Il progetto Lightweight Neural Network ++: un'implementazione C++ di una rete feed forward con backpropagation

Luca Cinti <lucacinti@supereva.it>
Lorenzo Masetti <lorenzo.masetti@libero.it>

23 aprile 2004

Indice

1 Caratteristiche e finalità della libreria

Il progetto consiste nella realizzazione di una classe `C++` che implementa una rete neurale di tipo feed forward con backpropagation.

Con la libreria *Lightweight Neural Network++* è possibile creare reti neurali feed forward con un numero qualsiasi di livelli e di neuroni per livello.

Sono disponibili i due tipi principali di funzioni sigmoidali: la funzione logistica, che permette di avere una risposta tra 0 e 1, e la funzione tangente iperbolica, che fornisce una risposta tra -1 e 1.

Per l'addestramento della rete sono disponibili le principali tecniche studiate in questo campo: il normale training on-line con backpropagation, la tecnica del momento, il training in batch e la tecnica Super SAB (Self Adaptive Backpropagation). Tutti questi metodi sono combinabili a piacere, e sarà compito dell'utente della libreria scegliere quello più adatto al problema che vuole risolvere.

La libreria vuole essere del tutto generale, e per questo l'utente a cui si rivolge è il programmatore `C++` che ha bisogno di una classe che implementa una semplice rete neurale. Inoltre viene fornita una classe che implementa le tecniche standard per addestrare una rete neurale, rendendo così relativamente facile la scrittura di un programma che addestra una rete neurale. Inoltre il progetto contiene due tipi di interfacce grafiche, che permettono anche all'utente senza esperienza di programmazione di addestrare una rete molto facilmente.

Gli obiettivi della libreria sono principalmente l'efficienza (cioè la velocità nel calcolo degli output e nell'apprendimento) e l'occupazione solo della minor quantità possibile di risorse (memoria, processore e spazio sul disco). Questa finalità è dichiarata fin dal nome "lightweight".

La libreria è rilasciata come free software, e pertanto open source, sotto la licenza LGPL. Questa licenza impone che non solo un software derivato dalla libreria, ma anche qualunque programma che usi la libreria linkandola nel suo codice debba essere anch'esso software libero. In questo modo vorremmo incoraggiare lo sviluppo di software libero in questo campo.

Bisogna infatti sottolineare che non sono molti i progetti di software libero su reti neurali feed forward. Molte librerie con funzionalità simili (e sicuramente più avanzate) a *lightweight neural net++* sono progetti commerciali di software a sorgente chiuso. Il pacchetto di reti neurali del Matlab, un software commerciale venduto a un prezzo molto alto, necessita di risorse di calcolo consistenti anche per risolvere problemi relativamente semplici.

Un'altra valida libreria rilasciata sotto LGPL, altamente ottimizzata per la velocità, è *fann* (Fast Artificial Neural Network). L'autore, che ha realizzato la libreria come tesi di laurea, propone alcuni benchmark e li confronta anche con la libreria *Lightweight Neural Network* di Peter Van Rossum (di cui il nostro progetto è un'estensione). Per maggiori dettagli si può leggere la documentazione di *fann* ([4]). Abbiamo utilizzati i benchmark predisposti da questa libreria per confrontarla con il nostro progetto. I risultati di queste prove sono presentati nella sezione 12.

Il sito di riferimento per scaricare l'ultima versione è
http://lwneuralnetplus.sourceforge.net.

2 Cenni teorici

2.1 Reti Multilayer Perceptron

Le reti *Multilayer Perceptron* furono introdotte da Rumelhart e altri rielaborando un metodo di apprendimento già introdotto da Werbos nel 1977. Questo tipo di rete estende il concetto di Percptron, introdotto da Rosenblat nel 1958, inserendo uno o più strati nascosti di neuroni tra lo strato di neuroni di input e quello di output. Ogni neurone dello strato k-esimo è legato a tutti i neuroni dello strato precedente $(k-1)$-esimo attraverso un peso. In tale modo i neuroni di ogni strato eseguono in modo sincrono la media pesata degli output di tutti i neuroni dello strato precedente ed applicano ad essa la *funzione di attivazione* o *funzione sigmoidale*. Questa funzione è generalmente la funzione logistica

$$f(x) = \frac{1}{1+e^{-x}}$$

ma può anche essere la tangente iperbolica,

$$f(x) = \tanh(x) = \frac{e^x + e^{-x}}{e^x - e^{-x}}$$

la gaussiana, o semplicemente una funzione lineare. Strati diversi possono avere funzioni di attivazione diverse. L'andamento della funzione logistica è mostrato in figura 1, dove è mostrato che essa assume valori compresi tra 0 e 1 ed è già molto vicina a questi valori per $x < -10$ e $x > 10$. Sia una rete Multilayer Perceptron con n strati, indicati con $k = 1 \ldots n$ e supponiamo di dover riconoscere i pattern $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_P$ con uscite desiderate $\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_P$, possiamo indicare con

- $O_j^{p,n}$ l'output dell'j-esimo neurone nello strato n-esimo quando nello strato di input è posto il p-esimo pattern
- y_j^p la componente j-esima della risposta desiderata per il pattern p-esimo.

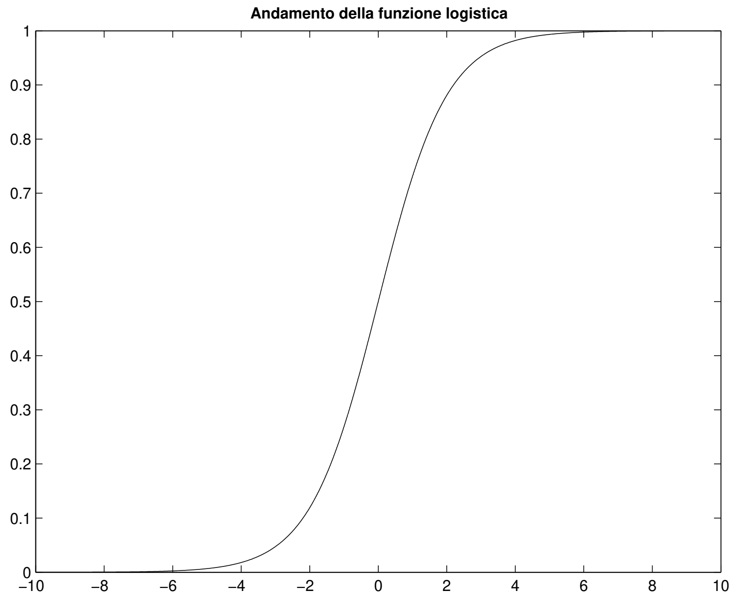

Figura 1: *Andamento della funzione logistica*

- W_{ij}^n il peso che lega il neurone i-esimo dello strato $n-1$-esimo e il neurone j-esimo dello strato n-esimo.

- t_j^n soglia associata al neurone j-esimo dello strato n-esimo.

Se si presenta quindi alla rete il pattern p-esimo si deve minimizzare lo scarto quadratico medio

$$E^p = \frac{1}{2}\sum_{j=1}^{m}(y_j^p - O_j^{p,n})^2$$

dove m è la dimensione del vettore di output e la funzione calcolata dalla rete, che dipende da tutti i pesi della rete, è

$$O_j^{p,n} = f(\sum_i W_{ij}^n O_i^{p,n-1} - t_j^n) =$$

$$= f(\sum_i W_{ij}^n\, f(\sum_k W_{ki}^{n-1} O_k^{p,n-2} - t_i^n) - t_j^n) =$$

$$= f(\sum_i W_{ij}^n\, f(\sum_k W_{ki}^{n-1}\, f(\sum_l W_{lk}^{n-1} \ldots - t_k^{n-2}) - t_i^{n-1}) - t_j^n)$$

Le soglie possono essere riassorbite con un neurone fittizio con uscita unitaria e legato ai neuroni dello strato successivo da pesi di valore t_j^n. Le soglie verranno quindi supposte nulle nella trattazione successiva per cui per lo strato di output posso scrivere

$$O_j^{p,n} = f(\sum_i W_{ij}^n O_i^{p,n-1})$$

Variando i pesi che legano il penultimo strato con l'ultimo di una quantità ΔW_{ij}^n, la variazione sullo scarto quadratico E^p è

$$\Delta E^p = \sum_{i,j} \frac{\partial E^p}{\partial W_{ij}^n} \Delta W_{ij}^n$$

Quindi considerando che $\frac{\partial E^p}{\partial W_{ij}^n} = \frac{\partial E^p}{\partial O_j^{p,n}} \frac{\partial O_j^{p,n}}{\partial W_{ij}^n}$, devo calcolare

$$\frac{\partial E^p}{\partial O_j^{p,n}} = -(P_j^p - O_j^{p,n})$$

$$\frac{\partial O_j^{p,n}}{\partial W_{ij}^n} = \left.\frac{\mathrm{d}\, f}{\mathrm{d}\, z}\right|_{z=\sum_i W_{ij}^n O_i^{p,n-1}} O_i^{p,n-1}$$

quindi si può usare come variazione dei pesi $\Delta W_{ij}^n = -\eta \frac{\partial E^p}{\partial W_{ij}^n}$, ovvero

$$\Delta W_{ij}^n = \eta (P_j^p - O_j^{p,n}) \left.\frac{\mathrm{d}\, f}{\mathrm{d}\, z}\right|_{z=\sum_i W_{ij}^n O_i^{p,n-1}} O_i^{p,n-1}$$

dove $\eta \in (0,1)$ è un parametro che regola la velocità dell'apprendimento detto *learning rate* In questo modo ho

$$\Delta E^p = -\eta \sum_{i,j} (\frac{\partial E^p}{\partial W_{ij}^n})^2 \leq 0$$

ovvero lo scarto quadratico diminuisce ad ogni aggiornamento dei pesi. L'ultimo problema da affrontare rimane quello di calcolare la derivata della funzione sigmoidale. Se si considera la funzione logistica osservo che

$$\frac{\mathrm{d}\, f}{\mathrm{d}\, x} = f(x)(1 - f(x))$$

In questo caso quindi l'aggiornamento dei pesi deve essere fatto in questo modo

$$\Delta W_{ij}^n = \eta (P_j^p - O_j^{p,n}) O_j^{p,n} (1 - O_j^{p,n}) O_i^{p,n-1}$$

Se invece la funzione sigmoidale è la tangente iperbolica $f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ ottengo

$$\frac{\mathrm{d}\, f}{\mathrm{d}\, z} = (1 - f(x))(1 + f(x))$$

e le formule per l'aggiornamento dei pesi dovranno essere modificate di conseguenza. Gli stessi ragionamenti possono essere applicati allo strato $n-1$-esimo ottenendo l'espressione per la variazione dei pesi che legano lo strato $n-2$-esimo all'$n-1$-esimo

$$\Delta W_{ki}^{n-1} = \eta \sum_l (P_l^p - O_l^{p,n})(1 - O_l^{p,n}) W_{li}^n O_i^{p,n-1}(1 - O_i^{p,n-1}) O_k^{p,n-2}$$

Le formule per l'aggiornamento dei pesi possono assumere una forma iterativa, largamente utilizzata per applicazioni pratiche, detta δ-rule

$$\Delta W_{ij}^m = \eta \delta_i^{p,m} O_j^{p,m} \tag{1}$$

dove

$$\delta_i^{p,m} = \begin{cases} O_i^{p,n}(1 - O_i^{p,n})(P_i^p - O_i^{p,n}) & \text{se} \quad m = n \\ O_i^{p,m}(1 - O_i^{p,m}) \sum_l W_{il}^{m+1} \delta_l^{p,m+1} & \text{se} \quad m < n \end{cases}$$

L'aggiornamento dei pesi, e quindi l'addestramento della rete, avviene dall'ultimo strato risalendo attraverso tutti gli strati fino all'input. Questo tipo di apprendimento prende il nome di *backpropagation*. Al contrario, il calcolo della della risposta della rete avviene ponendo il pattern sullo strato di input e propagato fino allo strato di output. Per questo si parla di reti *feed-forward*.

2.2 Apprendimento on-line, batch training e Super SAB

Con l'apprendimento finora considerato viene presentato alla rete un pattern e i pesi vengono aggiornati con l'espressione 1. Il procedimento deve essere iterato per tutti i pattern di apprendimento fino a raggiungere una risposta corretta da parte della rete. Questo modo di procedere è detto *apprendimento on-line*. In questo caso però se la distanza tra i pattern è ampia è possibile che i pesi della rete oscillino in modo eccessivo. Si deve allora usare un parametro η molto piccolo per smorzare le oscillazioni, rallentando di molto l'apprendimento. Può essere conveniente, quindi, utilizzare un approccio alternativo. Se abbiamo un insieme di pattern di N elementi si cerca di minimizzare lo scarto quadratico medio

$$E = \frac{1}{N} \sum_{p=1}^{N} E^p$$

quindi la modifica dei pesi viene fatta sulla media degli aggiornamenti calcolati per tutti i pattern. In simboli

$$\Delta W_{ij}^n = \eta \frac{1}{N} \sum_{p=1}^{N} ((P_j^p - O_j^{p,n}) O_j^{p,n}(1 - O_j^{p,n}) O_i^{p,n-1})$$

Il reale aggiornamento dei pesi avviene dopo aver presentato alla rete tutti i pattern di apprendimento, ovvero ogni *epoca*. Questo tipo di apprendimento prende il nome di *batch training* e, applicato ad alcuni problemi, fornisce una velocità di apprendimento superiore all'apprendimento on-line.

Un'ulteriore tecnica di apprendimento, che prevede di far muovere ogni peso verso il minimo ad una velocità diversa è la tecnica *Super SAB*, descritta più approfonditamente nella sezione 5.8.

3 Storia dello sviluppo della libreria

La libreria *Lightweight Neural Network++* deriva da un'analoga libreria scritta nel linguaggio *C*, *Lightweight Neural Network*. Questa libreria è stata inizialmente sviluppata dal professor Peter Van Rossum del dipartimento di Matematica dell'Università del New Mexico a Las Cruces, NM, USA ([3]). Cercando un'implementazione libera di una rete neurale feed forward, nell'ottobre scorso, abbiamo trovato questo progetto che era arrivato allora alla versione 0.5. La libreria aveva alcuni punti di forza ed un'interfaccia già organizzata con un approccio object-oriented, pur utilizzando un linguaggio non ad oggetti. Tuttavia le funzionalità erano ancora abbastanza limitate: forniva solo l'apprendimento on line con l'eventuale aggiunta di un momento, l'unica funzione di attivazione disponibile era quella logistica, una rete poteva essere salvata solo sotto forma di file testuale.

Grazie alla disponibilità del suo autore, abbiamo collaborato a questo progetto scrivendo il codice per memorizzare una rete sotto forma di file binario (riducendo così drasticamente la dimensione dei file contenenti le reti) e per l'addestramento in batch. Questa libreria è ancora disponibile all'indirizzo `http://lwneuralnet.sourceforge.net/`.

Per un primo periodo abbiamo continuato lo sviluppo della libreria originale ma poi abbiamo deciso di creare un nuovo progetto che utilizzasse un linguaggio di programmazione più avanzato, il `C++`, con il quale diventasse esplicita l'architettura ad oggetti del progetto. La classe *network* della libreria *lightweight neural network ++* quindi riutilizza molto del codice scritto per il progetto originale, ed offre un'interfaccia sostanzialmente analoga ma fornisce alcune caratteristiche in più:

- La possibilità di scegliere tra funzione logistica e tangente iperbolica come funzione sigmoidale
- L'addestramento Super SAB e la possibilità di combinarlo con il batch training

- La possibilità di combinare l'apprendimento in batch con il metodo del momento.

Inoltre il nostro progetto comprende anche una classe *trainer* che implementa gli algoritmi per addestrare una rete secondo le tecniche standard, due semplici interfacce grafiche per addestrare una rete e un framework per gestire in modo trasparente la lettura dei pattern da file.

4 Struttura del progetto

La libreria consiste di due classi principali: **network** che definisce una struttura dati che implementa una rete neurale feed forward, e **trainer** che consente di addestrare, validare e certificare una rete neurale su un set di pattern memorizzati su file. Le interfacce di queste classi sono descritte dettagliatamente nella documentazione della libreria.

La classe **trainer** ovviamente usa un oggetto della classe **network** che rappresenta la rete da addestrare.

Inoltre, per ottenere la massima flessibilità, la gestione della lettura dei pattern da file è gestita da un oggetto di classe **iomanage**. Quest'ultima è una classe astratta, cioè una classe che definisce i prototipi di due metodi: il primo permette di leggere da un file il numero di coppie input/output in esso contenute, e la dimensione degli input e dell'output (e quindi il numero di neuroni degli strati di input e di output di una rete che lavora su questi pattern); il secondo permette di leggere i valori dei pattern.

Non tutti i metodi di questa classe sono implementati, e quindi non possono essere istanziati oggetti di classe **iomanage**. È però possibile scrivere una classe derivata di **iomanage** che gestisca la lettura da un particolare tipo di file. Di default il trainer usa per la gestione della lettura da file un oggetto di classe **iomanagelwnnfann** (derivata da **iomanage**) che gestisce due tipi di formati per memorizzare i file: il formato utilizzato da *Lightweight Neural Network* ed il formato utilizzato dalla libreria *Fast Artificial Neural Network (fann)* ([4]). Inoltre viene fornita insieme alla libreria la classe **iomanagebinary**, che permette di salvare i pattern in formato binario. Ulteriori informazioni sul formato di questi file si trovano nella sezione 6.4.

Nelle prossime sezioni descriviamo le strutture utilizzate per mantenere le informazioni e le scelte fatte nell'implementazione dei metodi principali.

5 La classe network

5.1 Implementazione della struttura dati

Le principali informazioni da mantenere per implementare una rete neurale sono i valori dei pesi che connettono ogni neurone di uno strato con tutti i neuroni dello strato superiore.

Quindi la classe **network** contiene un campo di tipo intero di nome *no_of_layers* che contiene il numero di strati (si contano anche lo strato di input e quello di output, quindi una rete dovrà avere almeno due strati) e un array *layer* allocato dinamicamente di lunghezza *no_of_layers* di puntatori a strutture di tipo **layer_t**. Gli strati si contano a partire dallo strato di input (`layer[0]`), fino allo strato di output (che sarà `layer[no_of_layers-1]`). Si mantengono per comodità e per efficienza implementativa due puntatori diretti allo strato di input e allo strato di output.

Definizione della struttura layer_t

```
typedef struct
{
  int no_of_neurons;
  neuron_t *neuron;
}
layer_t;
```

Ogni livello contiene l'informazione sul numero di neuroni in quel livello e un puntatore a un array allocato dinamicamente di lunghezza pari al numero di neuroni che contiene i puntatori a strutture di tipo **neuron_t**.

Definizione della struttura neuron_t

```
typedef struct
{
  float output;
  float error;
  float *weight;
  float *delta;
  float *sumdeltas;
}
neuron_t;
```

Ogni neurone contiene tre array di numeri a virgola mobile (**float**) di lunghezza pari al numero di neuroni nello strato superiore. Questi array sono allocati solo per gli strati a partire dal primo strato nascosto (`layer[1]`).

Si può notare che l'implementazione della libreria non prevede l'uso di una soglia, neanche simulata da un neurone fittizio con uscita unitaria. Questa scelta, che permette una estrema semplificazione nell'implementazione della rete, non sembra comunque influire sulle prestazioni.

L'array *weight* contiene i valori dei pesi che collegano il neurone ai neuroni dello strato superiore, gli altri due array *delta* e *sumdeltas* ed il campo error sono usati nell'implementazione della δ-rule durante l'apprendimento.

Gli altri campi della classe sono: il parametro di apprendimento (*learning_rate*), la funzione di attivazione (*activation*), il parametro α per il momento (*momentum*), e l'errore globale della rete (*global_error*). Altri campi servono per l'apprendimento in batch (*no_of_patterns* memorizza il numero di pattern nell'epoca corrente) e per il Super SAB (descritti nella sezione 5.8).

5.2 Costruzione della rete e parametri di default

Per utilizzare una rete neurale bisogna istanziare un oggetto di classe **network** sul quale potranno poi essere richiamati i metodi che permettono di addestrare una rete o di usare una rete già addestrata per una simulazione.

Per costruire una nuova rete bisogna utilizzare un metodo costruttore. La classe fornisce tre costruttori diversi

Costruttore con numero di argomenti variabile Il primo parametro indica la funzione di trasferimento della rete (che deve essere specificata da una delle due costanti predefinite *network::LOGISTIC* o *network::TANH*), il secondo parametro indica il numero di strati della rete (compresi lo strato di input e lo strato di output, quindi dev'essere almeno due). Gli ultimi parametri indicano il numero di neuroni in ogni livello e devono essere tanti quanti sono gli strati. Ad esempio per istanziare una rete neurale che usa la funzione logistica, con 3 strati composti da 20, 30 e 10 neuroni rispettivamente si deve scrivere

```
network net(network::LOGISTIC, 3, 20, 30, 10);
```

Costruttore con vector Un'alternativa al primo tipo di costruttore è un costruttore che prende ancora come primo argomento la funzione sigmoidale e come secondo parametro un vettore di interi (**vector<int>**) che definisce il numero di neuroni in ogni strato (e, attraverso la sua lunghezza, il numero di strati). Nel secondo esempio viene costruito un puntatore a un oggetto di tipo network:

```
vector<int> layersvector = vector<int>(3);
layersvector [0] = 5; layersvector [1] = 9; layersvector [2] = 4;
network* net = new network(network::TANH, layersvector);
```

Costruttore da un file Il terzo tipo di costruttore prende il nome di un file su cui è salvata una rete neurale precedentemente creata. Un secondo parametro opzionale booleano, se settato a **false** permette di caricare una rete da un file in formato testuale (il vecchio formato utilizzato dalla libreria originale), ma è più conveniente utilizzare i file in formato binario che utilizzano meno spazio.

network mynet("trainednetwork.net ");

Quando una rete viene costruita per la prima volta i suoi pesi vengono inizializzati con numeri casuali in un intervallo $[-r; r]$ e quindi a media 0. Di default l'intervallo in cui variano i pesi inizializzati è $[-0.1, 0.1]$ se la funzione di attivazione è logistica e $[-0.05; 0.05]$ se la funzione di attivazione è la tangente iperbolica. Questa scelta, dettata dalla maggior sensibilità della tangente iperbolica, può comunque essere cambiata richiamando dopo la costruzione della rete il metodo **randomize (float range)** che inizializza i pesi a valori casuali nell'intervallo $[-range; range]$. Al momento della costruzione della rete vengono inizializzati anche gli altri parametri della rete: il learning rate a 0.3 se la funzione di attivazione è logistica e a 0.1 se la funzione di attivazione è la tangente iperbolica, il momento a 0. Queste impostazioni di default possono essere cambiate in seguito.

5.3 Implementazione della tecnica feed forward

Il campo *output* di ogni neurone contiene il suo output, dato, per lo strato l ($l = 1 \dots no\_of\_layers - 1$) e per ogni neurone $i = 0 \dots n_l - 1$ dalla formula

$$O_i^l = \sigma\left(\sum_{j=0}^{n_{l-1}-1} W_{ji}^l O_j^{l-1}\right) \tag{2}$$

dove con O_i^l indichiamo il valore dell'output memorizzato nel neurone i-esimo dello strato l (`layer[l].neuron[i].output`), con n_l indichiamo il numero di neuroni nelllo strato l (`layer[l].no_of_neurons`) mentre W_{ji}^l è il valore del peso che connette il neurone j-esimo dello strato $l - 1$ al neurone i-esimo dello strato l (nell'implementazione memorizzato in `layer[l].neuron[i].weight[j]`).

La funzione σ può essere sia la funzione logistica con $k = 1$

$$\sigma(z) = \frac{1}{(1 + e^{-z})}$$

sia la tangente iperbolica

$$\sigma(z) = \frac{(e^z + e^{-z})}{(e^z - e^{-z})}$$

Per calcolare l'output di una rete si salva il vettore di input nei campi *output* dei neuroni dello strato di input e poi si propaga il risultato con la tecnica feed forward fino allo strato di output, che conterrà nei campi *output* dei suoi neuroni il risultato della computazione.

5.4 Tecniche di ottimizzazione

Nell'implementazione di una libreria per le reti neurali l'efficienza è un punto critico. Infatti, anche se la velocità delle CPU e la disponibilità di memoria nei calcolatori continua aumentare, tanto che negli ultimi anni i personal computer economici forniscono anche agli utenti comuni possibilità fino a qualche tempo fa inimmaginabili (lo sviluppo della libreria e i vari test sono stati svolti su processori Athlon da 1.2 / 1.3 GHz e con 256 MB di memoria RAM), è pur sempre vero che in questo tipo di applicazioni le risorse sono utilizzate al massimo. Quindi un'ottimizzazione che riduca i tempi di computazione anche di un fattore 3 o 4 può essere cruciale.

Steffen Nissen sottolinea nella sua tesi di laurea ([4]) i possibili tipi di ottimizzazione in questo campo:

1. Ottimizzazione degli algoritmi
2. Ottimizzazione dell'architettura
3. Sfruttamento della cache e del principio di località
4. Tabulazione delle funzioni sigmoidali
5. In-lining del codice

L'ottimizzazione degli algoritmi è sicuramente il tipo di ottimizzazione che porta più benefici. Sostituendo un algoritmo di complessità $O(n^2)$ con uno di complessità $O(n)$ si otterrà sicuramente un beneficio maggiore di qualunque altro tipo di ottimizzazione. La libreria fornisce diversi tipi di algoritmi per l'addestramento, e spesso è difficile stabilire quale sia il più efficiente. Tuttavia in questo campo non c'è molto spazio per l'ottimizzazione.

Per quanto riguarda l'ottimizzazione dell'architettura, possiamo notare che avremmo potuto scegliere un approccio maggiormente orientato agli oggetti, ad esempio implementando una classe *layer* e una classe *neuron*. Tuttavia questo approccio, seguito ad esempio dalla *Jet's Neural Library* ([5]), può comportare degli svantaggi in termini di efficienza, dovuti alla scarsa località dei dati e all'overhead dovuto alla necessità di richiamare metodi in classi diverse.

Per sfruttare al meglio il principio di località è opportuno che i dati cruciali siano allocati sequenzialmente. Per questo motivo ad esempio la libreria *fann* ([4]) alloca tutti i pesi in un unico array. Comunque sembra che la nostra implementazione dia buoni risultati. Un approccio sequenziale è seguito anche in *lwnn++* per memorizzare i learning rates dell'apprendimento Super SAB.

Le ottimizzazioni chiave nell'implementazione di *Lightweight Neural Network++* riguardano gli ultimi due punti, ai quali dedichiamo i prossimi paragrafi.

5.4.1 Tabulazione delle funzioni sigmoidali

Un punto critico per la velocità di calcolo di una rete neurale è il calcolo della funzione di trasferimento. Questa funzione deve essere necessariamente non lineare, e quindi il suo calcolo con i metodi standard utilizzati da un calcolatore sarebbe altamente inefficiente. Un buon modo per calcolare le funzioni sigmoidali è la tabulazione in un numero sufficiente di punti delle funzioni per poi utilizzare il valore tabulato corrispondente al valore più vicino a quello che si vuole calcolare.

Di default le due funzioni sigmoidali vengono tabulate, al momento della compilazione della libreria, su 2000 punti distribuiti uniformemente (tra -12 e 12 per la funzione logistica, tra -3.11 e 3.11 per la tangente iperbolica). Il numero di punti in cui vengono tabulate le funzioni e i valori degli estremi possono essere cambiati in fase di compilazione. Questi valori vengono salvati in una matrice 2000×2 dove l'indice di colonna indica la funzione utilizzata (0 per la funzione logistica e 1 per la tangente iperbolica).

Se il valore x in cui si vuole calcolare la funzione di trasferimento non è compreso tra i due estremi di tabulazione, il valore della funzione viene approssimato con il suo valore limite (0 e 1 per la funzione logistica, -1 e 1 per la tangente iperbolica), altrimenti viene restituito il valore tabulato di indice

$$i = \lfloor \frac{x-a}{h} \rfloor \quad \text{con} \quad h = \frac{b-a}{n} \tag{3}$$

se la funzione è tabulata tra a e b in n punti.

Grazie a questo metodo di approssimazione il calcolo della funzione sigmoidale consiste in due operazioni elementari, due confronti e un accesso a un vettore.

5.4.2 In-lining del codice delle funzioni chiave

La chiamata a una funzione (o a un metodo) comporta un overhead nell'esecuzione del codice. Infatti la traduzione standard in linguaggio macchina di una chiamata di funzione comporta il passaggio dei parametri nello stack, un salto all'indirizzo della funzione, il recupero dei valori dei parametri, la scrittura dei valori restituiti nello stack e il loro recupero da parte della procedura chiamante.

Se una funzione viene richiamata nel ciclo più interno di un algoritmo critico, come nel caso della funzione che calcola il valore della funzione sigmoidale, è conveniente dichiarare la funzione *inline*. In questo caso il compilatore invece di implementare la chiamata di funzione, inserisce il direttamente il codice della funzione chiamata all'interno della funzione chiamante.

Lo svantaggio di questo approccio è che lo stesso codice verrà ripetuto per tutte le volte che la funzione verrà richiamata, creando quindi file eseguibili e file oggetto più grandi, per questo motivo deve essere utilizzato solo per le funzioni chiave e solo per funzioni molto semplici.

L'utilizzo della tecnica di in-lining ha portato a diminuire i tempi di computazione circa di un fattore 6.

5.5 Implementazione della backpropagation con δ-rule

La procedura fondamentale per l'apprendimento della rete è l'implementazione della δ-rule. Per addestrare la rete è necessario fornire un numero sufficiente di esempi di coppie di vettori (input, output). Per addestrare la rete ad associare a un vettore *input* l'output *target* si deve per prima cosa richiamare il metodo **compute (const float *input, float *output)**, lo stesso che serve per ottenere il risultato una volta che la rete è addestrata. Il secondo parametro può essere `NULL` se durante l'addestramento non interessa il valore restituito dalla rete perché i valori dell'output vengono mantenuti nei neuroni dell'ultimo strato, come è spiegato nella sezione 5.3.

Subito dopo si richiama il metodo **compute_output_error (const float *target)** che esegue il primo passo della δ-rule calcolando i valori dei δ dell'ultimo livello che sono definiti come

$$\delta_i^n = \begin{cases} O_i^n(1-O_i^n)(T_i - O_i^n) & \text{se si usa la funzione logistica} \\ (1-(O_i^n)^2)(T_i - O_i^n) & \text{se si usa la tangente iperbolica} \end{cases} \tag{4}$$

dove indichiamo con O_i^n l'output dell'i-esimo neurone dello strato di output (n =*no_of_layers*-1) e con T_i l'i-esimo elemento del vettore di apprendimento.

Il passo successivo, comune a tutte le tecniche di apprendimento, consiste nel calcolo dei δ di tutti gli strati precedenti. Questa operazione viene eseguita a partire dall'ultimo strato (da cui il nome backpropagation) per tutti i livelli l da *no_ of_ layers -2* a 0.

$$\delta_i^l = O_i^l(1 - O_i^l) \sum_{j=0}^{n_{l+1}-1} W_{ij}^{l+1} \delta_j^{l+1} \text{ se si usa la funzione logistica}$$

$$\delta_i^l = (1 - (O_i^l)^2) \sum_{j=0}^{n_{l+1}-1} W_{ij}^{l+1} \delta_j^{l+1} \text{ se si usa la tangente iperbolica}$$

I valori dei δ vengono memorizzati all'interno del campo *error* di ogni neurone.

5.6 Apprendimento on line con momento

Se si vuole addestrare la rete con il metodo on line per ogni coppia di addestramento si può richiamare il metodo **train ()** che aggiorna i pesi della rete secondo la regola $(W_{ij}^l)' = W_{ij}^l + \Delta W_{ij}^l$ con

$$\Delta W_{ij}^l = \eta \delta_i^l O_j^l \tag{5}$$

dove η è il *learning rate* della rete.

Per implementare la backpropagation con momento è sufficiente memorizzare il ΔW_{ij}^l al passo precedente (nel vettore *delta* di ogni neurone) e definire

$$\Delta W_{ij}^l(p) = \eta \delta_i^l O_j^l + \alpha \Delta W_{ij}^l(p-1) \tag{6}$$

dove $\Delta W_{ij}^l(p-1)$ è il delta al passo precedente memorizzato nella struttura e $\Delta W_{ij}^l(p)$ è il nuovo delta. Se α (memorizzato in *momentum*) è 0 si ha la backpropagation senza momento.

Lo scopo della backpropagation con momento è di ridurre l'entità della correzione nel caso che il gradiente abbia verso opposto a quello che aveva al passo precedente (tipica situazione in cui si hanno delle oscillazioni intorno al minimo) e invece di amplificare la correzione nel caso in cui la correzione ha lo stesso verso nei due passi (accelerando così lo scorrimento delle zone piatte della superficie dell'errore).

Riportiamo la porzione di codice che esegue l'aggiornamento dei pesi: precedentemente devono essere calcolati i valori dei δ_i^l che sono contenuti nel campo `layer[l].neuron[i].error`.

Aggiornamento dei pesi per l'apprendimento on line

```
int l, nu, nl;
float learning_factor, delta;

for (l = 1; l < no_of_layers; l++)
{
  for (nu = 0; nu < layer[l].no_of_neurons; nu++)
    {
      learning_factor = learning_rate * layer[l].neuron[nu].error;
      for (nl = 0; nl < layer[l - 1].no_of_neurons; nl++)
        {
          delta =
            learning_factor *
            layer[l - 1].neuron[nl].output +
            momentum * layer[l].neuron[nu].delta[nl];
          layer[l].neuron[nu].weight[nl] += delta;
          layer[l].neuron[nu].delta[nl] = delta;
        }
    }
}
```

5.7 Apprendimento in batch

L'apprendimento per epoche ha lo scopo di compensare le oscillazioni nelle correzioni dei pesi, mediando su tutte le correzioni provocate dai diversi pattern.

Dato un insieme di N coppie input / output si calcolano i δ definiti nella sezione 5.5 e se ne fa la media. Si utilizza questa media per calcolare le correzioni dei pesi.

$$\Delta W_{ij}^{l} = \eta \frac{1}{N} \sum_{p=1}^{N} \delta_i^{l(p)} O_j^{l(p)} \tag{7}$$

Nella nostra implementazione all'inizio di un'epoca si deve richiamare il metodo **begin_batch ()**, che pone il campo *no_of_patterns* a 0 e riazzera in tutti i neuroni della rete i valori dei campi *sumdeltas*.

Per ogni pattern della batch si deve quindi richiamare il metodo **train_batch ()** (con le stesse modalità del metodo **train ()**, cioè dopo aver richiamato **compute ()** e **compute_output_error ()**), che incrementa di un'unità il campo *no_of_patterns* e somma ai campi *sumdeltas* il valore $\delta_i^{l(p)} O_j^{l(p)}$ attuale (i pesi non vengono modificati).

Alla fine dell'epoca si deve richiamare il metodo **end_batch ()**, che esegue l'aggiornamento dei pesi usando come delta la media calcolata dividendo *sumdeltas* per *no_of_patterns*.

È possibile combinare l'apprendimento in batch con il metodo del momento usando come ΔW^l_{ij} al passo precedente il valore della correzione fatta nell'epoca precedente, che è memorizzata nel vettore *delta*.

Alcuni autori definiscono il training in batch diversamente, senza dividere per N la somma delle correzioni ai pesi.

$$\Delta W^l_{ij} = \eta \sum_{p=1}^{N} \delta_i^{l(p)} O_j^{l(p)} \tag{8}$$

L'espressione 8 è comunque equivalente alla 7 se si sceglie in (7) un learning rate $\eta' = N\eta$.

In effetti in alcuni esperimenti è risultato opportuno settare il learning rate a un valore significativamente più alto per addestrare la rete in batch. Tuttavia abbiamo lasciato all'utente la responsabilità di settare in modo opportuno il coefficiente di apprendimento. Questa ambiguità viene comunque superata se si usa l'addestramento in batch in combinazione con il metodo Super SAB, che adatta autonomamente i coefficienti d'apprendimento.

5.8 Apprendimento Super SAB (Self Adapting Backpropagation)

Il metodo Super SAB (Self Adapting Backpropagation) fa muovere ogni peso verso il minimo con una velocità differente. Ogni peso ha un suo coefficiente di apprendimento η_{ij}. Tutti questi coefficienti sono inizializzati al valore di default e vengono modificati durante l'apprendimento. Si definiscono due fattori $\eta_{up} > 1$ e un fattore $\eta_{down} < 1$ e ad ogni passo si modificano i learning rates in questo modo.

- Se la correzione nel peso mantiene lo stesso segno del passo precedente il coefficiente di apprendimento viene moltiplicato per η_{up}.
- Se la correzione nel peso cambia segno rispetto al passo precedente, il coefficiente di apprendimento viene moltiplicato per η_{down}.

La modalità Super SAB viene attivata chiamando il metodo **begin_ssab ()**, che alloca un vettore di learning rate di lunghezza uguale al numero di pesi della rete inizializzati al valore del learning rate corrente.

L'apprendimento in Super SAB può avvenire richiamando il metodo **train_ssab ()** che modifica i pesi usando il vettore dei learning rate e modificando i learning rate secondo la regola enunciata sopra.

È però più efficiente combinare il metodo Supoer SAB con l'apprendimento in batch. In questo caso dopo aver richiamato una sola volta il metodo **begin_ssab ()** si deve chiamare all'inizio di ogni epoca il metodo **begin_batch ()**, addestrare la rete con il metodo **train_batch ()** e richiamare alla fine dell'epoca il **end_batch_ssab ()**. Questo metodo modifica i pesi secondo la media delle variazioni calcolate sui vari pattern, usando i learning rate contenuti nel vettore e modificandoli secondo la regola del Super SAB.

La libreria permette di settare a piacere i parametri η_{up} e η_{down} e di definire un limite massimo e un limite minimo per i valori dei learning rate.

I valori dei learning rate possono essere salvati su un file binario ed essere ricaricati in seguito per proseguire l'addestramento.

5.9 Salvataggio della rete su file

La rete può essere salvata su un file per essere poi ricaricata e usata successivamente. Nel file binario si salva la funzione di attivazione (codificata come 0 o 1), il numero di livelli, il numero di neuroni per ogni livello, il momento, il learning rate, l'errore globale e tutti i pesi. Chiaramente non si salvano i valori degli array *delta* e *sumdeltas* che servono solo per l'apprendimento, mentre i parametri per l'apprendimento Super SAB possono essere salvati in un file separato.

È possibile caricare o salvare una rete anche in formato testuale. Questo formato non è generalmente conveniente perché utilizza più spazio su disco, tuttavia può essere utile per importare reti create con la libreria *Lightweight Neural Network* o per esportare delle reti su calcolatori con architetture diverse che utilizzano una diversa codifica binaria per i numeri a virgola mobile.

6 La classe trainer

6.1 Struttura

La classe **trainer** permette all'utente di addestrare una rete a partire da pattern memorizzati su file, con i vari metodi messi a disposizione dalla nostra libreria. Si distinguono tre insiemi di pattern: pattern di addestramento, pattern di validazione e pattern di certificazione (training set, validation set e certification set). Il training set serve per l'apprendimento, il validation set serve per verificare, durante l'addestramento, che la rete generalizzi in modo corretto, mentre i pattern di certificazione sono utilizzati al termine dell'addestramento per un'ulteriore verifica della correttezza dei risultati forniti dalla rete addestrata. Ognuno dei tre insiemi di pattern può essere caricato da un file.

Per garantire la sicurezza dell'addestramento ed evitare di addestrare una rete su pattern che non rispettano la dimensione degli strati di input e di output, la classe **trainer** mantiene il numero di input e il numero di output che vengono decisi quando l'oggetto viene costruito e non possono essere modificati in seguito. La rete usata dal trainer ed ogni insieme di pattern devono avere il numero di input e di output stabilito.

La lettura dal file avviene per mezzo di un oggetto di una classe derivata da **iomanage**, per permettere la massima flessibilità nel formato degli input.

Inoltre la classe **trainer** può essere specializzata per personalizzare le sue caratteristiche, come spiegato nella sezione 6.9. Grazie a queste possibilità di adattarsi a varie esigenze, questa classe rappresenta un piccolo *framework* per chi vuole scrivere un programma di addestramento per una rete neurale.

6.2 Costruttori della classe

La costruzione di un oggetto della classe **trainer** può avvenire in tre modi differenti.

Tutti e tre i tipi di costruttori hanno in comune gli ultimi due parametri che devono specificare i nomi dei file di log dove salvare l'andamento dell'errore e dell'accuracy (definita nella sezione 6.7). Se questi nomi non vengono specificati, non verrà scritto nessun file di log.

Il primo tipo di costruttore permette di costruire un trainer a partire da una rete, cioè da un oggetto di classe **network** che decide il numero di input e di output del trainer.

Il secondo tipo di costruttore prevede di specificare il numero di input, il numero di output, il numero di neuroni nello strato nascosto, e costruisce una rete con 3 strati e con le dimensioni specificate nel costruttore.

Il terzo tipo di costruttore richiede il nome di un file contenente i pattern di training da leggere, il numero di neuroni nello strato nascosto e costruisce una nuova rete a 3 strati con le dimensioni corrispondenti a quelle memorizzate sul file. È possibile anche specificare un oggetto di classe **iomanage** se si vuole utilizzare un formato diverso da quello standard per il file su cui sono memorizzati i pattern.

6.3 Specifica dei pattern di apprendimento, verifica e certificazione

La classe **trainer** mette a disposizione tre metodi per specificare il nome dei file in cui trovare i pattern di apprendimento, verifica e certificazione. Si tratta rispettivamente dei metodi **load_training**, **load_validation**, **load_certification**. I pattern caricati da file vengo memorizzati in tre diversi array di puntatori ad array di float e presentati alla rete da addestrare durante la fase di apprendimento della rete. Il formato di questi file dipende dall'oggetto di classe **iomanage** settato nel trainer, che può essere cambiato con il metodo **set_iomanager**.

6.4 Formato dei file contenenti i pattern

6.4.1 Formati testuali

La classe **iomanagelwnnfann** è quella usata di default dal trainer e supporta due formati di file: il formato *lwnn* ed il formato *fann*. Viene riconosciuto automaticamente il formato del file.

Formato "lwnn"

1. Prima riga: numero di pattern contenuto nel file
2. Seconda riga: lunghezza dell'input
3. Terza riga: lunghezza dell'output
4. Righe successive: Le righe successive devono contenere i pattern veri e propri. Deve essere specificato l'input in una riga e il target atteso nella riga successiva.
5. Commenti: le righe che iniziano con il carattere # sono considerati commenti e quindi ignorate. Non sono permessi commenti tra una riga che specifica un input e una che specifica un target.

Formato "fann"

Questo è il formato utilizzato dalla libreria *fann*. I file che specificano gli insiemi di pattern per questa libreria possono quindi essere facilmente importati.

1. Prima riga: Contiene tre interi che specificano il numero di patterns, il numero di input e il numero di output
2. Le successive righe contengono tutti i vettori di input
3. Le ultime righe contengono i corrispondenti vettori di output
4. Non sono ammessi commenti.

6.4.2 Formato binario

La classe **iomanagebinary** è un'altra classe derivata da **iomanage** che gestisce i file contenenti i pattern in formato binario.

In questo caso il file deve contenere, in formato binario, il numero di pattern, il numero di input e il numero di output. Seguono poi i vettori di input e di target, anch'essi in formato binario.

La classe fornisce anche un metodo **write_patterns** ed un metodo **convert** che permettono di convertire facilmente dei pattern memorizzati in un qualsiasi formato (ovvero con una qualunque implementazione di **iomanage**) in formato binario. In questo modo il formato binario può essere usato come formato standard e, se i dati sono memorizzati in un qualunque altro formato, si possono convertire nel formato binario e utilizzarli in un programma che utilizzi la classe **iomanagebinary**.

6.5 Addestramento di una rete

La classe **trainer** mette a disposizione le varie tecniche di addestramento implementate nella classe **network**. I metodi per addestrare una rete sono: **train_online**, per l'addestramento on-line, **train_batch** per l'addestramento in batch, **train_ssab** per l'addestramento con la tecnica Super SAB, **train_shuffle** per una variante dell'addestramento on line che prevede di presentare i pattern dell'insieme di addestramento in un ordine casuale diverso per ogni epoca. Ad ognuna di queste tecniche sarà applicato un momento, se previsto nella rete da addestrare. Questi metodi prendono come parametro il valore booleano `verbose` che permette di stampare sullo schermo l'andamento dell'errore.

Riportiamo di seguito il codice del metodo **train_online**

Codice del metodo train_online(bool verbose)

```
int trainer::train_online(bool verbose) {
  float error;
  int start = epoch;
  while (continue_training(start)) {
    error = 0.0;
    for (int i = 0; i < npattern_training; i++) {
      /* compute the outputs for inputs[i] */
      net->compute (training_set_input[i], NULL);
      /* find the error with respect to targets[i] */
      error += net->compute_output_error ( training_set_target[i] );
      /* train the network one step */
      net->train ();
    }
    error_on_training = error / npattern_training;
    epoch++;
    reports(verbose);
  }
  return epoch - start;
}
```

Come si può vedere, per ogni ciclo del `while` vengono sottoposti alla rete tutti i pattern del set di addestramento, e si calcola l'errore quadratico medio sui pattern di training. La condizione di arresto del metodo è implementata dal metodo **continue_training** e può dipendere da vari parametri secondo il tipo di addestramento che si vuole eseguire. In effetti la condizione di arresto dell'addestramento è il punto più delicato del processo di apprendimento. Infatti lo scopo dell'apprendimento è quello di creare reti che non solo riproducano bene i pattern di apprendimento ma siano anche capaci di generalizzare.

Una condizione d'arresto sempre valida è il limite al numero di epoche. Il metodo **set_max_epochs** permette di specificare un limite superiore al numero di epoche nell'addestramento. Questa condizione è importante perché garantisce in ogni caso la terminazione dell'apprendimento, dato che non è facile prevedere a priori quale sarà il minimo errore raggiungibile.

Se non si vuole usare un insieme di validazione si può specificare come ulteriore condizione di arresto il raggiungimento di un certo errore sull'insieme di training. Questo modo di procedere rischia però di generare il problema dell'overfitting, descritto nella prossima sezione.

Un possibile modo di usare l'insieme di validazione è quello di terminare l'apprendimento quando si raggiunge un certo errore sull'insieme di validazione. Per fare in questo modo, si può richiamare sul trainer il metodo **set_training_validation** che fa sì che l'errore desiderato non debba essere raggiunto sull'insieme di addestramento ma su quello di validazione. Per ragioni di efficienza l'errore sull'insieme di validazione non viene controllato ad ogni passo, ma ogni $n_{checking}$ epoche. Questo parametro viene specificato con il metodo **set_epochs_checking_error**.

Questo metodo di addestramento è utile se vogliamo che la rete garantisca una certa precisione sui pattern dell'insieme di validazione (e quindi, presumibilmente, sulla generalità dei pattern, se l'insieme di validazione è abbastanza rappresentativo). Tuttavia la scelta di questo parametro non è banale. Infatti scegliendo un valore troppo piccolo si rischia di non raggiungere mai questo errore sull'insieme di validazione, d'altra parte un errore più grande e facilmente raggiungibile non dà la garanzia di essere il valore ottimale per quella rete. In pratica si rischia di addestrare la rete troppo o troppo poco.

La classe trainer fornisce un'altra condizione di arresto alternativa: una volta definita l'accuracy (come descritto nella sezione 6.7) si può arrestare l'addestramento quando si raggiunge una certa accuracy sul *validation set*. Questo parametro viene specificato dal metodo **set_wanted_accuracy**. Per questa tecnica valgono tutte le considerazioni esposte sopra.

Al termine dell'addestramento è possibile richiamare il metodo **get_-stopping_cause()** che restituisce un codice che identifica univocamente il motivo per cui l'ultima sessione di addestramento è terminata.

6.6 Gestione dell'over-fitting

La strategia di continuare l'addestramento finché l'errore quadratico medio sul set di addestramento scende sotto una certa soglia è molto rischiosa, perché la backpropagation è suscettibile a riprodurre in maniera sempre migliore i pattern di addestramento, al costo di diminuire la capacità di generalizzazione a pattern non usati nell'addestramento.

Si osserva infatti che l'errore sull'insieme di validazione prima diminuisce, e poi aumenta, anche se l'errore sull'insieme di addestramento continua a diminuire. Come può succedere questo? La spiegazione è che i pesi della rete vengono sintonizzati per riprodurre le peculiarità degli esempi usati nell'addestramento, che non sono rappresentative della generalità dei pattern. L'overfitting tende a verificarsi negli stadi avanzati dell'addestramento e non

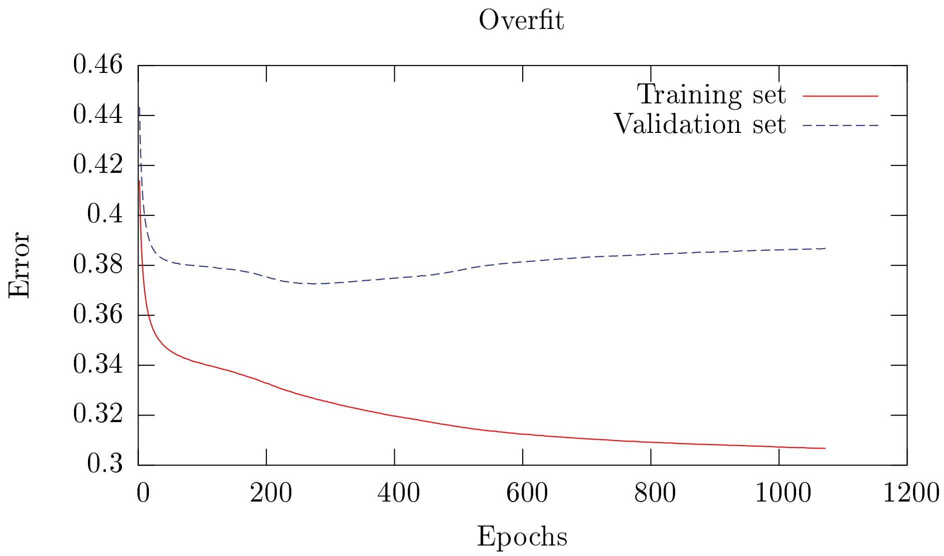

Figura 2: *Esempio classico di overfit. In questo caso la rete migliore è stata ottenuta all'epoca 273.*

nelle prime epoche, perché - dato che i pesi della rete sono inizializzati con valori piccoli e casuali - nelle prime iterazioni la superficie di decisione (cioè della funzione interpolante) è relativamente semplice, perché la funzione di trasferimento è pressoché lineare. Via via che l'addestramento procede alcuni pesi iniziano a aumentare per ridurre l'errore sui pattern di addestramento e la forma della superficie di decisione diventa più complessa. In un periodo di addestramento abbastanza lungo la rete sarà spesso in grado di generare interpolazioni sempre più precise nei punti su cui è addestrata ma che daranno risultati sempre peggiori per punti che non fanno parte del training set.

Uno dei metodi di maggior successo per evitare l'overfitting, descritto nel libro di Mitchell [1], prevede di usare un insieme di validazione insieme all'insieme di addestramento. L'algoritmo controlla l'errore sull'insieme di validazione e usa l'insieme di apprendimento per calcolare la discesa del gradiente. Il numero ideale di iterazioni sarà quindi quello che produce l'errore minimo *sull'insieme di validazione*, dato che è quest'ultimo il miglior indicatore del comportamento della rete su esempi mai visti.

Nella nostra implementazione si mantengono due copie della rete. Una è usata per l'addestramento, mentre si salva la rete che fino a quel momento ha prodotto il minimo errore sull'insieme di validazione. Quando per un certo numero di epoche l'errore sull'insieme di validazione non diminuisce, e diventa significativamente più grande dell'errore sull'insieme di addestramento, la procedura di addestramento termina e restituisce la rete che ha prodotto il minimo errore sull'insieme di validazione e che era stata salvata.

In figura 2 si vede come l'errore sull'insieme di validazione a un certo punto tende a aumentare. La rete trovata dall'algoritmo per evitare l'overfit è quella dell'epoca 273, corrispondente al minimo errore sull'insieme di validazione.

La classe **trainer** permette di specificare due parametri per l'addestramento con controllo dell'overfitting: l'apprendimento termina se è passato un certo numero di epoche (primo parametro) senza trovare una rete che produce un errore minore sull'insieme di validazione e se il rapporto tra l'errore sull'insieme di validazione e l'errore sull'insieme di addestramento è maggiore di un certo valore (secondo parametro). Questi parametri devono essere utilizzati correttamente per evitare di fermarsi non appena l'errore sull'insieme di validazione cresce; può succedere infatti che l'errore sull'insieme di validazione cresca per poi diminuire nuovamente, come in figura 3. In questo caso bisogna stare attenti a non fermare l'algoritmo non appena l'errore sul validation set comincia ad aumentare: facendo così si concluderebbe che la rete migliore è quella trovata intorno all'epoca 50, e non quella all'epoca 407. Bisogna inoltre tenere conto del fatto che, anche se si setta la modalità di arresto per overfit, il parametro *min_error* rimane significativo e l'addestramento si arresterà se il miglior errore sull'insieme di validazione scende sotto il valore dell'errore minimo. Quindi se si vuole eliminare questa condizione d'arresto bisognerà specificare un errore minimo molto basso (o addirittura nullo) con il metodo **set_min_error**. Al termine dell'addestramento, si deve richiamare il metodo **set_network_best()** per recuperare la rete con il minimo errore sull'insieme di validazione.

La figura 4 mostra un caso limite in cui all'inizio la rete generalizza talmente bene che l'errore sull'insieme di validazione è minore dell'errore sull'insieme di addestramento. Con il progredire dell'addestramento la situazione si inverte.

6.7 Definizione di accuracy

In molti casi le reti neurali vengono utilizzate per classificare dei pattern. In questi casi diventa importante, piuttosto dell'errore quadratico medio, la capacità della rete di classificare correttamente. Cosa voglia dire classificare correttamente dipende naturalmente dal tipo di problema, e dal significato attribuito al vettore di output. Tuttavia esistono due casi molto comuni, per i quali abbiamo implementato il calcolo automatico della precisione nella classe **trainer**.

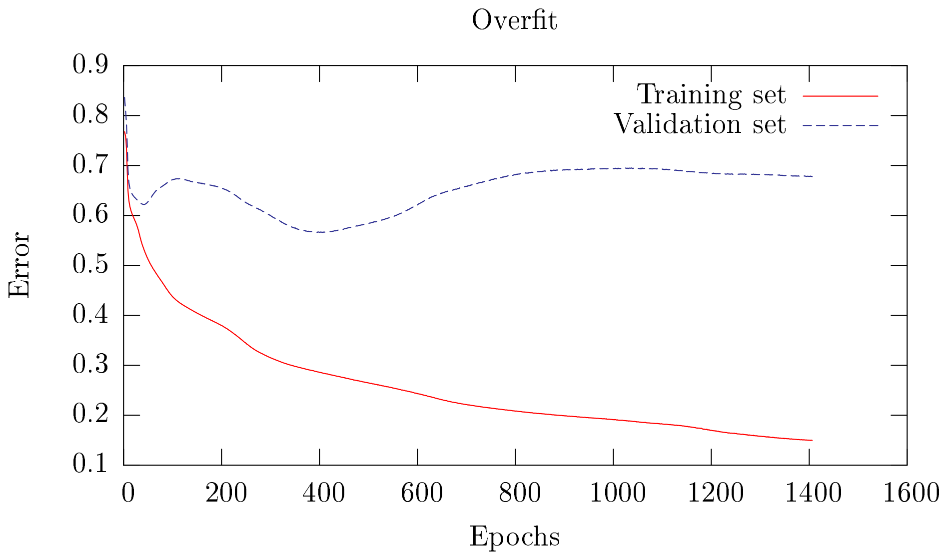

Figura 3: *In questo esempio l'errore sull'insieme di validazione aumenta, poi diminuisce e poi aumenta nuovamente. La rete migliore si ottiene all'epoca 407 (e non la prima volta che l'errore sul validation set inizia ad aumentare!)*

Binary Accuracy Si considera una rete che deve produrre come output un numero binario di lunghezza n. Dato che l'output non è binario si dovrà procedere a un arrotondamento, e si considereranno uguali a 0 tutti i neuroni che producono un output minore di 0.5 ed uguali a 1 tutti i neuroni che producono un output maggiore di 0.5. Dato quindi un insieme di N pattern e i corrispondenti target, per calcolare l'accuracy di una rete calcoliamo il suo output per ognuno di questi N input e contiamo il numero di risposte giuste, cioè di output per i quali, detto T_i l'i-esimo elemento del vettore target e O_i l'i-esimo elemento dell'output della rete vale

$$(T_i = 0 \land O_i < 0.5) \lor (T_i = 1 \land O_i > 0.5) \quad \forall i = 0 \dots n-1$$

L'accuracy sarà data dal numero delle risposte giuste diviso per il numero totale di pattern N.

In realtà nella nostra implementazione utilizziamo una definizione più larga dell'accuracy che prevede che anche l'output possa non essere precisamente binario (ad esempio, si potrebbero usare i valori 0.1 e 0.9 invece di 0 e 1 nella codifica binaria) e cioè

$$(T_i < 0.5 \land O_i < 0.5) \lor (T_i > 0.5 \land O_i > 0.5) \quad \forall i = 0 \dots n-1$$

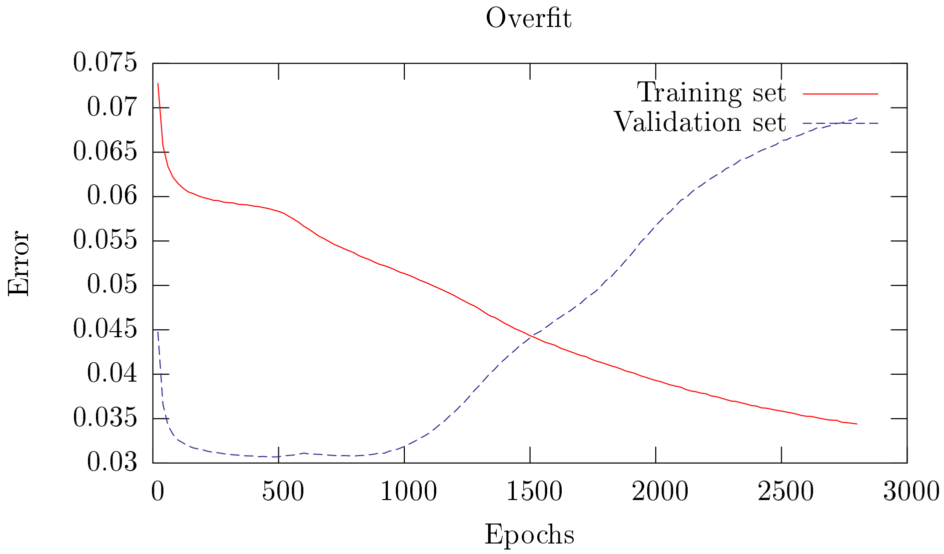

Figura 4: *In questo caso l'errore sull'insieme di validazione è inizialmente minore dell'errore sull'insieme di addestramento. La prova è stata fatta su una rete a 3 strati, con 9 neuroni di input, 9 neuroni nello strato nascosto e 2 neuroni nello strato di output. La rete migliore è stata trovata all'epoca 480*

Max Guess Accuracy Una definizione alternativa della precisione si ha quando le classi nelle quali si vogliono classificare gli input non sono codificate come numeri binari ma con un 1 nella posizione corrispondente (ad esempio quattro classi codificate come $1000, 0100, 0010, 0001$). Questo modo di codificare gli output è molto utilizzato nel campo delle reti neurali perché richiede generalmente una precisione minore per ottenere risposte corrette.

In questo caso per classificare correttamente un pattern, infatti, è sufficiente che la posizione del neurone di valore massimo corrisponda alla posizione dell'1 nel target. Da questo segue un'analoga definizione di *accuracy* che chiameremo *Max Guess* in cui una risposta è considerata giusta se

$$j_{max} = k_{max} \text{ dove } j_{max} \,|\, O_{j_{max}} > O_i \,\forall i \neq j_{max} \text{ e } k_{max} \,|\, T_{k_{max}} > T_i \,\forall i \neq k_{max}$$

La classe **trainer** permette di calcolare l'accuracy secondo queste due definizioni. Il metodo **set_accuracy_mode** permette di specificare il modo di calcolare l'accuracy: si può scegliere se non calcolarla (perché non è significativa) o se calolarla secondo la prima o la seconda definizione.

È inoltre possibile scrivere una classe derivata da **trainer**, come descritto nella sezione 6.9, che ridefinisce il metodo che si occupa di decidere se una risposta è da considerare giusta, personalizzando così la definizione di accuracy.

6.8 File di log

Nella costruzione di un oggetto di classe **trainer** si possono specificare i nomi di due file di log. Il primo conterrà l'andamento dell'errore, il secondo l'andamento dell'accuracy. Questi file vengono scritti ad ogni epoca se non si utilizza un insieme di validazione, solo quando viene fatta la validazione se invece si usa il validation set. In quest'ultimo caso i file di log conterranno in ogni riga tre numeri divisi dal carattere di tabulazione che rappresentano rispettivamente l'epoca, l'errore (o l'accuracy) sul training set e l'errore (o l'accuracy) sul validation set. Il file di log dell'accuracy viene scritto solo se l'*accuracy mode* non è settata a 0 (valore che significa "non calcolare l'accuracy").

6.9 Specializzazione della classe trainer

È possibile modificare ulteriormente il comportamento della classe **trainer**, scrivendo una classe derivata da **trainer** e ridefinendo alcuni metodi.

In particolare tre metodi sono stati pensati per essere specializzati. Questi metodi sono dichiarati **protected** (cioé non possono essere richiamati su un oggetto classe **trainer** ma possono essere chiamati e riscritti da una classe derivata) e **virtual** per garantire il binding dinamico nell'uso del polimorfismo.

Il primo metodo significativo che può essere riscritto facilmente è **check**. Questo metodo prende come parametri un output della rete e il corrispondente target e deve restituire un valore booleano che indica se la risposta deve essere considerata corretta. In questo modo si può dare una definizione diversa di accuracy, che tenga conto del significato che in una particolare rete ha il vettore di output. Per esempio per una rete che usa la tangente iperbolica si potrebbero usare i valori -1 e 1 invece di 0 e 1 per codificare output binari, oppure si potrebbe definire giusta una risposta se la norma della differenza tra l'output e il target è minore di una certa soglia.

Per garantire la massima flessibilità, abbiamo lasciato la possibilità di riscrivere anche il metodo **continue_training**. Questo metodo è quello che decide la condizione d'arresto per l'addestramento, ed è il punto più critico nell'addestramento di una rete. La classe **trainer** permette già di specificare molti parametri che determinano la condizione d'arresto, dal numero massimo di epoche, al minimo errore che si vuole ottenere, ed il metodo per evitare l'overfitting che dovrebbe essere adatto alla maggior parte delle applicazioni. Tuttavia è possibilie scrivere una classe derivata di **trainer** e ridefinire questo metodo per ottenere una condizione d'arresto diversa. Naturalmente in questo caso molti parametri della classe **trainer** potrebbero perdere di significato o assumere un significato diverso. Sarà compito del programmatore che scrive la classe che estende **trainer** far sì che l'interfaccia di questa nuova classe sia comunque coerente.

Altri due metodi che possono essere specializzati sono **print_input** e **print_output** che possono definire un modo diverso per stampare i vettori di input e di output più adatto al particolare significato che l'input e l'output assumono nella rete che si vuole addestrare.

7 Il pacchetto *lwneuralnetplus* e l'interfaccia grafica *klwnn++*

7.1 Installazione e utilizzo del pacchetto

Il pacchetto contenente la libreria, alcuni programmi utili per addestrare una rete, una semplice interfaccia grafica ed alcuni esempi può essere scaricato dal sito **http://lwneuralnetplus.sourceforge.net**.

La libreria e i programmi sono distribuiti sotto forma di codice sorgente e sono stati sviluppati e testati su un sistema *GNU/Linux* con il compilatore *gcc*, versione $\geq 2.95.4$, ma in linea di principio non dovrebbe essere difficile portarli su altri sistemi operativi o usare altri compilatori.

Per installare il programma, è sufficiente estrarre l'archivio, compilare la libreria con il comando *make* ed installarla con il comando *make install*. Informazioni più dettagliate sono contenute all'interno del pacchetto stesso.

Una volta installato il pacchetto, si avrà a disposizione la libreria *liblwnn-plus.so* che potrà essere utilizzata all'interno di qualsiasi programma `C++` e alcuni programmi utilizzabili da linea di comando:

lwnn_train Permette di addestrare una rete neurale, utilizzando la classe **trainer**. Le varie opzioni permettono di settare le caratteristiche della rete e le varie parametri della classe **trainer**.

lwnn_validate Prende come parametri un file che specifica i pattern e una rete salvata su file, usa la rete per calcolare gli output a partire dagli input, e calcola l'errore e l'accuracy della rete.

lwnn_printnet Stampa le caratteristiche principali di una rete memorizzata su un file.

lwnn_infospec Stampa le caratteristiche delle coppie di input/output memorizzate su un file (numero di coppie, numero di input, numero di output).

Inoltre è disponibile il comando **lwnn_visualtrainer** che lancia l'interfaccia grafica "Visual Network Trainer" descritta nella prossima sezione.

7.2 Interfacce

Per facilitare l'uso della nostra libreria, abbiamo sviluppato due interfacce grafiche che ne dimostrano le potenzialità.

La prima, distribuita insieme alla libreria, è una interfaccia scritta in tcl/tk che risulta essere più leggera e versatile, la seconda scritta sfruttando le librerie QT per KDE, distribuita separatamente e più facile e intuitiva da usare. Di seguito descriviamo brevemente entrambe.

7.2.1 Visual Network Trainer

Questa interfaccia grafica, fornita insieme alla libreria, è scritta nel linguaggio di scripting *tcl/tk*.

Visual Network trainer è una semplice maschera che permette di specificare i parametri dei programmi da linea di comando descritti precedentemente. Inoltre è possibile disegnare, a partire dai file di log prodotti dalla classe **trainer** (descritti nella sezione 6.8), i grafici dell'andamento dell'errore e dell'*accuracy* con il programma **gnuplot**.

L'interfaccia permette di specificare un insieme di addestramento (*training set*) e eventualmente un insieme di validazione (*validation set*), permette di settare i parametri della rete (numero di neuroni negli strati nascosti, learning rate, momento), di specificare le varie condizioni di arresto dell'apprendimento (massimo numero di epoche, errore da raggiungere, accuracy da raggiungere, arresto per overfit), e di scegliere il tipo di accuracy desiderata. È possibile creare una nuova rete oppure continuare l'addestramento a partire da una rete già addestrata.

L'interfaccia di questo programma è mostrata in figura 5.

7.2.2 KLWNN++

KLWNN++ è un'interfaccia alla nostra libreria scritta sfruttando le librerie QT, che si presenta come in figura 6. L'interfaccia è composta da quattro tab. Il primo, mostrato in figura 6 permette di settare i parametri di addestramento della rete di seguito elencati

- I file contenenti i pattern di addestramento, validazione e certificazione.
- Il tipo di addestramento (On line, batch training o Super SAB)
- Il numero massimo di epoche da eseguire
- L'errore richiestro
- Le condizioni di stop on over-fitting.

Il secondo tab permette di settare i parametri della rete neurale da addestrare, ovvero

- Il numero di neuroni nei vari strati nascosti, per un massimo di tre strati nascosti.
- Il momento
- Il learning rate
- Il tipo di funzione sigmoidale (Logistica o tangente iperbolica)
- Il tipo di riconoscimento (Binary round o Max Guess)

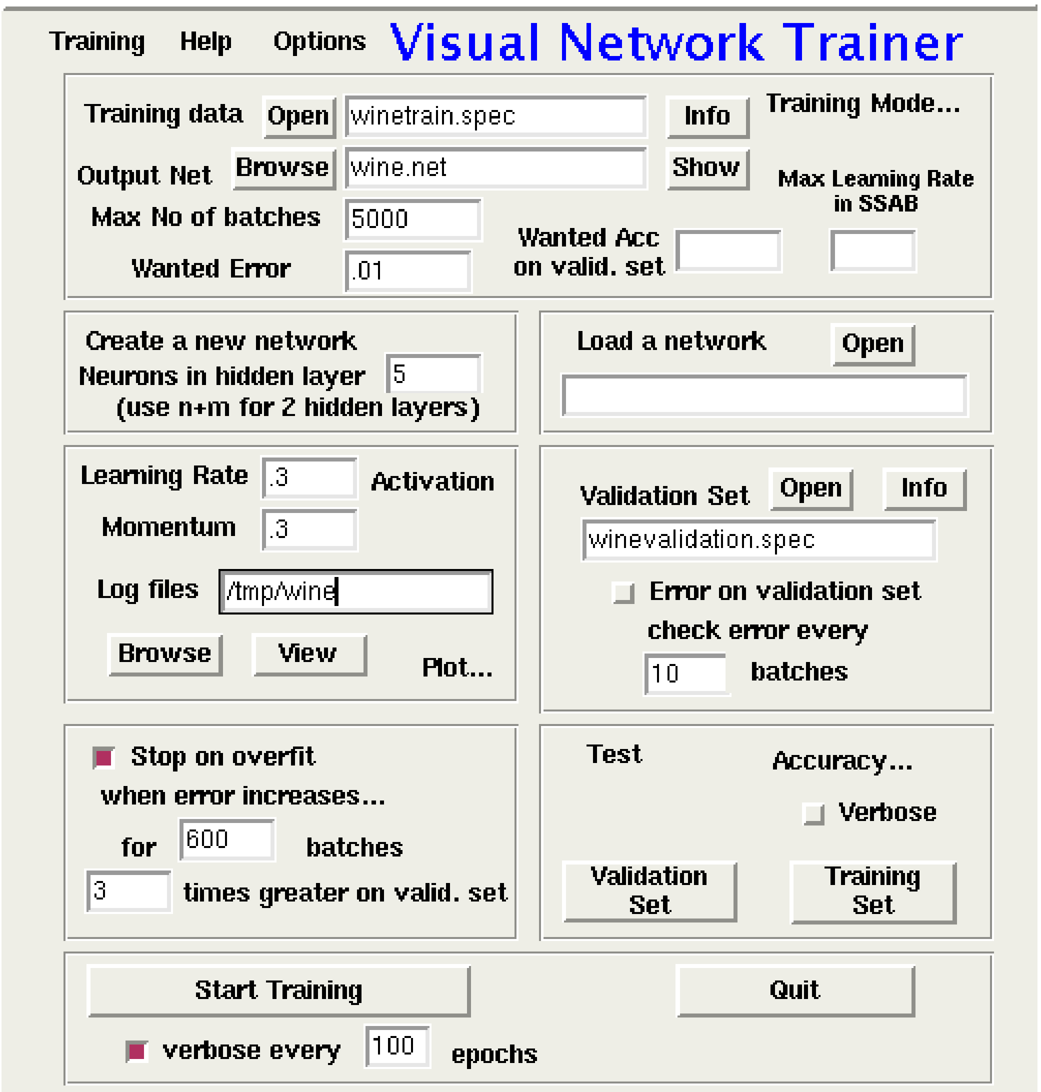

Figura 5: *L'interfaccia del programma* Visual Network Trainer

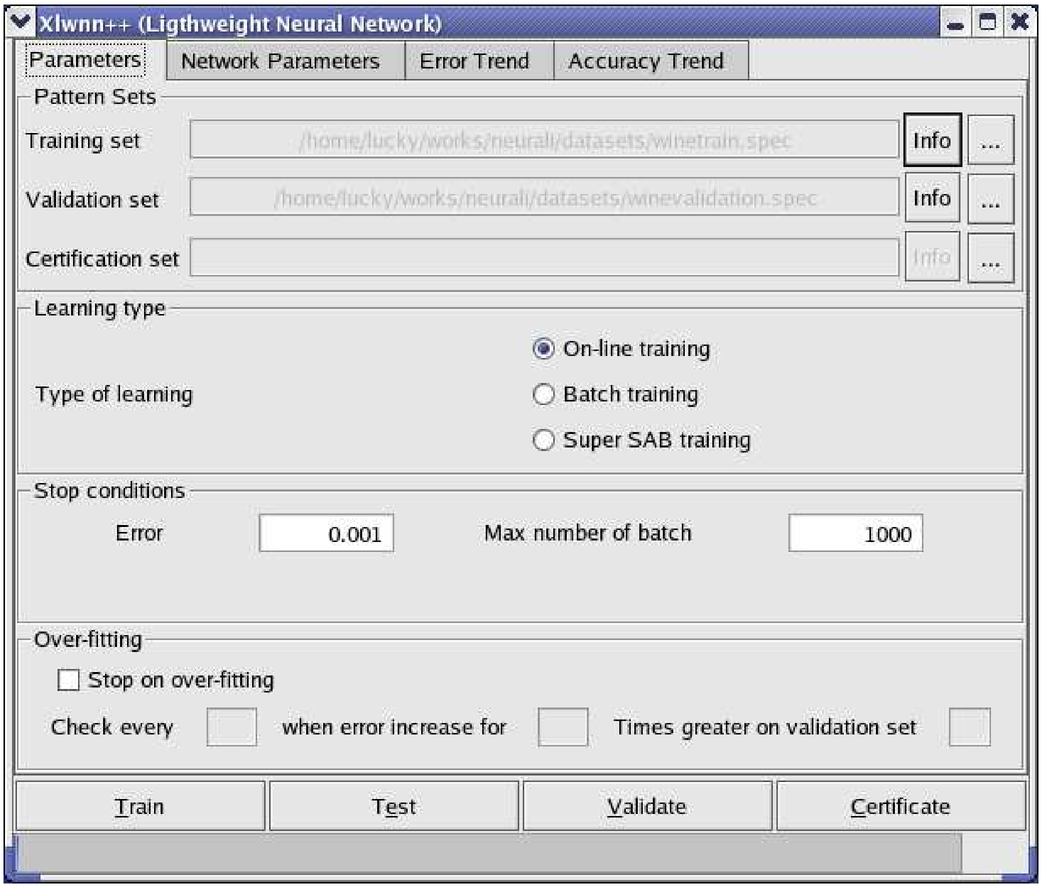

Figura 6: *L'interfaccia KLWNN, parametri di addestramento*

Inoltre in questo tab l'utente ha la possibilità di salvare la rete finora addestrata, caricare una rete da file o riniziare l'addestramento con una nuova rete (rispettivamente con i bottoni "Save network", "Load network" e "Clear network"). Questo tab è mostrato in figura 7

L'utente può iniziare l'addestramento agendo sul bottone "Train" e potrà controllare l'andamento dell'errore e dell'accuracy sull'insieme di addestramento e validazione spostandosi sul terzo e quarto tab dell'interfaccia. Il progredire dell'addestramento si può seguire anche grazie alla barra di progresso in fondo alla finestra. In figura 8 si può vedere l'andamento dell'errore durante l'addestramento di una rete. Al termine dell'addestramento l'utente verrà avvertito della causa della terminazione dell'addestramento (raggiungimento del numero massimo di epoche, raggiungimento dell'errore richiesto, over-fitting ...).

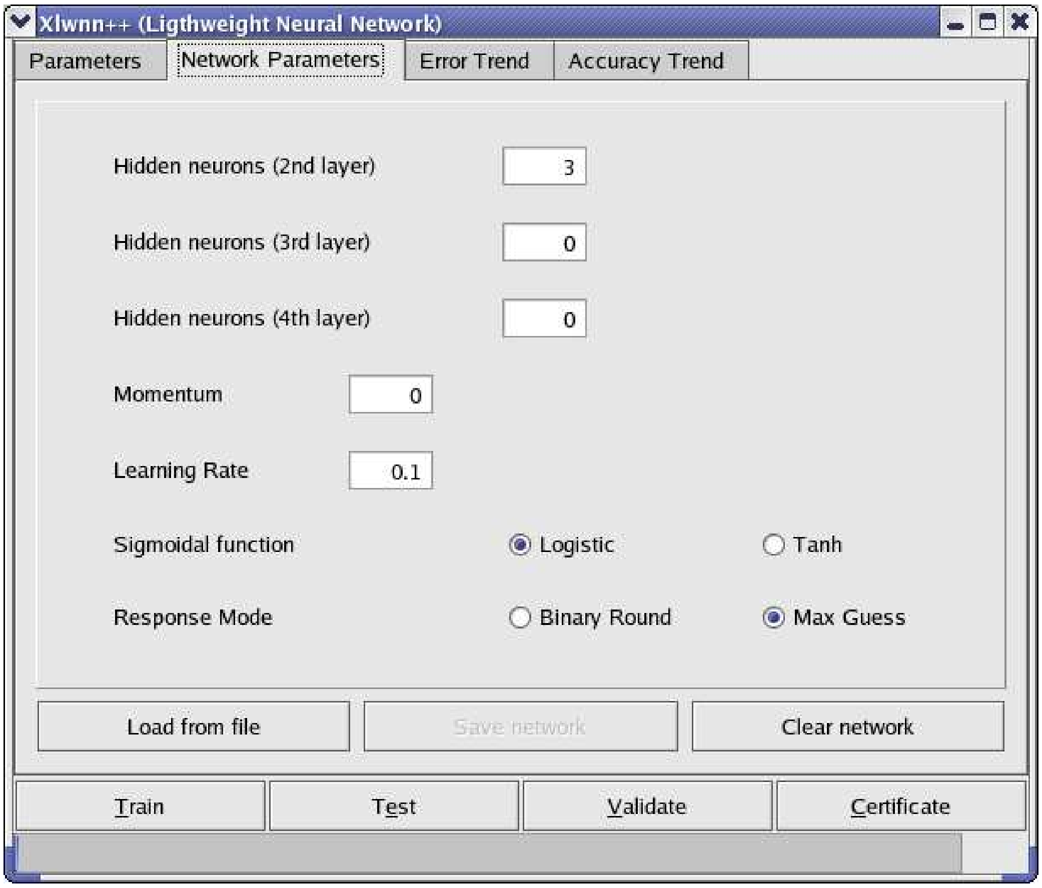

Figura 7: *L'interfaccia KLWNN, parametri di rete*

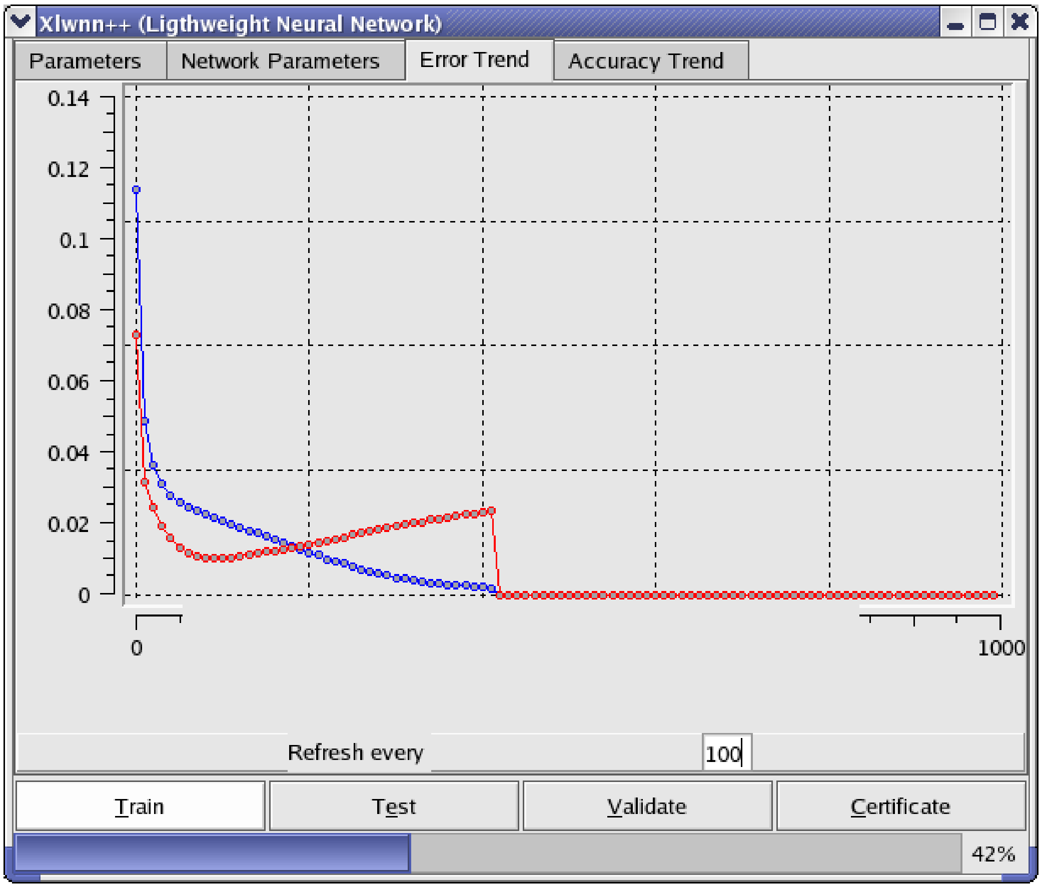

Figura 8: *L'interfaccia KLWNN, andamento dell'errore di addestramento*

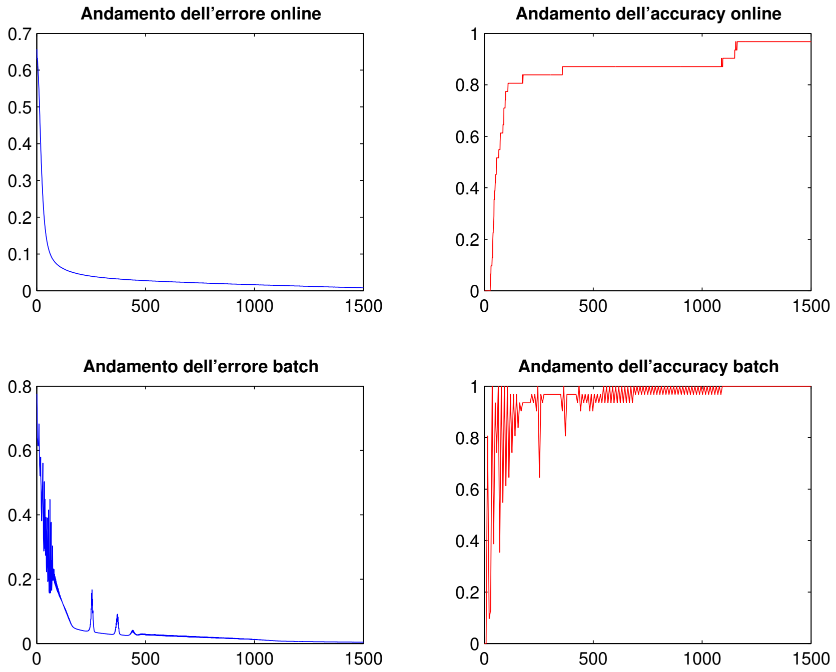

Figura 9: *Andamento dell'apprendimento on-line training e batch training*

8 Riconoscimento di vini

Utilizzando la nostra libreria, abbiamo sviluppato una semplice rete multi-layer perceptron che riconosce un vino dalle sue caratteristiche risultanti da una analisi chimica. Le caratteristiche del vino considerate sono: la gradazione alcolica, la quantità di acido malico, la cenere, l'alcalinità della cenere, la quantità di magnesio, l'acido fenico, flavonoidi, fenoli non flavonoidi, proantocianina, intensità di colore, tonalità, OD280/OD315 dei vini diluiti, prolina.

I vini possono essere distinti in tre classi di appartenenza. La rete realizzata ha quindi 13 neuroni di input, tre neuroni di output. Nell'addestramento le tre classi sono codificate come $(1,0,0),(0,1,0)$ e $(0,0,1)$. Per dare la risposta, si sceglie la classe con il valore maggiore. Per far funzionare la rete è stato sufficiente aggiungere uno strato nascosto formato da soli tre neuroni. Abbiamo ottenuto il set di pattern di apprendimento dall'UCI Repository of machine learning databases ([2]). Il set di apprendimento conteneva 178 pattern, 147 dei quali sono stati utilizzati per l'apprendimento vero e proprio, mentre i rimanenti 31 sono stati utilizzati come set di validazione. In figura 9 si può osservare l'andamento dell'errore e della accuracy (ovvero la frazione di pattern nel set di validazione riconosciuto dalla rete) ad ogni epoca. Con l'addestramento in batch si ottiene il riconoscimento di tutti i pattern nel set di validazione dopo meno di mille epoche. Con entrambi gli apprendimenti si evita comunque il problema dell'overfitting, ovvero la rete è in grado di generalizzare, riconscendo anche pattern di validazione che non facevano parte del set di training vero e proprio.

9 Il filtro antispam

I filtri antispam che si trovano su Internet o forniti insieme ad alcuni client di posta elettronica (ad esempio Mozilla Mail) tentano di riconoscere se una mail è da considerarsi spam principalmente in due modi diversi. Il primo è quello di collegarsi ad un database di indirizzi mail noti come sorgenti di spam, mentre l'altro è basato sul vero e proprio riconoscimento di spam in base a mail considerate spam già viste dal sistema. Le reti neurali possono essere utilizzate per implementare questo secondo tipo di riconoscimento di spam.

9.1 I pattern di spam

Abbiamo trovato un insieme di pattern di spam sul sito dell'*UCI Repository of machine learning databases* ([2]). Questo insieme di pattern è stato costruito raccogliendo un certo numero di email di spam e un certo numero di email non spam ritenuto significativo. Sono state considerate spam varie tipologie di mail: pubblicità indesiderata e non richiesta, catene di mail, mail pornografiche ecc... In tutto sono state raccolte 4601 mail, di cui 1813 mail di spam. Ogni pattern è descritto da 58 attributi.

I primi 48 attributi sono reali compresi tra 0 e 100 e indicano la frequenza percentuale di determinate parole. Sono del tipo word_freq_WORD ed indicano la percentuale di parole WORD che compaiono nel testo della mail rispetto al totale delle parole. Nel nostro caso abbiamo:

- word_freq_make
- word_freq_address
- word_freq_all
- word_freq_3d
- word_freq_our
- word_freq_over
- word_freq_remove
- word_freq_internet
- word_freq_order
- word_freq_mail

- word_freq_receive
- word_freq_will
- word_freq_people
- word_freq_report
- word_freq_addresses
- word_freq_free
- word_freq_business
- word_freq_email
- word_freq_you
- word_freq_credit
- word_freq_your
- word_freq_font
- word_freq_000
- word_freq_money
- word_freq_hp
- word_freq_hpl
- word_freq_george
- word_freq_650
- word_freq_lab
- word_freq_labs
- word_freq_telnet
- word_freq_857
- word_freq_data
- word_freq_415
- word_freq_85

- word_freq_technology
- word_freq_1999
- word_freq_parts
- word_freq_pm
- word_freq_direct
- word_freq_cs
- word_freq_meeting
- word_freq_original
- word_freq_project
- word_freq_re
- word_freq_edu
- word_freq_table
- word_freq_conference

I successivi 6 attributi considerano la frequenza di caratteri rispetto a numero totale di caratteri presenti in una mail. Nel nostro caso:

- char_freq_;
- char_freq_(
- char_freq_[
- char_freq_!
- char_freq_$
- char_freq_#

I successivi tre attributi considerano le sequenze di caratteri maiuscoli. Il primo dei indica la lunghezza media delle sequenze di caratteri maiuscoli, mentre il secondo la lunghezza della sequenza più lunga. Il terzo infine indica la lunghezza totale del numero di tutte le sequenze di caratteri maiuscoli.

L'ultimo attributo indica naturalmente se la mail è da considerarsi spam (1) o non-spam (0).

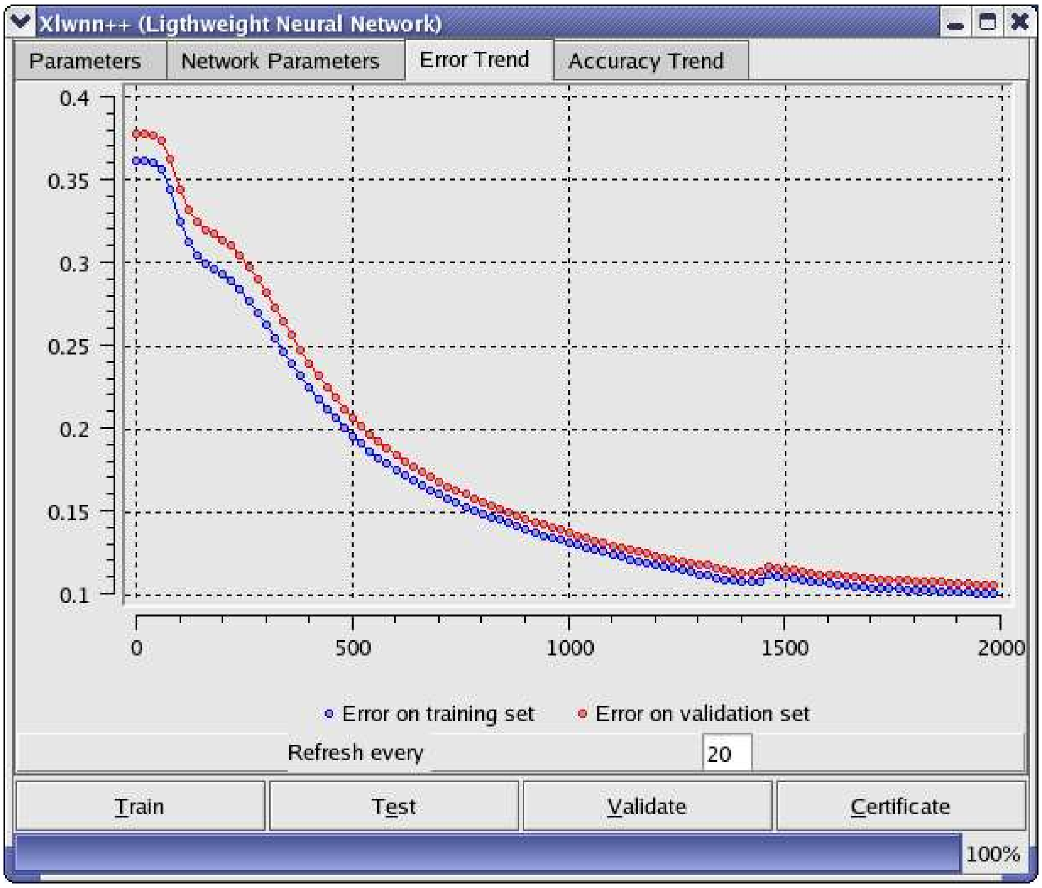

Figura 10: *Andamento dell'errore con una rete con uno strato nascosto di 10 neuroni*

9.2 Gli insiemi di addestramento, validazione, certificazione

L'insieme complessivo dei pattern di spam è stato diviso in tre insiemi. L'insieme di addestramento è formato da 2800 pattern. L'insieme di validazione contiene 920 pattern, mentre l'insieme di certificazione contiene 880 spam. Circa la metà dei pattern di ognuno degli insiemi è spam.

9.3 Addestramento della rete

I figura 10 si può vedere l'andamento dell'errore sull'insieme di addestramento e di validazione durante l'addestramento on line di una rete con un singolo strato nascoto di 10 neuroni su 2000 epoche. In figura 11 è mostrato

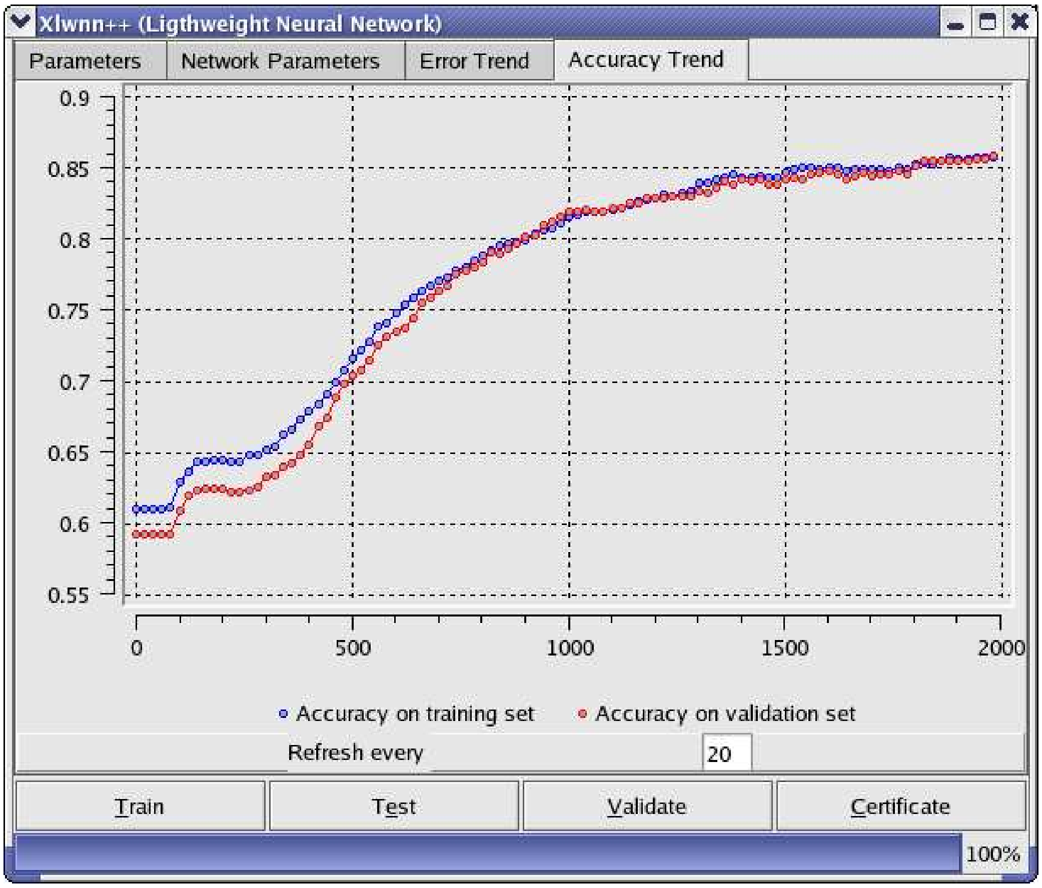

Figura 11: *Andamento dell'accuracy con una rete con uno strato nascosto di 10 neuroni*

invece l'andamento dell'accuracy (percentuale di risposte corrette della rete) sull'insieme di addestramento e di validazione.

Dalle figure si vede che l'accuracy della risposta della rete raggiunge l'85%. Sull'insieme di certificazione l'accuracy raggiunta è stata del'85.44%.

In figura 12 si può invece vedere l'andamento dell'errore con addestramento Super SAB. I risultati ottenuti sono migliori rispetto a quanto accade per l'addestramento on line, come si può anche osservare dall'andamento dell'accuracy sugli insiemi di addestramento e di validazione. Anche l'accuracy sull'insieme di certificazione migliora passando a oltre il 93%.

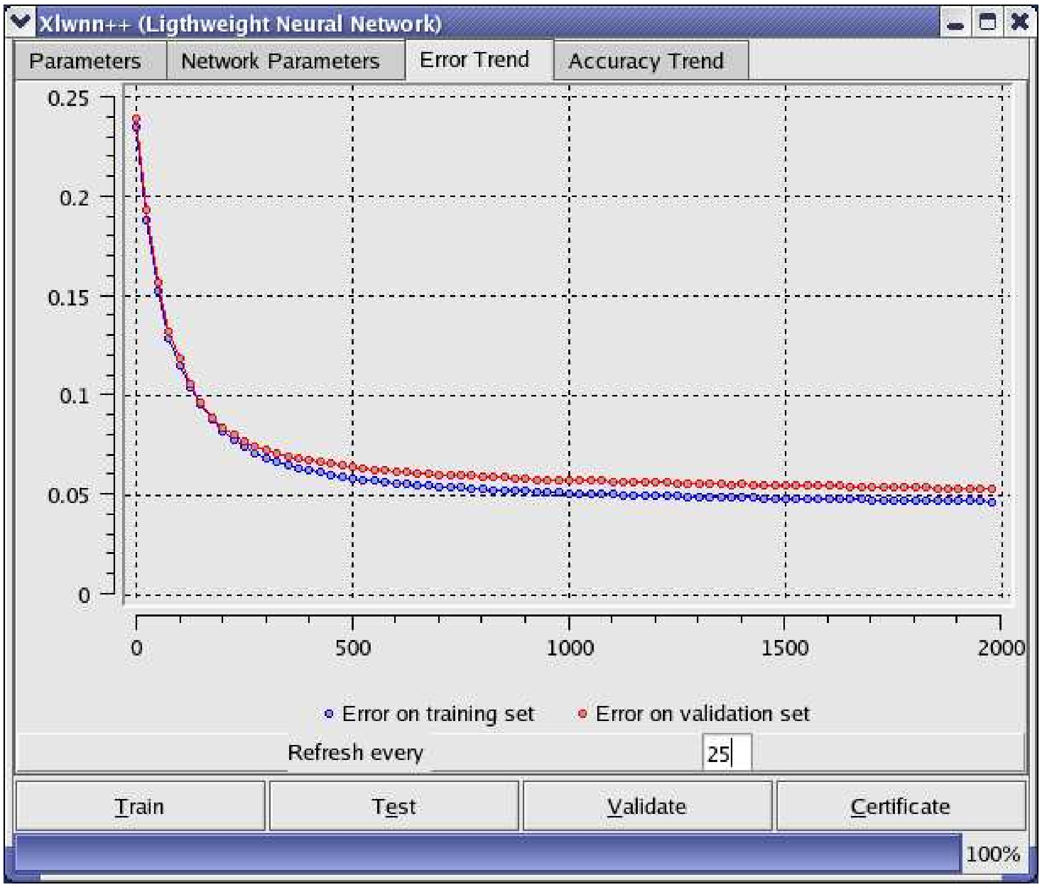

Figura 12: *Andamento dell'errore con una rete con uno strato nascosto di 10 neuroni con addestramento Super SAB*

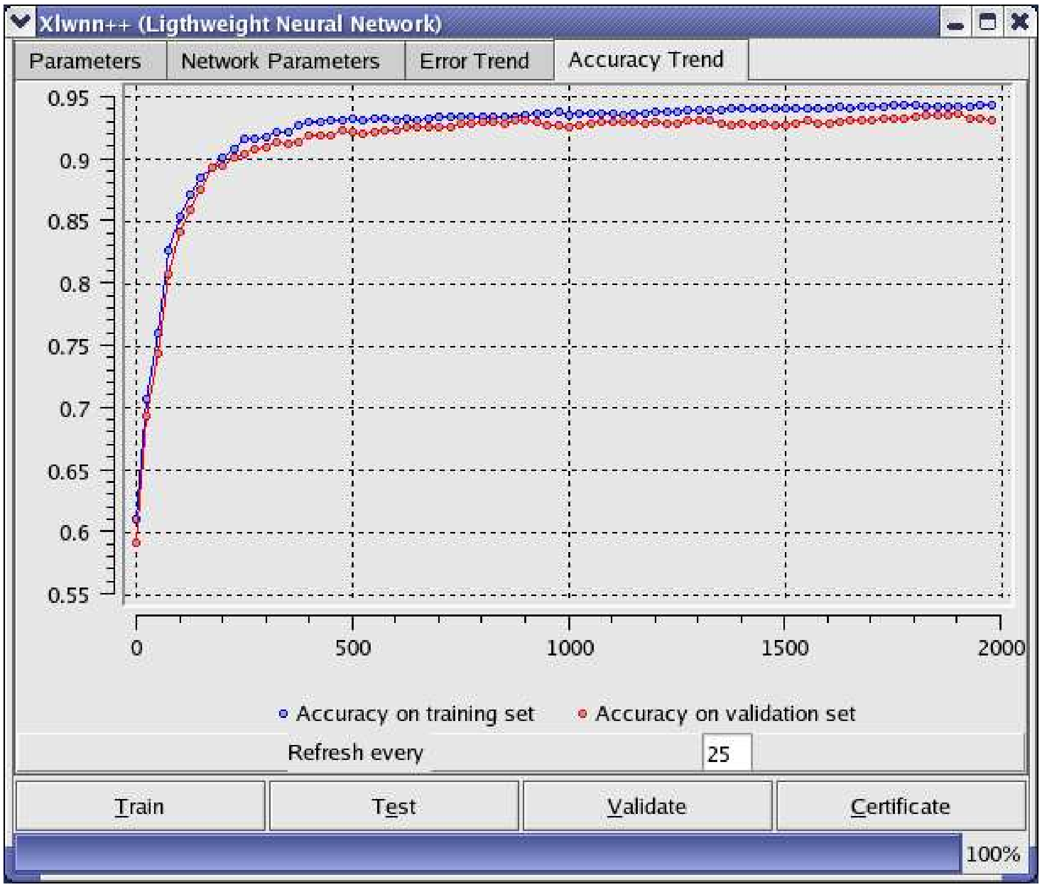

Figura 13: *Andamento dell'accuracy con una rete con uno strato nascosto di 10 neuroni con addestramento Super SAB*

10 Riconoscimento di volti

Questo esempio è tratto dal libro *Machine Learning* di Tom Mitchell ([1]).

Le specifiche dell'esercizio ([6]), insieme ai dati e al codice per leggere le immagini possono essere scaricate dal sito
`http://www-2.cs.cmu.edu/afs/cs.cmu.edu/user/mitchell/ftp/faces.html`

L'obiettivo della rete è quello di classificare immagini di volti di diverse persone riprese in pose diverse. Sono state scattate foto di 20 persone diverse, con circa 32 immagini per ogni persona, variando l'espressione (felice, triste, arrabbiato, neutro), la direzione verso cui stanno guardando (sinistra, destra, davanti, in alto) e se indossavano gli occhiali da sole.

Il campione disponibile su internet consiste in un totale di 604 immagini con una risoluzione di 30×32 pixels, ognuno dei quali è descritto da un'intensità in scala di grigio che varia da 0 a 255. In realtà le immagini originali avevano una dimensione di 120×128 pixels, ma sono state ridotte per diminuire la dimensione del vettore di input della rete. Come si può osservare in figura 14 ci sono altre variazioni nelle immagini, nello sfondo, nei vestiti indossati, e nella posizione della faccia all'interno dell'immagine.

Da questo campione si possono costruire tipi di target diversi, ad esempio si potrebbe addestrare la rete per riconoscere la direzione verso cui la persona guarda, oppure se indossa gli occhiali da sole.

Nel nostro esempio abbiamo scelto di addestrare la rete a riconoscere la persona. Il vettore di input di lunghezza $30 \times 32 = 960$ rappresenta i pixel dell'immagine scalati nell'intervallo $[0; 1]$. Il vettore di output contiene 20 neuroni, per classificare i vettori di input in 20 classi corrispondenti alle 20 persone. Ogni classe è codificata ponendo a 0.9 il neurone corrispondente alla risposta giusta, e a 0.1 gli altri neuroni. La scelta di questi valori, invece di 0 e 1, permette di evitare che i pesi debbano assumere valori eccessivamente alti per portare la funzione di trasferimento in una zona di saturazione. L'accuracy quindi è stata calcolata usando il metodo *Max Guess* cioè la risposta corrispondeva al neurone di valore massimo.

Negli esperimenti abbiamo utilizzato un *learning rate* η di 0.3 e un momento α pari a 0.3.

L'insieme totale dei pattern è stato diviso in tre insiemi. Il *training set* conteneva la metà delle immagini per ogni persona, il *validation set* un po' più di un quarto, e il *certification set* un po' meno di un quarto. Delle 604 immagini totali 315 sono state utilizzate per l'addestramento, 176 per la validazione e 113 per la certificazione.

Nelle figure 15 e 16 sono riportati l'errore e l'accuracy durante l'addestramento della rete. Si può notare che in questo esempio non si verifica l'overfitting, almeno durante le prime epoche. Comunque già dopo 200 epoche l'accuracy raggiunge il 100% anche sull'insieme di validazione. L'errore dopo 400 epoche raggiunge il valore di 0.032 sull'insieme di addestramento e 0.048 sull'insieme di validazione. Per l'insieme di certificazione abbiamo un errore pari a 0.051 e ancora un accuracy del 100%.

Visto gli ottimi risultati ottenuti con 10 neuroni nello strato nascosto abbiamo provato ad addestrare anche una rete con solo 5 neuroni nello strato nascosto. I risultati sono riportati nelle figure 17 e 18. In questo caso si verifica l'overfit e si ottiene la rete migliore all'epoca 722, raggiungendo un accuracy del 92.6% sul training set e del 90.9% sul validation set. Sull'insieme di certificazione si ottiene un errore vicino a quello sull'insieme di validazione ed un'accuracy del 92.0%.

Figura 14: *Le immagini di dimensione* 30×32 *per una delle persone.*

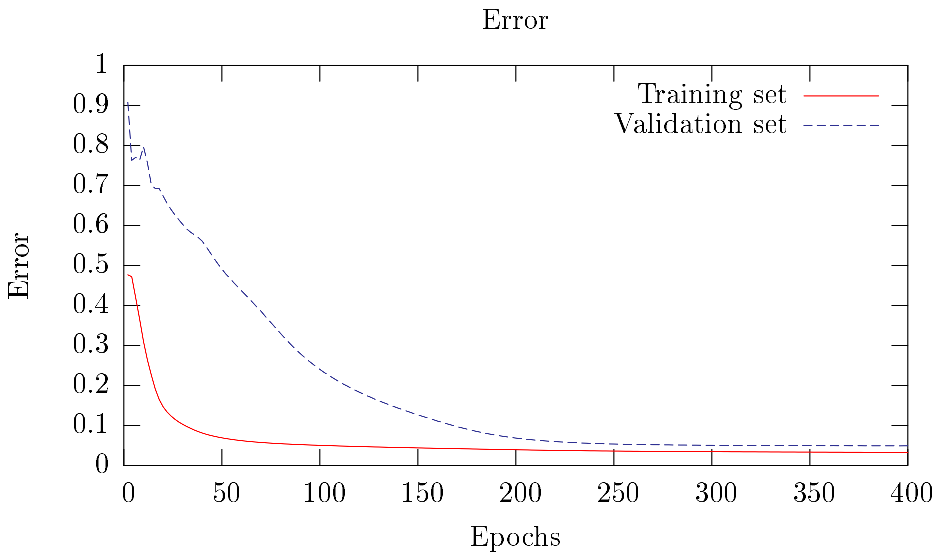

Figura 15: *Andamento dell'errore nell'addestramento della rete per il riconoscimento dei volti con 10 neuroni nello strato nascosto*

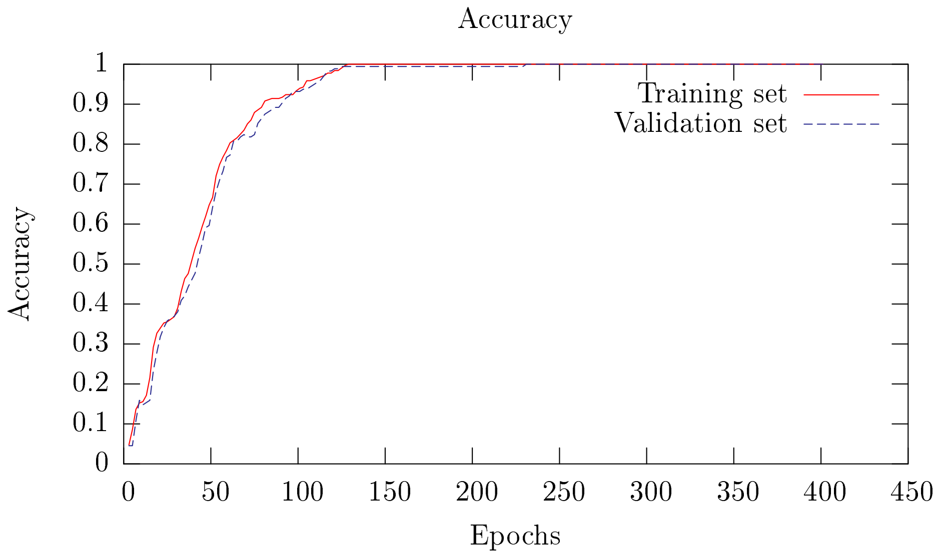

Figura 16: *Andamento dell'accuracy nell'addestramento della rete per il riconoscimento dei volti con 10 neuroni nello strato nascosto*

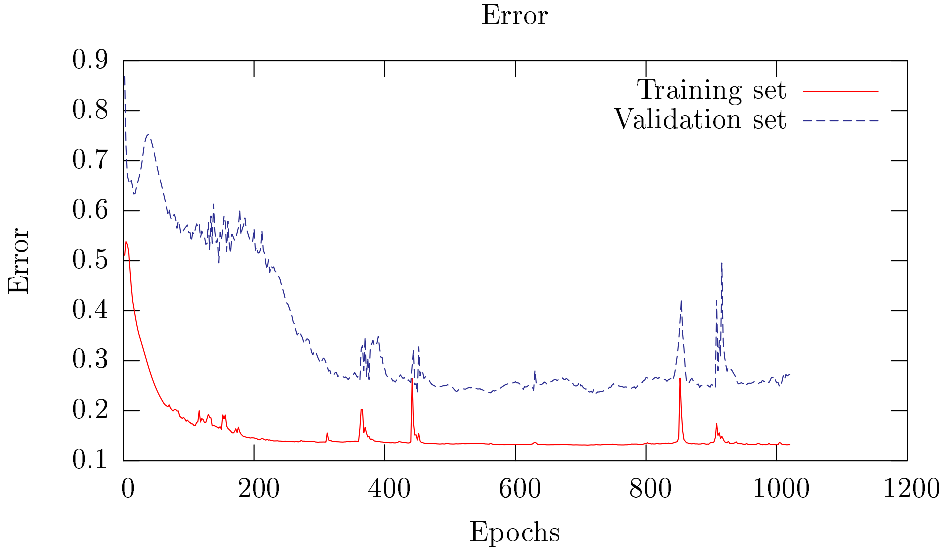

Figura 17: *Andamento dell'errore nell'addestramento della rete per il riconoscimento dei volti con 5 neuroni nello strato nascosto*

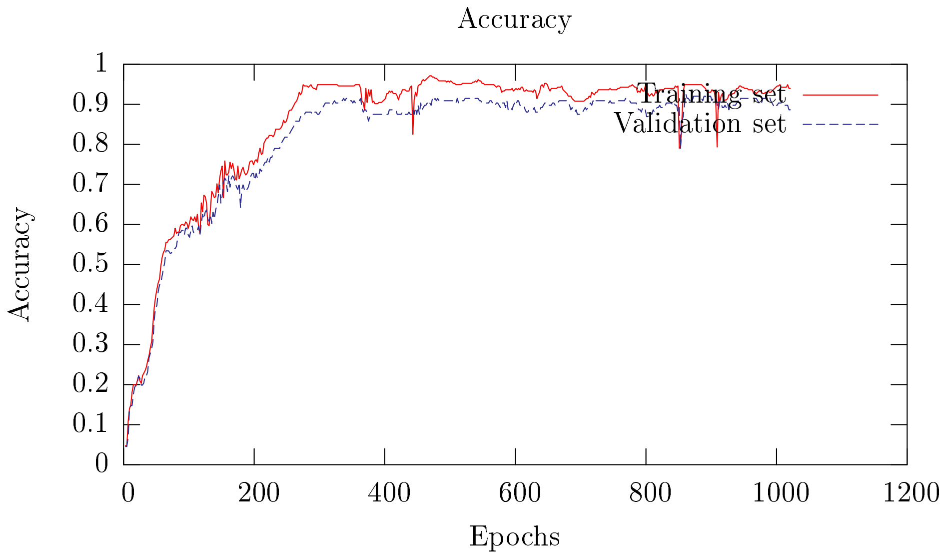

Figura 18: *Andamento dell'accuracy nell'addestramento della rete per il riconoscimento dei volti con 5 neuroni nello strato nascosto*

11 OCR: riconoscimento di cifre scritte a mano

In questa applicazione abbiamo addestrato una rete a riconoscere delle cifre scritte a mano. L'insieme dei campioni è stato scaricato dall'*UCI repository of machine learning databases* ([2]). Gli insiemi di pattern sono rappresentati da immagini in bianco e nero di dimensioni 32×32 pixels. Il *training set* contiene 1934 patterns, il *validation set* 946 patterns. Queste immagini sono state convertite in patterns di dimensione 16×16 in scala di grigi, in modo che ogni pixel dell'immagine ridotta corrispondesse a 4 pixel (un quadrato 2×2) dell'immagine originale (ed è ottenuto sommando i valori dei quattro pixel corrispondenti, e normalizzando i valori sull'intera immagine).

Come si può osservare nell'immagine 19, le cifre sono abbastanza variabili (sono state scritte da persone diverse), sia per la posizione che per l'inclinazione, la dimensione e la calligrafia.

La rete neurale ha 256 neuroni di input che rappresentano i pixel dell'immagine, ognuno con un valore compreso tra 0 e 1, 14 neuroni nello strato nascosto e 10 neuroni nello strato di output che codificano la cifra riconosciuta con un vettore che contiene 0 in tutte le posizioni eccetto in quella corrispondente alla cifra riconosciuta che è posto uguale a 1 nell'addestramento. Il riconoscimento avviene con la tecnica della *max guess accuracy*, cioè controllando quale sia il neurone con l'output maggiore.

La rete è stata addestrata con la tecnica *on line*, con *learning rate* 0.6 e momento 0.1, ma presentando ad ogni batch i patterns in un ordine casuale. Nelle figure 20 e 21 sono rappresentati gli andamenti dell'errore e dell'*accuracy*. La rete migliore è stata ottenuta dopo 29 epoche e commette 4 errori su 1934 sul *training set* (*accuracy* del 99.79%) e 20 errori su 946 sul *validation set* (*accuracy* del 97.89%).

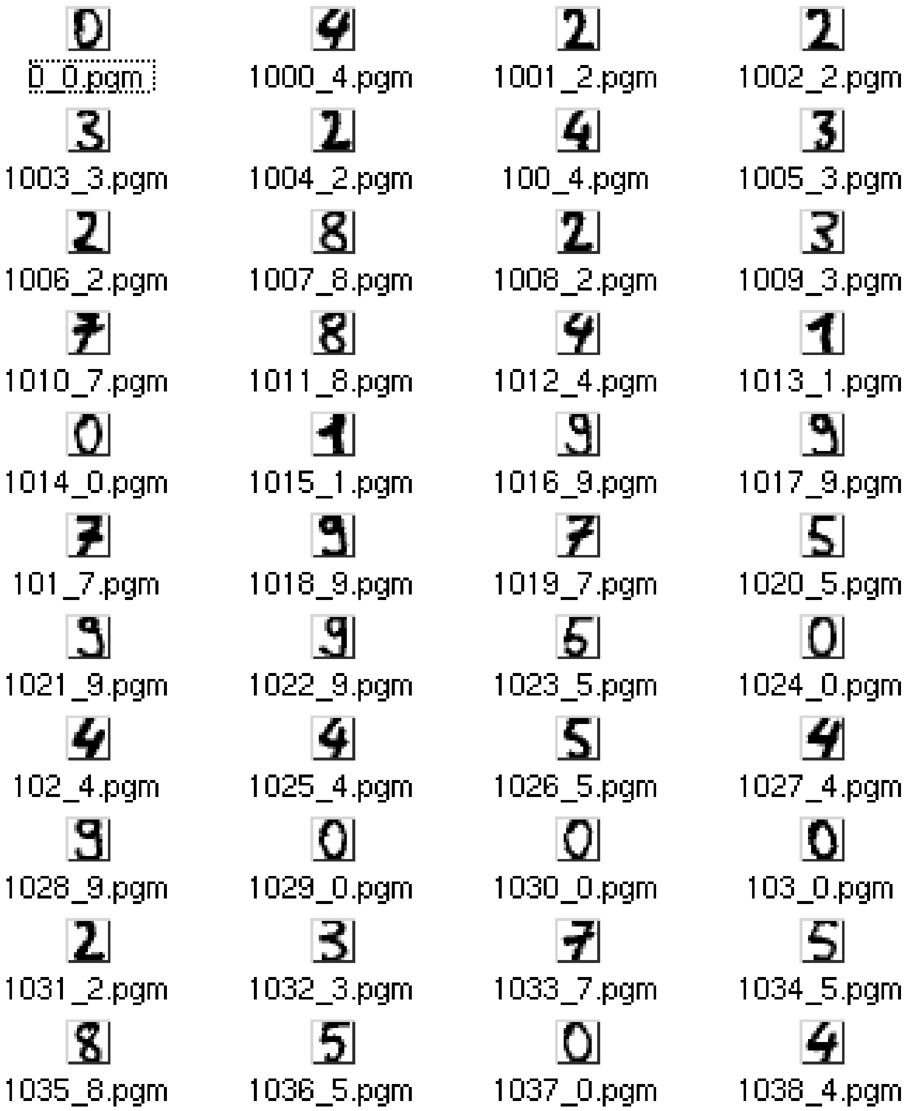

Figura 19: *Alcuni esempi di immagini di dimensione* 16×16 *usate nell'addestramento della rete*

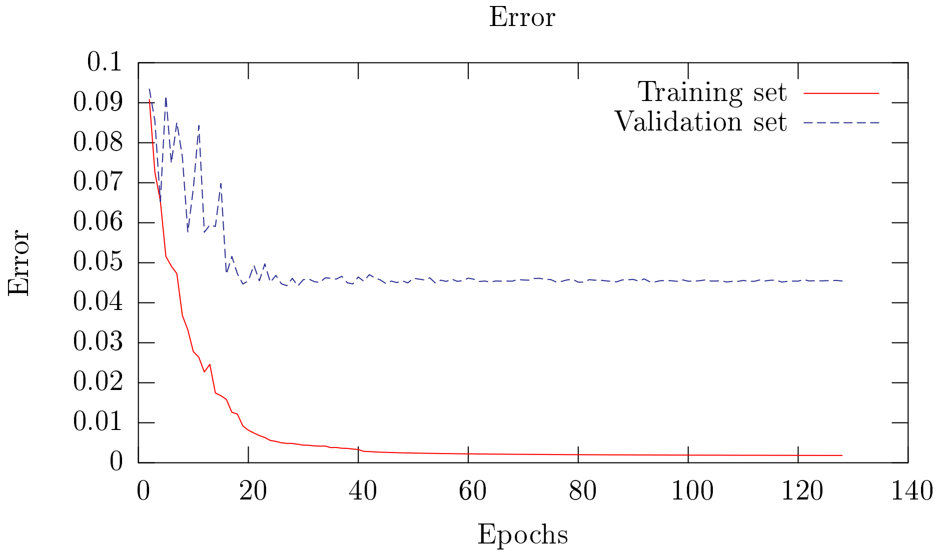

Figura 20: *Andamento dell'errore nell'addestramento della rete per il riconoscimento delle cifre scritte a mano con 14 neuroni nello strato nascosto. La rete migliore è stata trovata all'epoca* 29

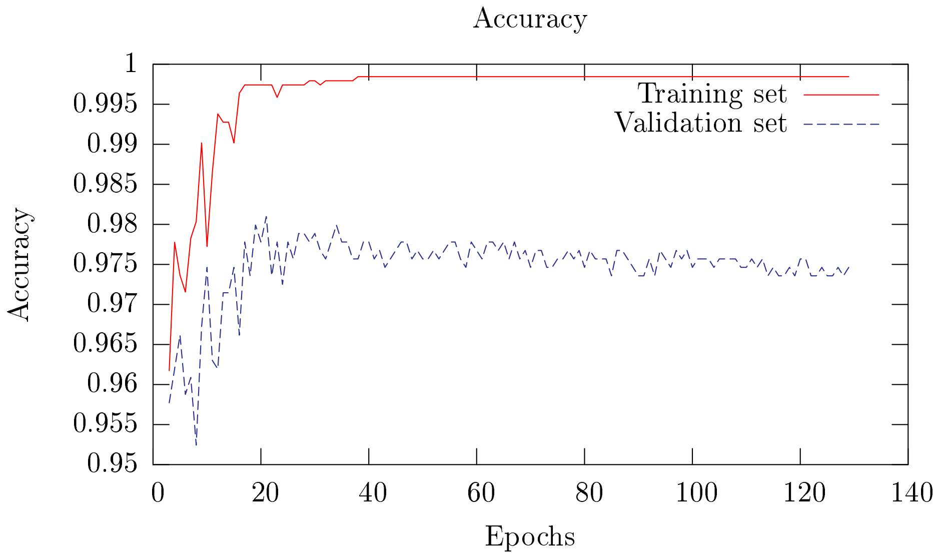

Figura 21: *Andamento dell'*accuracy *nell'addestramento della rete per il riconoscimento delle cifre scritte a mano con 14 neuroni nello strato nascosto*

12 Benchmark

Abbiamo testato la nostra applicazione su problemi di riconoscimento classici, estratti da "Proben1" ([7]). Le caratteristiche di questi problemi sono mostrate in tabella 1, per maggiori informazioni sugli insiemi di dati si può consultare [7].

Problema	Tipo	Numero di pattern	Input	Strato nascosto	Output
building	a	4208	14	16	3
card	c	690	51	32	2
gene	c	3175	120	4+2	3
mushroom	c	8124	125	32	2
soybean	c	683	82	16+8	19
thyroid	c	7200	21	16+8	3

Tabella 1: Caratteristiche dei problemi di benchmark. Il tipo può essere c (classificatore) o a (approssimazione) e $16+8$ nella colonna "strato nascosto" indica che ci sono due strati nascosti, il primo con 16 e il secondo con 8 neuroni.

I risultati sono stati confrontati con *fann* ([4]), una libreria simile alla nostra, sviluppata da uno studente danese per la sua tesi, il cui sito di riferimento è **http://fann.sourceforge.net/**.

Per ognuna delle prove eseguite abbiamo diviso l'insieme dei pattern in un insieme di addestramento e in un insieme di validazione, rilevando l'errore raggiunto dalle due librerie con reti identiche ogni secondo per 120 secondi di esecuzione. Dopo i 120 secondi abbiamo anche rilevato il numero di batch eseguite dalle due librerie. Riportiamo nella tabella 2 i risultati ottenuti con un Athlon 1.33 GHz con 256 MB di RAM.

Come si vede dalla tabella generalmente il numero di batch eseguite dalla nostra libreria è superiore a quanto fatto registrare da *fann* e l'errore è di solito paragonabile sia sull'insieme di validazione che sull'insieme di addestramento. Ad esempio nel primo problema test (*building*) la nostra libreria ha eseguito 12361 batches contro le 7788 batches di *fann*. L'errore raggiunto sul training e sul validation set è per entrambi dell'ordine di 7.5e-03 6.0e-03 rispettivamente (solo leggermente migliore per *fann*).

I grafici che seguono riportano sulle ascisse i secondi e sulle ordinate l'errore raggiunto a quel tempo. Generalmente la nostra libreria, essendo più veloce, riesce a completare un numero maggiore di batch e quindi a raggiungere un errore minore a parità di tempo. Fa eccezione il problema *gene* per il quale i risultati della nostra libreria sono peggiori di quelli di *fann*.

	lwnn++		fann	
Problema	Precisione	Batches	Precisione	Batches
building (test)	7.6196e-03	12361	7.5093e-03	7788
building (train)	6.0234e-03	12361	5.9211e-03	7788
card (test)	1.8808e-01	19149	1.9739e-01	14041
card (train)	3.3144e-02	19149	4.7671e-02	14041
gene (test)	4.4142e-01	11312	3.2057e-01	7233
gene (train)	2.6935e-01	11312	7.0173e-02	7233
mushroom (test)	9.2239e-07	727	1.0071e-06	588
mushroom (train)	5.1746e-07	727	5.2275e-07	588
soybean (test)	1.6387e-01	18197	1.4284e-01	13438
soybean (train)	1.7081e-05	18197	2.9258e-02	13438
thyroid (test)	4.9661e-02	4203	6.8455e-02	2867
thyroid (train)	1.1874e-02	4203	3.2830e-02	2867

Tabella 2: Benchmark

In figura 28 è rappresentato il numero di batch portate a termine dalle due librerie a parità di tempo. La nostra libreria risulta più veloce.

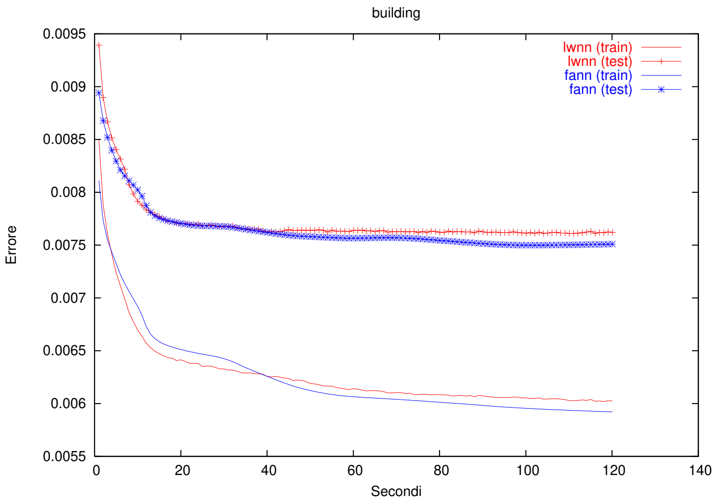

Figura 22: *Building. Andamento dell'errore sul set di training e di validation*

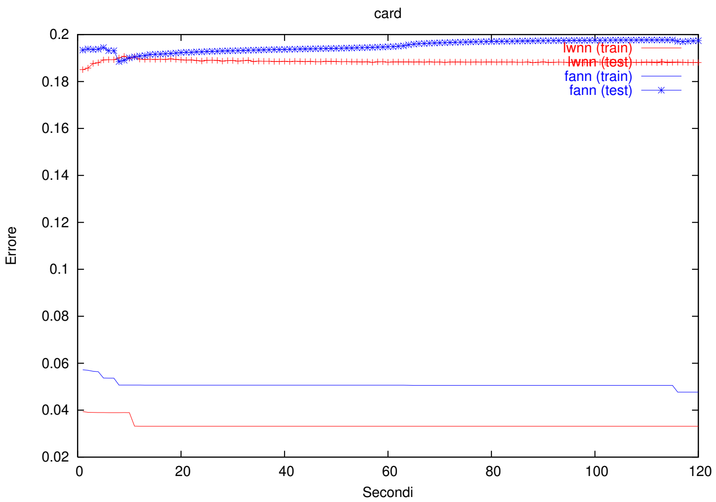

Figura 23: *Card. Andamento dell'errore sul set di training e di validation*

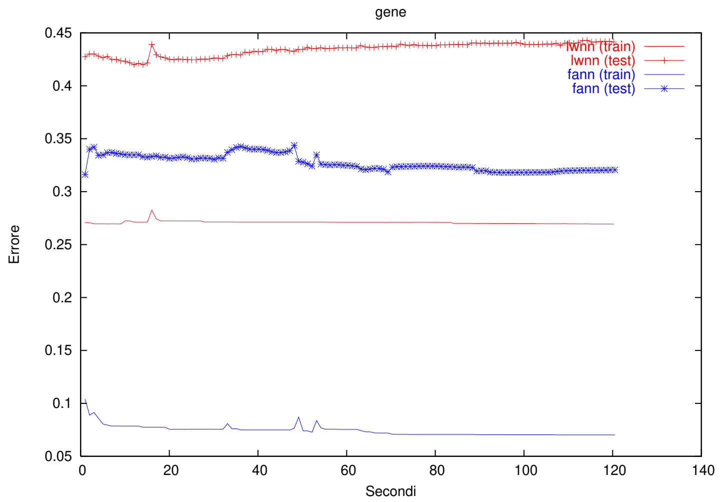

Figura 24: *Gene. Andamento dell'errore sul set di training e di validation*

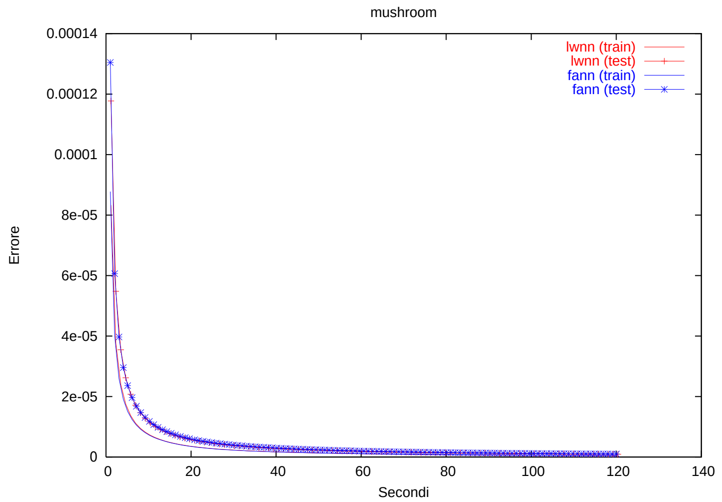

Figura 25: *Mushroom. Andamento dell'errore sul set di training e di validation*

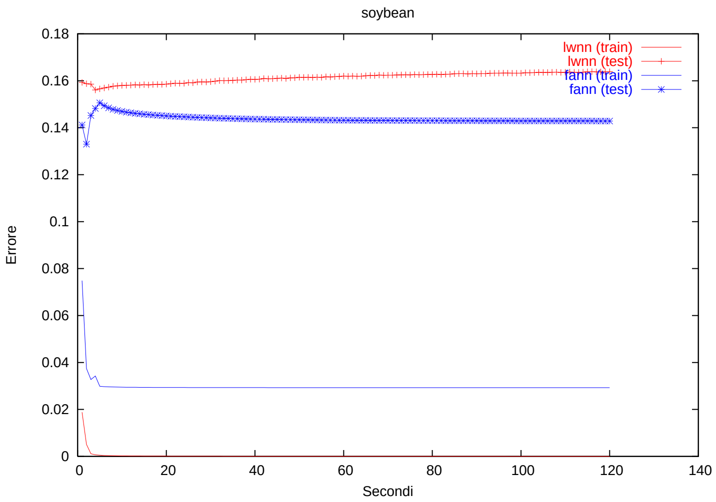

Figura 26: *Soybean. Andamento dell'errore sul set di training e di validation*

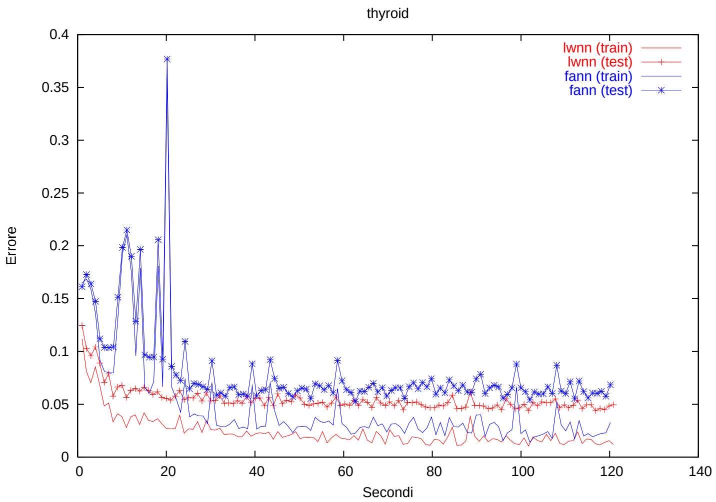

Figura 27: *Thyroid. Andamento dell'errore sul set di training e di validation*

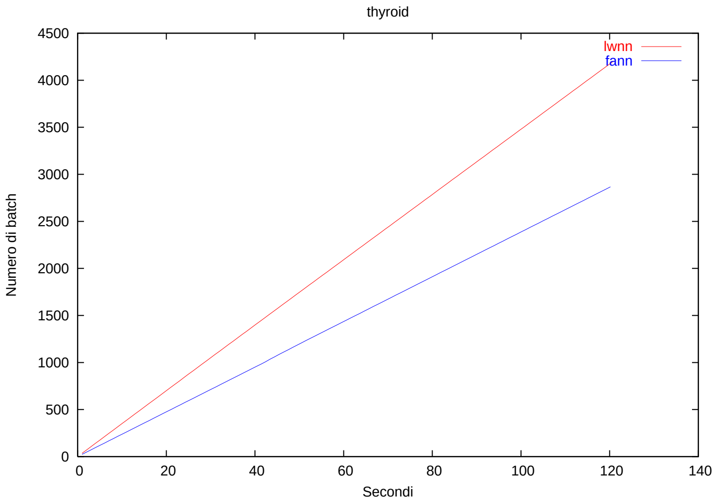

Figura 28: *Numero di batch eseguite nel tempo dalle due librerie per il problema "Thyroid". Si osserva che la libreria* Lightweight Neural Network++ *è più veloce*

Riferimenti bibliografici

[1] Tom M. Mitchell, *Machine Learning* (1997) (cap 4 “Artificial Neural Networks”). WCB / McGraw-Hill.

[2] Blake, C.L. & Merz, C.J. (1998). UCI Repository of machine learning databases [http://www.ics.uci.edu/ mlearn/MLRepository.html]. Irvine, CA: University of California, Department of Information and Computer Science.

[3] Peter Van Rossum, Luca Cinti, Lorenzo Masetti (2003). *Lightweight Neural Network* http://lwneuralnet.sourceforge.net

[4] Steffen Nissen (2003). *Implementation of a Fast Artificial Network Library* http://fann.sourceforge.net

[5] J. Heller (2002). *Jet's neural library* http://www.voltar.org/jneural/

[6] Tom M. Mitchell (1997). Homework 3: Neural Networks and faces images.
http://www-2.cs.cmu.edu/afs/cs.cmu.edu/user/mitchell/ftp/faces.html

[7] Prechelt, L. (1994). Proben1 - a set of neural network benchmark problems and benchmarking rules.