IEEE Proc. Int. Conf. Tools for AI,
TAI - 89, Fairfax, USA 1989

Neural Network Simulation using INES

R.Brause

J.W. Goethe University, FB 20 VSFT,
D-6000 Frankfurt, West-Germany

Abstract

In modern pattern processing systems like computer vision, speech processing and robotics, information processing is done in several stages or layers. It is widely agreed among uroscientists that basic sensor processing parts of the brain can be modelled in the same fashion.
This paper describes the Interactive NEtwork Simulation tool (INES) which supports the simulation of this kind of pattern processing. Since the sequence, the interconnections and functional characteristics of the layers depends on the ideas and the needs of the user, INES does not assume a special interconnection scheme but gives the means to set up visually an interconnection scheme of predefined units. The possible interconnections obey some restrictions (rules), representing a kind of visual programming language.
Thus the programming and simulation language system can be used for the design and evaluation of a pattern processing neural network computer.

1. Introduction

In modern parallel computer architectures a new eneration of highly parallel, real-time oriented architecture for artificial intelligence is at the horizon. These attempts favorize computers made by many, small processing elements of very low complexety and therefore very limited computing power, connected directly together contrary to a relative small number of complex processors, communicating with a high amount of overhead. An example of these attempts is the connection machine [5].
One important class of highly functional parallel models are those proposed since 30 years by neurological and cybernetical scientists for modelling brain functions. The models are based on the function of simple elements, the neurons, connected extensively in a specific manner (*Neural Networks*). Every connection is assigned a specific weight.
Contrary to some Neural Network models where all neurons are connected with each other ("full interconnection", e.g. the Hopfield network [6]) the Neural Network simulation of INES aims to such kinds of problems which can be solved by the interaction of whole functional groups of neurons with a small number of connections between the groups (*layers*). Some researchers even see the principle of information preserving between these layers as the basic principle of human information processing [9].
For example, in many models of artificial intelligence the problem is divided into subproblems in a layered manner. In figure 1 the layers of computer vision and speech recognition systems are shown. On each level the layer has to provide some basic fault-tolerant abilities like recognition of varied, noise-disturbed and incomplete patterns [1].

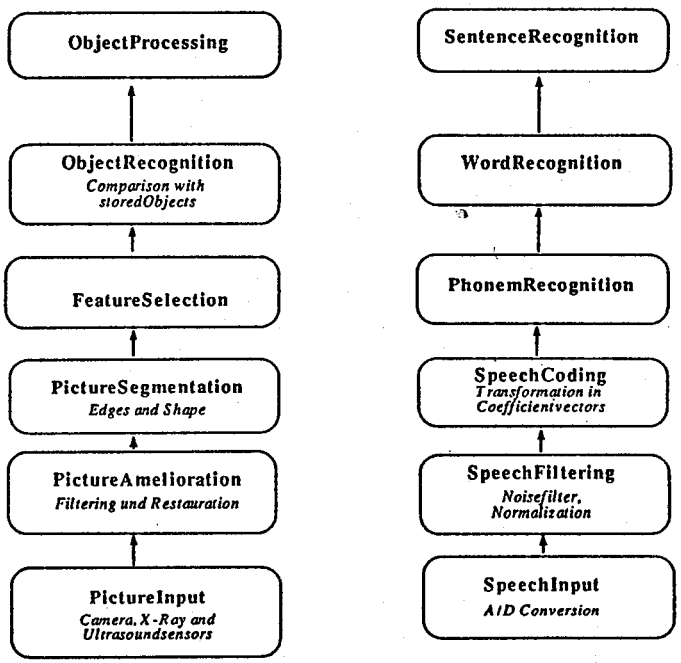

Fig 1 processing layers in computer vision and speech recognition

In the neural network approach each layer has nearly the same transmission function, e.g. a homogen structure of interconnected, very simple processing elements [3] or a Fourier transformation by optic stages [10].
In comparison to the view of AI-related problems the uniformity in the structure of the human pattern processing hardware, the brain, is striking. The brain consists of 80 micrometer wide histological columns which form functional units (*hypercolumns*) of 1-2 mm diameter. These columns consist of heavily connected pyramidal cells and other cell types, see [15].
The output of the columns is spread around a whole area of columns and further into other areas. Each column has three sources of input: short range signals, long range signals and unspecific, activity (gain) controlling signals.
Thus we have a hierarchy of functional units: neurons, columns, areas and cortex segments. Each unit can be functionally or histologically sharp distinguished from another unit of the same kind. In figure 2 such a network of layers (visual areas) is shown for the visual system.

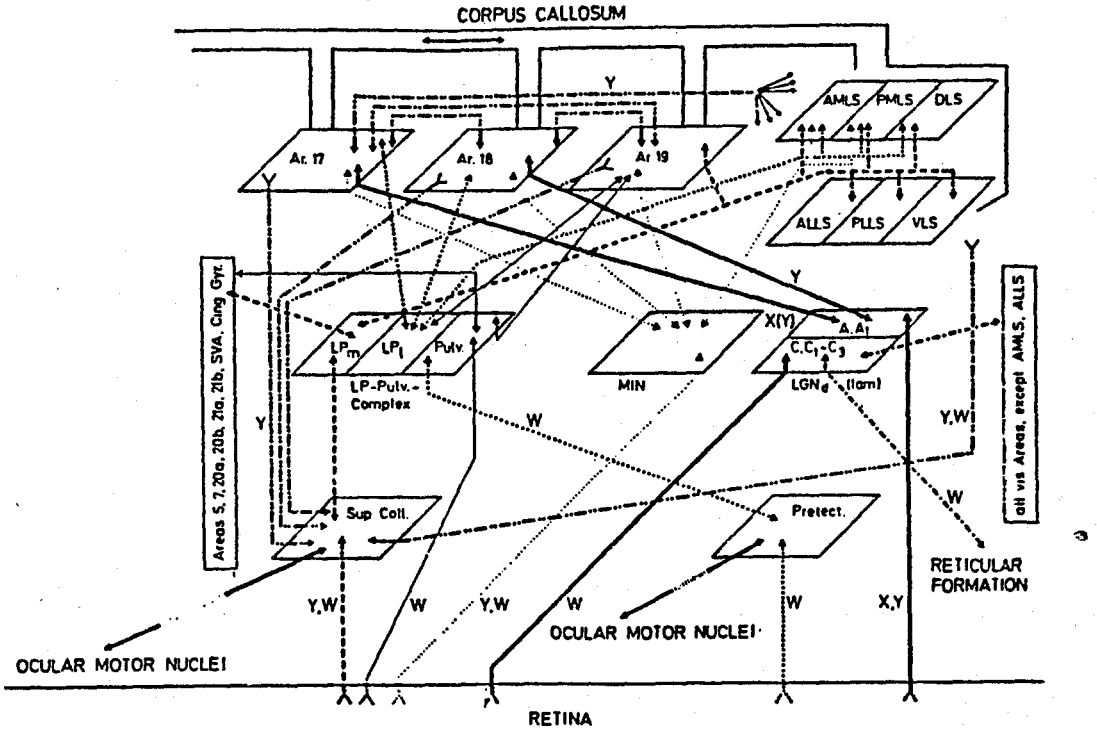

Fig 2. Interconnection schema of the primary visual cortex areas (from [16])

To program a complex network like this by a description of the whole network on the neuron level is not feasable because it is too complicated. This coincidents well with the experience in assembler programming of large programs. Instead of the "assembler style" of programming, as it is supported by conventional neural network programming languages like Spread-3 [2] or NNSIM [12], a structured, modularized approach is needed, as it is partially supplied for instance by the RCS language [4] .

2.0 The User View of INES

The user specifies his system by using the graphical editor shell of INES to interconnect user-defined, graphical units with user-defined lines or vectors. Every unit has several input and output ports and can be composed of interconnected units itself.
The whole structure is a graph, composed of hierarchically structured subgraphs. On every hierarchical level the user has a set of operations [13] like *move, copy, delete, insert...* which work on parts (i.e. a non-zero subset) of the network. The network may be sufficient specified if every input and output is assigned at least one connection.

2.1 Input and Output

The data to be processed are fed into the network by the connection to predefined data sources, represented by the icons of special, predefined input units.
Output data is received from the network by connecting in an analogous manner one output port of one unit to the icon of a special, predefined output unit.

The whole processing system can started by a menu-driven command, e.g. starting an A/D Converter as data source or initiating the reading of a disk file. As a special input data source a random generator is supplied.
In figure 3 a sample screen is shown.

selected set of units of the same type at programming time or store the state of the unit global data for restart purposes.
Each *input/output* connection consistes of a channel which is able to transfer a pattern vector (a number of bytes) at the same time on a parallel data path. The data path width

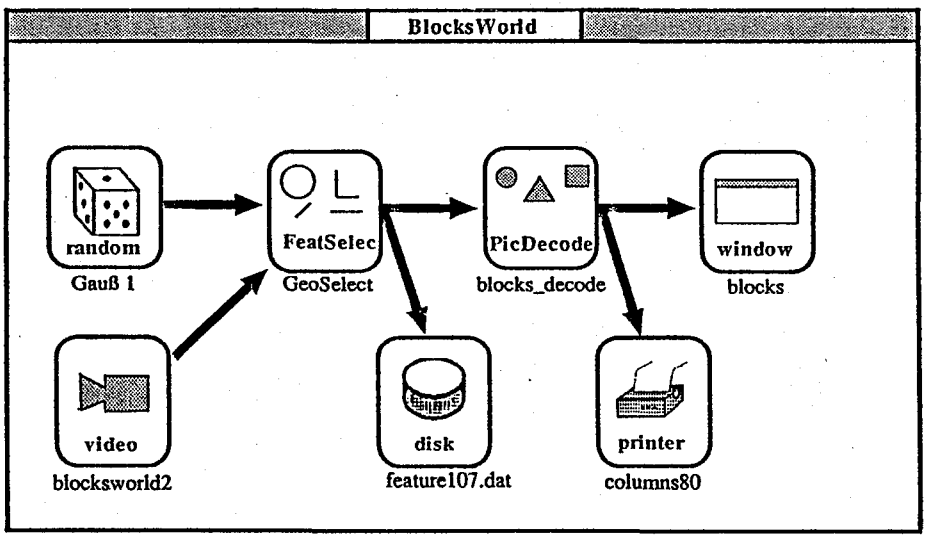

Fig 3 The user view of a network

2.2 The Specification of Units

We started our work from the observation, that the main problem in writing programs for the simulation of Neural Network algorithms is not the code for the basic neural functions but for everything to be set up around it. This is namely the generation of the input/output test pattern, the interconnection between well defined subfunctions ("subnets"), the monitoring of the activity for debugging and research purposes and the recording and the saving of the obtained output and network states.
So we decided to devide the whole programming code into common i/o code and the special function code, and put them into separate, standardized modules ("units") to allow new combinations of them. Every unit is specified by a subset of processing algorithm, globals and input/output parameters.
The *processing algorithm* determines the type of the processing unit; a change in the algorithm leads to a new type of unit. Initially, a set of standard units with standard algorithms like those of associative memory or back-propagation can be used to construct a new unit.
The *global data* segments are set to default values in the standard units. Each copy of a unit yields the same copied constants. The programmer can change the constants for a is a constant, specified by the user.
For the code of a base unit the programmer has to take some programming conventions into account. Since we want to link several data segments to the same code which signifys several instances of the same unit type, we have chosen a pointer structure as appropriate data structure.
The user puts his application code into one procedure, doing all input, output and references to global data of the base unit only by pointer references. Additional library functions provide an easy access to the three kinds of available data.
If a programmed module follows the restrictions above, it does not matter what kind of programming language is used. Principally, even special neural network languages (e.g. [8]) may be used to specify the activity of a net of neurons within a unit module.

2.3 Debugging the Program

The INES tool is dedicated to the developement of systems, models and theories. Therefore, the recording and presentation of intermediate processing results are of high interest. Certainly, a special output unit like a screen or a disk file can be connected directly to every output port of a unit in the network. Nevertheless, the raw output data are

normally difficult to understand and must be preprocessed or decoded (descrambled) [7]. Since all processing is done by units, the user can define special debugging units which can store for example the symbolic names of some patterns.

3.0 The Implementation

The implementation of the graphical specification and simulation language has to take some restrictions into account. These restrictions result from the demand for transparent operation, independance of the available hardware units and variable timing conditions due to the software nature of the implemented algorithm.

We devide the entire network into functional independant, only data coupled units.

We do not support coupling effects (neuro-chemical transducers etc) between the units (e.g. globals) beside the explicitely specified data path.

The principal operations in every base unit are parallel. So we assume that all input data is processed in a base unit in only one time step, either by hardware or by software.

Since there is no special wake-up time which can be specified for a unit, time simulations are not possible at this stage of implementation.

3.1 The Unit Module Concept

It was indicated above that all organizational entieties in the brain are treated by the INES approach as units in the same formal manner. Since units representing a subnet can be decomposed to a net of units, we finally end up with non-decomposable base units.

These units are represented by their algorithm, specified, implemented and compiled in a (most likely) procedural language as Modula-2, C++ or Pascal or any other stack-oriented language.

For instance, we can write some code specifying the function of one neuron, combine some units of that type into one net, use this net as a subnet unit again for the construction of a net of units, and so on until we arrive at a whole layer containing thousands of neurons. We only have to take into account that it takes much more time to simulate a network composed of thousands of neurons than a simulation of a unit representing a whole layer by an efficiently written assembler program or coprocessor chip. It is therefore a good practice to implement well known functional groups (layers) of a model as a single base unit, allowing the combination of the known functions to higher, unknown degrees of architectural complexity.

Software and Hardware Units

We aim to treat both kinds of algorithms, software and hardware units (e.g. coprocessor speed up boards), in the same manner. Since we can not adress hardware directly in an environment of a multi-user system as UNIX, and there is always some special software involved in the communication between the simulator and the hardware unit (the driver) which reflects the pecularitys of the hardware device, we put all driver software into one module which represents a software unit again and hides the hardware from the simulator.

Therefore, the simulator can assume that there exist only software units in the system.

3.2 The Graphical Editor

The graphical editor part of INES with the user interface is designed to be functionally separate from the neural network simulation engine. This is in contrast to many other implementations of neural network simulators, e.g. the popular MacIntosh tools such as Cognitron™ or MacBrain™ or other special purpose tools.

The INES graphical editor contains only the user interface, i.e. the graphical display engine, the menu handler and interactive procedures for moving, copying, inserting and deleting nodes, nets and subnets on the screen. Nets and subnets on any hierachical level can be saved or restored, using unit icons as representations.

The basic graphical functions are implemented using the high-level functions of the EDGE library [11], based on the X-Window system [14].

All special applications, e.g. programs or simulators, are linked to a predeclared directory path and are executable directly from the editor. On invocation the editor passes all its graphical information in a standard data file format to the forked application process. All interaction between the application (e.g. the neural network simulator) and the user is now managed by the application itself, not the editor.

Hence, the graphical editor is functionally separate from the application, allowing a wide range of applications such as printer-programs, simulators for Petri-nets or configuration programs for a general kind of parallel UNIX process pipe systems.

3.3 The Neural Network Simulator

Initialization

The first task of the interpreter initialization process is the setup and initialization of the base software units.
At this stage some very important design decisions must be considered. The first one deals with the question

? should the network of units be implemented as a set of interconnected UNIX processes (every software module is a precompiled program; the unit itself a UNIX process) or as one process, containing the code of the units as procedures?

This design decision is heavily dependant on the features of the available UNIX system. Since we use Berkeley Unix 4.3 a network of UNIX processes poses some problems:

- the interprocess communication can not use fast common memory operations but relays on the reletively slow file mechanism such as sockets, special files etc.

- passing the execution control from the simulator dispatcher to a unit and back again can not be implemented directly but by some send/wait mechanism using global semaphors, dynamic prioritys or file locking mechanism which are neither fast nor multi-user friendly.

Since the use of the operation system primitives is very specific, even in the UNIX system versions, we decided follow the second approach and to implement the units as procedures in one single process.
The second design decision reflects the following problem:

? should the procedures of the units ("unit code") loaded on demand dynamically into the simulator or should the unit code be linked statically to the interpreter code?

The first alternative needs relocatable code and is essentially a linking-loading operation. Because there are certainly some system library procedures used which have global variables (e.g. C-library functions for file i/o), the interpreter-simulator has to do the same thing as the standard system linker/loader. For this reason we chose the second alternative and let the job better be done by the standard system tools.
The initialization of the interpreter is therefore constructed as a separate initialization process which gets the list of all unit types, sets up and forkes off a system linking command and then executes the newly built, application customized interpreter-simulator, containing all unit type code modules, an address list of them and the interpreter core module.

Simulation

After the initialization of the interpreter and the units the interpreter starts the simulation. For this purpose the interpreter selects a unit according to the strategy discussed below, sets up the input and output pointers of the unit module according to the connection graph, pushes the arguments for the procedure call (see section 3.1) on the stack and then transfers the control to the unit by executing a procedure call. After the execution of the algorithm the interpreter receives the control and looks for the next unit to simulate or deals with newly arrived commands from the interrupting user. In figure4 a sample configuration of four units is shown.

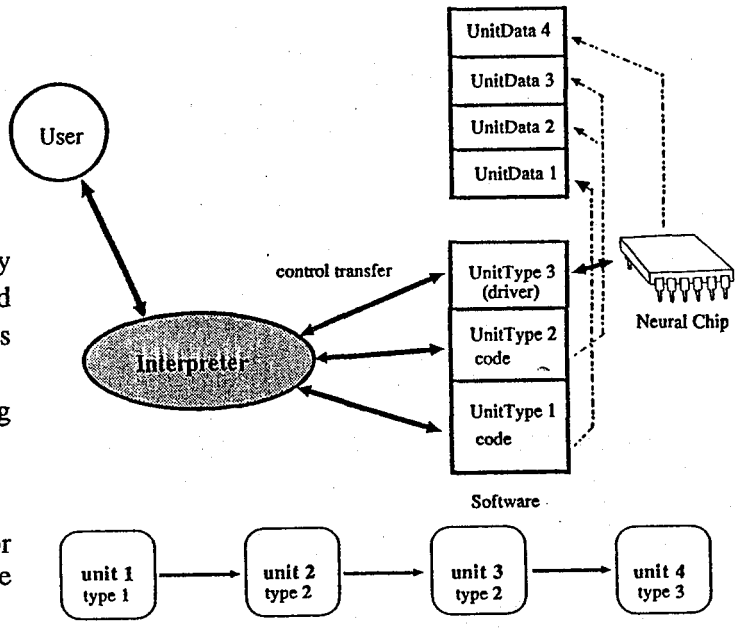

Fig 4 The interpretation and simulation of a network

The Simulation Strategy

The simulation takes every base unit module as a simulation module. Since the information processing is done in the sequence of the units, the most simple approach is to let a data pattern flow sequentially through all the units. But this strategy leads to problems when feedback loops are programmed; in this case the loop is continued while other parts of the network are "frozen".
Another approach might put all units in a list to guarantee an equal degree of activity in the whole network. This deterministic approach is very efficient, but can lead to propagation effects due to the numbering of the units.
A random approach guarantees us a random sequence of the units, but the equal pattern processing activity only in the mean.
As a compromise between the neutrality of the random ategy and the efficiency of the deterministic approach, we take as simulation strategy the rule that the execution sequence of the units is taken by a list containing the unit numbers in a randomized sequence.

4. Conclusion

The INES approach for Neural Network simulation supports the modularization of neural nets into functional units which have less communication (connectivity) between the units than within them.
This approach is in good coincidence with the description of AI and brain functions by functional layers.
Since the graphical editor, the network interpreter and the modules of the base units are independantly defined, the user can program his own base units (and even his own interpreter) in the programming language of his choice.
By the decomposition into seperate units the classical, historically grown (Fortran-based) simulation program which can only be used for one purpose becomes a collection of reusable, clearly structured software units. Standard problems like I/O, filter functions and monitors have to be implemented only once: the software productivity is increased.
Thus the whole system is highly flexible and can be extended and taylored to the user's needs and even used for other problems than that of Neural Network simulation.

References

[1] R. Brause, "Fehlertoleranz in intelligenten Benutzer schnittstellen", *Informatik Technologie* 3, Oldenbourg Verlag München 1988

[2] J.Diederich, C.Lischka, "Spread-3. Ein Werkzeug zur Simulation konnektionistischer Modelle auf Lisp-Maschinen", *KI-Rundbrief* Vol 46 pp.75-82, 1987

[3] K. Fukushima, "A Neural Network Model for selective Attention in Visual Pattern Recognition", *Biological Cybernetics* 55, p.5-15, Springer Verlag 1986

[4] N.Goddard, "The Rochester Connectionist Simulator, User Manual and Advanced Programming Manual", Dep. of Computer Sci., University of Rochester, April 1987

[5] D.Hillis, "The Connection Machine", MIT Press, Cambridge, Massachusetts, 1985

[6] J.J.Hopfield, "Neural Networks and physical systems with emergent collective computational abilities", *Proc. Natl. Acad. Sci.*, USA, 79, pp.2554-2558

[7] J.Kindermann, "Inverting Multilayer Perceptrons", Proc. DANIP Workshop on Neural Networks, GMD-St.Augustin, April 1989

[8] T.Korb, A.Zell, "A declarative Neural Network Description Language", Proc. Euromicro, Cologne 1989, *Microprocessing and Microprogramming*

[9] R. Linsker, "From Basic Network Principles to Neural Architecture", *Proc. Natl. Acad. Sci.*, USA, Vol 83, pp. 7508-7512, 8390-8394, 8779-8783

[10] H.Marko, "A Biological Approach to Pattern Recognition", *IEEE Transactions on Systems, Man and Cybernetics* Vol SMC-4/1, January 1974

[11] F.Newbery, "EDGE: An Extendible Directed Graph Editor", Internal report 8/88, University of Karlsruhe, W.-Germany

[12] J.Nijhuis, L.Spaanenburg, F.Warkowski, "Structure and Application of NNSIM: A General Purpose Neural Network Simulator", Proc. Euromicro, Cologne 1989

[13] D.Smith, C.Irby, R.Kimball, B.Verplanck, "Designing the Star User Interface", *Byte*, April 1982, pp.242-282

[14] R.W.Scheifler, J.Gettys, "The X window system", *ACM Transactions on Graphics*, 5(2), April 1986

[15] J. Szentagothai, "The 'Module-Concept' in Cerebral Cortex Architecture", *Brain Research*, 95, pp.475-496, Elsevier Sci. Publishing Company, Amsterdam 1975

[16] Segraves,Rosenquist, "The afferent and efferent callosal connections of retinotopically defined areas in cat cortex", *J.Neurosci*, vol 8, pp.1090-1107