

Intelligenza Artificiale Complementi ed Esercizi

Reti Neurali

A.A. 2008-2009

Sommario

- Esempio di costruzione di programma in linguaggio c per la backpropagation
- Neurosolutions: esempio di sistema commerciale per implementare reti neurali
- Joone: esempio di sistema open source per implementare reti neurali

Algoritmo BP in pseudolinguaggio

```

function APPRENDIMENTO-PROP-INDIETRO(esempi,rete) returns una rete neurale
  inputs: esempi, un insieme di esempi, ognuno con vettore di input x e vettore di output y
           rete, una rete feed-forward multistrato con L strati, pesi  $W_{i,j}$  e funzione di attivazione g

  repeat
    for each e in esempi do
      for each nodo k nello strato di input do  $a_k \leftarrow x_k[e]$ 
      for  $l=2$  to L do
         $in_j \leftarrow \sum_k W_{j,k} a_k$ 
         $a_j \leftarrow g(in_j)$ 
      for each nodo i nello strato di output do
         $\Delta_i \leftarrow g'(in_i) \times (y_i[e] - a_i)$ 
      for  $l=L-1$  to  $1$  do
        for each nodo j nello strato l do
           $\Delta_j \leftarrow g'(in_j) \times \sum_i W_{j,i} \Delta_i$ 
        for each nodo i nello strato l+1 do
           $W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i$ 
  until non è soddisfatto un qualche criterio di terminazione
  return IPOTESI-RETE-NEURALE (rete)
  
```

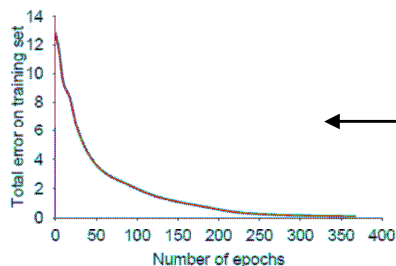
Calcolo del gradiente

$$\begin{aligned}
 \frac{\partial E}{\partial W_{k,j}} &= -\sum_i (y_i - a_i) \frac{\partial a_i}{\partial W_{k,j}} = -\sum_i (y_i - a_i) \frac{\partial g(in_i)}{\partial W_{k,j}} \\
 &= -\sum_i (y_i - a_i) g'(in_i) \frac{\partial in_i}{\partial W_{k,j}} = -\sum_i \Delta_i \frac{\partial}{\partial W_{k,j}} \left(\sum_j W_{j,i} a_j \right) \\
 &= -\sum_i \Delta_i W_{j,i} \frac{\partial a_j}{\partial W_{k,j}} = -\sum_i \Delta_i W_{j,i} \frac{\partial g(in_j)}{\partial W_{k,j}} \\
 &= -\sum_i \Delta_i W_{j,i} g'(in_j) \frac{\partial in_j}{\partial W_{k,j}} \\
 &= -\sum_i \Delta_i W_{j,i} g'(in_j) \frac{\partial}{\partial W_{k,j}} \left(\sum_k W_{k,j} a_k \right) \\
 &= -\sum_i \Delta_i W_{j,i} g'(in_j) a_k = -a_k \Delta_j
 \end{aligned}$$

Curve di addestramento di BP per l'esempio del ristorante

At each epoch, sum gradient updates for all examples and apply

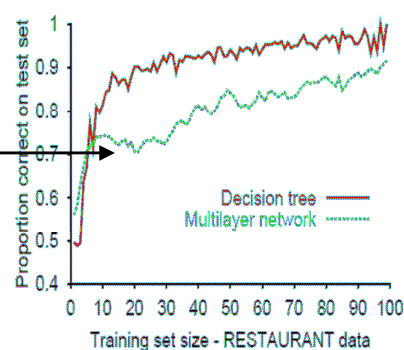
Training curve for 100 restaurant examples: finds exact fit



Typical problems: slow convergence, local minima

Curva dell'errore

Curva delle prestazioni sul test set



Un programma per la back propagation

Struttura del programma

- Variabili globali
- Macro e costanti
- Funzioni di supporto
- Algoritmo feed-forward
- Backpropagation

Variabili globali

```
#define INPUT_NEURONS      4
#define HIDDEN_NEURONS    3
#define OUTPUT_NEURONS    4

/* Input to Hidden Weights (with Biases) */
double wih[INPUT_NEURONS+1][HIDDEN_NEURONS];

/* Hidden to Output Weights (with Biases) */
double who[HIDDEN_NEURONS+1][OUTPUT_NEURONS];

/* Activations */
double inputs[INPUT_NEURONS];
double hidden[HIDDEN_NEURONS];
double target[OUTPUT_NEURONS];
double actual[OUTPUT_NEURONS];

/* Unit Errors */
double erro[OUTPUT_NEURONS];
double errh[HIDDEN_NEURONS];
```

Variabili globali

```
#define INPUT_NEURONS    4
#define HIDDEN_NEURONS  3
#define OUTPUT_NEURONS  4

/* Input to Hidden Weights (with Biases) */
double wih[INPUT_NEURONS+1][HIDDEN_NEURONS];

/* Hidden to Output Weights (with Biases) */
double who[HIDDEN_NEURONS+1][OUTPUT_NEURONS];

/* Activations */
double inputs[INPUT_NEURONS];
double hidden[HIDDEN_NEURONS];
double target[OUTPUT_NEURONS];
double actual[OUTPUT_NEURONS];

/* Unit Errors */
double erro[OUTPUT_NEURONS];
double errh[HIDDEN_NEURONS];
```

Struttura della rete

Variabili globali

```
#define INPUT_NEURONS    4
#define HIDDEN_NEURONS  3
#define OUTPUT_NEURONS  4

/* Input to Hidden Weights (with Biases) */
double wih[INPUT_NEURONS+1][HIDDEN_NEURONS];

/* Hidden to Output Weights (with Biases) */
double who[HIDDEN_NEURONS+1][OUTPUT_NEURONS];

/* Activations */
double inputs[INPUT_NEURONS];
double hidden[HIDDEN_NEURONS];
double target[OUTPUT_NEURONS];
double actual[OUTPUT_NEURONS];

/* Unit Errors */
double erro[OUTPUT_NEURONS];
double errh[HIDDEN_NEURONS];
```

Primo strato di pesi

Variabili globali

```
#define INPUT_NEURONS    4
#define HIDDEN_NEURONS  3
#define OUTPUT_NEURONS  4

/* Input to Hidden Weights (with Biases) */
double wih[INPUT_NEURONS+1][HIDDEN_NEURONS];

/* Hidden to Output Weights (with Biases) */
double who[HIDDEN_NEURONS+1][OUTPUT_NEURONS];

/* Activations */
double inputs[INPUT_NEURONS];
double hidden[HIDDEN_NEURONS];
double target[OUTPUT_NEURONS];
double actual[OUTPUT_NEURONS];

/* Unit Errors */
double erro[OUTPUT_NEURONS];
double errh[HIDDEN_NEURONS];
```

Secondo strato di pesi

Variabili globali

```
#define INPUT_NEURONS    4
#define HIDDEN_NEURONS  3
#define OUTPUT_NEURONS  4

/* Input to Hidden Weights (with Biases) */
double wih[INPUT_NEURONS+1][HIDDEN_NEURONS];

/* Hidden to Output Weights (with Biases) */
double who[HIDDEN_NEURONS+1][OUTPUT_NEURONS];

/* Activations */
double inputs[INPUT_NEURONS];
double hidden[HIDDEN_NEURONS];
double target[OUTPUT_NEURONS];
double actual[OUTPUT_NEURONS];

/* Unit Errors */
double erro[OUTPUT_NEURONS];
double errh[HIDDEN_NEURONS];
```

Valori dei nodi
(output)

Variabili globali

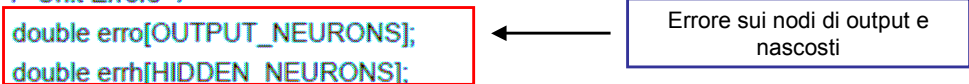
```
#define INPUT_NEURONS    4
#define HIDDEN_NEURONS  3
#define OUTPUT_NEURONS  4

/* Input to Hidden Weights (with Biases) */
double wih[INPUT_NEURONS+1][HIDDEN_NEURONS];

/* Hidden to Output Weights (with Biases) */
double who[HIDDEN_NEURONS+1][OUTPUT_NEURONS];

/* Activations */
double inputs[INPUT_NEURONS];
double hidden[HIDDEN_NEURONS];
double target[OUTPUT_NEURONS];
double actual[OUTPUT_NEURONS];

/* Unit Errors */
double erro[OUTPUT_NEURONS];
double errh[HIDDEN_NEURONS];
```



Macro e costanti

```
#define LEARN_RATE 0.2
#define RAND_WEIGHT ((float)rand() / (float)RAND_MAX) - 0.5)

#define getSRand() ((float)rand() / (float)RAND_MAX)
#define getRand(x) (int)((x) * getSRand())

#define sqr(x) ((x) * (x))
```

- Learning rate $\alpha = 0.2$
- Pesi iniziali selezionati in modo casuale nell'intervallo $[-0.5, 0.5]$
- La funzione *rand()* genera e restituisce un numero pseudo-casuale tra zero ed il valore RAND_MAX il quale deve essere almeno 32767.

Funzioni di supporto

```
void assignRandomWeights( void )
{
    int hid, inp, out;

    for (inp = 0 ; inp < INPUT_NEURONS+1 ; inp++) {
        for (hid = 0 ; hid < HIDDEN_NEURONS ; hid++) {
            wih[inp][hid] = RAND_WEIGHT;
        }
    }

    for (hid = 0 ; hid < HIDDEN_NEURONS+1 ; hid++) {
        for (out = 0 ; out < OUTPUT_NEURONS ; out++) {
            who[hid][out] = RAND_WEIGHT;
        }
    }
}

double sigmoid( double val )
{
    return (1.0 / (1.0 + exp(-val)));
}

double sigmoidDerivative( double val )
{
    return ( val * (1.0 - val) );
}
```

Funzioni di supporto

```
void assignRandomWeights( void )
{
    int hid, inp, out;

    for (inp = 0 ; inp < INPUT_NEURONS+1 ; inp++) {
        for (hid = 0 ; hid < HIDDEN_NEURONS ; hid++) {
            wih[inp][hid] = RAND_WEIGHT;
        }
    }

    for (hid = 0 ; hid < HIDDEN_NEURONS+1 ; hid++) {
        for (out = 0 ; out < OUTPUT_NEURONS ; out++) {
            who[hid][out] = RAND_WEIGHT;
        }
    }
}

double sigmoid( double val )
{
    return (1.0 / (1.0 + exp(-val)));
}

double sigmoidDerivative( double val )
{
    return ( val * (1.0 - val) );
}
```

Inizializzazione con pesi casuali
(attenzione al bias)

Funzioni di supporto

```
void assignRandomWeights( void )
{
    int hid, inp, out;

    for (inp = 0 ; inp < INPUT_NEURONS+1 ; inp++) {
        for (hid = 0 ; hid < HIDDEN_NEURONS ; hid++) {
            wih[inp][hid] = RAND_WEIGHT;
        }
    }

    for (hid = 0 ; hid < HIDDEN_NEURONS+1 ; hid++) {
        for (out = 0 ; out < OUTPUT_NEURONS ; out++) {
            who[hid][out] = RAND_WEIGHT;
        }
    }
}

double sigmoid( double val )
{
    return (1.0 / (1.0 + exp(-val)));
}

double sigmoidDerivative( double val )
{
    return ( val * (1.0 - val) );
}
```

← Calcolo sigmoide e sua derivata

Algoritmo feed-forward

```
void feedForward()
{
    int inp, hid, out;
    double sum;
    /* Calculate input to hidden layer */
    for (hid = 0 ; hid < HIDDEN_NEURONS ; hid++) {
        sum = 0.0;
        for (inp = 0 ; inp < INPUT_NEURONS ; inp++) {
            sum += inputs[inp] * wih[inp][hid];
        }
        /* Add in Bias */
        sum += wih[INPUT_NEURONS][hid];
        hidden[hid] = sigmoid(sum);
    }
    /* Calculate the hidden to output layer */
    for (out=0;out<OUTPUT_NEURONS;out++){
        sum=0.0;
        for (hid=0;hid<HIDDEN_NEURONS;hid++){
            sum+=hidden[hid]*who[hid][out];
        }
        /* add in bias */
        sum+=who[HIDDEN_NEURONS][out];
        actual[out]=sigmoid(sum);
    }
}
```

Algoritmo feed-forward

```
void feedForward()  
{  
    int inp, hid, out;  
    double sum; ← Variabili locali  
    /* Calculate input to hidden layer */  
    for (hid = 0 ; hid < HIDDEN_NEURONS ; hid++) {  
        sum = 0.0;  
        for (inp = 0 ; inp < INPUT_NEURONS ; inp++) {  
            sum += inputs[inp] * wih[inp][hid];  
        }  
        /* Add in Bias */  
        sum += wih[INPUT_NEURONS][hid];  
        hidden[hid]= sigmoid(sum);  
    }  
    /* Calculate the hidden to output layer */  
    for (out=0;out<OUTPUT_NEURONS;out++){  
        sum=0.0;  
        for (hid=0;hid<HIDDEN_NEURONS;hid++){  
            sum+=hidden[hid]*who[hid][out];  
        }  
        /* add in bias */  
        sum+=who[HIDDEN_NEURONS][out];  
        actual[out]=sigmoid(sum);  
    }  
}
```

Algoritmo feed-forward

```
void feedForward()  
{  
    int inp, hid, out;  
    double sum;  
    /* Calculate input to hidden layer */  
    for (hid = 0 ; hid < HIDDEN_NEURONS ; hid++) {  
        sum = 0.0;  
        for (inp = 0 ; inp < INPUT_NEURONS ; inp++) {  
            sum += inputs[inp] * wih[inp][hid];  
        }  
        /* Add in Bias */  
        sum += wih[INPUT_NEURONS][hid];  
        hidden[hid]= sigmoid(sum);  
    }  
    /* Calculate the hidden to output layer */  
    for (out=0;out<OUTPUT_NEURONS;out++){  
        sum=0.0;  
        for (hid=0;hid<HIDDEN_NEURONS;hid++){  
            sum+=hidden[hid]*who[hid][out];  
        }  
        /* add in bias */  
        sum+=who[HIDDEN_NEURONS][out];  
        actual[out]=sigmoid(sum);  
    }  
}
```

$$in_i = \sum_{j=0}^n W_{j,i} a_j$$

Algoritmo feed-forward

```
void feedForward()  
{  
    int inp, hid, out;  
    double sum;  
    /* Calculate input to hidden layer */  
    for (hid = 0 ; hid < HIDDEN_NEURONS ; hid++) {  
        sum = 0.0;  
        for (inp = 0 ; inp < INPUT_NEURONS ; inp++) {  
            sum += inputs[inp] * wih[inp][hid];  
        }  
        /* Add in Bias */  
        sum += wih[INPUT_NEURONS][hid];  
        hidden[hid]= sigmoid(sum);  
    }  
    /* Calculate the hidden to output layer */  
    for (out=0;out<OUTPUT_NEURONS;out++){  
        sum=0.0;  
        for (hid=0;hid<HIDDEN_NEURONS;hid++){  
            sum+=hidden[hid]*who[hid][out];  
        }  
        /* add in bias */  
        sum+=who[HIDDEN_NEURONS][out];  
        actual[out]=sigmoid(sum); ← Uscita della rete  
    }  
}
```

Backpropagation

```
void backPropagate(void )  
{  
    int inp, hid, out;  
    /* Calculate the output layer error */  
    for (out=0;out<OUTPUT_NEURONS;out++) {  
        erro[out]=(target[out]-actual[out])*sigmoidDerivative(actual[out]);  
    }  
    /* Calculate the hidden layer error */  
    for (hid=0;hid<HIDDEN_NEURONS;hid++){  
        errh[hid]=0.0;  
        for(out=0;out<OUTPUT_NEURONS;out++){  
            errh[hid]+=erro[out]*who[hid][out];  
        }  
        errh[hid]*=sigmoidDerivative(hidden[hid]);  
    }  
}
```

Backpropagation

```
void backPropagate(void )
{
int inp, hid, out;
/* Calculate the output layer error */
for (out=0;out<OUTPUT_NEURONS;out++) {
    erro[out]=(target[out]-actual[out])*sigmoidDerivative(actual[out]); ←  $\Delta_i$ 
}
/* Calculate the hidden layer error */
for (hid=0;hid<HIDDEN_NEURONS;hid++){
    errh[hid]=0.0;
    for(out=0;out<OUTPUT_NEURONS;out++){
        errh[hid]+=erro[out]*who[hid][out];
    }
    errh[hid]*=sigmoidDerivative(hidden[hid]);
}
}
```

Backpropagation

```
void backPropagate(void )
{
int inp, hid, out;
/* Calculate the output layer error */
for (out=0;out<OUTPUT_NEURONS;out++) {
    erro[out]=(target[out]-actual[out])*sigmoidDerivative(actual[out]);
}
/* Calculate the hidden layer error */
for (hid=0;hid<HIDDEN_NEURONS;hid++){
    errh[hid]=0.0;
    for(out=0;out<OUTPUT_NEURONS;out++){
        errh[hid]+=erro[out]*who[hid][out];
    }
    errh[hid]*=sigmoidDerivative(hidden[hid]); ←  $\Delta_j \leftarrow g'(in_j) \times \sum_i W_{j,i} \Delta_i$ 
}
}
```

Backpropagation

```
/* Update the weights for the output layer */
```

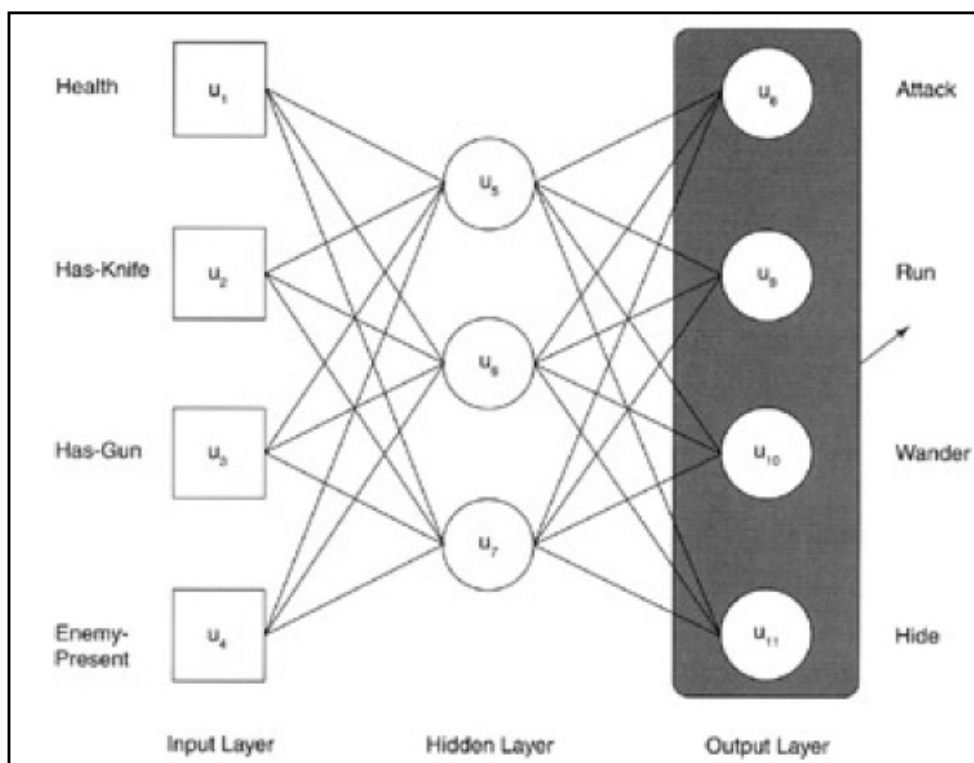
```
for (out=0;out<OUTPUT_NEURONS;out++){  
    for (hid=0;hid<HIDDEN_NEURONS;hid++){  
        who[hid][out]+=(LEARN_RATE*erro[out]*hidden[hid]);  
    }  
    /* Update bias */  
    who[HIDDEN_NEURONS][out]+=LEARN_RATE*erro[out];  
}
```

$$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i$$

```
/* Update the layer for the hidden layer */
```

```
for (hid=0;hid<HIDDEN_NEURONS;hid++){  
    for (inp=0;inp<INPUT_NEURONS;inp++){  
        wih[inp][hid]+=(LEARN_RATE*errh[hid]*inputs[inp]);  
    }  
    /* Update bias */  
    wih[INPUT_NEURONS][hid]+=(LEARN_RATE*errh[hid]);  
}
```

Esempio: Neurocontroller



Azione di un agente in base alla sua percezione

Esempio: Neurocontroller

- **Input**

- **healt of the agent** (0-poor to 2-healty)
- **has-knife** (1 if the agent has a knife 0 otherwise)
- **has-gun** (1 if in possession, 0 otherwise)
- **enemy-present** (number of enemies in field of view)

- **Output**

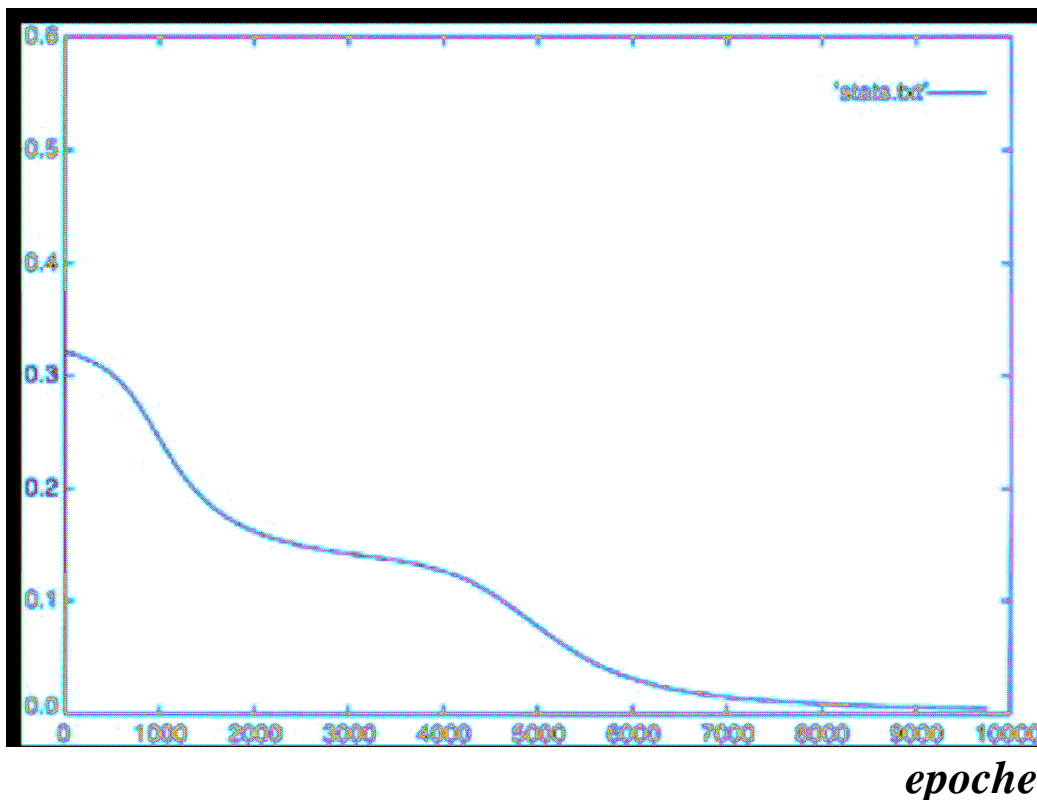
- *attack*: l'agente attacca i suoi pari nel suo campo visuale
- *run*: l'agente abbandona la sua posizione
- *wander*: l'agente vaga all'interno del suo ambiente
- *hide*: l'agente cerca un riparo

Training database

Health	Has-Knife	Has-Gun	Enemies	Behaviour
2	0	0	0	Wander
2	0	0	1	Wander
2	0	1	1	Attack
2	0	1	2	Attack
2	1	0	2	Hide
...
0	1	0	1	Hide

Addestramento

E



Rappresentazione del training set

```
typedef struct{
    double health;
    double knife;
    double gun;
    double enemy;
    double out[OUTPUT_NEURONS];
} ELEMENT;

#define MAX_SAMPLES 18
ELEMENT samples[MAX_SAMPLES]={
    {2.0,0.0,0.0,0.0,{0.0,0.0,1.0,0.0}},
    {2.0,0.0,0.0,1.0,{0.0,0.0,1.0,0.0}},
    {2.0,0.0,1.0,1.0,{1.0,0.0,0.0,0.0}},
    {2.0,0.0,1.0,2.0,{0.0,0.0,0.0,1.0}},
    {2.0,1.0,0.0,1.0,{1.0,0.0,0.0,0.0}},
    .....
    .....
```

MAIN

```
int main()
{
    double err;
    int i, sample=0, iterations=0;
    int sum=0;

    out=fopen("stats.txt", "w");
    /*seed the random number generator */
    srand(time(NULL));
    assignRandomWeights();
    /** train the network */
    while(1){
        if(++sample==MAX_SAMPLES) sample=0;
        inputs[0]=samples[sample].health
        inputs[1]=samples[sample].knife;
        inputs[2] = samples[sample].gun;
        inputs[3] = samples[sample].enemy;
        target[0] = samples[sample].out[0];
        target[1] = samples[sample].out[1];
        target[2] = samples[sample].out[2];
        target[3] = samples[sample].out[3];
        feedForward();
        err = 0.0;
```

MAIN

```
for (i = 0 ; i < OUTPUT_NEURONS ; i++) {
    err += sqrt( (samples[sample].out[i] - actual[i]) );
}
err = 0.5 * err;
fprintf(out, "%g\n", err);
printf("mse = %g\n", err);
if (iterations++ > 100000) break;
backPropagate();
}
```

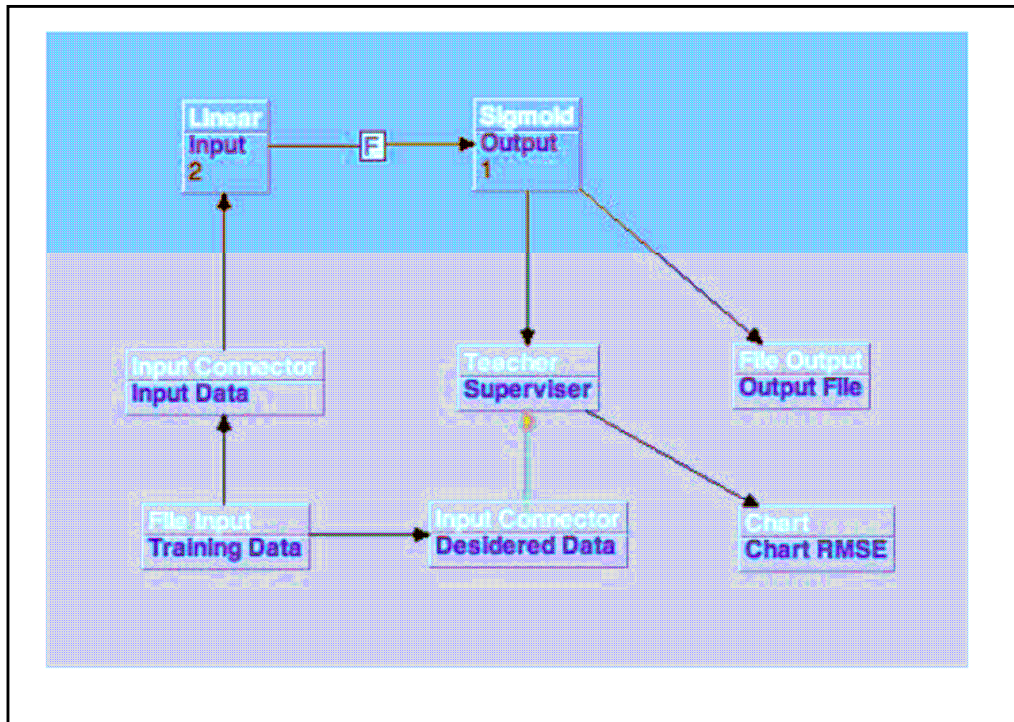

Joone

- Progetto italiano open source
- Ambiente per la simulazione di reti neurali
- Completa libreria di funzioni
- Editor grafico per sperimentazioni
- <http://www.jooneworld.com/>

Joone

The screenshot shows the Joone website in a Mozilla Firefox browser window. The browser title is "Joone - Java Object Oriented Neural Engine - Mozilla Firefox". The address bar shows "http://www.jooneworld.com/". The website has a blue header with the Joone logo (a circular arrangement of blue nodes) and the text "Java Object Oriented Neural Engine by Paolo Marrone". Below the header, there are navigation tabs for "Home", "Documentation", and "Community". A search bar is located in the top right corner. The main content area features a sidebar on the left with a "About" section containing links to "Introduction", "News", "Features", "Downloads", "Support", "Changes", "Todo", "Documentation", "Index", "The Core Engine", "The GUI Editor", "The DTE", and "Whole site". The main content area has a section titled "Joone - Java Object Oriented Neural Engine" with a list of links: "News", "About Joone", "Some features...", "The Framework", and "Resources". Below this is a "News" section with two entries: "Mar 14, 2007 - Nightly Builds now available" and "Jan 19, 2007 - Joone 2.0 RC1 is out!". The browser's status bar at the bottom indicates "Completato".

Joone



Apprendimento della funzione AND

Neurosolutions

- Strumento a pagamento
- <http://www.neurosolutions.com>
- disponibilità versione trial

The screenshot shows the NeuroSolutions website interface. At the top, there's a navigation bar with links for Home, Products, Downloads, Resources, Support, and Order. The main content area is dominated by a large banner for the 'Premier Neural Network Development Environment'. Below this, there are several product/service tiles: 'NeuroSolutions', 'NeuroSolutions for Excel', 'Custom Solution Wizard', 'NeuroSolutions for Matlab', 'Consulting Services', and 'Neural Network Course'. Each tile includes a brief description of the product. On the right side, there's a 'News and Reviews' section with a 'Product Advisor' widget. The browser's address bar shows the URL 'http://www.neurosolutions.com/'.

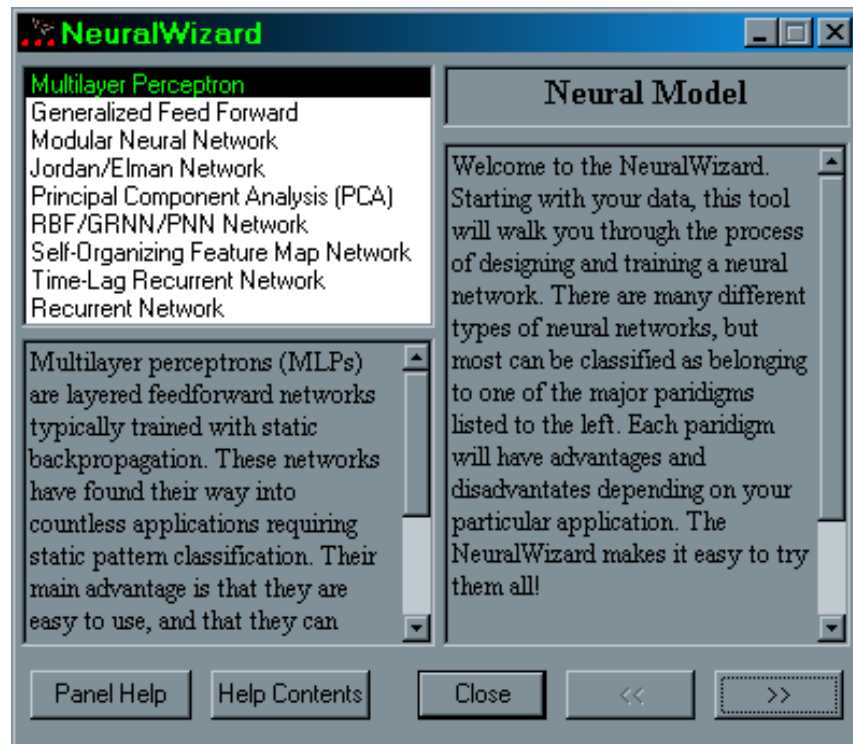
NeuralWizard – passo 1

COSTRUZIONE GUIDATA DI UNA RETE NEURALE

Scelta del tipo di rete:

Reti disponibili in
NeuroSolutions

Caratteristiche
della rete



NeuralWizard – passo 1

Tipologie di rete disponibili in NeuroSolution:

- Percettrone multistrato (MLP)
- Generalized Feedforward MLP
- Modular Feedforward
- Radial Basis Function (RBF)
- Jordan and Elman
- Principal Component Analysis (PCA)
Hybrids
- Self-Organizing Feature Map (SOFM)
Hybrid
- Time Lagged Recurrent
- General Recurrent

NeuralWizard – passo 2

Training:

Scelta file di training

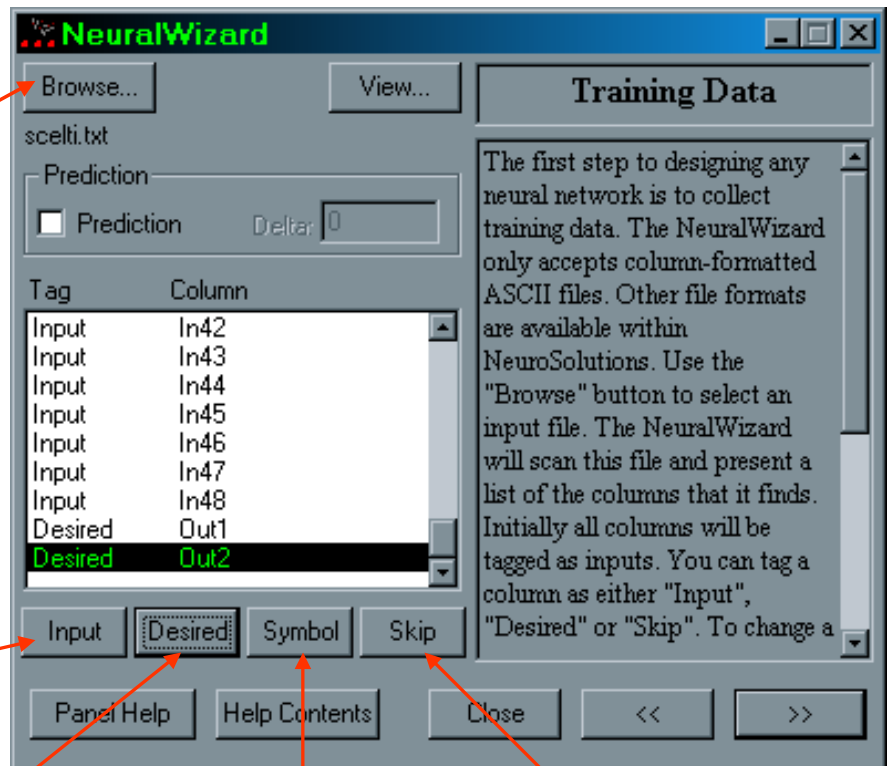
Tipologia dati (per colonne, cioè canali) nel file di training

Canale di input

Risposta desiderata

L'input è un simbolo

Canale ignorato



NeuralWizard – passo 3

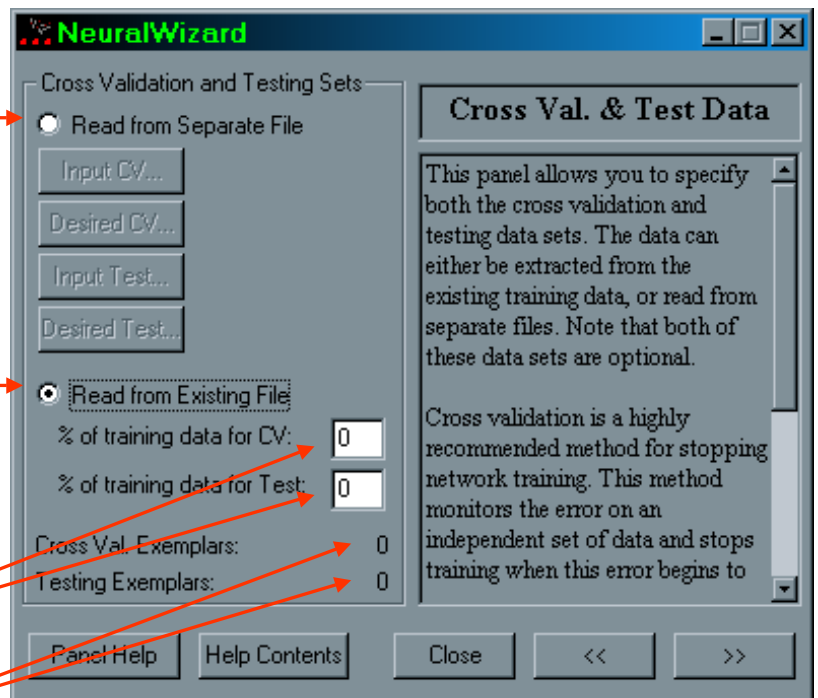
Cross Validation (CV) e Test:

Legge i pattern per CV e test da un file diverso da quello di training.

Utilizza una % dei pattern di partenza per CV e test.

Percentuali scelte

N° di pattern in base alla %



NeuralWizard – passo 3

Cross Validation (CV):

- insieme di pattern per il cross validation utilizzato per evitare l'*over-training* (la rete memorizza singoli esempi e non i dati nel loro complesso \Rightarrow non è in grado di generalizzare) \Rightarrow arresta l'addestramento

Se i pattern a disposizione sono pochi si utilizzano criteri alternativi per terminare l'addestramento:

- l'errore quadratico medio raggiunge una soglia
- l'errore quadratico medio non decresce in modo significativo

NeuralWizard – passo 3

Test:

Utilizzato per verificare le prestazioni al termine dell'addestramento.

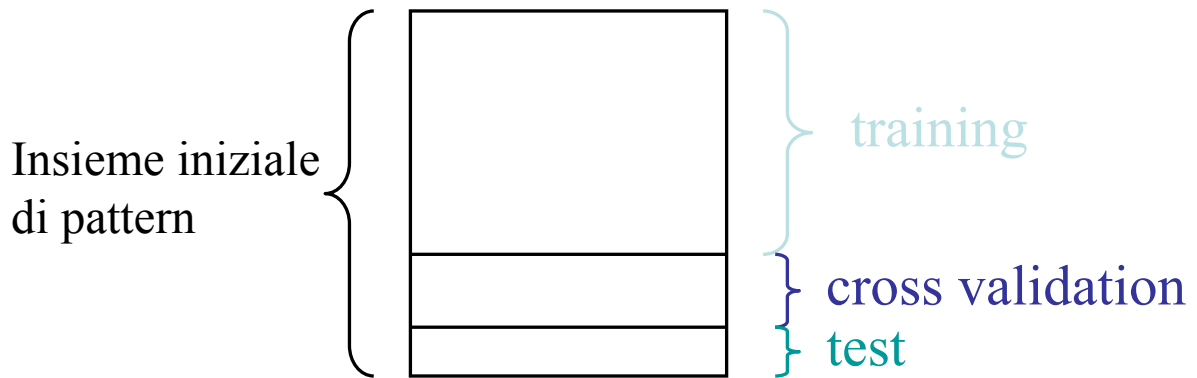
I pesi sinaptici vengono congelati.

Si può confrontare la risposta della rete con quella desiderata.

NeuralWizard – passo 3

Cross Validation (CV) e Test:

- a) pattern scelti dall'insieme iniziale, quello del passo 2 (\Rightarrow esclusi dal training); sono presi i pattern nelle posizioni finali



- b) pattern di un file separato

NeuralWizard – passo 4

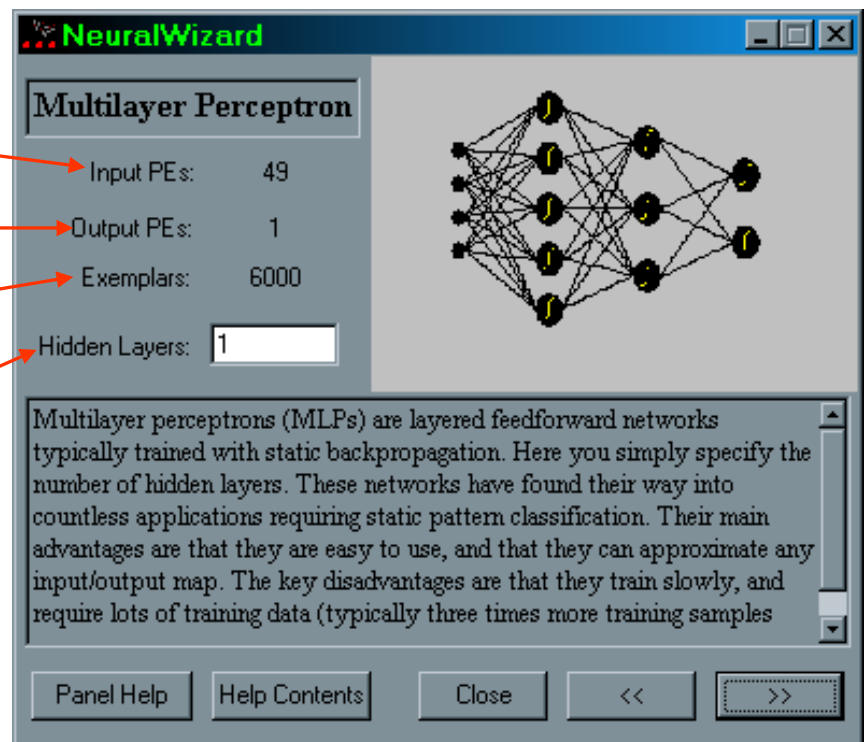
Topologia:

N° unità input

N° unità output

N° training record

N° strati nascosti



Il pannello cambia in base alla tipologia di rete.

NeuralWizard – passo 4

Per il MLP il n° di strati nascosti è l'unico parametro del pannello.

Se il problema non è particolarmente difficile, si può partire con il valore di default di un solo strato nascosto.

NeuralWizard – passo 5

Configurazione strati nascosti:

N° unità dello strato

Funzione di trasferimento

Criterio aggiornamento pesi

Parametro/i legati alla regola
(step size η o tasso di apprendimento - movimento sulla superficie dell'errore)
(momentum α – inerzia nella modifica dei pesi)

NeuralWizard

Hidden Layer #1

Processing Elements: 39

PE Transfer: TanhAxon

Learning Rule: Momentum

Step Size: 1.000000

Momentum: 0.700000

description of the learning rules that are available within the various Gradient Search components. The NeuralWizard selects the momentum component as default. In searching with the momentum component there are two parameters to be selected: the step size and the momentum. The NeuralWizard provides a default value for the learning rates. Be ready to modify this selection if you see that learning is unstable or very slow.

Panel Help Help Contents Close << >>

NeuralWizard – passo 5

Configurazione strati nascosti:

- *Numero di neuroni nascosti* - influenza le prestazioni della rete; scelta in base alla complessità del mapping fra input e output; determinato sperimentalmente.

In NeuroSolutions: è proporzionale al numero di ingressi (in genere ne determina più del necessario).

Ottimo: numero minimo di neuroni nascosti che risolve il problema per avere una buona generalizzazione sui nuovi dati.

NeuralWizard – passo 5

Funzioni di trasferimento:

Assone	Output	Caratteristiche
TanhAxon	-1..1	Non linearità principale
SigmoidAxon	0..1	Caratteristiche generali di TanhAxon
LinearTanhAxon	-1..1	Approssimazione di Tanh
LinearSigmoidAxon	0..1	Approssimazione di Sigm
SoftMaxAxon	0..1	$\Sigma_{output}=1$ (utile per classificazione)
BiasAxon	∞	Linear axon con soglia e pendenza adattabili
LinearAxon	∞	Linear axon con soglia adattabile
Axon	∞	identità come funzione di trasferimento

NeuralWizard – passo 5

La non linearità degli strati nascosti fornisce la capacità di apprendere problemi complessi.

Altri parametri

regola di apprendimento – utilizzata per calcolare l'aggiornamento dei pesi. Obiettivo: variare i pesi per trovare il minimo assoluto dell'errore. Strategie:

- step
- Momentum
- Quickprop
- DeltabarDelta

Momentum consigliata per utenti non esperti (meno rapida delle altre ma più stabile)

NeuralWizard – passo 5

Normalizzazione:

- NeuralWizard comunica automaticamente a NeuroSolutions di scalare e shiftare l'input nel range della funzione di trasferimento del primo strato.
- Pre-processing detto *normalizzazione*.

NeuralWizard – passo 6

Configurazione strato di uscita:

N° unità (fissato da NeuroSolution)

Funzione di trasferimento

Criterio aggiornamento pesi

Output Layer

Processing Elements: 26

PE Transfer: TanhAxon

Learning Rule: Momentum

Step Size: 0.100000

Momentum: 0.700000

This panel is used to specify the parameters a layer of processing elements (PEs). NeuroSolutions simulations are vector based for efficiency. This implies that each layer contains a vector of PEs and that the parameters selected apply to the entire vector. The parameters are dependent on the neural model, but all require a nonlinearity function to specify the behavior of the PEs. In addition, each layer has an associated learning rule and learning parameters. The number of PEs and learning

Panel Help Help Contents Close << >>

NeuralWizard – passo 6

Funzione di trasferimento in uscita in base al tipo di problema:

Problema	Descrizione	Assone in output
Classificazione multipla	classificazione da 1..N	SoftMaxAxon
Clasificazione binaria	classificazione con un solo canale in output	TanhAxon o Sigmoide
Regressione	la risposta desiderata è una funzione continua dell'ingresso	Axon Bias Axon LinearAxon

NeuralWizard – passo 6

Normalizzazione:

- opera come per lo strato di input
- la risposta è normalizzata per rientrare nel range della funzione di trasferimento dello strato di uscita

Denormalizzazione

NeuralWizard imposta automaticamente la normalizzazione dell'input e dell'output nel range delle funzioni di trasferimento.

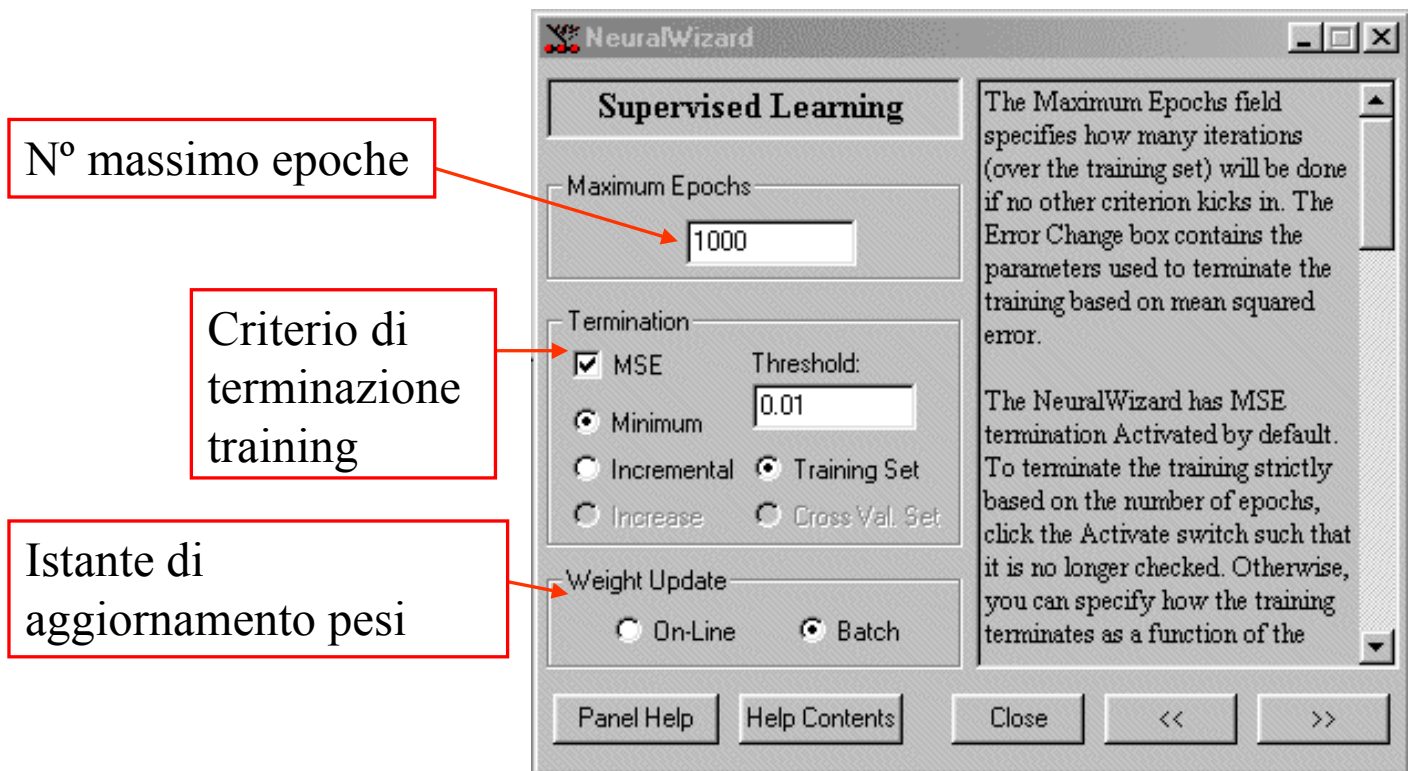
Meccanismo di denormalizzazione per osservare i valori originali dei dati.

Ogni insieme di dati genera il proprio file di normalizzazione per memorizzare scalamento ed offset.

La denormalizzazione non influenza il training.

NeuralWizard – passo 7

Addestramento supervisionato:



NeuralWizard – passo 7

- *Numero massimo di epoche* - per molti problemi il valore 1000 di default è sufficiente
- *Criterio di terminazione*: determina quando arrestare l'addestramento
 - dal solo numero di epoche (si disabilita MSE)
 - dall'errore quadratico medio tra risposta desiderata per l'input (pattern di training o CV) e risposta ottenuta:
 - *minimum** (quando il MSE scende sotto la soglia)
 - *incremental** (quando la differenza di MSE tra due iterazioni consecutive è inferiore alla soglia)
 - *increase* - (quando il MSE del CV set si incrementa della soglia specificata)

*: quando non si usa CV

NeuralWizard – passo 7

Aggiornamento dei pesi: stabilisce quando effettuare l'aggiornamento dei pesi

- *on-line* - dopo avere presentato in input un esemplare di addestramento
 - ☺ veloce
 - ☹ maggiore attenzione ai parametri di configurazione
- *batch* - dopo avere presentato l'intero training set ⇨ pesi mediati sull'intero training set
 - ☺ consigliato per utenti non esperti perché più stabile

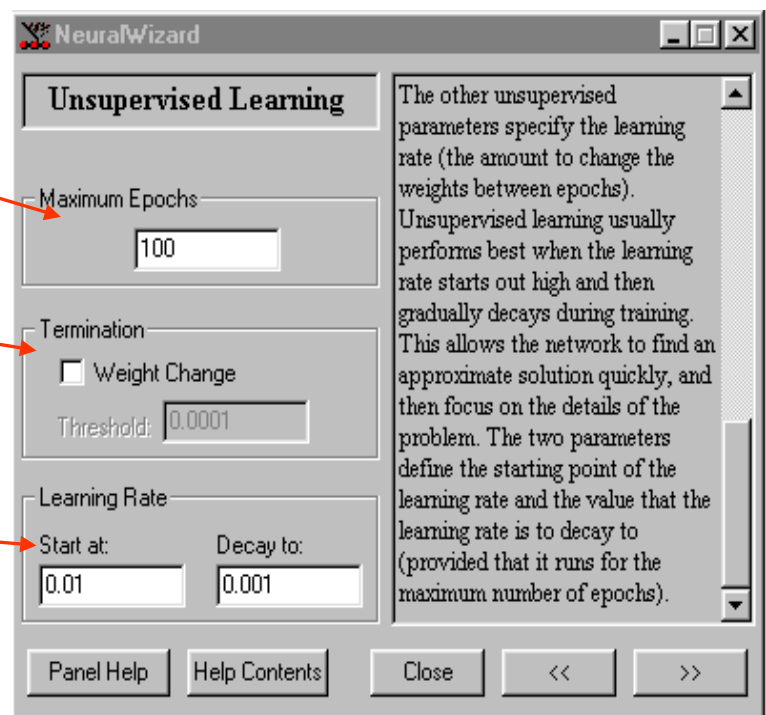
NeuralWizard – passo 7

Addestramento non supervisionato:

N° massimo epoche

Terminazione in base al cambiamento dei pesi

Learning rate variabile, a partire da un valore massimo ad uno minimo



NeuralWizard – passo 7

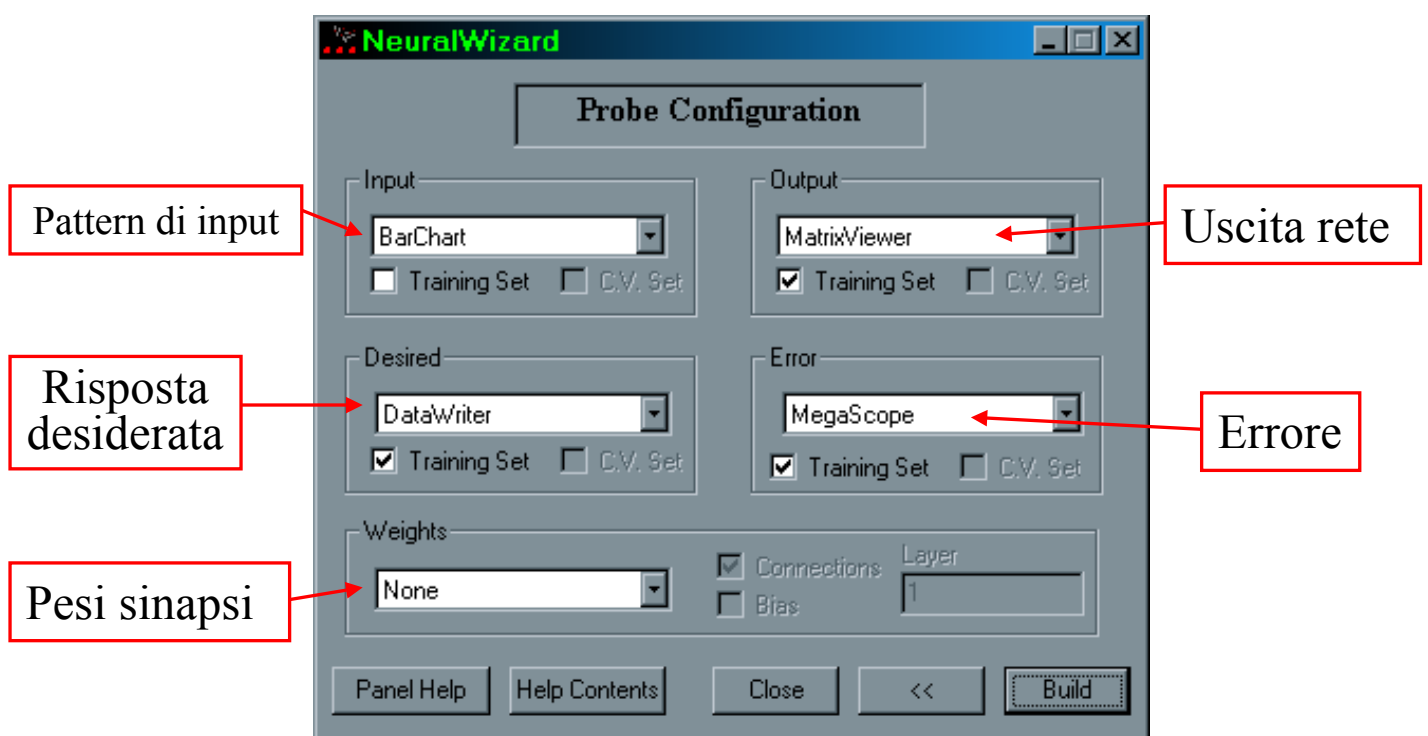
Reti ibride: hanno una parte supervised e una parte un supervised.

L'addestramento è effettuato separatamente.

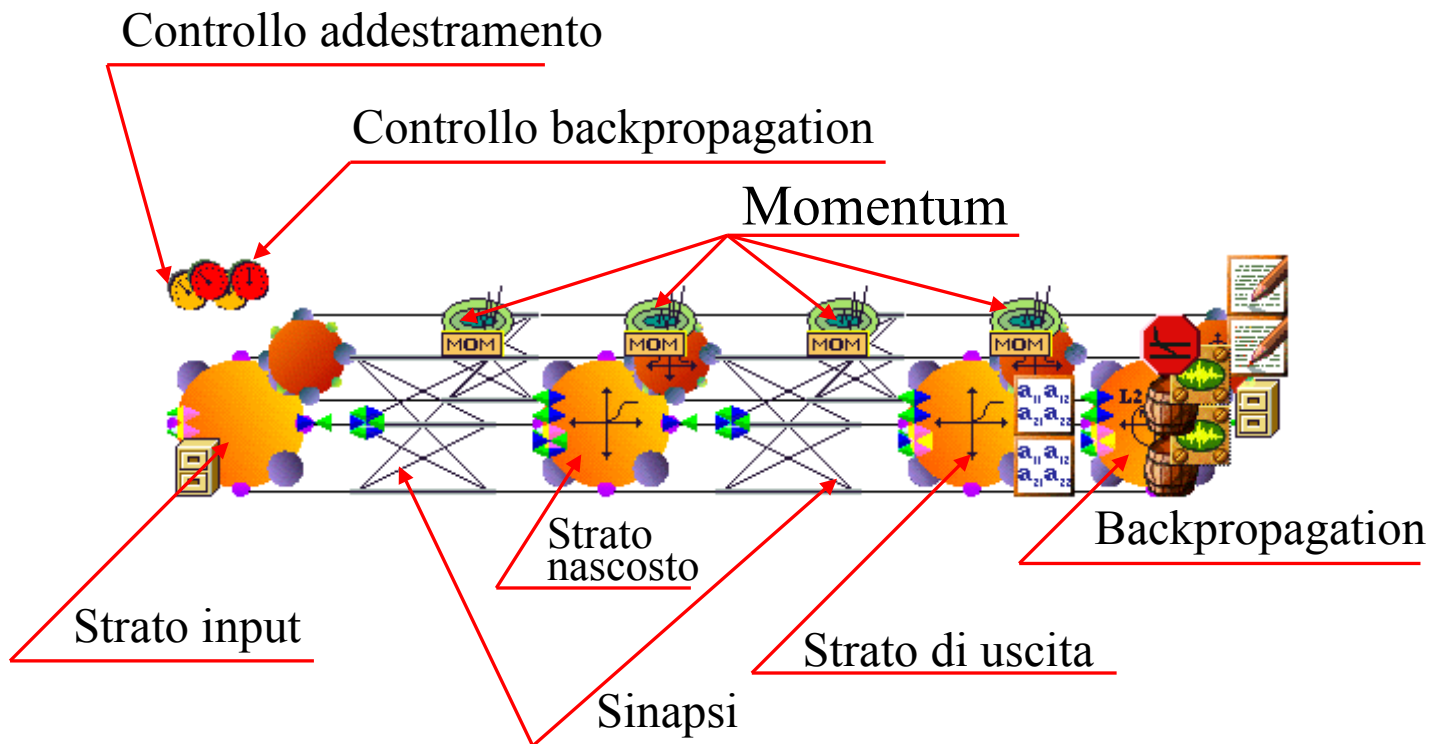
Il criterio che controlla la durata dell'addestramento è separato.

NeuralWizard – passo 8

Configurazione strumenti di visualizzazione:



Rappresentazione di una rete neurale



Rappresentazione di una rete neurale

Con il tasto destro del mouse si può accedere alle proprietà dei componenti

Data Writer – output (video o file) dati in tempo reale

Mega Scope (grafico)

Data Storage (buffer dati)

