The Lightweight Neural Network ++ project: a C++ implementation of a multilayer perceptron feed forward neural network with backpropagation learning

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

May 9, 2004

Abstract

This document is a technical description of the *Lightweight Neural Network++* project. The project provides a library implementing a feed forward neural network and was developed at the University of Firenze (Italy), as a project for the examination of Neural Networks, extending an original project by Peter Van Rossum of the University of New Mexico. The article gives a little introduction about the theory of feed forward neural networks, the description of the choices for implementing the library, a guide for using and extending the classes, and some example applications. The library was benchmarked and compared to an analogous free software library.

Contents

1 Features and objectives of Lightweight Neural Network++ Project

This project is the implementation of a general feed forward neural network with backpropagation training.

With *Lightweight Neural Network++* library it is possible to create feed forward neural networks with with any number of layers and any number of neurons per layer.

The two main types of sigmoidal functions are provided: logistic function (which gives an answer between 0 and 1) and tanh function (which gives an answer between -1 and 1)

For network training the main techniques, studied in this field, are available: normal on line training, the momentum technique, batch training and Super SAB (Self Adaptive Backpropagation) training. All these techniques can be used together, and the user should choose the technique that fits better the problem he has to solve.

The intended audience of the library is, at one level, the C++ programmer who needs a class implementing a simple neural network. He's given also a class which implements the standard techniques for training a network, making easy to write a program for training a neural network. But the project is also good for another type of audience, because it comes with two graphical user interfaces, which let the end user (without programming experience) train a network easily.

The library objectives are mainly efficiency (i.e. speed in computing output and in learning) and use of minimum quantity of resources (memory, CPU and disk space). This objective is declared in the name "lightweight".

The library is released as free software, and so open source, under the LGPL license. This license imposes that not only a derived software, but also every program which uses it by linking must be free software. This way we would like to encourage the development of free software in this field.

In fact there's not a lot of free software projects in the field of neural networks. A lot of libraries with similar (and, of course, more advanced) features to lwnn++ are commercial closed source projects. But not all of them are lightweight. For example the MATLAB's neural network tool box, sold at a very high price, needs a lot of computing resources, even to solve quite easy problems.

Another valid library, released under LGPL, highly speed-optimized, is *fann* (Fast Artificial Neural Network). The author (Steffen Nissen), who developed the library as degree thesis, makes some benchmarks using the *Lightweight Neural Network* library by Peter Van Rossum (of which our project is an extension) for comparing. For more details you can read the documentation of *fann* ([4]).

We used the benchmarks of fann library to test the speed of our project. Results are presented in section 11.

The site where you can download last version of our project is **http://lwneuralnetplus.sourceforge.net**.

2 A little theoric introduction

2.1 Multilayer Perceptron Networks

It should be clear that artificial neural networks have nothing in common with human or animal brain. They are just a way of approximating very complex functions $\Re^n \rightarrow \Re^m$. The network is given an input which is a vector of length n and it produces an output, which is a vector of length m. The interesting point is that the network has a training phase, when it "learns" to associate some outputs to some inputs. An artificial neural network is a sort of distributed associative memory. But, if properly used, neural networks can learn how to generalize, that is they can give a proper answer even to inputs they haven't seen before. This is a very interesting feature that let a computer perform tasks that are very difficult to solve with traditional approaches.

Multilayer Perceptron networks were introduced by Rumelhart and others, revising a learning method proposed by Werbos in 1977. This type of network extends the concept of Percptron, introduced by Rosenblat in 1958, by inserting one or more hidden neuron layers between the input layer and the output layer. Every neuron of the k-th layer is connected to every neuron of the previous ($k-1$-th) layer by a weight. The neurons of each layer compute synchronously the weighted mean of the outputs of the previous layer and then apply to the result the *activation function* (or *sigmoidal function*) that is usually the logistic function

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

but can be the tanh function

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

the Gaussian function, or a simple linear function. Different layers can have different activation functions. The graph of logistic function is shown in fig.

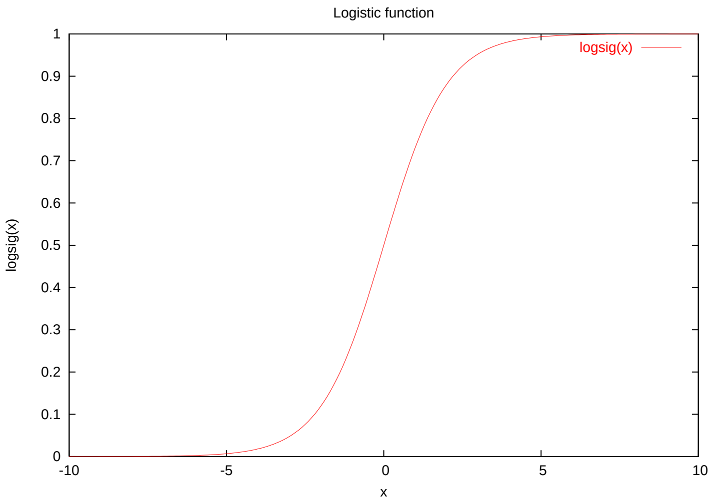

Figure 1: *Logistic function*

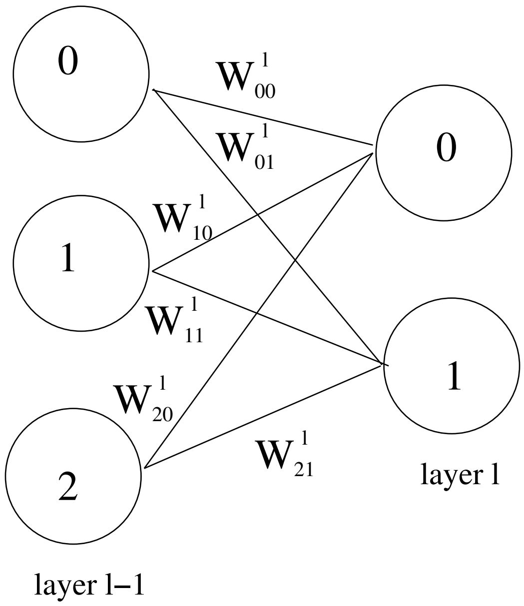

Figure 2: *Two layers of a network and the weights connecting them*

1, where we can see that it takes values between 0 and 1 and it's very close to the boundary values for $x < -10$ and $x > 10$.

Given a n-layers multilayer perceptron network, indexed by $k = 1 \dots n$, let's suppose we have to make the network learn to associate input vectors $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_P$ with the wanted outputs $\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_P$. We can call

- $O_j^{p,n}$ output of j-th neuron of n-th layer when input layer is set to p-th pattern
- y_j^p j-th component of wanted answer for p-th input.
- W_{ij}^n weight connecting i-th neuron of $n-1$-th layer to j-th neuron of n-th layer.
- t_j^n bias of j-th neuron of n-th layer.

If we do set the input layer with p-th pattern, we should minimize the mean square error:

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

where m is the output layer dimension. The function returned by the net, which depends on all the network weights, is

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

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

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

Biases can be simulated by an extra neuron with output 1 and connected to the next layer neurons by weights t_j^n. So we can assume $t_i^n = 0$ in the following, and we can write for the output layer

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

By varying the weights connecting the second-last layer with the last one of the quantity ΔW_{ij}^n, the variation on square mean error E^p is

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

Given that $\frac{\partial E^p}{\partial W_{ij}^n} = \frac{\partial E^p}{\partial O_j^{p,n}} \frac{\partial O_j^{p,n}}{\partial W_{ij}^n}$, we shall compute

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

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

choosing $\Delta W_{ij}^n = -\eta \frac{\partial E^p}{\partial W_{ij}^n}$ for weight variation, we obtain

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

where $\eta \in (0,1)$ is the *learning rate* parameter, which rules the learning speed.

With this choice we have

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

so that the mean square error decreases on every weight update. The last problem we have to deal with is to write the derivative of sigmoidal function. If we chose the logistic function, we can see that

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

and the weight update becomes

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

If, instead, the sigmoidal function is *tanh* $f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ we have

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

and the weight update expressions should be modified according to this result.

In the same way we can derive the expression for the variation of the weights connecting $(n-2)$-th to $(n-1)$-th layer:

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

These expression can be written in an iterative form, used by practical applications, the δ-rule

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

where

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

The network training takes place by weight updates, beginning from the last layer and going back through all the layers up to the input layer. This type of learning is called *backpropagation*. On the other hand computing the answer happens by setting the pattern on the input layer and propagating to the output layer. For this reason we talk about *feed forward networks*.

2.2 On line training, batch training and SuperSAB training

According to what we said before, training is done by giving the network an input pattern and by updating the weights according to expression 1. We keep on training this way for all training patterns, until we reach a correct answer. This is called *on line training*. If the patterns are very different it is possible for weights to vary too much, and we must choose a very small η, that makes the learning process very slow. It can be convenient to update the weights in batch, trying to minimize the value

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

and making the weight update be the average variation for each weight. For example for the last layer's weights we have

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

Real update happens after the network is given all training patterns: we call *epoch* the period in which we present to the network the whole training set, so we can say that weights are updated each epoch. This learning technique is called *batch training* and, in some particular problems, can lead to a higher speed. (But, in our experiments, this seems not to be true, and we found some articles claiming that the advantage of batch training is just an urban legend).

Another learning technique, which let every weight move to the minimum at a different speed, is the *Super SAB* technique, which is described in section 5.8.

3 Development history of the library

The *Lightweight Neural Network++* library is written in C++ and is derived by an analogous library written in *C*, *Lightweight Neural Network*. This library was developed by prof. Peter Van Rossum from Dept. of Mathematical Sciences at the New Mexico State University, Las Cruces, NM, USA ([3]). Looking for a free implementation of a feed forward neural network, on October 2003, we found this project which was in a development stage (version 0.5). The library was highly optimized and was organized with an object-oriented approach, although it was written in a non-object-oriented language. But the features were not much: it provided only on line training with the possibility to use a momentum, the only activation function was the logistic function, and a network could be saved only in a textual format.

Thanks to its author's helpfulness, we collaborated to the project, writing the code for saving a network on a binary file (reducing this way the dimension of files) and for batch training. This library is still available at the address `http://lwneuralnet.sourceforge.net/`.

In a first period we kept on developing the original library, but then we decided to create a new project written in a more modern language, C++, an object language, to make the object-oriented architecture of the project explicit. The **network** class of our library reuses a lot of code of the original project, providing a similar interface but it does provide some extra features:

- Possibility of choice between logistic and tanh function as activation function.
- Super SAB training and possibility to combine it with batch training.
- Possibility of combining batch training with momentum method.

Our project includes the class **trainer** which implements standard algorithms for training a net, two simple user interfaces, and a framework to read patterns from files in any format you like (if you know how to write a class which reads that format, of course!)

4 Project structure

The main classes are two: **network**, which defines a data structure implementing a multi-layer perceptron neural network, and **trainer** that let you train, validate and certificate a neural network on a set of patterns saved on a file. The interfaces of these classes are documented in detail in the library documentation.

The **trainer** class uses an object of class **network** that is the network to be trained.

In order to give the maximum flexibility, pattern reading is managed by an object of **iomanage** class. This is an abstract class, i.e. a class which only defines the signatures of two methods: the first one is used to read from a file the number of input/output couples it contains, and the dimensions of input and output vectors (that are as well the dimensions of input and output layer of a network that works on those patterns); the second one is used to read the values of patterns.

Since this class has some methods that are not implemented, it is not possible to create an object of **iomanage** class. But it is possible to write a derived class of **iomanage** that manages the reading of some file format. By default the trainer uses an object of class **iomanagelwnnfann** which manages two formats for saving patterns, that are described in section 6.4. The class **iomanagebinary** manages files in binary format.

In the following sections we describe the two main classes and the implementation choices.

5 The network class

5.1 Implementation of the data structure

Main informations we have to keep in order to implement a neural network are the values of weights connecting each neuron of a layer with all the neurons of the previous layer.

Class **network** has an integer field named *no_of_layers* that holds the number of layers (input and output layers are counted so that a network must have at least two layers) and a dynamically allocated array *layer* of length *no_of_layers* holding pointers to structures of type **layer_t**. Layers are counted starting from input layer (`layer[0]`), to the output layer (that is `layer[no_of_layers-1]`). For efficiency and readability two direct pointers to input and output layer are kept.

Declaration of layer_t structure

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

Each level holds the information about the number of neurons in that level and a pointer to a dynamically allocated array, whose length is equal to the number of neurons, of pointers to structures of type **neuron_t**.

Declaration of neuron_t structure

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

Every neuron holds three arrays of floating point numbers (**float**), whose length is equal to the number of neurons of previous layer. These arrays are allocated only in the layers starting from first hidden layer (`layer[1]`).

The implementation of the network does not use any bias, not even a simulated one. This choice, which makes the implementation very simple, does not imply a worse performance of the network.

The *weight* array holds the values of weights connecting the neuron to neurons of previous layer, other two arrays *delta* and *sumdeltas* and the *error* field are used in the implementation of δ-rule during the training phase.

Other fields of the network class are: the learning rate (*learning_rate*), the activation function (*activation*), the α parameter for the momentum (*momentum*), and the network global error (*global_error*). Some other fields are used for batch training (*no_of_patterns*) and for Super SAB training (see sec. 5.8).

5.2 Network constructor and default parameters

In order to use a neural network you have to construct an object of class **network**, on which you can call the methods that let you train the network or use a trained network for a simulation.

Network class provides three different constructors.

Varying length constructor In this constructor, the first parameter is the activation function of the network (must be one of the predefined constants, *network::LOGISTIC* or *network::TANH*), the second one is the number of layers of the network (must be > 2). The last ones are the number of neurons in every layer and must be the same number of the layers. For example in order to construct a neural network that uses the logistic function with 3 layers of 20, 30 and 10 neurons you have to write

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

Constructor by vector This constructor still takes as first argument the activation function, and as second parameter a vector of integers (**vector<int>**) which defines the number of neurons in each layer (and, by its length, the number of layers). In this example a pointer to a network object is constructed:

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

Constructor by a file The third type of constructor takes a file name on which a neural network was saved. A second optional boolean parameter, if set to false, tells the constructor to load the network from a text file. Binary files are smaller, of course!

```
network mynet("trainednetwork.net ");
```

When a network is constructed for the first time, its weights are initialized with small random numbers in range $[-r; r]$ (so that their mean is 0). By default the range of the initial weights is $[-0.1, 0.1]$ if the activation function is the logistic one, and $[-0.05; 0.05]$ if the activation function is tanh. This choice can be overwritten by calling the method **randomize (float range)** which assigns to the weights random values in the range $[-range; range]$. When a network is constructed other parameters are set: the learning rate to 0.3 if logistic function is used, and 0.1 if tanh is used, the momentum to 0. These default settings can be changed after the construction of the network.

5.3 Feed forward implementation

Field *output* of each neuron holds its output, given, for each layer l ($l = 1 \ldots no\_of\_layers - 1$) and for each neuron $i = 0 \ldots n_l - 1$ by

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

where O_i^l is the value of output of i-th neuron of l-th layer (`layer[l]-.neuron[i].output`), n_l is the number of neurons of l-th layer (`layer[l].-no_of_neurons`), and W_{ji}^l is the value of the weight connecting j-th neuron of $(l-1)$-th layer an i-th neuron of l-th layer (in our implementations it's `layer[l].neuron[i].weight[j]`).

The σ function can be the logistic function with $k = 1$

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

or the tanh function

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

In order to compute the network output we save the input vector in the *output* fields of input layer neurons and then feed forward propagate the result to the output layer, which will hold in *output* fields of its neurons the result of computing.

5.4 Optimizations techniques

In a neural network implementation, efficiency is a critical point. Although CPU speed and RAM availability is getting higher and higher, so that in the last years cheap personal computers give the common users a computing power that some years ago was just a dream (library development and testing was made on 1.2 / 1.3 GHZ Athlon processors with 256 MB RAM), it is still true that in this type of applications, resources are over-utilized. For these reasons, an optimization reducing computing times of a factor 3 or 4 can be essential.

Steffen Nissen underlines in his degree thesis ([4]) some possible types of optimization in this field

1. Algorithmic optimization

2. Architectural optimization

3. Cache optimization

4. Table lookup

5. In-lining of code

Algorithmic optimization is certainly the optimization type which gives more results. If you could change the running time of a piece of software, from $O(n^2)$ to $O(n)$, then this optimization would certainly be better than all other optimizations you can think of. But in the field of neural networks there's no much space for algorithmic optimization.

About architecture optimization, we could have chosen a more object-oriented approach, for example by implementing a **layer** class and a **neuron** class. But this approach (followed, for example, by *Jet's Neural Library* [5]) has some efficiency disadvantages, due to poor cache optimization and to overhead for calling methods in different classes.

In order to assure good cache performance, data should be accessed sequentially. For this reason, the *fann* library allocates all weights in a single array. However, our implementations gives good results too. A sequential approach is followed in our library to hold learning rates for Super SAB learning.

Key optimizations in the implementation of *Lightweight Neural Network++* are the last two ones, which we show in some detail in following sections.

5.4.1 Sigmoidal function table lookup

A critical point for computing speed of a neural network implementation is the computing of the activation function. This function has to be non linear, and so computing its value with standard methods used by a programming language would be highly inefficient. A good way of computing sigmoidal functions is to use table lookup of a sufficient number of points, and then use the nearest tabled value to the point where we want to compute the function.

By default sigmoidal functions are tabled, at compiling time, on 2000 points at fixed step intervals (between -12 and 12 for logistic function, between -3.11 and 3.11 for tanh). The number of points and values of bounds can be changed when you compile the library. These values are saved in a 2000×2 matrix, where column index is for the used function (0 for logistic function and 1 for tanh).

If the point x where we want to compute the sigmoidal function is not between the two table boundary values, the function value is approximated with its limit values (0 and 1 for logistic function, -1 and 1 for tanh), if it's between the boundaries the table value is returned of index

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

if the function is tabled between a and b in n points.

With this approximation method, computing the sigmoidal function costs two elementary operations, two comparisons and a vector access.

5.4.2 In-lining of key-functions code

Calling a function (or a method) implies a overhead in code execution. Standard translation in machine language of a function call requires passing the parameters on the stack, a jump to function address, taking parameter values, writing returned values on the stack, and taking them back in the caller procedure.

If a function is called in the inner loop of a critical algorithm, as in the case of function computing sigmoidal function value, it is useful to declare the function *inline*, so that the compiler inserts the code of called function everywhere it is called, instead of making a standard function call.

Of course, the same code will be repeated for all times the function is called, creating greater object files and executables, so that this method must be used only for key functions and for very simple functions.

Using in-lined code has brought to reduce computing times of a factor 6.

5.5 Backpropagation implementation with δ-rule

The main issue in network learning is the implementation of the δ-rule. To train the network you have to give it some examples of (input, output) couples. To make the network learn to associate *target* vector to *input* vector, you have to call the method **compute (coast float *input, float *output)**, the same one you can use to get the result once the network is trained. Second parameter can be `NULL` if, during training, the returned value is not interesting, because output values are kept in the last layer neurons, as we explained in section 5.3.

After calling this method, you have to call **compute_output_error (const float *target)** which executes the first δ-rule step, by computing values of last layer δ's according to expression:

$$\delta_i^n = \begin{cases} O_i^n(1-O_i^n)(T_i - O_i^n) & \text{if logistic function is used} \\ (1-(O_i^n)^2)(T_i - O_i^n) & \text{if tanh function is used} \end{cases} \tag{4}$$

where O_i^n is the output of i-th output layer neuron ($n = no\_of\_layers - 1$) and T_i is the i-th element of learning target vector.

The next step, common to all learning techniques, is to compute all the previous layers' δ's. This can be done starting from last layer, for all layers l from *no_of_layers -2* to 0.

$$\delta_i^l = O_i^l(1 - O_i^l) \sum_{j=0}^{n_{l+1}-1} W_{ij}^{l+1}\delta_j^{l+1} \text{ if logistic function is used}$$

$$\delta_i^l = (1 - (O_i^l)^2) \sum_{j=0}^{n_{l+1}-1} W_{ij}^{l+1}\delta_j^{l+1}\text{if tanh function is used}$$

The δ values are held in *error* field of each neuron.

5.6 Momentum learning

If you want to train the network with on line method, for each learning couple, you have to call method **train ()** that updates the weights according to rule $(W_{ij}^l)' = W_{ij}^l + \Delta W_{ij}^l$ where

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

and η is the network *learning rate*.

For implementing momentum backpropagation we just save previous step's ΔW_{ij}^l (in *delta* array of each neuron) and define

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

where $\Delta W_{ij}^l(p-1)$ is the previous step delta and $\Delta W_{ij}^l(p)$ is the new delta. If $\alpha = 0$ ($momentum = 0$) we have backpropagation without momentum.

Momentum is used for reducing the value of correction when the gradient at one step has opposite direction to previous step gradient (this situation happens when we have floating movements around a minimum of the energy), and for amplifying the correction when it has the same direction in the two steps (and so increasing the speed in the plain zones of the error surface).

This is the code portion which updates the weights: values of δ_i^l should be previously computed and held in the *layer[l].neuron[i].error* field.

Weights updating for on line training

```
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 Batch training

Batch training tries to reduce variations in weight corrections, making the mean of all the correction made by different patterns.

Given a set of N input / output couples we compute the δ's as defined in section 5.5 and we compute the average value. This mean is used to compute the corrections on weights.

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

In our implementation at the beginning of a new epoch, you have to call method **begin_batch ()**, which sets *no_of_patterns* to 0 and puts to zero all fields *sumdeltas* in all network neurons.

For each pattern of the batch you have, then, to call method **train_batch ()** (as you did with **train ()**, i.e. after calling **compute ()** and **compute_output_error ()**), that increases of one the field *no_of_patterns* and sums to *sumdeltas*, the current value of $\delta_i^{l(p)} O_j^{l(p)}$ (weights are not changed).

At the end of the epoch you have to call **end_batch ()**, which adjusts the weights according to the average variation for all patterns, computed by dividing *sumdeltas* by *no_of_patterns*.

It is possible to combine batch training with momentum method, by using as previous step's ΔW_{ij}^{l} the value of the previous epoch correction, which is saved in vector *delta*.

Some authors define batch training in a different way, i.e. without dividing by N the sum of weight corrections.

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

However expression 8 is equivalent to 7 if we choose in (7) a learning rate $\eta' = N\eta$.

In some experiments it turned to be appropriate to set learning rate to a higher value in order to train the network in batch mode. But we leave the user the responsibility to set properly the learning rate. Perhaps you don't have to mind this problem if you use batch training with Super SAB, which adapts the learning rates.

5.8 Super SAB (Self Adapting Backpropagation) training

Super SAB (Self Adapting Backpropagation) training makes each weight move to the minimum with a different speed. Each weight has its own learning rate, η_{ij}. These learning rates are initialized to the default value and are changed during the training. Let's define a factor $\eta_{up} > 1$ and a factor $\eta_{down} < 1$, at every step[1] learning rates are updated this way:

- If weight update keeps the same sign of previous step, learning rate is multiplied by η_{up}.
- If weight update changes its sign, learning rate is multiplied by η_{down}.

Super SAB mode is turned on by calling **begin_ssab ()**, which allocates a learning rates vector whose length is equal to the total number of weights, initialized to current learning rate value.

Super SAB training can be made by calling method **train_ssab ()**, which modifies the weights by using the learning rates vector and modifying learning rates according to the rule.

But it is more efficient[2] to use Super SAB method in combination with batch training. For using Super SAB mode in batch training you should call

[1] really at every epoch if you want SuperSAB to work...

[2] This is an euphemism: in our experiments using Super SAB technique updating the weights for each pattern led to very poor results: error increased instead of decreasing. For this reason the trainer class has no train_ssab_online() method

once **begin_ssab()**, then **begin_batch()** at the beginning of every epoch, train the network with **train_batch()** and then call **end_batch_ssab()** at the end of every epoch. The last method updates the weights according to the average variation computed on the patterns, using learning rates kept in the vector and changing the learning rates according to Super SAB rule.

The library allows you to set the parameters η_{up} and η_{down} and to define a maximum and a minimum value for the learning rates. Learning rates values can be saved on a binary fire and then reloaded to keep on training.

5.9 Network saving on file

A network can be saved on a file, and then reloaded and used again. In the binary file we save the activation function (coded as 0 or 1), the number of levels, the number of neurons in each level, the momentum, the learning rate, the global error and all the weights. Of course the values held in *delta* and *sumdeltas* arrays, which are only needed for training, are not saved on the file, while the parameters for Super SAB training can be saved on a separate file.

It is possible to load and save a network in textual format. This format is not usually suitable because it uses more disk space, but it can be useful for importing networks created with *Lightweight Neural Network* C-library (which originally supported only the textual format) or to export the networks on computers with different architectures using different coding for floating point numbers.

6 The trainer class

6.1 Structure

With the class **trainer** the user is allowed to train a network starting from patterns saved on file, with the different methods provided by the library. We distinct three patterns set: training set, validation set and certification set. Training set is used for training, validation set is used for verifying, during the training, that the network is properly generalizing, while certification set is used at the end of training for verifying once again the results. Each one of these three sets can be loaded from a file.

For making the learning safe and avoid training a network on patterns which have not the right dimension of input and output layers, class **trainer** holds the number of inputs and outputs which are set when the object is constructed and cannot be changed after. The network used by the trainer and every pattern set must have the same input and output dimension.

Reading from file is managed by an object of a derived class of **iomanage**, in order to allow maximum flexibility in file format.

Furthermore, class **trainer** can be specialized for customizing its features, as it is explained in section 6.9. Thanks to this ability to be customized, this class is a little *framework* for the programmer who wants to write a program for training a neural network.

6.2 Class constructors

A **trainer** object can be constructed in three different ways.

All three types of constructors have two common last parameters, that specify the names of log files where error and accuracy trends are saved (accuracy is defined in section 6.7). If the names are not given, no log file is written.

The first constructor takes a network object which sets the number of inputs and outputs.

With the second type of constructor you have to give the number of inputs, of outputs and the number of neurons in the hidden layer and the trainer will construct a new 3 layers network with the dimensions passed to the constructor.

The third constructor takes the name of a file containing the training patterns to be read, the number of neurons in hidden layer and constructs a new 3 layers network with dimensions equal to pattern file. It is possible to pass a **iomanage** object if you want to use a non standard format for pattern files.

6.3 Loading training, validation and certification sets

Class **trainer** has three methods to load from file the training, validation and certification set patterns. These methods are called **load_training**, **load_validation** and **load_certification**. The patterns are loaded from the file and are saved in three arrays of pointers to float arrays and presented to the network during the training phase. The format of these files depends on the object of class **iomanage** set in the trainer, which can be changed with method **set_iomanager**.

6.4 Formats for patterns files

6.4.1 Textual formats

Class **iomanagelwnnfann** is used by default in the trainer and supports two file formats: format *lwnn* and *fann* format. The class guesses the format of the file and reads properly a file without having to know its type.

"lwnn" format

This type of files supports comments, ignoring lines beginning with #. Comments are not allowed between a input line and a target line. The first (not comment) line is the number of pairs in the file. The second line is the number of inputs, the third one the number of outputs. Then there should be $2 * no\_of\_pairs$ lines containing inputs and matching target. Numbers are delimited by spaces

"fann" format

This is the format used by fann (Fast Artificial Neural Network) library (see `http://fann.sourceforge.net/`) by Steffen Nissen. The code for reading is borrowed and adapted from that library.

No comments are allowed. The first line should contains three integers being the number of pairs, the number of inputs and outputs Then there are no_of_pairs lines containing inputs and no_of_pairs lines containing outputs.

6.4.2 Binary format

iomanagebinary class is derived from **iomanage** that manages the patterns in binary format.

The file should contain, in binary format, the number of patterns, the number of input and output. Input and target vectors must follow.

Class provides a method **write_patterns** and a **convert** method that allows converting patterns saved in any format (i.e. with any implementation of **iomanage**) to binary format. This way binary format can be used as a standard format and data saved in any format can be converted in binary format and used in a program using binary format.

6.5 Network training

Class **trainer** allows you to train the network with the techniques implemented in **network** class. Methods for network training are: **train_-online**, for on line training (weights are updated every time a pattern is presented), **train_batch** for batch training, **train_ssab** for Super SAB training, **train_shuffle** for a version of on line training which presents patterns in a different random order each epoch. A momentum is used in each technique<, if it is set in the trained network. These methods take the boolean parameter `verbose` that allows to print on screen the error and accuracy trend.

This is the code of **train_online** method.

train_online(bool verbose) code

```
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;
```

}

As we can see, for each loop of the `while` the whole training set is presented to the network, and we compute the square mean error on training patterns. Stopping condition is implemented in **continue_training** method and depends on many parameters. Stopping condition is the most critical point in training process. In fact the objective of training is to create networks which give good results not only on training set but that are able to generalize as well.

A first stopping condition is a limit on maximum number of epochs. Method **set_max_epochs** allows to set a high boundary to number of training epochs. This condition is important because it guarantees that the training always stops, because it's not easy to foresee what will be the minimum reachable error.

If you don't want to use a validation set you can use as a stopping condition the reaching of a certain error on training set. This way of training exposes to risk of over fitting, described in next section.

A possible way of using validation is to stop training when an error is reached on validation set. To do this way, you have to call **set_training_validation** which makes the wanted error be computed on validation set and not on training set. For efficiency reasons error is not computed at every step but every $n_{checking}$ epochs. This parameter is set with method **set_epochs_checking_error**.

This training method is useful if you want the network to have a fixed precision on validation set (and so, on all patterns, if validation set is properly chosen). But it is not easy to choose this parameter. If you choose a too small value it can happen that this value is never reached on validation set. On the other hand a greater and easily reachable value could not be the optimal value for that network. The risk is to overtrain or undertrain the network.

Class **trainer** gives another stopping condition: once accuracy is defined (as it is explained in section 6.7) you can stop training when a fixed accuracy is reached on *validation set*. This parameter is set by the method **set_wanted_accuracy**. This technique has the same problems than stopping on a fixed error on validation set.

When training stops you can call **get_stopping_cause()** which returns a code identifying the reason why last training session stopped.

6.6 Avoiding over-fitting

The idea to keep on training until error on training set goes under a fixed value is very dangerous, because backpropagation is likely to reproduce in a better and better way the training patterns, at the cost of diminishing the ability to generalize to patterns never seen during the training.

In fact we can see that the error on validation set in a first period decreases, and then increases, while error on training set keeps on decreasing. How can it be possible? The answer is that the weights of the network get tuned for reproducing the peculiarities of the examples used for training, which are not representative of the general patterns. But why does overfit take place in the advanced phases of learning and not in the earlier ones? Since weights are initialized with small random values, in the first iterations the decision surface (that is the shape of the fitting function) is rather simple, because the activation function is almost linear. When training goes on, some weights begin to increase their value for reducing the error on training patterns and the shape of decision surface becomes more complex. In a long enough training period the network will be often able to produce very good fitting of the points on which it has been trained but will give poor results for the points not included in the training set.

One of the most successful methods for avoiding overfitting, explained in the book by Tom Mitchell [1], is to use a validation set besides the training set. The algorithm checks the error on validation set and uses the training set for computing gradient descent. The optimal number of batches will be the one giving the minimum error *on validation set*, because this last one is the best test of the behavior of the network on never seen examples.

In our implementation we keep two copies of the network. The first one is used for training, while we save the network that, by that moment, gave the best error on validation set. When for a some number of batches the error on validation set does not decrease, becoming rather greater than error on training set, training session ends and it's possible to retrieve the network which produced the best error on validation set.

In figure 3 you can see that error on validation set begins to increase. The best network found by the stop on overfit algorithm is the one at epoch 273, corresponding to best error on validation set.

Class **trainer** allows you to set two parameters for training with overfitting checking: training sessions stops if a fixed number of epochs has passed without finding a better error on validation set (first parameter) AND if the ratio between error on validation set and error on training set is greater than a fixed value (second parameter). These parameters have to be chosen properly in order to avoid stopping the first time the error on validation set

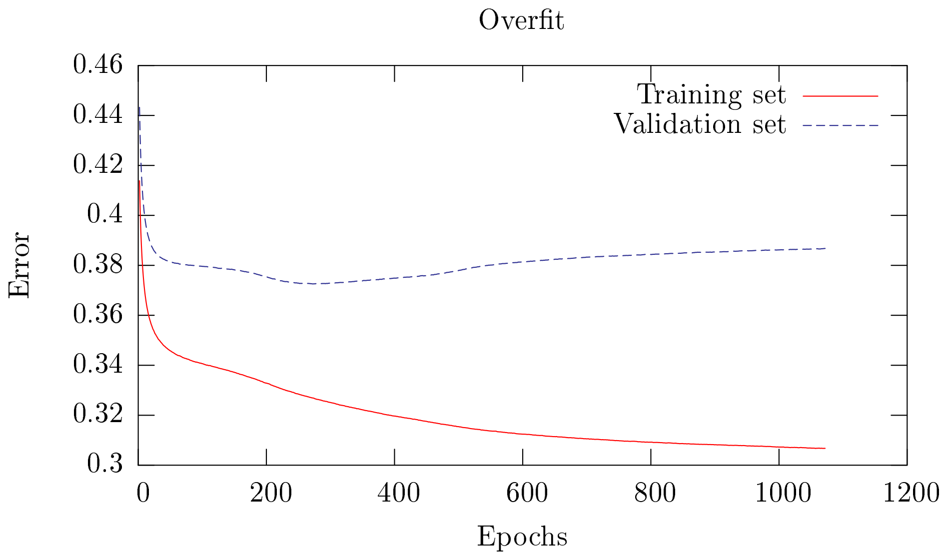

Figure 3: *Example of overfit. Best network was reached at epoch 273.*

increases; in fact it is common for error on validation set to increase and then to decrease once again, as in figure 4. So you must be careful not to stop the algorithm when error on validation set starts increasing: doing that way we would have come to the conclusion the best network is the one found around epoch 50, and not the one found at epoch 407.

You must also be aware that, even when you set the condition for stopping on overfit, the *min_error* parameter keeps on being meaningful and training session will stop if the best error on validation set is lesser than the value of minimum error. So, if you want to avoid that stopping condition, you should set a very small (or even 0) minimum error by the method **set_min_error**. At the end of training, you must call the method **set_network_best()** and then **get_network()** for retrieving the network with minimum error on validation set.

6.7 Definition of accuracy

In many applications, neural networks are used to classify patterns. And so it turns to be important, instead of square mean error, the ability of the network to properly classify. What "classify" means depends on the type of the problem and on the meaning given to the output vector. Nevertheless two very common situations arise very often, and for these ones we implemented the computing of accuracy inside the class.

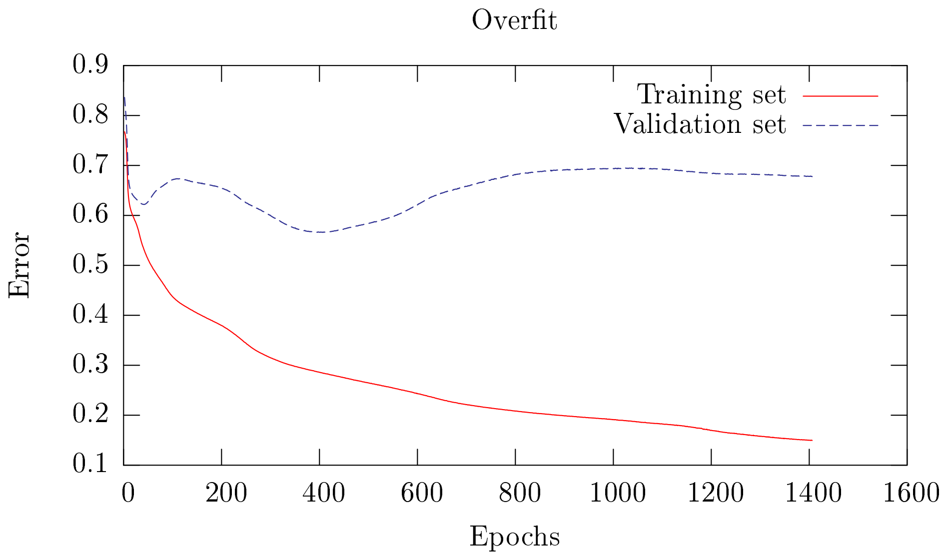

Figure 4: *In this example error on validation set increases, then decreases and increases once again. Best network is reached at epoch 407 (and not the first time that validation set increases!)*

Binary Accuracy Let's suppose our network must output a binary number of length n. Output is not binary so we need to round, and we take equal to 0 all the output neurons whose value is lesser than 0.5 and equal to 1 all the output neurons whose value is greater than 0.5. Given a set of input/output couples of dimension N, to compute accuracy we compute the output for each one of these N inputs and count the number of right answers. An answer is said to be right if, calling T_i the i-th element of target vector and O_i the i-th element of network output, the following condition holds:

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

Accuracy is given by the number of right answers divided by the total number of patterns, N.

In our implementation we really use a different definition of accuracy according to which the output can be not-exactly a binary output (for example we could use values 0.1 and 0.9 instead of 0 and 1 in binary coding). According to this definition an answer is said to be right if

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

Max Guess Accuracy A different definition of accuracy is useful when the classes for classifying inputs are not coded as binary numbers but with a 1 in the wanted position (for example four classes coded as $1000, 0100, 0010, 0001$). This coding for the output is often used in this field because it usually needs a lesser error for obtaining right answers.

In fact for properly classifying a pattern, it is sufficient that the positions of maximum value neuron is the right one. This considerations lead to the following definition of right answer

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

Accuracy is still defined as the number of right answers divided by the total number of patterns.

Class **trainer** allows you to compute accuracy according one of these two definitions. Method **set_accuracy_mode** allows to choose if accuracy should not be computed (because meaningless) or if it should be computed according to one or the other of the standard definitions.

It is possible as well to write a derived trainer class, as we show in section 6.9, redefining the method which says if an answer is right or not, and so customizing the definition of accuracy.

6.8 Log files

When you construct a **trainer** you can set the names of two log files. The first one will hold the error trend, the second one the accuracy trend. These files are written each epoch if you don't use the validation set, only when validation is done otherwise. In this last case log files will contain in each row three numbers divided by the tab char that are the epoch, the error (or accuracy) on training set and the error (or accuracy) on validation set.

Accuracy log is written only if the *accuracy mode* is not set to 0 (*NO_ACCURACY*)

6.9 Class specialization for the trainer

It is possible to change the behavior of **trainer** class, by writing a derived class of trainer and by rewriting some methods.

Three methods are thought to be specialized. These methods are declared **protected** (they cannot be called on a **trainer** object but can be rewritten and called in a derived class) and **virtual** to make polymorphism work.

The first method that can be easily rewritten is **check**. This method takes a network output and the wanted target and returns a boolean value saying if the answer has to be considered right. This way you can give a different definition of accuracy, which reflects the meaning of output vector. For example in a network using tanh as activation function you could use -1 and 1 instead of 0 and 1 for coding binary output, or you could define a answer to be right if the norm of the difference between output and target is lesser than a fixed value.

In order to give maximum flexibility we let the user rewrite method **continue_training**. This method implements the stopping condition for the training and is the most critical point in network training. Class **trainer** allows to set a lot of parameters for changing the stopping condition, so rewriting this method should not be necessary. If you want to do this you should be aware that some parameters of the class could change their meaning. It will be a responsibility for the programmer of the extended class to make its interface be clean and coherent.

Two other methods can be specialized: **print_input** and **print_output** for defining a different way of printing input and output vectors according to their meaning in a particular application.

7 *lwneuralnetplus* package and *klwnn++* GUI

7.1 How to install and use the library

The package contains the library, some utilities for training a network, a simple gui and some examples. It can be downloaded at **http://lwneuralnetplus.sourceforge.net**.

The package is distributed as source code. The library was developed on a *GNU / Linux* system with *gcc* compiler, version $\geq$ 2.95.4, but it should not be difficult to port them to other operating systems or compilers.

Type make to compile the library.

If you're connected to Internet make will also automatically download patterns for faces recognition example.

Choose the root installation directory for lwneuralnet++. Default is /usr (lib will be in /usr/lib, binaries in /usr/bin, documentation in /usr/doc/lwnnplus-VERSION, tests and examples in /usr/share/lwnnplus-VERSION), edit Makefile in this directory to change root installation dir (change value of BASE_DIR)

Type su to become root and give the `make install` command

If you want the documentation you must have doxygen available on your system.

After installing the library *liblwnnplus.so* will be available on your system and you will be able to link it in your application.

The following programs are available after installing:

lwnn_train Train a neural network, by using **trainer** class. There are a lot of options for setting network and trainer parameters (use the graphical interface if you don't want to remember all the options...)

lwnn_validate It takes two parameters: the first one is the pattern file, the second one a network file and uses the network to compute outputs and computes the error and the accuracy of the network.

lwnn_printnet Prints the parameters of a network saved on a file.

lwnn_infospec Prints information of input/output couples saved on a file (number of patterns, number of inputs and outputs)

It will be available the command **lwnn_visualtrainer** that launches the interface "Visual Network Trainer" described in the following section.

7.2 Graphical user interfaces

To make easy to use our library, we developed two graphical user interfaces.

The first one is distributed with the library, is written in tcl/tk and is very lightweight. The second one uses the QT libraries of KDE, is separately distributed and is easier to use.

7.2.1 Visual Network Trainer

This user interface is included in the package and is written in scripting language *tcl/tk*.

Visual Network trainer is a simple form to set the parameters of textual commands. It is also possible to draw, starting from log files, graphs of error and accuracy with **gnuplot**.

You can choose a training set and, if you want, a validation set. These files must be in lwnn (or fann) format or in binary format (if the file has the .bin extension).

Network parameters can be set (number of neurons in hidden layers, learning rate, momentum), as well as the stopping conditions (max number of epochs, wanted error, wanted accuracy, stop on overfit) and the type of accuracy. It is possible to create a new network or to continue training a trained network.

The interface is shown in figure 5.

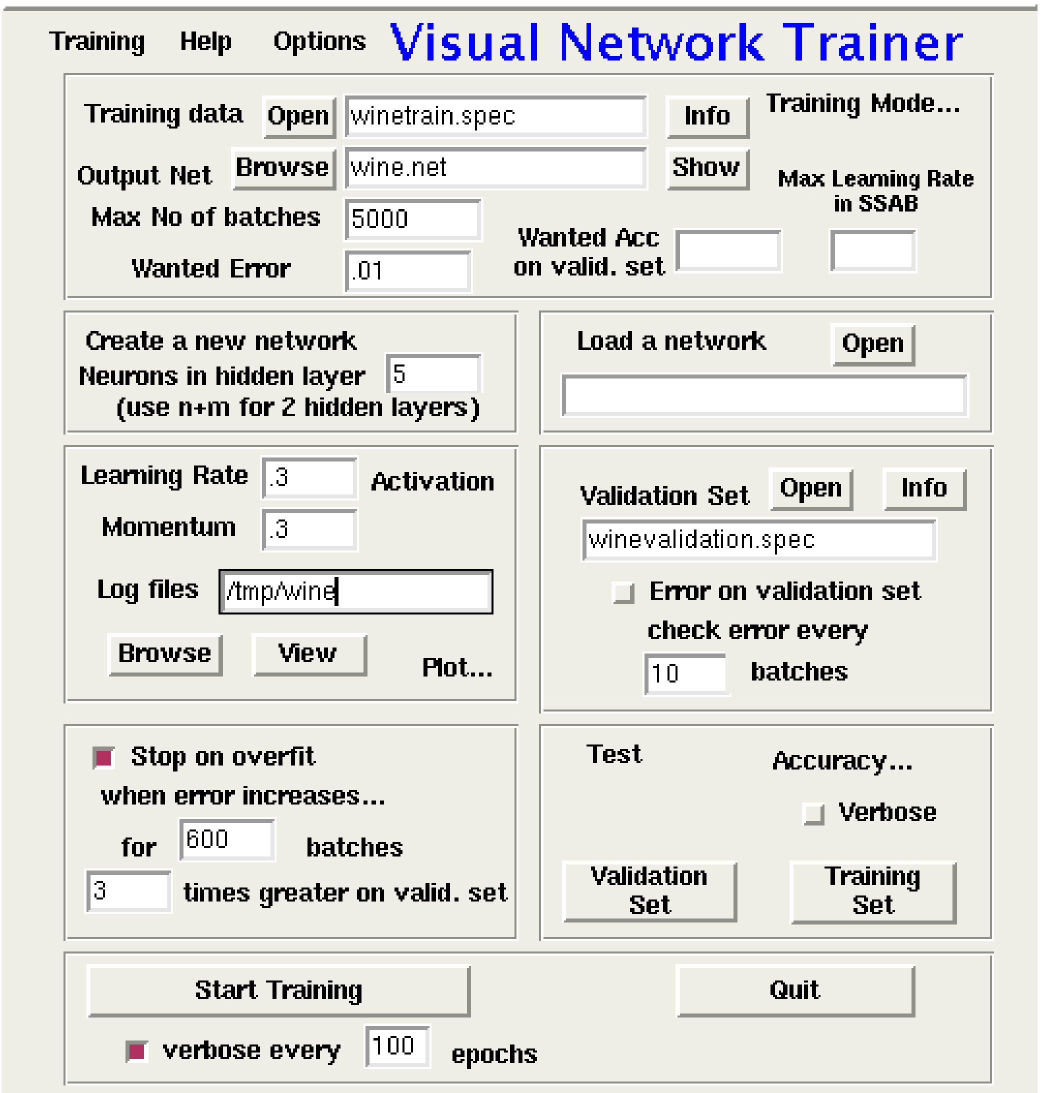

Figure 5: Visual Network Trainer *interface*

7.2.2 KLWNN++

KLWNN++ is an interface for our library using QT libraries. This interface is shown in fig. 6. The interface has four tabs. The first one, shown in fig. 6 is for setting the training parameters, that are

- Files of training, validation and certification patterns.
- Learning mode (On line, batch training or Super SAB)
- Max number of epochs
- Wanted error
- Stop on overfit conditions

The second tab allows you to set the network parameters, that are

- Number of neurons in the hidden layers (max. 3 hidden layers)
- Momentum
- Learning rate
- Sigmoidal function (Logistic or Tanh)
- Accuracy mode (Binary round or Max Guess)

From this tab the user can save the network trained so far, load a network from a file, or restart learning with a new network, as is shown in fig. 7.

The user can begin training pushing "Train" button and can check the error and accuracy trend in the last tabs of the interface. At the end of the training the user will be told the stopping cause of last learning session (max number of epochs reaches, wanted error reached, overfit, etc.)

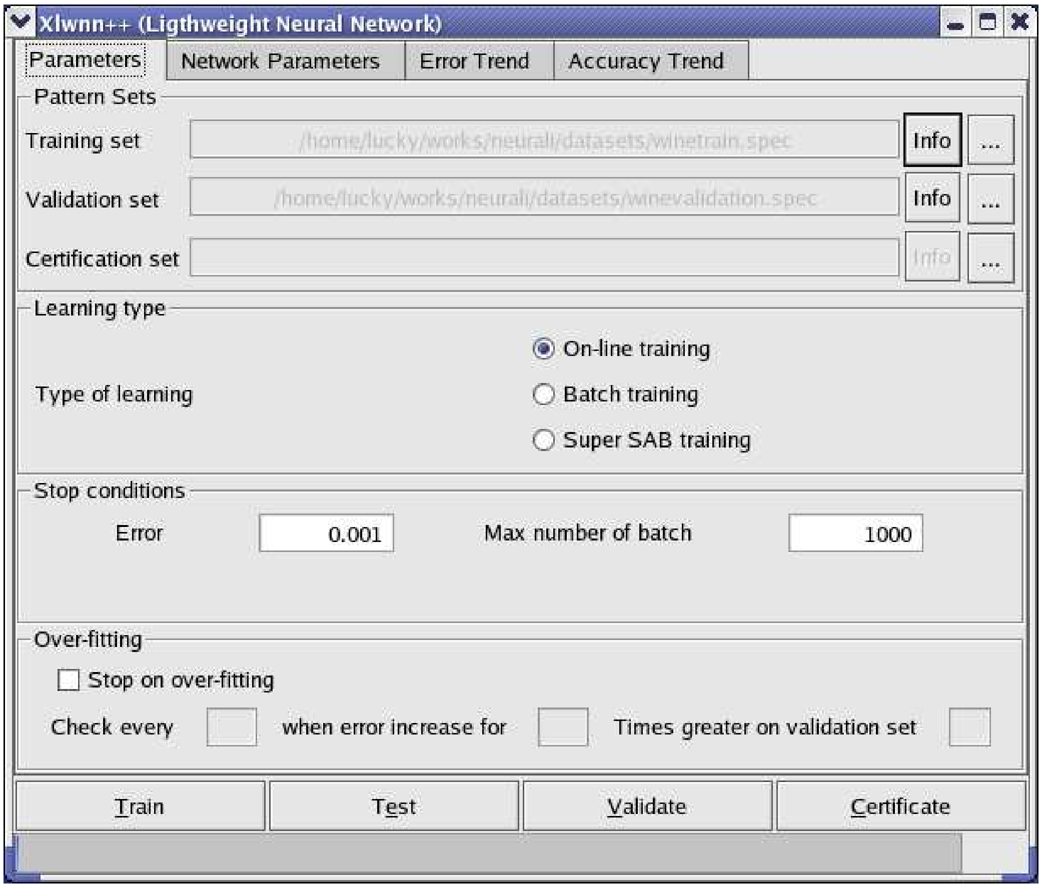

Figure 6: *KLWNN++: training parameters*

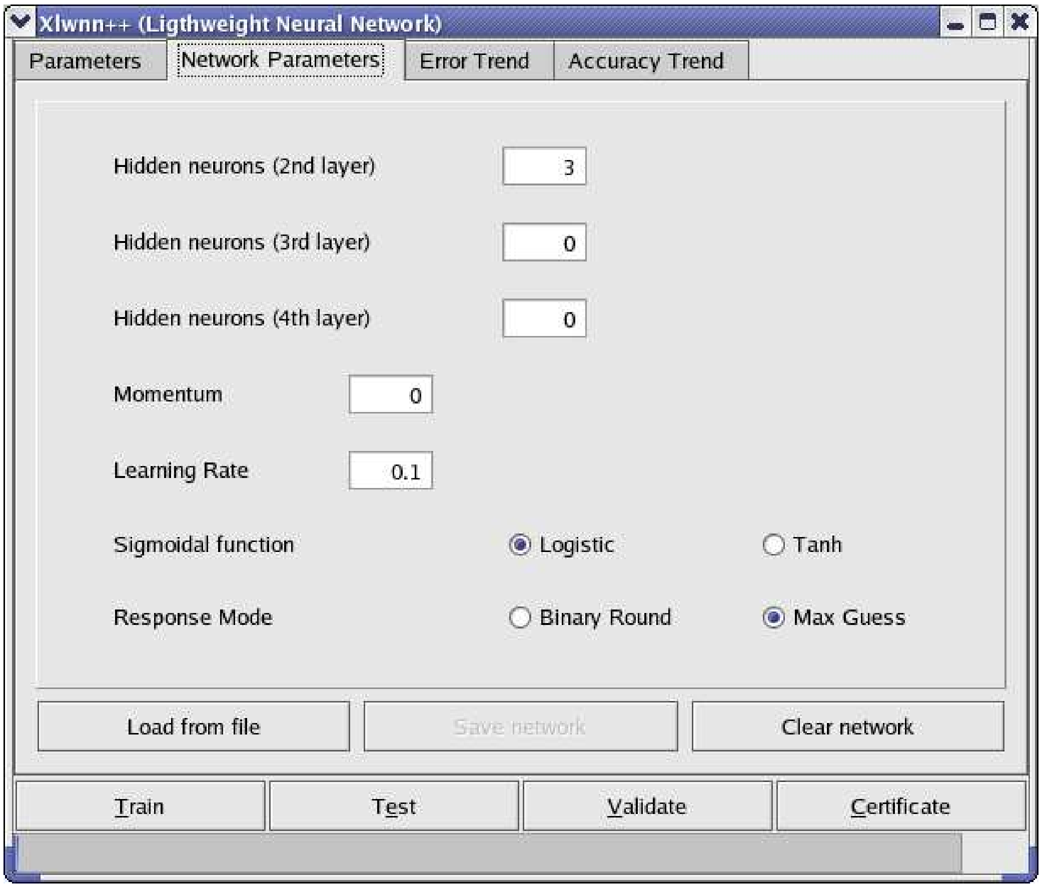

Figure 7: *KLWNN++: network parameters*

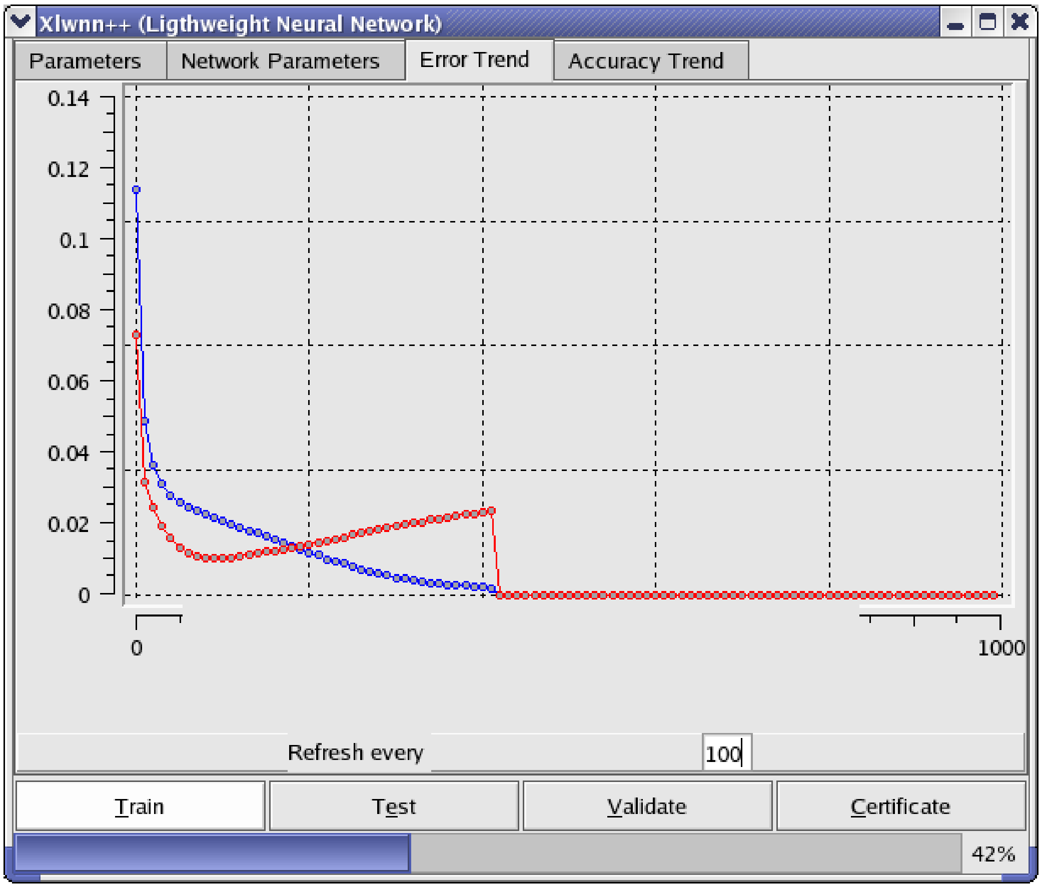

Figure 8: *KLWNN++: Error trend*

8 Anti-spam network

Antispam (junk) filters that can be found on the Internet, or given with some email clients (e.g. Mozilla Mail) try to recognize if an email has to be considered spam, mainly in two different ways. The first one is to connect to a database of addresses known as spam sources, while the other one focuses on spam emails that the system has already seen.

Neural networks can be used for implementing this second type of anti-spam.

8.1 Spam patterns

We found a set of spam patterns on the site of *UCI Repository of machine learning databases* ([2]). This set was built by taking some spam emails and some non spam emails. The set is made out of 4601 emails, of which 1813 were jumk emails. Every pattern is described by 58 parameters.

The first 48 are float numbers between 0 and 100 that are the frequency of some predefined words:

> make, address, all, 3d, our, over, remove, internet, order, mail, receive, will, people, report, addresses, free, business, email, you, credit, your, font, 000, money, hp, hpl, george, 650, lab, labs, telnet, 857, data, 415, 85, technology, 1999, parts, pm, direct, cs, meeting, original, project, re, edu, table, conference

Following 6 parameters give the frequency of some characters:

> ;, (, [, !, $, #

Other three parameters are constructed looking at the sequences of capital letters. The first one gives the average length of the sequences of capital letters, the second one gives the length of the longer sequence, while the third is the total length of all the sequences of capital letters.

The network is trained to output 0 or 1 (non spam or spam), so it has just one neuron in the output layer.

8.2 Training, validation and certification sets

The whole set of patterns was divided in three sets. The training set has 2800 patterns, the validation 920 patterns and certification set 880 patterns.

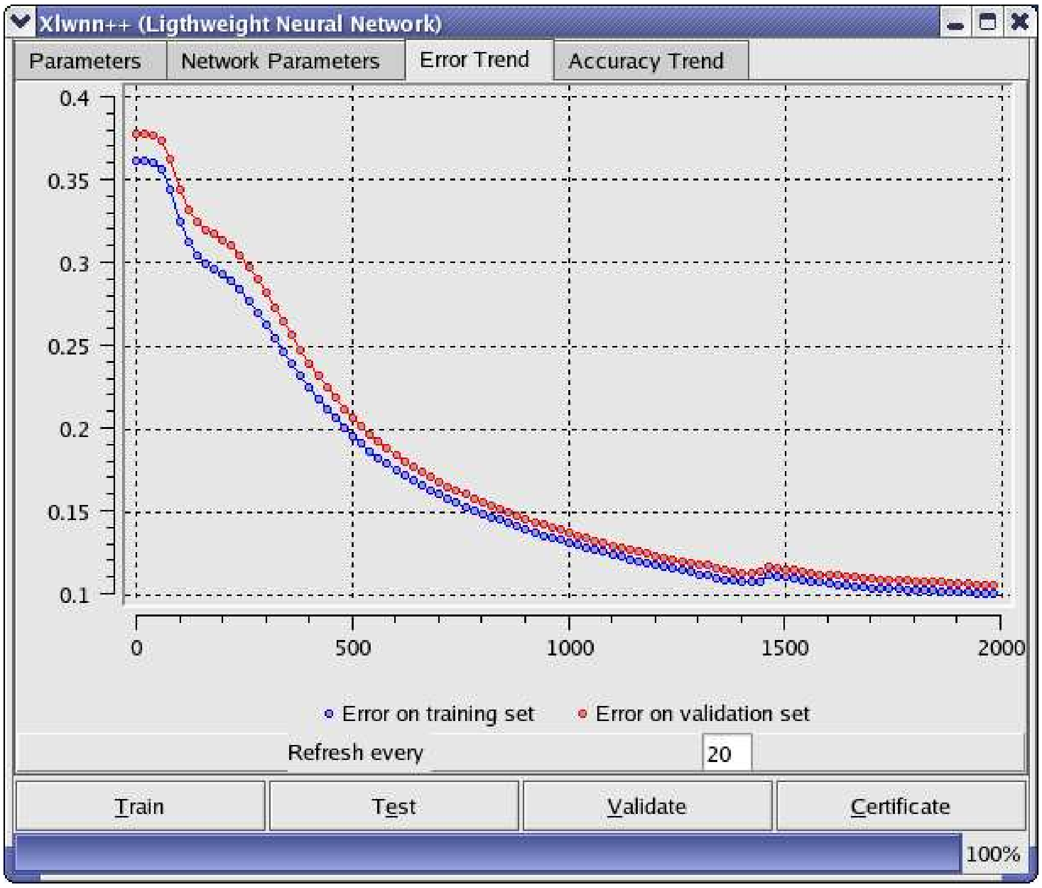

Figure 9: *Error trend for a network with 10 neurons in the hidden layer*

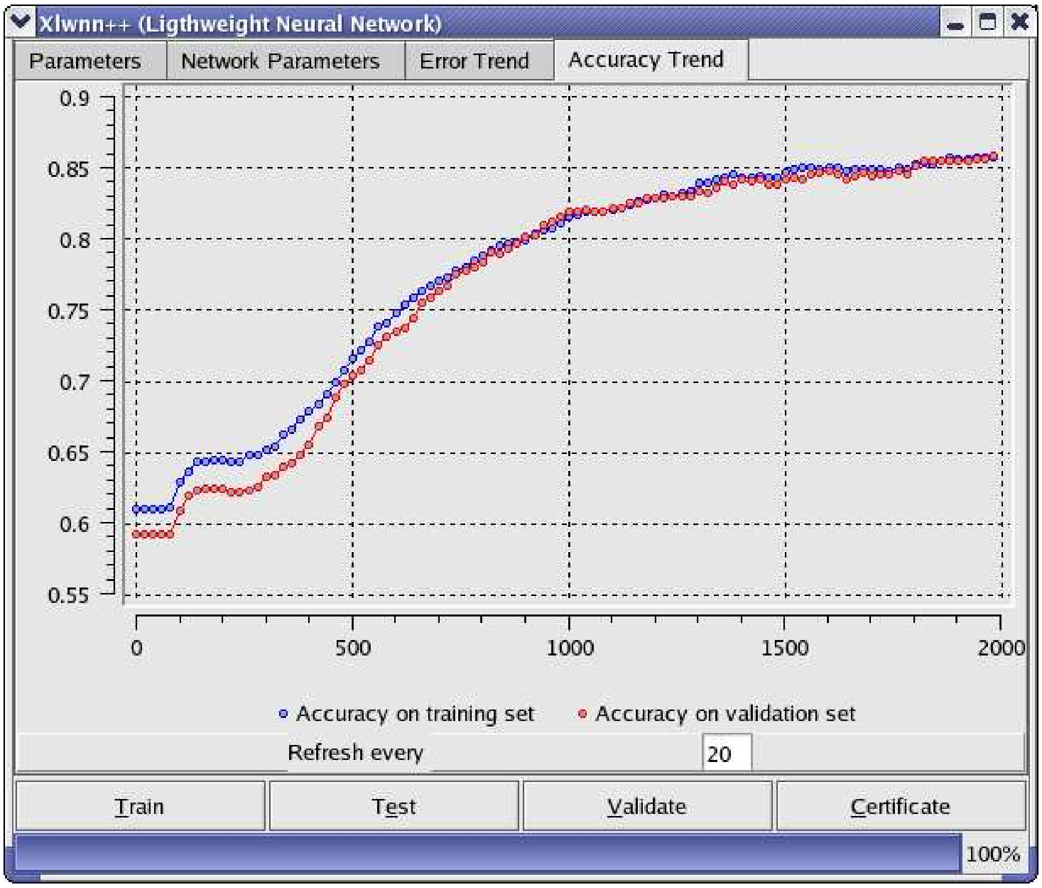

Figure 10: *Accuracy trend for a network with 10 neurons in the hidden layer*

8.3 Network training

In fig. 9 we can see the error trend on training and validation set in the training of a network with a single hidden layer of 10 neurons for 2000 epochs. In fig. 10 the accuracy trend is shown. Accuracy is computed according to the "Binary" definition (Max Guess does not make sense, because there's just one output neuron)

In figures we can see that the accuracy reaches 85%. On certification set we reached an accuracy of 85.44%.

In fig. 11 it is shown the error trend with Super SAB training. Results are better than on line training, as we can see in the accuracy trend. Accuracy on certification set is as well better, going up to more than 93%.

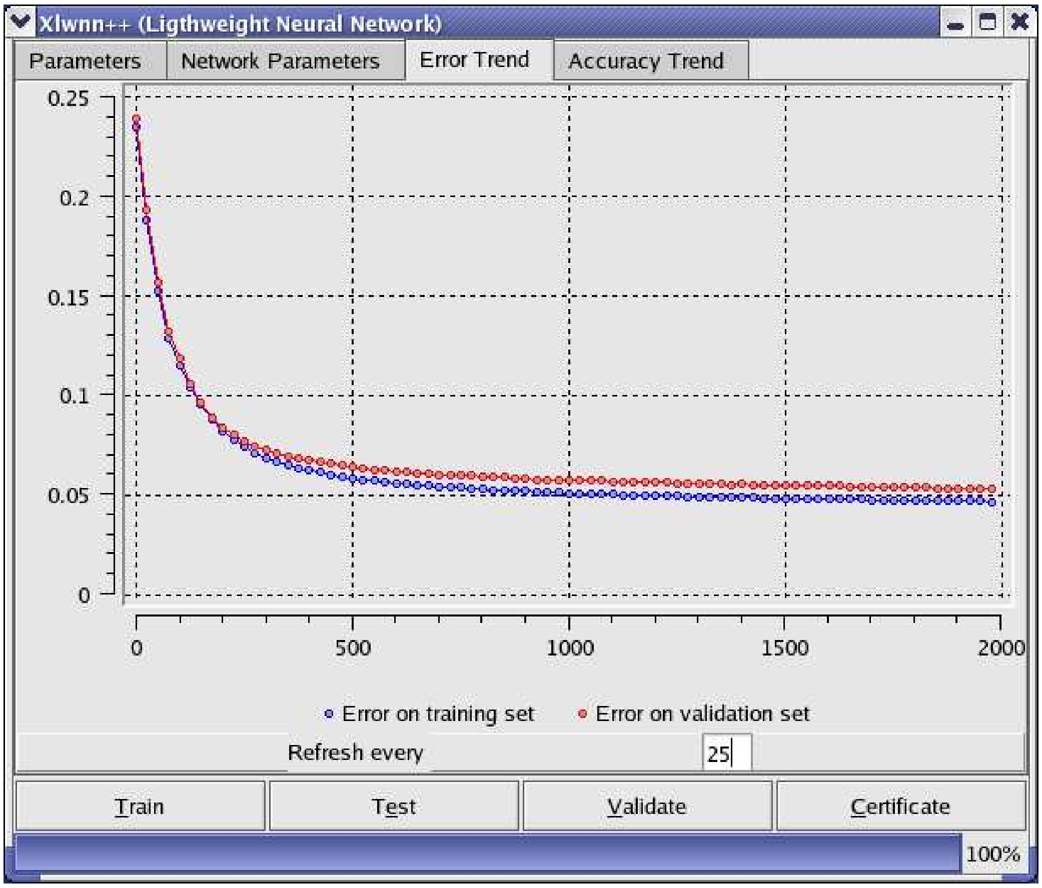

Figure 11: *Error trend for a network with 10 neurons in the hidden layer with SuperSAB training*

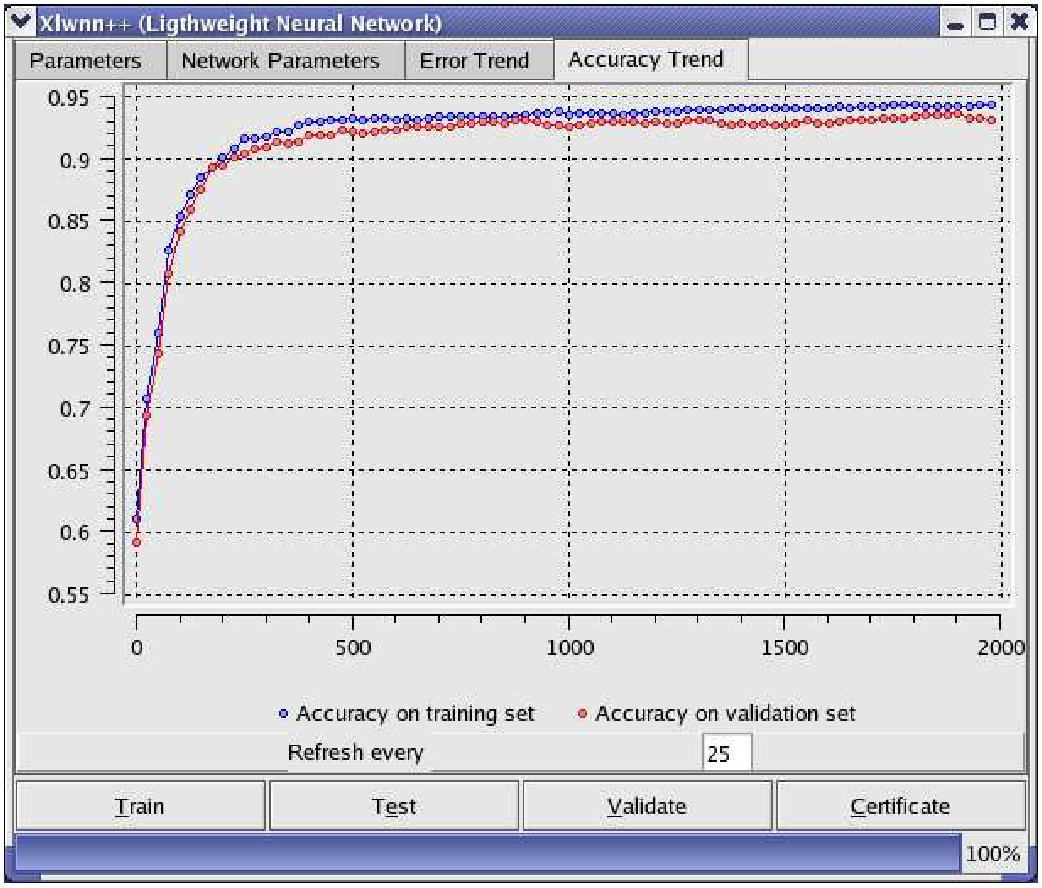

Figure 12: *Accuracy trend for a network with 10 neurons in the hidden layer with SuperSAB training*

9 Faces recognition

This example is taken from the book *Machine Learning* by Tom Mitchell ([1]).

The homework description ([6]), the image data and the code for reading images can be downloaded at the site
`http://www-2.cs.cmu.edu/afs/cs.cmu.edu/user/mitchell/ftp/faces.html`

We rearranged the code for including it in our example.

The neural network of this example is used for classifying face images of several persons taken in different positions. Photos of 20 different persons were taken, with up to 32 variations for each person, varying the facial expression (neutral, happy, sad and angry), the head position (straight, left, right, up), and if he/she wears sunglasses.

Data available on Internet consists of 604 images whose resolution is 30×32 pixels, each one described by a gray scale intensity varying from 0 to 255. Original images had a dimension of 120×128 pixels, but they were rescaled in order to reduce input layer size. As we can see in figure 13 there are some variations in the images: the background, the clothes worn, face position into the image.

Starting with this set of images we can build several different targets, for example we could train the network to recognize the direction where the person is facing, or if is he/she's wearing sunglasses.

In our example we chose to train the net to recognize the identity of the person. The input vector has length $30 \times 32 = 960$ and holds the pixels of the image rescaled in interval $[0; 1]$.

The output vector is 20 neurons long, in order to classify input vectors in 20 classes corresponding to the 20 persons. Each class is coded putting to 0.9 the neuron corresponding to the right answer, and to 0.1 the other neurons. The choice of these values, instead of 0 and 1, is made because this way the weights don't need to get to very high values for taking the function in a saturation zone.

Accuracy is computed according to the *Max Guess* definition, i.e. the answer depends on the neuron of maximum value.

In our experiments we used a learning rate $\eta = 0.3$ and a momentum $\alpha = 0.3$.

The whole set of patterns was divided in three sets. The *training set* holds half of the images of each person, the *validation set* a little more than a quarter, and the *certification set* less than one quarter.

Of the 604 total images, 315 were used for training, 176 for validation and 113 for certification.

In figures 14 and 15 we draw the error and accuracy trend during network training with 10 neurons in the hidden layer. We can notice that in this example no over-fitting happens, during the first epochs. However after 200 epochs of training accuracy reaches 100% on both training and validation set while error, after 400 epochs reaches 0.032 on training set and 0.048 on validation set. For certification set we have an error of 0.051 and again accuracy is 100%.

Very good results were obtained with 10 neurons, so we tried to train a network with only 5 neurons in the hidden layer. Results are shown in figures 16 and 17. This time there is over fit and we reached the best network at epoch 722, reaching an accuracy of 92.6% on training set and 90.9% on validation set. On certification set we reach an error near to the one on validation set and an accuracy of 92.0%.

Figure 13: *Images of dimension* 30×32 *for one person*

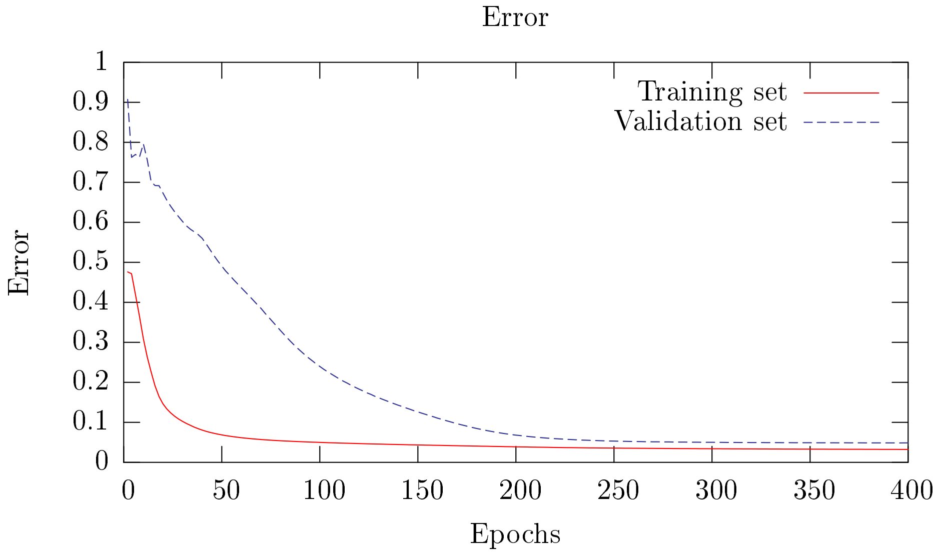

Figure 14: *Error trend in training a network for faces recognition with 10 neurons in the hidden layer*

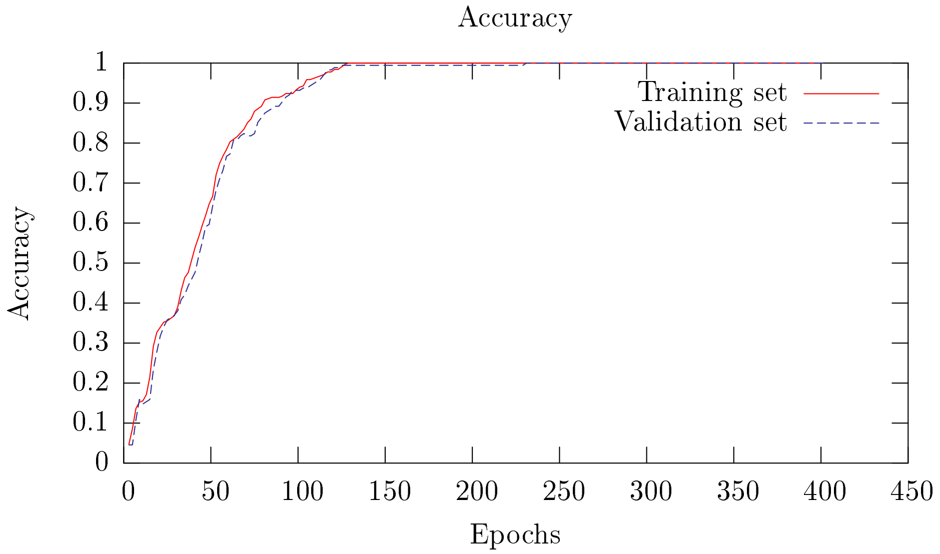

Figure 15: *Accuracy trend in training a network for faces recognition with 10 neurons in the hidden layer.*

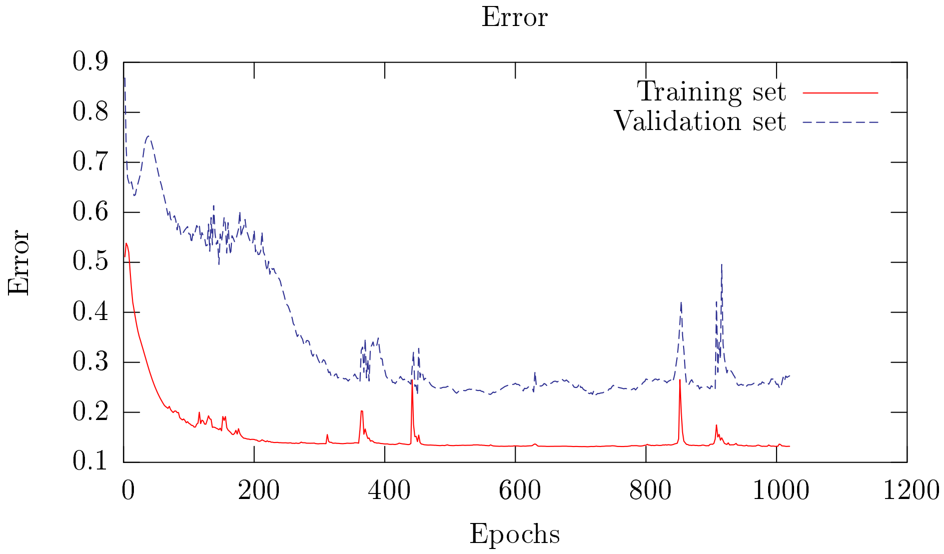

Figure 16: *Error trend in training a network for faces recognition with 5 neurons in the hidden layer*

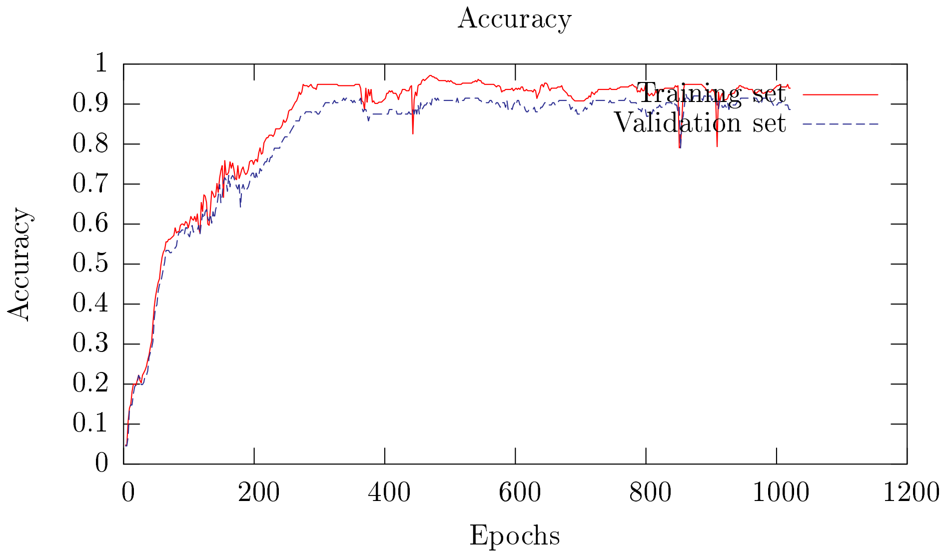

Figure 17: *Accuracy trend in training a network for faces recognition with 5 neurons in the hidden layer*

10 OCR: handwritten digits recognition

In this application we trained a network to read some handwritten digits. The set of examples were downloaded from the *UCI repository of machine learning databases* ([2]). The patterns set are made of black & white images of dimension 32×32 pixels. Training set contains 1934 patterns, validation set 946 patterns. These images were rescaled to patterns of dimension 16×16 in gray scale, making every pixel of the reduced images correspond to 4 pixels (a 2×2 square) of original image (obtained by summing values of the four pixels, and by normalizing the values on the whole image)[3].

As we can see in image 18, digits are quite variable (were written by different persons), in the position as well as in the dimension and handwriting.

Neural network has 256 neurons in the input layer which correspond to the image pixels, each one has a value between 0 and 1, 14 neurons in the hidden layer and 10 neurons in the output layer coding the recognized digit with a vector of 0's in all positions except the one corresponding to the right digit, which is 1 during the training. Classifying happens with the technique of *max guess accuracy*, i.e. checking the neuron with the higher output.

Network was trained using the on line technique, with learning rate 0.6 and momentum 0.1, but presenting in every batch the patterns in a random order. In figures 19 and 20 we show the error and accuracy trends. Best network is reached after 29 epochs and makes 4 mistakes out of 1934 on the training set (accuracy of 99.79%) and 20 errors out of 946 on validation set (accuracy 97.89%).

[3]The code for managing the images (actually saved in a simple textual format) is adapted from an example by Peter Van Rossum.

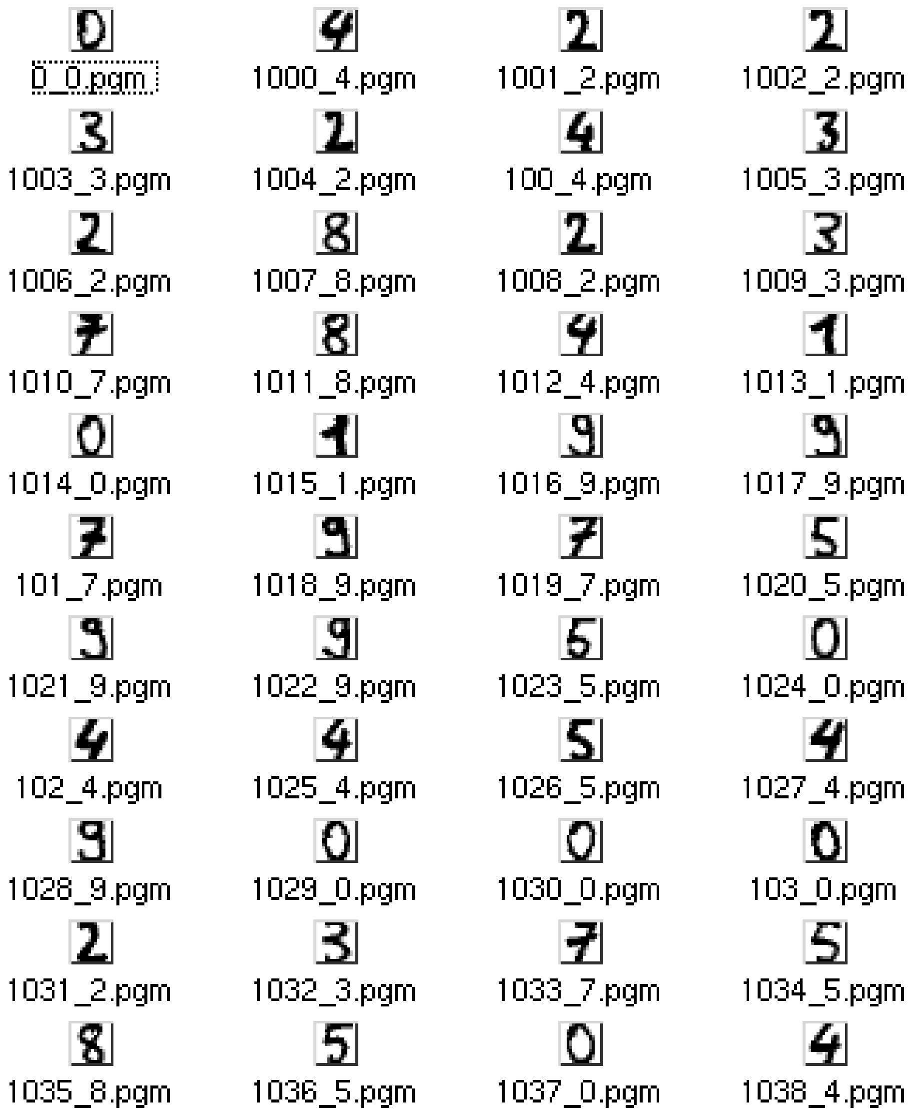

Figure 18: *Some examples of* 16×16 *images used for training*

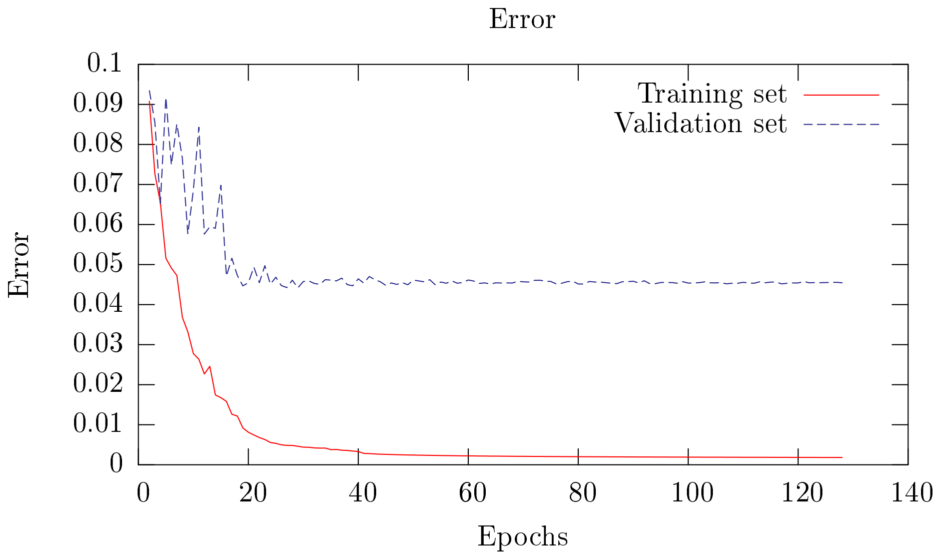

Figure 19: *Error trend in training a network for handwritten digits recognition with 14 neurons in the hidden layer. Best network was found at epoch 29*

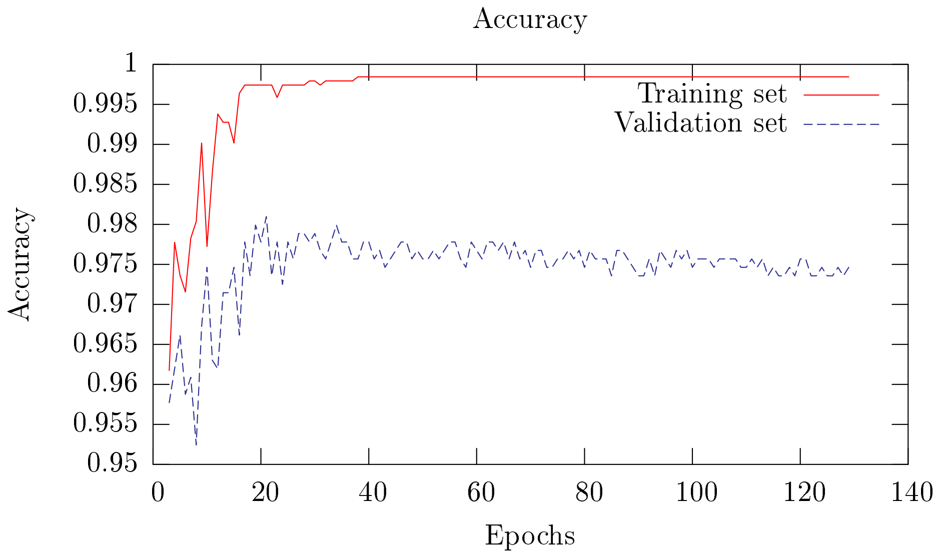

Figure 20: *Accuracy trend in training a network for handwritten digits recognition with 14 neurons in the hidden layer*

11 Benchmarks

We tested our application on some standard problems, taken from "Proben1" ([7]). The characteristics of these problems are shown in table 1, but for further information on the datasets please consult [7].

Dataset name	Type	Dataset size	Inputs	Neurons in hidden	Outputs
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

Table 1: The datasets. The type is either: a (approximation) or c (classification) and 16+8 in "Neurons in hidden" indicates that there are two hidden layers with 16 neurons in the first and 8 in the second

Performance was tested comparing to *fann* ([4]), a library developed by danish student Steffen Nissen you can find on **http://fann.sourceforge.net/**.

For each benchmark we divided the pattern set in a training set and a validation set, and we recorded the error reached on networks of the same dimensions every second for 120 seconds of execution by the two libraries. After 120 seconds we recorded the number of batches of the two libraries (the library which executes more batches during the same time is faster, of course!). On table 2 we show the results obtained on an Athlon 1.33 GHz with 256 MB RAM. The libraries were compiled with gcc 3.2 (this seems to be important!).

As we see in the table, the number of batches executed by our library is usually higher than the number of batches of *fann*. The error is usually of the same order, on training and on validation set. For exampe in the first problem our library executed 12361 batches versus 7788 batches of *fann*. Reached error is similar (a little smaller for fann).

In the following graphs in the x axis represents the seconds (from 0 to 120) and on the y axis the error reached at that time. Usually our library, because of its higher speed, completes more batches and so reaches a lesser error in the same time, except in the *gene* problem, on which our library behaves very badly.

In figure 27 we show the number of batches executed by the two libraries at each time. Our library is usually faster.

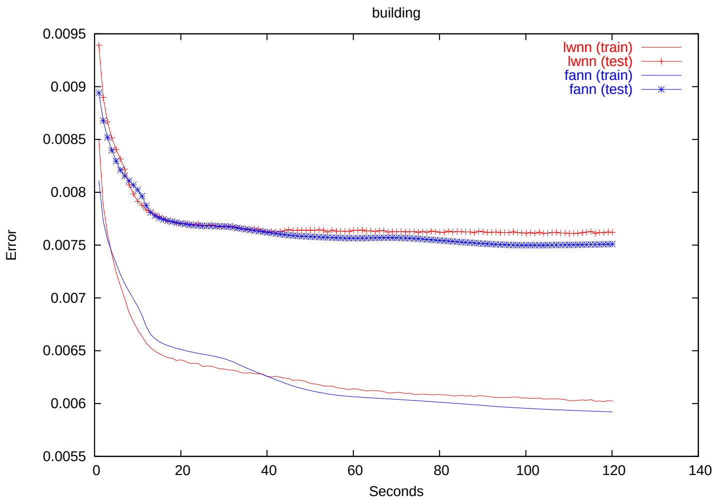

Figure 21: *Building. Error trend*

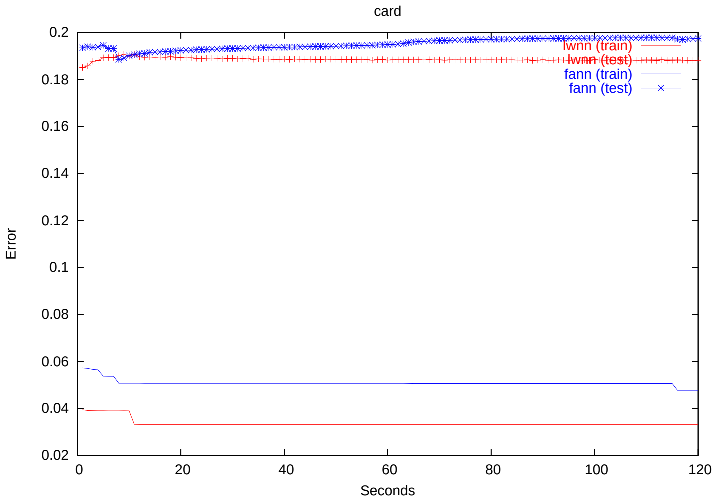

Figure 22: *Card. Error trend*

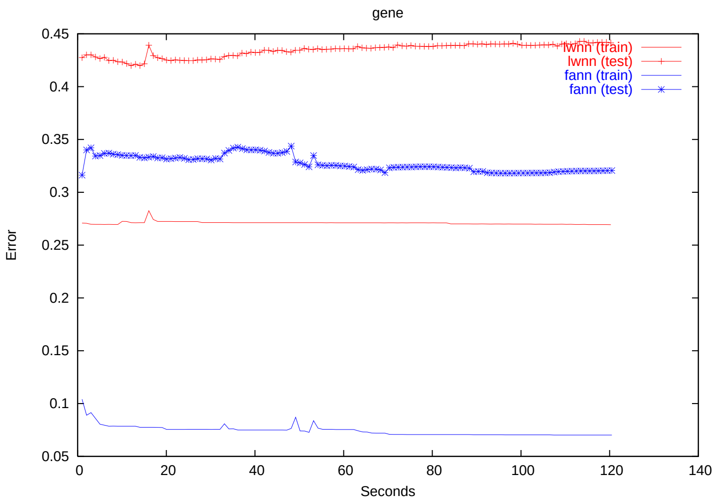

Figure 23: *Gene. Error trend*

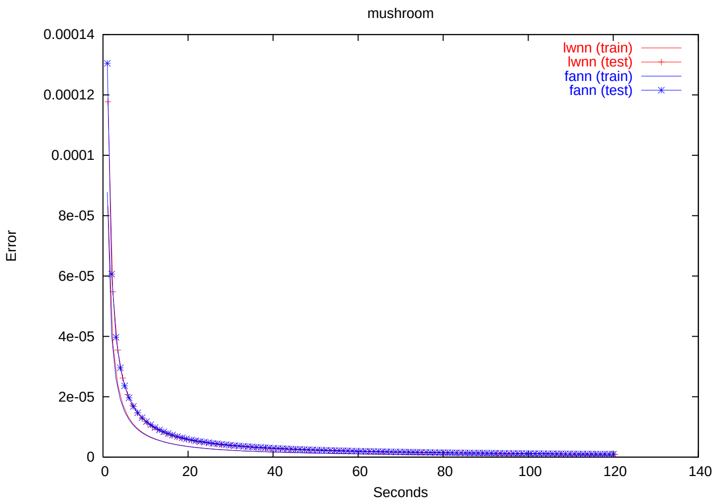

Figure 24: *Mushroom. Error trend*

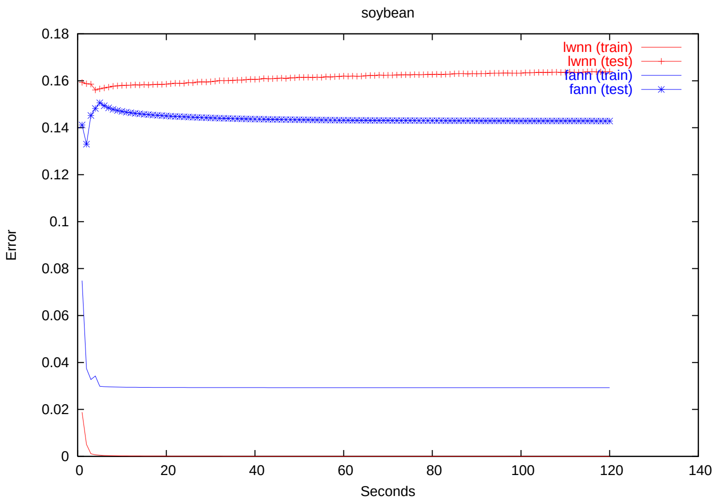

Figure 25: *Soybean. Error trend*

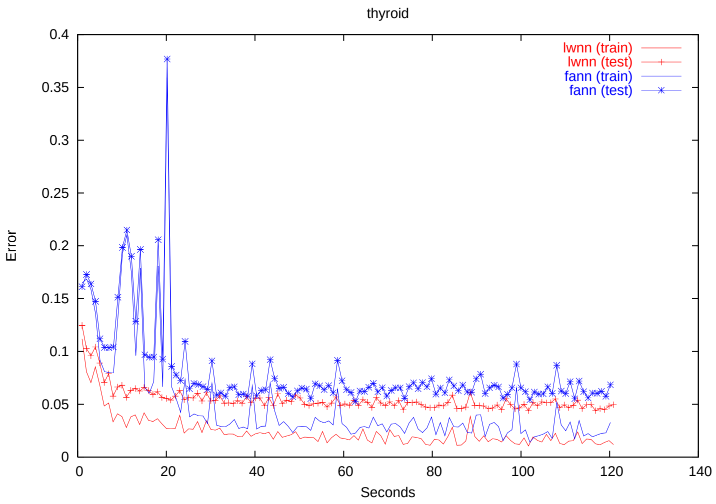

Figure 26: *Thyroid. Error trend*

	lwnn++		fann	
Dataset Name	Error	Batches	Error	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

Table 2: Benchmarks

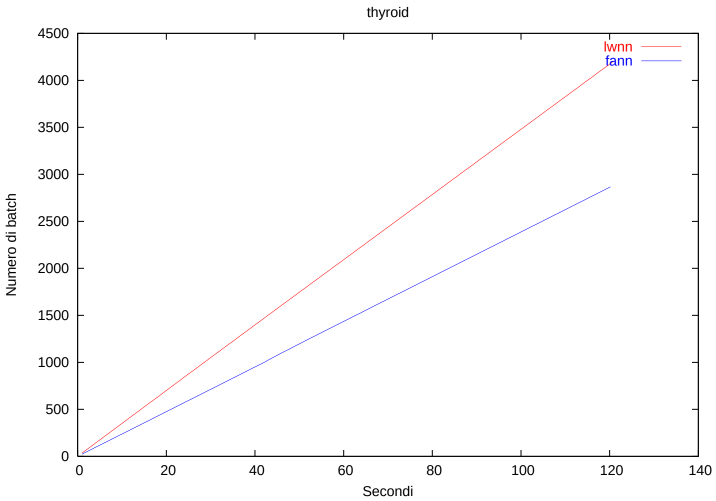

Figure 27: *Number of batches executed by the two libraries for the thyroid problem. As we can see* Lightweight Neural Network++ *is faster*

References

[1] Tom M. Mitchell, *Machine Learning* (1997) (chapter 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.