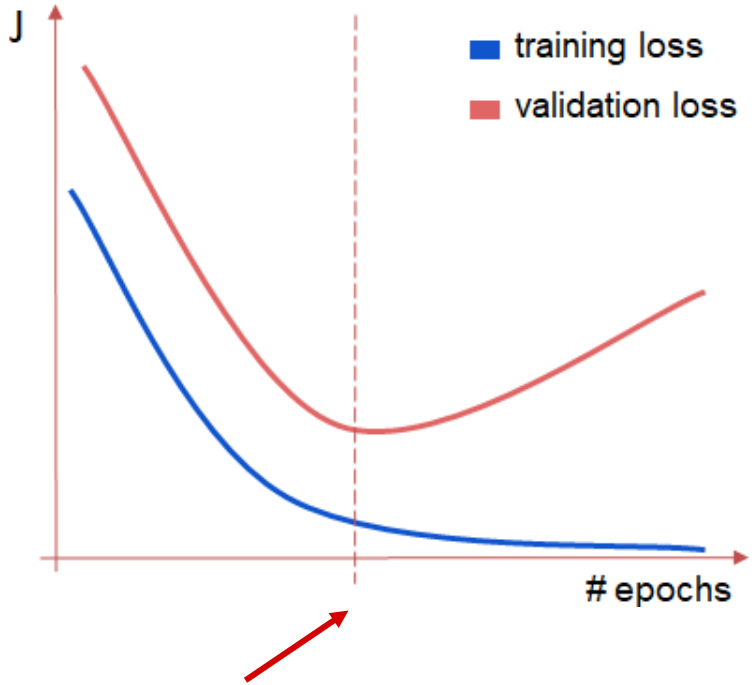
L'unica differenza è che devo regolarizzare $m + k$ vettori di pesi anziché un solo vettore

# Early stopping

Tipicamente, durante il training, il valore della loss calcolata sul training set decresce in maniera monotona

Se però calcolo la stessa loss function su un validation set esterno al training, quello che di solito accade è che il valore della loss sul validation diminuisce all'inizio e poi inizia ad aumentare *quando il modello inizia ad andare in overfitting*

L'idea dietro la regolarizzazione con early stopping è di fermare il training quando la loss sul validation set è al suo minimo



# Referimenti


- **A tu per tu col Machine Learning. L'incredibile viaggio di un developer nel favoloso mondo della Data Science**, Alessandro Cucci, The Dot Company, 2017 [cap.7]
- **Pattern Classification, second edition,** R. O. Duda, P. E. Hart, D. G. Stork, Wiley-Interscience, 2000
- **Pattern recognition and machine learning**, Christopher Michael Bishop, Springer Science+ Business Media, 2006 [cap. 5]