



# Intelligenza Artificiale

## Deep Learning



# Outline

---

- ▶ ML vs DL
- ▶ Reti Neurali
- ▶ Training
- ▶ Regolarizzazione
- ▶ CNN
- ▶ RNN

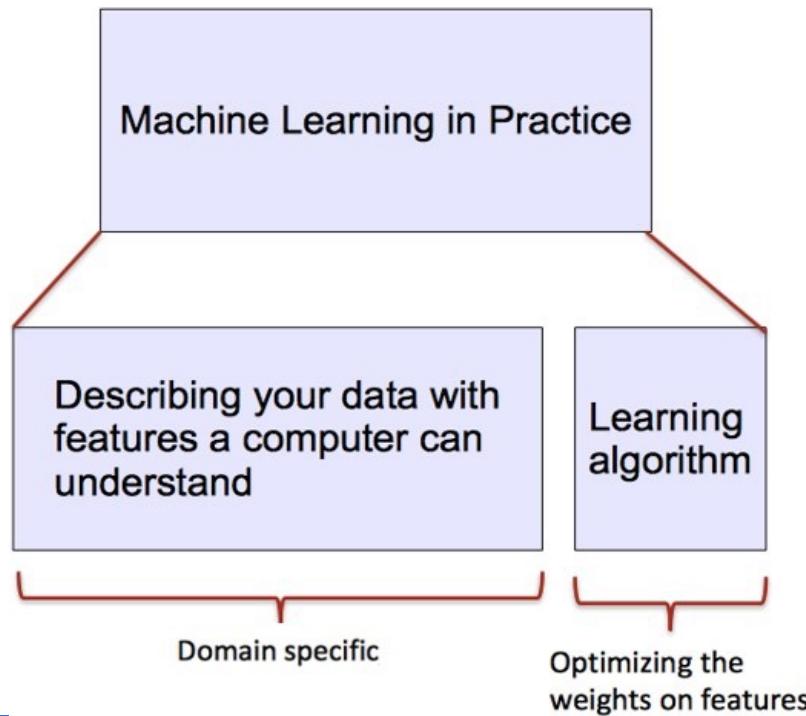
# Teorema del no free lunch

---

- I modelli di classificazione derivati per l'apprendimento supervisionato sono semplificazioni della realtà
  - Le semplificazioni si basano su certe ipotesi
  - Le ipotesi falliscono in alcuni situazioni
    - Ad esempio, a causa dell'incapacità di stimare perfettamente i parametri del modello ML da limitati dati
- In sintesi, *il teorema del no free lunch* afferma:
  - **Nessun singolo classificatore funziona al meglio per tutti i possibili i problemi**
  - Dal momento che dobbiamo fare ipotesi per generalizzare

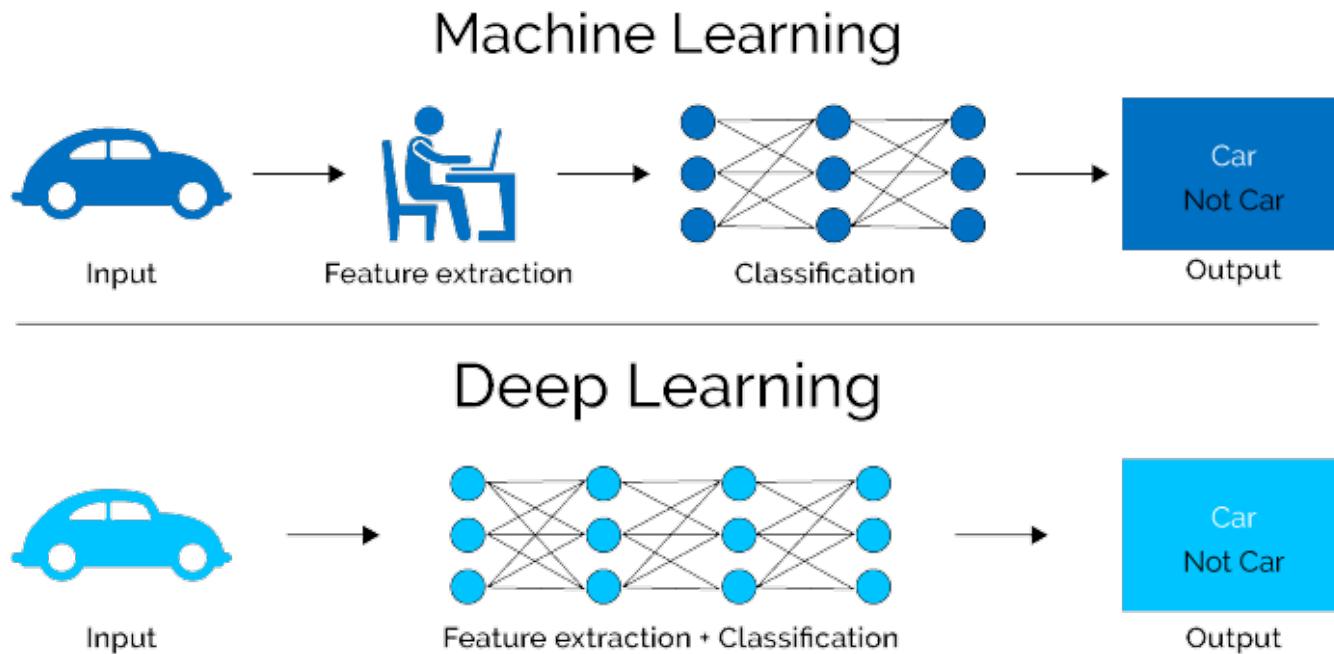
# ML contro Deep Learning

- I metodi di apprendimento automatico convenzionali si basano su **rappresentazioni di features progettate dall'uomo**
  - ML diventa solo l'ottimizzazione dei pesi per fare al meglio le predizioni



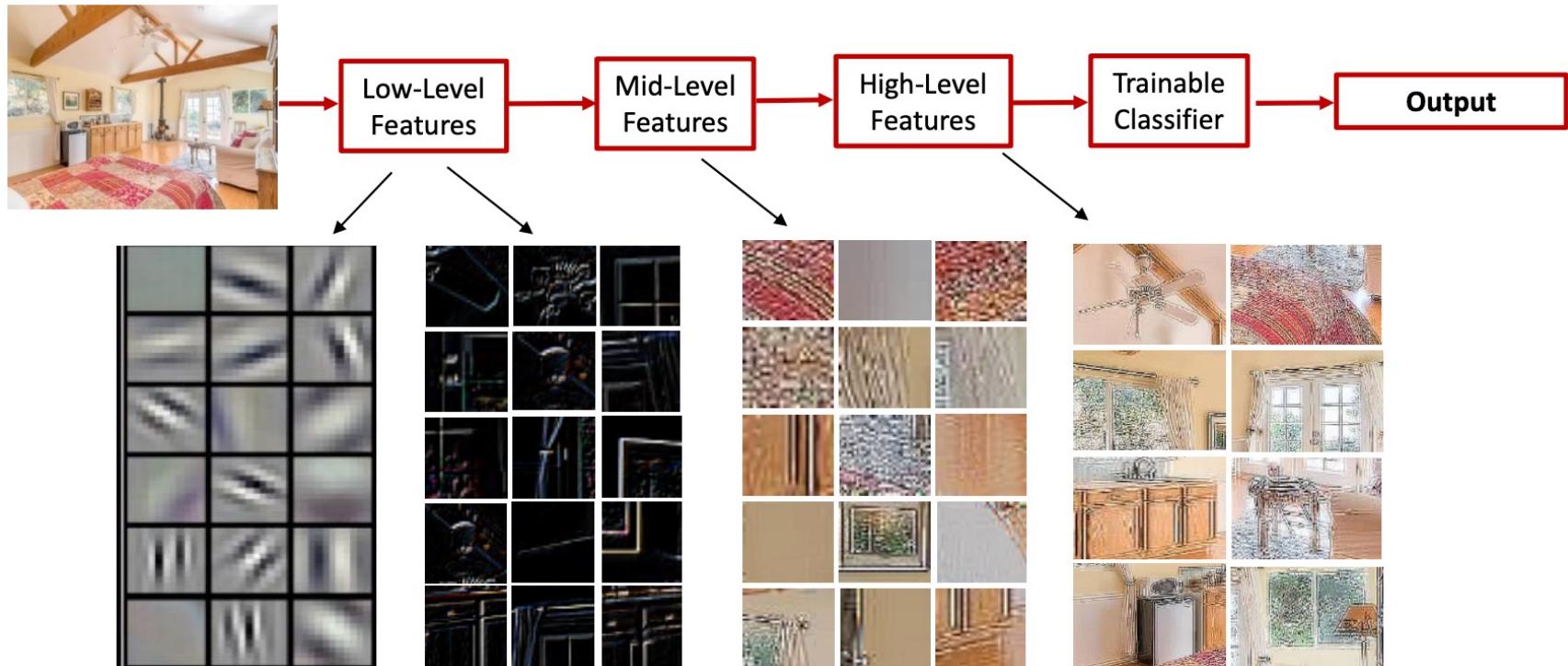
# ML vs. Deep Learning

- *Il deep learning* (DL) è un sottocampo di machine learning che utilizza più livelli per apprendere le rappresentazioni dei dati
  - DL è eccezionalmente efficace nell'apprendimento di pattern



# ML contro Deep Learning

- DL applica un processo multistrato per l'apprendimento di ricche features gerarchiche (es. rappresentazioni dei dati)
  - Pixel di una immagine → Bordi → Texture → Parti → Oggetti



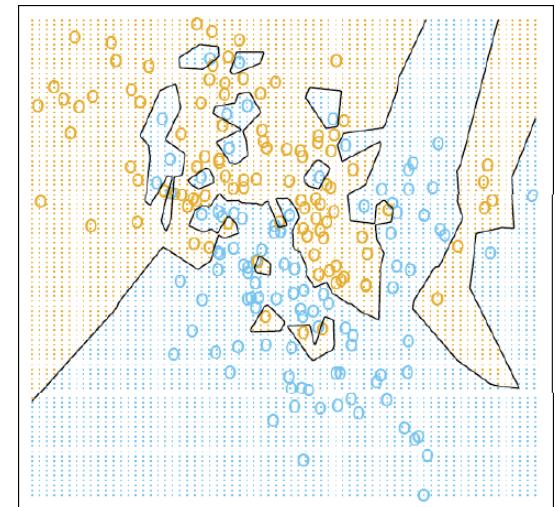
# Perché il DL è utile?

---

- DL fornisce un framework flessibile e di apprendimento per la rappresentazione di informazioni visive, testo, linguistiche
  - Può apprendere in modo supervisionato e non
- DL rappresenta un efficace sistema di apprendimento end-to-end
- Richiede grandi quantità di training
- Dal 2010 circa, DL ottiene prestazioni migliori delle altre tecniche di ML
  - Prima nella visione e nel parlato, poi nel NLP ed altre applicazioni

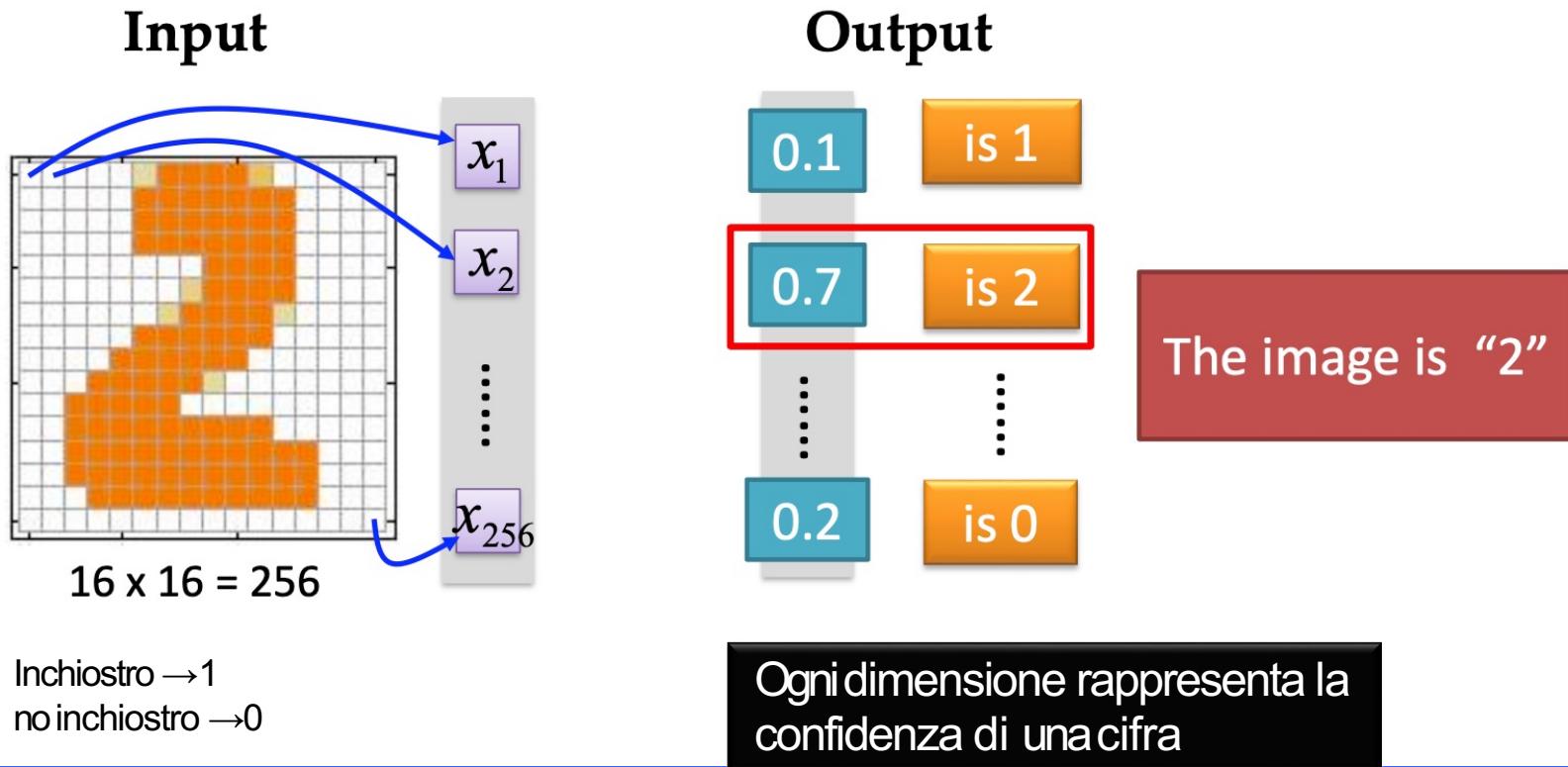
# Potere Rappresentativo

- Le NN con almeno un livello nascosto sono **approssimatori universali**
  - Data una qualsiasi funzione continua  $h(x)$  e qualche  $\epsilon > 0$ , esiste una NN con uno strato nascosto (e con una ragionevole scelta di non linearità) descritto con la funzione  $f(x)$ , tale che  $\forall x, |h(x) - f(x)| < \epsilon$
  - Cioè, le NN può approssimare qualsiasi funzione continua complessa
- Le NN utilizzano la mappatura non lineare degli input  $x$  agli output  $f(x)$  per *calcolare* confini decisionali complessi
- Ma allora, perché usare NN più profonde?
  - Il fatto che le NN profonde funzionino meglio è un'osservazione empirica
  - Matematicamente, le NN profonde hanno lo stesso potere rappresentativo di una NN ad un livello



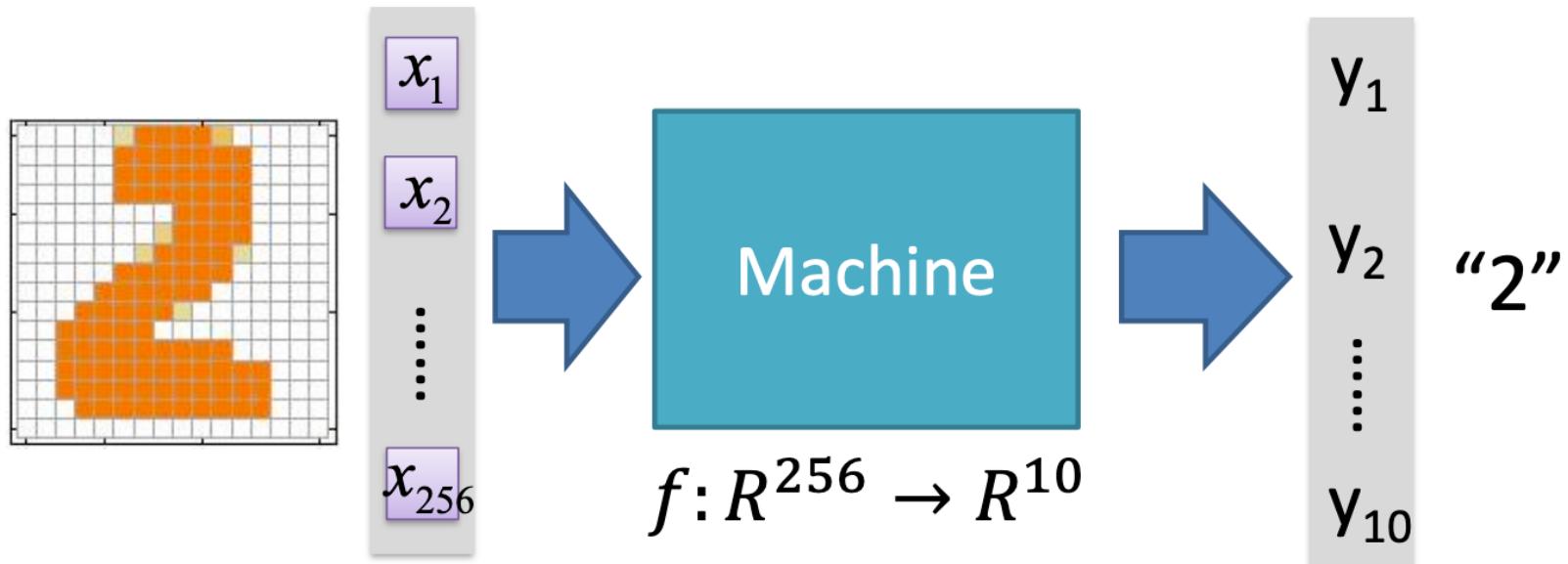
# Introduzione alle reti neurali

- Riconoscimento delle cifre scritte a mano ( [set di dati MNIST](#) )
  - L' intensità di ciascun pixel è considerato un elemento di **input**
  - L' **output** è la classe della cifra



# Introduzione alle reti neurali

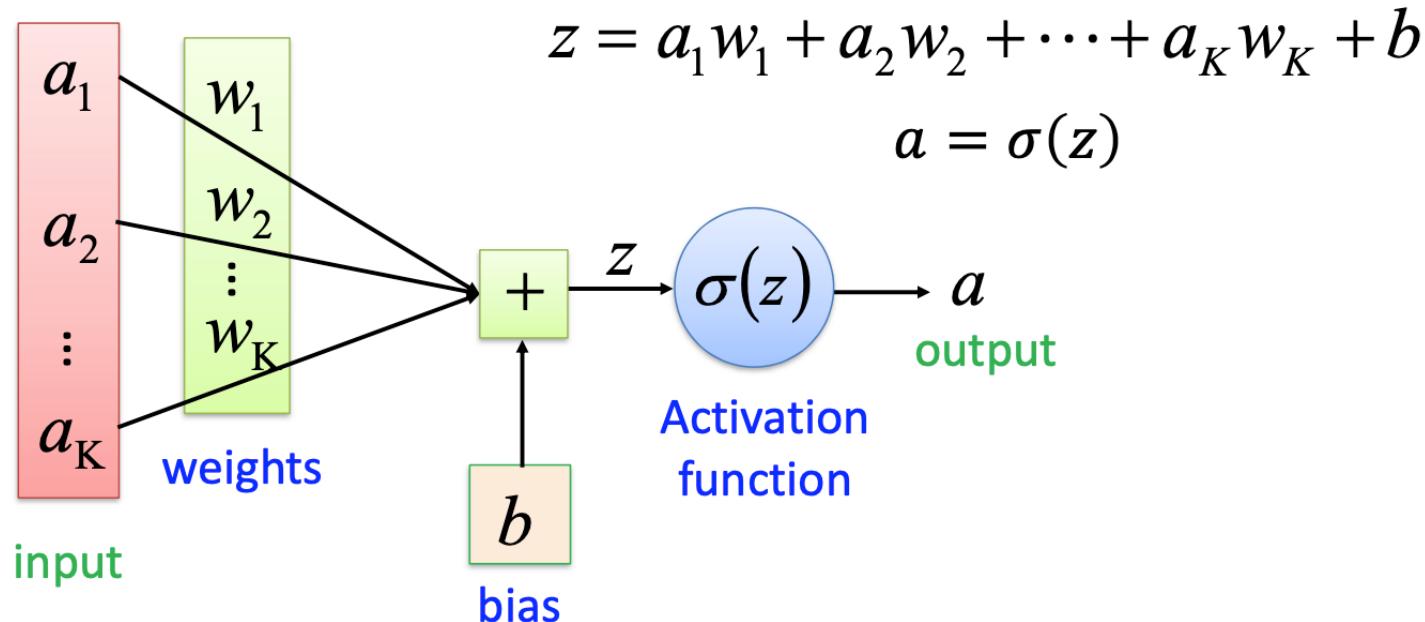
- Riconoscimento delle cifre scritte a mano



La funzione  $f$  è rappresentata da una rete neurale

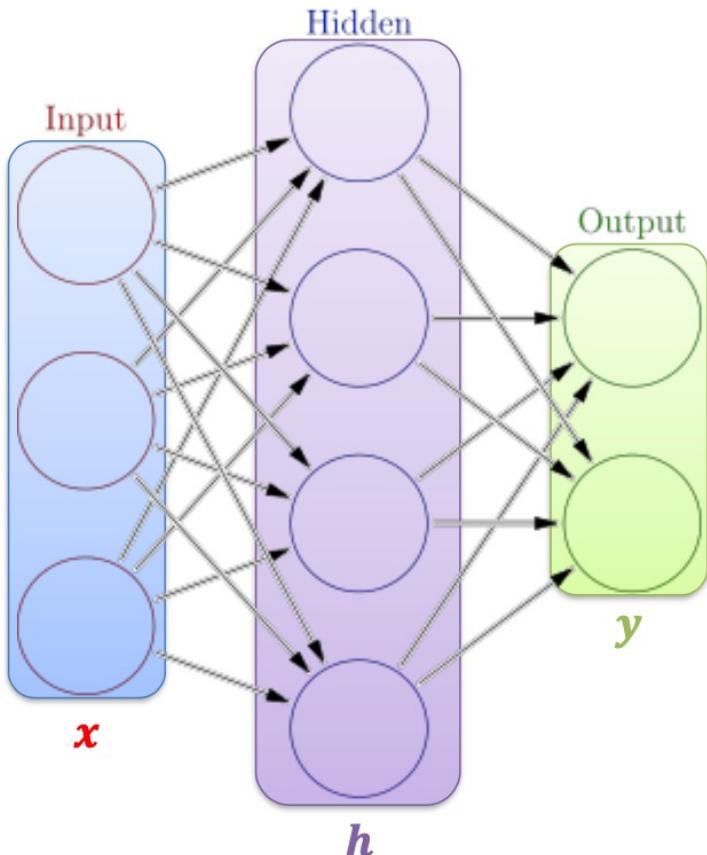
# Elementi di reti neurali

- Le NN sono costituite da strati nascosti con neuroni (cioè unità di calcolo)
- Un singolo **neurone** mappa un insieme di input in un numero di output , o  $f: R^K \rightarrow R$



# Elementi di reti neurali

- Una NN con un livello nascosto e uno strato di output



Weights      Biases

$$\text{hidden layer } h = \sigma(W_1 x + b_1)$$
$$\text{output layer } y = \sigma(W_2 h + b_2)$$

Activation functions

$4 + 2 = 6$  neurons (not counting inputs)

$[3 \times 4] + [4 \times 2] = 20$  weights

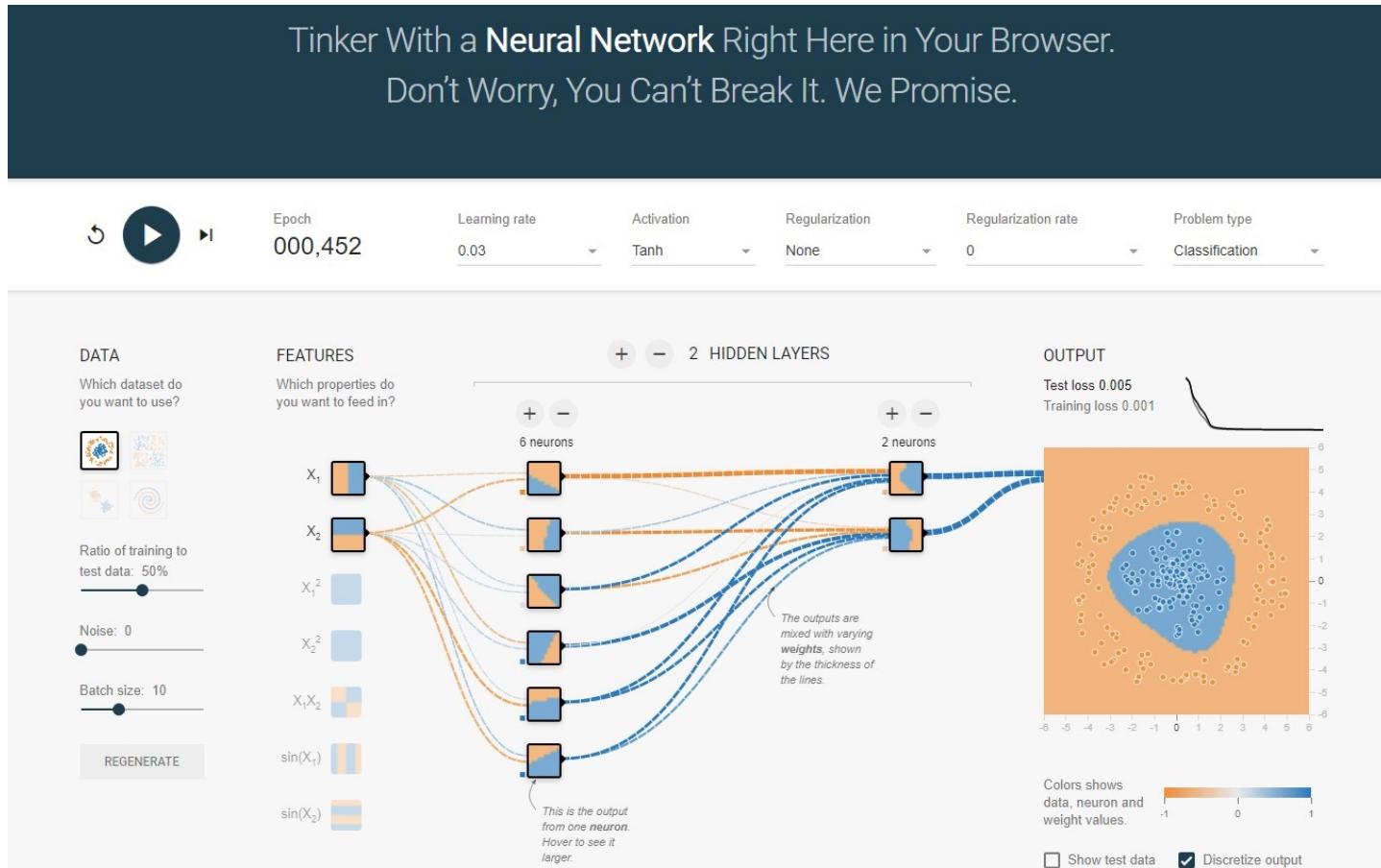
$4 + 2 = 6$  biases

---

26 learnable parameters

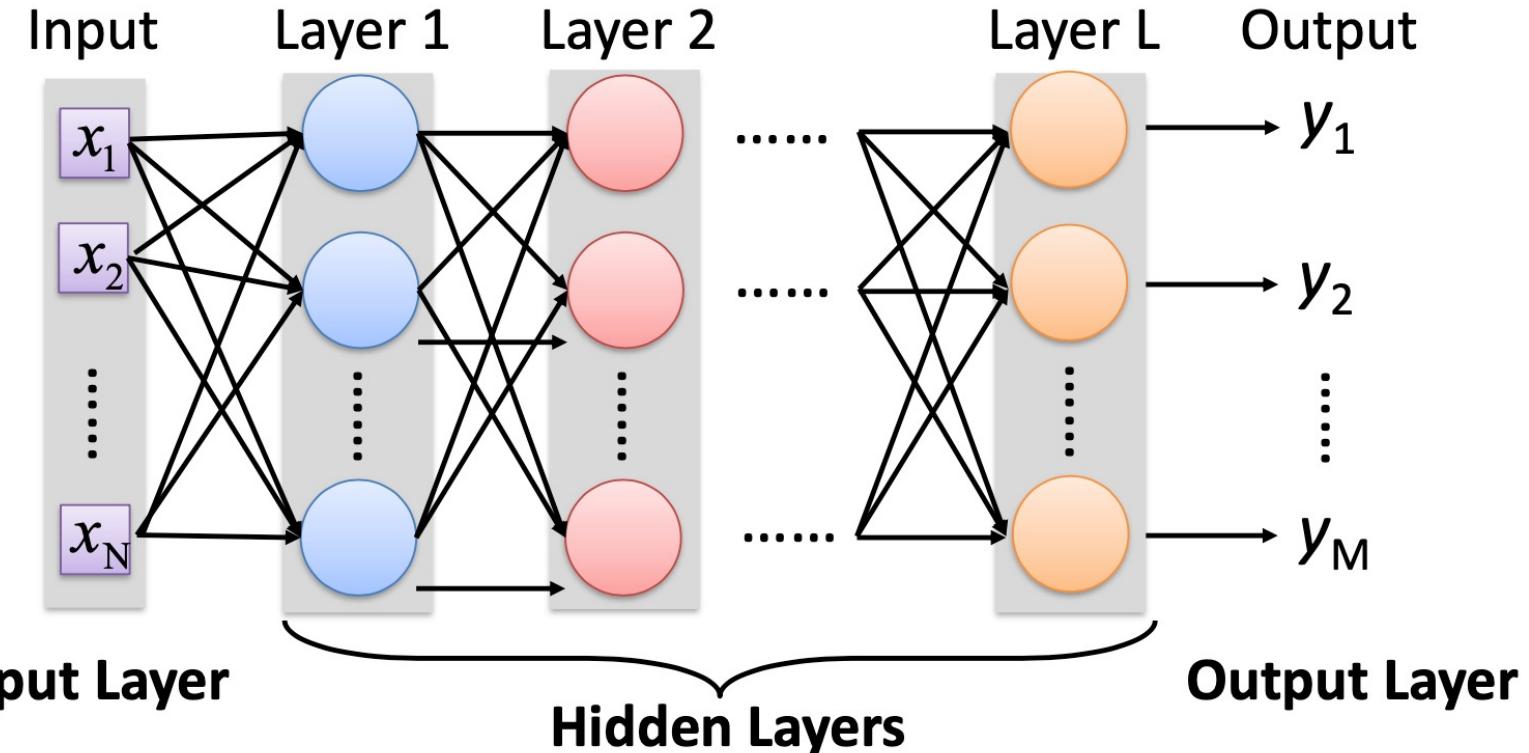
# Elementi di reti neurali

- Un playground della rete neurale [link](#)



# Elementi di reti neurali

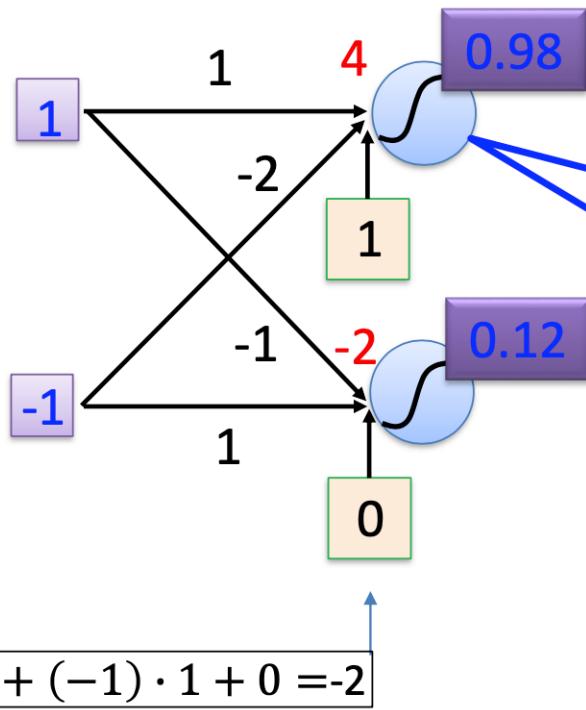
- Le Deep NN ne hanno molti di strati nascosti
  - Strati completamente connessi (densi) (aka Multi-Layer Perceptron o MLP)
  - Ogni neurone è connesso a tutti i neuroni nello strato successivo



# Elementi di reti neurali

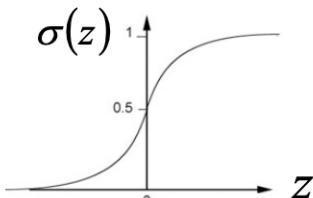
- Una rete semplice, toy example

$$(1 \cdot 1) + (-1) \cdot (-2) + 1 = 4$$



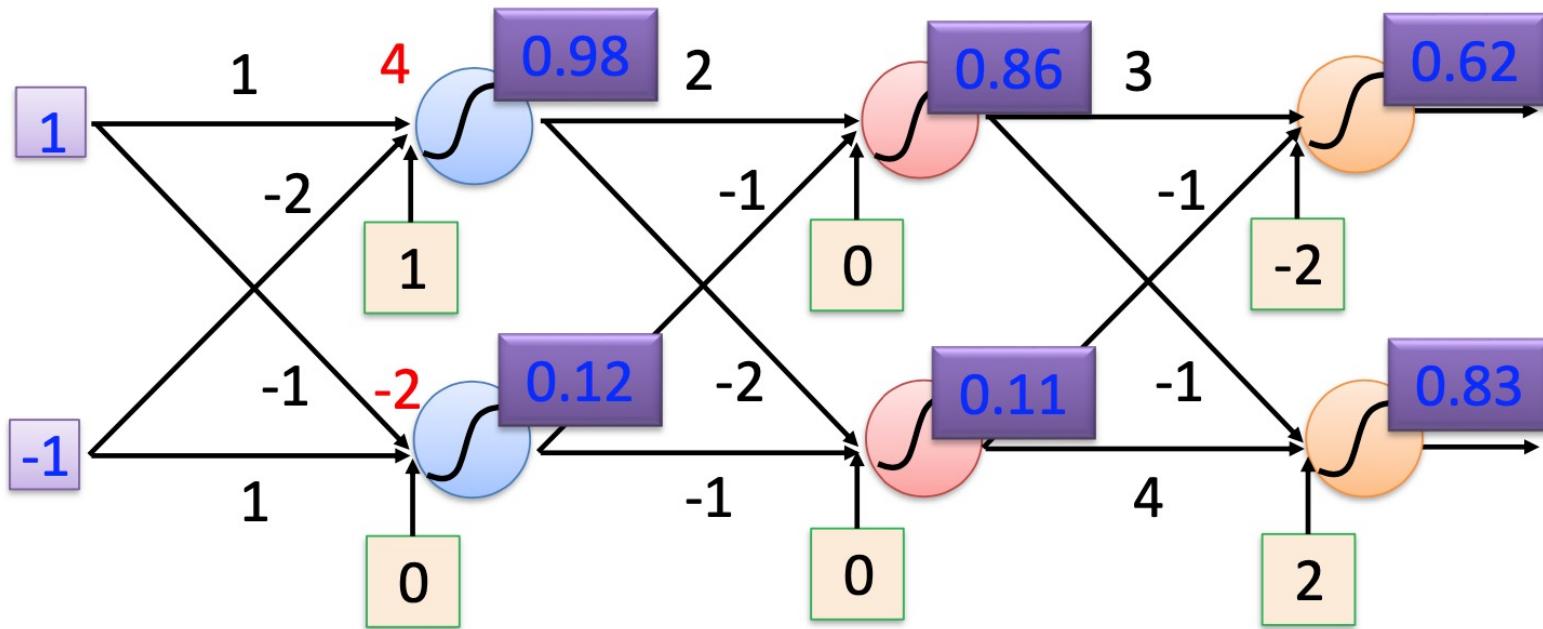
Sigmoid Function

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



# Elementi di reti neurali

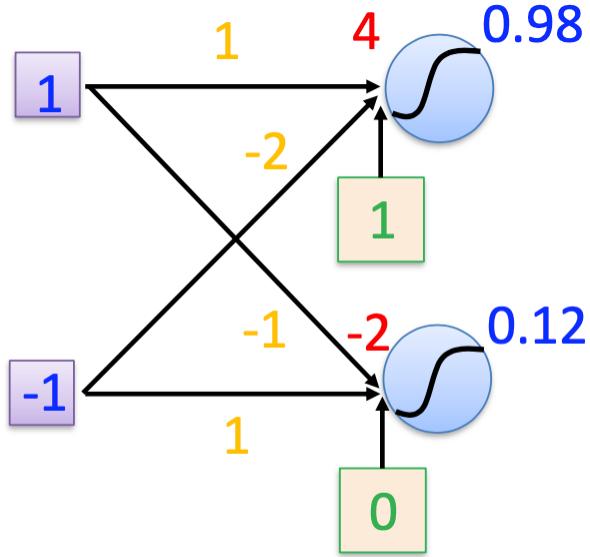
- Un semplice esempio di rete (continua)
  - Per un input vettore  $[ 1 \ -1 ]^T$ , l'output è  $[ 0.62 \ 0.83 ]^T$



$$f: R^2 \rightarrow R^2 \quad f \left( \begin{bmatrix} 1 \\ -1 \end{bmatrix} \right) = \begin{bmatrix} 0.62 \\ 0.83 \end{bmatrix}$$

# Operazioni su Matrici

- Le operazioni su matrici sono utili quando si lavora con input e output multidimensionali

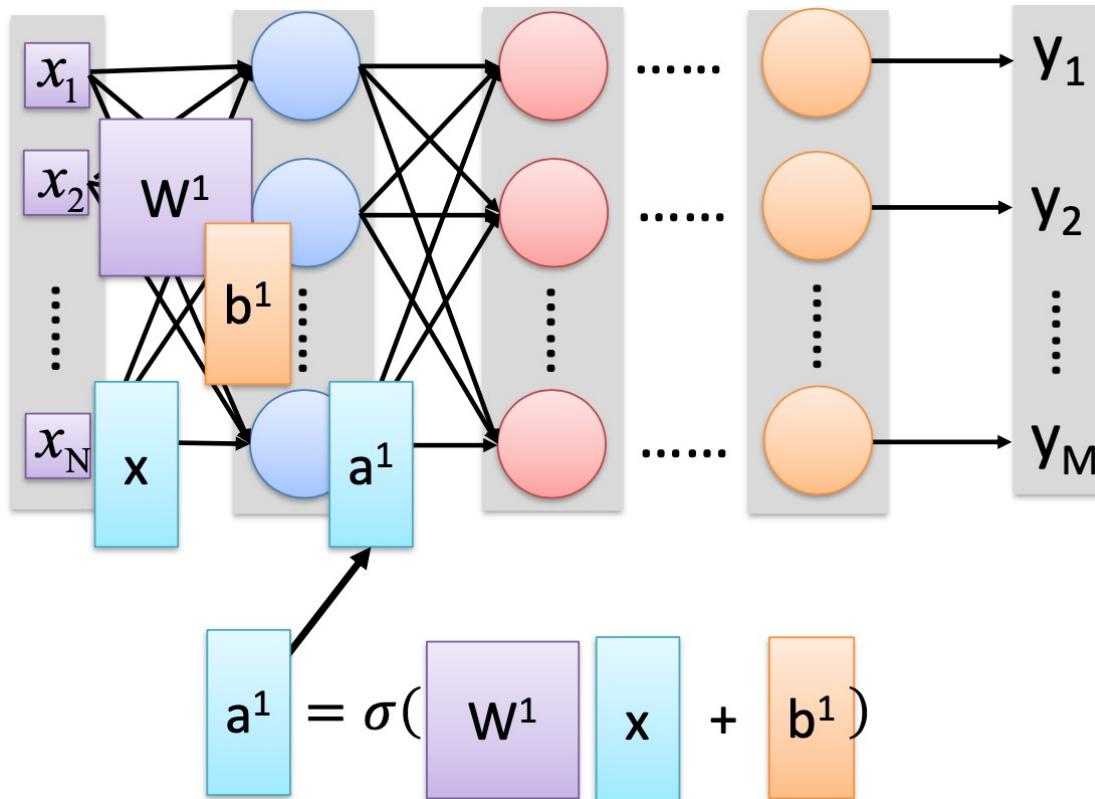


$$\sigma(\underbrace{Wx + b}_{a}) = a$$

$$\sigma(\underbrace{\begin{bmatrix} 1 & -2 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix}}_{\begin{bmatrix} 4 \\ -2 \end{bmatrix}}) = \begin{bmatrix} 0.98 \\ 0.12 \end{bmatrix}$$

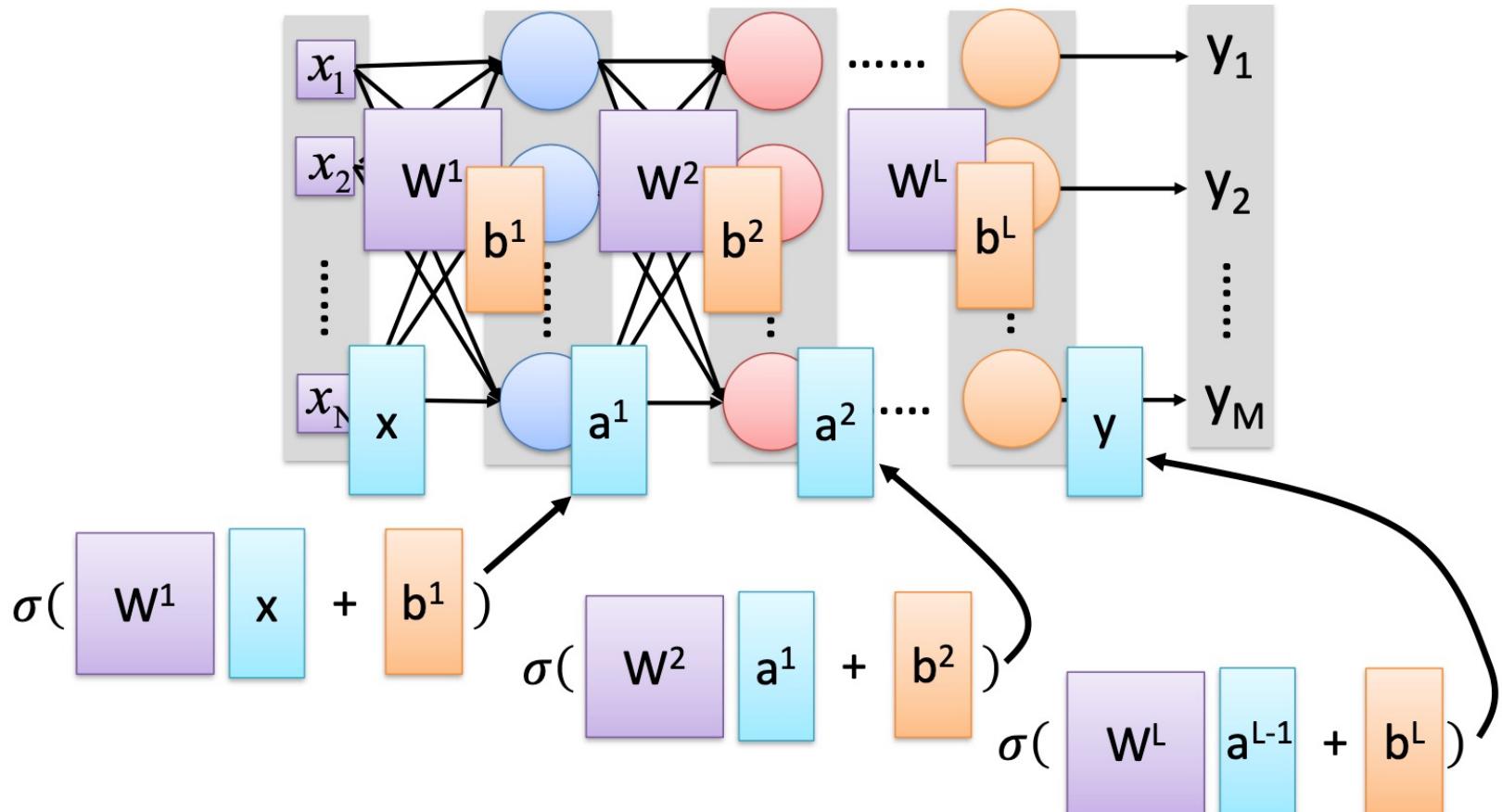
# Operazioni su Matrici

- Multilayer NN, calcoli matriciali per il primo strato
  - Vettore di input  $x$ , matrice di pesi  $W^1$ , vettore di bias  $b^1$ , vettore di output  $a^1$



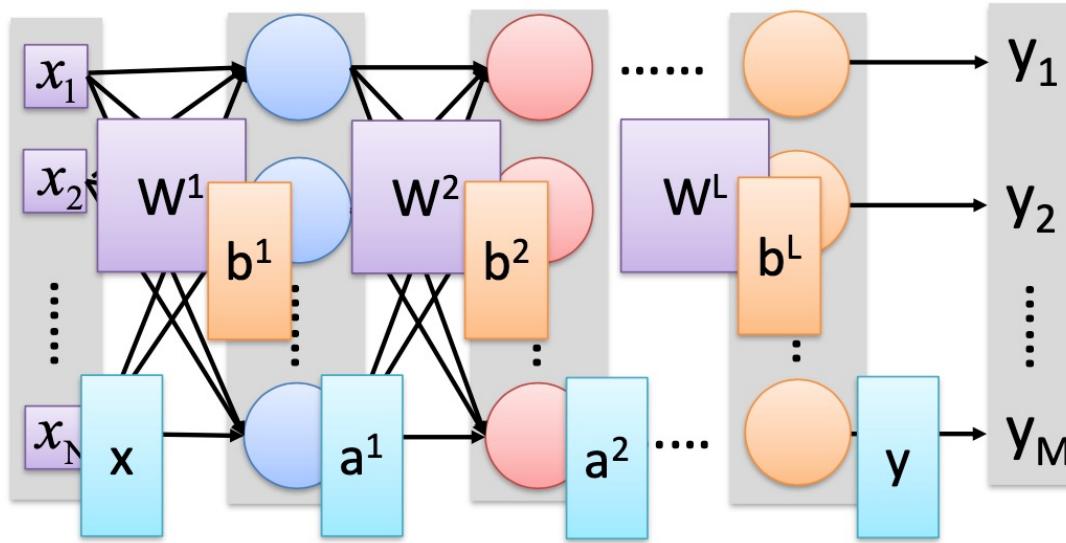
# Operazioni su Matrici

- Multilayer NN, calcoli matriciali per tutti strati



# Operazioni su Matrici

- Multilayer NN, la funzione  $f$  mappa gli input  $x$  agli output  $y$ , cioè  $y = f(x)$

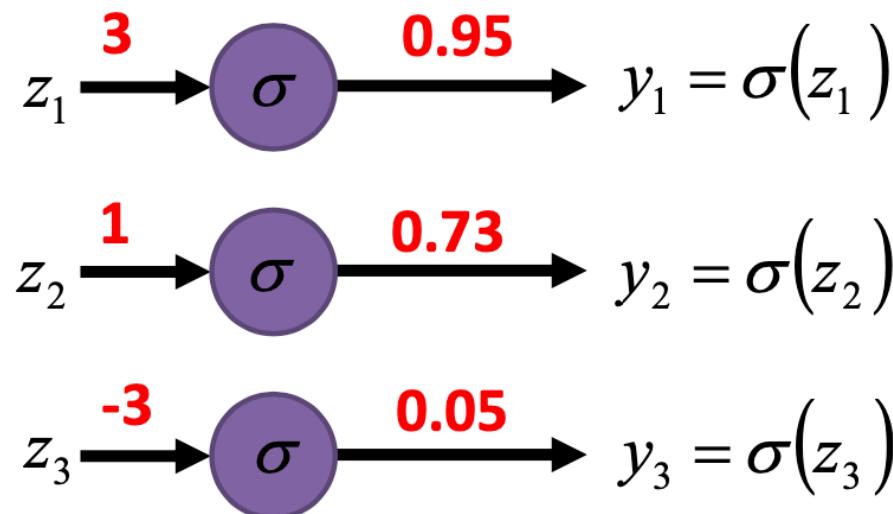


$$y = f(x) = \sigma(W^L \dots \sigma(W^2 \sigma(W^1 x + b^1) + b^2) \dots + b^L)$$

# Softmax Layer

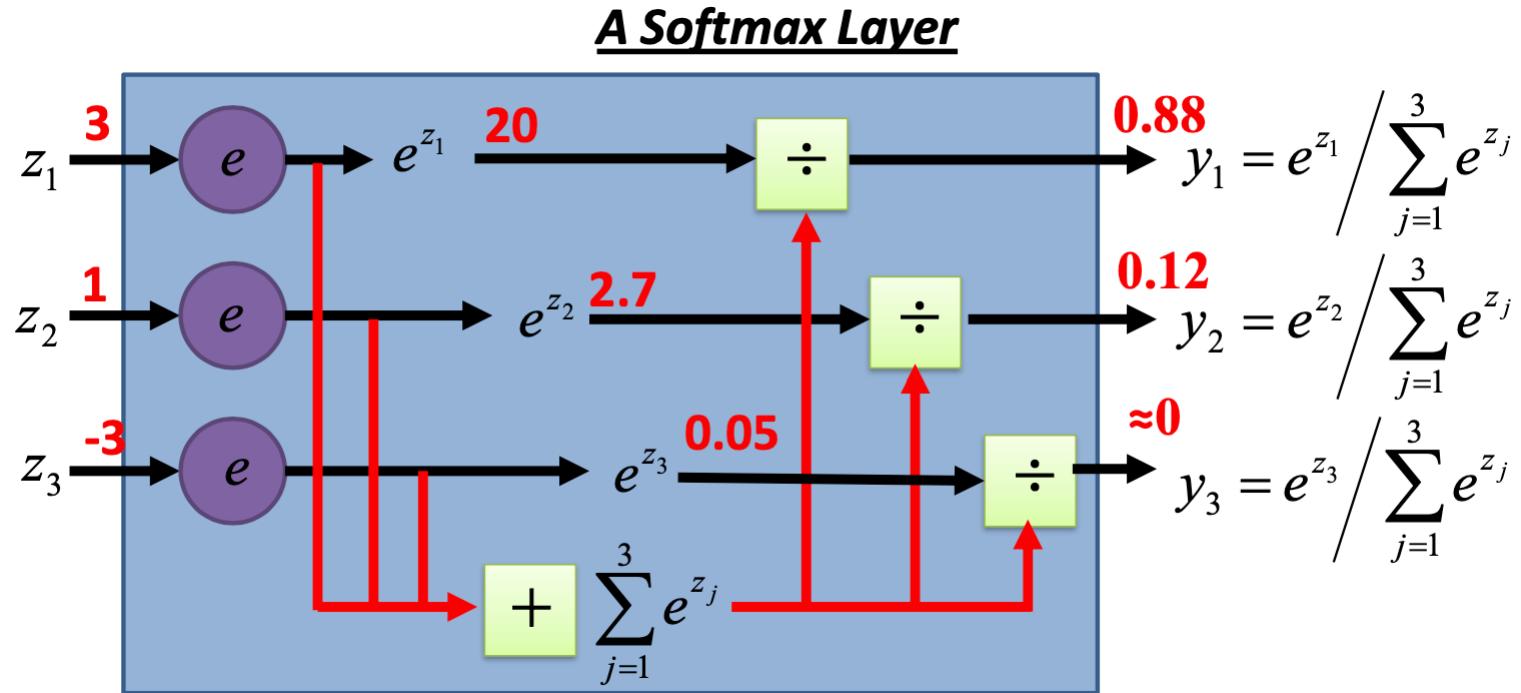
- Nelle attività di **classificazione multiclasse**, il livello di output è in genere un **strato softmax**
  - Cioè , impiega **una funzione di attivazione softmax**
  - Se invece viene utilizzato uno strato con una funzione di attivazione sigmoidea come strato di output , le previsioni della NN potrebbero non essere facili da interpretare

## A Layer with Sigmoid Activations



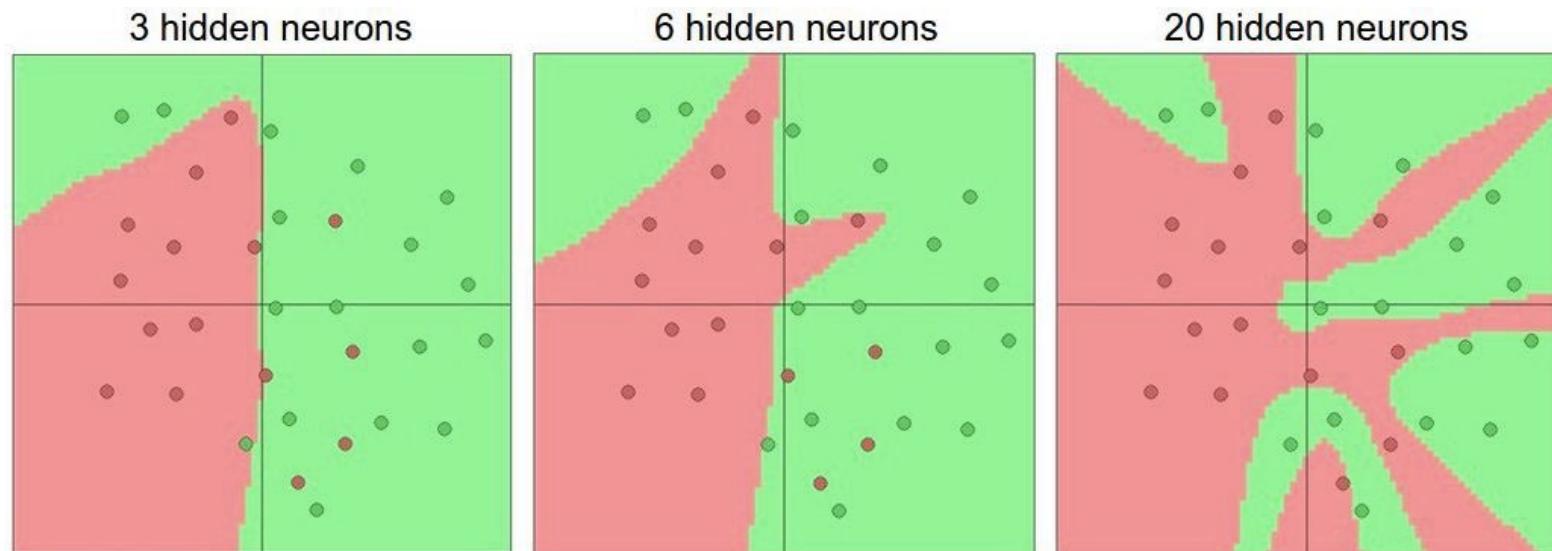
# Softmax Strato

- Lo **strato softmax** applica le attivazioni softmax per produrre un valore di probabilità nell'intervallo  $[0, 1]$ 
  - Si fa riferimento ai valori  $z$  immessi nel livello softmax come *logits*



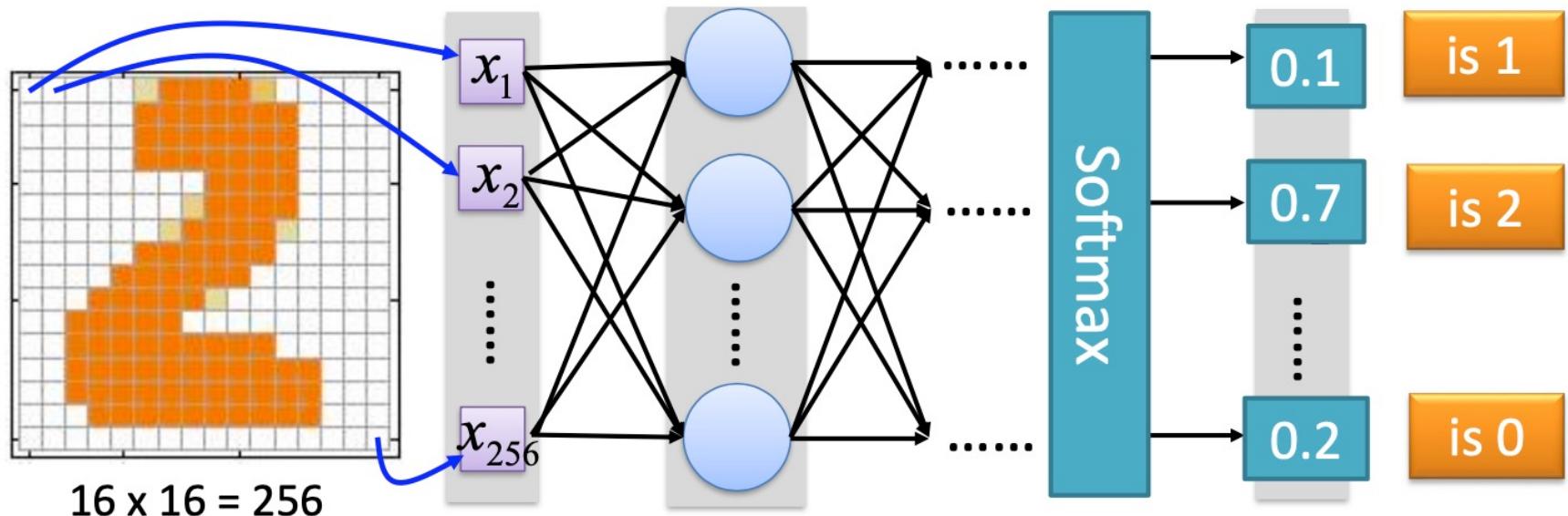
# Funzioni di attivazione

- Sono necessarie **attivazioni non lineari** per apprendere rappresentazioni di dati complessi (non lineari)
  - Altrimenti, NNs sarebbe solo una funzione lineare (come  $W_1W_2x = Wx$ )
  - NN con un gran numero di strati (e neuroni) possono approssimare funzioni più complesse, più neuroni migliorano la rappresentazione (ma potrebbero andare in overfit)



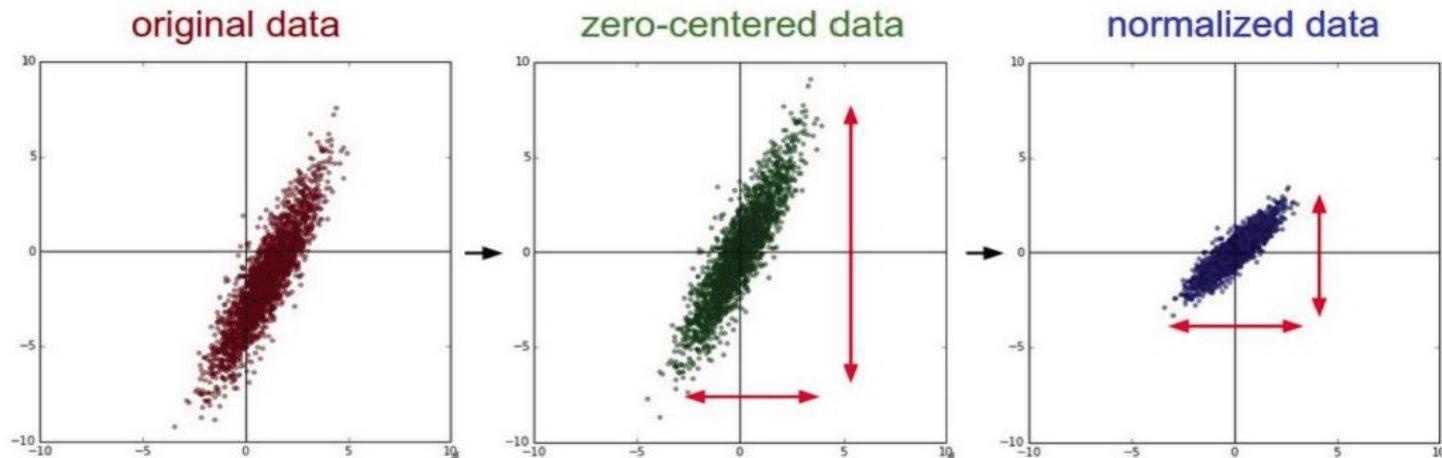
# Training NN

- I **parametri** di rete  $\theta$  includono le **matrici dei pesi** e i **vettori di bias** per tutti i layer  
$$\theta = \{W^1, b^1, W^2, b^2, \dots, W^L, b^L\}$$
  - Spesso i parametri del modello  $\theta$  sono indicati come **pesi**
- Addestrare un modello per apprendere una serie di parametri  $\theta$  che sono ottimali (secondo un criterio) è una delle maggiori sfide nel ML



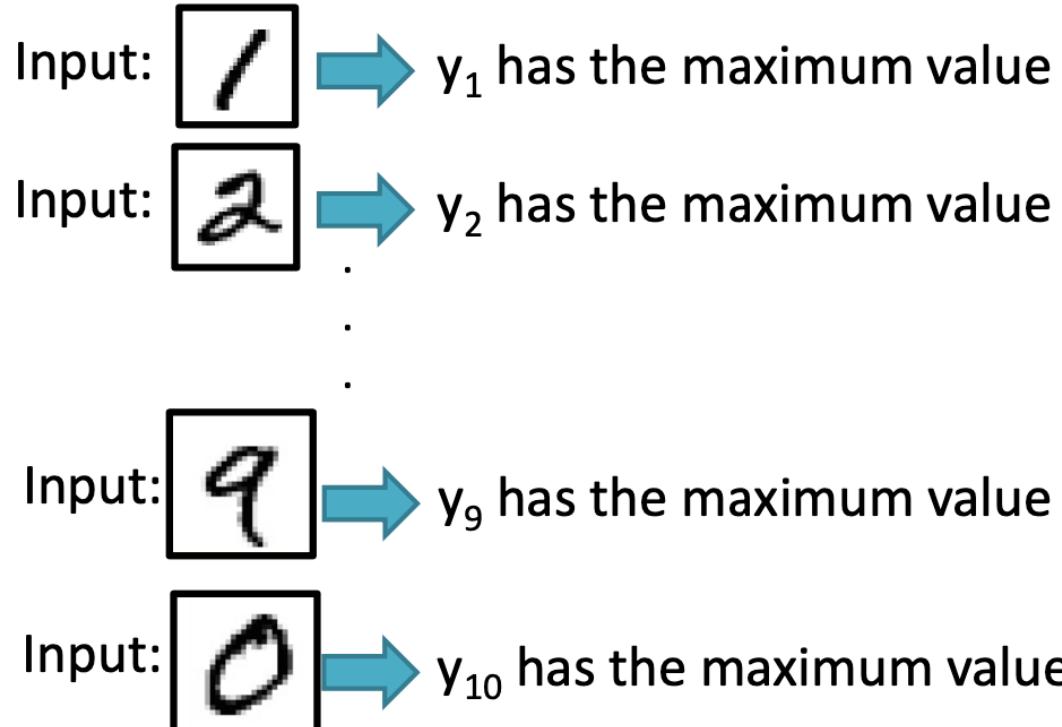
# Training NN

- **Preelaborazione dei dati** - aiuta la convergenza durante addestramento
  - **Sottrazione della media**, per ottenere dati incentrati sullo zero
    - Sottrarre la media per ogni singola dimensione dei dati (feature)
  - **Normalizzazione**
    - Dividi ogni feature per la sua deviazione standard
      - Per ottenere una deviazione standard di 1 per ciascuna dimensione dei dati (feature)
    - Oppure, ridimensionare i dati all'interno dell'intervallo  $[0,1]$  o  $[-1, 1]$ 
      - Ad esempio, le intensità dei pixel dell'immagine sono divise per 255 per ridimensionare in  $[0,1]$



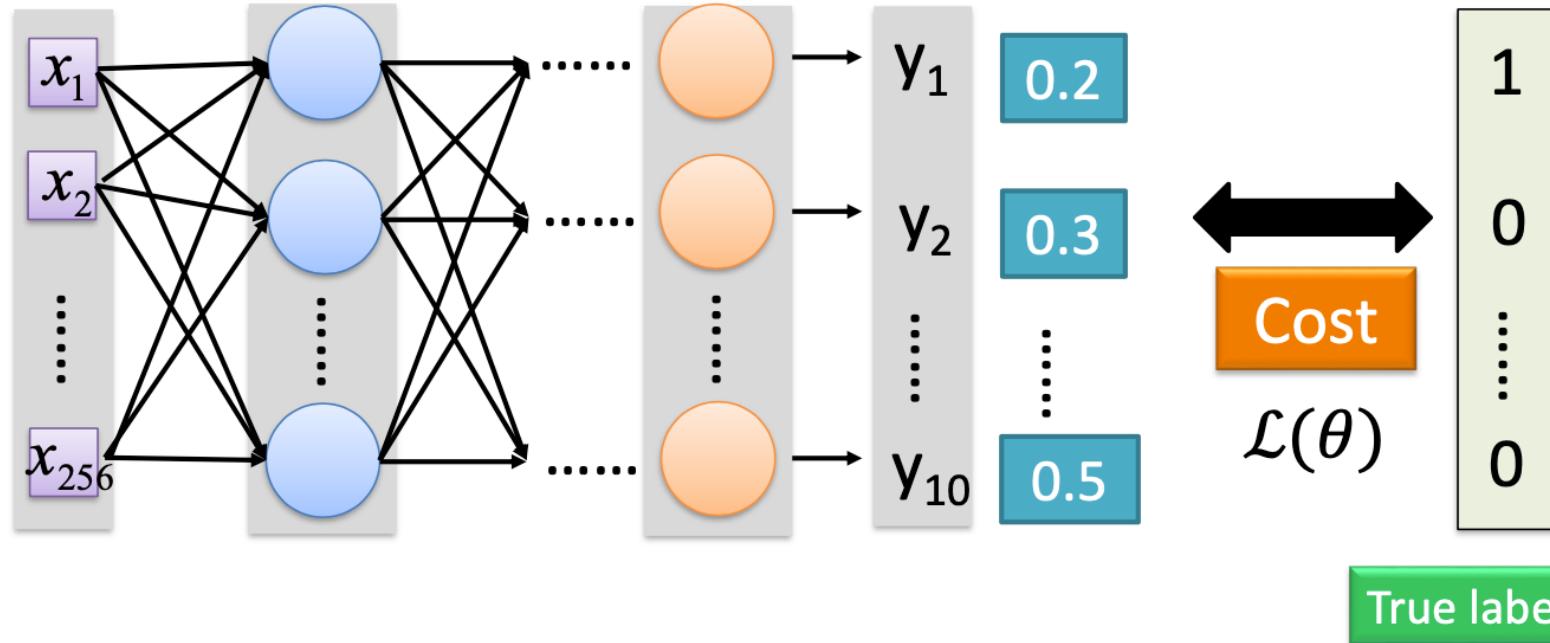
# Training NN

- Per addestrare una NN, bisogna imposta i parametri  $\theta$  in modo che per un sottoinsieme di immagini di addestramento, gli elementi corrispondenti nell'output previsto hanno i valori massimi



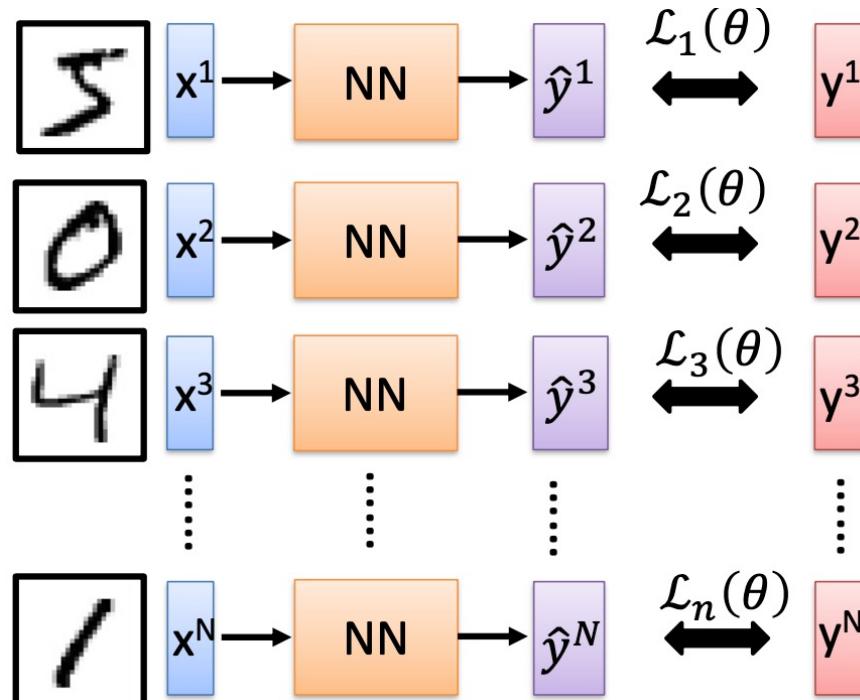
# Training NN

- Definire una *funzione di perdita / funzione obiettivo / funzione di costo*  $\mathcal{L}(\theta)$  che calcola la differenza (errore) tra la previsione del modello e l'etichetta vera
  - Per esempio, può essere errore quadratico medio, cross-entropy, eccetera.



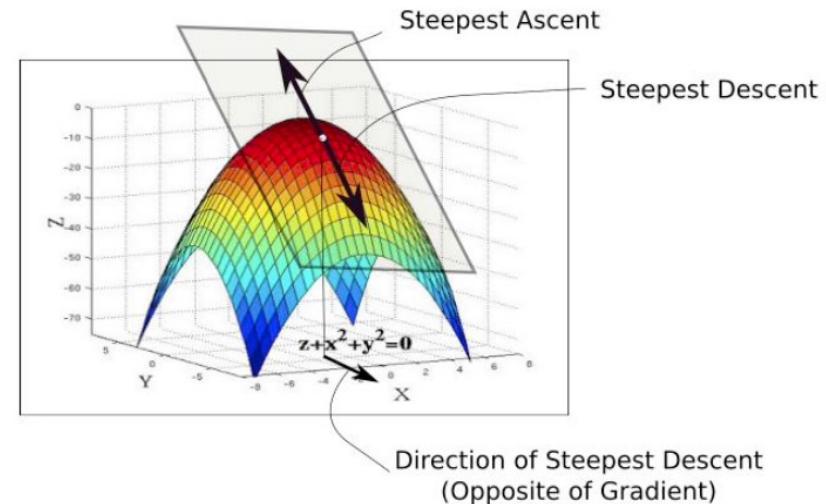
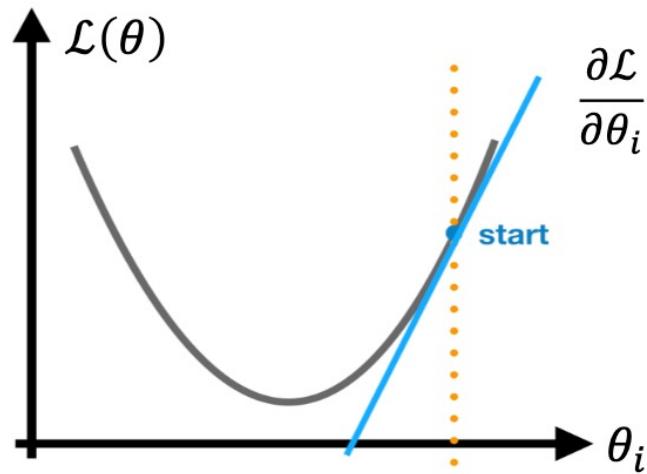
# Training NN

- Per un training set di  $N$  immagini, calcola la loss totale su tutte le immagini  $\mathcal{L}(\theta) = \sum_{n=1}^N \mathcal{L}_n(\theta)$
- Trova i parametri ottimali  $\theta^*$  che minimizzino la perdita totale  $\mathcal{L}(\theta)$



# Training NN

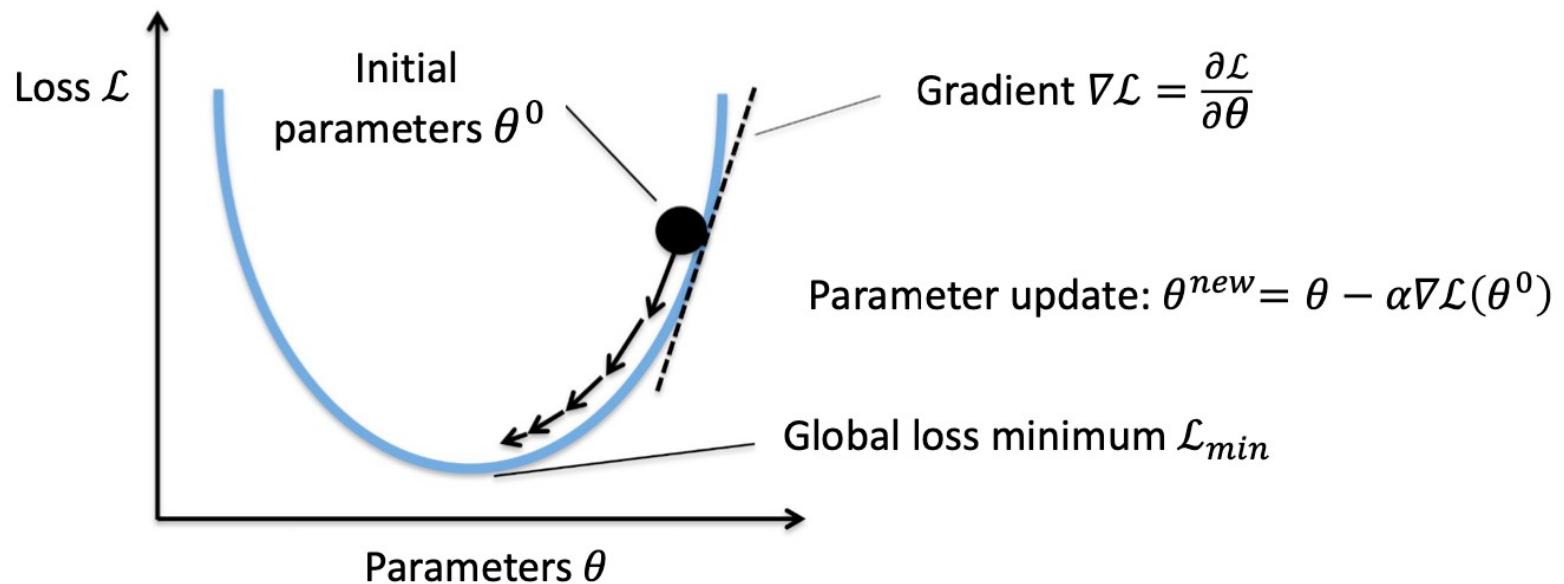
- Ottimizzazione della funzione di perdita  $\mathcal{L}(\theta)$ 
  - Quasi tutti i modelli DL oggigiorno vengono allenati con una variante dell'algoritmo di *discesa del gradiente (GD)*
  - GD applica il raffinamento iterativo dei **parametri** di rete  $\theta$
  - GD utilizza la direzione opposta del **gradiente della perdita** rispetto ai parametri NN (cioè,  $\nabla \mathcal{L}(\theta) = [\partial \mathcal{L} / \partial \theta_i]$ ) per l'aggiornamento di  $\theta$ 
    - Il gradiente della funzione di perdita  $\nabla \mathcal{L}(\theta)$  dà la direzione dell'aumento più rapido della funzione di perdita  $\mathcal{L}(\theta)$  quando i parametri  $\theta$  sono cambiati



# Algoritmo di discesa del gradiente

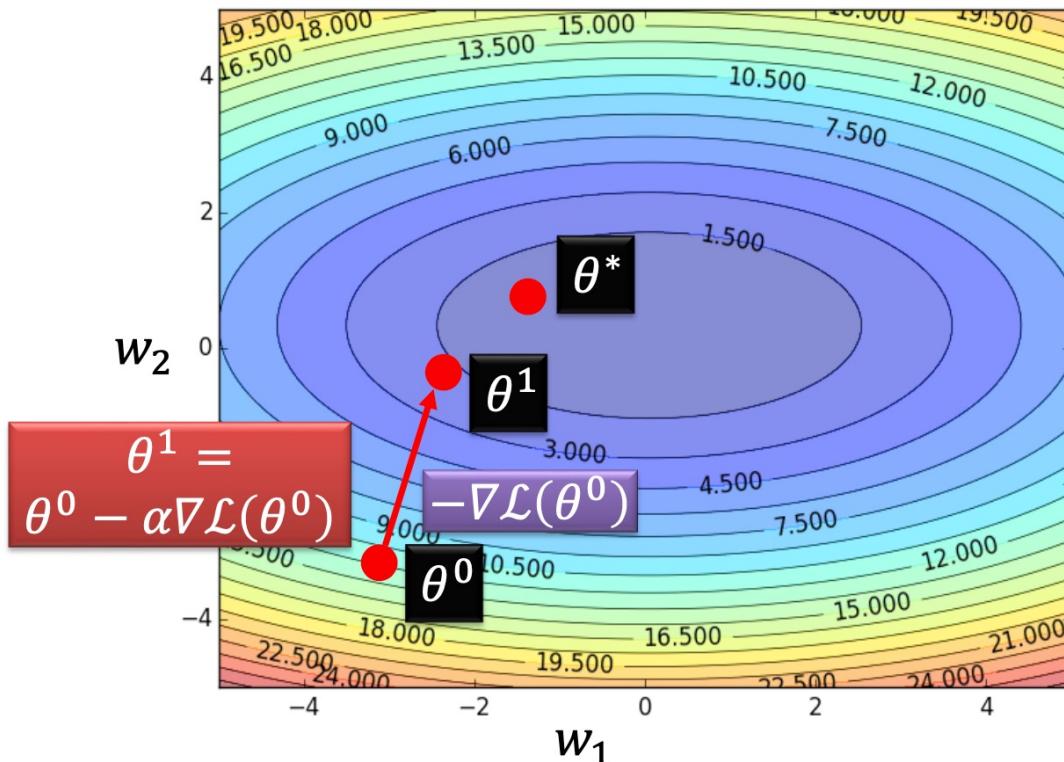
- Steps in the *gradient descent algorithm*:

1. Randomly initialize the model parameters,  $\theta^0$
2. Compute the gradient of the loss function at the initial parameters  $\theta^0$ :  $\nabla \mathcal{L}(\theta^0)$
3. Update the parameters as:  $\theta^{new} = \theta^0 - \alpha \nabla \mathcal{L}(\theta^0)$ 
  - o Where  $\alpha$  is the learning rate
4. Go to step 2 and repeat (until a terminating criterion is reached)



# Algoritmo di discesa del gradiente

- Esempio: una NN con solo 2 parametri  $w_1$  e  $w_2$ , cioè  $\theta = \{w_1, w_2\}$
- I diversi colori rappresentano i valori della perdita (perdita minima  $\theta^*$  è  $\approx 1.3$ )

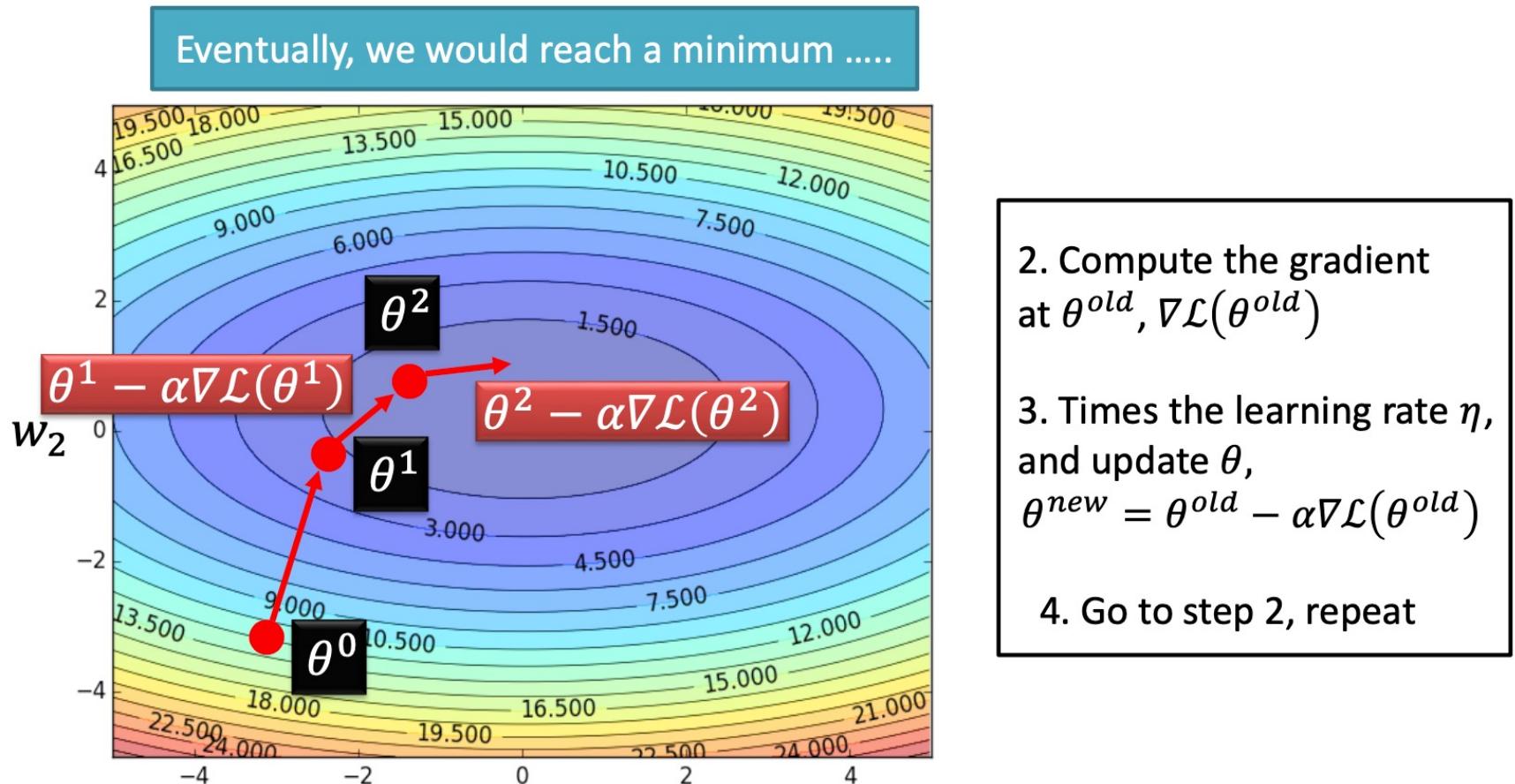


1. Randomly pick a starting point  $\theta^0$
2. Compute the gradient at  $\theta^0$ ,  $\nabla L(\theta^0)$
3. Times the learning rate  $\eta$ , and update  $\theta$ ,  
 $\theta^{new} = \theta^0 - \alpha \nabla L(\theta^0)$
4. Go to step 2, repeat

$$\nabla L(\theta^0) = \begin{bmatrix} \partial L(\theta^0)/\partial w_1 \\ \partial L(\theta^0)/\partial w_2 \end{bmatrix}$$

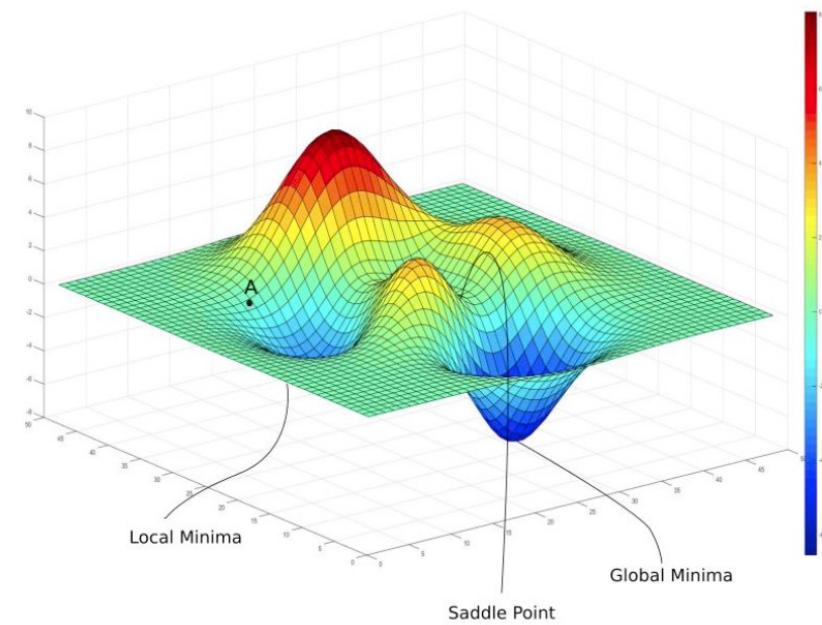
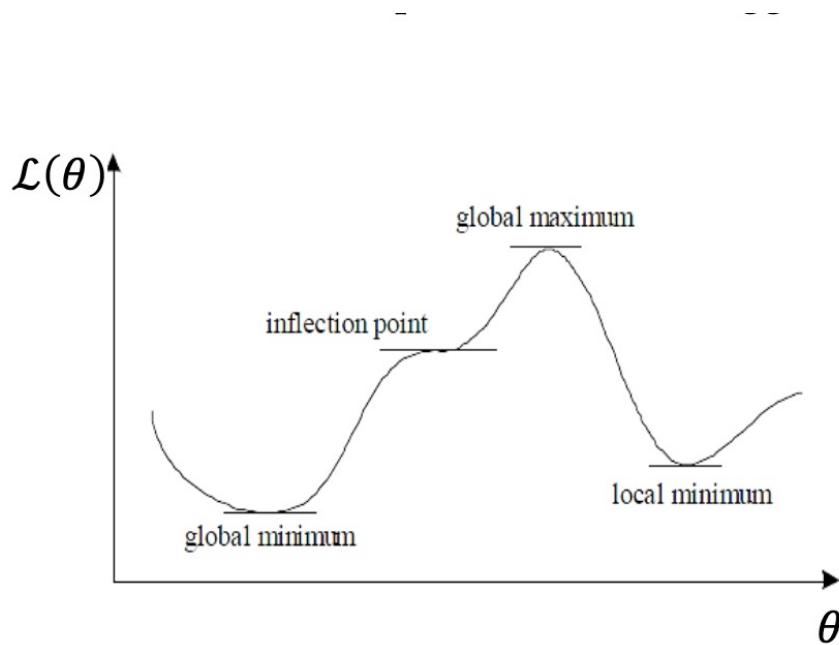
# Algoritmo di discesa del gradiente

- Esempio (continua)



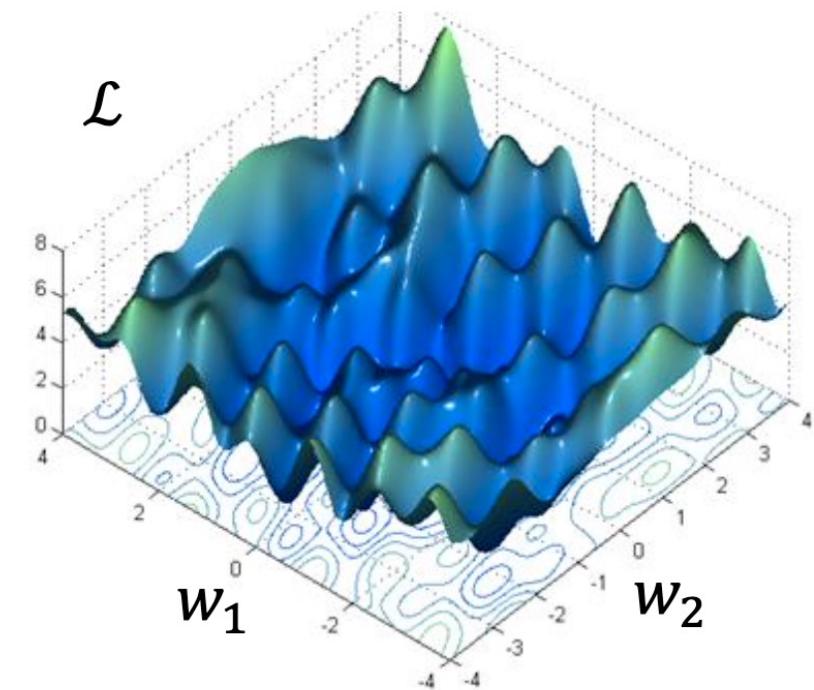
# Algoritmo di discesa del gradiente

- L'algoritmo di discesa del gradiente si ferma quando è raggiunto un **minimo locale** della superficie di perdita
  - GD non garantisce di raggiungere un **livello globale minimo**
  - Tuttavia, l'evidenza empirica suggerisce che GD funziona bene per NN



# Algoritmo di discesa del gradiente

- Per la maggior parte delle attività, la **superficie di perdita** è molto complessa (e non convessa)
- L'inizializzazione casuale in NN risulta in diversi parametri iniziali  $\theta^0$  ogni volta che la NN è allenata
  - La discesa del gradiente può raggiungere minimi diversi ad ogni esecuzione
  - Pertanto, la NN produrrà differenti output
- Inoltre, attualmente non disponiamo di algoritmi che garantiscano il raggiungimento di un **minimo globale per una** funzione di loss arbitraria



# Backpropagazione

---

- Le moderne NN utilizzano il metodo di *backpropagation* per calcolare i gradienti della funzione di perdita  $\nabla \mathcal{L}(\theta) = \partial \mathcal{L} / \partial \theta_i$ 
  - Backpropagation è l'abbreviazione di "backward propagation"
- Per addestrare NN, la **forward propagation** (forward pass) si riferisce al passaggio degli input  $x$  attraverso i livelli nascosti per ottenere gli output del modello (predizioni)  $y$ 
  - E' calcolata la perdita  $\mathcal{L}(y, \hat{y})$
- La **backpropagation** attraversa la rete in ordine inverso, dagli output  $y$  verso gli input  $x$  per calcolare i gradienti della perdita  $\nabla \mathcal{L}(\theta)$ 
  - La chain rule serve per calcolare le derivate parziali della funzione di perdita rispetto ai parametri  $\theta$  nei diversi strati della rete
- Ogni aggiornamento dei parametri del modello  $\theta$  durante l'allenamento effettua un passaggio in avanti e uno indietro (ad es. per un batch di inputs)
- Il calcolo automatico dei gradienti ( **differenziazione automatica** ) è disponibile in tutte le librerie di deep learning
  - Semplifica notevolmente l'implementazione di algoritmi di deep learning, poiché evita di derivare le derivate parziali della funzione di perdita a mano

# Discesa del Gradiente in mini batch

---

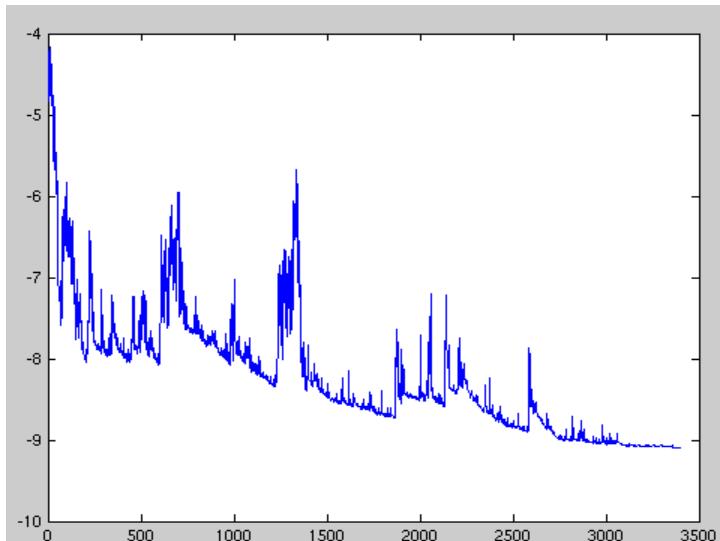
- È dispendioso calcolare la perdita sull'intero set di **dati di training** per eseguire un aggiornamento di un singolo parametro per grandi dataset
  - Ad esempio, ImageNet ha 14 milioni di immagini
  - Pertanto, GD (aka vanilla GD) viene quasi sempre sostituito con mini-batch GD
- *Mini-batch gradient descent*
  - Approccio:
    - Calcola la perdita  $\mathcal{L}(\theta)$  su un mini-batch di immagini, aggiorna i parametri  $\theta$  e ripeti finché non sono usate tutte le immagini
    - All'epoca successiva , mescola i dati di training e ripeti il processo precedente
  - Il mini-batch porta ad un addestramento molto più veloce
  - Tipiche taglie di mini-batch : da 32 a 256 immagini
  - Funziona perché il gradiente di un mini-batch è una buona approssimazione del gradiente dell'intero training set

# Discesa del gradiente stocastico

---

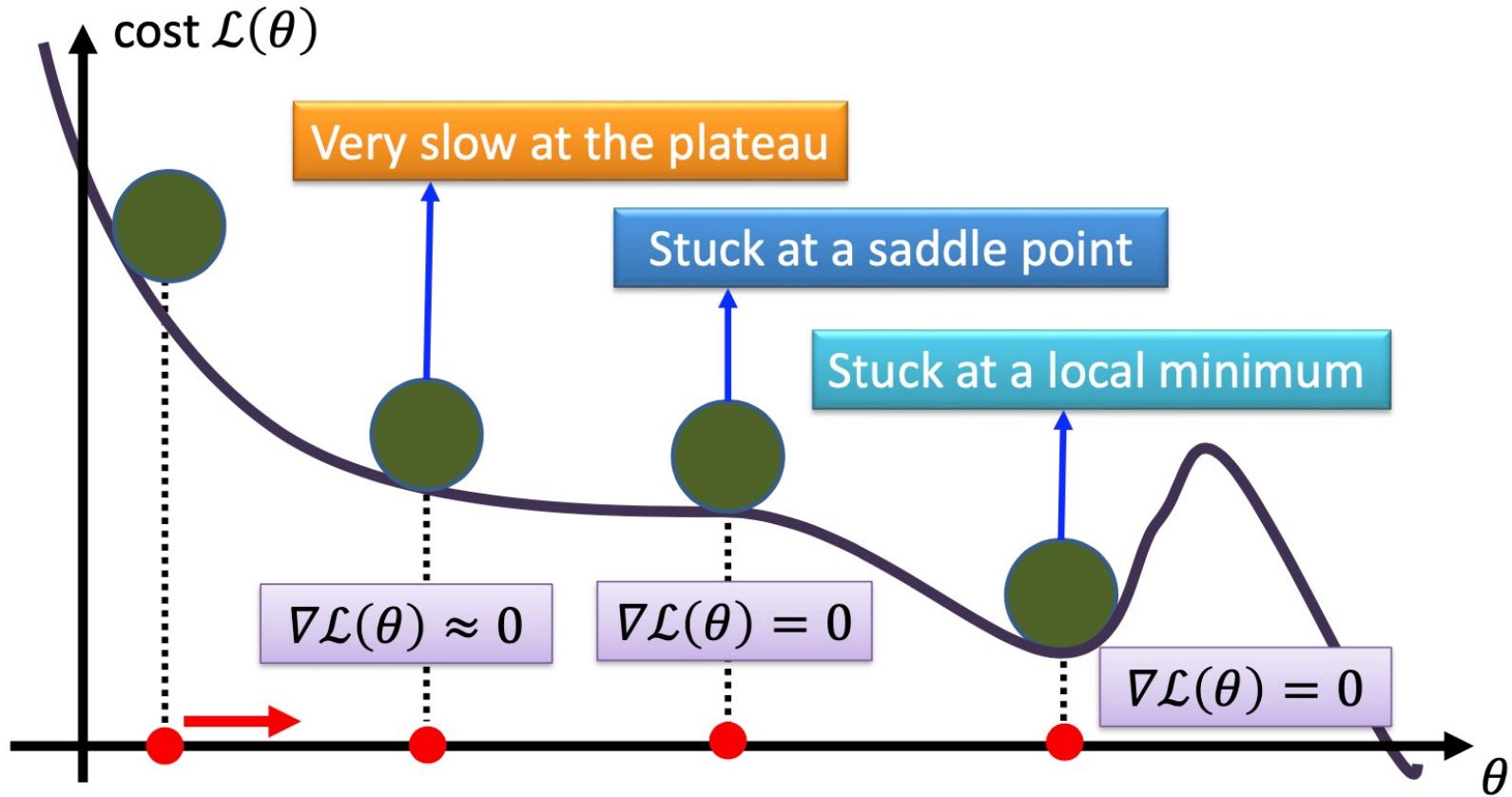
- *Discesa del gradiente stocastico*

- SGD utilizza mini-batch costituiti da un **unico esempio di input**
  - Ad esempio, una immagine di mini-batch
- Sebbene questo metodo sia molto veloce, può causare fluttuazioni significative nella funzione
  - Pertanto, è meno comunemente usato, viene preferito GD mini-batch



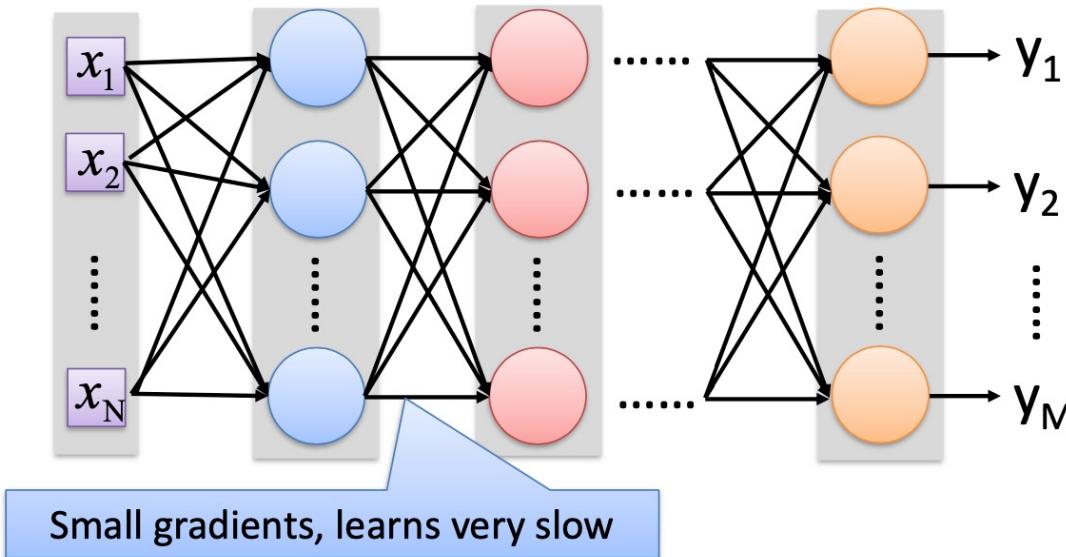
# Problemi con la Discesa del gradiente

- Oltre al problema dei minimi locali, l'algoritmo GD può essere molto lento ai **plateaus**, e può rimanere bloccato in **punti saddle**



# Problema del Gradiente evanescente

- In alcuni casi, durante l'allenamento, i gradienti possono diventare o molto piccoli (gradienti evanescenti) o molto grandi (esplosione del gradiente)
  - Portano ad un aggiornamento molto piccolo o molto grande dei parametri
  - Soluzioni: cambio del tasso di apprendimento, attivazioni ReLU, regolarizzazione, unità LSTM in RNN



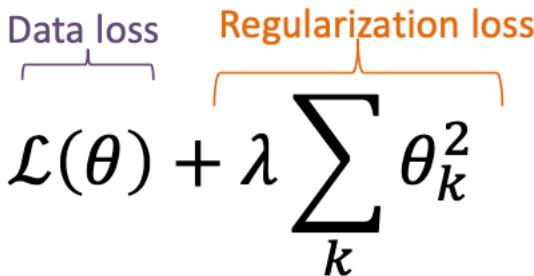
# Regularization: Weight Decay

## $\ell_2$ weight decay

- Alla loss function si aggiunge un termine di regolarizzazione che penalizza i pesi grandi

$$\mathcal{L}_{reg}(\theta) = \mathcal{L}(\theta) + \lambda \sum_k \theta_k^2$$

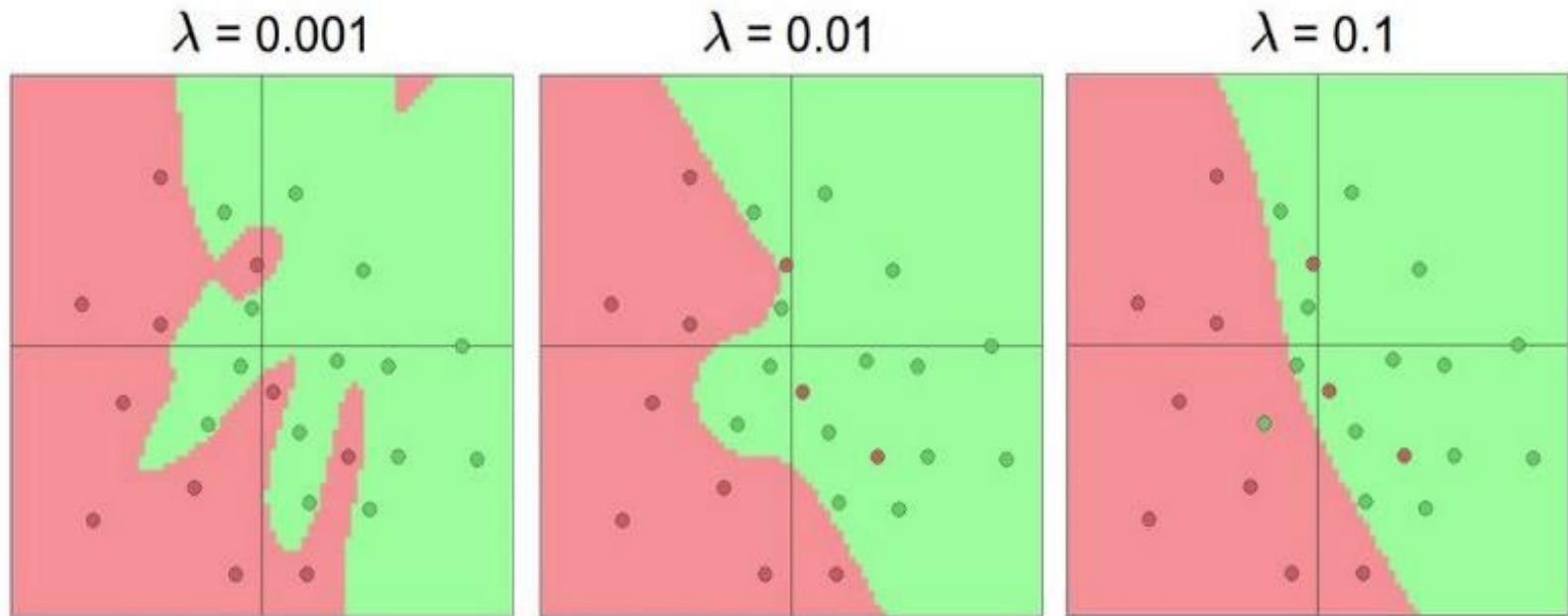
Data loss      Regularization loss



- Per ogni peso della rete, alla loss aggiungiamo il termine di regolarizzazione
  - Durante l'aggiornamento dei parametri in GD, ogni peso viene decaduto linearmente verso zero
- Il **coefficiente di decadimento del peso**  $\lambda$  determina quanto sia dominante la regolarizzazione durante il calcolo del gradiente

# Regolarizzazione: Weight Decay

- Effetto del coefficiente di decadimento  $\lambda$ 
  - Coefficiente di decadimento con peso elevato  $\rightarrow$  penalità per i pesi con valori elevati



# Regolarizzazione: Weight Decay

---

## $\ell_1$ weight decay

- Il termine di regolarizzazione si basa sulla norma  $\ell_1$  dei pesi

$$\mathcal{L}_{reg}(\theta) = \mathcal{L}(\theta) + \lambda \sum_k |\theta_k|$$

- Il decadimento del peso  $\ell_1$  è meno comune con NN
  - Spesso ha prestazioni peggiori di  $\ell_2$
- È anche possibile combinare  $\ell_1$  e  $\ell_2$ 
  - Chiamata **elastic net regularization**

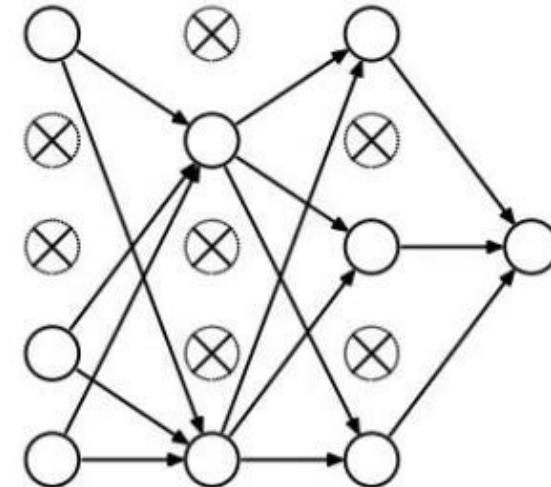
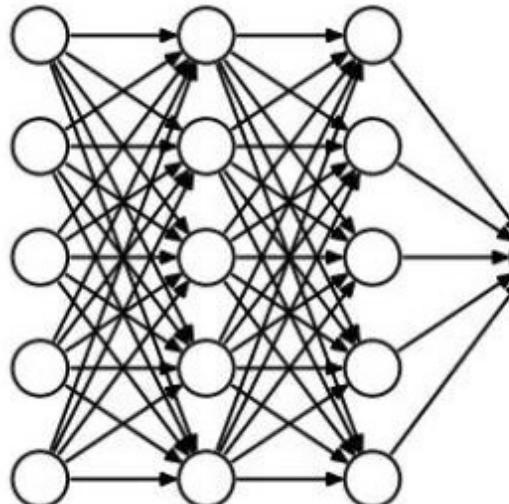
$$\mathcal{L}_{reg}(\theta) = \mathcal{L}(\theta) + \lambda_1 \sum_k |\theta_k| + \lambda_2 \sum_k \theta_k^2$$

$$\mathcal{L}_{reg}(\theta) = \mathcal{L}(\theta) + \lambda_1 \sum_k |\theta_k| + \lambda_2 \sum_k \theta_k^2$$

# Regolarizzazione: Dropout

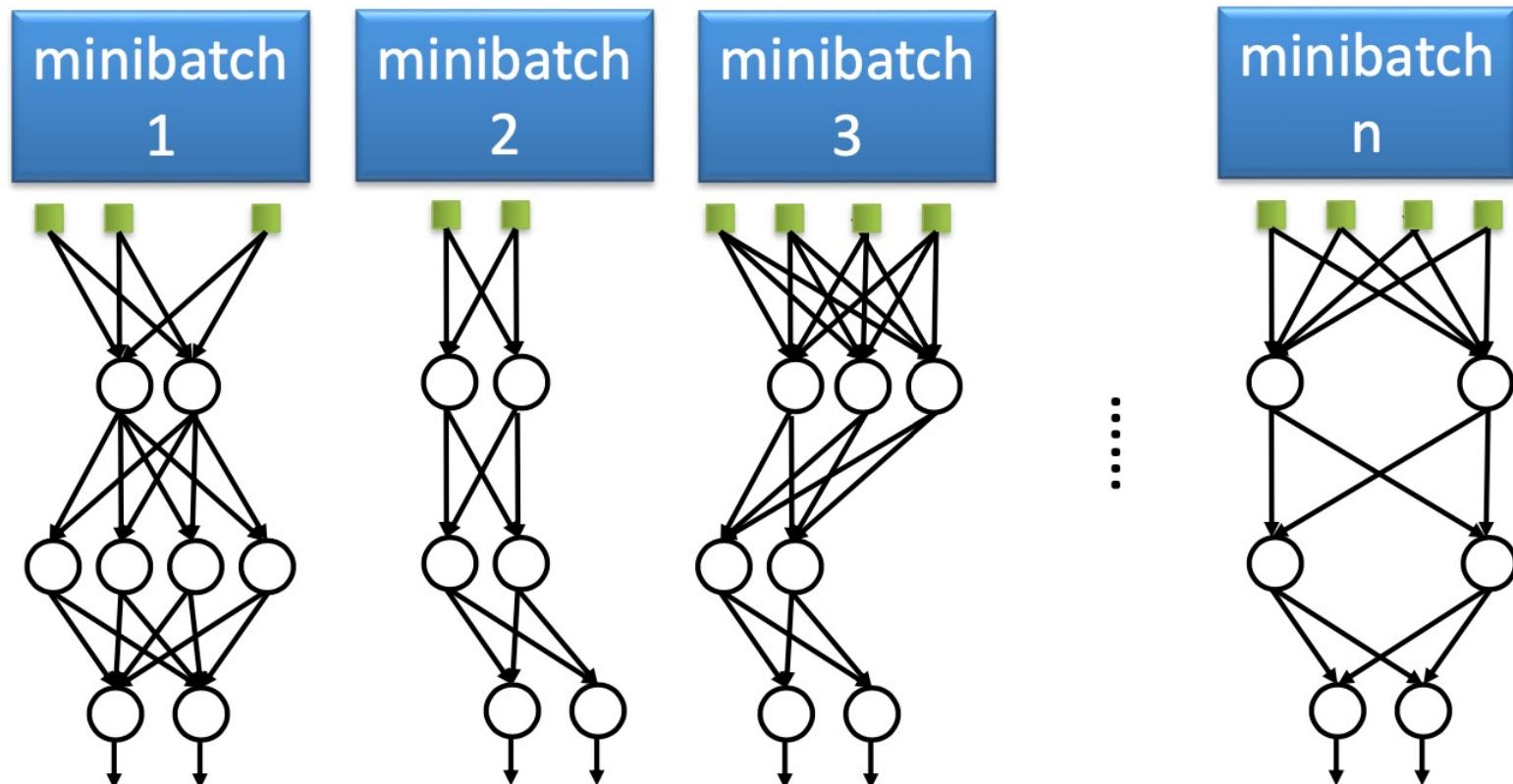
- *Dropout*

- Rilascia casualmente le unità (insieme alle loro connessioni) durante il training
- Ciascuna unità viene spenta con un **tasso di dropout  $p$** , indipendente dalle altre unità
- E' necessario scegliere l' iperparametro  $p$  (tuned)
  - Spesso, tra il 20% e il 50% delle unità sono dropped



# Regolarizzazione: Dropout

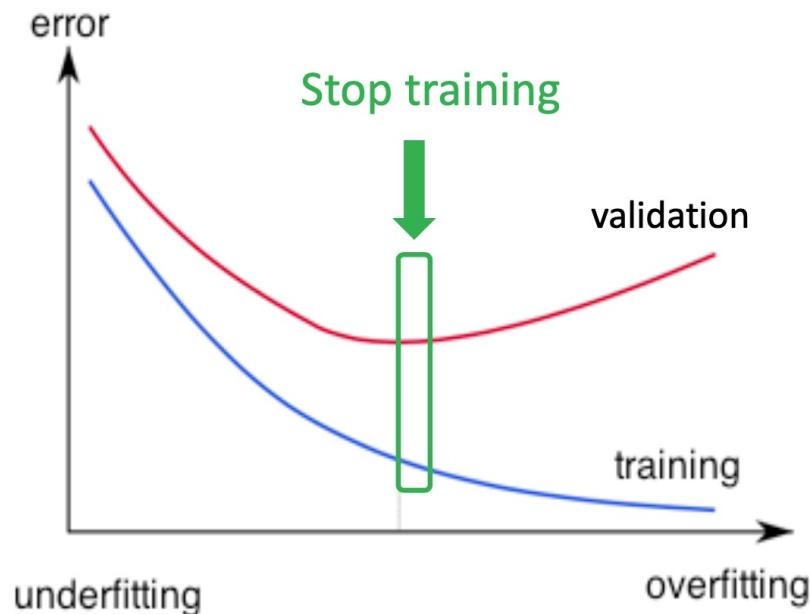
- Dropout è una specie di ensemble learning
  - Usando un mini-batch per addestrare una rete con un'architettura leggermente diversa



# Regolarizzazione: Fermata anticipata

## Early-stopping

- Durante l'addestramento del modello, utilizzare una **set di validazione**
- Stop quando l'accuratezza (o la perdita) della convalida non è migliorata dopo  $n$  epocha
  - il parametro  $n$  viene chiamato **patience**



# Batch Normalization

---

- *I livelli di normalizzazione batch* agiscono in modo simile alla pre-elaborazione dei dati menzionati prima

- Calcolano la media  $\mu$  e la varianza  $\sigma$  di un batch di dati di input e normalizzano i dati  $x$  su una media zero e una varianza unitaria

$$\text{I.e., } \hat{x} = \frac{x - \mu}{\sigma}$$

- I layers BatchNorm alleviano i problemi di una corretta inizializzazione dei parametri e iperparametri
  - Portano ad convergenza più rapida del training, consentono un tasso di apprendimento più ampio
- livelli BatchNorm vengono inseriti immediatamente dopo i livelli convoluzionali o i livelli completamente connessi e prima degli strati di attivazione
  - Sono molto comuni con le NN convoluzionali

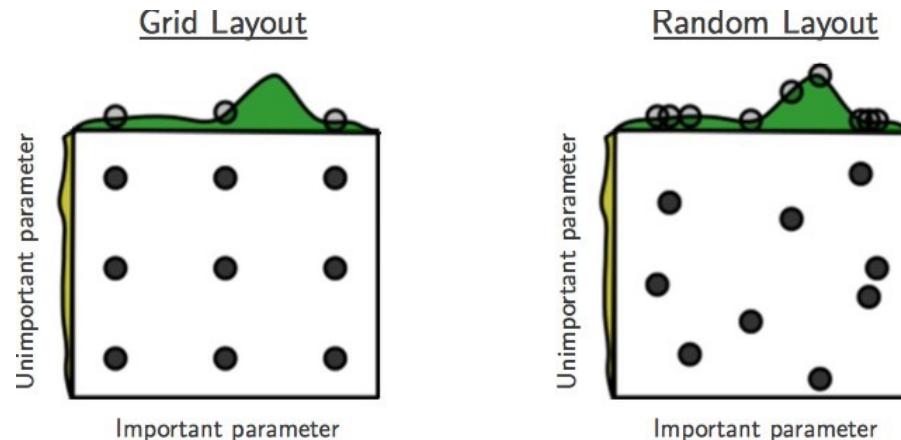
# Hyper-parameter Tuning

---

- Il training di NN può comportare il settaggio di molti *iperparametri*
- Gli iperparametri più comuni sono:
  - Il numero di layers ed il numero di neuroni per strato
  - Tasso iniziale del learning rate
  - Tasso di decadimento del learning rate (ad es costante)
  - Genere di ottimizzatore
- Altri iperparametri possono includere:
  - Parametri di regolarizzazione (penalità  $\ell_2$ , dropout rate)
  - Batch size
  - Funzioni di attivazione
  - Funzione di loss
- L'ottimizzazione degli iperparametri può richiedere molto tempo per NN grandi

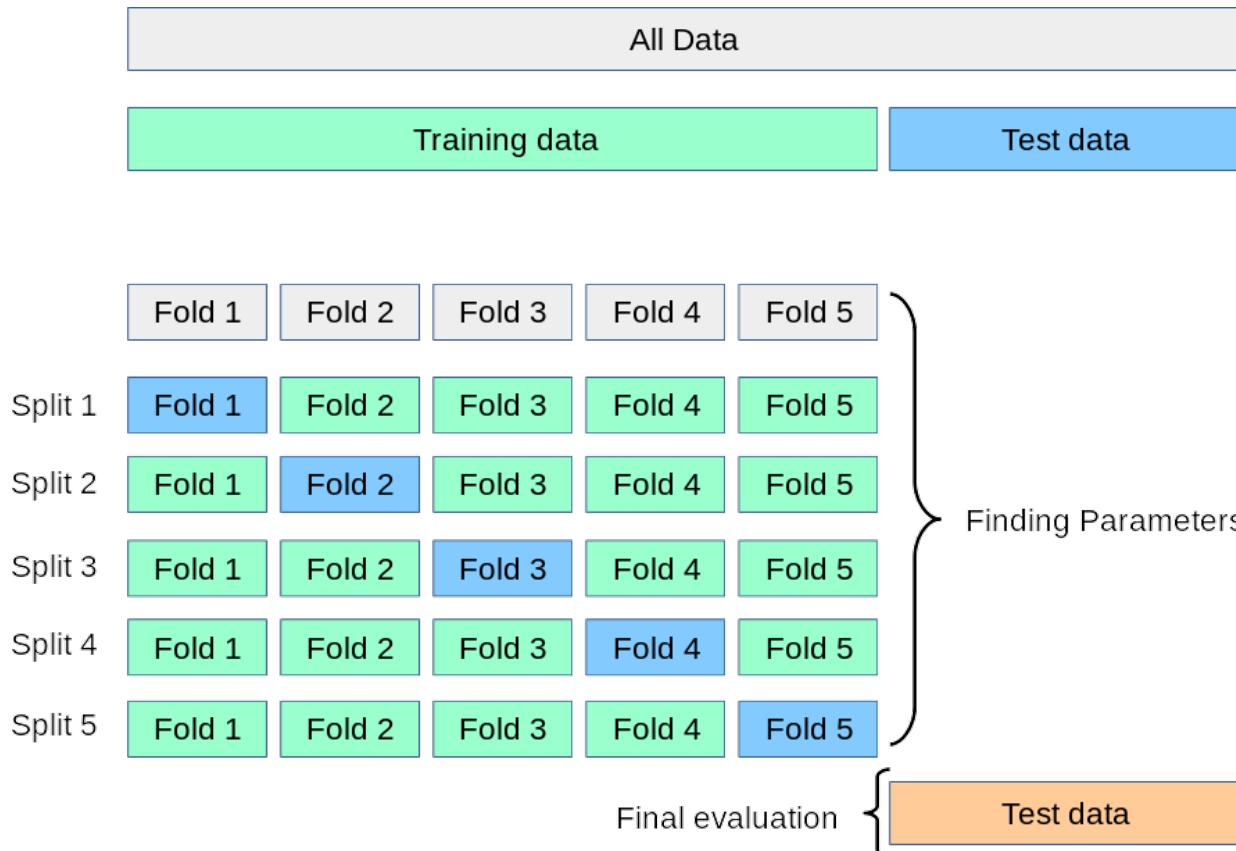
# Hyper-parameter Tuning

- Grid search
  - Controllare tutti i valori in un intervallo con un valore di step
- Random search
  - Campiona casualmente i valori per il parametro
  - Spesso preferito alla ricerca nella griglia
- Ottimizzazione bayesiana degli iperparametri
  - È un'area attiva di ricerca



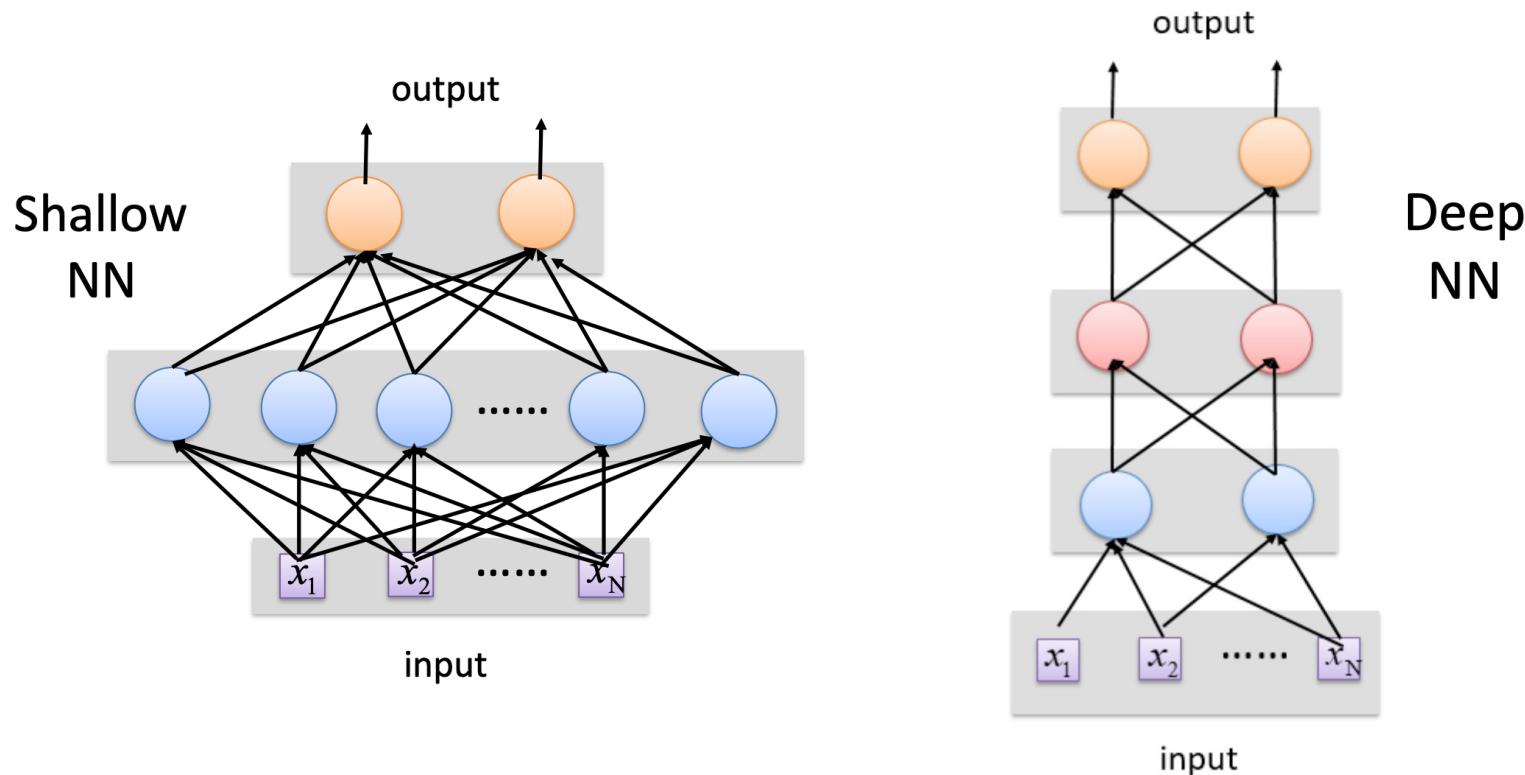
# *k-Fold Cross-Validation*

- Illustrazione di un 5-fold CV



# Deep vs Shallow Networks

- Le reti più profonde funzionano meglio di quelle poco profonde reti
  - Ma solo fino a un certo limite: dopo un certo numero di strati, le prestazioni delle reti più profonde si stabilizzano



# Reti neurali convoluzionali (CNN)

- *Le reti neurali convoluzionali* (CNN) sono state progettate principalmente per immagini
- Le CNN utilizzano **un operatore convoluzionale** per estrarre le feature
  - Consente la **condivisione dei parametri**
  - Efficiente da allenare
  - Hanno **meno parametri** rispetto a NN con strati completamente connessi
- Le CNN sono **robuste per le traduzioni spaziali** di oggetti in immagini
- Un filtro convoluzionale scorre attraverso l'immagine

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

Input matrix

1	0	1
0	1	0
1	0	1

Convolutional  
3x3 filter



1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

Image

4		

Convolved  
Feature

# Reti neurali convoluzionali (CNN)

- Quando i filtri convoluzionali vengono scansionati sull'immagine, essi catturano features utili
  - Ad esempio, rilevamento dei bordi

Filtro



$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$



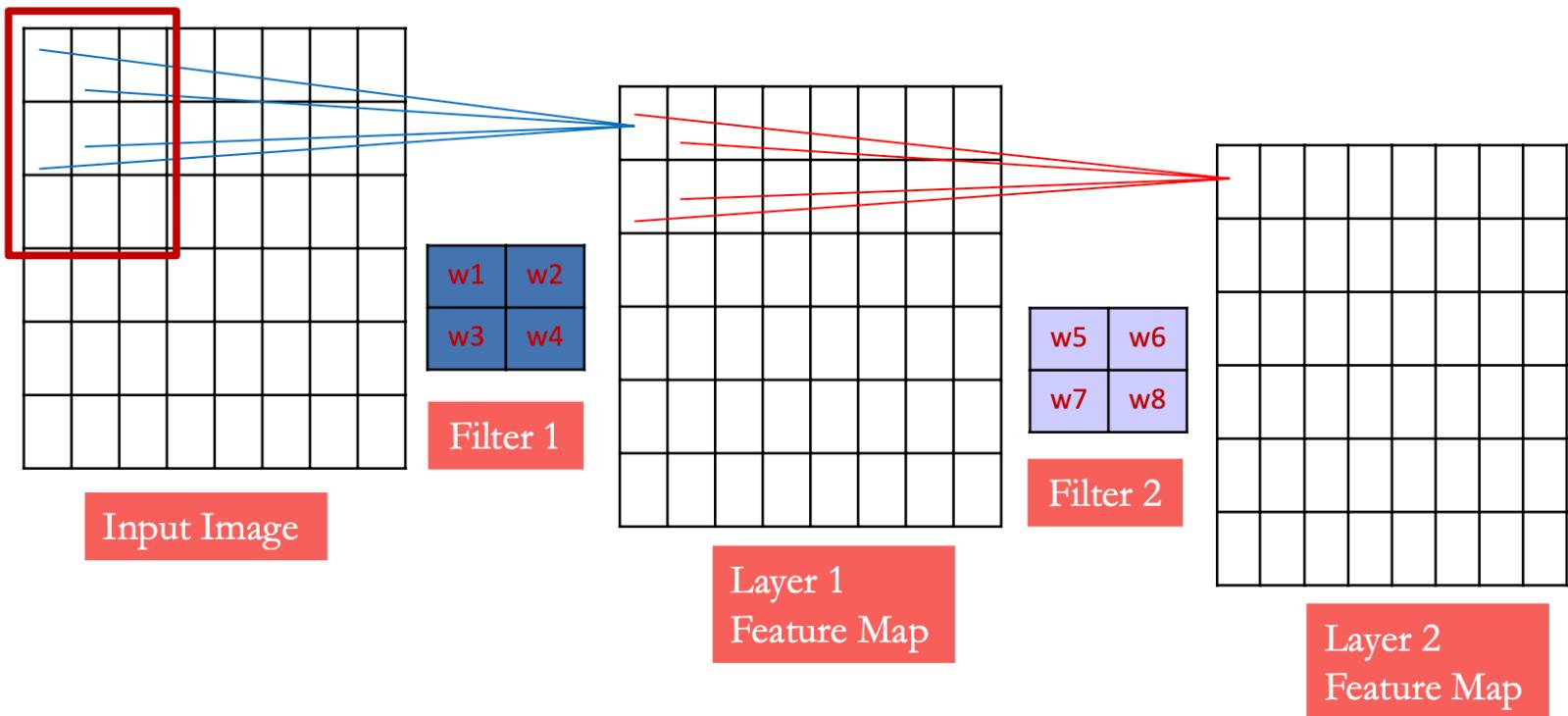
Immagine Input



Immagine Convoluta

# Reti neurali convoluzionali (CNN)

- Nelle CNN, le unità nascoste in un livello sono collegate solo a una piccola regione del strato precedente (chiamato **campo ricettivo locale**)
  - La profondità di ciascuna **feature map** corrisponde al numero di filtri convoluzionali utilizzati in ciascuna strato



# Reti neurali convoluzionali (CNN)

- *Max pooling*: riporta l'output massimo all'interno di un rettangolo (vicinato)
- *Average pooling*: riporta l'output medio di un rettangolo vicinato
- livelli di pooling riducono la dimensione spaziale delle feature map
  - Ridurre il numero di parametri, prevenire overfitting

MaxPool with a  $2 \times 2$  filter with stride of 2

1	3	5	3
4	2	3	1
3	1	1	3
0	1	0	4

Input Matrix

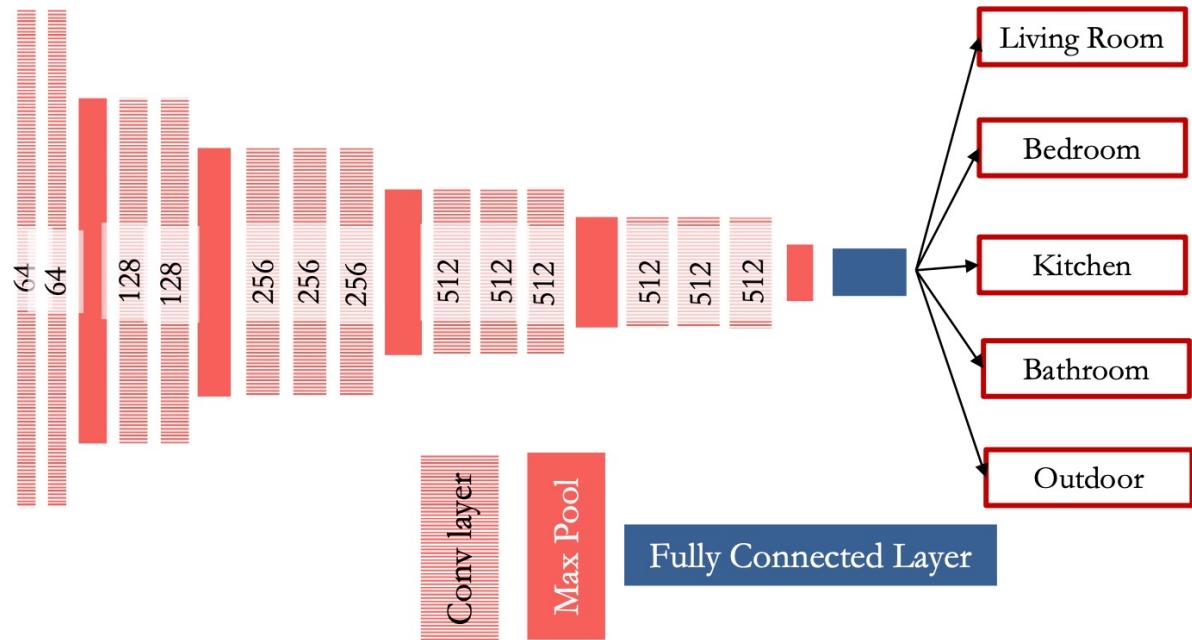
4	5
3	4

Output Matrix

# Reti neurali convoluzionali (CNN)

- Architettura

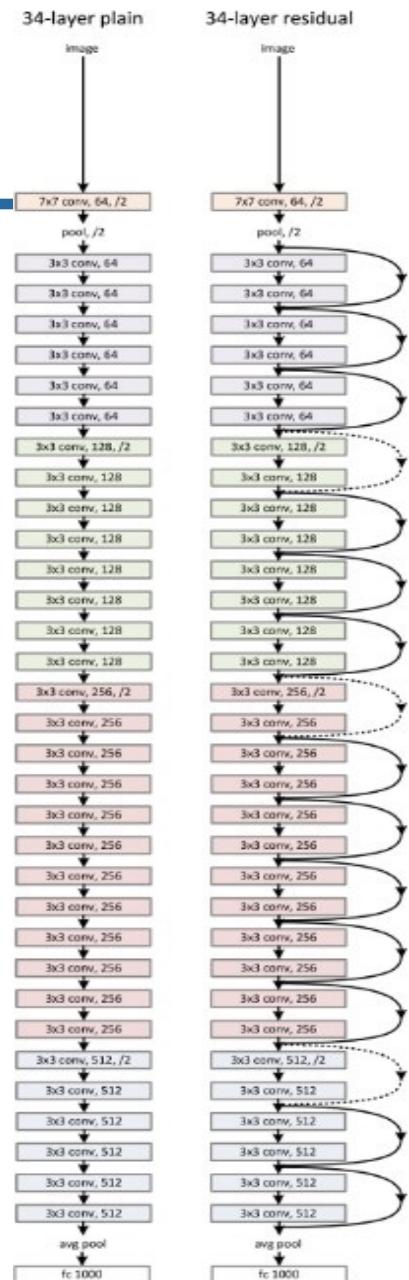
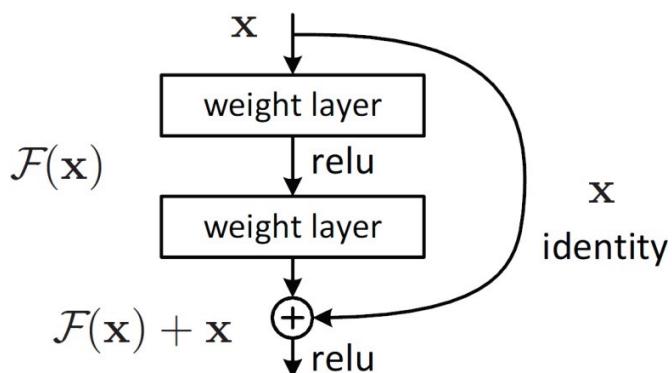
- Dopo 2 livelli convoluzionali, un livello di max-pooling riduce la dimensione delle feature map (in genere di 2)
- Uno strato completamente convoluzionale e uno strato softmax vengono aggiunti per ultimi per eseguire la classificazione



# CNN Residuali

- **Reti residue** (ResNet)

- Introducono il **salto connessione «identità»**
  - Uno strato dovrebbe perturbare non sostituire il precedente
  - Layer di input vengono propagati e aggiunti al livello output
  - Mitiga il problema del vanishing gradient durante il training
  - Consente un addestramento molto profondo NN (con oltre 1.000 strati)
- Esistono diverse varianti ResNet : 18, 34, 50, 101, 152 e 200 strati
- Sono usati come modelli base di altre NN dello stato dell'arte
  - Altri modelli simili: ResNeXT, Rete densa



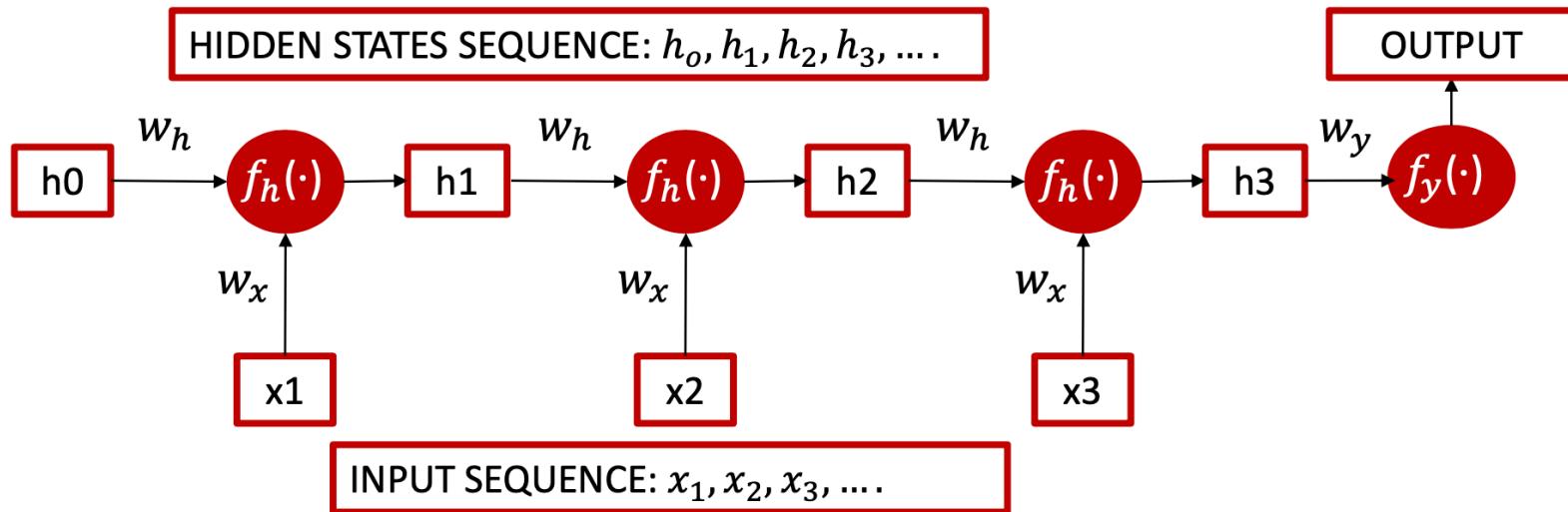
# Reti neurali ricorrenti (RNN)

---

- *Le Recurrent NN* vengono utilizzate per modellare **dati sequenziali** e dati con input e output di lunghezza variabile
  - Video, testo, parlato, sequenze di DNA, dati di scheletri umani
- Le RNN introducono connessioni ricorrenti tra i neuroni
  - Ciò consente di elaborare i dati sequenziali un elemento alla volta passando selettivamente le informazioni attraverso una sequenza
  - La memoria degli input precedenti viene archiviata nello stato del modello ed influenzano le predizioni del modello
  - Possono catturare le correlazioni in dati sequenziali
- Le RNN usano la **backpropagation-through-time** per il training
- Le RNN sono più sensibili al problema del gradiente evanescente rispetto alle CNN

# Reti neurali ricorrenti (RNN)

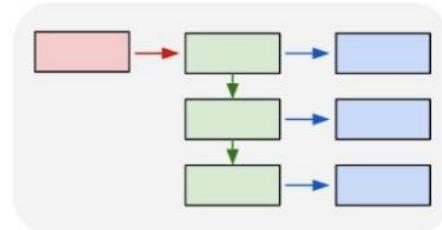
- RNN usa lo stesso set di pesi  $w_h$  e  $w_x$  **in tutti i passi temporali**
  - Viene appresa una sequenza di **hidden state**  $\{h_0, h_1, h_2, h_3, \dots\}$  che rappresenta la memoria della rete
  - Lo stato hidden al passo  $t$ ,  $h(t)$ , viene calcolato in base al precedente hidden state  $h(t-1)$  e l'input al passo corrente  $x(t)$ , cioè  $h(t) = f_h(w_h * h(t-1) + w_x * x(t))$
  - La funzione  $f_h(\cdot)$  è una funzione di attivazione non lineare, e.g., ReLU o tanh
- RNN nel tempo



# Reti neurali ricorrenti (RNN)

- Le RNN possono avere uno dei tanti input e uno dei tanti output

RNN



Applicazione

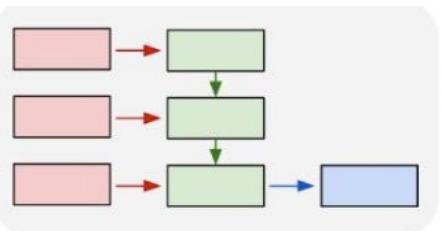
Image  
Captioning



Input

Output

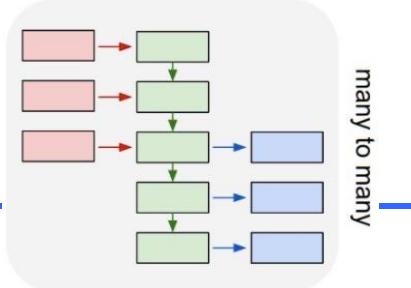
Una persona in sella a una moto strada



Sentiment  
Analysis

Awesome movie.  
Highly recommended.

Positivo



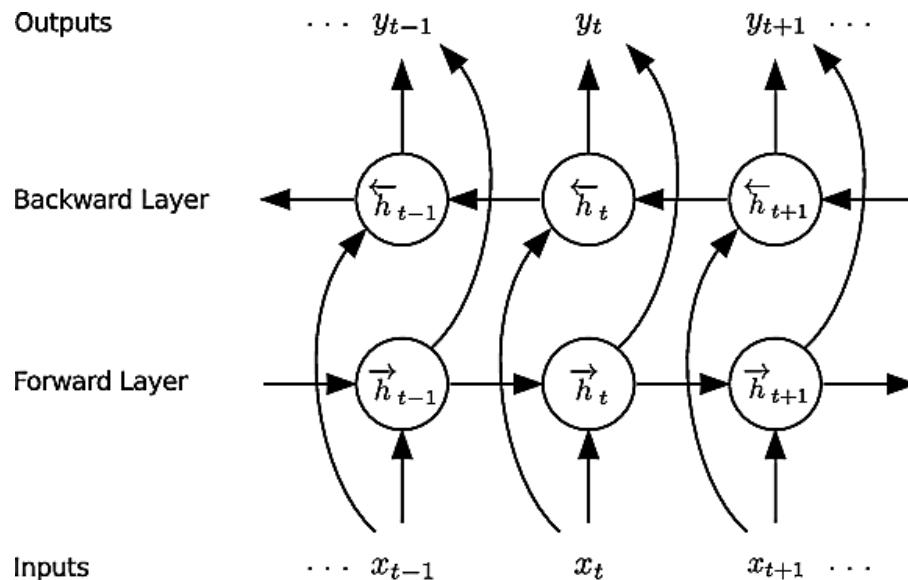
Machine  
Translation

Happy Diwali

शुभ दीपावली

# Bidirezionale RNN

- *Le RNN bidirezionali* incorporano passaggi sia in avanti che all'indietro attraverso dati sequenziali
  - L'output può dipendere non solo dagli elementi precedenti nella sequenza, ma anche dagli elementi futuri nella sequenza
  - Assomiglia a due RNN impilate una sull'altra



$$\vec{h}_t = \sigma(\vec{W}^{(hh)}\vec{h}_{t-1} + \vec{W}^{(hx)}x_t)$$

$$\overleftarrow{h}_t = \sigma(\overleftarrow{W}^{(hh)}\overleftarrow{h}_{t+1} + \overleftarrow{W}^{(hx)}x_t)$$

$$y_t = f([\vec{h}_t; \overleftarrow{h}_t])$$

Produce sia elementi passati che futuri

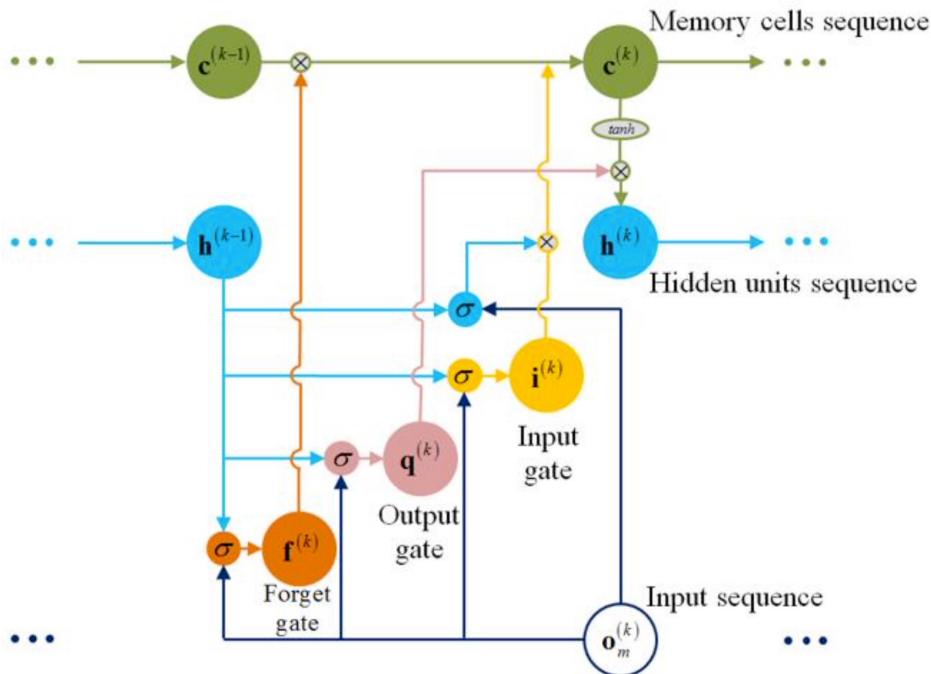
# LSTM Reti

---

- *Long Short-Term Memory* (LSTM) sono una variante di RNN
- LSTM attenua il problema della fuga del gradiente/esplosione
  - Soluzione: una *Memory Cell*, aggiornata ad ogni passaggio della sequenza
- Tre porte controllano il flusso di informazioni da e verso la *Memory Cell*
  - *Input Gate*: protegge il passo corrente dagli input irrilevanti
  - *Output Gate*: impedisce al passo corrente di trasmettere informazioni irrilevanti ai passi successivi
  - *Forget Gate*: limita le informazioni passate da una cella alla prossima
- La maggior parte dei modelli RNN moderni utilizza unità LSTM o altri tipi più avanzati di unità ricorrenti (ad esempio, unità GRU)

# Reti LSTM

- Cella LSTM
  - Input gate, output gate, forget gate, memory cell
  - LSTM può apprendere correlazioni a lungo termine all'interno delle sequenze di dati



$$\begin{aligned}\mathbf{i}^{(k)} &= \sigma(W_{oi}o_m^{(k)} + W_{hi}h^{(k-1)} + b_i) \\ \mathbf{f}^{(k)} &= \sigma(W_{of}o_m^{(k)} + W_{hf}h^{(k-1)} + b_f) \\ \mathbf{q}^{(k)} &= \sigma(W_{oq}o_m^{(k)} + W_{hq}h^{(k-1)} + b_q) \\ \mathbf{c}^{(k)} &= \mathbf{f}^{(k)}\mathbf{c}^{(k-1)} + \mathbf{i}^{(k)}\sigma(W_{oc}o_m^{(k)} + W_{hc}h^{(k-1)} + b_c) \\ \mathbf{h}^{(k)} &= \mathbf{q}^{(k)}\tanh(\mathbf{c}^{(k)})\end{aligned}$$