

**RETI NEURALI
SU PERSONAL COMPUTER
+ FUZZY LOGIC**

di

Luca Marchese

Nota di Copyright

Potete distribuire questo libro in qualsiasi forma (web, cartacea, file, ecc), alle seguenti condizioni:

- Il contenuto non venga alterato
- Sia riportato il nome dell'autore
- In caso di diffusione a scopo di lucro, l'autore sia anticipatamente contattato.

Scritto da Luca Marchese --- marchese@mbox.ulisse.it

E' possibile trovare questo libro in formato HTML presso i seguenti indirizzi:

- <http://www.ulisse.it/~marchese/book/neurbook.html>
- <http://space.tin.it/clubnet/blimar/neurbook/neurbook.html>
- <http://digilander.iol.it/lucamarchese/neurbook/neurbook.html>
- http://web.tiscalinet.it/luca_marchese/neurbook/neurbook.html

La versione PDF è stata adattata da Stefano Arcidiacono, ed è disponibile all'indirizzo:

- <http://gpi.eden.it>

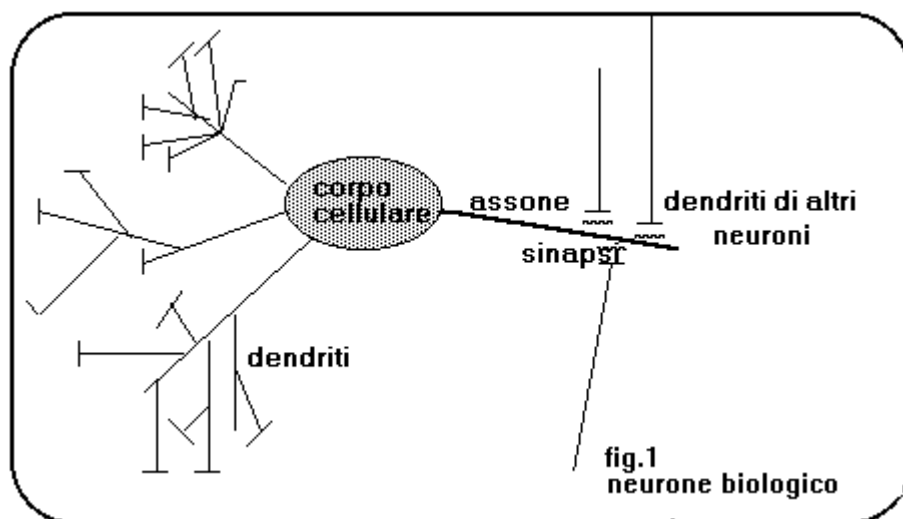
IL CERVELLO UMANO

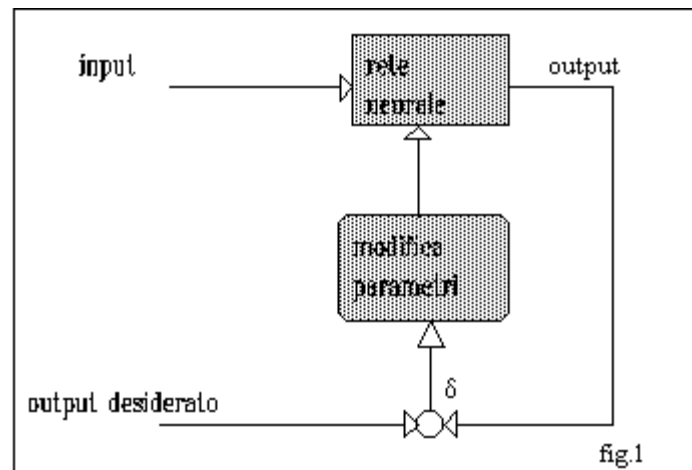
Il cervello umano è sicuramente la struttura più complessa dell'universo e può essere considerato come una enorme rete neurale. Circa 100 miliardi di neuroni costituiscono i nodi di tale rete. Ciascun neurone è collegato a decine di migliaia di altri neuroni ed esistono pertanto milioni di miliardi di connessioni. Un neurone biologico è composto da un corpo cellulare o "soma" dal quale partono molti collegamenti(dendriti)che ricevono segnali da altri neuroni, e un collegamento di uscita(assone) con il quale il neurone trasmette informazioni ad altri neuroni(attraverso i loro dendriti)([fig.1](#)). Ogni neurone ha una soglia di attivazione caratteristica: se i segnali provenienti da altri neuroni la superano, il neurone si attiva e trasmette un segnale elettrico sull'assone che arriva ad altri neuroni. Fra assone e dendrite esiste una sottile intercapedine detta "sinapsi" che permette la trasmissione del segnale attraverso un processo elettrochimico. Lo spessore della sinapsi può variare nel tempo rafforzando o indebolendo il collegamento tra due neuroni. Il contenuto informativo momentaneo del cervello è rappresentato dall'insieme dei valori di attivazione di tutti i neuroni, mentre la memoria è rappresentata dai valori di collegamento(più o meno forte) di tutte le sinapsi. Due sono le caratteristiche fondamentali del cervello: la plasmabilità e la scomposizione dell'informazione in informazioni elementari contenute in ogni singolo neurone. La plasmabilità deriva dal fatto che le sinapsi possono modificarsi nel tempo interagendo con segnali dal mondo esterno. Non è assolutamente ancora chiaro il meccanismo di apprendimento del cervello, ma è chiaro che il rafforzamento e l'indebolimento dei collegamenti sinaptici costituisce la memorizzazione di una informazione.

RETI NEURALI ARTIFICIALI

Le reti neurali sono lo stato dell'arte nel trattamento dell'informazione. Sono basate su principi completamente differenti da quelli normalmente utilizzati nell'AI classica per il trattamento dell'informazione e il supporto alla decisione. In effetti, in una rete neurale le informazioni sono scomposte in informazioni "elementari" contenute all'interno di ogni singolo neurone. Una rete neurale può essere vista come un sistema in grado di dare una risposta ad una domanda o fornire un output in risposta ad un input. La combinazione in/out ovvero la funzione di trasferimento della rete non viene programmata, ma viene ottenuta attraverso un processo di "addestramento" con dati empirici. In pratica la rete apprende la funzione che lega l'output con l'input attraverso la presentazione di esempi corretti di coppie input/output. Effettivamente, per ogni input presentato alla rete, nel processo di apprendimento, la rete fornisce un output che si discosta di una certa quantità DELTA dall'output desiderato: l'algoritmo di addestramento modifica alcuni parametri della rete nella direzione desiderata ([fig.1](#)). Ogni volta che viene presentato un esempio, quindi, l'algoritmo avvicina un poco i parametri della rete ai valori ottimali per la soluzione dell'esempio: in questo modo l'algoritmo cerca di "accontentare" tutti gli esempi un po' per volta. I parametri di cui si parla sono essenzialmente i pesi o fattori di collegamento tra i neuroni che compongono la rete. Una rete neurale è infatti composta da un certo numero di neuroni collegati tra loro da collegamenti "pesati", proprio come lo sono i neuroni del cervello umano. Ciò che ha portato alla realizzazione delle reti neurali è stato il tentativo di realizzare delle simulazioni delle strutture nervose del tessuto cerebrale. Tale obiettivo è, però, sfociato nella identificazione di modelli matematici che non hanno molte affinità con i modelli biologici. Un neurone del tessuto cerebrale può essere visto come una cella (corpo cellulare) che ha molti ingressi (dendriti) e una sola uscita (assone): una rete neurale

biologica è composta da molti neuroni dove gli assoni di ogni neurone vanno a collegarsi ai dendriti di altri neuroni tramite un collegamento (la cui forza varia chimicamente in fase di apprendimento e costituisce una "microinformazione") che viene chiamato sinapsi. La [fig.2](#) è la rappresentazione formale di un neurone biologico: come si può notare il neurone ha una sua interna funzione di trasferimento. Non sono ancora chiari i meccanismi di apprendimento del cervello degli esseri viventi e le reti neurali artificiali sono attualmente solo un sistema di trattamento dell'informazione in modo distribuito con algoritmi di apprendimento dedicati. Bisogna sottolineare però che le reti neurali hanno caratteristiche sorprendentemente simili a quelle del cervello umano, come capacità di apprendere, scarsa precisione associata ad alta elasticità di interpretazione dell'input e quindi capacità di estrapolazione. Quella che abbiamo chiamato elasticità di interpretazione dell'input viene comunemente chiamata "resistenza al rumore" o "capacità di comprendere dati rumorosi": un sistema programmato ha bisogno di un input ben preciso per dare una risposta corretta, mentre una rete neurale è in grado di dare una risposta abbastanza corretta ad un input parziale o impreciso rispetto a quelli utilizzati negli esempi di addestramento.





TIPI DI RETI NEURALI

Esistono molti tipi di reti neurali che sono differenziati sulla base di alcune caratteristiche fondamentali:

- tipo di utilizzo
- tipo di apprendimento (supervisionato/non supervisionato)
- algoritmo di apprendimento
- architettura dei collegamenti

La divisione fondamentale è quella relativa al tipo di apprendimento che può essere supervisionato o non supervisionato: nel primo caso si addestra la rete con degli esempi che contengono un input e l'output associato desiderato, mentre nel secondo caso la rete deve essere in grado di estrarre delle informazioni di similitudine tra i dati forniti in input (senza associazioni con output desiderati) al fine di classificarli in categorie. Dal punto di vista del tipo di utilizzo possiamo distinguere tre categorie basilari:

- memorie associative
- simulatori di funzioni matematiche complesse (e non conosciute)
- classificatori

MEMORIE ASSOCIATIVE: possono apprendere associazioni tra patterns (insieme complesso di dati come l'insieme dei pixels di una immagine) in modo che la presentazione di un pattern A dia come output il pattern B anche se il pattern A è impreciso o parziale (resistenza al rumore). Esiste anche la possibilità di utilizzare la memoria associativa per fornire in uscita il pattern completo in risposta ad un pattern parziale in input.

SIMULATORI DI FUNZIONI MATEMATICHE: sono in grado di comprendere la funzione che lega output con input in base a degli esempi forniti in fase di apprendimento. Dopo la fase di apprendimento, la rete è in grado di fornire un output in risposta ad un input anche diverso da quelli usati negli esempi di addestramento. Ne consegue una capacità della rete di interpolazione ed estrapolazione sui dati del training set. Tale capacità è facilmente verificabile addestrando una rete con una sequenza di dati input/output proveniente da una funzione nota e risulta, invece, utile proprio per il trattamento e la previsione di fenomeni di cui non sia chiaro matematicamente il legame tra input e output. In ogni caso la rete si comporta come una "black box", poiché non svela in termini leggibili la funzione di trasferimento che è contenuta al suo interno. Di questo tipo fa parte la rete a retropropagazione dell'errore o error back propagation che è quella attualmente più utilizzata per efficacia e flessibilità.

CLASSIFICATORI: con essi è possibile classificare dei dati in specifiche categorie in base a caratteristiche di similitudine. In questo ultimo tipo di rete esiste il concetto di apprendimento non supervisionato o "autoorganizzante", nel quale i dati di input vengono distribuiti su categorie non predefinite.

L'algoritmo di apprendimento di una rete neurale dipende essenzialmente dal tipo di utilizzo della stessa, così come l'architettura dei collegamenti. Le reti multistrato prevedono ad

esempio l'algoritmo a retropropagazione dell'errore o sono addestrate tramite algoritmi genetici. I classificatori normalmente derivano dall'architettura delle mappe autorganizzanti di Kohonen. Esistono diverse regole di base per l'apprendimento ma sono sempre in fase di studio nuovi paradigmi: quello delle reti neurali è un campo in cui c'è ancora molto da inventare e da capire. Questo volume non vuole certamente coprire l'intero panorama delle problematiche, delle applicazioni e dei paradigmi delle reti neurali ma vuole avvicinare il lettore ad una promettente tecnologia analizzando i paradigmi di rete più utilizzati con il supporto di simulazioni software contenute in versione eseguibile su dischetto e listati o suggerimenti per i programmatori che vogliono cimentarsi nella realizzazione di simulazioni proprie.

CARATTERISTICHE GENERICHE DI UNA RETE NEURALE

1) DEVE ESSERE COMPOSTA DA UN CERTO NUMERO DI NEURONI

2) OGNI NEURONE DEVE AVERE INGRESSI E USCITE E UNA PRECISA FUNZIONE DI TRASFERIMENTO

3) GLI INGRESSI E LE USCITE DEI NEURONI DEVONO ESSERE COLLEGATI TRAMITE "COLLEGAMENTI SINAPTICI" MODIFICABILI IN FASE DI ADDESTRAMENTO

4) DEVE ESISTERE UNA PRECISA LEGGE DI APPRENDIMENTO PER LA MODIFICA DEI PESI

Esistono anche reti neurali che non sono basate su specifici collegamenti tra i neuroni ma si evolvono modificando un parametro della funzione di trasferimento di ogni neurone sulla

base delle attivazioni dei neuroni di un opportuno vicinato (come il "vicinato di Von Neumann": neuroni sopra, sotto, a destra e a sinistra in una griglia) Esistono inoltre reti che possono essere studiate appositamente per risolvere problemi di ottimizzazione combinatoria con opportuni vincoli e una funzione di costo(energia) da minimizzare:non tratteremo in questo libro tale argomento che meriterebbe una trattazione a parte insieme ad algoritmi genetici e altre tecniche usate nell AI per la soluzione di problemi di ottimizzazione.

GUIDA AI CAPITOLI DEL LIBRO

Il capitolo 1 descrive il funzionamento di una memoria associativa a collegamenti bidirezionali. Dopo una analisi teorica della rete vengono presentate in metalinguaggio le procedure necessarie per la simulazione software. Viene inoltre presentato il programma didattico ASSOCIA che è contenuto nel dischetto.

il capitolo 2 descrive il principio di funzionamento delle reti neurali multistrato a collegamenti unidirezionali e la teoria della retropropagazione dell' errore. Le reti neurali basate su questo paradigma sono quelle attualmente più utilizzate in tutti i campi per flessibilità ed efficacia e sono in grado di comprendere la funzione matematica che lega coppie di dati input/output. Sul dischetto si può trovare il programma di addestramento/esecuzione di reti neurali error back propagation con i files di addestramento degli esempi proposti. Sul dischetto è contenuto anche il codice sorgente in linguaggio C del programma di simulazione "backprop".

il capitolo 3 approfondisce alcune delle possibili applicazioni delle reti neurali,prestando maggiore attenzione alle reti a retropropagazione dell' errore e al "forecast", cioè alle capacità di previsione delle reti neurali proficuamente utilizzate in campo

finanziario e commerciale. Sul dischetto si trovano i programmi e i files relativi agli esempi di forecast che possono essere sperimentati con il programma "backprop" del capitolo 2.

Il capitolo 4 introduce l'argomento delle reti neurali autoorganizzanti, utilizzate come classificatori, analizzando le reti di Kohonen e passa poi all'analisi dei paradigmi supervisionati LVQ e LVQ2. Vengono presentati listati in metalinguaggio facilmente comprensibili per la realizzazione di tali reti. Sul dischetto è presente un programma di simulazione di una rete autoorganizzante con sorgente in linguaggio "C" e relativi files per addestramento su "problemi giocattolo".

Il capitolo 5 tratta un argomento che esula dalle reti neurali ma è una tecnologia parallela utilizzata nell'AI che spesso si integra con esse: stiamo parlando di FUZZY LOGIC o "teoria del ragionamento sfumato" che sta ottenendo sempre maggiori attenzioni nel mondo dell'industria elettronica. Anche in questo capitolo vengono analizzati listati in semplice metalinguaggio finalizzati alla comprensione dei metodi di realizzazione software di tali sistemi.

Nel capitolo 6 viene presentato il programma NEURFUZZ 1.0 contenuto nel dischetto. Si tratta di un programma in grado di addestrare una rete neurale error back propagation e generare il codice C relativo che può essere inserito all'interno di altri programmi. Inoltre, NEURFUZZ 1.0 può generare motori inferenziali basati su regole fuzzy in codice C, che possono essere interfacciati con reti neurali.

Il capitolo 7 è una guida al dischetto contenuto nel libro, e il lettore vi può accedere in ogni momento per verificare i listati e i programmi concernenti i vari capitoli.

Il capitolo 8 è una breve guida ai più diffusi software presenti sul mercato.

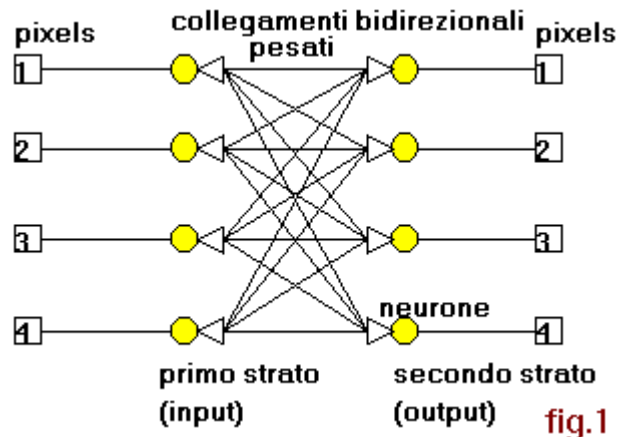
ESEMPIO DI MEMORIA ASSOCIATIVA

In questo capitolo vedremo brevemente il funzionamento di una memoria associativa realizzata in linguaggio c e contenuta nel dischetto in forma eseguibile. La rete è in grado di apprendere una associazione di immagini a coppie realizzate su files ascii con i caratteri "*" e "spazio" che rappresentano rispettivamente pixel acceso e pixel spento. Questa rete è una rete ciclica poiché presenta delle connessioni bidirezionali, cioè gli output ritornano in "retroazione" sugli input ([fig.1](#)).

Un sistema retroazionato può presentare differenti comportamenti:

- 1) si produce una variazione continua dell' output tale che il sistema "esplode": si dice che è un sistema non convergente.
- 2) l' output del sistema continua ad oscillare all' infinito e quindi non converge anche in questo caso.
- 3) l' output converge verso un determinato valore o verso una determinata configurazione dimostrando che il sistema è a tutti gli effetti una "macchina deterministica".

Quindi, da una rete neurale ciclica ci aspettiamo un comportamento come quello descritto al punto 3: la rete, in risposta ad un input, avrà un certo numero di oscillazioni prima di raggiungere uno stato di stabilità che coinciderà con una risposta in output. Il programma della memoria associativa contenuto nel dischetto si chiama associa.exe, ed è un programma esclusivamente didattico/dimostrativo: contiene una memoria associativa tipo BAM (Bidirectional Associative Memory).



Regola di apprendimento

La formula di addestramento utilizzata in questa memoria associativa è la regola di Hebb:

$$DW(j,k)=P(j)*P(k)*t$$

dove

$\Delta W(j,k)$ = variazione del peso di connessione tra il neurone j e il neurone k

$P(j)$ =output del neurone j

$P(k)$ =output del neurone k

t =fattore o tasso di apprendimento

Questa regola si può spiegare in questi termini: aumenta il peso che connette due neuroni in modo proporzionale al prodotto degli output da loro forniti per un determinato input. Non è una regola universalmente valida ed è applicabile solo per particolari tipi di reti neurali, come quella di cui parliamo in questo capitolo. Esistono diversi tipi di regole di apprendimento destinate a scopi specifici, come la regola delta (dalla quale deriva la retropropagazione dell'errore di cui parleremo approfonditamente), o la regola di Kohonen: voglio sottolineare che la regola di apprendimento è il punto critico di ogni rete neurale al di là della architettura sulla quale viene applicata.

ASSOCIA.EXE

Il programma associa.exe è in grado di memorizzare la associazione di coppie di immagini realizzate con files tipo ascii. Per comodità si possono usare immagini di lettere e numeri e associare in fase di addestramento lettere minuscole con corrispondenti lettere maiuscole o lettere con numeri. Il programma presenta il seguente menù:

- a) apprendimento
- b) esecuzione
- c) display immagine input
- d) display immagine output
- e) set dimensionale immagine
- f) load pesi sinaptici da file
- g) store pesi sinaptici su file
- h) editor(per creare immagini)
- i) edit su matrice dei pesi
- l) mappare semigraficamente matrice pesi su file
- m) calcolo automatico della distanza di Hamming
- n) terminazione programma

Utilizzando la funzione edit disegnate su files le immagini che vi interessano utilizzando i caratteri spazio(pixel spento) e "*" (pixel acceso)(vi sono sul dischetto immagini di lettere già preparate che potete seguire come esempio). Effettuate la fase di apprendimento fornendo i nomi dei due files da associare e provate il richiamo con la funzione "esecuzione". Provate poi il richiamo modificando leggermente qualche pixels dell' immagine di input: potrete vedere come output l'immagine corretta associata. In questo punto sta il vero significato di una memoria associativa realizzata con una rete neurale: una associazione tra due immagini potrebbe essere realizzata con un qualsiasi semplice algoritmo, ma una leggera imprecisione nei dati di input non porterebbe ad una

risposta corretta. La rete neurale è in grado di essere insensibile al rumore presente nell' input e risalire all' immagine di input corretta, partendo da quella rumorosa, per poi fornire come output l'immagine associata. Abbiamo realizzato un robusto sistema di accesso alla memoria per contenuto e non per indirizzo. Provate a visualizzare l'immagine di input con "c" dopo avere eseguito la rete con un input rumoroso e vedrete che l'immagine è stata corretta sui neuroni del primo strato della rete dai cicli dovuti ai segnali di riporto dall'output. Potete addestrare la rete neurale con altre coppie di immagini e verificare in esecuzione, ma noterete che la capacità di memoria è molto limitata per cui dopo quattro o cinque coppie cominceranno malfunzionamenti. La funzione di trasferimento di ogni neurone di questa rete è del tipo a gradino ([fig.2](#)). Una rappresentazione schematica della rete è visibile in [fig.1](#) dove si possono notare due strati di neuroni in cui i neuroni di ogni strato sono tra loro scollegati, mentre sono collegati i neuroni di strati differenti. La funzione di trasferimento va intesa come bidirezionale:

feedforward (ciclo input>output) backforward(ciclo di retroazione)

$$U(k) = -1 \text{ se } SI(k) < 0 \quad U(k) = -1 \text{ se } SI(k) < 0$$

$$U(k) = +1 \text{ se } SI(k) \geq 0 \quad U(k) = +1 \text{ se } SI(k) \geq 0$$

dove $U(k)$ = uscita o attivazione del neurone k e $SI(k)$ = sommatoria degli inputs del neurone k

La rete in esecuzione oscilla finché non è stato verificato uno stato di stabilità che corrisponde all' assenza di variazioni di attivazione di ogni neurone della rete. In una realizzazione software si devono utilizzare alcune semplici procedure:

1) caricamento dei dati input nei neuroni del primo strato (vettore int) da file (pixels immagine)

2) visualizzazione dello stato dei neuroni del secondo strato(vettore int) mappati sull' immagine

3) procedura apprendimento contenente regola di Hebb

4) procedura esecuzione che cicla finché non viene verificata la stabilità

Tralasciamo le procedure 1 e 2 che dipendono dall' utilizzo specifico della rete che può essere differente dall' associazione di immagini e, comunque, è abbastanza banale per un programmatore realizzare una procedura che porti i dati da un file grafico o di testo su un vettore binario (perché la rete accetta in input solo valori 0 o 1),o che visualizzi i dati binari contenuti in un vettore in forma grafica. Vediamo invece più in dettaglio la realizzazione delle funzioni di apprendimento ed esecuzione:

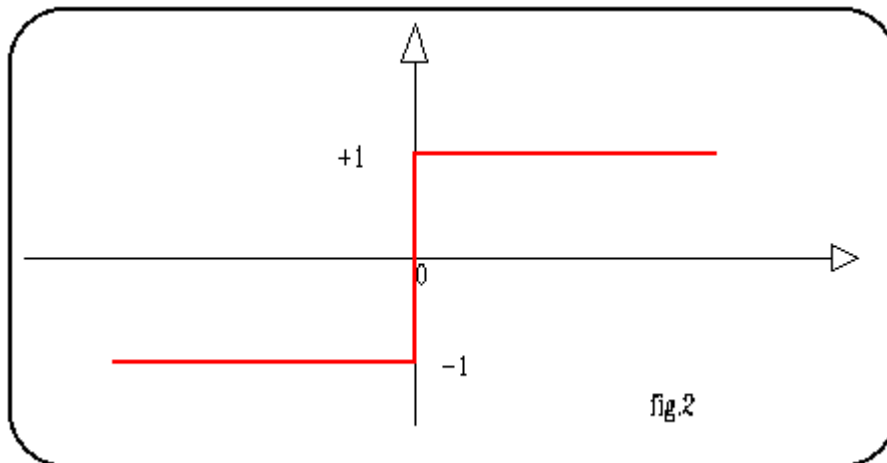
```
note: peso[k][j] =peso del collegamento
      tra il k_esimo neurone del primo strato
      e il j_esimo neurone del secondo strato
A[j]= somma dei segnali di ingresso del j_esimo neurone
U[j]= uscita del j_esimo neurone
```

```
apprendimento()
{
  carica i dati input nel vettore neuroni primo strato di dimensione k
  carica i dati d'associare nel vettore neuroni secondo strato di dimensione j
  per ogni neurone output(j)
  {
    per ogni input(k)
    {
      peso [k][j]=peso[k][j] + input[k]*output[j]   (regola di Hebb)
    }
  }
  return
}
```

```

esecuzione()
{
  carica i dati di input nel vettore input di dimensione k
  finchè non è verificato stabilità=vero
  {
    stabilità=vero
    dall input verso l output:
    per ogni neurone output(j)
      {
        per ogni neurone input(k)
          {
            A2(j)=A2(j)+U1(k) * peso[j][k]
          }
        if (A2(j)>=0) then U2(j)=1
          (funzione di trasferimento del neurone)
        else U2(j)=-1
      }
    dall output verso l input (retroazione):
    per ogni neurone input(k)
      {
        per ogni neurone output(j)
          {
            A1(k)=A1(k)+U2(j) * peso[j][k]
          }
        if ((A1(k)<0 and U1(k)=1) or (A1(k)>=0 and U1(k)=-1))
then stabilità=falso (cambiamento)
        if (A1(k)>=0) then U1(k)=1          (funzione di
trasferimento del neurone)
        else U1(k)=-1
      }
    } (chiude il ciclo verifica stabilità)
  }
  return
}

```



ALTRE FUNZIONI DI ASSOCIA.EXE

Per misurare la resistenza al rumore della rete potete modificare i files che contengono le immagini di input, un pixel alla volta e presentarle in esecuzione. Quando avete il primo risultato scorretto calcolate la distanza di Hamming* tra l'immagine rumorosa e l'immagine originale: questa può essere una misura della resistenza al rumore di questa memoria associativa. Con questo programma è possibile editare il file di pesi delle connessioni tra i neuroni, cambiandone qualcuno per simulare un "guasto neuronale" (che potrebbe esistere in una implementazione hardware di rete neurale).

LIMITI DI QUESTA MEMORIA ASSOCIATIVA

I grandi limiti di questa memoria associativa sono:

- scarsa capacità di memoria
- possibilità di lavorare con soli inputs booleani (0-1)
- necessità di avere dati di input tutti ortogonali tra loro: questo significa, per esempio nel caso di immagini, che circa la metà dei pixels di ogni immagine devono essere differenti da quelli di tutte le altre immagini. Anche in questo caso è possibile calcolare la ortogonalità delle immagini con la distanza di Hamming: per funzionare bene la nostra memoria associativa richiede immagini che abbiano tra loro distanze di Hamming comprese tra 0.40 e 0.60.

*distanza di Hamming= $\sum_k |p_x(k) - p_y(k)|$ cioè sommatoria dei moduli della differenza pixel - pixel di due immagini

RETI NEURALI ERROR BACK PROPAGATION

INTRODUZIONE

Nel capitolo precedente abbiamo analizzato il funzionamento di una rete neurale strutturata come memoria associativa. Ricordiamo che una memoria associativa realizzata con tecnica neurale deve presentare una delle seguenti proprietà:

-un contenuto parzialmente corretto in input deve richiamare comunque l'output associato corretto

-un contenuto sfumato o incompleto in input deve trasformarsi in un contenuto corretto e completo in output

Quando parliamo di contenuto dell'input di una memoria associativa intendiamo una configurazione di valori booleani ai quali però possiamo dare significati più articolati tramite opportune interfacce software sugli input/output della rete. Le memorie associative analizzate hanno comunque una serie di limitazioni abbastanza gravi:

1)capacità di memoria bassa

2)spesso è necessaria ortogonalità dei vettori di input

3)le forme riconosciute devono essere linearmente separabili

4)incapacità di adattamento alla traslazione, all'effetto scala e alla rotazione.

5)possibilità di operare solo con dati booleani.

Spieghiamo meglio i punti 3 e 4 che forse possono risultare i più oscuri:

-la separabilità lineare di una forma da riconoscere si ha quando una retta(in due dimensioni) o un piano (in tre dimensioni) o un iperpiano(in n dimensioni) può separare i punti $X(k)=\{x_1(k),x_2(k),\dots,x_n(k)\}$ (corrispondenti alla forma k da riconoscere) da tutti gli altri punti che rappresentano forme differenti e quindi da discriminare. Ad esempio le due forme logiche AND e OR sono linearmente separabili,mentre non lo è la forma logica XOR (or esclusivo), come rappresentato in [fig.1](#).

-l'incapacità di adattamento alla traslazione e alla rotazione o all'effetto scala è importante nel riconoscimento di immagini di oggetti che dovrebbero potere essere riconosciuti indipendentemente da questi tre fattori.

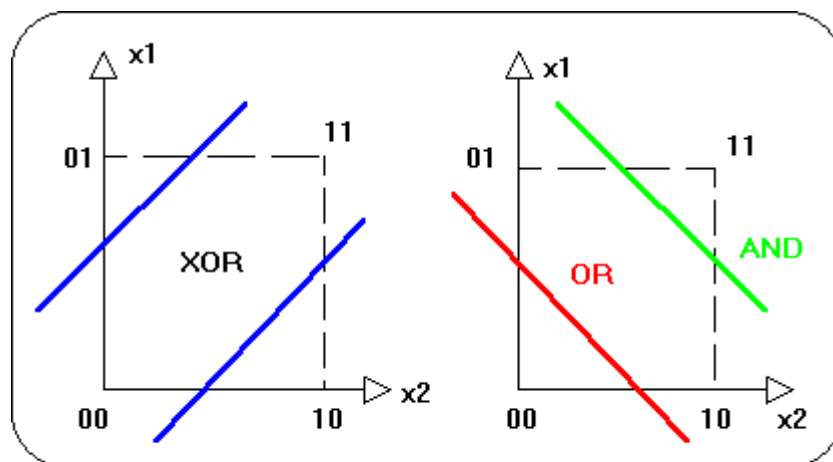
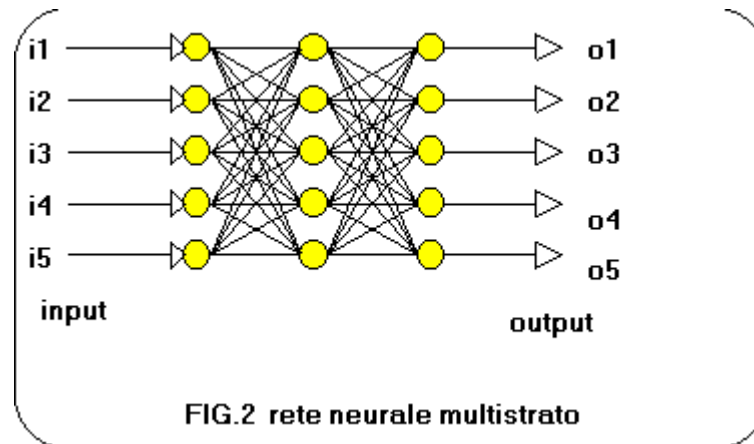


fig.1 SEPARAZIONE AND E OR CON UNO STADIO E
SEPARAZIONE XOR CON DUE STADI DECISIONALI

RETI NEURALI MULTISTRATO

Prossimamente ci occuperemo di reti neuronali che non sono impiegate come memorie associative ma sono in grado di svolgere funzioni più complesse, quali il riconoscimento di forme qualsiasi e la estrapolazione di correlazioni tra insiemi di dati apparentemente casuali. Nel fare questo passaggio dalle memorie associative alle reti multistrato tralasciamo il Perceptron, un tipo di rete neurale a due strati che ha senz'altro una importanza storica come capostipite delle attuali reti backprop e che vale la pena di menzionare. Il tipo di reti neurali che analizzeremo è unidirezionale nel senso che i segnali si propagano solamente dall'input verso l'output senza retroazioni che sono invece presenti nella memoria associativa BAM vista nel capitolo precedente (ricordate che la rete raggiungeva la stabilità come minimo energetico quando le oscillazioni dovute alla retroazione erano completamente smorzate?). Nel tipo di reti che vedremo i neuroni possono assumere valori reali (compresi tra 0.0 e 1.0) e non più valori booleani 0 e 1 per cui la flessibilità della rete risulta nettamente migliorata nella applicabilità a problemi reali. Ogni neurone di ogni strato sarà collegato ad ogni neurone dello strato successivo, mentre non vi saranno collegamenti tra i neuroni dello stesso strato ([fig.2](#)). Quella di figura 2 è la configurazione più semplice di rete multistrato, dotata di un solo strato intermedio normalmente chiamato "hidden layer": gli strati decisionali di tale rete sono lo strato intermedio e lo strato di output. Questa è un'affermazione che può sembrare scontata ma qualcuno di voi si sarà chiesto: "perché lo strato di neuroni di input non ha potere decisionale?" Ottima domanda! Facciamo un passo indietro ricordando che la memoria associativa del capitolo precedente imparava gli esempi dell'addestramento attraverso un particolare algoritmo che modificava i pesi dei collegamenti tra i neuroni. Ciò che una rete neurale impara sta proprio nel valore dei pesi che

collegano i neuroni opportunamente modificato in base ad una legge di apprendimento su un set di esempi. Allora comprendiamo che lo strato di input non è uno strato decisionale perché non ha pesi modificabili in fase di apprendimento sui suoi ingressi.



NOTE SULLA FORMA DELLE ESPRESSIONI MATEMATICHE DEI PROSSIMI PARAGRAFI

$\sum_{k=1}^N X(k)$ INDICA SOMMATORIA DI INDICE K DI X DA $K=1$ A $K=N$

(i limiti possono non essere presenti. Es: $\sum X(k) \cdot Y(j)$)

FUNZIONE DI TRASFERIMENTO DEL NEURONE

Ogni neurone ha una precisa funzione di trasferimento come avevamo già accennato parlando delle memorie associative nelle quali i neuroni hanno una funzione di trasferimento a gradino. In una rete `error_back_propagation` dove vogliamo avere la possibilità di lavorare con valori reali e non booleani, dobbiamo utilizzare una funzione di trasferimento a sigmoide ([fig.3](#)) definita dalla formula:

$$O = 1/(1+\exp(-I))$$

dove O =output del neurone, I =somma degli input del neurone

e in particolare $I = \sum_{k=1,n} w(j)(k) * x(k)$

La sigmoide di [fig.3](#) è centrata sullo zero ma in molte applicazioni è decisamente opportuno che il centro delle sigmoide di ogni neurone sia "personalizzato" dal neurone stesso per garantire una maggiore flessibilità di calcolo. Si può ottenere ciò modificando l'ultima formula nel seguente modo:

$$I = \sum_{k=1,n} w(j)(k) * x(k) - S(k)$$

dove $S(k)$ è il punto in cui è centrata la sigmoide del k -esimo neurone. Affinché tale soglia sia personalizzata è necessario che essa venga appresa (cioè si modifichi durante la fase di apprendimento) esattamente come i pesi delle connessioni tra i neuroni dei diversi strati. Con un piccolo trucco possiamo considerare tale soglia come un input aggiuntivo costante al valore 1 che è collegato al neurone k con un peso da "apprendere": la nostra rete si trasforma pertanto in quella di [fig.4](#). La formula per calcolare l'attivazione del neurone diventa:

$$I = \sum_{k=1,n+1} w(j)(k) * x(k)$$

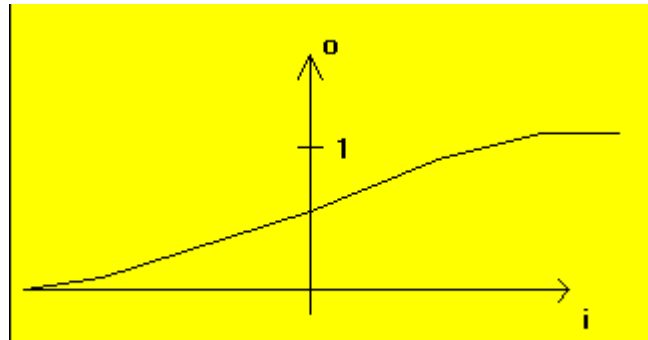


fig.3 funzione di trasferimento a sigmoide

ALGORITMO DI APPRENDIMENTO

Il tipo di addestramento che si esegue su questa rete è chiamato "supervised" (supervionato) perché associa ogni input ad un output desiderato:

```

ciclo epoche:
{
per ogni esempio:
{
a) si fornisce l'input alla rete
b) si preleva l'output da questa fornito
c) si calcola la differenza con l'output desiderato
d) si modificano i pesi di tutti i collegamenti
tra i neuroni in base a tale errore con una
regola (chiamata regola delta) in modo che tale
errore diminuisca.
}
e) calcolo dell'errore globale su tutti gli esempi
f) si ripete il ciclo epoche finché l'errore non raggiunge
il valore accettato
}

```

Naturalmente per input e output si intende un insieme di n esempi ciascuno composto da k input (k sono gli input fisici della rete) e j output (j sono gli output fisici della rete). Ogni ciclo come quello sopraesposto viene chiamato "epoca" di apprendimento. Voglio ricordare il concetto che nel capitolo precedente avevamo

chiamato "resistenza al rumore" (nel caso di memorie associative) e qui chiameremo "potere di generalizzazione" intendendone la capacità della rete dopo l'addestramento di dare una risposta significativa anche ad un input non previsto negli esempi. In pratica durante l'addestramento, la rete non impara ad associare ogni input ad un output ma impara a riconoscere la relazione che esiste tra input e output per quanto complessa possa essere: diventa pertanto una "scatola nera" che non ci svelerà la formula matematica che correla i dati ma ci permetterà di ottenere risposte significative a input di cui non abbiamo ancora verifiche "sul campo" sia all'interno del range di valori di addestramento (interpolazione), che all'esterno di esso (estrapolazione). Naturalmente l'estrapolazione è più difficoltosa e imprecisa che l'interpolazione e comunque entrambe danno risultati migliori quanto più è completo e uniformemente distribuito il set di esempi. Potremmo insegnare ad una rete neurale di questo tipo a fare delle somme tra due valori di input fornendole in fase di apprendimento una serie di somme "preconfezionate" con risultato corretto in un range più vasto possibile: la rete riuscirà a fare somme anche di valori diversi da quelli degli esempi. Addestrare una rete neurale a fare una somma è ovviamente inutile se non per motivi di studio o sperimentali in quanto essa è utile per studiare fenomeni di cui non sia conosciuta la correlazione matematica tra input e output (o sia estremamente complessa).

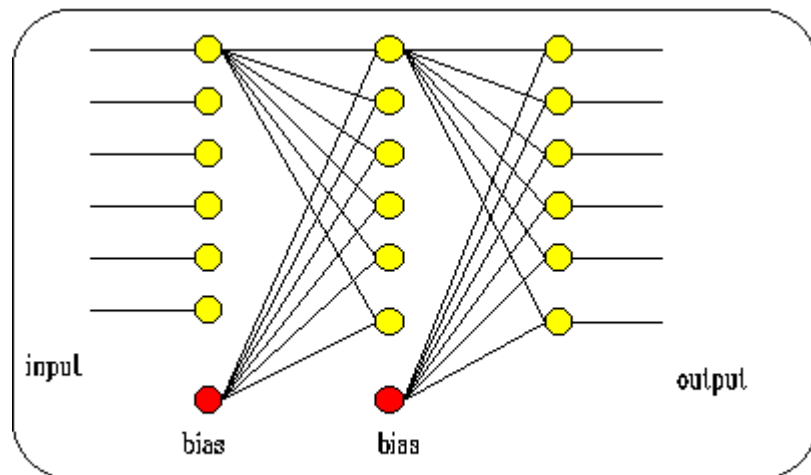


fig.4 inserimento del bias

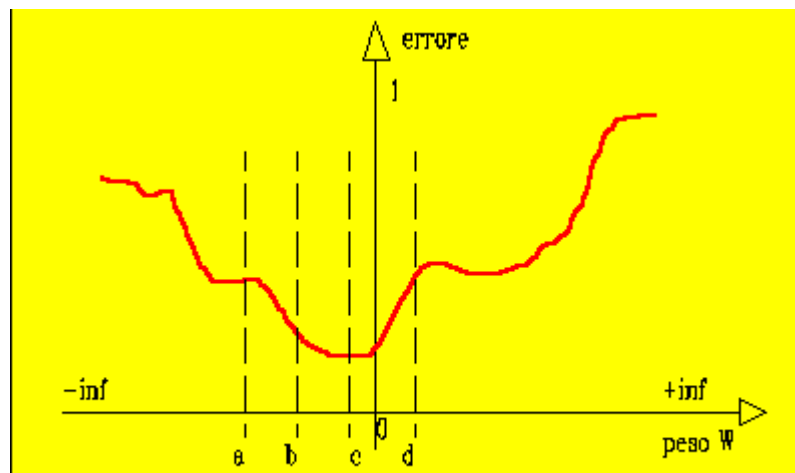


fig.5 andamento dell'errore in funzione di un peso

LA REGOLA DELTA

Ci troviamo esattamente al punto c del ciclo visto sopra e calcoliamo la differenza dell'output avuto all'unità j con quello desiderato: $err(j)=D(j)-y(j)$ Se per ogni esempio facessimo la somma algebrica degli errori di ogni output, poiché questi sono compresi tra $+1$ e -1 , rischieremmo di annullare l'errore globale e, comunque, commetteremmo un errore concettuale dato che per noi l'errore è tale sia nel senso negativo che positivo. Possiamo

ottenere una misura dell'errore globale facendo una somma dei quadrati degli errori (o in un programma in c usando una funzione che ci restituisca il valore assoluto di un float).

$err(j)=absolute(D(j)-y(j))$ o $err(j)=((D(j)-y(j))^{**2})/2$ Supponiamo che l'errore globale della rete così calcolato sia $e=0.3$, mentre noi desideriamo che l'errore sia 0.01 , ciò significa che dobbiamo cambiare i pesi delle connessioni ...ma come? Consideriamo il peso della connessione specifica che collega $y(k)$ dello strato di output al neurone $h(j)$ dello strato intermedio: naturalmente variando questo peso l'errore su $y(k)$ varia, supponiamo con una legge come quella di [fig.5](#). Il valore al quale dovremmo settare il peso della connessione è ovviamente il punto c che corrisponde al minimo dell'errore. Pertanto se noi riuscissimo a cercare tale punto minimo per ogni connessione avremmo ottenuto il risultato, cioè la rete sarebbe addestrata. La difficoltà di tale procedimento sta nel fatto che non possiamo analizzare singolarmente ogni connessione in modo sequenziale perchè la forma della funzione che lega ciascun peso con l'errore varia al variare degli altri pesi. Il problema della ottimizzazione dei pesi di una rete di tale tipo è in realtà la ricerca di un minimo di una funzione in uno spazio n -dimensionale che può essere ricondotto alla ricerca di un insieme di minimi coerenti di n funzioni in uno spazio bidimensionale. Esiste solamente una tecnica empirica per ottenere la soluzione che viene denominata "discesa del gradiente". Ricordando che la derivata di una funzione è una misura della pendenza della funzione in quel punto, possiamo spiegare tale tecnica come segue: partiamo da un punto casuale (i pesi all'inizio hanno valori casuali) e facciamo piccoli spostamenti di segno opposto e proporzionali alla derivata della funzione in quel punto (significa che se ci troviamo in A faremo un piccolo spostamento verso B e se ci troviamo in D faremo un piccolo spostamento verso C). In questo modo ci avviciniamo sempre più al minimo della funzione e quando sarà raggiunto, essendo la derivata in quel punto 0 , non effettueremo più spostamenti. Con questa tecnica noi possiamo

anche immaginare la forma della funzione modificarsi lentamente (a causa delle modifiche degli altri pesi) mentre noi ci muoviamo su di essa con spostamenti abbastanza piccoli da consentirci di tenere conto di tale movimento. Naturalmente il minimo raggiunto potrebbe non essere il minimo assoluto ma un minimo locale, però nell'insieme di tutti i minimi raggiunti è molto probabile che la grande maggioranza siano minimi assoluti. Esprimiamo in formula matematica il principio esposto nel seguente modo:

$$\Delta w_2 = -\epsilon \left(\frac{d \text{err}}{d w_2} \right)$$

dove $\frac{d \text{err}}{d w}$ = derivata dell' errore rispetto al peso

ϵ = costante di apprendimento che incide sulla misura dello spostamento a parità di "pendenza" nel punto specifico (è consigliabile un valore compreso tra 0.1 e 0.9)

Proviamo a sviluppare questa derivata nel seguente modo:

$$\Delta w_2(k)(j) = -\epsilon \left(\frac{d \text{err}}{d I(j)} \right) \left(\frac{d I(j)}{d w_2(k)(j)} \right)$$

definendo $\Delta(j) = \frac{d \text{err}}{d I(j)}$ si semplifica in

$$\Delta w_2(k)(j) = -\epsilon * \Delta(j) * \left(\frac{d I(j)}{d w_2(k)(j)} \right)$$

e ricordiamo che la formula di attivazione di un neurone dello strato di output è

$$I(j) = \sum_k w_2(k)(j) * h(k)$$

dove $h(k)$ = output del neurone k -esimo dello strato hidden

la sua derivata risulta $\frac{d I(j)}{d w_2(k)(j)} = h(k)$

ritorniamo adesso a Δ impostando

$$\delta(j) = (d \text{ err} / d y(j)) * (d y(j) / d I(j))$$

e risolviamo separatamente le due derivate in essa contenute:

$$\text{prima derivata) } d \text{ err} / d y(j) = d (((D(j)-y(j))^{**2})/2) / d y(j) = -(D(j)-y(j))$$

$$\text{seconda derivata) } d y(j) / d I(j) = d (1/(1+e^{**(-I(j))})) / d I(j) = y(j) * (1-y(j))$$

per cui dato che

$$\delta_w2(j) = -\epsilon * \delta(j) * (d I(j) / d (w2(k)(j)))$$

$$\text{si ha } \delta_w2 = -\epsilon * \delta(j) * h(k)$$

$$\text{e } \delta(j) = (D(j)-y(j)) * y(j) * (1-y(j))$$

abbiamo la formula che ci consente di calcolare i pesi delle connessioni tra i neuroni dello strato di output e quelli dello strato intermedio:

$$\delta_w2[j][k] = \epsilon * (D[j]-y[j]) * y[j] * (1-y[j]) * h[k]$$

dove conosciamo

D[j] come valore di output desiderato

y[j] come valore di output ottenuto

h[k] come stato di attivazione del neurone k del livello intermedio

LA RETROPROPAGAZIONE DELL'ERRORE

Con la formula sopraesposta potremmo già costruire un programma che effettui l'addestramento di una rete priva di strati nascosti (cioè con il solo strato di output decisionale), sostituendo allo strato nascosto lo strato di input. Se consideriamo una rete con uno strato hidden dobbiamo invece ripetere il calcolo precedente (tra output e hidden), tra lo strato hidden e lo strato di input. A questo punto sorge una difficoltà in più: nel primo calcolo noi conoscevamo l'errore sullo strato di output dato da $(D[j]-y[j])$, mentre adesso non conosciamo l'errore sullo strato hidden. Qui entra in gioco il concetto di retropropagazione dell'errore proprio per calcolare quale errore è presente nello strato nascosto (uso indifferentemente i termini hidden e nascosto) in corrispondenza dell'errore nello strato di output. La tecnica è quella di fare percorrere all'errore sull'output un cammino inverso attraverso le connessioni pesate tra output e hidden (da qui il nome "retropropagazione"). Tradurre in termini matematici ciò che sembra un concetto fisicamente banale risulta invece un lavoro un po' più complesso ma ci proviamo lo stesso iniziando con il definire la formula di modifica per i pesi tra hidden e input analoga alla precedente (discesa del gradiente):

$$\Delta w_{1[j][k]} = -\epsilon * (d \text{ err} / d w_{1[j][k]})$$

sviluppiamo la derivata nel seguente modo:

$$\Delta w_{1[j][k]} = -\epsilon * (d \text{ err} / d I[k]) * (d I[k] / d w_{1[j][k]})$$

$$\text{chiamiamo } \Delta[k] = -(d \text{ err} / d I[k])$$

e applicando la regola di composizione delle derivate possiamo scrivere

$$\Delta[k] = -(d \text{ err} / d I[j]) * (d I[j] / d h[k]) * (d h[k] / d I[k])$$

dove $h[k]$ =attivazione del k -esimo neurone hidden

quindi anche

$$\delta[k] = -(E(j) \frac{d \text{err}}{d I[j]} \frac{d I[j]}{d h[k]}) \frac{d h[k]}{d I[k]}$$

dato che l'operatore sommatoria agisce solo sui fattori aggiunti che si annullano.

Notiamo ora che il primo termine $(\frac{d \text{err}}{d I[j]} = \delta[j])$ che abbiamo incontrato nei calcoli precedenti e che il secondo $(\frac{d I[j]}{d h[k]})$ può essere calcolato come

$$\frac{d E(k)}{d w_2[k][j]} \frac{d w_2[k][j]}{d h[k]} = w_2[k][j]$$

mentre il terzo $\frac{d h[k]}{d I[k]} = \frac{d h[k]}{d (1/(1+e^{-h[k]}))} = h[k] * (1-h[k])$

otteniamo finalmente:

$$\delta_{w_1[j][k]} = \epsilon * \delta[k] * x[j]$$

in cui

$x[i]$ = i -esimo input

$$\delta[k] = (E[j] \delta[j] * w_2[k][j]) * h[k] * (1-h[k])$$

in cui

$h[k]$ = attivazione neurone k dello strato hidden (conosciuto ovviamente eseguendo la rete, cioè moltiplicando gli input per i pesi delle connessioni e sommando tutto)

$$\delta[j] = (D[j] - y[j]) * (y[j]) * (1 - y[j])$$

già calcolato nella parte precedente (quella relativa alla modifica dei pesi tra output e hidden). Notate che $\delta[j]$ contiene effettivamente la retropropagazione dell'errore e infatti è l'unico

termine che contiene dati relativi allo strato di output contenuto nel calcolo del Δw_1 .

SIMULAZIONE IN C DI ESECUZIONE E APPRENDIMENTO

Il listato contenuto nel dischetto è un programma di prova di reti neurali `error_back_propagation` che permette di addestrare la rete con un file `ascii` contenente i dati `input` e `output` desiderato in sequenza come mostrato in [fig.6](#).

```
.2345 input1 esempio1
.3456 input2
.1234 output
.6789 input1 esempio2
.4567 input2
.4567 output
.....
.5678 input1 esempioN
.4567 input2
.8976 output
```

fig.6


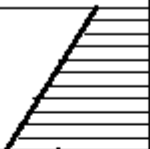
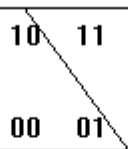

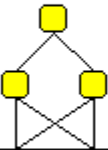
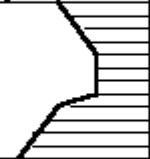
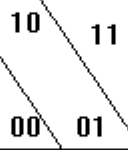

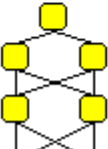

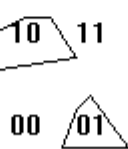

numero strati	forme	xor	note	regioni incuneate
 1			semipiani	
 2			regioni convesse	
 3			regioni complesse	

fig.7 forme riconoscibili con diversi numeri di strati

Questo programma prevede l'utilizzo di due hidden layers e quindi la retropropagazione dell'errore avviene due volte e non una sola: dallo strato di output allo strato hidden2 e dallo strato hidden2 allo strato hidden1. La scelta di inserire due strati hidden di neuroni non è casuale, infatti, nonostante che la grande maggioranza dei problemi possano essere risolti con un solo strato nascosto, esiste una netta differenza tra le reti a due strati decisionali (output+1strato hidden) e quelle con tre strati decisionali (output+due strati hidden): come scritto in un rapporto della D.A.R.P.A (Defensive Advanced Research Project Agency) sulle reti neurali, con uno strato decisionale(output) è possibile realizzare una separazione lineare, mentre con due strati (output+hidden1) possono essere separati spazi convessi e, con tre strati decisionali (output+ +hidden1 +hidden2) possono essere riconosciute forme qualsiasi ([fig.7](#)). Possiamo pensare che il primo strato decisionale sia in grado di separare i punti appartenenti a forme diverse con un semipiano o un iperpiano (separazione lineare) per ogni nodo e che il secondo strato decisionale faccia altrettanto intersecando i suoi semipiani con quelli del primo strato a formare regioni convesse(per ogni nodo): il terzo strato decisionale associa regioni convesse di nodi differenti dello strato precedente creando forme qualsiasi la cui complessità dipende dal

numero di nodi dei primi due strati e il cui numero dipende dal numero di nodi del terzo strato. I vantaggi relativi all'uso di più di due strati nascosti sono decisamente inferiori e comunque ne discuteremo eventualmente in un prossimo capitolo parlando, più approfonditamente, di potere di generalizzazione delle reti. Le due parti di programma che eseguono le due retropropagazioni sono assolutamente simmetriche dato che i calcoli da fare sono esattamente gli stessi: basta cambiare gli operatori.

Il programma contiene le seguenti procedure

`exec`: esegue la rete con un input

`back_propagation`: esegue la retropropagazione dell'errore modificando i pesi delle connessioni

`ebp_learn`: carica i parametri dell' addestramento e chiama `learn`

`learn`: procedura che esegue l' addestramento della rete nel seguente modo...

```

ciclo epoche
{
    ciclo esempi
    {
        chiama exec(esecuzione della rete)
        chiama back_propagation(modifica pesi)
        calcola errore esempio
        calcola errore epoca=max(errori esempi)
    }
    break se errore epoca < errore ammesso
}

```

`input`: preleva i dati dal file `net.in` in test

`output`: inserisce i dati nel file `net.out` in test

`weight_save`: salva i pesi della rete su file

`load_weight`: carica i pesi della rete da file

Bisogna innanzitutto definire i parametri della rete che sono numero di input e numero di output (dipendenti ovviamente dal problema applicativo) e numero di neuroni che vogliamo negli strati hidden. Esistono delle formule empiriche per calcolare il numero dei neuroni degli strati nascosti in base alla complessità del problema ma, generalmente, è più la pratica che suggerisce tale numero (spesso è inferiore al numero degli input e output). Con più neuroni negli strati hidden l'apprendimento diventa esponenzialmente più lungo, nel senso che ogni epoca occupa un tempo maggiore ($n_pesi = n_neuroni1 * n_neuroni2$), ma non è escluso che il risultato (raggiungimento del target) venga raggiunto in un tempo simile a causa di una maggiore efficienza della rete.

UN ESPERIMENTO: INSEGNAMO ALLA RETE A FARE LA SOMMA DI DUE NUMERI

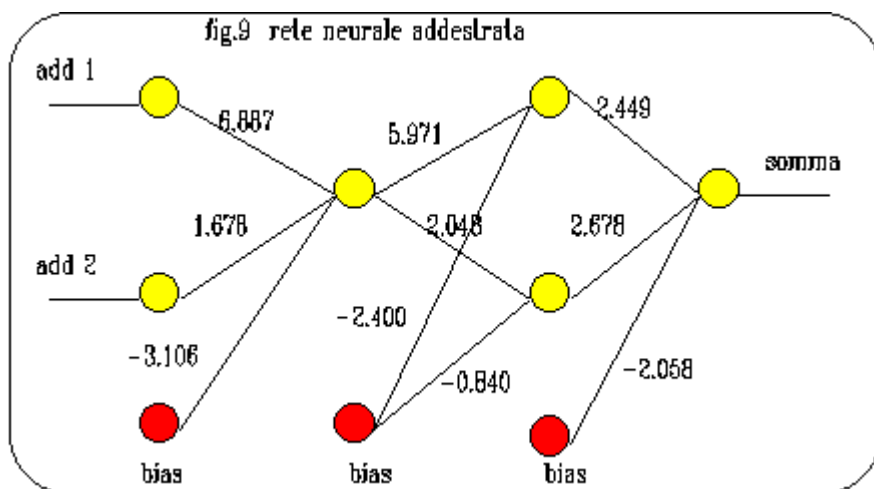
Utilizziamo il training set di [fig.8](#) che contiene 40 esempi normalizzati di somma di due numeri: il programma presentato in questo capitolo, su una SPARK station, è arrivato al raggiungimento del target 0.05 dopo pochi minuti, in circa 4300 epoche con un epsilon (tasso di apprendimento) = 0.5 e con quattro neuroni per ogni strato intermedio (ovviamente $n_input = 2$ e $n_output = 1$).

Il raggiungimento del target_error 0.02 è stato ottenuto nelle stesse condizioni all'epoca ~13800. Se non volete attendere molto tempo e verificare ugualmente la convergenza della rete potete ridurre il training set a 10 o 20 esempi curando che siano adeguatamente distribuiti (8 è dato dalla somma 1+7 ma anche 4+4 e 7+1). Con un training set ridotto la rete converge molto più velocemente ma è chiaro che il potere di generalizzazione risulta inferiore, per cui presentando poi alla rete degli esempi fuori dal training set l'errore ottenuto potrebbe essere molto più elevato del target_error

raggiunto. È buona norma lavorare con delle pause su un numero di epoche non elevato al fine di poter eventualmente abbassare il tasso di apprendimento epsilon quando ci si accorge della presenza di oscillazioni (errore che aumenta e diminuisce alternativamente) dovute a movimenti troppo lunghi (epsilon alto) intorno ad un minimo (che speriamo sia il target e non un minimo locale). Il raggiungimento del target 0.02 è relativamente rapido mentre da questo punto in poi la discesa verso il minimo è meno ripida e i tempi diventano più lunghi (è il problema della discesa del gradiente con movimenti inversamente proporzionali alla derivata nel punto). Inoltre avvicinandosi al minimo è prudente utilizzare un epsilon basso per il motivo esaminato prima. È sicuramente possibile che un errore target non sia raggiungibile in tempi accettabili se è troppo basso in relazione alla complessità del problema. In ogni caso bisogna ricordare che le reti neurali non sono sistemi di calcolo precisi ma sistemi che forniscono risposte approssimate a inputs approssimati. Se ripetete due o più volte lo stesso training con gli stessi dati e gli stessi parametri potreste sicuramente notare differenze di tempi di convergenza verso il target dovuti al fatto che inizialmente i pesi della rete sono settati a valori random (procedura `weight_starter`) che possono essere più o meno favorevoli alla soluzione del problema. Nella [fig.9](#) è visualizzato il `know_how` di una rete con un neurone nello strato `hidden1` e due neuroni nello strato `hidden2`, dopo l'addestramento alla somma fino al `target_error=0.05`: sono evidenziati i numeri che rappresentano i pesi dei collegamenti.

add1	add2	sum	add1	add2	sum
0.40	0.30	0.70	0.11	0.44	0.55
0.22	0.33	0.55	0.22	0.33	0.55
0.37	0.37	0.74	0.33	0.22	0.55
0.49	0.37	0.86	0.44	0.11	0.55
0.12	0.12	0.24	0.33	0.11	0.44
0.11	0.11	0.22	0.22	0.22	0.44
0.13	0.13	0.26	0.11	0.33	0.44
0.11	0.88	0.99	0.22	0.11	0.33
0.22	0.77	0.99	0.11	0.11	0.22
0.33	0.67	1.00	0.16	0.16	0.32
0.44	0.56	1.00	0.05	0.05	0.10
0.55	0.45	1.00	0.77	0.11	0.88
0.66	0.34	1.00	0.66	0.22	0.88
0.77	0.23	1.00	0.55	0.33	0.88
0.88	0.12	1.00	0.44	0.44	0.88
0.11	0.55	0.66	0.33	0.55	0.88
0.22	0.44	0.66	0.22	0.66	0.88
0.33	0.33	0.66	0.11	0.77	0.88
0.44	0.22	0.66	0.11	0.66	0.77
0.55	0.11	0.66	0.22	0.55	0.77

fig.8 somma di valori normalizzati(compresi tra 0 e 1)



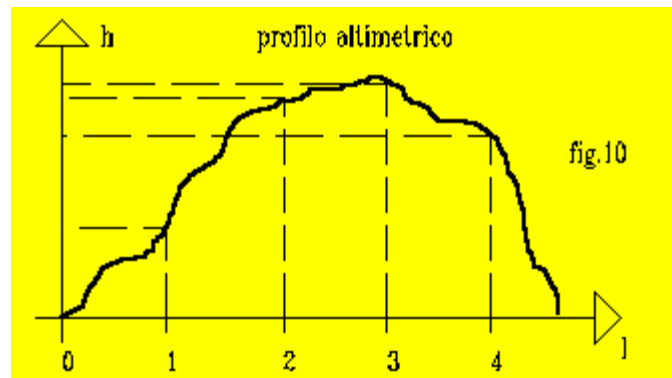
UN ESEMPIO APPLICATIVO

Una rete neurale può essere addestrata al riconoscimento di profili altimetrici: utilizziamo una rete con 5 input che costituiscono 5 valori di altezza consecutivi e 5 output corrispondenti alle seguenti scelte:

- 1)monte,
- 2)valle,
- 3)pendenza negativa,
- 4)pendenza positiva,
- 5)piano(pendenza nulla)

[\(fig.10\)](#)

Utilizziamo il training set di [fig.11](#) per addestrare la rete e il validation set di [fig.12](#) per verificare la capacità di generalizzazione della rete. Nelle figure 11 e 12 sono presentati i risultati ottenuti su una rete con cinque neuroni su ogni strato hidden. Si arriva al raggiungimento del $\text{target_error}=0.02$ in ~ 5600 epoche e in ~ 23000 epoche si raggiunge l'errore 0.01 con un $\text{epsilon}=0.5$. Come si può constatare il potere di generalizzazione, almeno per il validation set utilizzato, è ottimo nonostante che il training set sia costituito di soli 20 esempi(i valori ottenuti sono riferiti all' esecuzione della rete con i pesi relativi al $\text{target_error}=0.01$). Come noterete negli esempi i valori sono equamente distribuiti tra le cinque categorie per ottenere un buon risultato, poiché la rete neurale (un po' come il nostro cervello) tende a diventare più sensibile agli eventi che si verificano più spesso o anche a quelli che si sono verificati più spesso nell' ultimo periodo. In questo semplice esempio applicativo abbiamo utilizzato una rete ebp in modo un po' anomalo, cioè come classificatore, nel senso che non abbiamo una risposta "analogica" al nostro input ma una risposta booleana che assegna ad una classe il nostro input pattern(configurazione degli input).



inputs					output atteso					output ottenuto				
h1	h2	h3	h4	h5	M	V	P+	P-	P0	m	v	p+	p-	p0
0.5	0.6	0.7	0.1	0.0	1	0	0	0	0	.99	.00	.00	.00	.00
0.3	0.2	0.1	0.5	0.6	0	1	0	0	0	.00	.99	.00	.00	.00
0.6	0.7	0.8	0.9	1.0	0	0	1	0	0	.00	.00	.99	.00	.00
0.8	0.6	0.5	0.3	0.1	0	0	0	1	0	.00	.00	.00	.99	.00
0.1	0.1	0.1	0.1	0.1	0	0	0	0	1	.00	.00	.00	.00	.99
0.4	0.5	0.6	0.4	0.3	1	0	0	0	0	.99	.00	.00	.00	.00
0.6	0.5	0.3	0.4	0.7	0	1	0	0	0	.00	.99	.00	.00	.00
0.4	0.5	0.7	0.8	0.9	0	0	1	0	0	.00	.00	.99	.00	.00
0.7	0.5	0.4	0.3	0.1	0	0	0	1	0	.00	.00	.00	.99	.00
0.0	0.0	0.0	0.0	0.0	0	0	0	0	1	.00	.00	.00	.00	.99
0.1	0.4	0.6	0.2	0.1	1	0	0	0	0	.99	.00	.00	.00	.00
0.8	0.5	0.5	0.7	0.9	0	1	0	0	0	.00	.99	.00	.00	.00
0.0	0.1	0.3	0.6	0.9	0	0	1	0	0	.00	.00	.99	.00	.00
0.9	0.5	0.4	0.1	0.0	0	0	0	1	0	.00	.00	.00	.99	.00
0.4	0.4	0.4	0.4	0.4	0	0	0	0	1	.00	.00	.00	.00	.99
0.3	0.6	0.9	0.6	0.5	1	0	0	0	0	.99	.00	.00	.00	.00
0.8	0.6	0.5	0.7	0.9	0	1	0	0	0	.00	.99	.00	.00	.00
0.5	0.6	0.7	0.9	1.0	0	0	1	0	0	.00	.00	.99	.00	.00
0.7	0.5	0.4	0.3	0.1	0	0	0	1	0	.00	.00	.00	.99	.00
0.8	0.8	0.8	0.8	0.8	0	0	0	0	1	.00	.00	.00	.00	.99

fig.11 training set

inputs					output atteso					output ottenuto				
h1	h2	h3	h4	h5	M	V	P+	P-	P0	m	v	p+	p-	P0
0.0	0.1	0.5	0.4	0.2	1	0	0	0	0	.99	.00	.00	.00	.00
0.5	0.3	0.1	0.3	0.4	0	1	0	0	0	.00	.98	.00	.00	.03
0.1	0.2	0.5	0.7	0.9	0	0	1	0	0	.00	.00	.99	.00	.00
0.5	0.4	0.2	0.1	0.0	0	0	0	1	0	.00	.00	.00	.99	.00
0.5	0.5	0.5	0.5	0.5	0	0	0	0	1	.00	.00	.00	.00	.99
0.3	0.4	0.5	0.3	0.2	1	0	0	0	0	.99	.00	.00	.00	.01
0.6	0.4	0.1	0.3	0.8	0	1	0	0	0	.00	.99	.00	.00	.00
0.4	0.5	0.6	0.7	0.9	0	0	1	0	0	.00	.00	.99	.00	.00
0.4	0.3	0.2	0.1	0.0	0	0	0	1	0	.00	.00	.00	.98	.00
0.1	0.1	0.1	0.1	0.1	0	0	0	0	1	.00	.00	.00	.00	.99
0.4	0.5	0.6	0.4	0.3	1	0	0	0	0	.99	.00	.00	.00	.00
0.5	0.4	0.1	0.6	0.8	0	1	0	0	0	.00	.99	.00	.00	.00
0.1	0.2	0.3	0.4	0.6	0	0	1	0	0	.00	.00	.99	.00	.00
0.9	0.5	0.3	0.2	0.1	0	0	0	1	0	.00	.00	.00	.99	.00
0.7	0.7	0.7	0.7	0.7	0	0	0	0	1	.00	.00	.00	.00	.99

fig.12 validation set(test potere generalizzazione)

CONCLUSIONI

Prossimamente analizzeremo un programma che permette di addestrare un rete neurale ebp e di generare il codice c relativo come i più sofisticati programmi di simulazione. In quel programma saranno presenti particolari tools che ci permetteranno di approfondire alcune problematiche e tecniche di addestramento avanzate relative alle reti neurali, tra le quali il superamento di minimi locali con "simulated anealing" e "termoshok" o l'addestramento con algoritmi genetici.

APPLICAZIONI PRATICHE DI UNA RETE

INTRODUZIONE

I campi di applicazione delle reti neurali sono tipicamente quelli dove gli algoritmi classici, per la loro intrinseca rigidità (necessità di avere inputs precisi), falliscono. In genere i problemi che hanno inputs imprecisi sono quelli per cui il numero delle possibili variazioni di input è così elevato da non poter essere classificato. Ad esempio nel riconoscimento di un'immagine di soli 25 pixels (5×5) in bianco e nero senza toni di grigio abbiamo 2^{25} possibili immagini da analizzare. Se consideriamo una immagine di 1000 pixels con 4 toni di grigio dovremo analizzare 4^{1000} immagini possibili. Per risolvere questi problemi si utilizzano normalmente algoritmi di tipo probabilistico che, dal punto di vista della efficienza risultano, comunque, inferiori alle reti neurali e sono caratterizzati da scarsa flessibilità ed elevata complessità di sviluppo. Con una rete neurale dovremo prendere in considerazione non tutte le immagini possibili ma soltanto quelle significative: le possibili variazioni in un range intorno all'immagine sono già riconosciute dalla proprietà intrinseca della rete di resistenza al rumore. Un altro campo in cui gli algoritmi classici sono in difficoltà è quello dell'analisi di fenomeni di cui non conosciamo regole matematiche. In realtà esistono algoritmi molto complessi che possono analizzare tali fenomeni ma, dalle comparazioni fatte sui risultati, pare che le reti neurali risultino nettamente più efficienti: questi algoritmi sfruttano la trasformata di Fourier per scomporre il fenomeno in componenti frequenziali e pertanto risultano molto complessi e riescono ad estrarre un numero limitato di armoniche generando notevoli approssimazioni. Come vedremo, una rete neurale addestrata con i

dati di un fenomeno complesso sarà in grado di fare previsioni anche sulle sue componenti frequenziali e ciò significa che realizza al suo interno una trasformata di Fourier anche se nessuno le ha mai insegnato come funziona!

RICONOSCIMENTO IMMAGINI

Con una rete neurale tipo `error_back_propagation`, al contrario di una memoria associativa (che ammette solo valori 1/0 di input), possiamo pensare di analizzare immagini con vari livelli di grigio abbinando ad ogni input un pixel dell'immagine e il livello di grigio definito dal valore (compreso tra 0.0 e 1.0) dell'input. L'utilizzo potrebbe essere quello di riconoscere un particolare tipo di immagine o di classificare le immagini ricevute in input: il numero degli output pertanto è uguale al numero di classi previste ed ogni output assume un valore prossimo a 1 quando l'immagine appartiene alla classe ad esso corrispondente, altrimenti un valore prossimo a 0. Utilizzando una rete con 100 input si possono classificare immagini di 10*10 pixel ([fig.1](#)). Dovendo analizzare immagini più complesse (esempio 900 pixel 30*30) risulta più pratico utilizzare uno schema di apprendimento modulare anziché una singola rete con 900 input. Ad esempio possiamo utilizzare 9 reti da 100 input ciascuna che riconoscono una parte definita della immagine divisa in zone, ed infine una rete neurale con 9 input viene addestrata con le combinazioni di uscita delle 9 reti ([fig.2](#)). In [tab.1](#) si vedono i dati di training della rete finale che ha il compito di riconoscere se il numero di reti precedenti che ha riconosciuto la stessa classe di appartenenza è sufficiente a stabilire che l'immagine appartiene a tale classe: in pratica la tabella rivela una funzione di AND che però risulta "elasticizzato" dalla rete (per semplicità consideriamo solo 3 zone). In un tale sistema la resistenza al rumore della rete finale viene posta "in

serie" con quella delle reti precedenti e pertanto è conveniente effettuare addestramenti con raggiungimento di errori abbastanza piccoli al fine di evitare una eccessiva flessibilità del sistema.

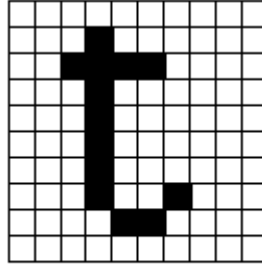


fig.1

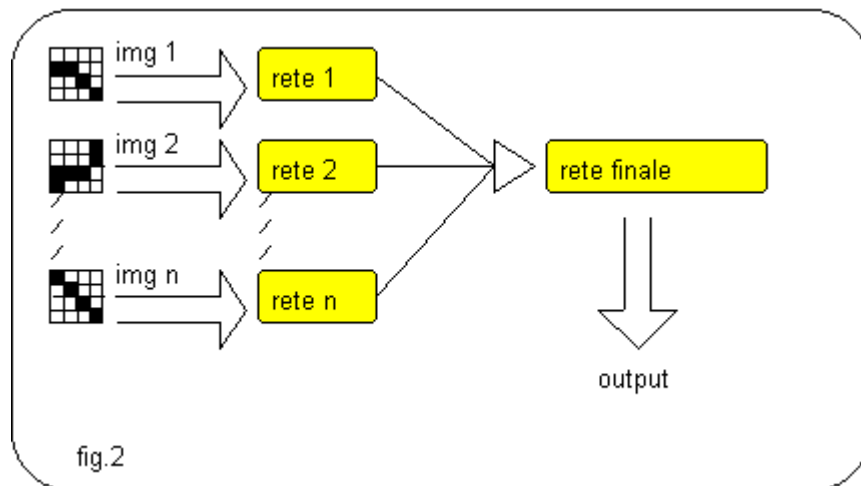


fig.2



FIG.3

net1	net2	net3	out
0	0	0	0
1	0	0	0
0	1	0	0
1	1	0	0
0	0	1	0
1	0	1	0
0	1	1	0
1	1	1	1

tab.1

RICONOSCIMENTO SCRITTURA

E abbastanza evidente come sia facilmente trasportabile il problema del riconoscimento scrittura manuale al problema del riconoscimento immagini. Se pensiamo alle lettere manoscritte incasellate in un modulo quadrettato e letto da uno scanner, il problema si riduce al riconoscimento dell'immagine contenuta in ogni quadrato in modo sequenziale ([fig.3](#)). Se la scrittura è invece completamente libera il problema diventa più complesso e si rende necessario un preprocessor che sia in grado di ricomporre la scrittura scomponendo le singole lettere su matrici definite: i risultati sono ovviamente meno affidabili.

PREVISIONE DI FENOMENI COMPLESSI

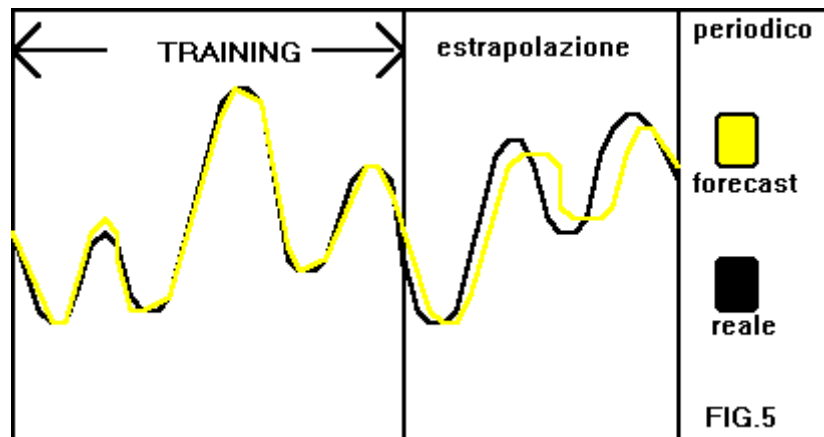
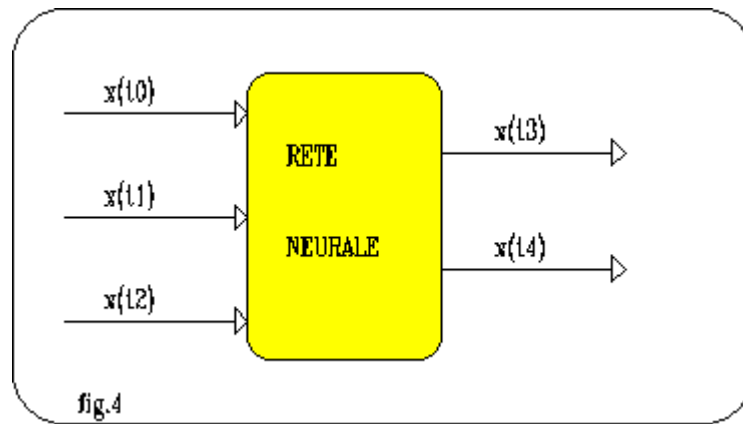
Una delle applicazioni più importanti delle reti neurali è sicuramente quella delle previsioni di fenomeni complessi come i fenomeni meteorologici o quelli finanziari o socio-economici. Esistono metodi per la previsione di tali fenomeni che si basano su tre diverse linee di principio:

- 1) classico
- 2) frequenziale
- 3) moderno

Non vogliamo analizzare in questa sede queste tre teorie, ma accennare solo alla seconda che si basa sulla scomposizione in componenti armoniche secondo la legge di Fourier. Il principale difetto di questo metodo è che i calcoli relativi alle componenti frequenziali più alte appesantiscono eccessivamente l'algoritmo al punto che si rende necessario accontentarsi di selezionare le armoniche che si ritengono maggiormente influenti, ottenendo un evidente errore di approssimazione. Con una rete neurale è possibile fare previsioni analizzando le serie storiche dei dati

esattamente come con questi sistemi ma non è necessario fare supposizione alcuna per restringere il problema ne, tanto meno, applicare la trasformata di Fourier. un difetto comune ai metodi di analisi sopra elencati è quello di essere applicabili solamente se si attuano delle restrizioni nel problema utilizzando delle ipotesi che talvolta potrebbero rivelarsi errate. In pratica si addestra la rete neurale con successioni di serie storiche di dati del fenomeno che si vuole prevedere. Supponiamo di voler prevedere, sulla base di n valori consecutivi che la variabile $x(t)$ ha assunto, i successivi m valori che assumerà: addestriamo la rete di [fig.4](#) con la prima serie storica di n dati della variabile in input e la seconda serie di successivi m dati in output e così via per altre coppie di serie storiche ciascuna delle quali rappresenta un esempio. In questo modo la rete apprende l'associazione che esiste tra i valori della variabile in n punti e quelli in m punti successivi ma anche l'associazione tra le derivate in quanto l'informazione di due soli valori vicini della variabile è un indice della derivata in quel punto. Esistono due tipi di previsione: univariata e multivariata. La prima riguarda la previsione dei valori di una sola variabile di un fenomeno in base ai dati storici della variabile stessa. La seconda riguarda la previsione dei valori di più variabili in base ai dati storici delle stesse ed eventualmente anche di altre variabili: in questo secondo caso è l'insieme dei valori delle sequenze storiche di tutte le variabili di input che concorre alla determinazione dell'output di ognuna delle variabili su cui si vuole fare la previsione. Esistono casi in cui le sequenze storiche elementari (quelle che rappresentano un esempio) devono essere composte da molti dati consecutivi per avere dei buoni risultati e ciò comporta la necessità di utilizzare reti con un numero di inputs molto elevato: è possibile fare questo ma, talvolta, si preferisce utilizzare "reti ricorrenti" che sono reti neurali dotate di "memorie sugli input" tali che ogni variabile possa avere un solo input fisico ma la rete ad ogni istante t sia condizionata non solo dagli input

dell'istante t ma anche da quelli degli istanti precedenti ($t-1 \dots t-n$, dove n rappresenta l'ampiezza delle sequenze storiche).



PREVISIONE UNIVARIATA

Si tratta in pratica di ciò di cui si è già parlato riferendoci alla [fig.4](#). Vediamo due esempi di previsione di due processi dei quali il primo è di tipo periodico (naturalmente la previsione di un processo periodico ha senso solo a livello didattico), e il secondo aperiodico. Possiamo effettuare previsioni sulla funzione: $x(t) = \sin(2 \cdot \pi \cdot t / 40) + \sin(2 \cdot \pi \cdot t / 100)$ [due sinusoidi con periodi 40 e 100]. Utilizziamo il generatore di file di esempi `sin_gen` e addestriamo una rete neurale `error_back_propagation` con il file di esempi generato. Usiamo una rete neurale con 2

inputs che corrispondono ad una serie storica di 2 valori e con 1 solo output che corrisponde al valore successivo previsto: con un training set di 40 esempi(120 valori del file period.lrn composto di 200 valori) e periodo 40 (la seconda senoide assume automaticamente periodo 100), otteniamo il risultato di [fig.5](#) (il target_error=0.01. E stato ottenuto con un numero considerevole di epoche su una spark station). Il risultato viene ottenuto ripetendo piu volte la funzione exec del programma di simulazione di rete neurale con valori di input scelti nel range dei valori possibili della funzione Utilizzando una rete con due input e un output abbiamo una informazione su due valori precedenti ma anche sulla derivata in quel punto. Per effettuare l'addestramento con con il programma presentato nel capitolo precedente si può utilizzare lo stesso training set impostando diversamente il numero degli input e degli output(tenendo presente che deve essere $(n_{in} + n_{out}) * n_{es} < n_{gen}$). Avendo il programma utilizzato una funzione di trasferimento a sigmoide che fornisce valori compresi tra 0 e 1, il programma sin_gen genera degli esempi normalizzati con valori che variano tra 0.0 e 1.0.

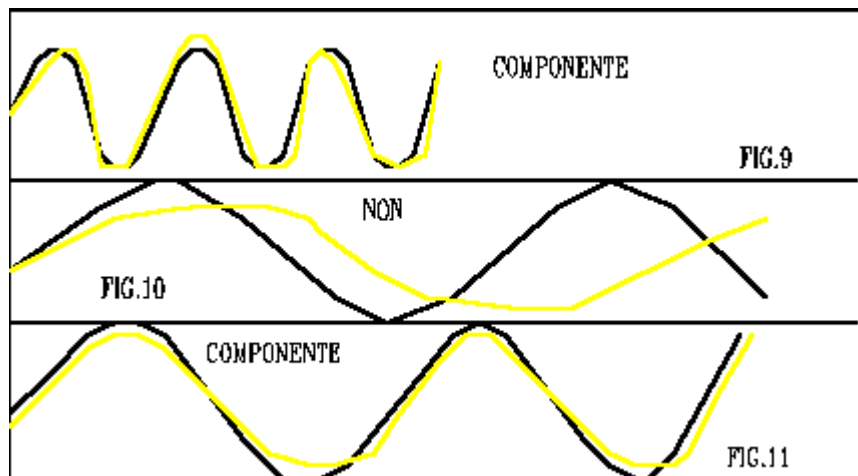
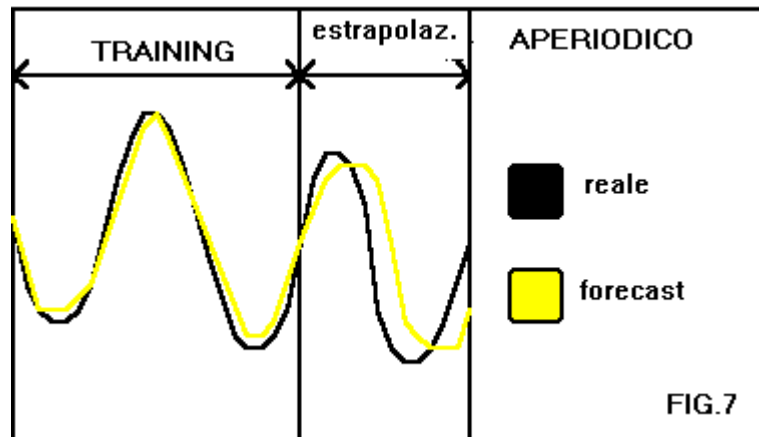
NOTA IMPORTANTE: bisogna distinguere le previsioni fatte nel range del training set(interpolazioni), che non sono previsioni vere e proprie, da quelle effettuate al di fuori di esso che rappresentano la reale capacità predittiva della rete neurale.

IN PRATICA: nella previsione di fenomeni complessi si utilizzano serie storiche composte da un numero molto maggiore di dati e i valori da predire sono sempre più di uno a seconda del campo predittivo che l'applicazione richiede.

Passiamo adesso ad analizzare una funzione aperiodica che risulta essere sicuramente più interessante ai fini delle previsioni. Utilizziamo una funzione composta dalla sovrapposizione di due sinusoidi di cui una con periodo incommensurabile:

$$x(t)=\sin(2*\text{pigreco}*t/40)+\sin(\text{sqrt}(2*\text{pigreco}*t/40))$$

Generiamo il file di esempi con il generatore `sin_gen` selezionando la funzione 2 e addestriamo una rete con tre inputs e un output (usiamo 5 neuroni in ogni strato hidden). Il risultato di previsione che otteniamo dopo un addestramento che porta ad un errore=0.01 è quello di [fig.7](#). L'addestramento è stato effettuato con 16 esempi che costituiscono 64 generazioni (16*(3input+1output)) delle 100 generate da `sin_gen` utilizzando l'opzione `aperiodica` e `periodo=40`. Adesso proviamo a verificare se la rete addestrata con il processo `aperiodico` è in grado di fare predizioni anche sulle componenti frequenziali del processo stesso. Possiamo generare con `sin_gen` due files di riferimento con i valori reali delle componenti frequenziali e prelevare da essi delle serie storiche di tre dati consecutivi da usare come input della funzione `exec` e costruire la funzione in base al valore dato come output nel file `net.out`. Sia nel caso della prima componente che nel caso della seconda abbiamo risultati di previsione analoghi a quelli della funzione composta e ne deduciamo che la rete è in grado di riconoscere le componenti frequenziali del processo, proprio come se applicasse una trasformata di Fourier "virtuale". Per avere una controprova di ciò possiamo tentare di fare una previsione su una senoide di frequenza differente da quelle componenti: la rete fa delle previsioni con errori tangibilmente maggiori dimostrando di essere sensibile solamente alle componenti frequenziali del processo relativo ai dati di addestramento ([fig.9](#) [fig.10](#) [fig.11](#)). Effettivamente questa controprova risulta molto più evidente se l'addestramento avviene con serie storiche più ampie di quella utilizzata, attraverso le quali la rete assume informazioni più precise in merito alla frequenza delle varie componenti.



PROCESSI NON STAZIONARI

Negli esempi precedenti abbiamo esaminato processi stazionari, cioè processi in cui la variabile considerata può variare entro un range ben definito. Se consideriamo processi non stazionari la variabile da predire può variare al di fuori di qualunque range di valori e ciò comporta un problema nella normalizzazione dei dati. La normalizzazione infatti consiste nel riportare valori reali del processo a valori compresi tra 0 e 1 o tra -1 e 1 a seconda del tipo di rete e di funzione di trasferimento dei neuroni: si ottiene questo nel modo più semplice dividendo tutti i valori del processo per il valore più alto di essi, ma in un processo non stazionario quest'ultimo non è conosciuto. Per risolvere questo problema si rende necessario effettuare una normalizzazione non lineare del

processo in modo che la variabile analizzata, che può teoricamente variare tra $-\infty$ e $+\infty$, vari tra $-k$ e $+k$. Si possono utilizzare funzioni logaritmiche, radici cubiche e funzioni non lineari come la sigmoide utilizzata nella funzione di trasferimento dei neuroni di una rete e_b_p. Naturalmente l'imprecisione aumenta se ci si avvicina agli estremi della funzione dato che il rapporto tra la variazione della variabile normalizzata e quella non normalizzata decresce.

esempio 1 di normalizzazione per proc. non stazionario:

$$x_normaliz = 1/(1+\exp(-x))$$

dove $-\infty < x < +\infty$ e $0 < x_normaliz < 1$

esempio 2 di normalizzazione per proc. non stazionario:

$$x_normaliz = \log_n(x)/k$$

dove $0 < x < +\infty$ e

k =costante valutata in base alla probabilità che la variabile oltrepassi un certo valore.

Naturalmente con una normalizzazione di tipo 1 abbiamo la certezza che tutti i valori normalizzati cadano entro il range 0.0/1.0, mentre con il secondo metodo dobbiamo fare una ipotesi restrittiva sui valori massimi che la variabile potrà assumere. In ogni caso, l'utilizzo di funzioni di questo tipo è più utile per effettuare delle "ridistribuzioni dei dati" (non tratteremo qui questo argomento). Per aggirare il problema dei processi non stazionari si possono considerare serie storiche di derivate anziché di valori della variabile: in questo modo siamo sicuri che i dati in ingresso sono compresi tra due valori definiti. Analogamente al caso di serie storiche di valori della variabile, gli intervalli di campionamento dei dati per l'addestramento dovranno essere tanto più piccoli quanto più alta è la frequenza di variazioni significative della variabile.

PREVISIONE MULTIVARIATA

Abbiamo precedentemente analizzato le problematiche relative alla previsione univariata, cioè quella in cui si cerca di stimare valori futuri di una variabile in base ai valori assunti precedentemente. Il caso della previsione multivariata non è concettualmente molto più complesso in quanto in esso esistono soltanto delle variabili che sono tra loro correlate per cui la previsione di valori assunti da una o più variabili dipende dall'insieme delle serie storiche di ognuna delle variabili di ingresso. Il contesto di analisi della rete neurale diventa molto più ampio, dato che l'output dipende da

$n_{\text{informazioni}} = n_{\text{variabili_input}} * n_{\text{dati_serie_storica}}$.

Ciò che rende differente e più interessante il problema è che i fenomeni complessi che si possono studiare analizzando l'evoluzione delle variabili appartenenti sono in genere sistemi nei quali le uscite non sono direttamente influenzate solo dalle variazioni degli ingressi ma sono funzione dello stato del sistema associato agli eventi negli ingressi. In realtà dobbiamo considerare tre tipi di variabili:

- 1) di ingresso
- 2) di stato del sistema
- 3) di uscita.

Le variabili di stato sono funzione di se stesse e degli input e le variabili di uscita sono funzione delle variabili di stato e degli inputs:

$dS(t)/dt = f(S,i)$ dove S =stato del sistema i =input

$u(t) = f''(S,i)$ dove u =output

f e f'' sono rispettivamente la funzione di evoluzione dello stato

rispetto agli ingressi e la funzione di evoluzione dell' uscita del sistema

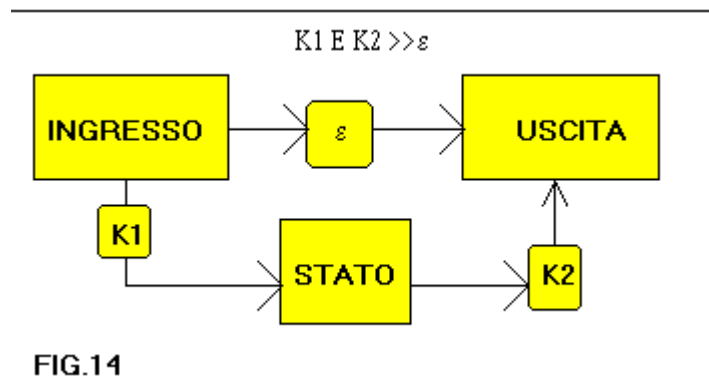
Nella realtà è abbastanza lecito semplificare nel seguente modo:

$$u(t) = f''(S)$$

perché le uscite sono, nella maggior parte dei casi, scarsamente influenzate direttamente dalle variazioni degli ingressi ma sono significativamente influenzate dalle variazioni di stato del sistema [\(fig.14\)](#). Considerando un sistema in cui gli ingressi variano lentamente, si possono calcolare le uscite all'istante t sulla base dello stato e degli ingressi all'istante $t-1$:

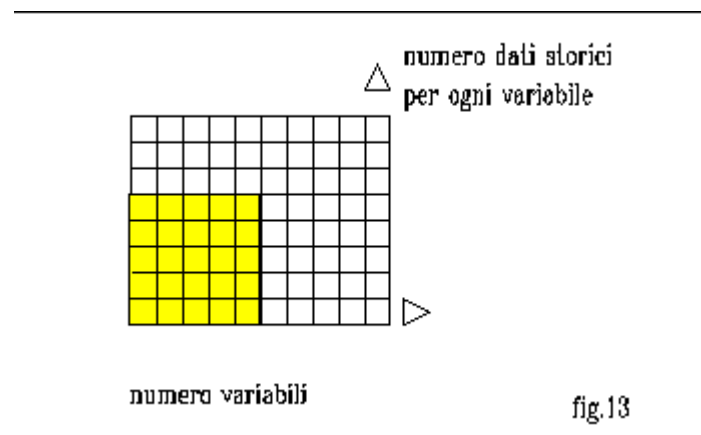
$$S(t) = f(S(t-1), i(t-1))$$

$$u(t) = f''(S(t-1)) = f''(f(S(t-1), i(t-1)))$$



Considerando un sistema in cui gli ingressi variano molto velocemente bisogna tenere conto dell'errore che deriva dal trascurare l'evoluzione dello stato dovuta alla variazione degli ingressi dall'istante precedente a quello relativo alla previsione. Per fare questo è bene inserire come variabili di input della rete,

non soltanto variabili di stato del sistema ma, anche variabili di input del sistema in modo che la rete possa fare previsioni sulle evoluzioni degli stessi. In ogni caso la previsione multivariata risulta molto più precisa e affidabile della previsione univariata, dato che si basa su un numero maggiore di informazioni. Per questo motivo è possibile utilizzare serie storiche ridotte rispetto alla previsione univariata: possiamo parlare di quantità di informazione verticale (numero variabili analizzate) e orizzontale (estensione delle serie storiche) e considerare come misura della completezza dell'informazione l'area del rettangolo di [fig.13](#).



CONCLUSIONI

Per uscire un po' dalla teoria vi fornisco un elenco di applicazioni pratiche realizzate con reti neurali in maggioranza di tipo `error_back_propagation`:

- sistema di guida autonoma di automobili che può guidare alla velocità di circa 5 Km/h nei viali della Carnegie Mellon University, avendo come input l'immagine della strada. E' una rete neurale `error_back_propagation` addestrata con 1200 immagini in 40 epoche su un supercomputer Warp. Il tempo di addestramento

è stato di 30 minuti mentre la sua esecuzione richiede circa 200 msec su Sun-3.

- classificatore di segnali radar che raggiunge prestazioni che con tecniche Bayesiane non sono mai state raggiunte.

- lettore di testi che può pronunciare parole mai viste prima con una accuratezza del 90%: si tratta di una rete error_back_propagation con 309 unità e 18629 connessioni implementata in linguaggio c su VAX_780.

- riconoscitore di oggetti sottomarini attraverso sonar realizzato da Bendix Aerospace

- un sistema di scrittura automatico su dettatura in lingua finlandese o giapponese che ha una accuratezza compresa tra l'80% e il 97%. L'hardware prevede un filtro passa_basso e un convertitore analogico_digitale, mentre la rete neurale è di tipo autoorganizzante realizzata su pc AT con coprocessore TMS-32010.

- sistema di supporto alla decisione per la concessione prestiti realizzato con rete error_back_propagation con 100 input e 1 output (prestito concesso o no). L'apprendimento è stato effettuato con 270.000 casi di prestiti in 6 mesi.

- la rete Neuro-07 sviluppata da NEC riconosce al 90% caratteri stampati e la rete sviluppata dalla Neuristique riconosce il 96% dei caratteri numerici scritti a mano. La sua architettura prevede 256 input (16*16 pixels), uno strato nascosto di 128 neuroni ed un secondo di 16, uno strato di output di 10 neuroni (cifre da 0 a 9).

- la Hughes Research Lab ha realizzato una rete che può riconoscere visi umani anche con disturbi

(barba/occhiali/invecchiamento).

- è stata realizzata una rete avente come input una immagine di 20×20 pixel con 16 livelli di grigio proveniente da proiezioni tomografiche di polmoni che restituisce in output l'immagine depurata del rumore (che è tanto più ampio quanto minore è il numero delle proiezioni). In pratica la rete fa una interpolazione di immagini discontinue sulla base di un addestramento di 60 immagini di polmoni con validation set di 120 immagini.

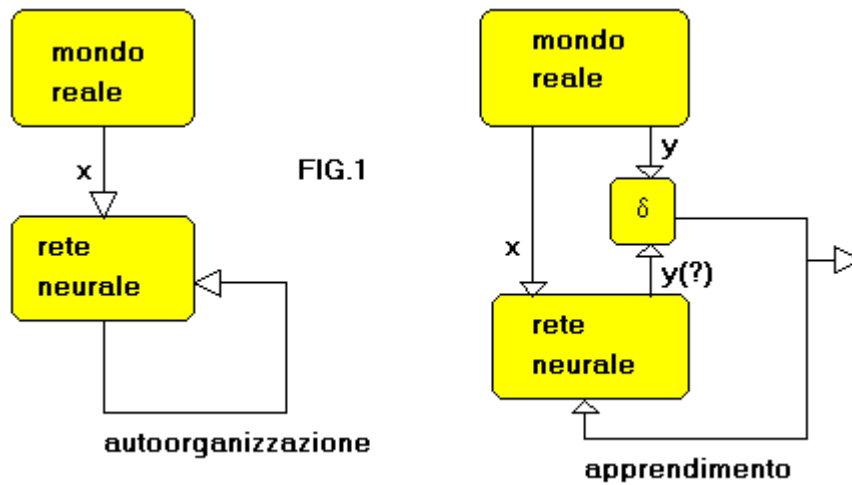
- la PNN (Probabilistic Neural Network) sviluppata da Lockheed interpreta correttamente il 93% dei segnali sonar con un addestramento di 1700 esempi di segnali riflessi da sommergibili e di 4300 esempi di echi riflessi da navi o da movimenti dell'acqua in superficie.

- una rete neurale è stata applicata in robotica per risolvere un problema di "cinematica inversa" del tipo: un braccio meccanico con due snodi deve raggiungere un target di coordinate R_t e α_t (R =raggio α =angolo) partendo da una situazione degli snodi α_{1_b} e α_{2_b} . Tale problema per essere risolto richiede il calcolo di funzioni trascendenti (trigonometriche inverse) ma con una rete neurale di 4 input ($R_t, \alpha_t, \alpha_{1_b}, \alpha_{2_b}$), due strati intermedi rispettivamente di 6 e 4 neuroni e due neuroni di output ($\alpha_{1_b}, \alpha_{2_b}$) si ottiene un risultato migliore con poco sforzo. Tale rete che comanda il manipolatore INTELLEDEX 605T è simulata su un PC_AT ed è stata addestrata con 64 esempi contenenti posizioni relative diverse tra target e braccio. Ciò che ha reso ancora più interessante l'applicazione è stato il fatto che la rete è risultata in grado di correggere disturbi al moto di avvicinamento del braccio e seguire un target in movimento ([fig 15](#)).

RETI NEURALI AUTOORGANIZZANTI

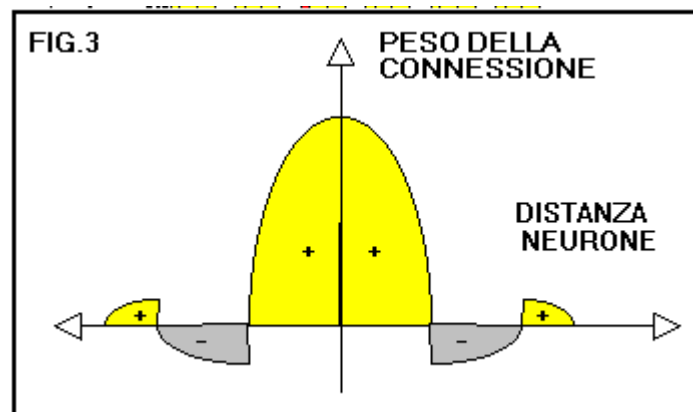
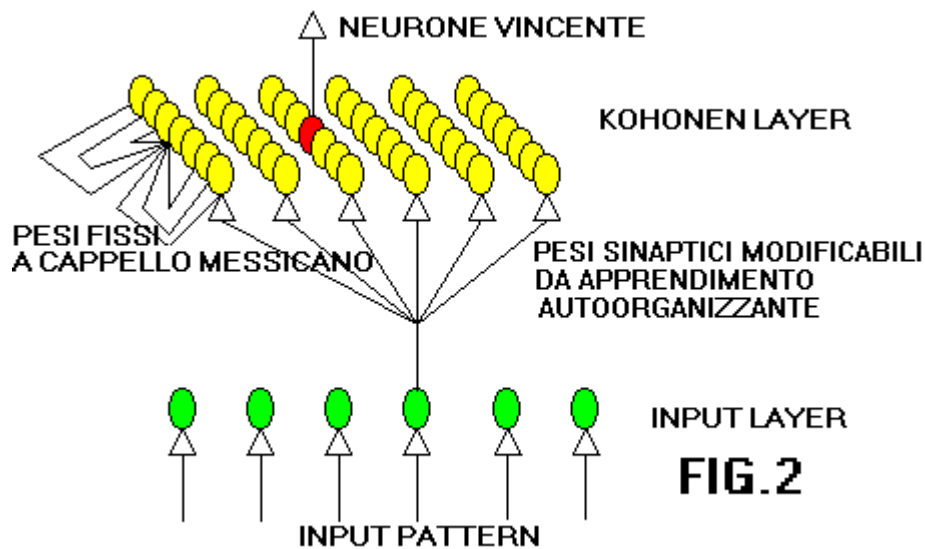
INTRODUZIONE

Le reti neurali esaminate nei capitoli precedenti erano una memoria associativa e una Error Back Propagation, cioè due paradigmi completamente differenti per quanto riguarda utilizzo e principi di funzionamento ma avevano, effettivamente, qualcosa in comune: entrambe utilizzavano un tipo di apprendimento "supervised". Per apprendimento supervised o supervisionato si intende un tipo di addestramento della rete tramite coppie input/output_desiderato che, chiaramente, sono esempi conosciuti di soluzione del problema in particolari punti nello spazio delle variabili del problema stesso. In taluni casi non esiste la possibilità di avere serie di dati relativi alla soluzione del problema ed esistono invece dati da analizzare e capire senza avere a priori un "filo guida" di informazioni specifiche su di essi. Un tipico problema di questo tipo è quello di cercare classi di dati aventi caratteristiche similari(per cui associabili) all' interno di un gruppo disordinato di dati. Questo problema può essere affrontato con una rete neurale autoorganizzante, cioè in grado di interagire con i dati, addestrando se stessa senza un supervisore che fornisca soluzioni in punti specifici nello spazio delle variabili. La [fig.1](#) è una rappresentazione schematica degli apprendimenti tipo "supervisionato" e "non supervisionato"(reti autoorganizzanti).



APPLICAZIONI

Naturalmente le reti neurali autoorganizzanti sono adatte alla soluzione di problemi differenti rispetto alle reti con apprendimento supervisionato. Il principale utilizzo di tali reti è precisamente quello di analisi di dati al fine di riscontrarne gruppi aventi similitudini (preprocessing di dati) o classificazione di forme come riconoscimento immagini o segnali. Ad esempio una rete di tale tipo può essere impiegata in riconoscimento scrittura o riconoscimento di segnali radar. Inoltre sono state realizzate molte applicazioni per ottimizzazione di processi industriali, talvolta interpretando i risultati, non solo in base alla mappa dei neuroni dello strato output, ma anche in base alla mappa dei vettori dei pesi della rete. Una applicazione interessante di questo tipo è stata realizzata per la ottimizzazione della produzione di circuiti elettronici integrati: è stata studiata in Germania da K.Goser e K.Marks e si tratta di una rete di Kohonen con una matrice di 2500 neuroni con 10 inputs dei quali uno rappresenta la qualità del circuito, mentre gli altri nove rappresentano parametri di fabbricazione.



LE RETI DI KOHONEN

Il più conosciuto ed applicato modello di rete neurale autoorganizzante prende il nome dal suo inventore (Kohonen, T) ed è costituito da una rete a due strati, dei quali uno è di input e l'altro (di output) viene comunemente chiamato strato di Kohonen. I neuroni dei due strati sono completamente connessi tra loro, mentre i neuroni dello strato di output sono connessi, ciascuno,

con un "vicinato " di neuroni secondo un sistema di inibizione laterale definito a "cappello messicano". I pesi dei collegamenti intrastrato dello strato di output o strato di Kohonen non sono soggetti ad apprendimento ma sono fissi e positivi nella periferia adiacente ad ogni neurone. Le figure [fig.2](#) e [fig.3](#) rappresentano una rete di Kohonen e il collegamento a cappello messicano. Nello strato di output un neurone soltanto deve risultare "vincente" (con il massimo valore di attivazione) per ogni input pattern che viene fornito alla rete. In pratica il neurone vincente identifica una classe a cui l'input appartiene. Il collegamento a cappello messicano, nella versione originale di questo tipo di rete, tende a favorire il formarsi di "bolle di attivazione" che identificano inputs simili. Ogni neurone del Kohonen layer riceve uno stimolo che è pari alla sommatoria degli inputs moltiplicati per il rispettivo peso sinaptico:

$$A(j)=S(k)w(j,k)*x(k)$$

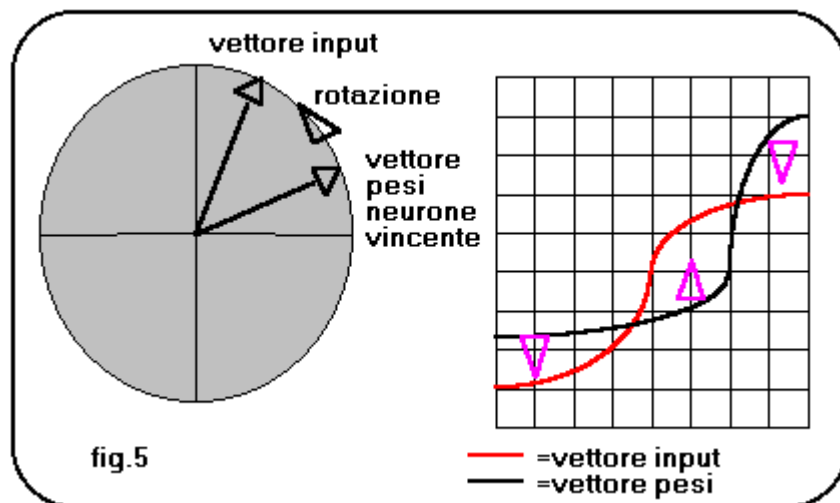
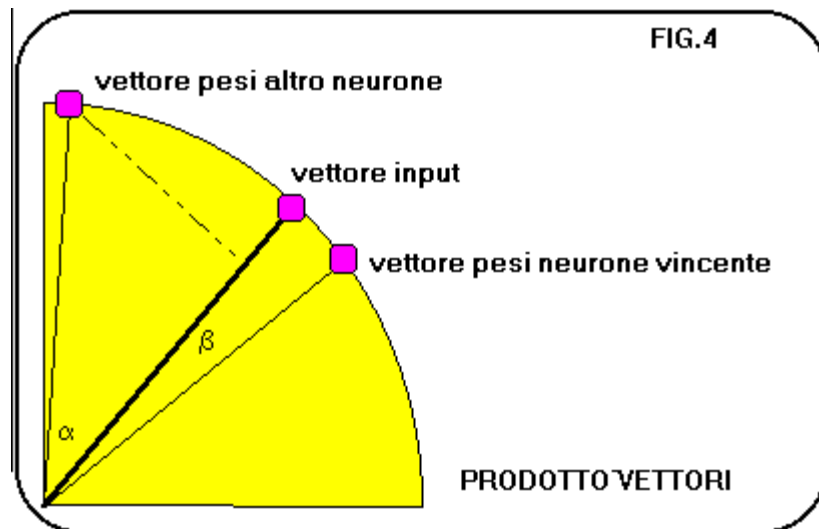
Tra tutti i neuroni di output viene scelto quello con valore di attivazione maggiore che assume quindi il valore 1, mentre tutti gli altri assumono il valore 0 secondo la tecnica "WTA"(Winner Takes All). Lo scopo di una rete di Kohonen è quello di avere, per inputs simili, neuroni vincenti vicini, così che ogni bolla di attivazione rappresenta una classe di inputs aventi caratteristiche somiglianti.

APPRENDIMENTO AUTOORGANIZZANTE

Il comportamento sopra descritto si raggiunge dopo la presentazione di molti inputs per un certo numero di volte alla rete, modificando, per ogni presentazione di input solo i pesi che collegano il neurone vincente(Kohonen layer) con i neuroni dello strato di input secondo la formula:

$$W(k, j)_{\text{new}} = W(k, j)_{\text{old}} + \epsilon * (X(k) - W(k, j)_{\text{old}})$$

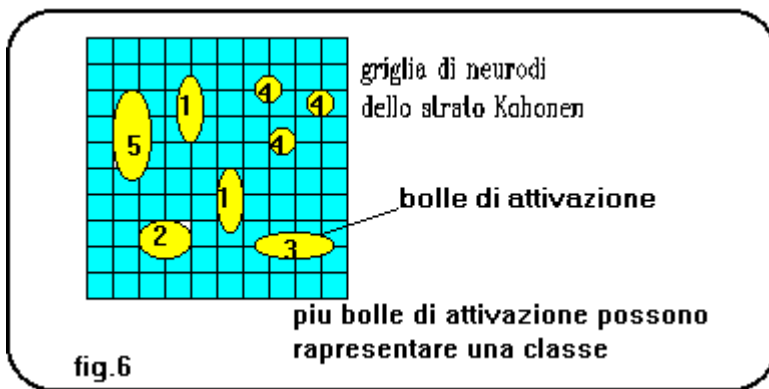
dove $W(k, j)$ = peso sinaptico del collegamento tra
input k e neurone vincente
 $X(k)$ = input k-esimo dell'input pattern
 ϵ = costante di apprendimento (nel range 0.1/1)



In pratica i pesi da aggiornare per ogni presentazione di input sono tanti quanti sono i neuroni di input. In taluni casi si aggiornano con lo stesso criterio, magari usando valori di epsilon decrescenti, anche i pesi dei neuroni di un opportuno vicinato (dimensionalmente sovrapponibile alla parte positiva del cappello

messicano). Leggendo con attenzione la formula di apprendimento, ci si accorge che il suo ruolo è quello di ruotare il vettore dei pesi sinaptici verso il vettore di input. In questo modo il neurone vincente viene ancor più sensibilizzato al riconoscimento della forma dell'input presentato ([fig.4](#) e [fig.5](#)). Questo tipo di reti è stato utilizzato con profitto in applicazioni di riconoscimento immagini e riconoscimento segnali. Quando viene impiegata una rete autoorganizzante, la matrice di neuroni (o neurodi*) di output si presenta con un insieme di bolle di attivazione che corrispondono alle classi in cui la rete ha suddiviso gli inputs per similitudine. Se noi addestriamo una rete con dati di cui conosciamo a priori la appartenenza a specifiche classi, potremo associare ciascuna delle bolle di attivazione a ognuna di queste classi ([fig.6](#)). Il vero fascino dei paradigmi non supervisionati è, però, quello di estrarre informazioni di similitudine tra input patterns (esempio un segnale sonar o radar) molto complessi, lavorando su grandi quantità di dati misurati nel mondo reale. Questo tipo di rete evidenzia molto bene il fatto che le reti neurali in genere sono processi tipo "data intensive" (con molti più dati che calcoli), anziché "number crunching" (con molti calcoli su pochi dati). Come si può vedere in [fig.6](#), più bolle di attivazione possono rappresentare una classe di inputs: questo può essere dovuto alla eterogeneità di una classe (che deve essere ovviamente conosciuta) o anche alla sua estensione nello spazio delle forme possibili (in questo caso due pattern agli estremi della classe possono dare origine a bolle di attivazione separate).

*NEURODO: questo termine che deriva dalla composizione di "neurone" e "nodo" è stato proposto da Maureen Caudill, una autorevole esperta statunitense di reti neurali, con lo scopo di distinguere il neurone biologico (la cui funzione di trasferimento è notevolmente più complessa) dal neurone artificiale. Da ora in poi utilizzeremo questo termine che è probabilmente più corretto.



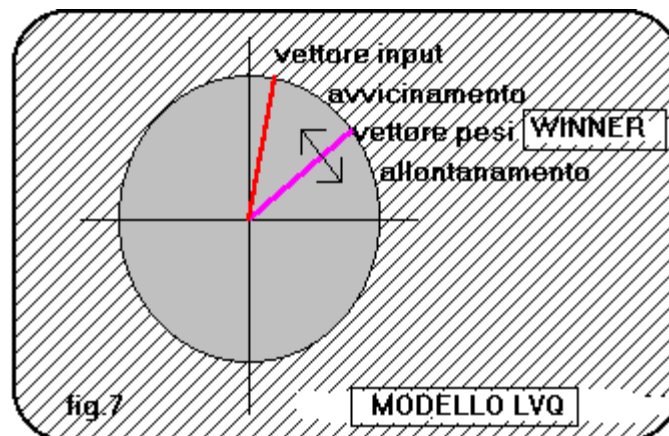
CLASSIFICATORI SUPERVISIONATI

Da questo modello autoorganizzante sono stati derivati classificatori supervisionati sicuramente più adatti ad un lavoro di riconoscimento/classificazione su problemi che consentono di avere una casistica di associazioni input/classe. Per addestrare un modello supervisionato occorre fornire alla rete, non la semplice sequenza degli input pattern ma la sequenza delle associazioni input/classe desiderata. In problemi "giocattolo" una classe può essere rappresentata anche da un solo neurone dello strato di output ma in problemi reali ogni classe sarà rappresentata da un numero piuttosto elevato di neuroni sulla griglia (anche diverse centinaia). Il numero di neuroni necessari a rappresentare una classe è proporzionale alla estensione della classe stessa nello spazio delle forme degli input pattern. Nello scegliere i neuroni di uscita atti alla rappresentazione di una classe si deve tenere conto della estensione di essa, fornendo più neuroni alle classi più ampie. La formula di apprendimento della rete nella versione supervisionata si sdoppia, ottenendo effetti opposti a seconda che il neurone risultato vincente appartenga alla classe desiderata o meno. In pratica il vettore dei pesi del neurone vincente viene avvicinato al vettore input se rappresenta la classe corretta, altrimenti viene allontanato:

1) $W(k,j)_{new} = W(k,j)_{old} + e * (X(k) - W(k,j)_{old})$
 se neurodo vincente appartiene alla classe corretta

2) $W(k,j)_{new} = W(k,j)_{old} - e * (X(k) - W(k,j)_{old})$
 se neurodo vincente non appartiene alla classe corretta

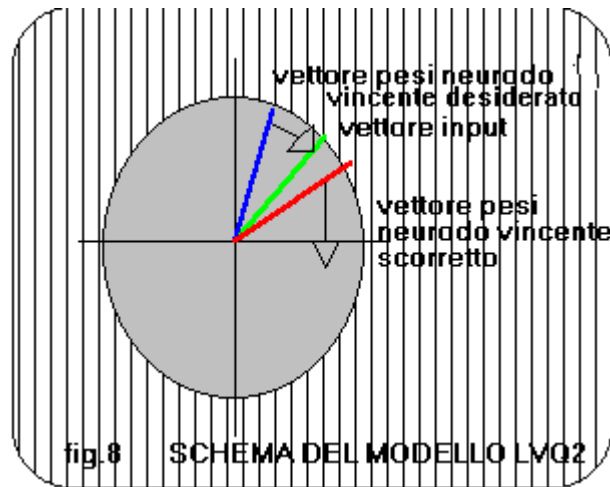
Questo modello di rete viene denominato LVQ che sta per Learning Vector Quantization e il procedimento di aggiornamento dei pesi nel neurodo vincente viene schematizzato in [fig.7](#).



All'idea di allontanare il vettore dei pesi del neurodo scorretto dal vettore input si è aggiunta anche quella di avvicinare il vettore dei pesi del neurodo desiderato come vincente. In questo modo la seconda formula (quella di allontanamento) vista prima deve andare in coppia con la seguente:

$W(k,j)_{new} = W(k,j)_{old} + e * (X(k) - W(k,j)_{old})$
 con $neurodo(j) = neurodo\ vincente\ desiderato$

In [fig.8](#) viene rappresentato lo schema del modello sopra descritto e comunemente chiamato LVQ2.



NORMALIZZAZIONE DELL'INPUT

I dati che vengono forniti in input ad una rete di Kohonen devono essere compresi tra 0 e 1, quindi è necessario fare un preprocessing dei dati al fine di normalizzarli. Questa operazione si può facilmente ottenere dividendo ogni elemento del vettore per la lunghezza del vettore stesso. La normalizzazione o comunque il preprocessing dei dati può essere effettuato in modi differenti a seconda del tipo di problema: in alcuni casi è molto importante riconoscere la forma dell' input pattern (es: riconoscimento segnali/immagini), mentre in altri casi è importante anche mantenere intatta una relazione dimensionale tra i pattern di input (es: distinguere i punti definiti da coordinate x/y dentro certe aree). Anche i pesi della rete devono essere inizializzati con pesi random compresi tra 0 e 1: durante la fase di apprendimento è possibile che i valori dei pesi vadano fuori dal range di normalizzazione ma, normalmente, tendono a rientrare. Comunque è possibile effettuare una normalizzazione di tutti i vettori dei pesi dopo ogni aggiustamento dei pesi del vettore del neurodo vincente (lvq/lvq2) e del neurodo desiderato (lvq2).

REALIZZAZIONE SOFTWARE

Non è particolarmente difficile la realizzazione di una rete autoorganizzante "giocattolo". Bisogna utilizzare due vettori di m e n float per input e output e un vettore di $n*m$ float per i pesi tra input e output. All' inizio i pesi devono essere inizializzati in modo random con valori compresi tra 0 e 1. Una funzione di apprendimento `learn()` deve leggere, da un file contenente il training set, un pattern e inserirlo normalizzato nel vettore input. Poi deve chiamare una funzione di esecuzione della rete `exec()` che calcola l'output per ogni neurone effettuando una moltiplicazione dei vettori input con i vettori pesi. Una altra funzione effettuerà uno scanning dei valori di attivazione dei neurodi dello strato output scegliendo il maggiore e passerà alla funzione `learn()` il numero identificativo(indice del vettore) del neurodo vincente. A questo punto la funzione `learn()` può applicare la regola di aggiornamento dei pesi su quel neurodo. Questa operazione completa il ciclo su un pattern e il passo successivo è la lettura di un altro pattern. Tutti i pattern del training set devono essere appresi in questo modo dalla rete per un numero consistente di iterazioni. Normalmente il collegamento a cappello messicano dei pesi sullo strato Kohonen viene trascurato anche perché dovrebbe essere dimensionato in modo opportuno a seconda del problema. Più importante può risultare l'aggiornamento dei pesi dei neurodi vicini al neurodo vincente, tuttavia le variazioni che si possono attuare su questo modello sono molteplici e consentono di avere risultati migliori per specifici problemi: ad esempio epsilon decrescente e vicinato del neurone vincente decrescente sono accorgimenti che possono favorire la convergenza della rete. Nel listato 1 sono rappresentate in metalinguaggio le funzioni principali semplificate di una simulazione di rete neurale Kohonen autoorganizzante.

LISTATO 1

LEARN:

```

FOR1 K=1 TO NUMERO ITERAZIONI
  FOR2 J=1 TO NUMERO PATTERNS
    LETTURA PATTERN(J)
    EXEC
    SCANNING_OUTPUT-->NEURODO WINNER
    UPDATE_WEIGHTS(NEURODO_WINNER)
    NORMALIZZ_WEIGHTS (NON INDISPENSABILE)
  ENDFOR2
ENDFOR1

```

EXEC:

```

FOR1 J=1 TO NUMERO NEURODI MATRICE OUTPUT
  FOR2 K=1 TO DIMENSIONE INPUT PATTERN
    ATTIV_NEURODO(J)=ATTIV_NEURODO(J)+W(J,K)*X(K)
  ENDFOR2
ENDFOR1

```

SCANNING_OUTPUT:

```

ATT_MAX=0
FOR J=1 TO NUMERO NEURODI MATRICE OUTPUT
  IF (ATTIV_NEURODO(J)>ATT_MAX)
    ATT_MAX=ATTIV_NEURODO(J)
    NEURODO_WINNER=J
  ENDIF
ENDFOR

```

UPDATE_WEIGHTS(J):

```

FOR K=1 TO DIMENSIONE PATTERN
  W(J,K)NEW=W(J,K)OLD+EPSILON*(X(K)-W(J,K)OLD)
ENDFOR

```

CONCLUSIONI

Prima di riuscire a far funzionare una simulazione di rete neurale autoorganizzante, è normale incontrare problemi sulla modalità di normalizzazione dei dati e avere perplessità sull'impiego stesso di essa per risolvere il problema che si sta analizzando. Il migliore modo per capire e conoscere queste reti è quello di realizzare delle piccole simulazioni software e cercare di capire quali tipi di problemi sono risolvibili apportando modifiche al tipo di preprocessing dei dati. Non abbiamo esaminato la realizzazione

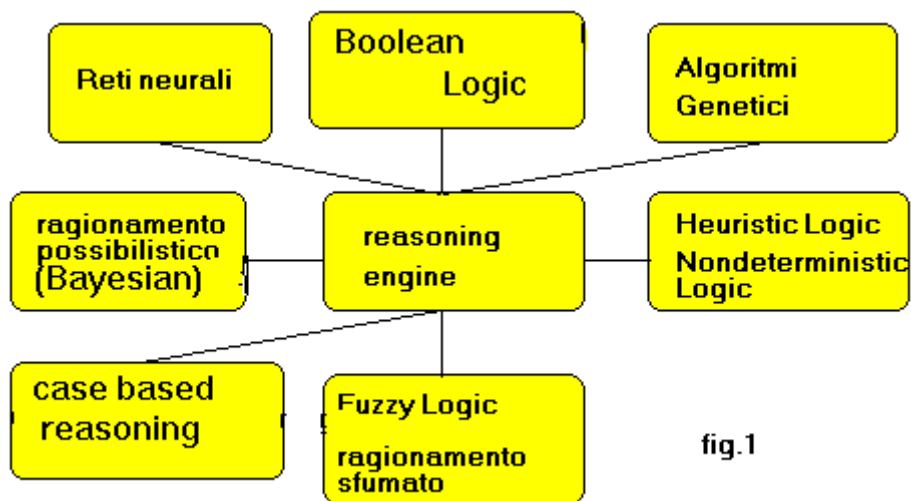
software di LVQ e LVQ2 ma credo che chi riesca a realizzare una Kohonen non dovrebbe avere problemi nel trasformarla in questi modelli (nb: tenere presente che per applicazioni reali ogni classe deve essere mappata su più neuroni dello strato output e che il numero di essi deve essere proporzionale alle dimensioni della classe stessa).

FUZZY LOGIC

LA TEORIA DEL RAGIONAMENTO SFUMATO

DOVE SI APPLICA LA FUZZY LOGIC

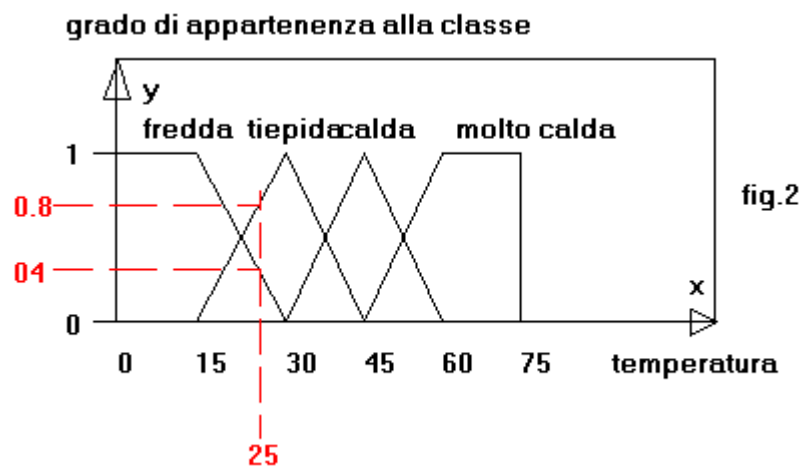
Innanzitutto dobbiamo precisare che questa tecnica fa parte dei sistemi decisionali basati su regole del tipo "if ...then...else", o più semplicemente di quei sistemi basati sulla conoscenza (Knowledge Based Processing) come i cosiddetti Sistemi Esperti. La base di conoscenza di un sistema esperto è come una specie di database che anziché essere composto di dati è composto di relazioni tra i dati intese come associazioni di conseguenze determinate da particolari condizioni di ipotesi: tutto ciò viene chiaramente espresso nelle regole tipo "if then else". Non voglio in questo capitolo approfondire problematiche relative ai sistemi esperti ma concludo ancora dicendo che il programma che "aziona le regole" contenute nel "knowledge base" viene spesso chiamato motore inferenziale o reasoning engine e può essere basato su diverse tecniche di supporto alla decisione. In [fig.1](#) potete vedere un tipico protocollo di ragionamento per un sistema esperto ibrido con la estensione delle ultime tecnologie come reti neurali e algoritmi genetici.



Come potrete immaginare, un sistema esperto puo essere basato anche solamente su regole fuzzy e quindi è il caso di capire cosa significa ragionamento fuzzy o sfumato. Se prendiamo in esame una regola del tipo if(acqua bolle) then (butto la pasta), non vi sono sicuramente incertezze a decidere quando buttare la pasta dato che (acqua bolle) è sicuramente vero o sicuramente falso. Se prendiamo in esame invece la regola if(acqua è calda) then (abbassa la fiamma) ci troviamo di fronte a dati incerti: che cosa significa calda? E quanto devo abbassare la fiamma? Supponendo di poter misurare la temperatura con un termometro, come faremo a decidere se quel valore rientrerà nella definizione "calda"? Dovremo iniziare a definire un range di valori in cui la temperatura può variare e dei subranges che rappresentino definizioni come "calda" o "molto calda". Appare quindi evidente che non sono i dati input a essere imprecisi ma bensì le relazioni che esistono tra input e output: nei problemi trattati con fuzzy logic infatti non esiste un facile modello matematico che collega input e output . In pratica con un motore inferenziale fuzzy si puo empiricamente sintonizzare una funzione matematica che vincola input e output, tramite regole tipo " if ...then ...".

FUZZIFICAZIONE DELL INPUT

Il ragionamento cosiddetto sfumato dei sistemi fuzzy è dovuto al fatto che un'ipotesi non è mai completamente vera né completamente falsa ma ha un suo "grado di verità" che inciderà sulla forza con cui verrà eseguita la regola e quindi applicata la conseguenza. Il grado di verità dell'ipotesi è in realtà il grado di appartenenza del valore di input a quel determinato range (es: calda) e ciò è dovuto al fatto che questi ranges non sono nettamente separati ma si sovrappongono con funzioni di appartenenza normalmente di tipo triangolare/trapezoidale, per cui un valore di una variabile di input potrebbe appartenere a due ranges con due differenti gradi di credibilità ([fig.2](#)).



Il valore 25 gradi della temperatura appartiene alla classe tiepida con un degree of membership (grado di appartenenza) pari a 0.8 e alla classe fredda con un grado di appartenenza uguale 0.4. Come vedete, definendo delle classi triangolari o trapezoidali (le più usate) si hanno dei gradi di appartenenza che diminuiscono verso gli estremi delle classi costruendo così un inizio di quello che è un ragionamento sfumato. Quello che rappresenta la [fig.2](#) è la operazione che viene comunemente definita "FUZZIFICAZIONE

DELL INPUT" , cioè trasformazione del dato preciso in dato tipo fuzzy:

dato preciso temp=25 gradi centigradi

dato fuzzy temp =tiepida con credibilità 0.8

temp = fredda con credibilità 0.4

Per realizzare matematicamente il processo di fuzzificazione bisogna effettuare alcuni semplici calcoli sulle funzioni triangolari/trapezoidali . Il primo passo da fare nel processo di fuzzificazione è uno scanning dei valori estremi di ogni classe per cui quando si ha che

$\text{limite_inf_class} < x < \text{limite_sup_class}$

allora x appartiene alla classe e bisogna calcolare il grado di appartenenza. Bisogna definire uno "slope" cioè una pendenza , sui lati dei triangoli o dei trapezi che definiscono le classi, che sia fisso per ogni classe e che ci consenta di calcolare il grado di appartenenza di un input x alla classe y. A questo punto bisogna vedere se il valore x cade nella parte bassa della classe (pendenza positiva) o nella parte alta(pendenza negativa): per far questo è sufficiente verificare se

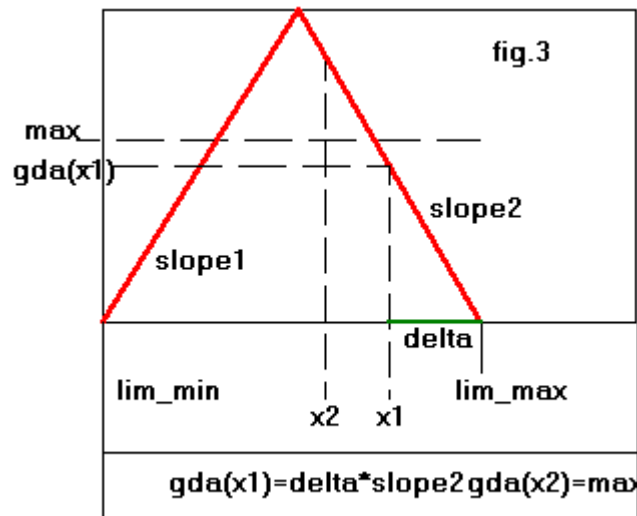
$x > \text{oppure } x < \text{di } (\text{lim_sup}-\text{lim_inf})/2$

che rappresenta il centro della classe. Nel primo caso si effettuerà il calcolo con lo slope positivo e nel secondo con lo slope negativo([fig.3](#)):

1) $\text{gda} = (x-\text{lim_inf})*\text{slope}$ (se $\text{gda} > \text{max}$ allora $\text{gda}=\text{max}$)

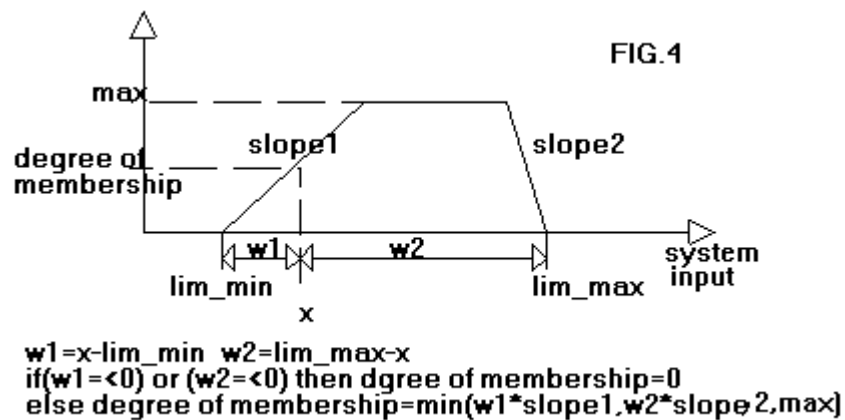
2) $\text{gda} = (\text{lim_sup}-x)*\text{slope}$ (se $\text{gda} > \text{max}$ allora $\text{gda}=\text{max}$)

(gda=grado di appartenenza o "degree of membership")



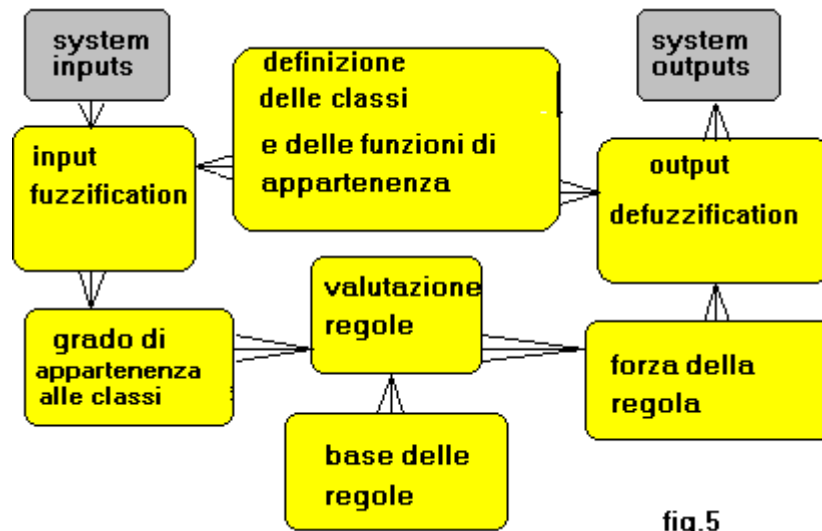
Bisogna precisare che questo è solamente un possibile approccio alla soluzione del problema di fuzzificazione, infatti è possibile implementare funzioni di qualunque tipo per definire il grado di appartenenza di un input ad una classe; si tratta solamente di calcolare il valore di una funzione in un particolare punto definito dal valore dell' input. Personalmente ho sviluppato interfacce fuzzy per reti neurali error_back_propagation con funzioni circolari (seno ,coseno ecc) ma ciò che va tenuto presente è che, normalmente, il range dei valori di appartenenza o "degree of membership" deve essere normalizzato tra il valore 0 e il valore 1. Ciò previene evidenti problemi tra variabili del sistema che hanno range di valori completamente differenti e sarebbe comunque auspicabile una soluzione che preveda la normalizzazione delle variabili di input su scala 0-1. Rimanendo comunque nelle classiche funzioni trapezoidali/triangolari si possono effettuare moltissime varianti. Ad esempio si può fare in modo che la pendenza (slope) dei triangoli sia differenziabile per ogni classe e all' interno di ogni classe quella di salita da quella di discesa. Si può anche effettuare il calcolo di fuzzificazione in modo differente senza calcolare il centro della classe ([fig.4](#)). Risulta evidente che quanto più complesse sono le forme delle funzioni scelte per definire le classi, tanto maggiore è la quantità di informazione che

deve essere memorizzata per definire una classe e la quantità di calcoli da effettuare per ottenere un valore di "membership degree". Ad esempio la semplice scelta di differenziare le pendenze dei lati dei triangoli o dei trapezi obbliga a memorizzare tale informazione per ogni classe presente.



LA VALUTAZIONE DELLE REGOLE

Un a volta che siamo in possesso di dati fuzzy provenienti dal processo di fuzzificazione dobbiamo inserire nel motore decisionale delle regole che ci diano degli output fuzzy particolari per particolari situazioni (fuzzy) degli input. Una di queste regole può avere la forma: if (input n appartiene a classe k) then output m appartiene a classe j con forza pari al grado di appartenenza di n a k. Spesso nella applicazione delle regole alcune di esse portano alla medesima conseguenza con livelli di forza differenti: in questi casi è pratica comune scegliere il valore maggiore(list.1). Avere dei dati fuzzy in uscita ci servirebbe a ben poco però, perciò si rende necessario trasformare i dati che derivano dalla valutazione delle regole in dati numerici reali: questo processo è il processo opposto alla fuzzificazione dell' input e infatti si chiama defuzzificazione dell' output. In [fig.5](#) vedete rappresentato lo schema di un sistema fuzzy completo.



LISTATO 1

REGOLA 1: IF (A&&B) THEN X E Y

REGOLA 2: IF (C&&D) THEN X E Z

FORZA DELLA REGOLA 1= MIN(A,B)

FORZA DELLA REGOLA 2= MIN(C,D)

Y= FORZA DELLA REGOLA 1

Z= FORZA DELLA REGOLA 2

X=MAX(FORZA DELLA REGOLA 1,FORZA DELLA REGOLA 2) = MAX(MIN(A,B),MIN(C,D))

DOVE A,B,C,D SONO IPOTESI DEL TIPO "DEGREE OF MEMBERSHIP" OVVERO GRADO DI APPARTENENZA DI UN INPUT DEL SISTEMA AD UNA SPECIFICA CLASSE

E

X,Y,Z SONO CONSEGUENZE INTESE COME GRADO DI APPARTENENZA DI UN OUTPUT AD UNA PARTICOLARE CLASSE

DEFUZZIFICAZIONE DELL OUTPUT

Una volta che abbiamo fuzzificato gli inputs e abbiamo fornito le informazioni in formato fuzzy al motore decisionale del nostro sistema (il programma che fa girare le regole) e che questo ci ha fornito in risposta degli output in formato fuzzy del tipo output 3 appartiene a classe2 con grado 0.6 , dobbiamo ora trovare un sistema che ci consenta di estrarre dati numerici precisi da applicare alle uscite. Questa operazione detta defuzzificazione dell' output deve risolvere il problema della conflittualità che nasce dal fatto che alcune regole possono avere generato conseguenze contrastanti tipo:

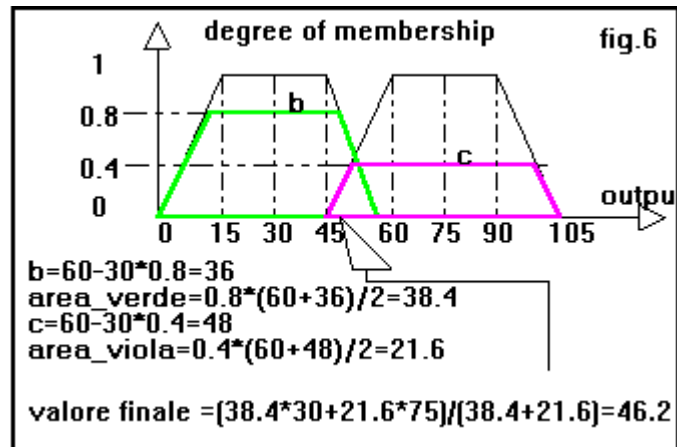
conseguenza1: out1 appartiene a classe2 con grado 0.6

conseguenza2: out1 appartiene a classe3 con grado 0.3

dove, cioè lo stesso output viene assegnato a classi differenti (normalmente adiacenti o vicine nel range della variabile). Viene comunemente utilizzato un metodo chiamato "center of gravity method" che consiste nel calcolare le aree "attive" dei trapezi o triangoli che definiscono le classi e, una volta posizionato il valore a metà tra i centri delle classi in conflitto farlo "attrarre" dai rispettivi centri_classe in modo proporzionale alle aree attive associate, come mostrato in [fig.6](#). Una possibile semplificazione può essere quella di prendere i centri di tutte le classi in conflitto e far attrarre il valore numerico finale da tutti i centri in modo proporzionale alla forza delle regole che hanno determinato le conseguenze di appartenenza ad ogni classe:

valore finale = $(f_1 * c_1 + f_2 * c_2 + f_3 * c_3 \dots + f_n * c_n) / (f_1 + f_2 + f_3 \dots + f_n)$

dove c_n = centro numerico della classe n f_n = grado di appartenenza (o forza della regola)



REALIZZAZIONE SOFTWARE

La realizzazione software di un sistema a regole basato su ragionamento sfumato non è particolarmente difficile in quanto non presenta particolari difficoltà tecniche. Naturalmente a seconda del tipo di applicazione che può avere esigenze di velocità esecutiva o meno, si potranno scegliere soluzioni differenti sia dal punto di vista strettamente teorico relativo alla scelta delle funzioni, sia dal punto di vista della programmazione. Quando si progetta un sistema fuzzy bisogna tenere presente alcuni passi fondamentali :

- 1) definizione delle classi per ogni variabile di ingresso
- 2) definizione delle classi per ogni variabile di uscita
- 3) definizione della forma che descrive il "degree of membership" e delle formule da usare per fuzzificazione e defuzzificazione (decidendo anche se utilizzare una normalizzazione dei dati)

- 4) definizione delle regole di base del motore decisionale
- 5) realizzazione del programma in modo che sia possibile aggiungere regole e modificare i limiti delle classi molto facilmente.
- 6) test del programma sul problema e sintonizzazione del sistema sul risultato desiderato tramite operazioni successive di affinamento tipo aggiunta/modifica di nuove regole e modifica dei limiti delle classi (talvolta anche ridefinizione del numero di classi presente sul range di una variabile)

Questo ultimo passo è senza dubbio il più difficile e il più lungo, tanto che ultimamente sono stati realizzati sistemi che permettono di sintonizzare un sistema fuzzy sulla soluzione del problema tramite reti neurali e sono in fase di studio anche sistemi che cercano di pilotare la evoluzione di un fuzzy_system tramite algoritmi genetici, cioè algoritmi che seguono i principi della teoria di Darwin. Dal punto di vista strettamente inerente la tecnica di programmazione credo che le soluzioni stilistiche possano essere ben differenziate anche a seconda dell'utilizzo o meno di un linguaggio object oriented come il c++: in ogni caso uno schema di base da seguire può essere quello seguente.

dati definiti in partenza:

float inp_class_min[n][m]
(vettore con limiti minimi di m classi su n input)

float inp_class_max[n][m]
(vettore con limiti massimi di m classi su n input)

float out_class_min[n][m]
(vettore con limiti minimi di m classi su n output)

float out_class_max[n][m]
(vettore con limiti massimi di m classi su n output)

programma:

```

per ogni input
{
    leggi il valore
    fuzzificazione:
        per ogni classe
            {
                if (min < valore < max) calcola grado
                appartenenza
            }
}

per ogni regola
{
    if(ipotesi input) imposta tesi output con grado di forza
    associato
}

defuzzificazione:
    per ogni output
    {
        calcola i conflitti con il "gravity center method"
    }

```

APPLICAZIONI

La maggior parte delle applicazioni di sistemi fuzzy riguarda il controllo di processi o meccanismi in genere e nell'industria americana e, giapponese soprattutto, se ne è fatto largo uso in questi ultimi tempi con risultati, a quanto pare, estremamente soddisfacenti: Minolta, Panasonic, Hitachi hanno fatto uso di fuzzy logic nella realizzazione di oggetti come macchine fotografiche o semplici aspirapolvere. In ogni caso la logica fuzzy può essere il nucleo di sistemi di supporto alla decisione o sistemi di controllo, in tasks dove l'estrema complessità non permette un approccio di tipo analitico: in questo senso la logica fuzzy è concorrente alla tecnologia delle reti neurali, ma sempre più spesso queste due tecnologie vengono utilizzate contemporaneamente in sistemi ibridi.

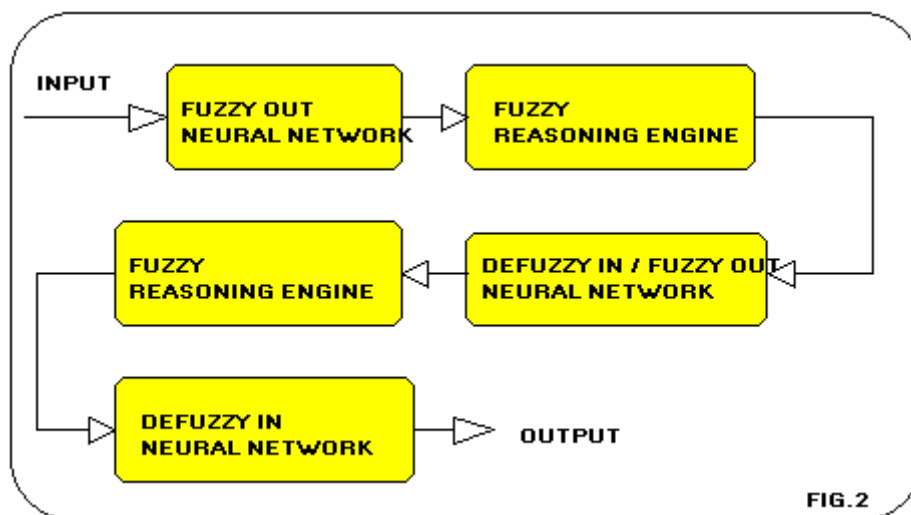
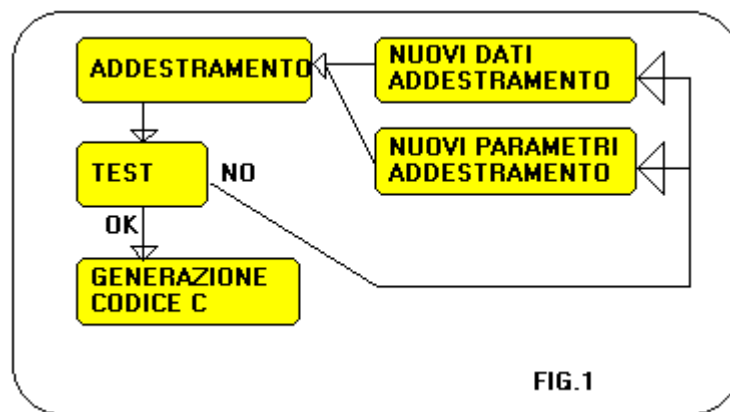
Neurfuzz 1.0

Generatore di codice c per reti neurali error back propagation e fuzzy systems

Introduzione

Neurfuzz è un tool composto da due programmi complementari, "Neuronx" e "Fuzzkern" che hanno, rispettivamente, le funzioni di generatore di reti neurali error back propagation e fuzzy systems. Neuronx è un programma in grado di addestrare una rete neurale ebp prelevando i dati relativi al training set da un file in formato ascii. Raggiunto il target error si può passare alla fase di test utilizzando l'apposito ambiente integrato. Successivamente, è possibile generare un file in codice c che costituisce la rete neurale addestrata e può comunicare con altri programmi tramite files, ma può anche essere integrato come funzione all' interno di altri programmi ([fig.1](#)). Fuzzkern permette di generare programmi in codice c che contengono il "reasoning engine" di un sistema fuzzy. Tali programmi devono essere utilizzati in combinazione con reti neurali generate con Neuronx utilizzando l' opzione delle interfacce fuzzy: in pratica le operazioni di fuzzificazione e defuzzificazione vengono eseguite all' interno della rete, mentre la esecuzione delle regole avviene all' interno del "reasoning engine". In questo modo è possibile, anche grazie alla architettura modulare di tipo object oriented del sistema, creare sistemi ibridi contenenti strati alternati delle tecnologie fuzzy e neuronale([fig.2](#)). Le reti generate hanno due strati hidden che possono avere fino a 100 neuroni ciascuno e un massimo di 100 inputs e 100 outputs.

Neuronx può girare su un 286 con 560 kb di memoria disponibile per programmi, facilmente ottenibile su DOS 6.0; la presenza del coprocessore matematico 287 accelera notevolmente il processo di apprendimento.



ADDESTRAMENTO

La prima cosa che bisogna analizzare è il formato che devono avere i dati nel training file che, deve essere un normale file ascii

organizzato con gli esempi in sequenza. Ciascuno è composto dalla sequenza degli input e degli output, come mostrato in tabella 1. I valori devono essere normalizzati, cioè portati alla scala 0.0 -1.0 , e ciò è fattibile anche usando il tool "data normalization" presente nel menu alla voce "data preprocessing". Se si ha un file di dati non normalizzato, bisogna inserire i valori minimo e massimo contenuti nel file ed eseguire la normalizzazione. Preparato il file normalizzato, è necessario definire il numero degli inputs, degli outputs (tipici del problema) e il numero dei neuroni di ognuno dei due strati hidden. Fatta questa operazione si può passare alla fase di addestramento vero e proprio con l'opzione "ebp learning" che chiederà il nome del training file, il numero degli esempi in esso contenuti, la costante di apprendimento (è opportuno scegliere valori intorno a 0.5) e il target error (un errore 0.05 rappresenta un errore del 5% e può andare molto bene in numerose applicazioni). Durante la fase di addestramento vedrete l'errore, prodotto dalla rete in esecuzione, diminuire ad ogni epoca a causa delle correzioni effettuate sui pesi dall'algoritmo di retropropagazione degli errori. In un primo momento, potreste vedere che l'errore cresce anziché diminuire: questo fenomeno è dovuto ad uno stato confusionale che la rete ha inizialmente a causa della inizializzazione random dei valori dei pesi. Quando la rete raggiunge il target error ferma l'addestramento e, a questo punto, è possibile testare la rete con degli esempi.

tabella 1

0.2 input1 esempio 1

0.3 input 2

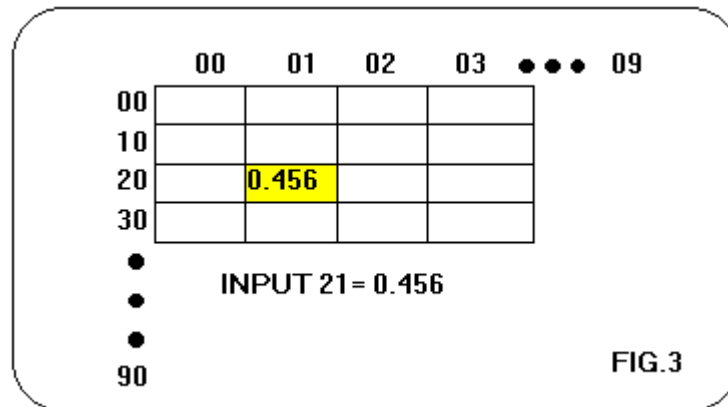
0.5 output 1

0.1 input 1 esempio 2

0.6 input 2

0.7 output1

sequenza dei dati nel file di addestramento



AMBIENTE DI TEST

In fase di test la rete legge i dati di input dal file NET.IN e scrive i dati di output nel file NET.OUT. Scegliendo la opzione "test" eseguite la rete con i dati contenuti nel file suddetto e modificateli da interfaccia dopo ogni esecuzione([fig.3](#)). Potete usare dati che non erano presenti nel training set, testando il potere di generalizzazione della rete in interpolazione e in estrapolazione. Se siete soddisfatti, potete scegliere di salvare in file il contenuto dei pesi della rete che potrete ricaricare in un secondo tempo per fare aggiornamenti con un nuovo training set migliorato o aggiornato, senza dover partire da zero: potete fare queste operazioni con i comandi "save weight on file" e "load weight from file". A questo punto è possibile generare il codice c .

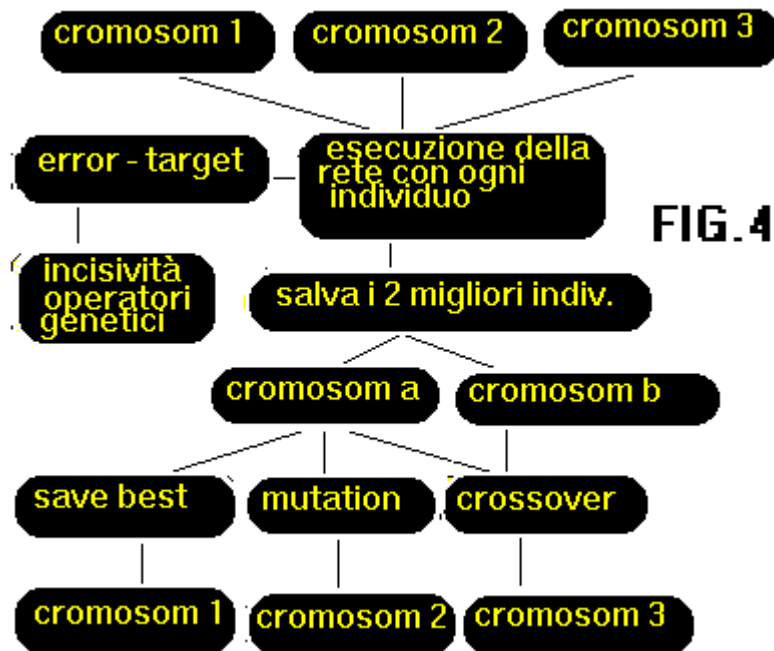
GENERAZIONE CODICE C

Si può immediatamente generare il codice c relativo alla rete neurale addestrata scegliendo se deve essere un programma indipendente o una funzione da inserire dentro altri programmi. In ogni caso, una volta scelto il nome del file (senza estensione), verranno generati i file "nome.c" che, contiene il codice, "nome.h", che è un header file che contiene il know how della rete. Sia che si abbia una funzione o un programma indipendente la rete comunica tramite i file "nome.in" e "nome.out". Non è possibile inserire più di una rete neurale come funzione all'interno di un programma a causa della definizione globale di alcuni dati, ma sarebbe comunque difficile farlo a causa della dimensione delle matrici dei pesi. Dovendo costruire un sistema complesso, comprendente più reti neurali, è consigliabile realizzare programmi indipendenti da collegare tramite files batch o tramite system() da qualunque linguaggio. Un esempio di file batch che utilizza tale sistema è rappresentato nel [listato 1](#).

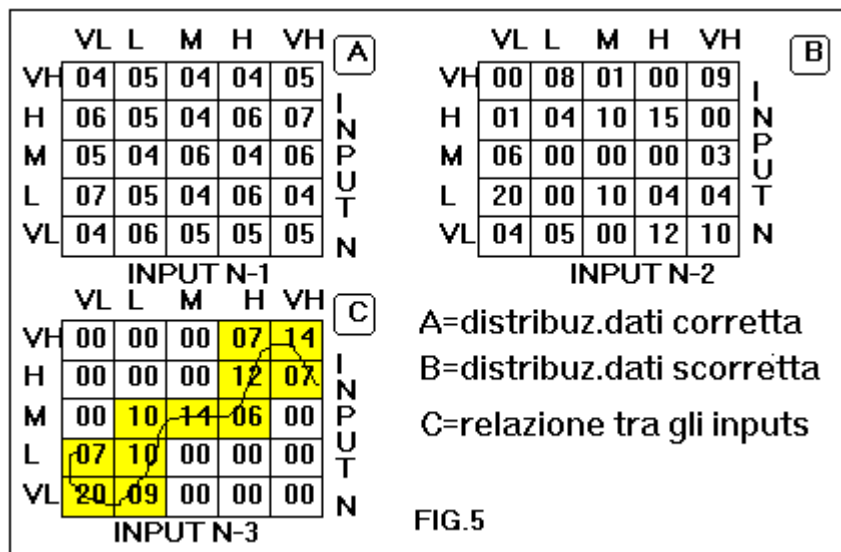
ADDESTRAMENTO CON ALGORITMO GENETICO

Si può addestrare la rete utilizzando il tipico algoritmo di retropropagazione dell' errore, ma anche tramite un algoritmo che sfrutta i principi dell' evoluzione di Darwin. Gli algoritmi genetici sono ampiamente utilizzati nella moderna intelligenza artificiale per risolvere problemi di ottimizzazione. La base fondamentale di un algoritmo genetico è una popolazione di individui (cromosomi) composti, ciascuno, da un certo numero di geni che rappresentano caratteri dell' individuo. Un individuo può avere caratteri più adatti alla soluzione del problema di un altro: si dice che la "fitness function" per quell' individuo porta ad un valore più vicino alla soluzione. Per fitness function si intende una funzione che lega i caratteri del cromosoma (le variabili indipendenti nel

problema) alla variabile che rappresenta la soluzione del problema. Un algoritmo genetico deve calcolare la fitness function per ogni individuo della popolazione e salvare gli individui migliori. Partendo da questi individui, deve generare una nuova popolazione tramite gli operatori "crossover" e "mutation" che, rispettivamente, scambiano geni tra cromosomi e creano piccole mutazioni casuali su alcuni geni dei cromosomi. Viene ricalcolata la fitness function per ogni individuo della nuova popolazione e salvati gli individui migliori. Questo ciclo viene ripetuto un numero molto elevato di volte creando sempre nuove "generazioni di individui". Nel nostro caso, un individuo o cromosoma è rappresentato dal vettore contenente tutti i pesi dei collegamenti tra gli strati neuronali della rete e ogni singolo peso rappresenta un gene del cromosoma. Ogni individuo viene testato e, in questo caso, la fitness function coincide con l'esecuzione della rete stessa. Gli individui migliori sono quelli che portano ad un errore globale più vicino al target. Si parte con tre individui e si salva il migliore e, come sopra esposto, si crea da esso una nuova popolazione di tre individui con gli operatori crossover e mutation, lasciando invariato un individuo per impedire possibili involuzioni. Questo ciclo viene ripetuto fino al raggiungimento del target error. Un addestramento con algoritmo genetico è utile su livelli molto bassi di errore della rete per due motivi: il primo è il fatto che un algoritmo genetico può superare i minimi locali e, il secondo è che su livelli alti di errore il classico algoritmo a retropropagazione dell'errore è molto efficiente mentre su livelli di errore bassi, cioè quando ci si avvicina al minimo assoluto (o comunque ad un minimo), tale algoritmo produce avvicinamenti al target molto lenti (ricordate la tecnica della discesa del gradiente e il fatto che la derivata di una funzione su un punto di minimo è nulla?)



Per questo motivo l'apprendimento con algoritmo genetico può partire solamente da errori inferiori a 0.5, perché per errori maggiori la retropropagazione degli errori risulta più efficiente. Si può anche utilizzare un tool ibrido che inizia l'addestramento con algoritmo ebp e continua con algoritmo genetico, automaticamente, in prossimità del 10% di distanza dal target. Un diagramma di flusso dell' algoritmo genetico è rappresentato in [fig.4](#).



TEST DISTRIBUZIONE DATI

Si può effettuare un test sulla distribuzione dei dati contenuti nel training set al fine di verificare se esistono concentrazioni di dati non uniformemente distribuite sul range di ciascun input. Una disuniforme distribuzione dei dati potrebbe compromettere la capacità di generalizzazione della rete, anche dopo un risultato soddisfacente in fase di addestramento. Scegliendo la opzione "data distribution test" contenuta in "data preprocessing", si ottiene una relazione di distribuzione per ogni coppia di input dove il range di ogni input viene diviso in cinque regioni come mostrato in [fig.5](#). Una corretta distribuzione si ha quando la quantità di dati è equamente distribuita in ogni casella per ogni coppia di input. Tramite questo test è possibile anche scoprire una possibile relazione tra due inputs: ciò rappresenterebbe un appesantimento inutile per il training della rete ed è conveniente eliminare una delle variabili correlate.

INTERFACCE FUZZY

Come precedentemente accennato, esiste la possibilità di definire sia input che output della rete in formato fuzzy. Questo significa che la rete accetta dati in input in formato fuzzy ed effettua una defuzzificazione, sulla base delle classi definite dall'utente, per presentare agli input effettivi dei valori numerici. Si può scegliere anche la fuzzificazione degli output che prevede la operazione inversa: i valori numerici in uscita vengono trasformati in formato fuzzy, secondo le classi definite dall'utente, per ogni output. Defuzzificazione in ingresso e fuzzificazione in uscita sono indipendenti, cioè possono essere selezionati separatamente. Il formato di rappresentazione dei dati fuzzy è il seguente:

input /output n

0 classe 0

.34 degree of membership

\$ separatore di classe

1 classe 1

.56 degree of membership

& separatore di input/output(nuovo input/output)

1 classe 1

.45 degree of membership

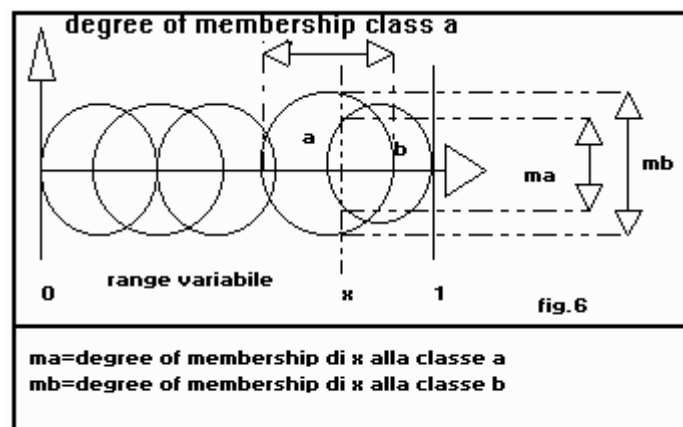
\$ separatore di classe

2 classe 2

.20 degree of membership

& separatore di input/output

Come vedremo è possibile collegare reti neurali con queste interfacce con programmi "fuzzy reasoning engine" generati da Fuzzkern che utilizzano, naturalmente, le stesse convenzioni di input / output. La fuzzificazione avviene attraverso funzioni "circolari" con uso di funzioni seno e coseno, anziché le classiche triangolari/trapezoidali, e per questo motivo sono state chiamate "bubble". Le classi circolari scendono verso gli estremi in modo non lineare ([fig.6](#)). La defuzzificazione avviene attraverso la regola del "center of gravity" semplificata che abbiamo visto nel capitolo dedicato al ragionamento sfumato.



ma e mb vanno normalizzati sulla dimensione della classe

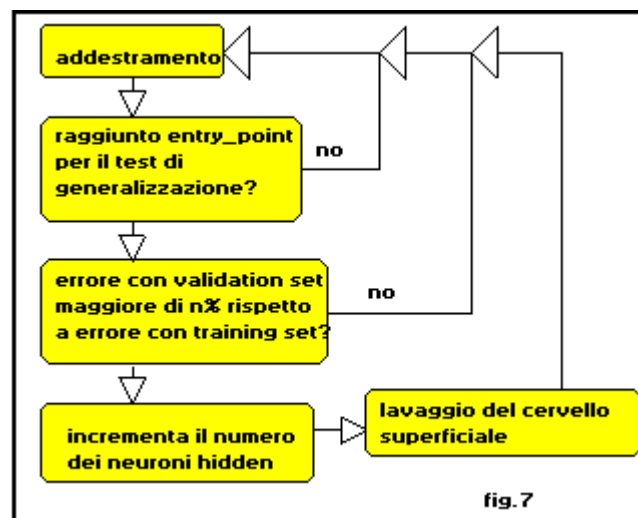
TEST SUL POTERE DI GENERALIZZAZIONE

Come ormai ben sapete, una delle caratteristiche salienti delle reti neurali è quella di godere di una certa elasticità di ragionamento che consente di avere risposte significative anche fuori dal training set. Più precisamente, possiamo dare come input alla rete valori non presenti nel training set chiedendole di effettuare una interpolazione o una estrapolazione. Il problema in questione è molto legato alla possibilità di avere un "overtraining" della rete, cioè un addestramento troppo incisivo. Effettivamente quando si

addestra una rete neurale raggiungendo errori molto piccoli si rischia di insegnare alla rete a riconoscere il "rumore", per cui dati estranei al training set possono diventare dati estranei in senso assoluto: la rete in questo modo ha perso potere di generalizzazione. Bisogna tenere presente che una rete neurale non è un preciso sistema di calcolo ma sistema di calcolo impreciso anche se intelligente: quando addestriamo una rete neurale, dobbiamo trovare una soluzione bilanciata tra la necessità di raggiungere un errore piccolo sul training set e la richiesta di potere di generalizzazione che, comunque, dipende anche dalla validità dei dati scelti per costruire il training set. Normalmente un minore numero di connessioni sinaptiche in rapporto al numero di esempi del training set induce una maggiore capacità di generalizzazione. Questo è dovuto al fatto che la rete non ha abbastanza memoria per "imparare" anche il "rumore". Una tecnica utilizzata è quella di eliminare connessioni sinaptiche che subiscono scarse variazioni durante il processo di addestramento, attuando così una ottimizzazione "a finestre" del complesso delle connessioni sinaptiche. Il programma Neurfuzz nasce con obiettivi sperimentali e segue, in questo caso, una strada completamente differente, che ha dato risultati positivi solo in certe condizioni, con particolari training sets. Questa opzione è rimasta nel programma sebbene non sia stata validata in alcun contesto definito e specificabile. La procedura utilizzata in questa opzione sperimentale è descritta di seguito. Per ogni epoca, a partire da un certo errore (che può essere scelto dall'utente), viene calcolato l'errore globale, ottenuto con il validation set, e confrontato con quello ottenuto con il training set: se esiste una differenza percentuale maggiore di quella scelta dall'utente, il numero dei neuroni degli strati hidden viene automaticamente incrementato. Dopo questa procedura, viene effettuata una operazione molto particolare che, consiste in un superficiale "lavaggio del cervello", ottenuto con piccole variazioni casuali e distribuite sui pesi degli strati hidden. Questa operazione ha come

primo effetto evidente, quello di elevare notevolmente l'errore della rete, ma errori molto piccoli vengono nuovamente raggiunti in tempi brevissimi (effettivamente è rimasta una matrice profonda dell'addestramento precedente) e con una "ridistribuzione" del know how su tutti i neuroni, indispensabile per il raggiungimento dello scopo.

Il diagramma di flusso di tutta la operazione è rappresentato in [fig.7](#).



TEST SULLA RESISTENZA AL RUMORE

Si possono effettuare due operazioni che simulano "guasti neuronali" al fine di verificare la robustezza della rete basata sul fatto che l'informazione è realmente frammentata e distribuita sul complesso dei collegamenti sinaptici. L'operazione di "lobotomy", ovvero asportazione di una piccola parte del cervello o, diverse operazioni di "microchirurgia" su singole connessioni sinaptiche, possono essere effettuate a questo scopo.

SIMULATED ANNEALING(*): REGIME TERMICO DINAMICO

Uno dei problemi che si possono presentare nell' addestramento di una rete neurale è quello della caduta in un minimo locale (rimando al capitolo sulla retropropagazione dell' errore). Uno degli espedienti che sono stati sperimentati per superare tale problema è quello di adottare una legge di tipo probabilistico del neurone, tale che le variazioni di attivazione vengano accettate con un certo grado di probabilità crescente nel tempo. Facciamo un esempio che dia un'immagine "fisica" del problema: immaginate di avere un piano di plastica deformato con "valli" e "monti" di varie estensioni e profondità, e di porvi sopra una pallina con l'obiettivo di farla cadere nella valle più profonda (minimo assoluto). Molto probabilmente la pallina cadrà nella valle più vicina al punto in cui la abbiamo messa che potrebbe non essere la valle più profonda. Il processo di simulated annealing consiste nel muovere il piano di plastica in modo che la pallina lo percorra superando monti e valli e diminuire la intensità del movimento gradualmente fino a zero. Molto probabilmente, la pallina rimarrà nella valle più profonda dalla quale, a un certo punto, il movimento non sarà più sufficiente a farla uscire. Nella nostra rete tutto questo non viene realizzato con un vero algoritmo probabilistico, ma si utilizza una legge di attivazione a sigmoide modificata che contiene un parametro "temperatura":

$$A = 1 / (1 + e^{**P/t})$$

A=valore di uscita del neurone

P=valore di attivazione del neurone(sommatoria degli ingressi)

t=temperatura

Nella cosiddetta Macchina di Boltzmann, che fa uso di una legge di attivazione probabilistica, la temperatura viene normalmente

fatta diminuire gradualmente nel tempo. Nel nostro programma la temperatura viene correlata alla differenza tra errore della rete e target error in modo che diminuisca con il diminuire di questa differenza. In principio la sigmoide ha una pendenza molto lieve garantendo risposte di output poco intense a variazioni di input anche elevate; in seguito la sigmoide si avvicina sempre più al gradino per cui il neurone ha risposte intense a piccole variazioni di input([fig.8](#)). Con questo sistema, in questo specifico programma, è possibile avere risultati molto interessanti nella risoluzione di certi problemi e avere risultati negativi con altri tipi di problemi. Ciò dipende, effettivamente, da come si presenta la "superficie" ottimale dei pesi per la risoluzione del problema stesso. Utilizzando questo processo, si può notare un mantenimento più lungo di livello di errore molto elevato, che talvolta (problema adatto a questo tipo di soluzione), scende a valori estremamente bassi in tempi rapidissimi([fig.9](#)). Bisogna precisare che, la Macchina di Boltzmann nulla ha in comune con questa rete e, il principio del Simulated Annealing è stato, in questo contesto, applicato con modalità differenti([**](#)).

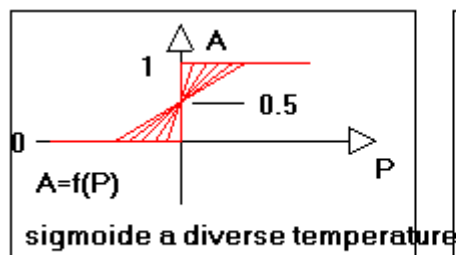


fig.8



fig.9

THERMOSHOCK

Questa funzione è orientata a risolvere il problema dei minimi locali, come quella precedente, ma con un criterio differente. Viene effettuato un test sulla derivata dell'errore e viene effettuato uno shock termico sulla rete se l'errore ha una discesa troppo lenta o ha una oscillazione con media costante ed è lontano dal target error. Lo shock termico viene realizzato tramite una variazione casuale dei pesi della rete. Ritengo che siano veramente pochi i problemi che possano trarre beneficio da questa utility che, comunque, resta interessante come curiosità didattica.

FUZZKERN : DEFINIZIONE DELLE REGOLE DI UN REASONING ENGINE

Fuzzkern permette la costruzione di programmi che effettuano la scansione ed esecuzione di regole tipo "if (x is in class a) then y is in class b". Tutto ciò ha senso solamente se, in input, esistono dati in formato fuzzy e se, in output, esiste qualcosa in grado di capire dati in formato fuzzy. Questi programmi (generati da Fuzzkern) devono infatti essere interfacciati con reti neurali create con Neuronx utilizzando la defuzzificazione degli input e la fuzzificazione degli output. Il programma chiede in sequenza le regole che devono essere inserite nel rules_base ed esiste la possibilità di inserire degli operatori di tipo and tra una regola e la successiva:

if ((x1 is in class a) && (x2 is in class c)) then y1 is in class e

Una limitazione durante la stesura delle regole è che, un eventuale gruppo di regole linkate con operazione "and" deve essere inserito

prima di regole libere che condividano la stessa conseguenza. Questo è dovuto al fatto che il calcolo della forza della regola (e quindi il degree of membership nella conseguenza), nel caso di regole `and_linkate`, viene effettuato tramite una operazione di `fuzzy_and` (scelta del valore più basso), mentre nel caso di due regole non vincolate da "and" si effettua una operazione di `fuzzy_or` (scelta del valore più elevato): il bug consiste nella mancanza di una traccia che permetta un feedback di verifica sul `and_link` flag delle regole. In una successiva versione questo problema potrebbe essere corretto, ma per il momento la presenza di regole `and_linkate` dopo regole libere che esprimano la stessa conseguenza può alterare il calcolo della "strength of the rule".

LISTATO 1

```
input      (programma lettura dati input da dispositivo o file
           e scrittura su file network1.in)
network1   (fuzzy out network)
ren network1.out sysfuzz.i n      (trasferimento dati)
sysfuzz1   (fuzzy reasoning engine)
ren sysfuzz.out network2.in      (trasferimento dati)
network2   (fuzzy in network)
output     (programma di lettura dati da file network2.out
           e scrittura su dispositivo o file)
```

(*) **SIMULATED ANNEALING**: significa ricottura simulata e si ispira al processo di ricottura dei vetri di spin ("spin glasses"). Un vetro di spin è un metallo contenente delle impurità di un altro metallo in percentuali molto basse ed ha caratteristiche magnetiche particolari. Un metallo puro può essere ferromagnetico (ferro) o non ferromagnetico (cromo). I metalli ferromagnetici hanno tutti gli atomi con lo stesso momento magnetico o spin (vettori con stessa direzione e stesso verso). Un metallo non ferromagnetico ha tutti gli atomi adiacenti con spin in posizione di equilibrio (stessa direzione ma verso differente). Portando un metallo al di sopra di una certa temperatura critica, gli spin degli atomi assumono direzioni e versi casuali (un metallo

ferromagnetico perde le sue proprietà magnetiche); il raffreddamento del metallo al di sotto della temperatura critica porta tutti gli spin verso una posizione di equilibrio (minima energia) ferromagnetico o non. Nei vetri di spin il raffreddamento congela, invece, gli spin in uno stato caotico, pertanto, a bassa temperatura, non esiste uno stato di minima energia, ma molti minimi relativi. Un raffreddamento lento permette di raggiungere un minimo ottimale (più vicino alla energia minima assoluta).

(**) **MACCHINA DI BOLTZMANN:** Si tratta di una rete neurale derivata dalla rete di Hopfield che, attraverso opportune formule di apprendimento, converge verso uno stato di equilibrio compatibile con i vincoli del problema. Può essere una memoria associativa ma è impiegata, in particolare, per la risoluzione di problemi di ottimizzazione, in cui la fase di apprendimento si basa sulla dimostrazione (di Hopfield) che, alla rete è associata una funzione "energia" da minimizzare rispettando i vincoli del problema (ottimizzazione). La Macchina di Boltzmann è sostanzialmente una rete di Hopfield in cui i neuroni adottano una legge di attivazione di tipo probabilistico: se da uno stato $S(t)$ la rete evolve verso uno stato $S(t+1)$ allora, questo nuovo stato viene sicuramente accettato se la energia della rete è diminuita (o è rimasta invariata), altrimenti viene accettato, ma solo con una certa probabilità che, aumenta al diminuire della temperatura ([figg.10/11](#)). Questo sistema consente il raggiungimento di una soluzione molto vicina all'ottimo assoluto nei problemi di ottimizzazione, secondo la teoria precedentemente esposta. La nostra rete non è una memoria associativa ma una rete multistrato "feed-forward" (propagazione del segnale da input verso output attraverso gli strati intermedi), utilizzata principalmente per estrazione di funzioni matematiche non conosciute da esempi input/output (si può comunque utilizzare anche per problemi di classificazione e ottimizzazione). Questa rete utilizza un algoritmo di apprendimento basato sulla

retropropagazione degli errori e non sulla evoluzione verso stati energetici bassi. La applicazione del Simulated Annealing su questa rete è stata realizzata in via sperimentale con risultati soddisfacenti nella soluzione di diversi problemi; la filosofia di trasporto di questa teoria su questo tipo di rete (non potendosi basare su stati energetici) è stata quella di fornire i neuroni di una legge di attivazione a sigmoide variabile in funzione della temperatura: i neuroni si attivano (a parità di segnali di ingresso) maggiormente alle basse temperature (questo comporta anche una differente attività di modifica dei pesi delle connessioni sinaptiche da parte dell' algoritmo di error_back_propagation alle diverse temperature).

ANALISI DEL CODICE C GENERATO

I listati 2 e 3 sono i files prova.c e prova.h generati da Neuronx, dopo un addestramento sul problema "giocattolo" del riconoscimento di profili altimetrici con cinque inputs. Il [listato 2](#) contiene le procedure di esecuzione della rete e quelle di input/output e normalizzazione. Le procedure di fuzzificazione e defuzzificazione contengono solo un return perché questa rete non ha interfacce fuzzy (defuzzy_in_flag=0 e fuzzy_out_flag=0 nei dati pubblici). Le procedure di normalizzazione e di denormalizzazione contengono il codice relativo perché sono attivate dai flag normaliz_flag=1 e denormaliz_flag=1 (dalla scala reale della variabile alla scala 0.0 - 1.0 e viceversa). Il [listato 3](#) è il file prova.h (include in prova.c) che contiene le matrici w1, w2, w3 di valori float corrispondenti ai pesi sinaptici che connettono lo strato input con lo strato hidden1 di neuroni, lo strato hidden1 con lo strato hidden2 e lo strato hidden2 con lo strato di output. Questi valori corrispondono ai valori ottenuti dopo un addestramento con algoritmo a retropropagazione dell' errore sul problema dei profili altimetrici. Tale addestramento è stato effettuato in regime termico

dinamico, come dimostra la dichiarazione "float t=0.513044" nei dati pubblici: tale valore sarebbe uguale a 1 in caso contrario. Questo valore di temperatura è quello associato al momento in cui è stato fermato il processo di addestramento. Nel caso in cui vengano scelte caratteristiche differenti nel pretrattamento dei dati (come la normalizzazione quadratica) o l'interfacciamento con dati sfumati, il codice c avrà differenti valori nei flags definiti nei dati pubblici e saranno presenti le funzioni relative complete.

LISTATO 2

```

/* THIS PROGRAM IN C LANGUAGE IS A EBP
   NEURAL NET CREATED WITH NEURONX1
   (NEURAL NETS C CODE GENERATOR)
   BY LUCA MARCHESE */

#include <stdio.h>
#include <math.h>
#include <prova.h>          /*BUG: correggere in #include "prova.h"*/
#define maxline 100

int nx=5;
int ny=5;
int nh1=9;
int nh2=9;
float func_res;
int normaliz_flag=1;
int denormaliz_flag=1;
int square=0;
float offset=0.000000;
float normak=100.000000;
int defuzzy_in_flag=0;
int fuzzy_out_flag=0;
float t=0.513044;
float x[100];
float h1[100];
float h2[100];
float y[100];

main()
{
  if(input())
  {
    exec();
    output();
    return 1;
  }
}

```

```

    }
    return 0;
}

exec()
{
    int k=0;
    int j=0;
    float a;
    while(k!=nh1) h1[k++]=0;           /*hidden1 initalization = 0*/
    h1[k]=1;                           /*bias*/
    k=0;
    while(k!=nh2) h2[k++]=0;           /*hidden2 initialization = 0*/
    h2[k]=1;                           /*bias*/
    k=0;
    while(k!=ny) y[k++]=0;             /*output initialization = 0*/
    x[nx]=1;                           /*input bias initialization*/
    k=0;
    while(j!=nh1)                       /*h1 calculation*/
    {
        k=0;
        a=0;                           /*any h1=0 initialization*/
        while(k!=nx+1) a=a+(w1[j][k]*x[k++]); /*h1 activation calculation*/
        f(a);
        h1[j++]=func_res;              /*output calculation*/
    }
    j=0;
    while(j!=nh2)                       /*h2 calculation*/
    {
        k=0;
        a=0;                           /*any h1=0 initialization*/
        while(k!=nh1+1) a=a+(w2[j][k]*h1[k++]); /*h2 activation calculation*/
        f(a);
        h2[j++]=func_res;              /*output calculation*/
    }
    j=0;
    while(j!=ny)                       /*y calculation*/
    {
        k=0;
        a=0;                           /*any y=0 initialization*/
        while(k!=nh2+1) a=a+(w3[j][k]*h2[k++]); /*y activation calculation*/
        f(a);
        y[j++]=func_res;              /*output calculation*/
    }
    return;
}

f(a)
float a;
{
    float b;
    b=1/(1+exp(-a/t));
    func_res=b;
    return;
}

input()
{

```

```

char *file="prova.in";
FILE *fpin;
int k=0;
if(!(fpin=fopen(file,"r")))
{
    printf("ERROR OPENING INPUT FILE\n");
    return 0;
}
if(!defuzzy_in_flag)while(k!=nx) fscanf(fpin,"%f",&x[k++]);
x[k]=1;
if((normaliz_flag)&&!defuzzy_in_flag) input_data_normalizer();
if((defuzzy_in_flag)&&!normaliz_flag) defuzzification_in(fpin);
fclose(fpin);
return 1;
}

```

```

output()
{
    char *file="prova.out";
    FILE *fpout;
    int k=0;
    if(!(fpout=fopen(file,"w")))
    {
        printf("ERROR OPENING OUTPUT FILE\n");
        return ;
    }
    if((fuzzy_out_flag)&&!denormaliz_flag) fuzzification_out(fpout);
    if((denormaliz_flag)&&!fuzzy_out_flag) output_data_denormalizer();
    if(!fuzzy_out_flag)while(k!=ny) fprintf(fpout,"%f\n",y[k++]);
    fclose(fpout);
    return ;
}

```

```

input_data_normalizer()
/*normalization to values v  0 < v < 1*/
{
    int k=0;
    while(k!=nx) x[k]=(x[k++]+offset)/normak;
    k=0;
    if(square) while(k!=nx);
    {
        x[k]=x[k]*x[k];
        k++;
    }
    return;
}

```

```

output_data_denormalizer()
{
    int k=0;
    if(square) while(k!=nx) y[k]=sqrt(y[k++]);
    while(k!=ny) y[k]=y[k++]*normak-offset;
    return;
}

```

```
fuzzification_out(fpout)
FILE *fpout;
{
    return;
}
```

```
defuzzification_in(fpin)
FILE *fpin;
{
    return;
}
```

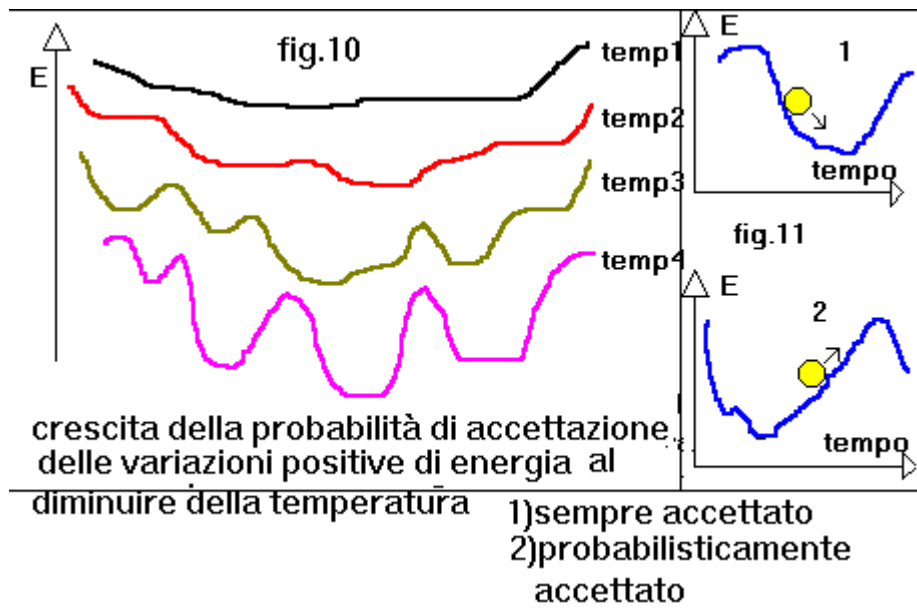
LISTATO 3

```
/*LEARNING_HEADER_FILE*/
```

```
float far w1[9][6]={3.248578,-0.168329,-4.854014,-0.148738,0.435997,1.908123,
-2.819315,-1.173162,-3.325368,2.926156,7.184569,-1.274087,2.919860,2.775792,
6.331062,-2.334243,-8.806980,-0.069663,-6.937530,0.348315,2.925305,1.283903,
3.154221,1.400293,7.618520,0.423969,-3.707562,-1.197842,-2.593067,0.737628,
0.106994,0.145101,-0.007456,-0.039538,0.008107,0.094729,-0.098390,0.133408,
0.338889,-0.056152,0.173400,0.021692,0.019445,-0.286478,0.103537,0.111047,
0.314026,0.006780,-0.013501,0.176355,-0.021135,0.181239,0.006862,-0.105471,
};
```

```
float far w2[9][10]={-2.077883,0.767727,-0.584588,3.150063,-3.833194,-0.389686,
0.229002,0.096596,0.114084,0.163734,-1.022036,3.448938,-5.098722,1.568256,
-2.456146,0.526446,0.266967,0.595630,0.455010,0.131174,1.849635,-3.917503,
5.342775,-2.050846,4.432072,1.498487,0.475523,-0.349084,0.128589,-0.072857,
-3.624948,-3.981793,7.178647,-2.805038,1.329446,-2.564022,0.162812,0.613991,
0.123508,0.177819,-3.550605,-2.135846,4.345431,4.950200,-3.462358,-1.334133,
0.587248,-0.124747,0.002104,0.181317,0.211475,0.871656,-0.030261,0.460907,
0.048518,1.088949,0.513699,0.319656,0.529848,-0.166475,-0.023568,0.813581,
0.321707,0.788963,-0.412579,-0.013629,-0.024483,-0.412023,0.575393,0.366901,
-0.029459,0.582926,0.173883,-0.244128,0.175112,0.569956,0.577285,-0.222923,
0.170567,0.376047,0.019090,0.387615,0.012755,-0.027384,0.024187,0.404193,
0.007966,0.404439,0.404142,0.403559,
};
```

```
float far w3[5][10]={0.716600,-2.232694,-0.664696,0.841474,7.530255,-5.672661,
-0.214670,-0.129960,-0.109924,-0.072957,-3.450566,4.449509,1.359750,-2.661987,
-7.623822,-2.760651,-0.088083,0.320317,0.132743,-0.177554,2.527434,4.761584,
-7.463439,-2.054665,-1.135034,-1.482672,0.270768,-0.048533,0.245478,-0.065364,
-1.509188,-8.215578,0.146611,4.062209,-3.707586,-2.923457,-0.348058,0.133766,
0.665961,0.090000,-1.948043,-2.391392,4.478242,-5.598677,2.806209,-2.877502,
-0.256568,-0.355556,-0.072116,-0.164903,
};
```



GUIDA AL FLOPPY DISK

CAPITOLO N.1

Nella directory "cap.1" sono presenti i seguenti files:

- associa.c: sorgente in codice C del programma dimostrativo associa.exe
- associa.exe: programma dimostrativo che simula una memoria associativa tipo b.a.m (bidirectional associative memory).
- a.min, a.cap, b.min, b. cap, c.min, c.cap: sono files ascii che contengono le "immagini" di lettere dell'alfabeto che possono essere associate.

Per associare le immagini è sufficiente eseguire la funzione apprendimento e fornire i nomi dei files che si vogliono associare come img.x(input) e img.y (output). Dopo un ciclo di apprendimento si può effettuare una esecuzione della rete fornendo come input la immagine usata come img.x in apprendimento. Si può visualizzare l'immagine y (output) che la rete ha fornito come "immagine associata" all' input. Possono essere modificati i files delle immagini di input in modo da inserire del "rumore": dopo una esecuzione della rete con un input rumoroso si può constatare, tramite la funzione di visualizzazione dell' immagine x, che lo stesso contenuto dell' input è stato corretto dai cicli di retroazione della rete. Questo programma contiene anche una funzione per calcolare la distanza di Hamming tra due immagini, nel formato ascii proposto, come spiegato nel capitolo.

CAPITOLO N.2

Nella directory cap.2 si trovano i seguenti files:

- backprop.c: sorgente in linguaggio C della rete neurale error back propagation backprop.exe.

- backprop.exe: programma di simulazione di una rete neurale a retropropagazione dell'errore con due strati nascosti.

- profil.lrn: file contenente gli esempi di addestramento della rete inerenti al problema "giocattolo" del riconoscimento di profili altimetrici: un esempio è raffigurato in tab.1.

- profil.val: file contenente gli esempi del validation set necessari alla verifica della capacità di generalizzazione della rete dopo l'apprendimento.

- profil.wgh: contiene i valori sinaptici della rete dopo un addestramento sul file profil.lrn e può essere direttamente caricato con la funzione "load" (la rete risulta come addestrata).

TAB.1

0.345 cinque valori input=monte

0.480

0.897

0.389

0.123

1 monte

0 valle

0 pendenza +

0 pendenza -

0 piano

La prima operazione che bisogna eseguire è quella di definire il numero di inputs e outputs (determinati dal problema) e il numero di neuroni per ognuno degli strati hidden: nel capitolo è stato presentato un esempio in cui si sono utilizzati cinque neuroni per ogni strato hidden. Il problema dei profili altimetrici nel file `profil.lrn` presenta cinque inputs ma si possono creare altri files con più inputs e per problemi differenti. Dopo tale passo si può procedere all'addestramento fornendo il nome del file contenente gli esempi, il valore desiderato per il tasso di apprendimento (consigliabile intorno a 0.5), il numero di epoche che devono essere eseguite senza pausa e il valore di errore che si desidera raggiungere (esempio `target error = 0.03` indica errore del 3%). Una volta raggiunto l'errore desiderato si può testare la rete inserendo dei dati nel file di input: tramite l'editor interno si editi il file `"net.in"` inserendo dei dati inerenti all'addestramento come mostrato in tab.2.

TAB.2

esempio di file input con un profilo altimetrico a cinque valori che rappresenta un "monte".

0.2
0.4
0.8
0.5
0.2

Dopo avere salvato il file `net.in` ed eseguito la rete con la funzione `"exec"` si può editare il file `"net.out"` per verificare il risultato. Quando è stato terminato un valido addestramento è bene salvare il contenuto dei collegamenti sinaptici in un file che potrà essere ricaricato in ogni momento senza dover ripetere il processo di

apprendimento: le funzioni "save" e "load" permettono, rispettivamente, queste due operazioni.

CAPITOLO 3

Il capitolo 3 utilizza ancora il programma Backprop ma fornisce nuovi e più interessanti files di addestramento che rivelano le capacità sorprendenti di una rete neurale multistrato come questa. I files contenuti nella directory cap.3 del dischetto sono:

- backprop.exe: è lo stesso programma del cap.2
- sin_gen.c: sorgente in linguaggio C del programma sin_gen.exe.
- sin_gen.exe: programma che genera files contenenti valori che descrivono sinusoidi e funzioni derivate dalla somma di sinusoidi: una delle due funzioni è periodica mentre l'altra è aperiodica perché una delle due sinusoidi componenti ha periodo incommensurabile.
- period.lrn, aperiod.lrn, compon1, compon2: sono files generati con il programma sin_gen.

CAPITOLO 4

In questo capitolo sono trattate le reti neurali autoorganizzanti e nel dischetto, nella directory cap.4, è contenuta una simulazione di una rete classificatrice autoorganizzante. Complessivamente sono contenuti i seguenti files:

- som.c: file sorgente in linguaggio C del programma som.exe
- som.exe: simulazione di una rete autoorganizzante costruita sulla base di una rete di Kohonen con alcune variazioni che non ne modificano, comunque, la struttura fondamentale.

- profil.lrn: è un file di addestramento per la rete "som" relativo al problema visto nel capitolo 2 del riconoscimento di profili altimetrici.

Come potrete notare il file contiene esempi di cinque valori di altitudine disposti consecutivamente, senza nessun valore di output desiderato: il training set di una rete autoorganizzante come questa non deve contenere valori di output desiderato come invece avviene nei training set di reti ad apprendimento supervisionato(tab.3).

- profil.val: contiene gli esempi di validazione dell'apprendimento. Viene comunemente chiamato validation.set e il programma "som" inizia a testare l'apprendimento appena terminato(completate le iterazioni sul training set) con gli esempi contenuti in questo file.

TAB.3

0.956 cinque valori esempio 1

0.767

0.486

0.800

0.900

0.300 cinque valori esempio 2

0.400

0.500

0.600

0.700

Eseguendo il programma "som" vengono richiesti il nome del file di addestramento, il nome del file di validazione, il numero di inputs(in questo caso sono cinque) e il numero di "iterazioni" che si vuole far compiere all'algoritmo di addestramento. Con un file così semplice e una rete così piccola (100 neuroni sulla griglia

output) sono sufficienti poche iterazioni(5-10) per creare una traccia nei pesi sinaptici. Terminata l'ultima iterazione sul file contenente il training set (es profil.lrn) il programma inizia una fase di test con gli esempi contenuti nel validation.set (es profil.val). In fase di test è visibile la matrice 10x10 dei neuroni di output con un simbolo "*" sul neurone winner (vincente). Sul lato sinistro del video compare l'insieme dei valori dell'input pattern e ciò vi permette di verificare su quali neuroni vengono "classificati" i vari patterns. Con poche iterazioni sono stati ottenuti i risultati di [fig.1](#). Bisogna notare che il problema del riconoscimento di profili altimetrici presenta, per il tipo di rete che abbiamo realizzato, il problema del "pattern piano" che richiederebbe la modifica della legge di trasferimento dei pattern di input sull'output. Nella rete realizzata con il sistema classico si effettua una moltiplicazione vettoriale del vettore di input con il vettore dei pesi che connettono l'input con l'output: è evidente che la moltiplicazione di due vettori con forme simili porta ad un vettore in cui la somma degli elementi è più elevata.

Esempio:

1) monte / monte

$$\text{input}(0.2-0.4-0.5-0.2-0.1) * \text{pesi}(0.3-0.4-0.6-0.4-0.3) = \\ = (0.06-0.16-0.3-0.08-0.03)$$

$$\text{e output} = S(0.06, 0.16, 0.3, 0.08, 0.03) = 0.63$$

2) monte / valle

$$\text{input}(0.2-0.4-0.5-0.2-0.1) * \text{pesi}(0.6-0.4-0.3-0.4-0.6) = \\ = (0.12-0.16-0.15-0.08-0.06)$$

$$\text{e output} = S(0.12, 0.16, 0.15, 0.08, 0.06) = 0.57$$

Da questo esempio risulta evidente che il pattern piano si pone in una situazione di neutralità tra le varie forme e potrà pertanto essere classificato in neurodi di output appartenenti ad altre classi e viceversa. Per una soluzione efficace del problema di profili altimetrici come quello esposto, con una rete autoorganizzante si può intervenire con le seguenti azioni:

1) modificare il trasferimento del vettore di input attraverso il vettore pesi in modo che si effettui una differenza vettoriale anziché una moltiplicazione: questa modifica permette al pattern piano di assumere una "forma propria".

2) effettuare una operazione di "stretching" dei vettori di input in modo che la rete diventi sensibile solamente alla "forma" del vettore e non ai valori assoluti dei suoi elementi. Mentre l'algoritmo di apprendimento resta invariato, la funzione di trasferimento diventa:

$$O(k) = 1 / (S |x(j) - w(k)(j)| + e)$$

$$e = 0.01$$

Lo stretching del vettore di input si ottiene invece come preprocessing prima dell'inserimento nella rete. Lo stretching può essere effettuato in questo modo:

vettore iniziale = 0.2, 0.3, 0.4, 0.2, 0.1 (monte)

Si sottrae ad ogni elemento il valore più basso tra tutti gli elementi, in modo da portare la base del vettore sullo zero e poi si stira il vettore in modo che l'elemento di valore maggiore diventi uguale a 1.0. Per fare ciò si imposta a 1.0 l'elemento di valore più elevato e si dividono tutti gli altri elementi per il suo valore. In questo esempio valore min = 0.1 e la prima operazione porta al nuovo vettore:

0.1,0.2,0.3,0.1,0.0

Adesso valore max=0.3 diventa 1.0 e

0.1 diventa $0.1/0.3=0.33$

0.2 diventa $0.2/0.3=0.66$

Il nuovo vettore è 0.33,0.66,1.0,0.33,0.0

Nel caso di valore max = valore min (es piano) si avrebbe un vettore di elementi nulli non accettabile per questa rete: si può ovviare aggiungendo 0.1 ad ogni elemento quando tale condizione si verifica. Questo procedimento è rappresentato graficamente in [fig.2](#) dove si evidenzia come di "enfaticizzare" la forma del pattern di input amplificandolo fino alla saturazione del range di lavoro. In questo modo si amplificano le differenze di forma tra i patterns e si equalizzano i valori assoluti. Esistono molti tipi di preprocessing dei dati per la presentazione di essi alle reti neurali e sono strettamente vincolati al tipo di problema da esaminare e al tipo di rete utilizzata. Un corretto preprocessing dei dati è molto spesso determinante per il raggiungimento di risultati affidabili con qualsiasi rete neurale. Quello appena visto può essere utilizzato nel riconoscimento di segnali, di profili, di immagini o comunque in tutte quelle applicazioni in cui è importante distinguere forme indipendentemente da valori assoluti.

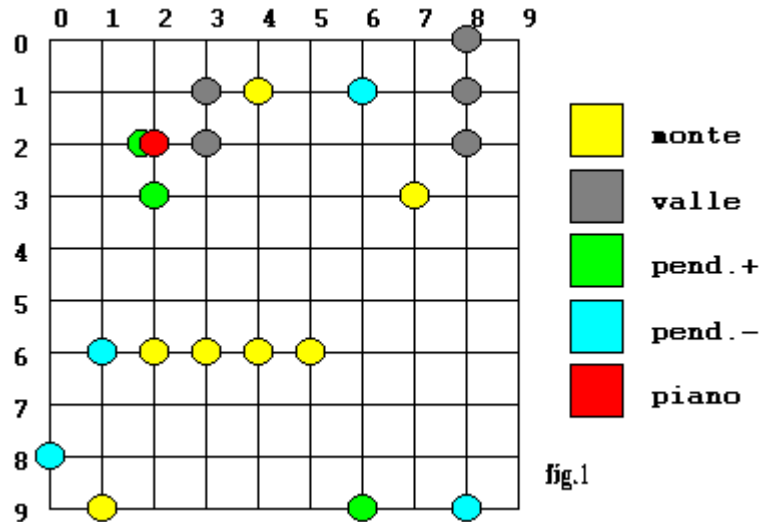


fig.1

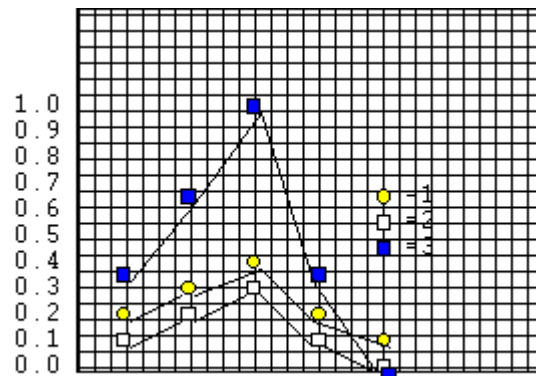


fig.2

CAPITOLO 6

Nella directory cap.6 sono contenuti i files eseguibili di un completo programma di generazione di reti neurali a retropropagazione dell'errore in codice C. Inoltre è presente anche un programma che permette di generare codice C per motori inferenziali basati su regole tipo "if then" operanti su dati fuzzy. Il programma Neuronx è in grado di addestrare un rete neurale a retropropagazione dell'errore con due strati nascosti e di generare il codice c "addestrato". Il programma Fuzzkern è in grado di generare motori inferenziali in codice C editando le regole nella apposita shell. I programmi generati da Neuronx e da Fuzzkern sono interfacciabili, per cui i due tools sono stati raggruppati nel

menù Neurfuzz: per avere il menù principale è sufficiente digitare "neurfuzz". Nei file con estensione "hlp" sono contenuti gli help in linea di Neurfuzz e, pertanto, non devono essere editati. Il file manual.doc contiene un manuale in inglese per l'utilizzo di Neurfuzz che approfondisce maggiormente alcuni aspetti delle interfacce fuzzy, rispetto al help in linea (manual.doc è visibile e stampabile attraverso Word per Windows). Nel file Neurfuzz.zip sono contenuti tutti gli altri files zippati con Pkzip.

GUIDA AL SOFTWARE

Esistono negli USA molti pacchetti software di simulazione di reti neurali su Pc che vanno dal livello didattico al massimo livello professionale. In genere i programmi ad alto livello sono specificamente orientati al "forecasting" (previsione) per applicazioni finanziarie o commerciali. Esistono programmi che permettono di creare reti neurali software relative ai principali paradigmi classici (error back propagation, Kohonen, lvq1, lvq2 ecc), mentre altri utilizzano paradigmi propri, talvolta anche più efficienti, ma meno adatti all'uso didattico. Molti di questi programmi permettono di generare codice in linguaggio c o c++, esattamente come il programma Neuronx contenuto nel dischetto: molti di questi programmi operano e possono generare codice per DOS ma esistono quasi in ogni caso le versioni per Windows. Alcune case produttrici di programmi per simulazione di reti neurali offrono anche parti hardware definite "acceleratori neurali" che altro non sono che schede da inserire sul bus del Pc che funzionano in parallelo ad esso come coprocessori. Queste schede sono basate principalmente su processori RISC come Intel 860 o DSP (digital signal processor) come il TMS320C25 e permettono di accelerare notevolmente i tempi di apprendimento, generalmente molto critici in applicazioni su problemi reali con reti error back propagation. Di seguito sono elencati alcuni dei pacchetti software più diffusi negli USA con una breve descrizione:

Brain Maker

produttore: California Scientific Software

10024 Newton Rd., Nevada City, Calif. 95959-9794,

(916) 4789040, (800)264-8112, fax(916)478-9041

è un sistema di sviluppo di reti neurali fornito in diverse versioni. La versione base permette di avere 512 inputs e otto strati di neuroni intermedi ed ha un costo di approssimativamente 195\$ con opzionale un programma di accelerazione dell' apprendimento fornito al prezzo di 150\$. La versione Brain Maker Professional 2.5 permette di avere 8192 inputs sfruttando la memoria estesa ed espansa. La versione 3.0 gira su Windows ed ha le stesse prestazioni della 2.5 per Dos più alcune migliorie nell' analisi dei dati e operazioni con dati ricorrenti. Per Brain Maker Professional esiste anche la opzione GTO (Genetic Training Option) che permette l' addestramento delle reti neurali con algoritmi genetici. A livello hardware sono disponibili due acceleratori su scheda da inserire su ISA-bus di Pc basati sul digital signal processor TMS 320c25 :il più semplice è costituito da un solo DSP e permette di operare a tre milioni di connessioni per secondo° gestito da Brain Maker ,mentre il più sofisticato viene gestito da Brain Maker Professional ed opera a 500 Mflops e 40 milioni di connessioni per secondo. Il prezzo di Brain Maker Professional 2.5 è circa \$750; il prezzo di Brain Maker professional 3.0 per Windows è circa \$795; il prezzo di Brain Maker + acceleratore è circa \$ 1995 il prezzo di Brain Maker professional + acceleratore va da \$9.750 (5 Mbyte ram) a 13.000 (32 Mbyte ram). Con più ram sulle schede acceleratrici si possono realizzare reti neurali di maggiori dimensioni; per esempio sul Brain Maker Professional accelerator si possono avere 600.000 connessioni con 5 Mbyte di ram contro i 3.6 milioni della versione con 32 Mbyte. La misura di velocità

delle schede acceleratrici viene spesso fornita in Mflops(milioni di operazioni in floating point per secondo) o in Mips(milioni di istruzioni per secondo),ma molto spesso anche in connessioni per secondo:questo dato sta a rappresentare il numero di operazioni che si riescono ad effettuare sulle connessioni della rete neurale nell'unità di tempo.

ExploreNet 3000

produttore:HNC 5501 Oberlin Dr.,San Diego,Calif.92121-1718,
(619)546-8877,fax (619) 452-6524

Si tratta di un programma che permette di realizzare 20 architetture di reti neurali senza programmazione (come quasi tutti i pacchetti software qui descritti).Il suo prezzo si aggira intorno a \$395.Si può comperare anche la versione completa di scheda acceleratrice a 80 Mflops che si chiama Balboa Developer s System e costa circa \$9.950.

NeuroShell

produttore:Ward Sytems Group,

Executive Park West,5 Hillcrest Dr.,Frederik,Md 21702 ,

(301) 662-7950 ,fax(301) 662-5666

Esiste nelle versioni 1 (per Dos) e 2(per Windows) e permette di generare codice C, Pascal, Fortran, Basic su diversi paradigmi di reti neurali come backpropagation, Kohonen, PNN(Probabilistic Neural Network), GRNN (General Regression Neural Network). Neuroshell1 costa \$195,mentre neuroshell2 costa \$495. La stessa casa fornisce inoltre NeuroWindows, un tool di programmazione con una Windows DLL che permette di creare fino a 128 reti

neurali interattive in una stessa applicazione al prezzo di \$369. Si può acquistare anche l'acceleratore neurale NeuroBoard 100 volte più veloce di un Pc basato su 386 a 20MHz. Sotto NeuroWindows possono girare reti neurali multiple su una NeuroBoard e fino a 10 NeuroBoard possono essere installate su un Pc.

DynaMind 4.0

produttore: NeuroDynamX

P.O.Box 323, Boulder, Colo.

80306, (303)442-3539,

(800)747-3531, fax(303)442-2854

Permette di generare reti neurali error back propagation con 16000 neuroni per layer e 32000 inputs e outputs. Permette anche la generazione di reti ricorrenti (sempre a retropropagazione dell'errore) e reti basate sull'algoritmo Madaline III. Funziona in ambiente Dos e il prezzo è di circa \$295. La stessa casa produce inoltre Dyna Mind Developer 4.0 che include anche una libreria di routines in C che possono essere "incapsulate" entro applicazioni custom. Dynamind Developer 4.0 può suddividere il problema complesso in piccole parti che possono essere collegate in serie e in parallelo. Il prezzo di Dynamind Developer è di \$1.295. NeuroDynamix fornisce anche gli acceleratori NDX Neural Accelerator XR25 e NDX Neural Accelerator XP50. Il primo è basato su un processore Risc Intel 860 XR che può operare a 22.5 milioni di connessioni per secondo e deve essere installato su uno slot per espansioni standard ISA di un Pc. Il secondo è basato su un processore RISC Intel 860 XP che permette di operare su 45 milioni di connessioni per secondo e come il primo deve essere installato su uno slot standard ISA. Il prezzo di NDX XR25 va da circa \$3.500 a circa \$10.400 a seconda della quantità di ram

installata. Il prezzo di NDX XP50 va da circa \$9.000 a circa \$15.000 a seconda della quantità di ram installata.

NeuroForecaster 2.1

produttore: NIBS Ptd.Ltd.,

62 Fowlie Rd., Republic of Singapore
1542, (65)3442357, fax(65)3442130.

Si tratta di un tool di utilizzo generale, particolarmente indicato per applicazioni di "forecast", che utilizza le tecnologie fuzzy e neurale. Permette la realizzazione di 12 modelli di reti neurali, compreso il tipo back propagation, radial basis function, Fast Prop, e NeuroFuzzy. Si tratta di un prodotto particolarmente orientato alle previsioni commerciali e finanziarie. Prezzo \$380.

NeuralWorks Professional II/Plus 4.10

produttore: NeuralWare,

Penn Center West, Bldg.IV, Pittsburgh, Pa. 15276, (412)787-8222, fax(412)787-8220

Distributore per l'Italia: UNIPLAN SOFTWARE S.R.L 84018 Scafati(SA)-Via G.Oberdan,52 telefono 081/8501197-8507171 telefax 081/8561979 Si tratta di un tool di sviluppo e addestramento di reti neurali che permette di utilizzare i seguenti paradigmi: Logicon Projection Network, Fuzzy Art Map, Radial Basis Functions, Quik Prop, General Regression Neural Network, Cascade Correlation e Probabilistic Neural Network. Prezzo: contattare distributore.

BIBLIOGRAFIA IN LINGUA INGLESE

T.Kohonen, "Associative memory: a system theoretic approach", Springer Verlag, Berlino 1977

B.Kosko: "Bidirectional Associative Memories", IEEE Transactions on Systems, Man and Cybernetics, 1987

R. Hecht-Nielsen: "Theory of Back Propagation Neural Networks" International Joint Conference on Neural Networks, 1989

S.Kirkpatrick, C.D.Gelatt, M.P.Vecchi: "Optimization by simulated annealing" in "Science", 220, 1983

T.Kohonen: "Self-Organizing formation of topologically correct feature map", Biological Cybernetics, 43, 1982

Timothy Masters, "Practical Neural Network Recipes in C++", Academic Press

Maureen Caudill, Charles Butler, "Understanding Neural Networks: Computer Explorations", "Vol.2 Advanced Networks". Cambridge, Mass.: MIT Press, 1992

e molti altri....

BIBLIOGRAFIA IN LINGUA ITALIANA

- **FONDAMENTI DI RETI NEURALI - Tarun Khanna - Addison-Wesley Editore**
- **RETI NEURALI ARTIFICIALI - A.Mazzetti - Apogeo Editore**
- **RETI NEURONALI - S.Cammarata - Etaslibri Editrice**
- **SISTEMI FUZZY - S.Cammarata - Etaslibri Editrice**
- **RETI NEURALI PER LE SCIENZE ECONOMICHE - G.Fabbri, R.Orsini - Franco Muzzio Editore**
- **IL FUZZY PENSIERO - B. Kosko - Baldini & Castoldi Editori**
- **L' OFFICINA NEURALE - G. Carrella - Franco Angeli Editore**
- **ASPETTI APPLICATIVI DELLE RETI NEURALI - A. Ferrari - F. Angeli Editore**
- **INTERVISTA SULLE RETI NEURALI - D. Parisi - Mulino Editrice**
- **IL CERVELLO COMPUTAZIONALE - P.S.Churchland, T.J.Sejnowski - Mulino Editrice**
- **SQUASHING THEORY - M. Buscema - Armando Editore**
- **RETI NEURALI AUTORIFLESSIVE - M.Buscema, G.Didonè, M.Pandin - Armando Editore**

PUBBLICAZIONI MENSILI /RIVISTE

**"AI Expert" 600 Harrison St. S.Francisco, California 94107
(415)905-2200 FAX:(415)905-2234 Si tratta di una ottima
rivista mensile di tipo divulgativo che tratta tutti gli argomenti
relativi all intelligenza artificiale ed è reperibile,in Italia,
anche in alcune edicole.**

**"Journal of Artificial Neural Networks" Ablex Publishing
Corp.,355 Chestnut Street,Norwood,NJ 07648. Rivista
disponibile solo in abbonamento,dedicata alle reti neurali.**

**"Neural Networks" Pergamon Press 660 White Plains Rd.
Tarrytown,NY 10591,USA
Si tratta della più autorevole rivista mondiale per gli addetti ai
lavori.**

Enti/Società

**International Neural Network Society,
1250,24th Street,N.W,Suite 300 Washington,Dc 20037,USA
Phone:202-466-4667 Fax:301-942-1619**

**SIREN(Società Italiana Reti Neuroniche)
via Pellegrino,19 - Vietri sul Mare(SA)
Tel. 089-761167**
