Metodi Computazionali 2

Francesca Birocchi

12 febbraio 2008

Il mio progetto si è articolato in tre fasi:

1. implementazione dell'algoritmo delle *Radial Basis Function, RBF* nella classe `Parametrization`;
2. implementazione della classe `Clustering`, classe che viene poi utilizzata all'interno dell'algoritmo delle RBF;
3. confronto tra le reti neurali RBF e MLP;

1 Radial Basis Function, RBF

Le reti a simmetria radiale (*Radial Basis Function*, RBF) costituiscono una particolare famiglia delle reti *feed-forward multistrato*, così chiamate perché il flusso di informazioni tra i neuroni procede in modo unidirezionale. Si tratta quindi di reti parzialmente connesse, nelle quali i neuroni sono organizzati in *strati* (*layer*) ordinati che, nel caso delle RBF, sono solamente tre: uno *strato d'ingresso*, che riceve gli stimoli dall'esterno, uno *strato nascosto* (*hidden layer*) e uno *strato di uscita*, che fornisce la risposta della rete.

L'output di una rete neurale a simmetria radiale è:

$$y(\mathbf{x}) = \sum_{i=1}^{m} \beta_i \ \phi(\|\mathbf{x} - \mathbf{c}_i\|) \tag{1}$$

dove m è il numero di neuroni nello strato nascosto, c_i è il centro corrispondente all'i-esimo neurone, β_i sono i pesi che connettono l'*hidden layer* con lo strato di uscita (*output*) e $\phi(\cdot)$ è la funzione di attivazione.

L'output delle RBF, dunque, è una combinazione lineare di m funzioni non lineari $\phi(\cdot)$ che dipendono dalla distanza tra un vettore $\mathbf{x}$ ed un vettore $\mathbf{c}_i$, definito centro dell'i-esima funzione radiale. La distanza $d(\mathbf{x}, \mathbf{c}_i)$ usata più spesso, e che ho scelto per l'implementazione dell'algoritmo della RBF, è

quella euclidea, per cui si pone:

$$d(\mathbf{x}, \mathbf{c}_i) = \|\mathbf{x} - \mathbf{c}_i\| = \sqrt{\sum_{i=1}^{m} (\mathbf{x} - \mathbf{c}_i)^2} \tag{2}$$

Inoltre, come dice il nome stesso, la funzione di attivazione delle RBF è una funzione a simmetria radiale; nell'implementazione dell'algoritmo delle RBF ho scelto la funzione maggiormente usata, quella gaussiana:

$$\phi(\|\mathbf{x} - \mathbf{c}_i\|) = exp\Big(-\frac{1}{2}\frac{\|\mathbf{x} - \mathbf{c}_i\|^2}{\sigma_i^2}\Big) \tag{3}$$

dove σ_i è l'ampiezza dell'i-esima funzione base radiale.
È importante notare che le funzioni radiali sono funzioni localizzate nel senso che:

$$\lim_{\|x\| \to \infty} \phi(\|\mathbf{x} - \mathbf{c}_i\|) = 0$$

ossia, cambiando i parametri di un solo neurone, si ottengono piccole variazioni per i valori degli input che sono molto lontani dal centro di quel neurone.

L'algoritmo che ho utilizzato all'interno del mio programma (nella funzione di output) si ottiene unendo la (1) e la (3):

$$y(\mathbf{x}) = \sum_{i=1}^{m} \beta_i \; exp\Big(-\frac{1}{2}\frac{\|\mathbf{x} - \mathbf{c}_i\|^2}{\sigma_i^2}\Big) \tag{4}$$

Per quanto riguarda il *training* delle reti RBF, ci sono tre tipi di parametri che devono essere determinati in modo da ottimizzare il fit tra $\phi(\cdot)$ e i dati: il vettore dei centri $\mathbf{c}_i$, i pesi β_i e le ampiezze σ_i. Per la stima dei parametri si possono seguire due strategie alternative; quella che ho scelto di implementare nel mio codice è una *procedura di stima 2 stadi.* Nel primo stadio, attraverso un metodo di apprendimento non supervisionato, vengono determinati i centri e le sigma delle funzioni base, mentre nel secondo stadio, si stimano i pesi β_i con un procedimento di apprendimento supervisionato, utilizzando le stime dei parametri delle funzioni base ottenute in precedenza. I metodi di apprendimento non supervisionato, usati per stimare i parametri che definiscono le funzioni base, sono diversi. In un primo momento ho implementato un algoritmo banale che, data una distribuzione uniforme di dati (input) in un'intervallo, calcolasse i centri delle funzioni base; in seguito però, ho deciso di stimare i parametri utilizzando *l' algoritmo di clustering delle K-medie* e, data la complessità del codice che ne risultava, di costruire una classe di `Clustering` (da utilizzare poi all'interno della classe `RBF`).

Una volta stimati i centri delle funzioni di base, bisogna determinare le ampiezze delle gaussiane. Anche in questo caso esistono numerosi algoritmi, tra i quali ho scelto di implementare l'algoritmo dei *P-nearest-neighbour*: si sceglie un numero P e, per ogni centro, si trovano i P centri più vicini. Il valore della j-esima ampiezza è quindi dato dalla radice quadrata della media della distanza tra il j-esimo centro e i P centri più vicini:

$$\sigma_j = \sqrt{\frac{1}{P} \sum_{i=1}^{P} (c_j - c_i)^2} \tag{5}$$

Stimati i parametri delle funzioni base, i pesi β_i vengono infine determinati mediante un procedimento di apprendimento non supervisionato contenuto nella classe `Minimization` (implementata durante le lezioni).

Il codice

Ho implementato la classe `RBF` all'interno delle classe `Parametrization`, restringendomi al caso unidimensionale, nel seguente modo.

Membri pubblici

- **Rbf (const *Data&*)**: costruttore che riceve in input un membro della classe `Data`.
- **~Rbf ()**: distruttore
- **vector<double> output (*vector<double>*)**: riceve come input il vettore che contiene i dati d'ingresso e restituisce, sotto forma di vettore, la risposta della rete (vedi equazione (4)).

Membri privati

- **Rbf ()**: costruttore
- **Rbf (const *Rbf&*)**: costruttore di copie
- **void read ()**: legge da un file i parametri della rete neurale RBF, in particolare:
  - il numero m di neuroni nell'*hidden layer* (`n[1]`);
  - il numero di *nearest neighbours* (`nnn`), che utilizzo poi per stimare le sigma delle funzioni base;
  - il tipo di procedimento che voglio utilizzare per stimare i centri delle funzioni di base (`type`): posso scegliere tra un procedimento banale o il metodo delle K-medie;

- **double activation_function (*size_t*, *double*)**: riceve come input un intero (`size_t i`), che indica quale degli m neuroni dell'hidden layer sto considerando, e un double (`double x`), che rapprensenta uno dei dati di input. Questo metodo della classe restituisce il valore della funzione di attivazione relativa all'input x e all'i-esimo neurone dello strato nascosto.

- **void assign_centres (*size_t*, *vector<double>*)**: stima i centri delle funzioni base, ricevendo in input il tipo di procedimento di apprendimento da utilizzare (`type`). Non restituisce nulla in quanto inizializza il membro della classe vector<double> xc, il vettore dei centri delle m gaussiane.

- **void sigma2 ()**: inizializza vector<double> sigmasq, un vettore membro della classe `RBF` che contiene le ampiezze delle m gaussiane. Non necessita di input in quanto utilizza membri della classe `RBF`: il numero dei *nearest neighbour* e il vettore dei centri. L'algoritmo usato da questo metodo per stimare le ampiezze delle gaussiane di base è quello dei *P-nearest neighbour*, spiegato in precedenza.

Attributi pubblici

- **vector<double> input**: vettore degli input

Attributi privati

- **size_t n[3]**: vettore di tre elementi che contiene, in ordine: il numero di input, il numero di neuroni nell'*hidden layer* e il numero di output.

- **vector<double> xc**: vettore che contiene i centri delle funzioni base

- **vector<double> sigmasq**: vettore che contiene le ampiezze delle gaussiane

- **size_t nnn**: numero dei *nearest neighbour*

- **size_t type**: indica il tipo di procedimento di apprendimento da utilizzare per la stima dei centri

2 Algoritmo di clustering delle K-medie

Il clustering consiste nel dividere degli oggetti, definiti da alcune proprietà numeriche, in gruppi (*clusters*) tali che gli oggetti che appartengono allo stesso gruppo siano più simili degli oggetti appartenenti a gruppi differenti. Esistono numerosi metodi di clustering. Quello che ho implementato io nella classe `Clustering` è chiamato *clustering delle K-medie.* Il criterio attraverso

cui questo algoritmo stima la somiglianza tra gi oggetti è la distanza euclidea: minore è la distanza tra due oggetti, maggiore è la loro somiglianza. Noto a priori il numero dei gruppi (*clusters*), il *clustering delle K-medie* minimizza la totale distanza Euclidea al quadrato, E, che presenta la seguente forma:

$$E = \sum_{i=1}^{N} \sum_{j=1}^{m} M_{ij} \|\mathbf{x}_i - \mathbf{c}_j\| \tag{6}$$

Dove $\mathbf{x}_i$ con $i = 1, 2, \ldots N$ sono gli N oggetti, $\mathbf{c}_j$ con $j = 1, 2, \ldots m$ sono gli m centri e M_{ij} è una matrice di zeri con un solo 1 per riga che identifica il gruppo a cui un dato oggetto appartiene. Per minimizzare questa espressione, bisogna porre a zero la derivata parziale rispetto ad ogni centro; svolgendo i calcoli scopriamo che, nella condizione che minimizza la totale distanza euclidea, abbiamo:

$$\mathbf{c}_j = \frac{1}{NP} \sum_{i=1}^{NP} \mathbf{x}_i \tag{7}$$

dove *NP* è il numero di oggetti che appartengono al *cluster* del centro j-esimo.

Il clustering delle K-medie può essere usato per stimare i centri delle funzioni RBF; la versione che ho implementato nella classe di `Clustering` si svolge in tre fasi:

1. si scelgono come centri delle funzioni base m elementi scelti in modo casuale tra i dati di entrata
2. si costituiscono i *clusters*, assegnando ogni elemento tra gli input al centro più vicino
3. si calcolano nuovi centri facendo la media tra i valori degli elementi di ogni *cluster* secondo la fomula (7)

Gli ultimi due punti si ripetono fino a quando la somma delle distanze tra ogni punto dei dati e ogni centro non si stabilizza.

Il codice

Ho implementato la classe `Clustering` nel seguente modo.

Membri pubblici

- **~Clustering ()**: distruttore
- **Clustering (*size_t*)**: costruttore che inizializza `ncentres`, il numero di centri

- **void sort (*vector<double>&*)**: riordina, in ordine crescente, gli elementi del vettore che riceve in input
- **void assign_points (*vector<double>*)**: riceve in input un vettore e assegna ogni elemento del vettore al centro piú vicino. Non restituisce nessun valore perché inizializza i membri privati `vector<size_t> np` e `vector<double> boundering`
- **void assign_centres ()**: assegna i centri secondo la formula (7), spiegata in precedenza
- **void generate_centres (*vector<double>*)**: inizializza il vettore dei centri prendendo come valori m elementi, scelti in modo casuale, del vettore che riceve in input. Controlla inoltre che non siano stati scelti due centri uguali; se ciò avviene ripete il procedimento fino a quando i centri non sono tutti diversi tra di loro
- **double objective_function (*vector<double>*)**: riceve in input un vettore e, partendo da questo, calcola il valore dell'equazione (6)
- **bool control_distance (*double*, *double*)**: controlla se la distanza tra i due `double` che riceve in input è minore di un certo valore; in caso affermativo restituisce `true`, se no `false`
- **void kmeans (*vector<double>*)**: riceve un vettore con gli input e stima i centri mediante *l'algoritmo di clustering delle K-medie*, illustrato in precedenza
- **void basic (*vector<double>*)**: riceve un vettore con gli input e stima i centri mediante un procedimento banale che consiste nel distribuire uniformemente i centri nell'intervallo in questione. Questo metodo, naturalmente, può essere utilizzato con buoni risultati solo se i dati sono distribuiti uniformemente sull'intervallo

Membri protetti

- **Clustering ()**: costruttore
- **Clustering (const *Clustering&*)**: costruttore di copie

Attributi pubblici

- **size_t ncentres**: numero dei centri(m)
- **vector<double> centres**: vettore che contiene i centri delle m gaussiane

Attributi privati

- **vector<size_t> np**: vettore che nella funzione `assign_points` tiene conto di quanti punti sono stati assegnati ad ogni centro
- **vector<double> np**: vettore la cui j-esima componente è la somma di tutti i punti assegnati al centro j-esimo

3 Confronto fra reti RBF e reti MLP

Le *Radial Basis Function* e le *Multilayer Perceptron*, pur essendo entrambe reti *feed-forward*, presentano numerose differenze:

- Le reti MLP considerano come funzioni base, funzioni *sigmoidali* con argomento il prodotto interno tra i vettori $\mathbf{x}$ e β_i (pesi), mentre le reti RBF considerano funzioni *radiali* con argomento la distanza (euclidea) tra i vettori $\mathbf{x}$ e $\mathbf{c}_i$ (centri); la funzione di attivazione delle MLP, dunque, è costante su iperpiani di R_d, mentre quella delle RBF su iperellissoidi di R_d.
- L'architettura di una rete MLP è più complessa di quella di una rete RBF, infatti ci possono essere più di uno strato nascosto; è uno strumento versatile ma può presentare dei problemi di minimi locali. Le reti RBF, invece, una volta fissati i centri, si riducono a una stima lineare
- I tempi di training di una rete MLP sono in genere più lunghi di quelli di una rete RBF: per queste ultime si usa infatti un algoritmo a due stadi, mentre per le reti MLP si utilizza un processo simultaneo di stima per tutti i pesi

Il codice

Alla luce di queste differenze ho implementato un programma (`main.cpp`) che, utilizzando le classi viste a lezione (`Parametrization, Minimization`, etc), prende un file contenente 30 dati, fa il training di una delle due reti neurali e poi usa i risultati ottenuti per stimare il valore della funzione *seno*. Confrontando i grafici ottenuti usando, a parità di parametri, i due tipi di reti neurali, ho potuto notare che:

- Per bassi valori dei parametri (`npar=3`) i due fit sono entrambi imprecisi;
- all'aumentare del numero dei parametri il fit della rete RBF è migliore (oltre che più veloce) di quello della rete MLP che, soprattutto alle estremità, non segue la curva della funzione.

Non ho potuto assegnare allo strato interno della rete RBF un numero di neuroni superiore a sei perché, in questo caso, l'algoritmo di `Clustering` presentava dei problemi. Per `ncentres=7`, per esempio, ho notato che c'era un centro al quale non venivano assegnati punti nel *clustering*; ciò faceva girare l'algoritmo che esegue il metodo delle K-medie all'infinito. Questo problema è sicuramente legato al rapporto tra il numero dei dati e il numero di neuroni dell'*hidden layer*: provando con un numero maggiore di dati (60) questo problema non si è verificato con `ncentres=7`, ma con `ncentres=10`.

Nelle pagine seguenti sono mostrati i fit ottenuti usando per ogni valore dei parametri, sia la rete naurale RBF sia quella MLP:

Figura 1: MLP con un hidden layer, 3 neuroni nello strato nascosto

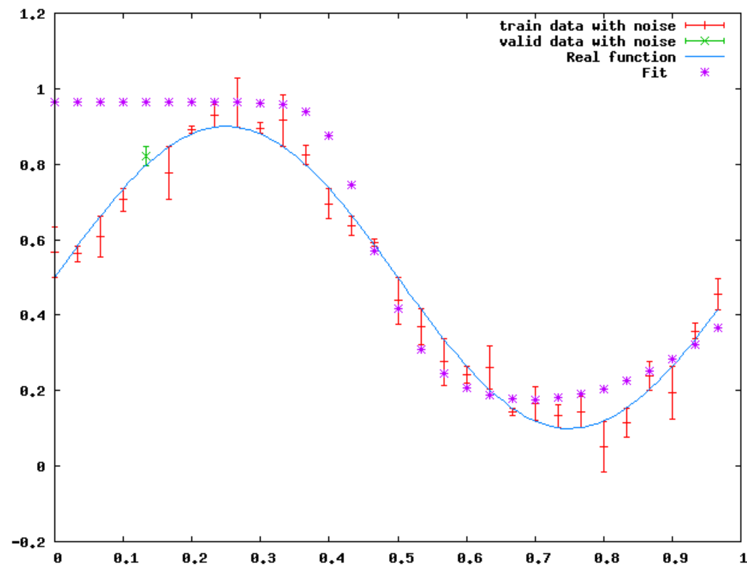

Figura 2: RBF con 3 neuroni nello strato nascosto

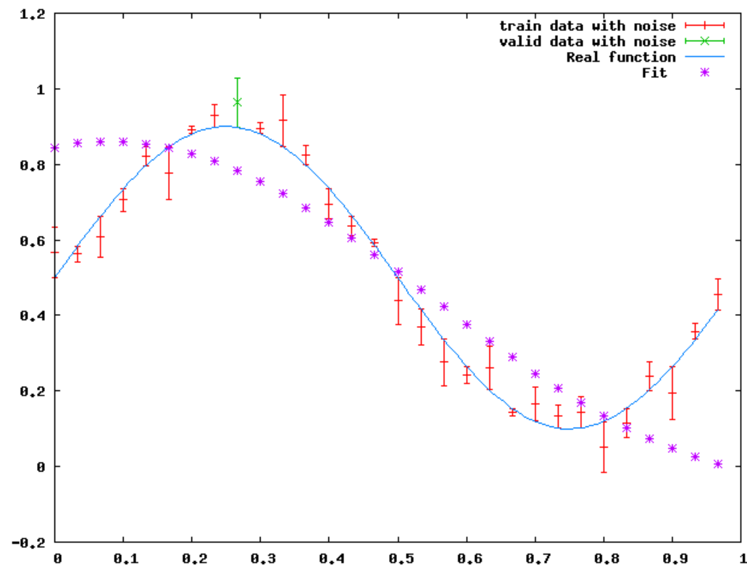

Figura 3: MLP con un hidden layer, 4 neuroni nello strato nascosto

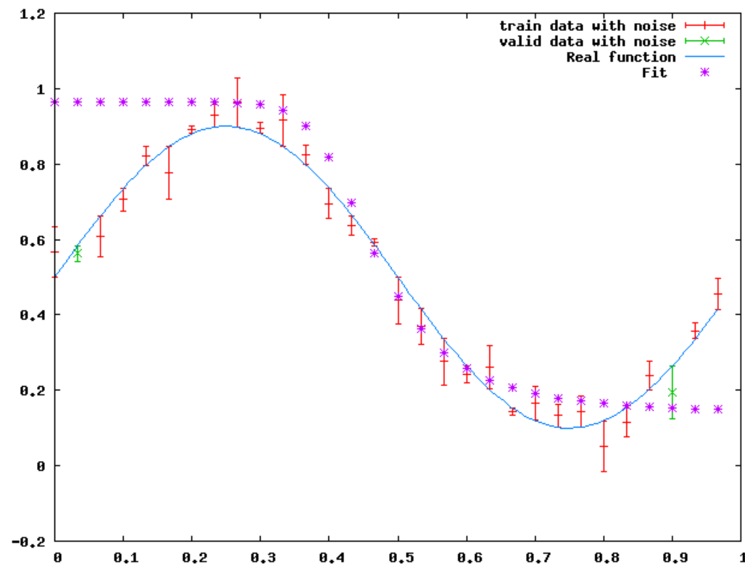

Figura 4: RBF con 4 neuroni nello strato nascosto

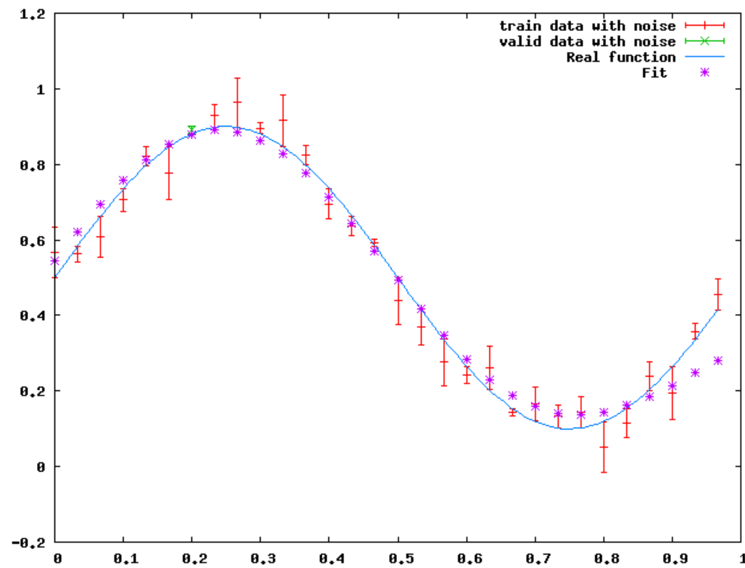

Figura 5: MLP con un hidden layer, 5 neuroni nello strato nascosto

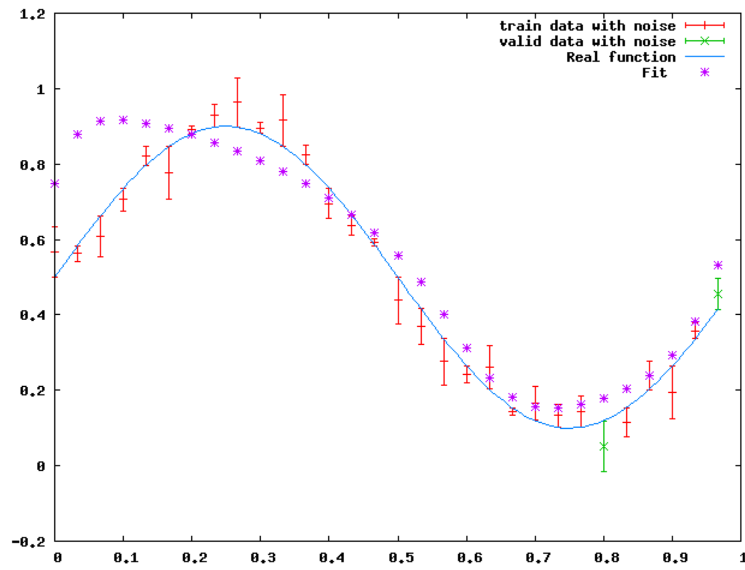

Figura 6: RBF con 5 neuroni nello strato nascosto

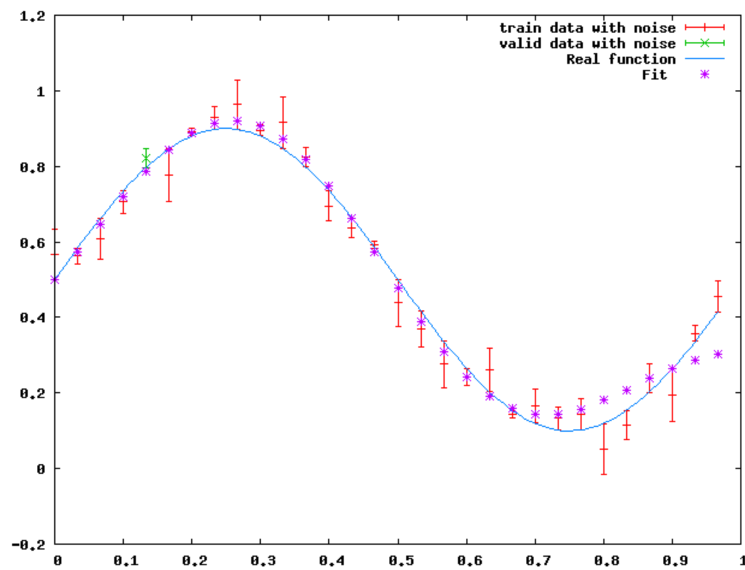

Figura 7: MLP con un hidden layer, 6 neuroni nello strato nascosto

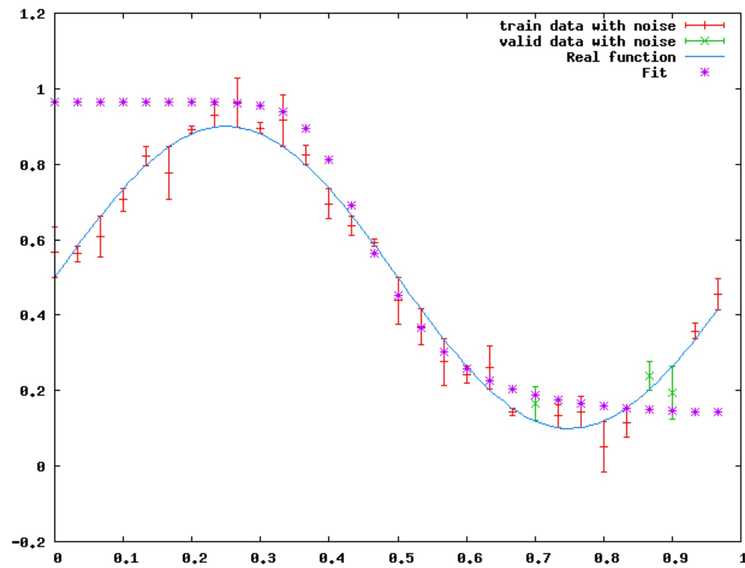

Figura 8: RBF con 6 neuroni nello strato nascosto