MiND: An Environment for the Development, Integration, and Acceleration of Connectionist Systems[1]

Gerd Kock
GMD FIRST Berlin
Rudower Chaussee 5
12489 Berlin, Germany

Thomas Becher
INCO Systeme GmbH
Stöhrerstrasse 17
04347 Leipzig, Germany

Keywords: Artificial Neural Networks, Simulation, Hardware, Software

ABSTRACT

The system MiND (Multipurpose integrated Neural Device) is a development system for artificial neural networks. It includes a neuroboard based on the new neuroprocessor SAND (Simple Applicable Neural Device), which can be used to accelerate networks like Backpropagation, Radial-Basis-Function or Kohonen networks (up to 800 MCPS). The user interface of MiND is menu and graphic oriented. The system is supplied with many "predefined" simulators. However, based on the specification language CONNECT and on abstract C++ graphic and menu classes a user can easily adapt predefined simulators or develop own ones.

The language CONNECT is based on a generic connectionist model. It allows to specify connectionist systems in an abstract and, at the same time, complete way. I.e., all relevant aspects (e.g. learning algorithms) are given explicitely, but in a compact and readable way. A CONNECT specification is translated into a C++ network class and then can easily be glued together with the graphic and menu classes to constitute a simulator. All "predefined" simulators of the MiND system are based on this mechanism. However, this mechanism can also be exploited by a user to define his own simulators and to develop custom applications.

The MiND system will be available on different platforms. The current version is running on PCs under Windows 95 and Windows NT, and includes a PCI board with the neuroprocessor.

INTRODUCTION

The demands on simulation tools for artificial neural networks are manifold. Most users ask for a comfortable graphical interface aiding in developing, analysing, and applying networks. In this, standard network models should be provided as predefined components of the system. For other users, flexibility and extendability is essential. E.g., it should be possible to modify existing learning algorithms or to include selfdefined ones. If a tool is used in the context of a complex project development, support for custom applications should be given. E.g., after having trained and tested a neural component, it should be easy to integrate this component into an external environment. Last but not least, pragmatic aspects like scalability and performance play a role. The latter aspect may be supported well by specific hardware only.

The simultaneous fulfilment of the items from above is a non trivial task. For discussing this matter in more detail, let us, at first, observe that simulation tools for artificial neural networks roughly can be divided into three categories [3,4]: (1) systems, the first design goal of which have been a comfortable graphical user interface, (2) systems which provide a user with a large library of modules written in languages like C or C++, and, finally, (3) systems which offer a special programming language for artificial neural networks.

[1]IMACS'97, 15th World Congress 1997 on Scientific Computation, Modelling and Applied Mathematics, August 24–29, 1997, Berlin, Germany

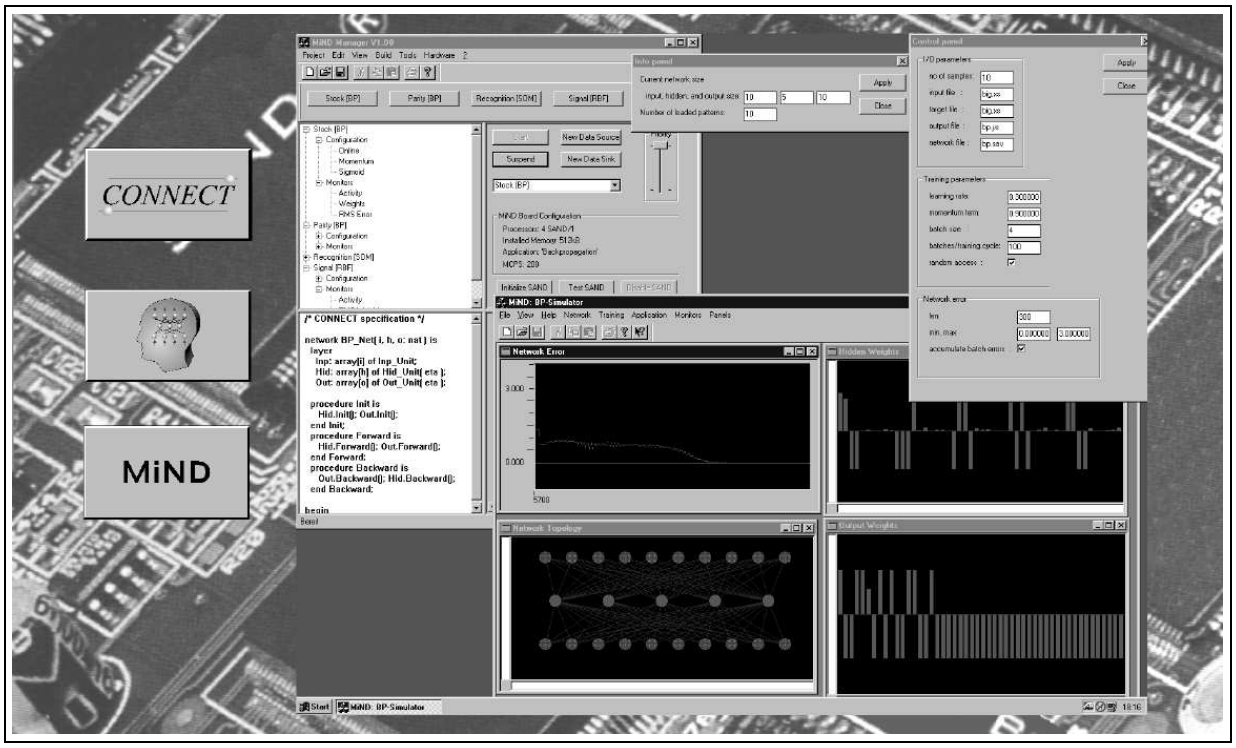

Figure 1: *The MiND system*

A fundamental problem with many systems is flexibility and extendability. The systems of the first category are build around internal data structures which, when new models have to be supported, may turn out to be too restrictive. Anyway, for a user, in general, it is a too complex task to deal with the structures behind the graphical user interface. Libraries of C or C++ modules conceptually are flexible and extendable. However, to exploit this advantage, detailed knowledge is necessary, the development of which takes time. Systems based on special neural network programming languages may be a good solution, if the language is based on an appropriate generic connectionist model, and if networks easily can be equipped with a graphical interface. However, some languages are not based on a connectionist model, but in fact are vector and matrix manipulation languages, and in other cases the underlying connectionist model is too restrictive or the programming language concepts used to "implement" a language model are too low level.

The goal in the development of the MiND system was to fulfil all demands from above. The system includes the neural network specification language CONNECT [2], a library of graphical user interface classes, and a PCI neuroboard based on the neurochip SAND [1]. In the next section, the overall structure of the MiND system is explained. After that, the SAND neurochip and the PCI neuroboard are outlined. Then, the CONNECT language is presented in more detail. The paper is closed by an evaluation of the MiND system.

MIND OVERVIEW

The components of the MiND system are: (1) the MiND PCI neuroboard which is equipped with up to four SAND neurochips, (2) the CONNECT specification language and a compiler which generates C++ classes from CONNECT specifications, (3) a collection of abstract graphical user interface (GUI) classes, and (4) the MiND manager which administrates the pool of predefined and selfdefined simulators.

The MiND manager and each simulator administrated by the MiND manager have comfortable

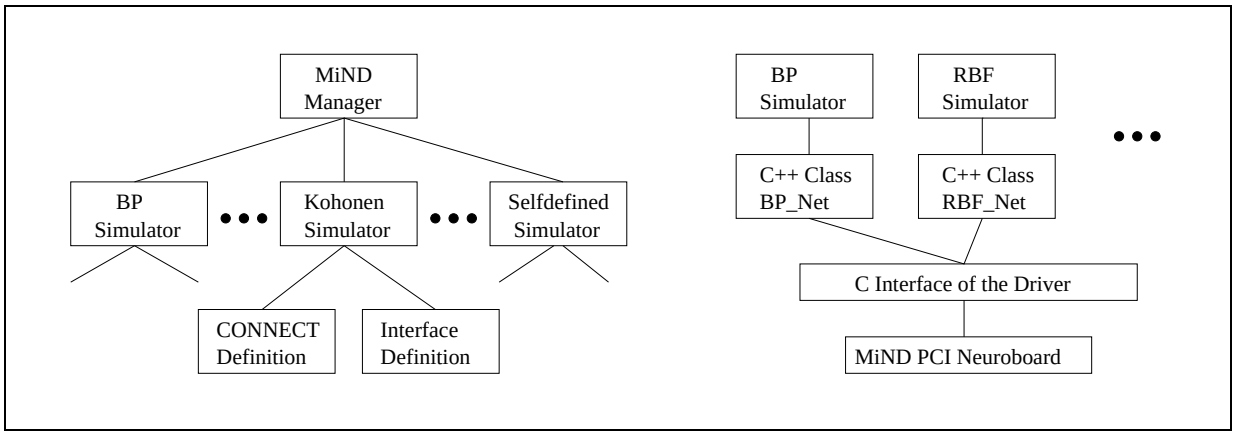

Figure 2: *MiND architecture and software layers for the PCI neuroboard*

graphical user interfaces. Each simulator is constituted by a CONNECT definition and an interface definition script (Figure 2). The CONNECT definition describes a neural network. The interface definition is build on the C++ class generated from the CONNECT definition and on the MiND GUI classes. Among the GUI classes, there are classes for menus, dialogue boxes, panels, and monitors for observing elements like network weights etc.

A user is allowed to modify or extend CONNECT network specifications and interface scripts. This specifically opens up the way for extending the pool of predefined simulators by selfdefined neural algorithms. The development of custom applications is supported twofold. It is possible to export a generated C++ network class or to develop a (prototype) application by modifying the graphical user interface of a simulator.

Figure 2 also shows the software layers for the PCI neuroboard. The driver's C interface offers direct access to the functionality of the MiND PCI neuroboard. From the user's point of view it is a simple to use software interface and hides all hardware specific aspects. In addition, for all networks supported by the neurochip, the corresponding simulators can be used to exploit the PCI neuroboard via a graphical user interface. This is achieved by adapting the generated C++ network classes on the base of the driver's C interface. Note that a C++ network class using the neuroboard can be exported like any other C++ network class.

THE SAND NEUROCHIP AND THE PCI NEUROBOARD

The SAND neurochip [1] supports fully connected feedforward networks, RBF networks, and Kohonen feature maps. To give an impression about what calculations are performed by SAND, the organization of the neural processor for feedforward networks is outlined.

The calculations performed by a complete layer can be described as a matrix/vector multiplication of form $y = f(W * x)$. In this, x is the input vector of the layer, W is the weight matrix holding all connection weights between two consecutive layers, f is the activation function, and y is the output of the complete layer. If, instead of one input activation vector x, a number of vectors is considered, then the equation from above turns into a matrix/matrix multiplication $Y = f(W * X)$. And this is what SAND does: it is a systolic array with four processing elements (PEs) and treats four neurons and four input activation vectors at the same time.

Every cycle, one weight and one activation are transfered. Each PE is working four cycles with the same weight. The activations are transfered through registers from one PE to the next. As a result, four PEs compute 16 multiplications within 4 cycles. There is a continuous

flow of data on both the activity and the weight bus.

To support data and command handling in the way described above, each PE needs a multiplier and an adder for the calculations sketched above. For RBF networks and Kohonen networks it is necessary to calculate the Euklidean distance between two vectors. Therefore SAND's PEs are equipped with a second adder. The two adders and the multiplier are placed in a pipeline.

The MiND PCI neuroboard contains up to four SAND chips running at 50 MHz. The performance can be scaled from 200 MCPS (one SAND processor) up to 800 MCPS (four SAND processors). Beside the neurochips the board is equipped with memory blocks for activities and weights, with a lookup table to store the activation functions and with controllers.

```
SAND_Load_LUT( sigmoid );
SAND_Load_MLP1( i, h, o, ws_h, ws_o, bs_h, bs_o );
SAND_Apply_Net( n, x );
while ( !SAND_Read_MLP( y ) ) { ... }
```

Figure 3: *The driver's C interface*

Figure 3 sketches the simplicity in which the neuroboard can be applied. At first, a sigmoidal function is used to initialize the lookup table. Then, a feedforward network (size `i-h-o`) with one hidden layer is laoded; `ws_h`, `ws_o`, `bs_h`, and `bs_o` are pointers to fields of weights or bias variables, respectively. After that, the network is applied to the field `x` containing `n` input vectors. At the end, the results are written into the field `y`.

THE CONNECT DEFINITION OF A RBF NETWORK

The CONNECT language is based on a generic connectionist model. Each CONNECT specification consists of a number of unit type definitions followed by a network type definition. In addition, procedures, functions, and relations can be defined. Unit and network types are defined in an object oriented manner, relations and functions are defined in a functional style. Relations are collections of index pairs used for connecting (and disconnecting) layers of units.

A unit type is defined from a "local" point of view. Fanout signals, input and output vectors describe the I/O behaviour of a unit, local variables are used to provide for local memory (e.g. for weights), and unit activation procedures are used to define the possible activations of units. On the other hand, a network type is defined from a "global" point of view. It consists of a number of layers, which are (multidimensional) arrays of units, other variables, network activation procedures, and an initialization part, which can be considered as a special network activation procedure. Connect and disconnect statements can be used to dynamically establish and modify topologies. During the execution of these statements, I/O entities of units are (dis)related with each other. Using a dot notation, a network activation procedure can execute a certain unit activation procedure for all units of a layer. Similarly, a certain I/O entity or variable defined in a unit type can be accessed simultaneously for all units of a layer.

As an example, consider the Radial–Basis–Function network `RBF_Net` in Figure 4. The training procedures of this network are defined according to [5], except for the fact that `RBF_Net` provides for online learning instead of batch learning. The specification starts with the declaration of some functions and a relation. Function `dist` computes the Euklidean distance, function `min` extracts the minimal component of a vector, `gauss` implements a Gaussian kernel, `sum` accumulates vector components, and relation `Full` relates all pairs of indices.

`RBF_Net` has three layers `Inp`, `Hid`, and `Out`. An input unit has a fanout signal `y` only (outgoing signal for hidden units). A hidden RBF unit has connections with input units (port `I`) and output units (port `O`). In addition, hidden units are connected with each other (via fanout signals `d` and input vectors `ds`). The input vector `I.x` (incoming signals from the input units)

is accompanied by the vector `I.c` standing for the center of the RBF unit. The center and width `w` characterize an RBF unit. Fanout signal `O.y` (outgoing signal for output units) is accompanied by an input vector `O.err` (incoming error signals from output units). An output unit is simpler than a RBF unit. It has a fanout signal `y` (network result) and a port `I` used for connections with the hidden layer. Here, input vector `I.x` is accompanied by an output vector `I.err` (outgoing error signals for the hidden units) and a weight vector `I.w`. The weight vector and the bias `b` are the coefficients of the linear mapping performed by the output unit. Variable `t` is used for storing the training target. The runtime parameter `eta` of both the hidden and output units stands for the learning rate.

```
// functions and relations

function dist: vector of real,vector of real -> real;
   ...
function min: vector of real -> real;
   ...
function gauss: real, real -> real;
   gauss(d,w) = exp( -(d*d)/(w*w) );
function sum: vector of real -> real;
   sum(<>)     = 0;
   sum(<x:xs>) = x + sum(xs);

relation Full;
   x Full y;

// unit types

unit Inp_Unit is
   fanout y : real;
end Inp_Unit;

unit RBF_Unit(rt eta: real) is
   port O is
      fanout  y : real;
      input err : vector of real;
   end O;
   fanout d : real;
   input ds : vector of real;
   port I is
      input x : vector of real;
      var   c : vector of real;
   end I;
   var w : real;
   procedure Distance is
      d := dist(I.x, I.c);
   end Distance;
   procedure Adapt is
      if (d = min(ds)) then
         I.c := I.c + eta*(I.x-I.c);
         w   := (1-eta)*w + eta*2*d;
      end;
   end Adapt;
   procedure Forward is
      d   := dist(I.x, I.c);
      O.y := gauss(d, w);
   end Forward;
   procedure Backward is
      var delta : real;
   begin
      delta := 2*O.y*sum(O.err);
      I.c   := I.c + (eta*delta)/(w*w)*(I.x-I.c);
      w     := w   + (eta*delta*d*d)/(w*w*w);
   end Backward;
end RBF_Unit;

unit Out_Unit(rt eta: real) is
   fanout y : real;
   port I is
      input    x : vector of real;
      output err : vector of real;
      var      w : vector of real;
   end I;
   var b, t : real;
   procedure Init is
      I.w := 0; b := 0;
   end Init;
   procedure Forward is
      y := I.w*I.x + b;
   end Forward;
   procedure Backward is
      I.err := (t-y)*I.w;
      I.w   := I.w + eta*(t-y)*I.x;
      b     := b   + eta*(t-y);
   end Backward;
end Out_Unit;

// network type RBF_Net

network RBF_Net (i, h, o : nat) is
   var eta : real;
   layer
      Inp : array[i] of Inp_Unit;
      Hid : array[h] of RBF_Unit(eta);
      Out : array[o] of Out_Unit(eta);
   procedure Init_Hidden(i: int; x: array of real) is
      Hid[i].I.c := x; Hid[i].w := 0.1;
   end Init_Hidden;
   procedure Init_Output is
      Out.Init()
   end Init_Output;
   procedure Adapt_Hidden(x : array of real) is
      Inp.y := x;
      Hid.Distance(); Hid.Adapt();
   end Adapt_Hidden;
   procedure Adapt_Output(x,t : array of real) is
      Inp.y := x; Out.t := t;
      Hid.Forward(); Out.Forward();
      Out.Backward(); Hid.Backward();
   end Adapt_Output;
   procedure Apply(x: array of real):array of real is
      Inp.y := x;
      Hid.Forward(); Out.Forward();
      return Out.y;
   end Apply;
begin
   connect Hid.d with Hid.ds using Full;
   connect Hid.I with Inp.y  using Full;
   connect Out.I with Hid.O  using Full;
end RBF_Net;
```

Figure 4: *CONNECT definition of a RBF network*

The network topology is established in the initialization part of network `RBF_Net`. The first

connect statement means that fanout signal `d` of unit `Hidden[i]` is connected to input vector `ds` of unit `Hidden[j]`, iff relation `[i] Full [j]` is fulfilled — which for `Full` always is the case. Accordingly, the other connect statements have to be understood. Note that connect (and disconnect) statements can be executed in network activation procedures.

`RBF_Net` contains five network activation procedures. `Init_Hidden` can be used to initialize the centers of RBF units with different input examples. `Init_Output` initializes the output layer. In this, the call `Out.Init()` executes the unit activation procedure `Init` for all output units. There are two training procedures. At first, `Adapt_Hidden` can be used to estimate the centers and width factors of the RBF units. The centers are calculated by a LVQ1 algorithm and the width of a unit is taken to be twice the "mean distance" between the center and those input vectors "belonging" to the unit. Secondly, `Adapt` can be used to tune the centers and width factors of the hidden and the coefficients of the output units by gradient descent.

The CONNECT compiler generates a C++ class `RBF_Net`. This class can be instantiated (e.g. `RBF_Net net(10,100,3)`), and then its methods can be called (e.g. `net.Adapt(x,t)`).

DISCUSSION

The MiND manager and the simulators administrated by the MiND manager have comfortable graphical user interfaces. Flexibility and extendability is supported by the concept that each simulator is described by a CONNECT network definition and an interface definition based on the generated C++ network class and on abstract GUI classes. The two definition scripts constitute a level of abstraction which enables a user to define own neural algorithms and interfaces for his needs. The development of custom applications is supported specifically by the possibility to export the generated C++ classes. Using these classes, an application not only can apply a trained network, but also can retrain it etc. Finally, the integrated PCI neuroboard can be used to accelerate networks. Three software layers encapsulate the neuroboard: the driver's C interface, C++ network classes, and MiND simulators built on these classes.

It can be concluded that the MiND system fulfils the demands on neural network simulators established above, and that it is a useful tool for education, research, and industry.

ACKNOWLEDGEMENTS

Many thanks go to K. Tiedemann (involved in the CONNECT development), D. Heinrich (GUI classes and MiND manager), for the SAND neurochip H. Gemmeke, W. Eppler, T. Fischer (all from FZK) and S. Neusser (IMS), and for the PCI neuroboard H. Runkewitz.

REFERENCES

[1] H. Gemmeke; W. Eppler; T. Fischer; A. Menchikov; S. Neusser: Neural Network Chips for Trigger Purposes in High Energy Physics, *Proceedings of Nuclear Science Symposium (NSS)* (1996).

[2] G. Kock; N.B. Šerbedžija: Artificial Neural Networks: From Compact Descriptions to C++, *Proc. of the Int. Conf. on Artificial Neural Networks*, pp. 1372–1375 (1994).

[3] G. Kock; N.B. Šerbedžija: Simulation of Artificial Neural Networks, *Systems Analysis — Modelling — Simulation (SAMS)* **27**(1):15–59 (1996).

[4] J.M.J. Murre: Neurosimulators, *The Handbook of Brain Theory and Neural Networks* (Michael A. Arbib, Ed.), MIT Press (1995).

[5] M. Verleysen; K. Hlavackova: An optimized RBF network for approximation of functions, *Proc. of the European Symposium on Artificial Neural Networks, ESANN'94* (1994).