THE MetaNet NETWORK ENVIRONMENT FOR THE DEVELOPMENT OF MODULAR NEURAL NETWORKS

Jacob M.J. Murre and Steven E. Kleynenberg[1]

Leiden University
Unit of Experimental and Theoretical Psychology
P.O. Box 9555, 2300 RB Leiden
The Netherlands

Abstract The MetaNet network environment allows users to build and examine modular neural networks, and to specify and run complex simulations. It consists of a graphical editor, a network compiler and a graphical (de)compiler, a network specification language (*MetaNet*), and hardware drivers. Its requirements are based on experiences with a text based network environment, which has been in use at our department since early 1988. Using the environment requires minimal programming experience. Currently, the system is implemented only on PCs. Off-loading of calculation processes to other machines is achieved through hardware drivers. It is possible to convert MetaNet code into ANSI C for direct compilation to stand-alone applications on any machine.

Writing code for the implementation of large neural networks can be a tedious and time consuming process. The *MetaNet network environment* emerged from our efforts to provide a general environment for the development of modular neural networks. The primary aim was that users with few programming skills be able to easily specify, run, and evaluate complicated simulations. First, we shall describe MetaNet's predecessor. Then, the objectives of the current environment will be described, which were largely based on experience with the older system. Finally, we shall give a brief description of the different components of the environment.

MetaNet's predecessor: the CALM development system

The MetaNet network environment is based on almost two years of experience with a system that was developed in early 1988: the *CALM development system*. (CALM stands for Categorizing And Learning Module). This system was build primarily for the simulation of CALM networks. CALM networks are learning neural networks that consist of CALM modules (for a general description, see Murre, Phaf, and Wolters, 1989; Murre, Phaf, and Wolters, 1990; Murre, Phaf, and Wolters, submitted; for theoretical and practical applications, see Happel, Phaf, and Murre, 1990; Phaf, Postma, and Wolters, submitted; Phaf, Postma, and Wolters, 1990). A CALM module can categorize and learn an arbitrary activation pattern. To explain the requirements of the system we shall briefly review the network architecture of CALM. A CALM module consists of two rows of n nodes, R(epresentation)-nodes and V(eto)-nodes, and two single nodes, the A(rousal)-node and the E(xternal)-node. We call n the size of a CALM module. All of these nodes are connected by non-modifiable connections according to a certain wiring scheme, as a result of which learning and categorization in CALM are sensitive to the novelty of the input activation pattern. Interconnecting two CALM modules F and T means connecting all R-nodes in F to all R-nodes in T. These connections are always learning.

Designing CALM networks invites thinking in terms of modules. One of the main objectives of the CALM development system, therefore, was to enable users to generate networks on the basis of high level (modular) specifications. For this purpose an interactive component was developed with a textbased user interface. Some of the

---

[1] The second author is also a partner in the Kleynenberg & Kleynenberg Company.

Copyrighted material

features of the interactive component of the CALM development system were:

- Generating a CALM network on the basis of a high level specification
- Saving and loading a network, including modified connections and activations
- Inspecting a network: activations, connections, modular structure, parameters, node status
- Changing a network: activations, parameters, node status
- Initializing a network: activations, connections, parameters
- Running a network: one cycle at a time, *i* cycles, until convergence

The interactive component was adequate for designing and testing CALM networks. It did not suffice, however, for extensive simulations of, for instance, psychological experiments. Therefore, the system was extended with a non-interactive, batch type component. Files with often used commands could be prepared in advance using a few simple syntactic rules. These command files could then be run from the interactive component. The results could be observed directly, or the file could be processed in a background process and be evaluated after the job had ended. In addition to this list of features, the non-interactive component also supported the specification of patterns (stimuli) in terms of node activations and status, as well as a variety of output formats. A minimal control structure was available for repetition of blocks of commands. The CALM development system has been used by numerous users during a two-year period. It has been implemented on VAXs (VMS), on PCs (MS-DOS), on Amiga computers (Amiga-DOS), and on T800 Transputer networks.

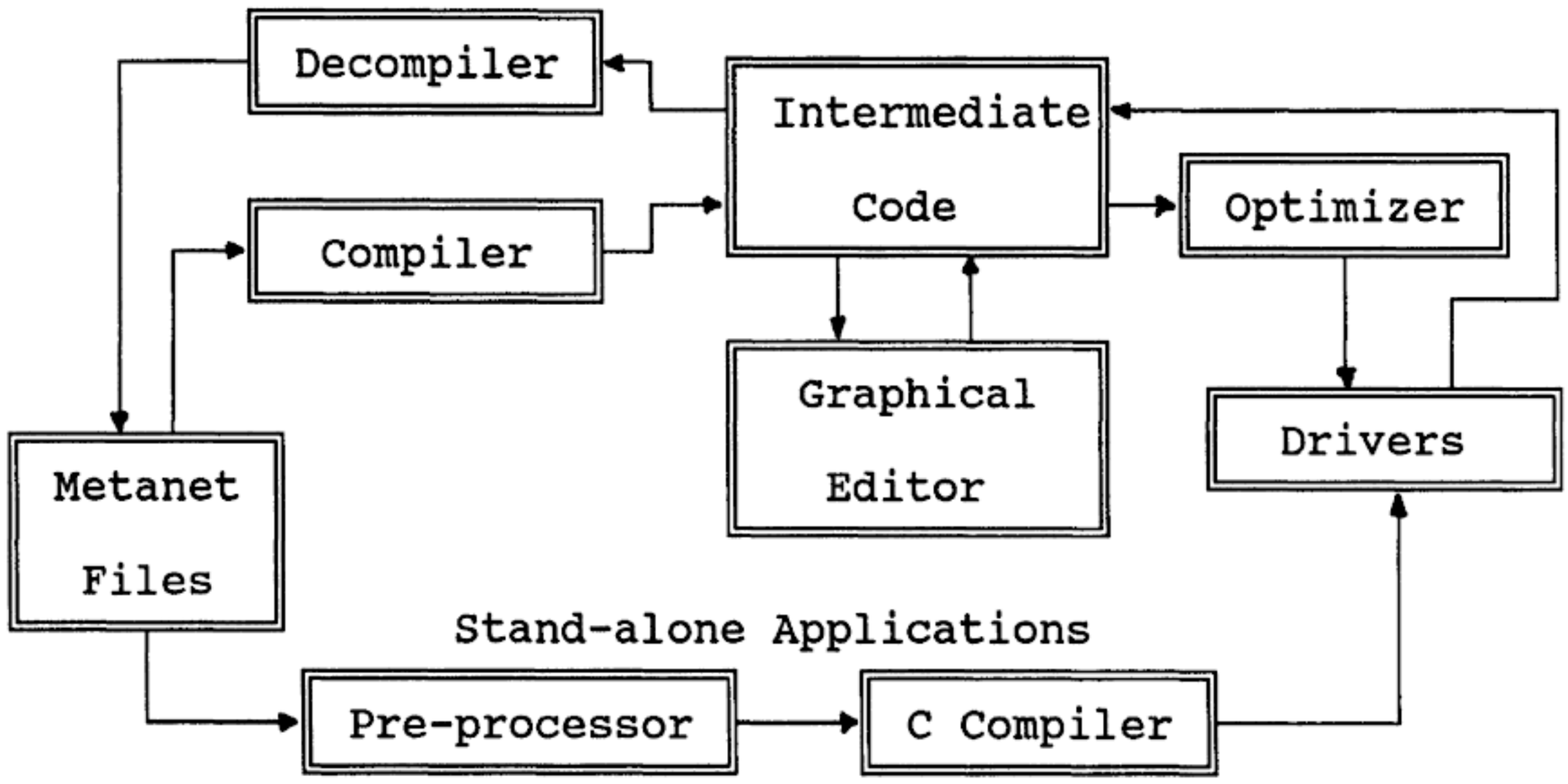
The MetaNet system

The CALM development system had a number of limitations: (1) It lacked graphical facilities. (2) The control structure of the non-interactive facility was very limited. (3) It could only run CALM type networks and it did not allow experimentation with other networks. (4) The system itself was very large, so that only relatively small networks could be run on a PC. (5) It was not possible to control processing in various kinds of hardware by a single system. The need for this was prompted, among others, by the development of a 400-processor network implementation of CALM (see Hoekstra, Heemskerk, Klaassen, Phaf, Knoppers, and Hudson, 1990). To overcome these limitations, the MetaNet network environment provides the following. (1) A graphical interface, which includes a graphical editor. (2) A graphical compiler and a network compiler. (3) MetaNet, a higher programming language for the specification of networks and simulations. (4) The option to convert MetaNet code to ANSI C code for compilation to stand-alone applications. (5) Hardware drivers to buffer the communication in the various processes, possibly on different machines. The MetaNet system is schematically drawn in the figure. A brief description of each component follows below.

Graphical editor. The current implementation of the graphical editor is aimed at PCs running MS-DOS. We judged X-Windows to be too slow on a PC for our purposes. Instead, we have tried to keep the system portable by using a graphical shell with functions found in most other graphical programming environments. The editor is event driven, and it is operated more or less as a 'paint' program. It enables the user to graphically edit a network. Nodes, modules, and other objects may be placed on the screen, and (generalized) connections formed. Objects are manipulated by means of a mouse. Activations, weights, thresholds, etc., can easily be adjusted. Activation, learning, and transfer functions can be chosen from a menu. Experienced users may define their own functions. Hooks with graphs or other status reports may be placed on objects, such as nodes and connections. A special feature is the possibility to automatically position objects using an 'charged-particle algorithm', in which the objects mimic the motions of charged particles.

Copyrighted material

The MetaNet Network Environment



Compiler/Decompiler. Networks and complete simulations may be specified in MetaNet, a high level network language (see below). Inside the MetaNet environment networks and control flow are represented in *intermediate code*. This code is processed by an optimizer to increase performance, and is subsequently send to the hardware drivers, which actually run the networks. To convert a MetaNet file into intermediate code a graphical (de)compiler has been developed. Graphical compilers have been built, among others, for the purpose to directly convert logical structures into object code. An example is the GRASPL programming system, which generates code for micro controller chips based on direct compilation of state diagrams drawn by the user (Zieleman, 1988). In a similar fashion, the graphical decompiler enables the user to graphically edit networks and convert them to MetaNet files. It is also possible to go the other way. A network compiler takes MetaNet files as input and generates intermediate code. This code can directed to the graphical editor, where it can be changed (and possibly decompiled to MetaNet again), or it can be send directly to the optimizer and drivers. The results of the ongoing calculations can be directed to the graphical interface without enabling the editor. In this way, demos can easily be built. A version of the network compiler is implemented separately as a preprocessor, which converts MetaNet to ANSI C for compilation to stand-alone applications.

MetaNet language. MetaNet is a simple network specification language. It is a subset of C extended with network specific syntax, which has an object-oriented flavor. Networks can be defined by using the following MetaNet objects: *NODE*, *GROUP* (of nodes), *MODULE*, and *MODEL*. Each of these types has specific properties. They form a strict hierarchy (a *MODEL* consists of *MODULEs*, which consist of *GROUPs*, which consist of *NODEs*). *GROUPs* roughly correspond to layers, such as used in back-propagation and other layered networks. MetaNet supports a recursive structure as well, the *NETWORK*, which may consist of *NODES* or other *NETWORKs*. MetaNet's network specific syntax focusses more on structures than on processes. The idea is that for calculations (activation and learning rules, etc.) and sequential control (iterations and conditionals) a small subset of C is more appropriate. An example of MetaNet, defining the internal wiring scheme of the CALM module, is given below. Generalized connections between object (*GROUPS*, *MODULES*, etc.) are expressed using the symbols ->, -+, and -\. In this syntax, A -> B : 1.0 means: A is connected to B in an all-to-all fashion with weights 1.0. For instance, if A and B are *GROUPs*, all nodes in A are connected to all nodes in B. The -+ operator indicates one-to-one connections and the -\ operator all-to-all-but-one, the corresponding nodes are not connected.

Copyrighted material

```
typedef MODULE (int size)          /*  define a new MODULE (sub)type      */
{
    GROUP   R[size],               /*  R(epresentation nodes              */
            V[size],               /*  V(eto) nodes                       */
            A[1],                  /*  A(rousal) node                     */
            E[1];                  /*  E(xternal) node                    */
    PARAMETER mu : .005;           /*  learning parameter                 */

    EXTERN -> R : LEARN;           /*  learning intermodular connection   */
    R -+ V : 1.0;                  /*  R-nodes feed V-nodes (one-to-one)  */
    V -\ V : -1.0;                 /*  V-nodes inhibit each other         */
    V -\ R : -10.0;                /*  V-nodes inhibit neighboring R-nodes*/
    V -+ R : -1.0;                 /*  V-nodes inhibit R-nodes            */
    V -> A : -.6;                  /*  V-nodes inhibit A-node             */
    R -> A : .4;                   /*  R-nodes excite A-node              */
    A -+ E : 1.0;                  /*  A-node feeds E-node                */
    E -> R : RANDOM(0.0,1.0);      /*  random activations to R-nodes      */
} CALM_TYPE;                       /*  name of new MODULE (sub)type       */
...
GROUP       input[5];              /*  create GROUP                       */
CALM_TYPE little[10];              /*  create MODULE of CALM_TYPE         */
...
input -> little : .6;              /*  connect with initial weight .6     */
```

Network drivers. To increase machine independence hardware drivers are used. This also saves memory, because only the relevant drivers need to be loaded. The drivers buffer network calculation processes, possibly running on remote machines. Drivers may also do calculations themselves. The communication protocols with the other components of the MetaNet environment are handled by means of intermediate code used inside the MetaNet system. Networks may also be dumped directly (full dump) in a file using intermediate code.

The different constituent parts work together to provide an optimal performance environment, both for inexperienced and advanced users. The MetaNet environment still has to pass several testing cycles, so that we cannot yet reach a conclusion about its range of applicability. We expect that modular, or similarly structured, neural networks will benefit most from the approach taken in MetaNet.

References

Happel, B.L.M, R.H. Phaf, and J.M.J. Murre (1990) Categorization in multi-module CALM networks: recognition of handwritten digits. *INNC-90*, Paris, July 9-13.

Murre, J.M.J., R.H. Phaf, en G. Wolters (1989) CALM networks: a modular approach to supervised and unsupervised learning. *IJCNN-89*, Washington D.C., June 18-22, 1, 649-656.

Murre, Phaf, and Wolters (1990) Novelty dependent categorization and learning in CALM modules. *INNC-90*, Paris, July 9-13.

Murre, J.M.J., R.H. Phaf, and G. Wolters (submitted) CALM: Categorizing And Learning Module.

Phaf, R.H., E.O. Postma, and G. Wolters (submitted) ELAN-1: a connectionist model for implicit and explicit memory tasks.

Phaf, R.H., E.O. Postma, and G. Wolters (1990) Connectionist simulations of the dissociation between implicit and explicit memory. *INNC-90*, Paris, July 9-13.

Zieleman, J.G. (1988) A graphical editor and compiler for the GRASPL programming system. Master's thesis, University of Texas, Austin, Texas.

Copyrighted material