ISSN 0803-6489

An Object-Oriented Approach to Neural Networks

Barry Kristian Ellingsen

REPORT NO. 45 JUNE 1995

Department of Information Science,
University of Bergen,
N-5020 Bergen, Norway
Email: barry.ellingsen@ifi.uib.no

Abstract

The aim of this report is to present an object-oriented approach to the design of a neural network simulation system. Although there are several object-oriented neural network systems available, only a few presents their design by using a consistent and uniform object-oriented methodology. Object-oriented programming alone is not sufficient to obtain the advantages of object-orientation, such as improving reuse, and emphasizing extensibility and flexibility. A neural network simulation software, called MANN (Modular Artificial Neural Network), is developed by using the OMT (Object Modeling Technique) methodology [6]. An important experience gained from the development process is that the concepts of reuse, extensibility, and flexibility must be included in an early stage of the development to be fully exploited.

Keywords: Neural networks, object-oriented design, C++

1 Introduction

For some years, considerable effort has been done in the development of software simulation systems for neural networks. Most software simulation systems available are specialized in terms of architecture and application domain. Only a few software simulator systems are designed to have configurable architectures for a multitude of neural network models, learning algorithms, and application domains. Typical application domains are: pattern recognition, speech recognition, vision, and forecasting.

By using an object-oriented approach to the design of software systems, advantages in terms of flexibility, extensibility, and portability can be achieved. This is also the case for an object-oriented neural network simulation system. System development ought to start with an analysis of the problem to conceive a precise and understandable description of the system. Herein are the requirements of the system, what the system should do, and its constraints. In other words, the result of an analysis phase is a model of the real-world problem at hand.

In this report I will address the design of an object-oriented neural network simulation system, called *Modular Artificial Neural Network* (MANN). The motivation for designing MANN is not to design a better and more efficient system than what already exists, but to present a solution using an object-oriented approach by taking fully advantage of the object-oriented concepts of abstraction, reusability, and extensibility. Section 2 presents the motivation and the requirements of the system. Next, in section 3, the system is designed using the Object Modeling Technique (OMT) [6], concentrated on object modeling and some aspects of dynamic modeling. An overall description of the solution and the MANN system is presented in section 4. Suggestions for future work is presented in section 5.

2 System requirements

In a previous work, I have developed a backpropagation neural network in plain C [1], by using the traditional and functional approach. The design of the neural network was based on incremental steps of adding and modifying bits and pieces of code. Clearly, any major revision of the system was too complex to accomplish without a significant amount of labour and redesigning involved. The backpropagation network had a reasonable amount of flexibility due to a descriptive and simple configuration mechanism. Unfortunately its internal structure did not have a corresponding flexibility with the intention of system maintenance and extensibility.

The backpropagation network, like most other neural network systems, is spe-

cialized concerning learning algorithm and application domain. A consequence of this is a lack of flexibility; for example, if a recurrent network architecture is required for an application domain, the system has to be redesigned to accomplish this. By and large, this is not a satisfactory situation. Further, if several neural network models were required, such as feedback networks, unsupervised networks etc., none or only a small amount of the existing code could be reused for this purpose. The main motivation for an object-oriented approach to a neural network simulation system is: to design reusable components that facilitate a flexible configuration of an application.

By using an object-oriented modeling approach, one is encouraged to model the world *as one sees it*, – to use a one-to-one approach to identify objects of the real-world. Whenever such real-world objects are possible to conceptualize, the object-oriented modeling process is unquestionably useful. However, in neural networks, the identification of objects is not a trivial process. The question is: What are the objects of a neural network? This really depends on one's conceptual view of a neural network. A neuro-physiologist tends to see neurons, synapses, dendrites, chemical and electrical transmissions, etc. A mathematician probably tends to see matrices, vectors and linear algebra. My conceptual view of a neural network is somewhere between these two conceptual views.

From a computational point of view, neural network objects consist of matrices, vectors and mathematical operations on these. This view concentrates on a functional approach to neural networks, which is not what the object-oriented paradigm encourages. On the contrary, object-oriented methodology focuses on modeling *concepts* in terms of collaborating objects, rather than *functions*. In a neural network system, concepts can be represented by the objects: model, neuron, weight, and pattern.

Current object-oriented systems, such as SESAME [3], emphasize on *flexibility* as one of their main objectives. Although SESAME has an object-oriented implementation, with a powerful and flexible usage, its design philosophy is nevertheless based on data flow graphs and block-diagrams and their functional relationships. SESAME's design approach is clearly functional, which is not according to the philosophy of object-oriented methodology.

By using a well-known and mature object-oriented methodology in the development of the MANN system, a unified view of the problem domain can be achieved in all development phases – from analysis to maintenance. The focus in this work is on analysis and object design of the MANN system. The requirements for the MANN system can be summarized by the three concepts:

- Abstraction
- Flexibility

- Extensibility

An important requirement for the MANN system is to provide different levels of abstraction. The neural network is conceptually separated in general and specific properties. The general properties are mainly layers of neurons and weights among them, and the specific properties are the multitude of learning algorithms. The learning algorithms are broadly grouped into supervised and unsupervised learning, feedback and feed-forward networks. A flexible solution is required to obtain a comprehensible and powerful user interface. This implies that a descriptive language is required for configuration of models; any modification of the network model should be done by changing the existing network definition, not the code itself. Thus, no programming skills are required to use the system. Not only is a powerful user interface needed but also a system interface to different pattern formats. These flexibility requirements does not differ substantially from other systems available, such as SNNS [10], Asprin/MIGRAINES [2], PlaNet [5], and Xerion [9]. Some of these systems also provide a graphical user interface; the MANN system does not – yet. A major achievement of the MANN design approach is extensibility. For example, if a new learning algorithm is added to the system, only learning specific behavior is needed; the rest, i.e., the general properties, is provided by the MANN system. Thus, is a high degree of *reuse* is obtained.

3 MANN design

In this report I have concentrated on OMT's object design in terms of a formal graphic notation for modeling classes and their relationships to one another. The classes in the object diagrams are provided with necessary attributes for describing the class' properties, and useful class operations are provided for modeling dynamic behavior. Section 3.1 describes the object model, and section 3.2 presents simple event trace diagrams for modeling of dynamic behavior. Functional modeling is beyond the scope of this report.

3.1 Object modeling

As described in the previous section, a requirement for the MANN system is to separate the general properties of neurons and weights, from the specific properties of different learning algorithms. Figure 1 illustrates the two levels of abstractions. The specific properties are different specializations of a neural network. Examples of specializations are backpropagation, bidirectional associative memory (BAM), and the Boltzmann machine [7, 8].

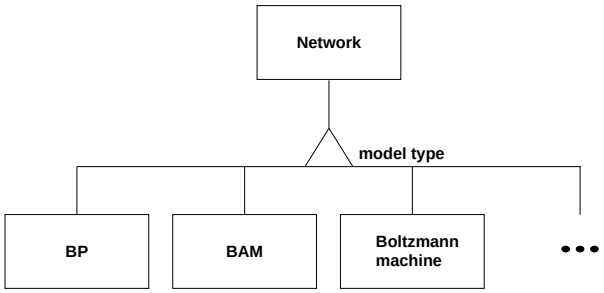

Figure 1: *Specializations of a neural network*

The general properties are partitioned in modules for logical grouping of classes, associations, and generalizations. There are basically four modules in this design: *layer*, *weight*, *pattern*, and *parameter*. Each of the modules are explained in this section.

The layer module

The neurons are organized into layers, in which the neurons in each layer are of a certain type. Figure 2 illustrates the layer module, containing the classes: *Layer*, *Neuron* and specializations of it.

There are three basic types of neurons included in the design: linear, gaussian, and sigmoid. The derived neuron type in figure 2, *SigmoidsymNeuron*, uses a symmetrical sigmoid activation function, whose output range is [-0.5,0.5] instead of [0,1] as for the standard sigmoid (logistic) activation function.

The weight module

The weight module shown in figure 3 represents the strength between neurons. As for the neurons, a weight can be of a certain type too. A weight type reflects what operation can be done on it during learning. Moreover, the weight type is associated with and dependent on the type of neurons it connects. Thus, a certain neuron type requires a similar weight type. For example, a linear neuron requires linear weights connected to it. A *Connection* class is provided to facilitate arbitrary connections among the neurons in the layers. A connection is regarded as a scheme containing pointers to sending and receiving neurons. This particular design solution is a key to achieve the flexibility of the MANN system.

The modules in figure 2 and figure 3 comprise a general and important property of a neural network. This is generally called the network topology. It should

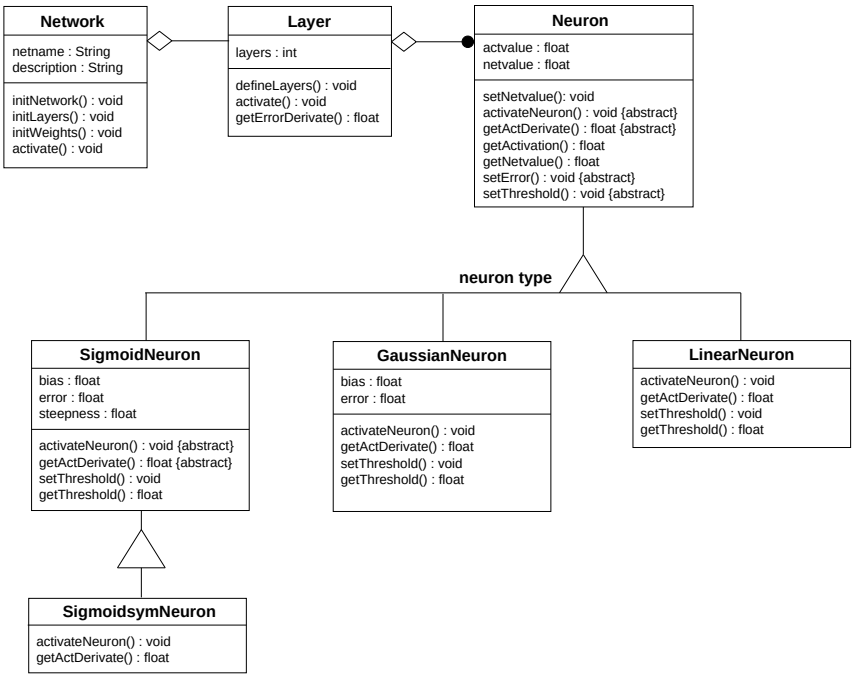

Figure 2: *The layer module*

be noted that the two modules are not perfectly general though, because they contain classes of specific neuron and weight types concerning particular learning algorithms. A consequence of this is somewhat less favorable: Extension of the MANN in terms of a new network model may imply installation of new neuron and weight types, i.e., adding more specializations of the class hierarchies shown in figure 2 and figure 3. This constraint compromises the conceptual division of general and specific properties of the MANN system, and is probably a weakness of this design. Nevertheless, this trade-off is feasible because introducing new neuron and weight specializations have no side effects on the existing classes.

The pattern module

A neural network simulator system needs an interface for pattern input and presentation of network output results. The MANN pattern input/output module is illustrated in figure 4.

There are three specializations of the *Pattern* class: one unsupervised, one supervised, and one for the pattern input. The first two classes are used for network output, and the last one is used for pattern input from files. There are basically two types of pattern formats that the MANN system can recognize: digits

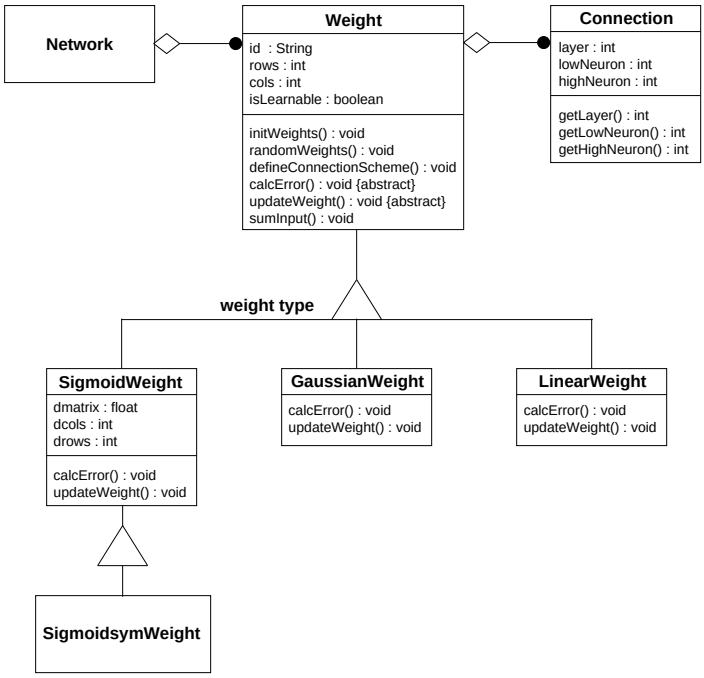

Figure 3: *The weight module*

and floating point values. The two types of pattern formats can be organized as vectors or matrices. A network output contains both statistical measures of the learning or test process, and patterns. Currently, there are three types of the supervised network output formats[1]:

- *SupClassOutput*; classification scheme in terms of a confusion matrix containing as many rows and columns as there are classes, plus an extra column reserved for a rejection category [4]. Any quantity in an off-diagonal position in the confusion matrix represent misclassifications.
- *SupDigitOutput*; vectors and matrices of digit values representing binary patterns.
- *SupFloatOutput*; vectors and matrices of floating point values representing analog patterns.

[1]There are no unsupervised models designed yet.

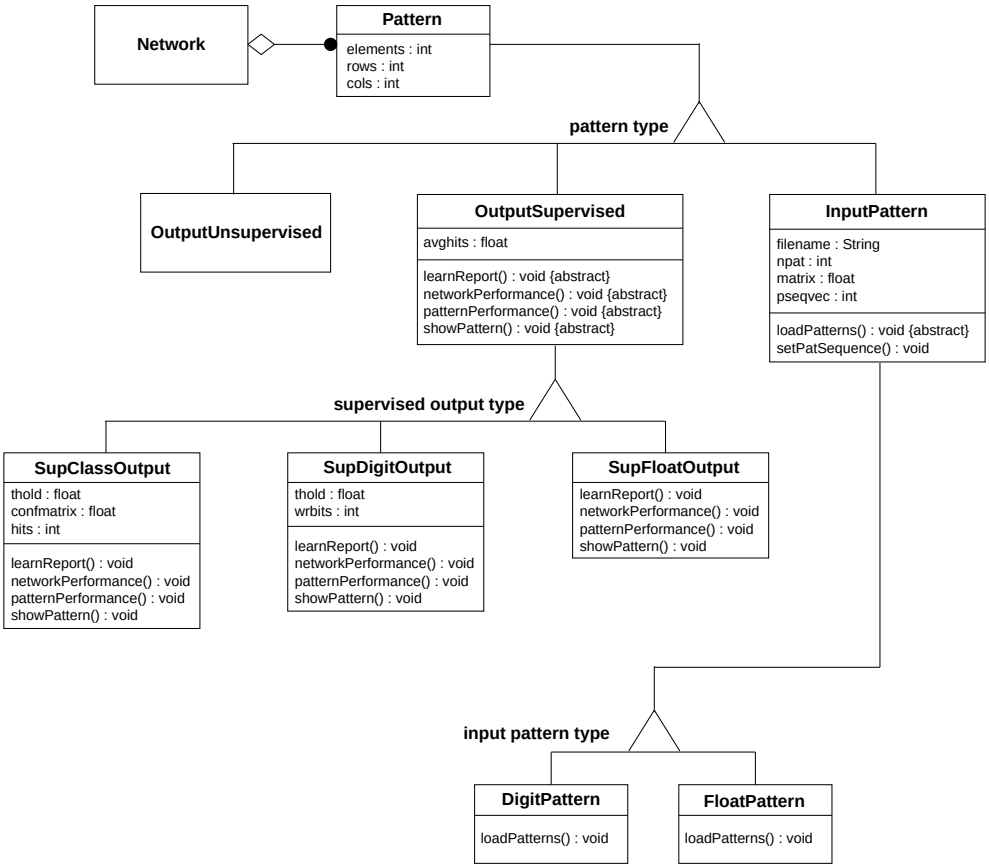

Figure 4: *The pattern module*

The parameter module

Finally, the parameter module provides the facility for managing the network parameters. This module is illustrated in figure 5. The parameters are commands that can be used to configure a network model and to control the MANN system. A network configuration is a process of defining a model in terms of neuron, layers, connections, pattern interfaces, and learning specifications for a particular application. The parameters are specified in a network definition file and read into the MANN system upon startup.

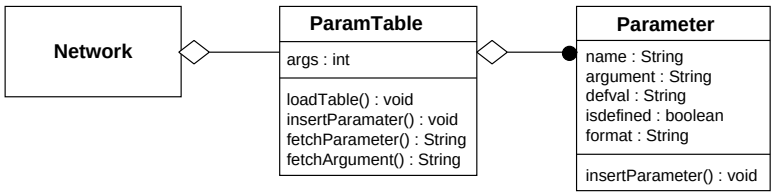

Figure 5: *The parameter module*

Some network parameters, those concerning configuration of neuron and connection among them, are managed by the layer and weight module. Other parameters are more specialized towards a network model, such as the backpropagation model, and are managed by the model specific class, e.g., the *Backprop* class shown in figure 1. Installing a new neural network model implies installation of parameters for that particular network model. Thus, the parameters are installed in a network model class, but managed and validated by the parameter module.

Functional dependencies

There exists a functional dependency between the layer module and the weight module because the weights uses the activation value in the neurons for updating the weights. A similar functional dependency exists between the layer module and the pattern module because network output is formatted by the pattern output classes (see figure 8 for module diagram). None of the client/supplier relationships are explicitly shown here[2].

[2]In OMT, the client/supplier relationship indicates a functional dependency among classes. This is part of the functional modeling and, hence, not included in this report.

3.2 Dynamic modeling

To illustrate the MANN system's dynamic behavior, two event trace diagrams are presented. The diagram in figure 6 shows a typical scenario for defining a network model, which in this case is the backpropagation model. The diagram in figure 7 shows a typical scenario for a learning epoch.

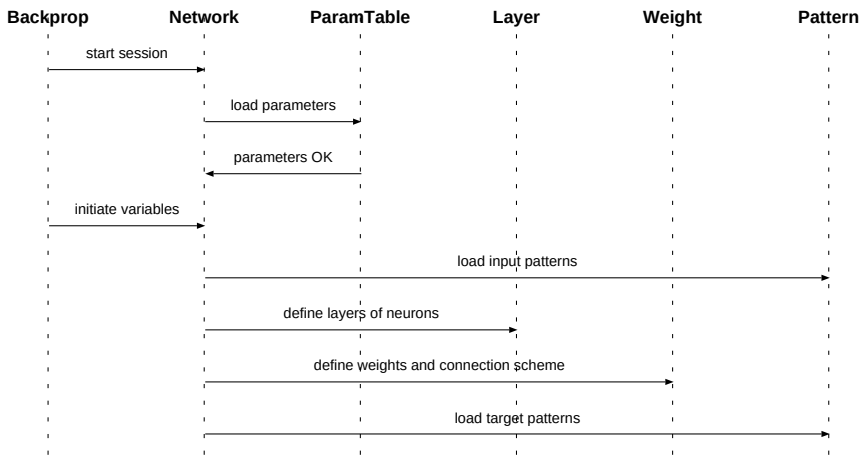

Figure 6: *Event trace for network definition*

The learning scenario in figure 7 illustrates a typical neural network learning session. However, the learning behavior in the figure indicates some irregular events. Most learning algorithms operates primarily on neurons [8]. This means that there is a primary indexing of neurons, and a secondary indexing of weights connected to the neurons. The event trace diagram in figure 7 however, indicates an opposite organization of indexing: Both the activation of neurons event and the calculation of their error derivate event, are based primarily on weight indexing, indicated by the events: *sum weight values* and *calculate error derivate*. By using primary indexing on weights, the flexibility of arbitrary connections, discussed in section 3.1, are obtained.

4 Solution

MANN modules

The MANN modules can be summarized as following:

- *layer*; organization of neurons into layers.

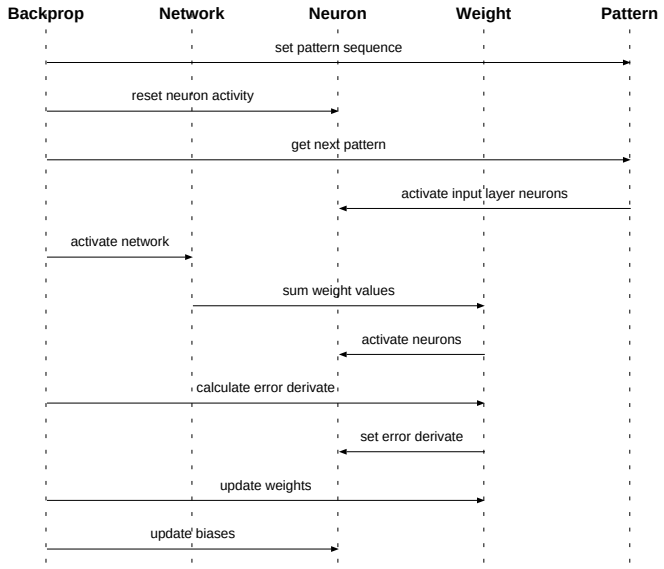

Figure 7: *Event trace for network learning*

- *weight*; definition and management of weight matrices and connection schemes.
- *pattern*; interface to the patterns, and specification of network output formats.
- *parameter*; parameter management, e.g., insertion and deletion of parameters for model configuration.

The modules can be illustrated as a block diagram shown in figure 8.

The backprop module, i.e., the *Backprop* class, uses the services from the general properties of a neural network, shown as a shaded area in the figure. The dashed modules at the back of the backprop module suggest that various network models can be added to the MANN system. Each of the new network model can then take advantage of the services from the general properties.

A neural network model is specified by including parameters in a *Network Definition File*. This file is read into the network by the parameter module upon system startup. The parameters are used for specifying the topology, the interface to the patterns, and to control the learning process. Input patterns and target patterns (in case of supervised learning) are read from separate files where the file names and their formats are specified in the network definition file. After the learning

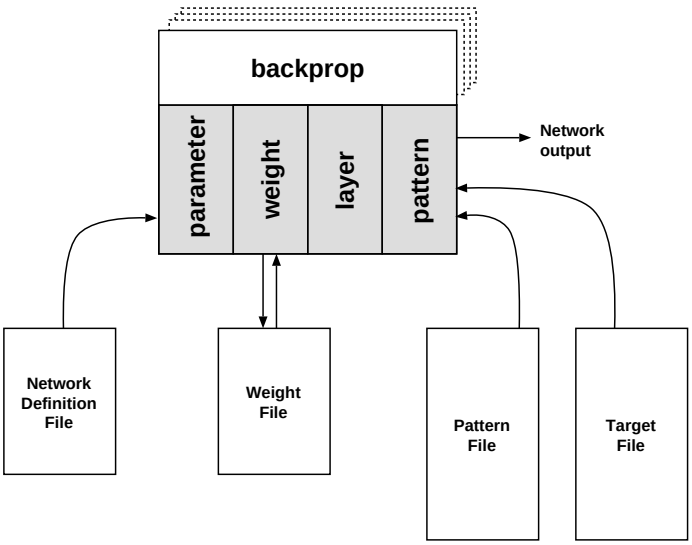

Figure 8: *MANN modules*

has terminated, the weights and the current network configuration are saved to a separate file (weight file). This file is read into the MANN system upon testing.

MANN usage

Below is an example of a network definition file for a pattern recognition application. A semicolon in the definition indicates comments. Only a few of the parameters are discussed in the example, most of which concerns topology and the pattern interface. A complete description of the full parameter syntax is beyond the scope of this report.

```
description("Letter recognition network")


;; network layer specification
layer(output, nodes=26, sigmoid_sym)
layer(hidden, nodes=40, sigmoid_sym)
layer(input, nodes=1024, linear)


;; connection scheme
connect(hidden, all, input, all, random=1.0, a)
connect(output, all, hidden, all, random=1.0, b)
weight_ranges(low=-100.0, high=100.0)   ; lower and upper range of weight values


;; pattern input interfaces
input("letters.pat", rows=32, columns=32, digits, randomized)
target("letters.tar", rows=1, columns=26, digits)
```

```
;; network output interface
output(classification, threshold=0.0)
learn_report(1)                          ; report learning statistics after each epoch


;; digit translation
input_translate("1", 0.5, "0", -0.5)
target_translate("1", 0.5, "0", -0.5)
output_translate(0.5, "1", -0.5, "0")


;; network save file
weight_file("letters.wgt")


;; BP algorithm parameters
equilibrium(stacksize=0)                 ; no equilibrium criteria is provided
max_epochs(1000)                         ; maximum number of iterations
outer_learning_rate(0.5)                 ; learning rate in weight set connected to the
                                         ; output layer
hidden_learning_rate(0.5)                ; learning rate in hidden weight sets
momentum(0.0)                            ; momentum term
output_error_function(exponential)       ; use an exponential function for error values
learn_criteria(hits=100)                 ; stop learning when number of hits is 100 percent
```

The network definition example shown above specifies a backpropagation model for recognition of 26 different letters represented as binary images of 32×32 pixels. An interface to the input patterns is specified in the *input* and *target* parameters with a file name and a format of rows, columns, and data type. The keyword `randomized` in the *input* statement indicates that the letters in the data set are being represented in a randomized order in each learning epoch.

A *layer* statement enables definition of neurons organized in different layers. In the example, a two-layer network containing 1024 input neurons, 40 hidden neurons, and 26 output neurons are defined. There is no limit to the number of neurons in a layer, nor is it a limit to the number of hidden layers; each hidden layer is labeled sequentially, e.g., `hidden0`, `hidden1`, etc. All neurons in a layer must have a common type. Here, all neurons in the hidden and output layers are of a `sigmoid_sym` type, while the neurons in the input layer are of a `linear` type. A `sigmoid_sym` has an activation value range of [-0.5,0.5]. Therefore, the pixel values need to be translated from 0 and 1 (as represented in the files) to -0.5 and 0.5 accordingly. This translation is specified by the parameters: *input_translate*, *target_translate*, and *output_translate*. A scaling term can be used optionally for certain neuron types. For example, a sigmoid neuron type can be scaled by the steepness of the activation curve. The statement below illustrates a definition of a hidden layer of sigmoid neurons with a less steeper curve of the activation function.

```
layer(hidden, nodes=20, sigmoid=0.5)
```

The definition of a connection scheme is obtained by the *connect* parameters. In the main example above, the connection scheme defines a fully connected network, in which all neurons at a lower layer are connected to all neurons at a higher layer. This yield two weight matrices, labeled `a` and `b` by the user. All weights are here initialized by random values in the range from -0.5 to +0.5, defined by `random=1.0` in the *connect* statement. Other connection schemes are possible. Feedback connections from the output layer back to the hidden layer can be obtained by defining:

```
connect(hidden, all, output, all, fixed=1.0, c)
```

The keyword `all` indicates that the connection includes all neurons of the specified layer. Ranges of neurons can also be specified, such as `0..19`. The keyword `fixed=1.0` indicates that the weight values are fixed at 1.0 and no updates are allowed. Another feature of the flexible connection facility of the MANN system is weight sharing. Weight sharing is when two or more groups of neurons at the same or different layers share a common set of weights. This can easily be obtained by using a similar label (`a`) of the connection, as illustrated in the connect statements below.

```
connect(hidden, all, input, 0..511, random=1.0, a)
connect(hidden, all, input, 512..1023, ignore, a)
```

The example of shared weights given above defines one weight matrix shared between two equal sized groups of sending and receiving neurons. A constraint here is that the dimension, i.e., the number of sending and receiving neurons of the groups, must be the same. When several groups of neurons share a common weight set, only one group of neurons ought to be given privileges for updating the weights. This option is specified by using the keyword `ignore` in all but one of the shared *connect* statements.

The *output* parameter specifies the format of the network results. In addition to digit and floating point format types, a *classification* format specifies that the output is presented as a confusion matrix of the classes produced by the network output and the target description [4]. The remaining parameters in the example controls the backpropagation learning process, and will not be discussed here.

Connection schemes

The type of connection schemes currently provided by the MANN system is called *inter-layer* connections, which are connections between neurons in different layers.

For example, in a fully connected backpropagation network, a neuron in one layer is connected to all the neurons in the next layer. This is called feed-forward connections. Another type of inter-layer connection is a recurrent connection, in which a connection loop from a neuron in a layer back to a neuron in a lower layer. This is called feedback connections.

Implementation

The MANN system is successfully implemented under UNIX using the GNU C++ compiler and class library (Libg++). In total, 22 classes comprises the MANN system, included the *Backprop* class. The total number of code lines is less than 3500. The GNU class library is used in the following cases:

- a double linked list template class for parameter management
- a string container class
- a regular expression container class for parameter parsing

Beyond use of the GNU C++ class library, the MANN code is compatible with C++ version 2.0 compiler and later.

Advantages of the MANN design

A major advantage of the MANN system is its development based on a well-known and comprehensive object-oriented design methodology. The concepts of *abstraction*, *flexibility*, and *extensibility* were established as requirements of the MANN system in an early stage of the development, and recognized through the entire life-cycle. As a result, a high degree of reuse is obtained in that general properties of a neural network can be reused by several different network models.

The modular design of the system makes the MANN system easy to maintain and extend. New modules can be added without significant interference with the existing modules. A natural extension of the MANN system is to add new network models. Currently, it is only the backpropagation model that is implemented. The flexibility of the *layer* and *weight* modules makes it easy to configure a wide range of connection schemes, including a recurrent structure.

An interface to different type of patterns is facilitated by the *pattern* module. This module also makes it possible to present the network output in different formats, herein a confusion matrix format for pattern classification applications. The *parameter* module provides a facility for simple network configuration.

5 Future work

Even though the MANN system facilitates a flexible and simple configuration of connection schemes through the layer and weight module, it is currently limited to *inter-layer* connection. However, certain network models require *intra-layer* connections, that is, connections between neurons in the same layer. For example, Hopfield network and competitive learning models are based on intra-layer connections that uses lateral connections [7, 8]. Here, a neuron is connected to itself and to its surrounding neighbors. Intra-layer connections are currently not provided by the MANN system. A possible extension of the MANN system would be to implement lateral connections.

To enhance the user friendliness of the MANN system, a graphical user interface (GUI) module is a natural extension. By this, a neural network model is configured and modified by using the GUI. From a design point of view, adding the GUI module does not cause significant interference with the existing core modules. This can be obtained by letting the GUI module communicate with the parameter module through the parameter set specified in the *Network Definition File*. This extension is illustrated in figure 9.

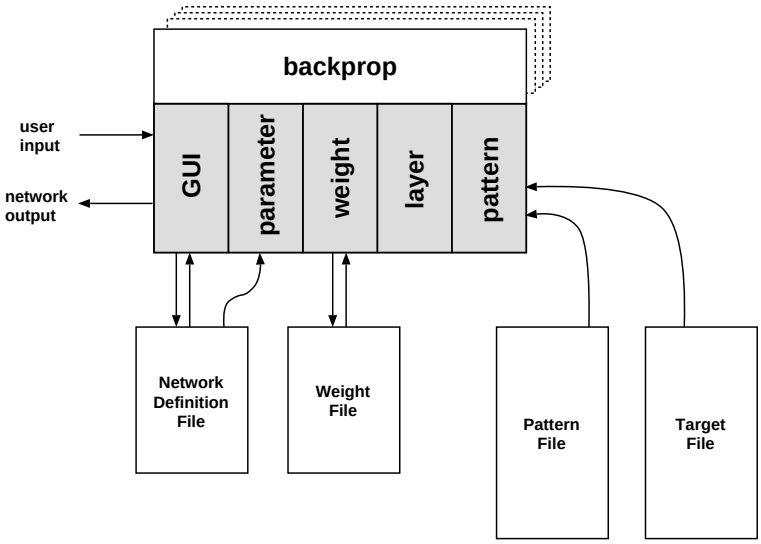

Figure 9: *Extension of the MANN system*

Instead of issuing parameters to the network definition file, the user interacts with the MANN system through the GUI. Upon network model execution, the parameters are read from a network definition file as previously, but they have been defined through the GUI. Network output can also be presented through the

GUI. Further, the GUI can also be used for inspection of the neuron and weight contents.

References

[1] B. K. Ellingsen. A comparative analysis of backpropagation and counterproagation networks. *Neural Network World*, 4(6):719–733, 1994.

[2] R. R. Leighton. *The Asprin/MIGRAINES Neural Network Software. V6.0*, October 1992.

[3] A. Linden, T. Sudbrack, C. Tietz, and F. Weber. An object-oriented framework for the simulation of neural nets. In S. Hanson, J. Cowan, and C. Giles, editors, *Advances in Neural Information Processing Systems 5: Proceedings of the 1992 Conference*, pages 797–804, San Mateo, CA, 1993. Morgan Keufmann Publishers.

[4] T. Masters. *Practical Neural Network Recipies in C++.* Academic Press, San Diego, CA, 1993.

[5] Y. Miyata. *A User's Guide to PlaNet Version 5.6.* Computer Science Department, University of Colorado, Boulder, January 1991.

[6] J. Rumbauch, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, Englewood Cliffs, NJ, 1991.

[7] D. E. Rumelhart and J. L. McClelland. *Parallel Distributed Processing: explorations in the microstructure of cognition*, volume I: Foundations of *Computational models of cognition and perception.* MIT Press, Mass., 1986.

[8] P. K. Simpson. *Artificial Neural Systems: foundations, paradigms, applications, and implementations.* Pergamon Press, Elmsford NY, first edition, 1990.

[9] D. van Camp. *The Xerion Neural Network Simulator–Version 3.1.* Department of Computer Science, Toronto, 1993.

[10] A. Zell, N. Mache, T. Sommer, and T. Korb. Recent developements of the snns neural network simulator. In *SPIE Conference on Application of Artificial Neural Networks.* Universität Stuttgart, April 1991.