ELSEVIER Neurocomputing 48 (2002) 623–646

NEUROCOMPUTING

www.elsevier.com/locate/neucom

NEURObjects: an object-oriented library for neural network development

Giorgio Valentini[a,b,*], Francesco Masulli[a,b]

[a]*Istituto Nazionale per la Fisica della Materia, Genova, Italy*
[b]*DISI—Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, Via Dodecaneso 35, 16146 Genova, Italy*

Received 23 August 1999; accepted 13 June 2001

Abstract

NEURObjects is a set of C++ library classes for neural network development, exploiting the potentialities of object-oriented design and programming. The main goal of the library consists in supporting experimental research in neural networks and fast prototyping of inductive machine learning applications. We present *NEURObjects* design issues, its main functionalities, and programming examples, showing how to map neural network concepts into the design of library classes. © 2002 Elsevier Science B.V. All rights reserved.

Keywords: Neural network development; Fast prototyping of neural network applications; Object-oriented programming; Library of C++ classes

1. Introduction

The availability of software libraries for the development of neural network applications can significantly speed up the development time of a specific application, especially when neural network algorithms are implemented from scratch in order to embed them in new software products.

In neural network and machine learning research, we often need to compare different learning algorithms, evaluating their performances on different data sets. Moreover, research and experimentation with new neural network algorithms and

* Corresponding author. DISI—Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, Via Dodecaneso 35, 16146 Genova, Italy.
E-mail addresses: valenti@disi.unige.it (G. Valentini), masulli@disi.unige.it (F. Masulli).

applications sometimes requires to modify existing algorithms, or to recombine them, reusing components from existing learning machines.

In such cases, a set of C++ library classes, for neural network development, seems to be a proper solution to these problems, exploiting the modularity, reusability and versatility characteristics of C++ design and coding [36].

It is worth noting that only a few libraries in C++ for neural network development are easily accessible. Among the available ones, some of them use C++ as a "better C", according to a well-known Stroustrup's statement [37], employing just in part the potentialities of object-oriented (OO) programming (see, e.g., the libraries of the Timothy Masters's book [22]). Others need remarkable computational resources, as well as considerable training time for the software developer [16,17]. Other kinds of C++ libraries concern those developed with emphasis on specific neural network models. NNO [19], e.g., has focused its efforts on self-organizing incremental networks [13], providing only a simple implementation of classes dealing with multi-layer perceptrons (MLP).

In this paper we present *NEURObjects*, a set of C++ library classes for neural network development, exploiting the potentialities of OO design and programming [36]. The main aim of the project consists in providing a set of basic neural network classes, supporting design, implementation and comparative testing of classification algorithms. Hence *NEURObjects* is a general set of classes supporting research, application and training in neural networks. The library also provides high level programs for experimental research in neural networks and fast prototyping of applications in inductive learning tasks. Source code, compiled libraries, applications and HTML documentation are freely downloadable for educational and research purposes at the *NEURObjects* web site: `http://www.disi.unige.it/person/ValentiniG/NEURObjects`.

The paper is structured as follows: In Section 2 the reasons for using C++ classes in neural network development and the main goals of *NEURObjects* are discussed. In Section 3 the general library design is presented, showing also some basic ideas about neural network OO design. In the next section related works are summarized and compared with the present work. Section 5 describes the main way end users can use the library, and simple examples of *NEURObjects* usage and programming are given in Section 6. Conclusions and future developments of this work end the paper.

2. *NEURObjects* aims

2.1. C++ classes for neural network development

There are many reasons to choose C++ in order to build a library supporting fast development of neural network based systems:

- C++ is a good general-purpose language with many useful OO features [36,37].
- C++ has also a large flexibility, allowing to avoid the OO paradigm when needed, as we can use it as an "improved C language".

- The language's OO features allow us to decompose the library into a set of modular objects which can be tested independently.
- It is easy to expand and reuse C++ code.
- C++ is very efficient: none of his own features (overriding, overloading, function inlining, etc.) lowers execution speed.

Concerning the specific application to neural network development, OO design provides classes corresponding to neural network concepts; in particular:

- C++ linguistic constructions enable us to represent in a concise way features and functionalities of neural networks.
- Data abstraction allows us to define in a comprehensive way sets of neural structures and operations.
- Ensembles of neural nets can be easily accomplished by net-objects generated by C++ constructors.
- Polymorphism of virtual functions leads up to generalization and flexibility of the classes related to neural networks.

Moreover, by inheritance, we can easily extend existing classes in order to experiment with new learning algorithms, without wasting time on programming from scratch.

2.2. NEURObjects goals

The main goal of *NEURObjects* is to support experimentation, training, test and comparative study of neural network based systems. *NEURObjects* permits to perform in an easy and quick way the following tasks:

- Application of standard neural network algorithms to specific classification and regression problems.
- Fast prototyping of applications in neural network domain.
- Implementation of new neural learning algorithms or variants of already existing neural algorithms.
- Implementation of neural nets ensembles in order to recombine them and improve overall system's classification capabilities.
- Comparison of learning algorithms on specified classification problems.
- Generation of performance statistics such as accuracy, confidence level, rejection curves, error rates and confusion matrices.

In particular, the library includes classes implementing methods for ensembles of learning machines, such as Output Coding decomposition methods [28,23] and classes (under construction) implementing bagging and boosting [7,12].

3. Design of *NEURObjects* library

In this section, we outline the main characteristics of the library, showing how to apply OO design and programming principles to neural networks. Efficiency is

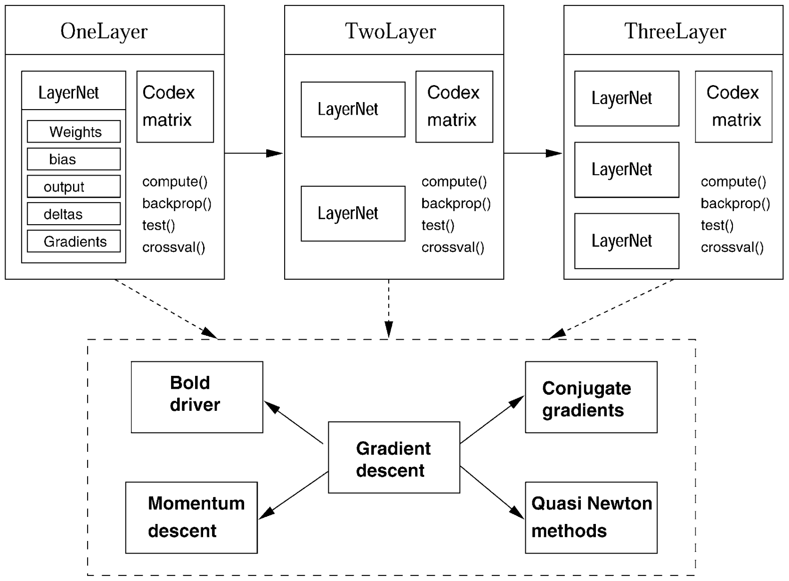


Fig. 1. *NEURObjects*: examples of inheritance, is-part-of and use relations among concept/classes (see text).

a key-feature in neural network applications, and the end of the section deals with this issue in the framework of C++ programming.

3.1. General design approach

The general design approach consists in mapping main concepts of the application domain into library classes: For instance, OO design allows the direct translation of the concept of neural network into the neural network class or from the concept of network layer directly into the layer class.

The concepts/classes are related each other by OO relations of *inheritance*, *is-part-of* (composition) and *use*, as illustrated in the examples of Fig. 1 regarding the structure of three neural network classes (OneLayer, TwoLayer and ThreeLayer) and some learning algorithms (grouped in a dotted box). In the figure, inheritance relation is represented by solid arrows, is-part-of relation by boxes inside other boxes (i.e. LayerNet is-part-of OneLayer), and use relations by dotted arrows.[1]

In Table 1 we show the matching between neural concepts relations and OO relations. We exploit this matching for the design of the library.

It is worth noting that a multilayer perceptron can be made, for instance, up by one, two or three layers, but the general features of these different MLP are

[1] The direction of the arrows in the figures is from parent to child, i.e. it is the opposite of the usual OO inheritance graphs.

Table 1
Design of *NEURObjects*: neural concepts and OO relations

Neural concepts relations	Object-oriented relations
Different MLPs own similar features and are hierarchically related.	**Inheritance:** A two-layer perceptron can *inherit* from a one-layer perceptron and a three-layer can *inherit* from a two layer.
MLP is made up by connected layers of units and every layer computes using only inputs coming from the layer below.	**Is-part-of relation:** A layer class *is-part-of* a neural network class.
Each MLP can use different learning algorithms.	**Use relations:** Each MLP class can *use* methods of different learning functional classes.

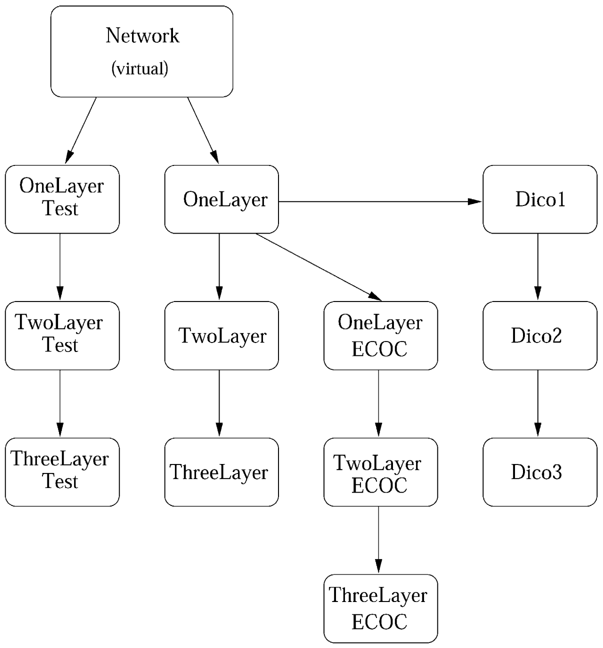


Fig. 2. Inheritance hierarchic tree of MLP classes.

similar; they can be designed using inheritance in such a way that a two-layer perceptron inherits from a one-layer perceptron and a three-layer inherits from a two layer (see Fig. 1). The relations of inheritance model the design of MLP classes: All classes inherit from *Network*, a basic abstract neural network class (Fig. 2). *OneLayerTest* and its derived classes are MLP with functionalities reduced to testing only; *OneLayer* and its derived are standard MLP with, respectively, 0, 1 and 2 hidden layers; *OneLayerECOC* and its children classes are MLP ECOC (multi-layer perceptrons with error correcting output coding) [9]; *Dico1* are MLP specialized for dichotomic classification problems.

The library directly supports perceptrons with three layers or less, but it is easy to build, by inheritance, perceptrons with more layers.

Moreover, as an MLP is made up by connected layers of units and every layer computes using only inputs coming from the layer below, we can decompose the overall computation of the net in the individual computations performed by each layer. As a consequence, the whole network can compute connecting the results of each layer in a suitable way. These simple concepts are transposed into *NEURObjects* by designing a layer class and using the relation "a layer is-part-of a neural network" (or in the same way the network is made up by layers of neurons). The method for computing the output of the neural net is given by the corresponding computing method used by its components (the layers):

```
/*
  compute: computes network output,
  using vinput as input vector.
*/
vector TwoLayer::compute (vector vinput)
{
  // hidden layer computation
vector hidout = layer[HIDDEN1].compute(vinput);
  // output layer computation
return (layer[OUTPUT].compute (hidout));
}
```

In this case also OO design directly agrees with the concept of the neural network domain, since connecting the layers of the net corresponds to the mathematical neural concept of function composition.

In order to illustrate an example of *use* relation, let us observe that the same net can use different learning algorithms. Most of the training algorithms involve an iterative procedure in order to minimize an error function, with adjustments of weights made in a sequence of steps. At each step we can distinguish between two distinct stages: Evaluating the error function derivatives and updating the weights by using the computed derivatives [5]. As these stages are substantially separated, we can design learning algorithms as functional classes whose main task is updating the weights of the net, no matter the way this task is done. The same neural network now can *use* different learning algorithms, using their methods to update the weights. Moreover, defining the fundamental operations of an abstract learning algorithm, we can set out pure virtual methods of a basic abstract learning class; as a consequence, it is easy to add new learning classes, overriding virtual pure methods of the basic abstract class, without any modification in other parts of the code. A hierarchic tree of learning functional classes is given in Fig. 3: *Learning* is the basic virtual class; *Gradient*

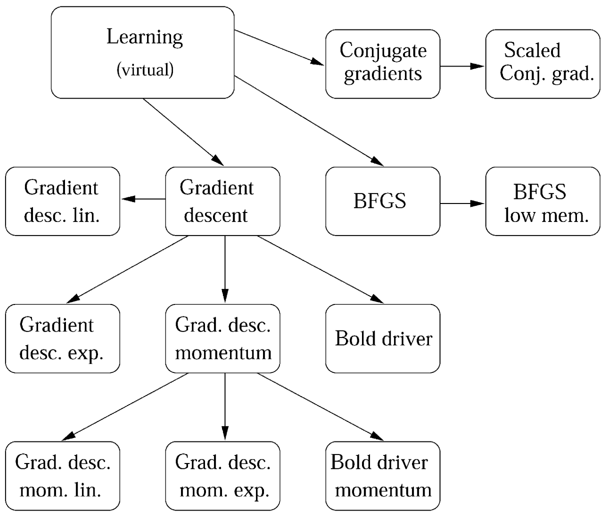


Fig. 3. Inheritance hierarchic tree of learning functional classes.

descent lin. and *Gradient descent exp.* are variants of the classic gradient descent algorithm (*Gradient descent*); *Grad. desc. mom. lin.* and *Grad. desc. mom. exp.* are variants of the gradient descent with momentum algorithm (*Grad. desc. momentum*).

3.2. *NEURObjects classes*

NEURObjects develops classes representing concepts in neural network domain and uses public domain software for general operations not specific to neural networks. Using a functional approach, we can divide the library in the following groups of classes:

- *General support classes*. These classes are not specific to neural networks. For instance, LEDA's [31] library classes allow the implementation of the basic classes used by *NEURObjects* (lists, queues, vectors, matrices, balanced trees, dictionaries).
- *I/O and pre-processing classes*. Classes for reading and writing data files, converting attribute types among formats (e.g. automatic conversion of classes from string-labeled into numeric-labeled format), normalizing attribute values, etc.
- *Automatic data generation classes*. Classes generating synthetic data files for training and testing by using controlled random or deterministic techniques.
- *Neural network building classes*. These classes are intended to build neural networks ranging from simple perceptrons to different types of multi-layer perceptrons.

- *Learning algorithms classes*. They implement several standard algorithms, from simple gradient descent to variations of conjugate gradients [3] or quasi-Newton methods (such us BFGS algorithm) [4], [2] and others methods.
- *PND cleasses*. Classes implementing multi-class classifiers through output coding decomposition methods [9,28,23]. Polychotomies are performed recombining parallel independent dichotomizers based upon perceptrons. These classifiers are called *Parallel Non-linear Dichotomizers* (PND) [25] and are implemented in specific template classes whose parameter is the type of decomposition [28]. Using single layer perceptrons as dichotomizers, we obtain classifiers similar to that proposed by Alpaydin and Mayoraz [1], while using MLP we obtain the PND studied in [25].
- *Performance statistics classes*. Classes handling performance statistics such as accuracy, confidence levels, rejection curves, learning rates and confusion matrices, and classes implementing estimations of dependence among output errors in learning machines, using measures based on mutual information [26,27].
- *Comparing algorithms classes*. Classes handling statistical test for comparing algorithms on a particular learning task: they detect whether exists a real (statistically significant) difference among different learning algorithms [8].

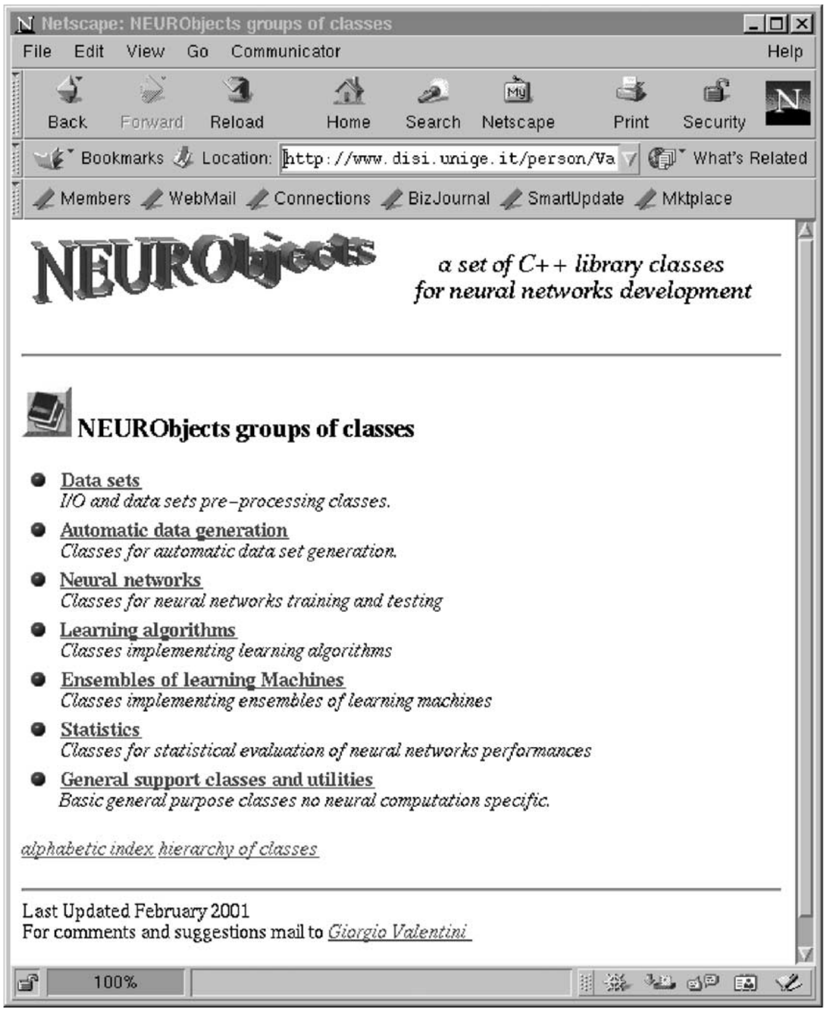
Actually *NEURObjects* consists of about 10 000 lines of code developed using GNU g++ compiler on PC-Linux and Sun-Solaris workstation platforms. Each class and related code are documented in HTML and Postscript using DOC++, a documentation system for C++ and Java [42] (an example is shown in Fig. 4).

3.3. Efficiency of C++ libraries

A key-feature of neural network software is its efficiency. C++ combines flexibility, reusability and extendibility with this important feature.

In fact, C++ has been designed to devolve compactness and efficiency of executable code to compilers technology. The new C++ features have added no substantial overhead with respect to C. Considering, for instance, one of the new key-features of C++, i.e. the implementation of proper functions calls whose type is unknown at compilation time, no overhead is added. In fact, the call mechanism needs to examine the object and obtain information that is added to the object by the compiler: A simple implementation mechanism consists in converting the name of the function into a table of function pointers (virtual functions table). This mechanism with current compilers and microprocessors technology implements this key feature of OO programming with no overhead. Moreover, C++ avoids features that can add overhead to run-time execution. For instance, a pure OO language (e.g. Small-Talk) checks during execution if it is possible to perform a specific operation (i.e. to execute a proper function) on a specific object. As this dynamic type checking introduces overhead, C++ uses a static type checking, assuring the

[2] At present, some of these classes are not included in the on-line distribution of *NEURObjects*.


Netscape: NEURObjects groups of classes
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security
Bookmarks Location: http://www.disi.unige.it/person/Va What's Related
Members WebMail Connections BizJournal SmartUpdate Mktplace
100%

NEURObjects

a set of C++ library classes for neural networks development

NEURObjects groups of classes

- Data sets
  I/O and data sets pre-processing classes.
- Automatic data generation
  Classes for automatic data set generation.
- Neural networks
  Classes for neural networks training and testing
- Learning algorithms
  Classes implementing learning algorithms
- Ensembles of learning Machines
  Classes implementing ensembles of learning machines
- Statistics
  Classes for statistical evaluation of neural networks performances
- General support classes and utilities
  Basic general purpose classes no neural computation specific.

alphabetic index hierarchy of classes

Last Updated February 2001
For comments and suggestions mail to *Giorgio Valentini*

Fig. 4. An example of HTML documentation of *NEURObjects*.

users that the operations specified in class declarations are the only one accepted by the compiler.

Moreover, C++ interfaces well with other programming languages, and we can easily encapsulate functions and libraries implemented in other languages. In fact, in future releases we plan to use *BLAS* [10] and *LAPACK* [2] libraries for speeding-up the computation: *BLAS* offers the most efficient routines (implemented in FORTRAN) for vector and matrix multiplications, operations that are fundamental for speeding-up execution both in the forward and in the backward step of the backpropagation algorithm. *LAPACK*, that uses *BLAS* for low-level subroutines, offers a set of very efficient implementation of linear algebra functions, such matrix

QR decomposition or pseudoinverse matrix calculation. We have also planned to use specialized versions of these libraries for family of microprocessors, such as Pentium III or Pentium IV Intel optimized *BLAS* libraries.

4. Related work

Different standard network simulation packages are available for experimentation with neural nets. We cite only the most popular such as SNNS developed at Stuttgart University [43], Aspirine developed at MITRE Lab [21], or Open Simulator, developed at ETH Zurich [20]. These packages represent good solutions for neural network simulation, but in this section we deal with software libraries for neural network development. In particular, we focus on C++ libraries, as they join good run-time execution speed with the flexibility and reusability features of OO programming.

4.1. Libraries that use C++ as a "better C"

There are not many C++ libraries for neural network development, and many of them use C++ as an imperative language, exploiting only in part its OO characteristics, reducing C++ to an extended version of the C language. For instance, the Rao & Rao's library [33], while it offers different learning algorithms and applications with neural networks and fuzzy systems, presents most of the code as "masqueraded" C++ code; moreover, code is structured in very large blocks, with only limited efforts for decomposing and organizing it into modules. Master's book and its included software [22] suffers from similar problems, even if we have a more rational modularization of the code, but with classes and C functions mixed without a clear OO design and implementation.

Different is the approach to OO programming in Welsted's library, where programmatically the author declares that the neural code is written in C and only the graphical interface is written using a C++ toolkit [41]. Watson's book shows examples of application of OO design and programming concepts to neural networks, but it mainly deals with general C++ programming techniques applied in different contexts and applications, offering only a limited set of classes specific to neural network development [40].

4.2. ANNDE library

An interesting pioneering example of applications of OO concepts to neural networks is represented by the paper of Hung and Adeli [15]. Although this work is oriented to specific domains (structural engineering applications), general object-oriented principles are applied to the design of a C++ neural network library (*ANNDE*: an artificial neural network development environment). Similarly to our library, the main neural network concepts are mapped into neural network classes, but the overall design is very compact: Only few classes model the entire system, while *NEURObjects* presents a more articulated architecture. The separation between learning strategies and learning processes (corresponding to different classes) allows the separation between general learning approaches (supervised, reinforcement and competitive learning) and their actual implementation using learning algorithms. The inheritance prevails among the other relations between classes and models the design of the library. This in some cases seems to limit the extendibility and the reusability of the library. For instance, the learning domain (LD) class, i.e. the class implementing input and output of the nets, is the base class for Neural Net (NN), i.e. the class implementing the structure of the neural networks. This design corresponds to the wrong relation "the net *is-a* learning domain", while the correct relations are "the net *has-a*" or "*uses* a learning domain". On the other hand, this compact design modeled mainly through inheritance relations could be appropriate in specific domain applications, where reusability and extendibility are not the main goals. In fact, this library shows good results in applications related to structural engineering problems.

4.3. Blum's library

Blum's book is a good introductory book to OO neural network programming [6]. Many of Blum's concepts and design features are present in *NEURObjects*. For instance, at the lower level fields of neurons are represented as vectors, synapses are represented as matrices and also, at this level, inheritance and composition relations are used in the same way, e.g. the matrices representing neural synapses are components of the neural network objects. Moreover, as in *NEURObjects*, a base abstract class is proposed for encapsulating the basic abstract characteristics of all neural nets. However, there are also substantial design and implementation differences. For instance, Blum chooses to model the backpropagation algorithm as a derived class of the base abstract class net, mixing in the same class the structure and the properties of a multi-layer perceptron with a single simple version of the backpropagation algorithm. Even if this choice can simplify the overall design and can be justified from an educational point of view, it is not the right one for a library for neural network development. In fact, programmers build their own algorithms, using the classes as basic components or developing new classes from the existing ones. On the contrary, in our implementation we have exploited the fact that we can distinguish two stages in the backpropagation algorithm: evaluating the error function derivatives and updating the weights. In *NEURObjects* we have developed a hierarchy of learning classes separated from the net classes, that simply *use* a learning algorithm: In such a way, an end user can select between different learning algorithms, without developing new derived classes, and a programmer can develop new learning algorithms without modifying the neural net classes.

Some implementation details also highlight that *NEURObjects* is conceived for neural network development in research and fast prototyping of applications, while Blum's library limits itself to demonstrative applications. For instance, in Blum's library neural network parameters are stored in files. It could be a good and clean choice, but if we have to train large number of neural networks with a large range of different parameters (e.g. for model selection), it is useful to provide classes methods or, better, general purpose applications with multiple parameters, and easily change them inside shell or Perl scripts. On the other hand, we outline that these different approaches are not mutually exclusive and can be easily combined in the same library.

4.4. Neural Network Objects

Neural Network Objects (*NNO*) [19] is a C++ library specialized on self-organizing incremental networks [13]. The abstract classes are used for modeling the library: an abstract neural network class at the root of the hierarchy has two derived abstract classes, one for supervised and another for unsupervised neural networks. From each of the two abstract children classes are derived the other classes of the library. This library provides implementations of Growing Neural Gas and Growing Cell Structures [14] together with Kohonen Feature Maps [18], but only a basic implementation of a MLP. Comparing the overall architecture of *NNO* with *NEURObjects*, we can note that the relations of compositions are not extensively used in *NNO*. As a consequence, the design is not as modular as in *NEURObjects*: For instance, inside the MLP class, layers are not designed as component classes and learning algorithms are not implemented as separated classes. However, *NNO* is a very nice library for experimenting and developing applications with self-organizing incremental networks.

4.5. MLC++

We conclude this brief review of C++ neural network libraries with *MLC++* that it is not in a proper sense a library for neural network development [16,17]. In fact *MLC++* probably represent the best effort to provide a set of C++ library classes for the development of machine learning applications. In fact this project, started at Stanford in the middle of 1990, has produced a large library with classes implementing a large set of learning machines, but with only a limited implementation of neural networks. Nevertheless, the excellent design and implementative solutions proposed in this library can be successfully used for designing and implementing neural network libraries.

Different induction algorithms are implemented in *MLC++*, such as decision trees, Naive–Bayes, nearest-neighbor and others; more interestingly, the library permits to interface *external inducers*, i.e. induction algorithms written by users. Another interesting issue is represented by *wrappers*, i.e. C++ objects that treat algorithms (implemented as separate objects) as black boxes and act on their outputs. For instance, accuracy estimators wrappers use any of a range of methods, such as bootstrap or cross-validation to estimate the performance of an inducer. This wrapper adds flexibility and power to the library, because we can use the same wrapper with different induction algorithms. This feature is only partially present in *NEURObjects*: e.g. we have no wrapper for cross-validation, that it is simply implemented as a public method in neural network classes. However, in boosting classes, we use this feature to wrap a general and unspecified learning machine in order to use it as base learner in boosting algorithms. In this way, we can easily add new learning machines to our library, using them as base learners in boosting algorithms, without changing code for boosting itself. *MLC*++ allows us also to build easily hybrid algorithms, i.e combinations of different learning algorithms; *NEURObjects*, being a neural network library, does not offer the same rich variety of learning algorithms, but it provides classes and facilities to build ensembles of neural networks, e.g. output coding decomposition, bagging and boosting ensembles.

Considering the overall software architecture, *MLC*++ library is designed through sets of independent units, encapsulating in different classes different concepts related to learning machines. Inheritance is used only when it helps the programmer: analogously to *NEURObjects* a set or relatively independent classes is given, and the library is built exploiting all the relations of inheritance, composition and use between classes proper of OO design and programming. The result is a set of classes with clear-cut interfaces, easily re-usable and extendible for developing and comparing learning machine algorithms. From an educational standpoint, our library, smaller and limited to neural networks, takes only a little training time, while *MLC*++ is a large machine learning library and consequently, it requires a relatively long training time. In conclusion, these libraries are not comparable for dimensions and also cover different areas of machine learning, but share similar applications of OO concepts and principles to software library design.

5. *NEURObjects* for neural network developers

Developing programs for training and testing neural networks using *NEURObjects* requires just a few lines of C++ code, each corresponding to a concept or an operation of the application domain:

Multilayer perceptron building, training by means of a learning algorithm and testing the resulting neural network need only one or few lines of code.

Programming with *NEURObjects* consists in selecting the proper components useful for the desired task and, when it is necessary, in overriding the behavior of some components to fully fit them to a specified task. The work can be done producing at each step a fully working program writing at each stage only few lines of code. To show that, in the next sections, we will present some simple example of *NEURObjects* programming.

In detail, we can distinguish three main ways end-users can use *NEURObjects*:

(1) *Using high level applications*. Some general purpose applications, implemented using the library classes, allow to easily and quickly experiment on classification problems. These basic applications allow neural networks training and testing using different types of neural nets, learning algorithms, and testing methods. Each application has a basic simple use and it is possible to choose

different options in order to solve a problem. Some examples are showed in the next section of this paper.

(2) *Building applications by NEURObjects classes*. It is possible to use *NEURObjects* classes to build customized applications. This task requires at least a bit of C++ programming know-how, but applications gain in power and pliability. Some simple examples are showed in the next section of this paper.

(3) *Developing new classes*. Programmers can develop their own classes, learning algorithm, set of neural nets, or other similar advanced programming tasks, using the library classes as parents. It is also possible to extend the existing classes by inheritance or class composition.

6. *NEURObjects* usage and programming

In this section, we show some simple usage examples of *NEURObjects* classes and its basic general purpose applications in order to solve different classification problems: The first one by using synthetic data, generated by an utility program of *NEURObjects* itself, and the others by using data from UCI repository [29].

6.1. Examples of synthetic data generation

The *NEURObjects* class *Patgen* can generate synthetic data files in ASCII format and an info file with information about the generated data. These data consist of clusters, each centered around a point in the input space. Each class is connected with each cluster (but it is possible to connect one class with more clusters). The clusters are generated by a normal distribution, centered around the generator point.

For the first classification problem we edited an info file (p6.info) using the class *Patgen* in order to generate a data file with six clusters. In this way, we can define the dimension and the number of patterns, the centers of the clusters, and the standard deviation of the normal distribution along main axes. These parameters may also be randomized, according to the options supplied to the application. For instance, the following commands generate two six-class tridimensional data files p6.train and p6.test using information written in the file p6.info:

```
dodata -t readinfo p6.train -ninfo p6
dodata -t readinfo p6.test -ninfo p6
```

6.2. Examples using general-purpose applications

We now show how to use general-purpose applications in classification problems. From a general point of view, these applications require only, as parameters, the name of data set files, but supplying more parameters we can select more specific options. More examples are on-line available at *NEURObjects* web site.

6.2.1. Examples with the application nn

By using the application *nn*, we can train and test an arbitrary multilayer perceptron (MLP) on specified train and data sets. Program options allow to define the structure of the net, its learning algorithms and parameters. The line command for *nn*, using the earlier generated data set, is

```
nn p6.train -test p6.data -nc 6 -d 3 -rate 0.02 -s p6.net
```

and the corresponding output is

```
Type: Multi layer perceptron
Layers number: 2
Input dimension: 3
Units hidden layer: 5
Classes number: 6
Stop conditions:
Maximum number iterations: 1000
Error threshold: 0.1
Learning algorithm type: Gradient descent
Learning rate: 0.02
By pattern learning.

Training results:
RMS error = 0.0970926
Iterations = 21
ELAPSED CPU time : Minutes : 0 seconds : 7

Testing results:
Errors:
Output - computed class - target class
6 0.31583 0.48789 0.14634 0.02548 0.19459 0.04472 - 2 - 1
6 0.04162 0.02127 0.53876 0.44078 0.01896 0.33760 - 3 - 4
Network saved in file p6.net

Confusion matrix :
199      0      0      0      0      0
  1    200      0      0      0      0
  0      0    200      1      0      0
  0      0      0    199      0      0
  0      0      0      0    200      0
  0      0      0      0      0    200

Total errors:
Errors number : 2 : Error percent : 0.166667 %
```

Table 2
Testing results on p6 data

H. units	Algorithm	Errors	% Error
11	Grad. desc.	3	0.25
11	Grad. desc. mom.	3	0.25
11	Bold	2	0.17
11	Bold mom.	3	0.25
9	Grad. desc.	2	0.17
9	Grad. desc. mom.	3	0.25
9	Bold	2	0.17
9	Bold mom.	2	0.17
7	Grad. desc.	2	0.17
7	Grad. desc. mom.	2	0.17
7	Bold	2	0.17
7	Bold mom.	2	0.17
5	Grad. desc.	2	0.17
5	Grad. desc. mom.	2	0.17
5	Bold	2	0.17
5	Bold mom.	2	0.17

It is possible to save the history of learning errors during the computation, done by epoch, and the whole output of the net for following statistical elaborations, by using the following options that produce the files p6.herr (history errors) and p6.out (output of the net with test file as input):

```
nn p6.train-test p6.data -nc 6 -d 3 -rate 0.02 \
           -s p6.net -serr p6 -out p6
```

Several algorithms can be used by the option -alg. Table 2 shows some results obtained by training and testing with the synthetic data sets p6.train (training set) and p6.test (test set), both containing 1200 three-dimensional examples.[3] Neural networks with two-layer perceptrons have been used. Hidden units vary between 5 and 11 and training is performed using classical gradient descent [34], gradient descent with momentum, bold driver [39], and bold driver with momentum. The same initial seed for random generation of initial weights is used. In the table, for different number of hidden units (H. units) and different training algorithms (Algorithm), the absolute error (Errors) and the error rate (%Error) are reported. Training and testing time require few seconds on a Linux PC platform with 32K RAM memory and AMD K6 microprocessor.

6.2.2. Examples with the application pnd

Using the application *pnd*, we can train and test an ensemble of MLP generated by output coding decomposition methods [28]. A PND [24] is a multi-class

[3] The *p6* synthetic data sets are available at `ftp://ftp.disi.unige.it/person/ValentiniG/Data/p6.tgz`.

Table 3
Testing results and computation time on *optdigits* data, using output coding decomposition ensembles of neural networks

Ensemble algorithm	No. of base learners	No. of hidden units	% Error rate	Total time (min.s)	Avg. base learner time (min.s)
CC-gd	45	20	2.17	86.45	1.55
CC-bold	45	20	2.34	97.30	2.10
CC-gd	45	30	1.89	116.44	2.36
CC-bold	45	30	1.95	133.06	2.58
CC-gd	45	40	2.12	149.36	3.19
CC-bold	45	40	2.06	174.06	3.52
ECOC-gd	15	20	2.00	22.21	1.29
ECOC-bold	15	20	2.06	26.24	1.46
ECOC-gd	15	30	2.89	29.35	1.52
ECOC-bold	15	30	2.95	36.54	2.28
ECOC-gd	15	40	2.45	38.35	2.35
ECOC-bold	15	40	2.28	47.09	3.08
OPC-gd	10	20	3.00	22.09	2.13
OPC-bold	10	20	2.89	23.28	2.20
OPC-gd	10	30	2.17	29.28	2.56
OPC-bold	10	30	2.23	32.31	3.14
OPC-gd	10	40	2.45	44.30	4.26
OPC-bold	10	40	2.45	47.21	4.42

classifier composed by a set of dichotomizers implemented through MLP; each base classifier of the ensemble executes a subtask (a dichotomy) of the overall task of multi-class classification.

Table 3 summarizes some results of PND implemented by MLP with one hidden layer on the UCI data set *optdigits*. This data set is composed by 10 classes with 64-dimensional input patterns: we have used for training and testing the two separated data sets of, respectively, 3823 and 1797 samples available at UCI repository web site [29]. The first column specifies the output coding decomposition ensemble type: CC stands for correcting classifiers [30], ECOC for error correcting output coding [9] and OPC for one per class, while gd specifies that a standard backpropagation algorithm with fixed learning rate $\eta = 0.3$ has been used, and bold stands for bold driver algorithm [4]. In the second column, there is the number of the base learners of the ensemble and in the third, the number of hidden units of the hidden layer. In the next column are shown the classification error on the test set and the overall computational time (in minutes and seconds) required for training all the MLP composing the ensemble. In the last column is shown the average computational time for training a single base learner of the ensemble. The experimentation has been performed using a RedHat 6.0 Linux system with a 350 MHz Pentium II microprocessor and 64 Mbytes of RAM.

6.2.3. Examples with the application pnd_cv

The application *pnd_cv* can train and test a PND using the method of cross validation [35]. The folds can be prepared using the program *dofold* in a simple way:

```
dofold glass.data -nf 10 -na 9 -name glass
```

This command build the folds for a tenfold cross-validation test. The data set is *glass* from the UCI Machine Learning repository [29]. Ten folds from the data file glass.data (named glass.fi.train and glass.fi.test varying i from 1 to 10) are extracted (the option -na specifies the number of attributes of the data sets).

Using *pnd_cv*, it is possible now to exploit a tenfold cross-validation, using for example, a decomposition scheme one-per-class (OPC), generating a number of dichotomizer equal to the number of classes, each dichotomizer choosing a class versus all others:

```
pnd_cv glass -res glass -nc 6 -d 9
```

The option -nc specifies the number of classes and -d the dimension of the attributes. The results of the tenfold cross-validation are saved in the file glass.cv (option -res). There are also many more options that can be used in order to change the default behavior of the PND, such as the kind of decomposition, the structure and learning algorithms of the dichotomizers and so on (see the *NEURObjects* web site for a full description of the available options).

NEURObjects high level applications are well suitable for neural network experimentations requiring extensive training and testing tasks: Cross-validation of many MLPs can be performed launching a single high level application of *NEURObjects*. These applications have been used in different research and application works, such as comparative evaluation of output coding decomposition methods [23], evaluation of different architectures in ECOC learning machines [24], analysis of the dependence among output errors in ECOC learning machines [27], applications of MLP and PND to electronic noses [32], and in applications to bioinformatics problems [38].

6.3. Programming examples using library classes

In order to build-up a neural network application using *NEURObjects* classes, at least a bit of C++ programming know-how is necessary, but this will increase power and pliability of the application itself.

The main steps necessary to build an application, i.e. to train and test a MLP for a classification task using *NEURObjects*, are

(1) Calling general initializing functions,
(2) Data sets preparation,
(3) Building the desired MLP,
(4) Building the desired learning algorithm,
(5) Training the net,
(6) Testing the net.

Each of these steps requires only a few lines of code:

```
int main(int argc, char* argv[])
{
 int      num_train;
 unsigned nclass   = 6;    // number of classes
 unsigned nhidden  = 5;    // number of hidden neurons
 unsigned num_attr = 3;    // dimension of attributes
 double   eta      = 0.02; // learning rate
 unsigned iter     = 0;    // iteration number
 double   err      = 0.0;  // training error

// 1. General initialization of the NEURObjects environment
NEURObjectsInit();

// 2. Construction and preprocessing of the training and test set
num_train = wc ("trainfile");
TrainingSet trainset(num_attr, num_train, "trainfile",
TrainingSet::last);
trainset.normalize();

num_train = wc ("testfile");
TrainingSet testset( num_attr, num_train, "testfile",
TrainingSet::last);
testset.normalize();

// 3. Build the desired MLP, here a two layer MLP
//    with nhidden hidden units,
//    num_attr inputs and nclass outputs
TwoLayer mynet(nclass, nhidden, num_attr);

// 4. Build the desired learning algorithm, here
//    a simple gradient descent algorithm with learning rate eta
GradientDescent gd(eta);

// 5. Init the weights and train the net by pattern using
//    the previous learning algorithm
mynet.init_weights_norm();
mynet.Learn_by_pattern(trainset, gd, iter, err);

// 6. Test the net using the test set and print errors
mynet.test(testset);
mynet.print_errors();
}
```

This example is a working program of about 10 lines of code. It is possible to use other algorithms to train and test the net adding three or four lines of code. For instance, one can re-train the previous net by a bold driver algorithm and test it by appending to the previous file the following lines:

```
BoldDriver bd();
mynet.init_weights_norm();
mynet.Learn_by_pattern(trainset, bd, iter, err);
mynet.test(testset);
```

Adding only some lines of code to the previous example, we can save the trained net and reload it to continue training with different parameters:

```
// save the structure, parameters and weights of the net
mynet.save_weights(FileNameoftheNet);
BuildNet buildnet(); // contructor of net builder
// read the structure of the net and
// its weight and builds it
OneLayer* reloadnet = buildnet.build(FileNameoftheNet);
// continue training and testing
reloadnet-> Learn_by_pattern(trainset, gd, iter, err);
reloadnet-> test(testset);
```

Moreover it is simple to build a PND and train it by cross validation:

```
...
// number of neurons in each layer
unsigned neurons_num[3]= {num_attr, nhidden2, nhidden1;}
// type of dichotomizer activation functions
act_func t_fun [3]= {sigmoid, sigmoid, sigmoid;}

// dico is an object for specifying dichotomizer type
DicoType dico(nlayers, neurons_num, t_fun);

// Constructor of a PND with
// ECOC decomposition and dichotomizer of type dico
PoliDico<d_ECOC_ex > * pd=
new PoliDico<d_ECOC_ex>(nclass, dico, sigmoid);
// n-fold cross validation
pd->CrossValidate(namefile, nfold, num_attr,
                              resultfile, learninfo);
// destruction of the polychotomizer
delete pd;
```

It is also possible to build array of more complex structures of MLPs or dichotomizers and train and test each of them by different learning algorithms.

7. Conclusions

NEURObjects is a set of C++ library classes for fast neural network development, exploiting the potentialities of OO design and programming. The library provides a set of basic neural network classes for fast prototyping of inductive machine learning applications based on neural networks, cutting off the costs related to implementation from scratch in systems embedding neural network technology.

The set of *NEURObjects* library classes enable neural network researchers to speed up experimentation and testing of new learning methods, and neural network developers can easily use existing classes to build scientific applications. Moreover, the library has been proved to be helpful in teaching neural networks, providing facilities in order to visualize neural network parameters, during training, and to analyze training and testing results.

At the actual development stage, the library consists of about 10 000 lines of code, and provides a set of basic neural network classes, including an implementation of the PND ensemble model. In the coming release, we will add classes for bagging and boosting ensembles of learning machines. In future releases, we plan also to use *BLAS* libraries for speeding-up the computation. Another planned development consists in the parallel implementation of PND classes using Message Passing Interface libraries [11], in order to exploit the parallel nature of output coding decomposition ensembles.

Acknowledgements

We would like to thank the anonymous reviewers for their suggestions and constructive criticism. This work has been partially funded by Progetto finalizzato CNR-MADESS II and University of Genova.

References

[1] E. Alpaydin, E. Mayoraz, Combining linear dichotomizers to construct nonlinear polychotomizers, Technical Report IDIAP-RR 98-05, IDIAP—Dalle Molle Institute for Perceptual Artificial Intelligence, Martigny, 1998.

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorensen, LAPACK Users' Guide, 3rd Edition, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1999.

[3] E. Barnard, Optimization for training neural nets, IEEE Trans. Neural Networks 3 (2) (1992) 232–240.

[4] R. Battiti, F. Masulli, BFGS optimization for faster and automated supervised learning, INCC 90 Paris, International Neural Network Conference, Kluwer Academic Publisher, Dordrecht, Germany, 1990, pp. 757–760.

[5] C.M. Bishop, Neural Networks for Pattern Recognition, Clarendon Press, Oxford, 1995.

[6] A. Blum, Neural Networks in C++, Wiley, New York, 1994.
[7] L. Breiman, Bagging predictors, Mach. Learning 24 (2) (1996) 123–140.
[8] T.G. Dietterich, Approximate statistical test for comparing supervised classification learning algorithms, Neural Comput. 10 (7) (1998) 1895–1924.
[9] T.G. Dietterich, G. Bakiri, Solving multiclass learning problems via error-correcting output codes, J. Artif. Intell. Res. 2 (1995) 263–286.
[10] J.J. Dongarra, J. Du Croz, I.S. Duff, S. Hammarling, A set of level 3 basic linear algebra subprograms, ACM Trans. Math. Software 16 (1990) 1–17.
[11] J.J. Dongarra, P. Kacsuk, N. Podhorszki (Eds.), Recent Advances in Parallel Virtual Machine and Message Passing Interface: Seventh European PVM/MPI Users' Group Meeting, Springer, Berlin, 2000.
[12] Y. Freund, R.E. Schapire, Experiments with a new boosting algorithm. Proceedings of the 13th International Conference on Machine Learning, Morgan Kaufmann, Los Altos, CA, 1996, pp. 148–156.
[13] B. Fritzke, Kohonen feature maps and growing cell structures—a performance comparison, in: L. Giles, S. Hanson, J. Cowan (Eds.), Advances in Neural Information Processing Systems, Vol. 5, Morgan Kaufmann, San Mateo, CA, 1993.
[14] B. Fritzke, A growing neural gas network learns topologies, in: G. Teauro, D.S. Tourtzky, T.K. Leen (Eds.), Advances in Neural Information Processing Systems, Vol. 7, MIT Press, Cambridge, MA, 1995.
[15] S.L. Hung, H. Adeli, Object-oriented backpropagation and its application to structural design, Neurocomputing 6 (1) (1994) 45–55.
[16] R. Kohavi, G. John, R. Long, D. Manley, K. Pfleger, MLC++: a machine learning library in C++. Tools with Artificial Intelligence '94, IEEE Computer Society Press, Silver Spring, MD, 1994, pp. 740–743.
[17] R. Kohavi, D. Sommerfeld, J. Dougherty, Data mining using MLC++: a machine learning library in C++. Tools with Artificial Intelligence '96, IEEE Computer Society Press, Silver Spring, MD, 1996, pp. 234–245.
[18] T. Kohonen, The self-organizing map, Neurocomputing 21 (1998) 1–6.
[19] M. Kunze, J. Steffens, The neural network objects, Proceedings of the Fifth AIHENP Workshop, Lausanne, World Scientific, Singapore, 1996.
[20] J.F. Leber, G.S. Moschytz, An acoustical signal signal recognizer implemented on a novel interactive object-oriented neural network simulator, in: I. Alexander, J. Taylor (Eds.), Artificial Neural Networks, Vol. 2, Elsevier, Amsterdam, The Netherlands, 1992, pp. 1291–1294.
[21] R. Leighton, A. Wieland. The Aspirine/MIGRAINES Software Tool User's Manual Release V4.0, 1991.
[22] T. Masters, Practical Neural Network Recipes in C++, Academic Press, Boston, 1996.
[23] F. Masulli, G. Valentini, Comparing decomposition methods for classification, in: R.J. Howlett, L.C. Jain (Eds.), KES'2000, Fourth International Conference on Knowledge-Based Intelligent Engineering Systems & Allied Technologies, Piscataway, NJ, IEEE Press, New York, 2000, pp. 788–791.
[24] F. Masulli, G. Valentini, Effectiveness of error correcting output codes in multiclass learning problems, in: J. Kittler, F. Roli (Eds.), Lecture Notes in Computer Science, Vol. 1857, Springer, Berlin, Heidelberg, 2000, pp. 107–116.
[25] F. Masulli, G. Valentini, Parallel non linear dichotomizers. IJCNN2000, The IEEE-INNS-ENNS International Joint Conference on Neural Networks, Vol. 2, Como, Italy, 2000, pp. 29–33.
[26] F. Masulli, G. Valentini, Dependence among codeword bits errors in ECOC learning machines: an experimental analysis, in: J. Kittler, F. Roli (Eds.), Lecture Notes in Computer Science, vol. 2096, Springer, Berlin, 2001, pp. 158–167.
[27] F. Masulli, G. Valentini, Quantitative evaluation of dependence among outputs in ECOC classifiers using mutual information based measures, in: Proceedings of the International Joint Conference on Neural Networks, IJCNN'01, vol. 2, Washington, DC, USA, 2001.
[28] E. Mayoraz, M. Moreira, On the decomposition of polychotomies into dichotomies, The XIV International Conference on Machine Learning, Nashville, TN, July 1997, pp. 219–226.

[29] C.J. Merz, P.M. Murphy, UCI repository of machine learning databases, 1998, www.ics.uci.edu/mlearn/MLRepository.html.

[30] M. Moreira, E. Mayoraz, Improved pairwise coupling classifiers with correcting classifiers, in: C. Nedellec, C. Rouveirol (Eds.), Lecture Notes in Artificial Intelligence, Vol. 1398, Berlin, Heidelberg, New York, 1998, pp. 160–171.

[31] S. Naeher, LEDA: a library of efficient data types and algorithms, 1998, http://www.mpi-sb.mpg.de/LEDA/leda.html.

[32] M. Pardo, G. Sberveglieri, F. Masulli, G. Valentini, Decompositive classification models for electronic noses, Anal. Chim. Acta 2001, in press.

[33] V.B. Rao, H.V. Rao, C++ Neural Networks and Fuzzy Logic, M& T Books, New York, 1993.

[34] D.E. Rumelhart, G.E. Hinton, R.J. Williams, Learning internal representations by error propagation, in: D.E. Rumelhart, J.L. McClelland (Eds.), Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1, MIT Press, Cambridge, MA, 1986 (Chapter 8).

[35] M. Stone, Cross-validatory choice and assessment of statistical prediction, J. Roy. Statist. Soc. 36 (1) (1974) 111–147.

[36] B. Stroustrup, The Design and Evolution of C++, Addison-Wesley, Reading, MA, 1994.

[37] B. Stroustrup, The C++ Programming Language, Addison-Wesley, Reading, MA, 1997.

[38] G. Valentini, Classification of human malignancies by machine learning methods using DNA microarray gene expression data, in: G.M. Papadourakis (Ed.), Proceedings of the Fourth International Conference on Neural Networks and Expert Systems in Medicine and HealthCare, Technological Institute of Crete, Milos island, Greece, 2001, pp. 399–405.

[39] T.P. Vogl, T.P. Mangis, A.K. Rigler, W.T. Zink, D.L. Alkon, Accelerating the convergence of the back propagation method, Biol. Cybernet. 59 (1988) 257–263.

[40] M. Watson, C++ Power Paradigms, McGraw-Hill, New York, 1995.

[41] S.T. Welstead, Neural Network and Fuzzy Logic Applications in C/C++, Wiley, New York, 1994.

[42] R. Wunderling, M. Zockler, DOC++: a documentation system for C++ and Java, 1998, http://www.zib.de/Visual/software/doc++/index.html.

[43] A. Zell, et al., SNNS: Stuttgart Neural Network Simulator. User manual, version 4.2. Technical report, Institute for Parallel and Distributed High Performance Systems, University of Stuttgart, 1998.

Giorgio Valentini received from the Università degli Studi of Genova, Italy, the "laurea" in Biological Science (1981) and the "laurea" in Computer Science (1999), both with *summa cum laude*. He is Ph.D. student at DISI, Department of Computer Science, University of Genova (Italy) and research assistant of the National Institute for Physics of Matter (INFM). His research interests include pattern recognition, neural networks, bioinformatics, and software engineering.

Francesco Masulli is an Assistant Professor at the University of Genova (Italy) and an Associate Researcher of the National Institute for Physics of Matter (INFM). He has been a visiting researcher at the University of Nijmegen and at the International Computer Science Institute in Berkeley (California). His main research interests are Neural Networks and Fuzzy Systems. He is a co-editor of the books "Neural Networks in Biomedicine" and "New Trends in Fuzzy Logic", and serves as an Associate Editor of "KES, International Journal of Knowledge-Based Intelligent Engineering Systems" and of the international journal "Intelligent Automation and Soft Computing". He was the Chair of SOCO'99, Third International ICSC Symposium on Soft Computing (Genova, 1999). He is a Board Member of the International Graphomomics Society (IGS), of the Italian Neural Network Society (SIREN), of the SIG Italy of the International Neural Network Society (INNS), a member of the Academic Advisory Board of the ICSC (International Computer Science Conventions), and a member of IEEE, IAPR and SIREN.