Appunti di Artificial Intelligence

Ivan Masnari*

Facoltà di Informatica, UniMi, Milano

Ultima modifica: 30 settembre 2020

1 Introduzione

Dato un qualunque sistema, se disponiamo di un insieme di leggi o regole che lo descrivono completamente (nel caso di un sistema fisico avremmo delle equazioni differenziali) potremmo, in teoria, calcolarne in ogni momento lo stato e, quindi, prevederne l'evoluzione nel tempo. Tuttavia, nella vita di ogni giorno capita spesso di non avere a disposizione una conoscenza perfetta di un certo sistema. Tale informazione:

1. può mancare.
2. possiamo averne una conoscenza approssimata.

L'intelligenza artificiale nasce con lo scopo di estrarre conoscenza direttamente dai dati in nostro possesso attraverso strumenti automatici. Questo modello si differenzia rispetto alla descrizione *a priori* del sistema, in quanto lo simula per comprenderne *a posteriori* il suo comportamento. Per far questo, è stato utile studiare come gli esseri viventi interagiscano con l'ambiente circostante e come vi si adattino. Vari modelli di intelligenza artificiale sono stati proposti lungo la storia della disciplina. Una categorizzazione preliminare che si fa in letteratura è quella tra modelli:

- *simbolici*, in cui i dati vengono sottoposti a codifica e solo dopo manipolati. Storicamente questo è stato il primo approccio adottato (vedi sistemi esperti degli anni '70).
- *pre-simbolici*, in cui i dati vengono manipolati direttamente, senza la mediazione di una codifica. Fanno parte di questa famiglia: le reti neurali, i sistemi fuzzy e gli algoritmi evolutivi.

Nel corso ci concentreremo sui secondi.

*e-mail: ivan.masnari@studenti.unimi.it

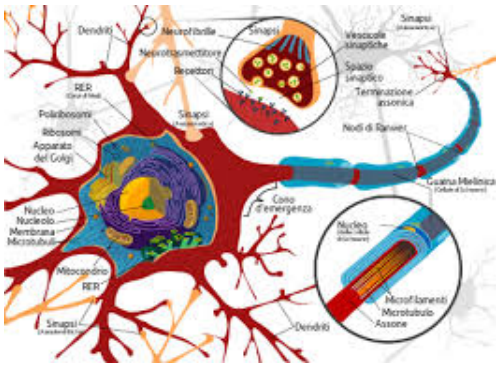

Figura 1: Neurone

2 Reti neurali

2.1 Background biologico

Il nostro cervello ci permette di analizzare in maniera molto sofisticata l'ambiente in cui ci troviamo per agire nel miglior modo possibile (esempio: se riconosciamo un leone nella savana, scappiamo nell'altra direzione). Queste analisi sono basate sul funzionamento del cervello: come estrae informazioni, come queste interagiscono con l'informazioni contenute in memoria, etc. Lo studio di questi processi è un campo di ricerca molto attivo e multidisciplinare dove convergono gli interessi della biologia, della medicina e della psicologia. Tali studi ci offrono dei modelli che simulano l'attività celebrale. Proprio questi modelli, vengono poi utilizzati dall'informatica per offrire strumenti di predizione, ottimizzazione e problem-solving in vari campi applicativi (guida automatizzata, smart cities, etc.). Il successo di questi modelli è condizionato dal fatto che il nostro cervello è un potente computer capace di computare in parallelo grandi porzioni di dati. Ma come funziona esattamente?

Il cervello è composto da miliardi di cellule dette *neuroni* (Figura 1). Il neurone a sua volta è costituito da:

- i *dendriti*, i quali sono filamenti raggiunti dalle terminazioni di altri neuroni e che gli permettono di raccogliere informazioni grazie a processi biochimici originati dai così detti *neurotrasmettitori*.
- l'*assone*: un lungo filamento che parte dal corpo centrale della cellula e trasmette segnali elettrici che, a loro volta, vanno ad attivare altri neuroni attraverso il rilascio di neurotrasmettitori.

Quando e come il neurone trasmetta il segnale di attivazione dipende dal particolare modello fisiologico che si voglia adottare. Solitamente si considera un *threshold*, superato il quale, l'assone viene depolarizzato e la differenza di potenziale provoca il passaggio di una corrente. Un diverso modello prende in considerazione non tanto la potenza dello stimolo quanto il loro numero. Questa struttura a network offre ottime prestazioni. Per un confronto con una CPU classica alleghiamo la seguente tabella:

	Personal computer	Human brain
Processing units	1 CPU, 2–10 cores 10^{10} transistors 1–2 graphics cards/GPUs, 10^3 cores/shaders 10^{10} transistors	10^{11} neurons
Storage capacity	10^{10} bytes main memory (RAM) 10^{12} bytes external memory	10^{11} neurons 10^{14} synapses
Processing speed	10^{-9} seconds 10^9 operations per second	$>10^{-3}$ seconds < 1000 per second
Bandwidth	1012 bits/second	10^{14} bits/second
Neural updates	106 per second	10^{14} per second

I vantaggi delle reti neurali sono:

1. Alta velocità di calcolo, grazie al parallelismo.
2. Tolleranza ai guasti: la rete rimane funzionale anche quando molti neuroni smettono di funzionare.
3. La performance degrada in modo lineare con il numero di neuroni danneggiati.
4. Ottimo per l'apprendimento induttivo.

2.2 Threshold logic unit

Per implementare una rete neurale artificiale occorre trovare un analogo del neurone naturale. Tale compito è svolto dalle *threshold logic unit*, nel seguito TLU. Una TLU è costituita da n variabili di input $x_1 \dots x_n$ e un output y. Ad ogni unità viene assegnato un *threshold* θ e ad ogni variabile di input un peso w_i dove $i \in \{1, \dots, n\}$ che rappresenta la rilevanza ai fine della computazione di quel particolare input. L'output della TLU viene calcolato secondo la seguente formula:

$$y = \begin{cases} 1 & \text{se } \sum w_i x_i \geq \theta \\ 0 & \text{altrimenti} \end{cases} \tag{1}$$

Attraverso questo semplice meccanismo possiamo simulare alcune funzioni booleane. Se volessimo computare l'AND logico tra due input x_1 e x_2 basta assegnare valori ai pesi e al threshold in modo che soddisfino il seguente sistema di disequazioni:

$$\begin{cases} w_1 + w_2 \geq \theta \\ w_1 < \theta \\ w_2 < \theta \end{cases} \tag{2}$$

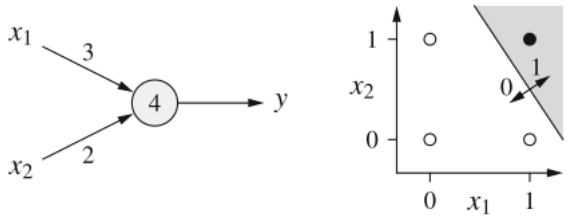

Figura 2: Rappresentazione geometrica della TLU per $x_1 \wedge x_2$

Risulta evidente che l'unica circostanza in cui l'output della TLU verrà posto ad 1 sarà quando entrambi gli input si trovano a 1. Inoltre, si noti che esistono varie scelte possibili di pesi e threshold che verificano le diseguaglianze.

2.3 Interpretazione geometrica

La condizione che calcola l'output della TLU somiglia molto da vicino all'equazione di un iperpiano (ovvero, un piano in n dimensioni):

$$\sum w_i x_i + \theta = 0 \tag{3}$$

Se pensiamo al caso precedente dell'AND logico e consideriamo i valori di input come coordinate in uno spazio bidimensionale, possiamo vedere che la retta definita da $x_1 w_1 + x_2 w_2 + \theta = 0$ corrisponde al confine che separa quelle combinazioni di valori che restituiscono come output 1 e quelle che, invece, restituiscono 0 (vedi Figura 2).
Da quanto detto, tuttavia, si può dedurre che una singola TLU potrà computare solo funzioni *linearmente separabili*, ovvero funzioni in cui le coordinate associate agli input che restituiscono 1 possono essere separate da quelle che restituiscono 0 da una funzione lineare (punto, retta, piano o iperpiano a seconda della dimensione).

Definizione 1 *Un insieme di punti X in uno spazio euclideo si dice* convesso *se e solo se non è vuoto, è connesso e ogni coppia di punti può essere congiunta da un segmento.*

Definizione 2 *Un* guscio convesso *di un insieme di punti X in uno spazio euclideo è il più piccolo insieme convesso che contiene X.*

Teorema 1 *Due insiemi di punti X e Y si dicono linearmente separabili se e solo se i loro gusci convessi sono tra loro disgiunti.*

Questo significa che già all'interno delle funzioni booleane ne esistono alcune che non possono essere simulate da una TLU. Come, per esempio, la doppia implicazione. Sebbene solo due funzioni booleane a due argomenti non siano linearmente indipendenti, al crescere degli argomenti il numero di funzioni che sono linearmente indipendenti diminuisce rapidamente. Per un numeri di argomenti arbitrariamente grande, una singola TLU non può calcolare "quasi" nessuna funzione.

Il problema può essere ovviato attraverso la costruzione di network di TLU più complessi. Come esempio consideriamo il network che simula la doppia implicazione (vedi figura 4).

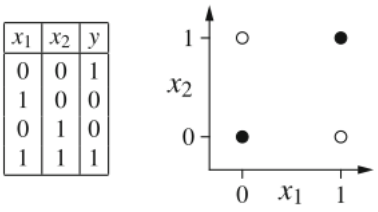

Figura 3: La doppia implicazione non è linearmente separabile

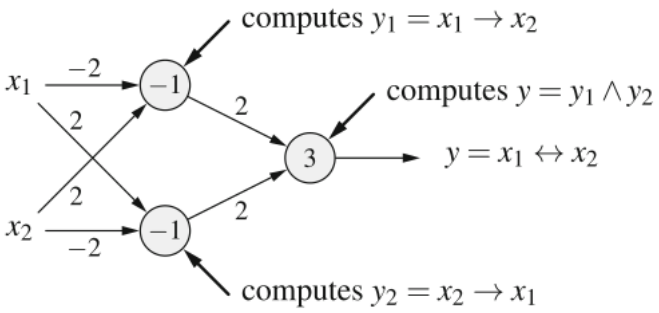

Figura 4: network di TLU che simula la doppia implicazione

2.4 Training delle TLU

L'interpretazione geometrica ci da una intuizione su come costruire una TLU avente 2 o 3 input, ma non è un metodo scalabile, nè automatizzato. Come far evolvere una TLU affinchè converga in modo autonomo ad una soluzione? Un algoritmo che ci permette di automatizzare il processo è il seguente:

1. Inizializzare i pesi e il threshold con valori randomici.
2. Determinare l'errore nell'output per un insieme di controlli. L'errore viene calcolato come una funzione dei pesi e del threshold $e(w_1, \ldots, w_n, \theta)$.
3. Aggiornare i pesi e il threshold per correggere l'errore.
4. Iterare finchè l'errore si annulla.

Mostriamo il comportamento dell'algoritmo nel caso più semplice, in cui abbiamo un threshold ed un unico input (quindi, un unico peso associato). Poniamo che si voglia allenare il nostro neurone a calcolare la negazione booleana. Sia x l'input, w il peso associato e θ il threshold, allora l'output y sarà definito come:

$$y = \begin{cases} 1 & \text{se } 0w = 0 \geq \theta \\ 0 & \text{se } 1w = w \geq \theta \end{cases} \tag{4}$$

Calcoliamo la funzione errore al variare di w e θ. Nel caso che $x = 0$ l'errore sarà 0 per un θ negativo e 1 per un θ negativo. Il peso non avrà alcuna influenza

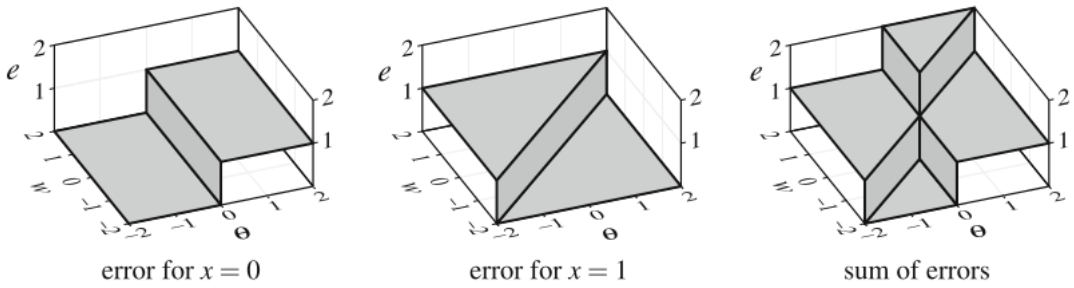

Figura 5: funzione di errore per la negazione booleana

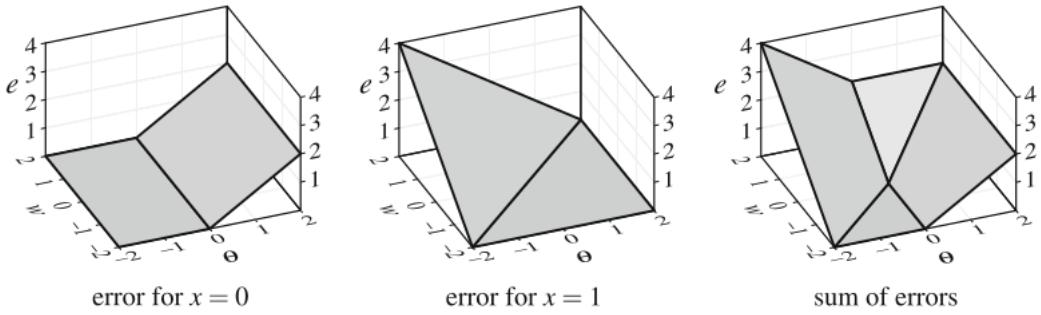

Figura 6: funzione di errore differenziabile

perchè viene annullato nella moltiplicazione con l'input. Quando, invece, $x = 1$, avremo che la funzione dipenderà da entrambi i parametri (vedi Figura 5).

La funzione di errore così calcolata non può essere usata direttamente nella nostra computazione perchè è composta da plateau e, quindi, non è ovunque derivabile. La soluzione è quella di calcolare la funzione di errore in modo tale che ci offra una misura di "quanto sbagliata" sia la relazione tra pesi e threshold. Otterremo così una funzione di errore che, seppur ancora non differenziabile, (vedi Figura 6) lo sia localmente nei punti in cui l'errore si discosta da 0. Ciò che faremo per correggere l'errore, dunque, sarà discendere verso l'area dove la funzione di errore si annulla. Questo è possibile esattamente perché abbiamo costruito una funzione derivabile nei punti in cui ci interessa, e cioè possiamo sempre calcolare la direzione migliore da prendere perchè si "scenda". Ci sono due modi di immaginare il processo di allenamento del neurone:

- *Online learning*: dove correggiamo l'errore individualmente per ogni scelta dell'input.
- *Batch learning*: dove prendiamo in considerazione l'errore cumulato su una sequenza di input prima di applicare le correzioni.

Definiamo di seguito la *delta rule* o *procedura di Widrow-Hoff* per allenare le TLU:

Definizione 3 *Sia* $\mathbf{v} = (x_1, \ldots, x_n)$ *il vettore di input di una TLU,* o *l'output aspettato e* y *il valore attuale. Se* o = y*, abbiamo finito. Al contrario, per*

ridurre l'errore computeremo nuovi valori per il threshold e i pesi nel seguente modo:

$$\theta^{(new)} = \theta^{(old)} + \Delta\theta \text{ con } \Delta\theta = -\eta(\mathrm{o} - \mathrm{y})$$

$$\forall i \in \{1, \ldots, n\} : w_i^{(new)} = w_i^{(old)} + \Delta w_i \text{ con } \Delta w_i = \eta(\mathrm{o} - \mathrm{y})x_i$$

dove η è il learning rate*. Più è alto, più i cambiamenti sui pesi e sui threshold sono drastici.*

Abbiamo visto prima, tuttavia, che non tutte le funzioni possono essere computate. Per le funzioni linearmente separabili esiste un teorema che ci garantisce che applicando la *delta rule* l'algoritmo converga ad una soluzione.

Teorema 2 *Sia $L = \{(\mathbf{v}_1, \mathrm{o}_1), \ldots, (\mathbf{v}_n, \mathrm{o}_n)\}$ una sequenza di pattern di allenamento per la TLU, dove $\mathbf{v}_i$ sono i vettori di input e o_i l'output atteso. Siano inoltre $L_0 = \{(\mathbf{v}, \mathrm{o}) \in L | \mathrm{o} = 0\}$ e $L_1 = \{(\mathbf{v}, \mathrm{o}) \in L | \mathrm{o} = 1\}$ rispettivamente gli insiemi delle coppie di pattern che hanno come output atteso 0 e quelle che hanno come pattern atteso 1. Se L_0 e L_1 sono linearmente separabili, allora esiste un $\mathbf{w}$ vettore di pesi e un θ threshold t.c.:*

$$\forall (\mathbf{v}, 0) \in L_0 : \mathbf{wv} < \theta$$

$$\forall (\mathbf{v}, 1) \in L_1 : \mathbf{wv} \geq \theta$$

Osservazione 1 *Negli esempi precedenti abbiamo codificato il valore booleano* falso *come 0 e* vero *come 1. Questa scelta non è felice perchè ha lo svantaggio che, nel caso di* falso*, i pesi corrispondenti perchè la formula contiene l'input come fattore. Per evitare il problema si ricorre in letteratura ad una diversa codifica chiamata* ADALINE *(ADAptive LINear Element), dove* falso *viene ad assumere il valore -1 e il* vero 1*.*

Notiamo che questa procedura di allenamento vale solo per le singole TLU, ma abbiamo prima visto che le TLU possono computare solo funzioni linearmente separabili. Sebbene questo inconveniente si possa evitare prendendo in esame *network* di TLU, questa procedura non si estende naturalmente a quel caso.

3 Sistemi fuzzy

4 Algoritmi evolutivi