



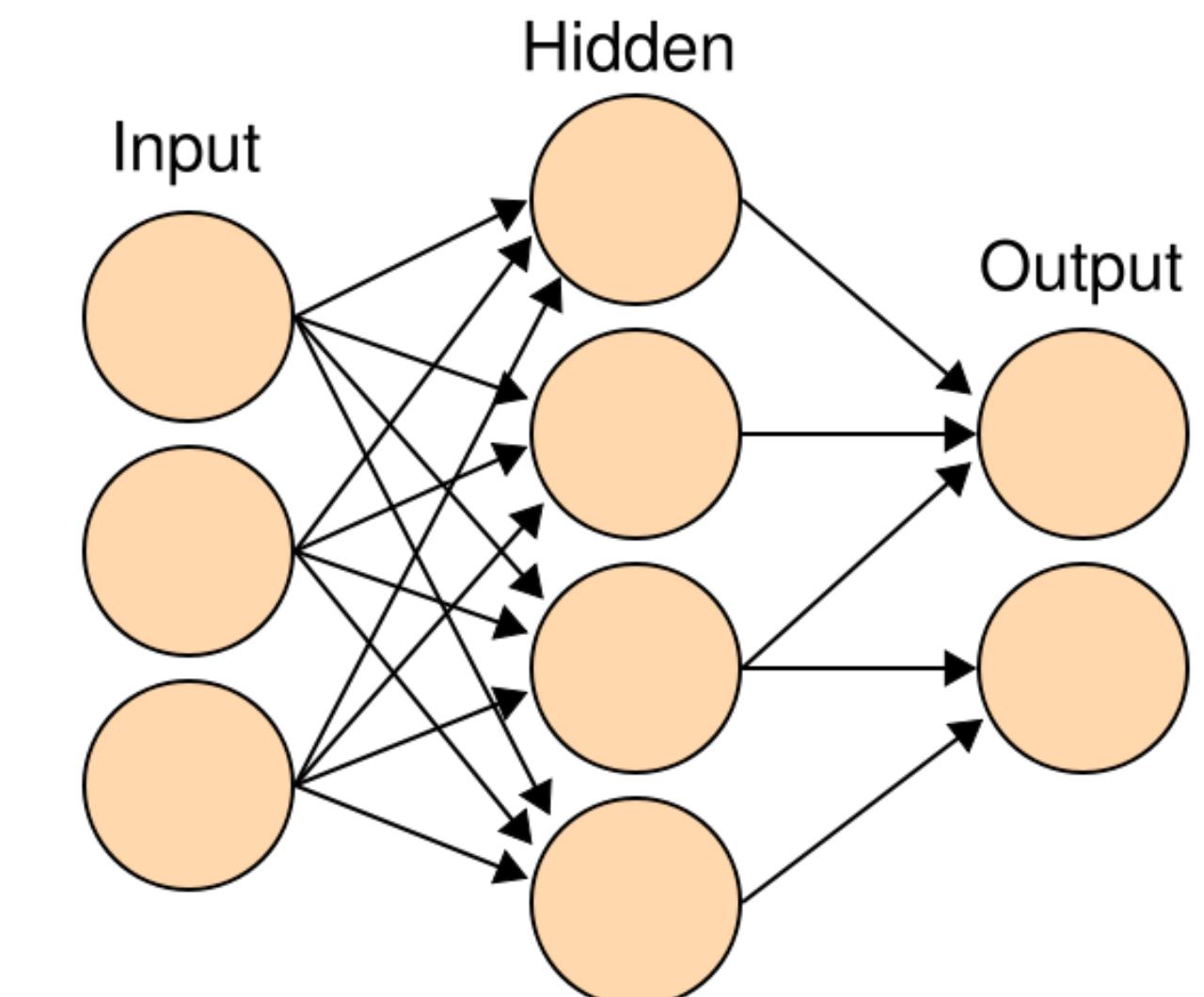
SAPIENZA
UNIVERSITÀ DI ROMA

METODI DI INTELLIGENZA ARTIFICIALE E MACHINE LEARNING IN FISICA

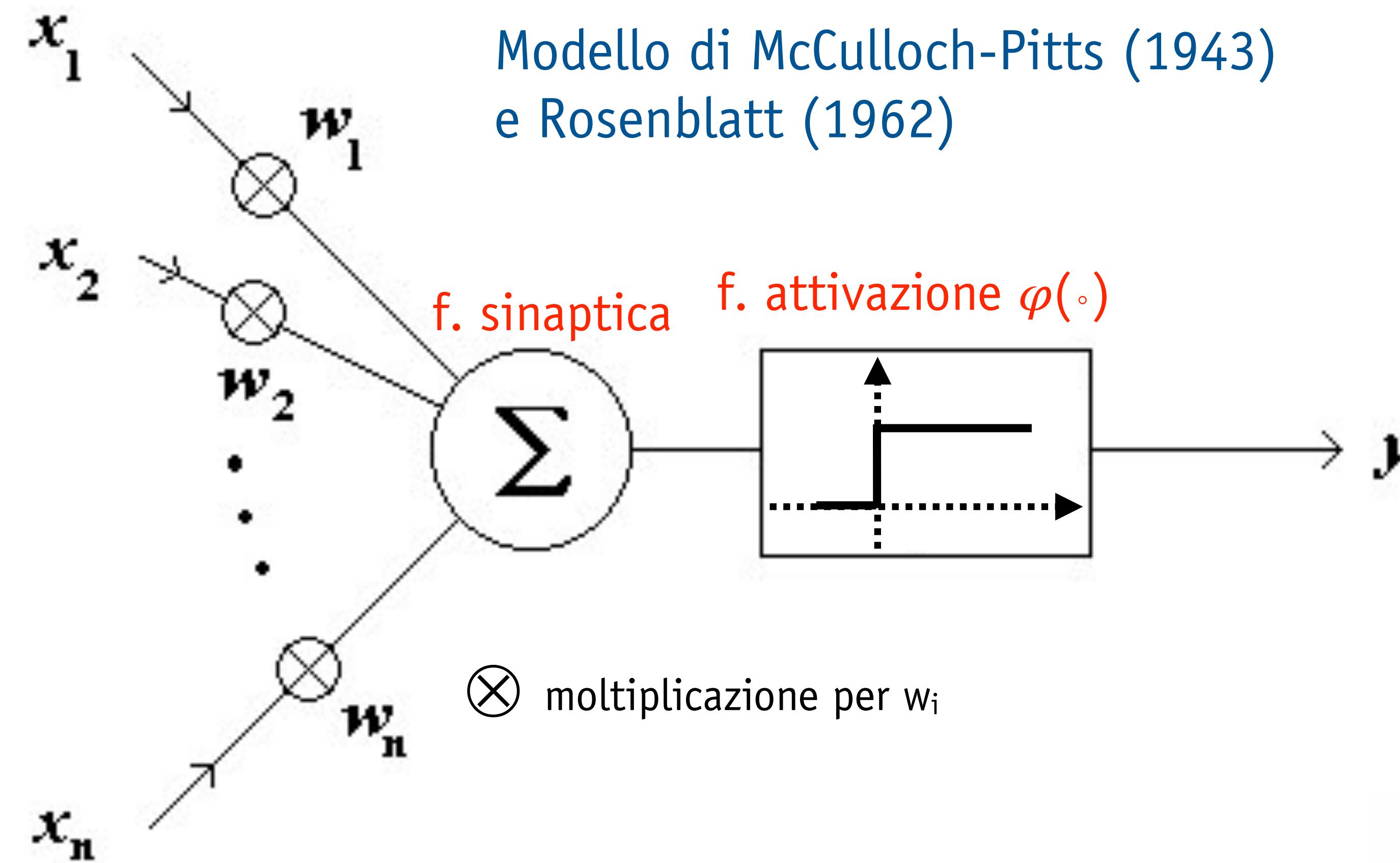
S. Giagu - AA 2019/2020
Lezione 13: 8.4.2020

ARTIFICIAL NEURAL NETWORKS

- costituiscono l'esempio più noto di modelli non-lineari e sono alla base delle moderne tecniche di deep learning
- un ANN è un **modello matematico in grado di approssimare con grande precisione qualsiasi forma funzionale**
 - è possibile introdurre il concetto di NN sia in una sorta di **analogia con le reti neurali biologiche**, sia in modo più formale come **composizione di funzioni connesse in catene descritte da grafi** (per esempio i Feed-Forward NN possono essere rappresentati come grafi diretti aciclici)
 - consiste in un **gruppo interconnesso di neuroni artificiali**
 - analizza informazioni in input in accordo ad un **approccio computazionale di tipo connessionista** → azioni eseguite in modo collettivo ed in parallelo da unità semplici (i neuroni)
 - si comporta come un **sistema adattivo**: modifica la sua struttura basandosi su un insieme di informazioni che fluiscono attraverso la rete durante la fase di apprendimento
 - risposta non lineare ottenuta attraverso l'uso di funzioni non lineari nella risposta di ogni neurone
 - apprendimento gerarchico delle rappresentazioni ottenuto tramite l'uso di architetture complesse



MODELLO DI NEURONE ARTIFICIALE



con una TLU è possibile risolvere problemi con classi linearmente separabili:

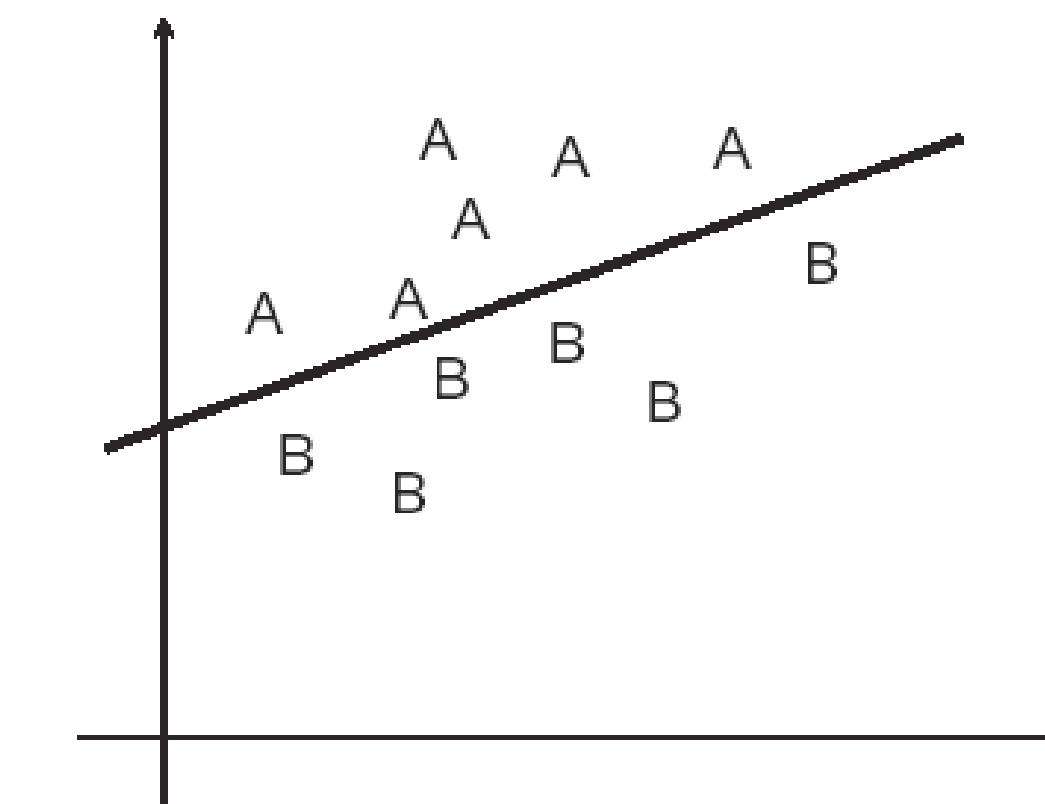
Caratteristiche:

- unità di processamento che prende in input n segnali x_i e produce in uscita un output y ottenuto come composizione di una **funzione sinaptica**:

$$a = \sum_{i=1}^n w_i x_i = \mathbf{w}^t \mathbf{x}$$

- e una **funzione di attivazione** (Heaviside):

$$y = \varphi(a) = H(a - w_0) = \begin{cases} 1 & \text{if } a \geq w_0 \\ 0 & \text{if } a < w_0 \end{cases}$$

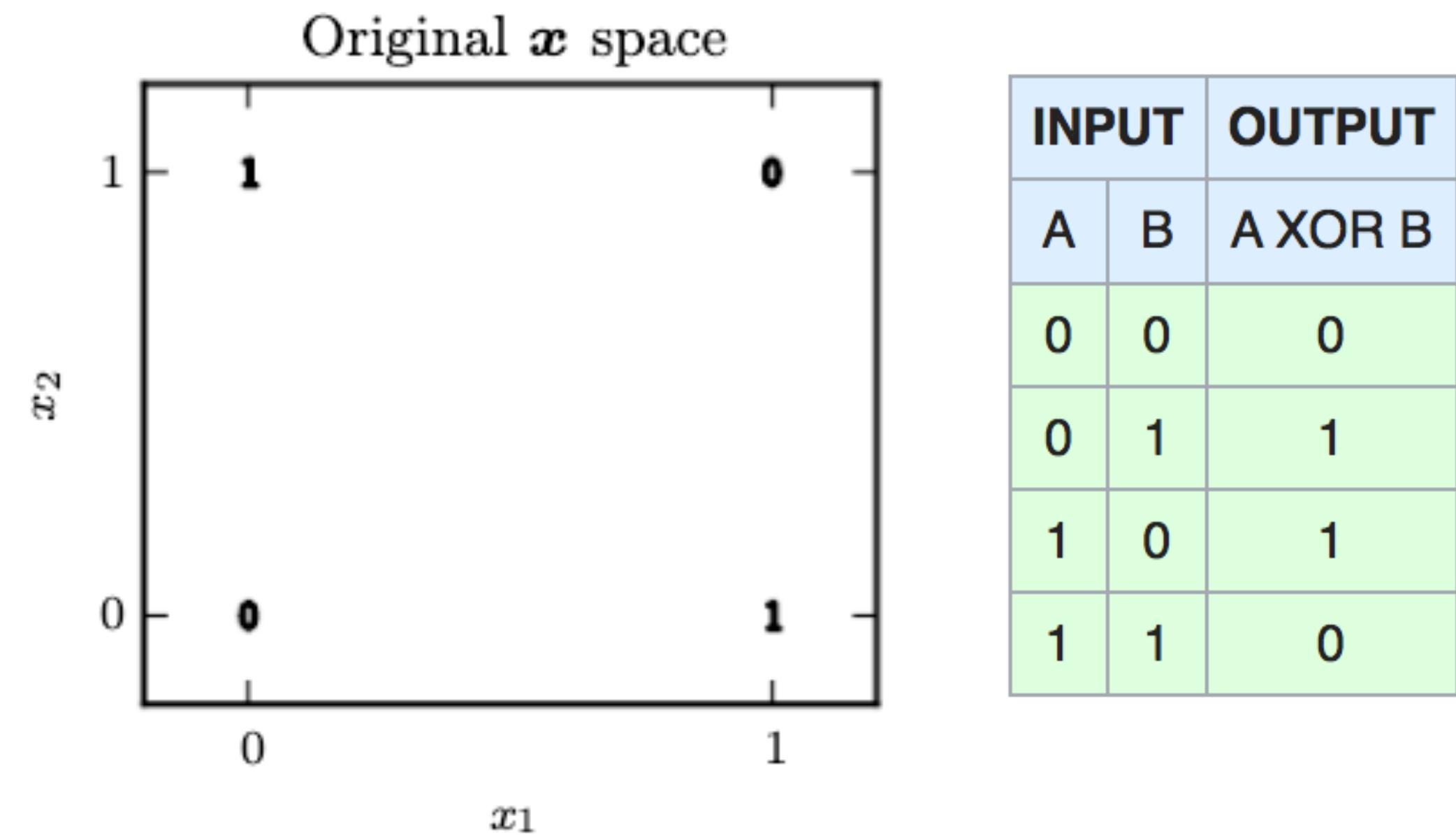


L'ESEMPIO PIÙ SEMPLICE DI PROBLEMA NON RISOLUBILE CON UN SOLO NEURONE: XOR GATE

$$(A \cdot \bar{B}) + (\bar{A} \cdot B)$$

$$y = f(\mathbf{x})$$

funzione f che ritorna 1 quando esattamente uno tra x_1 e x_2 è uguale a 1



- vogliamo approssimare la funzione f con una NN: $y = f^*(\mathbf{x}; \mathbf{w})$
- training set: $T = \{[0,0], [0,1], [1,0], [1,1]\}$
- loss function MSE: $L(\mathbf{w}) = 1/4 \sum_T (y - f^*(\mathbf{x}; \mathbf{w}))^2$

singola TLU con attivazione lineare

$$f^*(\mathbf{x}; \mathbf{w}, b) = \mathbf{x}^t \mathbf{w} + b$$

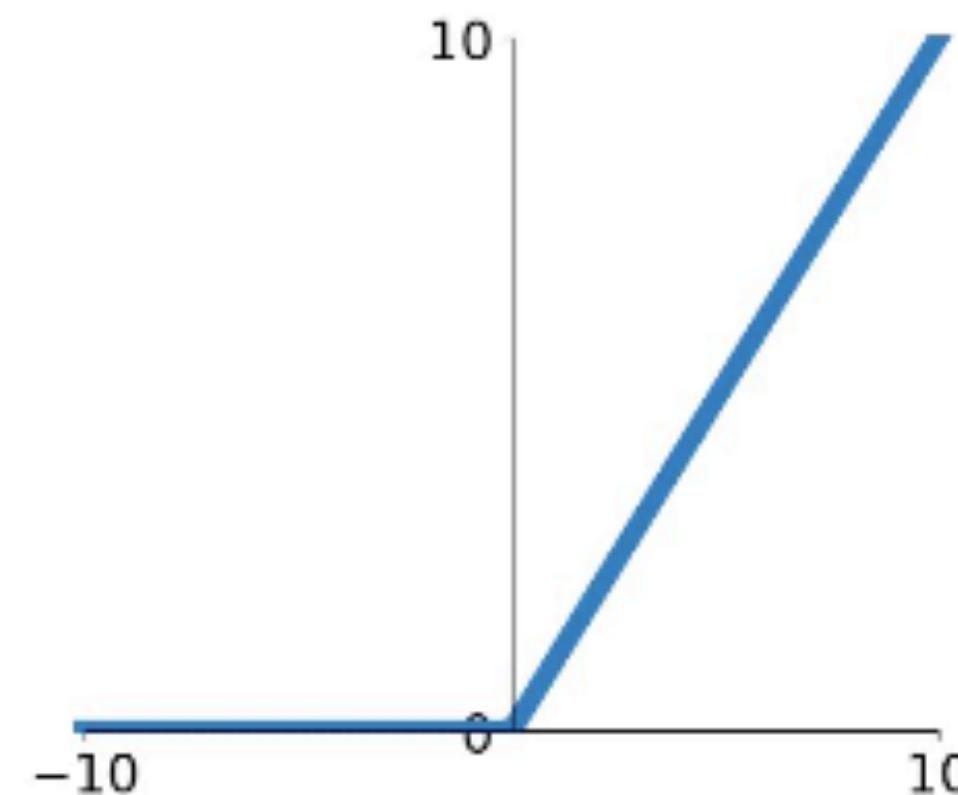
calcolando le derivate rispetto a \mathbf{w} e b e
eguagliando a zero si ottiene $\mathbf{w}=0$ e $b=1/2$
ovviamente una singola retta nel piano non può
risolvere il problema \Rightarrow serve una funzione non
lineare ...



definiamo la funzione 2-dim (hidden layer) h : $h = \varphi(\mathbf{x}^t \mathbf{W} + \mathbf{c})$

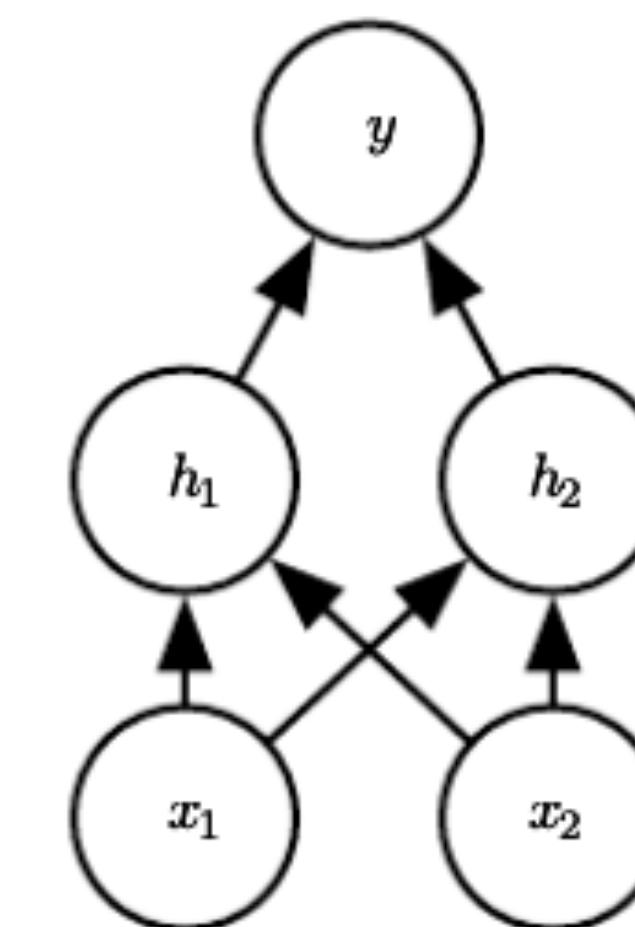
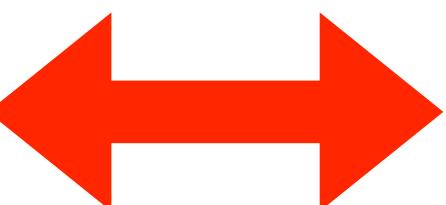
NOTA: ora W è una matrice di pesi 2×2 e il bias \mathbf{c} è un vettore bi-dimensionale visto che ci serve per trasformare \mathbf{x} in una nuova rappresentazione \mathbf{h} con stessa dimensione (in questo caso specifico) di \mathbf{x}

come attivazione scegliamo: Rectified Linear Unit (ReLU): $\varphi(x) = \max\{0, x\}$



la nostra equazione completa del NN diviene:

$$y = f^*(\mathbf{x}; \mathbf{w}, \mathbf{c}, b) = \mathbf{w}^t \max\{0, \mathbf{W}^t \mathbf{x} + \mathbf{c}\} + b$$



partiamo con:

$$\mathbf{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix},$$

$$\mathbf{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix},$$

$$\mathbf{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix},$$

calcoliamo \mathbf{XW} :

$$\mathbf{XW} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix}$$

infine applichiamo ReLU per ottenere la rappresentazione di \mathbf{X} nello spazio trasformato \mathbf{h} :

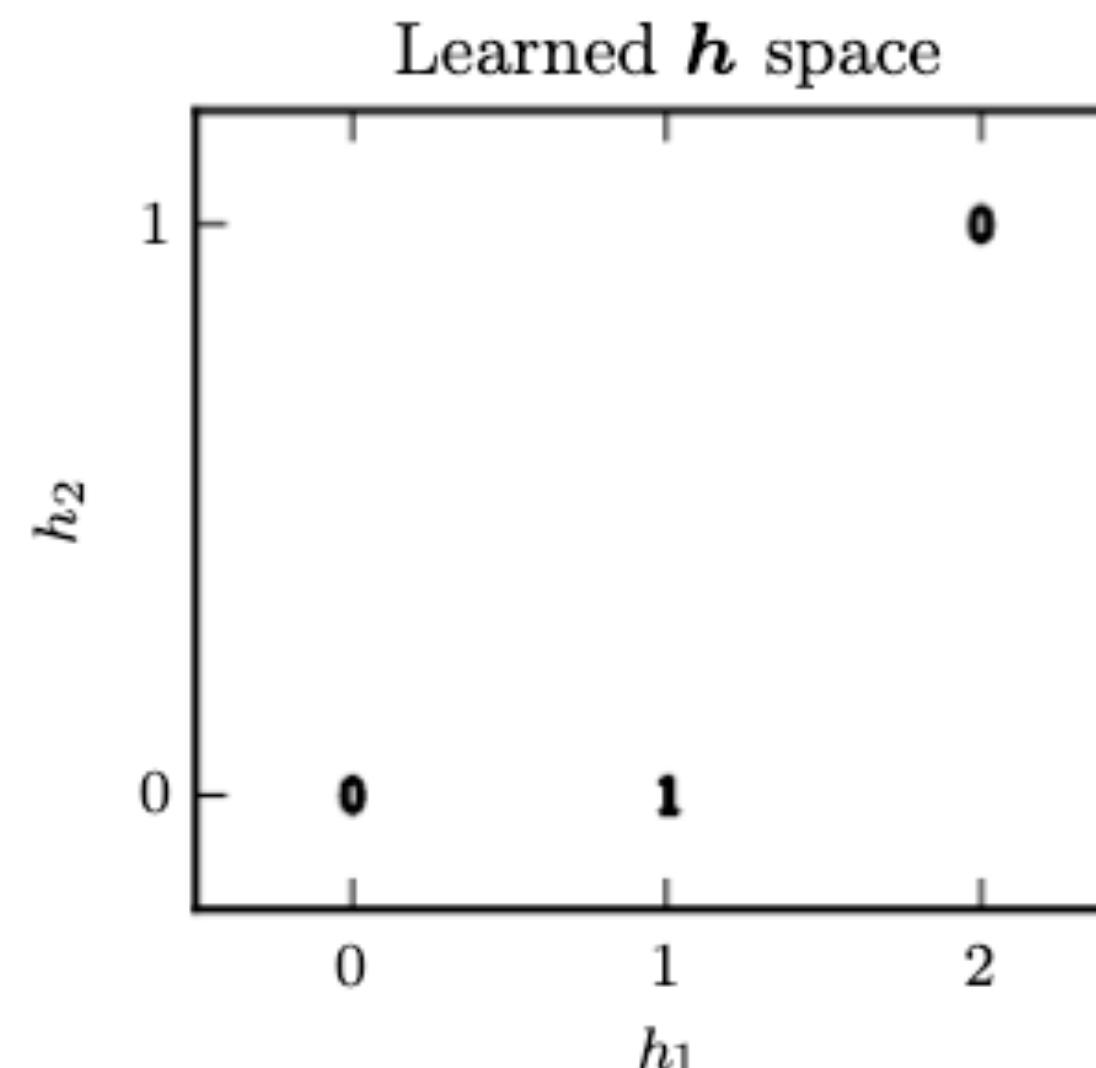
$$\begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

con set di eventi di training:

$$\mathbf{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$$

e aggiungiamo \mathbf{c} :

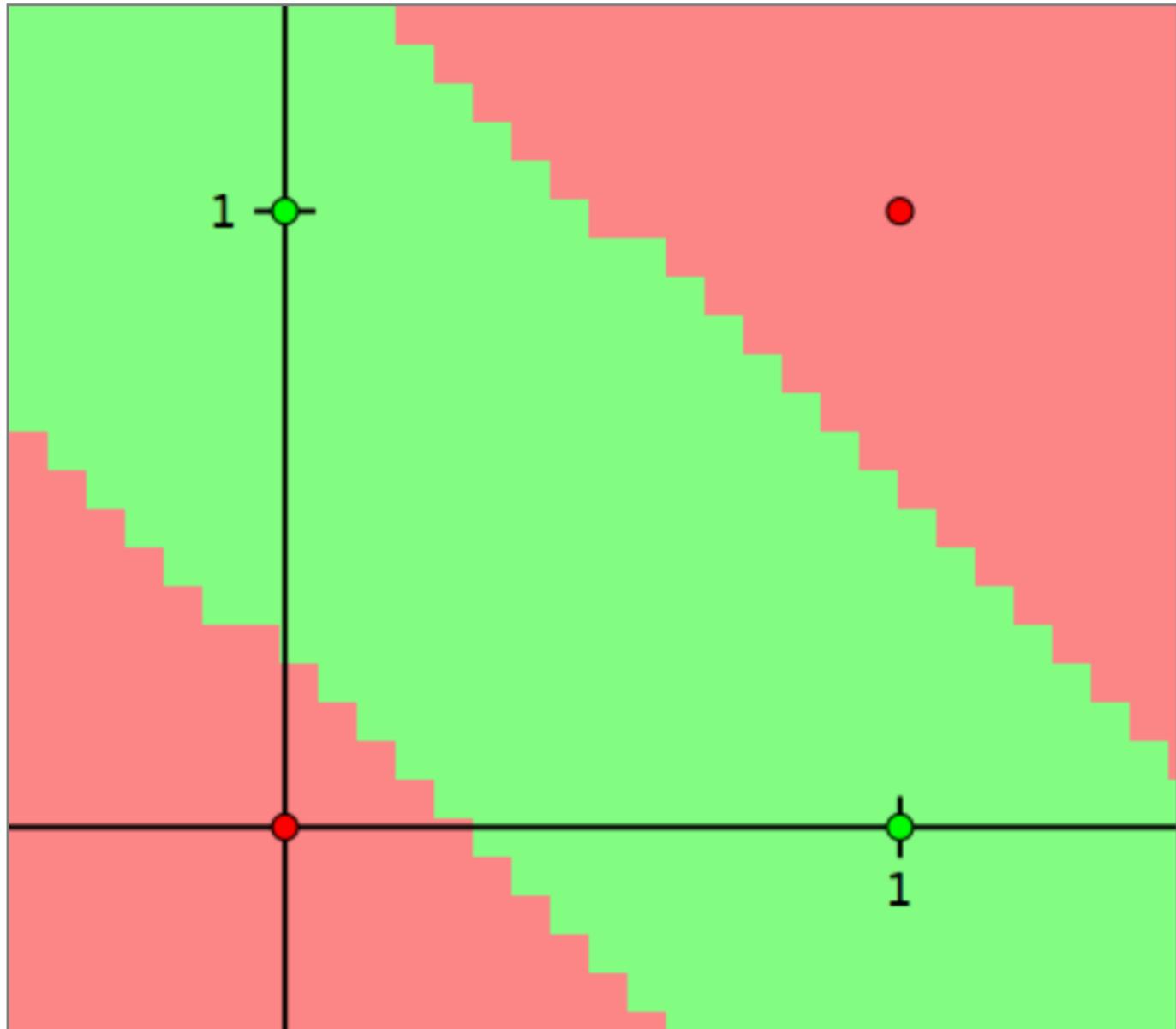
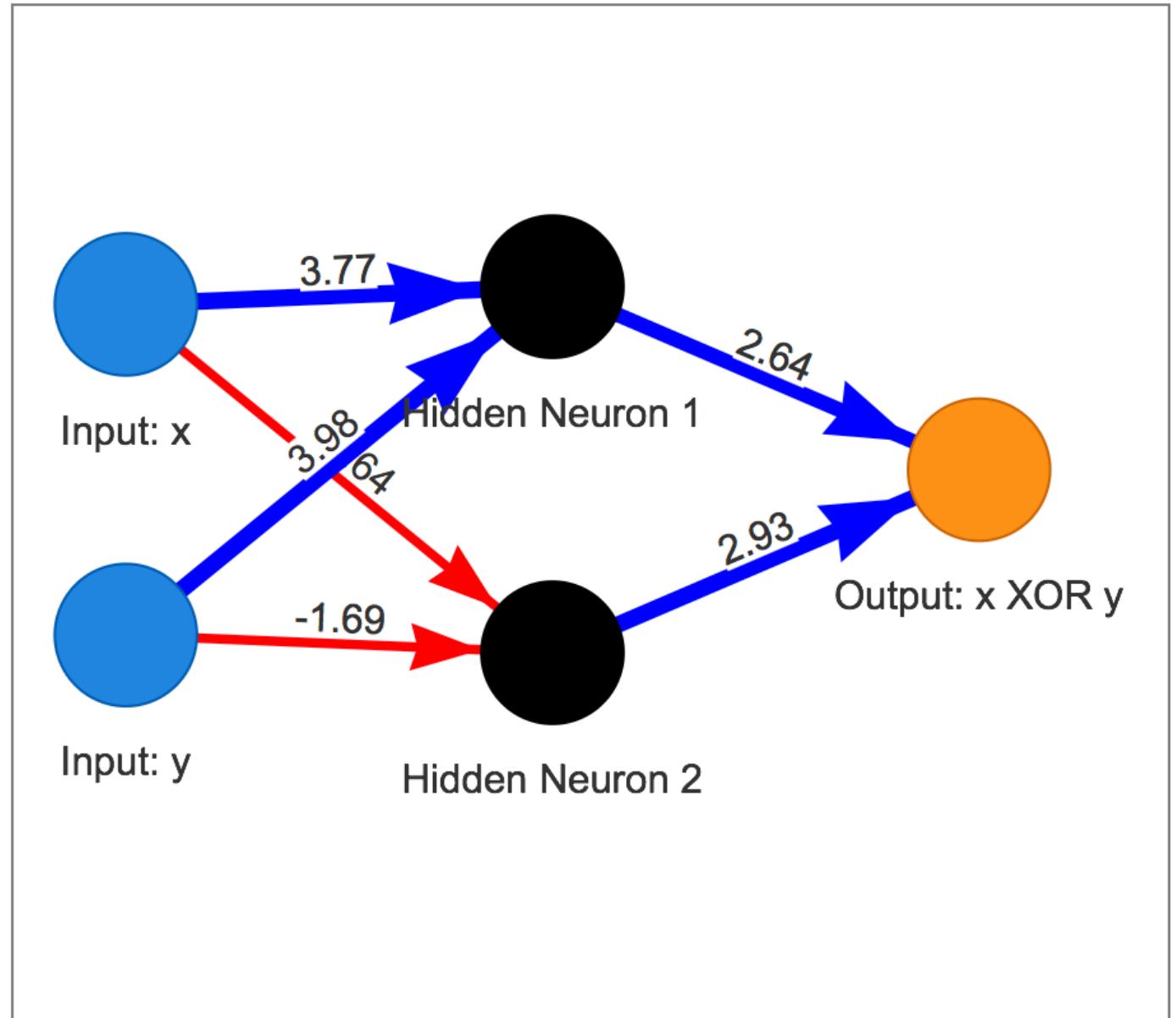
$$\begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$



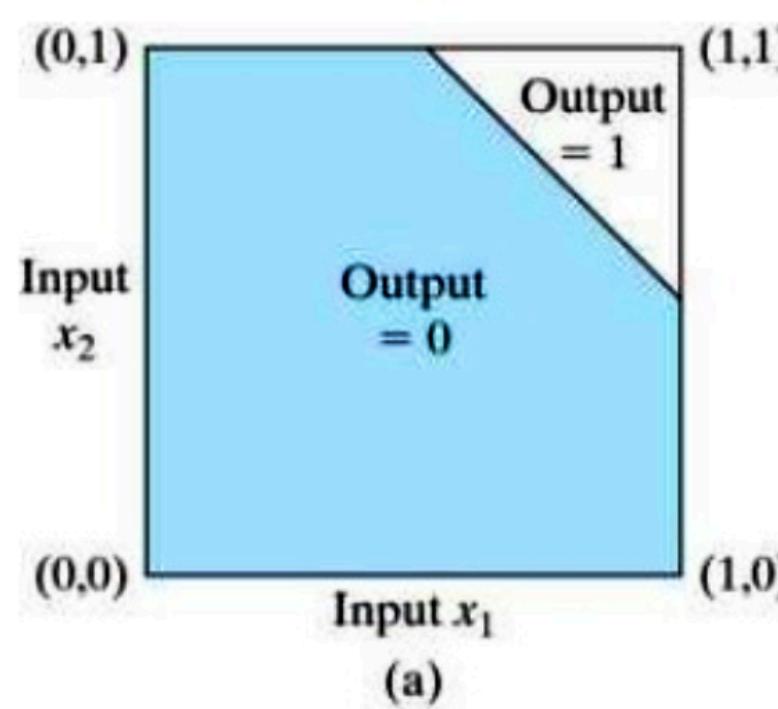
il problema ora è linearmente separabile!

se moltiplichiamo per \mathbf{w} :

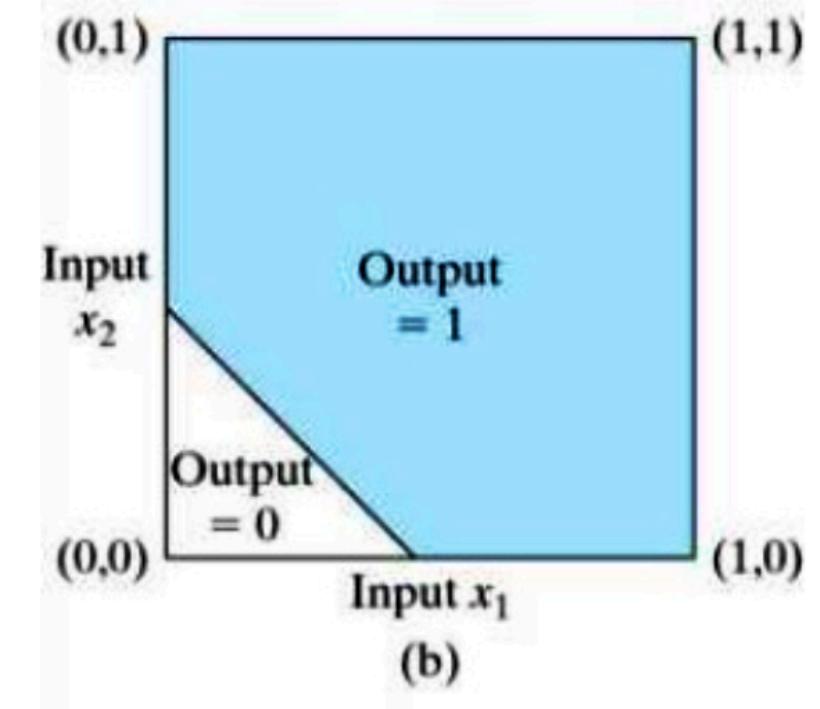
$$\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \text{ XOR}$$



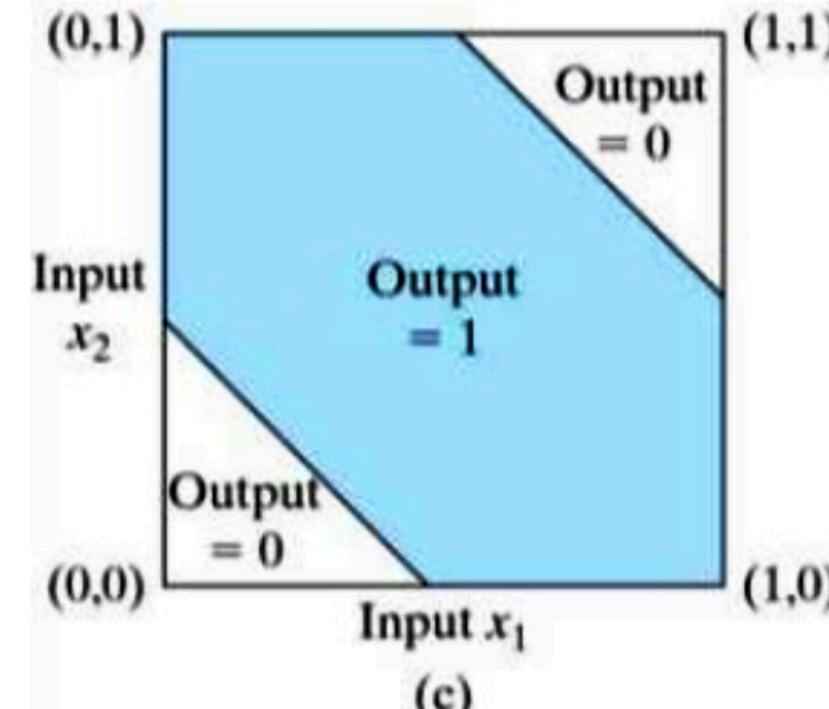
MLP con attivazione sigmoide
con ReLU si ottiene lo stesso
risultato ...



primo neurone



secondo neurone



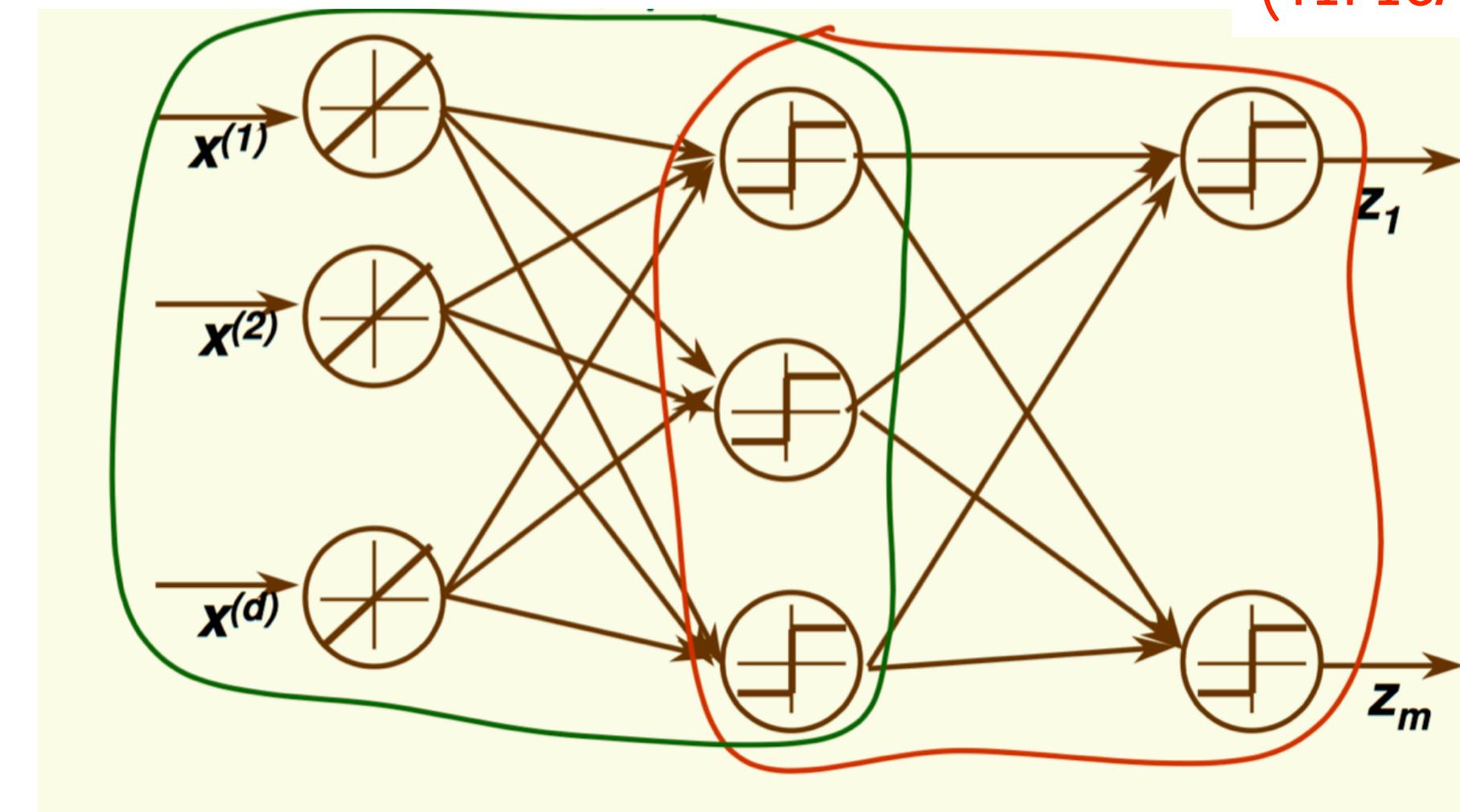
ANN: INTERPRETAZIONE COME ALGORITMO DI MAPPING NON LINEARE

- Un NN può anche essere pensato come un algoritmo che apprende simultaneamente due compiti:

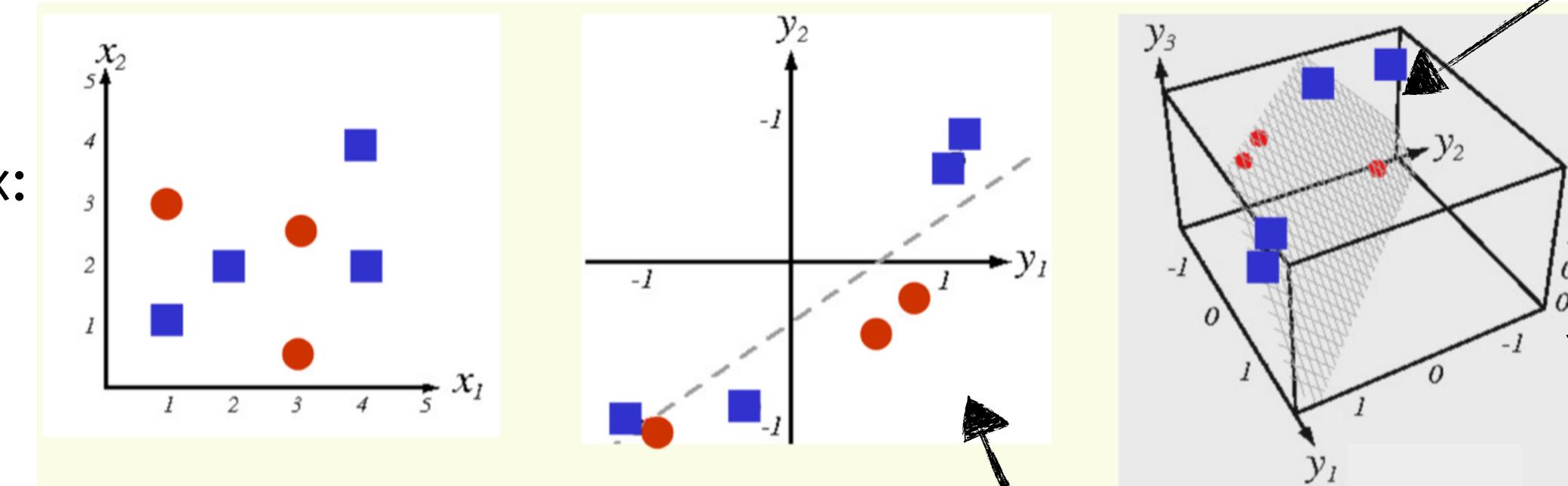
UN MODULO APPRENDE UN MAPPING (NON LINEARE)
DELL'INPUT

$$\mathbf{x} \rightarrow \phi(\mathbf{x})$$

i pesi e le funzioni di attivazione
del primo layer permettono di
apprendere ϕ in modo generale



spazio originale:
pattern x non linearmente separabili x:



NN con 2 neuroni hidden: apprende il mapping non lineare $y = \Phi(x)$ nello spazio 2-dimensionale nel quale i pattern sono "quasi" linearmente separabili ...

QUESTO MODULO APPRENDE UN MAP
(TIPICAMENTE LINEARE), AD ESEMPIO UN CLASSIFICATORE

$$y = \mathbf{w}^t \phi(\mathbf{x}) + w_0$$

i pesi w del layer nascosto permettono di
apprendere il map da ϕ al target desiderato

NN con 3 neuroni hidden: apprende il
mapping $y = \Phi(x)$ in 3-dimensioni in cui i
pattern sono linearmente separabili

REGIONI DI SEPARAZIONE COMPLESSE

si ottengono combinando tra loro più neuroni ...

una rete neurale feed-forward con un singolo layer nascosto contenente un numero finito di neuroni può approssimare funzioni continue in un sottoinsieme compatto di R^n sotto deboli assunzioni sulla funzione di attivazione (teorema di approssimazione universale)

$$F(\mathbf{x}) = \sum_{i=1}^N v_i \varphi(\mathbf{w}_i^t \mathbf{x} + b_i)$$

Struttura	Regioni di decisione	Forma generale
	Semispazi delimitati da iperpiani	
	Regioni convesse	
	Regioni di forma arbitraria	

NOTA IMPORTANTE: il teorema non dice nulla sull'effettiva possibilità di apprendere in modo semplice i parametri della rete!



UNIVERSAL APPROXIMATION THEOREM

- enunciato formale per shallow networks (unbounded width):

Let $\varphi(\cdot)$ be a nonconstant, bounded, and monotonically-increasing continuous function. Let I_m denote the m -dimensional unit hypercube $[0, 1]^m$. The space of continuous functions on I_m is denoted by $C(I_m)$. Then, given any $\varepsilon > 0$ and any function $f \in C(I_m)$, there exist an integer N , real constants $v_i, b_i \in \mathbb{R}$ and real vectors $w_i \in \mathbb{R}^m$, where $i = 1, \dots, N$, such that we may define:

$$F(x) = \sum_{i=1}^N v_i \varphi(w_i^T x + b_i)$$

as an approximate realization of the function f where f is independent of φ ; that is,

$$|F(x) - f(x)| < \varepsilon$$

for all $x \in I_m$. In other words, functions of the form $F(x)$ are dense in $C(I_m)$.

This still holds when replacing I_m with any compact subset of \mathbb{R}^m .

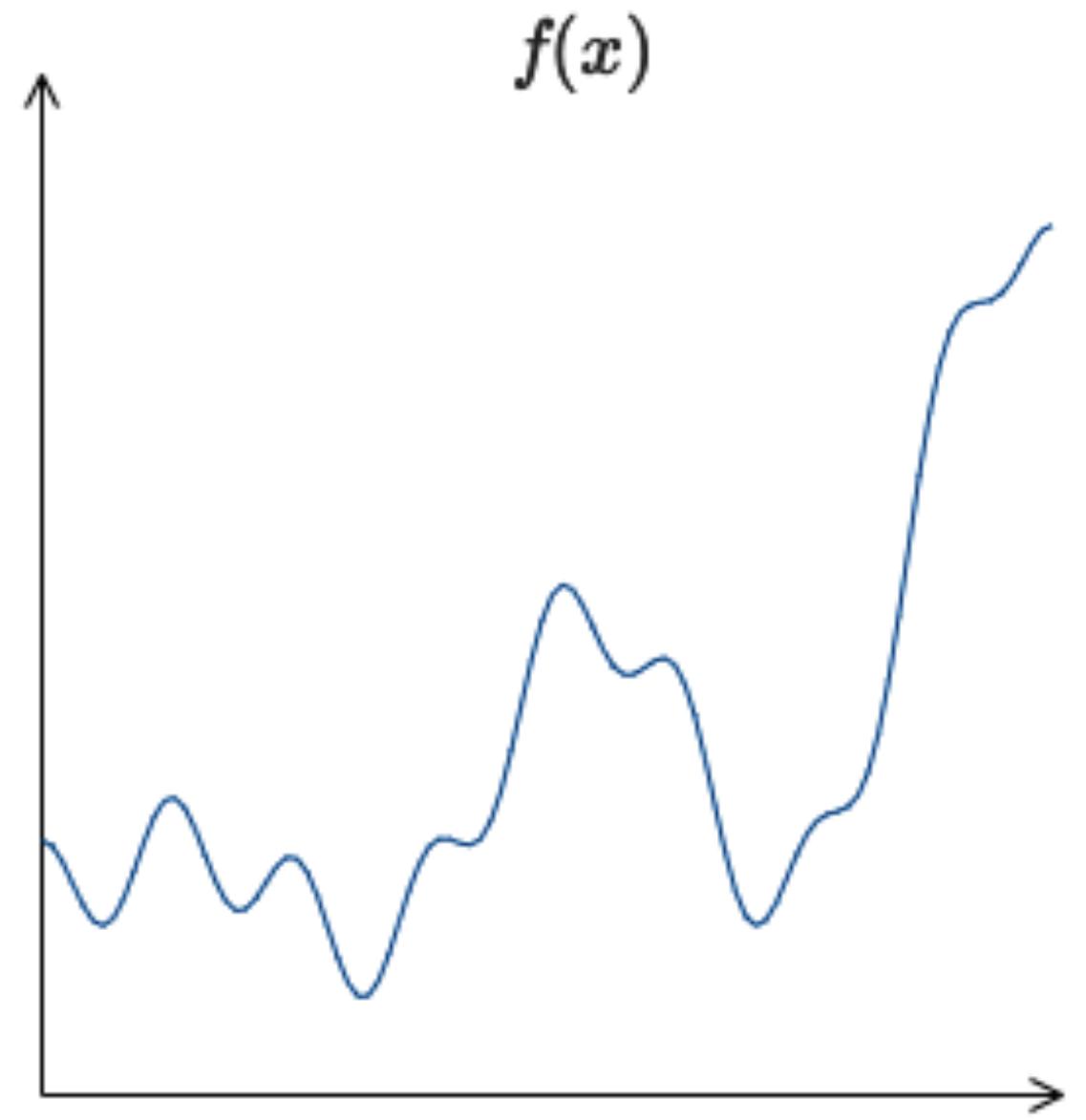
se interessati
dimostrazione [qui](#)

- unbounded width implica che il numero di neuroni nel layer nascosto N cresca esponenzialmente con la dimensione della funzione da approssimare
- si può dimostrare per deep-NN una versione bounded valida per funzioni lebesgue integrabili e per attivazioni ReLU in cui N cresce linearmente con la dimensione della funzione da approssimare

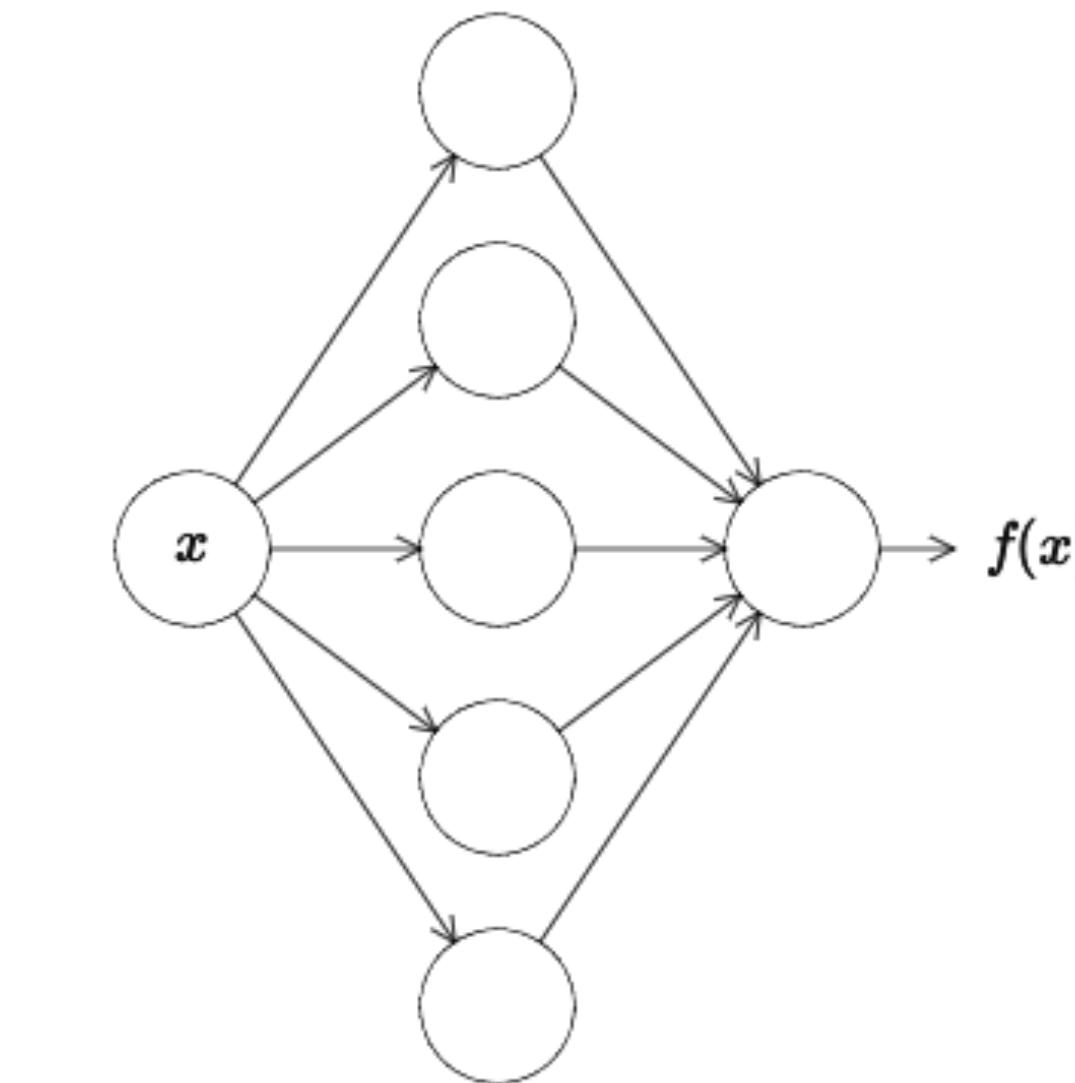


DIMOSTRAZIONE VISUALE

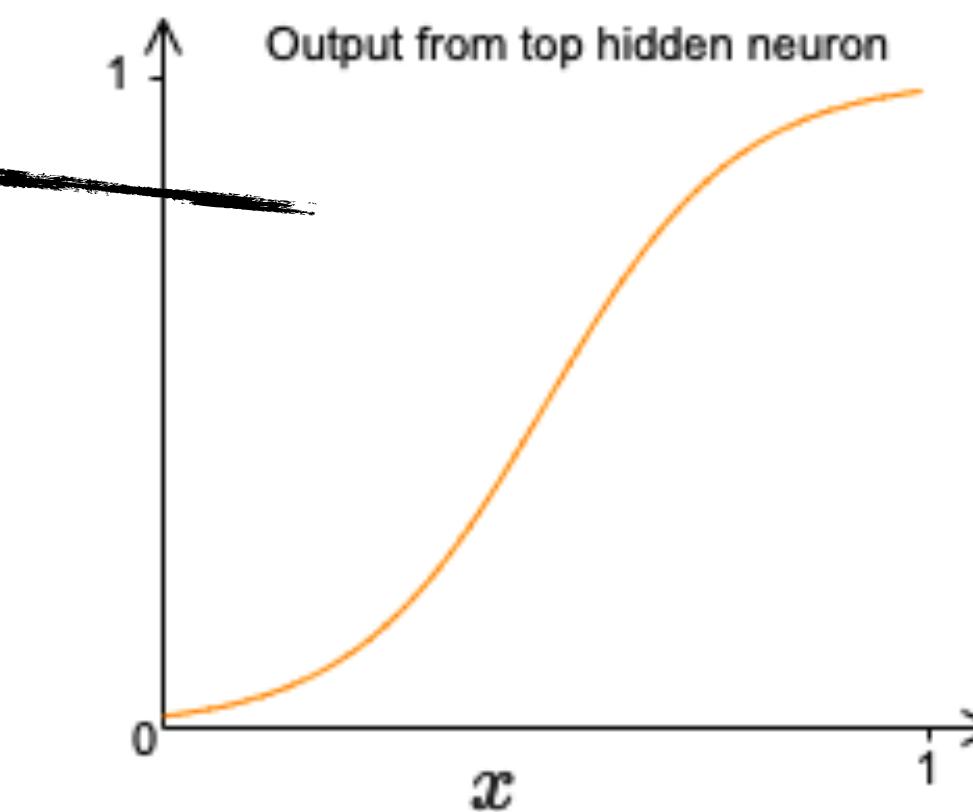
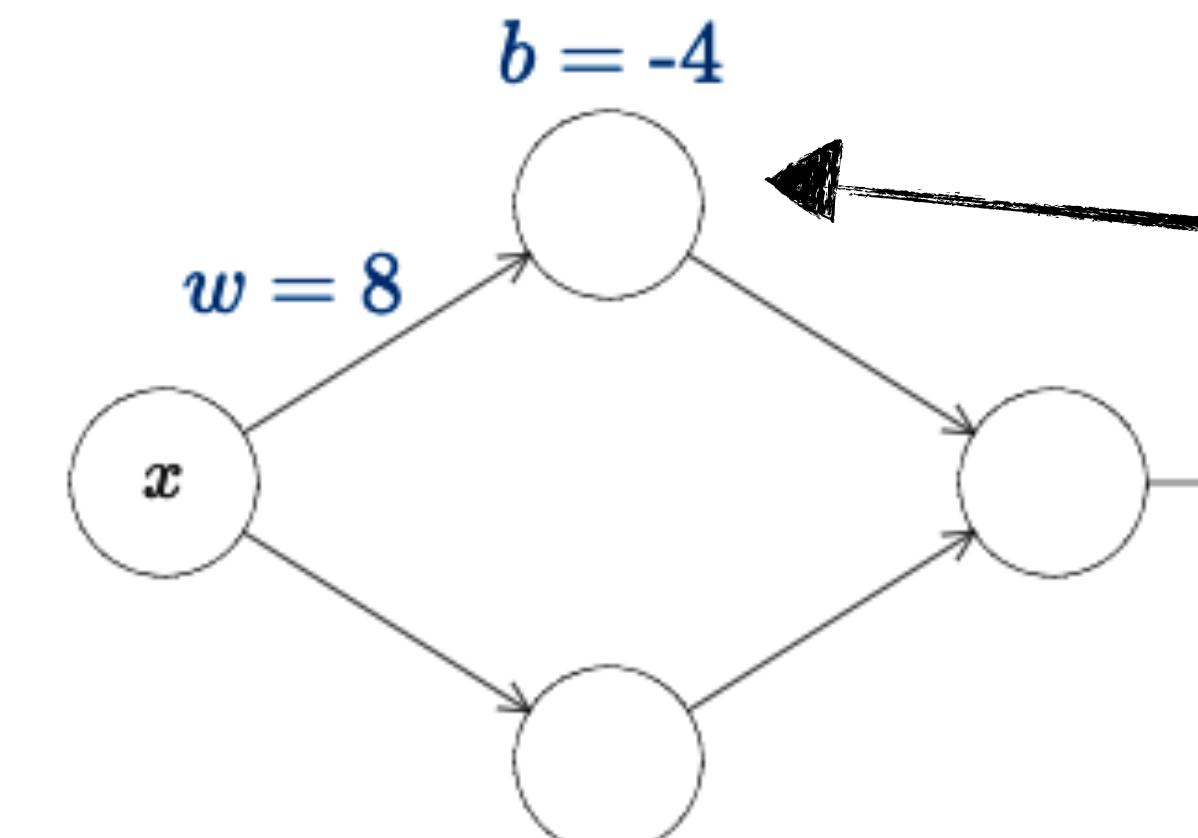
credit to l'ottima descrizione divulgativa di [M.Nielsen](#)



funzione continua che vogliamo approssimare con il NN

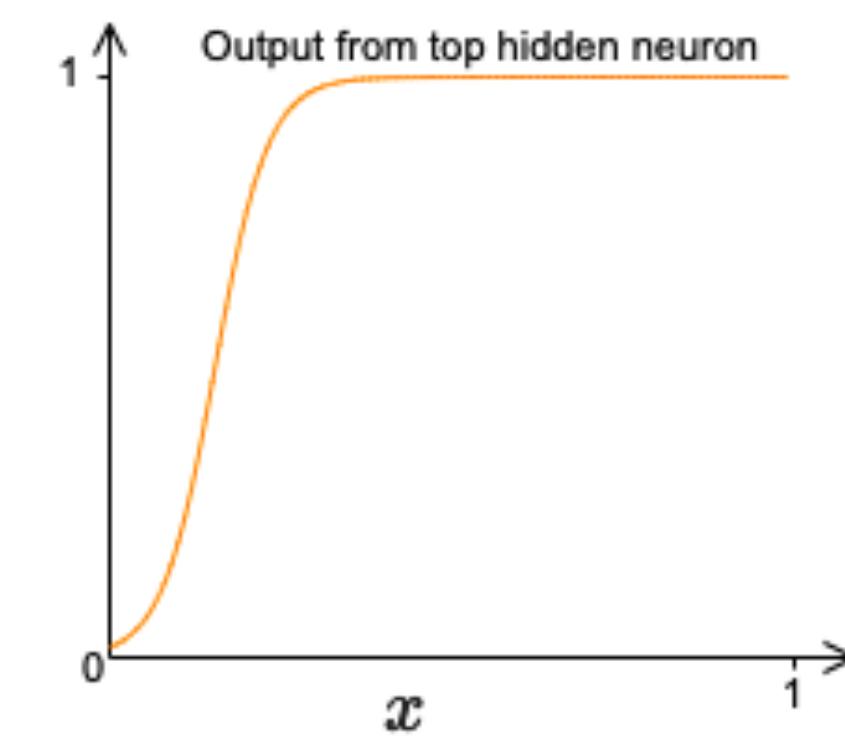
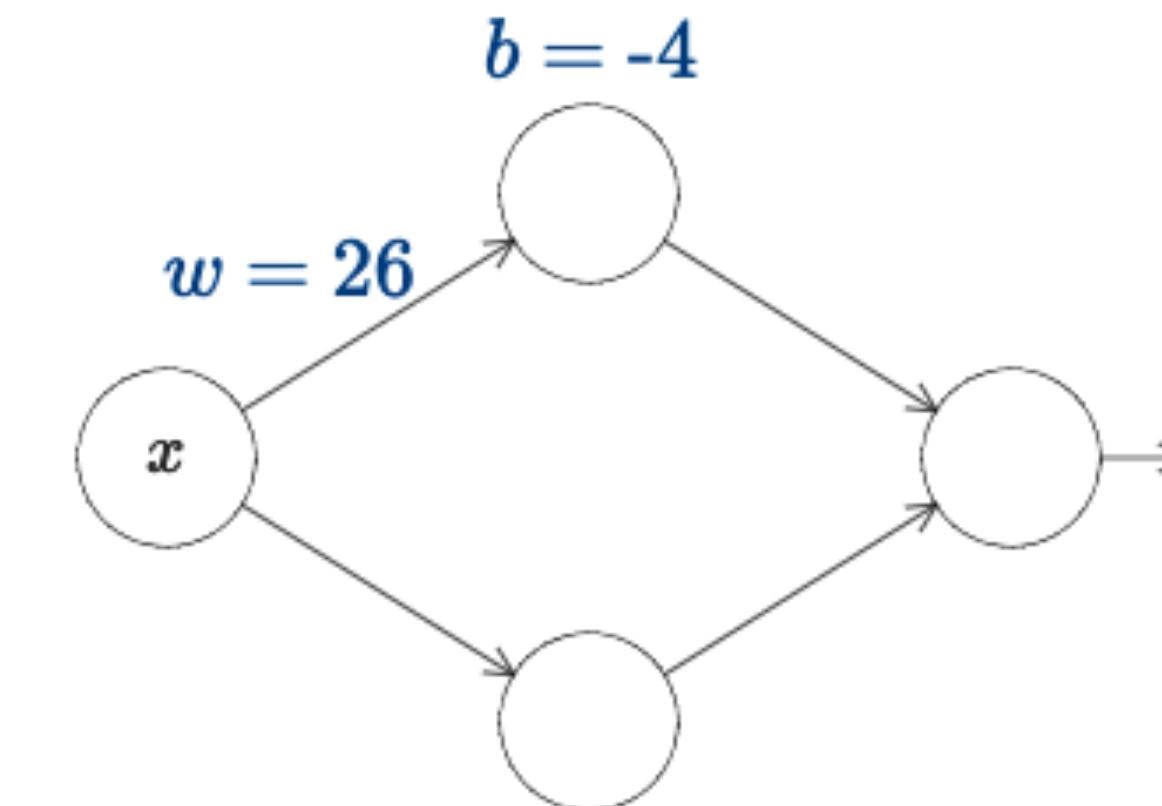
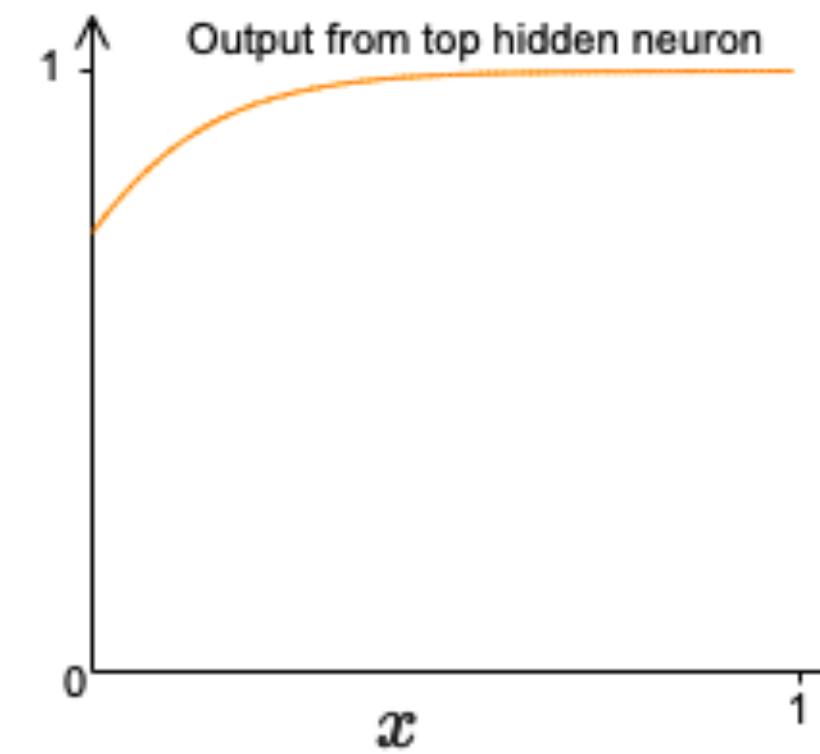
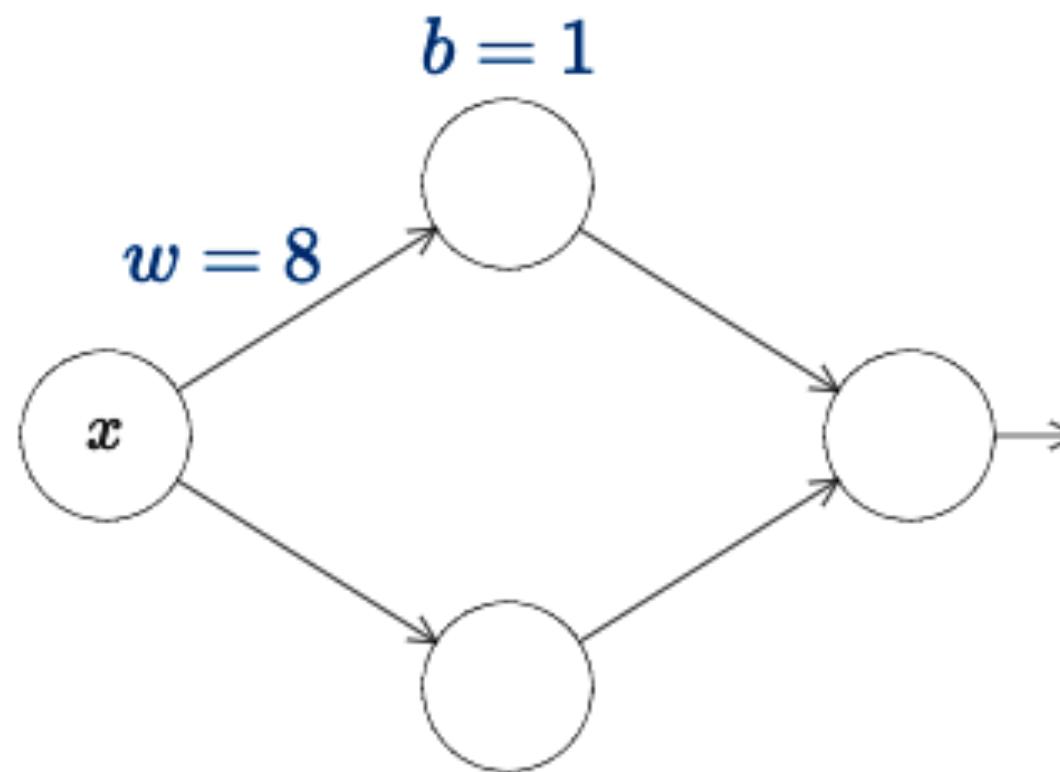
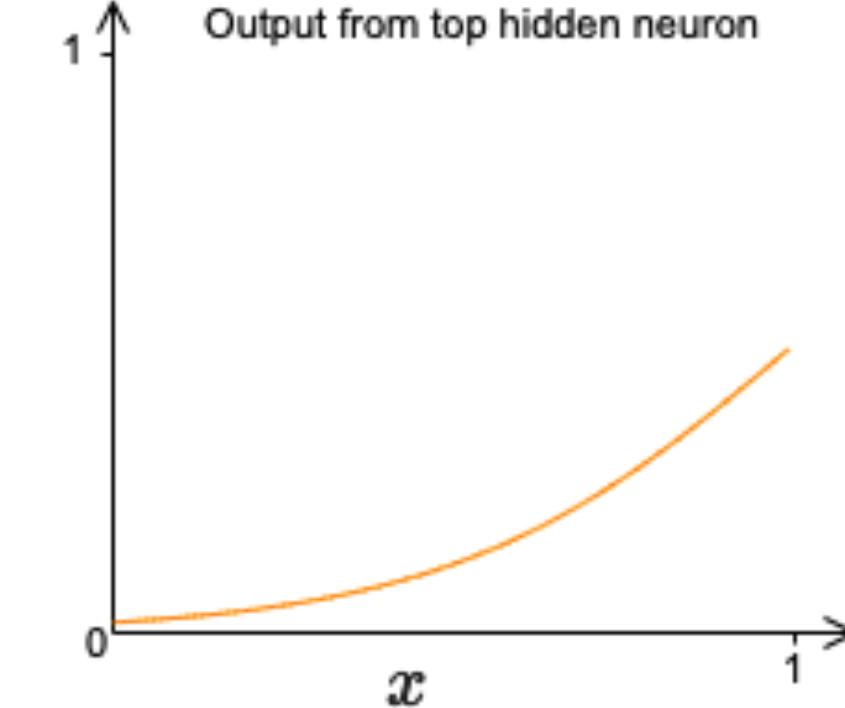
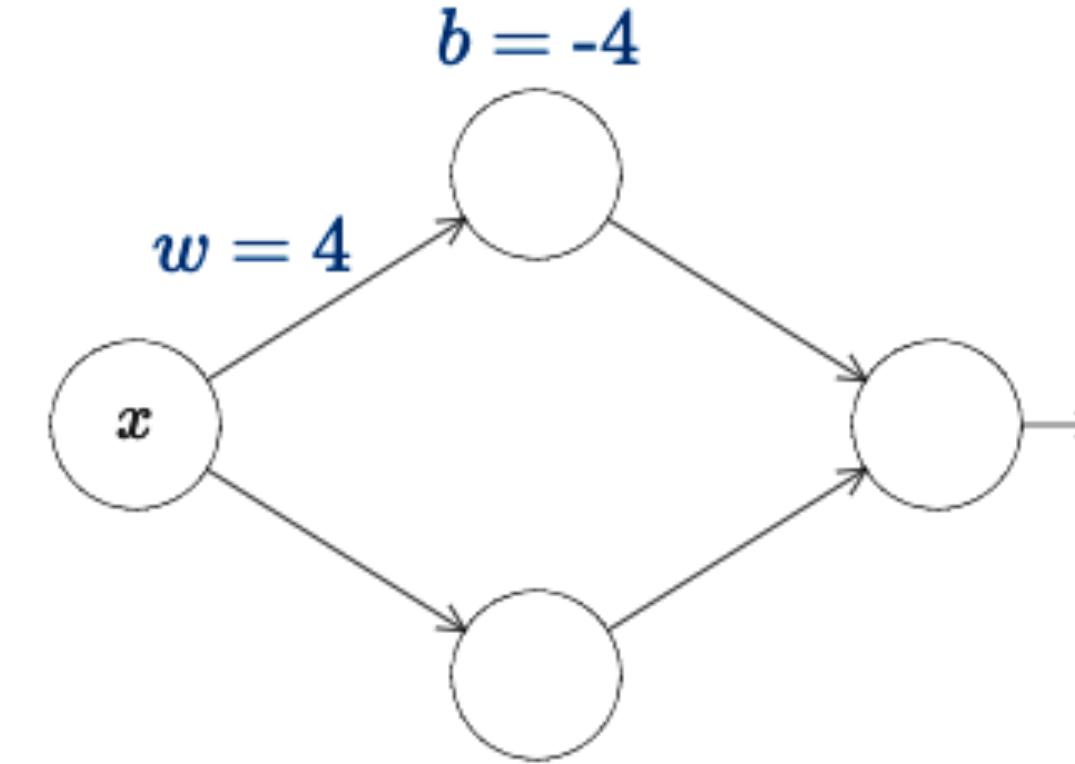
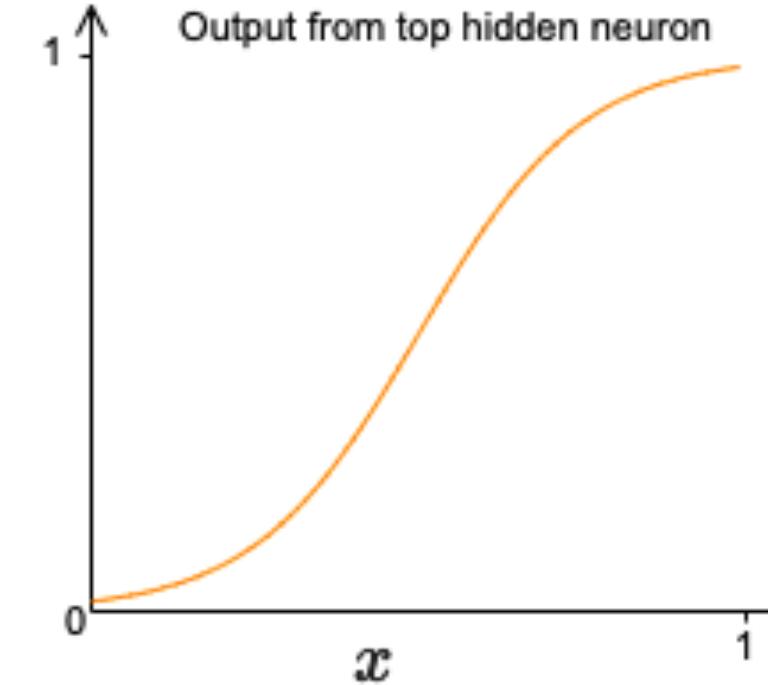
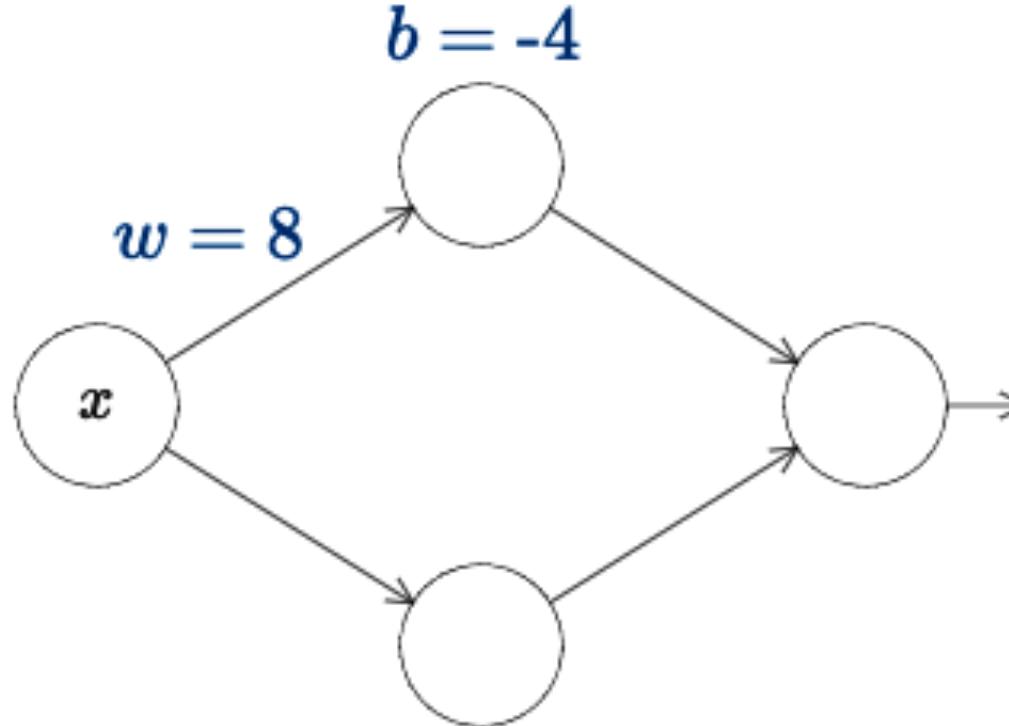


iniziamo con il considerare un semplice NN con 1 solo hidden layer formato da 2 soli neuroni



cambiando i valori dei pesi (w) e del bias (b) l'uscita con attivazione sigmoide $\sigma(wx+b)$ cambia ...

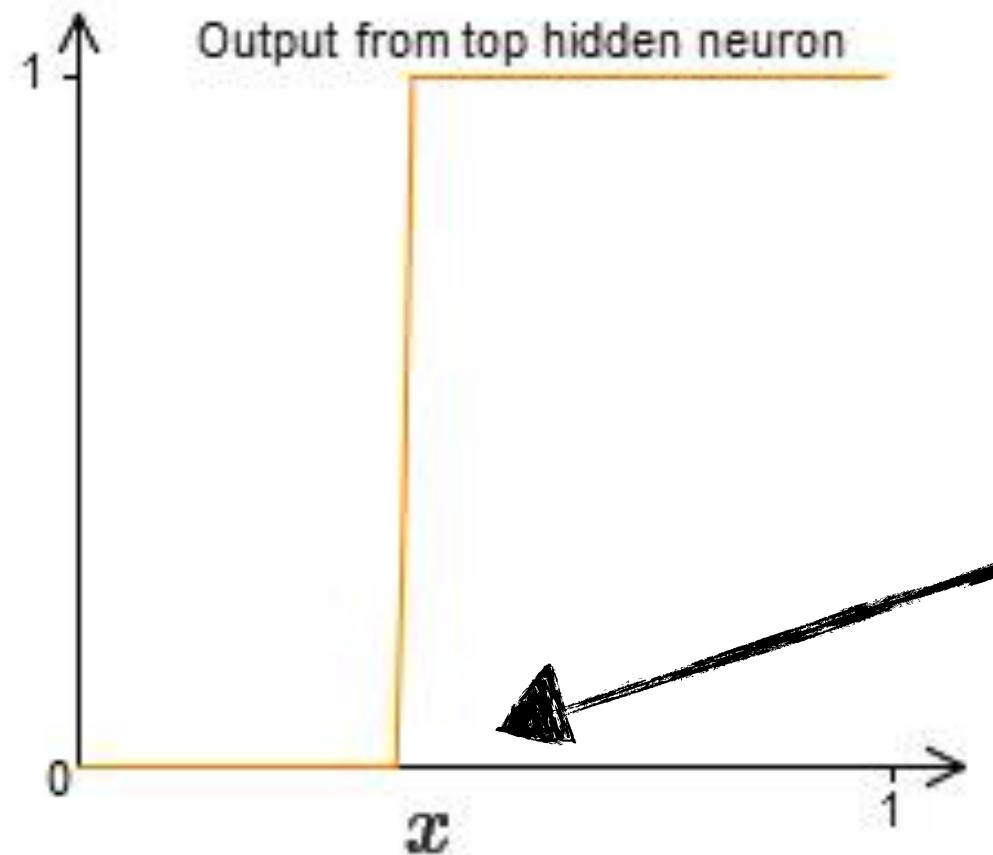
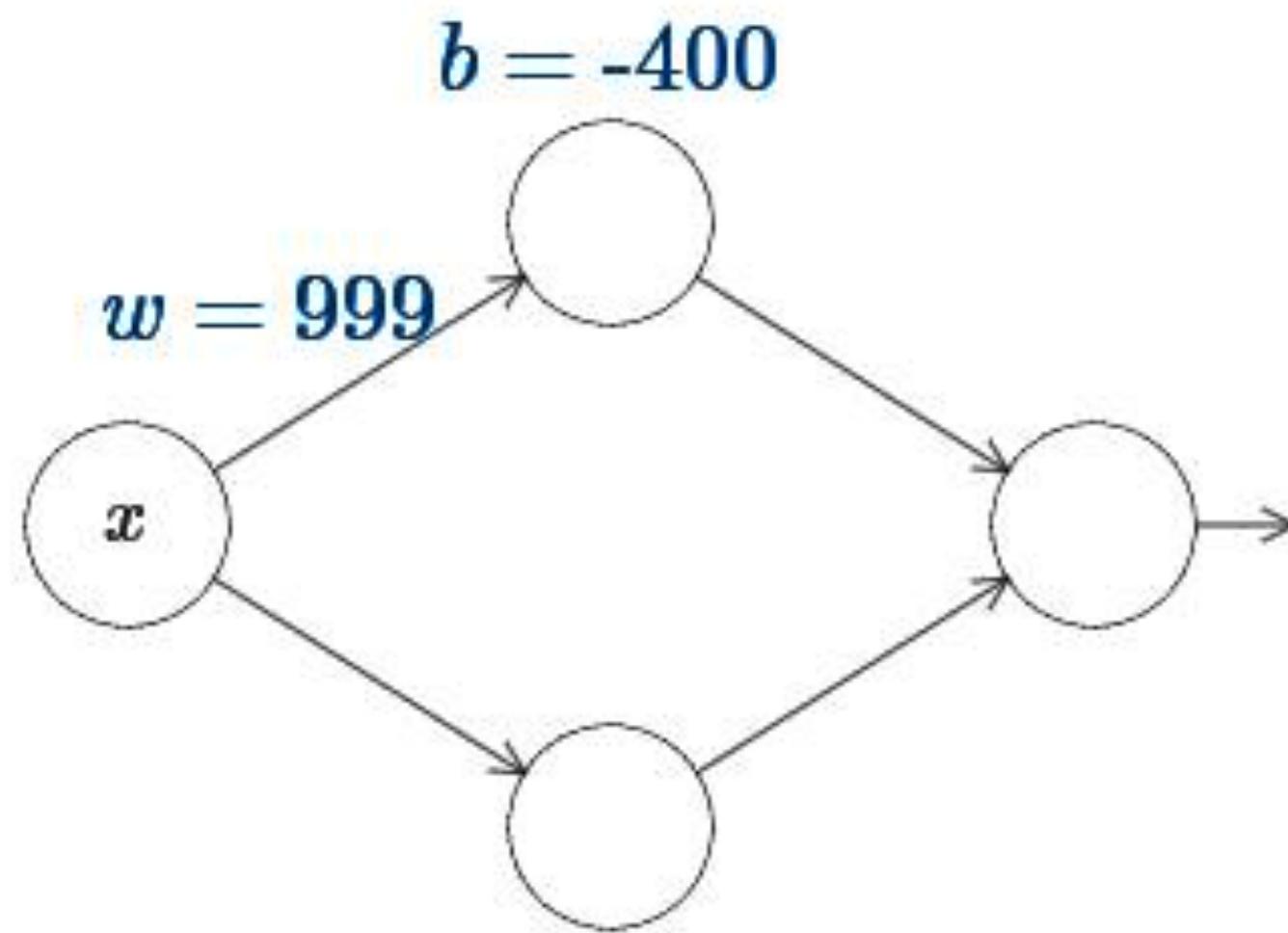
$$\sigma(z) = \frac{1}{1 - e^{-z}}$$



b sposta lungo x la curva senza
cambiarne la shape

w cambia la shape (pendenza) da step function (w
molto grande) a forma allargata (w piccolo)

giocando con b e usando w grandi si possono ottenere quasi step function in qualsiasi punto ...

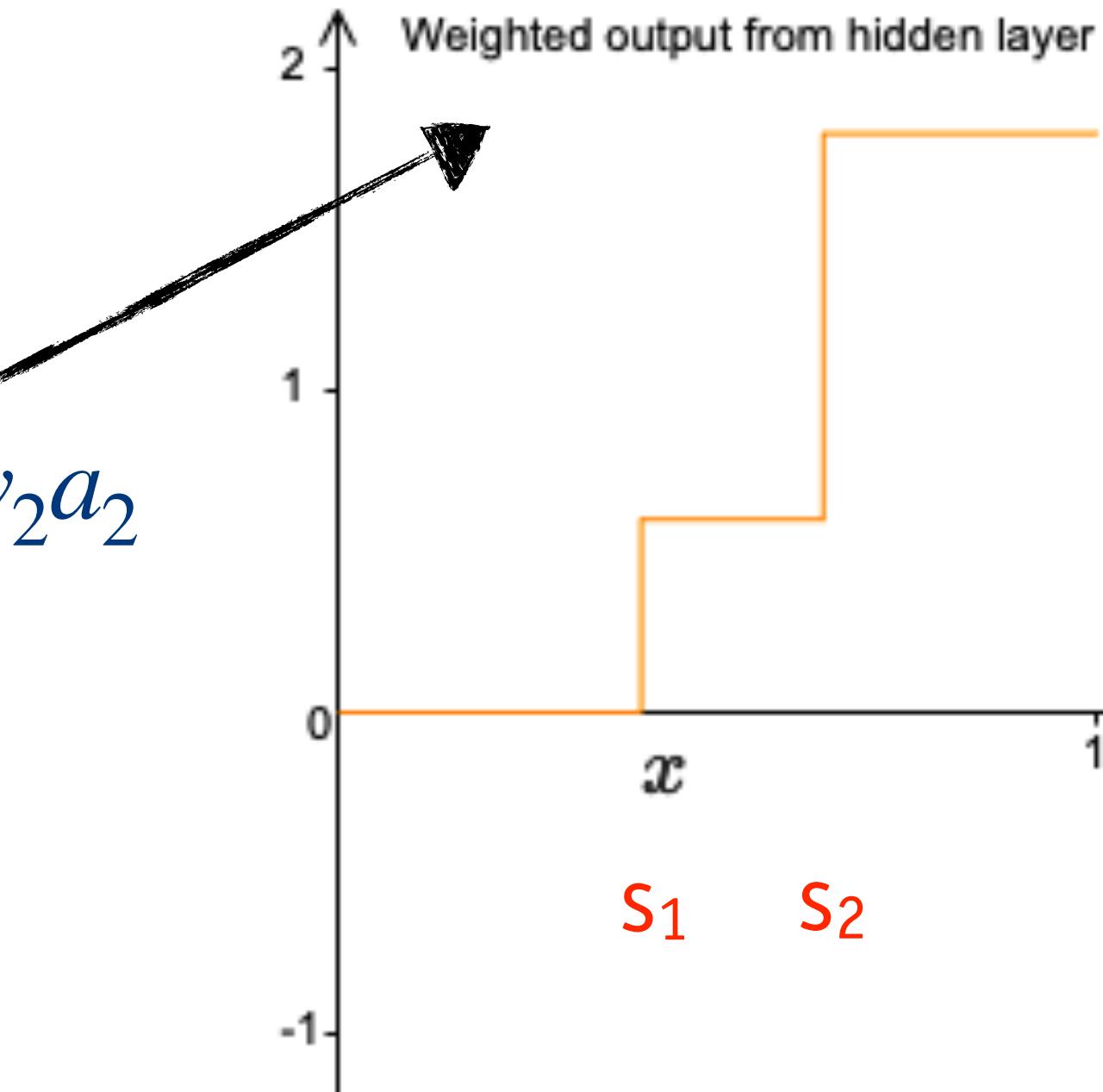
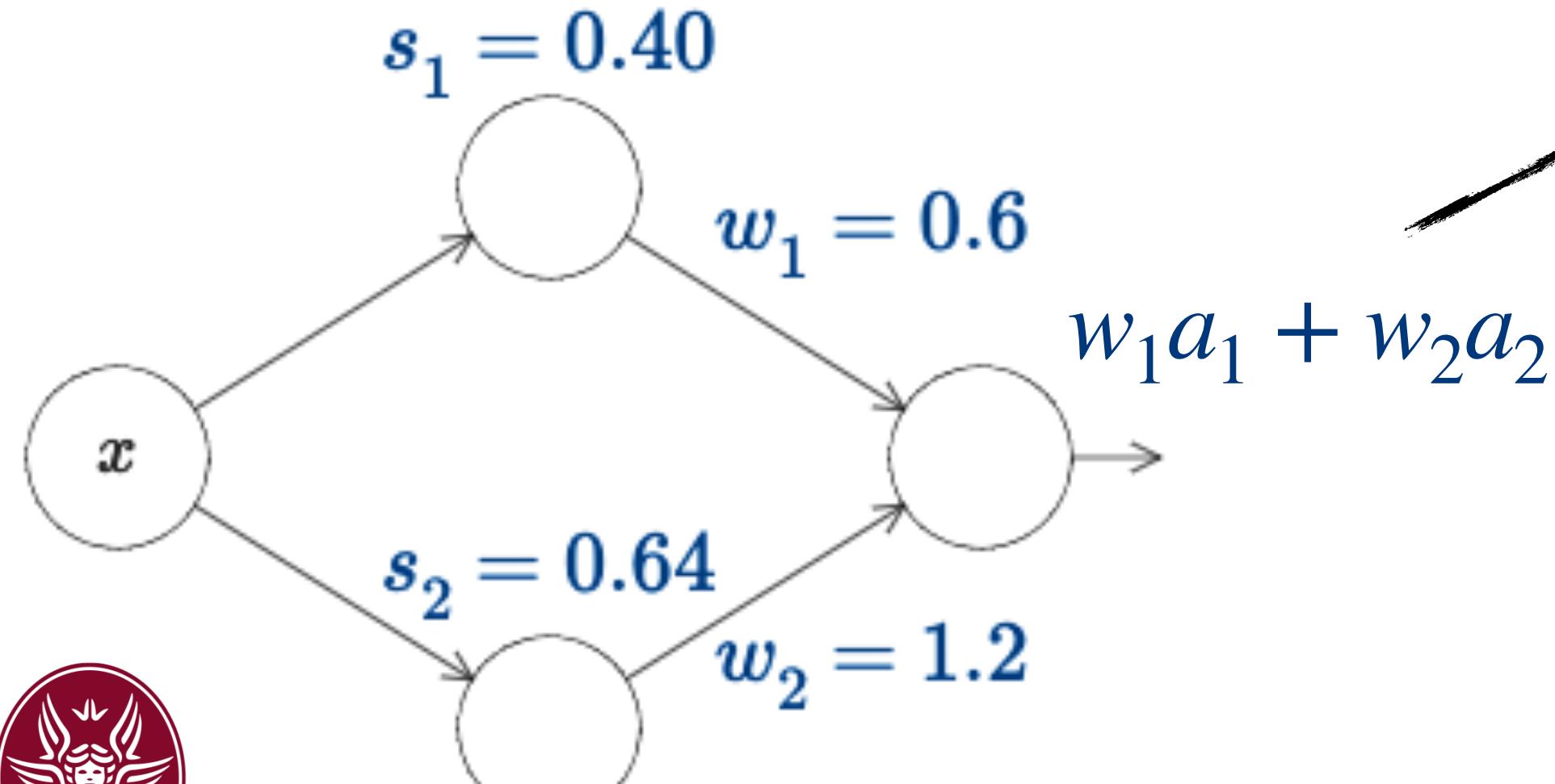


posizione dello
step

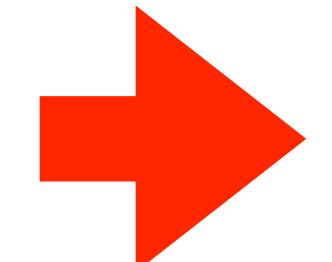
$$s = -\frac{b}{w}$$

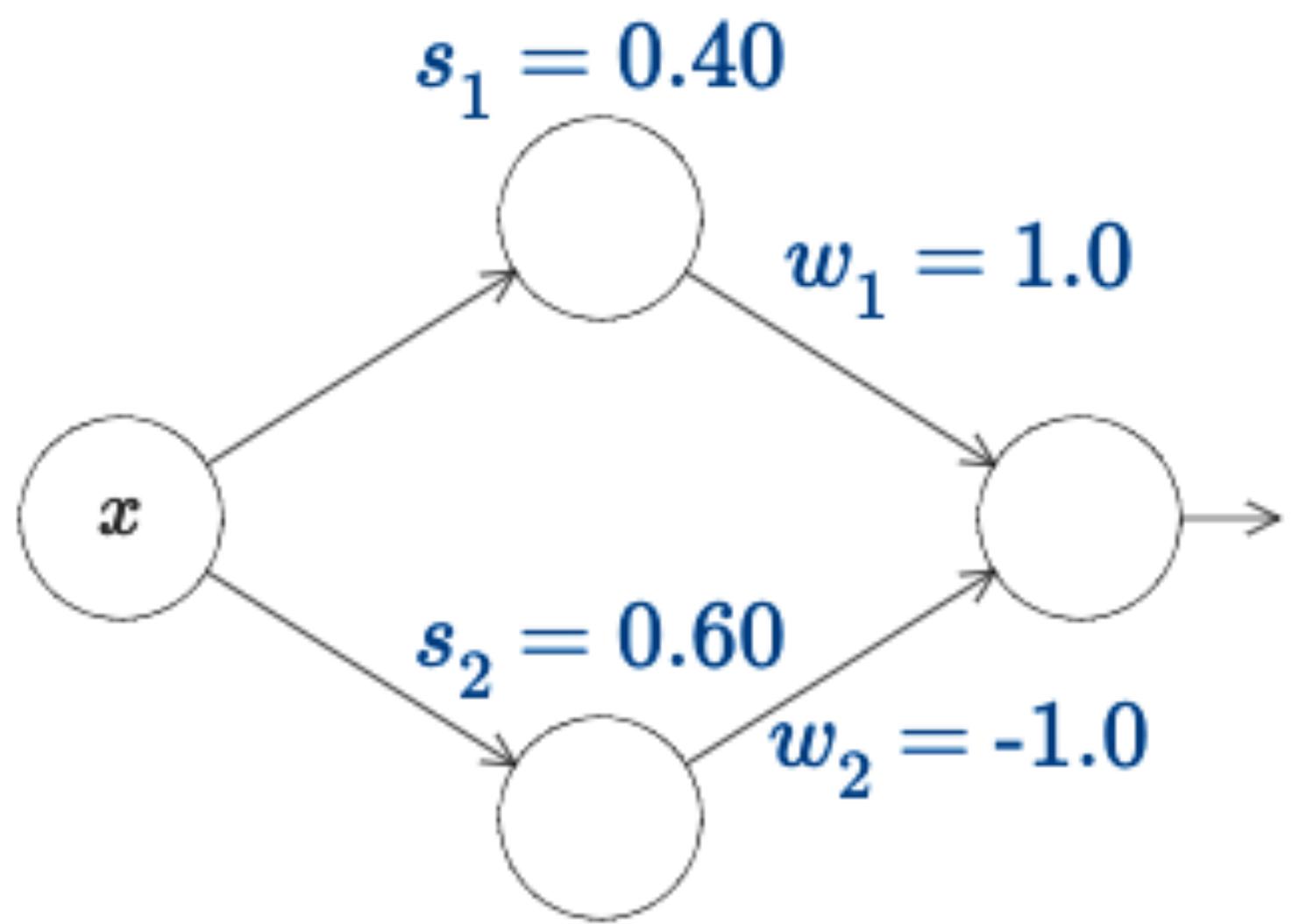
usiamo w fisso
grande e la
posizione è solo
controllata da b

usiamo ora anche il secondo neurone:

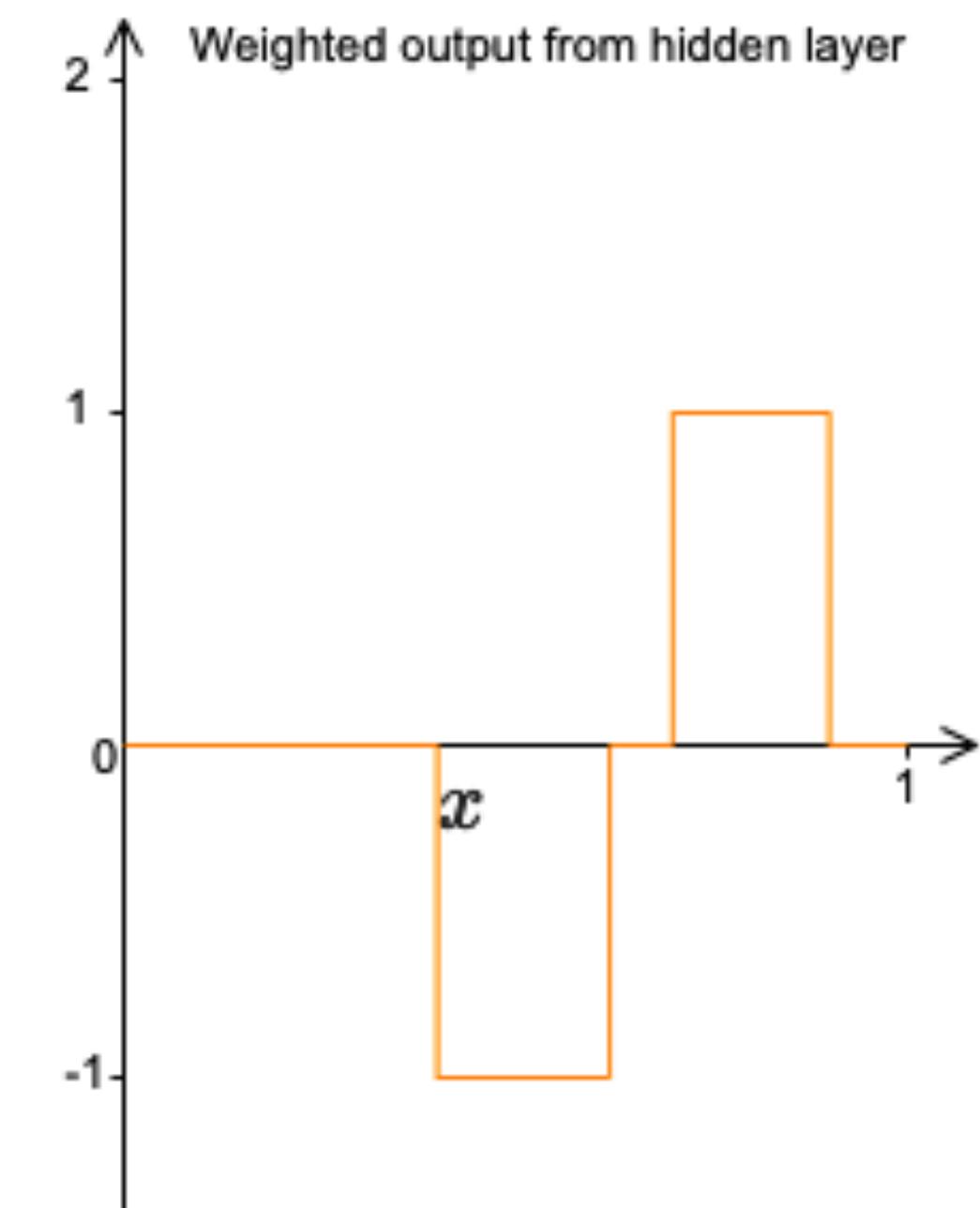
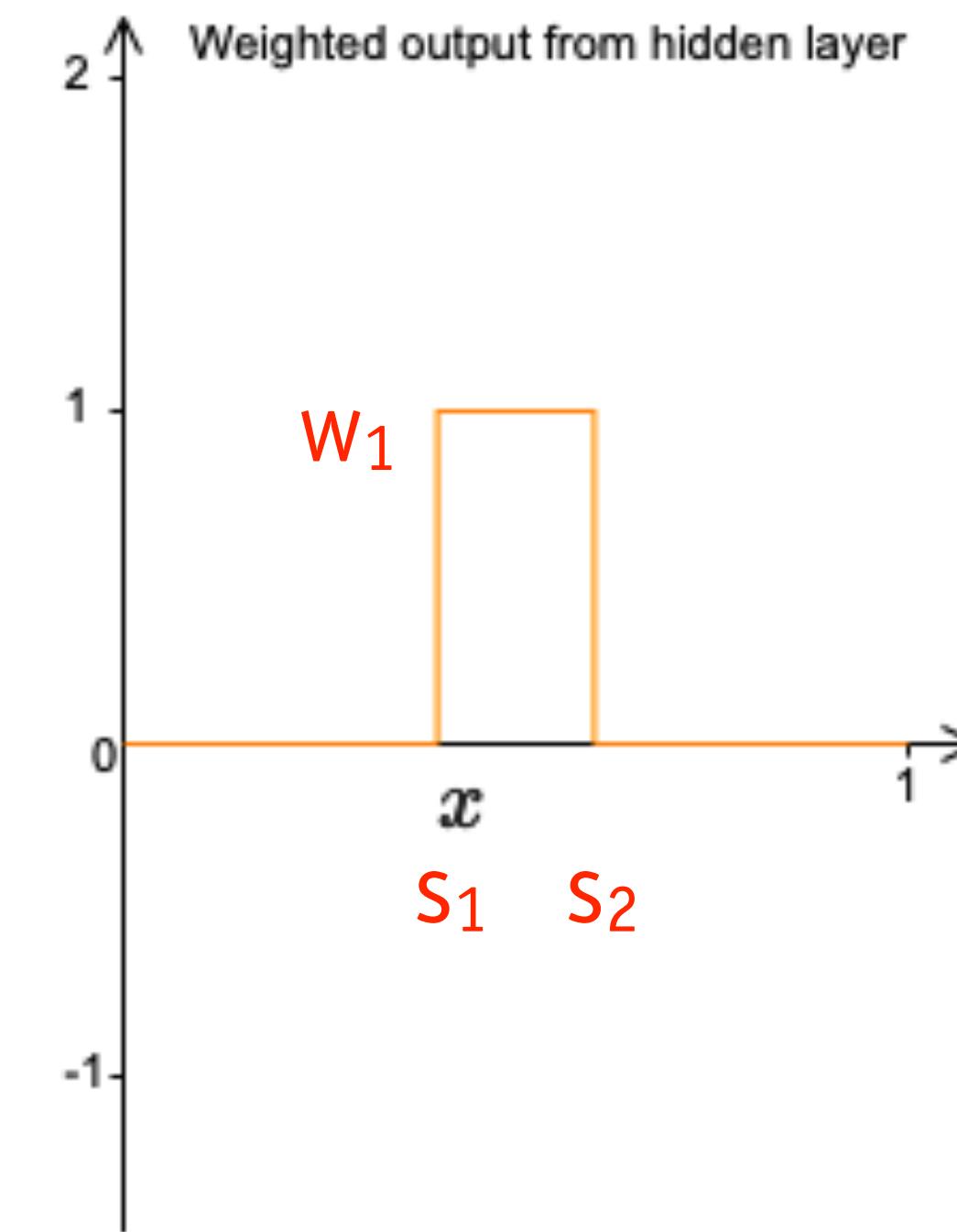
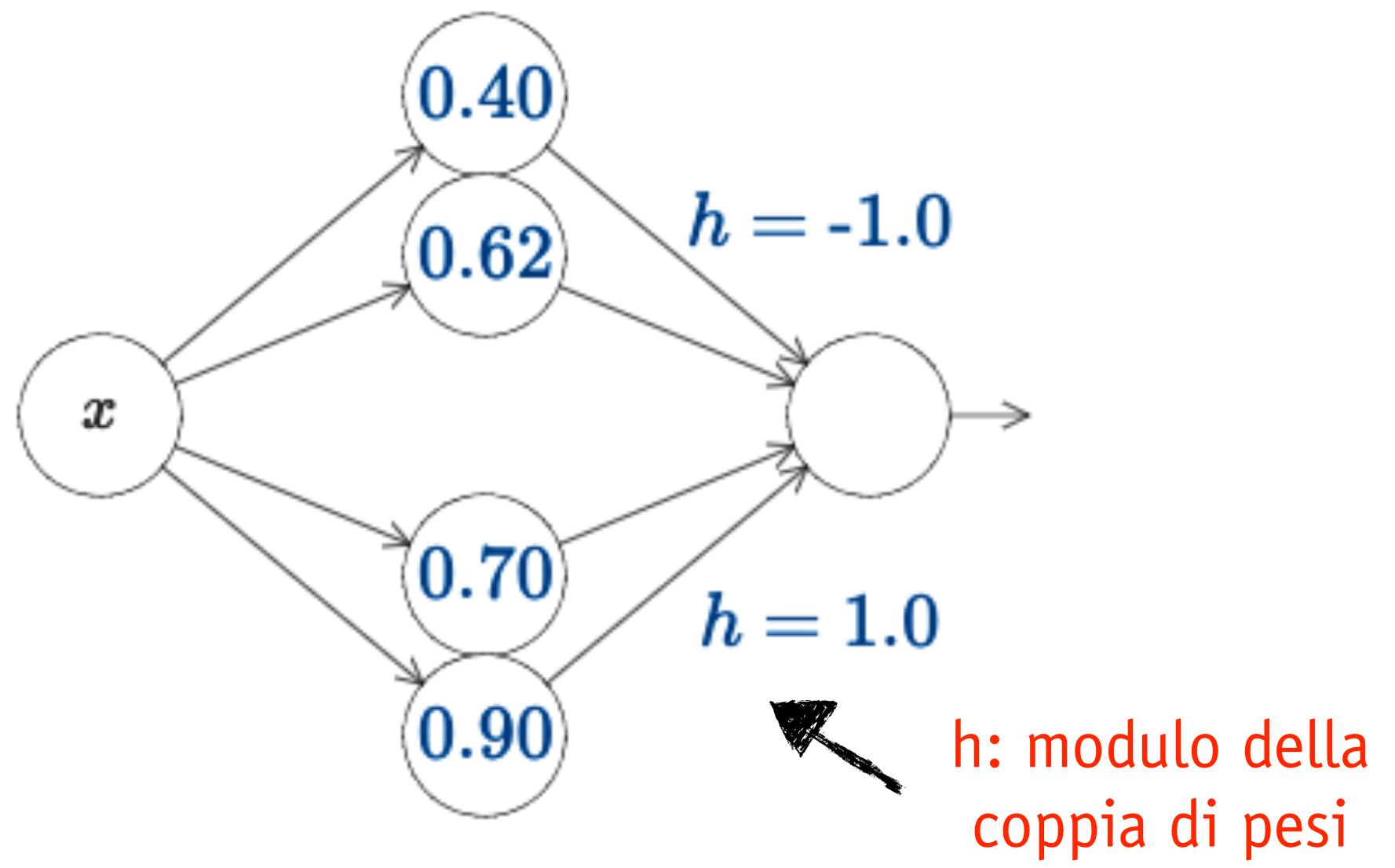


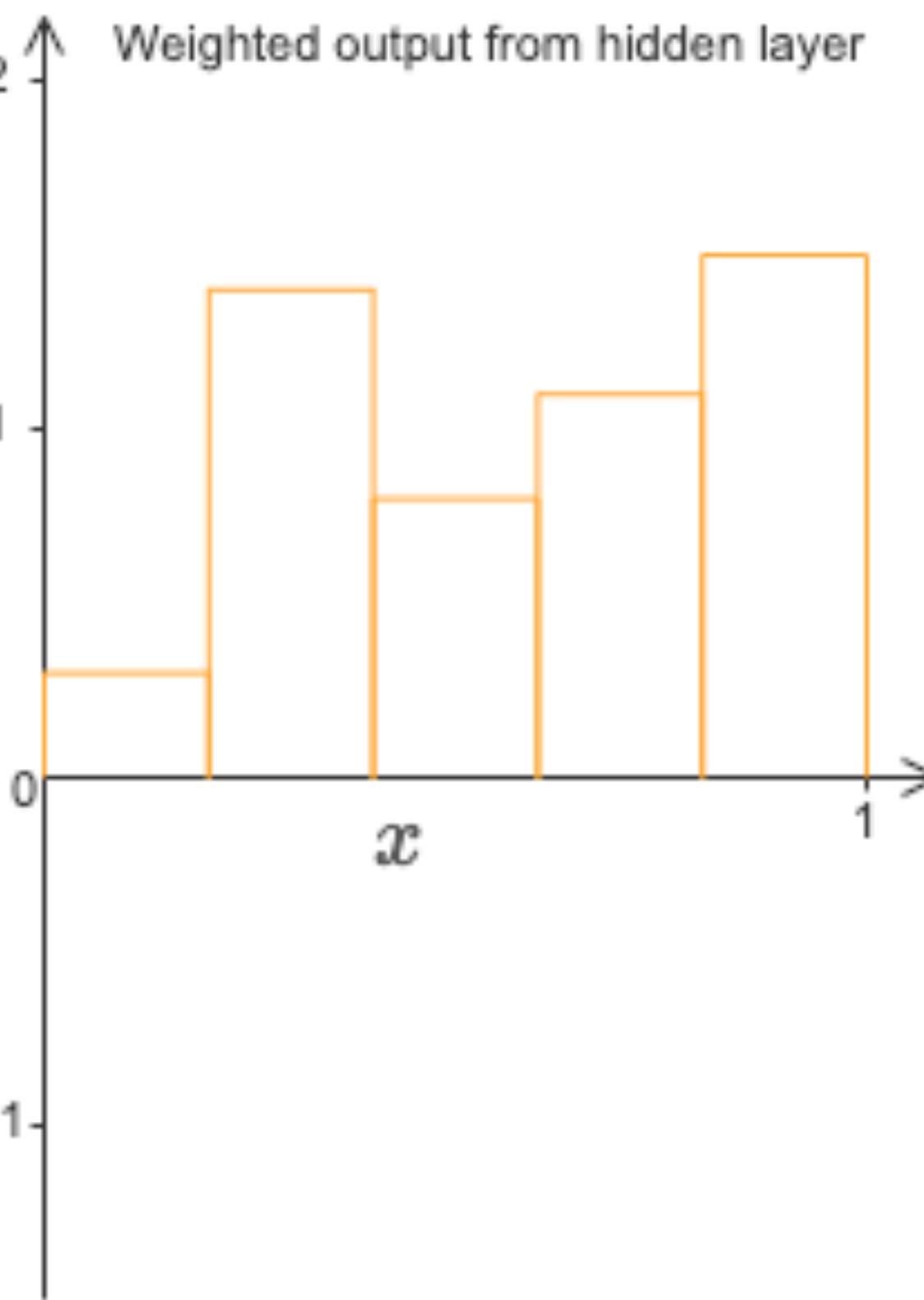
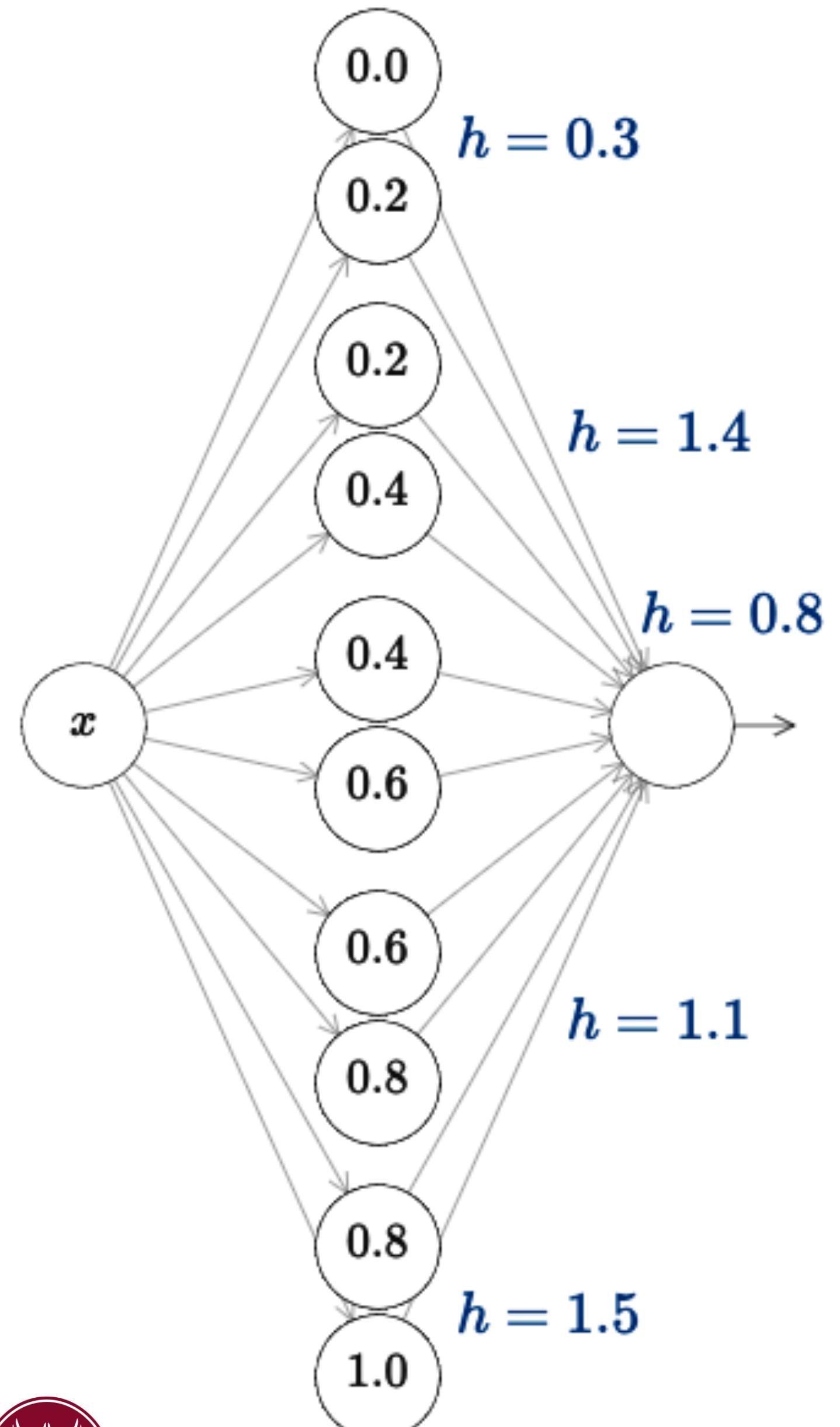
fissando $w_2 = -w_1$
possiamo ottenere
una output uniforme
che va da s_1 a s_2
alto w_1



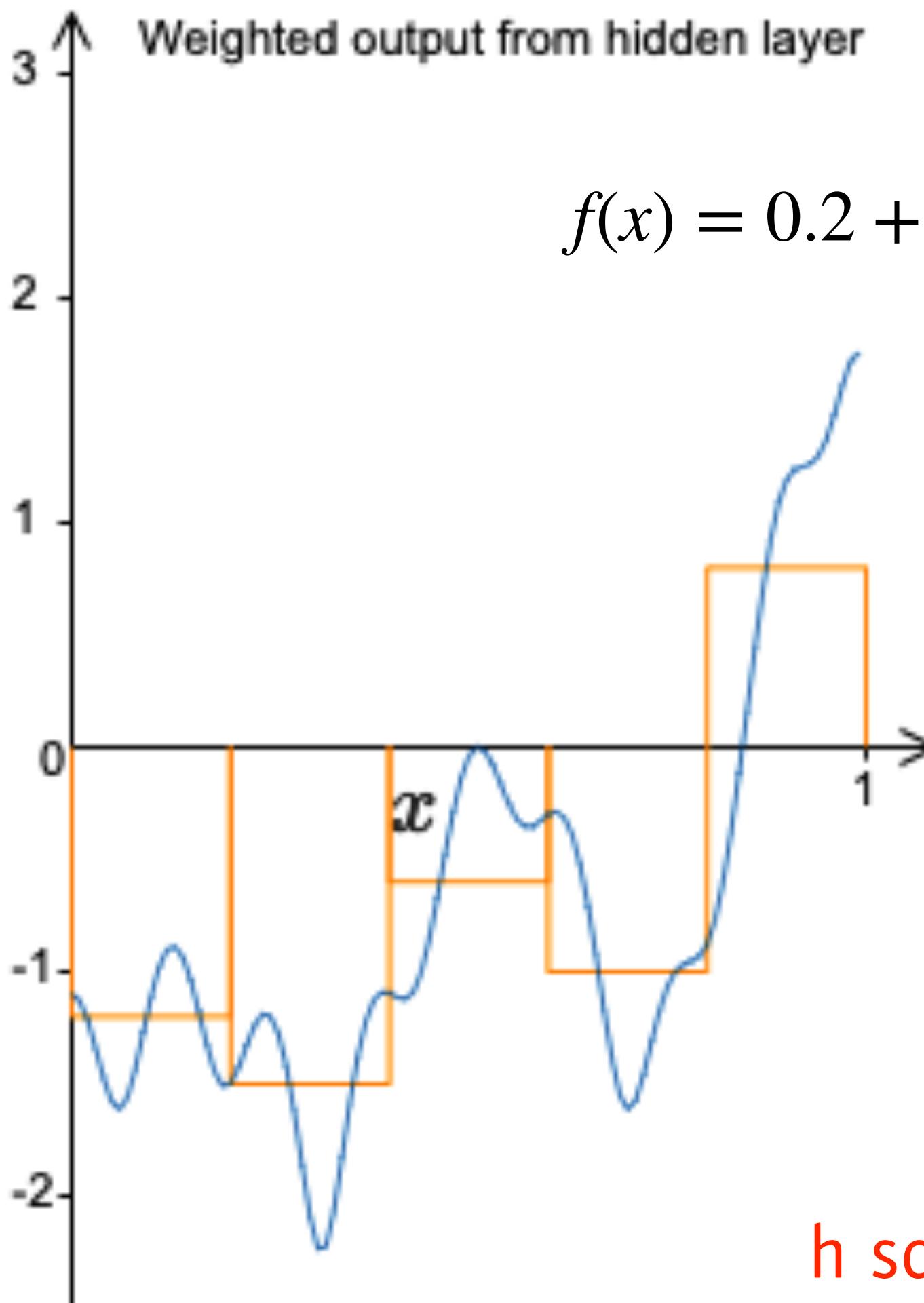
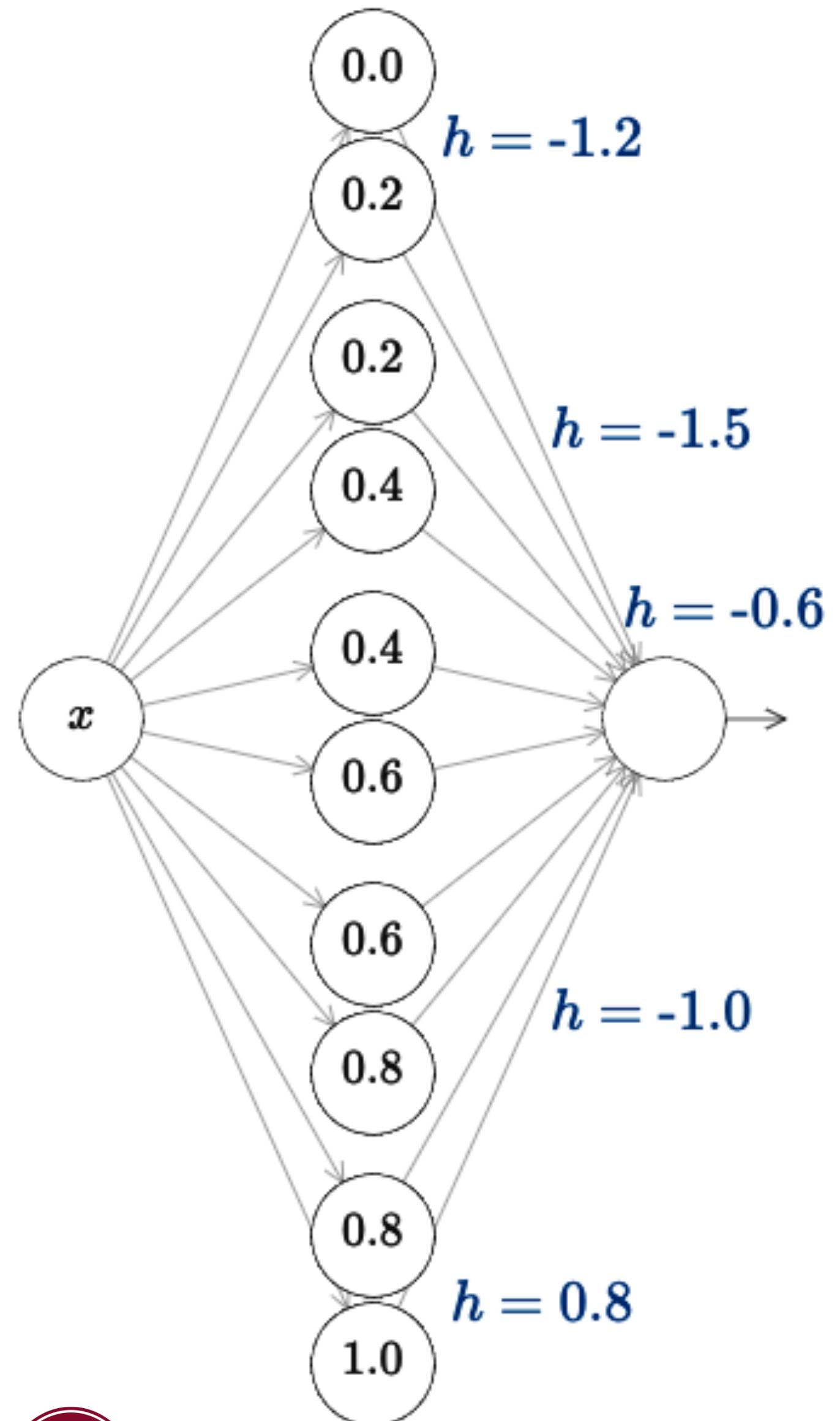


possiamo ora accoppiare coppie di neuroni per creare coppie di bump:



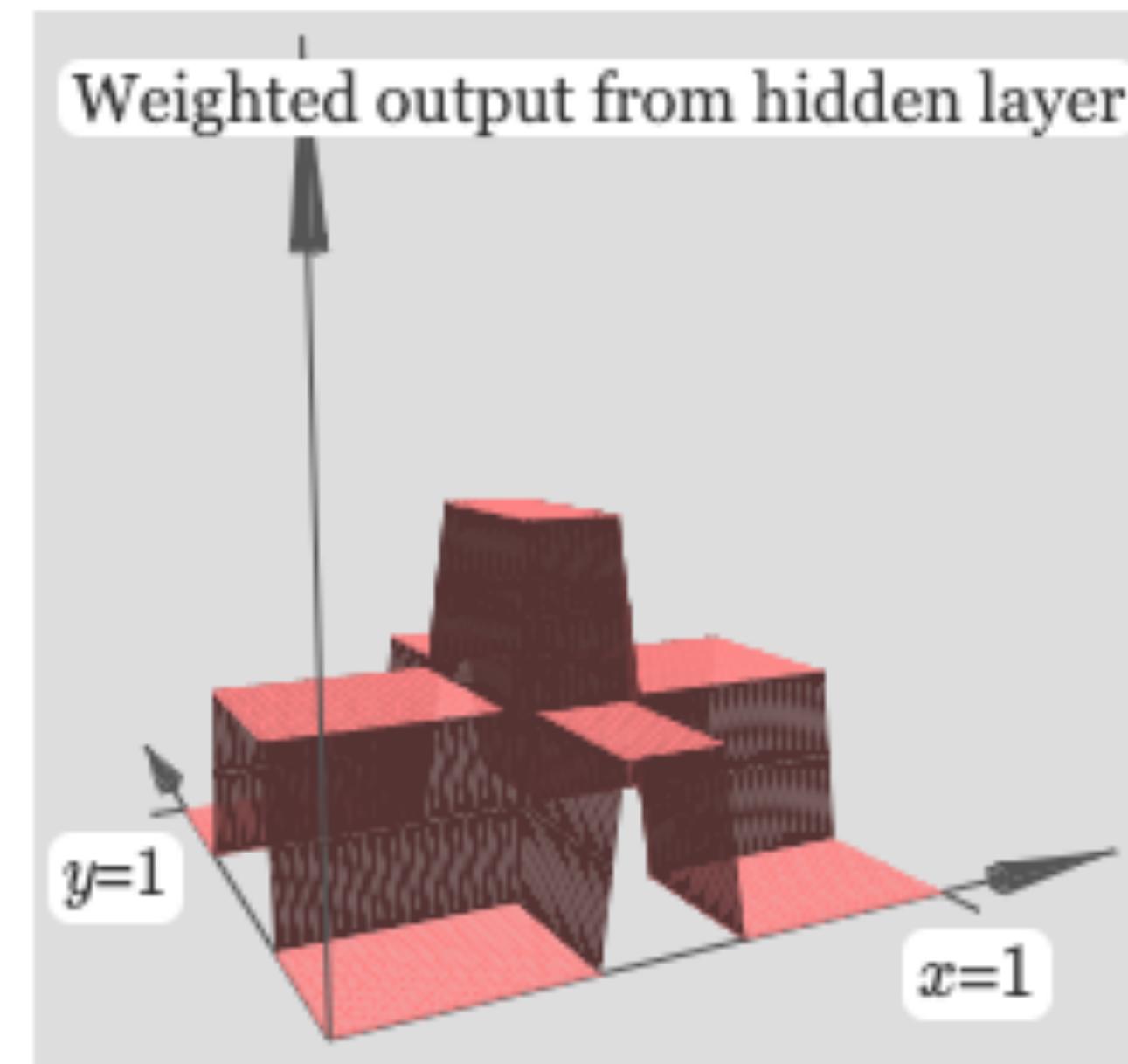
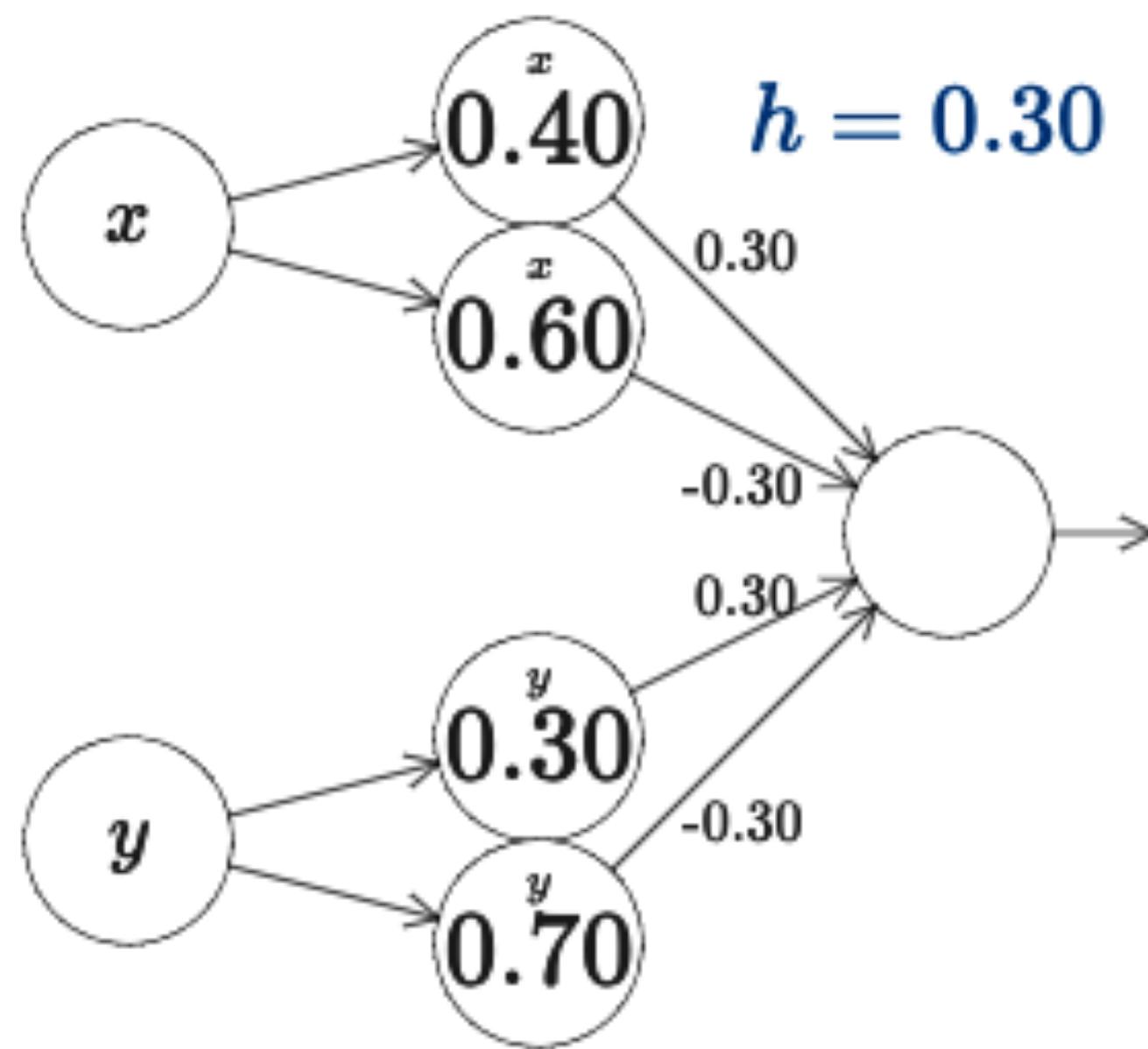


modificando i pesi possiamo approssimare la forma funzionale con rettangoli ...



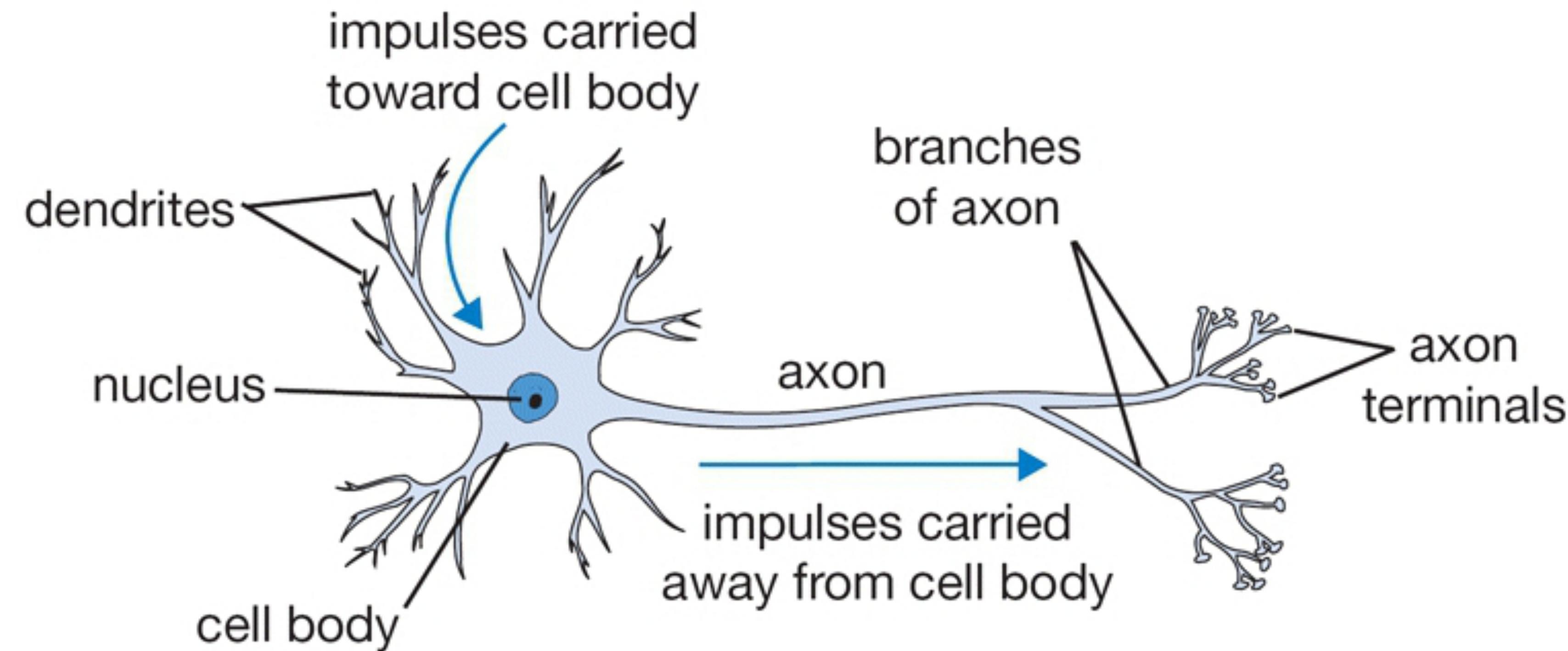
h scelti in modo da minimizzare la deviazione media nel intervallo x tra funzione vera e altezza del picco (h)

la procedura si generalizza ad un input multidimensionale



infine l'utilizzo completo di funzioni di attivazione sigmoidali al posto del loro limite come step function aiuta a mantenere basso il numero di neuroni grazie ad una approssimazione più smooth della funzione continua (in modo simile all'utilizzo di kernel gaussiani al posto di distribuzioni uniformi nella stima non parametrica di pdf)

NN ARTIFICIALI E RETI NEURALI BIOLOGICHE

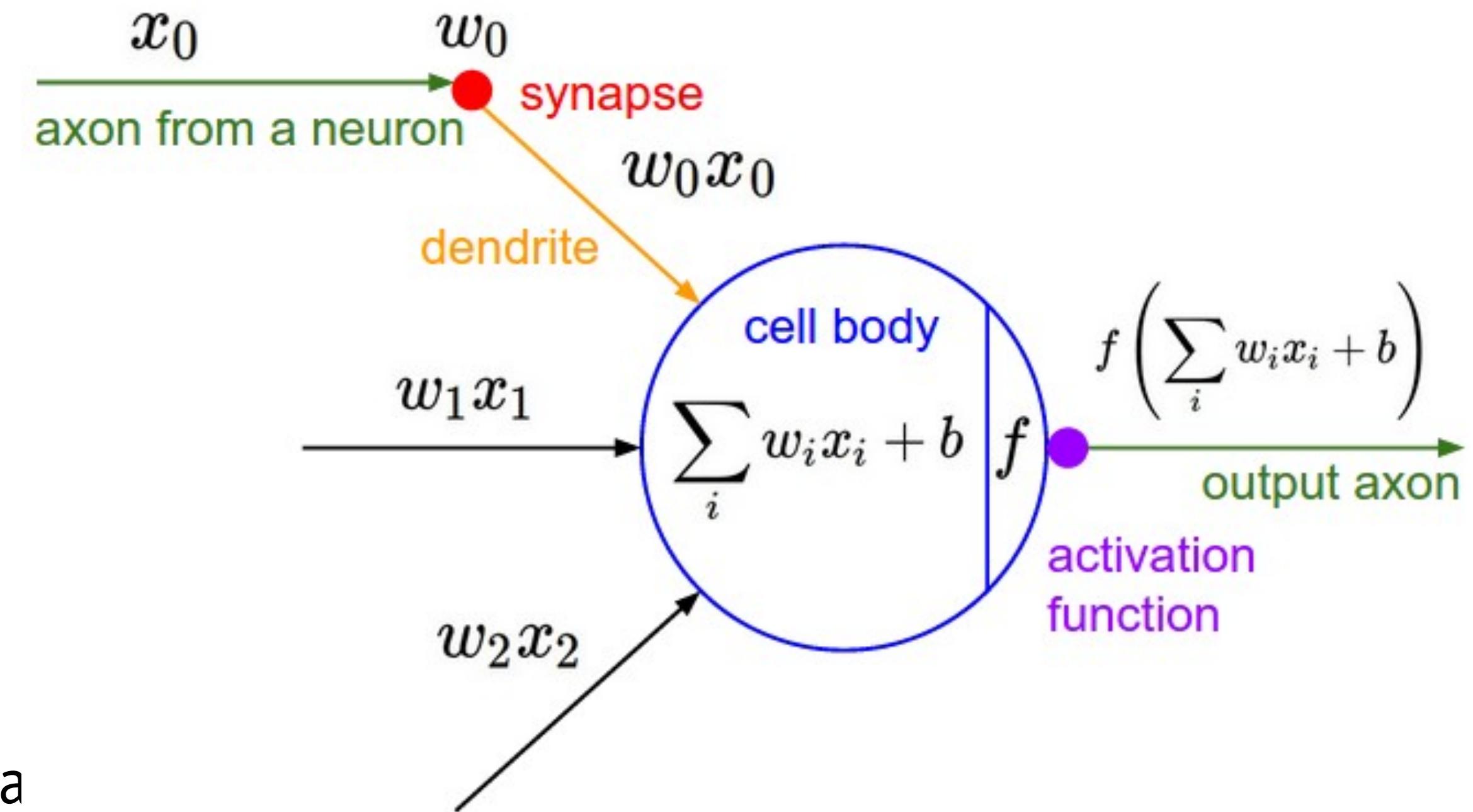
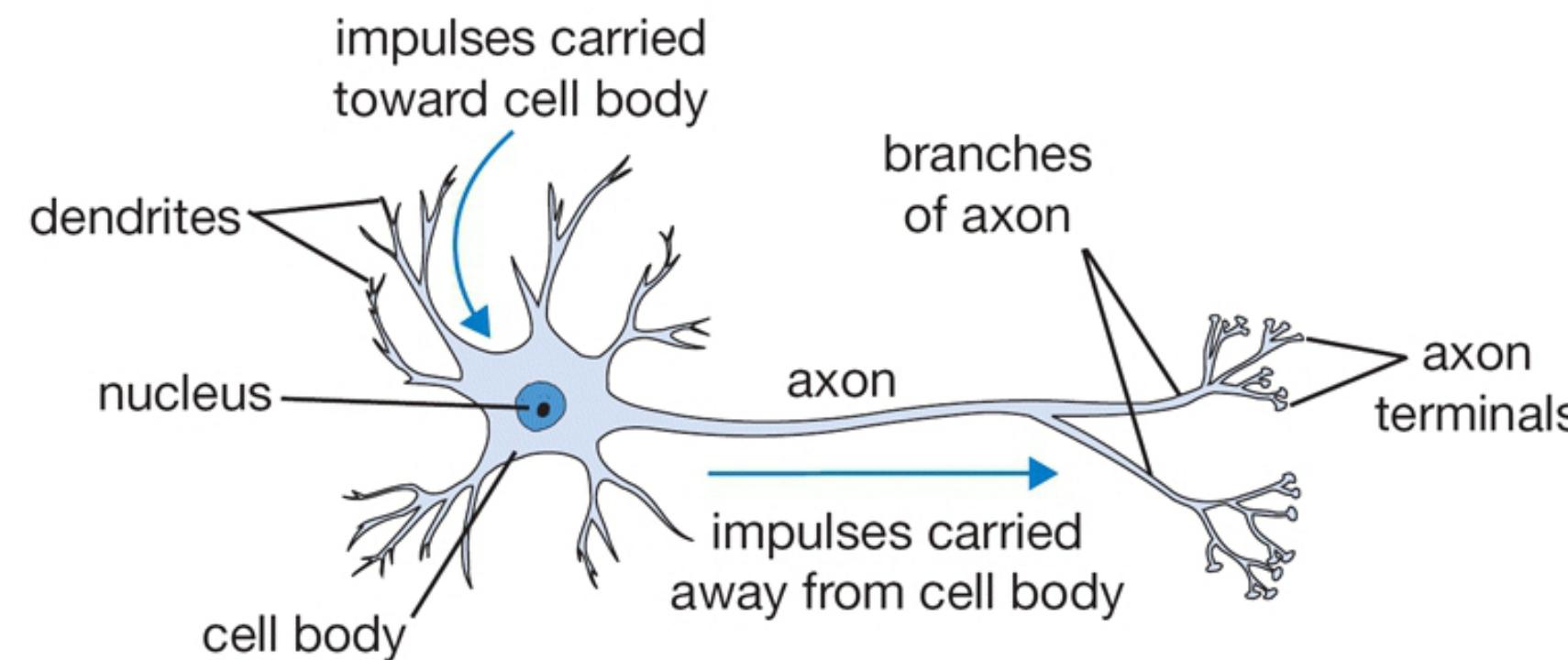


unità computazionale di base nel cervello animale: **neurone** ($\sim 10^{11}$ neuroni nel cervello umano)

- neurone composto di tre parti: i **corpo cellulare (soma)**, i **dendriti** e l'**assone**
- ogni neurone riceve segnali di input dai suoi dendriti e produce segnali di output lungo il suo (singolo) assone
- l'assone infine si dirama e si collega tramite **sinapsi** ai dendriti di altri neuroni

NN ARTIFICIALI E RETI NEURALI BIOLOGICHE

modello computazionale del neurone biologico:



- segnali che viaggiano lungo gli assoni (ad es. x_0) interagiscono in modo moltiplicativo (ad es. w_0x_0) con i dendriti dell'altro neurone in base alla cosiddetta strength sinaptica in quella sinapsi (ad es. w_0)
- le strength sinaptiche (i pesi w) sono apprendibili e controllano come un neurone influenza (sia in intensità che in direzione (eccitatoria (peso positivo) che inibitoria (peso negativo)) gli altri neuroni
- le reti neurali biologiche sono reti di tipo **spiking** in cui i dendriti portano il segnale al corpo cellulare dove vengono sommati e solo se la somma supera una certa soglia il neurone può “sparare” inviando un picco lungo il suo assone

NN ARTIFICIALI E RETI NEURALI BIOLOGICHE

- l'analogia con le reti neurali biologiche è suggestiva, ma bisogna essere molto cauti ...
 - esistono molte differenti tipologie di neuroni
 - i dendriti possono eseguire calcoli non-lineari molto complessi
 - le sinapsi nella realtà non sono singoli pesi ma sistemi dinamici complessi e non lineari
 - il timing delle spikes può avere un peso nel comportamento del sistema (nel modello semplificato si assume che il timing non abbia importanza e solo la firing rate contenga l'informazione)

Letture consigliate per chi fosse interessato all'argomento:

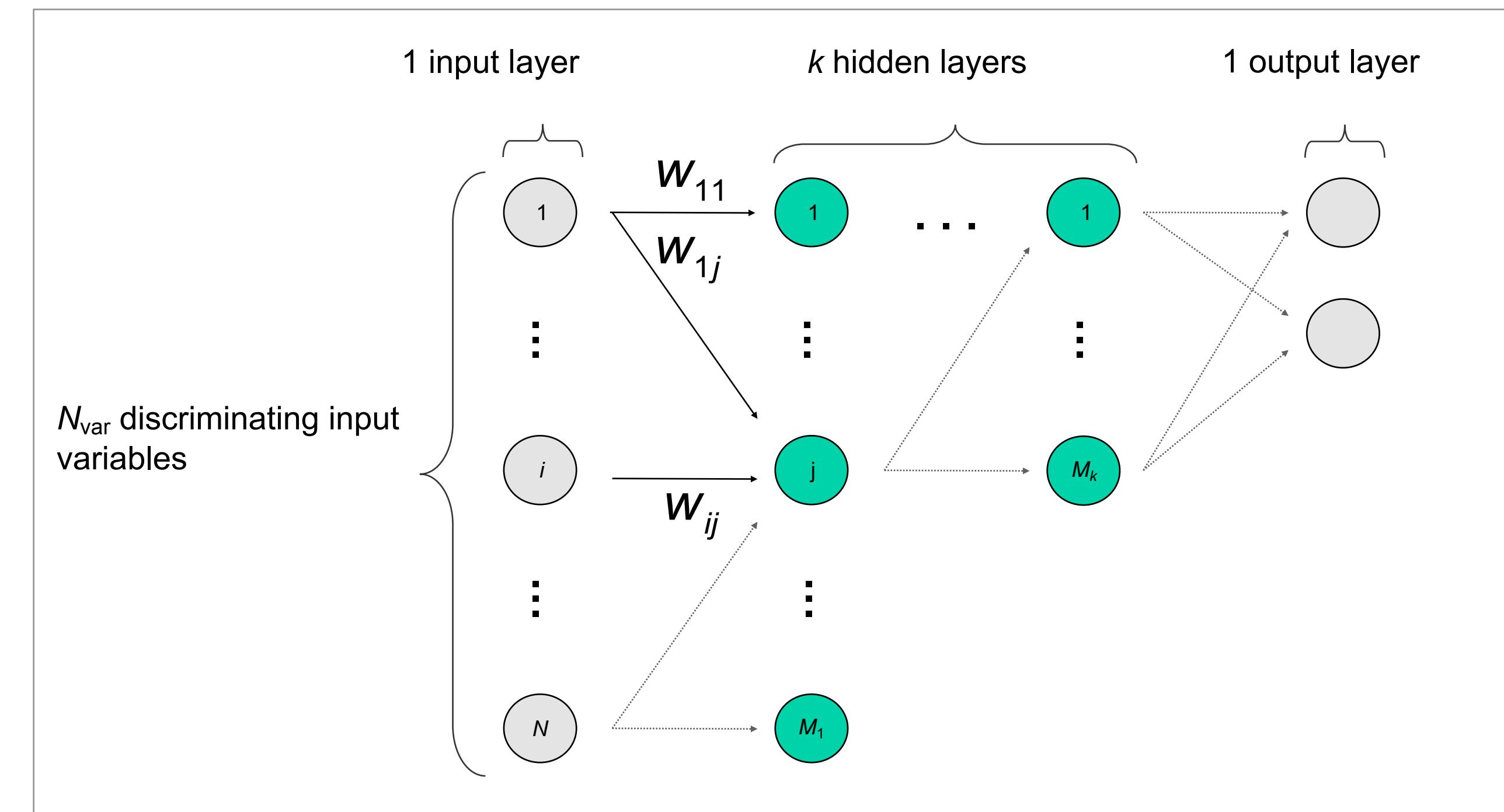
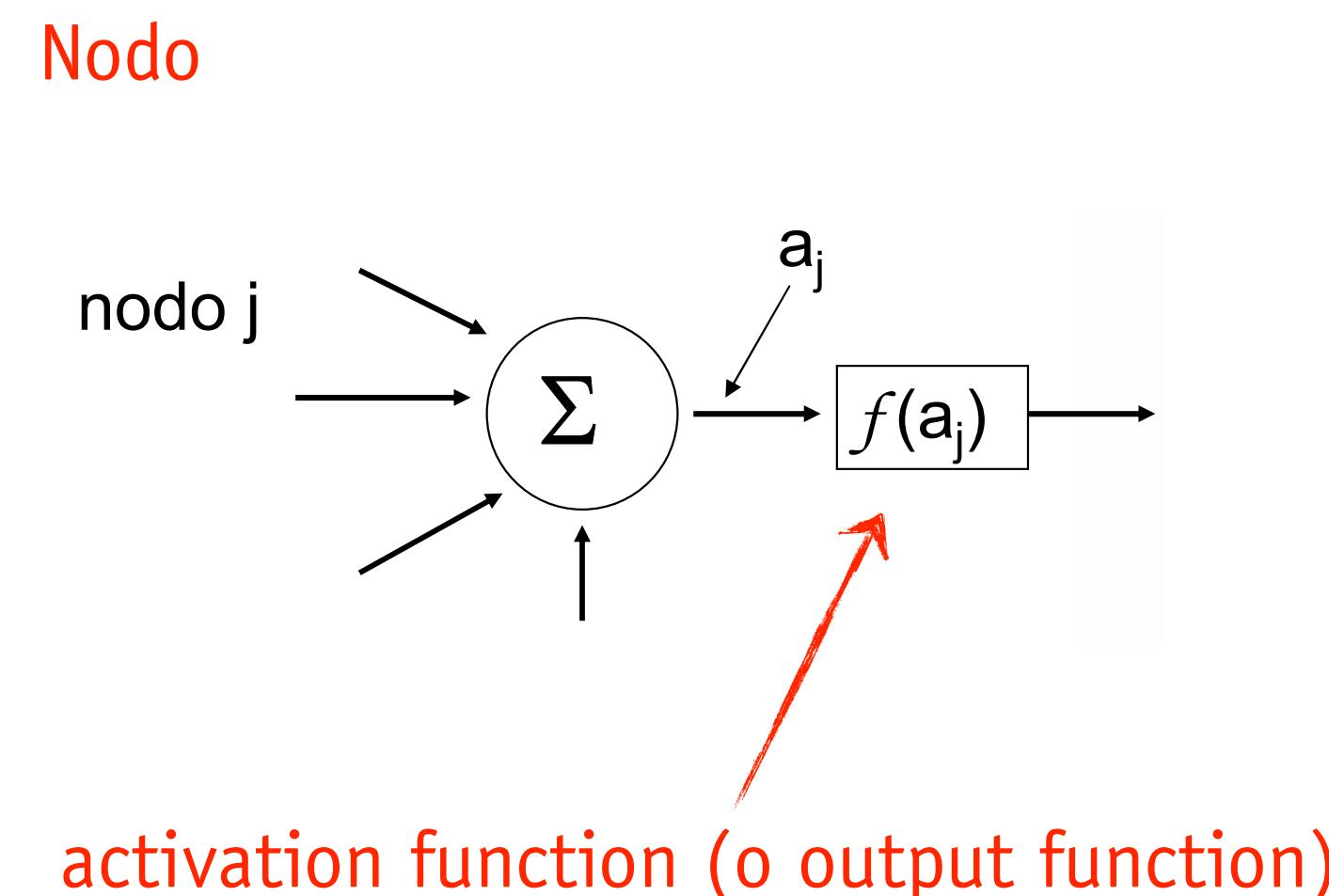
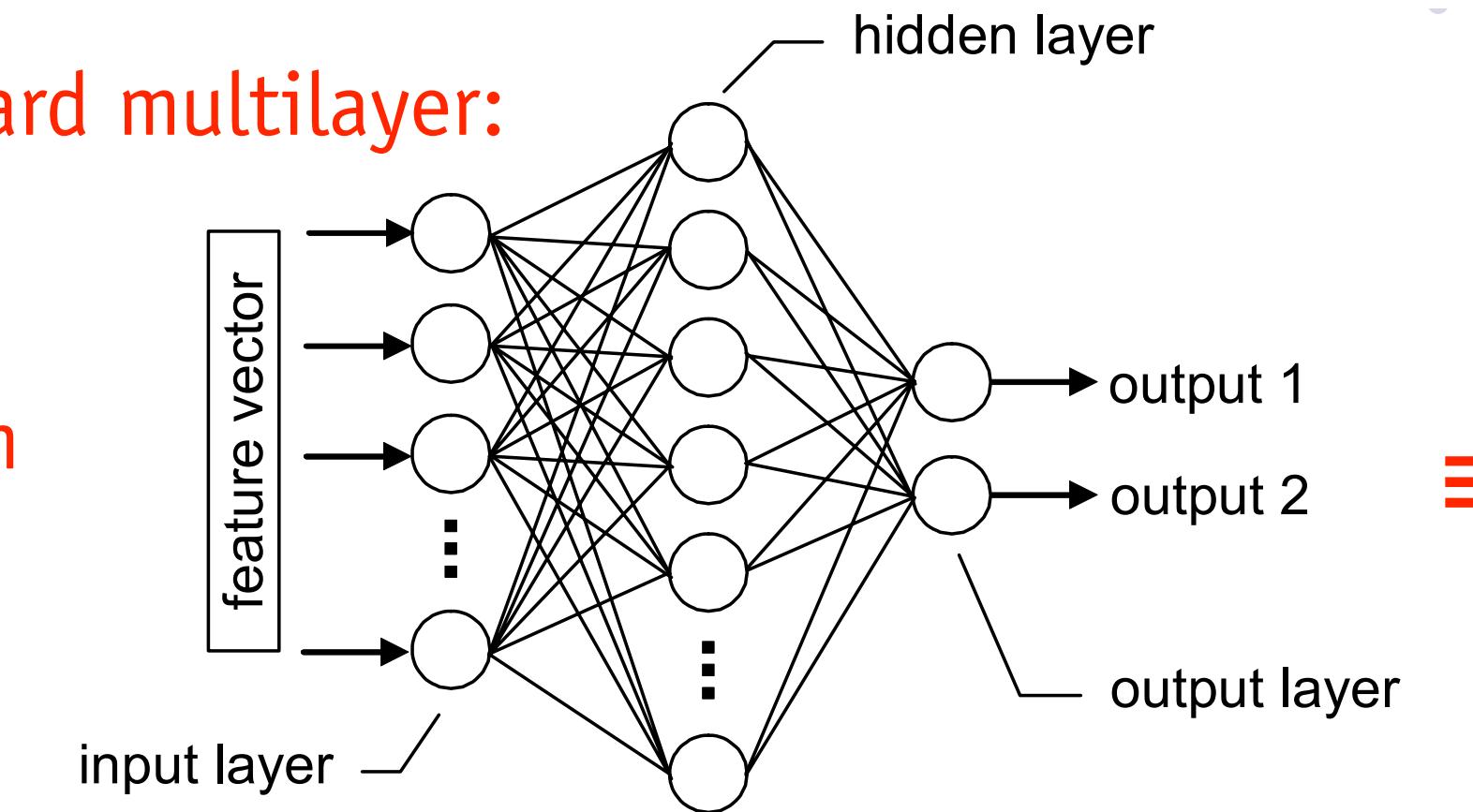
- London & Häusser: [Dendritic Computation](#)
- Brunel et al.: [Single neuron dynamics and computation](#)



FEED-FORWARD ANN

- la topologia di rete neurale ANN più utilizzata è una struttura Feed-Forward multilayer:

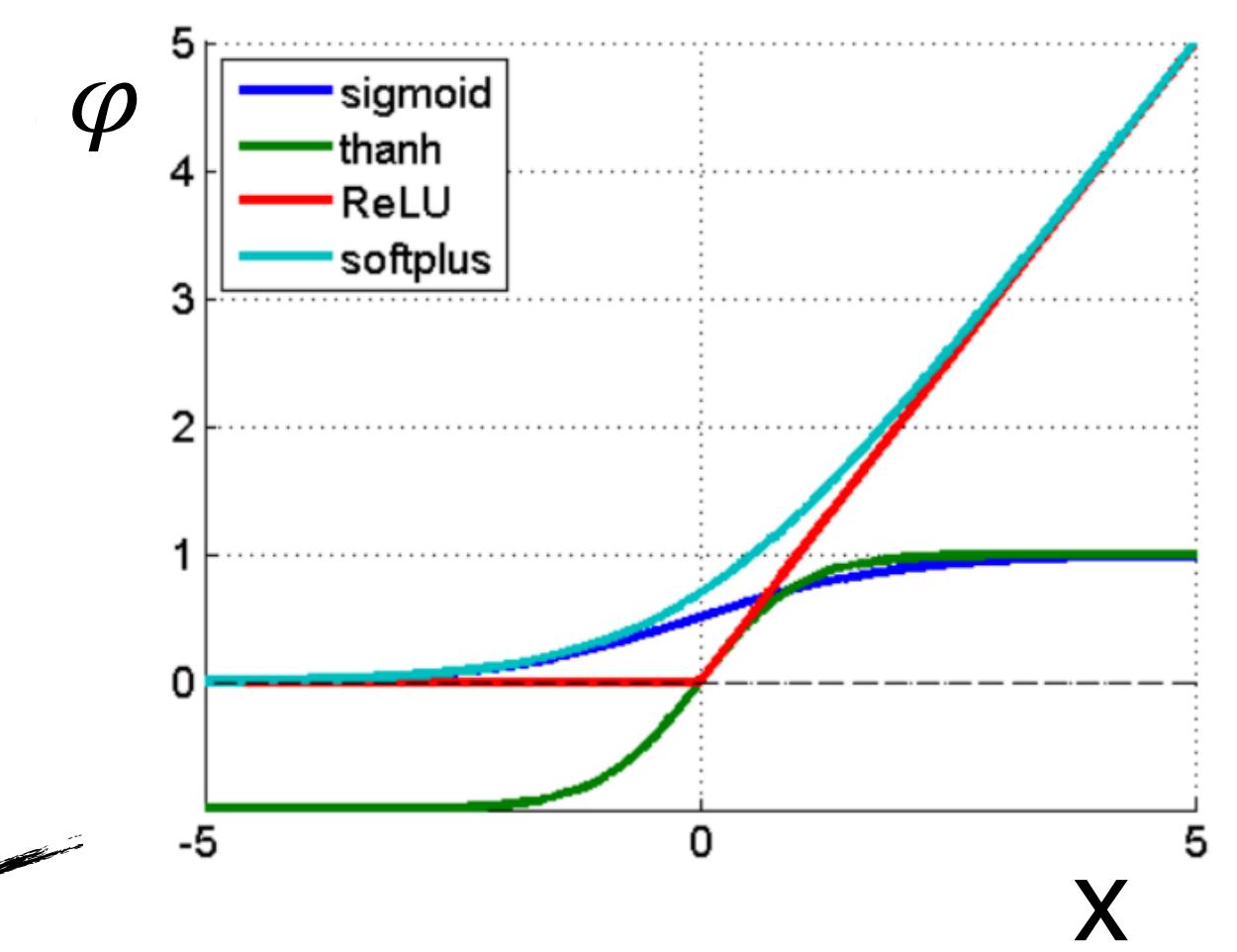
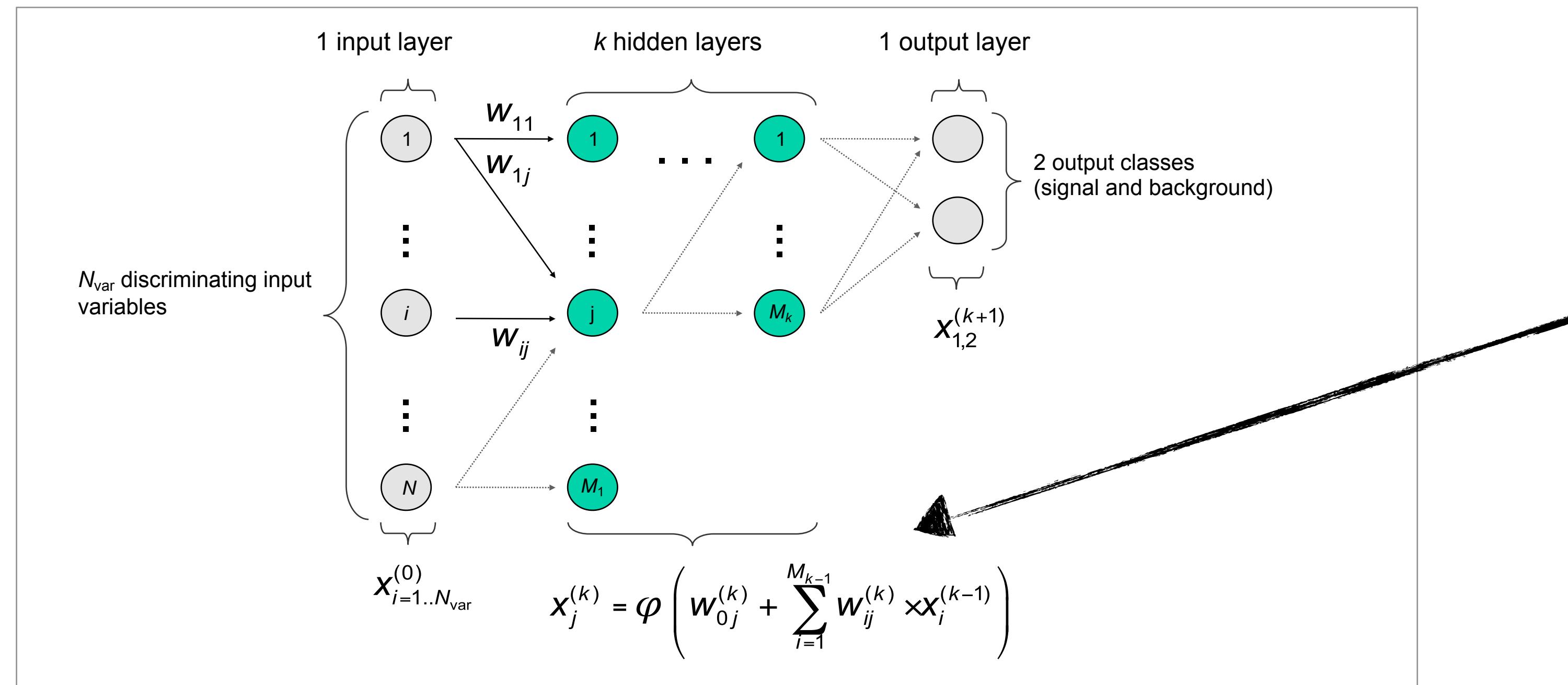
- neuroni organizzati in layers: input, hidden-1, ..., hidden-K, output
- possibili solo connessioni da un dato layer al successivo: direct acyclic graph



Feed-forward Multilayer Perceptron

FUNZIONE DI RISPOSTA

- comportamento della rete determinato da:
 - struttura topologica dei neuroni (architettura della rete)
 - pesi (weight) associati a ciascuna connessione
 - funzione di risposta di ciascun neurone ai dati di input
- funzione di risposta ρ :
 - mappa l'input del neurone n-esimo: $x^{(k-1)}_1, \dots, x^{(k-1)}_n$ nell'output $x^{(k)}_j$
 - divisa in **funzione sinaptica** $k: \mathbb{R}^n \rightarrow \mathbb{R}$ e in **funzione di attivazione neurale** $\varphi: \mathbb{R} \rightarrow \mathbb{R}$: $\rho = k \bullet \varphi$



$$\varphi : x \rightarrow \begin{cases} \text{linear: } x \\ \text{sigmoid: } 1/(1+e^x) \\ \text{Tanh}(x) \\ \text{ReLU: } \max(0,x) \\ \text{softplus: } \log(1+e^x) \end{cases}$$

TRAINING

- addestrare la rete consiste nell'aggiustare il valore dei pesi (e di tutti gli altri iperparametri) in accordo ad una data loss function che ottimizza le prestazioni del modello rispetto ad uno specifico compito
- tecnica più utilizzata: discesa lungo il gradiente con back-propagation

Output per un ANN con:

- un singolo hidden layer con φ : tanh
- un output layer con φ : lineare

n_h : numero di neuroni nel layer hidden

n_{var} : numero di neuroni nel layer di input

$$y_{ANN} = \sum_{j=1}^{n_h} x_j^{(2)} w_{j1}^{(2)} = \sum_{j=1}^{n_h} \tanh \left(\sum_{i=1}^{n_{var}} x_i w_{ij}^{(1)} \right) w_{j1}^{(2)}$$

peso associato alla connessione tra il j-esimo neurone del hidden layer e il neurone di output

peso associato alla connessione tra l'i-esimo neurone del input layer e il j-esimo neurone del hidden layer

NOTA: il modello matematicamente corrisponde ad una composizione di funzioni: $f(x) = f^{(2)}(f^{(1)}(x))$ con $f^{(i)}$ i-esimo layer ed ultimo layer detto output layer ... per un deep-NN con d-layer nascosti: $f(x) = f^{(d+1)}(\dots f^{(4)}(f^{(3)}(f^{(2)}(f^{(1)}(x)))))$



TRAINING

- durante il training vengono presentati alla rete N esempi: $T\{x_a\}$ ($a=1,\dots,N$)
- per ogni evento viene calcolato l'output del modello $y_{ANN}(x_a)$ e confrontato con il target atteso
 - esempio task classificazione a binaria: $Y_a \in \{0,1\}$
 - esempio regressione $Y_a = y \in \mathbb{R}$
- viene definita una opportuna funzione di perdita che misuri la “distanza” tra $y_{ANN}(a)$ e Y_a :

$$\Delta(x_1, \dots, x_N | \mathbf{w}) = \sum_{a=1}^N \Delta_a(x_a | \mathbf{w}) = \sum_{a=1}^N \frac{1}{2} (y_{ANN}(a) - Y_a)^2 \quad \text{Esempio MSE}$$

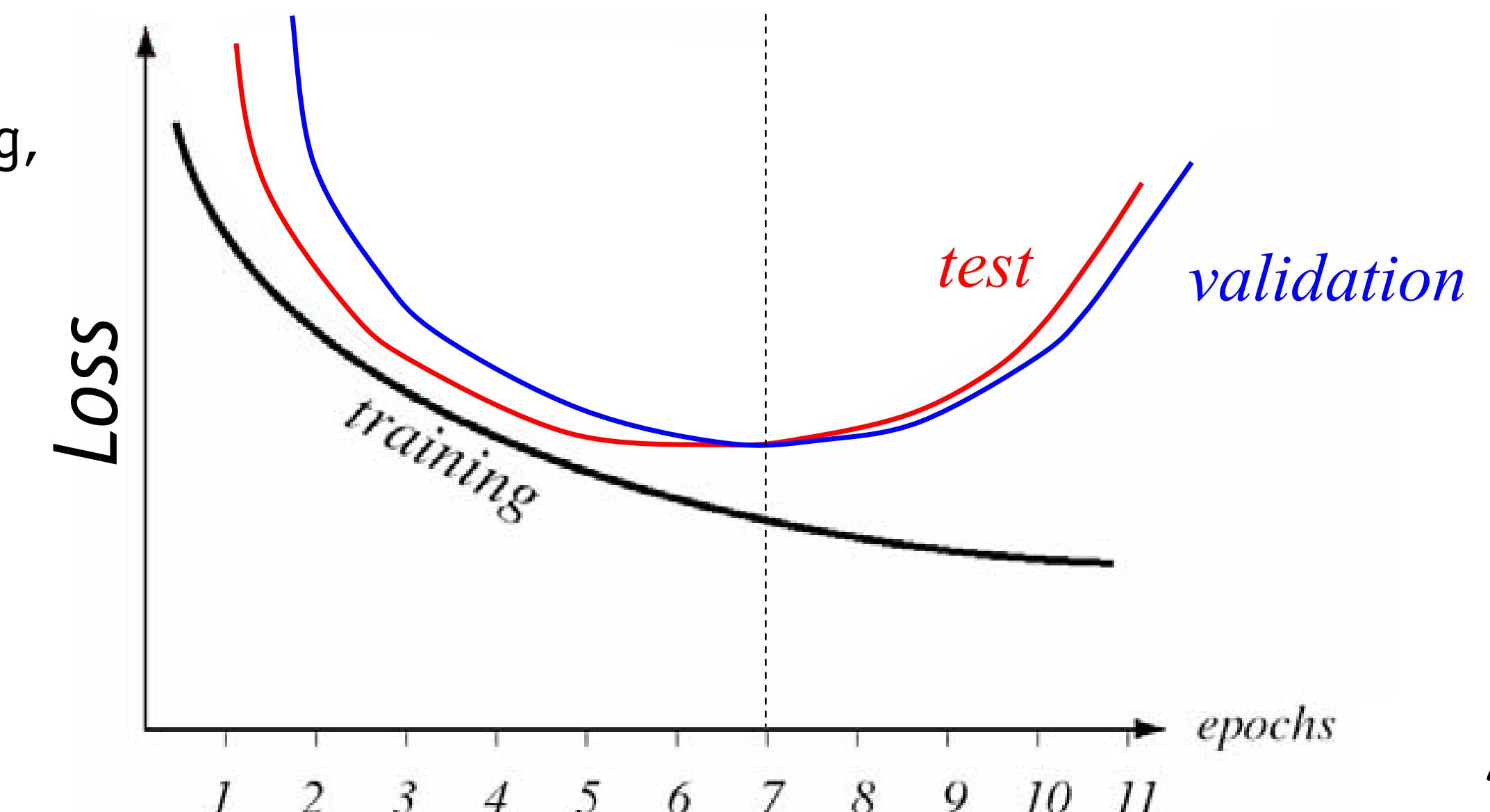
- il vettore di pesi viene scelto come quello che minimizza l'errore Δ
- il minimo può essere trovato con tecniche di GD/SGD ...
$$\mathbf{w}^{(\rho+1)} = \mathbf{w}^{(\rho)} - \eta \nabla_{\mathbf{w}} \Delta$$

PROBLEMA: per aggiornare i pesi di tutti i layer della rete è necessario calcolare il gradiente di funzioni complicate per ogni neurone della rete e di valutarne il valore numerico. Farlo in modo semplice ed efficiente prende il nome di procedura di **Back-Propagation o Backprop** (nella prossima lezione ...)



CURVE DI APPRENDIMENTO

- se si grafica il valore della loss function sul training set questo risulta tipicamente grande all'inizio dell'ottimizzazione quando i pesi della rete sono stati inizializzati in modo casuale (a piccoli valori random)
- con il passare del numero di iterazioni (epoch) l'errore decresce fino a raggiungere (tipicamente) un valore di plateau che dipende da:
 - la dimensione del training set
 - l'architettura della rete (i.e. il numero di neuroni (pesi) in un shallow FFNN)
 - il valore iniziale dei pesi
- il progresso nel training viene visualizzato tramite curve di apprendimento (learning curve) che riportano il valore della loss (o anche l'accuracy o qualsiasi indice di prestazioni in funzione dell'epoca)
- come per gli altri algoritmi di ML, per addestrare la rete sono necessari più dataset (e/o l'uso di k-fold cross validation): per il training, per ottimizzare gli iperparametri, decidere il criteri di stop, e infine per valutare le prestazioni finali del modello ...

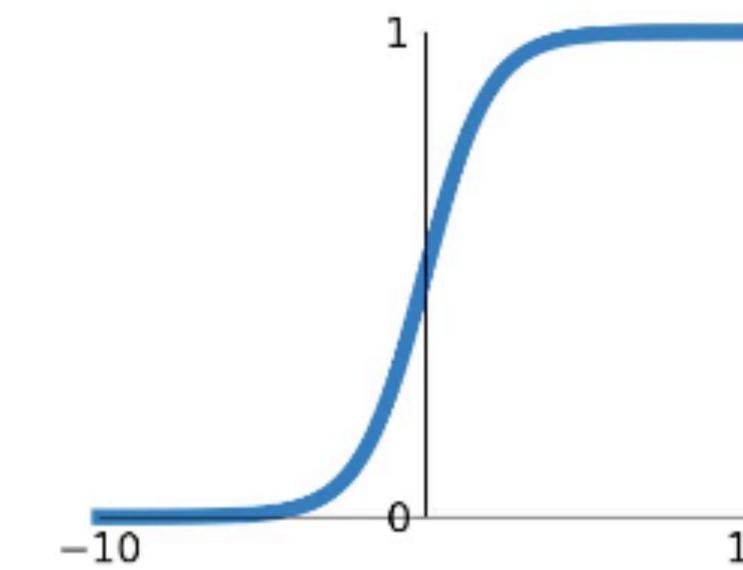


ACTIVATION FUNCTIONS

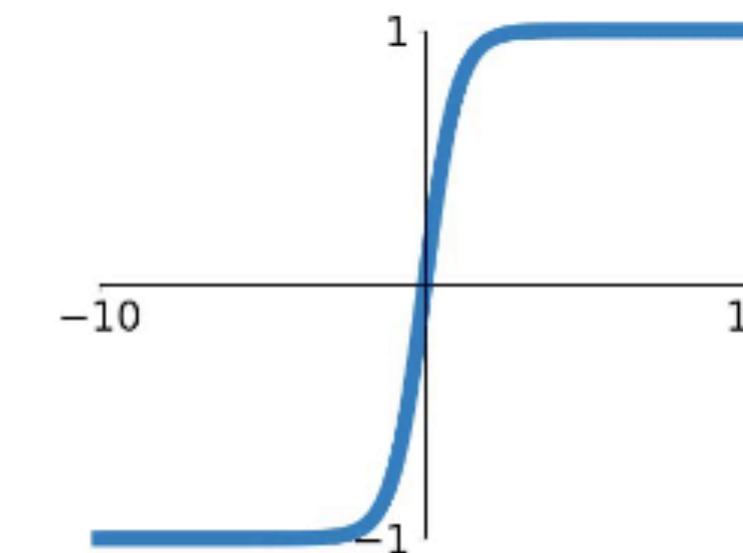
In generale qualsiasi funzione che rispetti il teorema di approssimazione universale va bene, nella realtà dei fatti esistono moltissime proposte di funzioni di attivazione spesso giustificate solo su base euristica ...

Sigmoid

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



tanh

$$\tanh(x)$$


preferibilmente da non usare:

- ha gradiente che si annulla lontano da $x=0 \rightarrow$ diluizione del gradiente (vanishing g \rightarrow gradient problem)
- l'output non è centrato in zero \rightarrow effetti sulla dinamica durante la discesa lungo il gradiente (ziga-zag, instabile)

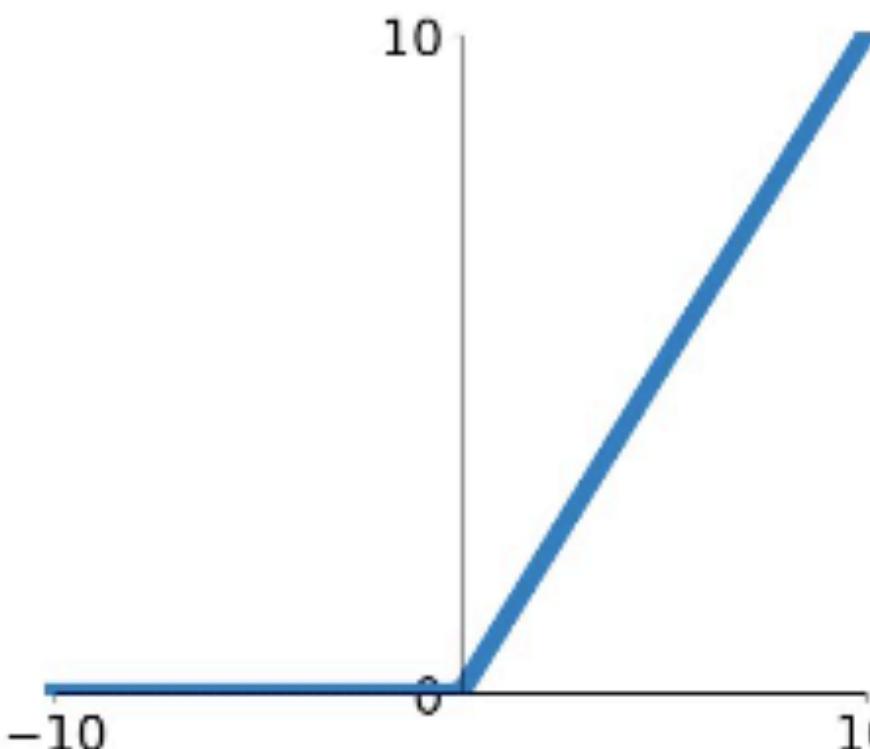
meglio:

- l'output è centrato in zero
- è esprimibile come sigmoide: $2\sigma(2x)-1$

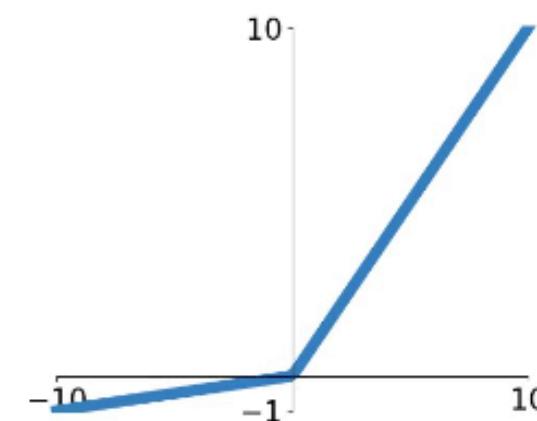
superate, specie nel DL, da ReLU e variazioni ...



ReLU

$$\max(0, x)$$


Leaky ReLU

$$\max(0.1x, x)$$


Maxout

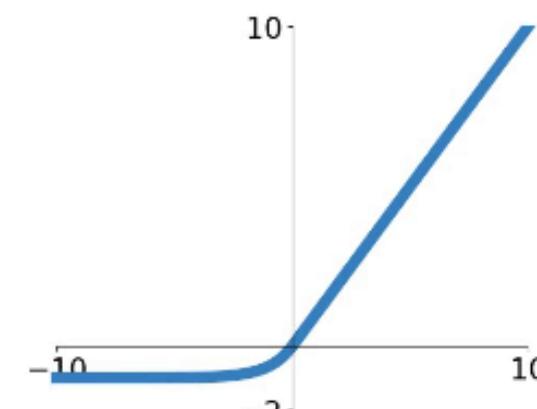
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

- la più usata:
- dinamica non lineare
 - accelera notevolmente la convergenza del NN durante il training poiché non satura
 - non soffre del problema del vanishing gradient
 - problemi: crea sparsificazione del gradiente (pesi nulli per un dato neurone) e morire durante il training, va monitorata e in caso di problemi (una grande parte della rete morta) usare funzioni diverse ...



variazioni da testare durante la fase di tuning degli iperparametri della rete

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$


ex. maxout ha tutti pregi di LeakyReLU/ELU e le generalizza ... a spese di due volte il numero dei parametri ...

LOSS FUNCTIONS

I moderni ANN sono addestrati usando il principio di massima verosimiglianza, di conseguenza la loss function più utilizzata è semplicemente l'equivalente della negative log-likelihood:

$$L(\mathbf{w}) = -E_T[\log p_{model}(y | x; \mathbf{w})]$$

forme più utilizzate:

MSE

binary cross-entropy

categorical cross-entropy

$$MSE = \frac{1}{n} \sum_{i=1}^n (\Upsilon - Y)^2$$

$$-t \log(p) - (1 - t) \log(1 - p)$$

$$L_i = -\sum_j t_{i,j} \log(p_{i,j})$$

$$p_{model}(y | x; \mathbf{w}) = N(y, x, \mathbf{w})$$

per problemi di regressione

p = predizione del modello

t = target

generalizzazione del caso precedente
per problemi multclasse

NOTA: cross-entropy:

date due distr. p e t, misura del numero medio di bit necessari per identificare un evento estratto dall'insieme, nel caso sia utilizzato uno schema ottimizzato per la distribuzione di probabilità p, piuttosto che per la distribuzione "vera" t

è di norma la loss function migliore per NN che presentano probabilità in output (esempio: softmax)



FFNN (SHALLOW) IMPLEMENTATION IN TENSORFLOW/KERAS

A sequential model with 1 dense layer used to classify handwritten numbers (0 to 9) from the standard MINIST dataset

https://www.dropbox.com/s/5tcjrgcq3yhe88/FFNN_shallow.ipynb?dl=0

```
[1] #import models & layers from TF/keras
import tensorflow as tf
from tensorflow import keras

# Helper libraries
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
print(tf.__version__)

2.2.0-rc2
```

richiede TF2.X,
testato su colab

```
#retrieve training data (MNIST dataset)
#60k images for training and 10k images for testing
mnist = keras.datasets.mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

#verify:
print ('images shape: ', train_images.shape)
print ('labels shape', train_labels.shape)

class_names = ['zero', 'uno', 'due', 'tre', 'quattro', 'cinque', 'sei', 'sette', 'otto', 'nove']

images shape: (60000, 28, 28)
labels shape (60000,)
```

MNIST dataset

NOTE: keras fornisce quattro arrays numpy:

- `train_images` e `train_labels`: training set e target (label) per il modello (vanno suddivisi in training e validation)
- `test_images` e `test_labels`: per il test delle prestazioni del modello (non devono mai essere usati per training o tuning degli iperparametri)

Le immagini sono array numpy formato 28x28 con pixel in scala di grigio con valori nel range [0,256]. Le label sono vettori di interi nel range [0,...,9] che corrispondono al numero rappresentato in ogni immagine.



```
[8] #Data exploration

print(train_images.shape)
print(len(train_labels))
print(train_labels)

print(test_images.shape)
print(len(test_labels))
print(test_labels)

plt.figure()
plt.imshow(train_images[33])
# plt.imshow(train_images[33], cmap=plt.cm.binary)
plt.colorbar()
plt.grid(False)
plt.show()
```

↳ (60000, 28, 28)

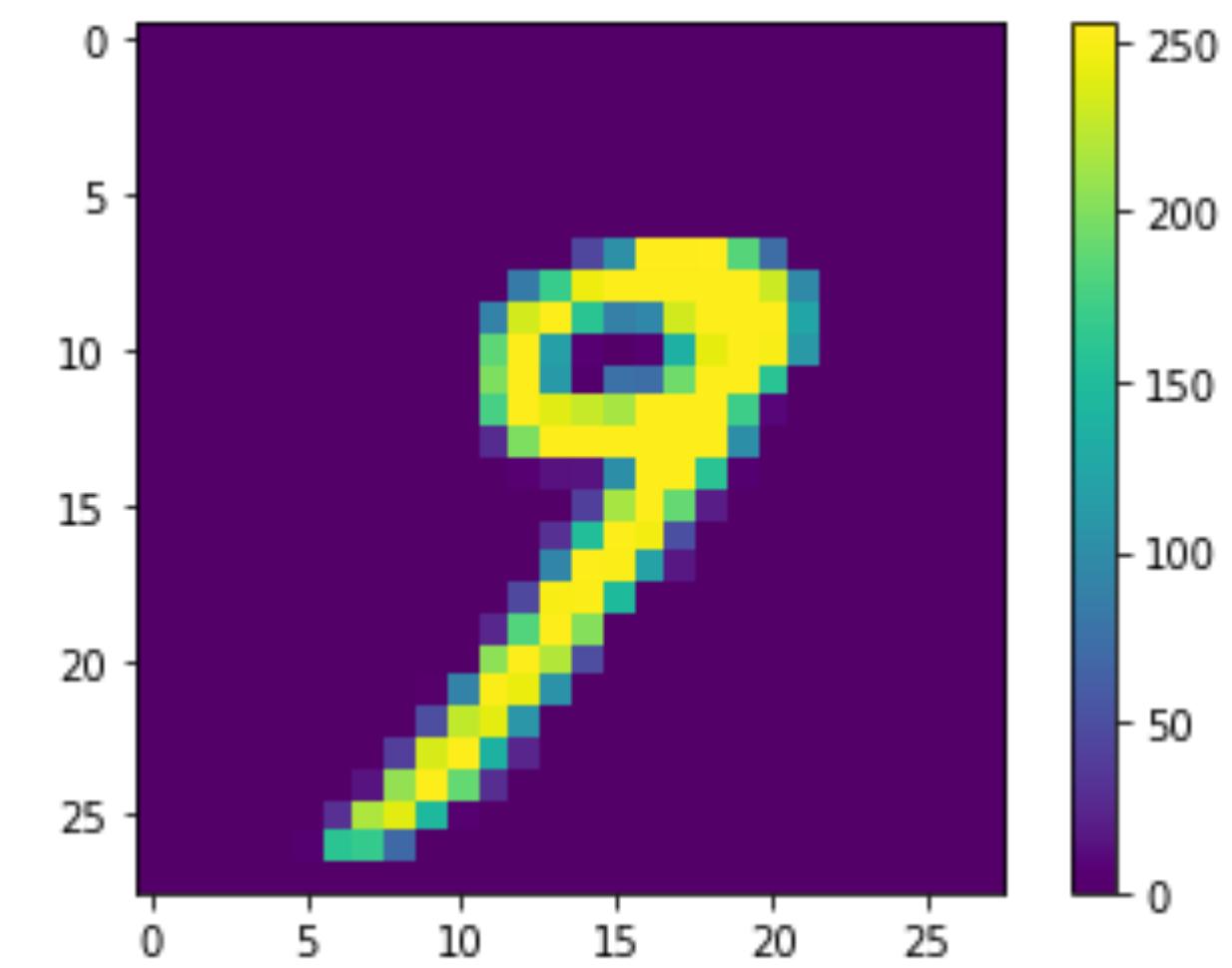
60000

[5 0 4 ... 5 6 8]

(10000, 28, 28)

10000

[7 2 1 ... 4 5 6]

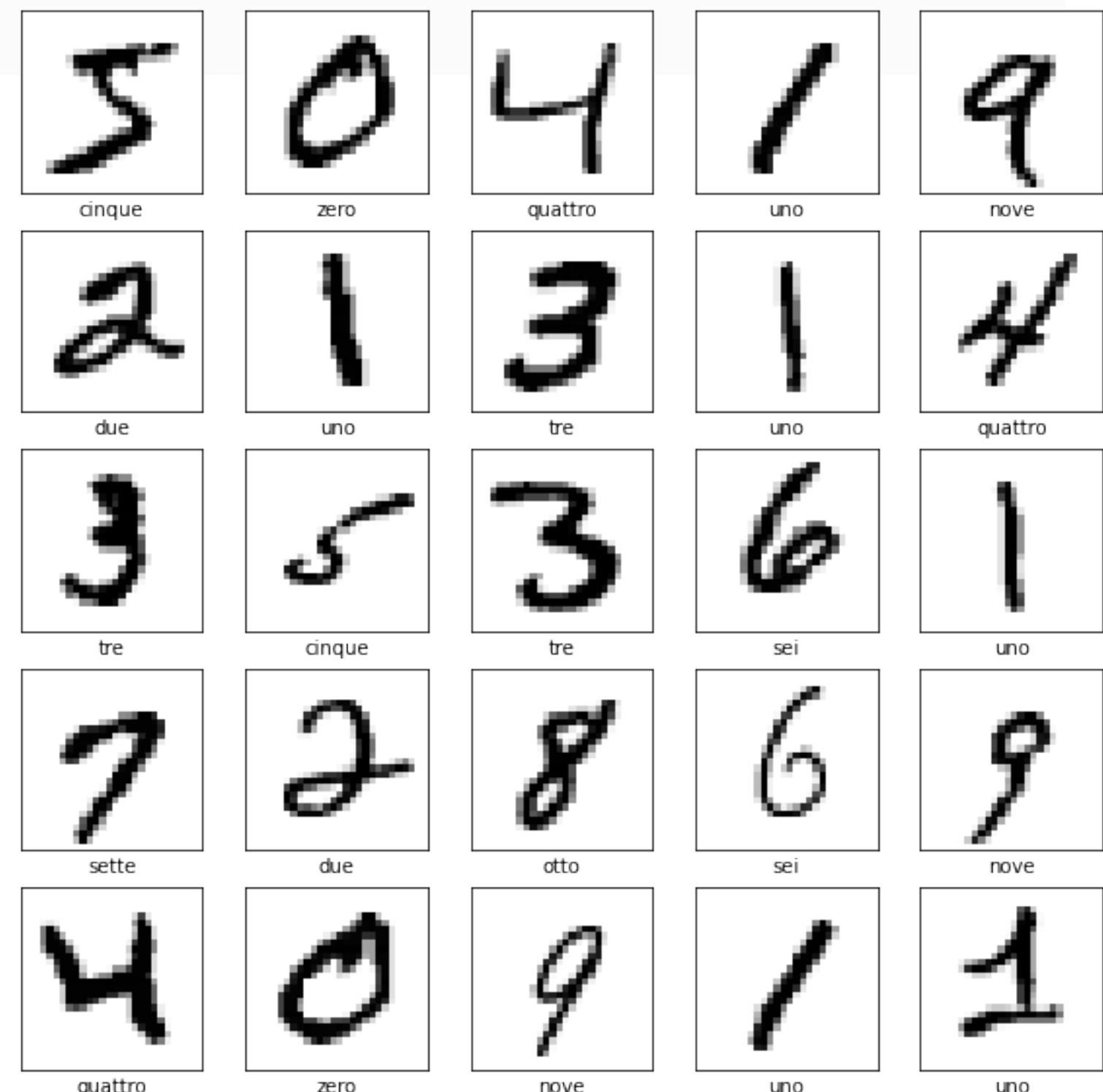


```
[9] print('label: ', train_labels[33])
```

↳ label: 9

```
[11] # plot 25 images to check everything is fine ...
```

```
plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i], cmap=plt.cm.binary)
    plt.xlabel(class_names[train_labels[i]])
plt.show()
```



Preprocessing:

```
[10] #training data preprocessing  
# normalize images intensities (pixel values) to [0,1] (be sure intensities are floating-points first)  
  
train_images = train_images.astype('float32') / 255.0  
test_images = test_images.astype('float32') / 255.0
```



```
[12] # encode the labels in category using one_hot encoding (builtin in keras)
```

```
train_labels = keras.utils.to_categorical(train_labels)  
test_labels = keras.utils.to_categorical(test_labels)
```

```
print('label: ', train_labels[33])
```

```
⇒ label: [0. 0. 0. 0. 0. 0. 0. 0. 1.]
```

rappresentazione “one-hot”
(i.e. versori assi principali)

```
[13] # convert images to flat vector of 28x28 pixels to be used in a dense layer
```

```
# NOTE you can do that here or (better approach) to use a Flatten layer in the NN model as first layer  
# in this example we use the Flatten layer ...
```

```
#train_images = train_images.reshape((60000, 28 * 28))  
#test_images = test_images.reshape((10000, 28 * 28))
```

la conversie da matrici (28,28) in vettori di feature
“flat” (28*28) si può fare come preprocessing o
direttamente nel modello di rete neurale in Keras
usando un layer apposito detto “Flatten”

Definizione del modello: ANN di tipo Feed-Forward Shallow (1 solo layer nascosto)

```
[16] # Model definition (aka define the architecture of the network)
```

```
#sequential (aka Feed-Forward Neural Network)
```

```
model = keras.Sequential([
```

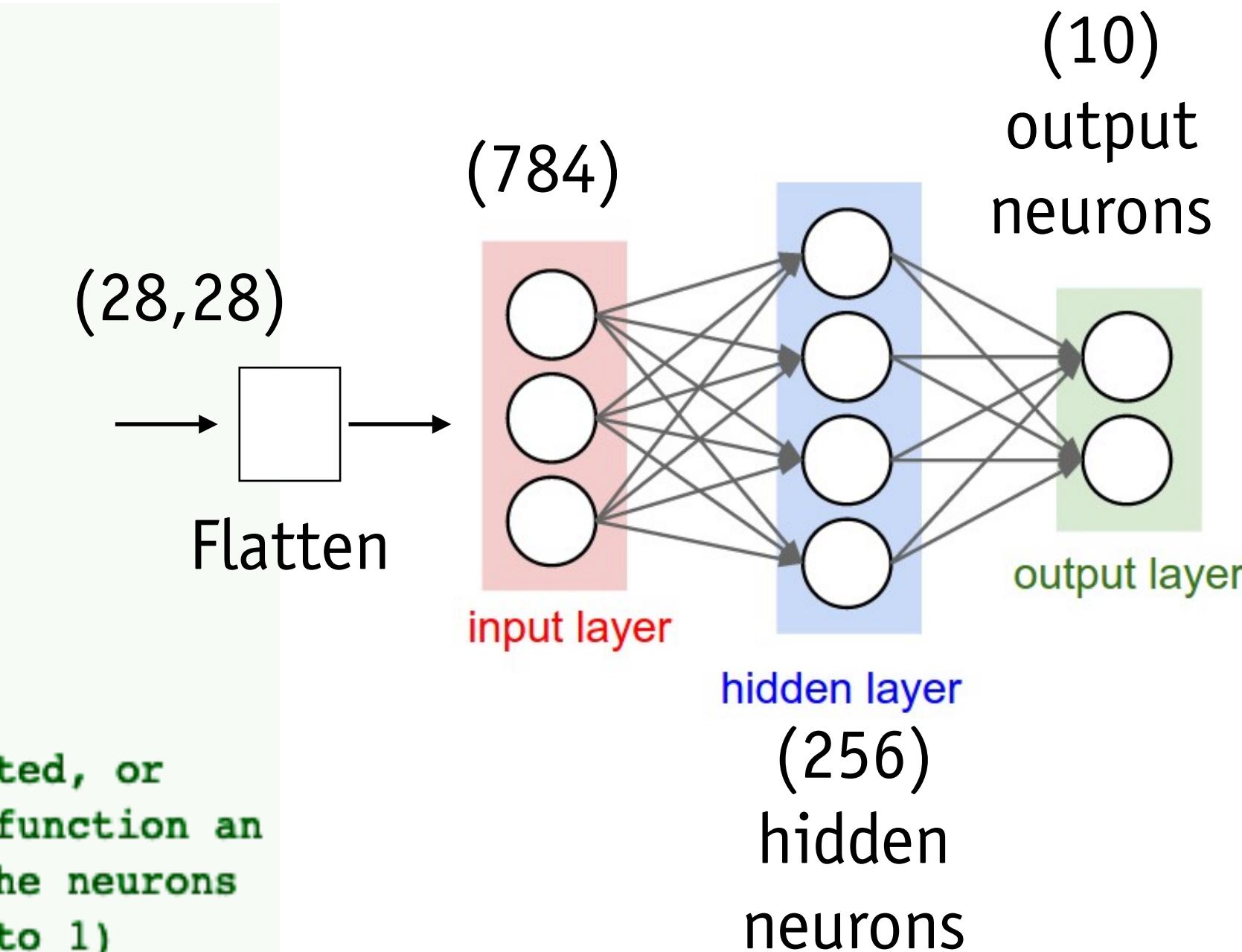
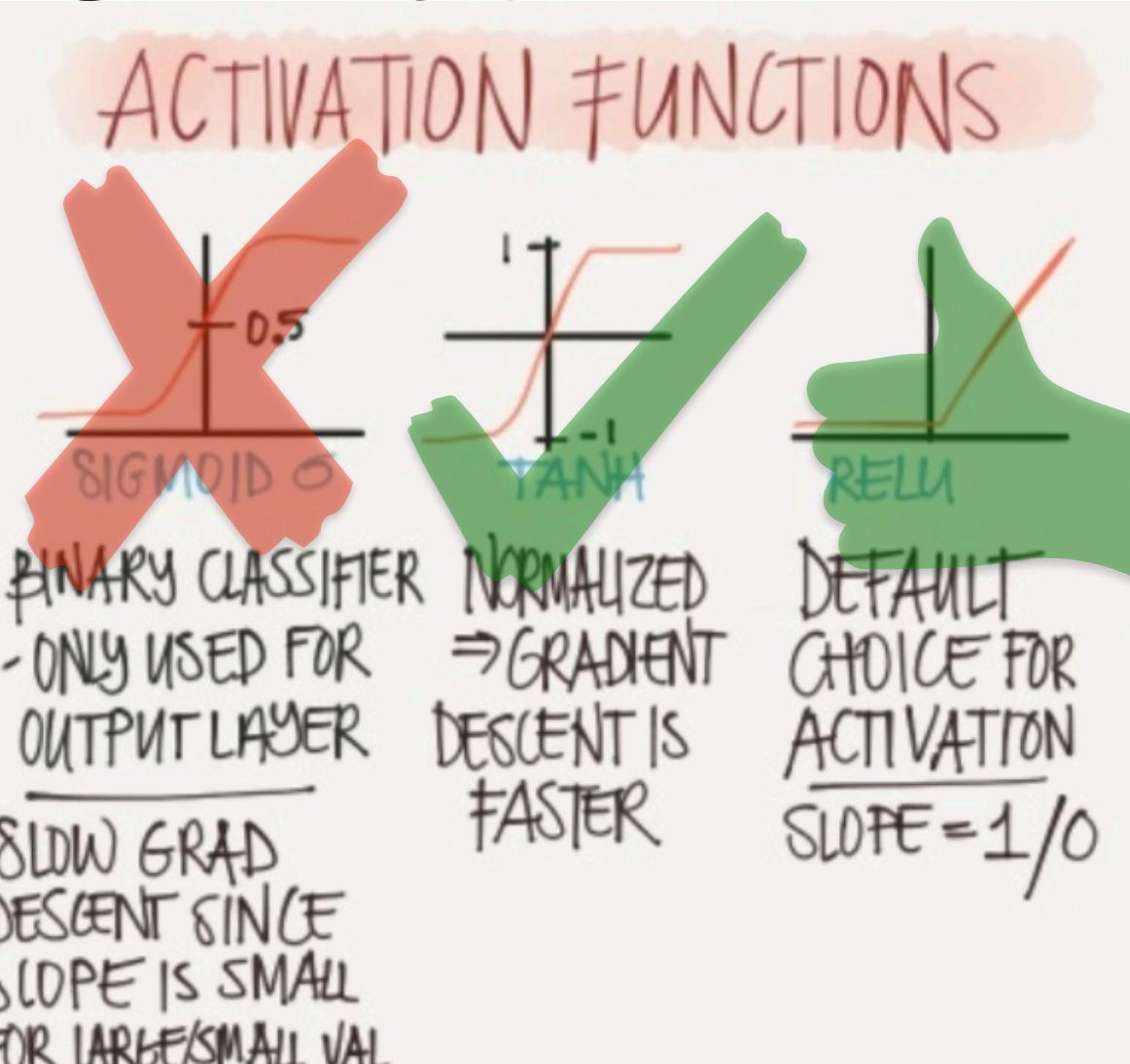
```
#first layer in this network, tf.keras.layers.Flatten, transforms the format of the images from a  
#two-dimensional array (of 28 by 28 pixels) to a one-dimensional array (of 28 * 28 = 784 pixels).  
#This prepare the flat input feature vector containing all the pixels of each image for the NN  
#This layer has no parameters to learn; it only reformats the data.
```

```
    keras.layers.Flatten(input_shape=(28, 28)),
```

```
#the network consists of a sequence of two tf.keras.layers.Dense layers. These are densely connected, or  
#fully connected, neural layers. The first Dense layer has 256 nodes (or neurons) and activation function an  
#hyperbolic tangent, the second layer has 10 neurons with activation softmax: means each one of the neurons  
#output a probability [0,1] with the constraint sum(outputs) = 1 (i.e. the output neurons add up to 1)
```

```
    keras.layers.Dense(256, activation='tanh'),  
    keras.layers.Dense(10, activation='softmax')
```

```
])
```



Nota: uso alternativo di TF/Keras
(Functional definition)

```
inputs = keras.Input(shape=(28,28,))
```

```
x = keras.layers.Flatten()(inputs)
```

```
x = keras.layers.Dense(256, activation='tanh')(x)
```

```
outputs = keras.layers.Dense(10, activation='softmax')(x)
```

```
model = keras.Model(inputs=inputs, outputs=outputs, name='my_mnist_model')
```

```
[17] #Compile the model
```

```
# Training parameters:  
# we need to define:  
# * Loss function: this measures how accurate the model is during training  
# * Optimizer: this is how the model is updated based on the data it sees and its loss function  
# * Metrics: used to monitor the training and testing steps  
  
#sgd: stochastic gradient descent  
#categorical_crossentropy: loss that measure the the distance between two probability distributions  
# (the probability output from the network and the true distribution of the labels)  
model.compile(optimizer='sgd',  
              loss='categorical_crossentropy',  
              metrics=['accuracy'])
```

```
[51] #print summary of the model  
model.summary()
```

↳ Model: "my_mnist_model"

Layer (type)	Output Shape	Param #
=====		
input_4 (InputLayer)	[(None, 28, 28)]	0
flatten_4 (Flatten)	(None, 784)	0
dense_13 (Dense)	(None, 256)	200960
dense_14 (Dense)	(None, 10)	2570
=====		
Total params: 203,530		
Trainable params: 203,530		
Non-trainable params: 0		



Training ...

```
[52] # Train phase  
# define batch_size for the stochastic gradient and number of epochs  
# and the fraction of training events used for validation (validation_split)  
  
history = model.fit(train_images, train_labels, epochs=500, batch_size=128,  
validation_split=0.2, shuffle=True, verbose=1)
```

```
Epoch 1/500  
375/375 [=====] - 2s 5ms/step - loss: 1.1026 - accuracy: 0.7268 - val_loss: 0.6394 - val_accuracy: 0.8544  
Epoch 2/500  
375/375 [=====] - 2s 5ms/step - loss: 0.5782 - accuracy: 0.8569 - val_loss: 0.4780 - val_accuracy: 0.8791  
Epoch 3/500  
375/375 [=====] - 2s 5ms/step - loss: 0.4750 - accuracy: 0.8754 - val_loss: 0.4161 - val_accuracy: 0.8912  
Epoch 4/500  
375/375 [=====] - 2s 5ms/step - loss: 0.4265 - accuracy: 0.8858 - val_loss: 0.3831 - val_accuracy: 0.8980  
Epoch 5/500  
375/375 [=====] - 2s 5ms/step - loss: 0.3969 - accuracy: 0.8920 - val_loss: 0.3618 - val_accuracy: 0.9022  
Epoch 6/500  
375/375 [=====] - 2s 5ms/step - loss: 0.3766 - accuracy: 0.8964 - val_loss: 0.3463 - val_accuracy: 0.9048  
...  
...  
...  
Epoch 496/500  
375/375 [=====] - 2s 5ms/step - loss: 0.0218 - accuracy: 0.9969 - val_loss: 0.0794 - val_accuracy: 0.9763  
Epoch 497/500  
375/375 [=====] - 2s 5ms/step - loss: 0.0218 - accuracy: 0.9969 - val_loss: 0.0794 - val_accuracy: 0.9764  
Epoch 498/500  
375/375 [=====] - 2s 5ms/step - loss: 0.0217 - accuracy: 0.9969 - val_loss: 0.0795 - val_accuracy: 0.9764  
Epoch 499/500  
375/375 [=====] - 2s 5ms/step - loss: 0.0216 - accuracy: 0.9970 - val_loss: 0.0794 - val_accuracy: 0.9765  
Epoch 500/500  
375/375 [=====] - 2s 5ms/step - loss: 0.0216 - accuracy: 0.9970 - val_loss: 0.0795 - val_accuracy: 0.9764
```

CLASSIC ML

100-10000 SAMPLES

TRAIN	DEV	TEST
60%	20%	20%

DEEP LEARNING

1M SAMPLES

TRAIN	D	T
98%	1%	1%



Test ...

```
[53] #checks accuracy and loss on test sample  
  
test_loss, test_acc = model.evaluate(test_images, test_labels, verbose=2)  
print('\nTest accuracy:', test_acc)
```

```
313/313 - 0s - loss: 0.0689 - accuracy: 0.9780
```

Test accuracy: 0.9779999852180481 ~98% accuracy sul test sample ...

```
[54] # plot traing and validation loss  
import matplotlib.pyplot as plt  
  
#retrieve the History object produced by the fit: a dictionary containing data about  
# everything that happened during training  
history_dict = history.history  
loss_values = history_dict['loss']  
val_loss_values = history_dict['val_loss']  
  
epochs = range(1, len(loss_values) + 1)
```

```
#'bo' = blue dot  
#'b' = solid blue line  
plt.plot(epochs, loss_values, 'bo', label='Training loss')  
plt.plot(epochs, val_loss_values, 'b', label='Validation loss')
```

```
plt.title('Training and validation loss')  
#plt.axis([0,40,0.1,0.3])
```

```
plt.xlabel('Epochs')  
plt.ylabel('Loss')  
plt.legend()  
plt.show()
```

▶ #plotting trainig and validation accuracy

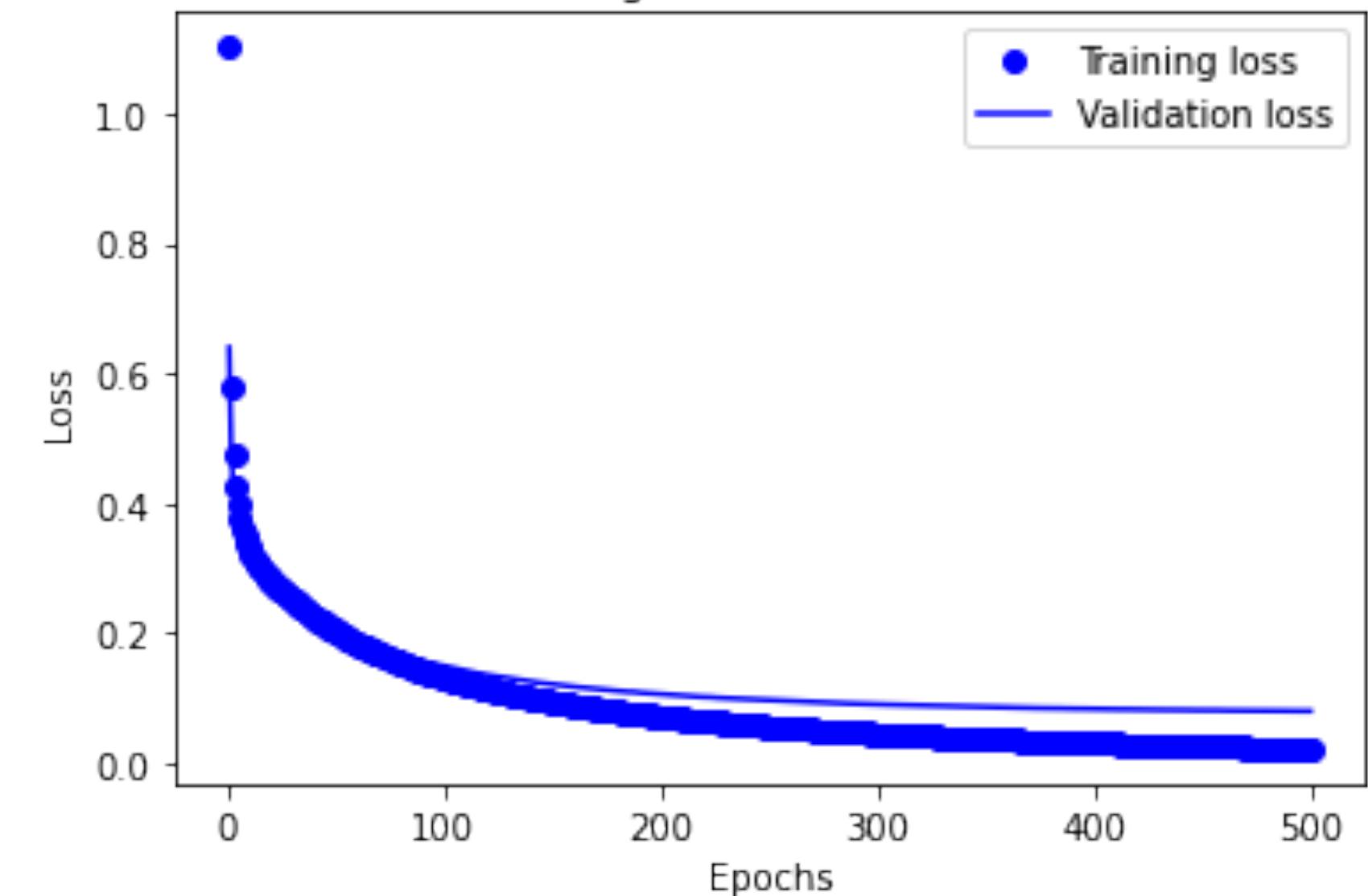
```
acc_values = history_dict['accuracy']  
val_acc_values = history_dict['val_accuracy']
```

```
plt.clf()  
plt.plot(epochs, acc_values, 'bo', label='Training acc')  
plt.plot(epochs, val_acc_values, 'b', label='Validation acc')  
plt.title('Training and validation accuracy')  
#plt.axis([0,11,0.9,1])
```

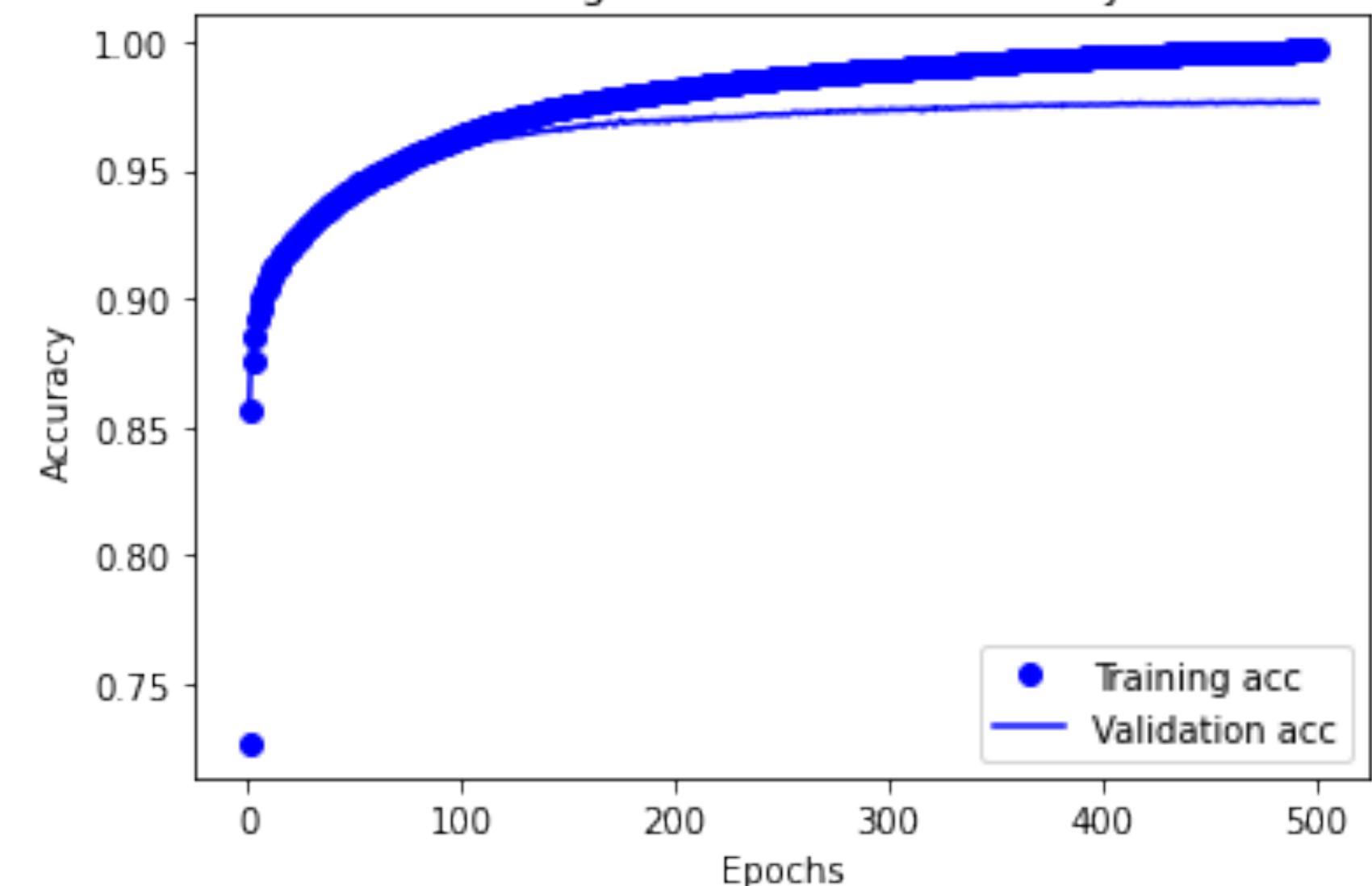
```
plt.xlabel('Epochs')  
plt.ylabel('Accuracy')  
plt.legend()
```

```
plt.show()
```

Training and validation loss



Training and validation accuracy



Uso della rete addestrata in Prediction ...

```
[56] #use the trained newtrok for predictions on additional dataset

predictions = model.predict(test_images)

print(predictions[0])
print('Max probability for number:', np.argmax(predictions[0]))
print(test_labels[0])
print('True label is:',np.argmax(test_labels[0]))

[3.4481525e-07 3.4682763e-09 5.7345255e-06 7.8907848e-04 2.2902189e-09
 2.8293982e-08 2.8238625e-11 9.9918634e-01 1.3844269e-06 1.7078450e-05]
Max probability for number: 7
[0. 0. 0. 0. 0. 0. 1. 0. 0.]
True label is: 7
```



Uso della rete addestrata in Prediction ...

plot (codice nel notebook)
per ogni immagine della
probabilità predetta dalla rete
per tutte le 10 classi

