MANUALE di introduzione al BPN

(Back-Propagation Neural networks)

Brunella Antodaro

Rapporto Interno I.R.A. n. 217/96

RAPPORTO INTERNO

CONSIGLIO NAZIONALE DELLE RICERCHE

ISTITUTO DI RADIOASTRONOMIA

Via P. Gobetti, 101 - 40129 BOLOGNA (Italy)

Indice

Introduzione

Questo documento descrive il simulatore di reti neurali BPN (Back-Propagation Network)ed é una guida per la sua installazione il il suo utilizzo.

Il pacchetto BPN é stato sviluppato presso l'Istituto di Radioastronomia del CNR di Bologna sotto la supervisione del Dott.Mauro Nanni e in collaborazione con il Dott. Giuseppe Stanghellini.

L'obiettivo era quello di creare un simulatore efficiente, flessibile e generale per la ricerca e le applicazioni delle reti neurali.

Il simulatore è composto da tre moduli logici fondamentali e da una libreria in linguaggio C.: il kernel, un'interfaccia grafica X11 e un'interprete di comandi per la programmazione degli esperimenti.

Questo documento contiene

- accenni teorici sulle reti neurali back-propagation;
- una guida all'installazione e alla compilazione del pacchetto;
- una descrizione delle caratteristiche del pacchetto;
- una descrizione dell'interprete di comandi che interfaccia il simulatore;
- una descrizione dell'interfaccia grafica.

Ogni domanda, commento o bug report puó essere inviato al seguente indirizzo email:

antodaro@astbo1.bo.cnr.it

1. Reti neurali back-propagation

Le reti neurali propongono un approccio al problem-solving alternativo a quello classico, basandosi sul concetto di apprendimento di un compito.

L'espressione "rete neurale" indica un modello della rete neuronale biologico. Naturalmente, i modelli finora elaborati sono semplificate imitazioni delle reti naturali, sia per la conoscenza incompleta che si ha del sistema nervoso, sia per la difficoltá di modellizzazione della sua struttura e del suo funzionamento.

Un neurone biologico si compone di un corpo cellulare (soma) dal quale partono un certo numero di propaggini (dendridi), che raccolgono i segnali provenienti da altri neuroni e un filamento (assone) attraverso cui il neurone trasmette l'impulso elettrico da lui prodotto in seguito all'elaborazione dei segnali ricevuti.

L'assone si divide in migliaia di diramazioni alle cui estremitá le sinapsi trasformano l'attivitá dell'assone in fenomeni elettrici inibitori o eccitatori per l'attivitá dei neuroni riceventi.

L'apprendimento di un'esperienza avviene attraverso la modifica delle forze di connessione tra i neuroni.

Dato un problema, le reti neurali consentono di esplorare uno spazio di soluzioni potenziali molto piú ampio di quello che si puó esaminare con metodologie convenzionali, grazie alla loro capacità di generalizzazione e di adattamento agli stimoli esterni.

Al pari di un sistema naturale di neuroni, sono in grado di riconoscere configurazioni, riorganizzare dati e di apprendere un compito attraverso non piú la descrizione dei passi da compiere per poterlo realizzare, ma attraverso la descrizione degli esempi che rappresentano il mondo del problema e il conseguente adattamento della rete di connessioni dei neuroni all'esempio proposto.

Le reti sono costituite da un insieme di unitá di elaborazione dette neuroni artificiali, che comunicano tra loro attraverso segnali prodotti da ciascuno di essi e trasportati mediante delle connessioni: si tratta di parallele distributed processing (PDP).

Costruire una rete neurale artificiale addestrata alla risoluzione di un problema specifico significa definire:

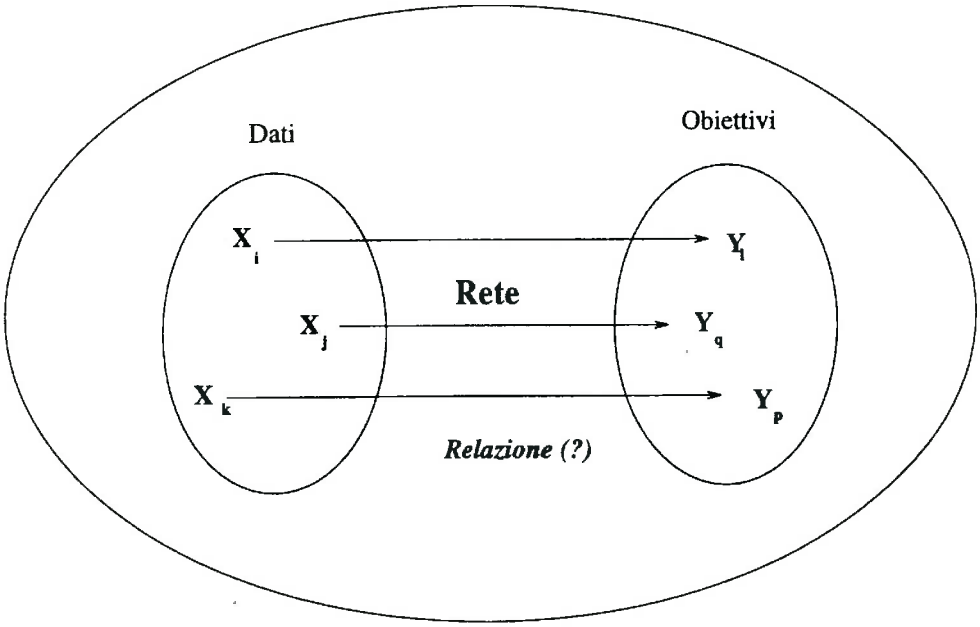
- un insieme di unità di elaborazione, dette neuroni;

- lo stato di attivazione di ciascuna unità, che determina il valore del segnale prodotto in output;
- il numero e la direzione delle connessioni tra le unità, rappresentate dai pesi;
- la funzione di attivazione che determina il nuovo valore di attivazione di ciascuna unità;
- un'unità detta Bias connessa a tutti i neuroni della rete che trasmette sempre un segnale pari ad 1;
- la regola di apprendimento del compito, che determina le modalità di modifica dei pesi di connessioni, in dipendenza dagli stimoli ricevuti in input;
- un insieme di esempi che copra il dominio del problema che si intende risolvere.

Il processo di elaborazione puó essere sintetizzato nel modo seguente: dato un problema per il quale si dispone di un insieme di esempi, ciascuno dei quali indica la risposta desiderata ad un certo impulso, si addestra la rete a risolvere l'insieme dei casi conosciuti, ripresentandoli alla rete piu' volte e modificando le connessioni allo scopo di minimizzare l'errore alla risposta ed individure cosi' la relazione che lega ogni coppia di valori.

Problema

Esempi

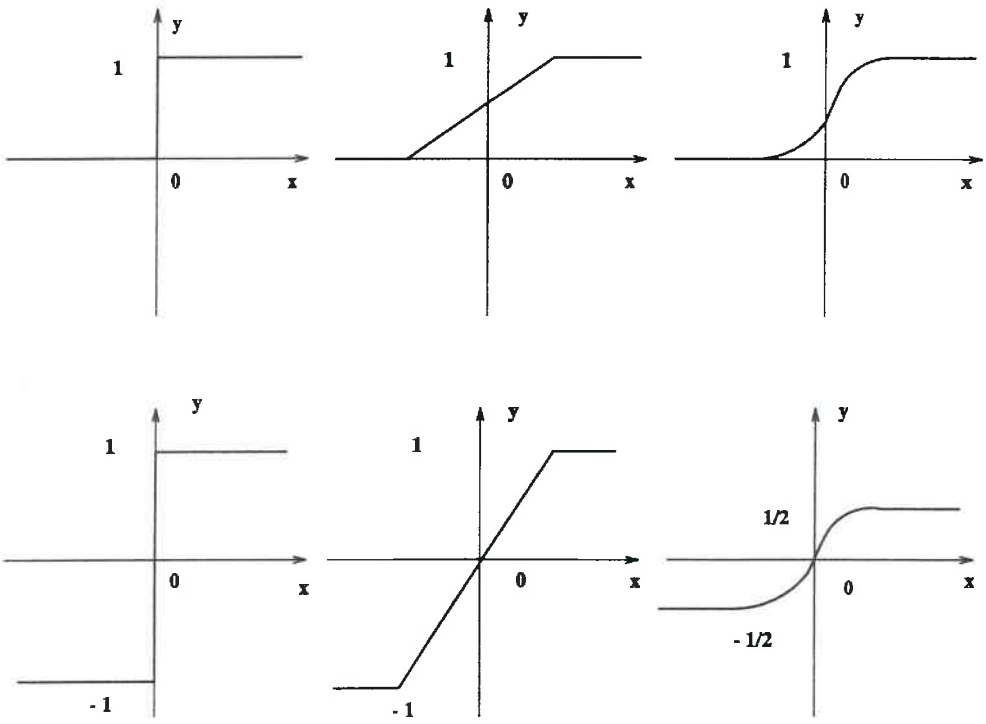


Dopo aver definito il problema e la struttura della rete, i pesi vengono inizializzati in modo casuale, per simulare la situazione di totale "ignoranza" di una rete priva di esperienza. A questo punto, ciascun neurone riceve dall'esterno o da altri neuroni un impulso (esempio), esegue una somma pesata su di essi, e trasforma tale valore applicando una funzione di attivazione.

Le più usate sono quelle mostrate in figura.

Il BPN implementa l'ultimo tipo parametrizzando l'ampiezza e la "forma" della curva (vedi Appendice).

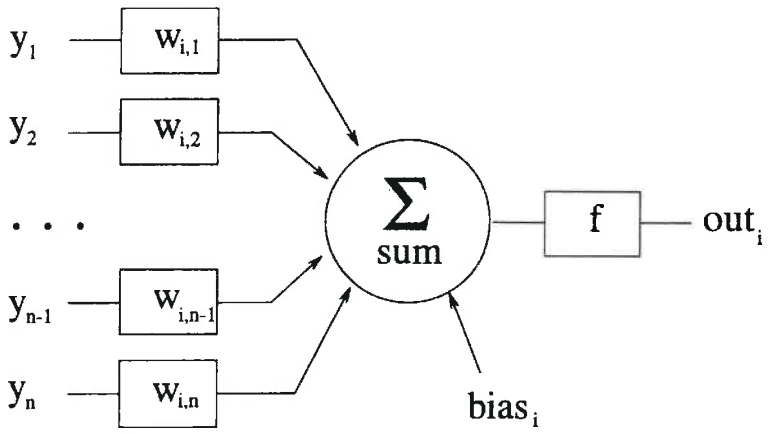
Funzioni di attivazione



Funzione a soglia logica **Hard-limit** **Funzione sigmoide**

Il risultato costituisce il segnale di uscita al neurone che, pesato con le rispettive forze di connessione, parteciperá alla elaborazione dei segnali dei neuroni riceventi l'impulso.

Neurone artificiale



W : peso di connessione del neurone i-esimo con i neuroni dello strato precedente

Y : segnale proveniente dallo strato precedente a quello di appartenenza del neurone i-esimo

bias : neurone che trasmette segnale pari ad 1 a tutti i neuroni di ogni strato tranne a quello di input

out : segnale prodotto dal neurone i-esimo

sum : somma pesata dei segali in ingresso al neurone i-esimo

f :funzione di attivazione

L'apprendimento si realizza nella modifica del sistema di trasporto dei segnali, conseguente all'esperienza e quindi in un adattamento del sistema agli stimoli esterni.

Nelle reti neurali back-propagation, tale adattamento avviene attraverso il confronto tra la risposta data e quella attesa e la minimizzazione della funzione di errore al variare delle forze (pesi) di connessione dei neuroni.

La rete "impara" cosi' a creare una rappresentazione interna del "mondo" descritto, modificando la propria configurazione, in funzione della risposta attesa ad un certo stimolo.

Una volta che la rete ha appreso a risolvere il problema sugli esempi forniti, essa è capace di risolvere i casi non previsti, che appartengono anche'essi al dominio del problema, ma sono completamente nuovi per la rete.

Quindi, la capacità di generalizzare diventa fondamentale per la risoluzione di problemi che non sono definibili perfettamente a priori, e del cui dominio non è possibile enumerare tutti i casi possibili.

Lo schema secondo cui i neuroni sono organizzati, il loro numero, le direzioni delle connessioni, i parametri della funzione di attivazione, il metodo di apprendimento sono profondamente legati al tipo di problema.

Costruire una rete significa definire la struttura e le caratteristiche del singolo neurone (funzione di attivazione), l'organizzazione della rete (numero di strati, numero di neuroni per ogni strato, connessioni, flusso di informazioni) e, infine, il metodo di apprendimento.

L'assenza di una teoria che guidi nella scelta della tipologia della rete é un limite, in quanto non è sempre facilmente individuabile dalle caratteristiche del problema da risolvere.

Grazie alla loro architettura parallela e distribuita, le potenzialitá di tali simulatori possono essere ben espresse su macchine parallele in molti campi che necessitano l'elaborazione di una grande mole di dati, quali la classificazione, il riconoscimento delle forme, il filtraggio di segnali, la visione artificiale, la robotica, la compressione di dati, ecc...

I vantaggi nel loro utilizzo si possono così sintetizzare:

- velocità elevata di elaborazione dopo l'apprendimento;
- affidabilità elevata (una rete addestrata ha un'alta tolleranza all'errore di elaborazione di una singola o di poche unità);
- gli algoritmi di apprendimento sono generali e coprono sufficientemente un'ampia classe di problemi;
- modellizzano perfettamente un computer parallelo (ciascun neurone diventa in tal caso un processore e le connessioni diventano i flussi di informazioni tra i vari processori);
- programmazione per esempi: superamento della programmazione algoritmica.

Gli svantaggi sono:

- metodo euristico per la ricerca della rete ottimale (trial & error);
- tempi di addestramento molto lunghi;
- applicazioni single purpose;
- assenza di una teoria solida.

Per una spiegazione teorica del funzionamento di una rete neurale back-propagation si rimanda all'Appendice.

2. Installazione e compilazione del BPN

L'intero pacchetto software BPN e' disponibile via "anonymous ftp" nel server FTP dell' Istituto di Radioastronomia del CNR di Bologna, all'indirizzo internet "terra.bo.cnr.it/BPN".

Gli utenti dell'Istituto possono disporre dell'eseguibile connettendosi al server terra e lanciando l'applicazione con il comando `bpn`. I sorgenti sono scritti nel linguaggio di programmazione C e sono disponibili sotto /usr/local/src/BPN.

Per la compilazione è necessario disporre di un compilatore C e dell' interfaccia grafica X11.

Il pacchetto BPN è stato concepito in modo tale da permettere anche l'aggiunta di nuovi comandi all'interprete e di routine grafiche di visualizzazione dei dati, per cui è necessaria la lettura del capitolo 6 e una minima conoscenza del C.

Essendo disponibile anche una libreria di comandi in C, nel caso in cui si voglia usare il simulatore delle reti neurali con interfaccia grafica X11 compilare eseguendo il comando `BPNbuild`, mentre per utilizzare in un programma in C le routine disponibili in libreria occorre compilare il programma con `BPNbuild_lib` avendo cura di aggiornare il file Imakefile: aggiungere, quindi, il nome del file sorgente (nomefile.c) e del file oggetto (nomefile.o) nelle rispettive liste SRCS e OBJS. (come esempio sono presenti nella versione originale prog_pr.c e prog_pr.o)

Lanciare l'applicazione con `bpn`.

Il programma é composto dai seguenti moduli:

- BPN_INTERP.C: implementa un Interprete di comandi;
- BPNM.C: implementa le procedure di simulazione di una rete neurale back-propagation;
- BPN_IO.C: contiene le procedure di input/output alla rete;
- BPN_INT.C: contiene le procedure per l' interfaccia X11 generica del pacchetto;
- BPN_X11_INT.C: contiene le procedure di creazione e distruzione di finestre X11;
- BPN_USER.C: questo modulo contiene le procedure per la manipolazione dei dati e le procedure utente;

- BPN_LIB.C: contiene la libreria di funzioni BPN;
- PROG_PR.C: esempio di utilizzo di alcune funzioni di libreria.

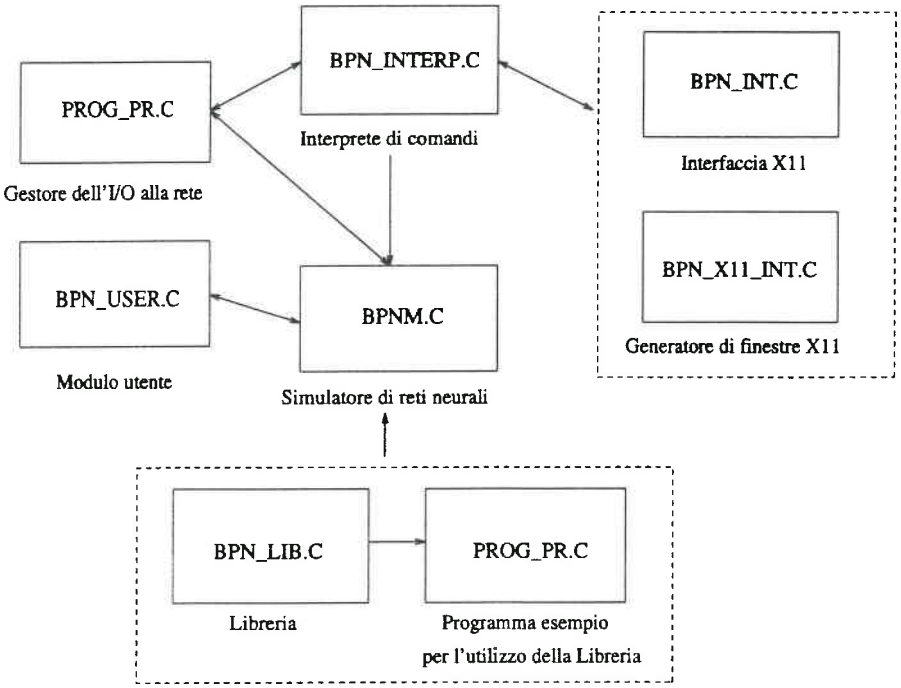


Diagramma a blocchi del BPN

3. BPN

Il programma BPN permette di simulare il comportamento di una rete neurale back-propagation e fornisce un interprete di comandi e un'interfaccia grafica che hanno lo scopo di facilitare la manipolazione e l'interpretazione dei dati in input ed in output alla rete.

Questo pacchetto nasce dall'esigenza di disporre di uno strumento che permetta di essere modificato agevolmente a seconda del problema che si vuole risolvere e soprattutto di essere interfacciato con pacchetti preesistenti.

Il simulatore costituisce il kernel del pacchetto e la definizione delle caratteristiche della rete (nr. neuroni, nr. strati, tipo di apprendimento, file di esempi, ecc..) viene realizzata tramite comandi dell'interprete, con assegnazioni dei valori desiderati per ciascun parametro o scrivendo un programma nel linuaggio dell'interprete ed eseguendolo dall'interno dell'ambiente stesso (vedi ESEMPI).

Le caratteristiche principali dell'interprete sono:

- È possibile aggiungere dei comandi al linguaggio dell'interprete, allo scopo di ampliarlo e permettere una maggiore flessibiltà nel suo utilizzo. Si possono, quindi, creare nuovi comandi la cui sintassi viene definita dall'utente.
- L'interprete dispone di comandi per la definizione, modifica e visualizzazione del contenuto di nuove variabili di vario tipo.

  Molte delle funzioni già disponibili sono state implementate allo scopo di mostrare tale caratteristica dell'ambiente di sviluppo di questo pacchetto.

  Pertanto, a scopo puramente dimostrativo, é stato realizzato l'insieme dei comandi per la creazione di finestre in iterfaccia X (vedi modulo BPN_XT_INT.C)
- Altra caratteristica importante, a cui si è già fatto cenno, è la possibilità di programmare nel linguaggio dell'interprete l'esecuzione di un processo (vedi ESEMPI pag. 12), nel senso che la definizione di una rete, la fase di addestramento, la visualizzazione dei dati, la stampa di un report dei valori delle variabili di cui si desidera la "storia", il test della rete e il suo utilizzo per un'applicazione successiva possono essere eseguite da linea di comando o scrivendo i relativi comandi in un file nella successione cronologica desiderata.

L'interfaccia grafica è costituita da una finestra dedicata ai comandi dell'interprete, una finestra per l'editing di file di testo, una finestra per la definizione di una

rete neurale e una finestra che visualizza l'andamento dell'errore nella fase di apprendimento.

Inoltre, il BPN contiene una libreria di funzioni in C.

Alle funzioni già disponibili ne possono essere aggiunte altre (vedi capitolo 6).

Per la compilazione di un programma in C in cui si fa uso delle funzioni di libreria fare riferimento il capitolo 2.

3.1 Interprete dei comandi

Al lancio dell'applicazione, viene aperta la finestra COMMANDWIN, in cui possono essere editati i comandi dell'interprete.

Lo standard output viene visualizzato sulla finestra da cui viene lanciata l'applicazione, in modo tale da tenere separati i comandi dai loro effetti sullo standard output.

L'interprete ha una sintassi molto simile a quella del linguaggio BASIC e nella versione originale fornisce comandi per la definizione e la visualizzazione dei valori delle variabili utente (la lista di tali variabili si ottiene con il comando **help v**), i costrutti di ciclo **for** e **while** e di controllo dell'**if**, l'esecuzione di task e dei comandi dell'ambiente UNIX e i comandi fondamentali per l'addestramento e l'utilizzo di una rete neurale back-propagation.

Lista dei comandi dell'interprete:

```
help
print
input
if
for
while
defsub
defstr
defchar
defuchar
defint
defuint
defflt
defdbl
show
sh
@
$
end
```

- **help** [i/f/u/v [command]]
  Le opzioni `i, f, u, v` listano rispettivamente i comandi interprete, le funzioni matematiche e i comandi utente disponibili. L'ultima opzione lista le varibili a cui si ha accesso da linea di comando.

Se alla opzioni `i` e `u` si fa seguire il nome del comando, viene listata la sintassi e la descrizione talvolta con esempio di quest'ultimo.

- **input** *var value*

  Esegue il comando di input del BASIC sulla variabile *var*.

- **print** *var [> filename][>> filename]*

  Stampa sullo standard output il valore della variabile *var*.

  L' output può essere ridirezionato con > o concatenato >> su un file di nome *filename*.

- **if** *expression* **then** *actions block*

  [**else** *actions block*] **endif**

  ESEMPIO:

  if a==5 then print "ciao" : endif

  Oppure

  scrivere in uno script quanto segue ed eseguirlo con *@script* (vedi pag. 13).

```
IF(i!=0)
    THEN print "giallo"
ELSE print "rosso"
     print "blu" : print "verde"
ENDIF
```

  È possibile scrivere i comandi del blocco sulla stessa riga con il separatore ":"

- **for** *variable* **to** *n actions block* **next**

- **while** *expression actions block* **wend**

  ESEMPIO (`FOR, WHILE`)

  Il **for** e il **while** non possono essere usate da linea di comando, ma solo all'interno di uno script eseguito poi con *@script*.

```
WHILE( i < 5 )
    FOR j = 1 TO 2
          sh j
    NEXT
    sh i
```

```
    i = i + 1
WEND
```

- **defchar** *var*

  Definisce e alloca una variabile utente di tipo carattere.

- **defuchar** *var*

  Definisce e alloca una variabile utente di tipo carattere senza segno.

- **defstr** *var*

  Definisce e alloca una variabile utente di tipo stringa.

- **defint** *var*

  Definisce e alloca una variabile utente di tipo intero.

- **defuint** *var*

  Definisce e alloca una variabile utente di tipo intero senza segno.

- **defsub** *namesub blocco* endsub

  Definisce una subroutine.

- **deffl** *var*

  Definisce e alloca una variabile di tipo float.

- **defdbl** *var*

  Definisce e alloca una variabile di tipo double.

- **show** [*string*][> *filename*][>> *filename*]

  Visualizza sullo standard output il valore delle variabili utente. Se non viene specificato il parametro *string* viene visualizzata la lista di tutte le varibili.

  É possibile usare il carattere jolly "*" e ridirezionare l'output con > per creare un file non ancora esistente e con >> per aggiornare con accodamento un file giá esistente *filename*.

  Termina l'applicazione.

- **$***command*

  Esegue il comando *command* di sistema.

  ESEMPIO

  $ls -l

- **@***filename*

  Esegue il programma in *filename*

- **end**

Termina l'applicazione.

Lista delle funzioni matematiche:

Le funzioni matematiche disponibili corrispondono nella sintassi e nella sematica alle corrispondenti funzioni del linguaggio di programmazione C.

- **sin(x)**

  calcola il seno, misurato in radianti.

- **sinh(x)**

  calcola il seno iperbolico di x (equivale a (exp(x) - exp(-x))/2)).

- **asin(x)**

  calcola i principali valori dell' arc sine di x, nell'intervallo [-pi/2,pi/2] radianti. Il valore di x deve essere nel dominio [-1,1].

- **cos(x)**

  calcola il coseno, misurato in radianti.

- **cosh(x)**

  calcola il coseno iperbolico di x (equivale a (e**x +e**(-x))/2)).

- **acos(x)**

  calcola i principali valori dell' arc cosine di x, nell'intervallo [0,pi] radianti. Il valore di x deve essere nel dominio [-1,1].

- **tan(x)**

  calcola la tangente, misurata in radianti.

- **tanh(x)**

  calcola la tangente iperbolico di x. (equivale a (e**x - e**(-x))/(e**x + e**(-x))).

- **atan(x)**

  calcola i principali valori dell' arco tangente di x, nell'intervallo [-pi/2,pi/2] radianti.

- **atan2(y,x)**

  calcola i principali dell'arco tangente di y/x nell'intervallo [-180,180]. Il segno e' determinato dal segno di y.

- **log10(x)**
  calcola il logaritmo in base 10 di x.
- **log(x)**
  calcola il logaritmo naturale (base e) di x.
- **exp(x)**
  calcola la funzione esponenziale di x.
- **sqrt(x)**
  calcola la radice quadrata di x.
- **pow(x,y)**
  eleva x all'esponente y. Equivale a e**(y ln(x)).
- **abs(x)**
  restituisce il valore assoluto di x intero.
- **x mod y**
  restituisce il resto della divisione di x per y.
- **int(x)**
  restituisce la parte intera di x

Caratteri speciali:

- **"i"**

  E' possibile eseguire una macro-espansione all'interno delle variabili stringa.

  Per esempio, data la variabile **i**, per comporre delle **stringhe** al cui interno si vuole venga inserito il valore corrente di **i**, eseguire uno script del tipo:

```
defint i
for i= 1 to 10
   save_net rete"i".net
next
```

  Eseguendo un **$ls rete*.net**, si puo' verificare che il ciclo di **for** ha salvato la configurazione della rete nei file **rete1.net rete2.net ... rete10.net**.

- *****

  Questo carattere, oltre ad essere un operatore matematico, se usato all'inizio di riga in un file che si intende eseguire con @, commenta tutta la riga.

- **=**

  Questo carattere, oltre ad essere un operatore logico, è anche operatore di assegnazione, per cui data una variabile, l'operazione di assegnazione diventa semplicemente:

  N_slab = 3

Table 0.1: Operatori

Operatore	Descrizione
**	operatore unario di elevamento a potenza
^	operatore unario di elevamento a potenza
=	operatore binario di uguaglianza
==	operatore binario di uguaglianza
:	separatore di istruzioni
+	operatore binario additivo
−	operatore binario di sottrazione
*	operatore binario moltiplicativo
/	operatore binario di divisione
>	maggiore: operatore bin. di confronto nelle espressioni o di ridirezione su file
>>	ridirezione con accodamento su file
<	minore: operatore bin. di confronto nelle espressioni
>=	maggiore uguale: operatore bin. di confronto nelle espressioni
<=	minore uguale: operatore bin. di confronto nelle espressioni
not	operatore logico unario di negazione
!	operatore logico unario di negazione
!=	non uguale: operatore bin. di confronto nelle espressioni
<>	non uguale: operatore bin. di confronto nelle espressioni
><	non uguale: operatore bin. di confronto nelle espressioni
and	operatore logico binario di and
&&	operatore logico binario di and
or	operatore logico binario di and
|	operatore logico binario di or
+ =	operatore di assegnazione con addizione
− =	operatore di assegnazione con sottrazione
* =	operatore di assegnazione con moltiplicazione

3.2. Simulatore

Il simulatore implementa una rete neurale offrendo quattro versioni dell'algoritmo di apprendimento back-propagation (vedi APPENDICE):

- Normal
- Smoothing
- Batching
- with Momentum.

Le caratteristiche generali di una rete neurale Back Propagation sono:

- Funzione di attivazione del neurone sigmoidea (vedi APPENDICE).
- Organizzazione dei neuroni stratificata.
- Connessione in avanti di ogni strato con il successivo, partendo dallo strato di input fino a quello di output.
- Flusso di informazioni con stessa direzione delle connessioni.
- Direzione di modifica delle forze di connessione inversa a quella di queste ultime.
- Algoritmo di apprendimento back-propagation (vedi APPENDICE).

Lista dei comandi del simulatore

- **edit_p** *fileconf*

  Apre una finestra editor su un file (template) in cui sono presenti tutti i parametri di configurazione di una rete neurale.

  Deve precedere l'uso di questo comando l'assegnazione del valore 1 (es. Normal = 1) ad una delle variabili per la scelta della modalità di apprendimento.

- **init_net**

  Inizializza i pesi di connessione con valori casuali tra 0 e 1.

- **del_net**

  Dealloca la rete im memoria e pone a 0 tutti i parametri di rete.

- **set_bound** *filelearn*

  Legge dal file di addestramento specificato da *filelearn* l'header contenente i domini dei valori che descrivono gli esempi, per permettere la normalizzazione.

- **learning** *C numiter*

  Esegue *numiter* iterazioni di apprendimento con il tipo di apprendimento scelto. C puo' assumere il valore N, S, B o M (N = Normal, S = Smoothing; B = Batching; M = with Momentum).

- **save_net** *filename*

  Salva tutti i parametri e i valori dei pesi della rete nel file *filename*.

  Questo comando è necessario dopo la fase di apprendimento, per salvare la configurazione "addestrata" della rete e poter rendere possibile il suo utilizzo.

- **save_subnet** *filename start_layer end_layer*

  Salva una sottorete della rete allocata nel file *filename* dallo strato *start_layer* allo strato *end_layer* di quest'ultima.

- **get_net** *filename*

  Alloca la rete memorizzata nel file *filename*.

- **query_net** *yes / no*

  Questo comando apre una finestra da cui è possibile interrogare la rete dopo l'addestramento su un singolo valore.

  A seconda che si conosca o meno la risposta corretta, si sceglie l'opzione yes o no. Il primo caso è dato per valutare la precisione della rete, in quanto viene fornito il valore dell'errore commesso rispetto alla risposta desiderata. Nella finestra compaiono tanti input da inizializzare, quanti sono i neuroni di dello strato di input e tanti output quanti sono i neuroni dello strato di output (questo solo nel caso di scelta dell'opzione yes).

  La valutazione avviene in tale modo: si addestra la rete su un insieme di esempi del problema e si conserva un certo numero di esempi noti che non partecipano all'apprendimento, ma che possono fornire una misura del grado di generalizzazione raggiunto dalla rete.

  Terminato l'apprendimento si usano tali esempi, per calcolare l'errore commesso dalla rete.

- **close_testwin**

  Tale comando distrugge la finestra creata dal comando **query_net**.

- **test_net** *filename fileout fileerr*

  Questo comando è equivalente al **query_net** solo che processa un insieme di valori contenuti in *filename*, scrive i risultati in *fileout* e se nel *filename* accanto alla colonna dei valori c'e' anche quella dei valori desiderati.

  Nel *filename* di test non è necessario alcun header. Il formato vuole solo per ciascuna riga un valore o una coppia valore-risultato.

- **end**

  Termina l'applicazione.

3.3. Interfaccia grafica.

L'interfaccia grafica si compone di due gruppi di finestre.

Al primo appartengono:

- finestra dei comandi;
- finestra editor;
- finestra per la configurazione di una rete;
- finestra dell'errore dell'apprendimento.

Al lancio dell'applicazione viene creata la finestra dei comandi COMMANDWIN.

In essa vanno digitati i comandi dell'interprete.

Con il comando **edit** *filename* viene creata una finestra che utilizza l'editore EMACS per la scrittura di un testo.

Con **close_edit** viene chiusa la finestra e salvato il suo contenuto nel file specificato con il comando **edit**.

La finestra per la configurazione di una nuova rete viene aperta dal comando **edit_p** *filename*.

Precedentmente si deve aver assegnato 1 ad una delle variabili che permettono la scelta della modalità di apprendimento.

Tali variabili sono Normal, Batching, Smoothing e Momentum.

A seconda della scelta, l'editore verrà aperto sul template relativo in cui sono presenti tutte le variabili da inizializzare per definire la nuova rete.

Con **close_edit_p** viene chiusa la finestra e salvato il suo contenuto nel file specificato con il comando **edit_p**.

In tal modo è pronto il file contenente la struttura della rete e con il comando **@filename**, seguito da **init_net**, verrà creata la nuova rete.

Se durante l'apprendimento si volesse seguire l'andamento dell'errore che descrive l'"evoluzione" della rete, prima di lanciare il comando **learning**, eseguire il comando **open_errwin**.

Compare una finestra con un grafico su cui l'asse delle X indica il numero dell'iterazione e sull'asse delle Y, una volta lanciato il learning, compare l'errore scalato nel range specifico del problema.

Per chiudere questa finestra usare il comando **close_errwin**.

Al secondo gruppo di finestre appartengono le finestre "generabili" da comandi utente.

In BPN_X11_INT.C c'é qualche esempio di comandi che creano e distruggono delle finestre.

Tale modulo viene fornito unicamente allo scopo di mostrare l'estendibilitá dell'interprete.

I comandi implementati in questo modulo sono:

- **create_win** *numberwin name x_init y_init x_width y_height*

  Possono essere create in un'applicazione al massimo 10 finestre. All'atto di creazione bisogna specificare il numero della finestra (tale numero servirà per identificare la finestra da distruggere con il comando **destroy_win**), il nome che comparirà sulla barra superiore della finestra, le coordinate dello spigolo superiore sinistro, l'altezza e l'ampiezza desiderate.

- **destroy_win** *numberwin*

  Chiude la finestra il cui numero è *numberwin*.

- **write_str** *numberwin x y string*

  Scrive una stringa nella finestra *numberwin* partendo dalle coordinate *x y* specificate.

4. Libreria

La libreria contiene funzioni richiamabili in un programma in C.

- **at_BPN:**
  PROTOTIPO
  `void at_BPN(char *filename);`
  Esegue il programma scritto nella sintassi del BPN, contenuto in `file_name`, evitando i comandi che richiamano le funzioni X11.

- **bound_BPN:**
  PROTOTIPO
  `void bound_BPN(char *filename);`
  Carica i valori *max* e *min* contenuti nel file di addestramento necessari alla normalizzazione dei dati prima che siano processati dalla rete.

- **init_BPN:**
  PROTOTIPO
  `void init_BPN();`
  Reinizializza in modo casuale i pesi della rete caricata in memoria.

- **load_BPN:**
  PROTOTIPO
  `void load_BPN(char *filename);`
  Carica il file contenente i parametri di struttura della rete neurale e i valori dei pesi.

- **fprocess_BPN:**
  PROTOTIPO
  `void fprocess_BPN(char *fileinp, char *fileout, char *fileer );`
  Processa con la rete neurale caricata da `load_BPN` i dati contenuti in *fileinp*, restituisce il risultato dell'elaborazione in *fileout*, e se desiderato il file di errori, nel caso in cui si disponga, dei valori desiderati relativi al problema da risolvere.
  Nel caso contrario si ricordi di porre a NULL il puntatore al file *fileer*.

- **process_BPN:**

  PROTOTIPO

  ```
  double* process_BPN(double* pattern);
  ```

  Processa con la rete neurale caricata da `load_BPN` il pattern specificato, la cui dimensione deve essere uguale a quella dello strato di input della rete caricata con `load_BPN`.

- **save_BPN:**

  PROTOTIPO

  ```
  void save_BPN(char *s);
  ```

  Salva la rete in memoria su un altro file di cui si deve specificare il nome.

- **save_sub_BPN:**

  PROTOTIPO

  ```
  void save_sub_BPN(char *filename, char *inp_layer, char *out_layer);
  ```

  Salva una sotto-rete della rete in memoria su un altro file di cui bisogna specificare il nome. I parametri *inp_layer* e *out_layer* servono ad indicare da quale strato e fino a quale strato si vuole che venga salvata la rete.

- **shell_BPN:**

  PROTOTIPO

  ```
  void shell_BPN(char *s);
  ```

  Processa un comando di shell da programma.

  ES:

  shell_BPN("rm filename");

5. Esempi

Supponiamo di voler addestrare una rete a simulare la funzione identica su interi tra 0 e 10.

È necessaria prima una descrizione del problema mediante esempi.

La funzione identica è descritta da coppie di valori tali che ad ogni x corrisponde se stesso.

Con un editore qualsiasi scrivere un file del tipo:

```
#### Domini della funzione identica #######
0 0
10 10
##########################################
0 0
1 1
2 2
3 3
4 4
5 5
6 6
7 7
8 8
9 9
10 10
```

La prima riga è di commento e segna l'inizio dell'header, in cui viene specificato il valore minimo (0) e il valore massimo (10) per ogni colonna.

Nella prima colonna compaiono i valori di x a cui corrispondono i valori della seconda colonna.

Quindi, nell'ordine, viene descritto prima l'input e poi l'output desiderato.

Il numero di colonne per la rappresentazione dell'input e dell'output è uguale alla dimensione dello spazio della funzione.

Nell'esempio la dimensione dell'insieme di definizione e dell'insieme di valori è uguale ad 1.

Notare che tali dimensioni sono rispettivamente quelle dello strato di input e dello strato di output della rete neurale da addestrare successivamente.l

Supponiamo di aver salvato il file con il nome **esempi**.

Lanciare `bpn`.

Compare la finestra COMMANDWIN, in cui vanno editati i comandi dell' interprete.

L'editor di questa finestra offre la possibilità di usare un sottoinsieme di funzioni dell' EMACS.

A questo punto bisogna iniziare la fase di addestramento.

C'è la possibilità di scegliere la modalita' di apprendimento assegnando 1 alla variabile Normal per un apprendimento classico, o alla variabile Batching per un aggiornamento per ogni sottoinsieme degli esempi, o alla variabile Smoothing o ancora alla variabile Momentum.

Digitare `edit_p` *filename* e premere il tasto "invio".

Si aprirà una finestra EDIT su un template in cui ci saranno tutti i parametri da impostare per poter costruire una rete neurale BPN con la modalità scelta.

Notare che il numero massimo di strati della rete è pari a 10 e che la dimensione massima di ciascuno strato è pari a 100 neuroni.

Per il problema di esempio, basta una rete di tre strati: la dimensione del primo strato e dell'ultimo è 1 perchè la funzione identica è definita su uno spazio monodimensionale con valori in uno spazio sempre monodimensionale.

La dimensione dello strato intermedio la si sceglie, ad esempio di 100.

I template sono salvati rispettivamente nei file templateN, templateS, templateB, templateM.

Per operare la scelta assegnare 1, da linea di comando dalla finestra COMMANDWIN, alla variabile Normal o Smoothing o Batching o ancora Momentum.

Per un apprendimento **Normal** il template risulta essere il seguente:

```
*            STRUCTURE Network

*  number of layers
      N_slab = 0

*  number of neurons per layer
      N_neuron = {0,0,0,0,0,0,0,0,0,0}

*            LEARNING TYPE: NORMAL

*            PARAMETERS OF THE Learning Function

*  learning rate (typical values 0.1...1.0)
      Alpha = 0.0


*  activation function parameters (typical value 0..1.0)
      P1 = 0.0
      P2 = 0.0
      P3 = 0.0

*  flag for  remixing of  training set
      Mix = 0

*            FILES

*  file of learning
      FileTrain =""
```

Per un apprendimento **Smoothing**, il template risulta essere il seguente:

```
*            STRUCTURE Network

*  number of layers
      N_slab = 0

*  number of neurons per layer
      N_neuron = {0,0,0,0,0,0,0,0,0,0}

*            LEARNING TYPE: SMOOTHING

*            PARAMETERS OF THE Learning Function

*  learning rate (typical values 0.1...1.0)
      Alpha = 0.0

*  amount of weigth variation (values = 1.0 - Mu)
      Beta = 0.0


*  amount of old weight change (typical values 0...1.0)
      Mu = 0.0

*  activation function parameters (typical values 0... 1.0)
      P1 = 0.0
      P2 = 0.0
      P3 = 0.0

*  flag for  remixing of  training set
      Mix = 0

*            FILES

*  file of learning
      FileTrain =""
```

Per un apprendimento **Batching**, il template risulta essere il seguente:

```
*            STRUCTURE Network

*  number of layers
     N_slab = 0

*  number of neurons per layer
     N_neuron = {0,0,0,0,0,0,0,0,0,0}

*           LEARNING TYPE: BATCHING

*  PARAMETERS OF  Learning Function

*  learning rate (typical values 0.1...1.0)
     Alpha = 0.0

*  batching size

     Batch = 0

*  activation function parameters (typical value 0...1.0)
     P1 = 0.0
     P2 = 0.0
     P3 = 0.0

*  flag for  remixing of  training set
     Mix = 0

*            FILES

*  file of learning
     FileTrain =""
```

Per un apprendimento con **Momentum**, il template risulta essere il seguente:

```
*            STRUCTURE Network

*  number of layers
     N_slab = 0

*  number of neurons per layer
     N_neuron = {0,0,0,0,0,0,0,0,0,0}

*           LEARNING TYPE: WITH MOMENTUM

*            PARAMETERS OF THE Learning Function

*  learning rate (typical values 0.1...1.0)
     Alpha = 0.0

*  amount of old variation weight (typical values 0.1...1.0)
     Eta = 0.0


*  flat spot elimination value (typical values 0...0.25)
     C = 0.0

*  amount of old weight change (typical values 0...1.0)
     Mu = 0.0

*  activation function parameter (typical values 0...1.0)
     P1 = 0.0
     P2 = 0.0
     P3 = 0.0

*  flag for  remixing of  training set
     Mix = 0

*            FILES

*  file of learning
     FileTrain =""
```

Supponiamo di aver scelto la modalità Smoothing:

```
*            STRUCTURE Network

*  number of layers
     N_slab = 3

*  number of neurons per layer
     N_neuron = {1,100,1,0,0,0,0,0,0,0}

*            LEARNING TYPE: SMOOTHING

*            PARAMETERS OF THE Learning Function

*  learning rate (typical values 0.1...1.0)
     Alpha = 0.5

*  amount of weigth variation (values = 1.0 - Mu)
     Beta = 0.7


*  amount of old weight change (typical values 0...1.0)
     Mu = 0.3

*  activation function parameters (typical values 0... 1.0)
     P1 = 1.0
     P2 = 1.0
     P3 = 0.0

*  flag for  remixing of  training set
     Mix = 1

*            FILES

*  file of learning
     FileTrain ="esempi"
```

Con `close_edit_p` viene salvato il file *filename* con la configurazione definita.

Con **`@`**`filename` le variabili di rete vengono inizializzate.

Con `init_net` viene realizzato un check sulla consistenza dei dati di struttura della rete e inizializzati i pesi di connessione.

Deve seguire il comando **set_bound** *nomefile* per la lettura dal file di addestramento dei limiti del dominio di ogni dato ($min_i <= xi <= max_i$), perché venga fatta una normalizzazione tra 0 e 1 di ogni valore.

Il file di addestramento deve avere un header cosi' fatto, come mostrato nel file esempi:

######### Dominio del problema X #######

min_1 min_2 ... min_n $dmin_1$ $dmin_2$ ... $dmin_m$

max_1 max_2 ... max_n $dmax_1$ $dmax_2$ ... $dmax_m$

##############################

min_i e max_i rappresentano i limiti per colonne della caratteristica i_{esima} dell'esempio.

n è la dimensione dello strato di input.

$dmin_i$ e $dmax_i$ rappresentano i limiti per colonne della caratteristica i_{esima} del campione desiderato

m la dimensione dello strato di output.

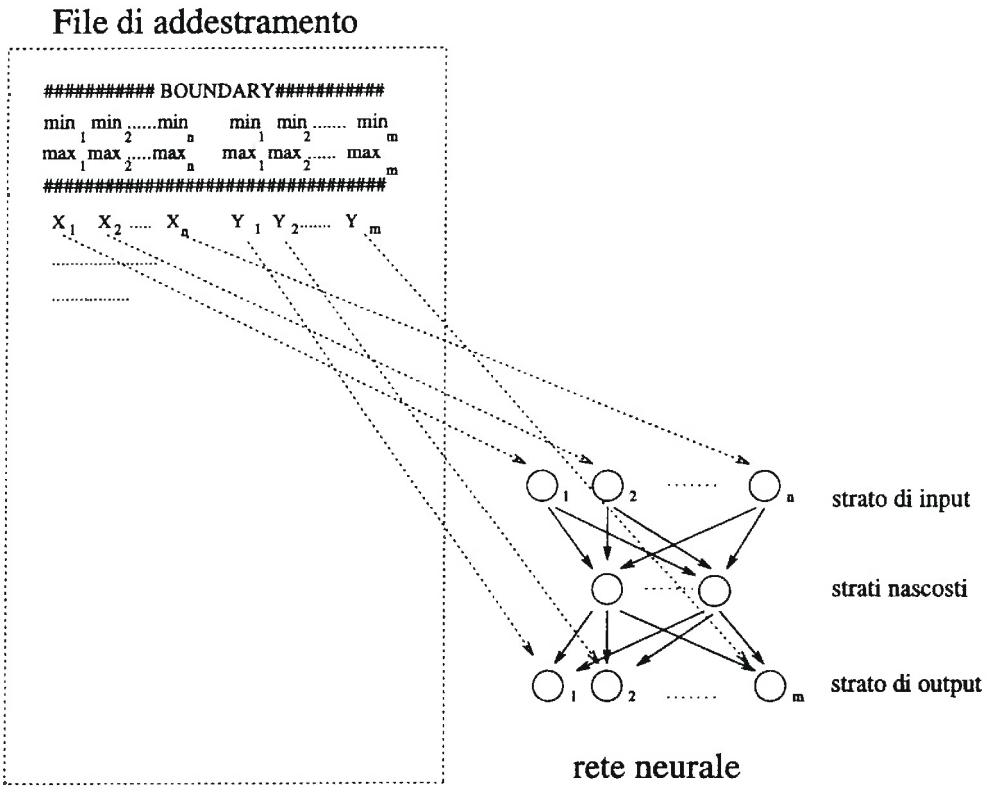
Seguono i dati nei seguenti formato e ordine per riga

x_1 x_2 ... x_n y_1 y_2 ... y_m

dove

x_1 ... x_n rappresentano l'input alla rete

y_1 ... y_m rappresentano l'output desiderato



$X_1 X_2 \ldots X_n$: input alla rete

$Y_1 Y_2 \ldots Y_m$: output desiderato all'input X

min_i : valore minimo del campo di esistenza del valore della colonna corrispondente

max_i : valore massimo del campo di esistenza del valore della corrispondente

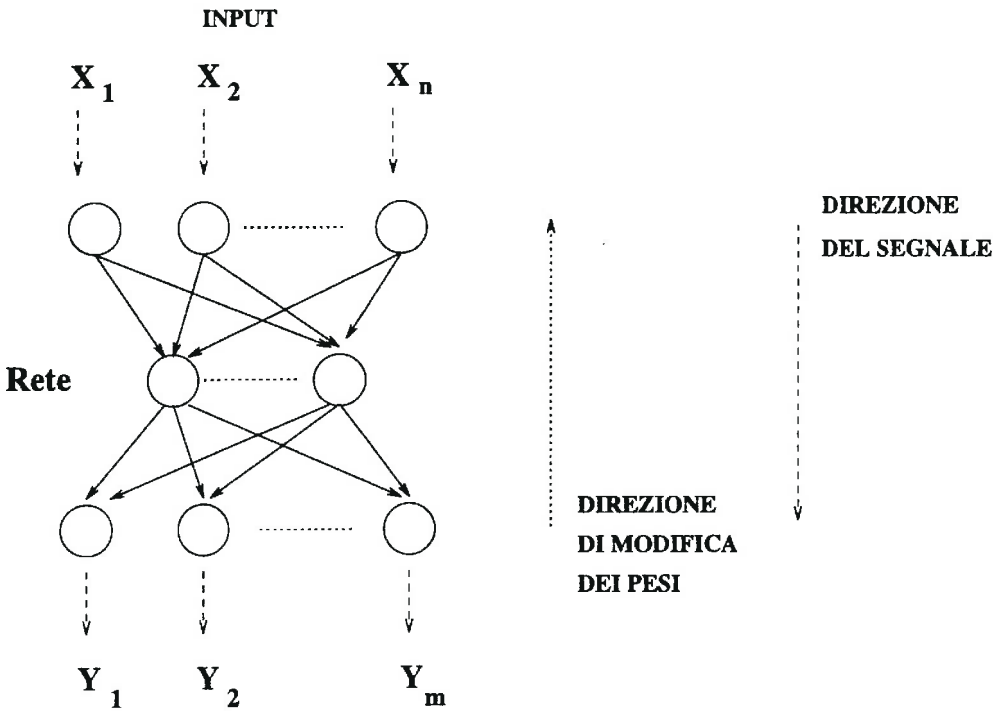
A questo punto, lanciare il comando `learning` *C num*, dove *C* è il carattere che specifica il tipo di apprendimento desiderato e può assumere il valore N per Normal, B per Batching, S per Smoothing, M per Momentum e *num* e' il numero di esempi che si vuole vengano proposti alla rete per la fase di apprendimento.

Lo schema logico di ciò che avviene in fase di apprendimento è mostrato dalla figura che segue:

Addestramento

$$(X_1 X_2 \ldots X_n) \dashrightarrow (Y_1 Y_2 \ldots Y_m)$$

Esempio



Nel caso in cui la variabile `Mix` venga posta ad 1, ogni qualvolta termina un ciclo di presentazione degli esempi contenuti nel file di addestramento, il loro ordine viene cambiato, in modo che la "storia" dell'apprendimento non dipenda da esso.

Se si desidera la visualizzazione dell'errore che descrive l'evoluzione dell'apprendimento, eseguire il comando `open_errwin`, prima del learning.

Ogni 5000 iterazioni, sullo standard output viene stampato il valore dell'errore commesso dalla rete sull'esempio corrente.

L'apprendimento può essere sospeso cliccando sul bottone STOP LEARNING, nel caso la convergenza del processo di apprendimento venga ottenuta prima del termine delle iterazioni specificate col parametro *num* o si voglia impostare diversamente una variabile di rete visibile dall'esterno.

Terminata la fase di apprendimento, eseguire la fase di test sul file di apprendimento, con il comando

test_net *inpfile outfile [errfile].*

A questo punto è possibile, ad esempio:

- processare un solo pattern chiamando la **query_net**;

- salvare la rete "istruita" chiamando la **save_net** *netname*;

- salvare una sottorete con **save_subnet** *netname inp_layer out_layer*

- vedere il suo contenuto e il formato: **$more** *netname.*

Nell'header del file di rete ci sono tutte le informazioni relative alla struttura della rete e i parametri di apprendimento, a cui seguono i valori dei pesi in binario.

Per utilizzare una rete già creata si usi la **get_net** *netname.*

6. Come fare cosa

In questo capitolo vengono descritte le modalitá da seguire per la modifica del sorgente, per aggiungere delle variabili "visibili" run-time e dei nuovi comandi.

6.1. Come aggiungere variabili e nuovi comandi all'interprete

È stato accennato alla possibilità di definire le variabili.

È possibile farlo in due modi:

- da linea di comando;
- in una tabella (vedi BPN_USER.C).

Nel primo caso, la variabile è visibile e modificabile solo con l'interprete, ad esempio, con un'assegnazione, o un incremento in un ciclo da linea di comando.

Esempio: da linea di comando definire una variabile intera di nome **a** con **defint a**. Con **sh a** si può verificare se la sua creazione è avvenuta con successo.

A questo punto, la variabile **a** è utilizzabile nel corso dell'applicazione corrente e viene distrutta all'atto di uscita dall'applicazione stessa.

Nel secondo caso, la variabile viene utilizzata all'interno delle procedure del sorgente, il suo valore dipende quindi da esse. È visibile in qualunque momento dall'interprete e modificabile da comando.

Viene creata ad ogni lancio dell'applicazione.

Il modulo da modificare è lo BPN_USER.C, per cui è bene copiare in un nuovo file l'originale come esempio da conservare nelle altre applicazioni.

Nel modulo BPN_USER.C ci sono due tabelle:

```
/************************************
 * variabili visibili dall'Interprete
 ************************************/

int n1;
char n2[10];
double n3;

DVAR LocalUserTable[100]={
        {"new_var1", INTEGER, 1, &n1},
        {"new_var2", VARSTRING, 1, &n2},
        {"new_var3", DOUBLE, 10, &n3},
        {"", 0, 0, 0, 0},
         };

/************************************************************
 * prototipi delle funzioni richiamabili da linea di comando
 * dell'Interprete
 * **********************************************************/

USERDCOMM LocalUserComTable[100]={
 /* Commands must be entered lowercase */
       {"new_com", (void*)newcom},
       {"", 0},         /* mark end of table */
};

/*****************************************************************
*                         nuovo comando
*****************************************************************/

void newcom()
{
   char *s;

   if(h==1)
     {
        printf("\nSYNTAX\n");
        printf("newcom  filename num\n\n");
        printf("DESCRIPTION\n");
        printf("Example of user command\\n");
        return;
     }
```

```
    s = GetStrPar();

    if(GetParErr)
      {
         printf("No parameter 1: put string\\n");
         return;
      }

    strcpy(n2,s);
    n1 = GetNumPar();

    if(GetParErr)
      {
         printf("No parameter 1: put int number\\n");
         return;
      }

    n1 = n1 + 18;
    n3 = 13.6;

    return;
}
```

La tabella `LocalUserTable` contiene le nuove variabili del secondo tipo, descritto in precedenza.

Ciascuna di esse è definita da una stringa che contiene il nome con cui la si userà da interprete, il tipo, il numero di elementi di cui è composta e il puntatore alla variabile che viene usata all'interno del programma.

Da notare che questa ultima deve essere di tipo globale.

I tipi disponibili sono l'intero (INTEGER), la stringa (VARSTRING) e il double (DOUBLE).

Per definire dei vettori, come nel caso di `n2` basta specificare il numero di elementi che la compone.

Ciascun elemento della tabella `LocalUserComTable` è composto dalla stringa che permetterà all'interprete di eseguire il comando definito.

I parametri dei comandi, invece, vengono passati mediante l'uso delle procedure `GetStrPar` per la variabile stringa e `GetNumPar` per i parametri di tipo numerico (vedi il corpo della procedura `newcom`). Gli errori commessi alla chiamata del comando viene rilevato da `GetParErr`, che gestisce l'uscita dalla procedura.

Per vedere l'effetto della funzione **newcom** sulle variabili $n1$, $n2$, $n3$ eseguire prima, ad esempio, il comando **new_com** `ciao 2` a cui far seguire i comandi `sh ciao`, `sh new_var1`, `sh new_var2`, `sh new_var3`.

6.2. Come e dove modificare il sorgente

Il modo in cui i dati descriventi il dominio del problema vengono proposti alla rete neurale e' spesso determinante per un buon apprendimento, per cui durante lo studio del problema che si vuole risolvere con una rete neurale, e' abbastanza frequente la necessità di elaborare i dati di cui si dispone per rappresentare il dominio del problema, senza per questo modificarli fisicamente.

Ad esempio, consideriamo uno dei problemi più semplici che una rete neurale puo' risolvere: dati due insiemi si vuole simulare la funzione che associa a ciascun elemento del primo insieme l'elemento del secondo insieme la cui differenza dal primo sia minima.

Possiamo pensare due modi alternati di presentazione del problema: si può richiedere alla rete che da un certo input, "impari" a restituire in output il valore desiderato, oppure data in input la coppia di valori che minimizzi la differenza restituendo il valore zero.

Sempre nel modulo BPN_USER.C ci sono due funzioni molto importanti per la manipolazione dei dati prima della loro presentazione alla rete.

Il file degli esempi deve avere il seguente formato:

su ciascuna riga ogni dato rappresenta l'input al neurone di input

Le funzioni in questione sono:

```
/***************************************************
*         manipola i dati in ingresso alla rete
***************************************************/

void process_training_example()
{
     int i;

     for( i = 0; i<N_neuron[0]; i++ )
        {
          X_vet[i]=cX_vet[i];
        }

     if(question==2)
        {
           for( i = 0; i<N_neuron[N_slab-1]; i++ )
```

```
            {
              t[i]=ct[i];
            }
        }
    return;
}

/************************************************
*         manipola i dati in uscita alla rete
************************************************/

void process_out_example()
{
    int i;

    for( i = 0; i<N_neuron[N_slab-1]; i++ )
       {
          oX_vet[i]=X_vet[i];
       }

    return;
}
```

Cosi' come sono tali procedure forniscono gli esempi alla rete senza nessuna elaborazione preventiva.

Supponiamo che, nel corso dello studio di un problema tipo l'associazione degli elementi di due classi, si tenti la strada della minimizzazione delle differenze delle distanze degli elementi delle coppie da associare: può risultare utile non modificare il file originario e manipolarne la lettura da programma.

In uscita si potrebbe voler applicare all'uscita una qualsiasi funzione matematica.

Ad esempio:

```
/************************************************
*         manipola i dati in ingresso alla rete
************************************************/

void process_training_example()
{
    int i;

    for( i = 0; i<N_neuron[0]; i++ )
       {
```

```
            X_vet[i]=cX_vet[i];
          }

      if(question==2)
         {
           for( i = 0; i<N_neuron[N_slab-1]; i++ )
      {
        t[i]=ct[i]-cX_vet[i];
      }
 }
      return;
}

/************************************************
*          manipola i dati in uscita alla rete
************************************************/

void process_out_example()
{
    int i;

    for( i = 0; i<N_neuron[N_slab-1]; i++ )
        {
          oX_vet[i]=X_vet[i]+15.0;
        }

    return;
}
```

Si possono quindi modificare solo le assegnazioni alle variabili

`X_vet`

`t`

`oX_vet`

In `cX_vet` (vedi procedura process_training_example) ci sono i valori dell'esempio letti direttamente da file, in `ct` in valori del pattern desiderato.

`X_vet` invece è la struttura che passa i dati alla rete e riceve il suo output, che puo' essere modificato e passato esternamente da `oX_vet`.

Infine la variabile `t` contiene i valori con cui viene ad ogni passo confrontato l'output della rete.

La variabile **question** serve per la fase di utilizzo di una rete già addestrata, cioè per la sua interrogazione senza che sia noto il target da raggiungere.

Si supponga, ad esempio di avere una rete addestrata a risolvere il problema della somma. Se si vuole sommare due addendi senza conoscere a priori il risultato, alla rete non deve essere fornito il target. Se invece si desidera conoscere la precisione con cui la rete risolve un problema già noto, la variabile **question** deve essere posta a 2.

6.3. Come programmare un processo

Editare in un file di testo, ad esempio *ident.run*, un programma nel linguaggio dell'interprete.

Per eseguirlo digitare dalla finestra dei comandi **@***ident.run*.

Al suo interno si noti l'uso di **@***ident.conf*.

```
* una linea di commento inizia con un asterisco
* ident.conf contiene i comandi di assegnazione dei parametri di struttura della rete.
@ident.conf
* Prepara l'header del file di report del processo
print "Rete neurale che simula la " > report
print "Funzione identica su numeri interi tra 0 e 10" >> report
print "                " >> report
* visualizza  i parametri della rete
sh Net*  >> report
sh N_n* >> report
sh N_s* >> report
sh A* >> report
* alloca e inizializza i pesi della rete con valori random tra 0 e 1
init
* crea la finestra del grafico dell'errore di apprendimento
open_winerr
* carica i domini del problema dal file di addestramento ident
set_bound ident
* applica l'algoritmo di apprendimento 10000 volte sugli esempi
* del file ident
learning N 10000
* stampa nel file di report una riga, il n. corrente di iterazioni,
* una riga vuota e il valore dell'errore alla 10000esima iterazione
print "-----------------------------------------" >> report
print "Iteration = 10000
print "                " >> report
sh Error >> report
* stampa nel file di report una riga, il n. corrente di iterazioni,
* una riga vuota e il valore dell'errore alla 10000esima iterazione
print "-----------------------------------------" >> report
learning  N 10000
* stampa nel file di report una riga, il n. corrente di iterazioni,
* una riga vuota e il valore dell'errore alla 10000esima iterazione
```

```
print "-------------------------------------------" >> report
print "Iteration = 20000
print "                " >> report
sh Error >> report
* stampa nel file di report una riga, il n. corrente di iterazioni,
* una riga vuota e il valore dell´errore alla 10000esima iterazione
print "-------------------------------------------" >> report
learning N 10000
print "-------------------------------------------" >> report
* salva la rete addestrata  nel file ident.net
save_net ident.net
* esce dall´applicazione
end
```

6.4. Come usare ed estendere la libreria

Consideriamo un esempio, fornito dal modulo *prog_pr.c*

```
extern void init_lib_BPN();
extern void load_BPN();
#ifdef BPN_LIB

void main(int argc, char *argv[] )
     {
        double* ovet;
        double* ivet;
        int i;

        init_lib_BPN();

        /* caricamento dlla rete addestrata alla risoluzione della
           Funzione Identica su interi tra 0 e 10 */
        load_BPN("ident.net");

        /* lettura dei domini dal file di addestramento */
        bound_BPN("ident1");

        /* esecuzione della Funzione Identica sui dati nel file ident1 */
        fprocess_BPN("ident1", "ident1.out", "ident1.err" );
        fprocess_BPN("ident", "ident.out", "" );

        /* esecuzione della Funzione Identica sul valore 5 */ I
        ivet=(double*)malloc(sizeof(double)*2);
        ivet[0]=5;
        ovet=process_BPN(ivet);
        printf("risultato: %lf\n",ovet[0]);

        /* salvataggio della rete in memoria sul file ident1.net */
        save_BPN("ident1.net");

        /* esempi di esecuzione del comando more del S.O. */
        shell_BPN("more ident1");
        shell_BPN("more ident1.out");
        shell_BPN("more ident1.err");
        shell_BPN("more ident");
        shell_BPN("more ident.out");
```

```
        return;
    }
#endif
```

Per l'uso corretto delle funzioni di libreria, esiste una sequenza fondamentale di chiamata.

Supponiamo di aver risolto il problema X con il simulatore e di aver salvato la rete addestrata in un file.

La rete deve essere caricata in memoria dalla funzione `load_BPN` a cui bisogna passare la stringa costante con il nome del file.

Secondo passo necessario è il caricamento dei limiti del dominio di ciascuna coordinata dei pattern costituenti l'insieme degli esempi del problema, per la normalizzazione dei dati.

Ció si ottiene chiamando **bound_BPN** sul nome del file degli esempi su cui è stato eseguito l'apprendimento. Se si desidera che la rete processi i dati contenuti in un file, si deve usare la funzione **fprocess_BPN**. Bisogna passare il nome del file di input e quello di output. Eventualmente, se si volesse solo testare la precisione della rete e si dispone dei valori desiderati, il nome del file degli errori commessi dalla rete su ciascun esempio del problema.

Se invece si desidera il calcolo della rete su un unico pattern si deve usare la **process_BPN**.

La funzione **save_BPN** salva la rete in memoria su un file di cui si deve specificare il nome, mentre la **save_sub_BPN** salva una sottorete. Questa funzione e' usata sopratutto per il problema della compressione.

Per reinizializzare i pesi di connessione mantenendo inalterata la struttura della rete e i parametri di apprendimento chiamare la `init_BPN`.

Di particolare utilitá sono le funzioni **at_BPN** e **shell_BPN**.

La prima permette di eseguire un programma scritto nella sintassi del linguaggio del BPN, contenuto in un file di testo. Tale funzione risulta utile, ad esempio, se si desidera realizzare la fase di apprendimento da programma. Bisogna evitare nel programma le routine di visualizzazione, in quanto la compilazione condizionata non permette il loro utilizzo.

La **shell_BPN** permette di usare i comandi del tuo S.O. da programma.

Notare che si può estendere la libreria aggiungendo delle funzioni nel modulo BPN_USER.C

Esempio:

Si supponga di avere nel modulo BPN_USER.C la funzione **void esempio()** che realizza il comando dell'interprete **example** *par1 par2* con par1 stringa e par2 intero. Per aggiungere la routine nella libreria nel modulo BPN_LIB.C:

```
void example_BPN(char *s, char *p)
{
   char *line;

   line=(char *)malloc(sizeof(char)*80);
   strcpy(line, "example "); /*nota il blank dopo il nome*/
   strcat(line, s);
   strcat(line, "  ");
   strcat(line, p);
   XT_interf(line);

   return;
}
```

APPENDICE

Algoritmo Back-Propagation

Le reti artificiali di neuroni implemetate dal BPN sono costituite da tre tipi di unitá.

- Unitá o neurone di ingresso.
- Unitá o neurone nascosto.
- Unitá o neurone di uscita.

I neuroni sono disposti in strati o layers.

Le unitá di ingresso ricevono informazioni dal file di input. quelle di uscita emettono segnali in output e quelle nascoste fungono da intermediarie tra quelle di ingresso e quelle di uscita: non ricevono informazioni dall'esterno e non producono segnali di uscita rilevabili.

I neuroni dello strato di input e quelli di ogni strato intermedio sono connessi con tutti i neuroni dello strato seguente, permettendo cosi' la propagazione del segnale in avanti.

Il principio generale di funzionamento del singolo neurone (PE Processing Element) di una rete neurale BP è il seguente:

il neurone riceve un segnale da ogni neurone dello strato precedente con cui è connesso, somma i segnali dopo averli pesati con la forza della connessione e applica al risultato una funzione di attivazione. Questo avviene per i tutti i neuroni che non appartengono allo strato di input. Per questi ultimi, invece, il segnale ricevuto è il valore dell'input normalizzato tra 0 e 1.

Il valore così ottenuto costituisce la "attivazione" del neurone e rappresenta il segnale che verrá propagato in avanti. I segnali prodotti dall'ultimo strato costituiranno l'output della rete.

A questo punto l'output viene confrontato con il valore desiderato e i pesi di connessione vengono aggiornati in modo da minimizzare l'errore tra l'output corrente e il valore desiderato.

Analizziamo piú in dettaglio quanto detto:

dato un insieme di esempi di risposte corrette al problema, che ne descrivono il dominio, la rete apprende per iterazioni successive, cercando di commettere ad ogni passo errori sempre piú piccoli.

La funzione per il calcolo dell' errore usata nel nostro caso è la seguente:

$$E = \frac{1}{2} \sum_{i=1}^{out} (y_i - d_i)^2$$

dove

d_i è l'uscita desiderata dell'unità i.

out è la dimensione dello strato di output.

y_i è l'uscita effettiva, con y_i dato dalla funzione sigmoidea :

$$\frac{(P2-P3)}{(1.0+e^{-P1\,x})+P3}.$$

dove $P3$ rappresenta il limite inferiore e $P2$ quello superiore, per cui $P2$ deve essere necessariamente maggiore di $P3$.

$P1$ determina la pendenza della curva ad $x = 0$. Deve essere sempre positivo.

Per rendere minimo l'errore, se ne considera la derivata rispetto al peso w_{ij} della connessione tra le unità i e j:

$$\frac{\partial E}{\partial w_{ij}} = y_i y_j (1 - y_j)\beta_j$$

dove

$\beta_j = (y_j - d_j)$ per le unità di uscita.

$\beta_j = sum_k(w_{jk}(y_k(1 - y_k) + C)\beta_k)$ per le unità nascoste.

(k rappresenta il numero delle unità dello strato successivo con cui l'unità j è connessa).

L'errore viene, quindi, direttamente calcolato per le connessioni che entrano nelle unità di uscita, mentre per le unità nascoste la derivata dipende dai valori calcolati a tutti gli strati successivi.

Questo significa che, per calcolare la derivata, il valore di *3* deve essere retropropagato lungo la rete.

Usando queste equazioni, l'algoritmo può essere implementato come segue. dopo aver scelto il tipo di apprendimento.

Normal

1. Scegliere:

   α: parametro di apprendimento. Specifica l'ampiezza del passo del gradiente discendente.

   (Dominio: 0.1...1.0)

Finchè la rete non termina il numero di iterazioni specificato dal comando di `learning`:

2. Per ciascuna configurazione campione:

   2.1 Fare un passo in avanti nella rete ricavando una configurazione in uscita

   2.2 Per tutte le unità di uscita calcolare $\beta_j = (y_j - d_j)$

Per tutte le altre unità calcolare $\beta_j = \sum_k (w_{jk}(y_k(1-y_k)\beta_k)$

2.3 Per tutti i pesi della rete calcolare la quantità: $\Delta w_{ij}(t+1) = \alpha\delta_j o_j$

dove

$\delta_j = -y_j(1-y_j)\beta_j$ o_j è l'output della rete.

2.4 Aggiornare i pesi nel modo seguente: $w_{ij}(t+1) = w_{ij} + \Delta w_{ij}(t+1)$

Batching

Nell'apprendimento Normal, per ogni singola coppia input/output-desiderato avviene l'aggiornamento del peso di una quantita' pari al gradiente Δw calcolato.

Il processo di media dei gradienti di un sottoinsieme di esempi è chiamato Batching. La regola di apprendimento, in tal caso diventa

1. Scegliere:

α: parametro di apprendimento. Specifica l'ampiezza del passo del gradiente discendente.

(Dominio: 0.1...1.0)

Batch: Numero di esempi dopo la cui eleborazione si vuole avvenga il processo di aggiornamento dei pesi.

(Dominio:1...*numerodiesempi*)

Finchè la rete non termina il numero di iterazioni specificato dal comando di **learning**:

2. Per ciascuna configurazione campione:

2.1 Fare un passo in avanti nella rete ricavando una configurazione in uscita

2.2 Per tutte le unità di uscita calcolare $\beta_j = (y_j - d_j)$

Per tutte le altre unità calcolare $\beta_j = \sum_k w_{jk}(y_k(1-y_k)\beta_k)$

2.3 Per tutti i pesi della rete calcolare la quantità: $\Delta w_{ij}(t+1) = \alpha\delta_j o_j$

dove

$\delta_j = -y_j(1-y_j)\beta_j$ o_j è l'output della rete.

2.4 Aggiornare i pesi nel modo seguente: $\frac{w_{ij}(t+1)=w_{ij}+\Delta w_{ij}(t+1)}{Count}$ se $Count = Batch$

altrimenti $w_{ij}(t+1) = w_{ij}(t)$

Lo svantaggio di questo metodo è dato dal fatto che, poichè l'aggiornamento avviene ogni Batch esempi. il tempo di convergenza ad un minimo locale è spesso più grande di quello necessario alla regola Normal.

Un'alternativa al Batching è lo Smoothing.

Lo Smoothing è un compromesso tra il Batching e l'apprendimento Normal, in quanto come il primo utilizza nel calcolo del gradiente il valore al passo precedente, ma permette un aggiornamento ad ogni coppia

di esempio.

Smoothing

1. Scegliere:

   α: parametro di apprendimento. Specifica l'ampiezza del passo del gradiente discendente.

   (Dominio: 0.1...1.0)

   μ: il momento, cioe' il coefficiente moltiplicativo della modifica del peso al passo precedente.

   (Dominio:0...1.0)

   Finchè la rete non termina il numero di iterazioni specificato dal comando di **learning**:

2. Per ciascuna configurazione campione:

   2.1 Fare un passo in avanti nella rete ricavando una configurazione in uscita

   2.2 Per tutte le unità di uscita calcolare $\beta_j = (y_j - d_j)$

   Per tutte le altre unità calcolare $\beta_j = \sum_k w_{jk}(y_k(1 - y_k)\beta_k)$

   2.3 Per tutti i pesi della rete calcolare la quantità: $\Delta w_{ij}(t+1) = (1-\mu)\alpha\delta_j o_j + \mu\Delta w_{ij}(t)$

   dove

   $\delta_j = -y_j(1 - y_j)\beta_j$ o_j è l'output della rete.

   2.4 Aggiornare i pesi nel modo seguente: $w_{ij}(t+1) = w_{ij} + \Delta w_{ij}(t+1)$

Ultimo tipo di apprendimento implementato è quello con **Momentum** che vuole i seguenti parametri.

1. Scegliere:

   α: parametro di apprendimento. Specifica l'ampiezza del passo del gradiente discendente.

   (Dominio: 0.1...1.0)

   η: coefficiente moltiplicativo del valore precedente del peso da aggiornare.

   (Dominio:0.1...1.0)

   μ: il momento, cioe' il coefficiente moltiplicativo della modifica del peso al passo precedente.

   (Dominio:0...1.0)

   C: parametro di *flat spot elimination*, costante additiva alla funzione di attivazione, che permette di attraversare la superficie della curva degli errori, qualora diventi piana. In tal caso la derivata è nulla e la convergenza dell'algoritmo verrebbe compromessa.

   (Dominio:0...0.25)

   Finchè la rete non termina il numero di iterazioni specificato dal comando di **learning**:

2. Per ciascuna configurazione campione:

2.1 Fare un passo in avanti nella rete ricavando una configurazione in uscita

2.2 Per tutte le unità di uscita calcolare $\beta_j = (y_j - d_j)$

Per tutte le altre unità calcolare $\beta_j = \sum_k (w_{jk}(y_k(1 - y_k) + C)\beta_k)$

2.3 Per tutti i pesi della rete calcolare la quantità: $\Delta w_{ij}(t + 1) = \alpha \delta_j o_j + \mu \Delta w_{ij}(t)$

dove

$\delta_j = -(y_j(1 - y_j) + C)\beta_j$ o_j è l'output della rete.

2.4 Aggiornare i pesi nel modo seguente: $w_{ij}(t + 1) = \eta w_{ij} + \Delta w_{ij}(t + 1)$

Al termine del processo di apprendimento, la rete acquisisce la capacità di generalizzare e quindi di fornire una risposta su esempi del problema che non appartengono all'insieme di apprendimento.