

Istituto di Istruzione Superiore
“M. Buonarroti”
Via Velio Spano, 7 – 09036 Guspini (VS)

Le Reti Neurali Artificiali

Giacomo Trudu

Classe 5°A Industriale
Anno scolastico 2010 – 2011

<http://www.hackyourmind.org/>

Premessa

L'obiettivo di questa tesi è fornire al lettore le conoscenze necessarie a comprendere e a progettare una rete neurale artificiale di tipo feedforward. Ho scelto di trattare questo argomento perché da sempre suscita in me grande interesse e perché desideravo produrre un documento che fosse utile a chiunque volesse avvicinarsi per la prima volta a questi innovativi sistemi per il trattamento dell'informazione. Ho trovato diversa documentazione sul *Web*, alcune tesi di laurea, per la maggior parte in lingua inglese, e alcuni tutorial strettamente pratici, ma non un unico documento che introducesse le reti neurali artificiali in modo chiaro e sufficientemente preciso. Non sono sicuro di esserci riuscito e non ho alcuna pretesa in merito. Anzi, il tempo a mia disposizione era limitato ed è giusto ribadire che io (autore) non ho alcuna competenza particolare, se non la “passione per il sapere” e la determinazione a migliorarmi.

Il documento è suddiviso sostanzialmente in tre parti: un *capitolo introduttivo*, che descrive in termini generici il modello computazionale a partire dai suoi fondamenti biologici; un *capitolo sugli aspetti teorici*, che fornisce al lettore le basi matematico-scientifiche necessarie a comprendere il funzionamento delle reti neurali artificiali; ed una *parte conclusiva*, nella quale è presentata la libreria *Serotonina* insieme ad un esempio di applicazione delle reti neurali artificiali.

La *parte teorica* descrive alcuni modelli di rete neurale feedforward, a partire da quello più elementare, il *perceptrone*, fino a quelli che sono considerati lo *stato dell'arte*, ossia le *reti di perceptroni multistrato* che usano algoritmi di addestramento basati sulla *Resilient Backpropagation*. Ogni paragrafo di questa sezione è correlato da una semplice implementazione in linguaggio C++.

Serotonina è una libreria libera per la creazione e la gestione di reti neurali artificiali feedforward ed è stata scritta come parte integrante del lavoro svolto in questa tesi. Gli esempi contenuti in questo documento, infatti, sono stati scritti all'insegna della semplicità didattica e non secondo le rigide regole dell'efficienza e degli standard. *Serotonina* costituisce da questo punto di vista una buona base di partenza per chi volesse avvicinarsi seriamente alla progettazione di reti neurali artificiali ad alte prestazioni.

Le immagini presenti in questo documento sono state realizzate attraverso gli strumenti grafici di *LibreOffice Writer 3.4*, del software *gnuplot 4.4* e di *GIMP 2.6*.

Indice

<u>0x01</u> Introduzione.....	1
<u>0x02</u> Fondamenti biologici.....	1
<u>0x02:01</u> Il cervello umano.....	1
<u>0x02:02</u> Il neurone biologico.....	1
<u>0x03</u> Le reti neurali artificiali.....	3
<u>0x03:01</u> Il perceptrone.....	4
<u>0x03:02</u> Il perceptrone (codice).....	8
<u>0x03:03</u> L'apprendimento del perceptrone.....	9
<u>0x03:04</u> L'apprendimento del perceptrone (codice).....	12
<u>0x03:05</u> Il perceptrone multistrato.....	16
<u>0x03:06</u> Il perceptrone multistrato (codice).....	19
<u>0x03:07</u> L'algoritmo di retropropagazione dell'errore.....	22
<u>0x03:08</u> L'algoritmo di retropropagazione dell'errore (codice).....	25
<u>0x03:09</u> La retropropagazione elastica.....	29
<u>0x03:10</u> Le funzioni di trasferimento.....	31
<u>0x04</u> La libreria Serotonina.....	34
<u>0x04:01</u> Installazione su sistemi GNU/Linux.....	34
<u>0x04:02</u> Installazione su sistemi MS Windows.....	34
<u>0x04:03</u> Creazione e addestramento di una rete neurale.....	35
<u>0x04:04</u> Un'applicazione reale.....	36
<u>0x05</u> Conclusione.....	38
<u>0x06</u> Bibliografia.....	38

0x01 Introduzione

Con il passare degli anni i computer si sono dimostrati strumenti estremamente efficaci nell'eseguire calcoli matematici e nella risoluzione di problemi strutturati, ossia in tutti quei compiti che richiedono la ripetizione di una serie di operazioni ben definite. Si tratta, infatti, di *elaboratori sequenziali*, fondati sul concetto di *macchina di Turing*¹ e progettati per eseguire sequenze di calcoli numerici, anche a frequenze molto elevate (attualmente dell'ordine dei *GHz*). Tuttavia, apparentemente, queste macchine non sembrano essere in grado di svolgere quei lavori che per un essere umano appaiono estremamente semplici e immediati: distinguere un volto familiare in una foto, riconoscere un suono, identificare un fiore dal suo profumo, etc; sono tutti compiti che richiedono un'elevata capacità di segmentazione dell'informazione, che non può avvenire esclusivamente attraverso delle operazioni di basso livello (come ad esempio l'analisi delle variazioni di luminosità in un'immagine o la ricerca di corrispondenze in un database). Il volto di una persona, ad esempio, può apparire in infinite pose, con differenti espressioni e sotto diverse illuminazioni, senza contare che le immagini, come i suoni, sono soggette a “rumore”². Per cui diventa estremamente complicato, se non del tutto impossibile, formalizzare questo tipo di problemi in un algoritmo di risoluzione sequenziale che possa essere inserito in un programma.

Per far fronte a questi limiti sono stati introdotti dei nuovi *paradigmi computazionali*, biologicamente ispirati, attraverso i quali è stata possibile la realizzazione di “sistemi intelligenti” per il trattamento dell'informazione, con capacità di *apprendimento*, *adattamento* e *human-like reasoning*.

L'*intelligenza computazionale* (o *soft computing*) è la disciplina che studia questi nuovi paradigmi di ispirazione biologica e che comprende diversi modelli di elaborazione dell'informazione, tra i quali figurano le *reti neurali artificiali*, gli *algoritmi evolutivi*³ e i *sistemi a logica fuzzy*⁴.

0x02 Fondamenti biologici

0x02:01 Il cervello umano

Il cervello umano è la struttura più complessa dell'universo conosciuto. Ha un peso di circa 1300-1400 grammi ed un volume che va dai 1100 ai 1300 cm³. Con le sue ridotte dimensioni è in grado di svolgere operazioni estremamente complesse: al suo interno, miliardi di unità processanti, dette *neuroni*, sono interconnesse tra di loro per formare un'unica grande *rete processante*, chiamata appunto *rete neurale* (o *rete neuronale*). Questa singolare struttura consente al cervello di elaborare le informazioni in modo *massivamente parallelo e distribuito* e li conferisce, inoltre, la straordinaria capacità di “generalizzare” le soluzioni apprese, ossia la capacità di risolvere un problema mai incontrato prima attraverso analogie con i problemi già imparati.

0x02:02 Il neurone biologico

Il *neurone* (o *cellula neuronale*), come abbiamo visto, è l'unità fondamentale del sistema nervoso. Si tratta di una cellula altamente differenziata e specializzata per la raccolta, l'integrazione e la conduzione di *impulsi nervosi*. All'interno del cervello umano se ne trovano circa *100 miliardi*, delle

1 Pensata nel 1936 dal matematico inglese *Alan Turing*, è un dispositivo ideale in grado di risolvere un numero molto vasto di problemi attraverso una sequenza di istruzioni.

2 Perdita di informazioni dovuta ad interferenze durante la digitalizzazione di un'immagine o di un suono.

3 Un metodo euristico ispirato al principio della selezione naturale di *Charles Darwin*.

4 Sistemi a logica polivalente, come alternativa ai sistemi basati sulla logica tradizionale.

dimensioni di *10 micron* ciascuno.

Esistono tre classi principali di neuroni:

- i *neuroni sensori*, con sottotipi visivi, uditivi, olfattivi, tattili, etc; sono i neuroni incaricati di raccogliere gli stimoli dall'ambiente esterno e costituiscono l'input della rete neuronale;
- i *neuroni motori* o *moto-neuroni*, che controllano generalmente le fibre muscolari e che costituiscono l'output della rete neurale;
- i *neuroni intermedi* o *inter-neuroni*, connessi sia ai primi, sia ai secondi, sia ad altri inter-neuroni, che servono all'elaborazione vera e propria dell'informazione e alla sua trasmissione.

Ogni neurone possiede un corpo cellulare, detto *soma*, dotato come tutte le cellule del patrimonio genetico dell'individuo (*DNA*). Dal soma si diramano una fibra nervosa principale, chiamata *assone*, lunga da un millimetro a un metro, e diverse fibre secondarie, dette *dendriti*. Sia le dendriti che l'assone si connettono a molti altri neuroni in punti detti *sinapsi*⁵.

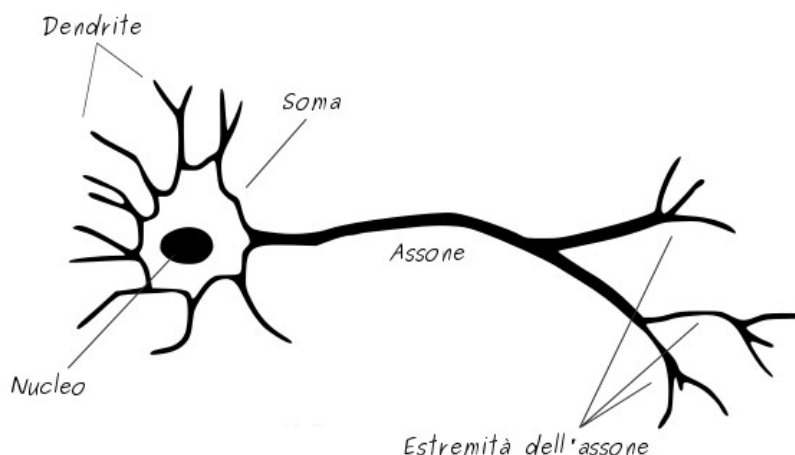


Fig. 1: Struttura di un neurone biologico

Tali contatti possono essere di tipo *eccitatorio* o *inibitorio* e la loro eccitazione/inibizione avviene grazie all'emissione locale di speciali sostanze chimiche dette *neurotrasmettitori* (ad esempio: acetilcolina, dopamina, acido GABA, etc).

Il compito dei *dendriti* è di ricevere l'informazione dai neuroni comunicanti (sotto forma di *impulso nervoso*) e di portarla verso il corpo cellulare; se la somma di tutti gli impulsi ricevuti, negativi e positivi, supera una certa soglia caratteristica, detta *soglia di attivazione*, il neurone invia in uscita un impulso corrispondente che, passando attraverso il suo *assone*, viene smistato alle dendriti di molti altri neuroni. Questo significa che il comportamento di ogni neurone è influenzato da quello dei suoi vicini e che, di conseguenza, l'elaborazione dell'informazione è determinata dal comportamento complessivo dell'intera rete. Non esiste perciò un "*grandmother cell*"⁶ (un *neurone della nonna*), ovvero un neurone che da solo detiene le informazioni su nostra nonna, ma, al contrario, l'informazione è delocalizzata nella topologia stessa della rete e nei *pesi sinaptici*⁷ dei neuroni che la compongono.

5 Si ipotizza che ogni neurone del cervello umano abbia circa 10.000 connessioni con altri neuroni, anche a lunghe distanze, per un totale di quasi 10^{14} sinapsi. Un numero davvero impressionante.

6 Termine coniato intorno al 1968 dal prof. *Jerry Lettvin* per illustrare l'inconsistenza delle teorie preconessioniste diffuse fino agli anni '80 che sostenevano l'esistenza di una gerarchia tra le reti del cervello tale che un singolo neurone potesse essere in grado di rappresentare concetti specifici ma complessi, come addirittura il volto di una persona.

7 La proprietà eccitatoria o inibitoria di una *sinapsi*.

0x03 Le reti neurali artificiali

Le *reti neurali artificiali* (*artificial neural network*, o più comunemente solo *reti neurali*) sono dei sistemi di elaborazione dell'informazione che cercano di riprodurre, più o meno in parte, il funzionamento dei sistemi nervosi biologici e, in particolare, i processi computazionali che avvengono all'interno del cervello umano durante le fasi di apprendimento e di riconoscimento.

La caratteristica più importante di questi sistemi è, appunto, quella di poter apprendere modelli matematico-statistici attraverso *l'esperienza*, ossia tramite la lettura dei dati sperimentali, senza dover determinare in modo esplicito le relazioni matematiche che legano le soluzioni al problema. La rete neurale artificiale non viene quindi programmata, bensì “addestrata” attraverso un processo di apprendimento basato su dati empirici.

Ne deriva una modellistica di tipo “*black box*” (scatola nera), che si contrappone a quella “*white box*” (scatola bianca) perché non si conoscono le componenti interne al sistema e non è possibile spiegare in termini logici come esso giunga ad un determinato risultato.

Per addestrare una rete neurale artificiale esistono tre grandi paradigmi di apprendimento:

- *l'apprendimento supervisionato*, in cui si utilizzano opportuni algoritmi atti a minimizzare l'errore di previsione della rete su un insieme finito di esempi tipici ripartiti in coppie input-output (detto *training set*); esso può essere di tipo *online*, se la correzione avviene in modo incrementale utilizzando un esempio alla volta, o di tipo *batch*, se la correzione dei pesi sinaptici è effettuata sulla totalità dell'errore degli esempi; se l'addestramento ha successo la rete impara a riconoscere la relazione implicita che lega le variabili di ingresso a quelle di uscita ed è in grado di rispondere correttamente anche a stimoli che non erano presenti nell'insieme di addestramento;
- *l'apprendimento non supervisionato*, dove si ricevono le informazioni sull'ambiente esterno senza fornire alcuna indicazione sui valori di output, nel tentativo di raggruppare tali dati in *cluster*, riconoscendo *schemi* o *patterns* impliciti;
- *l'apprendimento per rinforzo*, un paradigma più realistico e flessibile dei precedenti, in cui la rete interagisce direttamente con l'ambiente esterno che risponde attraverso stimoli positivi o negativi (“premi” o “punizioni”) che guidano l'algoritmo nella fase di apprendimento; la rete viene quindi adattata in modo da aumentare le probabilità di ottenere dei “premi” e diminuire quelle di ricevere “punizioni”;

Come quelle *biologiche*, le reti neurali artificiali sono composte da un certo numero di unità processanti che operano in parallelo. Queste unità sono dette *neuroni artificiali* o *neurodi* e possono essere ripartite in più sottoinsiemi della rete, chiamati “strati”. I neurodi di uno strato e quelli di un altro possono comunicare tra di loro attraverso delle connessioni pesate simili alle sinapsi biologiche.

In base all'organizzazione di tali connessioni avremo:

- *reti totalmente connesse*, dove ogni neurone di uno strato è connesso ad ogni neurone di un altro strato;
- *reti parzialmente connesse*, dove ogni neurone di uno strato è connesso ad un particolare sottoinsieme di neuroni di un altro strato;

Oppure in base al flusso dei segnali:

- *reti feedforward*, dove le connessioni trasportano il segnale solo in avanti;
- *reti feedback* (o *reti ricorrenti*), dove le connessioni trasportano il segnale anche all'indietro.

Il comportamento della rete neurale è determinato da tutti questi fattori ed ogni tipo di rete è adatto a svolgere determinati lavori.

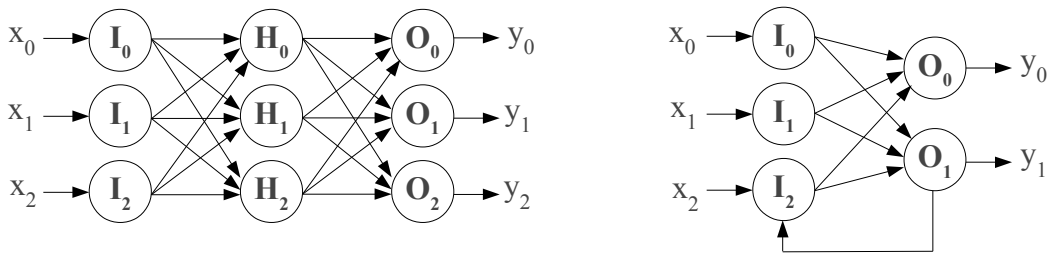


Fig. 2: Una rete feedforward a tre strati ed una rete feedforward ricorsiva a due strati

Per quanto riguarda le caratteristiche generali delle reti neurali, si può riassumere dicendo che esse sono in grado di:

- *apprendere per esempi*, come si è detto parlando del paradigma di apprendimento supervisionato;
- *generalizzare le soluzioni apprese*, ovvero rispondere correttamente a stimoli simili a quelli usati durante l'addestramento;
- *astrarre nuove soluzioni*, ossia rispondere correttamente a stimoli diversi da quelli usati per l'addestramento;
- *trattare dati "rumorosi"*, ossia rispondere correttamente anche in presenza di dati alterati o parziali.

In questa tesi, tuttavia, tratteremo unicamente le *reti neurali feedforward* che sono sicuramente il modello di rete neurale più diffuso e che ben si adattano all'approssimazione di funzioni matematiche e all'interpolazione statistica.

0x03:01 Il perceptrone

I primi a proporre una rete neurale per scopi computazionali furono *McCulloch e Pitts* nel 1943, i quali idearono un neurone artificiale, composto essenzialmente da un combinatore lineare a soglia, in grado di calcolare semplici funzioni booleane. Tuttavia il prototipo di rete neurale più significativo fu sicuramente il *perceptrone* pensato da *F. Rosenblatt* nel 1958; questo modello, come vedremo tra poco, introdusse per la prima volta i pesi sinaptici variabili permettendo alla rete neurale di "apprendere".

Il *perceptrone* (o *perceptron*) è considerato il modello più semplice di rete neurale artificiale feedforward, pensato per il riconoscimento e la classificazione di forme (o *patterns*). Si tratta di una rete neurale artificiale composta da un singolo neurone, avente "*n*" ingressi ed una sola uscita:

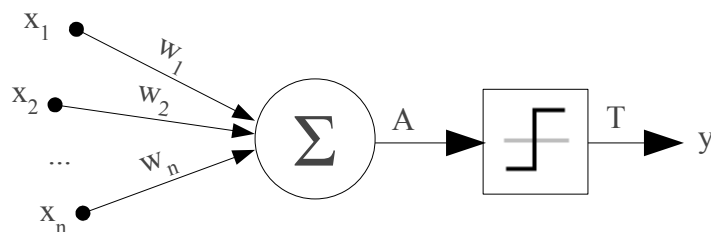


Fig. 3: Schema del perceptrone

Il neurone possiede una *funzione di attivazione "A"* e una *funzione di trasferimento "T"*: la prima serve a calcolare il *potenziale di attivazione* del neurone, corrispondente alla somma di tutti gli ingressi moltiplicati per i rispettivi pesi sinaptici; la seconda, invece, serve a calcolare il valore del

segnale di uscita sulla base del potenziale di attivazione ed è scelta in funzione delle grandezze numeriche da trattare; nel perceptrone originale era usata la *funzione di trasferimento a gradino*, la stessa che utilizzeremo in questo ambito.

Con queste semplici premesse è possibile scrivere la formula d'uscita del perceptrone:

$$y = T\left(\sum_{i=1}^n w_i x_i - \theta\right)$$

Fig. 4: Uscita del perceptrone

dove $[x_1, x_2, \dots, x_n]$ sono gli ingressi, $[w_1, w_2, \dots, w_n]$ sono i rispettivi pesi sinaptici e y è il valore dell'uscita; infine θ rappresenta una soglia caratteristica del neurone, detta *soglia di lavoro*.

La *soglia di lavoro* può essere considerata anche come il peso sinaptico di un ingresso fittizio "n+1" con valore unitario negativo, detto comunemente *Bias*. In questo modo la formula di uscita del perceptrone può essere riscritta come segue:

$$y = T\left(\sum_{i=1}^{n+1} w_i x_i\right)$$

Fig. 5: Uscita del perceptrone con l'ingresso fittizio del Bias

Come è facile intuire, questa rete neurale artificiale è in realtà un *classificatore* in grado di svolgere semplici problemi di separazione lineare.

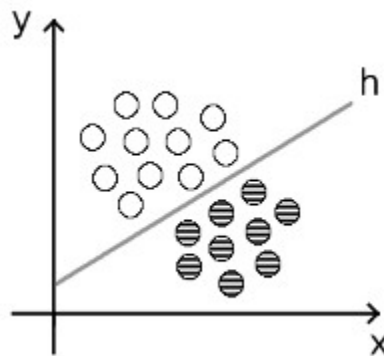


Fig. 6: Esempio di classificazione lineare in cui gli elementi di un insieme vengono ripartiti in due classi di appartenenza distinte

Scegliendo ad esempio la funzione di trasferimento $T(x) = \begin{cases} 1, & \text{se } x > 0 \\ 0, & \text{se } x \leq 0 \end{cases}$, che spinge il perceptrone ad attivarsi solo quando il suo potenziale di attivazione è maggiore di zero, l'uscita della rete sarà data dalla seguente espressione:

$$\sum_{i=1}^{n+1} w_i x_i = w_1 x_1 + w_2 x_2 + \dots + w_n x_n - \theta > 0$$

Fig. 7: Uscita del perceptrone con funzione di trasferimento booleana

Se consideriamo gli ingressi $[x_1, x_2, \dots, x_n]$ come le coordinate di un punto "P" di un iperspazio ad "n" dimensioni e i pesi $[w_1, w_2, \dots, w_n]$ come delle costanti numeriche, tale espressione identifica un *iperpiano*⁸ (che diventa un piano per $n=3$ ed una retta per $n=2$) che suddivide lo spazio degli ingressi in due semispazi distinti.

⁸ Un iperpiano è un'estensione della figura bidimensionale della retta e di quella tridimensionale del piano, una figura con più di tre dimensioni.

$$w_1 x_1 + w_2 x_2 + \dots + w_n x_n - \theta = 0$$

Fig. 8: Iperpiano dell'uscita del perceptrone

Questo significa che, scegliendo dei pesi sinaptici appropriati, è possibile fare in modo che il neurone si attivi solo quando il punto "P" è situato in una determinata regione dello spazio degli ingressi.

Per capire meglio come opera il perceptrone analizziamo ora una semplice rete neurale a due ingressi ed una sola uscita in grado di svolgere la *forma logica OR*⁹.

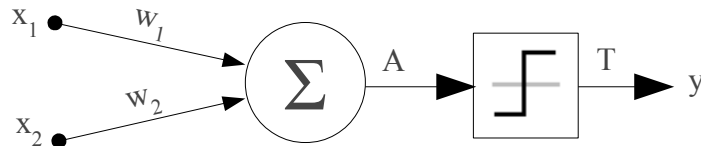


Fig. 9: Perceptrone a due ingressi

La sua uscita è data dalla solita espressione in Fig 7 che, in questo caso specifico, individua un semipiano nello spazio *bidimensionale* degli ingressi, corrispondente all'area di attivazione del neurone.

$$\sum_{i=1}^{n+1} w_i x_i = w_1 x_1 + w_2 x_2 - \theta > 0$$

Fig. 10: Uscita del perceptrone a due ingressi

Realizzare la forma logica *OR* attraverso il perceptrone significa fare in modo che esso restituisca un uscita alta quando almeno uno dei suoi ingressi è di valore "1". Nella pratica questo significa far corrispondere il semipiano di attivazione del neurone ai punti dello spazio degli ingressi in cui si desidera che l'uscita sia alta.

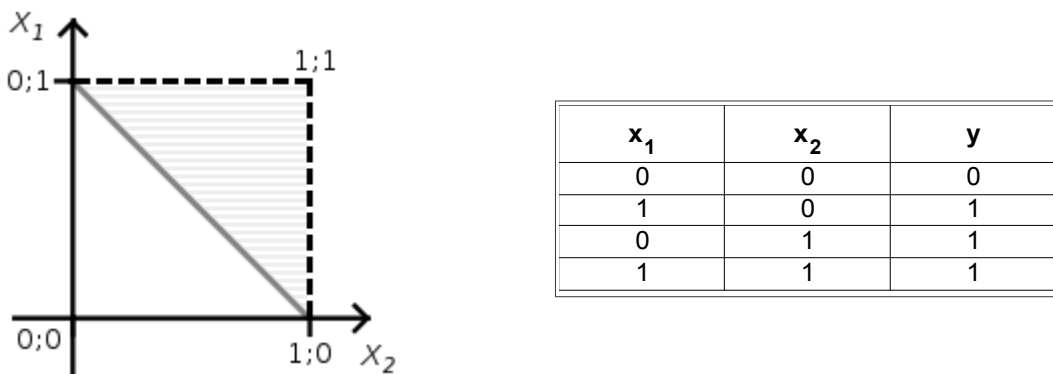


Fig. 11: Separazione lineare dell'operatore OR nello spazio degli ingressi e tabella riassuntiva dell'operatore

La retta individuata graficamente in Fig 11, ad esempio, soddisfa perfettamente il problema dell'operatore *OR* in quanto isola i punti $(0;1)$, $(1;0)$ e $(1;1)$, corrispondenti alle aree in cui il neurone deve attivarsi e restituire un'uscita alta.

Vista la semplicità della rete possiamo ricavare i pesi sinaptici in modo analitico partendo dalla sua formula di uscita, che corrisponde all'*equazione implicita di una retta*:

$$w_1 x_1 + w_2 x_2 - \theta = 0$$

Fig. 12: Retta di uscita del perceptrone

⁹ L'operatore logico *OR* restituisce 1 se almeno uno degli elementi è 1, mentre restituisce 0 in tutti gli altri casi.

Sarà sufficiente trasformarla nella sua *forma esplicita*:

$$x_2 = -\frac{w_1}{w_2} x_1 + \frac{\theta}{w_2}$$

Fig. 13: Retta dell'uscita del perceptrone (forma esplicita)

Quindi utilizziamo un sistema a due equazioni e tre incognite per ricavare i parametri della retta passante per i punti (0;1) e (1;0), coincidenti con i pesi sinaptici della nostra rete:

$$\begin{cases} 0 = -\frac{w_1}{w_2} \cdot 1 + \frac{\theta}{w_2} \\ 1 = -\frac{w_1}{w_2} \cdot 0 + \frac{\theta}{w_2} \end{cases} \quad \Rightarrow \quad w_1 = w_2 = \theta \neq 0$$

Le soluzioni del sistema ci indicano che è possibile far funzionare la rete neurale scegliendo dei pesi sinaptici eguali tra loro e diversi da zero, ad esempio $w_1 = w_2 = \theta = 1$; in realtà, per fare in modo che i punti (0;1) e (1;0) siano compresi nella soluzione, è necessario utilizzare una soglia di lavoro leggermente inferiore, visto che il neurone si attiva con un potenziale di attivazione maggiore di zero e non maggiore o uguale a zero. La *soglia di lavoro*, infatti, influisce sulla posizione verticale della retta rispetto all'origine, come è possibile notare dalla formula di uscita.

Un esempio funzionante di perceptrone che risolve la forma logica OR potrebbe essere il seguente:

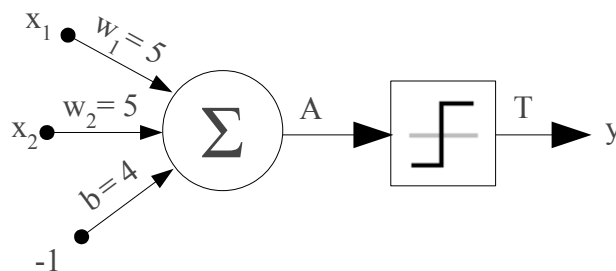


Fig. 14: Perceptrone dell'operator OR con Bias

la cui uscita è determinata dall'espressione:

$$y = (5x_1 + 5x_2 - 4 > 0)$$

Fig. 15: Uscita del perceptrone dell'operatore OR

Non ci resta quindi che mettere alla prova le capacità della nostra rete neurale provando tutte le possibili combinazioni degli ingressi e analizzando i risultati in uscita:

x_1	x_2	$5x_1 + 5x_2 - 4 > 0$	y
0	0	$5 \cdot 0 + 5 \cdot 0 - 4 > 0$	0
1	0	$5 \cdot 1 + 5 \cdot 0 - 4 > 0$	1
0	1	$5 \cdot 0 + 5 \cdot 1 - 4 > 0$	1
1	1	$5 \cdot 1 + 5 \cdot 1 - 4 > 0$	1

Fig. 16: Esecuzione del perceptrone dell'operatore OR

Ovviamente possono esistere diverse combinazioni dei pesi sinaptici che risolvono il medesimo problema, questo perché più rette possono isolare le aree della soluzione; di seguito sono forniti alcuni esempi di rette in grado di risolvere la forma logica OR:

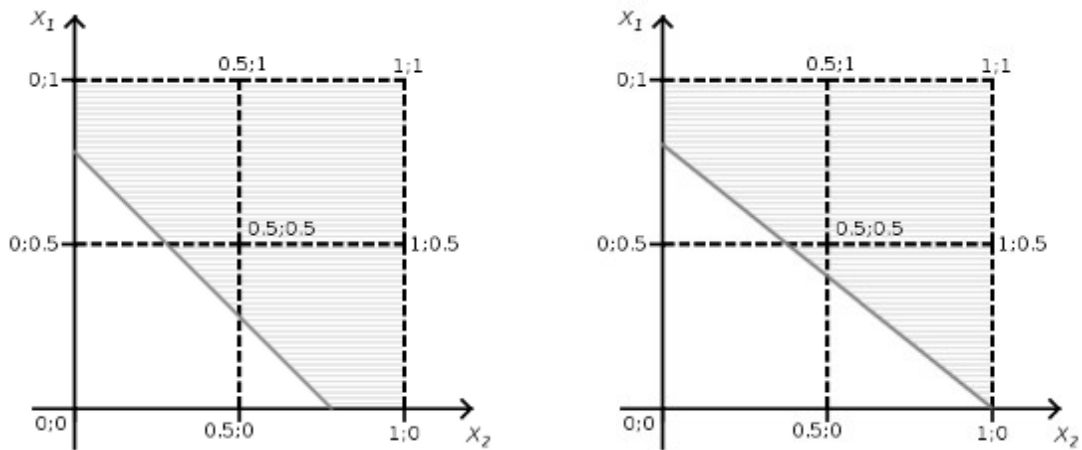


Fig. 17: Separazione lineare dell'operatore OR

0x03:02 Il perceptrone (codice)

Realizzare un perceptrone virtuale attraverso un linguaggio di programmazione strutturata è una procedura abbastanza semplice: si può rispettare la forma matematica, descrivendo i parametri del neurone come variabili (o vettori) e le funzioni di attivazione e di trasferimento come procedure e funzioni strutturate equivalenti.

Di seguito trovate una semplice implementazione in linguaggio C++:

```

1 #include <iostream>
2 #include <cstdio>
3 #include <cstdlib>
4
5 // Precisione dei valori
6 typedef double T_Precision;
7
8 T_Precision
9 activation_function( const T_Precision *x, const T_Precision *w, size_t n ) {
10
11     // Potenziale di attivazione
12     T_Precision potential = 0;
13
14     // Iteratore
15     size_t i = 0;
16
17     // Calcolo il potenziale di attivazione del perceptrone
18     for ( ; i <= n; i++ ) {
19
20         potential += x[i] * w[i];
21     }
22
23     return potential;
24 }
25
26 T_Precision
27 transfer_function( const T_Precision *x, const T_Precision *w, size_t n ) {
28
29     // Calcolo il potenziale di attivazione del perceptrone
30     T_Precision potential = activation_function( x, w, n );
31
32     // Calcolo l'uscita del perceptrone sulla base del potenziale di attivazione

```

```

33     return (T_Precision) ( potential > 0 );
34 }
35
36 int main( void ) {
37
38     // Numero di ingressi
39     const size_t n = 2;
40
41     // Ingressi del percettrone (più il bias)
42     T_Precision x[n + 1];
43     // Pesi sinaptici (più il peso del bias)
44     T_Precision w[n + 1];
45     // Uscita del percettrone
46     T_Precision y;
47
48     // Iteratorii
49     size_t i, j;
50
51     // Imposto il valore negativo unitario del bias
52     x[n] = -1;
53
54     // Imposto il valore dei pesi
55     w[0] = 5;
56     w[1] = 5;
57     w[2] = 4;
58
59     // Stampo l'intestazione della tabella
60     printf( "[x1]\t[x2]\t[y]\n" );
61
62     // Provo tutte le combinazioni degli ingressi
63     for ( j = 0; j <= 1; j++ ) {
64
65         for ( i = 0; i <= 1; i++ ) {
66
67             // Imposto gli ingressi
68             x[0] = i;
69             x[1] = j;
70
71             // Calcolo l'uscita del percettrone
72             y = transfer_function( x, w, n );
73
74             // Stampo gli ingressi e la rispettiva uscita
75             printf( "%.0f\t%.0f\t%.0f\n", x[0], x[1], y );
76         }
77     }
78
79     return 0;
80 }
81

```

Il percettrone è stato scomposto in due *vettori* ed una *variabile*: il primo vettore contiene gli ingressi, il secondo i pesi sinaptici, mentre la variabile è utilizzata per contenere il valore di uscita.

La funzione di trasferimento è la funzione a gradino vista nel paragrafo precedente.

Il sorgente è abbastanza intuitivo e non credo occorra dilungarsi ulteriormente.

0x03:03 L'apprendimento del percettrone

Per un percettrone “apprendere” significa modificare progressivamente i propri pesi sinaptici fino al raggiungimento della risposta desiderata. Fu lo stesso Rosenblatt, nel 1962, a proporre un algoritmo in grado di tarare automaticamente i pesi sinaptici sulla base dei dati sperimentali. La “*regola delta*”, questo è il nome dell'algoritmo, rientra tra i paradigmi di apprendimento supervisionato e

cerca di minimizzare l'errore della rete attraverso una procedura di *discesa del gradiente*¹⁰.

Dato un insieme di addestramento non vuoto formato da “S” coppie ingressi-uscite, si definisce una *funzione di errore* (o *funzione costo*) “E” corrispondente all'errore quadratico medio tra l'uscita reale “ y_c ” e l'uscita desiderata “ d_c ” di ogni esempio, dove “c” è appunto l'indice dell'esempio:

$$E = \sum_{c=1}^S \frac{1}{2} (d_c - y_c)^2$$

Fig. 18: Funzione di errore del perceptrone

Per minimizzare l'errore della rete e fare in modo che essa risponda correttamente agli stimoli è necessario trovare il *minimo*¹¹ di questa funzione. Scendere lungo il gradiente di una funzione significa partire da un punto scelto a caso nel suo spazio multidimensionale e spostarsi fino a convergere sul *minimo locale* più vicino. Il *gradiente* di una funzione, infatti, indica la direzione di massima crescita (o di massima decrescita) della funzione a partire da un determinato punto.

Il gradiente, in questo caso, è il vettore che ha per componenti le derivate parziali della funzione costo rispetto a tutti i pesi sinaptici della rete:

$$\nabla E(\mathbf{w}) = \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \dots, \frac{\partial E}{\partial w_n} \right]$$

Fig. 19: Gradiente della funzione di errore

Dal gradiente si ricava la *regola di discesa del gradiente*, utilizzata per la correzione dei pesi sinaptici del perceptrone:

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}; \quad w_i^t = w_i^{t-1} + \Delta w_i$$

Fig. 20: Regola della discesa del gradiente applicata al perceptrone

dove “t” è l'istante dell'addestramento, “ Δw_i ” rappresenta la modifica da applicare al peso sinaptico “ w_i ” e dove η è una costante chiamata *learning rate* (o *tasso di apprendimento*) positiva e minore di uno che determina la velocità di apprendimento del neurone. La derivata parziale determina la rapidità di crescita o decrescita dell'errore sul singolo peso “ w_i ”, in base alla pendenza della funzione costo.

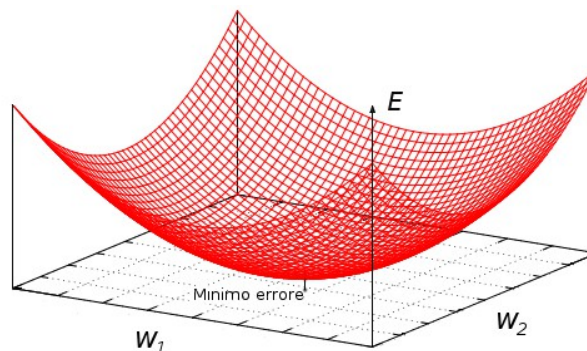


Fig. 21: Superficie dell'errore del perceptrone a due ingressi

10 Il *gradiente* di una funzione è il vettore che ha come componenti le derivate parziali della funzione rispetto a tutte le variabili indipendenti calcolate in un determinato punto.

11 Il *minimo* di una funzione è il più piccolo dei valori che essa può assumere al variare dei suoi parametri.

Se l'errore cresce con l'aumentare dei pesi sinaptici, questi vengono leggermente diminuiti e viceversa, fino al raggiungimento del minimo locale più vicino.

Usare un tasso di apprendimento troppo piccolo renderà l'addestramento molto lungo; con un tasso di apprendimento troppo grande, invece, non sarà possibile convergere sul minimo locale della funzione costo, che verrà saltato a causa della correzione troppo marcata.

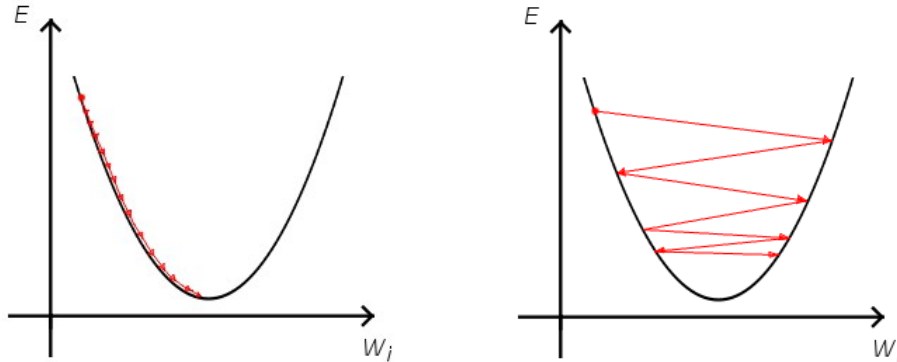


Fig. 22: Tasso di apprendimento troppo piccolo o troppo grande

Applicando la regola di *derivazione delle funzioni composte*¹² otteniamo che:

$$\frac{\partial E}{\partial w_i} = \frac{\partial E}{\partial y} \frac{\partial y}{\partial w_i}$$

Fig. 23: Sviluppo della regola di discesa del gradiente

Risolvendo le due derivate parziali:

$$\frac{\partial E}{\partial y} = \frac{\partial \left[\frac{1}{2} (d - y)^2 \right]}{\partial y} = \frac{1}{2} \cdot [2 \cdot (d - y) \cdot (-1)] = -(d - y)$$

$$\frac{\partial y}{\partial w_i} = \frac{\partial T \left(\sum_{i=1}^{n+1} w_i x_i \right)}{\partial w_i} = \frac{\partial \sum_{i=1}^{n+1} w_i x_i}{\partial w_i} = x_i$$

Fig. 24: Sviluppo della regola di discesa del gradiente

Avremo che:

$$\frac{\partial E}{\partial w_i} = -(d - y) x_i$$

Fig. 25: Derivata dell'errore sul peso "w_i"

Che, considerando tutti gli esempi dell'insieme di addestramento (modalità *batch*), diventa:

$$\frac{\partial E}{\partial w_i} = \sum_{c=1}^S [-(d_c - y_c) x_{ci}]$$

Fig. 26: Derivata dell'errore sul peso "w_i", considerando tutti gli esempi dell'insieme di addestramento

¹² La derivata di una funzione composta è data dal prodotto delle derivate delle funzioni elementari.

Infine, riscrivendo la formula di discesa del gradiente, otteniamo la *regola delta*, utilizzata per l'apprendimento del perceptrone con funzione di trasferimento a gradino:

$$\Delta w_i = -\eta \sum_{c=1}^S [-(d_c - y_c) x_{ci}]; \quad w_i^t = w_i^{t-1} + \Delta w_i$$

Fig. 27: La regola delta del perceptrone

Come verifica di tutto quello che abbiamo appena scritto, proviamo ad addestrare un perceptrone a due ingressi ed un a sola uscita affinché impari a svolgere la forma logica AND.

Innanzitutto prepariamo un insieme di addestramento come quello in Fig. 28.

x_1	x_2	y
0	0	0
1	0	0
0	1	0
1	1	1

Fig. 28: Tabella dell'operatore AND

Quindi usiamo la *regola delta* sulla rete fino a quando l'errore convergerà su un livello trascurabile.

Epoche	w_1	$\partial E/\partial w_1$	w_2	$\partial E/\partial w_2$	w_b	$\partial E/\partial w_b$	E
0	7,00	-1,00	2,00	-1,00	15,00	1,00	0,50
1	7,50	-1,00	2,50	-1,00	14,50	1,00	0,50
2	8,00	-1,00	3,00	-1,00	14,00	1,00	0,50
3	8,50	-1,00	3,50	-1,00	13,50	1,00	0,50
4	9,00	-1,00	4,00	-1,00	13,00	1,00	0,50
5	9,50	-1,00	4,50	-1,00	12,50	1,00	0,00

Fig. 29: Addestramento di un perceptrone per la forma logica AND

Ogni ciclo di apprendimento della rete neurale è detto "epoca" e corrisponde ad un piccolo passo verso la risposta corretta.

A questo punto non ci resta che verificare in modo empirico se il perceptrone ha imparato ad eseguire correttamente la forma logica AND:

x_1	x_2	$9,5 x_1 + 4,5 x_2 - 12,5 > 0$	y
0	0	$9,5 \cdot 0 + 4,5 \cdot 0 - 12,5 > 0$	0
1	0	$9,5 \cdot 1 + 4,5 \cdot 0 - 12,5 > 0$	0
0	1	$9,5 \cdot 0 + 4,5 \cdot 1 - 12,5 > 0$	0
1	1	$9,5 \cdot 1 + 4,5 \cdot 1 - 12,5 > 0$	1

Fig. 30: Esecuzione del perceptrone dell'operatore AND

0x03:04 L'apprendimento del perceptrone (codice)

Per aggiungere l'addestramento supervisionato al perceptrone virtuale visto in precedenza è necessario definire un'apposita funzione:

```

1 #include <iostream>
2 #include <cstdio>
3 #include <cstdlib>
4 #include <ctime>
5
6 // Precisione dei valori
7 typedef double T_Precision;
8

```



```

...
36
37 T_Precision
38 training_function( T_Precision *x, T_Precision *w, size_t n,
39                   const T_Precision *dx, const T_Precision *dy, size_t sn,
40                   const T_Precision eta, const T_Precision desired_error ) {
41
42     // Uscita calcolata
43     T_Precision y;
44
45     // Errore della rete
46     T_Precision error;
47
48     // Errore dell'uscita
49     T_Precision dEdy;
50
51     // Errore dei pesi sinaptici
52     T_Precision dEdw[n];
53
54     // Modifica del peso sinaptico
55     T_Precision deltaw;
56
57     // Contatore delle epoche
58     size_t epochs = 0;
59
60     // Iteratore
61     size_t i, t;
62
63     // Log di lavoro
64     printf( "Inizio l'addestramento (eta = %.2f, errore desiderato = %f).\n", eta,
desired_error );
65
66     // Inizializzo i pesi sinaptici con dei valori casuali
67     for ( i = 0; i < n; i++ ) {
68
69         w[i] = rand() % 10;
70     }
71
72     // Continuo l'addestramento finché non raggiungo
73     // l'errore desiderato (max 100 epoche)
74     do {
75
76         // Stampo il numero dell'epoca corrente
77         printf( "Epoca #%zu: ", epochs );
78
79         // Azzero l'errore della rete
80         error = 0.0;
81
82         // Azzero gli errori dei pesi sinaptici
83         for ( i = 0; i <= n; i++ ) {
84
85             dEdw[i] = 0;
86         }
87
88         // Ripeto per tutti gli esempi nell'insieme di addestramento
89         for ( t = 0; t < sn; t++ ) {
90
91             // Prendo gli ingressi dall'esempio
92             for ( i = 0; i < n; i++ ) {
93
94                 x[i] = dx[t * n + i];
95             }
96
97             // Calcolo l'uscita del percettrone
98             // con gli ingressi dell'esempio
99             y = transfer_function( x, w, n );

```

```

100
101 // Calcolo l'errore sull'uscita
102 dEdy = -(dy[t] - y);
103
104 // Calcolo l'errore dei pesi sinaptici
105 for ( i = 0; i <= n; i++ ) {
106     dEdw[i] += dEdy * x[i];
107 }
108
109 // Calcolo il quadrato dell'errore necessario
110 // per trovare l'errore quadratico medio
111 // E(x) = SUM( e^2 ) / n_samples
112 error += ( dEdy * dEdy ) / 2.0;
113 }
114
115 // Correggo i pesi usando la regola delta
116 for ( i = 0; i <= n; i++ ) {
117     // Calcolo la modifica del peso
118     deltaw = - eta * dEdw[i];
119
120     // Applico la modifica al peso sinaptico
121     w[i] = w[i] + deltaw;
122
123     // Stampa il nuovo valore del peso
124     printf( "w[%zu]=%.1f (E=%.1f), ", i, w[i], dEdw[i] );
125 }
126
127 // Termino la riga di log aggiungendo l'errore corrente
128 printf( "MSE=%f\n", error );
129
130 // Incremento il numero delle epoche
131 epochs++;
132
133 } while ( error > desired_error && epochs < 100 );
134
135 // Log di lavoro
136 printf( "Addestramento terminato dopo %zu epoche.\n\n", epochs );
137 }
138
139 int main( void ) {
140
141     // Inizializzo il generatore di numeri pseudocasuali
142     srand( (size_t) time( NULL ) );
143
144     // Numero di ingressi
145     const size_t n = 2;
146
147     // Ingressi del perceptrone (più il bias)
148     T_Precision x[n + 1];
149     // Pesi sinaptici (più il peso del bias)
150     T_Precision w[n + 1];
151     // Uscita del perceptrone
152     T_Precision y;
153
154     // Iterator
155     size_t i, j;
156
157     // Imposto il valore negativo unitario del bias
158     x[n] = -1;
159
160     // Numero di esempi dell'insieme di addestramento
161     const size_t sn = 4;
162
163     // Preparo l'insieme di addestramento per l'operatore AND
164
165

```

```

166     const T_Precision dx[sn * n] = {
167                                     0, 0,
168                                     0, 1,
169                                     1, 0,
170                                     1, 1
171     };
172
173     const T_Precision dy[sn] = { 0, 0, 0, 1 };
174
175     // Tasso di apprendimento
176     const T_Precision eta = 0.5;
177
178     // Errore desiderato
179     const T_Precision desired_error = 0.00001;
180
181     // Addestrò il perceptrone
182     training_function( x, w, n, dx, dy, sn, eta, desired_error );
183
184     // Stampo l'intestazione della tabella
185     printf( "[x1]\t[x2]\t[y]\n" );
186
187     // Provo tutte le combinazioni degli ingressi
188     for ( j = 0; j <= 1; j++ ) {
189         for ( i = 0; i <= 1; i++ ) {
190             // Imposto gli ingressi
191             x[0] = i;
192             x[1] = j;
193
194             // Calcolo l'uscita del perceptrone
195             y = transfer_function( x, w, n );
196
197             // Stampo gli ingressi e la rispettiva uscita
198             printf( "%.0f\t%.0f\t%.0f\n", x[0], x[1], y );
199         }
200     }
201
202     return 0;
203 }
204
205 }
206

```

Come il codice precedente anche questo dovrebbe essere abbastanza intuitivo.

La funzione *training_function* è quella che permette al perceptrone di apprendere in modo supervisionato. In essa i pesi sinaptici vengono inizializzati con dei valori pseudocasuali (attraverso la funzione *rand()*) e man mano calibrati sulla base degli esempi forniti, attraverso la *regola delta* vista precedentemente.

I dati del *training set* sono stati raccolti in un unico vettore monodimensionale per una questione di praticità e per facilitarne il passaggio alle funzioni di lavoro. L'accesso agli elementi del vettore, in ogni caso, è gestito attraverso l'uso di due indici, come si può vedere nell'equiparazione sottostante:

$$\begin{aligned}
 \text{matrice}[\text{colonne}][\text{righe}] &\rightarrow \text{matrice}[\text{colonne} * \text{righe}] \\
 \text{matrice}[i][j] &\rightarrow \text{matrice}[i * \text{righe} + j]
 \end{aligned}$$

0x03:05 Il perceptrone multistrato

I limiti del perceptrone diventano evidenti appena si cerca di risolvere un problema di *separazione non lineare*. Prendiamo ad esempio la forma logica *XOR* (o *esclusive-OR*): questa forma non è linearmente separabile in quanto non vi è modo di isolare i punti (1;0) e (0;1) dello spazio degli ingressi attraverso un'unica retta (vedi la Fig. 31). La separazione sarebbe possibile solo se si utilizzasse una funzione di attivazione non lineare, ma questa renderebbe il perceptrone estremamente più complesso.

In realtà esiste un modo per svolgere problemi non lineari mantenendo la linearità propria del perceptrone.

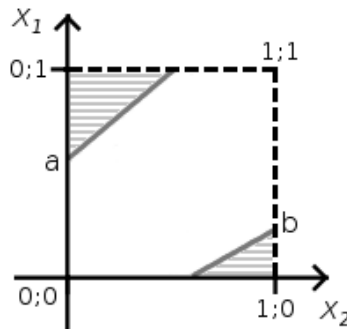


Fig. 31: Separazione della forma XOR attraverso due rette distinte

Utilizzando più perceptron interconnessi, infatti, è possibile aumentare il numero di rette che operano sullo spazio degli ingressi, scomponendo il problema di separazione non lineare in più sotto-problemi di lineari facilmente risolvibili dal perceptrone.

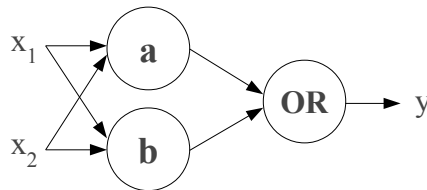


Fig. 32: Rete a tre perceptron dell'operatore XOR

Nel caso specifico dell'operatore *XOR*, ad esempio, potremmo utilizzare tre perceptron ripartiti in due strati, in modo tale da isolare i punti (0;1) e (1;0) con i primi due e, con l'ultimo, controllare se vi è stata l'attivazione di almeno uno degli altri due perceptron (forma logica OR).

Il problema, scomposto in forme linearmente separabili, apparirebbe così:

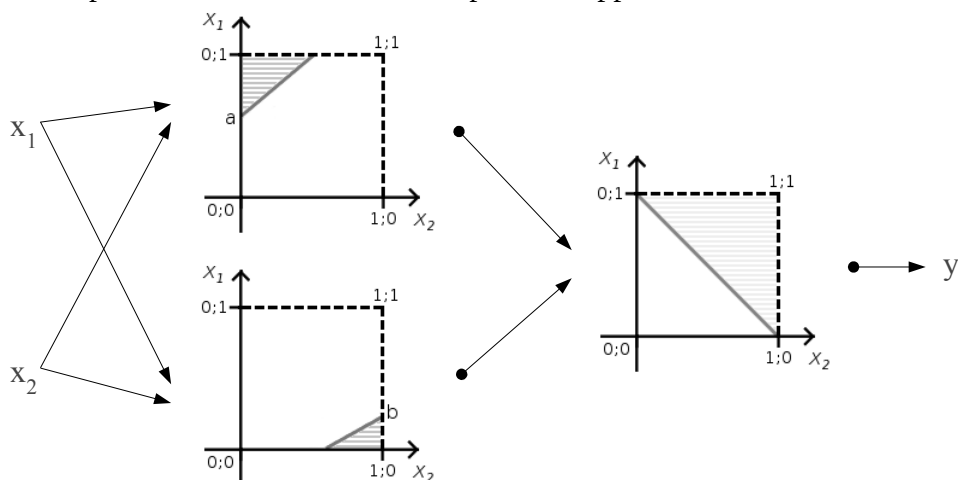


Fig. 33: Separazioni lineari necessarie a svolgere la forma logica XOR

Da questa idea nascono le *reti di perceptron a più strati* (o *Multilayer Perceptron*, o ancora *MLP*) composte tipicamente da uno strato di ingresso, uno o più strati intermedi (strati nascosti) ed uno strato di uscita, tutti completamente interconnessi.

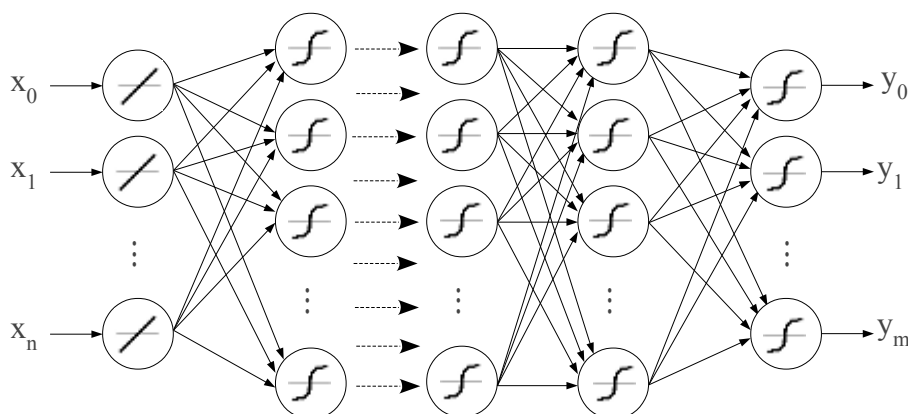


Fig. 34: Rete di perceptron a più strati

In questa nuova architettura le uscite dei perceptron di uno strato diventano gli ingressi dei perceptron dello strato successivo (rete *feedforward*), ad eccezione dell'ultimo dove le uscite dei perceptron corrispondono alle uscite effettive della rete. Ogni strato ha inoltre un nodo fittizio (*Bias*) avente come valore di uscita “-1”, che è comune a tutti i neuroni dello strato successivo.

I perceptron degli strati intermedi e quelli dello strato di uscita in genere utilizzano una funzione di trasferimento *sigmoide*, mentre i perceptron dello strato di ingresso fungono da “*buffer*” per gli ingressi della rete ed utilizzano una funzione di trasferimento a rampa, utilizzata per limitare l'intervallo dei valori che essi possono assumere; per questo motivo ogni perceptrone dello strato di ingresso accetta un un unico valore corrispondente ad un determinato ingresso della rete.

Utilizzando la funzione di trasferimento sigmoide l'uscita dei perceptron dell'ultimo strato della rete sarà data dalla seguente formula:

$$\text{sigmoid} \left(\sum_{i=1}^{n+1} w_i x_i = w_1 x_1 + w_2 x_2 + \dots + w_n x_n - \theta \right)$$

Fig. 35: Uscita del perceptrone con funzione di trasferimento sigmoide

Tale espressione identifica una *sigmoide* nello spazio degli ingressi, orientata secondo i pesi sinaptici del perceptrone.

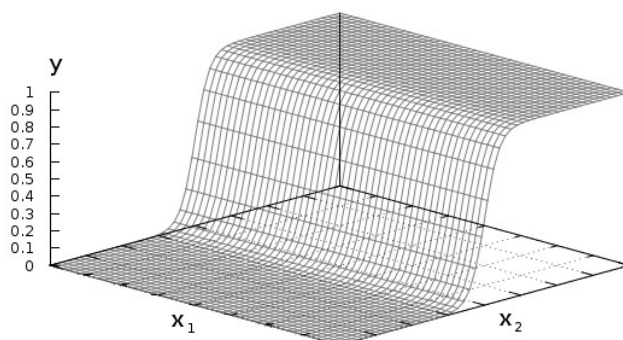
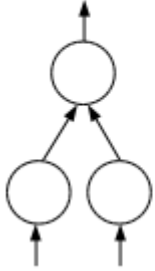

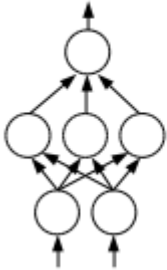

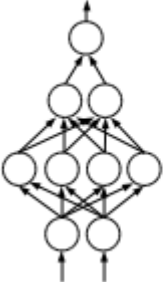



Fig. 36: Uscita del perceptrone a due ingressi con funzione di trasferimento sigmoide

Per calcolare le uscite di questo tipo di rete neurale si procede ordinatamente: prima si calcolano le uscite dei perceptron dello strato di ingresso, poi quelle dello strato successivo e così via fino all'ultimo strato, le cui uscite corrispondono a quelle effettive della rete.

Man mano che il problema da affrontare diventa più complesso è possibile aumentare il numero degli strati intermedi che compongono la rete al fine di incrementarne la potenza di calcolo. Tuttavia, secondo il *teorema di Kolmogorov*¹³ (1957), una rete di perceptroni con un unico strato nascosto dovrebbe essere in grado di rappresentare qualsiasi funzione continua e di approssimarla con qualsiasi grado di precisione semplicemente aumentando il numero dei neuroni che compongono lo strato. Dico “dovrebbe” perché tale teorema applicato alle reti neurali è tutt’oggi oggetto di dibattito. In ogni caso esso non fornisce alcuna indicazione su quale debba essere la struttura effettiva delle reti neurali per poter risolvere un determinato tipo problema.

Per avere almeno un punto di riferimento sulla topologia di rete da adottare possiamo classificare le diverse architetture in base ai problemi che esse sono sicuramente in grado di svolgere:

Struttura	Area di decisione	Tipo di problemi
0 strati intermedi 	Semipiano 	Separazione lineare Classificazione di aree lineari
1 strato intermedio 	Regioni convesse 	Separazione non lineare <i>(limitata alle figure convesse)</i> Classificazione di forme convesse
2 strati intermedi 	Regioni complesse 	Separazione non lineare Classificazione di forme complesse

In particolare l'ultima configurazione, caratterizzata da due strati nascosti, è in grado di isolare qualunque regione complessa nello spazio degli ingressi. Possiamo generalizzarne il funzionamento dicendo che:

- il *primo strato nascosto* partendo dal basso divide lo spazio degli ingressi in varie regioni delimitate da iperpiani ad n-dimensioni (uno per perceptrone);

¹³ Tale teorema afferma che qualsiasi funzione reale continua definita in un sistema “n” dimensionale può essere decomposta come la somma di più funzioni che hanno come loro argomento somme di funzioni continue a variabile singola.

- il *secondo*, successivamente, riconosce le eventuali intersezioni tra queste regioni, attraverso un'operazione di AND logico;
- l'*ultimo strato*, infine, effettua ulteriori unioni tra le precedenti intersezioni, attraverso un'operazione di OR logico.

Di seguito un esempio di elaborazione di una rete di perceptron avente due strati nascosti:

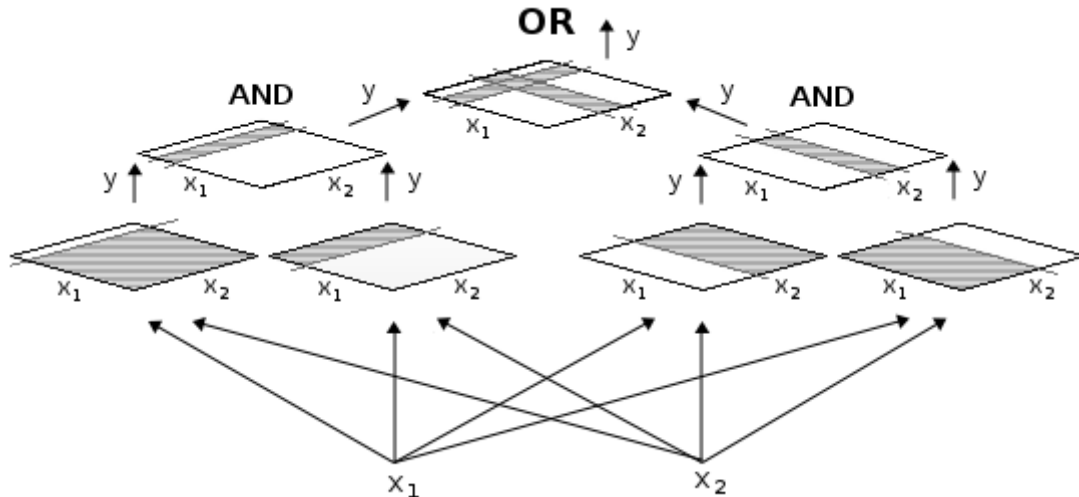


Fig. 37: Superficie di decisione non lineare ottenuta con una rete di perceptron con due strati nascosti

La rete in questione prende in ingresso due valori corrispondenti alle coordinate di un punto nello spazio bidimensionale degli ingressi, quindi, attraverso un'elaborazione in cascata, verifica se tale punto è situato nell'area decisionale della rete, delimitata dalle separazioni lineari effettuate dai perceptron e, in questo caso, rappresentata da una figura cruciforme.

0x03:06 Il perceptrone multistrato (codice)

Il seguente sorgente implementa una rete di perceptron a tre strati:

```

1  #include <iostream>
2  #include <cstdio>
3  #include <cstdlib>
4  #include <cmath>
5
6  #define __POW2__(x)      (x * x)
7
8  #define __SIGMOID__(x)  1.0 / ( 1.0 + std::exp(-x) )
9  #define __D_SIGMOID__(x) x * ( 1 - x )
10
11 // Precisione dei valori
12 typedef double T_Precision;
13
14 // Struttura di un neurone
15 struct Neuron {
16
17     T_Precision value; // Uscita del perceptrone
18 };
19
20 // Struttura di una connessione tra due neuroni (sinapsi)
21 struct Synapse {
22
23     T_Precision weight; // Peso della connessione
24 };
25

```

```

26
27 void
28 run_network( Neuron *i_neuron, size_t input_size,
29             Neuron *h_neuron, Synapse *h_synapse, size_t hidden_size,
30             Neuron *o_neuron, Synapse *o_synapse, size_t output_size ) {
31
32     // Potenziale di attivazione
33     float potential;
34
35     // Iteratori
36     size_t j, i;
37
38     // Calcolo le uscite dello strato intermedio
39     for ( i = 0; i < hidden_size; i++ ) {
40
41         // Azzero il potenziale di attivazione
42         potential = 0;
43
44         // Calcolo il potenziale di attivazione
45         for ( j = 0; j < input_size; j++ ) {
46
47             potential += i_neuron[j].value * h_synapse[i * ( input_size + 1 ) +
48 j].weight;
49         }
50
51         // Aggiungo il valore del bias
52         potential += h_synapse[i * ( input_size + 1 ) + j].weight;
53
54         // Calcolo l'uscita del neurone sulla base del potenziale di attivazione
55         h_neuron[i].value = __SIGMOID__( potential );
56     }
57
58     // Calcolo le uscite dello strato di uscita
59     for ( i = 0; i < output_size; i++ ) {
60
61         // Azzero il potenziale di attivazione
62         potential = 0;
63
64         // Calcolo il potenziale di attivazione
65         for ( j = 0; j < hidden_size; j++ ) {
66
67             potential += h_neuron[j].value * o_synapse[i * ( hidden_size + 1 ) +
68 j].weight;
69         }
70
71         // Aggiungo il valore del bias
72         potential += o_synapse[i * ( hidden_size + 1 ) + j].weight;
73
74         // Calcolo l'uscita del neurone sulla base del potenziale di attivazione
75         o_neuron[i].value = __SIGMOID__( potential );
76     }
77
78 int main( void ) {
79
80     // Dimensioni della rete
81     const size_t input_size = 2;
82     const size_t hidden_size = 4;
83     const size_t output_size = 1;
84
85     // Creo i neuroni degli strati
86     Neuron i_neuron[ input_size ];
87     Neuron h_neuron[ hidden_size ];
88     Neuron o_neuron[ output_size ];
89
90     // Creo le sinapsi degli strati (+ quelle del bias)

```



```

90 Synapse h_synapse[ (input_size + 1) * hidden_size ];
91 Synapse o_synapse[ (hidden_size + 1) * output_size ];
92
93
94 // Imposto i pesi sinaptici
95 h_synapse[0].weight = -3.9340242026;
96 h_synapse[1].weight = 8.2505052824;
97 h_synapse[2].weight = 0.3043062447;
98
99 h_synapse[3].weight = 6.8427240303;
100 h_synapse[4].weight = 6.8347133644;
101 h_synapse[5].weight = -1.4695877520;
102
103 h_synapse[6].weight = 8.2647545506;
104 h_synapse[7].weight = -4.1937195208;
105 h_synapse[8].weight = 0.5536933684;
106
107 o_synapse[0].weight = -12.2588589492;
108 o_synapse[1].weight = 13.6915483842;
109 o_synapse[2].weight = -12.2575228405;
110 o_synapse[3].weight = 4.8756901299;
111
112
113 // Iteratore
114 size_t i, j;
115
116 // Stampo l'intestazione della tabella
117 printf( "[x1]\t[x2]\t[y]\n" );
118
119 // Provo tutte le combinazioni degli ingressi
120 for ( j = 0; j <= 1; j++ ) {
121     for ( i = 0; i <= 1; i++ ) {
122         // Imposto gli ingressi
123         i_neuron[0].value = i;
124         i_neuron[1].value = j;
125
126         // Eseguo la rete neurale
127         run_network( i_neuron, input_size,
128                     h_neuron, h_synapse, hidden_size,
129                     o_neuron, o_synapse, output_size );
130
131         // Stampo gli ingressi e la rispettiva uscita
132         printf( "%.0f\t%.0f\t%.0f\n", i_neuron[0].value, i_neuron[1].value,
133               o_neuron[0].value );
134     }
135 }
136
137
138 return 0;
139 }
140

```

Sono state definite due strutture che rappresentano i *percettroni* e le rispettive *sinapsi*.

Le istanze di quest'ultime sono state raccolte in un unico vettore monodimensionale per una questione di praticità e per facilitarne il passaggio alle funzioni di lavoro.

La funzione che fa operare la rete neurale artificiale è *run_network* che calcola le uscite di ogni strato, un percettrone per volta, fino a determinare le uscite della rete.

0x03:07 L' algoritmo di retropropagazione dell'errore

L'algoritmo più usato per l'addestramento delle reti MLP è senza dubbio la *retropropagazione dell'errore* (o *Error Back Propagation*, o ancora *EBP*). Si tratta di un algoritmo di apprendimento supervisionato che estende la *regola delta* vista nel paragrafo precedente alle reti feedforward multistrato con funzioni di trasferimento non lineari (e derivabili); per questo motivo esso è conosciuto anche con il nome di *Regola Delta Generalizzata*.

L'obiettivo dell'addestramento è ancora una volta quello di minimizzare la *funzione costo* della rete, in questo caso definita come l'errore quadratico medio tra tutte le uscite della rete e i rispettivi valori desiderati, contenuti nell'insieme di addestramento.

Tale funzione può essere definita come:

$$E = \sum_{c=1}^S \sum_{j=1}^n \frac{1}{2} (d_{cj} - y_{cj})^2$$

Fig. 38: Funzione di errore della rete di perceptroni

dove “ y_{cj} ” è l'uscita j -esima della rete (corrispondente allo j -esimo perceptrone dello strato di uscita) al momento della presentazione dell'esempio “ c ”; “ d_{cj} ”, invece, è il rispettivo valore desiderato.

$$\nabla E(w) = \left[\dots, \frac{\partial E}{\partial w_{jk}}, \dots \right]$$

Fig. 39: Gradiente della funzione di errore

Attraverso il confronto con i valori dell'insieme di addestramento possiamo conoscere l'errore dei perceptroni dello strato di uscita. Invece non vi è alcuna indicazione su quali debbano essere le uscite dei nodi interni alla rete. Per determinare l'errore di un perceptrone nascosto è allora necessario retropropagare l'errore dallo strato di uscita, in modo da rendere possibile la correzione dei pesi sinaptici.

Prendiamo ad esempio una generica rete di perceptroni a 3 strati, composta da “ n ” neuroni nello strato di ingresso $X_i \rightarrow [i=1, 2, \dots, n]$, “ m ” neuroni nello strato nascosto $Z_k \rightarrow [k=1, 2, \dots, m]$ e “ p ” neuroni in quello di uscita $Y_j \rightarrow [j=1, 2, \dots, p]$.

Per correggere i pesi sinaptici di un perceptrone dello strato di uscita utilizziamo la regola:

$$\Delta w_{jk} = -\eta \frac{\partial E}{\partial w_{jk}}; \quad w_{jk}^t = w_{jk}^{t-1} + \Delta w_{jk}$$

Fig. 40: Regola della discesa del gradiente applicata ai perceptroni dello strato di uscita

dove “ Δw_{jk} ” è il peso k -esimo del perceptrone “ Y_j ”, ossia il peso sinaptico che connette il perceptrone “ Z_k ” a quello “ Y_j ”.

Sviluppando la formula attraverso la *regola di derivazione delle funzioni composte* otteniamo che:

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial P_j} \frac{\partial P_j}{\partial w_{jk}}$$

Fig. 41: Sviluppo della regola di discesa del gradiente

dove “ y_j ” è l'uscita j -esima della rete e “ P_j ” è il potenziale di attivazione del rispettivo nodo.

Quindi risolvendo le derivate parziali otteniamo:

$$\frac{\partial E}{\partial y_j} = \frac{\partial \left[\frac{1}{2} (d_j - y_j)^2 \right]}{\partial y_j} = \frac{1}{2} \cdot [2 \cdot (d_j - y_j) \cdot (-1)] = -(d_j - y_j)$$

$$\frac{\partial y_j}{\partial P_j} = \frac{\partial T(P_j)}{\partial P_j} = T'(P_j)$$

$$\frac{\partial P_j}{\partial w_{jk}} = \frac{\partial \left(\sum_{k=1}^{m+1} w_{jk} z_k \right)}{\partial w_{jk}} = z_k$$

Fig. 42: Sviluppo della regola di discesa del gradiente

dove “ z_k ” è l'uscita k -esima dello strato precedente, regolato dal peso “ w_{jk} ”.

Ricostruendo l'equazione iniziale abbiamo che:

$$\frac{\partial E}{\partial w_{jk}} = -(d_j - y_j) T'(P_j) z_k$$

Fig. 43: Derivata dell'errore sul peso sinaptico

Se poi poniamo:

$$\delta_j = -(d_j - y_j) T'(P_j)$$

Fig. 44: Sviluppo della regola di discesa del gradiente

Otteniamo la regola delta generalizzata per un nodo di uscita:

$$\Delta w_{jk} = -\eta \delta_j z_k$$

Fig. 45: Regola delta generalizzata (per un nodo di uscita)

Quando invece dobbiamo correggere il peso sinaptico di un nodo interno, abbiamo:

$$\frac{\partial E}{\partial w_{ki}} = \frac{\partial E}{\partial z_k} \frac{\partial z_k}{\partial P_k} \frac{\partial P_k}{\partial w_{ki}}$$

Fig. 46: Sviluppo della regola di discesa del gradiente

Dove:

$$\frac{\partial z_k}{\partial P_k} = \frac{\partial T(P_k)}{\partial P_k} = T'(P_k)$$

$$\frac{\partial P_k}{\partial w_{ki}} = \frac{\partial \left(\sum_{i=1}^{n+1} w_{ki} x_i \right)}{\partial w_{ki}} = x_i$$

Fig. 47: Sviluppo della regola di discesa del gradiente

Mentre la derivata dell'errore sull'uscita interna “ z_k ” è ottenuta retropropagando l'errore dallo strato successivo:

$$\frac{\partial E}{\partial z_k} = \sum_{j=1}^{p+1} \left[\frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial P_j} \frac{\partial P_j}{\partial z_k} \right] = \sum_{j=1}^{p+1} \left[-(d_j - y_j) T'(P_j) w_{jk} \right]$$

Fig. 48: Sviluppo della regola di discesa del gradiente

Formula che, come abbiamo visto, può essere riscritta nel seguente modo:

$$\frac{\partial E}{\partial z_k} = \sum_{j=1}^{p+1} \delta_j w_{jk}$$

Fig. 49: Sviluppo della regola di discesa del gradiente

La correzione del peso sinaptico sarà data quindi da una funzione ricorsiva che propaga all'indietro l'errore dello strato successivo:

$$\Delta w_{ki} = -\eta \sum_{j=1}^{p+1} (\delta_j w_{jk}) T'(P_k) x_i$$

Fig. 50: Sviluppo della regola di discesa del gradiente

Infine, ponendo:

$$\delta_k = \sum_{j=1}^{p+1} (\delta_j w_{jk}) T'(P_k)$$

Fig. 51: Sviluppo della regola di discesa del gradiente

Otteniamo una forma uguale a quella utilizzata per correggere i pesi sinaptici di un nodo di uscita:

$$\Delta w_{ki} = -\eta \delta_k x_i$$

Fig. 52: Regola delta generalizzata (per un nodo nascosto)

Possiamo quindi generalizzare la regola di addestramento in questo modo:

$$\Delta w_{ki} = -\eta \delta_k z_i; \quad w_{ki}^t = w_{ki}^{t-1} + \Delta w_{ki}$$

$$\delta_k = \begin{cases} -(d_k - y_k) T'(P_k) & \text{se è un nodo di uscita} \\ \sum_{j=1}^{p+1} (\delta_j w_{jk}) T'(P_k) & \text{se è un nodo interno} \end{cases}$$

Fig. 53: La regola delta generalizzata

E' interessante notare che quando si utilizza una funzione di trasferimento sigmoidale si ha:

$$T'(P_k) = T(P_k) \cdot [1 - T(P_k)] = y_k \cdot (1 - y_k)$$

Fig. 54: Derivata dell'errore sul potenziale di attivazione con una funzione di trasferimento sigmoidale

Per questa proprietà, estremamente favorevole dal punto di vista computazionale, la sigmoide è una delle funzioni di trasferimento più usate.

Ovviamente per aggiustare i pesi sinaptici di un nodo della rete considerando tutti gli esempi dell'insieme di addestramento (modalità *batch*) dobbiamo modificare la formula come segue:

$$\Delta w_{ki} = -\eta \sum_{c=1}^S (\delta_{ck} z_{ci}); \quad w_{ki}^t = w_{ki}^{t-1} + \Delta w_{ki}$$

Fig. 55: La regola delta generalizzata per tutti gli esempi dell'insieme di addestramento

0x03:08 L' algoritmo di retropropagazione dell'errore (codice)

Il seguente sorgente implementa la *regola delta generalizzata* per la rete di perceptroni multistrato vista in precedenza:

```

1  #include <iostream>
2  #include <cstdio>
3  #include <cstdlib>
4  #include <cmath>
5  #include <ctime>
6
7  #define __POW2__(x)      (x * x)
8
9  #define __SIGMOID__(x)  1.0 / ( 1.0 + std::exp(-x) )
10 #define __D_SIGMOID__(x) x * ( 1 - x )
11
12 // Precisione dei valori
13 typedef double T_Precision;
14
15 // Struttura di un neurone
16 struct Neuron {
17
18     T_Precision value; // Uscita del perceptrone
19     T_Precision dEdy; // Errore del neurone
20 };
21
22 // Struttura di una connessione tra due neuroni (sinapsi)
23 struct Synapse {
24
25     T_Precision weight; // Peso della connessione
26     T_Precision dEdw; // Errore della connessione
27 };
28
29 ... DEFINIZIONE DELLA FUNZIONE PER L'ESECUZIONE DELLA RETE NEURALE ...
30
31 void
32 init_weight( Synapse *synapse, size_t layer_size, size_t prev_layer_size ) {
33
34     // Valore casuale
35     T_Precision random_;
36
37     // Iteratorii
38     size_t j, i = 0;
39
40     // Inizializzo i pesi sinaptici con dei valori casuali
41     for ( ; i < layer_size; i++ ) {
42
43         for ( j = 0; j <= prev_layer_size; j++ ) {
44
45             // Prelevo un valore casuale

```

```

95         random_ = rand();
96
97         // Imposto il valore del peso tra 0 e 1
98         synapse[i * prev_layer_size + j].weight = ( sin(random_) * sin(random_)
99     );
100     }
101 }
102
103 void
104 backpropagation( Neuron *i_neuron, size_t input_size,
105                 Neuron *h_neuron, Synapse *h_synapse, size_t hidden_size,
106                 Neuron *o_neuron, Synapse *o_synapse, size_t output_size,
107                 const T_Precision *dx, const T_Precision *dy, size_t sn,
108                 const T_Precision eta, const T_Precision desired_error ) {
109
110     // Errore della rete
111     T_Precision error;
112
113     // Modifica del peso sinaptico
114     T_Precision delta_w;
115
116     // Contatore delle epoche
117     size_t epochs = 0;
118
119     // Iteratorii
120     size_t t, j, i;
121
122     // Log di lavoro
123     printf( "Inizio l'addestramento (eta = %.2f, errore desiderato = %f).\n", eta,
124     desired_error );
125
126     // Inizializzo i pesi sinaptici con dei valori casuali
127     init_weight( h_synapse, hidden_size, input_size );
128     init_weight( o_synapse, output_size, hidden_size );
129
130     // Continuo l'addestramento finché non raggiungo
131     // l'errore desiderato (max 50000 epoche)
132     do {
133         // Azzero l'errore della rete
134         error = 0.0;
135
136         // Ripeto per tutti gli esempi nell'insieme di addestramento
137         for ( t = 0; t < sn; t++ ) {
138
139             // Prendo gli ingressi dall'esempio
140             for ( i = 0; i < input_size; i++ ) {
141
142                 i_neuron[i].value = dx[t * input_size + i];
143             }
144
145             // Eseguo la rete neurale
146             run_network( i_neuron, input_size,
147                         h_neuron, h_synapse, hidden_size,
148                         o_neuron, o_synapse, output_size );
149
150             // Calcolo l'errore dello strato di uscita
151             for ( i = 0; i < output_size; i++ ) {
152
153                 // Calcolo l'errore dell'uscita del neurone
154                 //  $dE/dy_i = -(D_i - Y_i)$ 
155                 o_neuron[i].dE_dY = -( dy[t * output_size + i] - o_neuron[i].value );
156
157                 // Aggiungo l'errore al totale
158                 error += __POW2__( o_neuron[i].dE_dY );

```

```

159     }
160
161     // Calcolo l'errore dello strato intermedio
162     for ( i = 0; i < hidden_size; i++ ) {
163
164         // Azzero l'errore del neurone
165         h_neuron[i].dEdy = 0;
166
167         // Calcolo l'errore dello strato intermedio
168         // dE/dz_k = SUM( dE/dy_j * dy_j/dP_j * dP_j/dz_k )
169         for ( j = 0; j < output_size; j++ ) {
170
171             // Calcolo l'errore dell'uscita del neurone
172             h_neuron[i].dEdy += o_neuron[j].dEdy *
__D_SIGMOID__( o_neuron[j].value ) * o_synapse[j * ( hidden_size + 1 ) + i].weight;
173         }
174
175         // Calcolo l'errore dell'uscita del bias
176         h_neuron[i].dEdy += o_neuron[j].dEdy *
__D_SIGMOID__( o_neuron[j].value ) * o_synapse[j * ( hidden_size + 1 ) + i].weight;
177     }
178
179     // Aggiusto i pesi dello strato di uscita
180     for ( i = 0; i < output_size; i++ ) {
181
182         // Correggo il peso sinaptico
183         for ( j = 0; j < hidden_size; j++ ) {
184
185             // Calcolo la modifica del peso sinaptico
186             // delta_w = - eta * dE/dy_j * dy_j/dP_j * dP_j/dw_jk
187             delta_w = - eta * o_neuron[i].dEdy *
__D_SIGMOID__( o_neuron[i].value ) * h_neuron[j].value;
188
189             // Applico la modifica al peso sinaptico
190             // w = w + delta_w
191             o_synapse[i * ( hidden_size + 1 ) + j].weight += delta_w;
192         }
193
194         // Calcolo la modifica del peso del bias
195         // delta_w = - eta * dE/dy_j * dy_j/dP_j * dP_j/dw_jk
196         delta_w = - eta * o_neuron[i].dEdy *
__D_SIGMOID__( o_neuron[i].value );
197
198         // Aggiungo la correzione del bias
199         // w = w + delta_w
200         o_synapse[i * ( hidden_size + 1 ) + j].weight += delta_w
201     }
202
203     // Aggiusto i pesi dello strato intermedio
204     for ( i = 0; i < hidden_size; i++ ) {
205
206         // Correggo il peso sinaptico
207         for ( j = 0; j < input_size; j++ ) {
208
209             // Calcolo la modifica del peso sinaptico
210             // delta_w = - eta * dE/dz_k * dz_k/dP_k * dP_k/dw_ki
211             delta_w = - eta * h_neuron[i].dEdy *
__D_SIGMOID__( h_neuron[i].value ) * i_neuron[j].value;
212
213             // Applico la modifica al peso sinaptico
214             // w = w + delta_w
215             h_synapse[i * ( input_size + 1 ) + j].weight += delta_w;
216         }
217
218         // Calcolo la modifica del peso del bias
219         // delta_w = - eta * dE/dz_k * dz_k/dP_k * dP_k/dw_ki

```

```

220         delta_w = - eta * h_neuron[i].dEdy *
__D_SIGMOID__( h_neuron[i].value );
221
222         // Aggiungo la correzione del bias
223         // w = w + delta_w
224         h_synapse[i * ( input_size + 1 ) + j].weight += delta_w;
225     }
226 }
227
228 // Calcolo l'errore quadratico medio della rete (MSE)
229 // E(x) = SUM( e^2 ) / n_samples
230 error /= ( input_size * sn );
231
232 // Ogni 1000 epoche stampo il log di addestramento
233 if ( epochs % 1000 == 0 ) {
234     printf( "Epoca #%zu, MSE=%f\n", epochs, error );
235 }
236
237 // Incremento il numero delle epoche
238 epochs++;
239
240 } while ( error > desired_error && epochs < 50000 );
241
242 // Log di lavoro
243 printf( "Addestramento terminato dopo %zu epoche.\n\n", epochs );
244 }
245
246 int main( void ) {
247     // Inizializzo il generatore di numeri pseudocasuali
248     srand( (size_t) time( NULL ) );
249
250     // Dimensioni della rete
251     const size_t input_size = 2;
252     const size_t hidden_size = 4;
253     const size_t output_size = 1;
254
255     // Creo i neuroni degli strati
256     Neuron i_neuron[ input_size ];
257     Neuron h_neuron[ hidden_size ];
258     Neuron o_neuron[ output_size ];
259
260     // Creo le sinapsi degli strati (+ quella del bias)
261     Synapse h_synapse[ (input_size + 1) * hidden_size ];
262     Synapse o_synapse[ (hidden_size + 1) * output_size ];
263
264     // Numero di esempi dell'insieme di addestramento
265     const size_t sn = 4;
266
267     // Preparo l'insieme di addestramento per l'operatore XOR
268     const T_Precision dx[sn * input_size] = {
269         0, 0,
270         0, 1,
271         1, 0,
272         1, 1
273     };
274
275     const T_Precision dy[sn * output_size] = { 0, 1, 1, 0 };
276
277     // Tasso di apprendimento
278     const T_Precision eta = 0.5;
279
280     // Errore desiderato
281     const T_Precision desired_error = 0.0001;
282
283
284

```



```

285 // Addestrò la rete neurale
286 backpropagation( i_neuron, input_size,
287                 h_neuron, h_synapse, hidden_size,
288                 o_neuron, o_synapse, output_size,
289                 dx, dy, sn, eta, desired_error );
290
291 // Iteratore
292 size_t i, j;
293
294 // Stampo l'intestazione della tabella
295 printf( "[x1]\t[x2]\t[y]\n" );
296
297 // Provo tutte le combinazioni degli ingressi
298 for ( j = 0; j <= 1; j++ ) {
299     for ( i = 0; i <= 1; i++ ) {
300         // Imposto gli ingressi
301         i_neuron[0].value = i;
302         i_neuron[1].value = j;
303
304         // Eseguo la rete neurale
305         run_network( i_neuron, input_size,
306                    h_neuron, h_synapse, hidden_size,
307                    o_neuron, o_synapse, output_size );
308
309         // Stampo gli ingressi e la rispettiva uscita
310         printf( "%.0f\t%.0f\t%.0f\n", i_neuron[0].value, i_neuron[1].value,
311                o_neuron[0].value );
312     }
313 }
314
315 return 0;
316 }

```

La struttura dei percettroni e quella delle sinapsi è stata modificata per aggiungere, rispettivamente, la derivata dell'errore sull'uscita del nodo e la derivata dell'errore sul peso sinaptico.

La funzione *init_weight* inizializza un gruppo di pesi sinaptici che connettono due strati utilizzando dei valori pseudo casuali compresi tra zero e uno. La funzione *backpropagation*, invece, addestra la rete neurale sulla base degli esempi che le vengono passati come argomento. Al suo interno vengono calcolate prima le derivate degli errori sulle uscite dei percettroni e, successivamente, le derivate degli errori sui pesi sinaptici che vengono utilizzate per correggere questi ultimi.

0x03:09 La retropropagazione elastica

I due metodi di apprendimento visti finora utilizzano un tasso di apprendimento *costante*. Questo significa che, per tutta la fase di addestramento, la discesa della funzione di errore avviene ad una velocità direttamente proporzionale alla sua pendenza. I limiti di questa metodologia sono gli stessi messi in evidenza nel paragrafo [0x03:03](#), ossia le difficoltà nel trovare un tasso di apprendimento adeguato che non sia eccessivamente piccolo, il che porterebbe a tempi di addestramento lunghi, né troppo grande, che renderebbe impossibile la convergenza sul minimo locale della funzione di errore.

Per aggirare questo ostacolo, e cercare di ridurre i tempi dell'apprendimento, si è pensato bene di introdurre un *fattore di inerzia*, un parametro che stabilisca quando la velocità di apprendimento debba essere incrementata, ad esempio nelle lunghe discese, e quando, invece, diminuita, ad esempio dopo il superamento del minimo locale più vicino. Per capire meglio ciò che vogliamo ottenere, si pensi al movimento di una biglia che viene lasciata cadere da un lato di una rampa a

forma di “U”, che acquista velocità nelle lunghe pendenze e che ne perde al cambio di direzione impostoli dalla sfonda opposta, fino a fermarsi nel punto più basso della rampa.

Il metodo più semplice per introdurre questo nuovo fattore è quello di modificare la *Regola Delta Generalizzata* in questo modo:

$$\Delta w_{ji}^{t+1} = -\eta \delta_j x_i + \alpha \Delta w_{ji}^t$$

Fig. 56: Regola delta generalizzata con il momentum

dove “ α ” è un parametro arbitrario costante, maggiore o uguale a zero e minore di uno, detto *momentum*, ossia il fattore di inerzia accennato precedentemente.

Questa nuova formula tiene conto della correzione effettuata nell'epoca precedente, in modo tale da aumentare la velocità di apprendimento nel caso in cui le correzioni di due epoche consecutive abbiano segni algebrici concordi, ossia nei periodi in cui la pendenza è stata costante, e per diminuirla nel caso in cui i segni siano discordi, ovvero al cambio della pendenza e quindi dopo il superamento del minimo locale più vicino.

Questo accorgimento rende l'addestramento più preciso e, soprattutto, più rapido di quello originale.

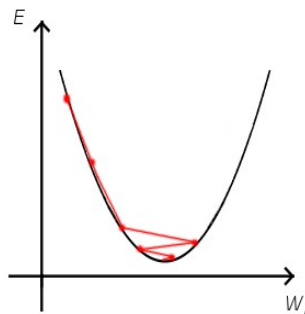


Fig. 57: Raggiungimento del minimo locale con l'adattamento del tasso di apprendimento

Tuttavia si è giunti allo *stato dell'arte* solo con l'introduzione dell'algoritmo *Rprop* (o *Resilient backpropagation*, o *retropropagazione elastica*), sviluppato da *Martin Riedmiller* e *Heinrich Braun* nel 1992. Si tratta di un metodo euristico che elimina del tutto l'influenza dannosa della derivata parziale dell'errore: in questo modo il tasso di apprendimento può crescere o diminuire indipendentemente da quale sia la pendenza della funzione costo.

In questo algoritmo, infatti, ogni peso sinaptico della rete ha il suo specifico tasso di apprendimento, che viene incrementato o diminuito a seconda della pendenza della funzione costo assunta nel corso dell'addestramento.

La prima parte dell'algoritmo è molto semplice ed usa il segno della derivata dell'errore sul peso sinaptico per stabilire quale sia il verso della correzione:

$$w_{ji}^t = w_{ji}^{t-1} + \Delta w_{ji}^t$$

$$\Delta w_{ji}^t = \begin{cases} -\eta_{ji} & \text{se } \frac{\partial E^t}{\partial w_{ji}} > 0 \\ \eta_{ji} & \text{se } \frac{\partial E^t}{\partial w_{ji}} < 0 \\ 0 & \text{se } \frac{\partial E^t}{\partial w_{ji}} = 0 \end{cases}$$

Fig. 58: Aggiornamento dei pesi sinaptici attraverso l'algoritmo Rprop

dove “ $\Delta\eta_{ji}$ ” è appunto il tasso di apprendimento del peso sinaptico “ Δw_{ji} ”.

La seconda parte, invece, è quella che modifica il tasso di apprendimento in modo euristico:

$$\eta_{ji}^t = \eta_{ji}^{t-1} + \Delta\eta_{ji}^t$$

$$\Delta\eta_{ji}^t = \begin{cases} \eta^+ \cdot \Delta\eta_{ji}^{t-1} & \text{se } \frac{\partial E^{t-1}}{\partial w_{ji}} \cdot \frac{\partial E^t}{\partial w_{ji}} > 0 \\ \eta^- \cdot \Delta\eta_{ji}^{t-1} & \text{se } \frac{\partial E^{t-1}}{\partial w_{ji}} \cdot \frac{\partial E^t}{\partial w_{ji}} < 0 \\ \Delta\eta_{ji}^{t-1} & \text{se } \frac{\partial E^{t-1}}{\partial w_{ji}} \cdot \frac{\partial E^t}{\partial w_{ji}} = 0 \end{cases}$$

Fig. 59: Modifica dei tassi di apprendimento attraverso l'algoritmo Rprop

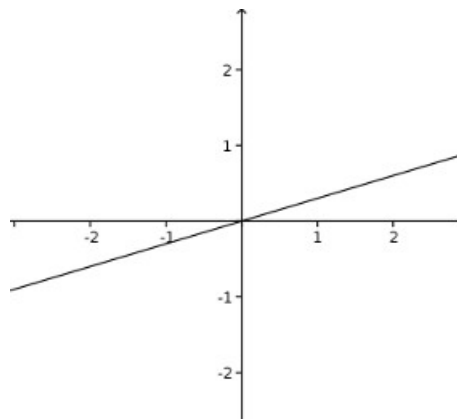
dove “ η^+ ” e “ η^- ” sono due parametri scelti arbitrariamente, tali che risulti $0 < \eta^- < 1 < \eta^+$, e sono detti, rispettivamente, *fattore di incremento* e *fattore di decremento*.

Quando la derivata della funzione costo sul peso sinaptico mantiene per due epoche consecutive lo stesso segno, e quindi la stessa pendenza, il tasso di apprendimento viene moltiplicato per il fattore di incremento che, essendo maggiore di uno, aumenta la velocità della discesa; viceversa, quando la pendenza varia da un'epoca alla successiva, il tasso di apprendimento viene moltiplicato per il fattore di decremento, e quindi diminuito, per avvicinarsi con più lentezza al minimo locale della funzione. I valori ritenuti ideali per “ η^+ ” e “ η^- ” sono, rispettivamente, 1.2 e 0.5.

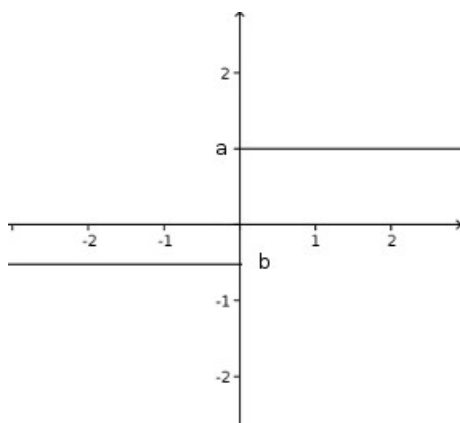
Altri algoritmi che cercano di eliminare l'influenza dannosa della derivata parziale sono il *SuperSAB* (*Super self-adjusting back-propagation*, Super retropropagazione autoregolante), sviluppato da *Tom Tollenaere*, e il *Quickprop* (*Quick propagation*, Propagazione rapida) proposto da *Scott E. Fahlman*; oltre ovviamente a tutte le evoluzioni dell'*Rprop* (*Rprop+*, *Rprop-*, *IRprop+*, *IRprop-*).

0x03:10 Le funzioni di trasferimento

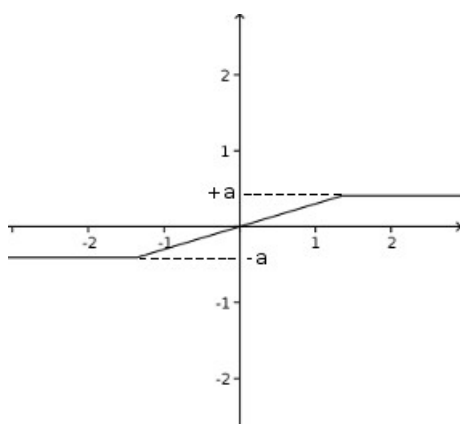
Di seguito trovate le più comuni funzioni di trasferimento usate per le reti di perceptron.



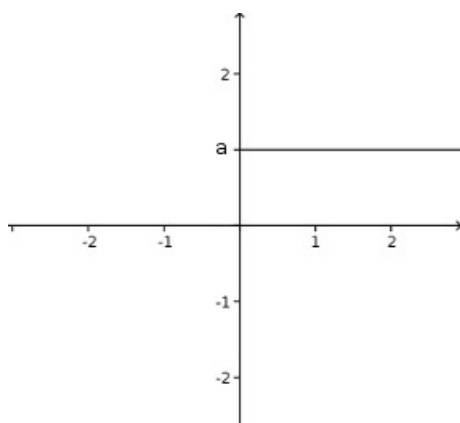
Funzione lineare: $T(x) = ax$, derivabile come $T'(x) = a$.



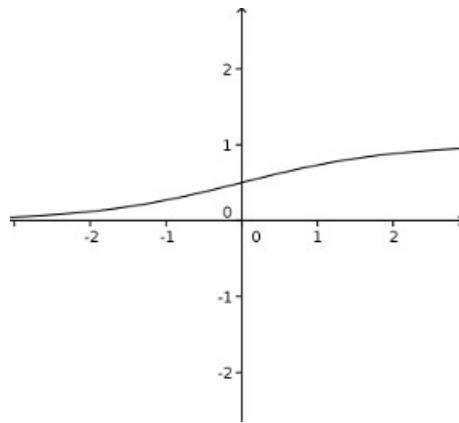
Funzione a gradino $T(x) = \begin{cases} +a, & \text{se } x > 0 \\ -b, & \text{se } x \leq 0 \end{cases}$, non derivabile.



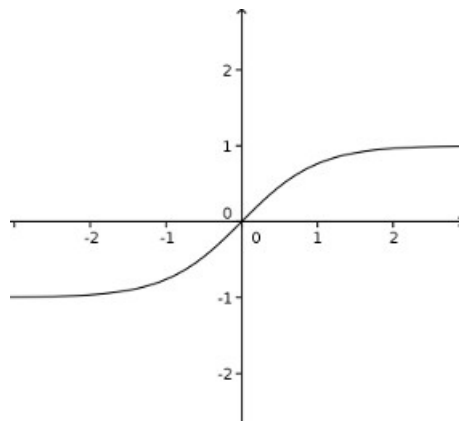
Funzione a rampa $T(x) = \begin{cases} +a, & \text{se } x \geq a \\ x, & \text{se } |x| < a \\ -a, & \text{se } x \leq -a \end{cases}$, derivabile come $T'(x) = 1$,
valori compresi tra $-a$ e $+a$.



Funzione booleana $T(x) = \begin{cases} 0, & \text{se } x \leq 0 \\ a, & \text{se } x > 0 \end{cases}$, non derivabile.



Funzione sigmoidea $T(x) = \frac{1}{1 + e^{-x}}$, derivabile nella forma $T'(x) = T(x)(1 - T(x))$,
valori compresi tra 0 e 1.



Funzione tangente iperbolica $T(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$, derivabile,
valori compresi tra -1 e +1.

0x04 La libreria Serotonina

Serotonina è una libreria libera per la creazione e la gestione di reti neurali artificiali. E' una libreria orientata agli oggetti, scritta in linguaggio C++ ed è stata pensata per gestire con semplicità ed efficienza qualsiasi tipo di rete neurale feedforward, attraverso vari metodi di apprendimento supervisionato implementati sotto forma di moduli.

I tratti distintivi, nonché gli obiettivi che essa si pone di raggiungere, sono:

- la *velocità*, per poter gestire con efficienza e rapidità qualsiasi tipo di rete neurale artificiale;
- la *modularità*, per consentire agli utilizzatori di estendere le funzionalità della libreria attraverso la creazione di moduli personalizzati;
- la *portabilità*, principalmente sui sistemi GNU/Linux e Microsoft Windows;
- l'*eleganza del codice*, secondo le regole della chiarezza e dell'essenzialità, ponendo sempre al primo posto l'efficienza.

La libreria è rilasciata sotto licenza *GNU LGPL v3* e i suoi sorgenti sono reperibili all'indirizzo <https://github.com/Wicker25/Serotonina-Neural-Network-Library> . Serotonina non necessita di dipendenze particolari, ma alcuni suoi demo si appoggiano alle librerie *FLTK* e *OpenCV*.

Alla stesura di questo documento la versione corrente è la *1.1.4*, la stessa a cui faremo riferimento in questa breve introduzione.

0x04:01 Installazione su sistemi GNU/Linux

Sui sistemi GNU/Linux possiamo ottenere una copia dei sorgenti di Serotonina attraverso lo strumento *git*, dando da terminale un semplice:

```
$ git clone http://github.com/Wicker25/Serotonina-Neural-Network-Library
```

Per la compilazione, invece, usiamo lo strumento *CMake*, dopo esserci spostati all'interno della cartella contenente i sorgenti:

```
$ cd Serotonina-Neural-Network-Library
$ cmake ./ -DCMAKE_INSTALL_PREFIX=<PATH>
$ make
```

Una volta che la compilazione è andata a buon fine possiamo installare la libreria sul nostro sistema digitando:

```
# make install
```

Mentre è possibile annullare una precedente installazione con:

```
# make uninstall
```

0x04:02 Installazione su sistemi MS Windows

Nei sistemi MS Windows l'installazione avviene praticamente allo stesso modo, poiché *CMake* genera il *Makefile* o qualsiasi altro “file progetto” necessario alla compilazione:

```
> cmake ./ -G <Generatore del Makefile> -DCMAKE_INSTALL_PREFIX=<PATH>
```

Per maggiori informazioni: http://www.cmake.org/Wiki/CMake_Generator_Specific_Information .

Al momento non esistono pacchetti autoinstallanti.

0x04:03 Creazione e addestramento di una rete neurale

Per aggiungere Serotonina ai nostri programmi è sufficiente includere l'intestazione *serotonina.hpp* ed utilizzare il rispettivo namespace:

```
1 // Includo l'intestazione di Serotonina
2 #include <serotonina.hpp>
3
4 // Uso il namespace di Serotonina
5 using namespace Serotonina;
6
```

Possiamo creare una nuova rete neurale artificiale istanziando la classe *Serotonina::Network*:

```
7 int main( void ) {
8
9     // Creo la rete neurale
10    Network network( 3, 2, 5, 1 );
11
```

Nel riquadro sovrastante, ad esempio, viene creata una rete neurale multistrato 2x5x1: il primo argomento del metodo costruttore, infatti, indica il numero totale degli strati della rete, mentre i successivi argomenti indicano il numero dei neuroni che li compongono, in modo ordinato dal primo all'ultimo strato.

Per addestrare la nuova rete neurale è necessario creare un *Trainer*, un “addestratore”, che permetta di calibrarne la risposta sulla base degli esempi forniti:

```
12     // Creo l'addestratore della rete neurale
13     Trainer trainer( network );
14
```

Una volta associato un *Trainer* alla rete neurale è possibile iniziare l'addestramento scegliendo l'algoritmo e i rispettivi parametri da utilizzare:

```
15     // Addestro la rete neurale con il metodo Batch
16     trainer.SetParameters( 0.5, 0.8 );
17     trainer.TrainOnFile< Algorithms::Batch >( "examples/train/and.train", 0.000001,
18     100000, 5000 );
```

Il metodo *SetParameters* serve ad impostare i parametri dell'addestramento, in questo caso specifico il *tasso di apprendimento* e il *momentum* (fattore inerzia) usati per la *Regola Delta Generalizzata*.

Per addestrare la rete neurale, invece, abbiamo a disposizione due metodi:

- *Train*, che legge l'insieme di addestramento da un vettore;
- *TrainOnFile*, che legge l'insieme di addestramento da un file esterno;

Il primo parametro del metodo *TrainOnFile*, lo stesso utilizzato nell'esempio precedente, è il percorso al file contenente l'insieme di addestramento, il secondo indica l'errore desiderato, il terzo le epoche massime per l'addestramento e l'ultimo l'intervallo di epoche tra un report e l'altro.

Il *parametro template*, invece, è la classe-modulo usata per l'addestramento.

Attualmente Serotonina mette a disposizione i seguenti algoritmi di addestramento: *Batch (Regola delta + Momentum)*, *Rprop*, *Rprop+*, *Rprop-*, *IRprop+* e *IRprop-*.

Il contenuto del file “*and.train*” usato per l'addestramento è il seguente:

```
1 2 1
2 0 0:0
3 0 1:0
4 1 0:0
5 1 1:1
```

La prima riga è l'intestazione del training set ed indica alla libreria che gli esempi forniti sono per una rete a due ingressi ed una sola uscita. Il corpo del file, invece, è costituito dagli esempi veri e propri, uno per riga, dove il carattere di separazione “:” indica la fine degli esempi di ingresso e l'inizio di quelli di uscita.

Per mandare in esecuzione la rete neurale usiamo il metodo *Run*, che prende come unico parametro un vettore di tipo *Serotonina::T_Precision* contenente gli ingressi della rete, e che restituisce un riferimento al vettore delle uscite:

```
19 // Imposto gli ingressi della rete neurale
20 Serotonina::T_Precision in[2] = { 1, 1 };
21
22 // Calcolo l'uscita della rete neurale
23 const std::vector< T_Precision > &out = network.Run( in );
24
25 // Stampo i risultati
26 std::cout.setf( std::ios::fixed, std::ios::floatfield );
27 std::cout << "\nRun ( " << in[0] << " AND " << in[1] << " ) = " << out[0] <<
std::endl;
28
29 return 0;
30 }
31
```

Infine, con la distruzione dell'istanza, vengono deallocate tutte le strutture della rete.

0x04:04 Un'applicazione reale

Insieme ai sorgenti della libreria Serotonina sono stati inclusi alcuni software d'esempio che fanno uso delle reti neurali artificiali: uno strumento grafico per l'addestramento delle reti neurali (*st_gym*), un riconoscitore di caratteri manoscritti (*st_ocr*) ed un software per l'autenticazione facciale (*st_face_recognition*, attualmente in costruzione).

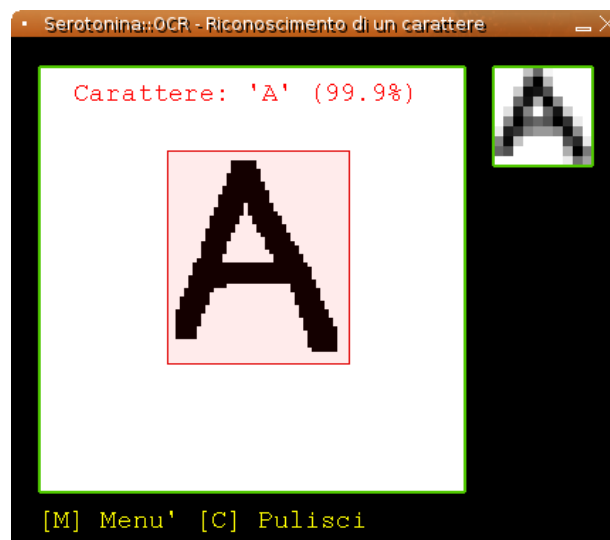


Fig. 60: Riconoscitore di caratteri manoscritti

Prendiamo in considerazione il *riconoscitore di caratteri manoscritti*, che rappresenta una possibile applicazione pratica delle reti neurali artificiali. Si tratta di un software in grado di riconoscere una lettera manoscritta su una tavola da disegno, una semplificazione dei programmi utilizzati per la digitalizzazione dei documenti cartacei (*OCR*).

All'avvio del programma (*Fig 60*) appare una semplice finestra *OpenGL* costruita attorno al *framework FLTK* (una libreria grafica degna di nota per la sua semplicità e portabilità).

In questa prima finestra possiamo distinguere la tavola da disegno, nella quale è possibile disegnare il carattere manoscritto, ed un riquadro più piccolo, contenente il carattere estrapolato dalla tavola; l'area di colore rosso-trasparente è la delimitazione automatica del carattere, ossia la più piccola area rettangolare che lo può contenere.

Quando disegniamo sulla tavola da disegno, l'area contenente il carattere viene ridimensionata e portata alle dimensioni di 10x10 px. Questa prima elaborazione, oltre ad estrapolare il carattere dal suo contesto, serve a “normalizzare” le informazioni prima della classificazione vera e propria, che avviene per mezzo di diverse reti neurali artificiali che operano in cascata.

L'immagine 10x10 px, infatti, può essere descritta come un vettore monodimensionale di 100 elementi, dove le righe dell'immagine sono messe in fila una dietro l'altra. Ognuno di questi elementi contiene un valore reale compreso tra 0 ed 1, corrispondente al livello di “oscurità” del pixel a cui fa riferimento; tuttavia, trattandosi di un'immagine monocromatica, i pixel neri corrisponderanno al valore 1 e quelli bianchi al valore 0.

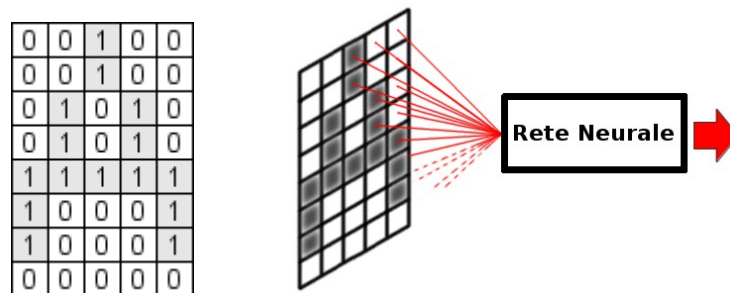


Fig. 61: Esempio di elaborazione del riconoscitore di caratteri manoscritti

Le informazioni sul carattere vengono quindi processate da diverse reti neurali delle dimensioni di 100x15x15x1 (che potete trovare al percorso “*data/character/*”), con un numero di ingressi pari al numero degli elementi del vettore che contiene le informazioni. Tra tutte le reti neurali, quella con l'indice di riscontro più alto è identificata come la rete di appartenenza del carattere.

Nel caso in cui, invece, l'identificazione non vada a buon fine, è possibile correggere la risposta delle reti neurali indicando manualmente la classe di appartenenza del carattere disegnato; potete farlo dal *menù di addestramento* (*Fig 62*) che può essere richiamato premendo il tasto “*m*” della tastiera.

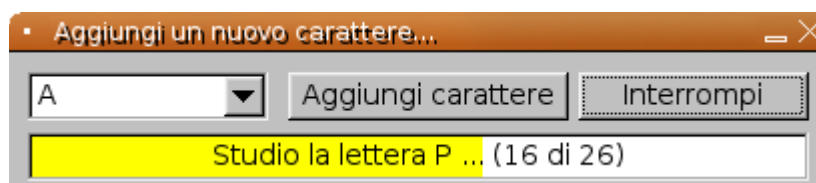


Fig. 62: Menù di addestramento del riconoscitore di caratteri manoscritti

Una volta selezionata la classe di appartenenza del carattere è sufficiente premere su “*Aggiungi carattere*” e, aggiunto un numero sufficiente di esempi, è possibile ripetere l'addestramento delle reti neurali premendo sul pulsante “*Addestra le reti*”; durante questa fase viene generato il training set di ogni carattere, associando alle immagini del carattere preso in considerazione un'uscita alta e

a tutte le altre un uscita bassa. Sono sufficienti pochi esempi per consentire il riconoscimento del carattere.

0x05 Conclusione

Applicazioni come quella del paragrafo precedente mettono in evidenza le straordinarie potenzialità delle reti neurali artificiali, in particolare la loro forte tolleranza agli errori e la straordinaria capacità di astrarre nuove soluzioni, là dove non siamo in grado di spiegare in termini logici la soluzione di un problema. Ma essa è solo una delle tante applicazioni.

Nella medicina moderna le reti neurali artificiali sono utilizzate in numerosissimi ambiti, dai sistemi di supporto per le persone portatrici di handicap, come i sistemi di *eye tracking* (tracciamento dell'occhio) per la video scrittura e le *interfacce vocali*, che permettono alle persone con limitata mobilità di interagire con dei dispositivi informatici; fino alla diagnosi precoce delle malattie, ad esempio i sistemi per il riconoscimento dei tessuti tumorali o delle malattie neurodegenerative come l'Alzheimer. Ma anche nella finanza, nella meteorologia, nella sismologia, nella biometria, etc. Le reti neurali artificiali trovano significative applicazioni in tutti quei contesti dove gli approcci tradizionali falliscono.

Tra gli svantaggi, invece, dobbiamo citare l'impossibilità di comprendere a pieno le soluzioni apprese dalle reti, poiché esse rimangono dei sistemi a "scatola chiusa". Inoltre reti di elevate dimensioni possono essere computazionalmente pesanti da elaborare, ma questo è un limite che sarà presto superato con il progresso dei calcolatori e con la costruzione di hardware dedicato.

Spero che questa tesi vi abbia stimolato o quantomeno incuriosito su un argomento che con gli anni si appresta a divenire un tema di grande attualità.

0x06 Bibliografia

- Alberto Marotta, Misura delle frazioni di produzione degli adroni charmati carichi in interazioni indotte da neutrini nelle emulsioni dell'esperimento CHORUS, Università degli Studi di Napoli "Federico II", 2002
- Crescenzo Gallo, Reti Neurali Artificiali: Teoria ed Applicazioni, Università degli Studi di Foggia, 2007, www.dsems.unifg.it/q282007.pdf
- Marco De Michele, Una breve introduzione alle reti neurali, <http://www.taolab.it/algorithms/nn/nn.htm>
- Marco Scarnò, Metodi per lo studio di variabili finanziarie: un confronto teorico ed empirico tra modelli statistici e modelli di reti neurali, Università degli Studi "La Sapienza", <http://www.caspur.it/~mscarno/accademica/TesiMarco.pdf>
- Salvatore Poma, Reti Neurali, <http://utenti.multimania.it/pomas/neural/neural.htm>
- Silvio Cammarata, Sistemi a logica fuzzy. Come rendere intelligenti le macchine, ETAS, Torino, 1997
- Stefano Cagnoni, Apprendimento Automatico, Reti Neurali Apprendimento supervisionato, http://www.ce.unipr.it/people/cagnoni/didattica/appraut/lucidi/appraut_6.ppt
- Susi Dulli, Sara Furini, Edmondo Peron, Data mining metodi e strategie, Springer, Milano, 2009
- Teoria e Tecniche del Riconoscimento, Reti Neurali Artificiali, Università di Trento, www.inf.unitn.it/Intranet/Didattica/Materiali_corsi/Laurea/3anno2semestre/3anno2semestre_file/TL

[C/download/Capitolo6.pdf](#)

- Vittorio Villasmunta,, Reti neurali e previsioni del tempo, 2001,
http://digilander.libero.it/vvillas/reti_neurali/reti_neurali_e_previsioni_del_te.htm
- <http://it.wikipedia.org/wiki/Neurone>
- <http://nsa.liceofoscarini.it/derivate/composta.html>
- <http://www.accademiaxl.it/Biblioteca/Virtuale/Ipertesti/neuroscienzeXL/golgididattica.htm>
- http://www.irccsdebellis.it/html/reti_neurali/Home_retineurali.htm
- <http://www.dontveter.com/bpr/faster.html>