Microprocessing and Microprogramming 36 (1993) 127–139
North-Holland

NSL: A language for neural network simulation

F. Panetsos[a,b], J. Alonso[a], E. Barja[a,c], P. Isasi[a,d] and V. Olmedo[a]

[a]*Escuela Politécnica Superior, Universidad Carlos III de Madrid, Av. del Mediterraneo 20, 28913 Leganés Madrid, Spain*
[b]*INFN (Istituto Nazionale di Fisica Nucleare), Sezione di Pavia, Via A. Bassi 6, 27100 Pavia, Italy*
[c]*Telefónica I+D, Emilio Vargas 6, 28043 Madrid, Spain*
[d]*S.E. Quinto Centenario, Vicente Jimenez 7, 28023 Madrid, Spain*

Abstract

Panetsos, F., J. Alonso, E. Barja, P. Isasi and V. Olmedo, NSL: A language for neural network simulation, Microprocessing and Microprogramming 36 (1993) 127–139.

Among the problems to be considered when designing a neural network simulator, a very important question is represented by network architecture definition and easy algorithm implementation. In this paper a new simulation-oriented programming language is presented. NSL (Neural Simulation Language) makes it possible to create an arbitrarily complex network and to define procedures in order to perform state update operations.

Keywords. Neural networks, simulation, parallel computing.

1. Introduction

A neural network is a set of elemental processors (neurons) interconnected by one-way signal transmission channels allowing communication between nodes. Each node may have any number of input lines (through which different signals pass) and any number of output lines, through which the same signal is transmitted. The nodes receiving signals from the outside world and those transmitting their output to the outside form the input and output layer respectively (not always distinct). The remaining nodes, if they exist, form the so-called hidden layers.

Correspondence to: F. Panetsos, Escuela Politécnica Superior, Universidad Carlos III de Madrid, Av. del Mediterraneo 20, 28913 Leganés Madrid, Spain.

*This work has been partially supported by a grant from the CEC to F. Panetsos, a grant from the Spanish Governement to J. Alonso and the Comunidad de Madrid (C145/91 project).

The neurons have an associate transference function which represents its operational capacity and which can generally be splitted into activation function (determining the state of the node) and output function. The connections have an associate weight, proportionally to which the signals passing through them are modulated. The weights of the connections to the neuron, along with a memory modification function, belong to what is referred to as its local memory composed of other parameters, such as threshold, state of the neuron at previous times, etc. (*Fig. 1*).

The state of the neuron at time t is given by the value of the activation function and depends on the input signals and the local memory. The state of the network at time t is given by all of the states of the simple neurons. Network functioning is determined by laws which describe the evolution of its state with time. The modification of the local memory of the simple neurons is named learning.

A network may be represented in several ways with the aim of defining a general model that can be

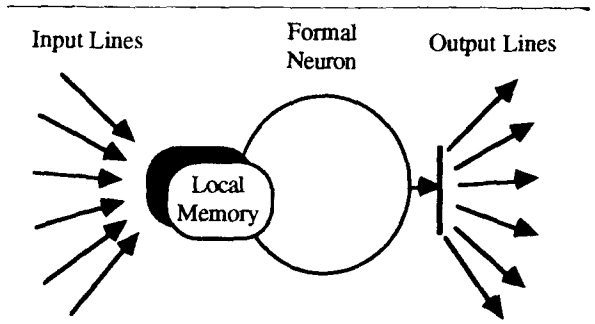


Fig. 1. Formal neuron, I/O lines and local memory.

used to create a programming language: for example, as a set of layers S_i, $(S_i \cap S_j = \varnothing,\ \forall i \neq j)$ of neurons with the same activation function and updated all together. The connections may also be defined as homogeneous groups of oriented lines connecting sets of neurons with given properties [1].

The dynamic behavior of the network is determined by the functions which enable a computational process to be associated to the network: local memories modification, (possible) elements synchronization, etc. An example of such a system is represented by the well-known feed-forward neural networks trained by the backpropagation learning algorithm:

A neuron element i, having n inputs (*Fig. 2*) is specified by n weights $w_{i1}, w_{i2}, \ldots, w_{in}$, and a threshold h_i which form its local memory. The n input variables $x_1, x_2, \ldots, x_n$, take on values between 0 and 1 and the output of i-neuron is defined by (Eq. 1) and (Eq. 2).

$$y_i = \frac{1}{1 + e^{-\text{net}_i}} \tag{1}$$

$$\text{net}_i = \sum_{j=1}^{n} w_{ij} y_j + h_i \tag{2}$$

A feed-forward multilayer network is a set of one-layer networks where neurons of one layer are only connected to neurons of the next layer and outputs of layer i are inputs to layer $i+1$ (*Fig. 3*). In time t_0 an input pattern of k components is presented to the input layer. In time t_1 (in a synchronous manner) neurons belonging to the first layer, update its output values which are transmitted to the neurons of the next layer. The process terminates when signals reach the output layer.

During learning, a pattern is presented to the network and weights are adapted according to whether the network output y_j matches a target output t_j imposed on the system. Their difference is defined as the error, and that difference is then fed back to adjuste the weights according to the backpropagation training algorithm which minimizes the mean square error [2].

2. Simulation language

2.1 Neural network design

The design of the networks is divided into two conceptually and structurally different stages:
(a) The description of the static part of the net-

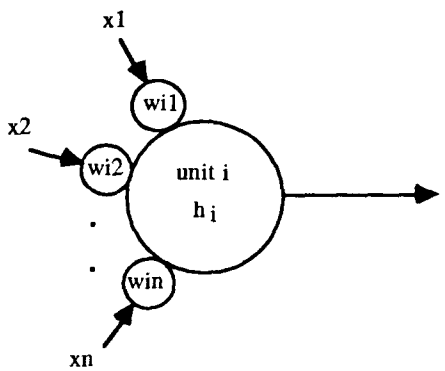


Fig. 2. Neuron i, with local memory represented by the weights of the n connections and the threshold of the neuron.

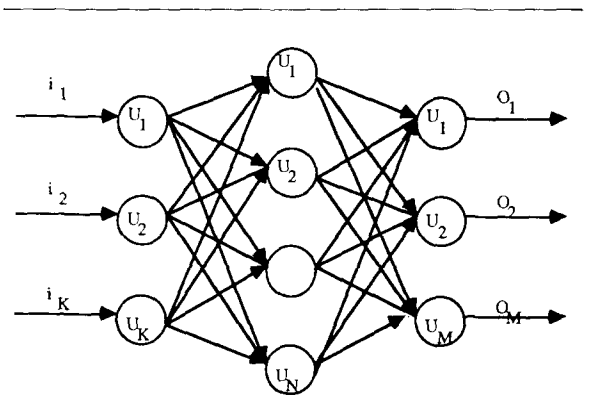


Fig. 3. Multi-layer feed-forward neural network.

work, i.e. the number and type of components and the communications between them;

(b) the definition of the laws which determine the dynamics of the elements making up the network. The two steps are usually independent, and this fact should be reflected in the structure of the programming language.

'Programming' a neural network implies varying the configuration of its elements in order to improve its performance. As a result, a language for defining the neural network is not enough. It has to give the user the opportunity to retrieve, alter, duplicate and combine networks. In this paper, a general purpose high-level language is presented which meets the above requirements. NSL (Neural Simulation Language) is oriented to the easy description of the network architecture and the implementation of algorithms based on a generalized concept of network. The evolution of neural networks with time is specified and processed by applying a series of operators to sets of interrelated objects. Only the conjunction of operators, objects and functions or procedures, as well as their use, gives the designed network its specific meaning. Using this concept, any type or kind of network can be implemented, ranging from biological models to artificial neural networks [3]. The different objects which constitute a network each one of which has associated functions and procedures are net, subnet, interface, connection and cell.

2.2 *Architecture*

The net is the most general object and is composed of an input interface, an output interface and either a set of subnets interconnected through interfaces or cells and connections. An instance of the net may, in turn, belong to another network (possibly created subsequently) and is then a subnet. A procedure for updating the state of its subnets, cells and connections is associated to each net.

The main advantages of this structure lie in:

- *Modularity*: A network may be built on the basis of other smaller previously trained nets. The functionality of a given net will be the sum of the functionalities of the different subnets of which it is composed.
- *Reusability*: Libraries containing networks with specific functionalities are available and may be used to build other more complex nets.
- *Locality of effect*: In a net composed of several subnets, one of them can be modified without affecting the rest of the structure.
- *Parallelization*: The use of interfaces in subnet communication is useful when the network has to be arranged in a multiprocessor system (for example a Transputer-based system).

The *interface* is the object through which the internal network communicates with its environment which may be a network, subnet or the outside world. We differentiate between input and output interfaces which differ with respect to their use but not to their structure, since both are made up of a set of the same type of cells.

The use of interfaces enables:

- The hidding of the structure by making the subnet operate like a black box.
- Easy duplication and modules interconnection.
- Easy substitution of subnets.

A *connection* embodies a relation between two cells and is composed of an ordered n-upla of numerical fields. Types of connection differ depending on the cells of which such connection is composed: connections between cells of the same network and connections between cells of a (sub)net and an interface. A connection is a set of time-variable values managed and transformed by a set of functions.

The use of this structure enables:

- *Different kinds of connections*: different types of specialized connections performing particular functions may be available within the same subnet.
- *Connections with memory*: since the fields of a connection are not defined a priori, these may be used as the states of the connection at time t or at K times prior to the observation, as well as other parameters and/or variables of the whole system.
- It also means that the process of updating the state of a connection can be made independent of the general processes of the network.

The *cell* object is composed of an ordered n-upla of numerical fields and may be an interface or network cell, these two differing both in structure and operation. The former belongs to the input or output interfaces and its function is to store provisionally the information flow between the internal cells

of two different nets, enabling both subnets to be processed as independent objects (modularity of the structure). A cell of this type may be connected with a single cell of another interface and a single cell of the network to which it belongs. The later is made up of a set of values which may change with time and is managed and transformed by a set of functions. There is no limit to the number of connections in which these cells may participate. Thus it is possible to have:

- Different types of cells: different types of specialized cells performing particular functions may be available within the same subnet.
- Cells with memory: since the fields of a cell are not defined a priori, these may be used as the state of the cell at time t or at K times prior to the observation.

2.3 *Functions and procedures*

As mentioned above, in order to define a neural network, in addition to the objects and the relations between them we need a set of functions to associate to the objects, completing the functionality of the net in question. Accordingly, we can define network, subnet, interface, connection and cell functions.

Network functions are the ones (possibly only one) which establish the dynamic behavior of the different elements making up the network (that is, the moment of activation and the time interval in which they are active), as well as the communication between them. The I/O data management functions are also network functions and the same principle is applied to the subnets.

This makes it possible to easily implement networks with heterogeneous synchronization, that is, establishing different time bases or clocks for each network (synchronous or asynchronous, deterministic or random) enabling individual synchronisms for each subnet and generating hierarchies (if necessary).

Interface functions determine the intercommunication between networks and the interactions between the net and the outside world. Moreover they make the operation of the net independent of the format of the data.

Connection functions determine the behavior and type of connections, transmit values from the source to the target cells and update or maintain internal values.

Cell functions are splitted into three clearly distinct types:

- Input functions for picking up and transforming the values provided by the connections of the target cells.
- Activation functions for management and updating of the different cell fields.
- Output functions which provide the value or set of values to the part of the network related to that cell.

2.4 *NSL: Description*

NSL, the syntax of which is supported on YACC and LEX, is made up of two languages. The first, called NDL (Neural Description Language), makes it possible to define the objects and the relations between them using a simple set of instructions and operators. The second, NAL (Neural Algorithm Language), has been developed to describe the functions and procedures associated with the operation of the network. It enables the dynamics of the network to be defined independently of the language which describes the static structure. Both languages are not directly related with the simulation process: while the NDL compiler translates the program into data structures, the NAL compiler translates the program into a low-level language called SCRIPT. Simulation is performed by an interpreter which reads data and SCRIPT instructions and then it executes them. The interpreter of this language is based on a stackoperated automaton and thus may be parallelized and executed on multiprocessor cards with task distribution (see App. 2).

NSL operates in a system development environment enabling programs to be prepared, compiled, simulated and the results to be displayed. From the operational point of view, the system is divided into two parts: 'High-level Languages' and 'Simulation Module'.

High-level Languages:

Data structures and the associated operational procedures are generated on the basis of a description of the features of a network, in such a way that they can be used by the interpreter. A set of primi-

tive functions which perform elemental operations (create, erase, modify, display, obtain information on objects created, etc.) are used to create networks and procedures.

The description of a neural network is not procedural, consequently a non-procedural declarative language [4] is the most suited for this task, and this is a feature of NDL. NDL creates the structures of the data to be used by the simulator together with the updating procedures generated by NAL. The syntax is designed to cover the set of features described above and enables the:

- definition of the structure of every network element
- definition of the relations between objects
- initialization of the variables
- incorporation of networks previously built and/or trained
- definition of sets of objects and arrangement of operators
- use of predefined objects.

NAL is a high-level typed procedural and structured language, whose syntax is similar to that of C or Pascal. NAL allows the declaration of global and local variables, as well as the use of library variables and is equipped with the set of sentences typical of high-level programming languages. Each type of data has a set of associated operators and it is possible to include SCRIPT sentences directly into the NAL code. NAL, like NDL, has been designed for prototyping different types of networks. Although the simulation speed obtained is satisfactory, we might, after an initial examination, wish for higher speed, which would be achieved by a specific simulator in C or Pascal. The syntax makes translation into these languages easy.

Simulation module:

It is a set of aiding programs for the design of networks, templates, etc., and the simulation. A graphic interface is used in order to visualize the structure of the network and the evolution of its values in real time. The interpreter module is the principal component of the simulator, since the NAL procedures are translated into the SCRIPT language interpreted by this module (SCRIPT is a suffix notation based language; it is a variant of FORTH adapted and oriented to neural network simulation). Its internal procedures (reserved words) are called up from files translated by the NAC and by library files and may be employed by the user whenever he needs any one of them.

SCRIPT is a low-level language with some elements of a high-level one and a fairly large vocabulary of reserved words which may be classified under the following headings:

- Control structures: for loops, generalized while loops, if-then-else, ...
- Building words: ranging from the most elemental ones (defining variables and procedures) to less usual ones (defining menus, function interpolators, etc.).
- Stack and other internal structures management (characteristic of this type of interpreters).
- Arithmetic and logical operators.
- Memory and input-output access words.
- Words for accessing the relevant fields of the network data structure.
- More complex function words oriented to neural network simulation such as variants of back-propagation, feedforward, sigmoids, ...

The system has been developed in C on UNIX machines and verified on other machines and operating systems such as MS-DOS and VMS.

3. Description of an NDL program

An NDL program starts with

NETWORK (Mandatory):

Here, the name of the network to be created is indicated and the program field is delimited.

```
NETWORK net_name
  Body of program
END
```

The program is divided into five sections:

IMPORT (Optional):

A previously validated or trained network can be incorporated in this section, elements may be added or removed.

```
NETWORK New_Net
IMPORT Old_Net
  Declarations
END
```

Creates the network 'New-Net' on the basis of the 'Old-Net'.

CELLS (Mandatory):

The network cells and conections are specified in this section:

- The names of the additional cell fields and, implicitly, their number (optional sentence).
- The names of the additional connection fields (as above).
- The number of network cells.

Once the number of cells and connections has been defined, different subsets of cells can be named:

- the set of input cells
- the set of output cells
- the sets of hidden cells.

Examples:

FCELL = (field1, field2, ...);

The additional cell fields are field1, field2, etc.

FCON = (field1, field2, ...);

The additional connection fields are field1, field2, etc.

CELNO = Cell_number;

The total number of the cells is 'Cell_number'.

Name_Set = [1..7, 9, 15, 17..19];

'Name_Set' is the set composed by the cells 1, 2, 3, 4, 5, 6, 7, 9, 15, 17, 18, 19.

If 'Name_Set' is the reserved word IN the cells belong to the input interface.

If 'Name_Set' is the reserved word OUT the cells belong to the output interface.

If 'Name_Set' is the reserved word INOUT the cells belong to the input and output interface.

Every network must include both the IN and OUT sets or an INOUT set. The cell reference numbers must belong to [1..CELNO].

STRUCTURE (Mandatory):

The connections between the cells are defined using a powerful set of operators. There are two types of connections: one-way and two-way. The difference between a two-way connection and two one-way connections with the same weights but going in different directions is only structural. In the first case there is one object and in the second two.

For the topology can be used predefined schemata and/or special purpose operators. There are currently two types of predefined topologies: FF (feedforward) and FULL, and they are declared with:

TYPE FF(n_level1, n_level2, ...);

TYPE FULL(n);

FF represents networks structured by layers. Every cell is connected with each the cell in the next layer. There are no connections between cells of the same layers or feedbacks.

FULL represents networks in which each cell is connected with all the others but not with itself.

If the desired network does not correspond to neither of the predefined ones, a set of operators is used with or without predefined topologies, e.g. (Op indicates one of the language operators):

Set1 Op Set2;	Create connections between Set1 and Set2.
DEL Set1 Op Set2;	Delete connections between Set1 and Set2.

Operators for individual and block connections are available.

Individual connection operators: For every cell belonging to Set1, only one connection with the corresponding Set2 cell is created.

->	Unidirectional connection.
-	Bidirectional connection.
<->	Two opposite unidirectional connections.
@	Feedback connection to the same cell.

Ex. Let S1 = [1, 2] and S2 = [3, 4]

S1->S2:	Means 1->3 and 2->4.
S1--S2;	Means 1--3 and 2--4.
S1<->S2;	Means 1->3, 3->1, 2->4 and 4->2.

Block connection operators: For every cell belonging to Set1 a link to every cell of Set2 is created.

=>	Unidirectional connection.
=	Bidirectional connection.
<=>	Two opposite unidirectional connections.

Ex. Let S1 = [1, 2] and S2 = [3, 4]

S1 => S2;	Means 1->3, 1->4, 2->3 and 2->4.
S1 = = S2;	Means 1--3, 1--4, 2--3 and 2--4.
S1 <=> S2;	Means 1->3, 1->4, 3->1, 4->1, 2->3, 2->4, 3->2, 4->2.

The WITHTHR operator is applied to a set of cells and specifies that they have an associate threshold. Ex.

WITHTHR [4..10];

INIT (Optional):

The different fields in each one of the network elements (the parameters of the different objects) are initialized. If they are not specified, the compiler will initialize them with default values. Similarly, some values, such as connection weights, can be indicated as fixed. The following operators may be used to do this: THR:, 'FieldCell', W. THR indicates that the threshold of a cell, is to be initialized.

'FieldCell' is the name of a given, previously defined additional cell field. W indicates the weight of the connection or connections between a pair or pairs of cells, respectively.

```
THR[i]=0.3*i:i IN 1..7;
```

The result of multiplying 0.3 to the number of the cell is assigned to the cells 1 to 7.

```
THR[*]=0.9;
```

All network cells are initialized with threshold 0.9.

```
Error_t1[7]=0;
```

The additional field Error_t1 of the cell 7 is initialized with the value 0.

```
Excit_t2[i-1]=i/0.9:i IN 2..4, 6, 7;
```

The additional fields Excit_t2 of the cells 1, 2, 3, 5, 6 are initialized with the result of the division between cell number and 0.9.

```
Inhib_t1[*]=0;
```

All additional fields Inhib_t1 of the network are initialized with 0.

```
W[*, *]=0;
```

All weights of the network are initialized with 0.

```
FIX W[*, i]=0.7:i IN 1..3;
```

All connections to the cells 1, 2, 3 are initialized with 0.7 and, moreover, these weights will be fixed during simulation.

```
W[i, j]=rand(5):i IN 1..4, j IN OUT;
```

All connections between cells 1, 2, 3, 4 and the output cells are initialized with a random integer value between 1 and 5.

GATEWAYS (Mandatory in the case of a network made up of several subnets):

Builds up a network from subnets by connecting them through input and output cells using the operator -〉(once the subnets have all been defined).

Ex. Let R1, R2 and R3 be identifiers of simple subnets.

```
GATEWAYS
  R1.[52..54]->R2.[2, 4, 6]
  R3.[81..85]->R2.[1, 3, 7..9];
END
```

The output interfaces of R1 and R3 are connected to the input interface of R2, connecting cells one by one.

4. Description of an NAL program

An NAL program always begins with the reversed word NETWORK followed by the name of the network to which it is referred. Its structure is:

```
NETWORK Net_Name
Global declarations
function_1()
{
  Local declarations
  List of sentences
}
....
MAIN
{
  Local declarations
  List of sentences
}
```

Each of the most important parts of the language is presented in more detail below.

Declaration of variables

As in Pascal and C there are two types of variables (global and local) which must be defined in the program before they are used. In addition to the variables defined by the programmer, there is a set of specific predefined variables which simplify the source code. Predefined variables cannot be declared in the program.

Type of data: INT, REAL, POINTER.

Predefined cell variables

Out[*i*]	Output value of cell *i*.
Excit[*i*]	Dot product of weights and input values to cell *i*.
CellError[*i*]	Difference between expected and calculated output value of cell *i*.

Predefined connection variables

(In the following variables the index *j* may appear depending on the inclusion or not into a block following the word 'withcell').

_w[i, j]	Weight of the connection between cells *i* and *j*.
w[i, j]	Weight of the *j*th connection to the cell *i*.

Some predefined variables

CountCells	Number of cells in the selected network.
Count-Connects	Number of processed connection in time *t*.

Additional fields

Following the above notations, the additional cell and connection fields defined in NDL can be used. Ex.

Cell_field[7] = 23;

The value 23 is assigned to the cell 7.

_Conex_field[11,5] = 0.8;

The value 0.8 is assigned to the connection between cells 11 and 5.

Conex_field[11,5] = 0.8;

The value 0.8 is assigned to the additional field of the fifth connection of cell 11.

Functions and procedures

FUNCTION f_name (declaration of variables): type of function

{
Body of function
RETURN(expression)
}

PROCEDURE p_name (declaration of variables)

{
Body of procedure
}

As in the case of the variables and in addition to those declared by the programmer, there is a set of predefined functions and procedures in a library.

Some predefined functions

Mathematical functions: sin(x), cos(x), tan(x), sinh(x), In(x), exp(x), sqr(x), abs(x), etc.

Activation functions: sigmo(x) = Sigmoid, hardlim-(x) = Hard limiter, thrlogic(x) = Threshold logic, etc.

Simulation functions:

NetError()	Sum of the differences between expected and obtained values of output cells.
Derivative(i)	Returns the value of the derivative of the activation function in i.
IsCellInput(i)	Returns TRUE if the cell i is an input cell.
x(i)	Returns the input value of the input cell i.
xto(i)	Returns the number of connections to the cell i.
FixActiv(i)	Returns TRUE if the cell i has a fixed activation.
_FixWeight(i,j)	Returns TRUE if the weight of the connection between cells i and j is fixed.
FixWeight(i,j)	Returns TRUE if the weight of the ith connection to the jth cell is fixed.

Some predefined procedures

SelectNet(NetName)	Selects a subnet.
SelPattern(i)	Selects the pattern i as input.
SetNetErrorO	Initializes the error of the net.
SaveOutO	Saves the output values of the net to the interface.
Inc(x, Exp)	$x = x +$ result of expression Exp.
Dec(x, Exp)	$x = x -$ result of expression Exp.

Sentences

Assignments.

IF-THEN-ELSE, WHILE, FOR.

EXEC (permits the execution of a function if starting direction and parameters are known).

Guides: scr Y endscr (permit the inclusion of SCRIPT code in a NAL code).

Operators

Arithmetical, logical, relational, ...

5. Discussion

Using NSL both biological and artificial neural networks can be simulated without restrictions on the type of neurons, connections, etc., or the learning functions which can be defined by the user or by the system. NSL is comparable to simulation systems and languages like P3 [5], RCS [6], NNSIM [7] and SLONN [8].

Compared to these latter, NSL offers the same advantages:

- Possibility of recycling software and a simple and rapid development stage.
- Possibility of building a network on the basis of smaller ones and of using independent updating procedures for each subnet. This facilitates simu-

lation on parallel machines by distributing the subnets between the different computing elements with communication via their interfaces. If parallel machines are not available, the interfaces have the disadvantage of needing an additional computing and memory load for communication between the subnets, which slows down the simulation speed particularly in the case of small computers (PC/AT, etc.).

- Topology/algorithm independence: different management algorithms are available for the same network and, in addition, the introduction of new elements in any of the two languages does not, in principle, require variations in the syntax of the other.
- Easy extendibility with respect to future needs: the definition of the syntax is supported on YACC and LEX (as well as being fully portable) as these standard tools are widely accepted.

In addition to the above mentioned characteristics, the user can control network simulation by control statements in the program as well as interactively; he can modify the structure and behavior of the network during simulation by adding new elements (like neurons and weights) or by adding a functions.

Interpretation leads to simulation times which are approximately double the time needed for programs written in C.

Appendix 1. An example

Programming in NAL standard learning algorithms, such as Backpropagation, can be very simple using the procedures in the internal (written in C) and external (written in SCRIPT) libraries of the simulator. The working mode of such procedures can be alterated using NAL sentences. The following example in NAL uses our standard Backpropagation algorithm:

```
NETWORK ex1;
#use backprop.scr
main.
{
  main_bp O ;
}
#associate   f_activation        u_polar_sigmoid
#associate   f_output            u_identity
#associate   f_activate          u_activate
#associate   f_load              u_feedforward
#associate   f_learn             u_backpropagation
#associate   f_correct_weights   u_correct_weights
#associate   f_correct_weight    u_correct_weight
#associate   f_cell_error        u_cell_error
```

The first sentence specifies the network to simulate. Each NAL sentence starting with a will be passed unprocessed to the SCRIPT interpreter, so **#use backprop.scr** tells the SCRIPT interpreter to use the definitions contained in file BACKPROP.SCR (external Backpropagation library). The only task of NAL main procedure is to call the Backpropagation main procedure **main_bp**. The following sentences define several links between the library procedures. The procedure **main_bp** calls the links **f_learn** and **f_load** which should be associated with procedures that perform Backpropagation and Feedforward (in our example: **u_backpropagate** and **u_feedforward**). **u_feedforward**, in turn, uses a link called **f_activate** each time it wants to compute the activation of a neuron. This is done in **u_activate**. The last procedure calls links **f_activation** and **f_output** (associated to **u_polar_sigmoid** and **u_identity**) in order to compute the activation and output values of a neuron (and so on). External procedures are identified with a starting **u_** while internal procedures (written in C) have a name starting with **i_**.

The use of this kind of links is very straightforward. To use another activation function for example, the following sentences in NAL are necessary:

```
rFUNC  my_activation (x):real
real x;
{
        if (x>0.0)    /* this is the polar threshold */
                      return (1.0) ;
        else
        return        (−1.0) ;
}
associate f_activation my_activation
```

The definition of a three layer network (4-9-1) with threshold connections in the hidden and output layers in NDL could be the following:

```
NETWORK ex1
  CELLS.
    CelNo = 14;
    input = [1..4];
```

```
    hidden = [5..13];
    output = [14];
  STRUCTURE.
    input =) hidden;
    hidden =) output;
    with_thr hidden;
    with_thr output;
  INIT.
    thr[i] = random(100.0)/50 - 1 : i in [5..14];
    w[*, *] = random(100.0)/50 - 1;
END
```

All weights are initialized between −1.0 and 1.0.

Appendix 2. Multitasking with SCRIPT

Nowadays there are several FORTH interpreters which allow multitasking. In these interpreters it's the programmers duty to define procedures and determine which ones could be executed as concurrent tasks [9]. Moving from such a system to another one with several processors does not present major difficulties.

Our main purpose is to parellelize loops that perform any type of vector processing, which is a very common operation in neural network simulation. the following examples are valid for multiprocessor systems with one master and several slave microprocessors. Each one has exactly the same copy of the SCRIPT interpreter (which is very similar to FORTH). The master processor executes the users main program, delivers to the slave processors some specific tasks for execution and collects the resulting data. The basic schema for this is very simple. As an introductory example we will discuss a small application for neural networks written in our SCRIPT language.

```
define feed_forward
  cell_number@  0  loop
    i  activate
  endloop
enddef
```

The purpose of this procedure (**feed_forward**) is to activate sequentially all the units (from 0 to cell_number) of a neural network.

The following sentence:

```
  i  activate
```

might be written:

```
  i  'activate exec
```

where

'activate	leaves on stack top the code address of the procedure **activate**
exec	takes an address from the stack and executes the code starting at that point.

This is just the same as using only **activate**, but it will be useful for our next example.

In a multiprocessor system, the first thing we can do is to activate different neural units on different slave processors. The master processor will send each slave the task to be done (the code address of **activate**) and its parameters (the number of the unit in this case). This parameters are placed on the stack and sent to a specific slave processor with the word **export**. This word needs some additional information, such as the number of parameters to be sent and the identifier of the target processor.

```
define feed_forward
  cell_number@  0  loop
    i 'activate 2 i mod_slave_proc  export
  endloop
enddef
```

2	is the number of parameters to export
i mod_slave_proc	leaves on the stack the number of one slave processor depending on the number of existing processors and the neuron to activate.

The slave processor builds up a list with the incoming data. If it is idle it converts this list to its parameter stack and performs an **exec** with the same result as executing **activate** on the current neural unit.

This schema is not very efficient, because the processing time of different units may vary. So the slave processors will not be working at the maximal rate. This problem can be avoided with a slight modification.

```
define feed_forward
  cell_number@  0  loop
    i 'activate  2  on_idle_slave  export
  endloop
enddef
```

The new word **on_idle_slave** waits for any slave processor identifying itself as in idle state and leaves its identifier on the stack. The resulting data of an

exported task can be collected by the master processor with the word **import**.

⟨**number_of_parameter_to_import**⟩ ⟨**slave_number**⟩ **import**

retrieves the stack of the slave and leaves the data on the master's own stack.

Appendix 3. Benchmarks

The purpose of the following benchmarks is to show the performance of the simulator using different combinations of its library components. Experiments were done with Backpropagation learning. Since we are not interested in sampling the convergence time of several trials with the same network, only two types of experiments are presented.

A: using a backpropagation procedure written in C.

B: using a backpropagation procedure written in NAL.

The machines used for the benchmarks are a 25Mhz 80386 with a 80387 coprocessor and a 33MHz 80486.

The first example is a 16-4-16 Codec with the following results on the 25MHz 80386 & 80387:

A:	93 seconds	7815 Connections/sec.
B:	230 seconds	3390 Connections/sec.

The second example is a 2-1-2 XOR with the following results on the 33MHz 80486:

A:	115 seconds	15217 Connections/sec.
B:	256 seconds	8357 Connection/sec.

(Learning rate = 0.005, RMS Error = 0.01)

Appendix 4. Grammars

NDL grammar

```
program          : NETWORK Ident subnet_body
                   END
                 | subnet_list gateways
subnet_list      : subnet_decl
                 | subnet_list subnet_decl
subnet_decl      : SUBNET Ident subnet_bo-
                   dy  END
subnet_body      : import_section cell_section
                   struct_section init_section
import_section:
                 | IMPORT  Ident set_subsection
                   set_stmt_list imp_stmt_list
set_subsection:
                 | set_stmt_list
imp_stmt_list:
                 | imp_stmt_list imp_stmt ';'
imp_stmt         : DEL '['set']' setof
                 | ADD Integer before_after Inte-
                   ger  setof
setof            :
                 | OF Ident
before_after     : BEFORE
                   AFTER
cell_section     : CELLS '.' cell_fields connect-
                   _fields celno set_stmt_list im-
                   p_stmt_list
cell_fields      :
                   FCELL '=' '('field_list')' ';'
connect_fields:
                 | FCONEX '=' '('field_list')' ';'
cellnumber       :
                 | CELL_NUMBER '=' Integer ';'
set_stmt_list    : set_stmt ';'
                   set_stmt_list set_stmt ';'
set_stmt         : Ident '=' '['set']'
field_list       : Ident
                   field_list ',' Ident
set              : set_item
                   set ',' set_item
set_item         : Integer
                 | Integer '..' Integer
struct_section:
                 | STRUCTURE '.' struct_type
                   struct_stmt_list
struct_type      :
                 | TYPE FF '('layers')'
                 | TYPE FULL
layers           : Integer
                 | layers ',' Integer
struct_stmt_list:
                 | struct_stmt_list struct_stmt ';'
struct_stmt      : delete opset op opset
                 | delete '@' opset
                 | WITH_THR opset
delete           :
                   DEL
opset            : '['set']'
```

```
                | Ident
op              : '->'
                | '<->'
                | '=>'
                | '<=>'
                | '--'
                | '=='
init_section:
                | INIT '.' list
list            : stmt ';'
                | list stmt ';'
stmt            : asgn
                | asgn ':' cond
cond            : term
                | cond ',' term
term            : Ident IN opset
asgn            : Cell_field '['index']' '=' expr
                | flag THR '['index']' '=' expr
                | flag WEIGHT '['index', 'index']'
                '=' expr
                | Connect_field '['index', 'index']'
                '=' expr
flag            :
                | FIX
index           : Ident
                | Integer
                | '*'
expr            : Integer
                | Real
                | Ident
                | RANDOM '('expr')'
                | '('expr')'
                | expr '+' expr
                | expr '-' expr
                | expr '*' expr
                | expr '/' expr
                | '-' expr
gateways        :
                | GATEWAYS '.' gate_stmt_list
                END
gate_stmt_list  : gate_stmt ';'
                | gate_stmt_list gate_stmt ';'
gate_stmt       : Ident '.' opset '->' Ident '.' opset
```

Nal grammar

```
program         : selnet global_var_list pro-
                g_stmt_list main
selnet          : NETWORK Ident ';'
global_var_list:
                | global_var_list var_decl ';'
var_decl        : type ident_list
var_type        : INT
                | REAL
                | POINTER
ident_list      : ident_decl
                | ident_list ',' ident_decl
ident_decl      : Ident
                | Ident '=' val_type
prog_stmt_list:
                | prog_stmt_list prog_stmt
prog_stmt:
                | FUNC Ident '('param_list')' ':'
                var_type  func_body
                | PROC Ident '('param_list')'
                func_body
main            : MAIN '.' func_body
param_list      :
                | type identifiers ';'
                | list_param ';' type identifiers
identifiers     : identifiers ',' Ident
                | Ident
func_body       : '{'local_var_list stmt_list'}'
stmt_list       :
                | stmt_list stmt ';'
local_var_list  :
                | local_var_list local_var  ';'
local_var       : type identifiers
stmt            : '{'stmt_list'}'
                | var '=' expr
                | WHILE '('cond_expr')' stmt
                | IF cond stmt
                | IF cond stmt ELSE stmt
                | FOR Var_Ident '=' expr TO
                expr stmt
                | RETURN
                | RETURN '('expr')'
                | Proc_Ident '('args')'
                | Built_In '('args')'
                | WITHCELL expr
                | INC '('var ',' expr')'
                | DEC '('var ',' expr')'
cond            : '('cond_expr')'
expr            : val_type
                | var
                | '('expr')'
                | Ident_Func '('args')'
```

```
                | expr '+' expr
                | expr '-' expr
                | expr '*' expr
                | expr '/' expr
                | '-' expr
                | Built_In_Var
                | Built_In '('args')'
                | '&' Ident
                | '*' expr
cond_term       : expr '>' expr
                | expr '<' expr
                | expr '>=' expr
                | expr '<=' expr
                | expr '==' expr
cond_expr       : cond_expr '&&' cond_term
                | cond_expr '||' cond_term
                | '!' cond_expr
                | cond_term
                | '('cond_term')'
args            :
                | expr
                | args ',' expr
var             : Var_Ident
                | Cell_Field '['Integer']'
                | Connect_Field '['Integer ',' Integer']'
val_type        : Integer
                | Real
```

References

[1] R. Hecht-Nielsen, *Neurocomputing* (Addison-Wesley, Reading, MA, 1990).

[2] D. Rumelhart and J. McClelland, eds, *Parallel Distributed Processing* Vol. 1 (MIT Press, Cambridge, MA, 1986).

[3] F. Panetsos, J. Alonso, E. Barja and V. Olmedo, Programming languages for neural network simulation, TN No I.91.57, CEC-JRC, Ispra, 1991.

[4] T. Korb and A. Zelf, A declarative neural network description language, *Microprocessing and Microprogramming* 27 (1989) 181–188.

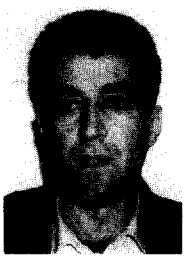
[5] D. Zipser and D. Rabin, P3: A parallel network simulating system, in: D.E. Rumelhart and J.L. McClelland, eds, *Parallel Distributed Processing Vol. 1: Foundations* (MIT Press, Cambridge MA, 1986).

[6] N. Goddard, K.J. Lynne and T. Mintz, Rochester connectionist simulator, Technical Report TR-233, Department of Computer Science, Univ. of Rochester, Rochester, New York, 1988.

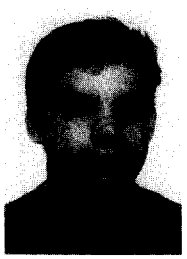
[7] J. Nijhuis, L. Spaanenburg and F. Warkowski, Structure and application of NNSIM: A general purpose neural network simulator, *Microprocessing and Microprogramming* 27 (1989) 189–194.

[8] D. Wang and C. Hsu, SLONN: A simulation language for modeling of neural networks, *Simulation* 56 (1990) 69–83.

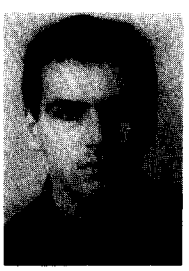
[9] L. Waring, Implementation of FORTH on a network of transputers, *Microprocessors and Microsystems* 15 (1) 49–53.



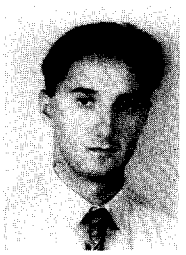
Fivos Panetsos is professor at Carlos III University, Madrid and responsible of the AI Laboratory. His interests are mainly in neural network architectures and their applications. His projects include neural control of chemical reactors, on-line pattern recognition in HEP experiments, machine vision and autonomous behaviour using NGST models.



Juan M. Alonso is a grantholder at the Engineering Department at Carlos III University, Madrid. He received his B.S. degree in Computer Science from the Madrid Polytechnic University, in 1990. He is carrying out his Ph.D. in Computer Sciences. His main research interests include neural networks, parallel computers, genetic algorithms and fuzzy logic applied to optimalization problems.



Pedro Isasi received his B.S. degree in Computer Science from the Madrid Polytechnic University, in 1990. From 1990 to 1991 he worked as a researcher in the Rank Xerox A.I. Department. Since 1991 he has been a faculty member of the Engineering School at the Carlos III, Madrid. He is carrying out his Ph.D. in Computer Sciences. His main research interests are knowledge engineering, neural networks and genetic algorithms.



Enrique Barja received his B.S. degree in Computer Science from the Madrid Polytechnic University, in 1990. Since 1990 he has been working in the R&TD department of Telefónica S.A. He is carrying out a Ph.D. degree in computer sciences. His main research interests include advanced computer interfaces and parallel computers.