

NESS: A SIMULATION ENVIRONMENT FOR BEHAVIORAL DESIGN OF NEURAL NETWORKS FOR PREDICTION AND CONTROL

Cesare Alippi, Fabio Casamatta, Leonardo Furlan, Andrea Pelagotti, Vincenzo Piuri
Department of Electronics and Information
Politecnico di Milano
Milano, Italy I-20133
e-mail address: piuri@elet.polimi.it

ABSTRACT

NeSS (Neural Systems Simulator) is a flexible software environment for high level simulation, specialized to analyze and model dynamic non-linear systems for identification and control applications. The behavior of each system constituting a project is simply defined in a graphic fashion by interconnecting atomic objects (e.g., algebraic functions, neural networks). Atomic objects are generally parameterized: their optimal values can be directly fixed by the user or identified by means of suitable training procedures.

1. INTRODUCTION

The design of the control architecture for a given plant requires modeling of the whole process by exploiting physical features (and thus writing physical-driven equations) and identifying the unknown parts with suitable identification and prediction tools. The latter aspect, namely identification and prediction of some parts of the plant, can be successfully solved by relying on new information technologies, which provide useful computational paradigms such as neural networks.

Each module composing a plant can be seen as a black box characterized by its behavior, i.e., the mathematical description of its input-output relationship. Some parts can be effectively modeled by analytical equations, ranging from simple algebraic to differential non-linear equations; for other parts, it may be difficult to identify the behavior by means of traditional linear approaches. The problem can be successfully tackled by considering neural networks [1], non-linear computational paradigms able to learn the input/output relationship from pairs of (input, output) couples: therefore, the learning procedure does not require any information about the internal and physical behavior of the component.

The integrated presence of neural and algorithmic descriptions of the components of a plant makes its control design a hard task: the difficulty increases whenever hybrid controllers (neural and PID-like) are taken into account. In this case, training must be applied to set the parameters of the neural controller in a closed-loop control configuration.

To cope with such problems, dedicated CAD tools have been developed mainly dealing with classical approaches; the most sophisticated support the design of hybrid control systems with an off-line training modality [2] (a neural component is trained off-line by using learning couples measured on the plant).

In this paper we present *NeSS* (Neural System Simulator), a flexible simulation environment for identification and control of dynamic systems. The simulator is capable of supporting both neural components and classical ones with an on-line training modality.

The behavior of the system is defined graphically by interconnecting a set of atomic components (e.g., algebraic and trigonometric functions, neural networks, data files) with a graphic editor. Each component is defined by a vector Φ of parameters representing the physical insights of the element: for example, for a sinusoidal function $y = A \sin(\omega t + \phi)$, the vector contains the variables $\Phi = [A, \omega, \phi]$. Such parameters may be either fixed or time-dependent. As far as a neural component (i.e., a neural network) is concerned, an algorithm library is provided, which contains the most popular and frequently used topologies (e.g., multilayered feedforward [1], virtual neurons topology [3]); for each neural element the operator has to define the topological complexity in terms of the number of layers and neurons: the Φ vector contains weights and biases.

A configuration procedure allows to train a generic atomic component (both algorithmic and neural); this implies that parameters contained in Φ can be identified via learning: this facility is particularly relevant in several application and it is an interesting feature and extension provided by *NeSS*.

The training procedure can be dedicated and tailored to the specific application; according to this philosophy the function to be minimized and the learning algorithm can be directly constructed by the user. Anyway, to ease the task, libraries of pre-defined error functions and training procedures (which cover the most common cases) are also provided.

In section 2, the overall architecture of *NeSS* is presented while details related to the data structures and the dynamic system model are described in section 3. In section 4, we present the features provided by the graphic editor and how to design a system; simulation tools are finally described in section 5.

2. THE OVERALL ARCHITECTURE

The main guidelines followed in designing *NeSS* can be summarized as readability, efficiency, portability, and reusability in order to simplify future software modifications and extensions. For these reasons the code was written in C++ in the MS-Windows system environment.

The *NeSS* architecture is characterized by four fundamental modules: the user interface, the graphic editor, the simulation

engine and the underlying data structure. Such separation allows portability since graphical aspects are separated from mathematical and algorithmic ones: a change in the host platform only interferes with graphical primitives.

Likewise, changes in the simulation engine can be made independently from modification both of the graphic interface and of the data structure.

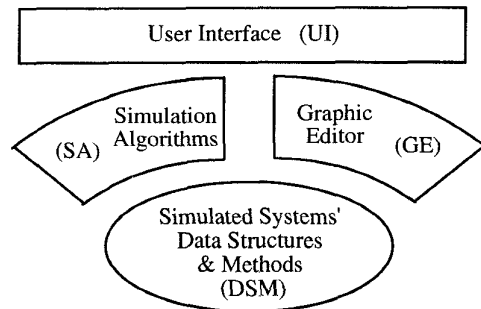


Fig. 1 - The overall architecture.

The whole environment results in the layered structure of Fig. 1. The information interchange between the user and NeSS is ruled by the user interface (UI). Such a module is designed to provide a friendly and powerful environment in which the user can easily design and interconnect systems, set up and run personalized simulation tasks.

The UI offers different pop-up menus which provide the access to all tools supplied by the graphic editor (GE) and by the simulation algorithms module (SA).

The graphic editor GE allows to design - easily and intuitively - the architecture of a generic system. In a window-based environment, the user can draw the internal architecture of the system to be modeled and simulated by placing all components (represented by blocks) and by connecting them. Utilities are provided which simplify the drawing and connecting operations as well as files management.

A simulation is executed by using the algorithms available in the SA module. This module implements all methods which are usually adopted to perform numerical analysis and learning. Simulation procedures can be tailored to the specific application by setting the configuration parameters.

Since NeSS is implemented according to the object-oriented paradigm, each system, each function and each graphical output window is an object that can be manipulated by using its own methods (i.e. the related procedures in the object oriented terminology). The module DSM groups these objects together.

3. THE SYSTEM MODEL AND THE DATA STRUCTURE

In the control theory, systems are classified in two categories: *static* and *dynamic* systems. The first ones are time-independent, i.e., their parameters are time-invariant and the behavior is described by algebraic and trigonometric equations. The second ones are time-dependent, namely, their parameters change in time according to a specific rule.

Such components can be modeled in NeSS at a high level: each of which is a black-box, having inputs, outputs and the associated transfer function. The transfer function of a component is described by using a simple language which provides elementary algebraic and trigonometric elements,

suitably composed to obtain the desired function or can be a different element such as a neural network.

The system model is hierarchically created by the user in NeSS: the system is partitioned into subsystems, connected together and towards the external world via connectors. Connectors represents the interface the primary input and output signals exchanged by the system with the external environment. The elementary operators are, for example, sums, products, logic operators and neural nets. Each subsystem is represented by a block that may contain either lower-level subsystems and elementary operators.

The representation of the dynamic system in NeSS must take into account these characteristics and must be coherent with them. A comprehensive and homogeneous internal representation must treat elementary operators, connectors and subsystems in the same way, because all of them are mathematical operators and the only relevant aspect is their input/output behavioral description.

Due to these remarks and to the intrinsic recurrence of the system model definition, the data structure representing a dynamic system can be naturally described by using an object-oriented inheritance-based class hierarchy. In particular, systems, connectors and elementary operators inherit from the same base class (generically called *TComponent*). Such a structure takes advantage from the dynamic binding mechanism which is used to call member functions by knowing only the pointers to previously-created objects.

Actually, inheritance has been exploited in NeSS in a more refined way by using *multiple inheritance* as shown in Fig. 2. Mathematical and graphical properties, which are shared by all components, are separated in two intermediate-level classes in the hierarchy (*TMComponent* and *TGComponent*). These properties are then extended and specialized for single components types by another class level in the hierarchy; the resulting classes for the mathematical properties are *TMConnector*, *TMFunction* and *TMSystem*, while the ones related to the graphic characteristics are *TGConnector*, *TGFunction* and *TGSystem*.

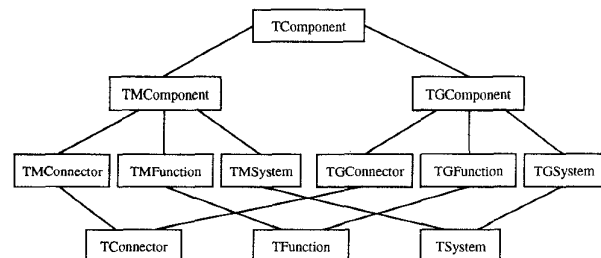


Fig. 2 - Architecture of the module DSM.

Final objects used in the system model belong to classes which inherit both from the mathematical (TM) and graphical (TG) classes; classes at the bottom of Fig. 2 are therefore in the C++ *is-a* relationship both with mathematical and graphical classes.

Graphical classes provide data structures and methods to support the graphic description of the dynamic systems, in particular for the graphic editor operations.

Mathematical classes provide data and methods to support simulation. In particular, we introduced methods to compute the output generated by a given set of input values of a component,

to evaluate the first derivative of the outputs with respect to the parameters of the operator, and to manage the component state. To each elementary operator is associated an output buffer in which the last computed vector is stored.

Evaluation of the component's output vector is performed in three steps. First of all, the input vectors are built by using the connection information and the output buffers of other operators connected to the component's inputs. Then, the output vector is computed by applying the rules associated with the component itself; finally, the resulting vector is stored in the output buffer.

For a *TSystem* object, the computation method produces, in a recursive way, a sequence of calls to the computation methods of operators and *TSystem* objects contained in the object itself. All of these calls are executed by exploiting the polymorphism mechanism.

The *TFunction* is the base class for the hierarchy of elementary operators. The user can choose among about fifty elementary analysis functions, algebraic and logic operators, and other various widely-used functions. In the present version, we implemented two types of neural networks which are most used in identification and control: the multi-layered perceptron (with sigmoidal or linear output) and the memory neuron networks [2].

NeSS has been designed as an "open" system in order to deal efficiently with particularly complex mathematical models. In such a case, a dedicated procedure can be written in any programming language and put into the operators class library, so that simulation is sped up considerably.

4. THE GRAPHIC EDITOR

The structure of a dynamic system can be easily defined by the user through a powerful graphic editor GE. It provides several drawing features and filing management support.

In NeSS, the system graphic representation adheres to the hierarchical structure of the system. The whole system, the subsystems, the connectors and the elementary operators are represented by blocks. The one at the highest abstraction level is the *project*: it contains the information related to system specifications and interconnections and the simulation options. Functions and connectors are terminal blocks (i.e., they are atomic objects), while subsystem blocks contain interconnected blocks associated with the definition of the subsystem.

To each project and all internal blocks is associated a window; for instance, in the example given in Fig. 3, the upper window is associated with the project while the lower window characterizes one of its components.

The graphic editor allows to define the project architecture by drawing the rectangles representing the subsystems in the window working space and by connecting them through interactive mouse-based operations. Atomic objects (i.e., the elementary operators) are represented by rounded blocks, as shown in Fig. 3. Each subsystem has its own internal architecture, which can be defined, in turn, in the same way. The global architecture of a project can thus be viewed as a tree whose root is the project itself and whose leaves are atomic functions. A non-atomic object is exploded in its own components, which can be edited in a separate window.

A connection between two blocks can be easily drawn by selecting the blocks with the mouse. Connections are relocable around the blocks in order to support possible restructuring both of the graphic representation and of the signals that must be linked.

The menu items *File* and *Component* allow the access to the filing management primitives, i.e., to load the project and components and save them in a file. Saving a project means to store its architecture (components and interconnections) and its state, namely, the current value of the dynamic parameters and the global features of the simulation. This allows to stop a simulation, save its state on a file, and then restarting it exactly from the point at which it was pre-empted without losing any information.

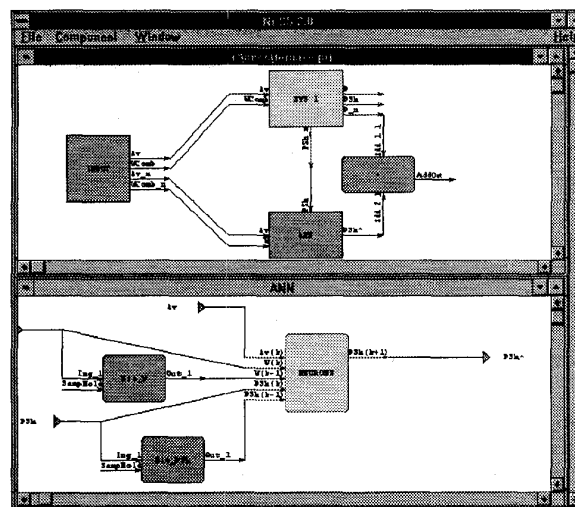


Fig. 3 - The project's architecture (upper part) and the subsystem's internal structure (lower part).

Besides, each component can be independently saved on a file, in order to create a personalized library of reusable objects for future projects. The *Component* menu supplies blocks management facilities such as component renaming, searching in working space, I/O relocation, and creation of new blocks.

5. THE SIMULATION ENGINE

NeSS provides a sequential simulator for the dynamic system designed by the user. It includes a simulation engine and the auxiliary procedures for simulation setup and printing result of both in text and graphic formats. The structure of the simulation module is shown in Fig. 4.

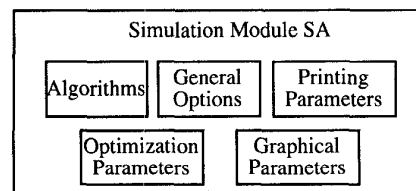


Fig. 4 - The module SA.

Simulation uses information and elementary methods contained in the system data structure, as defined by the user. Two kinds of simulation can be generally envisioned: *training* and *validation*. Validation is directed to observe and evaluate the performance of a given system on a data set or directly on data coming from the plant. Training allows to tailor the system

model on the specific application by adapting the model parameter to the application data.

In both cases, simulation is performed on the system selected and loaded by the user. All components and subsystems are serially simulated starting from the primary inputs of the system towards the primary output. This avoids all problems related to synchronization, which constitute the main drawback in parallel simulators working on a shared-memory data structures.

In the *validation mode*, the dynamic system configuration (i.e., its parameters) is assumed known. Actually it is usually the result of a suited training phase.

Validation is performed by exciting the top-level system with user-defined input signals. According to the system's internal architecture, signals are propagated to the system's components until atomic ones are reached. At this point, the computation can be performed by using atomic components' methods and results are forwarded to the upper level. The computation propagates from the bottom to the top until the most external system is solved.

In the *training mode*, the configuration parameters of the elementary operators (including the neural networks) are optimized according to a user-selected goal. Since elementary operators contain methods to compute the first derivatives of the outputs with respect to their parameters we can define training algorithms using gradient information.

As in the validation mode, the gradient of the top-level system is evaluated recursively by propagating the computation downward to the atomic components. Gradient computation is completed in two steps. In the first one, the error function derivatives with respect to the outputs of the operator under training are computed by using a numerical algorithm to guarantee the simulator generality. In the second step, the operator outputs derivatives with respect to the operator characteristic parameter are computed. The gradient vector is then obtained by multiplying these two derivative matrices.

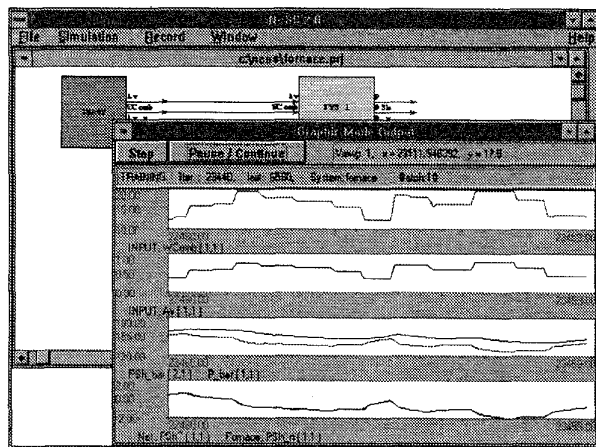


Fig. 5 - Data output in the graphic format.

NeSS supports different techniques for gradient computation [4] to match the specific characteristics of the envisioned application. Gradient may be computed according to different granularities of training data representation, i.e., to different (single or multiple) sizes of the training window on sampled

data. We can evaluate the gradient at every individual pattern or at the end of every training pattern batch. Besides, both fixed-step gradient and conjugate gradient algorithms have been implemented in a training procedure library.

Different training methods have been implemented to take in account effects induced by the component dynamic behavior. Recurrent neural networks can be configured by using the Williams-Zipser algorithm and Teacher Forcing correction [5].

The user can save or load files containing the status of the simulation, the data produced during the simulations and the parameters of the optimized operators.

Simulation options are set in the menu item *Simulation* and by selecting the desired configuration through dialog boxes. Adaptation of each system parameter is performed according to the error obtained by comparing a selected signal with the reference value. If required, no optimization is performed: this is useful to test the outputs of a non-adaptive classical control system. The user can select one or more signals that will be displayed in text and/or graphic format (see Fig. 5); data outputs can be sampled at a user-defined frequency.

Menu item *Record* allows to save data outputs on files in the Microsoft Word or Excel format, so that further data processing may be performed by using these tools or other which can load the same file format.

6. CONCLUSIONS

NeSS is a new flexible simulation environment which has been proved very effective to design complex systems for signal identification and system control. By supporting both various neural networks and algebraic/trigonometric functions, NeSS allows to create, configure, and simulate complex dynamic non-linear systems in which algorithmic components are integrated with neural networks. Optimization processes have been defined to adapt system parameters in order to minimize a user-defined error function.

Current research addresses problems related to connect the simulator directly to data coming from the real plant. This can be obtained by using I/O devices connected with the PC running the simulator. This facility is particularly attractive since it allows real time monitoring and supports users' on-line decisions.

REFERENCES

- [1] J. Hertz, A. Krogh, R. G. Palmer, "Introduction to the Theory of Neural Computation", Addison-Wesley Publishing Co., 1991.
- [2] SIMULINK, MATLAB with SIMULINK, The MathWorks, 1993.
- [3] P. S. Sastry, K. P. Unnikrishnan, G. Santharam, "Memory Neuron Networks for Identification and control of Dynamical Systems", IEEE Transactions on Neural Networks, March 1994.
- [4] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, "Numerical Recipes in C, the Art of Scientific Computing", 2nd Edition, Cambridge University Press, 1992.
- [5] R. Williams, D. Zipser, "A Learning Algorithm for continually running fully recurrent networks", Neural Computation 1, 270-280, 1989.