Deep Learning

Daniele Avolio

September 26, 2023

Contents

1 Introduzione

Appunti

2 Deep Learning 101

In questo corso affronteremo diverse tematiche, il che può sembrare assurdo se ci si pensa.

- Classificazione (binaria, cioè sì o no)
- Multi-class classification (non più binaria)
- Regressione (il guessing viene fatto su un valore numerico)
- Gestione di immagini e riconoscimento
- Serie numeriche (predizioni di mercato e trend)
- Classificazione di testi

2.1 Architetture e strumenti nel deep learning

- Autoencoder
  - Tutti i possibili tipi
  - Qui si fa anche **Clustering** e **Anomaly detection**
- Architetture generative
  - Tutti i possibili tipi
- XAI: Explainable AI

2.2 Libri utili

- "Deep Learning in Python"
- "Tensorflow tutorial"

2.3 Strumenti che useremo

- Tensorflow
  - High-level più di altri
- Keras
  - High-level API basato su Tensorflow
  - Ci saranno cose che non possiamo fare con Keras perché è troppo ad alto livello

2.4 Schema generale di un problema di deep learning

Abbiamo delle coppie $(x_0, y_0), (x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ dove x_i è un vettore di features e y_i è un valore numerico (regressione) o una classe (classificazione).

$$y_i = f(x_i)$$

Non conosciamo la funzione f, quindi dobbiamo impararla.

$$y = \alpha x + \beta$$

Con una rete neurale puoi approssimare praticamente qualsiasi funzione.

Una rete neurale permette di collegare un input di dati a una funzione di output.

Abbiamo diversi tipi di reti neurali a seconda del tipo di problema che vogliamo risolvere. È importante essere in grado di selezionare l'architettura giusta per risolvere il problema.

Definiamo alcuni concetti che useremo:

- N: rete neurale
- w: valori dei pesi della rete neurale
- f: funzione di output della rete neurale

$$f \in N(w)$$

2.5 Perché si usa il termine "Tensore"?

Un *tensore* non è altro che una matrice.

- 0D tensor: scalar
- 1D tensor: vector
- 2D tensor: matrix
- 3D tensor: tensor

2.6 AI vs DL

- AI: è un ampio insieme di tecniche per risolvere problemi che richiedono "intelligenza".
  - Esempio: Stockfish, un programma che gioca a scacchi.
- DL: È un sottoinsieme di AI che si concentra sull'*astrazione*.
  - L'astrazione consiste nel fornire una funzione che traduce dati di input in dati di output senza conoscere la funzione stessa.
  - È un approccio induttivo: fornisci un input e ti aspetti un output, senza conoscere la funzione che li collega.
  - Questo è completamente diverso dall'AI basata sulla logica.

3 Introduzione alle Reti Neurali

3.1 Il modello di McCulloch-Pitts

Un modello pensato dai due tizi qui presenti,

- Warren McCulloch
- Walter Pitts

Era formato da:

- Un insieme di neuroni
- Un insieme di connessioni tra i neuroni
- Un insieme di pesi associati alle connessioni
- Una funzione di attivazione
- Una funzione di output

Quindi immaginiamo $x_1, x_2, \ldots, x_n$ che vengono dati come input, e quello che viene fuori è un valore di $y \in \{0, 1\}$. Questo è un modello di classificazione binaria.

In soldoni, in deep learning si usa un vettore di numeri per tirare fuori un altro vettore di numeri.

Nella maggior parte del tempo, però, non lavoriamo solamente con i dati.

- immagini
- Testo
- Audio
- ...

Non ci sono concetti di foto, video, immagini. L'unico concetto che esiste è quello dei numeri.

3.2 Modello di Rosenblatt

Qui il modello è leggermente diverso. Gli elementi in questo modello sono i seguenti:

- Valori di input: x_i
- Funzione di attivazione: ϕ
- Pesi degli archi: w_i
- Bias: b

$$h(x|w,b) = h(\sum_{i=1}^{l} w_i \cdot x_i - b) = h(\sum_{i=1}^{l} w_i \cdot x_i) = sign(w^T x) \tag{1}$$

Nota: Quando il **bias** non viene specificato, allora si assume che sia 0.

I pesi w_i sono collegati archi che vanno da x_i al prossimo neurone. Sia il valore di input che il peso sono **numeri reali**. Non sono lo stesso valore, hanno solamente il formato di *reale* che è uguale tra loro. Ciò che viene fatto è solamente la somma della prodotto tra ogni peso w_i e x_i **meno** il bias.

La funzione di attivazione: Dipende. Ogni funzione che ha *2 stati* va bene per noi. Una funzione di attivazione può essere una qualsiasi che in un punto ha valore 1 e in un altro ha valore -1.

3.3 Esempio con 2 neuroni

.

Immaginiamo di avere un piano cartesiano con una retta che interseca in 2 punti.

$$sign(w_1 \cdot x_1 + w_2 \cdot x_2) \tag{2}$$

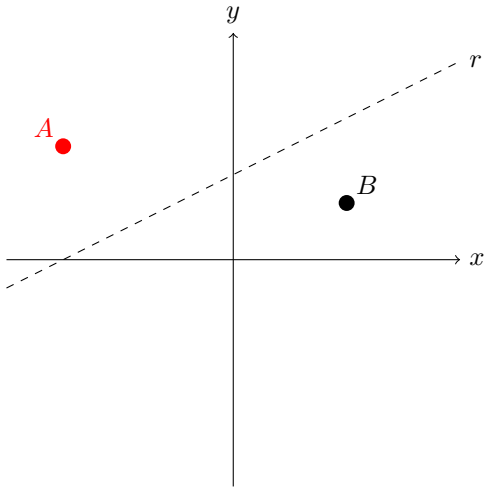
Il parametro della funzione non è altro che **l'equazione di una retta.**

In particolare, se consideriamo la reta che separa gli spazi del piano, vediamo che la retta è **capace di separare 2 punti nel piano.**

Cosa abbiamo:

- Un neurone
- 2 valori in input
- 2 pesi

Con la funzione di attivazione **sign** si avrà come output una retta che separa



dei punti.

Nota: Quando è utile avere un valore output che non è una separazione di punti in un piano? Nel caso della **regressione**.

Oltre la funzione sign:

- Funzione di attivazione lineare
- Funzione di attivazione sigmoid
- Funzione di attivazione Tanh

3.4 Rappresentazione le funzioni logiche

:

3.4.1 AND

Immaginiamo di avere:

- $x_1, x_2 \in \{0, 1\}$
- Bias= -30
- Funzione di attivazione: Logistica o Sigmoid

Come facciamo a rappresentare un AND?

$$h(x) = g(-30 + 20x_1 + 20x_2) \tag{3}$$

x_1	x_2	$h(x)$
0	0	1
0	1	0
1	0	0
1	1	1

Lo stesso ragionamento vale per:

- OR
- NOT
- (NOT x_1) AND (NOT x_2)

3.4.2 Il problema dello XOR

Il percettrone non può imparare regioni che non sono linearmente separabili.

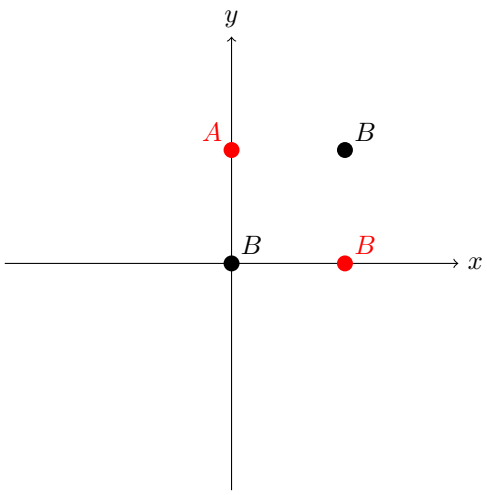


Figure 1: XOR non possibile con 1 percettrone

Come vediamo qui non possoamo tracciare una retta per dividere i due punti. In questo caso ci serve una funzione **non lineare**.

La soluzione a questo problema è **aggiungere LAYER** alla rete neurale. Aggiungere un layer significa aggiungere un neurone successivamente ad un altro.

Maggiore è il numero di layer, maggiore diventa la potenza espressiva della rete. Praticamente possiamo catturare qualsiasi cosa aggiungendo layer alla rete. Questo è **il vero potere del deep learning**.

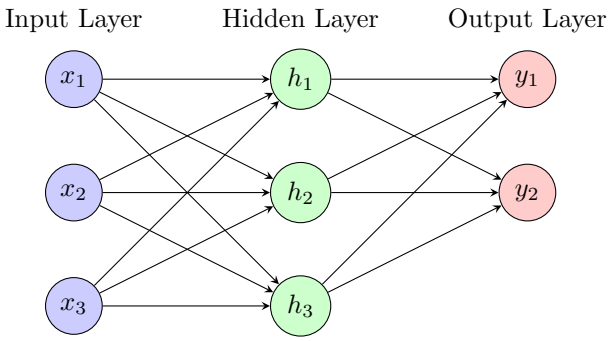


Figure 2: Neural Network con 1 hidden layer

3.5 Gli ingredienti di una rete neurale

3.5.1 Il grafo g

Un grafo $g = \{N, E\}$ è un grafo diretto pesato con label.

Ogni nodo $i \in N$ viene chiamato **neurone** o **percettrone**. Per ogni nodo ci sono 2 targhette:

- Un valore a_i che viene chiamato **attivazione**
- Una funzione di attivazione f_i che viene applicata all'attivazione, che produce un output z_i

Ogni arco $e = \{j \in N \to i \in N\} \in E$ associato con un peso $w_{j,i}$.

Ogni nodo i è anche coinvolto con un arco speciale, con un nodo fantasma, che viene chiamato **bias** b_i.

Nota: $z_i = f_i(a_i)$. E $a_i = b_i + \sum_{j:j\to i\in E} w_{j,i} z_j$

La combinazione dei neuroni connessi costruisce il grafo. I nodi che condividono gli stessi input sono raggruppati in **layers**.

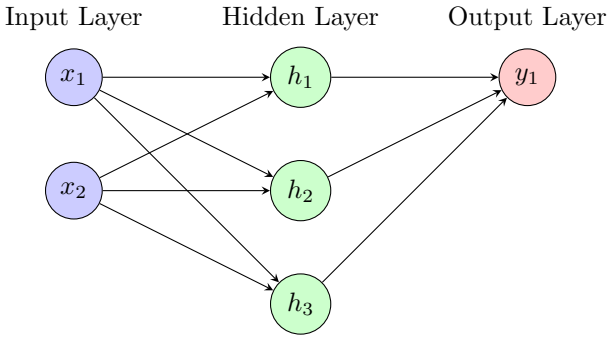


Figure 3: Neural Network con 1 hidden layer

Il risultato finale dipende da tutti i parametri del grafo, e anche dal tipo di funzione di attivazione che viene usata.

> Ma come facciamo a scegliere il corretto valore dei parametri del grafo?

3.5.2 La funzione di loss

Il grafo è un operatore algebrico non lineare: $g(\vec{x}|W, B)$. In questo grafo non sappiamo il valore di **B e W**. La fase di apprendimenti di una rete consiste nel trovare il **migliore** valore per W e per B. Ma cosa intendiamo con **migliore?**.

Formalmente, l'unico modo che abbiamo per definire la conezione di migliore, è quello di *approssimare al meglio la funzione che vogliamo trovare.*

Consideriamo il valore di una funzione F su x_i e il vero valore di y_i. Noi vogliamo minimizzare la differenza tra $F(x_i)$ e y_i.

$$\min_{W,B} \frac{1}{n} \sum_{i=1}^{n} loss(\vec{y_i}, g(x_i|W,B)) \tag{4}$$

con:

- $loss(\vec{y_i}, g(x_i|W,B))$ è una funzione che misura la differenza tra y_i e $g(x_i|W,B)$
- n è il numero di esempi
- W e B sono i parametri del grafo
- $g(x_i|W,B)$ è il valore di output del grafo
- $\vec{y_i}$ è il valore di output vero

La funzione di **loss** dipende dal tipo di task che dobbiamo svolgere.

3.5.3 Funzione di loss per Regressione

Mean Absolute Error:

$$\frac{1}{n} \sum_{i=1}^{n} |y_i - g(\vec{x_i}|W,B)| \tag{5}$$

- Considera tutti gli errori con lo stesso peso
- **Non è differenziabile in 0**

Mean Squared Error:

$$\frac{1}{n} \sum_{i=1}^{n} (y_i - g(\vec{x_i}|W,B))^2 \tag{6}$$

- Gli errori più grandi hanno un peso maggiore
- **È differenziabile in 0**
- **È più sensibile agli outliers**

Domanda: quale si usa tra le due? Dall'approccio greedy, **usa entrambe**.
Esistono anche altre funzioni:

- Smooth Absolute Error
- Huber Loss

3.5.4 Funzione di loss per classificazione

Binary Cross Entropy [BCE] Viene usata se $y_i \in \{0,1\}, g(\vec{x}|W,B) \in [0,1]$.

$$BCE = -\frac{1}{n}\sum_{i=1}^{n} y_i \log(g(\vec{x_i}|W,B)) - (1-y_i)\log(1-g(\vec{x_i}|W,B)) \tag{7}$$

Categorical Cross Entropy [CCE]

$$CCE = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{m} y_{i,j} \log(g(\vec{x_i}|W,B))_j \tag{8}$$

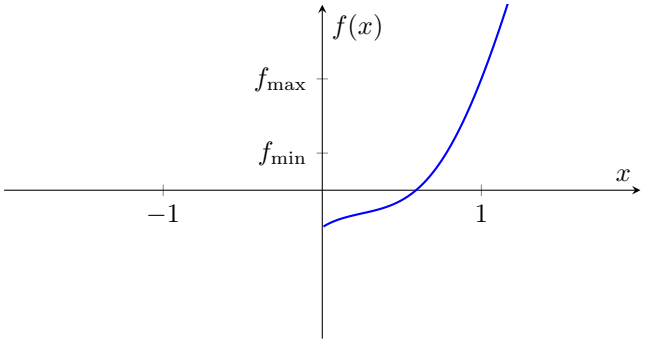
3.6 L'ottimizzatore o

Come risolviamo il problema di:

$$\min_{W,B} \frac{1}{n}\sum_{i=1}^{n} loss(\vec{y_i}, g(\vec{x_i}|W,B)) \tag{9}$$

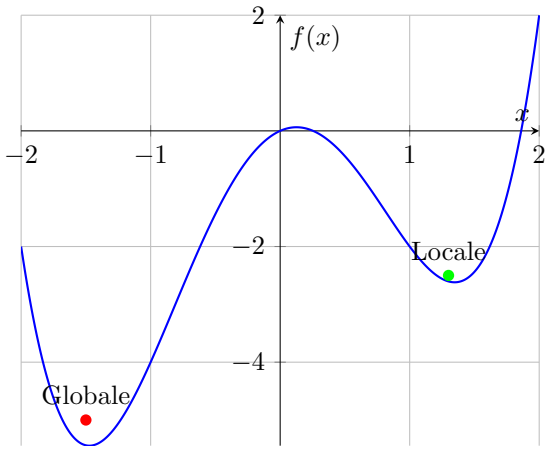
Il problema lo risolviamo calcolando il **gradiante** della funzione loss, lo poniamo uugale a 0 e controlliamo se siamo in un punto di **massimo, minimo o punto di sella**.

$$\frac{\partial loss}{\partial W} = 0 \tag{10}$$



Abbiamo bisogno di un metodo iterativo per trovare una soluzione. Ci vuole un'**euristica**. Tipicamente, siamo soddisfatti di un **minimo locale**, e si utilizza, appunto, il **metodo di discesa del gradiente**.

Differenza minimo locale e globale



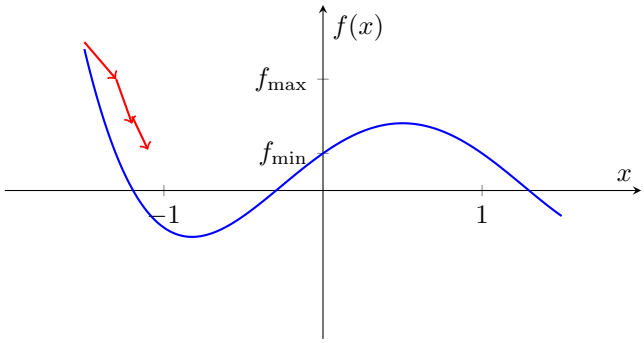
3.6.1 Il metodo di discesa del gradiente

Sia $F(\vec{x})$ una funzione differenziabile.

- $F(\vec{x})$ decresce più veloce nella direzione del gradiente negativo
- $F(\vec{x})$ cresce più veloce nella direzione opposta al gradiente

$$a^{new} = a^{old} - \eta \cdot \nabla F(a^{old}) \tag{11}$$

Il parametro η viene chiamato **learning rate** e determina il comportamento del'ottimizzazione. Da notare che e' l'unico **parametro**.



Ci sono ancora problemi: Questo metodo non è esente da problemi, quindi abbiamo:

- Dipendentemente dal punto di inizio, abbiamo un risultato diverso. Dobbiamo capire che, giustamente, se ci accontentiamo di un minimo locale, non avremo quasi mai lo stesso minimo per ogni allenamento della rete.

Nota: con **topologia** intendiamo il numero di layer e il numero di neuroni per layer.

Diciamo che in generale, gli steps per allenare una rete sono:

```
for each topology T in C_t
    for each η in C_η
        for each initialization I in C_I
            Applica la discesa del gradiente
```

3.6.2 Il metodo di discesa del gradiente stocastico

La differenza del metodo di discesa del gradiente stocastico è che, invece di calcolare il gradiente su tutti i dati, si calcola il gradiente su un sottoinsieme di dati.

Ciò che viene fatto, per ogni step, invece di prendere la derivata di ogni loss function e poi fare i calcoli, prendo **un sample del mio dataset**, tipicamente chiamato **batch**. Ogni volta che calcolo la discesa del gradiente, calcolo la derivata **NON PER TUTTO IL DATASET**, ma solamente del **batch**. Questo OVVIAMENTE non mi assicura che ottimizzare ogni batch mi ottimizza anche l'intero dataset, ma non c'è modo di lavorare sull'intero dasaset, poiché questo è troppo grande e richiede troppo tempo.

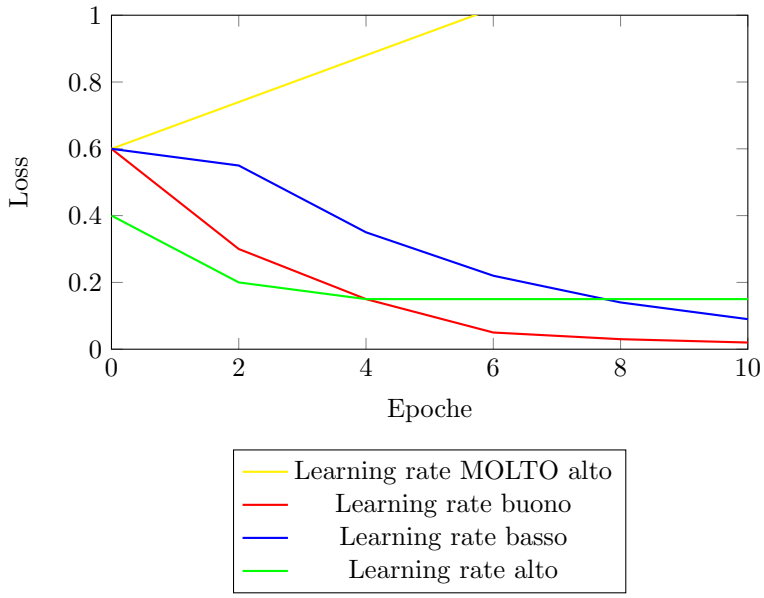
Nonostante tutto, utilizzando questa tecnica **si ottengono risultati comunque accettabili**.

Il *workflow* allora cambia in:

```
for each topology T in C_t
    for each η in C_η
      for each initialization I in C_I
       for each batch size b ∈ C_b
         Applica la discesa del gradiente stocastica
```

3.6.3 Linee guida sul learning rate

Epoca: Un'epoca è un passaggio dell'approssimazione della funzione



Analisi dei learning rate:

- learning rate basso: troppo lento!
- learning rate alto: fermo con un minimo locale immediatamente
- learning rate MOLTO alto: non converge mai
- learning rate buono: decresce sempre e eventualmente converge

Nota: Il learning rate è un parametro che va **tunato**.

- Se il learning rate è troppo alto, non si riesce a convergere poiché si salta il minimo
- Se il learning rate è troppo basso, si rischia di non convergere mai

Il nostro obiettivo è quello di avere un learning rate che sia **giusto** e che permetta di avere un andamento della funzione di loss come quello in rosso, cioé avere una buona diminuizione della loss andando avanti con le epoche. Sempre decrescente e con la distanza minore dall'asse delle x.

Domanda: Vogliamo sempre la loss uguale a 0? **No.** Perché? Perché a differenza che in statistica dove vogliamo avere un errore il minimo. In deep learning non è così. Se la loss è 0, allora vuol dire che la rete ha imparato a memoria i dati, e non è quello che vogliamo. Vogliamo che la rete sia capace di generalizzare.

In particolare, noi **vogliamo fare predizioni**, sopratutto su **istanze ancora non viste.** Se avessimo una loss uguale a 0, probabilmente non saremmo in grado di avere delle predizioni decenti, poiché la rete non è in grado di generalizzare. Avendo la loss non a 0 rendiamo la rete capace di poter introdurre istanze ancora non viste in futuro.

Quindi, per statistica **loss = 0 : nice**, per **deep learning: insomma**.