Appunti di Artificial Intelligence

Ivan Masnari*

Facoltà di Informatica, UniMi, Milano

Ultima modifica: 12 ottobre 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

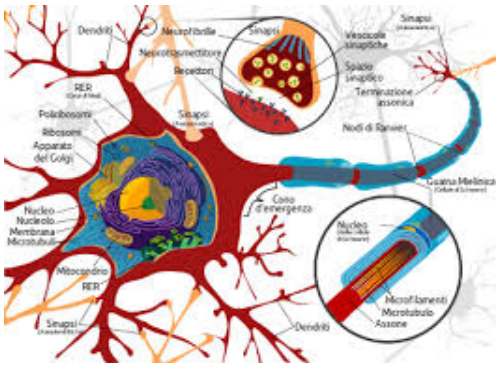


Figura 1: Neurone

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:

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

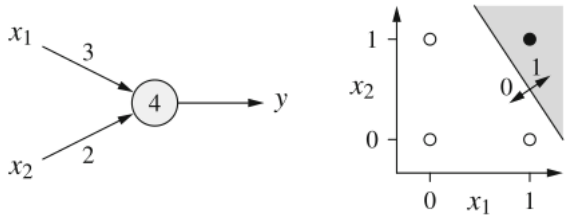


Figura 2: Rappresentazione geometrica della TLU per $x_1 \wedge x_2$

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).

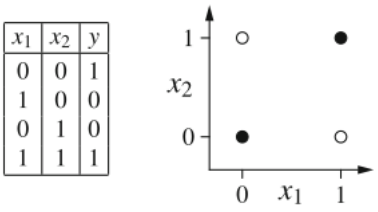


Figura 3: La doppia implicazione non è linearmente separabile

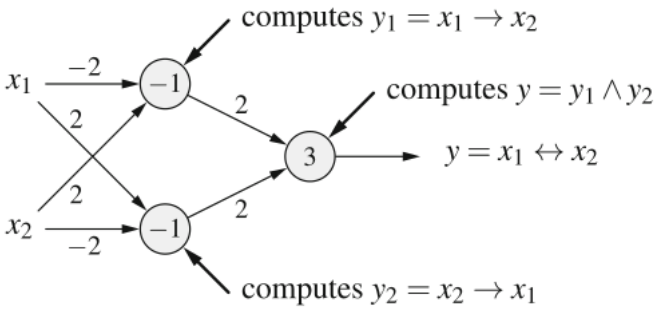


Figura 4: network di TLU che simula la doppia implicazione

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

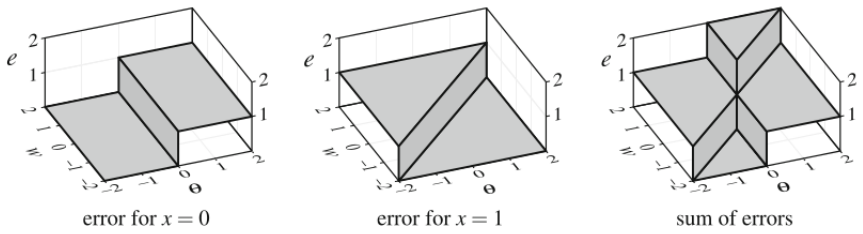


Figura 5: funzione di errore per la negazione booleana

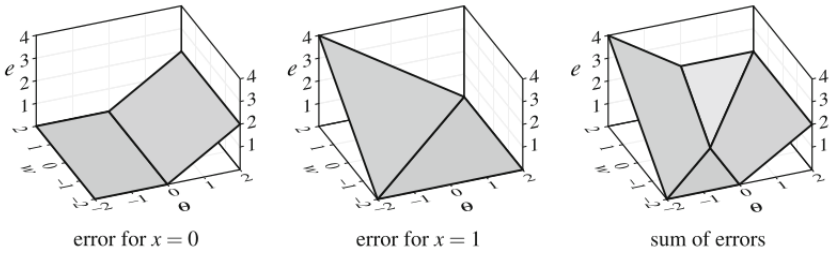


Figura 6: funzione di errore differenziabile

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* $\mathrm{o} = \mathrm{y}$*, abbiamo finito. Al contrario, per ridurre l'errore computeremo nuovi valori per il threshold e i pesi nel seguente modo:*

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

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

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.

2.5 Artificial neural network

Un artificial neural network (in quello che segue ANN) può essere rappresentata come un grafo diretto $G = (U, C)$ dove i nodi sono TLU e gli archi sono le connessioni tra le varie unità. L'insieme dei nodi U può essere partizionato in tre sottoinsiemi:

- $U_{(in)}$: è l'insieme dei nodi di input, i quali ricevono in modo diretto l'informazione dall'ambiente.
- $U_{(out)}$: è l'insieme dei nodi di output, i quali sono i soli nodi a comunicare con l'esterno.
- $U_{(hidden)}$: è l'insieme dei nodi interni, i quali propagano la computazione.

Ogni connessione $(u, v) \in C$ possiede un peso w_{uv} che definisce l'importanza del dato originato da v per il neurone u. Ad ogni neurone $u \in U$ vengono, invece, assegnate quattro variabili: il *network input* net_u, la *activation* act_u, l'*output* out_u e l'*external input* ext_u (vedi Figura 7). Le prime tre variabili vengono calcolate in ogni momento dell'evoluzione dell'ANN grazie a tre funzioni associate:

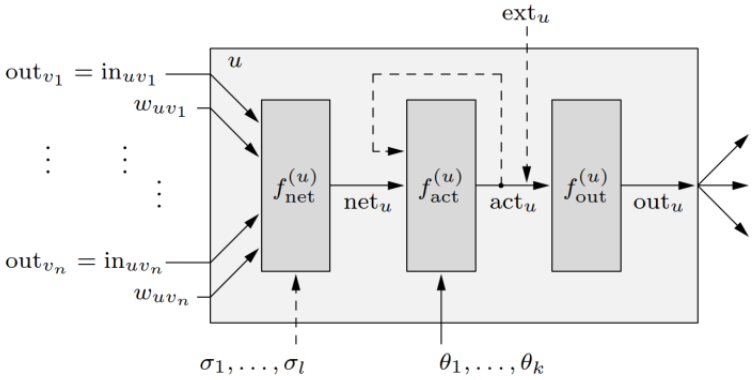
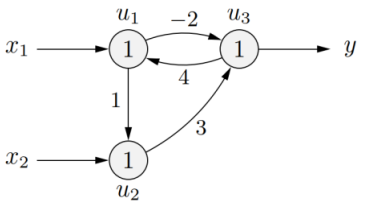


Figura 7: rappresentazione di un singolo neurone



	u_1	u_2	u_3		
input phase	**1**	**0**	**0**		
work phase	1	0	**0**	net$_{u_3}$ = −2	< 1
	1	**1**	0	net$_{u_2}$ = 1	≥ 1
	0	1	0	net$_{u_1}$ = 0	< 1
	0	1	**1**	net$_{u_3}$ = 3	≥ 1
	0	**0**	1	net$_{u_2}$ = 0	< 1
	1	0	1	net$_{u_1}$ = 4	≥ 1
	1	0	**0**	net$_{u_3}$ = −2	< 1

Figura 8: computazione di una recurrent neural network che non giunge ad uno stato stabile

1. La *network input function* f_{net}^u: calcola la somma pesata dell'input.
2. La *activation function* f_{act}^u: ne esistono vari modelli (gaussiana, sigmoide, etc.) a seconda dell'applicazione.
3. La *output function* f_{out}^u: definisce l'output a seconda che il neurone venga o meno attivato.

Se il grafo che rappresenta l'ANN è aciclico si parla di *feed forward network* e la computazione procede in modo unidirezionale da $U_{(in)}$ a $U_{(out)}$ seguendo l'ordine topologico[1] del network. Nel caso, invece, il grafo contenga un ciclo, allora si parla di *recurrent network*. I processi all'interno di un ANN si dividono in due fasi:

1. La *input phase*: dove gli input esterni vengono acquisiti dai neuroni di input.
2. La *work phase*: dove i neuroni di input vengono spenti e un nuovo output viene computato da ogni neurone. La *work phase* continua finchè gli output sono stabili o si raggiunge un timeout.

[1]L'ordine topologico è una numerazione dei vertici di un grafo diretto tale che tutti gli archi partano da un nodo associato ad un numero minore rispetto a quello associato al nodo di arrivo. Un ordine topologico esiste solo per grafi aciclici.

Nel caso delle recurrent neural network, potrebbe accadere che non si giunga mai ad uno stato stabile a seconda di quale ordine di update dei neuroni si scelga di seguire. In Figura 8 abbiamo un esempio di una computazione con risultato oscillante in un recurrent neural network. L'ordine seguito per l'update è: $u_3, u_1, u_2, u_3, u_1, u_2 \ldots$. Se si fosse seguito un diverso ordine la computazione avrebbe raggiunto uno stato stabile.

2.6 Training delle ANN

Abbiamo visto in precedenza che è possibile allenare in modo automatico una singola TLU grazie alla delta rule. Come abbiamo già avuto modo di osservare questo procedimento non può essere generalizzato alle ANN. Tuttavia, i principi a cui ci ispiriamo sono i medesimi: calcolare correzioni ai pesi ed ai threshold dei singoli neuroni e aggiornarli di conseguenza. A seconda del tipo dei dati che utilizziamo per allenare le nostre ANN e dei criteri di ottimizzazione distiguiamo due tipi di apprendimento:

1. *fixed learning task* o apprendimento con supervisione
2. *free learning task* o apprendimento senza supervisione

Nel caso di una fixed learning task avremo un insieme $L = \{(\mathbf{i}_1, \mathbf{o}_1), \ldots, (\mathbf{i}_n, \mathbf{o}_n)\}$ di coppie che assegnano ad ogni input un output desiderato. Una volta completato il processo di apprendimento, la ANN dovrebbe essere in grado di restituire l'output adeguato rispetto all'input che le viene presentato. In pratica, questo accade raramente e bisogna accontentarsi di un risultato approssimativo. Per giudicare in che misura una ANN si avvicina alla soluzione della fixed learning task si adotta una funzione di errore. Solitamente tale funziona viene calcolata come il quadrato della differenza tra l'output desiderato e quello attuale:

$$e = \sum_{l \in L} \sum_{v \in U_{(out)}} e_v^l$$

dove

$$e_v^l = (o_v^l - out_v)^2$$

è l'errore individuale per una particolare coppia l e un neurone di output v. Il quadrato delle differenze viene scelto per vari motivi. Per prima cosa, errori positivi e negativi altrimenti si cancellerebbero a vicenda e non sarebbero presi in considerazione. In secondo luogo, questa funzione è ovunque derivabile, semplificando così il processo di aggiornamento dei pesi e dei threshold. Nel free learning task avremo, invece, solo una sequenza di input $L = \{\mathbf{i}_1, \ldots, \mathbf{i}_n\}$. Questo comporta che, a differenza del fixed learning task, non avremo modo di calcolare una funzione di errore rispetto ad un output atteso. In linea di principio, infatti, l'obiettivo di un free learning task sarà quello di produrre un output "simile" per input "simili". Un caso particolare potrebbe essere quello del *clustering* dei vettori di input. Qualsiasi processo di apprendimento si scelga esistono alcune buone pratiche che è utile seguire. Una è quella di normalizzare il vettore di input. Comunemente lo si scala in modo tale che abbia media uguale a 0 e la varianza ad 1. Per fare questo uno deve calcolare per ogni neurone $u_k \in U_{(in)}$ la media aritmetica μ_k e la deviazione standard σ_k degli input esterni:

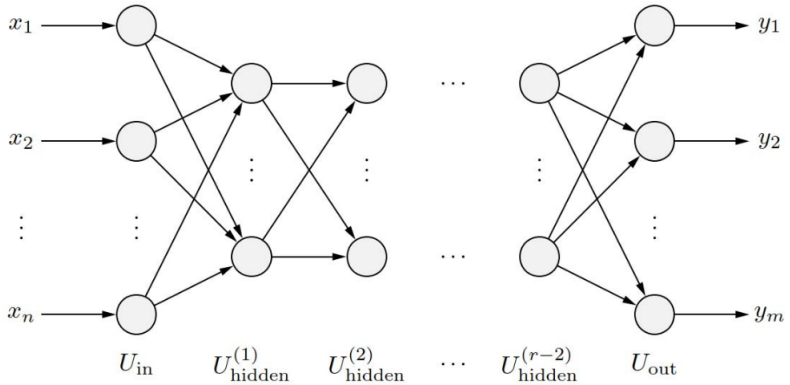


Figura 9: multi-layer perceptrons

$$\mu_k = \frac{1}{|L|}\sum_{l\in L} ext^l_{u_k} \qquad \sigma_k = \sqrt{\frac{1}{|L|}\sum_{l\in L}(ext^l_{u_k} - \mu_k)^2}$$

Quindi gli input esterni vengono ricalcolati secondo questa formula:

$$ext^{new}_{u_k} = \frac{ext^{old}_{u_k} - \mu_k}{\sigma_k}$$

2.7 Multi-layer perceptrons

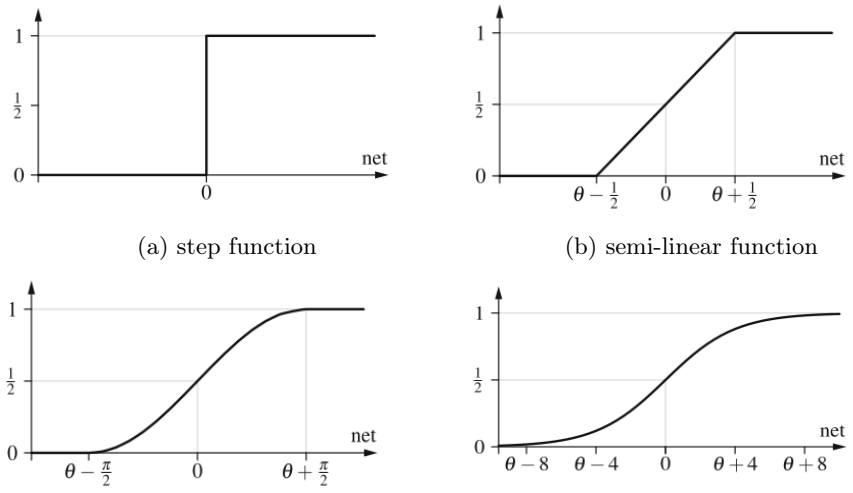
Una delle prime ANN sviluppate furono i *multi-layer perceptrons* (nel seguito MLP). Le MLP sono particolari feed-forward network in cui le unità base (i percettroni) sono organizzati in *layer* e ogni layer ha connessioni solo con il layer successivo (vedi Figura 9). Questo permette di minimizzare il fenomeno delle continue ricomputazioni che avverrebbero durante la propagazione del segnale nei normali feed-forward network. La network input function di ogni neurone $u \in U_{(hidden)\cup U_{(out)}}$ viene calcolata come la somma pesata degli input, come:

$$f^u_{net}(\mathbf{w}_u, \mathbf{i}_u) = \sum_{v\in pred(u)} w_{uv} out_v$$

L'activation function, invece, è una così detta *funzione sigmoide*, ossia una funzione monotona non descrescente tale che:

$$f : \mathbb{R} \to [0,1] \quad \text{con} \lim_{x\to-\infty} f(x) = 0 \quad \text{e} \lim_{x\to\infty} f(x) = 1$$

La funzione di output può essere sia una sigmoide oppure una semplice funzione lineare.
La struttura a layer di un MLP suggerisce che si possa descrivere il network con l'aiuto di una matrice dei pesi. In questo modo la computazione del MLP può essere rappresentata attraverso la moltiplicazione tra matrici e vettori. Tuttavia, noi non abbiamo utilizzato in classe una matrice per l'intero network, ma una per ogni singolo layer. Siano $U_1 = \{v_1, \ldots, v_n\}$ e $U_2 = \{u_1, \ldots, u_m\}$ due layer



(a) step function

(b) semi-linear function

(c) sine up to saturation

(d) logistic function

Figura 10: Alcune funzioni sigmoidi

consecutivi di neuroni. I pesi delle loro connessioni sono codificati in una matrice W di dimensioni $n \times m$:

$$W = \begin{pmatrix} w_{u_1 v_1} & w_{u_1 v_2} & \cdots & w_{u_1 v_n} \\ w_{u_2 v_1} & w_{u_2 v_2} & \cdots & w_{u_2 v_n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{u_m v_1} & w_{u_m v_2} & \cdots & w_{u_m v_n} \end{pmatrix}$$

Se due neuroni u_i e v_j non sono connessi, è sufficiente porre $w_{u_i v_j} = 0$. Il vantaggio di questa matrice sta nel fatto che è possibile scrivere il network input di un layer come:

$$\mathbf{net}_{U_2} = W\mathbf{in}_{U_2} = W\mathbf{out}_{U_1}$$

dove $\mathbf{net}_{U_2} = (net_{u_1}, \ldots, net_{u_m})^\top$ e $\mathbf{in}_{U_2} = \mathbf{out}_{U_1} = (out_{v_1}, \ldots, out_{v_n})^\top$. Fino ad adesso abbiamo visto che le ANN possono rappresentare funzioni booleane, ma quando si parla di funzioni a valori continui?

Teorema 3 *Ogni funzione Riemann-integrabile è approssimata con precisione arbitraria da un MLP avente quattro layer.*

Ogni funzione, infatti, può essere approssimata da una step function (come in Figura 11). Ad ogni pivot x_i associamo nel nostro MLP un neurone nel primo hidden layer (vedi Figura 12). Nel secondo hidden layer creiamo un neurone per ogni scalino, il quale riceverà input dai due neuroni del primo livello che sono assegnati ai valori x_i e x_{i+1} che definiscono i bordi dello scalino. A questo punto, scegliamo pesi e threshold in modo tale che il neurone venga attivato se e solo se l'input è maggiore di x_i e minore di x_{i+1}. Siccome la funzione di attivazione del neurone di output è la funzione di identità, il valore così calcolato viene emesso così come è ricevuto. Dovrebbe essere chiaro che

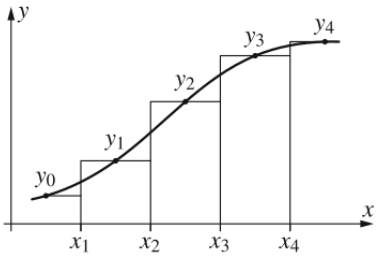


Figura 11: Approssimazione di una funzione continua con una step function

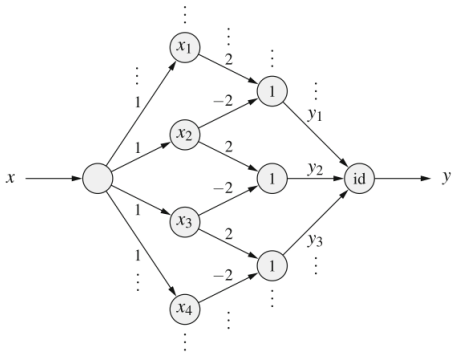


Figura 12: MLP che calcola la step function in Figura 11

l'approssimazione può crescere a piacere semplicemente aggiungendo neuroni e diminuendo la lunghezza dei gradini. Possiamo, inoltre, risparmiarci un layer se non utilizziamo nel calcolo l'altezza assoluta ma quella relativa come peso della connessione al neurone di output. Bisogna notare, comunque, che questo risultato non ha natura costruttiva, ossia non ci dice come deve essere fatto un MLP che approssimi con una data accuratezza una certa funzione. Tutto ciò che afferma il Teorema 3 è che limitare il numero di layer non pregiudica la proprietà del MLP di essere un *approssimatore universale*.

2.8 Regressione

Abbiamo visto che per allenare un ANN occorre minimizzare la funzione di errore, la quale si calcola solitamente come il quadrato della differenza tra output aspettato e attuale. Questo avvicina il problema dell'apprendimento nelle reti neurali a quello più generale della *regressione*. La regressione è una tecnica molto usata in analisi e in statistica per estrapolare la retta (o, più in generale, il polinomio) che meglio approssima la relazione esistente in un insieme di dati/osservazioni. Detto in modo più formale, se $G = \{(\mathbf{w}_0, y_0), \ldots, (\mathbf{w}_n, y_n)\}$ è il nostro dataset e immaginiamo esista una relazione funzionale tra il vettore di input $\mathbf{w}_i$ e l'ascissa y, allora la regressione ci aiuterà a trovare i parametri di quella funzione. A seconda del diverso genere di funzione avremo diverse forme di regressione.

2.8.1 Regressione lineare

Se ci aspettiamo che le nostre due quantità x e y esibiscano una dipendenza lineare, allora dovremo identificare i parametri a e b che individuano la retta $y = g(x) = a + bx$. In generale, tuttavia, non sarà possibile trovare una singola retta che passi per tutti i punti del nostro dataset. Quello che faremo sarà trovare la retta che devi dai punti il meno possibile e che, quindi, minimizzi l'errore calcolato come segue:

$$F(a, b) = \sum (g(x_i) - y_i)^2 = \sum (a + bx_i - y_i)^2$$

Il teorema di Fermat ci dice che una condizione necessaria perchè un minimo della funzione $F(a, b)$ esista è che la derivata parziale in entrambi i parametri si annulli:

$$\frac{\partial F}{\partial a} = \sum 2(a + bx_i - y_i) = 0$$

$$\frac{\partial F}{\partial b} = \sum 2(a + bx_i - y_i)x_i = 0$$

Questo sistema può essere risolto con alcune semplici tecniche di algebra lineare (vedi pag. 174 del libro). La soluzione così trovata sarà unica a meno che ogni valore x_i sia identico.

2.8.2 Regressione polinomiale e multilineare

Il metodo precedente può essere esteso in modo ovvio a polinomi di ordine arbitrario. In questo caso, si prende come ipotesi che la funzione indotta dal dataset approssimi un polinomio di ordine n:

$$y = p(x) = a_o + a_1 x + \cdots + a_n x^n$$

E si cercherà di minimizzare la funzione F tale che:

$$F(a_1, \ldots, a_n) = \sum (p(x_i) - y_i)^2 = \sum (a_o + a_1 x + \cdots + a_n x^n - y_i)^2$$

Come nel caso della regressione lineare, la funzione potrà essere minimizzata solo se le derivate parziali rispetto ai parametri a_i si annullano:

$$\frac{\partial F}{\partial a_1} = 0 \quad \ldots \quad \frac{\partial F}{\partial a_n} = 0$$

Inoltre, non siamo limitati a calcolare funzioni ad un solo argomento. Con alcune minori modifiche questo metodo è capace di approssimare funzioni in un numero arbitrario di argomenti. In quel caso, la chiameremo *regressione multilineare*.

2.8.3 Regressione logistica

Nel situazione in cui il nostro dataset non sia approssimato con sufficiente accuratezza da una funzione polinomiale, potremmo dover utilizzare funzioni di generi diversi. Data, per esempio, una funzione della forma:

$$y = ax^b$$

possiamo trasformarla in una equazione lineare applicando l'operazione di logaritmo:

$$ln(y) = ln(a) + b \cdot ln(x)$$

Nel caso delle ANN ci interessiamo in particolare alla funzione logistica (vedi Figura 10(d)):

$$y = \frac{Y}{1 + e^{a+bx}}$$

Siccome molte ANN utilizzano come funzione di attivazione del neurone proprio la funzione logistica, se trovassimo un modo di applicarci il metodo della regressione potremmo determinare i parametri di qualsiasi network a due layer con un unico input. Il valore a nella funzione corrisponderebbe al threshold del neurone di output e la b al peso dell'input. Possiamo "linearizzare" la funzione logistica applicandoci le seguenti trasformazioni (comunemente chiamata *logit transformation*):

$$y = \frac{Y}{1 + e^{a+bx}} \leftrightarrow \frac{1}{y} = \frac{1 + e^{a+bx}}{Y} \leftrightarrow \frac{Y - y}{y} = e^{a+bx} \leftrightarrow ln(\frac{Y - y}{y}) = a + bx$$

Se estendiamo il nostro approccio fino a comprendere funzioni con più argomenti, in analogia a quanto accade nella regressione multilineare, possiamo utilizzarlo per computare i pesi di network a due layer con arbitrari neuroni di input. Tuttavia, siccome il metodo della somma degli errori funziona solo quando parliamo di neuroni di output, questo approccio non può essere esteso a network con più di due layer.

2.9 Backpropagation

Come abbiamo appena visto la regressione logistica funziona solo per MLP con due layer di neuroni. Un approccio più generale è quello del *gradient descent*. Il metodo consiste nell'utilizzare la funzione di errore per calcolare la direzione in cui cambiare i pesi e il threshold per minimizzare l'errore. Condizione necessaria per il suo utilizzo è che la funzione sia differenziabile. Tuttavia, un MLP ha una funzione logistica come funzione di attivazione e, quindi, la funzione di errore sarà differenziabile (posto che la funzione di output sia la funzione identità). Intuitivamente, il *gradiente* descrive la pendenza di una funzione. Questo è calcolato assegnando ad ogni punto del dominio della funzione un vettore, i cui componenti sono le derivate parziali rispetto ai agli argomenti(un esempio in Figura 13). L'operazione di calcolare il gradiente (di un punto o di una funzione) è comunemente denotata con l'operatore differenziale ∇ (pronuncia: nabla).

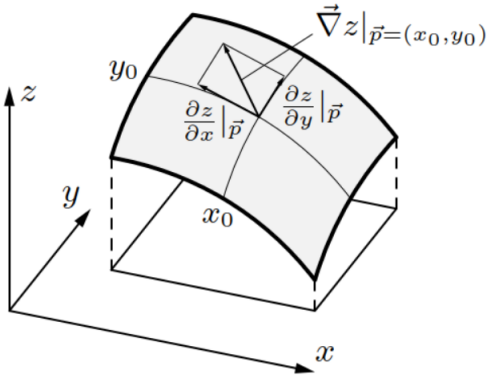


Figura 13: Il gradiente di una funzione a due argomenti.

Nel caso delle MLP, calcolare il gradiente della funzione di errore si traduce nel calcolare la derivata parziale della funzione di errore rispetto ai pesi e i threshold presi come parametri. Sia $\mathbf{w}_u = (-\theta, w_{u_1}, \ldots, w_{u_k})$ il vettore dei pesi di un singolo layer esteso così da includere anche il threshold, calcoliamo il gradiente come segue:

$$\nabla_{\mathbf{w}_u} e = \frac{\partial e}{\partial \mathbf{w}_u} = (-\frac{\partial e}{\partial \theta}, \frac{\partial e}{\partial w_{u_1}}, \ldots, \frac{\partial e}{\partial w_{u_k}})$$

Siccome l'errore totale e è dato dalla somma degli errori individuali rispetto a tutti i neuroni e tutti i training pattern l, otteniamo che:

$$\nabla_{\mathbf{w}_u} e = \frac{\partial e}{\partial \mathbf{w}_u} = \frac{\partial}{\partial \mathbf{w}_u} \sum_{l \in L} e^l = \sum_{l \in L} \frac{\partial e^l}{\partial \mathbf{w}_u}$$

Osservazione 2 Se abbiamo come $f_{(act)}$ la funzione logistica avremo che i cambiamenti operati sul vettore $\mathbf{w}_u$ saranno proporzionali alla derviata della funzione $f_{(act)}$. Più vicini allo 0 della funzione sono i valori, più ripido sarà il pendio della funzione e, per tanto, più rapido sarà l'apprendimento.

Come facciamo dopo aver trovato l'errore a calcolare la correzione necessaria per ogni peso e threshold di ogni singolo neurone? Il processo che ci permette di fare questo viene chiamato *error backpropagation* ed è schematizzato in Figura 14. Si assume che la funzione di attivazione sia la funzione logistica per ogni neurone $u \in U_{(hidden)} \cup U_{(out)}$ tranne che per quelli di input.

Inizialmente, (1) applichiamo l'input ai neuroni di input che lo restituiscono senza modifiche in output al primo dei layer hidden. (2) Calcoliamo per ogni neurone dei seguenti layer la somma pesata degli input e al risultato applichiamo la funzione logistica generando così l'output che verrà propagato in tutto il network fino ai neuroni terminali. A questo punto, (3) calcoliamo la differenza tra l'output atteso e quello attuale e, dato che la funzione di attivazione è invertibile, risaliamo dal vettore di errore a quale fosse l'input che ha condizionato

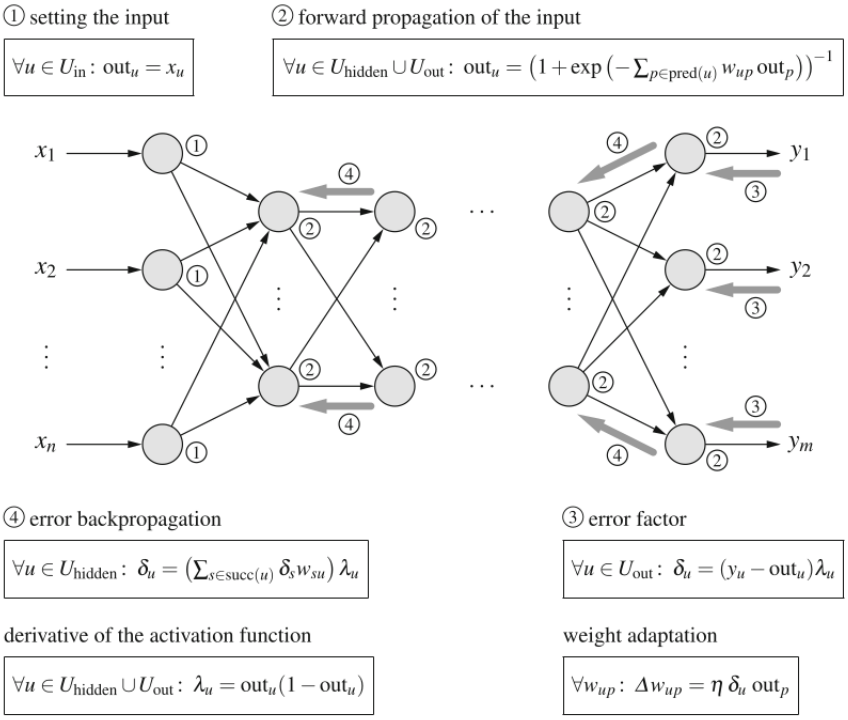


Figura 14: Propagazione dell'errore in un MLP.

quel particolare errore (la variabile δ_u, nell'immagine). Avendo, ora, (4) trasformato l'errore della variabile di output out_u in quello della variabile di input net_u possiamo distribuire l'errore (e la correzione necessaria) in modo proporzionale al ruolo del singolo neurone nel calcolo del seguente output. Propago a ritroso l'errore fino ai neuroni di input. Bisogna osservare comunque che data la forma della funzione logistica l'errore non può sparire completamente, in quanto il gradiente approssimerà il vettore nullo più si avvicinerà allo zero.

Osservazione 3 Se si inizializza il learning rate η ad un valore troppo alto, al posto di discendere la curva si corre il rischio di saltare da un "picco" della funzione all'altro senza convergere mai al minimo. Inoltre, non è affatto detto che il minimo raggiunto in questo modo sia il minimo globale della funzione. La causa sarà piuttosto da ascrivere alla scelta dei valori iniziali. Una soluzione al problema può essere quella di ripetere l'apprendimento, inizializzando il sistema con una diversa configurazione di pesi e threshold, e scegliere alla fine quale configurazione risulta in un miglior minimo.

2.9.1 Variazioni sul gradient descent

Esistono varie sofisticazioni della tecnica del gradient descent che permettono un più veloce apprendimento e, nello stesso momento, un miglior controllo sulla lunghezza dei singoli step di apprendimento. Alcuni esempi sono:

- *Manhattan training*: utlizza al posto del valore del gradiente solo il suo segno per calcolare la direzione. Questo permette di semplificare notevolmente la computazione.
- *Flat spot elimination*: cerca di limitare l'abbattimento della lunghezza degli step di apprendimento quando ci si avvicina ad un plateau della funzione "sollevando" artificialmente la derivata della funzione in quel punto.
- *Momentum term*: ad ogni successivo step aggiungo al gradiente una frazione del precedente cambiamento di pesi così da avere una memoria di quanto velocemente stava cambiando nel passato.
- *Self-adaptive error backpropagation*: permetto ad ogni parametro di avere un diverso learning rate in modo da avere un più fine controllo rispetto alle caratteristiche del singolo parametro.
- *Resilient error backpropagation*: combina il Manhattan training con l'approccio self-adaptive.
- *Quick propagation*: al posto di utilizzare il gradiente approssimo la funzione con una parabola e salto direttamente all'apice della parabola.
- *Weight decay*: riduce i pesi per evitare di rimanere intrappolato in una regione già saturata.

2.9.2 Overfitting e underfitting

Quanti neuroni ho bisogno per avere un buon network? Come regola di massima si dovrebbe scegliere il numero di neuroni negli hidden layer seconda la seguente formula:

$$\#\text{hidden neurons} = (\#\text{input neurons} + \#\text{output neurons})/2$$

Non esiste una spiegazione teoretica soddisfacente del perchè questo sia un buon numero, ma è stato dimostrato empiricamente. Se, infatti, il numero dei neuroni negli hidden layer è troppo basso rischiamo l'*underfitting*, ossia che il nostro MLP non riesca a catturare la complessità della funzione che vogliamo catturare. Al contrario se ne ho troppi rischio di incorrere nell'*overfitting*, ossia che il nostro MLP si adatti agli esempi che gli abbiamo fornito durante il periodo di apprendimento, ma anche alle loro specificità accidentali (errori e deviazioni). Per evitare questi fenomeni è buona pratica dividere il nostro data set in modo da avere due sottoinsiemi di dati: alcuni dati per l'apprendimento ed altri per la validazione del processo di apprendimento. I primi verranno usati per allenare il nostro network e i secondi per giudicare se effettivamente il network approssimi la funzione desiderata. É possibile iterare a piacere questo procedimento suddividendo i dati non in due sottoinsiemi, ma in un numero arbitrario così da ottenre una conferma incrociata dei progressi nell'apprendimento del nostro network. Un diverso metodo per evitare l'overfitting è quello di terminare l'apprendimento quando il differenziale dell'errore tra un epoca ed un altra si abbassi sotto una certa soglia, oppure se l'apprendimento si protrae per un periodo troppo lungo.

3 Sistemi fuzzy

4 Algoritmi evolutivi