NEURAL WORKBENCH: AN OBJECT–ORIENTED NEURAL NETWORK SIMULATOR

by
Athanasios Margaris, Efthymios Kotsialos, Athanasios Styliadis, Manos Roumeliotis

Abstract: The aim of this paper is the presentation of a neural network simulator, the “Neural Workbench ”.The structure of this simulator is based on the principles of Object –Oriented Design. This facilitates the implementation of complicated neural network structures that can be used to address a variety of problems and applications. In addition to the description of the simulator structure, specific task screen-shots of the running application are presented, and typical network paradigms and examples are studied.

Keywords: Neural Networks, Object-Oriented Programming, Simulators, Software Engineering

1. INTRODUCTION

The development of neural network simulators has received great attention in the last few years. Many such systems are commercial applications, but a few of them are available in the research community free of charge. The design of such programs follows many programming approaches and dominant among them are the Procedural Design and the Object –Oriented Design. The common feature of these design types is the modeling of a large number of network characteristics, such as the description of processing elements, the network topology and dynamics, and the weight adaptation rules. A comparison between the most well known freely obtainable neural network simulators can be found in (Dengel and Lutzy, 1993). The main problem associated with most of the above network simulators is their strong dependency to a specific problem. This fact means that, in general, each network model is dedicated to a specific task, and if a new requirement becomes necessary, many things have to be redefined and many procedures have to be reimplemented (Gegout, et al., 1994). In other words, the existing simulators are not flexible and they are characterized by the absence of design properties such as efficiency, extendibility, and maintainability (Giles, et al., 1996). The remarks above justify the claim that a neural network simulator has to be designed in such a way that becomes independent of any specific problem. In order to simulate a network with this characteristic, the use of object – oriented approach is recommended. The reason for this choice is the fact that this approach is characterized by the most important properties of procedural programming such as the reusability of code, and furthermore provides the user with a lot of other characteristics such as information hiding, polymorphism and inheritance (Riel, 1996).This last feature allows the construction of a neural network tree, with an abstract network type as root, and specialized types of networks as leafs (Shikuta, 1995).

2. REVIEW OF PREVIOUS WORK

The design and implementation of neural network simulators gave rise to a large variety of such programs; each one has its own features and addresses a specific class of problems. The most important of these simulators are the PlaNet Simulator from the University of Colorado at Boulder, USA (Miyata, 1989), the Rochester Connectionist Simulator (RCS) from the University of Rochester, UK (Goddard et al., 1989), the Pygmalion Simulator from the Computer Science Department of the University College at London, UK (Hewetson, 1990), and the Stuttgart Neural Network Simulator (SNNS) from the University of Stuttgart, Germany (Zell et al., 1995).Besides these applications, there are many other simulators available as commercial or free programs, such as the NSK (Neural Simulator Kernel) (Gegout, et al., 1994), the PDP++ (Dawson, et al., 2003), and the SPRLIB/ANNLIB (Statistical Pattern Recognition and Artificial Neural Network Library) (Hoekstra, et al., 1998).This last one has a low – level programming interface in C that supports the easy construction and simulation of pattern classifiers, as well as the simulation of an extensive list of network models and learning rules. In the following subsections, we present a brief description of the most significant, according to our experience, neural network simulation tools, among those mentioned above.

2.1 PlaNet Neural Network Simulator

The PlaNet System (at the time of this writing in version 5.6), is a tool for constructing, running and examining neural network structures. PlaNet has previously been known as SunNet. The most significant aspect of PlaNet is that it allows the user to deal with a network at a fairly high level of conceptualization, and yet provides the flexibility of constructing networks of almost arbitrary structures and size, and to "run" the network in many different ways. The user defines the network by specifying layers of units and connections between layers. In the next step, the user can program the network by defining procedures that specify the way it should be activated. The network specification language of PlaNet is general enough to allow many different types of networks to be constructed. It also includes high level routines for various neural networks tasks, based on the backpropagation learning algorithm. Another important aspect of PlaNet is that it allows the examination of the neural network state through graphical display of various network characteristics, such as activation patterns or weight matrices in the connections. We can use this to plot the learning curve (error or other network states as a function of learning cycles) in a graph. PlaNet is an environment with which the user can interact by giving commands that read in network specifications and input/target patterns, run and train the network using user-defined procedures, or display the state of the network in various ways. The modification of the network parameters that affect the nature of the network or affect the interface of PlaNet can be effected through an Options mechanism.

2.2 Rochester Connectionist Simulator (RCS)

RCS is a flexible and powerful tool for simulating networks of highly interconnected information processing units. This is a unix application, written in C, and its current version is 4.2. In the RCS application, a connectionist network consists of simple computational elements which communicate by sending their level of activation via links to other elements. The units have a small number of states, and compute simple functions of their inputs. Associated with each link is a weight, indicating the significance of activation arriving over that link. The behavior of the network is determined by the pattern of connections, the weights associated with the links, and the unit functions. The Rochester Connectionist Simulator supports construction and simulation of a wide variety of networks, the most significant ones being the backpropagation networks. It is characterized by the existence of a graphical user interface (GUI) that allows the construction and training setup of neural network structures in an easy and convenient way. Finally, the simulator kernel can be embedded in other programs or used as a separate procedure, a fact that makes possible the integration of the neural network technology in a great variety of applications.

2.3 The Pygmalion Neural Network Simulator

The aim of Pygmalion NNS is to provide an open programming environment that can be easily extended and interface with other tools. For this reason the core of the environment is the platform of X-Windows and the programming languages C and C++. The basic architecture of this application consists of five major parts:

- an X –graphics interface, for controlling the execution and monitoring of a neural network application simulation.
- an algorithm library that allows the implementation of common neural networks such as the backpropagation network, the Hopfield net, the Kohonen Self –Organizing Map (SOM), and Boltzman machines.
- the high level languages N and nC that are based on C++and are used to define a neural network architecture, by describing the network topology and its dynamics.

- an intermediate language, nc-Code, that serves as a low–level, machine–independent network specification language for representing the partially or fully trained neural net- works.
- various compilers for the target UNIX-based workstations and parallel transputer-based machines.

The Pygmalion environment has been implemented as a part of an ESPRIT-II project in order to facilitate the implementation of key real-world applications, such as image and speech processing.

2.4 SNNS (Stuttgart Neural Network Simulator)

The SNNS application is a software simulator for neural networks on Unix workstations - a Windows version is also available - and its current version is 4.2.The goal of the SNNS project is to create an efficient and flexible simulation environment for research and applications of neural networks. The SNNS simulator consists of two main components, namely, a simulator kernel written in C, and a graphical user interface under X11Rx.The simulator kernel manipulates the internal data structures of the neural networks and performs all operations of learning and recall. It can also be used without the other modules comprising the system, as a C program embedded in custom applications. SNNS is extensible with user –defined activation functions, output functions, and learning procedures. Those can be written as simple C programs and linked to the simulator kernel. SNNS supports many network architectures and learning procedures, such as the backpropagation, the counter –propagation, the QuickProp, ART1 and ART2, among others. Additional network architectures such as the Dynamic LVQ, the Self Organizing Maps (SOM) and the Time Delay Neural Networks (TDNN)are also available. The graphical user interface X –GUI is built on top of the kernel and gives a 2D and 3D graphical representation of the neural networks. It controls the kernel during the simulation run. In addition, the 2D user interface has an integrated network editor which can be used for direct creation, manipulation and visualization of neural networks in various ways.

3. INTRODUCTION TO NEURAL WORKBENCH

The main reason for the implementation of the Neural Workbench, abbreviated to NW in most places from now on, is the support of neural network facilities not provided by the existing neural network simulators. Such facilities are the concatenation of “small” networks in order to create a large one (the import facility, as well as the inversion of the neural network at hand in order to create its mirror version. Furthermore, NW supports some other specialized functions, such as the recording of the absolute minimum training set error during the backpropagation training. Generally speaking, the implementation of a custom neural network simulator is faced with arbitrary variations regarding network design and training methodology. The programmer knows its structure, and can modify it to cover any new requirements. On the other hand, the disadvantage of using an existing application is the limitation of the user to those capabilities that are provided by the application. Even when the source code of the simulator is available—a fact that is especially true for the majority of Unix-based simulators used in the academic community—the modification and the enhancement of this code is a non –trivial task. For this reason, scientists and practitioners in the neural network field more often than not write their own simulators. The most important characteristic of NW is the dynamical structure of the implemented neural networks. These networks can be constructed easily, using the mouse to add layers, neurons and synapses, as well as to define their properties. The flexibility that characterizes the network structure is based on the fact that this structure is implemented as a multi-layered linked list, i.e. a group of nested linked lists, each node of which contains a whole list structure. More specifically, the kernel of the current implementation of NW is based on a template class, named TList (Adams, et al., 1995), that implements the single linked list. Using this class, the construction of a single linked list of objects of type T is possible. The most important part of the definition of this class is showed in Listing 1.

```
template <class T >class TList {
private :
    struct Node {
        Node *Next;
        T*DataVal;
    }*First, *Last;
    int NodeNumber;
public :
    Insert (T*Item, int Position);
    Delete (int Position);
    Search (T *Item)const;
}//TList
```

Listing 1
TList class definition

Based on the TList class template described above, we can define all other classes that the neural network simulator contains. These classes model all the simulator parts, such as the single neuron processing element, the layer of neurons, the whole neural network, as well as the synapse between neurons, the bias unit, and the training set class. A short description of all these classes is given below, followed by the presentation of the main dialogs that allow the interaction between the user and the neural network simulator.

3.1 The TNeuron class

The most important part of the TNeuron class definition is shown in Listing 2. For each neuron we store its basic parameters, i.e. the threshold, the output value and the type of activation function it uses. Other parameters that are not shown here are associated with the various training algorithms in which a neuron participates. Further-more, since a neuron belongs to a specific layer and it is connected with other neurons through synapses; two single linked lists are maintained for each neuron. The first list, named OutLinks, stores the synapses to which the neuron under consideration is the source neuron, while the second list, named InLinks, stores the synapses to which the neuron under consideration is the target neuron:

```
class TNeuron {
private:
    int NeuronId;
    int LayerId;
    double Threshold;
    double Output;
    int Type;
    int FunctionType;
public :
    TList <TSynapse >InLinks;
    TList <TSynapse >OutLinks;
};//TNeuron
```

Listing 2
TNeuron class definition

Besides the basic parameters that are shown in Listing 2, there are many other parameters that are defined and maintained for each neuron processing element. These parameters are mainly associated with the various algorithms that can be used for the neural network training —the learning rate, the sigmoidal slope and the momentum for the back propagation algorithm, to name a few. An interesting property of NW is that each neuron can be assigned its own parameter values, allowing thus the assignment of different behavior per network neurons; even through they belong to the same layer. We can use this as an elementary modeling technique for the concept of *diversification*.

As a last interesting feature of the applications associated with the network neurons, the use of functional link neurons is considered. These neurons belong exclusively to the input network layer; they are not fed with training set samples as the normal neurons do, but their output is a function of the output of the remaining input layer neurons. More specifically, if x is the output of some input neuron, then the output of a functional link neuron, can have the forms sin(kBx), cos(kBx), xsin(kBx) or xcos(kBx), where the parameter k assumes integer values. For a detailed description of functional link neurons see (Pao, 1989).

3.2 The TLayer class

Each neural network layer is defined as a linked list of TNeuron objects. A sort section of the TLayer class definition is shown in Listing 3. This class provides the most important operations associated with each layer such as the insertion, deletion, and search for neurons. These functions are based on the corresponding functions of the. TList template -Insert, Delete and Search; this holds for every class based on that template.

```
class TLayer {
private :
     int LayerId;
public :
     TList <TNeuron >Neurons;
     InsertNeuron (TNeuron *N);
     DeleteNeuron (int Pos);
     CopyNeuron (int old, int new);
     MoveNeuron (int old, int new);
     FindNeuron (int Pos);
};//TLayer
```

Listing 3
TLayer class definition

The neural network layers as they are represented by the TLayer objects are characterized by their own icons in the running NW application. If the user displays the layer property sheet, by double clicking on the icon of some layer, the properties that are set via this dialog are applied to the group of neurons belonging to that layer.

3.3 The TNetwork class

A neural network is defined as a linked list of TLayer objects. Listing 4 shows a part of the TNetwork class definition. Since each neural network is a linked list of TLayers, and each one of them is a linked list of TNeurons, it is possible to insert, delete, copy and move layers, as well as neurons between these layers. Synapses can also be created between these neurons, regardless of their positions in the network structure, a fact that makes possible the creation of feedforward as well as of recurrent networks. There are also two additional classes that are used in the TNetwork class definition: the TBias class that simulates the bias unit, and the TSet class that describes a training set. There is a single linked list of such TSet objects, meaning that more than one training sets can be attached and used during training of each neural network. The TNetwork class is the most complicated class of Neural Workbench, since it includes all the properties that can be assigned to each TNetwork object. Besides the attributes and the functions shown in Listing 4, there are many other members of that class. Due to the aims of this short exposition those can not be presented analytically here. The most important of them are appropriate functions that are used in order to save and load the network to and from a hard disk file, and the procedures Import and Inverse that allow the concatenation of small networks to a larger one and construction of the mirror network.

```
class TNetwork {
    int NetworkId;
public :
    TBias *biasUnit;
    TList <TSet >setList;
    TList <TLayer >Layers;
    AddLayer (TLayer *L, int Pos);
    DeleteLayer (int Pos);
    CopyLayer (int old, int new);
    MoveLayer (int old, int new);
    AddNeuron (TNeuron *N,
                int LPos, int NPos);
    DeleteNeuron (int LPos, int NPos);
    AddSynapse (TNeuron *Source,
                TNeuron *Target);
    DeleteSynapse (TNeuron *Source,
                TNeuron *Target);
    AddTrainingSet (TSet *S);
    InsertBias ();
    RemoveBias ();
    ...
};//TNetwork
```

Listing 4
TNetwork class definition

```
class TSynapse {
private :
    double Input;
    double Weight;
public :
    TNeuron *Source;
    TNeuron *Target;
};//TSynapse
```

Listing 5
TSynapse class definition

3.4 The TSynapse class

The TSynapse class simulates the behavior of a synapse between two neurons. Each synapse is characterized by an input and a weight value for that input, and in order to be described completely, one has to determine the source as well as the target neuron for that synapse. A part of the definition of the TSynapse class is shown in Listing 5. Each synapse can be enabled or disabled during simulation, and its weight can be fixed, varied, or conditionally fixed. In the last case, the synapse weight is fixed when a predefined condition is satisfied during simulation.

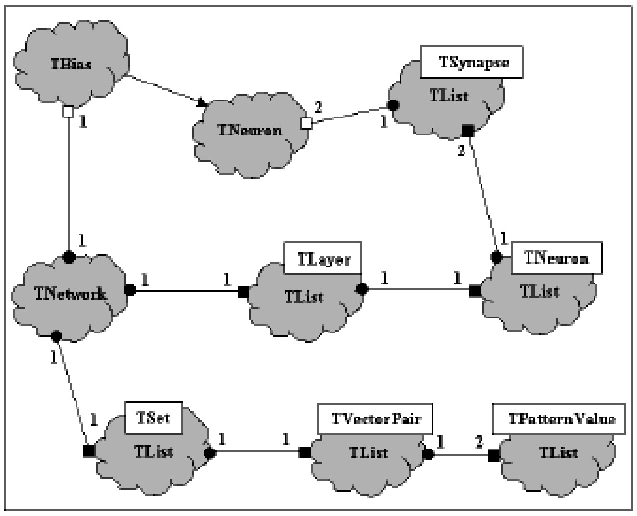


Fig. 1. Class Diagram of the Neural Workbench Simulator

3.5 The TBias class

The TBias class simulates the behavior of the bias unit which is connected to a single neuron in order to provide for it a variable threshold. Since the bias is actually a special type of neuron, it is de- rived from the TNeuron class through inheritance. So, the definition of the TBias class, not shown in Listing format here, has the form

class TBias :public TNeuron {..... }

4. THE TRAINING SET STRUCTURE

The TList class template is also used for the representation of the training set. Each training set is defined as a linked list of TVectorPair objects. The TVectorPair object is composed from two other linked lists, one for input values and one for the corresponding output values. In the current implementation these atomic values can be of integer or double data type and they are represented as objects of another class called TPatternValue. This dynamic structure of the training set, allows the insertion and deletion of training patterns, as well as the variation of the number of inputs and outputs for each pattern. Figure (1) shows the class diagram of the neural simulator structure, in Booch notation (Booch, 1994).

5. USING SIMULATOR COMPONENTS

In order to access the components of the neural networks created through Neural Workbench, the overloaded operator [·] of the class TList is used;

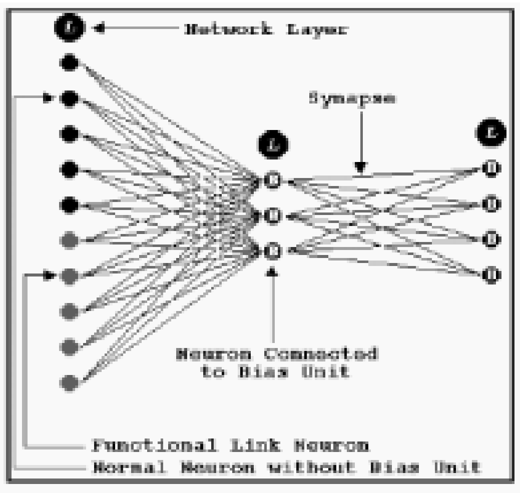

Network Layer
Synapse
Neuron Connected to Bias Unit
Functional Link Neuron
Normal Neuron without Bias Unit

Fig. 2. A typical feed forward neural network structure created in the Neural Workbench Simulator

this returns a specific node of the list. So, in order to get the i th layer of a TNetwork object

```
tNet 6Layers [i ]
```

is used. The j th neuron of that layer is returned by

```
tNet 6Layers [i ] 6Neurons [j ].
```

In order to retrieve the weight of the k th input synapse of that neuron, we write

```
tNet 6Layers [i ] 6Neurons [j ] 6\
InLinks [k ] 6GetWeight(),
```

and so on.

6. CREATING AND MANIPULATING NEURAL NETWORKS

The creation and manipulation of neural networks with NW can be performed using the key- board and the mouse. The program runs under the graphical environment of Win32 platforms (a linux port is under development), and the con- struction of networks can be done via a network editor which has been designed for that purpose. A typical neural network structure implemented via NW is shown in fig.(2). The neural network shown there is a feedforward one, meaning that the net synapses are directed from the input layer to the output layer. However, in general, the program allows the association of two neurons that can be located anywhere in the network structure. Figure (3) shows various connection types between neurons in a neural network.

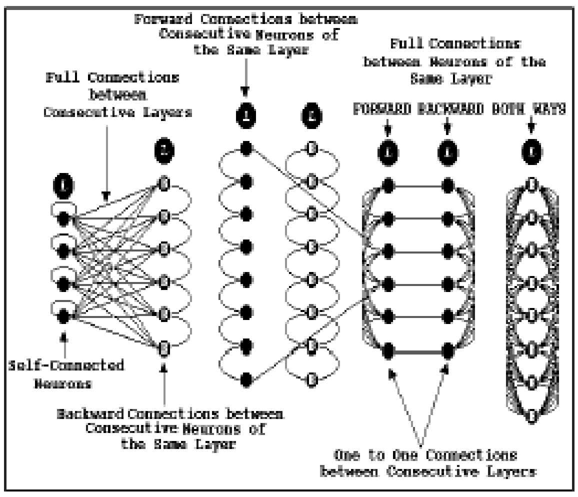
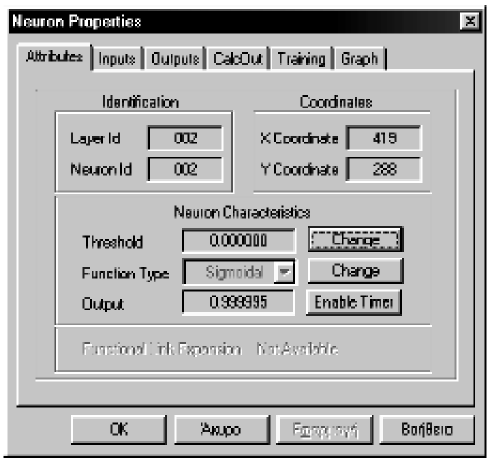

Forward Connections between Consecutive Neurons of the Same Layer
Full Connections between Neurons of the Same Layer
Full Connections between Consecutive Layers
FORWARD BACKWARD BOTH WAYS
Self-Connected Neurons
Backward Connections between Consecutive Neurons of the Same Layer
One to One Connections between Consecutive Layers

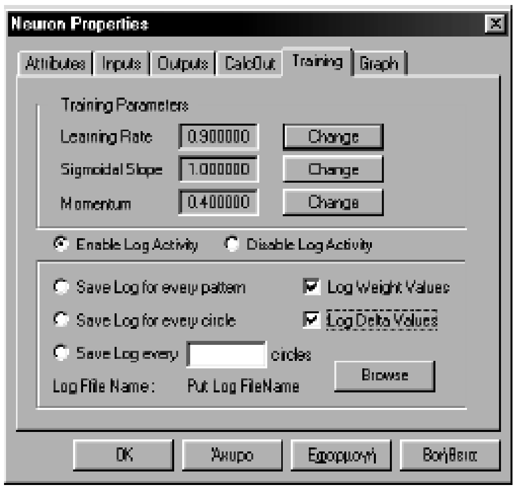
Fig. 3. Various connection types between neurons of a neural network

These types include the connection of a neuron to itself (self-connected neurons) and the connections of consecutive neurons that belong to the same layer in both forward and backward directions. This last connection type allows the construction of Hopfield –type neural networks. Regarding the connections between two layers, these can be full connections –in the sense that all the neurons of the source layer are connected to all neurons of the target layer –or one-to-one, if the two layers are characterized by the same number of neurons. Finally, a bias unit can be connected to each neuron that provides it with a variable threshold. The neurons that are connected to the bias unit contain the letter B in the bitmap that represents them in those figures. The assignment of properties to each network element can be performed using a property sheet that can be displayed by left –clicking on them. There are different types of such sheets that can be used to configure the various network elements such as the layers, the neurons and the synapses of the network. If a property value is determined for a layer, then this value is assigned to all the neurons that belong to that layer. Neural Workbench, however, allows the assignment of different values to the neurons of the same layer, using the neuron property sheet; a part of it is shown in figures (4) and (5). Using the property page of the figure (4), the user can determine the threshold value and the function type of the selected neurons, while the property page of figure (5) allows the determination of important neuron properties such as the learning rate, the sigmoidal slope and the momentum, as well as the activation (or deactivation) of the log procedure, that allows the recording of various neuron parameters during the training operation. Neural Workbench contains similar property dialogs that allow the configuration of the network layers details as well as the synapses of the current neural network structure.



Layer Id 002
Neuron Id 002
X Coordinate 419
Y Coordinate 288
Threshold 0.000000
Function Type Sigmoidal
Output 0.999995
Functional Link Expansion Not Available

Fig. 4. The *Attributes* page of the property sheet for the current neuron.



Training Parameters
Learning Rate: 0.900000
Sigmoidal Slope: 1.000000
Momentum: 0.400000

[✓ Enable Log Activity] Disable Log Activity

Save Log for every pattern
Save Log for every circle
Save Log every ____ circles
[✓ Log Weight Values]
[✓ Log Delta Values]
Log File Name: Put Log FileName

Fig. 5. The *Training* page of the property sheet for the current neuron.

7. TRAINING SET MANIPULATION

Neural Workbench provides an advanced interface for the creation and manipulation of training set structures. As it has been already mentioned above, the current neural network can be associated with more than one training set at the same time. These sets must be compatible with the network structure; each one of them can be used during the training phase. The software component which is responsible for the interaction between the user and the training sets of the network is shown in fig.(6). From this figure, it is clear that the user can modify the contents of the training set by inserting and deleting data values, as well as its structure, by varying the number of inputs and the corresponding desired outputs of the current training set.

8. NEURAL NETWORK TRAINING AND RECALL

The current NW version supports the most commonly used learning algorithms, namely the back – propagation algorithm and the Kohonen self – organizing maps (SOM). These algorithms have been implemented as separate threads, a fact that makes possible the interaction between the user and the neural network, during training. Due to page limitations, in the next paragraphs the back propagation interface is going to be described in short. The main window of the back propagation algorithm is shown in figure 7. This window includes the graph of the global error as a function of the iteration cycle and it also displays the current minimum and maximum values of that error. An interesting feature of this dialog is the scaling of the error graph, if the new error value does not belong to the interval defined by the current minimum and maximum error values. In this case the whole curve is rescaled in order to fit to the plot area of the back propagation window. The operation of the back propagation algorithm is controlled by means of many child windows with the most important of them to be the control panel dialog, the training parameters dialog, and the back propagation results dialog. The control panel is used to start, suspend, continue, and abort the neural network simulation. The training parameters window allows the variation of the back propagation parameters such that the learning rate, the sigmoidal slope, and the momentum.

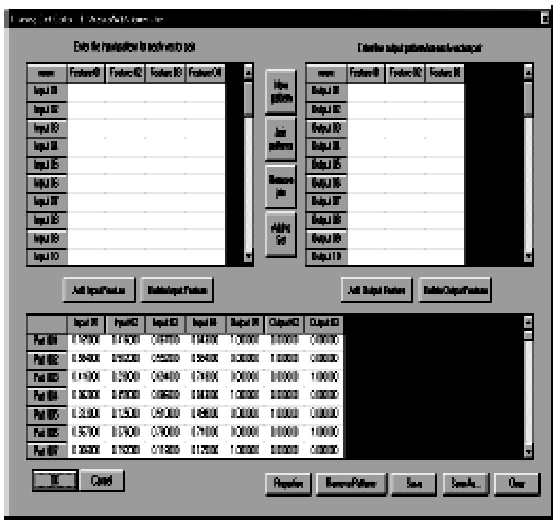


Fig. 6. The training set editor that allows the creation and the manipulation of training set structures.

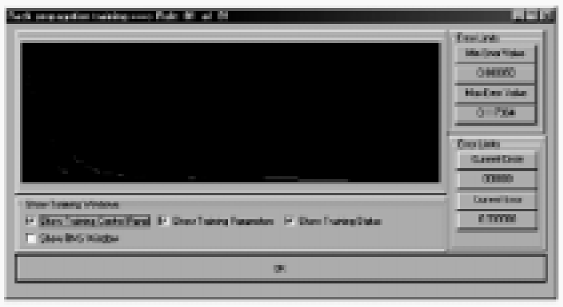
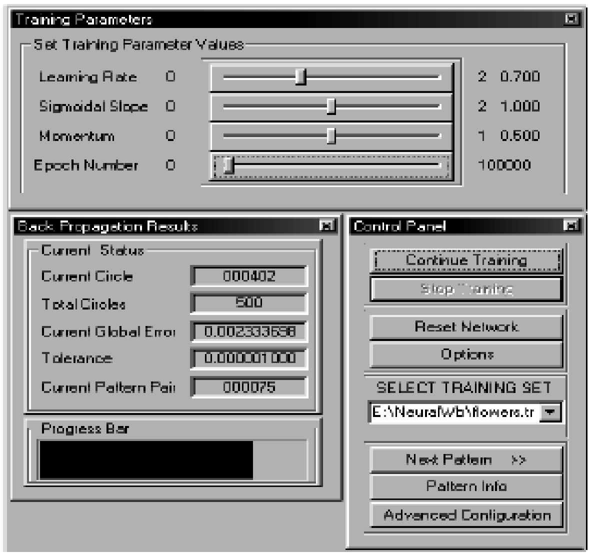


Fig. 7. The back propagation main window.



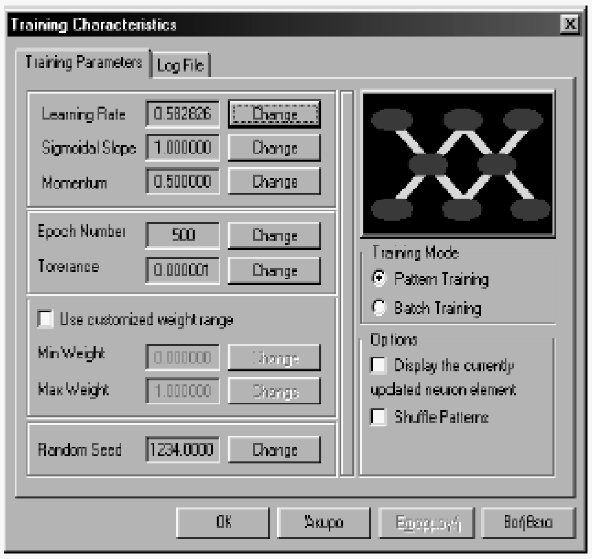
Learning Rate	0	2	0.700
Sigmoidal Slope	0	2	1.000
Momentum	0	1	0.500
Epoch Number	0		100000

Current Circle	000402
Total Circles	500
Current Global Error	0.002333638
Tolerance	0.000001000
Current Pattern Pair	000075

E:\NeuralWb\flowers.tr

Fig. 8. The back propagation child windows.

Since the algorithm is running in a multi-thread environment the variation of these parameters at run time and the study of the consequences of this variation is possible. Finally, the back propagation results dialog displays for each simulation cycle, the values of important parameters such as the current cycle and pattern, the current global error value and the tolerance value that indicates the level of simulation accuracy that has to be reached by the neural network after the training operation. Another important dialog that controls the operation of the back propagation algorithm is the Options property sheet that allows the configuration of the back propagation parameters -the corresponding property page is shown in figure 9 -and the creation of the log data files. The parameter values that are set via this dialog, are associated with the learning rate, the sigmoidal slope and the momentum, as well as the number of iterations, the tolerance and the learning mode (pattern shuffling or not). Regarding the creation of the log files, the user has the ability to determine the parameters that have to be logged during training. These parameters include the neuron delta values, the synapse weight values, the neuron output values, and the current global error (which is the default option). Except from the parameter value to be recorded, the user can also specify the frequency of this recording, which can take place every pattern, every N training cycles, or only once, after the end of the training operation.



Learning Rate 0.582826
Sigmoidal Slope 1.000000
Momentum 0.500000
Epoch Number 500
Tolerance 0.000001
☐ Use customized weight range
Min Weight 0.000000
Max Weight 1.000000
Random Seed 1234.0000
Training Mode
[✓ Pattern Training]
Batch Training
Options
☐ Display the currently updated neuron element
☐ Shuffle Patterns

Fig. 9. The back propagation Options Dialog.

After the termination of the training operation, the trained neural network can be used for the recall process. In this process, the user supplies to the network input data values that may belong to the training set or not, and the network tries to es timate the correct output based on the knowledge obtained during training. In Neural Workbench, the recall operation is performed via the dialog shown in figure 10.

9. CONCLUSIONS AND FUTURE WORK

Neural Workbench is a multi-threaded neural network simulator that allows the construction and training of neural network structures. The architecture of this application follows the principles of the object-oriented design, and therefore, its implementation is based on the powerful characteristics of this approach such as the polymorphism and class inheritance. According to this architecture, a neural network is defined as a single linked list of TLayer objects (representing the network layers) each one of them contains a single linked list of TNeuron objects (representing the processing element known as neuron). Each network neuron can be either a source or a target neuron for a synapse (which is represented by a TSynapse object), and for this reason it includes two linked lists of synapses: a list of synapses to which the current neuron is the source neuron, and another list of synapses, to which the current neuron is the target neuron. From this description it is clear that each synapse is stored twice in the neural network structure: one time to the synapse list of the source neuron, and another time to the synapse list of the target neuron. This double storage of each synapse leads to the following disadvantages: (a) the programming of the neural network is more complicated since every time that the characteristics of a synapse are modified, the characteristics of its conjugate synapse have to be modified, too and (b) the neural network data file demands mope space in the memory and in the secondary storage, since the synapse information is stored twice. However these disadvantages are considered unimportant, since today, computers are characterized by very fast processors and very large (and cheap) storage devices. On the other hand, this design allows the creation of synapses between any processing elements of the neural network, regardless of its position inside the network structure. The neural network simulator described in previous sections, has been used in various problems and it worked fine in the case of backpropagation, counterpropagation and Kohonen training (Freeman and Skapura, 1991). One of its most important applications was the implementation and training of neural models for chaotic maps, such as the logistic equation. Besides the characteristics described in the previous sections, the simulator o fers much more features, such as the use of variable learning rate and the plotting of many network features vs. time. Future work on this project includes the re-programming of the simulator from the beginning in order to support more features such as the creation of more than one neural network at the same time that can be connected together and work in parallel -the current version of the application allows the usage of only one network. The design of the whole simulator is going to be modified, too, in such a way that each network type will be produced by an abstract network class, through inheritance. The result of this approach is a hierarchical neural network tree, with an abstract net class as root, and specialized network types as nodes and leafs. Finally, the graphical user interface is going to be enhanced by implementing useful functions that will allow the interaction between the user and the application in a more convenient way.

References

[1].Joel Adams, Sanford Leestma, Larry Nyho . (1995).C++, An Introduction to Computing , First Edition, Prentice-Hall Inc, ISBN 0-02- 369402-5.

[2].Grady Booch (1994).Object-oriented Analysis and Design with Applications , Second Edntion, The Benjamin/Cummings Publishing Com- pany Inc, ISBN 0-8053-5340-2.

[3].Chadley K.Dawson, Randall C.O ’Reilly, James McClelland (2003).The PDP++Software Users Manual , Carnegie Mellon University.

[4].Andreas Dengel and Ottmar Lutzy (1993).A Comparison of Neural Net Simulators , IEEE Expert, Volume 8, Number 4, pp.43-51.

[5].J.Freeman, D.Skapura (1991).Neural Net- works, Algorithms, Applications and Program- ming Techniques , Addison Wesley Publishing Company.

[6].Cedric Gegout, Bernard Girau, Fabrice Rossi (1994).NSK, an Object-Oriented Simulator Kernel for Arbitrary Feedforward Neural Net- works in Proceedings

of IEEE International Conference on Tools with Arti .cial Intelligence, New Orleans, pp.93-104.

[7].C.Lee Giles, Steve Lawrence and Ah Chung Tsoi (1996).Correctness, E .ciency, Extendability and Maintainability in Neural Networks Simu- lation , International Conference on Neural Net- works, ICNN 96, Washington DC, IEEE press, pp.474-479.

[8].Nigel H. Goddard, KentonJ.Lynne, TobyMintz, Liudvikas Bukys (1989).Rochester Connection- ist Simulator , The University of Rochester, Computer Science Department, Rochester, New York, Technical Report 1989.

[9].Mike Hewetson (1990).Pygmalion Neural Net- work Programming Environment:Reference Manual , ESPRIT Project 2059.

[10].A.Hoekstra, M.A.Kraaijveld, D.de Ridder, W.F.Schmidt, A.Ypma (1998).The Complete SPRLIB &ANNLIB Version 3.1 User ’s Guide and Reference Mannual , Delft University of Technology, Faculty of Applied Physics, Pattern Recognition Group.

[11].Yoshiro Miyata (1989).A User Guide to PlaNet Version 5.6 , University of Colorado, Boulder, Computer Science Department.

[12].Pao Y (1989).Adaptive Pattern Recognition and Neural Networks , Addison Wesley Publishing Company.

[13].Arthur Riel (1996).Object-Oriented Design Heuristics , Addison-Wesley Publishing Com- pany Inc, ISBN 0-201-63385-X.

[14].Erich Schikuta (1995).A Software Engineering Approach to Neural Network Speci .cation , in Proc.of 3 rd Annual SSN Symposium on Neural Networks:Arti .cial Intelligence and Industrial Applications, pp.381-384, Nijmegen, Nether- lands, Springer-Verlag, Berlin.

[15].Andreas Zell et al (1995).SNNS Version 4.2 User Manual , University of Stuttgart, Institute for Parallel and Distributed High Performance Systems (IPVR), Applied Computer Science, Stuttgart.

Authors:

Athanasios Margaris, corresponding author, email: amarg@uom.gr
Efthymios Kotsialos, email :ekots@uom.gr
Athanasios Styliadis, email : styl@it.teithe.gr Technological Institute of Thessaloniki, Sindos GR 541 01, Hellas
Manos Roumeliotis, email: manos@uom.gr University of Macedonia Department of Applied Informatics Thessaloniki GR 540 06, Hellas 3