A COMPONENT ARCHITECTURE FOR ARTIFICIAL NEURAL NETWORK SYSTEMS

Fábio Ghignatti Beckenkamp

**Dissertation zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften (Dr.rer.nat.)
an der Universität Konstanz**

**eingereicht in der
Mathematisch-Naturwissenschaftlichen Sektion
im Fachbereich Informatik und Informationswissenschaft**

Juni 2002

Acknowledgements

I dedicate this work to my parents Valecy and Jalda that have made the education of their kids their first priority in life.

I express my most profound gratitude to my wife Ana who supported my initiatives and goals from the beginning of our relationship, though it meant even sometimes being geographically separated but close enough for love.

Special thanks go to my advisor Prof. Dr. Wolfgang Pree for providing the opportunity of doing this work, and for his expertise, enthusiasm and friendship that helped in the course of doing the PhD.

Thanks to my brother Tarcísio and my sister Mariele for being nearby whenever I needed.

Thanks to Ana's family for having comprehended the importance of this effort.

This work would not have being possible without the direct support of many people and institutions for several reasons. My thankfulness goes to the University of Constance, its professors, colleagues and staff; and to the Federal University of Rio Grande do Sul, especially to Prof. Dr. Paulo Engel.

Some people filled my life during this time with unconditional friendship and help, this includes: Sergio Viademonte; Altino Pavan; Egbert Althammer; André Ghignatti and the Mercador colleagues; Michael Beckenkamp and family; César De Rose; Gustavo Hexsel; Ênio Frantz; Beatriz Leão; Miguel Feldens; Brazilian friends in Germany and European friends.

Deutsche Zusammenfassung

Diese Arbeit stellt zuerst die Architekturbausteine eines Komponentenframeworks dar, das im Rahmen der Dissertation implementiert wurde und das die Wiederverwendung der Kernteile von Modellen für künstlichen neuronalen Netze (artificial neural networks, ANN) erlaubt. Obwohl es eine Reihe von verschiedenen ANN-Modellen gibt, wurde ein wesentlicher Aspekt bisher kaum untersucht, nämlich der der Bereitstellung von wiederverwendbaren Komponenten, die eine effiziente Implementierung von entsprechende Systemarchitekturen für diese Domäne ermöglichen. Das Komponentenframework wird mit bestehenden Implementierungsansätzen für ANN-Modelle und -Simulationen verglichen.

Die Anwendung von ANN sieht sich mit Schwierigkeiten konfrontiert, wie zum Beispiel Begrenzungen von Hardwareressourcen und passende Softwarelösungen. Die Tatsache, wie sich die ANN-Komponenten die Parallelisierung von vernetzten Computern zunutze machen, stellt einen Beitrag zum Stand der Technik im mobilen Code und in verteilten Systemen dar. Die Software-Architektur wurde so definiert, dass sie die Parallelisierung sowohl der internen Ausführung eines ANNs wie auch der Simulation von unterschiedlichen ANNs, simultan auf derselben Maschine oder auf unterschiedlichen Maschinen verteilt, erleichtert. Das kombinatorische Netzmodell (combinatorial network model, CNM) wurde dabei als Fallstudie für die Implementierung von Parallelität auf der Ebene der ANN-Struktur gewählt.

Die durchgeführte Verbesserung eines der ANN-Modelle, nämlich des CNM, stellt einen Beitrag zum Bereich der ANNs selbst und zum Data-Mining dar. Der ursprüngliche CNM-Algorithmus konnte erheblich verbessert werden hinsichtlich der Optimierung des Suchraumes, mit dem Effekt einer höheren Ausführungsgeschwindigkeit und weniger Speicherverbrauch.

Das letzte Kapitel bietet einen Überblick über offene Forschungsfragen, die während der Dissertation aufgetaucht sind.

Schlüsselwörter: Framework, Komponenten, Wiederverwendung von Software, objektorientiertes Design, objektorientierte Architektur, künstliche neuronale Netze, intelligente Expertensysteme, hybride intelligente Systeme.

Abstract

The main focus of the PhD thesis is about automating the implementation of artificial neural networks (ANNS) models by applying object/ and component technology. Though various ANN models exist, the aspect of how to provide reusable components in that domain for efficiently implementing adequate system architectures has been barely investigated. The prototypical component framework that was designed and implemented in the realm of the dissertation is compared to existing approaches for generically implementing ANN models and simulations.

The application of ANNs faces difficulties such as limits of hardware resources and appropriate software solutions. How the ANN components harness parallelization on networked computers represents a contribution to the state-of-the-art in mobile code and distributed systems. The software architecture was defined in a way to facilitate the automated parallelization at the level of the inner execution of an ANN and at the level of the simulation of different ANNs at the same time, on one computing node or on different computing nodes in a distributed way. The Combinatorial Network Model (CNM) was chosen as case study for implementing parallelism at the level of the ANN structure.

The improvement of one of the ANN models, namely the CNM, represents a contribution to the area of ANNs itself and to data mining. The original CNM algorithm could be significantly enhanced regarding the aspect how it deals with the search space, which results in a faster execution and less memory allocation.

A sketch of research issues that result from the PhD work rounds out the thesis.

Keywords: artificial neural networks, object-oriented frameworks, components, software reusability, object-oriented design, software architectures, intelligent decision support systems, hybrid intelligent systems.

Summary

List of Figures

List of Tables

List of Source Code Examples

Acronyms

ABLE	Agent Building & Learning Environment
ANN	Artificial Neural Network
ANSI	American National Standards Institute
API	Application Program Interface
ART	Adaptive Resonance Theory
ASCII	American Standard Code for Information Interchange
BAM	Bi-directional Associative Memory Simulation
BP	Backpropagation Neural Network
CANN	Components for Artificial Neural Networks
CLOS	Common Lisp Object System
CNM	Combinatorial Neural Model
CPU	Central Processing Unit
ECANSE	Environment for Computer Aided Neural Software Engineering
GoF	"The gang of four", Gamma et al. 1995.
GUI	Graphical User Interface
HTML	Hypertext Markup Language
ID	Identifier
IO	Input and output
IP	Internet Protocol
JDK	Java Development Kit
JVM	Java Virtual Machine
LAN	Local Area Network
NN	Neural Network
OO	Object-oriented
ORB	Object Request Broker
PC	Personal Computer
PDP	Programmable Data Processor
RAD	Rapid Application Development
RAM	Random Access Memory
SOM	Self-Organizing Feature Maps
SQL	Structured Query Language
SWE	Software Engineering

TCP	Transmission Control Protocol
UFRGS	Federal University of Rio Grande do Sul, Brazil
UML	Unified Modeling Language
URL	Uniform Resource Locator
VMI	Vendor Managed Inventory
XML	Extensible Markup Language
XOR	Exclusive OR

1 Introduction

1.1 Motivation

The direction of the PhD thesis originated in the prototypes build in the realm of the Master thesis at the State University of Rio Grande do Sul (UFRGS – Brazil – http://www.inf.ufrgs.br). Some studied neural network models and their first prototype implementations drew the attention of companies. Those wanted to have an expert system be able to analyze historical data and to build knowledge about these data, in order to perform decision-making. The Hycones system (Leão and Reategui, 1993) was developed for this purpose. It was mainly applied to the medical area to support decision on heart diseases. Later, another application also appeared requiring the application of Hycones in areas such as credit scoring (Reategui and Campbell, 1994; and Quelle AG – http://www.quelle.de) and logistics (newspaper distribution control system at RBS – http://www.clickrbs.com.br).

1.2 Problem statement

The implementation of the solutions prototype exposed the fragility of the Hycones system as a software system. Applying it to different application areas showed the following difficulties:

- The artificial neural network's inner code had to be changed to adapt to each application.
- The input and output artificial neural networks data handling had to be coded nearly from scratch.
- A change from one artificial neural network model to another represented a huge coding effort.
- The limits of hardware and software resources.
- The different parts of the system were implemented on different hardware and software platforms.

- The artificial neural networks algorithms had limitations such as not dealing with combinatorial explosion.

The goal of the PhD thesis is to apply object-oriented component-based software engineering construction principles to overcome these problems. One of the results is a flexible architecture that facilitates the implementation of any artificial neural network model and that can be applied to different domain problems.

1.3 Overview of the proposed solution

This work starts with the identification of the software limitations of the typical ANN systems such as Hycones. The identified problems were solved by studying them in detail and by designing and implementing completely new software solutions to each of them.

Initially, an object-oriented design of an artificial neural networks software solution was built. A few so-called frameworks[1] were identified and built in order to permit the construction of any artificial neural network model based on them. Those frameworks formed the basis for building four different artificial neural network models as software components.

Other important frameworks were identified and implemented to perform tasks that complement the artificial neural network's functionality: A framework was created to build a different domain knowledge model that facilitates the fast adaptation of an artificial neural network model to any application problem at hand; another framework was built for fetching data for learning and testing the artificial neural networks models; and finally a framework for configuring, via user interface, the artificial neural network models was implemented.

Based on the whole set of frameworks implemented to build and support the artificial neural network models, an artificial neural networks simulation framework was defined and a complete simulation tool (CANN Simulation Tool) was built. On top of this simulation tool, many domain problems can be modeled in parallel and different artificial neural network models can run at the same time in order to solve the problems at hand. Four artificial neural network components (CANN) were built based on the ANN frameworks. They run in the simulation tool.

[1] A piece of software that is extensible through the callback style of programming

Two empirical studies were accomplished in the area of software parallelism. First, the study of the parallel software implementation of artificial neural networks, together with the implementation of a general-purpose solution for running artificial neural networks in parallel. Second, the study and implementation of a solution to run the simulation of the ANN instances in a distributed way using an ORB. The two implementations where done in order to give hints on how to improve the artificial neural networks performance by better using the software and hardware infrastructure.

Finally, the CNM model algorithm received special attention on its design and implementation using the given frameworks. Improvements on its algorithm and a parallel implementation solution were proposed and implemented in the realm of this PhD work.

1.4 Organization of the thesis

The thesis is organized in 8 chapters. Chapter 2 is dedicated to introduce the computer science areas involved in this thesis: Software Engineering (SWE) and Artificial Neural Networks (ANN). There, ANNs are motivated by biological perspective and the four ANN models implemented in the thesis are presented. Complementarily, the SWE concepts that are extensively applied along the thesis are also introduced.

The following five chapters form the core parts of this thesis. Chapter 3 shows the implementation of the ANN frameworks. It shows in detail the design decisions of each framework and its relevant implementation aspects. It finally compares it to other work in the area.

In sequence, the chapters 4 and 5 go deep on the parallel and distribution issues. Chapter 4 introduces ANN parallel implementation and shows what was the possible solution to have parallelism for the CANN simulation tool. Furthermore, this chapter shows in detail the proposed and implemented parallel solution to the CNM (Machado and Rocha, 1989) model.

Chapter 5 approaches the implementation of a distribution framework for the ANN components. The distribution solution is implemented in the CANN simulation tool in order to have the possibility of running different ANN models at the same time in different machines and centrally controlled.

Chapter 6 explains contributions of this thesis to the ANN field where the CNM artificial neural model has its learning algorithm optimized in order to be faster and allocate less memory during this process.

Chapter 7 shows in detail the characteristics and functionalities of the CANN simulation tool. It also compares this tool to other ones that are commercially available. Chapter 8 has the conclusions of this work and the future research possibilities for its continuation.

1.5 Statement of goals and contributions

The main goal of the thesis is to *come up with a flexible and efficient design for ANN implementations*. For this purpose, object-oriented framework technology has been applied to the ANN domain for the first time. A framework for ANN development is constructed and various ANN models are implemented using this framework in order to evaluate its applicability. It is an important goal of this thesis to give contributions on *how to better develop ANN software* in order to make the ANN functionality optimized as a software system. It is also part of the goals to come up with contributions on *how to implement ANN parallelism in software and code mobility for ANN architectures in order to provide ANN execution in a distributed system*.

To promote contributions to ANN models is another important goal and is concentrated on the CNM model, where the author has extensive experience regarding its development and applications.

To better measure the importance of these goals, the software development practices done so far are evaluated. It means the identification of main development problems. To solve these problems techniques from the SWE area are chosen. This work shows these techniques are appropriate to ANN software development.

The contributions of this work are concentrated on the areas of software engineering and artificial neural networks. In the software engineering area the main contribution is to show the applicability of object-oriented framework technology to the construction of ANN software. This work focused on:

- Analyzing the ANN domain area.

- Deriving from the design analysis the design for the construction of the ANN framework.
- Constructing different types of ANN architectures in order to prove the framework's applicability.
- Analyzing and implementing a solution for a parallel implementation of ANNs.
- Analyzing and implementing ANN code mobility and ANN distributed execution in a LAN.
- Implementing a simulation tool for ANN based on the defined frameworks.
- Deploying and applying the constructed ANN simulation tool to different application areas.
- Evolving the CNM artificial neural network by proposing and implementing a learning and testing algorithm that is faster and uses a smaller memory footprint.
- Proposing and implementing a parallel solution for the CNM algorithm.

Each of those contributions were carefully designed, implemented, tested and compared to related work. Some SWE techniques were extensively used and supported by this work. The hot-spot-driven design (Pree, 1995) was applied to the design of the flexible parts of the frameworks showing its applicability to the ANN area. Design-patterns (Gamma et al. 1995) and meta-patterns (Pree, 1995) proved to be an important vehicle of proper design communication among the involved developers. The design of the ANN frameworks can be shared with the whole community of ANN developers. The coining of the concept of Framelets (Pree and Koskimies, 1999 and 2000) was also supported by the design and implementation of the ANN basic frameworks that can be considered as Framelets. The accumulated experience in building the ANN framework components is an important contribution of this work and is shared with the research community through this text and the collection of publications produced along the development of this work.

The new ANN components have been used in much different application areas as: weather forecast (Viademonte et. al, 2001a and 2001b); personalization of Internet sites where the ANN components are used to build knowledge about the user preferences, navigation and transaction habits to build a personalized environment for a better user experience (http://www.godigital.com); and e-business such as the creation of an agent to analyze marketplace negotiation data and the optimization of the supply chain performance by implementing a VMI (Vendor Managed Inventory) algorithm based on ANN (http://www.mercador.com).

As mentioned before, this work resulted in several publications:

- (Pree, Beckenkamp and Da Rosa, 1997) introduces the software engineering challenges of the redesign of the Hycones system.
- (Da Rosa, Beckenkamp and Hoppen, 1997) approaches the use of fuzzy logic to model semantic variables in expert systems.
- (Beckenkamp, Pree and Feldens, 1998) introduces optimizations to the CNM algorithm.
- (Beckenkamp and Pree, 1999) describes the artificial neural networks frameworks components design.
- (Beckenkamp and Pree, 2000) exposes details of the artificial neural networks frameworks implementation.
- (Da Rosa, Burnstein and Beckenkamp, 2000) presents results of the application of the Voyager ORB on the distribution of ANN components.
- (Viademonte, Burnstein, Dahni, and Willians, 2001a and Viademonte and Burnstein, 2001b) presents the first results of a weather forecast expert system where the CANN simulation tool is applied.

2 Context and State-of-the-art

A first step to understand the origin of ANNs is to relate them to the biological paradigm. It is important to know a about the biological neuron and nerves to understand why the ANN models make certain approximations to the biology. It is important to understand when the approximations are poor and when they are reasonable.

2.1 Biological Motivation

The brain's elementary building blocks are the neurons. The study of the neuron, its interconnections and its role in the information processing is one important research field in modern biology. The research on ANN starts by trying to simulate the function of a neuron. ANN researchers adopt a minimal cellular structure that can be seen in Figure 2.1

The dendrites are the transmission channels for the incoming information. The synapses are the contact regions with other cells and are responsible for supplying the dendrites with information. Some organs inside the cell body are responsible for keeping the cell continuously working. The mitochondria are responsible for supplying the cell with energy. The cell has one axon that is responsible for transmitting the output signal to other neurons.

The information processing in the cell membrane is done via electrical signals produced by diffusion. In short, neurons transmit information using action potentials. The information processing involves a complex electrical combination and chemical process. The synapses control the direction of the information transmission. They can be inhibitory or excitatory depending on the kind of ion flowing through it.

The cell processes information by integrating incoming signals. If the flow of ions (membrane potential) reaches a certain threshold, an action potential is generated at the axon of the cell. The information is not only transmitted but also weighed by the cell. Rojas (Rojas, 1996) explains that "signals combined in an excitatory or inhibitory way can be used to implement any desired logic function". This explains the huge information processing capability of the neuron systems.

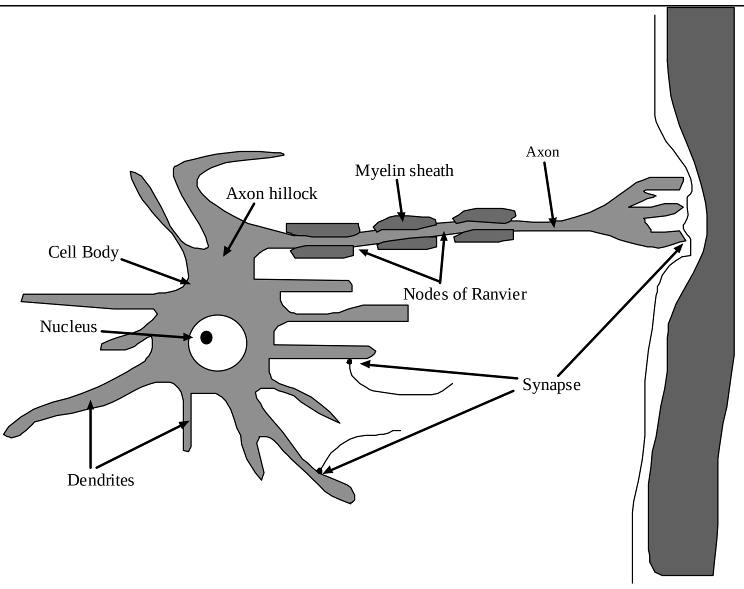


Figure 2.1 – General structure of a generic neuron (Freeman 1992)

Neuron information is stored at synapses. The synapses control the passage of ions, thus controlling the cell depolarization. The plasticity is the synaptic connection will determine the capacity of the cell in acting properly. Therefore, the synapses control is important information to the whole system's functionality. In ANN this synaptic control efficiency is simulated via a constant that is multiplied to the flowing information on the input channels, the weight.

The storage, processing and transmission of information at the neuron level are still not fully understood. Neurons form such complex nervous systems that researches in many areas such as mathematics, chemistry, medicine and psychology are trying to understand how cell nerves act. Computer science has played an important role on this research being an important test bed for the different concepts and ideas. Furthermore, ANN also can be seen as a computation paradigm that has much to be explored by scientists of computer science. For a deeper study on the biological foundations see Anderson, 1995 or Rojas, 1996.

2.1.1 The generic artificial neuron

The term artificial neuron is used in the literature interchangeably with: node, unit, processing element or even computational unit. Depending on the author's approach or goals, one of those will be used. Here the term neuron will be kept in order to maintain the analogy to the biological structures when modeling the framework objects. But whenever necessary, neurons will be distinguished by using the terms natural or artificial.

The artificial neuron is a pretty huge simplification of the natural neuron. It is important not to be too restrictive and not to try to make a one-to-one relationship between the natural and the artificial neurons. In this work there are no discussions about the simplifications done. There are no discussions whether the models are appropriate simplifications of the reality or not. This work is based on what is already accepted in the community, and tries to improve those concepts from a software-engineering point of view.

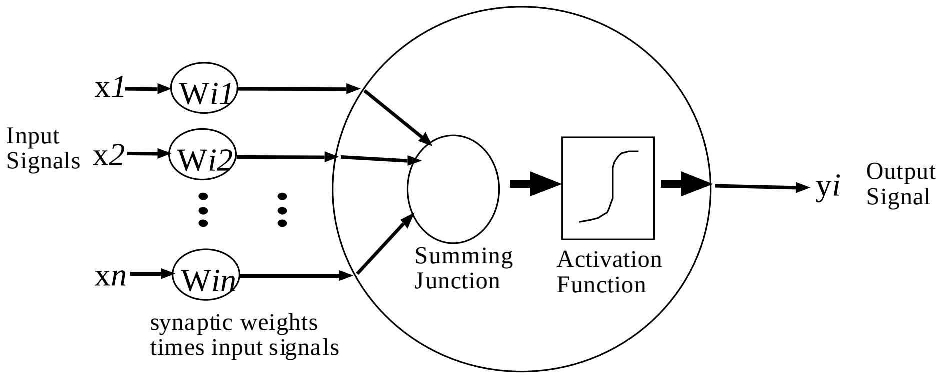


Figure 2.2 – The artificial neuron

The artificial neuron has input and output channels and a cell body. The synapses are the contact points between the cell body and the input or output connections, having a weight associated to them. The artificial neuron can be divided into two parts: the first is a simple integrator of synaptic inputs weighed by connection strengths; the second is a function that operates on the output of the integrator. The result of the second function will be the neuron output. The artificial neuron is schematically drawn in Figure 2.2.

There are several mechanisms for calculating the neuron output value, such as: linear combination, mean-variance connections and min-max connections (Simpson, 1992). The most common way of doing it is the linear combination where the dot product (inner product) of the input values with the connection weights is calculated. In general it is followed by a nonlinear operation, the activation function (also called neuron function).

Next a detailed description of the linear combination is shown:

1. The ANN has several inputs (x_j) and one output (y_i).

2. Each input connection has an associated weight that controls the connection strength (w_{ij}) and is usually a real number.

3. The weights can be inhibitory or excitatory (typically negative and positive values).

4. The net input (Equation 2.1) is calculated by summing the input values multiplied by the corresponding weights (inner product or dot product).

$$net_i = \sum_j x_j w_{ij}$$

Equation 2.1 - Calculating the net of the neuron

5. The output value (Equation 2.2) is calculated applying an activation function that uses the net_i:

$$y_i = f_i(net_i)$$

Equation 2.2 - Calculating the output of the neuron

Some possible activation functions are shown in the following section.

2.1.1.1 Activation function

There are many possible activation functions. The most common ones are: the Linear, the Step, the Ramp, the Sigmoid, and the Gaussian functions. The last four functions introduce nonlinearity in the network dynamics by bounding the output values within a fixed range.

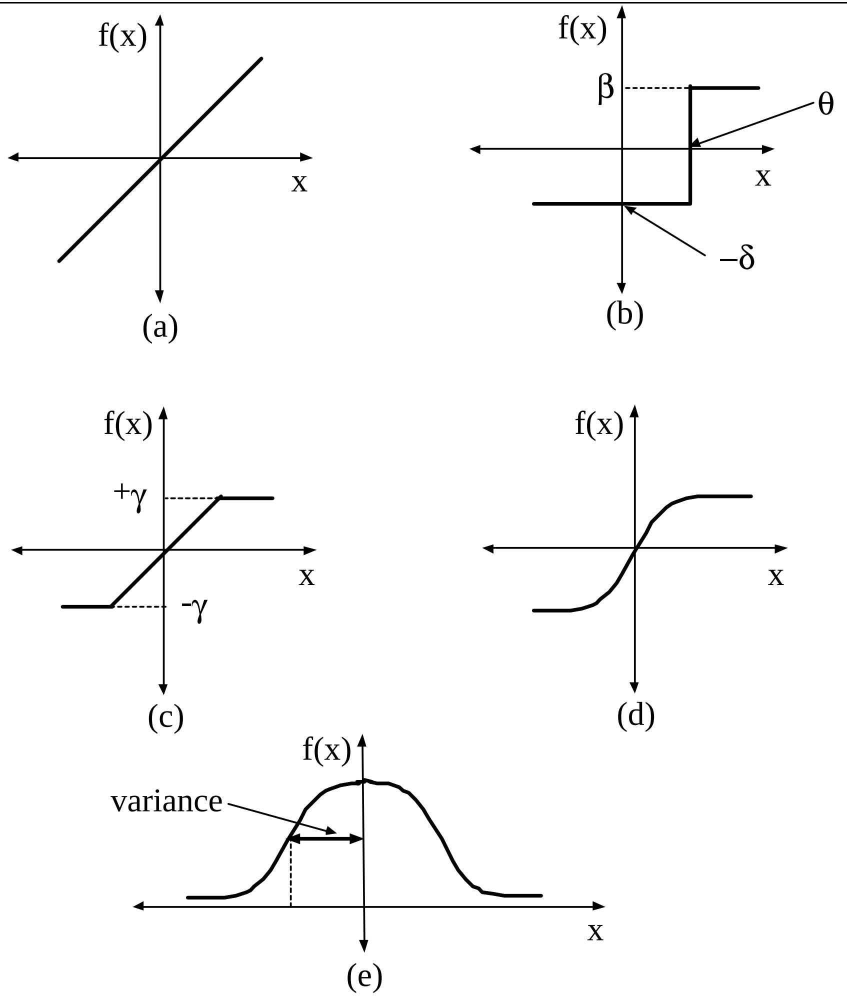


Figure 2.3 – Activation functions (Simpson, 1992)

Linear Function

The Linear function produces a linearly modulated output. It is described by Equation 2.3 and can be seen in Figure 2.3(a):

$$f(x) = \alpha x$$

Equation 2.3 – Linear function

Where α is a positive scalar.

Step Function

The Step function produces only two values, β and -δ. If x is equal or exceeds a predefined value θ the function produces β, otherwise it produces -δ. The values β and δ are positive scalars. This function is binary and is used in neural models such as the Hopfield (Hopfield, 1982) and the BAM (Kosko, 1988). The Step function is defined on Equation 2.4 and its result can be seen in Figure 2.3(b).

$$f(x) = \begin{cases} \beta & if \quad x \geq \theta \\ -\delta & if \quad x < \theta \end{cases}$$

Equation 2.4 – Step function

Ramp Function

The Ramp function is a combination of the linear and the step functions. It has upper and lower bounds and allows a linear response between them. It is defined on Equation 2.5 and can be seen in Figure 2.3(c). The value γ is the function saturation.

$$f(x) = \begin{cases} \gamma & if \quad x \geq \gamma \\ x & if \quad |x| < \gamma \\ -\gamma & if \quad x \leq -\gamma \end{cases}$$

Equation 2.5 – Ramp function

Sigmoid Function

The Sigmoid function is a continuous version of the Ramp function and provides a graded, nonlinear response within a specified range. The most common sigmoid function is the Logistic distribution function that provides an output value from 0 to 1.The value α > 0 and usually equal to 1. The Sigmoid function definition is shown at Equation 2.6 and its effect can be seen in Figure 2.3(d).

$$f(x) = \frac{1}{1 + e^{-\alpha x}}$$

Equation 2.6 – Sigmoid function

Gaussian Function

The Gaussian function is symmetric in its origin. It requires a variance value υ>0 to shape the function. The Gaussian function definition is shown at Equation 2.7 and its effect can be seen in Figure 2.3(e).

$$f(x) = \exp(\frac{-x^2}{\upsilon})$$

Equation 2.7 – Gaussian function

2.1.2 ANN Architectures

The natural neurons form the neural nerves when connected. In ANN the artificial neurons are connected in many different ways forming architectural characteristics. The learning algorithms and the architectures are closely related. It is important to have a clear concept of how the artificial neurons may be interconnected to form the specific architectures, because this will define how the computer implementations of the architectures can be done. The possible computer implementation solutions for the specific architectures and learning algorithms are going to be explained later. Following the description of the principal ANN architectures is as follows.

2.1.2.1 Single-Layer Feedforward Networks

In this simplest network, a layer of input neurons is connected to a layer of output neurons. The "single-layer" designation refers to the output layer. The layer of input neurons is not considered because it does not process any computation over the input values. It just bypasses the input values. An example of this kind of neural network is a linear associative memory where an input pattern is associated to an output pattern, both in form of a vector. Figure 2.4 shows a single-layer network of 3 output nodes.

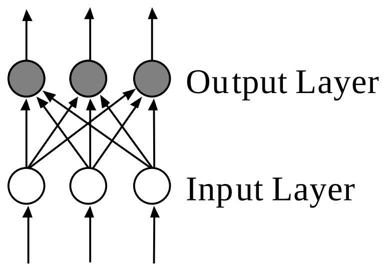


Figure 2.4 – Single-layer feedforward network

2.1.2.2 Multi-Layer Feedforward Networks

In this architecture, one or more hidden layers of neurons are present. Those networks are able to deal with higher-order problems because of the extra set of connections and the extra dimension of neural iterations (Churchland and Sejnowski, 1992; Haykin, 1994).

Figure 2.5 shows a multi-layer 4-2-3 network that means a network formed by an input layer with 4 neurons, only one hidden layer with 2 neurons and an output layer with 3 neurons. The network is fully connected because each neuron in one layer is connected to all neurons in the next layer. It also may happen that not all the neurons of one layer are connected to all neurons of the subsequent layer. This may happen when the user has a certain previous knowledge about the pattern being classified. He/she is then able to preset the network connections. Figure 2.6 shows a multilayer 4-2-3 network not fully connected.

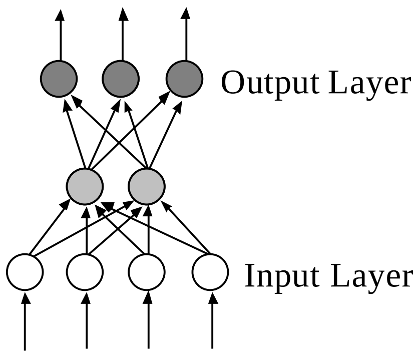


Figure 2.5 – Multilayer feedforward network fully connected

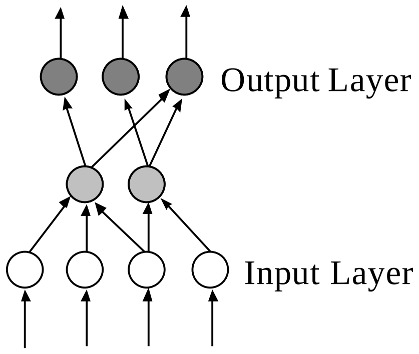


Figure 2.6 – Multilayer feedforward network not fully connected

2.1.2.3 Recurrent Networks

This network model has at least one feedback loop. It may have the same architecture as a layered network, but it is necessary to have the feedback. The feedback can happen from the output of one neuron back to the input of another neuron. This feedback may happen among neurons of the same layer or neurons of different ones. The feedback may also happen as a self-feedback when the output of the neuron is returned to its own input. The feedback deeply influences the network learning capability and its performance. Figure 2.7

shows a single-layer recurrent network where the output signals of the neurons are fed to the input of the other neurons in the same layer.

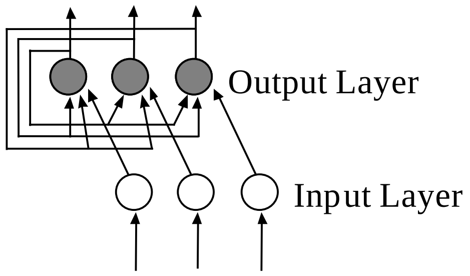


Figure 2.7 – Recurrent network with no self-feedback loops

2.1.2.4 Lattice Networks

Lattice Networks consist of networks formed by arrays of neurons at any dimension. An input neuron is associated to each array supplying the signal to the array. Figure 2.8 shows a two-dimensional lattice network of 3-by-3 neurons fed from a layer of 3 input neurons.

For each of the above architectures, various learning algorithms were proposed. In this work, the last 3 architectures were considered when choosing the test models once the single-layer is a simplification of the multi-layer. Two multi-layer models were chosen, one case of fully connected network and one case of not fully connected network. The fully connected one is the Backpropagation (Rumelhard and McClelland, 1986) and the other one is the CNM (Machado and Rocha, 1990). The chosen recurrent network is the ART1 model (Grossberg, 1976; Grossberg, 1987) and the chosen lattice model is the SOM (Kohonen, 1982). The description of these 4 models is presented in Section 2.2 below.

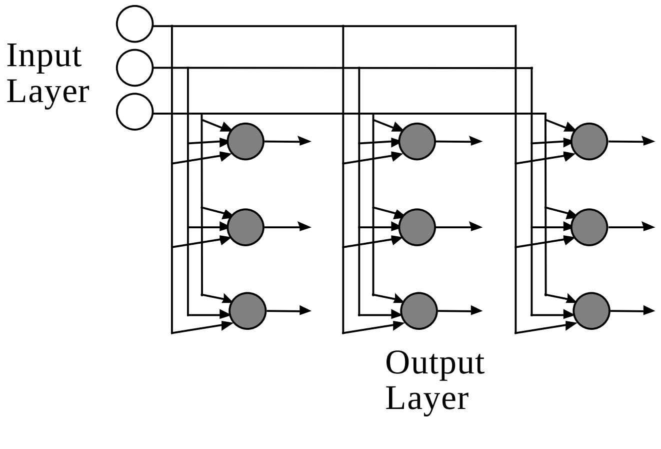


Figure 2.8 – Lattice network

2.2 ANN Learning Algorithms

Since the information of how the neurons process data is on the weights, the weight values will determine whether the neuron is able to accomplish a certain task or not. So learning is a process of finding weights that represent the system knowledge. In practice, the learning process is the modification of the weight values in order to make the output values the proper ones, given a certain input data. The learning law is determined by the modification of the strength of synaptic junctions based on a system of differential equations for the weight values. The system of equations is solved to an acceptable approximation solution.

The values of the connection weights will determine how well the neural network solves the problems. These values may be predefined and hardwired into the network. However, such method is rarely adopted because it is very difficult to know in advance the appropriate values for the weights. A learning algorithm often determines the values of the weights. The learning algorithm is an automatic adaptive method that tries to fit the appropriate weights to the system solution. There is no explicit programming for the solution achieved.

Several algorithms are available for changing the values of the connection weights. It is not the goal here to cover all the possible algorithms but to introduce the general ideas behind them. The learning algorithms are divided into two categories: supervised and unsupervised learning.

Supervised learning is a process that incorporates global information and/or a teacher. The teacher regulates the learning, informs the network what it has to learn and checks if it has properly learned or not. Supervised learning has information deciding when to turn off the learning, deciding how long and how often to present each datum for learning, and supplying performance information. Some well-known algorithms that implement supervised learning are error correction learning, reinforcement learning and stochastic learning.

Supervised learning can be subdivided into two subcategories: structural and temporal learning. The first tries to find the best input/output pattern relationship for each pattern pair. It is used to solve problems such as pattern classification and pattern matching. The second one is concerned with finding a sequence of patterns necessary to achieve some final outcome. The current response of the network is dependent on the previous inputs and responses. Examples of use are prediction and control.

The unsupervised learning uses only local information during the learning. It organizes presented data discovering their collective properties. Examples of unsupervised algorithms are Hebbian learning and competitive learning.

In this work, there is no necessity to build up a complete taxonomy of the possible learning algorithms. There are several algorithms with innumerable variations. More complete lists of learning algorithms can be found in the neural networks literature such as Haykin, 1994; Simpson, 1992; and Rojas 1996.

A brief explanation of some relevant supervised and unsupervised learning algorithms follows.

2.2.1 Hebbian Learning

It is the simplest way of adjusting the connection weight values. It is based on the work of the neuropsychologist Donald O. Hebb (1949) (Haykin, 1994). Hebb hypothesized that the change in a synapse's efficacy is prompted by a neuron's ability to produce an output signal. It proposes a correlation-based adjustment of the connection weight values. That means if the activation of a neuron A repeatedly and persistently caused a neuron B to fire, then the efficacy of the connection among those neurons is improved. This idea was expanded to the inverse sense where either uncorrelated or negatively correlated activity produces synaptic weakening.

The Hebbian synapses efficiency thus is a mechanism that is time dependent because it relies on the exact time of occurrence of the presynaptic and postsynaptic activities. It is also a highly local mechanism where the local available information is used to produce a local synaptic modification. Finally, the Hebbian synapse has an interactive nature because it depends on activities on both sides of the synapse, that is, it depends on the interaction among the presynaptic and postsynaptic activities.

The mathematical models of Hebbian learning can be found in (Haykin, 1994; and Simpson, 1992). Important neural network models that implement this kind of learning include:

- The Linear Associative Memory (Anderson, 1970; Kohonen, 1972; Haykin 1994) that employs this learning and analyzes its capabilities;

- The Hopfield network (Hopfield, 1982) which is a one-layer network that restricts the neuron activity to either binary {0,1} or bipolar {-1,1} values and introduces feedback connections forming a nonlinear, dynamic system;

- The Bidirectional Associative Memory (BAM) (Kosko, 1988), which implements this learning in a two-layer network.

2.2.2 Competitive Learning

Competitive learning is a method of automatically creating classes for a set of input patterns. This kind of learning was introduced and extensively studied by Grossberg (Grossberg 1970, and 1982). The basic idea behind this learning method is that the output neurons compete among themselves for being the one to fire. Typically the input data is organized as vectors and the neural network maps the input vectors into its synaptic weights. The classes are represented by the groups of neurons that have the nearest synaptic vectors to a given input vector pattern. Important implementations of this kind of learning are:

- The Adaptive Resonance Theory (ART) (Grossberg, 1987) where the neurons of the competitive layers shall compete to find appropriate pattern classifications without compromising the neural network capacities of generalization (stability) and discrimination (plasticity);

- The Self-Organizing Feature Maps (SOM) also known as the Kohonen model (Kohonen, 1984) where neurons in the competitive layer compete to map the input features trying to simulate the cerebral cortex areas of storing the knowledge.

2.2.3 Error Correction Learning

When applying a data value to the input layer of neurons of a neural network, this data value is processed by the network neurons sequentially until a signal reaches the output neurons of the network. The output values produced by the output neurons should be the desired output values. A learning algorithm can be used to gradually approximate the computed output to the desired output. The error correction learning adjusts the connection weights considering the proportional difference between the desired and the computed values.

A two-layer network implementing error correction learning is able to solve linear separable problems. Examples of two-layer neural networks that implement error correction learning are:

- The Perceptron (Rosenblatt, 1962);
- The ADALINE (Widrow and Hoff, 1960).

A multilayer network using error correction learning is able to capture nonlinear mappings between the input and output patterns. For a long time the problem was to establish an appropriate algorithm to do the error correction for the hidden neurons connection weights based on the output neurons output values. The point was to determine how dependent the network output is from the output generated by the hidden neurons. It is ultimately a problem of minimization of a cost-function based on the error signal produced by the network output. The solution is the use of partial differentiation to calculate weight changes for any weight in the network (the gradient descent method; Widrow and Stearns, 1985; Haykin 1994). An example of neural network using the multilayer error correction algorithm is:

- The Backpropagation (Werbos, 1974; Parker, 1985; leCun, 1985; Rumelhart, Hinton and Williams, 1986), which introduced an error correction algorithm for multilayer networks.

2.2.4 Reinforcement Learning

It is similar to error correction learning because the weights are also reinforced for correct performance, and punished for incorrect performance. The difference is that the error correction uses a more specific algorithm where the output of each neuron on the output layer is considered, while the reinforcement learning uses nonspecific error information to determine the performance of the network. Furthermore, in reinforcement learning the gradient descent is performed in probability space while in error correction learning is performed in error space. The reinforcement learning is ideal for using in prediction and control where there is no specific error information available, only overall performance information. Neural network examples that use reinforcement learning are:

- The Adaptive Heuristic Critic neural network (Barto et al.,1983) which introduced the use of a critic to convert the primary reinforcement signal received from the environment into a higher-quality reinforcement signal called the heuristic reinforcement signal. The learning system consists of three components very much used in Artificial Intelligence: learning element, knowledge base and performance element. The learning element is responsible for doing all changes in the knowledge base, and the performance element is responsible for selecting actions randomly on the basis of a distribution that is determined by the knowledge base;

- The Associative Reward-Penalty neural network (Barto, 1985).

2.2.5 Stochastic Learning

This learning method adjusts connection weights to a multilayer network using random process, probability and energy relationship. The neural network escapes local energy minima in favor of a deeper energy minimum via probabilistic acceptance of energy states. The learning process is governed by a "temperature" parameter that slowly decreases the number of probabilistically accepted higher energy states. An example of this learning is:

- The Boltzman machine (Ackley et al. 1985).

2.3 ANN Input and Output Data

The neural network has to be set with data to do the learning and testing processes. All models need an input data pattern to be applied in its input neurons in order to learn or test this data pattern. The data to be applied to the network must be prepared in a way that the network is able to understand and process it. The quality of what is learned by the neural network depends very much on what the patterns are representing. Typically, each data pattern must be transformed in a vector of values that represent the pattern to be applied. The appropriate transformation of the data in the input vector is essential to the learning process. Many different features can be modeled from the same problem at hand. For example, the age of a person can be modeled as discrete values, sets, or fuzzy sets. Depending on how it was modeled, the network can succeed or not in the learning process.

Furthermore, there are neural networks that need to have input and output patterns presented during the learning process, the networks based on supervised learning such as the error correction learning algorithm. The input pattern is applied on the input neurons, then the signal flows over the network producing an output. The produced output is then compared with the desired output for the given input pattern. The comparison results in a difference that is used to do the error correction of the network connection weights.

Some neural networks accept only binary data as input such as the discrete Hopfield model (Hopfield, 1982), other networks accept real numbers like the Backpropagation (Werbos, 1974; Parker, 1985; leCun, 1985; Rumelhart, Hinton and Williams, 1986). The number of neurons on the network input layer usually is determined by the size of the vector to be applied, so that each vector element value is applied to one input neuron. The number of neurons in the input and the output layers are not necessarily the same.

The data preparation (pre-processing) for applying to the neural network simulation is a problem that has to be faced by the developer. A programmer often has to code the data transformation to the chosen ANN model, take care of the chosen ANN architecture (the number of neurons on the input and output layers) and so on.

The hard coding of the data preparation can generate overhead in many situations:

- For each new problem that the ANN has to be applied, a specific program has to be created to do the appropriate data pre-processing.
- Frequently the ANN architecture is changed during the process of finding the appropriate solution because certain features are added, removed or applied in a different way.
- Sometimes the solution via a neural model is not possible and the programmer chooses to go for another model. This may imply changing completely the way the data has to be pre-processed.

To avoid reprogramming of the data pre-processing in each of the above situations, it is important to use appropriate pre-processing tools that can easily integrate with the ANN implementation. One solution to have a nice integration is to create a pre-processing framework that deals with the data independently of the ANN model and the selected features. The construction of such a framework is proposed in this work via the Domain framework that is explained in Chapter 3.

2.4 Chosen ANN Models

In the realm of this work, it is not possible to approach the various existing ANN models. Few models were picked up among the most important ones, based on some criteria such as: application interest (potential), architecture and kind of learning. The idea is to be able to cover a good variety of situations, being able to implement enough generalizations that could be useful for most of the ANN models without having to implement many models.

The first chosen model was the Backpropagation (Werbos, 1974; Parker, 1985; leCun, 1985; Rumelhart, Hinton and Williams, 1986). This model implements a supervised learning based on the error correction learning algorithm. Its architecture is feedforward multilayer and it is fully connected. The Backpropagation is a widely used model for structural and temporal classifications. There are many variations of this model.

The second chosen model is the Combinatorial Neural Model (CNM) (Machado and Rocha 1990). This model also implements supervised learning based on a variation of the error backpropagation learning algorithm. The network is feedforward and not fully connected. Besides being an important and interesting neural model by its concepts, the reasons to implement it in this work were twofold: the profound knowledge of the author on this model; and the special interest in the application of this model on the credit scoring problem of companies.

Third, a typical unsupervised competitive learning model was chosen, the Self-Organizing Feature Map (SOM) (Kohonen, 1984). The SOM architecture is based on a two-dimensional lattice map. The SOM model has been largely applied in different areas such as image and speech recognition.

Finally, another unsupervised competitive learning model was chosen, the Adaptive Resonance Theory (ART) (Grossberg, 1987). The extra important architectural aspect implemented by this model is the presence of feedback connections among its neurons, so the network has a recurrent architecture.

Next, each of the four models is introduced. The goal here is not only to introduce the theoretical aspects of each model but also to find out the main characteristics of each one and, specially, the commonalties among the models. Those characteristics and commonalties are important to properly create an object model to build the neural network components.

2.4.1 The Backpropagation

The Backpropagation neural network (Werbos, 1974; Parker, 1985; leCun, 1985; Rumelhart, Hinton and Williams, 1986) is a multilayer, feedforward network, using a supervised learning mode based on error corrections. Minsky and Papert (1969) have provided a very careful analysis of the capabilities of the neural models proposed so far. At that moment, the Perceptron was offered a very simple guaranteed learning rule (the delta rule – Widrow and Hoff, 1960) for all problems that can be solved by a two-layer feedforward network. What Minsky and Papert (1969) have shown was that this learning rule was capable of solving only linear separable problems. Later, Rumelhart, Hinton and Williams (1986) have shown that the addition of hidden neurons to the neural network architecture would permit the mapping of the problem representation in a way that non-linear separable problems could be solved.

The problem, at that moment, was that there was no specified learning rule to cope with the hidden neurons. The solutions that came up later to solve this problem were the following three:

- The addition of weight values on the hidden neurons by hand, assuming some reasonable performance;
- The competitive learning where unsupervised learning rules are applied in order to automatically develop the hidden neurons.
- The creation of a learning procedure capable of learning an internal representation (using the hidden neurons) that is adequate for performing the task at hand.

The Generalized Delta Rule represents the latest approach, which implements a supervised learning algorithm based on the error correction in a multilayer neural network and which is known as the Backpropagation neural network. The proposed learning procedure involves the presentation of a set of input and output patterns to the neural network. The input patterns typically correspond to a sample of the real patterns. For each input pattern one output pattern is determined. The output patterns are the known classification of the correspondent input patterns. The patterns are represented in the form of a vector. When learning, the system first uses the input vector to produce its own output vector and then compares this with the desired output (the output pattern). If there is no difference, no learning takes place, but if there is a difference, the weights of the network are changed in order to reduce this difference.

The generalized delta rule minimizes the squares of the differences between the actual and the desired output values summed over the output neurons and all pairs of input/output vectors. There are many ways of deriving the delta rule; derivations are detailed in (Rumelhart at al.1986).

Figure 2.9 shows a generic Backpropagation neural network architecture with one layer of hidden neurons. The network is fully connected, having all neurons of one layer connected to all neurons of the next layer. It has an input layer which number of neurons corresponds to the size of the input vector. The number of neurons for the output layer corresponds to the size of the output vector. The Backpropagation may have more than one layer of hidden neurons. The number of neurons on the hidden layer may define the capability of the network in mapping the problem properly. Usually the determination of the number of neurons on the hidden layers and the number of hidden layers is very difficult. Typically, the network developer or user determines them empirically.

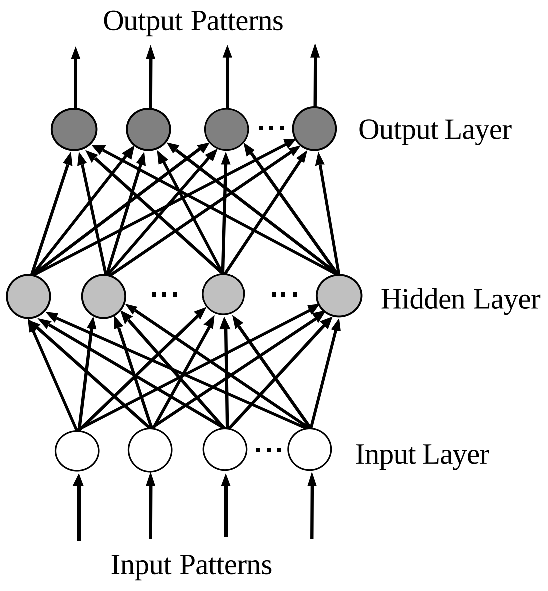


Figure 2.9 – Generic Backpropagation network

Next, a sequence of equations will be introduced showing the mathematical description of the generalized delta rule or the Backpropagation algorithm.

First the input vector $\boldsymbol{x}_p = (x_{p1}, x_{p2}, \ldots, x_{pN})^t$ is applied to the input layer of the network. The input neurons just bypass the input values to the hidden neurons via the connection synapses. The hidden neurons will calculate its net using Equation 2.8:

$$net_{pj} = \sum_{i=1}^{N} w_{ji} x_{pi} + \theta_j$$

Equation 2.8 – Hidden neurons net

In Equation 2.8, *p* is the pattern being learned, *j* denotes the *j*th hidden unit, w_{ji} is the weight on the connection from the *i*th input unit to the *j*th hidden unit, and θ_j is the bias value. The bias is useful to accelerate the network convergence. Equation 2.9 gives the activation of the hidden neuron.

$$i_{pj} = f_j\,(net_{pj})$$

Equation 2.9 – Hidden neurons activation

The calculation of the output neurons net (net_{pk}) and the corresponding output value (O_{pk}) is the same as shown in Equations 2.10 and 2.11. In these equations, the difference to Equations 2.8 and 2.9 is that the index *k* denotes the *k*th output unit.

$$net_{pk} = \sum_{j=1}^{N} w_{kj} i_{pj} + \theta_k$$

Equation 2.10 – Hidden neurons net

$$o_{pk} = f_k\,(net_{pk})$$

Equation 2.11 – Hidden neurons activation

Function *f* can assume several forms as seen previously on the activation functions exemplified in Figure 2.3 (Activation functions). Typically, two forms are of interest: linear and sigmoid output functions given by Equations 2.12 and 2.13 respectively. The same function forms are valid for both hidden and output neurons.

$$f_k\,(net_{pk}) = net_{pk}$$

Equation 2.12 – Linear output function

$$f_k\,(net_{pk}) = (1 + e^{-net_{pk}})^{-1}$$

Equation 2.13 – Sigmoid output function

After propagating the input signals over the network as explained by the equations above, it is time to calculate the local gradients for the output and hidden layers. This calculation is done based on the difference between the real and the desired output, that is the principle of the supervised learning as already explained. The calculation of the local gradients for the hidden neurons is done before the update of the connection weights to the output layer neurons. Equation 2.14 shows the calculation of the local gradients for the output neurons.

$$\delta_{pk} = (y_{pk} - o_{pk})\, f_k^{'}(net_{pk})$$

Equation 2.14 – Output neurons local gradients function

Equation 2.15 shows the calculation of the local gradients for the hidden neurons.

$$\delta_{pj} = f_j^{'}(net_{pj}) \sum_k \delta_{pk} w_{kj}$$

Equation 2.15 – Hidden neurons local gradients function

Equation 2.16 shows the f' function that is the derivative of the sigmoid activation function in respect to its total input, net_{pk}. The function shown is the one used for the output neurons, the same is valid for the hidden neurons changing the index k by index j.

$$f_k^{'}(o_{pk}) = o_{pk}(1 - o_{pk})$$

Equation 2.16 – Sigmoid function derivative

Having calculated the local gradients for all neurons on the output and hidden layers, it is time to go backwards recalculating the weights of the neural network based on those local gradients. It is necessary to calculate the negative of the gradient of E_p (error for the example pattern p), ∇E_p, with respect to the weights w_{kj}, to determine the direction in which to change the weights. Then the weights are adjusted so that the total error is reduced. Equation 2.17 calculates the error term E_p that is useful to determine how well the network is learning. When the error is acceptably small for each of the training examples, the training can be stopped. Figure 2.10 exemplifies a hypothetical error surface in weight space, its local and global minima and the gradient descendent.

$$E_p = \frac{1}{2}\sum_{k=1}^{M}\delta_{pk}^2$$

Equation 2.17 – Error term

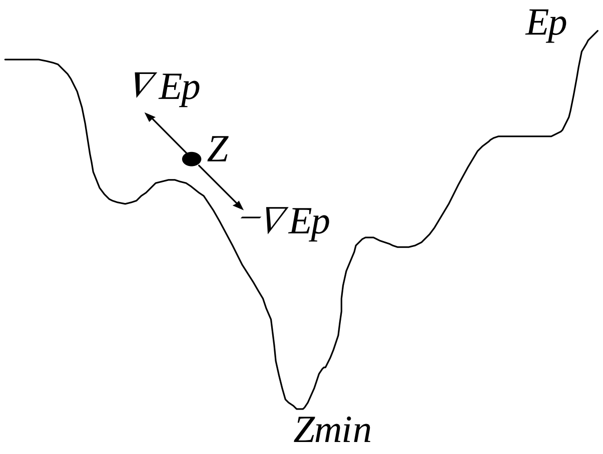


Figure 2.10 – Hypothetical error surface

The constant that assures the proportionality of the error adjustments steps is the learning rate η. The larger this constant, the larger the changes in the weights. The chosen learning rate shall be as large as possible without leading to oscillation. One way to avoid oscillation is to include to the generalized delta rule a momentum term that has a parameter α. Thereby Equation 2.18 gives the weight change for the connections among the output and hidden neurons in the iteration time t.

$$\Delta_p w_{kj}(t) = \eta(\delta_{pk} i_{pj}) + \alpha\Delta_p w_{kj}(t-1)$$

Equation 2.18 – Weight change magnitude for connections among output and hidden neurons

Equation 2.19 gives the update of the weight value for the connections among the output neurons and the hidden neurons.

$$w_{kj}(t+1) = w_{kj}(t) + \eta(\delta_{pk} i_{pj}) + \alpha\Delta_p w_{kj}(t-1)$$

Equation 2.19 – Weight update for connections among output and hidden neurons

Similarly, Equation 2.20 gives the weight change for the connections among the hidden and the input neurons in the iteration time t.

$$\Delta_p w_{ji}(t) = \eta(\delta_{pj} x_i) + \alpha \Delta_p w_{ji}(t-1)$$

Equation 2.20 – Weight change magnitude for connections among hidden and input neurons

Then Equation 2.21 gives the update of the weight value for the connections among the hidden neurons and the input neurons.

$$w_{ji}(t+1) = w_{ji}(t) + \eta(\delta_{pj} x_i) + \alpha \Delta_p w_{ji}(t-1)$$

Equation 2.21 – Weight update for connections among hidden and input neurons

Once the network learning reaches a minimum, either local or global, the learning stops. If it reaches a local minimum, the error produced at the network outputs can still be unacceptably high. The solution then is to restart the network learning from scratch with new weights for the connections. If this is still not the solution, the number of neurons on the hidden layer can be improved, or the learning parameters *learning rate* and *momentum* can be changed.

Important aspects to consider on the object model for the Backpropagation are:

- The input nodes just bypass the information. Usually this information is a normalized set of values between 0 and 1, resulting from a pre-processing of the input data.
- The hidden and the output neurons implement the Perceptron functionality where the *dot product* is applied and a *nonlinear function* gives the neuron activation.
- In the original model the connections among the neurons are *feedforward*. There are *no feedback* connections and the network is *fully connected.*
- The number of input, hidden and output neurons is determined based on the expert knowledge on the application domain.
- Even not having feedback connections the backward propagation of the error terms is a computation in the reverse direction. In fact, the backward

computation is as important as the forward computation on this model, being an important and interesting characteristic to be modeled.

The given Backpropagation equations presented here are from Freeman, 1992.

2.4.2 The Combinatorial Neural Model

The Combinatorial Neural Model (Machado and Rocha 1989, 1992; Machado et al. 1998) has been explored and developed during the past decade. Experiments with this model have demonstrated that it is well suited for classification problems, with excellent results in terms of accuracy (Leão and Rocha 1990; Feldens and Castilho 1997). The CNM integrates, in a straightforward architecture, symbolic and non-symbolic knowledge. This model has characteristics that are desirable in a classification system:

- Simplicity of neural learning - due to the neural network's generalization capacity.
- Explanation capacity – the model can map a neural network's knowledge into a symbolic representation.
- High-speed training - only one pass over the training examples is required.
- Immunity against some common neural network's pitfalls – i.e. local optima, plateau, etc.
- Incremental learning possibility - previously learned knowledge can be improved with new cases.
- Flexible uncertainty handling – it can accept fuzzy inputs, probabilistic inputs, etc, as inputs fall into the interval [0, 1].

The CNM includes mapping of previous knowledge to the neural network, training algorithms, and pruning criteria, in order to extract only significant pieces of knowledge. The detailed explanations on the possible learning algorithms are in (Machado et al. 1998). Here are considered the *Starter Reward and Punishment* (SRP) and the *Incremental Reward and Punishment* (IRP) learning algorithms, which are the original learning algorithms proposed for the model. Those learning algorithms offer the possibility of building a CNM network based on knowledge elicited from an expert (using SRP), and the possibility of refinement with new examples the knowledge existing in the network (using IRP).

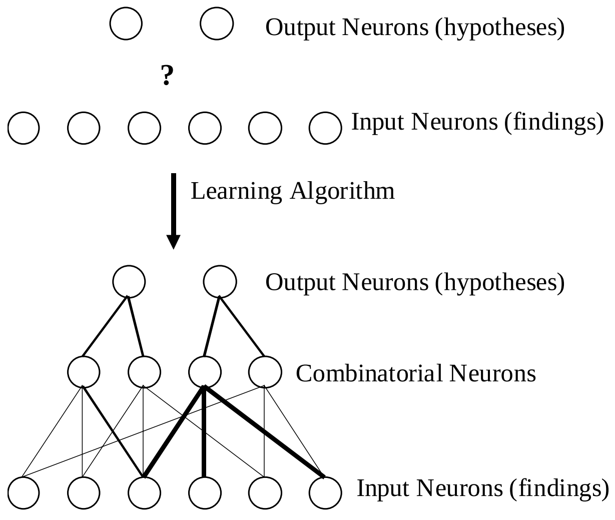


Figure 2.11 – The CNM network generation

The CNM is a 1-hidden layer, feed-forward network. It has particular characteristics in the way the topology is constructed, in neurons, in the connections among neurons, and in its training algorithm.

The domain knowledge is mapped to the network through evidences and hypotheses (Figure 2.11). The evidence may have many distinct values that must be evaluated separately by the neural network, called findings. The input layer represents the defined set of findings (also called *literals*). A finding can be a categorical or numeric pattern feature. The categorical feature can only take on a finite number of values while a numeric feature may take on any value from a certain real domain. A categorical feature requires at most one input neuron for each of the feature possible values. The state of each such neuron is either zero or one. A numeric feature has to be partitioned in fuzzy sets where each set will correspond to an input neuron and its state will correspond to the fuzzy set degree of membership. An example: if an evidence age is modeled, it probably has findings that can be modeled as fuzzy intervals (Kosko, 1992). The domain expert defines fuzzy sets for different ages (e.g. child, adolescent, adult, senior) (Figure 2.12). Each fuzzy set defined will correspond to a finding and, as a consequence, to a CNM input neuron.

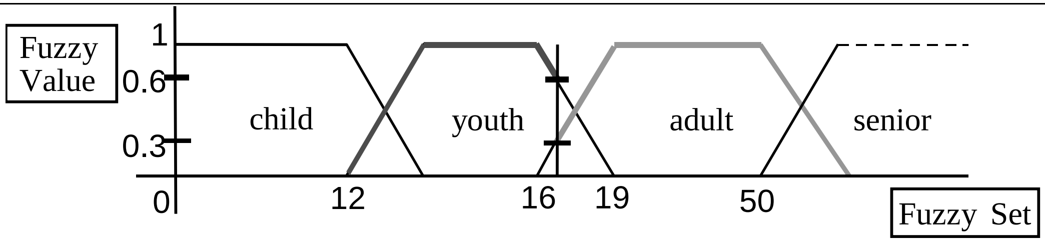


Figure 2.12 - Fuzzy Sets

In this case each input value is in the [0,1] interval (fuzzy set membership functions), indicating the pertinence of the training example to a certain concept, or the degree of confidence. In the example of Figure 2.12, the age 19 will correspond to a zero membership value for the child and senior fuzzy sets and to a 0.6 value for the adolescent fuzzy set and 0.3 to the adult fuzzy set.

The intermediate (combinatorial) layer is automatically generated. A neuron is added to this layer for each possible combination of evidences, from order 1 to a maximum order, given by the user.

The output layer corresponds to the possible classes (hypotheses) to which an example could belong. Combinatorial neurons behave as conjunctions of findings that lead to a certain class. For that reason, the *p*th combinatorial neuron propagates input values according to a fuzzy AND operator (Equation 2.22), taking as its output the minimum value received by the inputs.

$$\min_{i \in I_p} s_{ip}(x_i)$$

Equation 2.22 – Fuzzy AND

In Equation 2.22 above, $I_p \subseteq \{1,...,n\}$ indicates the appropriate input neurons, and either $s_{ip}(x_i) = x_i$ or $s_{ip}(x_i) = 1 - x_i$ (fuzzy negation). In the first case the synapse (i,p) is called excitatory and in the later case inhibitory.

Output neurons group the classification hypotheses implementing a fuzzy OR operator (Equation 2.23), propagating the maximum value received by its inputs.

$$\max_{1 \le p \le m} w_p \min_{i \in I_p} s_{ip}(x_i)$$

Equation 2.23 – Fuzzy OR

In Equation 2.23 above, *m* indicates the number of output neurons, and $w_p \in [0,1]$ is the weight associated with the connection from the *p*th combinatorial neuron to the output neuron.

The weights modifications are determined by using a supervised algorithm which attempts to minimize the mean square error (Equation 2.24) incurred by the network when presented with a set of examples.

$$mse(w) = \frac{1}{|E|} \sum_{e \in E} [y(e,w) - \hat{y}(e)]^2$$

Equation 2.24 – Mean Square Error calculation for the new weight set of values

In Equation 2.24 above, *w* is the weight vector of a CNM network, and the learning is done using a set of examples $E \subset [0,1]^n$. For an example $e \in E$, let $\hat{y}(e)$ denote the desired output, and *y(e,w)* the output generated by the network with the weight vector *w*.

The Incremental Reward and Punishment (IRP) and the Starter Reward and Punishment (SRP) learning algorithms are based on the concept of rewards and punishments. The connections between neurons (synapses) have weights and also pairs of accumulators for punishment (P_p) and reward (R_p). Before the training process, in absence of previous knowledge, all weights are set to one and all accumulators to zero. During the training, as each example is presented and propagated, all links that led to the proper classification have their reward accumulators incremented. Similarly, misclassifications increment the punishment accumulators of the path that led to wrong outputs. Weights remain unchanged during the training process, only accumulators are incremented.

The training process is generally done in one pass over the training examples. At the end of this sequential pass, the connections that had more punishments than rewards are pruned. The remaining connections have their weights changed using the accumulators.

2.4.2.1 The IRP learning

The IRP is used when the CNM neural network already exists and the goal is to add new knowledge to the network by learning new examples. In the case of the IRP, the learning proceeds as following.

If for a weight vector *w*, an example $e \in E$ yields $y(e,w) > \hat{y}(e)$, then every pathway *p* (connection among a combinatorial neuron and an output neuron), in the network is rewarded proportionally to:

- The response of the network, $y(e,w)$;
- The absolute value of the error, $\left| y(e,w) - \hat{y}(e) \right|$;
- The signal injected by *p* into the output neuron, $w_p \min\limits_{i \in I_p} s_{ip}(x_i(e))$.

On the other hand, when $y(e,w) < \hat{y}(e)$ each pathway of the network is punished in the same way.

At the end of each learning iteration, the synapses with the $R_p < P_p$ are deleted (pruned), and for the others, the accumulators are used to recalculate the value of w_p. The remaining pathways with $R_p > 0$ and $P_p = 0$ (*pathognomonic* pathways) have their weights updated based on Equation 2.25:

$$w_p = t + (1-t)\frac{R_p}{\max\limits_{1 \le q \le m} (R_q - P_q)}$$

Equation 2.25 – *Pathognomonic* pathway weight update

The constant *t* is an arbitrary acceptance threshold and $t \in (0,1]$. The other remaining pathways (*ordinary* pathways), have their weights updated based on Equation 2.26:

$$w_p = \frac{R_p - P_p}{\max\limits_{1 \le q \le m} (R_q - P_q)}$$

Equation 2.26 – *Ordinary* pathway weight update

At the end of the iteration, the values of R_p and P_p are passed on to the next iteration, enabling the incremental learning capability.

2.4.2.2 The SRP learning

The SRP learning applies the same general principle of rewards and punishments of the IRP learning in a one-iteration procedure. Before applying the SRP, the accumulators are set to zero and the weights are set to one. All the data examples are then applied to the network in a one-iteration procedure promoting the changes of the reward and punishment accumulators. After applying all examples, the same weight updates used on IRP are applied: the non-rewarded pathways are pruned and the others have their weights modified by Equations 2.25 and 2.26 above. The SRP is the starting point for the CNM learning. After its application, the IRP can be used to increment the CNM knowledge.

Important aspects to consider on the object model for the CNM are:

- The input neurons just bypass the information. Usually this information is normalized values among 0 and 1, resulting from a pre-processing of the input data.
- The combinatorial neurons implement the fuzzy AND function and the output neurons operate the fuzzy OR function.
- The connections among the neurons are feedforward, there are no feedback connections and the network is not fully connected.
- The number of input and output neurons is determined based on the domain knowledge the expert has, and the combinatorial neurons are created as the possible combinations, from zero to a given order number (typically 3), of the input neurons. The generation of the combinatorial layer may demand enormous memory footprint to be able to generate the neurons and synapses to all necessary combinations.
- The CNM synapses have reward and punishment accumulators that are used to decide whether the synapse must be pruned or not, and also to recalculate the new value for the synaptic weight.

2.4.3 The Self-Organizing Feature Maps

This kind of neural network is inspired on the brain's cortical structure. The different sensory inputs like visual or acoustic are mapped in different areas of the cerebral cortex in a topologically ordered manner. The Kohonen Self-Organizing Feature Map (SOM) (Kohonen, 1990) was built to operate in an analog way. The network is based on a usually two-dimensional lattice structure. The learning is competitive where the neurons are selectively tuned to various input patterns (vectors) or classes. The different input features force the locations of the winning neurons to be ordered in respect to each other in a meaningful way, so that the different features are mapped to different regions of the lattice. The spatial locations of the neurons correspond to the features given in the input patterns.

An important characteristic of the SOM neural network is the presence of lateral feedback in the learning process. The lateral feedback is a special form of feedback that is dependent on lateral distance from the point of its application. The lattice network has "imaginary" lateral connections among the neurons. The neuron on the map that best fits the feature of the pattern being presented to the network is considered the network *winner*. The neurons that are located around the winner neuron shall receive excitatory or inhibitory effects depending on the distance of the winner neuron through the lateral connections. This effect is due to the called *bubble activity* the network tends to have by concentrating the electrical activity into local clusters. Exciting or inhibiting the neurons around the winner forms the feature clusters. Typically the Gaussian function shown in Figure 2.3(e) (Activation functions) is used to describe the lateral feedback.

The Kohonen SOM neural model basic functionality can be described as follows:

- The input of the network can be of any dimensionality while a one or two-dimensional lattice of neurons is responsible for computing simple discriminant functions of the input.
- A mechanism compares these discriminant functions and selects the neuron with the largest discriminant function value.
- The selected neuron and its neighbors are activated simultaneously.
- An adaptive function increases the discriminant function values of the selected neurons in relation to the input signal.

The Kohonen SOM neural network structure can be seen in Figure 2.8 (Lattice network) that shows the lattice architecture or in Figure 2.13 below. Figure 2.13 tries to show that the SOM network is fully connected, all neurons on the input layer are connected to all neurons on the output (lattice) layer. It shows also the connections among the neurons on the output layer that will be responsible for the lateral feedback.

The number of input neurons is determined by the size of the input vector. The input vector represents the data pattern and can be of any dimensionality. The output map of neurons may have the dimensionality augmented to three, but the two-dimensional map is the most frequently used.

The first step on the SOM learning is to find the winner neuron or the neuron that best matches a certain criterion. The winner neuron will determine the location of the activation bubble on the map. The network connections are initialized with random values so that no representation is given on the map at the beginning.

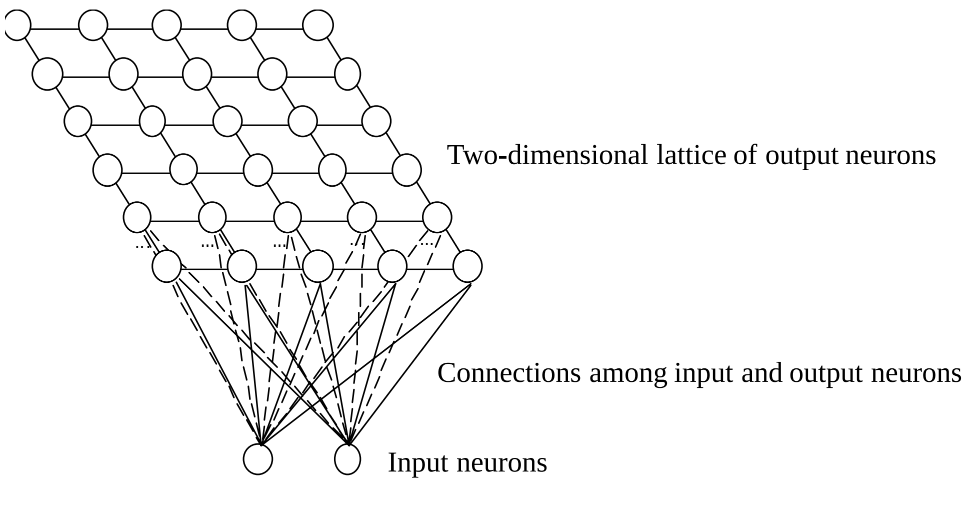

Two-dimensional lattice of output neurons
Connections among input and output neurons
Input neurons

Figure 2.13 – Kohonen SOM

When the input pattern is applied to the SOM input, the values are propagated to all neurons on the lattice layer. The lattice layer can compute the arriving input signals by simply calculating the inner product to all output neurons. Considering that the threshold is the same for all the output neurons, the output neuron that produces the largest inner product is the one that best fits to the pattern and is selected as the winner. A second form of best-matching is to use the *minimum Euclidean distance* between the vectors formed by the input signal and the weights at the arriving connections of each output neuron. In this case, the input vector and the network weights shall be normalized to constant Euclidean norm (length). Equation 2.27 shows how to calculate the winner neuron *i* for a given input vector *x* using the minimum Euclidean distance.

$$i(x) = \arg_j \min \left\| x - w_j \right\|$$

Equation 2.27 – Winner neuron calculation using the *minimum Euclidean distance*

In Equation 2.27 above, *x* is the learning vector and w_j is the vector formed by the connections weights among the input nodes and the output neuron with index *j*.

The Euclidean distance for a given output neuron with index *j* can be given by Equation 2.28 below, where *p* is the number of input neurons, $w_{j,i}$ is the weight of the connection among the *j*th output neurons and the *i*th input neuron, and x_i is the input value applied to the *i*th input neuron.

$$\left\| x - w_j \right\| = \sqrt{\sum_{i=1}^{p} (x_i - w_{j,i})^2}$$

Equation 2.28 – Euclidean distance

The topology of the lattice determines the neurons that are the neighbors of the winner neuron *i*. The weights of the connections to the winner neuron are to be rewarded by being this neuron the best matching to the given input. The neighbor neurons also shall receive some sort of reward forming the activation bubble together with the winner. By the lateral feedback connections, the neighbor neurons will be adapted to the given input. Let the lateral distance of the neuron *j* from the winning neuron *i* be denoted by $d_{j,i}$. The amplitude of the topological neighborhood centered on the winner neuron *i* is denoted by $\pi_{j,i}$.

Typically the neighborhood function is given by the Gaussian-type function shown in Equation 2.29.

$$\pi_{j,i} = \exp\left(-\frac{d_{j,i}^2}{2\sigma^2} \right)$$

Equation 2.29 – Gaussian-type function for the topological neighborhood

The parameter σ at Equation 2.29 is the efective width (variance) of the topological neighborhood. The topological neighborhood $\pi_{j,i}$ is maximum at the winning neuron ($d_{j,i}$ =

0). The amplitude of the topological neighborhood decreases with increasing lateral distance $d_{j,i}$, decaying to zero for $d_{j,i} \to \infty$.

The update of the synaptic weight vector w_j of neuron j at lateral distance $d_{j,i}$ from the winning neuron i*(x)* is given by Equation 2.30 below, where n denotes the discrete time and η is the learning-rate parameter of the algorithm.

$$w_j(n+1) = w_j(n) + \eta(n)\pi_{j,i(x)}(n)[x(n) - w_j(n)]$$

Equation 2.30 – Weight update function

The time-dependent learning-rate parameter $\eta(n)$ used to update the synaptic weight vector $w_j(n)$ shall be time-varying. During the first 1000 iterations it shall assume a value near to the unity and be decreased gradually until a value above 0.1. The width of the topological neighborhood $\sigma(n)$ shall also decrease slowly during the learning process. Typically, the function used for the calculation of both parameters is the exponential decay, described at Equations 2.31 and 2.32.

$$\eta(n) = \eta_0 \exp\left(-\frac{n}{\tau_1}\right)$$

Equation 2.31 – Learning-rate parameter update

$$\sigma(n) = \sigma_0 \exp\left(-\frac{n}{\tau_2}\right)$$

Equation 2.32 – Topological neighborhood parameter update

In Equations 2.31 and 2.32 above, the values σ_0 and η_0 are respective values of $\sigma(n)$ and $\eta(n)$ at the initiation of the algorithm (n=0). The values τ_1 and τ_2 are the time constants for the parameters η and σ respectively. Equations 2.31 and 2.32 shall be used only during the ordering phase, the first 1000 iterations or so, after that a small value shall be used for many iterations.

Important aspects to consider on the object model for the SOM are:

- The input nodes just bypass the information. Usually this information is normalized values among 0 and 1, resulting from a pre-processing of the input data.

- The output neurons implement the Euclidean distance to find out its activation value.

- The connections among the neurons of the input and output layers are feedforward and fully connected, there are feedback connections among the neurons of the output layer. The lateral feedback is a novel characteristic in relation to the previous models seen so far. There is no hidden layer.

- The number of input neurons is determined by the expert based on the domain to apply the model. The number of output neurons is also determined empirically and is typically a two-dimensional map.

- The activation bubbles form the chunks of knowledge about the problem being treated by the network. Each chunk tends to map an input feature.

The given SOM equations presented here are from the Haykin, 1994.

2.4.4 The Adaptive Resonance Theory

The Adaptive Resonance Theory (ART) was proposed by (Grossberg, 1976; Grossberg, 1987). When proposing the ART model, Grossberg attempted to solve what he called the *stability-plasticity dilemma*. This dilemma is concerned with the neural network capacities of generalization (stability) and discrimination (plasticity). The stability property is responsible for the neural network capability of grouping similar patterns in the same category. The plasticity is the network capacity of creating new categories when new patterns are presented. The dilemma lives in the difficulty of having a neural model that is able to develop the two capabilities at the same time. It means the network shall: remain adaptive (plastic) in response to relevant input, yet remain stable in response to irrelevant input; know to switch between its plastic and its stable modes; retain its previously learned information while continuing to learn new information.

Regarding the stability and plasticity properties of the ART model Grossberg stated: "An ART system can adaptively switch between its stable and plastic modes. It is capable of plasticity in order to learn about significant new events, yet it can also remain stable in response to irrelevant events. In order to make this distinction an ART system is sensitive to *novelty*. It is capable, without a teacher, of distinguishing between familiar and unfamiliar events, as well as between expected and unexpected events." (Grossberg, 1987)

Grossberg introduced a feedback mechanism between the competitive and the input layers of the network to cope with the stability-plasticity dilemma. This feedback mechanism controls the learning in a way that new information is learned without destroying old information. Several models were derived from the Adaptive Resonance Theory (Grossberg, 1987; Carpenter and Grossberg, 1987; Carpenter and Grossberg, 1992):

- ART1 – Is characterized by using only binary input.
- ART2 – Is able to process analog input.
- ART3 – Considers the action of the neurotransmitters on its synaptic mechanisms.
- Fuzzy ART – Implements fuzzy concepts in ART1 architecture.
- ARTMAP – Predictive architecture based on two ART modules.
- Fuzzy ARTMAP - Predictive architecture based on two Fuzzy ART modules.

The name "resonant" on the model comes from physics where the resonance occurs when a small-amplitude vibration of the proper frequency causes a large-amplitude vibration in a mechanical or electrical system. In the ART network, the signals reverberate back and forth between neurons of the input and competitive layers. In this process, the network tries to stabilize by developing the proper pattern. If it does so, a stable oscillation ensues, which is the neural network equivalent of resonance. During the resonant period the learning or adaptation occurs. Before the network has achieved the resonant state no learning occurs.

If a completely new pattern is presented to the ART network, it first tries to find on its internal representations the matching for the pattern. If the network does not find one, it enters in a resonant state in order to develop a new internal representation for the pattern. If the network has previously learned to recognize a pattern, then it will quickly enter in resonance and will reinforce the previously created internal representation.

In the ART model, two subsystems interact in the processing of the input patterns: the attention subsystem and the orientation subsystem. The attention subsystem is responsible for the processing of familiar input patterns, establishing responses and precise internal representations for those patterns. The orientation subsystem copes with the unknown patterns, resetting the attention subsystem when such patterns are presented to the network.

An important role performed by the ART systems is to distinguish in a certain set of patterns what is signal and what is noise. To turn this possible, the context of the patterns was considered on the ART definition. Certain characteristics can be considered as noise when presented in certain patterns and considered as important signal feature for other patterns. The relevant information that shall be distinguished is called *critical feature patterns*, or prototypes and represent invariants of the set of all experienced input patterns. As the learning process takes place, new equilibrium points are formed as the system discovers and learns the critical features. The learned features are stabilized by internal mechanisms that internally control the learned features and avoid possible sources of system instability.

Next, one model was picked from the ART family to be implemented in this work. The fourth model to be implemented is the ART1. The implementation of this ART model shall prove the capacity of the designed ANN framework to cope with feedback connections. Having this first ART model implemented, it is then easier to go for the implementation of the several other models of the ART family.

2.4.4.1 The ART1

In the ART architecture (Figure 2.14), two distinct processes occur when an input pattern is applied. The bottom-up process, also known as the adaptive filter or process of contrast enhancement, that produces the *Y* pattern, and the top-down process, that realizes a similarity operation to produce the pattern X*.

Given a set of input patterns, a certain pattern belonging to this set is represented by *I*. When pattern *I* enters *F1* (attention subsystem), it is then called pattern *X* (*I* and *X* are then identical). The neurons activated by the pattern *X* in *F1* generate output signals that flow through the connections *F1*→ *F2* (long-term-memory). This process produces pattern *Y* on layer *F2*. This pattern *Y* is the result of a *winner-take-all* process where only one neuron in *F2* can be the winner (the one with bigger activation). Only the winner neuron produces the value 1 on its output, the other neurons on the same layer produce 0. Here the bottom-up process finishes and the top-down process starts.

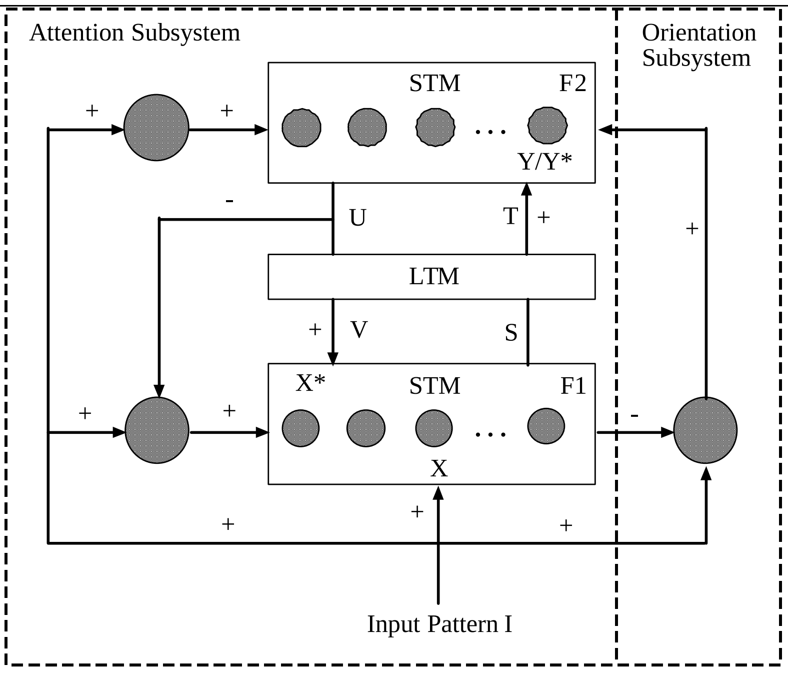


Figure 2.14 – ART1 Architecture

From the winner neuron a top-down pattern is produced through the *F2*→ *F1* connections (feedback connections). This new signal is called pattern *V*. Now, there are two stimulation sources for the input layer *F1*: the input pattern *I* and the feedback pattern *V*. Together they give origin on *F1* to pattern *X** that is typically different from pattern *X*. Pattern *X** indicates the similarity level between the input pattern and the stored prototype on the connection among layers *F2* and *F1*. The similarity process among those patterns can be simply the function AND (∩), or another similarity measure.

It is then necessary to decide whether the input pattern must be stabilized on the winner neuron or another neuron should be used. To verify the level of similarity represented on *X** it is necessary to define a rule that considers both *X** and *I*. It is possible to consider several different rules for accepting or rejecting the stabilization. This rule is called *the reset rule* and Inequality 2.33 commonly defines it.

$$\frac{|V \cap I|}{|I|} = \frac{|X^*|}{|I|} \geq \rho$$

Inequality 2.33 – Reset rule

In the Inequality 2.33, ϱ is the *vigilance parameter* that can assume values between 0 and 1, $|X^*|$ is the number of ones in the pattern X^*, and $| I |$ is the number of ones in the pattern *I*. If the result of this equation is true, the winner neuron is considered stable. Otherwise, it will not be considered by the orientation subsystem *A*, and the search for a stable pattern will continue. If no neuron is accepted as stable, then a neuron that is not used to any input pattern is then selected to learn this new input pattern. After choosing a stable pattern, the network learns the input pattern by adaptation of the bottom-up and top-down connections.

The 2/3 rule

Figure 2.15 shows the application of the *2/3 rule*. There are situations when the neurons on the *F1* layer must be deactivated, i.e., not producing output signals. For example, the neurons on *F1* do not have to generate outputs when excited only by the pattern *V*, but must be able to generate output when excited by the input pattern *I*. A mechanism called *gain control* (*B*) is implemented on the network to control such a situation. When *F2* is active, excitatory signals to *F1* are generated (pattern be *V*) and at the same time the gain control is inhibited (Figure 2.15(a)). The rule that controls layer *F1* is called *2/3 rule* and says that: two of the three signal sources available in *F1* (input pattern *I*, top-down pattern *V*, and the gain control pattern *B*), are necessary to activate the *F1* layer neurons producing an output pattern. These outputs from the neurons on the *F1* layer are called *supraliminal* signals. During the button-up process, *F1* receives an input that is the pattern *I* and an excitatory signal from the gain control, turning possible then the generation of the supraliminal signals (Figure 2.15(b)). During the similarity process of the top-down and bottom-up patterns, also supraliminal signals are generated (Figure 2.15(c)). Neurons that receive one bottom-up or one top-down input, not both, cannot generate the supraliminal signals.

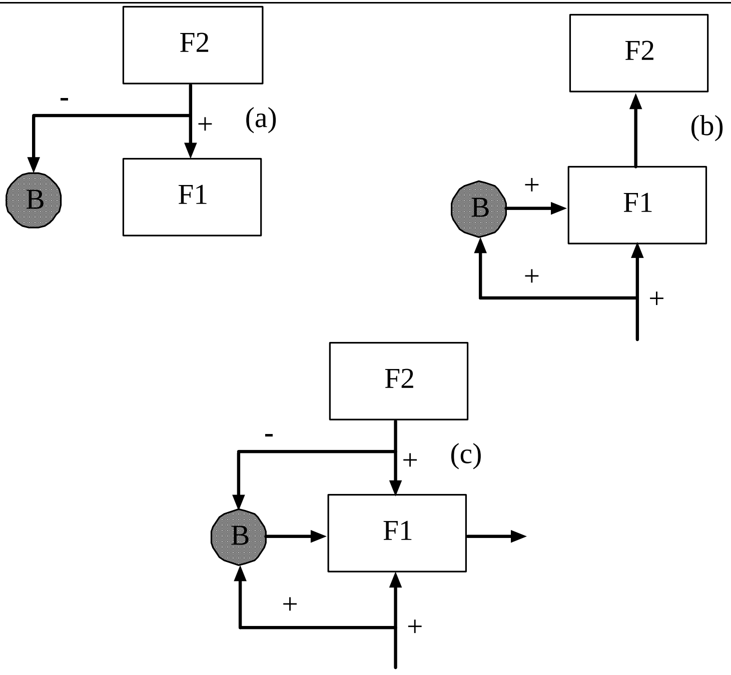


Figure 2.15 – The 2/3 rule

Learning on ART1

There are two ways of doing the learning on ART1: the slow and the fast learning. The slow learning allows for only a small amount of the input pattern to be represented by the weights. After repetitive presentations of the entire set of learning examples, the most important features of each pattern are captured by the LTM (long term memory). This form of learning is well suited to problems with high dimensionality, subject to learning sets with huge amounts of noise. The fast learning quickly encodes the input pattern features on the weights. This encoding typically is done in a one-shot-learning, which means, each input pattern of the learning set is presented to the network only once and the network is able to learn the pattern features. The fast learning is recommended to learning sets free of noise and when the learning must be done immediately.

Slow learning

The equations for the slow learning at the LTM and the STM (short-term-memory) described in (Carpenter and Grossberg, 1887) are as follows:

STM Equations

Equation 2.34 gives dynamic equation for calculating the activation of the STM xk of any neuron vk in F1 and F2.

$$\varepsilon \frac{d}{dt} x_k = -x_k + (1 - Ax_k)J_k^+ - (B + Cx_k)J_k^-$$

Equation 2.34 – STM activation

In Equation 2.34 above, J_k^+ is the total excitatory input to v_k, J_k^- is the total inhibitory input to v_k , and all parameters are positive. Parameter *A* controls the bottom-up and top-down signals, while *B* and *C* regulate the gain control. If *A > 0* and *C > 0,* then the STM activity $x_k(t)$ remains within the finite interval [$-BC^{-1}$, A^{-1}], regardless of the values of J_k^+ and J_k^-.

The neurons on F_1 are called v_i, where $i = 1,2,\ldots M$. The F_2 neurons are called v_j, where $j = M+1,M+2,\ldots N$. Thus Equation 2.34 above turns to be Equation 2.35 below for the neurons on the F_1:

$$\varepsilon \frac{d}{dt} x_i = -x_i + (1 - A_1 x_i)J_i^+ - (B_1 + C_1 x_i)J_i^-$$

Equation 2.35 – STM for neurons on the F_1

Similarly, Equation 2.34 above turns to be Equation 2.36 below for the neurons on the F_2:

$$\varepsilon \frac{d}{dt} x_j = -x_j + (1 - A_2 x_j)J_j^+ - (B_2 + C_2 x_j)J_j^-$$

Equation 2.36 – STM for neurons on the F_2

For a discrete calculation of F_1 an input vector I_i is applied and the activities are calculated according to Equation 2.37 below:

$$x_{1i} = \frac{I_i}{1 + A_1(I_i + B_1) + C_1}$$

Equation 2.37 – Activities for the neurons on the F_1

The J_k^+ input of the *i*th neuron v_i in F_1 is a sum of the bottom-up input I_i and the top-down input pattern V_i, represented by Equation 2.38 below:

$$J_i^+ = I_i + V_i$$

Equation 2.38 – F1 input

The top-down input pattern V_i is given by Equation 2.39 below:

$$V_i = D_1 \sum_j f(x_j) z_{ji}$$

Equation 2.39 – Top-down input pattern Vi

The *f(x)* on Equation 2.38 above is the signal generated by activity x_j of v_j, and z_{ji} is the LTM trace in the top-down connection from v_j of v_i. In the notation of Figure 2.14 (ART1 Architecture), the input pattern $I = (I_1, I_2,\ldots,I_M)$, the signal pattern $U = (f(x_{M+1}), f(x_{M+2}),\ldots, f(x_N))$, and the template pattern $V = (V_1,V_2,\ldots,V_M)$.

The inhibitory input J_i^- governs the attentional gain control signal and is calculated by Equation 2.40:

$$J_i^- = \sum_j f(x_j)$$

Equation 2.40 – F_1 attentional gain control

Thus $J_i^- = 0$ if and only if F_2 is inactive. When F_2 is active, $J_i^- > 0$ and hence term J_i^- in Equation 2.35 (STM for neurons on the F_1) has a nonspecific inhibitory effect on all the STM activities x_j of F_1.

The inputs and parameters of STM activities in F_2 are chosen so that the F_2 neuron that receives the largest input from F_1 wins the competition for the STM activity (winner-take-all).

The inputs J_j^+ and J_j^- to the F_2 neuron v_j have the form given in Equations 2.41 and 2.42 respectively:

$$J_j^+ = g(x_j) + T_j$$

Equation 2.41 – F_2 input

$$J_j^- = \sum_{k \neq j} g(x_k)$$

Equation 2.42 – F_2 attentional gain control

Input J_j^+ adds a positive feedback signal $g(x_j)$ from v_j to itself to the bottom-up adaptive filter input T_j given by Equation 2.43 below:

$$T_j = D_2 \sum_i h(x_i) z_{ij}$$

Equation 2.43 – Bottom-up adaptive filter input T_j

The $h(x_1)$ is the signal emitted by the F_1 neuron v_i and z_{ij} is the LTM trace in the connection from v_i to v_j . Input J_j^- adds up negative feedback signals $g(x_k)$ from all the other nodes in F_2.

In the notation of Figure 2.14 (ART1 Architecture), the output pattern $S = (h(x_1), h(x_2),\ldots, h(x_M))$, and the input pattern $T = (T_{M+1}, T_{M+2},\ldots, T_N)$.

LTM Equations

The LTM value for the bottom-up connection from v_i and v_j follows Equation 2.44:

$$\frac{d}{dt} z_{ij} = K_1 f(x_j)[h(x_i) - E_{ij} z_{ij}]$$

Equation 2.44 – LTM for bottom-up connections

Similarly, the LTM value for the top-down connection from v_j and v_i follows Equation 2.45:

$$\frac{d}{dt} z_{ji} = K_2 f(x_j)[h(x_i) - E_{ji} z_{ji}]$$

Equation 2.45 – LTM for top-down connections

On Equations 2.44 and 2.45 above, K_1 and K_2 are positive constants that control the learning rate, E_{ij} and E_{ji} are positive constants that control the decay rule (see Carpenter and

Grossberg, 1987), z_{ij} is the value for the connection from v_i to v_j , and z_{ji} is the value for the connection from v_j to v_i. The threshold function $h(x_i)$ applied in the F_1 neurons is the sigmoid. The term $f(x_j)$ is a postsynaptic sampling, or learning, signal because $f(x_j) = 0$ implies $\frac{d}{dt}z_{ij} = 0$. The term $f(x_j)$ is also the output signals of v_j to connections from v_j to F_1 , as in Equation 2.39 (Top-down input pattern V_i).

For the top-down connection, the simplest choice for the values of K_2 and E_{ji} is to make both equal to 1. For the bottom-up connection a more complex choice of determining E_{ij} is made. This choice considers the Weber Law Rule (Carpenter and Grossberg, 1987). This rule requires that positive bottom-up LTM values learned during the encoding of an F_1 pattern *X,* with a smaller number | X | of active neurons, be larger than the LTM values learned during the encoding of an F_1 pattern with a larger number of active nodes. At least those values shall be similar. Thus, the Weber rule ensures that input patterns I_1 that are subsets of other input patterns I_2 activate its own features.

Fast Learning

For the fast learning to work properly, the bottom-up and top-down connections must be properly initialized. It is necessary to ensure that the encoded patterns are alwa2ys properly accessed. The bottom-up connections z_{ij} from v_i to v_j shall follow initialization conditions called *Direct Access Inequality*, defined in Inequality 2.46:

$$0 < z_{ij} < \frac{L}{(L-1+n)}$$

Inequality 2.46 – Direct Access Inequality

In Inequality 2.46 above, the constant L is bigger than 1 and typically is equal to 2, and *n* is the number of neurons in F_1. This value is very critic because if it is too large, the network may allocate all the neurons of the F_2 for a unique input pattern.

The initial conditions for the values of the top-down connections z_{ji} from v_j to v_i shall follow the *Template Learning Inequality* defined in Inequality 2.47 below:

$$\frac{B_1 - 1}{D_1} < Z_{ji} \le 1$$

Inequality 2.47 – Template Learning Inequality

It is suggested that z_{ji} be initialized with 1.

The following Equation 2.48 gives the codification of the LTM bottom-up connections:

$$z_{ij} = \begin{cases} \dfrac{L}{(L-1+|X|)} & \text{if} \quad v_i = v_j = 1 \\ 0 & \text{if} \quad v_i = 0 \quad \text{and} \quad v_j = 1 \\ z_{ij} & \text{if} \quad v_j = 0 \end{cases}$$

Equation 2.48 – Bottom-up codification

In Equation 2.48 above, $|X|$ is the number of components in vector X. There is no modification in z_{ij} when v_j in F_2 is inactive.

Similarly, the following Equation 2.49 gives the codification of the LTM top-down connections:

$$z_{ji} = \begin{cases} 1 & \text{if} \quad v_i = v_j = 1 \\ 0 & \text{if} \quad v_i = 0 \quad \text{and} \quad v_j = 1 \\ z_{ji} & \text{if} \quad v_j = 0 \end{cases}$$

Equation 2.49 – Top-down codification

There is no modification in z_{ji} when v_j in F_2 is inactive.

Important aspects to consider on the object model for the ART are the following:

- The input nodes do not simple bypass the information as in the models seen so far. The input nodes have to cope with feedback information coming from the competitive layer.
- There are special equations for the processing of the neurons of both input and competitive layers.
- There are feedforward and feedback connections among neurons of the input and competitive layers. The layers are fully connected. The feedforward and feedback connections process the signals traveling on them.

- There is lateral feedback connection among the neurons of the competitive layer similarly to what happens to the competitive layer of the SOM model.
- The number of input neurons is determined by the expert based on the domain to apply the model. The number of output neurons is also determined empirically.
- The input values can be binary, analog or even fuzzy depending on the ART model implemented.
- There are connections among the neurons of the input and competitive layers, and also connections between the layers and the control mechanisms of the network that can be seen as special neurons drawn in Figure 2.14 (ART1 Architecture).

2.5 Software Engineering issues

This section introduces Software Engineering concepts that are relevant to the realm of the thesis. The goal here is not to extensively discuss these concepts, but only to put them together in a way that makes clear the engineering principles that form the basis of this work.

2.5.1 Software Quality

Software quality is the final objective of software engineering. A high quality software system is more stable, has fewer bugs, is easier to understand, maintain and extend, and attend user's needs in functional and efficiency terms. (Pree, 1995)

Software quality may be achieved through careful planning, design, implementation and testing. Each of these phases is expensive and time-consuming. So how can increasingly complex software be delivered in time and with the expected quality? The answer is: through reuse of well-defined, well-designed and well-tested units of software.

In order to achieve effective software reuse, the software must have some capability to adapt to a variety of situations. This aspect of software flexibility is one of the keys to high quality software. Flexible software is not only more reusable, it is also more easily maintained and extended as it was built with change in mind. Programming including flexible points in the software can sometimes increase the costs of initial development, but often this cost is amortized if the system needs to be adapted or extended.

2.5.2 Flexible software

To create a flexible piece of software, one must be able to spot which parts of the system are more likely to change, and which technique can be used more effectively in that situation. There are a variety of techniques. Some of them can be applied to different languages and methodologies, some only work in specific environments (e.g. object-oriented frameworks). This section sketches a selection of these techniques, with emphasis on the framework-based ones.

2.5.2.1 Flexibility based on data

The Von Neumann model, which most computing systems follow nowadays, states that programs and data share the same memory space. That view of the program as data leads to the duality of programs and data that "means that one program can process another as input. The output data could be a transformed program" (Pree, 1991). Using the duality concept, it is possible to make flexible software whose behavior can change based on the inputs that the software receives. This kind of flexibility requires no recompilation, no access to the software source code and can be implemented in virtually any language. Two ways of data-based flexibility can be cited:

Resources

The data that will be interpreted is stored in resource files. The kind of customization that can be done in this way ranges from simple data entry for a calculation program, up to complex workflow definition or graphic user interface customizations. In the latter cases, the resource files are often edited using special customization tools or editors.

For instance, a typical use of resource bundles is the internationalization of the GUI. The different words and sentences that form the GUI are stored in different resource bundles and loaded at runtime in order to build the user interface according to the user's preferences, in this case according to the particular language.

Scripting languages

Another kind of program that can be stored as data is a script. Although a script file can be viewed as a kind of resource file, it deserves a distinction because of some unique characteristics. An explicit interpreter module that is not strongly coupled with the rest of the application usually processes a script. A script language is the set of rules that describe whether a resource is valid, and what actions are to be taken for each command the script contains.

An example of scripting usage is the JavaScript that has been broadly used to bring interactivity to the Web application. It is used for creating live online applications that link together objects and resources on both clients and servers. JavaScript was designed for use by HTML page authors and enterprise application developers to dynamically script the behavior of objects running on either the client or the server. (http://java.sun.com/pr/1995/12/pr951204-03.html)

Microsoft's counterpart is VBScript, which offers active scripting to various environments, such as Web client scripting in Microsoft Internet Explorer and Web server scripting in Microsoft Internet Information Service. (http://msdn.microsoft.com/library/default.asp?url=/library/en-us/script56/html/vbswhat.asp)

2.5.2.2 State of the art programming concepts

Some of the techniques to achieve reusability and flexibility are based on language or paradigm specific features. This section explains some of them and shows some of their inherent advantages and disadvantages.

Class Libraries

Class libraries are probably the most common technique for making reuse happen, as virtually every object-oriented language has one or more. Their main goal is re-usability, so the flexibility of each subset of classes must be maximized. Ordinary class libraries don't provide any application structure. An example is the Java language API with libraries such as: swing (GUI), io, lang, math, etc.

Components

Szyperski (1998) defined software component as:

"A unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

The definition brings the attention to the core properties of a component such as: independence, contractual interfaces, composition with third-parties and deployment. The component shall be a unit of independent deployment. They are built to be used by different systems that need the functionality implemented by the component. A component shall not be dependent on other components or environments. Components are called units because they cannot be deployed partially and the third party is not expected to have access to a component's details.

The component contractual interface allows its composition with third-party components given that they match and that the interfaces are well documented. A component has to encapsulate its implementation and interact with the environment through well-defined interfaces.

Therefore the component properties are (Szyperski, 1998):

- The component is a unit of independent deployment.
- The component is a unit of third-party composition.
- The component has no persistent state.

Components can correspond to objects and their classes. However, there is no need for a component to be a class or consist of classes. The fact that components are used in software development does not directly imply that the resulting software is of high quality or has higher flexibility. The whole architecture must be consistent and the components must themselves have a sound structure and be flexible enough to attend the needs of that particular application. Like class libraries, the components tend to provide flexible, easy to use software, without imposing any specific application structure. The main advantage over simple class libraries is the higher level of encapsulation that is achieved.

Components require a standard for connecting them. There are currently three standards: CORBA, COM and JavaBeans. Each of these standards has its specific interfaces, services, platform dependencies, etc. For an extensive study and comparison the book by Szyperski (1998) is recommended.

Frameworks

A component that can fit the definition of well-defined, well-designed and well-tested units of software is the *framework*. Following, (Gamma et al. 1995) defined a framework as "a set of cooperating classes that make up a reusable design for a specific class of software". That definition regards frameworks as large pieces of software, with as many responsibilities as the whole application class for which it was built may have. Defining frameworks as above limits their overall usability, because it is unlikely that two or more large, complex sets of classes can be combined together without a significant effort to bridge them.

A less constrict and perhaps more useful definition can be found at (Wirfs-Brock and Johnson, 1990). It states, "A framework is a collection of abstract and concrete classes and the interface between them, is the design for a subsystem". This definition allows more reuse by allowing different modules, each of which is responsible for a related set of tasks.

But also this concept has some limitations when they have to be combined, as the designs of the different subsystems may not be fully compatible. Often a framework assumes that it has the main control of an application. Two or more frameworks making this assumption are difficult to combine without breaking their integrity. (Pree and Koskimies, 1999)

Frameworks predefine an architecture for the application, so they encapsulate part of the design, as well as part of the code. Frameworks also cause the code to suffer an "inversion of control", as the framework is responsible for the control flow instead of the application. Examples of framework are: ET++ application framework (Weinand et. al. 1989) and many of the Java libraries.

White-box frameworks

Socalled White-box frameworks consist of a collection of incomplete (abstract) classes, that is, classes that contain methods without meaningful default implementations (Pree, 1996). To adapt the framework, the application developer extends those incomplete classes, implementing appropriate methods to satisfy the application needs. As a direct consequence, the user of the framework must be aware of its workings and structure. Figure 2.16 shows the creation of a subclass A1 in order to add behavior not implemented in the superclass A. The new behavior is added by overriding the necessary methods of the superclass A.

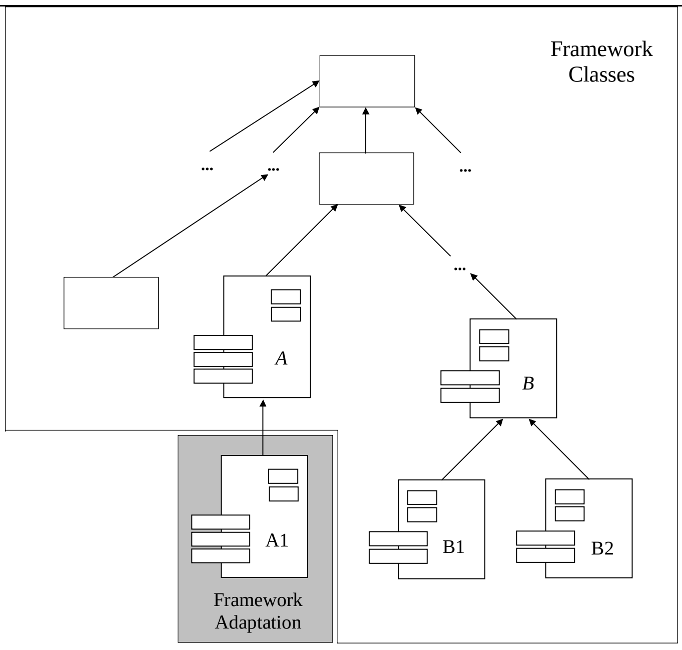


Figure 2.16 - Sample framework class hierarchy. (Pree, 1996)

Black-box frameworks

Black-box frameworks are similar to white-box frameworks, with the exception that they offer several ready-to-use components to be combined, instead of demanding them to be created through extension. Modifications are accomplished by *composition*, not by programming. Black-box frameworks are easier to reuse, if they have the right components. Most of the time, though, frameworks have many white-box aspects in the beginning and move towards black-box, as some often used components are incorporated.

The example below, picked from Pree (1996), shows the adaptation of a framework by composition of white-box and black-box components. In the framework class hierarchy in Figure 2.16, class B already has two subclasses B1 and B2 that provide default implementations of B's abstract method. Suppose that the framework components interacted as depicted in Figure 2.17(a), a programmer would adapt this framework, for example, by instantiating classes A1 and B2 and plugging in the corresponding objects (see Figure 2.17(b)). In the case of class B, the framework provides ready-to-use subclasses; in the case of class A the programmer has to subclass A first.

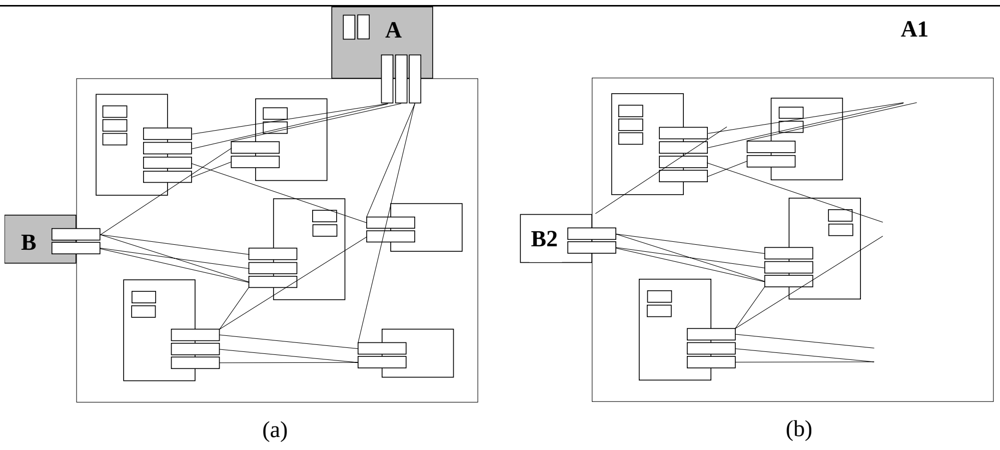


Figure 2.17 - Framework (a) before and (b) after specialization by composition. (Pree, 1996)

Hot Spots

Hot spots are the "aspects of an application domain that have to be kept flexible" (Pree, 1996 and 1997). To find the hot spots is a priority when designing a framework, as it is not cost-effective to make the framework flexible in every conceivable way. The complexity of keeping the framework exceedingly flexible also makes the framework harder to reuse and maintain. The design using hot spots is explained further in Section 2.5.4.

Framelets

A framelet is simply a small framework. According to (Pree and Koskimies, 1999), "In contrast to a conventional framework, a framelet is small in size (< 10 classes), does not assume main control of an application and has a clearly defined simple interface." So that framelets represent a way for modularizing frameworks.

Framelets, like classes in a class library, can be arranged into families, like a family of framelets that processes documents. There may be a framelet responsible for managing payment notices; another framelet that handles invoices; another that treats purchase orders, etc.

2.5.3 Framework construction patterns

It is necessary to know how to construct frameworks that can be reused in a variety of domains. So that it is important to understand the essential framework construction principles. Socalled design patterns apply the construction principles in various situations.

The implementation of frameworks relies on "abstract classes" and "abstract coupling". Abstract class is a class that contains one or more methods without a meaningful implementation. These methods are the socalled hooks that the application-defined components must implement, through extension of the abstract class, to provide their application-specific code. As the application-specific class extends the abstract class, it automatically inherits all method signatures. This relationship is called "abstract coupling", and is the basis for both white-box and black-box design. Abstract coupling allows the substitution of the base class for any other class that contains the expected method signatures, without any change in the code that exercises that base class.

2.5.3.1 Hook combination patterns

As was shown before, the main objective of using hook-template pairs is to provide flexibility. The code for the hook and template may be in the same class or in different classes. In this subsection, the class containing the code for the template method is called T, and the class containing the code for the hook method is called H. (Pree, 1995)

Unification

When the template and hook methods are unified in the same class, the combination principle is called "Unification". The implication of using this pattern is that it requires adaptations through inheritance, requiring an application restart to accomplish any change. Figure 2.18(a) shows the unification of the template and hook classes in the same class. (Pree, 1996).

Those methods that meant to be extended are called "hook methods", and the methods that call them are called "template methods" (Gamma et. al. 1995). Template methods define the behavior, flow of control or the interaction of objects that must be common for any of the classes that extend the base class. This allows the extended classes to change the hook method behavior without changing the source code for the class that implements the template method.

Separation

When the code for the template and hook methods is kept in different classes, it provides an implementation to the combination principle called "Separation". Separation corresponds to an abstract coupling between the template and hook classes, allowing the template class behavior to change by composition, instead of by inheritance. This also allows runtime substitution of H, instead of requiring a code change, which implies an application restart. Figure 2.18(b) shows the separation of the template and hook classes. (Pree, 1996)

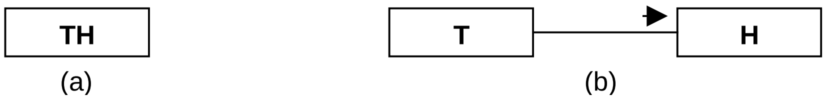


Figure 2.18 – (a) Unification and (b) Separation of template and hook classes. (Pree, 1996)

Recursive combination patterns

It is possible to make the template class as a descendant of a hook class (see Figure 2.19(a)). In the degenerated version of the recursive combination, template and hook classes are unified (see Figure 2.19(b)). This situation can lead to a set of recursive combinations that allow the building of directed graphs of objects. Then, instead of plugging a simple object, it is possible to compose directed graphs of objects. The template and hook methods must have the same signature to make the forwarding work, even if the code for the template and hook are separated. (Pree, 1996)

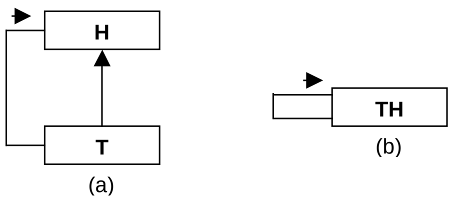


Figure 2.19 – Recursive combinations of template and hook classes. (Pree, 1996)

Chain of collaboration

The chain of collaboration happens when the template and hook classes are unified in the same class and each unified class can refer to another 0 or 1 class, leading to a line of objects that collaborate to solve a specific problem. The main difference between this chain of collaboration and the behavior composition is that "TH objects can be viewed as equally ranked and interchangeable in the sense that each TH object can refer to another TH object" (Pree, 1996). This construction also makes the T and H methods to be the same: the method should verify if it is possible to help to solve the problem and act, before it forwards the request to the next object in line.

2.5.3.2 Construction principles and the GoF design-patterns

Many of the design-patterns shown in the GoF ("The gang of four", Gamma et al. 1995) book are examples for small frameworks that apply the above construction principles. The design-patterns vary only on the semantics for the hot spots. (Pree, 1996)

GoF patterns with a template-hook unification

The most obvious pattern from the GoF catalog that is based on the unification principle is the Factory Method pattern. The unified template and hook class is called *Creator*, and the hook method *Factorymethod()*. (Pree, 1996)

GoF patterns with a template-hook separation

Most GoF patterns, though, are based on the Separation. Among them, Bridge, Abstract Factory, Builder, Command, Interpreter, Observer, Prototype, State and Strategy. Table 2.1 below shows how the catalog patterns and its corresponding hook method and class, besides its template class and hot spot semantic. (Pree, 1996)

GoF patterns with recursive template-hook combinations

Table 2.1 – Naming issues of catalog entry. (Pree, 1996)

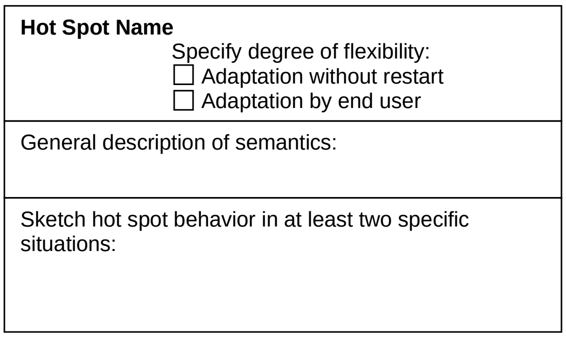
Catalog Entry	*Hook Class*	*Hook Method*	*Template Class*	*Hot Spot Semantics*
Abstract Factory	AbstractFactory	CreateProduct()	Client	Families of product objects
Builder	Builder	BuildPart()	Director	How a complex object is created
Command	Command	Execute()	Invoker	When and how a request is fulfilled
Interpreter	AbstractExpression	Interpret(…)	Client	Interpretation of a language
Observer	Observer	Update()	Subject	How the dependent objects stay up to date
Prototype	Prototype	Clone()	Client	Class of object that is instantiated
State	State	Handle()	Context	States of an object
Strategy	Strategy	AlgorithmInterface()	Context	An algorithm

Some of the design-patterns apply the recursive combinations shown before. When template and hook classes are separated and a template object can refer to any number of hook objects, the pattern is the Composite. When template and hook classes are separated, but each template object can refer to at most one hook object, the pattern is Decorator.

When the template and hook classes are unified and each object can refer to at most one other object, it is named Chain of Responsibility. (Pree, 1996)

2.5.4 Hot-spot-driven design

The problem with frameworks design is how to balance the flexibility versus the complexity of use of that framework. Less flexible solutions tend to keep the framework smaller and diminish the cost of maintenance and reuse. To find out the most important hot spots then becomes a major issue when designing a framework. One technique shown at (Pree, 1996, 1997 and 2000) is to use hot spot cards, to express both data and behavior variation points (see Figure 2.20). It is almost certain that a framework will need a number of iterations before it is considered well designed. Important hints to cut that number of iterations are: asking the right questions to the right people (it seems obvious, although it is not always easy to find who the right person may be); investigate similar use cases, trying to spot the differences between them; examine maintenance of old similar systems, the spots that change more often are likely the spots where flexibility is more required.



Hot Spot Name

Specify degree of flexibility:
☐ Adaptation without restart
☐ Adaptation by end user

General description of semantics:

Sketch hot spot behavior in at least two specific situations:

Figure 2.20 – Layout of function hot spot card. (Pree, 1996)

Hot spot cards are a tool to improve communication between domain experts and software engineers, but they lack the capability to express recursive combinations. So, according to (Pree, 1996), it is up to the software engineer to use them in order to produce more elegant and flexible architectures.

2.5.5 Applying Software Engineering Issues

Along this thesis, the above concepts are extensively applied. Especially in Chapter 3 where the CANN framework architecture and implementation is discussed, most of those concepts can be identified such as: classes, components, design patterns and framelets.

The CANN framework was completely built taking care of properly applying these concepts from the very beginning. The first design rounds of the ANN framework construction were done using the hot-spot-driven design to identify its flexibility points. By using UML the object model was refined many times during the sessions of discussion between the ANN and SWE experts. The result is an object model where some framelets are identified.

The identified framelets were carefully implemented in Java. The implementation took care of applying the proper design patterns for each desired semantic situation. As Java evolved to implement its components model, JavaBeans, the principal objects evolved to Java components.

In order to validate the components and framelets, four different ANN models were implemented using them. Besides this, an ANN simulation environment was built. To conclude, study cases under well-controlled environment validated the implementations.

The result of the whole implementation effort is a flexible ANN implementation and simulation environment, as well as a set of SWE implementation and design experiences gained from the application of the construction of ANN software.

3 The CANN solution

This chapter explains in details the implementation of the CANN (Components for Artificial Neural Networks). It shows how each of the core frameworks where defined and implemented. At the end, a comparison to other approaches are presented.

3.1 Introduction

There are numerous books and papers about ANN implementations and ANN development, besides tools and freely available programs on the Internet that address the development of ANN or even one or another ANN model in object-oriented fashion. But most of this work is only a wrapping of C to C++ or Java code being limited as object-oriented design and architecture. This code, in general, is not reusable besides sometimes being very usable and efficient. Rogers (1997) classifies this approach as coarse-grained because the ANN objects encompass a great deal of functionality, and do not exploit the inherent object nature found in ANN.

The implementation of this coarse-grained software for ANN may happen for many reasons:

- Difficulty in approaching the problem of programming the algorithms with objects that represent connectivity, functionality and cooperation.
- Opposed to the coarse-grained approach, the implementation of such classes could lead to a fine-grained solution, which perhaps can be an inefficient solution. On the other hand, the fine-grained approach can be ideal to implement the ANN architecture to enhance existing ones because it is more elegant and flexible from the software engineering point of view. It is difficult to balance the two approaches.

Thus, the goal of implementing an object-oriented solution for ANN software is to build a set of tools to develop ANN and to provide design and programming studies that can help gradually increase the flexibility, reusability and velocity of different ANN system development efforts. In this work, the developed prototypical tool set is not complete and the implemented models do not cover everything in the ANN domain but are sufficient to have a deep study of the field to meet the goals of this work.

The basic principle in building on object-oriented architecture is to find out the commonalties in the application area, so that it is necessary to find the common aspects among the various ANN models. The first common elements that show up are the neurons and synapses. Later, other elements can also be defined such as layers of neurons, mathematical functions, similar interfaces to the external world, etc.

3.1.1 Why build Components for ANN?

An ANN implementation should be flexible enough to solve problems in several application domains. This chapter focuses on this aspect i.e., constructing a flexible ANN system. In ANN software development, it is common to redevelop models from scratch each time a different application must be accomplished. There are tools that try to avoid this and help on the main ANN development aspects offering some pre-defined building blocks. Unfortunately, in general, these tools are commercially available software and their structure is not open for analysis. Furthermore, ANN software developers usually:

- Think about only one neural model to solve a specific application problem.
- Come up with very specific implementations for a particular problem.
- Concentrate on ANN performance and not on the construction of different ANN models and its reusability in different problem domains.

Thus, object-oriented design and implementation have hardly been applied, without compromises, so far in this domain area. The intention of this work is to build a flexible object-oriented architecture in order to solve the following problems related to the implementation of ANN:

- The architecture should be flexible/extensible to deal with various neural models.
- Flexible ANN architectures that can change their structure and behavior at run time allow experiments to gain better results.

- It should be easy to have different neural network models distributed over a network. In this way, a suitable solution to a decision problem can be found more quickly.
- In the long run, the object-oriented architecture could form the basis of building up hierarchies of ANNs working together, cooperating, and acting as intelligent agents in a distributed environment.

Many important aspects of ANN development were already covered that certainly explain some reasons for developing ANN components. But no reason is stronger than experience. The experience of developing software and continuously adapting it to different situations in an infinite loop is sometimes frustrating. This adaptation means having to frequently migrate code from one platform to another, to adapt code to a new application domain, to adapt code to a specific user necessity, to adapt code to new improvements on the ANN theory, to adapt code to implement new ANN models, and so on. Later, the experiences collected by implementing a traditional software solution for ANN (the Hycones expert system) are briefly explained.

3.1.2 The Hycones system as starting point

Hycones (short for Hybrid Connectionist Expert System; Leão, 1993) is a sample hybrid system that is especially designed for classification decision problems. The core technology is artificial neural networks based on the Combinatorial Neural Model (Machado and Rocha 1989 and 1990). The experiments with this model proved that it is powerful for classification problems, providing good results ranging from medical diagnosis to credit analysis (Leão 1993a, Reátegui 1994, da Rosa 1995). This section begins with a discussion of the principal features of Hycones. Based on this overview the problems regarding its flexibility are outlined. These problems were encountered when Hycones was applied to different domains.

Hycones is a generator system in the sense that its inherent ANN system, the CNM, generates, upon demand, as many neurons as necessary to solve the domain problem. It can also generate other AI structures such as frames or semantic rules based on the knowledge built by the ANN. The fact that Hycones is a generator system already indicates that the design of a generic architecture constitutes an important goal right from the beginning. Unfortunately, the conventional design and implementation did not provide the required flexibility as will be outlined in Section 3.1.2.1.

The first step in using Hycones is to specify the input and output nodes of the ANN. In the case of a customer classification system, the input nodes correspond to the information on the order form. Hycones offers different data types (e.g., string values, fuzzy value ranges) to specify the input neurons, which maps the so-called findings. The output neurons, called hypotheses, correspond to the desired decision support. For example, in the case of a customer classification system, the customer categories become the output neurons.

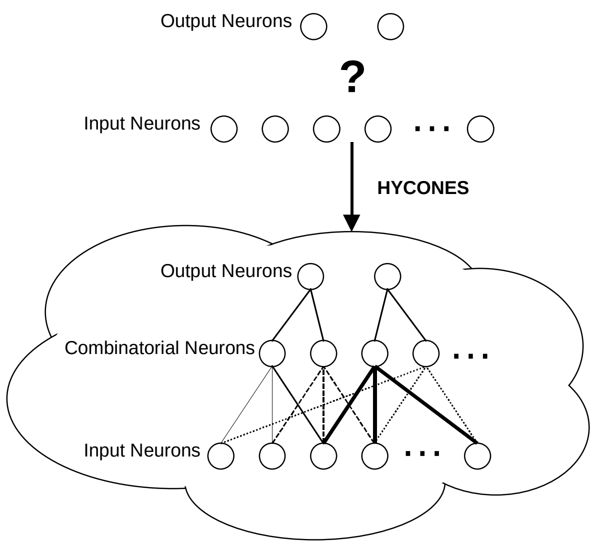


Figure 3.1 - Hycones as ANN generator.

Based on the information described above, Hycones generates the Combinatorial Neural Model (CNM) topology depending on some additional parameters (various thresholds, etc.). Figure 3.1 schematically illustrates this feature of Hycones. Each combination of input neurons contributes to the overall decision. CNM applies an inductive learning that is performed through the training of the generated ANN based on available data using a punishment and reward algorithm and an incremental learning algorithm (Machado and Rocha 1989). Inductive learning allows automatic knowledge acquisition and incremental learning.

Once the generated ANN is trained, Hycones pursues the following strategy to come up with a decision for one specific case (e.g. a customer): the ANN evaluates the case and calculates a confidence value for each hypothesis. The inference mechanism finds the winning hypothesis and returns the corresponding result.

Additional expert knowledge can be modeled in expert rules (Leão and Rocha 1990). For example, rules describing typical attributes of customers belonging to a particular category could be specified for the mail order decision support system. Such rules imply modifications of the weights in the ANN. Figure 3.2 exemplifies this Hycones property. The expert rule: *I3 & I4 & In => O2*, corresponds to the strengthened connections among the input nodes *I3*, *I4* and *In*, the combinatorial node *C3*, and the output node *O2* of an ANN.

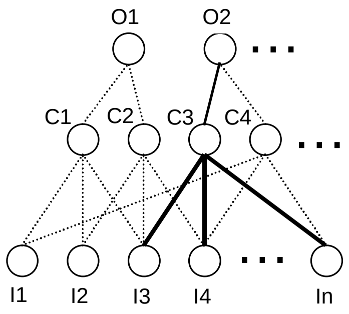


Figure 3.2 - Incorporating expert rules into the ANN topology.

3.1.2.1 Adaptation problems

Despite the intention of Hycones to be a reusable generator of decision support systems, the Hycones implementation had to be changed fundamentally for each application domain. In other words, the Hycones system had to be implemented almost from scratch for each new application domain. What are the reasons for this unsatisfying situation?

Limits of hardware & software resources

The first Hycones version was implemented in CLOS. CLOS simplifies the implementation of core parts of Hycones, but the execution time turns out to be insufficient for the domain problems at hand.

In subsequent versions of Hycones, parts of the system were even implemented on different platforms to overcome performance problems and memory limits. For example, the ANN training algorithm is implemented in C on a Unix workstation. C was chosen to gain execution speed. Other parts of Hycones, such as an interactive tool for domain modeling by means of specifying expert rules, are implemented on PCs and use Borland Delphi for building the GUI.

Complex conceptual modeling

This issue is also related to performance problems: Hycones manages the complex ANN topology by storing the information of all the connections and their corresponding weights in main memory. (Hycones provides no parallelization of ANN training and testing.) Due to memory limits, only parts of the ANN structure can be kept there and the rest is stored in a database. Roughly speaking, database records represent the connections and weights of one ANN topology. In general, this forms quite a complex conceptual model, involving overhead when swapping ANN weights between memory and database. The way in which the information about the generated ANN is stored in database tables had to be changed several times to be optimal for the database system in use. These changes are not only tedious, but also error-prone.

The fact that Hycones also became a hybrid system with regard to its implementation implies complex data shifting between different computing platforms. The parameters comprising the specification for the ANN generation are entered on PCs, but the ANN training and testing is done on Unix workstations. Finally, if the user prefers to work with the decision support system on PCs, the generated and trained ANN has to be transferred back from the Unix platform to the PC environment.

Neural network models

Hycones supports only one ANN model, the Combinatorial Neural Model, but it should be possible to choose from a set of ANN models, the one that is best suited for the decision support problem at hand. A version of Hycones (Guazelli and Leão, 1994) was developed using the ART (Grosberg, 1987). This version was hard coded, so the CNM and ART implementations are independent and nearly no reusability was applied.

Conversion of data

An application of Hycones requires providing of data for ANN training and testing. Of course, various different ways of dealing with these data have to be considered. For example, data are provided data in ASCII-format, in relational database tables, or in object databases. The data read from these sources must be converted to valid data for the ANN input. This conversion is done based on the domain knowledge, which also changes from application to application. Though this seems to be only a minor issue and a small part of the overall Hycones system, experience has proven that a significant part of the adaptation work deals with the conversion of training and test data.

In order to overcome the problems mentioned above, Hycones was completely redesigned based on framework construction principles. The problems of the conventional Hycones implementation form a good starting point for identifying the required hot spots. The next section presents the hot-spot-driven redesign and implementation of Hycones.

3.2 Design of a neural network framework architecture

The project where the components for ANN were developed was called Components for Artificial Neural Networks, CANN for short, usually referred in this work as *CANN project* and *CANN tool* for its software implementation. The goal of the CANN project was to build up a framework for implementing ANN and a simulator for running them. The simulation environment developed on the CANN was used as the test bed for the its components.

The hot spots of CANN can be summarized as follows:

- Data conversion: The simulation environment should provide flexible mechanisms for converting data from various sources. CANN should be able to support these mechanisms to feed data to the different ANN models.

- Domain modeling: It should be natural to model different problems and associate them to different data sources and ANN models.

- Inference engine: CANN should support several ANN models (as already pointed out, two separate Hycones versions implemented two different ANN models the Combinatorial Neural Model and the ART). The idea here is to have more than one model running on the same simulation environment at the same time. The ANN internal structure and behavior changes from model to model, but some significant aspects can be kept flexible in order to facilitate any ANN model implementation.

- Parallelism: In order to improve performance, parallelism shall be implemented at the level of ANN instances.

- Distribution and code mobility: Also to gain performance, a solution for the mobility of the different ANNs being trained on the simulation environment must be implemented.

- General GUI: The simulation of ANN shall be done in a very similar way for different ANN models. The actions applied over the ANN models are similar to the different ANN models so that a unique general GUI can be developed in order to manage any kind of ANN model. There are some aspects that are unique to each ANN model that shall be kept flexible such as the parameters configuration and the ANN structure visualization.

Figure 3.3 shows the frameworks considered essential to implement the hot spots listed above.

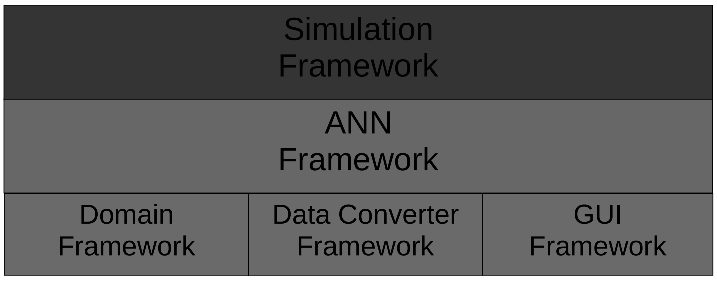


Figure 3.3 – CANN frameworks

Domain knowledge modeling and data conversion are parts of the architecture that are good candidates for defining general frameworks. The CANN shall concentrate on finding a generalized way of defining the domain so that it can be applied to any ANN model and domain problem. Data conversion shall be implemented in order to facilitate database and text file access for learning and testing data fetching. In this way, it is expected to solve the problem of implementing this part of the system for each new applied domain problem.

An ANN framework shall be defined in order to facilitate the implementation of different ANN models. Modeling the core entities of the ANN, that is, neurons and synapses, as objects solves the complex conceptual modeling of Hycones. Instead of storing the generated ANN in database tables, the topologies are saved as objects via Java's serialization mechanism. But the most important expected contribution of such a framework is to be able to reuse those core ANN components for implementing new ANN models.

It is also important to meet the goal of having different ANN instances of different ANN models running at the same time. The object-oriented model also forms the basis for implementing this simulation facility. Besides this, it forms the architecture for parallelizing and distributing the ANN learning and testing. These aspects are discussed separately in Chapters 4 and 5.

Finally, Java, being a portable language and system, solves the problem of splitting the original system into subsystems implemented in various programming paradigms on different platforms. CANN runs on all major computing platforms.

3.2.1 Summary of desired software characteristics and relation to other work

Section 3.3 analyzes and compares related work to the CANN project. Table 3.1 summarizes the considered software characteristics and shows whether the other authors also approach them.

Table 3.1 – Software characteristics and the analyzed related work

Software characteristic	CANN	Freeman	Masters	Vondrák	Rogers
Domain knowledge modeling	✓				
Accessing different data sources	✓				
Data conversion	✓				✓
Different ANN models implemented	✓	✓	✓	✓	✓
ANN as reusable components	✓		✓	✓	✓
ANN building components	✓	✓	✓	✓	✓
Different ANN instances running at the same time	✓				
Runtime addition of new ANN components	✓				
ANN GUI components	✓				
Parallelism	✓				
Distribution	✓				
Platform independent	✓				
Hierarchies of ANN's					

The related work analysis was done comparing the work of other four authors. The first is the simulation software based on structured programming in Pascal proposed by Freeman and Skapura (1992). The other authors develop ANN software based on the object-oriented paradigm. They are: Timothy Masters (1993), Ivo Vondrák (1994) and Joey Rogers (1997). Masters centers his work giving tips on how to implement ANN specific functionality in C++. Vondrák's work is fully concentrated on designing a fully object-oriented solution of an ANN software implementation. His language of choice is Smalltalk. Finally, Rogers also concentrates on designing object-oriented software to implement diverse ANN models in C++.

From Table 3.1 it becomes clear that CANN implements characteristics that are not addressed by the other authors. Their work concentrates on the construction of software artifacts to build ANN models and are not concerned about issues such as the ANN models applicability to the problem at hand, performance, integration to data and GUI control. CANN implementation goes beyond not only by proposing the ANN model software construction but also suggests the implementation of aspects that are important for the simulation and usability of the software infrastructure. The next section explains the CANN software architecture and design that support the desired software characteristics sketched in Table 3.1.

3.2.2 The ANN framework

The ANN framework was designed in order to reflect the necessary building blocks for creating different ANN architectures. The design takes into consideration the flexibility for reusing the core entities of an ANN.

3.2.2.1 Object-oriented modeling of the core entities of neural networks

Neurons and synapses of ANNs mimic their biological counterparts and form the basic building blocks of ANNs. CANN provides two abstract classes, *Neuron* and *Synapse*, whose objects correspond to these entities. Both classes offer properties that are common to different neural network models. The idea is that these classes provide basic behavior independent of the specific neural network model. Subclasses and associated classes add the specific properties according to the particular model.

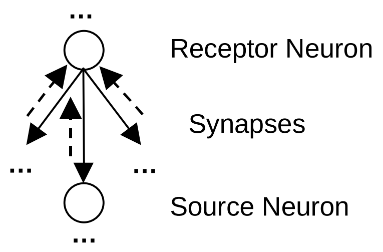


Figure 3.4 - The relationship between Neuron and Synapses objects.

An object of the Neuron class has the activation as its internal state. It provides methods to *calculate* its activation and to manage a collection of *Synapses* objects. A *Synapse* object represents a direct connection between two neurons (here distinguished by the names *Receptor* and *Source*). The receptor neuron manages a list of incoming synapses (represented by the solid arrows in Figure 3.4) and computes its activation from these synapses. A *Synapse* object has exactly one *Neuron* object connected to it, that is the source of the incoming sign. The dashed arrows in Figure 3.4 represent the computational flow (set of incoming signs from all source neurons). The incoming signal from one source neuron is processed by the synapse and forwarded to the receptor neuron on its outgoing side.

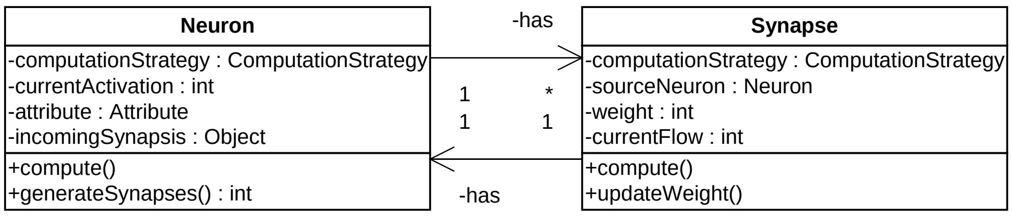


Figure 3.5 - Neuron and Synapses composition.

As the synapse knows its source neurons, different neuron network topologies can be built, *such* as multilayer feedforward or recurrent networks. The Figure 3.5 shows the implementation of the *Neuron* and *Synapse* classes. The process of creating the neural network architecture is controlled by a method called *generateNet()* and belongs to the interface *INetImplementation* that is explained later in Section 3.2.3. Each neural network model is responsible for its topological construction. Different neural models use the *Neuron* and *Synapse* classes as the basic building blocks for the neural network structure and behavior construction. Code 3.1 shows some coding aspects of the *Neuron* class and Code 3.2 the *Synapse* class to make clear how these classes are implemented. The core explanations for them come in the next sections.

Code 3.1 - The *Neuron* class

```
public abstract Neuron class extends Object implements Serializable {
      // stores the resulting activation computation
      int currentActivation;

      // vector of input Synapse objects
      Vector incomingSynapses;

      // strategy for processing input values (see explanation in the
      // text)
      ComputationStrategy compStrategy;

      Neuron() {…}

      Neuron(ComputationStrategy  cs) {…}

      abstract void compute(Neuron upNeuron, Vector parameters);
      abstract int generateSynapses(
         Vector sourceNeurons,
         ComputationStrategy  synapsesCompStrategy);

      float getCurrentActivation() {…}

      Vector getSynapses() {…}

      synchronized void setCurrentActivation(float newActivation) {…}
}
```

Code 3.2 - The *Synapse* class

```
public abstract class Synapse extends Object implements Serializable {
      // the neuron that the synapse receives computation
      Neuron sourceNeuron;

      // strategy for processing input values (see explanation in the
      // text)
      ComputationStrategy compStrategy;

      int weight;       // synaptic weight
      int currentFlow;  // stores the result of the synapse computation

      public Synapse(Neuron addtlSourceNeuron,
            ComputationStrategy  cs) {…}

      public Synapse(ComputationStrategy  cs) {…}

      abstract void compute(Neuron upNeuron, Vector parameters);

      void setWeight(float calcWeight) {…}

      float getWeight() {…}

      float getCurrentFlow() {…}

      void setCurrentFlow(float cf) {…}
}
```

3.2.2.2 Using **Neuron** *and* **Synapse** *classes to create neural network topologies*

In the case of implementing a multilayer feedforward neural network, the neurons for all necessary neuron-layers are created initially. Later, the necessary synapses to connect the neurons at different layers are created and correctly connected to the neuron layers. A list of synapses (called *incomingSynapses*) controls each instance of *Synapse* that connects an output neuron to a hidden neuron. The abstract Neuron class (see Code 3.1) implements this list. When creating instances of the *Synapse* class (see Code 3.2), it is informed in its constructor to which hidden neuron it must be connected. The reference to the hidden neuron is stored in the instance variable *sourceNeuron*. This process is repeated for all network layers. The abstract method *generateSynapses(Vector sourceNeurons, ComputationStrategy synapsesCompStrategy)* in the Neuron class, is responsible for the generation of *Synapse* instances and their appropriate connection to source neurons. As this method is specific to different neurons on different neural models, it is only implemented in its subclasses. The hierarchy of the classes derived from Neuron can be seen in Figure 3.6 below.

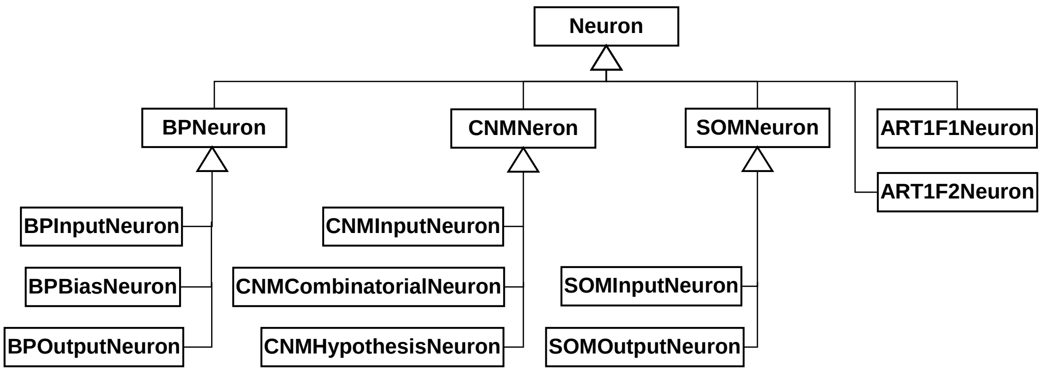


Figure 3.6 – Neuron hierarchy.

The *Neuron* and *Synapse* abstract method *compute()* needs a more detailed explanation, which will be done in the next section. The remaining methods in both classes are simply getter and setter methods to get and set the state of the instance variables. Special attention has to be given to the setter methods of the instance variable *currentActivation* (method *setCurrentActivation()*). As this method can be called by different synapses running in parallel, it was necessary to synchronize it in order to warrant the *currentActivation* value integrity.

The class hierarchy for the *Synapse* class is presented below in Figure 3.7. It is possible to find out in the two hierarchies, *Neuron* and *Synapse*, that the subclasses are created to implement each specific ANN model. The hierarchies are natural for an ANN expert. The subclasses of *Neuron* implement the abstract methods *generateSynapses()* and *compute()*. The *Synapse* subclasses implement the abstract method *compute()*. Both subclasses implement specific methods necessary for the specific ANN model at hand. For example, the CNM specific *Synapse s*ubclasses include punish and reward accumulators and methods to manage it.

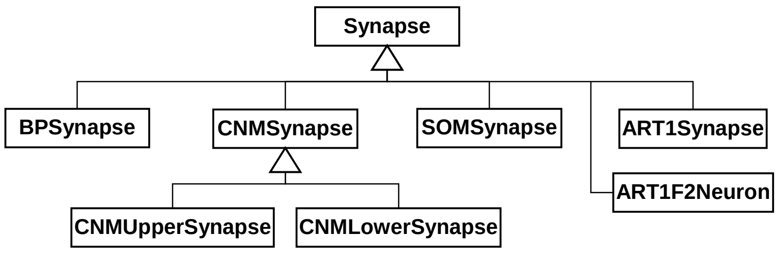


Figure 3.7 – Synapse hierarchy.

The software architecture explained above was successfully used to implement different neural network models involving different neural network architectures. The implementation of recurrent computation in the proposed architecture typically implies synchronization of the computational flow. It is necessary to select the next neuron to do the computation in a learning step. The typical solution is to choose the neuron randomly (Haykin 1994).

An interface called *INetImplementation* is tightly associated with the *Neuron* class (*INetImplementation* interface is explained in Section 3.2.3). Roughly speaking, an INetImplementation object harnesses the ANN in order to make decisions. The *INetImplementation* object represents the inference engine (= neural network model) of the running CANN system. Its interface reflects the needs of the decision making process. For example, a method *getWinner()* computes which output neuron has the maximum activation. Due to the design of *INetImplementation*, CANN supports different inference engines. How to switch between different ANN models is discussed in Section 3.2.3.

3.2.2.3 The neuron and synapse behavior

Specific ANN models imply the need for specific behavior of the *Neuron* and *Synapse* classes. A simple solution would be to create subclasses of *Neuron* and *Synapse*, but this solution would generate a nested hierarchy, because, for each neural model, many subclasses of *Neuron* and *Synapse* would be created to implement each specific behavior. For example, in case of Backpropagation the subclasses would have the names *BPNeuron* and *BPSynapse*. *BPNeuron* would factor out commonalties of the *Backpropagation*-specific classes *BPInputNeuron*, *BPHiddenNeuron* and *BPOutputNeuron*. If the user needed to change anything on the functionality of any of those classes, he would have to subclass it. The part of the neuron and the synapse that changes frequently is its activation calculation. For example, to change the activation function of a Backpropagation from a linear to the sigmoid function, there three *BPNeuron* subclasses would be needed for implementing each function. Quickly, there would be many subclasses of each kind of Backpropagation neuron and synapse. For each different neural model, the same would happen, generating an exploding hierarchy where the functionality of each one is hard-coded in the specific subclass.

To avoid this, the Bridge pattern was used (Gamma et al., 1995). This pattern is equivalent to the Separation Metapattern (Pree, 1997), and therefore has the ability to change neuron and synapse behavior at run-time. Figure 3.9 shows the application of this pattern to the *Neuron* class. Its application is analogous to the *Synapse* class. The necessity of the flexible behavior for Neuron and Synapse was detected during the hot spot analysis. The Figure 3.8 shows the hot spot used to determine the flexible behavior for these classes.

Title: *Making the neurons and synapse behavior flexible*
Description: *The neurons and synapses behavior is given by mathematical functions. The simply change of the mathematical function may change the results of the given ANN model*
Runtime: *Nice to have*
Changed by end user: *Yes, by programming*

Description of 2 instances:

- *A Neuron with a Fuzzy AND behavior may be used by a CNM ANN or any Fuzzy model*
- *A Synapse with a simply multiplication function can be used by both Backpropagation and CNM*

Figure 3.8 – CANN Hot Spot Cards for Neuron and Synapse behavior

Neuron and *Synapse* are implemented as abstract classes. Concrete classes are derived from those abstract classes for each ANN model. For example the already cited *BPNeuron* and *BPSynapse* would be the concrete classes to implement Backpropagation specific aspects to neuron and synapse. If necessary, subclasses of those concrete classes can be created to implement layer specific aspects, like *BPInputNeuron* and *BPOutputNeuron* (Backpropagation hidden neurons are implemented by the *BPNeuron* class). The Backpropagation specific ANN layers are then built using those classes. Those would be the final classes in the hierarchy, no other class should be necessary to be implemented.

The necessary behavior of each neural model is added to these classes by composition through the associated interface *ComputationStrategy* (see Figure 3.9). The different behaviors are implemented in classes that implement *ComputationStrategy* and can be used by different model implementations through the *Neuron* and *Synapse* classes. Examples of behavior-specific classes are functions that are typically implemented by the neurons and synapses to calculate their activation. In the CANN framework some components for this purpose where implemented, such as: *CSEuclidianDistance* for the SOM neuron and *CSPartialDistance* for the SOM synapse; *CSFuzzyAND* and *CSFuzzyOR* for the CNM neurons; *CSMultiplicator* for CNM and Backpropagation synapses; etc.

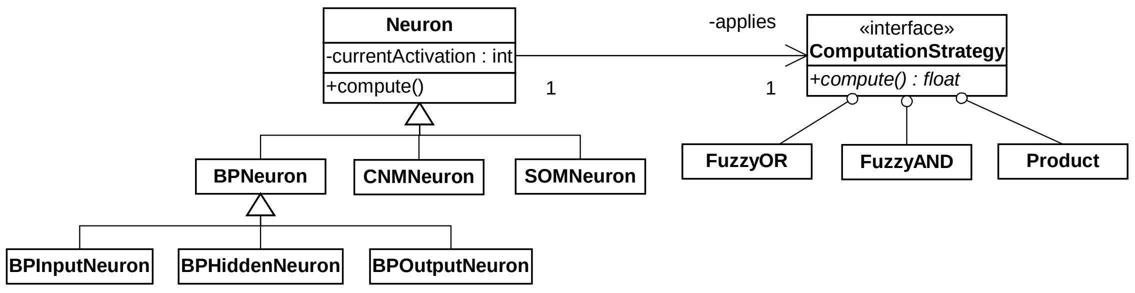


Figure 3.9 - Design of flexible behavior based on the Bridge Pattern.

The *ComputationStrategy* also implements the pattern *Flyweight* (Gamma et al. 1995). The framework has only one instance of each class that implements it. Each instance is shared by a large number of *Neuron* and *Synapse* instances. This keeps the memory footprint significantly smaller and improves the behavior reusability. For example, all CNM combinatorial neuron instances (can be thousands at the same time) would use the same instance of *CSFuzzyAND* computation strategy.

3.2.2.4 Support of different neural network models through the Separation pattern

One important aspect is to try to separate the logic of manipulating the domain and the data cases for learning and testing from the core ANN implementation aspects. In this sense, the creation of the class *NetManager* and interface *INetImplementation* was elaborated. The abstract class *NetManager* shall hold the aspects that belongs to the ANN process outside the ANN architecture boundaries such as: manipulating domain knowledge, fetching test and learn data, managing learn parameters, and managing the whole flow of learning and testing processes based on user defined parameters. The processes that are general to any ANN model are implemented directly in this class such as: GUI control, and learn and test cases manipulations. Specific processes shall be implemented inside its subclasses that are defined for each implemented ANN.

The classes that implement the *INetImplementation* interface hold the specific ANN model architectural implementation. They implement the construction of the ANN architecture by combining the *Neuron* and *Synapse* classes and implement each ANN inherent algorithms.

Figure 3.10 exemplifies how three ANN models can be incorporated in CANN by implementing the interface *INetImplementation*: Backpropagation (BPImplementation), SOM (SOMImplementation) and CNM (CNMImplementation).

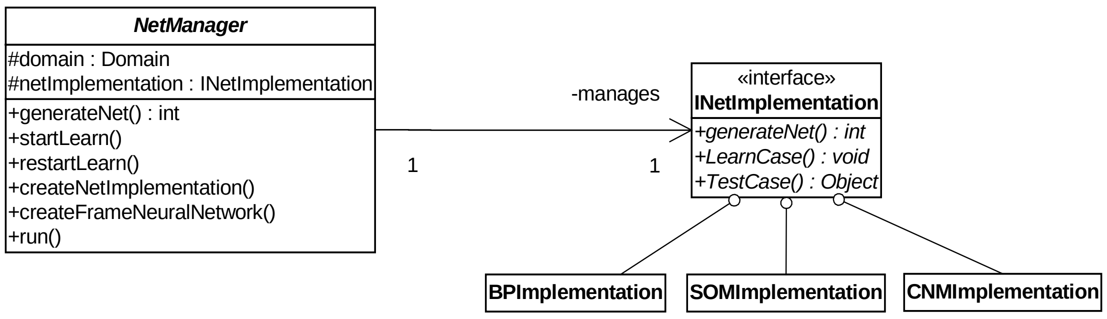


Figure 3.10 - ANN models that implement *INetImplementation*.

A specific ANN model is defined by implementing the interface *INetImplementation* and its corresponding hook methods such as *generateNet() and learnCase()*. The complete interface is shown in Code 3.3 below.

The first three methods of Code 3.3 are used for the distribution solution explained in Chapter 5. The *generateNet()* method shall implement the generation of the ANN architecture. If the network wasn't yet generated, the *getNetSize()* method returns the size the ANN will have in number of neurons and synapses if it is generated for a given domain problem at hand. If the ANN was already generated this method returns the size of the actual network. The *learnCase()* method is called when one case was already fed by the implemented domain, meaning that the network must perform its learning epoch for that case. More explanations on the relation of domain and the ANN framework are given in Section 3.2.3.

The *testCase()* method also works the same way as the *learnCase()* method: when a testing case is ready to be analyzed by the ANN, this method is called. It returns a Vector that can contain any object, in general an array of numeric results containing the outputs of the output neurons and, if possible, an array of strings containing the explanation about the results. The methods *setLearningParameters()* and *getLearningParameters()* are used to feed and get the appropriate ANN learning parameters that in general are defined by the user at runtime. The method *setParallelExecution()* is used to request to run in parallel, if possible. The method *getStopLearning()* is used to verify if the ANN has already finished its learning process.

Code 3.3 – The *INetImplementation* interface

```
public interface INetImplementation extends java.io.Serializable {

  public void setProxies(IDomain domain) throws java.rmi.RemoteException;
  public void atLocation();
  public boolean restoreObjectReferences();

  // return any number different than zero to indicate that the generation
  // succeed or zero that it failed
  public int generateNet(IDomain domain) throws java.rmi.RemoteException;
  public int getNetSize(IDomain domain);

  public void learnCase();

  // return a Vector with the results and explanation if possible
  public Vector testCase();

  public void setLearningParameters(Vector parameters);
  public Vector getLearningParameters();

  public void setParallelExecution(boolean pE, int threads);

  public boolean getStopLearning();
}
```

An important design issue is that a developer who reuses such CANN classes does not have to worry about which specific subclasses of *Synapse* and *Neuron* are associated with a particular ANN model. In order to solve this, the Factory pattern (Gamma et al. 1995) was applied. A concrete class that implements *INetImplementation*, such as *CNMImplementation*, already takes care of the correct instantiation of *Neuron* and *Synapse* subclasses (see below).

ANN-Adaptation of CANN

A sample adaptation of CANN exemplifies the necessary steps to adjust the framework components to a specific neural network model (see Figure 3.11):

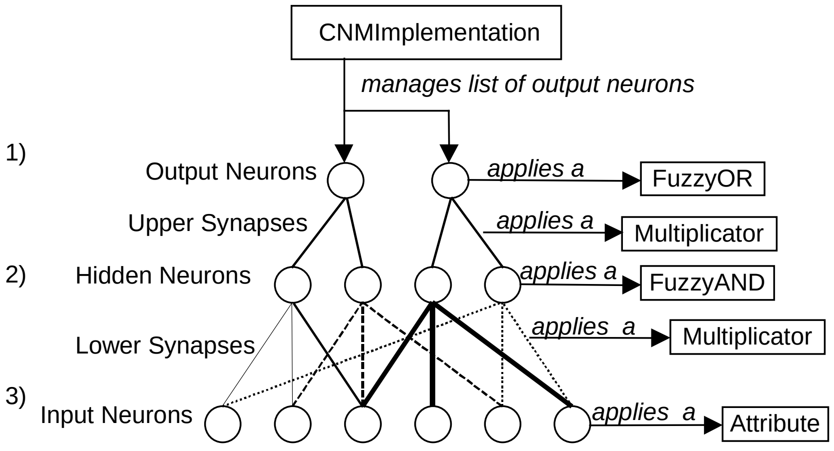


Figure 3.11 - Building CNM architecture.

1. The *CNMImplementation* object is responsible for the creation of *OutputNeuron* objects (called Hypothesis neurons in the CNM definition, Machado 1990), with the *CSFuzzyOR* behavior and for the creation of *InputNeurons*, whose behavior is explained in Section 3.2.6.

2. An *OutputNeuron* instance then creates Synapse objects, which automatically create *HiddenNeuron* instances (called combinatorial neurons in the CNM definition, Machado 1990), with the *CSFuzzyAND* behavior. As the CNM model has only one hidden layer, the neurons of the hidden layer are then directly connected to the input layer.

3. The connections between the *HiddenNeuron* instances and the *InputNeuron* instances are established in an analogous way. The *Synapse* instances of a CNM model have behavior similar to the Backpropagation model using the same *CSMultiplicator* computation strategy that simply does the multiplication of the input activation with the synaptic weight and returns the result.

For the neural network generation process, the classes that implement *INetImplementation* (in this case, the *CNMImplementation*) rely on the problem-specific domain knowledge, whose representation is discussed in Section 3.2.4. The basic idea behind the CNM inference machine is that numerous combinations (the small networks that form the CNM hidden layer) all test a certain case for which a decision is necessary. Adding the activation from all combinations amounts to the *OutputNeuron* activation. The *OutputNeuron* with the maximum accumulated activation is the winner (*CSFuzzyOR* behavior). The CNM object also provides an explanation by calculating those input neurons that most strongly influenced the decision. Machado and Rocha (1989 and 1990) discuss the CNM-specific algorithm.

Adding behavior to an ANN architecture

The sequence diagram in Figure 3.12 shows the sequence of computation that happens when the ANN, either for learning or for testing, evaluates a case. The class that implements *INetImplementation* knows the instances of output neurons that are implemented in the structure already created. The result value of a case computation is implemented by the output neurons' *compute()* method and can be retrieved by the *getCurrentFlow()* method (see Code 3.1 – *Neuron* class). The *INetImplementation* class requests computation from the output neurons by calling the *compute()* methods of all existing output neurons. When the output neuron is requested to do its computation, it first requires its list of incoming synapses to do the same. The synapses also have a source neuron that is requested to do its own computation (see Code 3.2 – *Synapse* class).

The source neuron is a hidden neuron in the architecture and its *compute()* method implementation also requests the computation of the connected synapses. In this way, the request of computation flows from the output neurons to the input neurons. The input neuron instance's behavior is simply to take the activation from outside to be able to start the ANN data processing. This activation comes from the class Attribute, explained in Section 3.2.4. In short, the instances of the Attribute class prepare the activation values from the data read in the data sources. These prepared data (activation) are transferred to the input neuron instances on demand.

When the computation flow goes back from the input neurons to the output neurons, each synapse and neuron object is then able to do the necessary calculation it is supposed to do and returns it to the object that requested it (other neurons and synapses). The instances of *ComputationStrategy* class do this calculation. Finally, the *compute()* method of each output neuron gets the computational results of all connected synapses and does its appropriate computation. Then the resulting values can be consulted through the output neurons *getCurrentFlow()* method. The *INetImplementation* implemented class is able to evaluate these values and to make a decision.

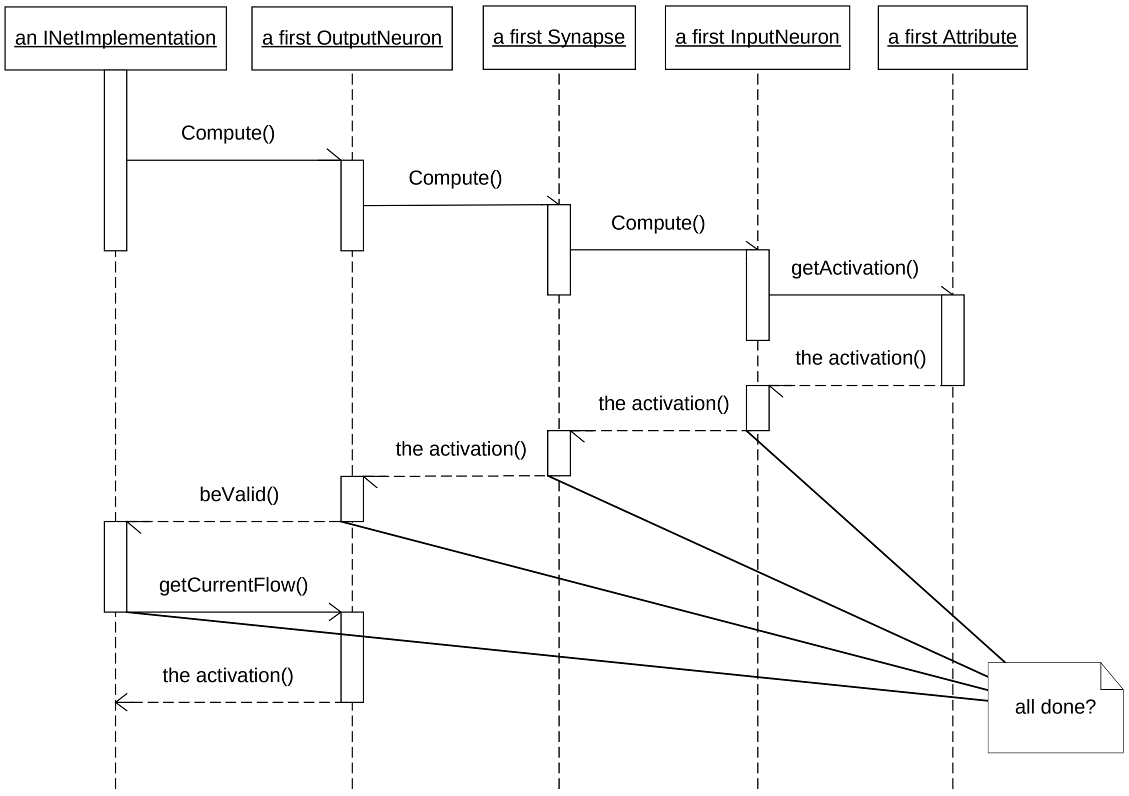


Figure 3.12 – Sequence diagram for a case computation.

The computational flow explained above is a parallel process internal to the neural network architecture. Instances of *Synapse* and *Neuron* in the same layer can be completely independent processes. Depending on the neural model, different synchronization must be implemented in order get the correct results and to have optimal performance. The parallel implementation strategy of the computational flow is specific to each model and a more detailed discussion is done in Chapter 4.

To complete the appropriate behavior of the implemented neural networks, it is necessary to have them related to the knowledge representation of the problem domain. The next section explains how the domain model influences and interacts with the neural network architecture.

3.2.3 The Simulation framework

CANN should be flexible regarding its underlying ANN model. The choice of the most appropriate ANN model depends on the particular decision problem at hand, and usually it is necessary to try some different ANN models to see which one performs better. Thus, the design should allow the trial of different ANN models. The hot spot card for this flexibility is shown on Figure 3.13.

Title: *Adding new ANN components at the simulations environment at runtime*
Description: *The CANN simulator should be able to allow the addition of new ANN models during the simulation of others in order to cope with longtime simulations. Also more then one ANN shall be able to run at the same time.*
Runtime: *Yes*
Changed by end user: *Yes*

Description of 2 instances:

- *A Backpropagation ANN is running a simulation for more than 24 ours and it is possible to add a RBF model to run at the same time. This can be useful to evaluate its performance to the same problem domain in the same simulation environment.*
- *A CNM model is running for evaluating new entries in a production environment and a new already learned CNM model is added to substitute the first one.*

Figure 3.13 – CANN Hot Spot Cards for different ANN models

CANN should be able to manage various ANN models trying the solution for a specific problem in a concurrent way: several neural models could run at the same time trying a solution to the problem. To handle this idea the *Project* and *NetManager* classes were created. The *NetManager* class is responsible for controlling, at run time, an instance of a neural model and the *Project* class is responsible for managing a set of *NetManager* instances. A program can run as many instances of the *NetManager* class as necessary to solve a problem. Instances of the same or different ANN models shall be possible to be created at runtime.

As the learning of a neural model can take days, it would be interesting to be able to add different ANN models at run-time. Thus, it would be possible to start different learning trials during the learning or testing process of other neural models, without stopping the processes already started. The added new models could be completely different from the others already running. This means the possibility of having new models that have added or deleted neurons and synapses on the ANN structure and/or changed its behavior by changing learning strategies and tuning learning parameters. To have these kinds of simulation characteristics, it is necessary to have quite a flexible architecture design. This design was obtained through the hot-spot-driven design methodology (Pree 1995).

Code 3.4 shows part of the implementation of the *NetManager* class where it is possible to see the instance variable *netImplementation* that holds the instance of a concrete class that implements the *INetImplementation* interface. It is also possible to see the instance variable *domain* that stores an instance of the Domain class. The Domain modeling affects directly the ANN architecture as can be seen in Section 3.2.4.

NetManager implements methods such as *initFrame()* and *createFrameNeuralNetwork()* that are used for controlling the associated GUI of an ANN instance. Methods like *generateNet()*, *startLearn()* and *restartLearn()* are directly related to the management of the controlled ANN instance implementation. Some methods are abstract such as *createNetImplementation()* and *run()* and shall be implemented by the *NetManager* subclasses. The subclasses of *NetManager* can be seen in Figure 3.14 below and implement functionality specific for the implemented ANN model.

Code 3.4 – The *NetManager* class implementation

```
public abstract class NetManager extends Object implements Serializable,
Runnable {

  INetImplementation netImplementation;
  Domain domain;
  // …

  public void initFrame(DialogSimulate relatedDialog) {…}
  public void createFrameNeuralNetwork() {…}

  int generateNet() {…}
  void startLearn() {…}
  void restartLearn() {…}

  abstract public void createNetImplementation();
  abstract public void run();

}
```

To permit the addition of new neural models at run-time, it is necessary to abstractly couple the *Project* with the class *NetManager*, that is, to rely on the Separation pattern. The Project has a list of instances of the *NetManager* class that are responsible for different ANN instantiation and execution. The relation between the *Project* class and the *NetManager* class can be seen in Figure 3.14 below.

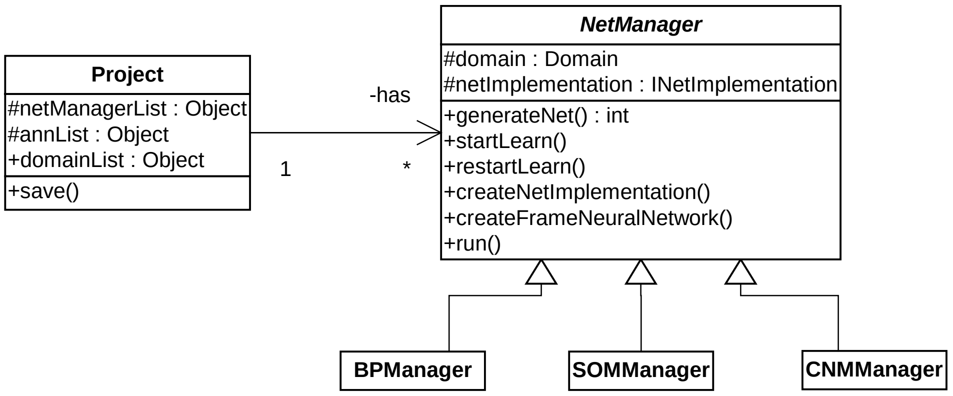


Figure 3.14 - *NetManager* abstractly coupled to *Project*

Code 3.5 below shows part of the class *Project* implementation. The instance variable *netManagerList* stores a vector of instances of the *NetManager* class. The instance variable *annList* has the list of possible ANN models that can be instantiated by the *Project* class and stored on the *netManagersList*. For example, the *annList* may store the classes: *CANNP.NeuralNetwork.CNMManager* and *CANNP.NeuralNetwork.BPManager* meaning that those two ANN models can be instantiated in that project. Note that the whole name is stored, keeping the packages path. The user can add ANN models in runtime and also create new instances in runtime.

Code 3.5 – The *Project* class

```
public class Project extends Object implements Serializable,
ProjectModifiedListener, IRemote {

  Vector netManagersList;
  Vector annList;
  public Vector domainList;
  …
}
```

The CANN shall also provide a way that the user can describe a problem domain in different ways or even have many problem domains simultaneously modeled to be analyzed by the different instances of the ANN models running into a project. Consequently, a *Project* may also manage a list of problem domain descriptions as can be seen in Figure 3.15 below. The Domain class is explained in Section 3.2.4. Code 3.5 above shows the instance variable *domainList* that is responsible for keeping a list of instances of the *Domain* class. How the different ANN instances relate to the Domain instances will be shown in Section 3.2.4 below.

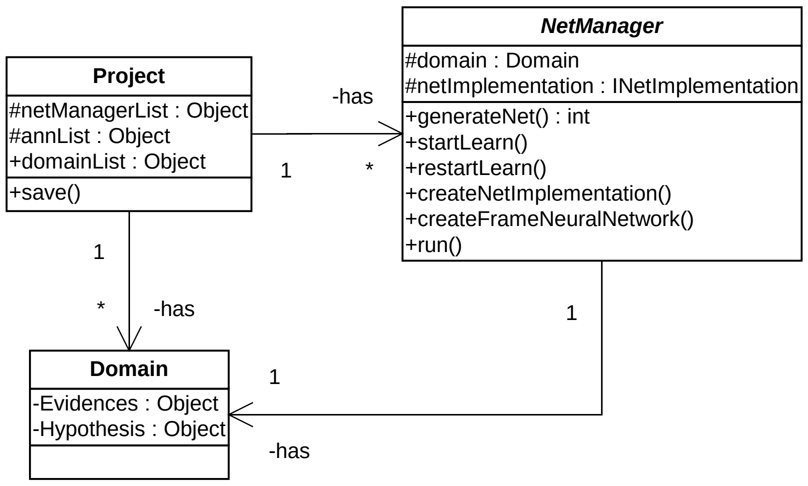


Figure 3.15 - *Project* coupling *Domain* instances

An important characteristic of the whole CANN architecture is that all classes that compound the frameworks are *Serializable* (see Code 3.5). Being the *Project* class *Serializable*, the user can store its characteristics, It means to store the instances of the modeled domain problems and the instances of the associated ANN as well.

3.2.4 The Domain representation framework

As the principal application domain of CANN is classification problems, the chosen object-oriented design of this system aspect reflects common properties of classification problems. On the one hand, the so-called evidences form the input data. Experts use evidences to analyze the problem in order to come up with decisions. Evidences in the case of the customer classification problem would be the age of a customer, his/her home address, etc. One or more Attribute objects describe the value of each Evidence object. For example, in the case of the home address, several strings form the address evidence. The age of a customer might be defined as a fuzzy set (Kosko 1992, da Rosa 1997) of values: child, adolescent, adult, and senior (see Figure 3.16).

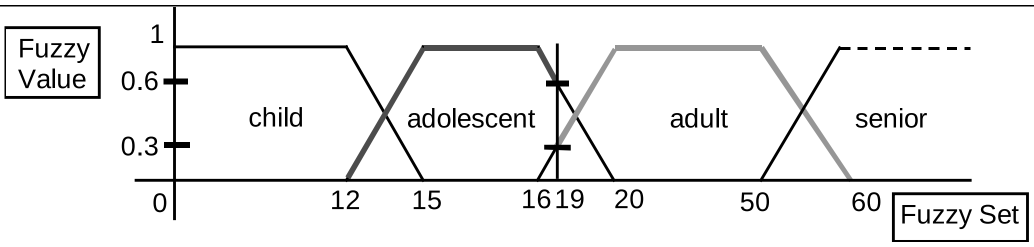


Figure 3.16 - Fuzzy set example.

On the other hand, the hypotheses (classification categories) constitute a further core entity of classification problems. In CANN an instance of class Domain represents the problem by managing the corresponding Evidence and Hypothesis objects. Even based on classification problems and focused on neural networks learning algorithms, the design presented here can also be extended to support general domain representation for symbolic learning strategies. Edward Blurock (1998) also works intensively on the domain representation for machine learning algorithms. Although being completely independent works, both lead to quite similar designs. Figure 3.17 shows the relationship among the classes involved in representing a particular domain.

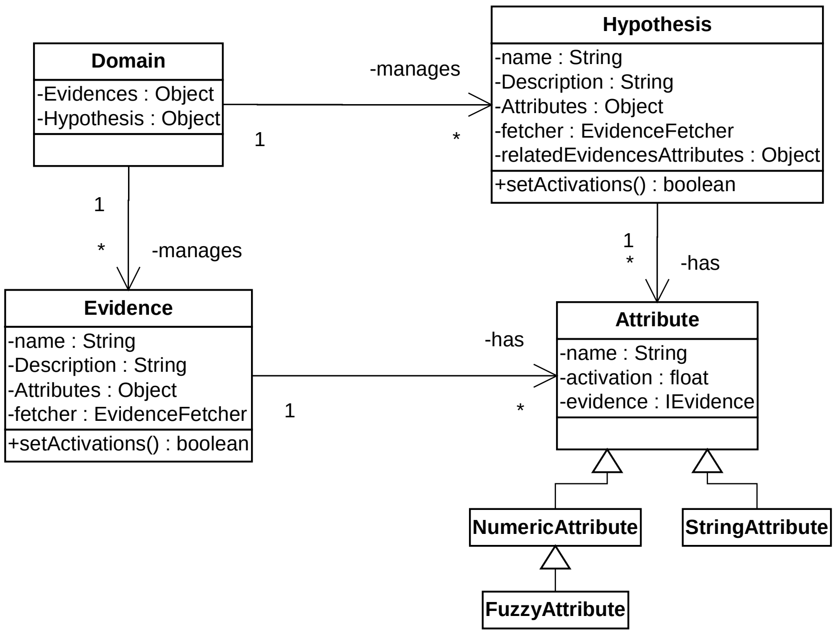


Figure 3.17 - Domain representation.

The training and testing of ANN's are the principal features of CANN. For both tasks, data must be provided. For example, for training an ANN to classify customers, data might come from an ASCII file. One line of that file represents one customer, i.e. the customer's evidences and the correct classification. After training the ANN, customer data should be tested. To do this, CANN gets the evidences of a customer as input data and must classify the customer. The data source might, in this case, be a relational database management system. It should be clear from this scenario that CANN has to provide a flexible data conversion subsystem. Data conversion must be flexible at run time, as the user may wish to change the data source anytime during learning or testing. Thus, the Separation pattern is the appropriate construction principle underlying this framework.

3.2.5 The Converter framework

Two abstract classes constitute the framework for processing problem-specific data, class Fetcher and class EvidenceFetcher. Class Fetcher is abstractly coupled with the class Domain (see Figure 3.18). A Fetcher object is responsible for the preparation/searching operations associated with a data source. If the data source is a plain ASCII file, the specific fetcher opens and closes the file. This includes some basic error handling.

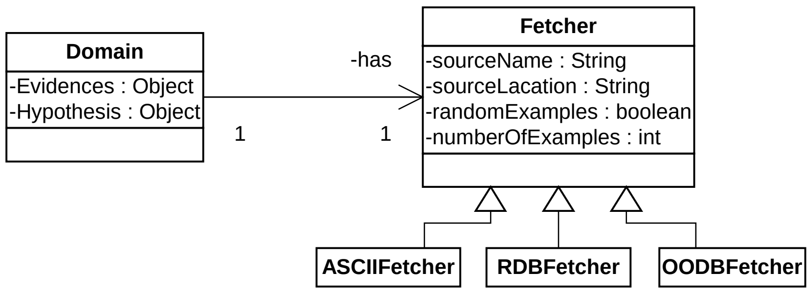


Figure 3.18 - Dealing with different data sources.

The Evidence class and the Hypothesis class are abstractly coupled with the EvidenceFetcher class (see Figure 3.19). Specific subclasses of EvidenceFetcher know how to access the data for the particular evidence. For example, an EvidenceFetcher for reading data from an ASCII file stores the position (from column, to column) of the evidence data within one line of the ASCII file. An EvidenceFetcher for reading data from a relational database would know how to access the data by means of SQL statements. Figure 3.19 shows the design of these classes, picking out only class Evidence. The Hypothesis class has an analogous relationship with the EvidenceFetcher class.

Note that the Attribute objects prepare the data from external sources so that they can be directly fed to the input neurons of the ANN (see Figure 3.11). This works in the following way: each Evidence instance fetches its value from the data source, and this value is applied automatically to all attributes of the evidence. Each attribute applies the conversion function that is inherent to the specific Attribute class. For example, the StringAttribute conversion function receives the string from the database and compares it to a given string modeled by the expert, returning 1 or 0 based on whether the strings match. This numeric value is stored by the attribute object and will be applied in the ANN input by request. The ANN input nodes have a direct relationship with the attributes of the evidence (see Figure 3.11). When the learning or testing is performed, each input node requests from its relative attribute the values previously fetched and converted. The attribute simply returns the converted value.

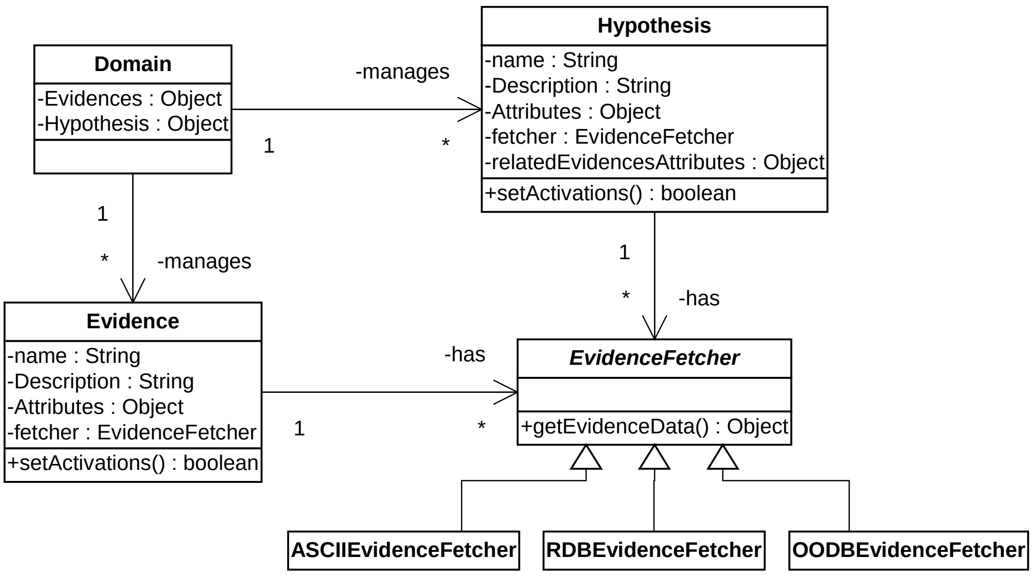


Figure 3.19 - Data conversion at the evidence level.

Visual/interactive tools support the definition of the specific instances of EvidenceFetcher and Fetcher subclasses. For example, in the case of fetching from ASCII files, the end-user of CANN who does the domain modeling, simply specifies the file name for the ASCIIFetcher object and, for the ASCIIEvidenceFetcher objects, specifies the column positions in a dialog box. Such visual tools can be seen on chapter 7 (Figure 7.8).

3.2.6 Describing problem domains using the Domain and converter frameworks

In order to better understand how to use the classes that form the Domain and the converter frameworks let's consider an example: building the necessary evidences and hypothesis for a given domain problem for two different ANN models. The chosen domain is the XOR problem. Both ANN models CNM and Backpropagation are able to solve this problem.

The Boolean table shown in Figure 3.20 gives the XOR problem for two variables. This is the data that will be fed to the ANN as learning cases. The Boolean table is created in a text file that will be read by the CANN simulator as an ASCII file so that the Domain shall have an instance of the *ASCIIFetcher* converter, and the converters for the implemented attributes shall be of the type *ASCIIEvidenceFetcher*. The fetcher for each evidence will indicate what position in the file record (line) is the data that shall be fed into the input neuron associated to it.

```
0 0 0
1 0 1
1 1 0
0 1 1
```

Figure 3.20 - XOR ASCII file for Backpropagation and CNM learning

3.2.6.1 Backpropagation domain modeling for the XOR problem

The Backpropagation ANN is modeled with two input neurons and one output neuron to solve the given XOR problem, so that it is necessary to model the evidences and hypothesis in order to represent this ANN structure. One Hypothesis instance and two Evidence instances are created as described in Figure 3.21. The Backpropagation *netGenerate()* method will be able to interpret this domain and create one output neuron associated to the hypothesis and two input neurons, each one associated to each evidence.

***Hypothesis* Output**
ASCIIEvidenceFetcher from = 4, to = 4
NumericAttribute Output
Related Evidences = Input1→I1; Input2→I2

***Evidence* Input 1**
ASCIIEvidenceFetcher from = 0, to = 0
NumericAttribute I1

***Evidence* Input 2**
ASCIIEvidenceFetcher from = 2, to = 2
NumericAttribute I2

Figure 3.21 - Modeling XOR Domain for Backpropagation

One hypothesis called "Output" is created with fetcher getting the data on the fourth column of the learning file. It has a numeric attribute that converts the read data from the file into its numeric value. The associated output neuron will request this value under demand. The related evidences indicate which ones must be considered to build the ANN architecture to the given hypothesis. In case of the Backpropagation, both evidences are considered for the construction of the ANN that leads to the output neuron.

The two created numeric attributes of the two evidences fetch values on columns 0 and 2 of the ASCII file. The neurons of the Backpropagation hidden layer are created based on the configuration value entered by the user at run-time. They will be appropriately connected to the created input and output neurons.

3.2.6.2 CNM domain modeling for the XOR problem

The CNM is modeled with four input neurons and two output neurons to solve the given XOR problem. Two Hypothesis instances and two Evidence instances are created as described in Figure 3.22. The possible output values are modeled on the CNM as symbolic values "Yes" or "No", meaning two possible hypotheses for the problem. Therefore, two *Hypothesis* instances are created. The fetchers for both Hypotheses are once again configured to get the values on the fourth column of the ASCII file. If the value is zero, the "No" hypothesis is considered the winner, if the value is one, the "Yes" hypothesis wins. Those values are fetched by the CNM output neurons whenever necessary during the learning process.

The attributes of the created evidences are considered for the modeling of the input neurons. CNM considers one input neuron for each modeled attribute. For instance on the problem domain at hand each input can assume the two Boolean values one or zero. Those values are also considered by the CNM as the symbolic values "Yes" or "No". One attribute must be built for each evidence in order to model the two possible values. One CNM input neuron is created for each attribute. Therefore the CNM will have four input neurons and two output neurons to the XOR problem.

The CNM *netGenerate()* method will be able to interpret this domain and create the two output neurons associated to the hypothesis string attributes and four input neurons associated to each string attribute of the modeled evidences.

***Hypothesis* Yes**
ASCIIEvidenceFetcher from = 4, to = 4
StringAttribute **Yes**; string = "1"
Related Evidences = Input1→Yes; Input1→No; Input2→Yes; Input2→No

***Hypothesis* No**
ASCIIEvidenceFetcher from = 4, to = 4
StringAttribute **No**; string = "0"
Related Evidences = Input1→Yes; Input1→No; Input2→Yes; Input2→No

***Evidence* Input 1**
ASCIIEvidenceFetcher from = 0, to = 0
StringAttribute **Yes**; string = "1"; morbidity = 0.9
StringAttribute **No**; string = "0"; morbidity = 0.9

***Evidence* Input 2**
ASCIIEvidenceFetcher from = 2, to = 2
StringAttribute **Yes**; string = "1"; morbidity = 0.8
StringAttribute **No**; string = "0"; morbidity = 0.8

Figure 3.22 - Modelling XOR Domain for CNM

3.2.7 Coupling Domain, ANN and simulation frameworks together

As it was already described, the class *Project* contains a list of *Domain* instances. Each such *Domain* instance is implemented in the way explained in Section 3.2.6 above. Consequently a *Project* may have many domain problem models, that means, modeling of the same problem domain in different ways and/or modeling different domain problems. The *Project* also contains a list of *NetManager* instances that represent the ANN models that shall be simulated in order to solve the modeled domain problems. Each instance of a NetManager can handle only one domain problem.

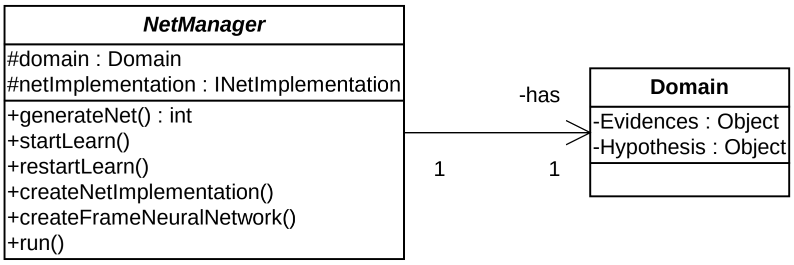


Figure 3.23 - *NetManager* is associated to a *Domain* instance

Figure 3.23 shows the relation between *NetManager* and *Domain* classes. The *NetManager* contains a *Domain* instance cloned from the list of modeled domains in the *Project*. The Domain instance is cloned because more than one ANN instance may use the same Domain model at the same time. The appropriate solution for this problem would probably be implementing transaction synchronization at the level of the simulation framework. Such a solution is much more complex so that simply cloning the Domain warrants that each ANN accesses the data via a separate control structure.

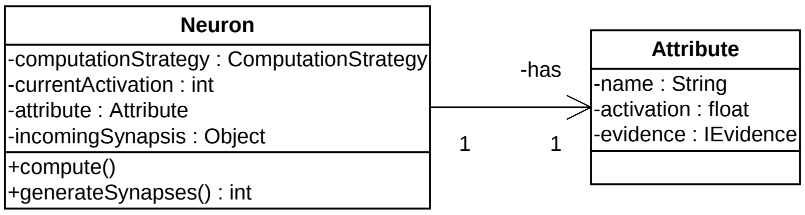


Figure 3.24 - *Neuron* fetches activation from its associated *Attribute* instance

The specific *NetManager* class implements how the ANN treats the modeled domain to build the appropriate ANN architecture. Each ANN model presumes modeling the domain in a determined way in order to properly perform the learning. Besides this, the constructed ANN structure will have relationships to the Domain in order to be able to fetch the learning and testing data whenever necessary. Typically the output neurons will have relations to the hypothesis attributes objects and the input neurons to the evidence attributes objects. Figure 3.24 shows the relations among those classes.

3.2.8 The ANN GUI framework

A complete simulation environment was built to facilitate the simulation of ANN. In this environment, a project can be created to manipulate different instances of ANN's and domain problems. A complete explanation for the CANN simulation environment characteristics can be found in Chapter 7.

The whole simulation GUI environment was also built based on the design principles that lead the development of the frameworks so far. Special dialogs for manipulating the creation of the domain data such as the hypothesis and evidences, and the runtime adding and creation of different ANN instances. Besides those, a small GUI framework was built for handling the ANN main functionalities: The ANN GUI framework.

The ANN GUI framework implements a small set of Java GUI components that group general ANN activities such as learning and testing data. The built GUI components are intended to be general for any type of ANN. Figure 3.25 shows the main GUI elements created for the ANN manipulation. It shows a class called *FrameNeuralNetwork* that implements a Java *Frame* class. *FrameNeuralNetwork* contains an instance of a *NetManager*, so that it is able to control the execution of any ANN component implemented on the CANN framework.

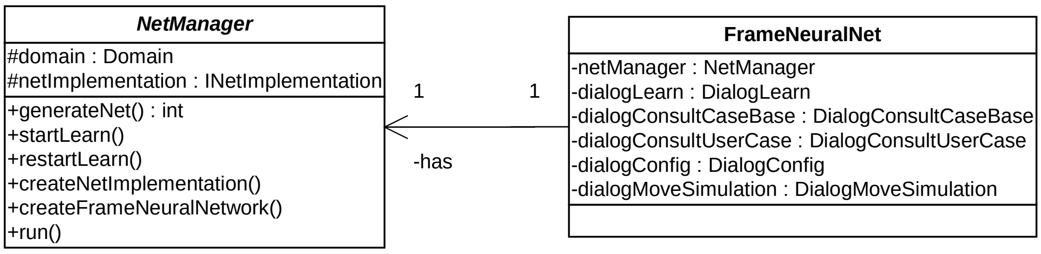

NetManager
#domain : Domain
#netImplementation : INetImplementation
+generateNet() : int
+startLearn()
+restartLearn()
+createNetImplementation()
+createFrameNeuralNetwork()
+run()
1
1
-has
FrameNeuralNet
-netManager : NetManager
-dialogLearn : DialogLearn
-dialogConsultCaseBase : DialogConsultCaseBase
-dialogConsultUserCase : DialogConsultUserCase
-dialogConfig : DialogConfig
-dialogMoveSimulation : DialogMoveSimulation

Figure 3.25 – GUI framework

FrameNeuralNetwork has a set of dialogs to manipulate an ANN instance execution: *DialogLearn* for controlling the execution of the ANN learning process; *DialogConsultCaseBase* to control the execution of the ANN testing based on a set of cases; *DialogConsultUserCase* allows the testing of an ANN given one case built at runtime by the user through this GUI; *DialogMoveSimulation* allows the user to make use of the mobility characteristic of the ANN instances and move it to a remote computer, this dialog is not obligatory; and *DialogConfig* to control the ANN configuration parameters. This dialog is an abstract class and its concrete classes are specifically implemented for each ANN model once the configured learning parameters are different for each ANN model.

Figure 3.26 shows the *FrameNeuralNetwork* frame instance. In this case, it is running a Backpropagation instance called "BP 1". The menu "Neural Net" provides the possibility to configure the inner ANN component, reset (meaning creating a new network structure), save the neural network instance and close the frame. The menu "Simulate" gives the alternatives: move the ANN component to a remote machine using the mobility characteristics explained in Chapter 5; perform the network learning; and perform the network testing. At the bottom of the dialog there is a status bar that in this example is explaining that the ANN instance is created and running at the local host at port 7000 where the Voyager ORB server is running (see Chapter 5 for details on the Voyager implementation).

Adicionar os menus abertos e referenciar como a e b abaixo.

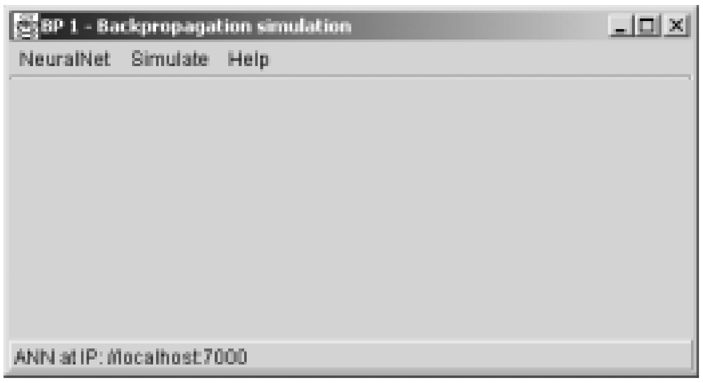
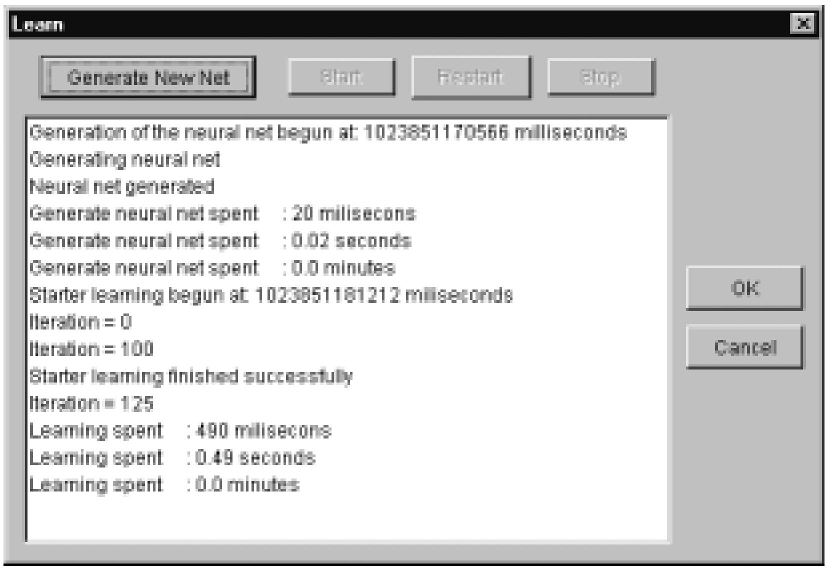

BP 1 - Backpropagation simulation
NeuralNet
Simulate
Help
ANN at IP: //localhost:7000

Figure 3.26 - *FrameNeuralNetwork* containing a Backpropagation ANN instance

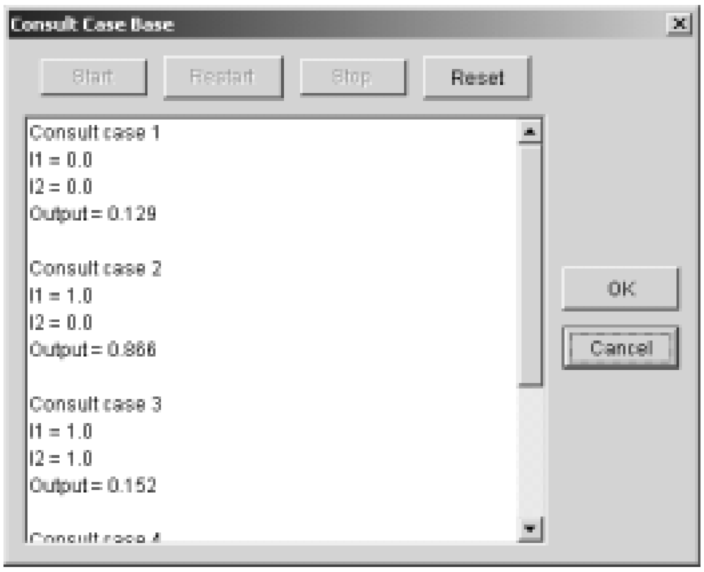
Figure 3.27 shows the *DialogLearn* class. In this example the Backpropagation network was generated for the given XOR domain problem in 20 milliseconds. The learning was performed for the XOR problem as well and succeeded at 144 epochs taking 1182 milliseconds. In this dialog the user can generate new nets, start, stop and restart the learning. The learning is performed based on an ASCII file that was already defined by the implemented *Fetcher* at the ANN associated *Domain* instance.

Figure 3.28 shows a *DialogConsultCaseBase* instance with a case base formed by the XOR problem cases being tested by the Backpropagation ANN. The test is performed based on an ASCII file that was already defined by the implemented *Fetcher* at the *Domain* instance as well as the learning. The cases show the input values for I1 and I2 (Input 1 and Input 2 evidences and I1 and I2 attributes of the evidences), meaning zero for false and one for true. The ANN output result is a numerical value between 0 (false) and 1 (true). The Figure shows the network performing properly for the first three cases. In this dialog it is possible to start, stop, restart and reset the testing of the case base.



```
Generation of the neural net begun at: 1023851170566 milliseconds
Generating neural net
Neural net generated
Generate neural net spent    : 20 milisecons
Generate neural net spent    : 0.02 seconds
Generate neural net spent    : 0.0 minutes
Starter learning begun at: 1023851181212 milliseconds
Iteration = 0
Iteration = 100
Starter learning finished successfully
Iteration = 125
Learning spent    : 490 milisecons
Learning spent    : 0.49 seconds
Learning spent    : 0.0 minutes
```

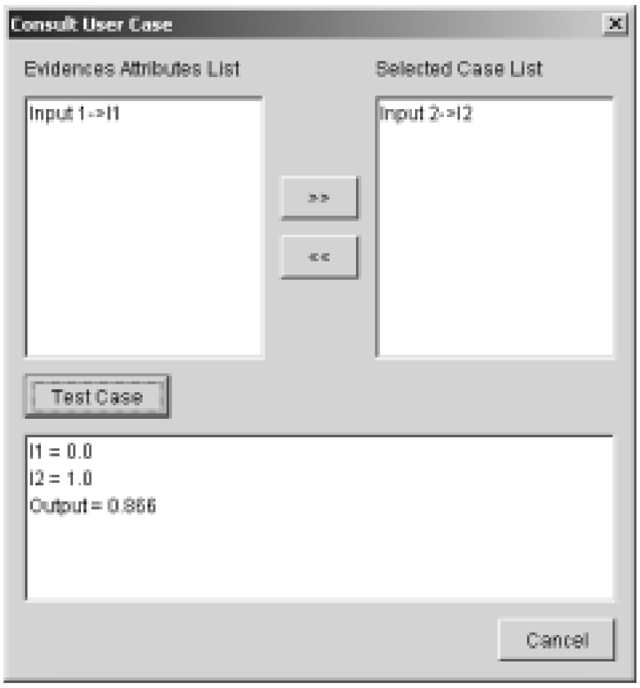
Figure 3.27 - *DialogLearn* performing the learning of the XOR problem



```
Consult case 1
I1 = 0.0
I2 = 0.0
Output = 0.129

Consult case 2
I1 = 1.0
I2 = 0.0
Output = 0.866

Consult case 3
I1 = 1.0
I2 = 1.0
Output = 0.152

Consult case 4
```

Figure 3.28 - *DialogConsultCaseBase* performing the testing of the XOR problem

Figure 3.29 shows the *DialogConsultUserCase* class where the user has the chance of building a case to be presented for the ANN at runtime. In that case, the user did not select the Input 1 meaning that this input evidence must have value zero (false) as activation. The selected Input 2 will have activation 1 (true). The evaluation of this case to the XOR learned Backpropagation network gave the output result of 0.866 that is a value next to the value one, meaning (true).



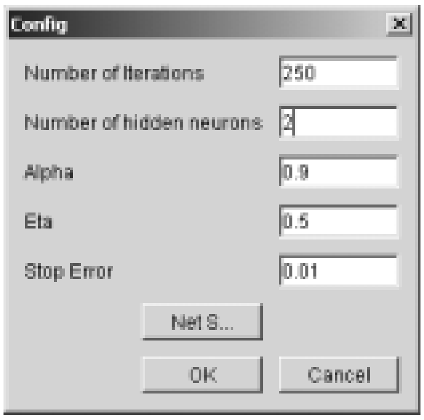
Input 1->I1

Input 2->I2

```
I1 = 0.0
I2 = 1.0
Output = 0.866
```

Figure 3.29 - *DialogConsultUserCase* performing the testing of a user case

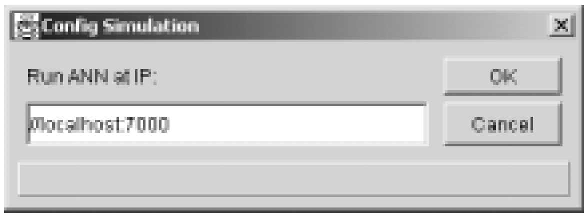
Figure 3.30 shows a concrete sub-class of *DialogConfig*, the *BPDialogConfig*. The *BPDialogConfig* implements a Java *Dialog* where the user can choose the appropriate learning parameters for the Backpropagation ANN instance. As already explained before, this dialog must be implemented for each different ANN model.



Number of Iterations	250
Number of hidden neurons	2
Alpha	0.9
Eta	0.5
Stop Error	0.01

Figure 3.30 – *BPDialogConfig* class for the Backpropagation configuration

Figure 3.31 shows the *DialogMoveSimulation* class that implements a Java *Dialog* where the user can specify a host IP where to move the ANN instance.



localhost:7000

Figure 3.31 – Moving the ANN component to run in a remote machine

The GUI components shown here are extensively used on the CANN simulation environment. They were built in order to form a basic GUI framework where the ANN programmer can use the GUI classes as they are and concentrate on the implementation of the ANN component. New ANN models can be easily added to this GUI framework in order to test its inner implementation.

3.2.9 Packaging the frameworks in reusable components

"Components are units of deployment" (Clemens Szypersky 1997). Szypersky phrase focuses on the main reason for building components, the deployment of software. The core idea is that a component is any piece of software that can be delivered as a single unit and reused in systems other than the one it was planned for. In that way a component may be a class, a procedure, a module, etc. The software reusability capacity is directly related to the quality of the design and implementation of the software pieces.

In this work the deliverable pieces of software are organized in classes and small frameworks that can be deployed and reused by a third party in order to solve problems involving ANN development. The created components were built in Java and packaged as JavaBeans, the components standard for the Java language. It was an important goal of this work to be able to provide the CANN components in a standard that other people have been adhering to, so that the components have a critical mass to use them. At the same time, it is not the intention of this work to compare different component standards or to provide the CANN components in other component standards. However, this work could be done in the future given the maturity of one or another component standard.

The actual CANN framework implementation has some coding dependencies among the frameworks like the ANN and Domain framework or even the GUI framework.

However, the developer that understands the code done so far is able to reuse each of the frameworks individually. To evolve in order to reach a *blackbox reuse* level the CANN framework still has to mature. As mentioned before, there is the possibility of some design review in order to improve reusability especially at the level of the simulation and GUI frameworks.

The CANN framework not only permits the reuse of source code but also the architecture design. The degree of reusability expected is then even bigger than considering only code reuse. For instance when developing the different ANN models in CANN, frameworks like Domain, Converter and GUI can be reused while code extension is necessary only to implement the specific characteristics of the ANN model at the level of ANN and Simulation frameworks, but keeping their architecture design as well.

Code mobility in the form of objects or agents is an open problem in the component research area. There are no clear architecture standards and design patterns that facilitate the design and implementation of general solutions for component mobility. Chapter 5 explains the design of a solution for making the CANN components for ANN mobile. The main design guidelines and implementation issues may be applied not only to ANN components but also to any other kind of components.

Two important aspects on the implementation of components were introduced quite late in the CANN implementation so that they still lack improvement. The first is the use of a tool for the proper documentation of the source code. In this case, the Javadoc technology was adopted. The second aspect is code versioning control. It is very important to keep the code under the guard of a versioning control system. It helps not only to keep the appropriate version relation among the components but also to track its design evolution.

The development of the CANN framework may still evolve in some aspects. It is important to introduce a better error control and logging where the error levels may be better managed. There are Java API's that can help with that implementation aspect such as the Log4J (Log4J Project). It is also important to build and/or find testing tools to test the components individually. The possibility of quickly creating tests for the new or extended components in order to verify the errors before coupling the components in a complete application helps the components developer. This kind of test reduces drastically the application errors. An interesting tool to be developed is also a RAD (Rapid Application Development) environment where the developer can quickly program by composition of the

CANN components. Such a tool may be useful not only for somebody who wants to apply ANN but also for didactic purposes.

The next section analyzes how other authors approached the problem of designing and implementing ANN software, and how they relate to the CANN framework.

3.3 Related Work

This chapter introduces a selection of software solutions for the construction of ANNs. The study of these solutions was important to consolidate the author's knowledge for the proposition of a software solution based on components. This study considers how the different solutions take care of design aspects such as modeling the ANN structure and implementing its functionality. It also considers how the different authors implement general code to be used by different models and by other people; in short, what the authors choose to be the reused code, and how it was engineered. It also analyzes simulation problems, such as memory and CPU allocation, implementation parallelism, etc.

The first studied solution is the simulation software proposed by Freeman and Skapura (1992). They developed a software simulation environment in Pascal. Then, attempts to develop ANN software based on the object-oriented paradigm are analyzed in particular the works of Timothy Masters (1993), Ivo Vondrák (1994) and Joey Rogers (1997). Masters centers his work on giving tips on how to implement ANN different functionality in C++. Vondrák's work is fully concentrated on designing a fully object-oriented solution to ANN software implementation. His language of choice was Smalltalk. Finally, Rogers also concentrates on designing object-oriented software to implement diverse ANN models in C++.

3.3.1 The Freeman and Skapura (1992) solution

In their work, Freeman and Skapura attempt not only to explain the ANN mathematics but also to give solid examples on how to implement ANNs. Undoubtedly they succeed on this task by offering a complete software solution for ANN development. The code is written in Pascal and is clear and precise. The authors propose two forms of organizing the ANN data structures: based on arrays and based on linked-lists. Each solution has its advantages and disadvantages. Both are analyzed in detail in this section.

3.3.1.1 Array-based ANN structures

In such a solution, data is arranged by groups of linearly sequential arrays, each containing homogeneous data, such as sequential arrays containing connection weight values or neuron activation values. In the first case, each index of the array corresponds to a connection and in the second case each index corresponds to a neuron. This approach is useful to be fast in stepping through the neurons or connections once they are "represented" by simple indexes in arrays, and stepping over array indexes is fast. This approach is faster than using linked-lists (next section) where an address lookup has to be accomplished for each step. The drawback of such a solution is the lack of generality in regard to the size of the allocated networks.

Figures 3.32 and 3.33 show how to map the ANN structure into arrays. The first figure shows a two-layer network where the lower layer of neurons produces individual outputs. Note that each neuron produces an output that is referred by *O* and indexed by a neuron's sequential number. The connection weights among the lower and the upper layers are represented by *W* and indexed by the lower and upper neurons it connecs. The lower neurons are represented by the numbers from *1* to *5* and the upper neurons by the variable *i*.

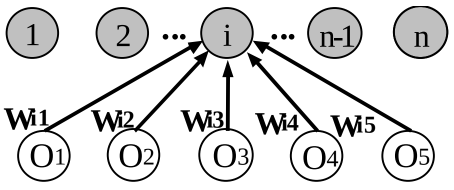


Figure 3.32 – Two-layer network weigh and output arrays (Freeman 1992)

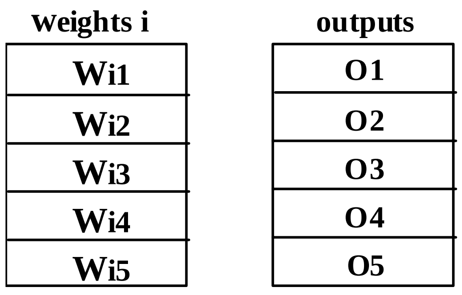


Figure 3.33 – Array data structures for computing net$_i$ (Freeman 1992)

The array-based architecture can be described as follows:

- Each layer of neurons is represented by an array that stores its output. That is, for each neuron output there is one position on the array that stores it. The array index means the neuron index.
- Each neuron output connects via a synapse to the neurons of the next layer.
- Each neuron of the next layer will receive the outputs of the previous layer as input.
- For each neuron of the receiving layer, there will be an array of weights. This array will have exactly the same size of the previous array of outputs.

The sum-of-products (net_i) is done in the following way:

1. Iterate over the two arrays operating the product of each weight value with the corresponding output value (take the values using the same array index).
2. The resulting product is stored in an auxiliary accumulator that must have been previously initialized to zero.
3. Then the next index is considered and the same operation is done. The weight is multiplied by the output and the result is summed to the accumulator value.
4. After iterating over all possible indexes (neurons), the calculation is finished.

The operations over those arrays are based on indexes. As the arrays are statically allocated, the index iteration is, in fact, simply a pointer increment on the memory, being a pretty fast operation.

It is clear that such a solution means building specific arrays for the specific ANN model in development. Also it is necessary to program-by-hard the relations among the arrays to form the proper model architecture. This solution is certainly not difficult to develop and may have good performance results, but it is not flexible for modifications. Any change of the model architecture may mean programming modifications at the level of the arrays constructions and relationships. Furthermore, the ANN cannot increase in size at runtime. All the necessary memory has to be allocated in advance. There are ANN models that can change in size dynamically (e.g. the CNM model), which makes this solution impractical. Sometimes the memory usage of an ANN model cannot be forecast. If the previously allocated memory is excessive, not being totally used, a waste of resource will be characterized. If the allocated memory is not enough, the process may be stopped in the middle, being a waste of time.

Another problem is that if the neurons of different layers are not fully connected, this solution will not work properly. For the connections that are not used it is necessary to introduce controls like indexes or consider connections (array positions) with null value. The sparser the net is or turns to be during the learning, the more losses in memory spaces and processing time will occur. The implementation turns out to be restrictive and complicated to extend.

3.3.1.2 Linked-list-based ANN structures

Here the arrays describing the neuron weights or the neurons layer outputs are implemented using linked lists. Each element on the list points to the next element. The elements can be allocated at run time making lists of any size possible. The connections among the different layers of the network may be also implemented via pointers and different approaches can be used to correctly build the architecture. Connection lists can be built to indicate to which inputs in an upper layer the output list neurons must be connected.

Such a solution brings the advantage of generality on the architecture implementation and processing, once any ANN model can use the list navigation algorithm. But the disadvantages are twofold in relation to the array-based solution:

- It allocates more memory to store the list pointers
- It consumes more processing time by having to navigate over the links to access the data structures.

Freeman then chooses to have arrays as data structures, but dynamically allocated, so that the arrays can be allocated at run-time, but their size is fixed once allocated. The advantage is also that the iteration inside the array is done via index, keeping the array performance quality.

Figure 3.34 shows the layered structure that is used to model a collection of neurons with similar function. The first neuron computed output value is stored in its correspondent output index (*O1*) on the *outputs* array. The weight values of all input connections to the first neuron are stored sequentially in the *W2j* array. Those values are accessed to calculate the *O1* output. The same happens to the other neurons in sequence. The neural network will manage as many layer arrays as necessary to implement the ANN structure.

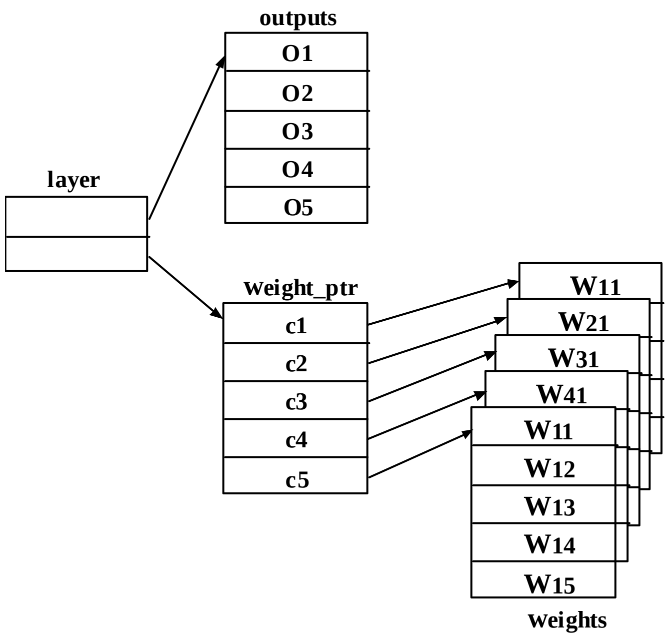


Figure 3.34 – Layered structure (Freeman 1992)

A 4x5x2 Backpropagation network can be taking as an example. It may manage three arrays for the input, hidden and output layers. The input layer would have only an *outputs* array since it does not have connections to a previous layer. The output values (4 values) would be filled with the input activation and are used to calculate the activation of the hidden neurons. The hidden and output layers would have an array of *outputs* and an array of weight pointers (*weight_ptr*) as seen in Figure 3.34. Each position of the weight_ptr array points to an array of weights. The number of positions on the *weight_ptr* array corresponds to the number of neurons on the layer (5 to the hidden layer and 2 to the output layer). The number of weights stored on each *weights* array corresponds to the number of connections each neuron of this layer has to the previous layer. In the example the hidden layer would have 5 elements on the *outputs* and *weight_ptr* arrays. Each *weight* array would have 4 weights corresponding to each connection to the input neurons. Similarly, it would happen to the output layer. It would have 2 elements on the *outputs* and *weight_ptr* arrays corresponding to its output neurons. Each *weight* array would have 5 weights corresponding to each connection to the input neurons.

However, Freeman and Skapura do not focus on analyzing the two solutions in order to verify how much one is better than the other. Their preoccupation is to show what must be done to simulate ANN and not how to implement the models. The empirical comparison of two methods such as those ones is not necessary in this work too. The work effort would be too high to find out results that can be easily forecast by previous knowledge on programming similar solutions for ANN building.

The simulation based on vectors and matrixes has as advantage the fact that this kind of computation is widely used in scientific computation. Many computers have special hardware to process vectors like supercomputers that operate very efficiently with long vectors at the same time. As the ANN processing is massively parallel, it is possible to define parallel algorithms to make use of this computer capability. However, the main disadvantage of such a solution is the reduced reusability. The structured programming approach presented here has its inherent difficulties, such as the necessity of reprogramming parts of the code in order to be able to implement new algorithms, ANN models, learning rules, data interaction, etc.

3.3.2 The Timothy Masters (1993) solution

In his book "Practical Neural Network Recipes in C++", Timothy Masters intends to address ANN beginners. The book quickly explains the several ANN paradigms introducing the mathematical aspects and briefly shows code examples for the relevant aspects. The coding is done in C++ and, in general, is formed by code pieces of the structured programming stile.

The author does not apply OO aspects in the software implementation. There aren't detailed explanations on the design used to implement the several models. He who wants to have more details about the implementation has to wade through to the full source code that comes with the book in an additional floppy disk. But this code is also not properly documented. There is a simple user's manual that helps to run the software and provides few code comments. Therefore, the study of the core implementation aspects turned out to be complicated.

Masters has defined a general top-level class for ANNs, called *Network*. The ANN structure is built in memory and as the implementation is done in C++, the programmer has to manage memory. Particular ANN methods are the following few ones:

1. *learn* – do the learning process.

2. *trial* – calculate the output for a given input by evaluating the network.

3. *trial error* – compute the training set error.

It also implements *save* and *restore* methods for making the ANN structures persistent in file streams. *It is possible to see in CANN INetImplementation class methods that implement very similar functionality to those above.*[2]

There are two subclasses of *Network*: *LayerNet* and *KohNet*. The first is the basis for building different multilayer-based ANN models. The second implements the Kohonen SOM model (Kohonen, 1984). Both subclasses have some methods for specific model computations such as "*finding gradient*" in *LayerNet* and "*winner*" in *KohNet*.

There are only two other classes for the whole simulation environment: *TrainingSet* and *SingularValueDecomp*. The first implements the management of a collection of samples, which will be used for training. This functionality is important and the author shows, by implementing a separate class, the necessity of having it independent of the core ANN classes. The *LayerNet* class for the implementation of the regression algorithm uses the second. This class is completely out of the scope of the whole design. It implements something specific for a particular ANN model and is not important for the rest of the code. In most cases the author chooses to implement model particularities inside the specific subclasses of *LayerNet*. The *SingularValueDecomp* class is isolated and shows lack of designs and clarity of the system.

The absence of more fine-grained classes to implement the ANN models and the presence of only few "virtual" (abstract) methods on the *Network* class shows that the design chosen by Masters is not much concerned with reusability of code. Most of the essential parts are coded inside each ANN subclass and not many can be reused. For example: the author implemented several mathematical functions in methods inside the specific ANN models such as: weight regression, simulated annealing, sigmoid function, gradient descent calculation, etc.

Furthermore, if a careful reuse design approach had been done, certainly some code could be made generic by improving the interface of the top-level class *Network*. For example,

[2] In Cursive character type the author comments that compares somehow the related work with CANN.

methods that manipulate weights of retrieve network results (*winner*), could be implemented in the generic class *Network* and be useful to other ANN subclasses.

On the other hand, some code could be moved to specific subclasses such as the *confusion* method in the class *Network*, which certainly shouldn't be there. The *confusion* is a method to help in classification networks, so that it could be added to a subclass that specifically groups networks that would be able to use it or need it. A class could be created to implement this facility and referred by any ANN that would use it.

If considering OO design, Masters approach can be considered a further step in relation to Freeman that did not implement OO aspects. But his approach is still not mature in terms of OO design. Unfortunately, the comments on the code are not enough to have a clear understanding about the meaning of certain attributes and methods, and sometimes even some classes. It is not possible to make a profound analysis of the implementation within a reasonable amount of time but it is clearly few steps behind of being characterized as OO design and implementation. Finally, the author does not provide any design diagrams, the defined classes are few compared to the problem at hand, the methods are not generic to the classes that contain them, the code is organized as libraries of structured code, not as objects.

3.3.3 The Ivo Vondrák (1994) solution

In his papers (Vondrák, 1994, 1994a), Ivo Vondrák makes a parallel between the OO concept of message exchange among objects via appropriate methods usage and the exchange of information at the ANN. The collection of messages an object is able to react to forms its supported protocol. The ANN has a similar behavior where the nets have neurons that communicate to each other by signals transmitted by the connections among them. Therefore, Vondrák's conclusion is that an ANN can be mapped into a computer model using the OO paradigm.

Analogous to CANN, that OO solution resembles the objects in an ANN. Vondrák proposes objects to represent neurons, the interconnections among them, the layer of the connected neurons and the whole network. The operations on these objects represent the ANN tasks such as passing the signal, adaptation, self-organization and changing the topology. A hierarchy of classes is built to represent the various types of objects originated from the various ANN models to implement.

3.3.3.1 Hierarchy of Neurons

The hierarchy of neurons is shown in Figure 3.35. *Neuron* is the top class, its subclasses define special behavior for different ANNs. The code that defines a *Neuron* is shown in Code 3.6.

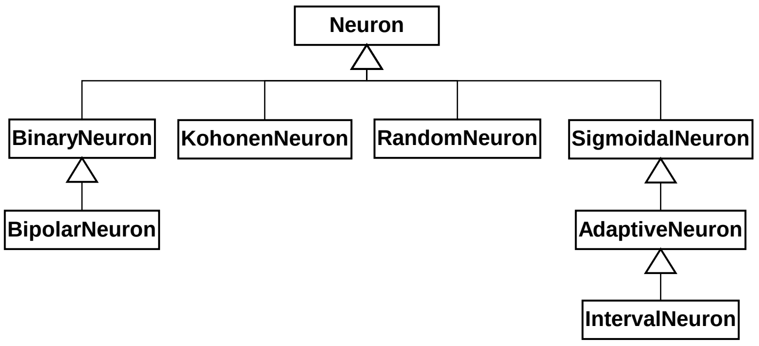


Figure 3.35 – *Neuron* hierarchy (Vondrák 1994)

The *Neuron* class stores data necessary to compute its activation. The abstract method that does the activation calculation is called *transfer*. The subclasses of *Neuron* implement this function in the appropriate way regarding its ANN model. Here Vondrák chooses the unification metapattern (Pree, 1995) where the different behaviors of the *Neuron* types are implemented by a method that, is dynamically bound and that reacts in different ways in the different *Neuron* subclasses. Whenever a different behavior is necessary, a new subclass of *Neuron* shall be implemented. The remaining methods described in Code 3.6, are implemented operations that are general to all *Neuron* types. One method is used to initialize the neuron (*initialize*), another to add a signal to the neuron activation calculation (*adjustPotential*) and, finally, a method is implemented to check the actual neuron activation (*getState*).

Code 3.6 – The *Neuron* class (Vondrák 1994)

```
class:       Neuron
superclass:  Object
data elements:
       potential    "inner potential"
       state        "state of the excitation"
       threshold    "threshold of the neuron"
       name         "represented by number"
message protocol:
       initialize: aName   "initialization of the neuron and setting a name"
       adjustPotential: aFloat   "add an input signal to the potential of the
neuron"
       transfer                 "abstract method for the activation function"
       getState                 "returns the state of the neuron"
```

Some *Neuron* types need to introduce other messages in its communication protocol, that means introducing new methods to perform activities specific to the ANN model it belongs to. In this case, once again the solution is to do subclassing. The subclasses will introduce the new method not implemented by the superclass. It is the case of *IntervalNeuron*, which introduces the method *transferInterval* to implement the possibility of assigning a state of excitation to the neuron controlled by an interval via its data elements *minState* and *maxState*.

3.3.3.2 Hierarchy of Connections

The connection between neurons is used to pass signals from one neuron to another. It also represents the first level of the topology of neurons defined by Vondrák. The implementation of the *Connection* class can be seen in Code 3.7. The *Connection* object stores the synaptic weight and references to the neurons it is connected to. *Connection* has a method to initialize it, a method called *adjust* to update the weight value and a third method called *passSignal* used to transfer the signal from the "first" to the "second" neuron connected to the *Connection* object. It is interesting that this connection is directional from one neuron to the other, from the first to the second, so that Vondrák's model does not take into consideration that a connection could be bi-directional, even though it has the information about the two involved neurons on the connection.

It is not clear whether it would be easy to change this "semantics" (direction) of the connection. Anyway the processing in the reverse direction, from "second" to "first" neuron could be done by overwriting the passSignal method or by defining a new method, in a *Connection* subclass, similar to *passSignal* but in the opposite direction. At least the ANN developer could create two connections, one from the "first" neuron to the "second" and the other from the "second" to the "first" neuron.

Code 3.7 – The *Connection* class (Vondrák 1994)

```
class:          Connection
superclass: Object
data elements:
      first           “first neuron from the couple of neurons”
      second          “second neuron”
      weight          “weight of the interconnection”
message protocol:
      initialize      “initialization of the connection”
      adjust: aFloat  “adjust a weight”
      passSignal      “pass a signal from the first to the second
neuron”
```

There is one *Connection* subclass called *IntervalConnection* which implements the same behavior defined for *IntervalNeuron* where an interval state of excitation is possible. A new method *passIntervalSignal* is added to implement this behavior.

Finally, it is possible to say that Vondrák’s *Connection* class is appropriate for the simplicity of its behavior.

3.3.3.3 Hierarchy of Interconnections

The class called *Interconnections* represents a set of connections among neurons defining a part of the whole neural network. Code 3.8 shows the implementation of this class in detail. The main role of this class is to determine the way the neurons shall be connected, thus being dependent of the specific ANN model. The class stores a dynamic collection of *Connection* objects. Similarly to the *Connection* class, it has the methods *adjust* and *passSignal* that have the same functionality yet applying it to the collection of connection objects it holds.

Code 3.8 – The *Interconnections* class (Vondrák 1994)

```
class:          Interconnections
superclass: Object
data elements:
      connections  “dynamic collection of the connection”
message protocol:
      initWeights  “initialization of the weights of the connection”
      adjust       “adjust weights of interconnections”
      passSignal   “pass a signal between neurons”
      add: aConnection      “add a connection”
      remove: aConnection  “remove a connection from the collection”
```

The subclasses of *Interconnections* implement the concrete solutions for the methods *adjust* and *passSignal*. Therefore, the hierarchy shown in Figure 3.36 is created. The subclasses also add specific methods to do the appropriate creation of the object and the appropriate connections, such as the method *connect*.

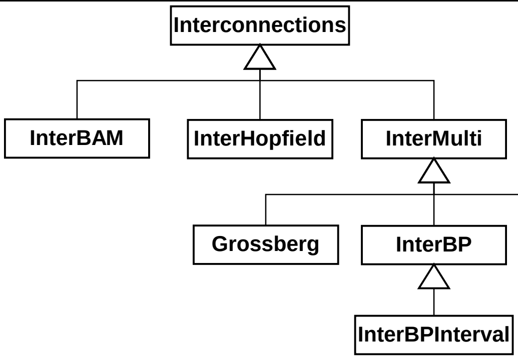


Figure 3.36 – Interconnections hierarchy (Vondrák 1994)

There is no class such as the Interconnections in CANN. The creation of the ANN structure is the responsibility of the components that implement INetImplementation interface (see Section 3.2.2.4). They shall implement the abstract method generateNet that is called when the ANN must be built. Each specific component implements this method in its appropriate way. At first sight, the Interconnections object did not seem to be important, but with the experience of implementing four different models using the CANN framework, it is clear that it makes sense. The Interconnections plays the role of the concept of an ANN layer and is a possible element of code reuse. The creation of layers of interconnected synapses and neurons makes sense for all models and the code implementation is very similar to the differently implemented models. In CANN the grouping of sets of neurons is done using the Java Vector class and the Neuron class controls the connection among neurons. A class like Interconnections can encapsulate this functionality avoiding repetitive code such as interacting over the neurons vector.

3.3.3.4 Hierarchy of Artificial Neural Networks

Figure 3.37 shows Vondrák's class hierarchy for ANNs. The class *NeuralNet* is the top class and its code is shown in Code 3.9. This class is responsible for putting together the layers of *Interconnections* to define the ANN topology. Its subclasses implement complementary methods to provide management for the functionality among the interconnections.

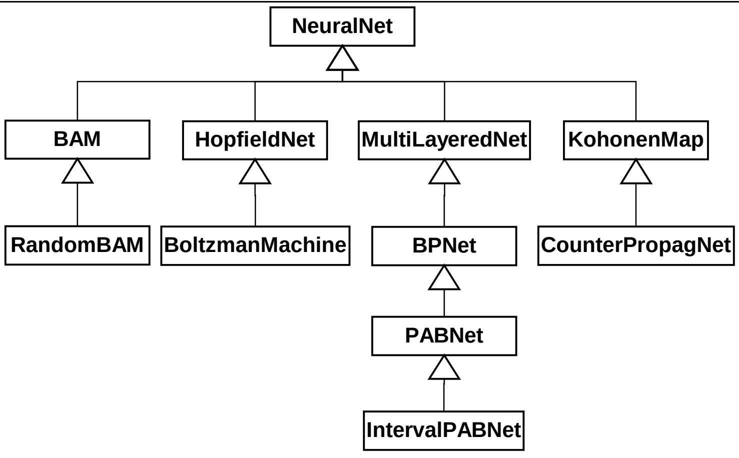


Figure 3.37 – Neural Network hierarchy (Vondrák 1994)

The *NeuralNet* class implements two important methods: *learning* and *run*. The method *learning* is responsible for the adaptation functionality of the model by implementing the learning procedure for a training set. The method *run* is simply a recall information method, which means, it is used to test the previously adapted network with one case. The *learning* method receives as parameter *aTrainingSet* and the *run* method receives *anInput*. It is not possible to exactly infer from the text what those data types are, but they certainly are objects that implement some functionality that the *NeuralNet* subclasses are able to cope with. Those objects can have, internally, training or testing data already pre-processed or not to be applied to the ANN. This means that the *NeuralNet* objects can also do the pre-processing of data before applying it to the ANN structure (e.g. a*djust* method for interconnections).

Code 3.9 – The *NeuralNet* class (Vondrák 1994)

```
class:          NeuralNet
superclass: Object
data elements:
      inter           “dynamic collection of the interconnections”
message protocol:
      initNet                     “initialization of the neural net”
      learning: aTrainingSet      “adaptation of the network”
      run: anInput                “recall the information”
```

Vondrák points out that the advantages of using the OO approach to build ANN software are “the direct reincarnation of the real nets into the computer model and the possibilities to reuse the already written code.” Also “the possibility to redefine or to extend hierarchies and adapt them for the solution of the concrete problem.” He finally points to the higher reliability of the code.

Even being the consulted material not extensive, it is clear that the design is elegant and promises to be a generic software solution for implementing ANN.

3.3.4 The Joey Rogers (1997) solution

In his book "Object-Oriented Neural Networks in C++" (1997), Joey Rogers probably wrote the most detailed publication so far about implementing ANN using the object-oriented paradigm. Rogers uses OO programming techniques "to find the inherent object nature found in all neural networks to create a flexible set of reusable classes that simplifies the implementation of any NN architecture. The gain one should have is not only a tool chest of reusable objects to aid in the implementation of NN architectures, but learn the development process for creating such objects and realizing any new architecture."

This inherent object nature is the presence of neurons and connections among neurons, the synapses. So Rogers (1997), like Vondrák (1994) already did, built these basic ANN elements as objects and reused them for all implemented models.

Rogers starts by implementing two base ANN classes: *Base_Node* and *Base_Link*. Those classes should have all functionality needed by any ANN model so that the programmer does not need to implement it again. The implementation of one model follows the other; the objects created for one are reused in the others.

3.3.4.1 The Base_Node *Class*

The principal characteristics of the *Base_Node* class are:

1. It processes input and produces output.
2. It does not define how the node object processes information; the task is done in the subclasses.
3. It maintains lists of connections to the associated nodes. Two linked lists are used to store associated nodes: one maintains the nodes that bring input to the node, the other the list of nodes that receive the output of this neuron.

The *Base_Link* objects (explained later) do the connections among *Base_Node* objects. The *Base_Link* objects store two pointers, one to the source and one to the destination neuron. Therefore, the *Base_Node* objects point only to the links. Figure 3.38 shows an example of network formed by *Base_Link* (rectangles) and *Base_Node* (circles) objects.

The creation of those links will generate an additional overhead that is softened by the flexibility of creating any architecture and the functionality capability associated to this structure once the *Base_Link* objects also store the connection weight and can process information.

The CANN Neuron class (see Section 3.2.2.1) does not have a list of input and output connections. It has only one list of input connections, e.g. a list of synapses that bring activation to the neuron. The neuron does not need to know what neurons are associated to its input because each synapse connected to it knows its source and destination neurons. There is no list of synapses associated to the neuron output. Once the synapse knows to which neurons it is associated, it is able to request the neuron output value whenever necessary. The list of synapses is maintained by Neuron as a Java Vector object.

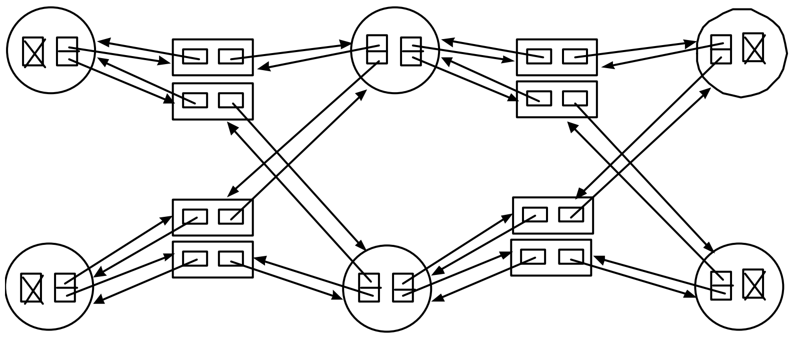


Figure 3.38 – Object representation of network topology (Rogers, 1997)

The disadvantage of Rogers' solution is the necessity of managing many pointers. When implementing in C++ keeping those pointers consistently allocated generates a programming overhead. The programmer is responsible for allocating and deallocating those pointers being a highly error-prune solution.

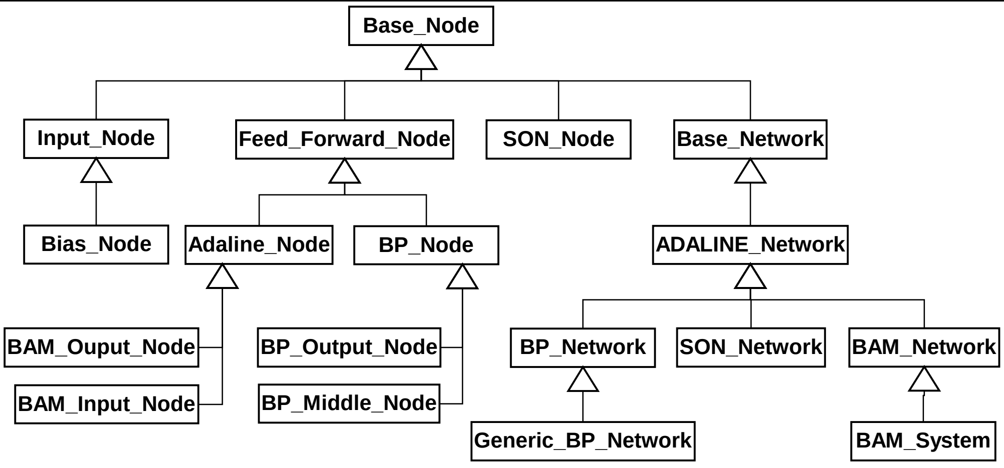


Figure 3.39 – Neural Network Node hierarchy (Rogers, 1997)

The *Base_Node* class hierarchy is shown in Figure 3.39 and its definition is shown in Code 3.10. The *Base_Node* stores a set of numeric "values" that is useful to store the results of any processing done inside the neuron. This set of "values" was created to make the class generic and to make it useful in several ANN architectures. It also stores "error" results in the same way. There are getter and setter methods for "values" and "errors". This class also stores the node name and ID. The ID is used for certain iteration controls and for the object serialization.

The Neuron class in CANN does not store a set of values but a single value, which is the neuron activation called currentActivation. The neurons also do not store name and ID because they are expensive data in terms of memory footprint. It would be waste of memory space to allocate a string to the name of each neuron in a CNM network where there could exist millions of combinatorial neurons.

The ID is not necessary once the neurons are stored in Java Vectors, which are classes that have iteration facilities via indexes. Furthermore, the Java serialization and reflection engines guarantee the unique access to the objects not being necessary to have IDs to identify them. Another important contribution of Java is the error handling mechanism that allows a seamless treatment of exceptions in classes and methods. Therefore it is not necessary to store error codes inside the classes.

Rogers' solution stores learning parameters in the node values instead of storing them in the ANN model structure. The *GetValue* and *SetValue* methods are used to get and set the necessary learning parameters for all nodes of the model. Sometimes some learning parameters are, in fact, necessary inside the neuron to be able to do the activation calculation.

However those values should not be stored inside the neurons because it causes a waste of memory by allocating space for storing the same value as many times as node instances exist. The natural solution would be to store those values at the ANN level and pass them by parameter to the nodes whenever necessary.

Base_Node has constructor and destructor methods for allocating and deallocating the object's internal variables. It also provides methods for saving its state to the disk - methods save and load that operate over an *iostream*, - and a print method to print its state to the standard IO.

In the Java solution there is no necessity to implement the class destruction method because the destruction is provided automatically by the Java's garbage collection engine. It is also not necessary to implement save and load methods for neurons and connections because the Java Persistence facility provides this automatically. Classes are only required to implement the Serializable interface in order to be automatically enabled to be serialized. In this case, instance variables that shall not be persistent shall be declared transient.

The most important methods defined by *Base_Node* are certainly *Learn*, *Run* and *Epoch*. Those methods are abstract so that they are implemented only by the subclasses. The *Run* method defines the node operation when evaluating an input in "normal" operation, typically testing. The *Learn* method is used during the training process and the third method *Epoch* is important for unusual situations to reset values or to do special operations.

Code 3.10 - The *Base_Node* class (Rogers, 1997).

```
class Base_Node            // Base Neural-Network Node
      {
      private:
           static int ticket;

      protected:
           int id;             // Identification Number
           double *value;      // Value(s) stored by this node
           int value_size;     // Number of Values stored by this node
           double *error;      // Error value(s) stored by this node
           int error_size;     // Number of Error values stored by this node

           LList in_links;     // List for input links
           LList out_links;    // List for output links

      public:
           Base_Node( int v_size=1, int e_size=1 );     // Constructor
           ~Base_Node( void  );                         // Destructor
           LList *In_Links( void );
           LList *Out_Links( void );
           virtual void Run( int mode=0 );
           virtual void Learn( int mode=0 );
           virtual void Epoch( int code=0 );
           virtual void Load( ifstream &infile );
           virtual void Save( ofstream &outfile);
           inline virtual double Get_Value( int id=NODE_VALUE );
           inline virtual void Set_Value( double new_val, int id=NODE_VALUE );
           inline virtual double Get_Error( int id=NODE_ERROR );
           inline virtual void Set_Error( double new_val, int id=NODE_ERROR );
           inline int Get_ID( void );
           inline virtual char *Get_Name( void );
           void Create_Link_To( Base_Node &to_node, Base_Link *link );
           virtual void Print( ofstream &out );

           friend void Connect( Base_Node &from_node, Base_Node &to_node,
                                Base_Link *link );
           friend void Connect( Base_Node &from_node, Base_Node &to_node,
                                Base_Link &link );
           friend void Connect( Base_Node *from_node, Base_Node *to_node,
                                Base_Link *link );
           friend int Disconnect( Base_Node *from_node, Base_Node *to_node);

           friend double Random( double lower_bound, double upper_bound );
      };
```

The CANN Neuron class has only one abstract method for both Run and Learn operations defined in Rogers' Base_Node class because both do the same thing, e.g. calculate the neuron activation. The calculated activation is the same independently whether the ANN is in the learning or testing process. There is no similar operation like the one defined by the Epoch method.

In case of the CANN Neuron class, the process of information is not left to the subclasses. The creation of subclasses of Neuron and Synapse is frequently done to implement the specialties of each ANN model. But the implementation of the activation function is done in separate subclasses of the class ComputationStrategy, which avoids the creation of nested subclasses for the same model just implementing various activation functions. Using the ComputationStrategy solution the activation *functions could be defined even at runtime by "plugging" in a new instance of a specific ComputationStrategy. That is the use of the separation metapattern instead of the unification metapattern.*

Finally there are "static" methods called *Connect* that implement the facility of connecting *Base_Node* objects through *Base_Link* objects. There is also a *Disconnect* method to undo such a connection.

The "static" Connect methods defined by Rogers' Base_Node class are also unnecessary in the CANN Neuron definition. The connection among neurons is done using a more generic method called generateSynapses that implements the connection of the input synapses to the neuron in the appropriate way the ANN model requires. The Disconnect method implemented by Rogers was not implemented in CANN Neuron class but it is certainly a good idea to have the chance to disconnect neurons on the fly.

3.3.4.2 The **Base_Link** *class*

The *Base_Link* class hierarchy can be seen in Figure 3.40 below.

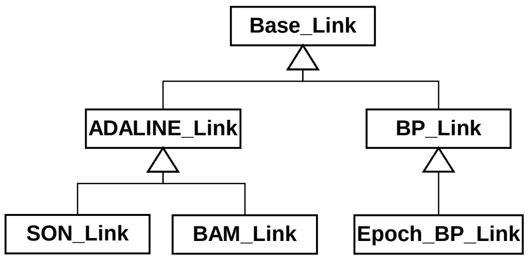


Figure 3.40 – Neural Network Links hierarchy (Rogers, 1997)

The *Base_Link* class definition can be seen in Code 3.11 and has basically the same methods as *Base_Node*. It has specific methods to manage connections among its input and output neurons and to access them. Another extra method *Update_Weight* is used to update the stored weight value.

Code 3.11 - The *Base_Link* class (Rogers, 1997)

```
class Base_Link   // Base Neural-Network Link class
     {
     private:

          static int ticket;

     protected:
          int id;                     // ID number for link
          double *value;              // Value(s) for Link
          Base_Node *in_node;         // Node instance link is comming from
          Base_Node *out_node;        // Node instance link is going to
          int value_size;

     public:
          Base_Link( int size=1 );       // Constructor
          ~Base_Link( void );            // Destructor for Base Links
          virtual void Save( ofstream &outfile );
          virtual void Load( ifstream &infile );
          inline virtual double Get_Value( int id=WEIGHT );
          inline virtual void Set_Value( double new_val, int id=WEIGHT);
          inline virtual void Set_In_Node( Base_Node *node, int id );
          inline virtual void Set_Out_Node( Base_Node *node, int id );
          inline virtual Base_Node *In_Node( void );
          inline virtual Base_Node *Out_Node( void );
          inline virtual char *Get_Name( void );
          inline virtual void Update_Weight( double new_val );
          inline int Get_ID( void );
          inline virtual double In_Value( int mode=NODE_VALUE );
          inline virtual double Out_Value( int mode=NODE_VALUE );
          inline virtual double In_Error( int mode=NODE_ERROR );
          inline virtual double Out_Error( int mode=NODE_ERROR );
          inline virtual double Weighted_In_Value( int mode=NODE_VALUE );
          inline virtual double Weighted_Out_Value( int mode=NODE_VALUE );
          inline virtual double Weighted_In_Error( int mode=NODE_VALUE );
          inline virtual double Weighted_Out_Error( int mode=NODE_VALUE );
          inline virtual int Get_Set_Size( void );
          inline virtual void Epoch( int mode=0 );
     };
```

3.3.4.3 The **Feed_Forward_Node** *class*

This class provides the neuron functionality for implementing feedforward networks. Those networks generally have nodes that apply a simple threshold function as its neuron activation function such as the Sigmoid function. From this class Rogers derives the classes needed for the ANN models implemented later.

3.3.4.4 The **Base_Network** *class*

Rogers's implementation of the *Base_Network* class can be seen in Code 3.12. It manages two arrays of *Base_Node* and *Base_Link* classes.

That is pretty similar to what is done in the CANN NetImplementation class by implementing input and output neuron vectors (see Section 3.2.2.4). But there is no necessity of having the control of the connections (Synapses) in the NetManager class once they are controlled by the Neuron classes.

The methods *Load_Nodes_Links* and *Save_Nodes_Links* implement persistence, which is, once again not necessary in a Java implementation because of the automatic persistence engine of the Java language. The *Base_Network* class also has a *Create_Network* method that is abstract (virtual) giving to its subclasses the responsibility of implementing the network specific topology.

The CANN NetManager class that defines the abstract method GenerateNet does the same.

The *Base_Network* method *Load_Inputs* is also an abstract method and is used to standardize the loading of input values into the input nodes. Here Rogers expresses the necessity of having a standard way of feeding data to the ANN for the learning and testing processes.

That is what was done by the implementation of the CANN Domain and Converter frameworks (see Sections 3.2.4 and 3.2.5).

Finally, similar to the *Base_Node* class, a method is implemented called *Epoch* to execute one network operation epoch for all links and nodes. Other methods inherited by *Base_Node* such as *Get_Value*, *Set_Value*, *Save*, *Load*, *Run* and *Learn*, are overridden in the *ADALINE_Network* class. In general, those operations are extended to cope with the arrays of *Base_Node* and *Link_Node* objects contained in the *Base_Network* class.

After checking the *Base_Network* definition, lets recheck Rogers's class hierarchy for ANN in Figure 3.37. The *BP_Network* class implements the Backpropagation model. It could be derived from *Base_Network* class, but "to simplify things even further" and "to take advantage of the operations that have already been defined", it was derived from the *ADALINE_Network* class. Nevertheless, the reason for deriving the *BP_Network* class from the *ADALINE_Network* class is not completely clear, but it is understandable. However, deriving models like Kohonen's SOM from the Adaline model is pretty complicated to understand. Rogers does not explain why he decided to define *SON_Network* class as a subclass to *ADALINE_Network*. He recognizes they are completely different and says nothing about the possible similarities that could justify his design. Taking a closer look at the *SON_Network* implementation, one can conclude that it could perfectly inherit directly from *Base_Network*, avoiding the design misunderstanding.

Code 3.12 - The *Base_Network* class (Rogers, 1997)

```
class Base_Network : public Base_Node     // Base Network Node
     {
     protected:
          int num_nodes;                  // Number of nodes in Network
          int num_links;                  // Number of links in Network
          Base_Node **node;               // Array of base nodes
          Base_Link **link;               // Array of base links

          virtual void Create_Network( void );
          virtual void Load_Inputs( void );
          virtual void Save_Nodes_Links( ofstream &outfile );
          virtual void Load_Nodes_Links( ifstream &infile );

     public:
          Base_Network( void );           // Constructor
          ~Base_Network( void );          // Destructor
          virtual void Epoch( int code=0 );
          virtual void Print( ofstream &outfile );
          virtual char *Get_Name( void );
     };
```

Clearly the class hierarchy constructed by Rogers does not reflect the real domain, which makes it difficult to understand, use and extend. The use of OO design in this case may be more complicated rather than helpful. Going even further, Figure 3.37 shows that Rogers's *ADALINE_Network* derives from *Base_Network* class that derives from *Base_Node* class. This could be read like: *Base_Network* is a *Base_Node;* or a network is a node. This design is unnatural because nobody can see such hierarchy in the domain. Coad and Jourdon (1991) advise not to use generalization-specialization hierarchies for relations that are not found on the problem domain. It is simply wrong that a network is a specialization of a node.

The justification of Rogers for such a design is: "Since the Adaline neural network receives inputs, processes those inputs, and produces an output, the Adaline neural network itself can be abstracted to the node model." This justification is not enough since virtually any computation process implements this functionality of receiving and processing input and producing output, so that any class could be a subclass of his *Base_Node* class.

The same kind of problem can be verified in the hierarchies for *Feed_Forward_Node* and *Base_Link*. Rogers derives *Feed_Forward_Node* class from the *Base_Node* class, and the definition for node classes of other models from the same *Feed_Forward_Node* class. He also derives *SOM_Link* and *BAM_Link* from *ADALINE_Link*. Both hierarchies do not reflect the problem domain.

Probably the true reason why Rogers opted for such a design was the possibility of inheriting attributes and methods (abstract at most), which is not enough. He come up with such a design because each model was built on the results of another previously implemented one. Because of this "sequential" implementation, he tried to reuse as much code as possible through inheritance, but ended up with an unnatural design.

This approach used by Rogers is certainly not appropriate for building OO systems. For instance, the class hierarchy would be different if he had chosen to implement the Backpropagation model before the Adaline model, and so on. His design would be different also if he had considered many ANN models in advance and built as many general classes as possible, implementing the commonalties of the several models. Then, the hierarchy would probably be more general and reusable.

Taking that idea into consideration, it is possible to imagine something similar to Rogers's design by considering that ANN and neurons are both learning units. Then, it would be possible to have more general (simpler) learning units on the top of the hierarchy, such as neurons, and more specific and complex units behind, such as ANN models.

3.3.5 Final Remarks

This chapter introduces related work that also comes up with generic ANN software solutions. The main focus here was to understand the design choices made by the authors on modeling an object-oriented ANN system. Thus the result of the evaluation of these related works was the possibility to mature a design where good and bad aspects of those related works were considered. The list below points out some relevant aspects of them that were considered in the design of CANN:

- The design should reflect the problem domain.
- As performance is an important aspect in an ANN simulation, the classes should not include extra information that make them unnecessarily big in terms of memory footprint and also they should be optimized in terms of code implementation.
- Allocation and deallocation of memory in C++ can be avoided using languages such as Java and Smalltalk. The drawback certainly is less performance.
- It is an important feature that the ANN structure can be made persistent.

- There must be a way of implementing training set classes to make the integration of the ANN with the learning and testing data easier.

- Neurons and synapses classes are the basic building blocks for constructing object-oriented ANN solutions.

- None of the discussed work considers the separation pattern. In general, the code reusability and flexibility is given by subclassing and overriding methods.

- None of the discussed work takes care of parallel and distributed solutions.

CANN considers all of the above in its design and implementation. For a future version of the CANN implementation, several ideas can be reevaluated as already cited along the text.

3.4 Conclusions

This chapter illustrates how the uncompromising application of framework technology leads to the construction of ANN software with appropriate flexibility. The implementation of such a framework corroborates that a sound object-oriented design of neural network components delivers the expected benefits, which could not be provided by a conventional solution.

An important goal of the component framework development was its usefulness in different application domains and reusability for different tools. The CANN components have been used for different purposes in different systems as expected. An early version of the CNM ANN component was applied to perform credit rating for the Quelle retail company. The very good results achieved in this work were one of the main motivations for constructing the CANN simulation environment. The numerical results of this work are not public domain.

The developed ANN framework, specially the optimized CNM component (see Chapter 6) was the base for the implementation of the AIRA data mining tool (http://www.godigital.com.br). AIRA has been applied mainly in the area of personalization of web sites.

The CANN simulation environment has been applied to weather forecast. In the work described in (da Rosa et. Al, 2001a and 2001b), it is applied to rare event weather forecasting at airport terminals. Rare events are difficult to forecast because, by definition, the experts do not have extensive experience with such events. Those events may make it difficult for an aircraft to land or take off, causing many problems like traffic controllers acting under stress situations, the aircraft having to land at another airport, etc.

Even in an early stage, the CANN simulation environment has been used as a simulation tool for implementing a VMI solution (Vendor Managed Inventory) for an E-Business company dedicated to implementing B2B solutions for supermarkets and its suppliers (http://www.mercador.com).

The framework design has provided flexibility and reliability to those systems. The CANN framework components expanded from a classificatory system with only one learning algorithm to the possibility of implementing many different learning algorithms. The design permits the straightforward application of the different ANN models to different ANN domain problems. Different data sources are easily coupled with the domain problem at hand and applied to the ANN learning and testing processes. The design also allowed the framework to add other implementation facilities such as parallelization and distribution. This plays an important role to overcome the limitations of hardware that the ANN learning may face. The Java implementation also permits the necessary platform independence.

The current design and implementation of CANN may be considered as a generic decision-making system based on neural networks. An ambitious goal would be to enhance the framework further, so that other decision support problems, like forecasting, can be supported. Also ambitious would be to allow the implementation of other learning mechanisms that do not rely only on neural networks, such as machine learning algorithms.

Currently, most excellent frameworks are products of a more or less chaotic development process, often carried out in the realm of research-like settings. In the realm of designing and implementing CANN *hot-spot-analysis* was particularly helpful for coming up with a suitable framework architecture more quickly. An explicit capturing of flexibility requirements indeed contributes to a more systematic framework development process.

4 ANN Parallel Implementation

One of the main concerns of implementing ANN is to take care of performance aspects. The software solution shall be carefully developed in the sense that it does its best to guarantee the ANN performance while performing learning and testing tasks. To improve the ANN performance, it is necessary to use the hardware platform as much as possible. The first step in this direction in the realm of the CANN project was to implement parallelism in the neural network learning and testing mechanisms. With such an implementation, machines with more than one processor (for instance 2 to 4 processors) improve the network performance.

The parallel program design in ANN is an important aspect to be considered while developing a software solution such as the CANN framework. Thus, the goals of exploring parallel software implementation in this work are twofold:

- To explore the possibility of having a generic parallel solution for the CANN framework.
- To propose and implement a parallel solution for the CNM.

The first goal is focused on exploring the state of the art in parallel implementations for ANN in order to understand the best solution for the simulation environment as a whole. The second goal is specific to the CNM which is an ANN model were parallelism still was not explored in detail, being a contribution to this model state of the art.

4.1 Introduction

The first step, in order to better understand how to implement parallel ANN solutions, is to take a look at structuring approaches for implementing parallelism in ANNs. Kock (1996) proposes how to break the ANN structure from large- to fine-grained pieces in order to run them in parallel:

- **Training session parallelism** – Train a given ANN simultaneously with different learning parameters in the same training examples. Typically different sessions are placed on different processors.

- **Training example parallelism** – Implementation of simultaneous learning in different training examples within the same session. A training set is split into a number of subsets and the corresponding number of network instances are trained simultaneously. The weights of each network instance are accumulated separately and at the end of the process they are brought together in one network. The different training subsets are distributed on the different processors.

- **Layer parallelism** – It provides concurrent computation for layers. Layers are pipelined so that learning examples are fed through the network in a way that each layer works in a different training example. The different layers are distributed on the different processors.

- **Node parallelism** – The ANN neurons perform weighed input summation and other computation in parallel.

- **Weight parallelism** – It refines node parallelism allowing the simultaneous calculation of each weighed input. This form of parallelism can be implemented in most of the ANN models.

- **Bit serial parallelism** – Each bit of the numerical calculations is processed in parallel. It is a hardware dependent solution.

Implementing a generic solution for the CANN framework means that any implemented ANN shall naturally have a parallel solution. The mentioned implementations shall be considered in a generic solution.

In general, there are four ways of implementing ANN algorithms: adapting or extending a preprogrammed simulator; developing the solution from scratch using a general purpose language; developing the solution using a ANN library based on a general-purpose language; and using an ANN specification language. Simulators in general lack flexibility and extensibility in terms of ANN parallel implementations so that the developers have to make their hands-on implementation. Parallel implementations of ANN simulation kernels are rare because a parallelization of the complete kernel is difficult. When doing so, the kernels have restricted functionality being applied to specific ANN architectures or models.

In the early nineties a major challenge of ANN development was to achieve maximum performance on parallel machines performing ANN tasks. The common strategy of these developments on general-purpose parallel computers (SIMD architectures like MasPar and MIMD architectures like IBM SP-2) was to speed up the processing using special characteristics of the target ANN architectures. That means implementing specific solutions to specific ANN models from scratch. Special-purpose parallel neurocomputer architectures like CNAPS (Hammerstrom, 1990) or Synapse-1 (Ramacher, 1992) were also used. Such implementations were done using machine-dependent languages or libraries. The programmer is responsible for choosing how to partition the ANN structures in order to better use the available processors. This may not be optimal and may collapse if any ANN architectural change has to be done.

Another alternative is to use libraries for the simulation of ANN like SESAME (Linden et al. 1993), MUME (Jabri et al. 1993) or PDP++ (Dawson et al. 1997). Such libraries are based on general-purpose languages and contain facilities for constructing ANN architectures and simulate them. They are neither suited for neurocomputers nor for parallel computers due to the underlying sequential programming language.

Trying to avoid this, some specific languages for building ANN solutions were created. One example is CuPit (Prechelt, 1994). This language was specifically designed to express neural network learning algorithms. Its programs can be compiled into efficient code for parallel machines. Its advantages in terms of software design are the high expressiveness, clarity and ease of programming for implementing ANN learning algorithms. As the language is domain specific, it can result in more efficient code because it applies optimizations unavailable to compilers for general-purpose parallel languages. However, when compared to solutions built in sequential languages such as C/C++, the resulting code may be less efficient. Furthermore, this language does not support characteristics of object-oriented languages such as inheritance and neurocomputers were not supported as target architectures.

Another more recent language for building ANN parallel solutions is the EpsiloNN (Strey, 1999). The EpsiloNN (Efficient parallel simulation of Neural Networks) was built to efficiently simulate ANNs on different parallel computer machines such as SIMD parallel computers and neurocomputers. Object-oriented concepts such as classes and inheritance are applied to describe the ANN architectures. Similarly to the CuPit language, EpsiloNN main building blocks for the ANN construction are neurons, synapses and networks. EpsiloNN does not rely on polymorphism and dynamic binding because they can only be analyzed at runtime and forbid the computation of an optimal mapping at compile-time. The ANN specification is transformed into an appropriate simulation source code. It generates C ANSI code for sequential computers and adequate C dialects for parallel computers or neurocomputers.

Neural networks in general execute few kinds of operations: local operations on neurons or synapses; reduction operations such as summing over all weighed incoming synapses; and broadcast operations such as applying a global parameter to all neurons. Those described operations may happen on local objects such as the neurons or on groups of them such as the incoming synapses of a neuron. Such kinds of operations lead to two nested levels of parallelism: *Weight parallelism* and *Node parallelism*. The already cited languages for implementing ANN algorithms CuPit-2 and EpsiloNN implement *Weight* and *Node parallelism* having achieved very good results. The CANN solution goes in the same direction by implementing a generic solution to *Weight parallelism* that is explained in the next section.

4.2 Towards a generic parallelization of the CANN framework

In the CANN framework, objects such as neurons and synapses are implemented. A natural match to that is to implement weight and node parallelism. In CANN the neurons are objects that act as controllers of the synapses execution. Each neuron requests the execution of the synapses that are associated to it. It performs its own computation only when all the dependent synapses return the results of their own computation. Furthermore, the execution of the synapses objects are clearly independent from each other and from the other objects of the framework being good candidates to be run inside a thread. In fact, the synapses are responsible for the weighed input calculation, being already prepared for implementing the *Weight parallelism*. Complementarily, the neuron object is a good candidate to group and coordinate the execution of the synapses threads.

Figure 4.1 below shows a CNM. Each synapse is drawn with a different line type representing one different running thread. During the CNM execution, the lower synapses that form one combination (leads to one combinatorial neuron) can run in parallel. The upper synapses executions also are completely independent from one another and are performed in parallel by independent threads coordinated by the hypothesis neurons.

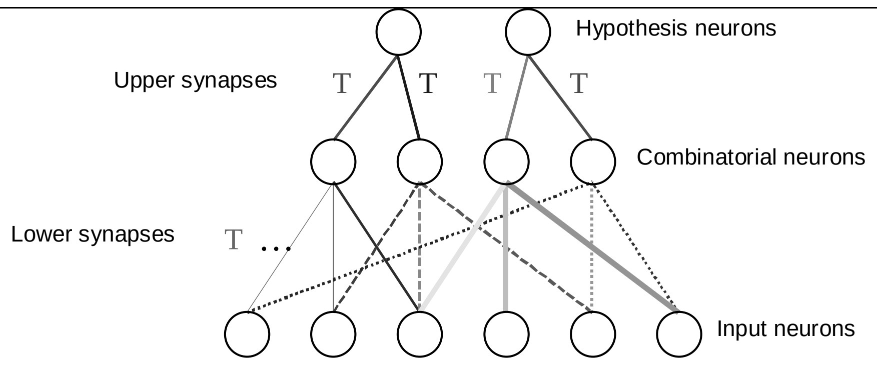


Figure 4.1 - Threads on CNM, each synapse becomes a thread

Such a solution can be applied to different ANN models by simply isolating the synaptic computations that are necessary to perform the computation of one neuron. The four ANN models implemented in this thesis could incorporate this solution (e.g. CNM, Backpropagation, SOM and ART1). However, in practice, tests of such implementation with the CNM model proved that this solution is too fine grained because a large number of threads with short processing times may be generated. The generation of an excessive number of threads turns the ANN performance to an unacceptable level and makes its management sometimes impossible at the level of operating system capabilities.

Implementing *Node parallelism* could be a good alternative as a general solution for the CANN framework, but clearly the problem of generating a huge number of threads still can happen, depending on the implemented ANN and domain problem.

Weight and *Node parallelism* led to good results on the already referred CuPit-2 and EpsiloNN languages certainly because they are capable to generate optimized code for parallel machines that are able to run efficiently such fine-grained operations contained on the synapses and neurons calculations. Such a solution in a general-purpose language like Java turned out to be inefficient. It is known that Java imposes a significant overhead on the threading execution, so that a comparison of the CANN solution and the other cited general-purpose solutions are unfruitful. Furthermore, the different target hardware of the solutions makes such a comparison improper.

The natural consequence of this first experiment is to look for more large grained parallel implementations. The CANN framework does not implement the concept of layer so that to implement a solution such as *Layer parallelism* is neither practical nor intuitive. It would be necessary to make changes to the inner code of the framework in order to implement it.

However, the implementation of layers as exposed ANN structures shall be done in future evolutions of the CANN framework.

A general solution for parallelism on the CANN framework succeeds at the level of *Training session parallelism*. This approach was used to implement the CANN simulation characteristic of having more than one ANN instance running at the same time. The next section explains how this solution was implemented and used in the CANN simulation environment.

4.2.1 The CANN parallel implementation

The CANN simulation environment allows the execution of several ANN instances at the same time (*Training session parallelism*). The ANN's are started in parallel inside separate threads. The termination of the execution of each ANN is independent from each other. The CANN simulation environment allows the parallel execution of different ANN instances from different ANN models.

The parallel implementation was done at the level of the neural networks manager class. The network management classes are subclasses of the abstract class *NetManager* that can be partially seen in Code 4.1 below. This class predefines that any of its subclasses implements the interface *Runnable* that means they can start thread executions. It also defines that its subclasses shall implement a method called *netManagerStarter* that shall be used to start the ANN learning inside a thread called *netLearnThread*. Code 4.1 also shows part of the class *CNMManager*, where the method *netManagerStarter* is implemented and the thread is created and started. The method *run* is also shown where the inner execution of the thread is defined. The variable *threadBody* defines what method to call inside the created thread defining the type of execution the ANN must run. The class may run the network inside a thread for the normal CNM learning algorithm, the optimized learning algorithm or even for the testing of the already trained network.

Code 4.1 – Parallel implementation

```
public abstract class NetManager extends Object implements Serializable, Runnable {
  //…
  transient Thread netLearnThread;
  abstract public void netManagerStarter();
  //…
}

public class CNMManager extends NetManager {
  void netManagerStarter() {
    netLearnThread = new Thread(this,"Thread Learn");
    netLearnThread.setPriority(parameters.thePriority);

    if (netOptimization == true) {// If Optimized learning
      // sets optimized learning parameters
      // …
    }
    else {
      // sets original learning parameters
      // …
    }
    netLearnThread.start();
  }

  // calls the appropriate simulation method
  public void run() {
    switch (threadBody) {
      case CONSULT_CASEBASE: // Consulting Case Base
        runConsultCaseBase();
      break;
      case STARTER_LEARN: // Learning Case Base
        runLearnCaseBase();
      break;
      case OPTIMIZED_LEARN: // Optimized Learning Case Base
        runOptimizedLearnCasebase();
      break;
    }
  }
//…
}
```

The called method inside the *run* method is responsible for calling the appropriate *INetImplementation* instance to execute its tasks. In this way, each ANN instance in the CANN framework can run inside an independent thread. Figure 4.2 sketches the implementation framework.

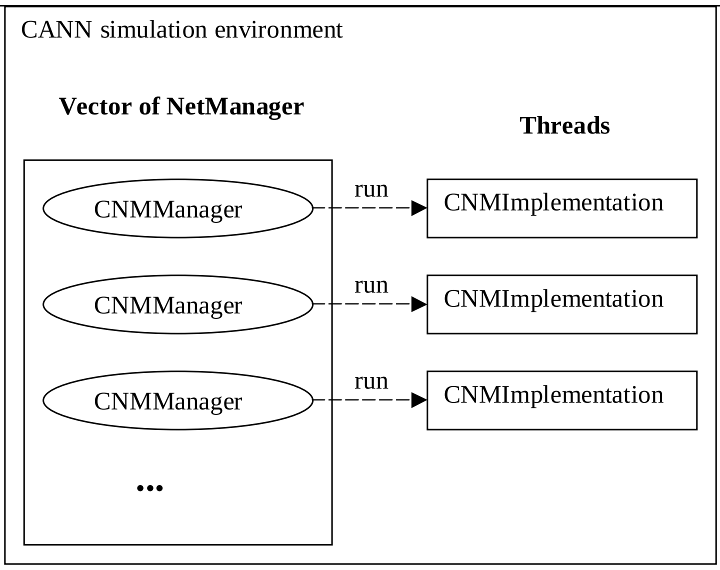


Figure 4.2 – The parallel architecture solution

The CANN simulation environment allows the user to create many ANN instances at the same time grouping them in a *Vector*. The user can start the execution of the ANN at any time.

4.2.2 CANN parallel solution test results

The main goal of the tests is to verify check whether the fact of having different instances of ANN's running at the same time will influence the ANN performances. The proposed test measures the performance of two different ANN instances when running alone and when running together, sharing the machine resources. The same test was performed on different machines with one and two processors, to verify the behavior of the solution when the CPU is shared and when two CPU's can be allocated. The first machine is a Pentium III 550 MHz with 256 Mb of RAM and the second one is an IBM Netfinity 3000 with 2 Pentium III 667 MHz processors and 512 Mbytes of RAM.

The two selected nets were the Backpropagation and the SOM. The time each network took to perform its learning process was measured. The Backpropagation network ran 10000 epochs to learn the XOR problem and the SOM ran 5000 epochs to learn the Bi-Dimensional problem. The start of the learning process is manual for both networks using the CANN GUI. Therefore, one shall be started before the other. The SOM network was started first.

Table 4.1 - Networks running on a machine with one CPU

Network	Standalone	Parallel
BP	14843 ms	20812 ms
SOM	10750 ms	21429 ms

Table 4.1 shows the results when the tests where performed on the single CPU machine. The column *Standalone* shows the time average the networks took to perform the learning running each one in a different time frame, not competing for the CPU. The column *Parallel* shows the time average when they were executed in parallel. The results show that when running in parallel both networks spend more time to perform the learning individually, however, as they are running together, the total time is simply the time the last net took to learn. When running them separately the time of both shall be summed. The Figure 4.3 shows the difference of running the ANN sequentially and in parallel. Running the Backpropagation and the SOM separately would take on average 25593 ms (14843 ms + 10750ms), while running together it would take on average 21429 ms (time SOM took to finish, the BP certainly had finished before). Thus, it is worth running the networks in parallel even with a one CPU machine.

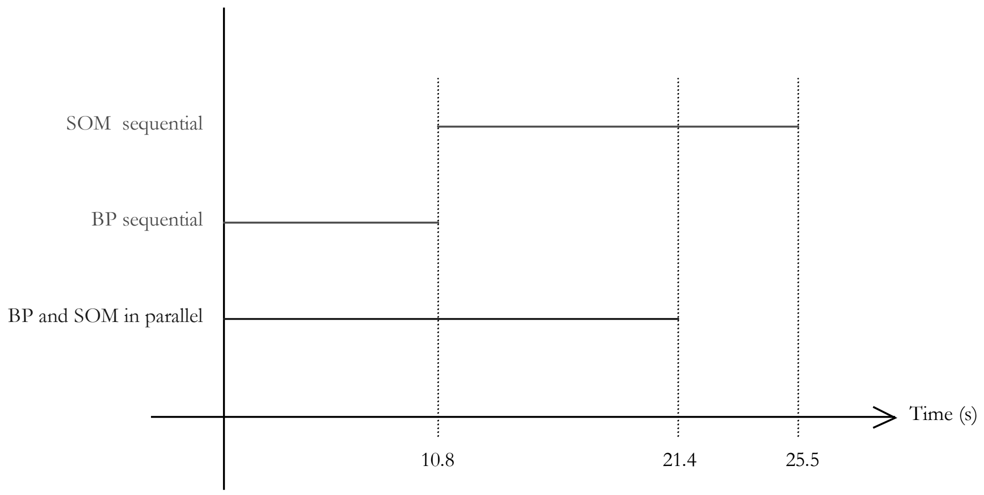


Figure 4.3 – The time difference between running in parallel and sequentially

The Speed-up (Sp) calculus (Hwang & Xu, 1998), reinforces this conclusion. It is a simple acceleration factor that is given by the reason of the sequential time (st) by the parallel time (pt) as can be seen on Equation 4.1 below.

$$Sp = \frac{st}{pt}$$

Equation 4.1 – Speed-up

Taking the sequential time as the addition of the sequential times for the Backpropagation and SOM execution (25593 ms) and for the parallel time the SOM time (21429 ms), the Speed-up will be 1.19 (Equation 4.2). It means that running the two networks in parallel it will be 1.19 times faster or 19% faster.

$$Sp = \frac{14843 + 10750}{21429} = 1.19$$

Equation 4.2 – Speed-up for running BP and SOM in parallel in a single CPU

For the machine with two CPU's, more tests were performed. The first test was to verify the performance of the Backpropagation running standalone and with two instances at the same time. Table 4.2 shows the standalone performance average of the Backpropagation.

Table 4.2 – Backpropagation running standalone in a 2 CPU's machine

Network	Standalone
BP	5500 ms

Table 4.3 shows the performance when two instances of the Backpropagation run during 10000 epochs on the 2 CPU's machine.

Table 4.3 – Two Backpropagation instances running in parallel in a 2 CPU's machine

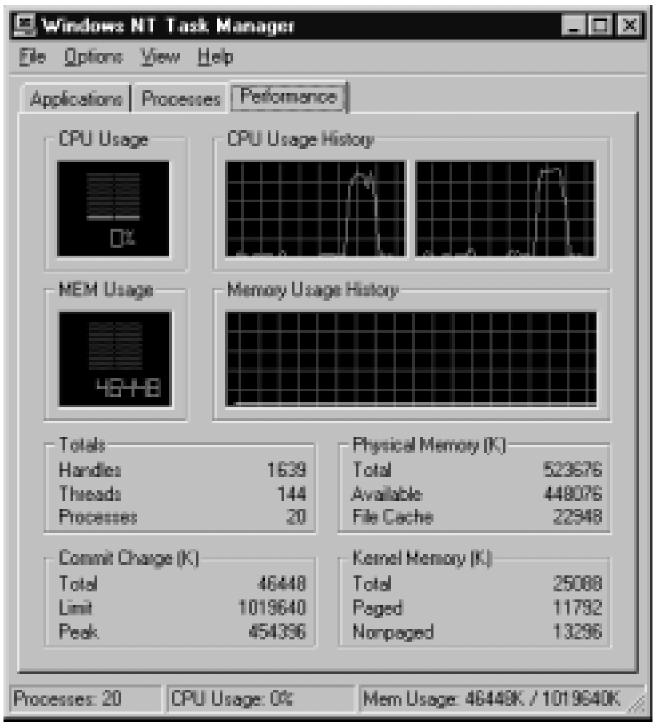
Network	Parallel
BP 1	8367 ms
BP 2	8586 ms

Once again it was worth running two ANN's at the same time, considering that, on average, running two instances of Backpropagation at the same time is faster than running them in sequence. The Speed-up for this execution is given in Equation 4.3 below. The sequential time is given by the execution of two Backpropagation simulations sequentially and the parallel time is given by the longest execution of the two parallel Backpropagation simulations. The Speed-up result is that the parallel execution is 28% faster than the sequential one.

$$Sp = \frac{5500 + 5500}{8586} = 1.28$$

Equation 4.3 – Speed-up for running BP in a 2 CPU machine

Figure 4.4 shows the processors being allocated to perform the execution of the two Backpropagation simulations in parallel. The two CPU's are nearly 100% allocated during the simulation period.



Totals
Handles 1639
Threads 144
Processes 20

Physical Memory (K)
Total 523676
Available 448076
File Cache 22948

Commit Charge (K)
Total 46448
Limit 1019640
Peak 454396

Kernel Memory (K)
Total 25088
Paged 11792
Nonpaged 13296

Processes: 20 CPU Usage: 0% Mem Usage: 46448K / 1019640K

Figure 4.4 – Two CPU's running two Backpropagation instances in parallel

Table 4.4 below shows the average time for running the learning of one instance of the SOM network in the machine with two CPU's. Table 4.5 shows the average time two SOM instances take to run in parallel on the same machine.

Table 4.4 – SOM running standalone in a 2 CPU's machine

Network	**Standalone**
SOM	8555 ms

Table 4.5 – Two SOM instances running in parallel in a 2 CPU's machine

Network	Parallel
SOM 1	8974 ms
SOM 2	8760 ms

There is no significant difference between running the SOM as a standalone learning process or running two learning processes at the same time. The Speed-up for this execution is given in Equation 4.4 below.

$$Sp = \frac{8555 + 8555}{8974} = 1.90$$

Equation 4.4 – Speed-up for running SOM in a 2 CPU machine

This clearly shows the significant advantage of having such a parallel solution for running the SOM learning. During the same time frame two networks can be learned instead of one, the parallel solution for the SOM simulation is, on average, 90% faster than the sequential one.

Table 4.6 shows the learning time when two instances of different ANN's run in parallel on the machine with two CPU's. The Backpropagation instance took a little more time than when running standalone and the SOM instance once again presented a similar performance average.

Table 4.6 – Backpropagation and SOM instances running in parallel in a machine with two CPU's

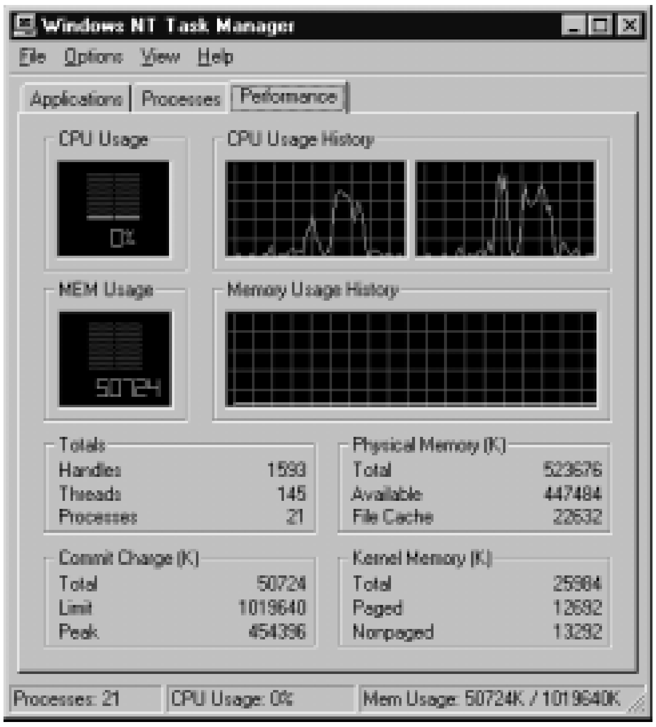
Network	Parallel
BP	6510 ms
SOM	9291 ms

The results confirmed that it is worthwhile simulating different ANN models at the same time on the same machine. The Speed-up for this execution is given in Equation 4.5 below. The parallel execution was 51% faster than running sequentially.

$$Sp = \frac{5500 + 8555}{9291} = 1.51$$

Equation 4.4 – Speed-up for running BP and SOM in a 2 CPU machine

Figure 4.5 below shows the allocation of the CPU's during the Backpropagation and SOM instances learning.



```
Totals                      Physical Memory (K)
Handles          1593       Total          523676
Threads           145       Available      447484
Processes          21       File Cache      22632

Commit Charge (K)           Kernel Memory (K)
Total           50724       Total           25984
Limit         1019640       Paged           12692
Peak           454396       Nonpaged        13292

Processes: 21    CPU Usage: 0%    Mem Usage: 50724K / 1019640K
```

Figure 4.5 –Two CPU's running Backpropagation and SOM instances in parallel

It is important to note that running more than one ANN in parallel inside the CANN simulation environment may lead to performance bottlenecks. Depending on the number of nets running simultaneously and the size of those nets, the machine resources can be dried out fast.

CANN runs in one Java runtime, receiving a main controller process. The threads created for the ANN's will allocate the resources of this process. For each created ANN instance, its threads will make use of the same memory and CPU footprint that was allocated for the other ANN instances inside the same process. So it is important, when running more than one ANN using CANN, to clearly understand the ANN necessities of CPU and memory footprint as well as the machine resources and Java runtime issues. It may be better to run some ANN instances on separate machines to avoid competing for the CPU and memory given the number of CPU's and threads to create and the available memory. Further tests could be elaborated and performed to measure such situations in order to map the exact implications of running parallel ANN's in the CANN simulation environment.

To be able to extend the possible resources for running the parallel ANN instances an extension of this general solution is presented in Chapter 5 where the *Training session parallelism* allows the implementation of parallelism by distributing the different ANN models on the networked computers for learning and testing processes.

The solution presented here is general enough as a good solution for the CANN framework. However, it does not take into consideration the parallelism inherent to the specific ANN models. As a general solution is difficult at such a level, as already explained, it is important to do some investigation on how to implement the parallelism at least for the CNM model, which is the model where parallelism could lead to significant gains due to the inherently structure of CNM nets. Such an investigation can also help evaluate the CANN architecture as well. Next section explores the possibilities for implementing parallelism for the CNM.

4.3 A parallel solution for the CNM

The CNM model is a CPU intensive network. Depending on the problem domain it is modeled for and the chosen combination order the network can have an enormous size and its learning time can take days. That's why it is important to invest time in the optimization of the CNM learning time. Fortunately the CNM has an inherent parallel structure, though implementing parallelism is quite straightforward.

Figure 4.6a shows a first approach to separate the CNM network into parts that can run in parallel. This subdivision is based on the fact that the whole network combinatorial structure is independent for each Hypothesis (upper neurons). Only the input neurons are shared resources that shall be synchronized. To this approach, one thread is created to manage the learning process of each Hypothesis Neuron instance. The set of combinations that forms the hypothesis is run inside the thread.

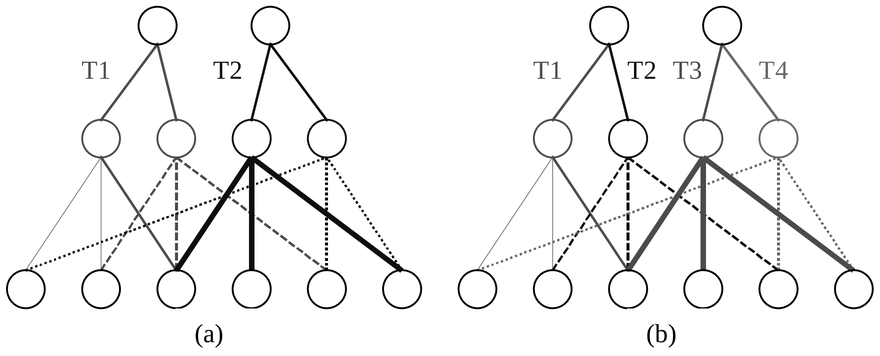


Figure 4.6 - Using threads on CNM

A natural evolution to this solution can be seen in Figure 4.6b where more than one thread can be defined for each hypothesis. Each combination that forms the set of combinations for each hypothesis is also independent of each other. As a consequence, the set can be subdivided into subsets of combinations. Each subset can run as an independent thread. Inside a subset, the combinations are evaluated in sequence. The user can define the number of threads to perform the set of combinations (number of subsets). This number cannot be bigger than the number of combinations of the set.

The solution proposed above can be considered as a variation of the *Training example parallelism*. The training set was not subdivided into subsets but the CNM was split into many parts where each part learns separately to be glued together at the end of the learning process. Constructing separate instances of parts of the CNM network is appropriate once the execution of each ANN combination is independent of each other. The constructed solution can also be considered a variation of the *Node parallelism* because different neurons in the same level run in parallel. Each neuron is used as a coordinator for the running of the synapses inside independent threads. However, the threads are created only at the level of the hypothesis neurons and not at the level of combinatorial neurons.

The implemented solution scales well. It optimizes the use of threads according to the size of the combination layer and the machine capabilities. In the next section, the implementation of this solution is explained in detail, and its results are also extensively evaluated in order to show that this is an appropriate approach to implement parallelism for the CNM model.

4.3.1 CNM Parallel implementation

The solution shown in Figure 4.6b and explained above can be implemented on the CANN framework in a systematic way. The object-oriented architecture and the implementation in the Java language facilitate the use of threads. The proposed solution implements the *Group Proxies* (Lea, 1999) design pattern that is applied for implementing partitioned concurrent activities in Java. A group consists of all members of some arbitrary set, for instance a set of CNM combinations. Group Proxies are protocol adapters that manage multiple threads controlled by multiple objects; for instance, the hypothesis neurons control the threads that contain the sets of combinations.

The Group-based design makes it easy to increase parallelism transparently when there are multiple CPU's, and whenever there is a good reason to partition a problem into parts that can be run concurrently. That is exactly the case of the CNM model. Group Proxies consider complex features such as the controlling of adding or removing members of the set that are not used in this work.

The Group Proxies pattern implements the design necessary to perform multithreaded delegation, also known as a form of scatter/gather processing (Lea, 1999). Figure 4.7 shows the interaction diagram of the Group Proxy pattern and Code 4.2 an algorithm for implementing it. The core algorithm logic is implemented in a method called *op()* that implements the scatter and gather parts of the thread control. The scatter part of the code is used to start the necessary threads and the gather shall make the execution join of the started threads. The gather part is also responsible for collecting results that will be returned by the *op()* method.

Code 4.2 - Group Proxies algorithm

```
public interface AnInterface {
      public ResultType op(ArgType arg);
}

class GroupProxy implements AnInterface {
      public ResultType op (ArgType arg) {
      // “Scatter” phase
      split the problem into parts;
      for each part {
            start up a thread performing its actions;
      }

      // “Gather” phase
      wait for some or all threads to terminate;
      collect and return results;
      }
}
```

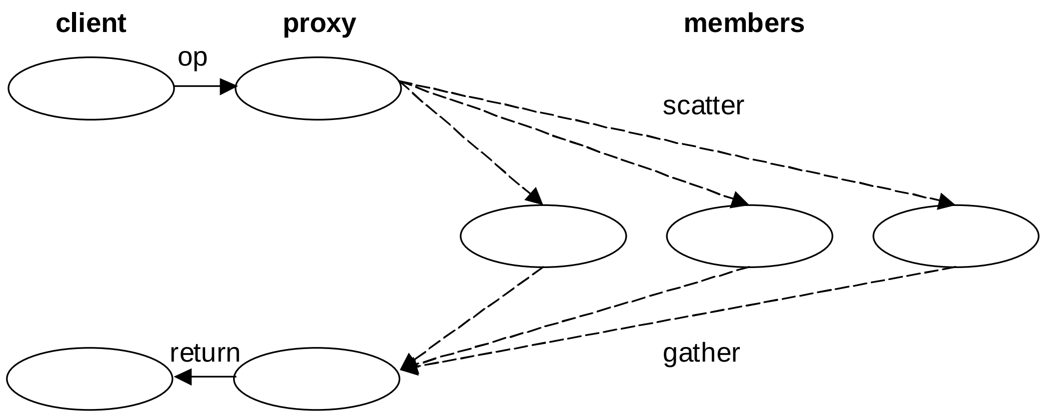


Figure 4.7 - Group Proxies interaction diagram

Code 4.3 sketches the *computeEvidentialFlow* method from the *CNMImplementation* class. The method implements the choice of performing the CNM learning or testing processes (evidential flow) serially or in parallel. The learning and testing processes are implemented in the *CNMManager* class. In case of running serially, for each hypothesis neuron the method *serialStartEvidentialFlow* of the class *CNMHypothesisNeuron* is called. The evaluation of each hypothesis is performed in sequence. The evaluation of each combination inside the set of combinations of each hypothesis is performed in sequence as well.

When running in parallel, the method *startEvidentialFlow* of the class *CNMHypothesisNeuron* is called for each hypothesis neuron. This method creates threads for running the evidential flow of the hypothesis neuron. The created threads are returned in a vector. The method *computeEvidentialFlow* is responsible for performing the *join* of all threads started by each hypothesis neuron.

The method *computeEvidentialFlow* implements the Group Proxy pattern by implementing the scatter and gather phases. When the method calls the *startEvidentialFlow* of the class *CNMHypothesisNeuron* it is controlling the starting of the parallel threads execution, implementing the scatter phase. When it makes the join of the started threads it is implementing the gather phase. The gather phase of the *computeEvidentialFlow* implements an AND termination where all the threads are waited to terminate. Besides this, it does not return any result because they are already contained on the *CNMHypothesisNeuron* class that implements the threads execution and can be accessed by the caller class *CNMImplementation*.

Code 4.3 – The *computeEvidentialFlow* method

```
/*
* Implements the calling of the evidential flow for hypothesis neurons
*/
void computeEvidentialFlow() {
  // compute the evidential flow = propagate evidences
  if (!parallelExecution) {
    for each hypothesisNeuron do {
      // sets initial variables
      //…
      hypothesisNeuron.serialStartEvidentialFlow(parameters);
    }
  }
  else {
    // create threads for running sets of combinations
    // scatter phase
    Vector hypothesisThreads = new Vector();
    for each hypothesisNeuron do {
      // sets initial variables
      //…
      hypothesisNeuron.startEvidentialFlow(parameters);
    }
    // make the join for all hypothesisNeurons Threads
    // gather phase
    try {
      for each hypothesisThread do
        hypothesisThread.join();

    }
    catch (InterruptedException ex) {
      for each hypothesisThread do
       hypothesisThreads.stop();
    }
  }
}
```

Code 4.4 was picked from the *CNMHypothesisNeuron* class. It shows the *startEvidentialFlow* method. The set of combinations associated with the hypothesis neuron is divided into subsets. Each subset is encapsulated in a thread that is created and started.

Code 4.4 – The *startEvidentialFlow* method

```
/*
* Computes Fuzzy And (minimum) or Fuzzy OR (maximum)
*/
Vector startEvidentialFlow(CNMHypothesisNeuron upNeuron, Vector auxParameters) {
  // sets initial variables
  //…
  neuronThreads = new Vector();

  if (incomingSynapses.size()<hypothesis.getNumberOfThreads())
    hypothesis.setNumberOfThreads((short)incomingSynapses.size());

  // gets the rest of the division; if odd the last loop must have one more element
  lastLoopRest = incomingSynapses.size()%hypothesis.getNumberOfThreads();
  loopSize = (int)incomingSynapses.size()/hypothesis.getNumberOfThreads();

  // starts the threads based on the number of threads set by the user
  // each thread controls the execution of a set of combinations
  for (int i=0; i<hypothesis.getNumberOfThreads(); i++) {
    // wait for variables loopBegin and loopEnd in use by the previous thread
    while (loopNotConsumed) {};

    neuronThreads.addElement(new Thread(this, "Thread "+hypothesis.getName()+"->"+i));
    loopNotConsumed = true;
    loopBegin = loopSize*i;
    if ((i == (hypothesis.getNumberOfThreads()-1)))
      loopEnd = (loopSize*(i+1)) + lastLoopRest;
    else
      loopEnd = loopSize*(i+1);
    ((Thread)neuronThreads.elementAt(i)).setPriority(5);
    ((Thread)neuronThreads.elementAt(i)).start();
  }
  return (neuronThreads);
}
```

4.3.2 CNM parallel solution test results

Table 4.7 shows the resulting tests of this parallel solution. The tests were performed on an IBM Netfinity 3000 with 2 Pentium III 667 MHz processors and 512 Mbytes of RAM. The CNM was configured to run the credit analysis problem with order 4 and without optimizing the learning. The learning parameters were acceptance threshold 0.6 and pruning threshold 0.4.

The network performance was tested first without any thread so that the neural network learning process was performed serially. The other four tests are parallel executions with 2, 4, 8 and 16 threads. Each configuration was tested three times and the time and memory usage averages can be seen in Table 4.7. The speed-up results of the parallel executions are also shown in Table 4.7 in order to clearly show when the parallel solution is desirable and when it is not. While the Speed-up is greater than 1.0 the parallel execution has better performance than the serial one.

Table 4.7 – Time and memory results

Net Configurations	Time	Memory	Speed-up

Mono (1)	57.30	31120	1.0
2 Threads (2)	36.68	35888	1.56
4 Threads (3)	49.58	35328	1.15
8 Threads (4)	69.05	35880	0.82
16 Threads (5)	99.84	35652	0.57

The memory usage is nearly the same for all cases. The time to perform the learning was significantly better when using the parallel solution for 2 threads. Using 4 threads the performance is still better. As expected when the number of threads increases, the performance starts to degrade. It can be even worse than running the network serially. Figure 4.8 below clearly illustrates this scenario where the polynomial curve describes the behavior of improving the number of threads when running in the same machine. The reasons for this behavior are twofold: the thread management takes time; and there is a limit for optimizing the performance using two processors. It is not just by chance that the best performance happens when there are two threads running, one in each processor. In fact, the management of many threads without the possibility of executing them in parallel decreases the performance progressively. When there are more active threads than there are CPU's, the Java run-time system occasionally switches from running one activity to running another, which also entails scheduling – figuring out which thread to run next (Lea, 1999).

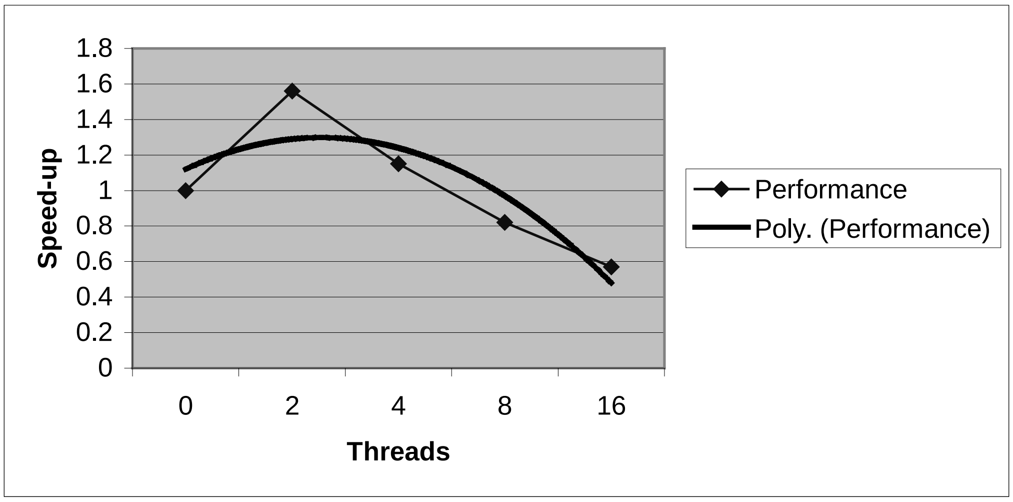


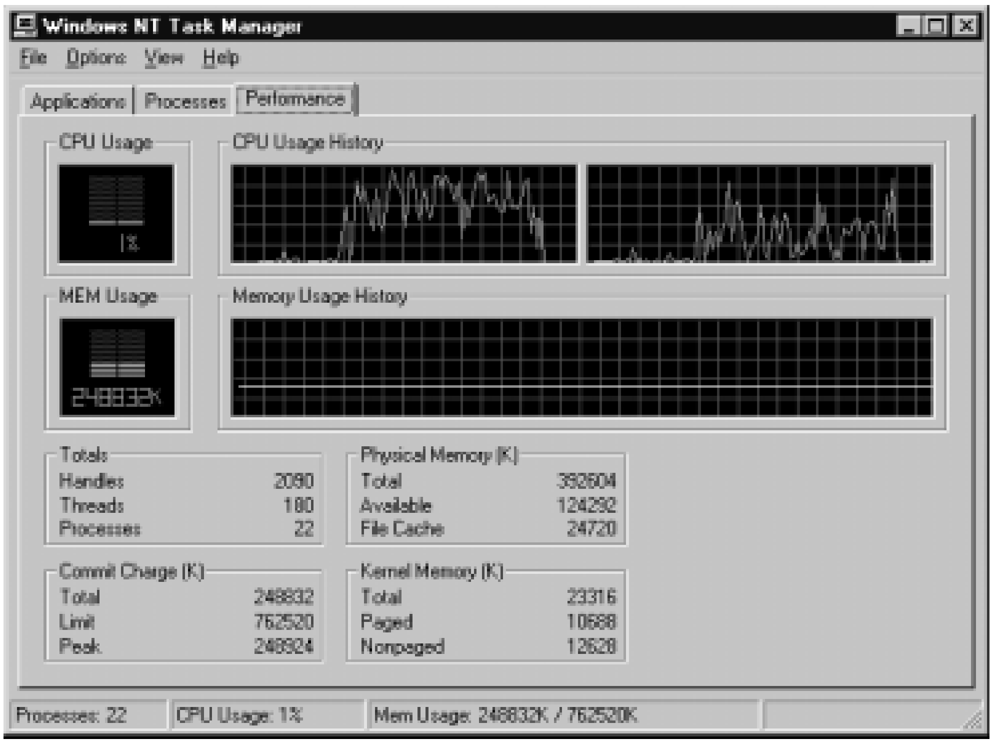
Figure 4.8 – Speed-up for the CNM parallel solution

Figures 4.9 to 4.13 below show snapshots of the Windows Task Manager for each of the tested network configurations. It starts with Figure 4.9 that shows the performance of the ANN when running serially.

4.3.2.1 Test 1 – Running serially

The first processor is responsible for running the ANN. The second processor also spends some time performing other tasks not related to the ANN but to the Java environment control/synchronization. It is clear that the performance of this configuration is not optimal because the processors are not 100% used during the processing time. The system varies the usage of the CPU's very much. It is known that Java methods employing synchronization can be slower than those that do not provide proper concurrency protection. The ANN below runs one thread only, but with synchronization implementation, so that it is not a simple implementation with any concurrency control. Between thread and synchronization overhead, concurrent programs can be slower than sequential ones even if the running computer has multiple CPU's (Lea, 1999).

```
Serial Test
Memory usage average: 31120 Mbytes
Time average: 57.30 s
```



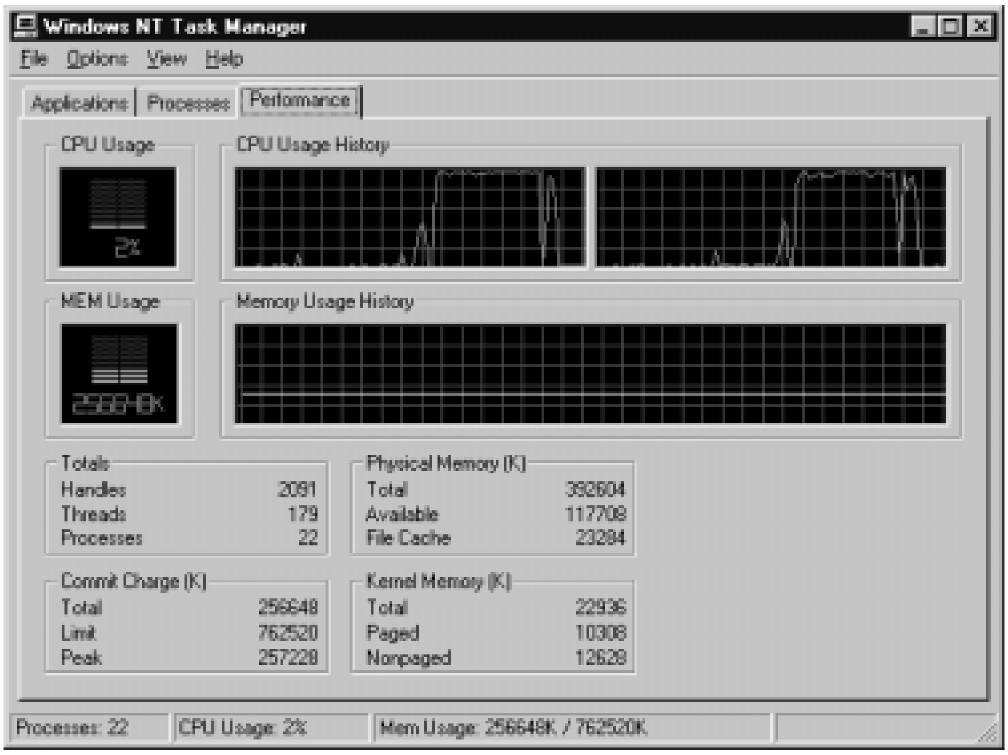
```
Totals                  Physical Memory (K)
Handles       2080      Total         392604
Threads        180      Available     124292
Processes       22      File Cache     24720

Commit Charge (K)       Kernel Memory (K)
Total       248832      Total          23316
Limit       762520      Paged          10688
Peak        248924      Nonpaged       12628

Processes: 22   CPU Usage: 1%   Mem Usage: 248832K / 762520K
```

Figure 4.9 – Time performance for serial implementation

4.3.2.2 Test 2 – Running with 2 threads

The next test configuration shows the results when using 2 threads to perform the learning. Figure 4.10 shows the usage of the two processors during the neural network execution. The execution for both processors is similar, each one taking care of the execution of one thread. The processors spend nearly 100% of its capacity performing the ANN learning during the time they are allocated to do that, in opposite to the serial solution where no processor is allocated to its maximum capacity during the learning process. The first small hill, that is very close to the main plateau and that can be seen in the performance of both processors, is the processing time used for generating the ANN combinations. The main plateau indicates that the ANN is performing the learning and the next small hill corresponds to the ANN pruning.

```
2 Threads
Memory usage average: 35888 Mb
Time average: 36.68 s
```



Totals
Handles 2081
Threads 179
Processes 22

Physical Memory (K)
Total 392604
Available 117708
File Cache 23284

Commit Charge (K)
Total 256648
Limit 762520
Peak 257228

Kernel Memory (K)
Total 22936
Paged 10308
Nonpaged 12628

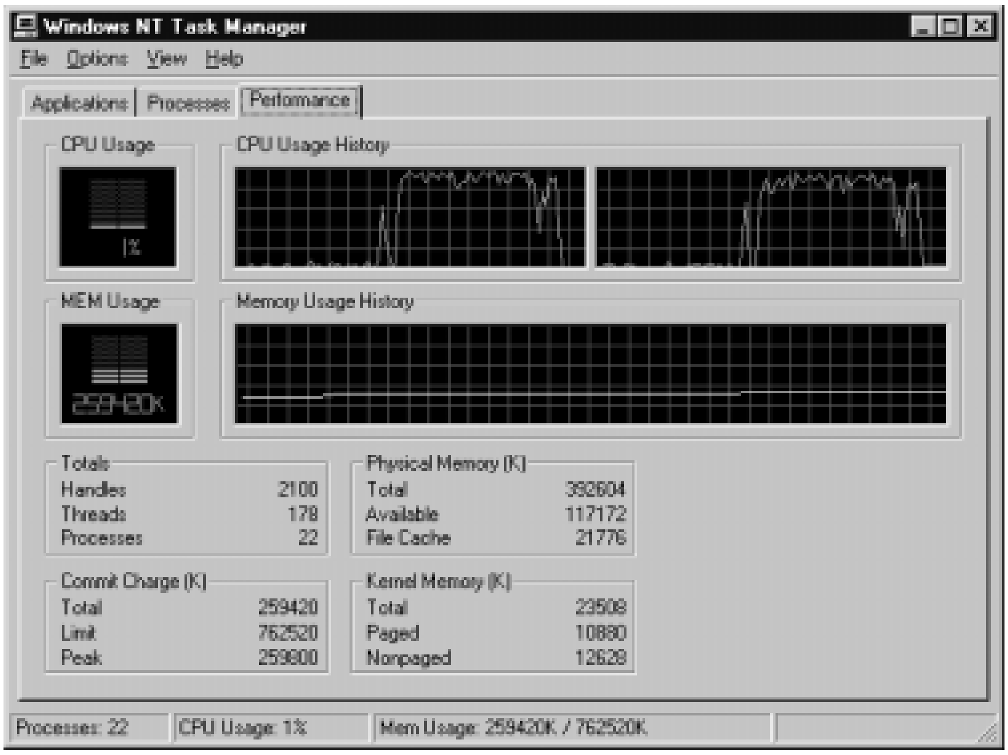
Processes: 22 | CPU Usage: 2% | Mem Usage: 256648K / 762520K

Figure 4.10 – Time performance with 2 threads

4.3.2.3 Test 3 – Running with 4 threads

When running with four threads, the performance decays a little bit. Figure 4.11 corroborates that the processors are not 100% allocated during the learning process as happened on the previous configuration with 2 threads. The reason for that is the time lost with scheduling the threads during the learning process. Each thread receives a time slice, and the change from one thread to another makes the processing time not as optimal as when there are no threads to swap.

```
4 Threads
Memory usage average: 35328
Time average: 49.58
```



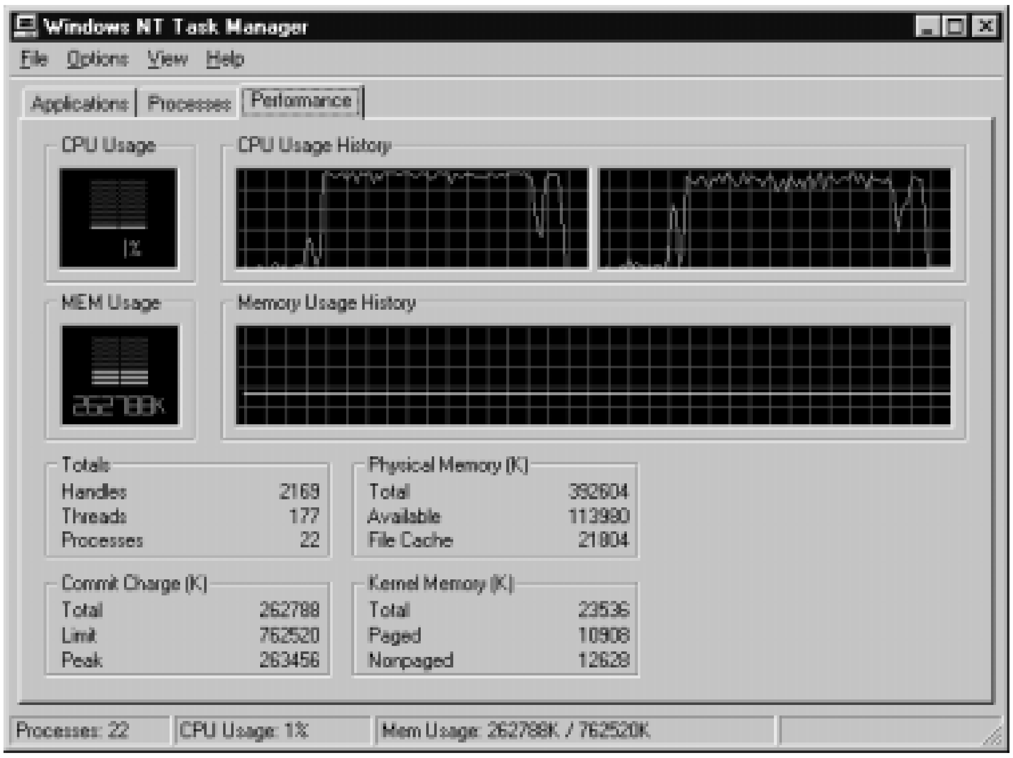
```
Totals                     Physical Memory (K)
Handles       2100         Total          392604
Threads        178         Available      117172
Processes       22         File Cache      21776

Commit Charge (K)          Kernel Memory (K)
Total       259420         Total           23508
Limit       762520         Paged           10880
Peak        259800         Nonpaged        12628

Processes: 22   CPU Usage: 1%   Mem Usage: 259420K / 762520K
```

Figure 4.11 – Time performance with 4 threads

4.3.2.4 Test 4 – Running with 8 threads

Having more than 4 threads does not improve the learning performance for the tested machine as can be seen in Figure 4.12. It would be necessary to have a machine with more processors to allow more threads at the same time to be able to keep improving performance.

```
8 Threads
Memory usage average: 35880
Time average: 69.05 s
```



Totals
Handles 2169
Threads 177
Processes 22

Physical Memory (K)
Total 392604
Available 113980
File Cache 21804

Commit Charge (K)
Total 262788
Limit 762520
Peak 263456

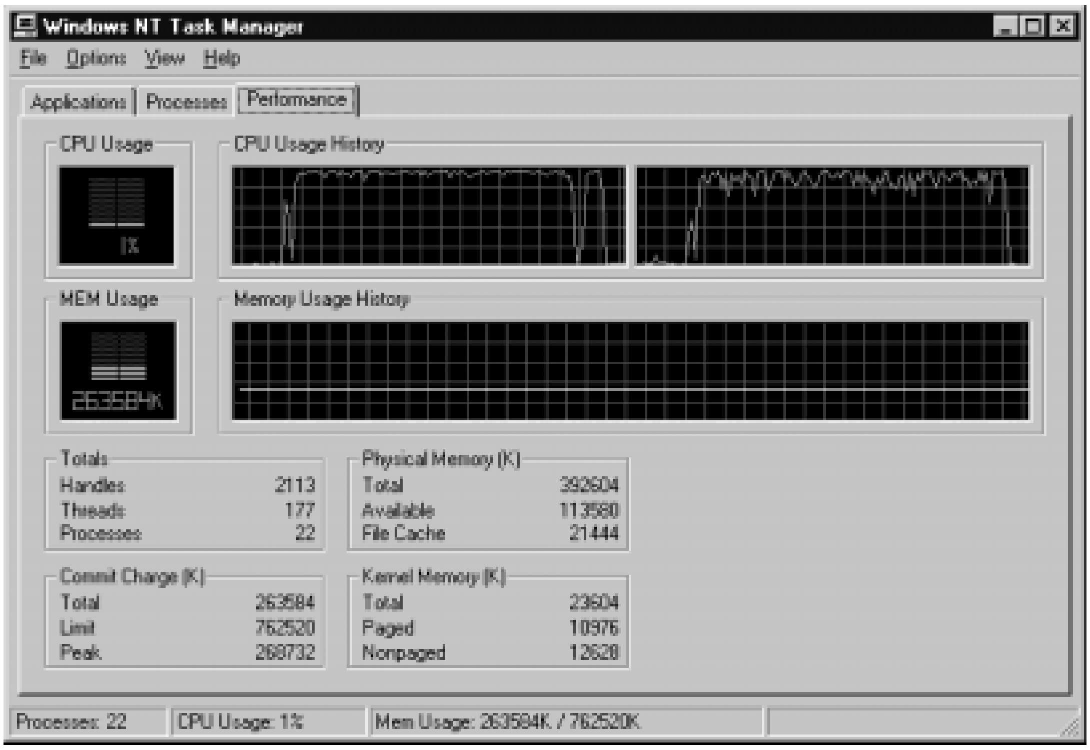
Kernel Memory (K)
Total 23536
Paged 10908
Nonpaged 12628

Figure 4.12 – Time performance with 8 threads

4.3.2.5 Test 5 – Running with 16 threads

After a certain number of threads, the performance is getting worse. The test with 16 threads (Figure 4.13) shows a performance more than 50% worse than the learning without parallelism.

```
16 Threads
Memory usage average: 35652
Time average: 99.84 s
```



1%

263584K

Totals
Handles 2113
Threads 177
Processes 22

Physical Memory (K)
Total 392604
Available 113580
File Cache 21444

Commit Charge (K)
Total 263584
Limit 762520
Peak 268732

Kernel Memory (K)
Total 23604
Paged 10976
Nonpaged 12628

Processes: 22 | CPU Usage: 1% | Mem Usage: 263584K / 762520K

Figure 4.13 – Time performance with 16 threads

4.4 Conclusions

Implementing parallelism in neural networks is not a straightforward task. In general it requires complete control on the ANN software architecture in order to reproduce specific parallel software structures and control mechanisms. Implementing such structures and mechanisms in a pre-defined framework may be difficult because it is necessary to deeply understand the framework in order to get its benefits. Typically, adequate extensions to core classes' become necessary.

The first attempt to implement a generic solution for the CANN framework failed. The implementation of parallelism at the level of each synapses (*Weight parallelism*) proved to be too fine-grained, leading to performance problems. The second solution was to implement the parallelism at the level of *Training session* that means to have different ANN instances running in parallel. This approach was successful showing that more than one ANN can share the CPU resources without degrading its performance. In that way many ANN instances can run in parallel in order to find a solution for a given problem. The number of ANN instances able to run in parallel depends only on the ANN models resource allocation and the available machine resources.

Another successful experiment in implementing parallelism in this work is to consider the parallel solution for a given ANN model, taking into consideration its specific architecture particularities. The architecture of each ANN model defines its possible parallel solutions. As the architectures differ very much from model to model, it is difficult to have a generic parallel solution. However, the CANN framework facilitates the parallel implementation by giving exact entry points for implementing thread control such as the *startEvidentialFlow* and *computeEvidentialFlow* methods. There is a clear separation of ANN architectural parts such as neurons, synapses and management components such as *NetImplementation* and *NetManager*. For instance, the management components implement methods that specifically control the ANN execution for learning and testing processes.

Given those facilities, it is straightforward to implement a parallel solution for the CNM model. It is important to reinforce that this implementation is unique; there are no other parallel implementations for the CNM model so far.

A positive consequence of having a specific parallel solution for the CNM model is that it performs properly, leading to performance improvements. The performance of the ANN execution can be better when running in parallel as shown by the tests. The CNM parallel solution can accommodate the usage of the available hardware resources leading to better hardware usage during the learning and testing processes. It is possible to create an appropriate number of threads in order to get the best results from the number of available CPU's. The drawback is that the solution is specific for this model. It is intrinsically implemented on its architecture and cannot be extended to other ANN models. It would be an important future implementation experiment to add specific parallel solutions to other ANN models inside the CANN framework such as the BackPropagation or SOM.

Complimentary tests were also performed on a 4 processor-machine and the CNM performed appropriately, allocating the 4 processors during learning and testing processes. Those tests were performed only to validate the implementation so that its results are not reported here. The machine was not available to perform long time performance tests in order to evaluate different CNM behavior with different domain configuration and large learning and testing files. In the future, it would be interesting to execute more complete tests when the parallel solution could be tested with different CNM configurations and the learning and testing load could be improved to more complicated application domains with bigger learning and testing sets.

5 Implementing distribution in the CANN framework

The main goal of having a distributed simulation environment for neural networks is to be able to use the networked computational capacity, which means to use the memory and processing capacity of the networked computers. The implementation was motivated by the possibility of optimizing the time spent to perform ANN learning and testing by using the networked capacity. Even being the possibilities of distributing code a very promising alternative to improve ANN simulation results, there is no similar work focusing on implementing distributed solutions for ANNs. Though the results presented along this chapter can be considered contributions to the state of the art in the software development of ANNs.

The simulation of ANN can be a very memory intensive and CPU intensive process. Each created ANN structure can allocate significant amounts of memory and its processing can take CPU hours or even days. Those are the two main reasons to distribute ANN instances. By sending different ANN instances to different machines, the computational capability is multiplied. Therefore, different ANN instances with different configurations can be built and tested in parallel. From an end user point of view, the CANN simulation environment should allow one user to distribute several ANN instances yet controlling them locally.

The implementation of the ANN distribution was built in the CANN framework as a facility for distributing any ANN model implemented in CANN framework. The CANN ANN instances were built as Java objects with mobility capabilities, being able to run in a remote machine.

5.1 Implementing distribution in the CANN simulation environment

This section describes the design and implementation of a software solution for implementing distributed simulation for Artificial Neural Networks (ANN). The CANN framework offers a group of classes that implement various ANN models allowing the simultaneous management of any number of instances of those ANN components. These ANNs being trained or tested at the same time in the simulation environment should be distributed over networked computing devices. In this section the architectural aspects that are relevant for understanding the implementation of the ANN distribution are explained in detail. Once again the software implementation has its distribution solution based on the Objectspace's Voyager library (http://www.objectspace.com).

5.1.1 Choosing the mobile component

One way of distributing ANN computation is to run different ANN models on different machines. The main implementation idea is to move only the core ANN objects in order to have them running on remote machines, allocating the remote machine CPU and memory. The objects that implement the ANN control shall be kept running locally. Class *NetManager* together with the interface *INetImplementation* (see Chapter 3), form the core entity for distributing different ANNs over a network by implementing such a separation of controller and implementation.

The NetManager objects remain in the local program. Each *NetManager* instance is independent of the other ones. The user may work with many *NetManager* instances at the same time in the simulation environment. Each instance of the *NetManager* has an independent GUI so that the user is able to control each ANN simulation independently.

The objects that implement the *INetImplementation* interface are appropriate for distribution because each one is independent, has no GUI, and is the only one responsible for creating, keeping and executing the core ANN functionality. The *INetImplementation* instances can be created either locally or remotely. In both cases, the instance can be moved later.

5.1.2 The ANN instance as a Voyager Agent

The clear separation of functionality provided by the object model helps to have a natural and straightforward implementation of the distribution. This can be seen in detail below, where the use of the Voyager distribution framework is explained.

Voyager implements a *forward mechanism* to deal with the agent's distribution. When a remote object is constructed using Voyager, a *proxy* object whose class implements the same interface as the remote object is returned to the server machine (from where the remote object was created). Voyager dynamically generates the *proxy* class at run time. The proxy can receive and forward messages, receive and return value, and pass the return value on to the original sender. The local machine in this case is used as a server computer to send the agents. In this way, the machine where the remote objects are created keeps the *proxy* of the remote objects.

The forward mechanism is implemented using Java interfaces. A special proxy object that implements the same interface as the local object represents the remote object (see Figure 5.1). Therefore, a variable whose static type is an interface may either refer to an instance of the actual object or a proxy object.

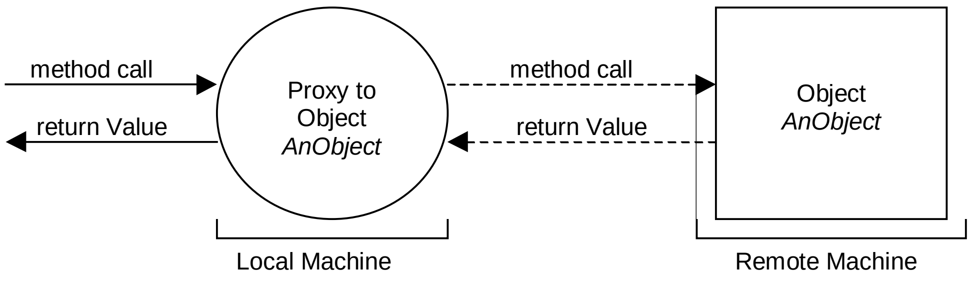


Figure 5.1 - Remote messaging using Proxy

The proxy solution perfectly fits the CANN architecture. The *INetImplementation* interface can refer to a local object or to a proxy to the remote object, and the NetManager implements the object that calls the remote object via the proxy. In such a case it is not necessary to change the CANN design. Figure 5.2 shows how the Voyager Proxy is added to the CANN framework to make remote references to a ANN component.

Figure 5.2 is the evolution of Figure 3.9 from Chapter 3 where the CANN architecture is explained in detail. The specific ANN implementation classes implement the interface *INetImplementation* that is used to act as the proxy interface for the ANN implementation code. It makes then possible for the ANN's that implements this interface to be moved using Voyager.

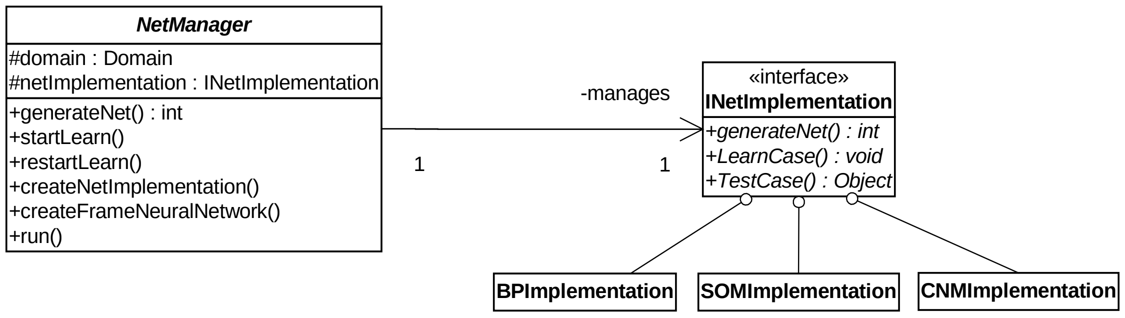


Figure 5.2 - ANN models as classes of *INetImplementation*.

It is important to refer here to the design aspects of having a proxy interface implementation. Whenever a voyager *Proxy* is implemented in this distribution framework it is implementing the "Proxy" design pattern (Gamma et al, 1995). This pattern makes clear that the function of the object that implements it is to act as a forward mechanism.

Code 5.1 shows the Java source code of the *NetManager* class for moving objects of static type *INetImplementation*.

Code 5.1 - The *NetManager* class

```
public abstract class NetManager extends Object implements Serializable, Runnable {
   INetImplementation netImplementation; // instance of the ANN model
   transient String URL = NetParameter.localHostURL; // sets to the default local host
   transient FrameNeuralNetwork frameNeuralNet; // generic ANN GUI
   Project project;

   //…

   abstract public void createNetImplementation();

   public void moveNetImplementation() throws Exception {
       netImplementation = (INetImplementation)Proxy.of(netImplementation);
       Agent.of(netImplementation).moveTo(getURL(),"atLocation");
       netImplementation.setProxies(project.domain);
   }
//…
}
```

The NetManager class defines the *netImplementation* variable that receives an instance of any object that implements the *INetImplementation* interface. The variable *netImplementation* may also contain a proxy object that will refer to the remote instance of the object.

The code continues with the declaration of a *String* that stores the URL where the code should move. The third variable is the declaration of the *NetManager* generic GUI implemented as a transient variable of the class *FrameNeuralNetwork*. Transient means that this *variable* is not made persistent. Finally, a reference to the *Project* class is maintained by the *NetManager* in order to be able to get access to the domain model and the learning and testing data.

Two methods of *NetManager* are shown in Code 5.1. The first is the abstract method *createNetImplementation()* that is responsible for creating the ANN instance. This method is implemented by the *NetManager* subclasses. So the variable *netImplementation* is set by the subclasses of *NetManager*. Those classes are specific managers for each ANN model and know how to use the specific ANN model implementation.

The second method shown is the *moveNetImplementation()*. This method implements the necessary code to move the object contained in the *netImplementation* variable. In the first method line a proxy of the *netImplementation* variable is created and assigned to it. Then, the variable does not refer anymore to the object but to its proxy.

The next step is to rely on the Voyager *dynamic aggregation*. It allows the attachment of secondary objects, termed *facets*, to a primary object at runtime. A primary object and its facets form an aggregate that is made persistent, moved, and garbage collected as a single unit.

Voyager offers the Agent facet to move objects. This facet is useful when it is necessary to give autonomy to the remote object. In such a case the task can be performed independently of the launching computer.

Code 5.1 illustrates the usage of the *Agent* facet in the second code line of the method moveNetImplementation*()*. Besides adding the *Agent* facet, the method *moveTo(String URL, String callback [, Object[] args])* is called. This method, defined in the interface *IAgent,* is responsible for moving the object to the remote program and requests two parameters:

- The first must be a string containing the *URL* where to move the object;
- The second is a string that specifies the callback method to be used to restart the code execution at the remote location.

5.1.3 Effects of moving the ANN objects

After running the *moveTo* method, the ANN object is already located in the remote program but is not able to perform properly before reorganizing its references to the local program. The method *setProxies* handles this problem. The last line of the *moveNetImplementation()* calls this method.

When moving an object in Voyager, the object and all of its non-transient elements are copied to the new location using Java serialization. Pass-by-reference interfaces like *java.rmi.Remote* are ignored. In this case, the object and its referenced objects are copied to the remote program. To avoid copying excessive and unnecessary objects it is possible to store proxies instead of the objects in the variables.

The ANN instances have references to instances of classes of the CANN project that are responsible for data access. Those are the *Evidence* and *Hypothesis* classes and its instances are controlled by an instance of the *Domain* class. For each ANN model, different references to *Evidence* and *Hypothesis* objects must be kept in order to have the appropriate access to the learning and testing data (Figure 5.3). Typically, the input and output neurons of the ANN have direct references to the evidences and hypothesis that map the problem. With such information an ANN instance can automatically get the appropriate learning and testing data whenever necessary. The references to these objects are also stored either as direct reference to the object or as a proxy in the case of having the instance in a remote program. Therefore, the remote ANN implementations are able to refer to the local *Domain* objects.

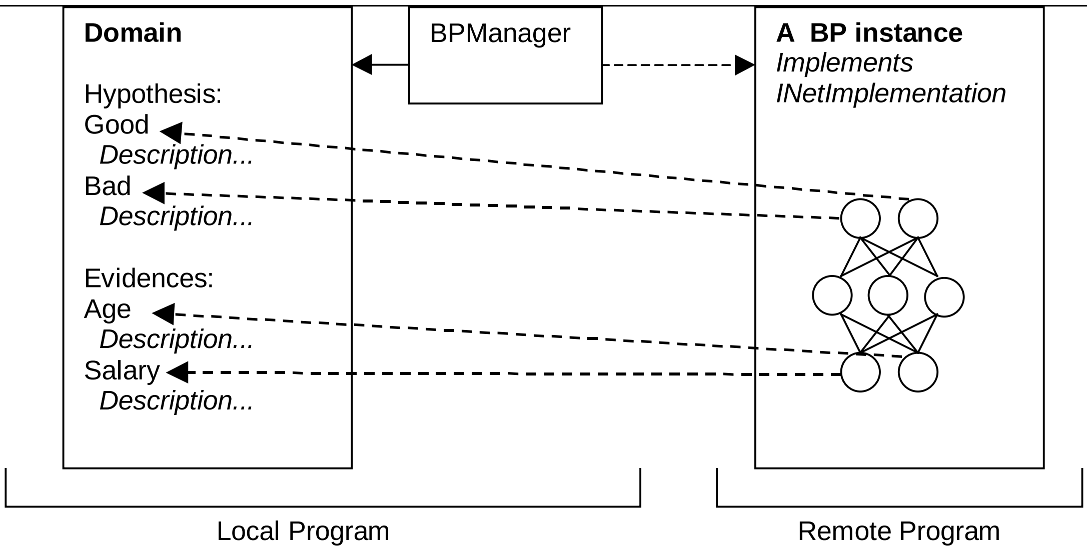


Figure 5.3 - Associating the ANN to the Domain class

If the ANN instance is created locally, then the object references are directly set to the variables, but if the instance is created remotely, proxies to the local objects are set to the variables (dashed arrows in Figure 5.3). The method *setProxies()* does this job. Each ANN model must implement this method in order to keep the correct proxies references when the ANN object is moved (the appropriate references from the *Neuron* instances to the *Evidences* and *Hypothesis* instances).

One project may have many *Domain* definitions. Each ANN instance must have its own *Domain* definition in order to make the parallelization more straightforward. It is an extra complication to handle different ANN instances referring to the same *Evidence* or *Hypothesis* instance and, in consequence, to the same data. Having different *Domain* definitions avoids the problem of synchronizing the access of one evidence or hypothesis by multiple ANN instances. Also the access to the learning and testing data is handled by each *Domain* instance independently. If different ANN instances have to use the same *Domain* definition, one *Domain* instance is cloned to each necessary ANN instance.

Substituting the variables by proxies when the object is in a remote program can be a problem when the user wants to save the project. Saving the project means saving all the domain definitions and all ANN instances. When saving an ANN instance located in a remote program, the persistence engine saves the referred proxies instead of the instances. To avoid this, it is necessary to move all the proxies to their original location before completing the saving operation. Part of the code that implements this is shown in Code 5.2.

The class *Proxy* offers the method *getLocal()* that returns a direct reference to the object if it is located in the same program. To have all the remote ANN instances in the same program, it is necessary to move them back to the local program before using the *getLocal()* method. It is possible to check whether an instance is local or not through the method *isLocal()* of the class *Proxy*.

When saving a project, it is necessary to move all ANN instances to the local program and then to restore the ANN internal domain references. After this, the ANN instances can be saved (serialized). After saving the project, the instances are returned back to the remote program and the proxies are rebuilt.

For each ANN instance the method *saveRemote()* is called. This method first returns the ANN instance to the local program by calling the method *moveNetImplementationHome()*. This method simply calls the *moveNetImplementation()* with the local URL as parameter. The local URL can be obtained from the *Agent* method *getHome()*.

After returning the ANN instance to the local program, it is possible to restore its original object. To implement the restoring of the internal proxies of an ANN instance all classes based on the *INetImplementation* interface must also implement a method called *restoreObjectsReferences()*. This method uses the methods *isLocal()* and *getLocal()* from class *Proxy* to move the ANN instance. Furthermore, this method calls the method *restoreObjectsReferences()* of the ANN instance which restores all internal proxies that an ANN instance has to the domain classes. This is the inverse effect of the *setProxies()* method.

Code 5.2 - The *NetManager* class (continued)

```
// …
public boolean saveRemote() {
   // code to get the remote code back and save it
   try {
      moveNetImplementationHome();
      if (restoreObjectsReferences()) {
         System.out.println("NetImplementation object restored");
         return true;
      }
      else {
         System.out.println("NetImplementation object not restored");
         return false;
      }
   } catch (Exception ee) {
      frameNeuralNet.statusBar.setText(ee.toString());
      System.err.println(ee);
      return false;
   }
}
public void moveNetImplementationHome() throws Exception {
    moveNetImplementation(Agent.of(netImplementation).getHome());
}
public boolean restoreObjectsReferences() {
   if (Proxy.of(netImplementation).isLocal()) {
      if (netImplementation.restoreObjectsReferences()) {
         // restores the object back to the netImplementation
        netImplementation =
          (INetImplementation)Proxy.of(netImplementation).getLocal();
        if (netImplementation == null)
           return false;
        else
           return true;
      }
      else
         return false;
   }
   else
      return false;
}
```

5.2 Testing the CANN distribution solution

The tests were performed by generating 3 instances of the CNM neural network for the credit analysis problem. The credit analysis is a classification problem where customer data from a retail company is analyzed in order to classify the customer as potentially good or bad for a credit offer. The network is trained with the portfolio data of 22 good customers and 22 bad customers (44 real cases). The customer data are organized in 32 different evidences (data attributes) including evidences such as age, sex and value of the order.

The Java JDK1.1.7b and the Voyager 2.0.1 version were used as software platform. At the moment of the CANN development and performance tests the Hotspot JVM was not available. The use of *Hotspot* (Java project on a Just in Time Compiler - http://www.javasoft.com) could significantly increase the overall performance of the system.

For testing the ORB, 4 machines connected in a LAN with velocity of 100 Mbits were used. The ideal situation would be to use 4 identical machines, but there was no such configuration available. Therefore the option was to have two groups of 2 identical machines as described in Table 5.1:

Table 5.1 – Computers used to test the distribution

Machine name	Processor	RAM Memory
Fire	PIII - 500 MHz	256 Mb
Ether	PIII - 500 MHz	256 Mb
Earth	K6II – 500 MHz	128 Mb
Water	K6II – 500 MHz	128 Mb

On the local machine (*Fire*), the instances of the CNM neural network were generated. Each instance works over the same domain model (the credit analysis), uses the same data for learning and testing, and the same default learning parameters. Therefore the generated networks have exactly the same size in neurons and the same behavior in learning and testing the examples. The first neural network is created on the local machine and its learning and testing are performed on the local machine as well. Another 3 neural networks were created at the local machine and each was moved to a different remote machine. The neural network structure (network generation) and the learning and testing were performed on the same remote machines. By doing this distribution, it is possible to verify the behavior of the CANN simulation tool, the implemented ANN framework and the Voyager ORB.

5.2.1 Measured results and discussion

One important measurement is to compare the time the same ANN spends to do the learning locally and remotely. The learning time measurements are expressed in minutes in the column "Learn" of Table 5.3. The time spent to perform the ANN test is also an important measurement once a previously trained network can be distributed to perform tasks on remote machines. How the ANNs perform in such a situation is useful in various applications from credit analysis to network traffic control. In such applications, the learned neural network is moved to the target machine where the measurement shall be done. It can perform the whole evaluation on the remote machine and simply inform the results both to the local and remote machines. The time results of the local and remote measurements of the

ANN's for testing are shown in column "Test" of Table 5.3. Both the learning and testing processes are performed for 44 cases. So the measurements are the total time the CNM network spend to learn 44 cases or to test 44 cases.

In the process of learning, it is always important to consider the network generation, that is a pre-condition to the learning. The time spent to perform the learning is also an important measurement because this time may be critical, depending on the domain application. In Table 5.3 the time spent to generate the networks locally and remotely is shown in the column "Generation".

Besides the time the ANN spends to perform the tasks remotely, it is important to consider the time the system spends to transfer the network from the local machine to the remote machine. It is necessary to consider if the neural network was already generated or not to account this time. This aspect is important because, before generating the ANN, no objects to represent the neural network architecture were created. After learning, the neural network structure was already generated and perhaps pruned, remaining only the learned ANN structure. In Table 5.3 the label "Transferring pre" shows the time spent by the CNM network for credit analysis to be transferred **before** the network generation and learning was performed. The label "Transferring pos" shows the time spent by the CNM network for credit analysis to be transferred **after** the network generation and learning was performed.

Table 5.2 – Number of CNM neurons after learning

Neurons Layer	Number
Input	32
Combinatory	679
Output	2

The CNM for credit analysis, after the generation and learning, has a total of 713 neurons (objects) remaining from the learning process, which means, those are the neurons that form the learned network. Table 5.2 shows the exact number of neurons on each CNM layer that remain after the credit analysis learning process. Besides those objects, the CNM network creates synapse objects to connect neurons from different layers. This is the network that contains the knowledge able to solve the credit analysis problem. Such a network can be transferred to different remote machines to perform credit analysis directly on these machines.

5.2.2 Performance Results

The performance evaluation involves tests for evaluating the time the ANN takes for running in the local and remote machines. Besides this, it is measured the usage of the CPU's and memory in the local and remote machines.

5.2.2.1 Time measurements

The results of the distribution tests can be seen in Table 5.3. The time the network spends to generate its structure for learning is much smaller in the local machine than in the remote ones. In the same way the time spent to do the learning is much smaller in the local machine than in the remote ones. Note that the local machine (*Fire*) and the remote machine number 1 (*Ether*) have the same hardware configuration. So the performance differences from running on the local machine and the remote machine 1 are independent of hardware capabilities. The difference in performance between the remote machine 1 and the remote machines 2 (*Earth*) and 3 (*Water*) can be delegated due to the difference among its hardware capabilities.

Table 5.3 – Time tests

CNM Credit	**Generation**	**Learn**	**Test**	**Transferring pre**	**Transferring pos**
Local	0.01 min	00.31 min	0.02 min	0.00 min	0.00 min
Remote 1	2.41 min	33.59 min	2.40 min	0.02 min	0.02 min
Remote 2	5.20 min	59.52 min	4.50 min	0.03 min	0.07 min
Remote 3	3.51 min	49.38 min	3.30 min	0.02 min	0.05 min

- Local machine = Fire
- Remote machines = Ether (1), Earth (2) and Water (3)

The time spent by the ORB to perform the neural network moving from one machine to another ("Transferring" columns), was considered to be normal and tolerable, interfering minimally on the overall performance. Even the time difference in transferring the ANN before and after the generation and learning phases haven't represented much overhead for the system performance. Therefore the transferring time is not subject of further time measurements and evaluation.

The performance of the remote neural networks for network generation, learning and testing is significantly below the expected values. The time difference between performing the learning locally and remotely is significant. It is necessary to understand why such performance degradation occurred. Was it caused by the distributed implementation at the application level or a problem generated by the overhead of using an ORB? Why did the first experiment have good performance and the second one not? The first attitude was to reanalyze the whole CANN distribution architecture and to perform complementary tests to try to get clues about the performance bottlenecks, which is detailed in Section 5.5.5 below.

5.2.2.2 Measuring CPU usage

The behavior of the CPU usage during the generation and learning processes was analyzed. The generation process has different behavior when the local and remote machines have different hardware. Table 5.4 shows the ANN generation when two similar machines are used. The CPU of the remote hardware is used more intensively, as expected. When the remote machine has a hardware inferior in performance than the local hardware, the CPU usage reaches 50% for both machines as can be seen in Tables 5.5 and 5.6. For the learning process (Table 5.7), the CPU usage on the local machine goes a little bit higher than the generation usage on Table 5.4 and the remote machine takes most of its CPU time for processing, which is the expected result. This performance was similar for all remote machines.

Table 5.4 – CPU usage for generation on similar hardware machines

Generation	CPU
Local	20
Remote 1	80

Table 5.5 – CPU usage for generation on different hardware machines

Generation	CPU
Local	50
Remote 2	50

Table 5.6 – CPU usage for generation on different hardware machines

Generation	CPU
Local	50

Remote 3	50

Table 5.7 – CPU usage for learning

Learning	CPU
Local	40
Remote 1,2,3	80

Table 5.8 – CPU usage for learning with local hardware inferior than the remote

Learning	CPU
Local “Water”	70
Remote “Fire”	50

Interesting was the result shown on Table 5.8. In this case, the chosen local hardware was the machine *Water* and for remote hardware the machine *Fire*. In this test case the local machine has inferior hardware than the remote machine. In this testing scenario, the CPU usage was much different than in the case shown in Table 5.7 where the local and remote machines where the opposite. While performing the learning process on the remote machine, the local machine kept a very high CPU usage and the remote machine did not increase much its processing even though the learning process was performed on its memory space. This behavior was not expected. It shows again that there is much communication/processing performed by the ORB among the local and remote machines. Being the local machine much inferior in processing capacity than the remote one, the result is that the remote finishes its process in advance and has to wait for the synchronized results of the local one.

Another CPU measurement performed was to evaluate what happens when another process (other running applications) takes the CPU on the local machine. The effect is that the Java ORB process slows down and the remote machine also diminishes its processing to something like 3%.

5.2.2.3 Memory measurements

The memory usage depends on the ANN model and the domain problem being solved. The memory can be precisely measured but the results cannot be generalized to any ANN model or domain problem. While testing the system the memory of all machines were monitored and no special demand was detected. Memory measurements for the CNM Model are presented in Chapter 6 where the CNM performance of the CANN system is evaluated. One type of memory test performed that gives important information is to verify the difference in allocated memory when generating the ANN network in the local machine and when generating it in a remote machine. To evaluate this, two tests were performed:

- Test 1: the CNM for credit analysis was generated on the local machine and its memory was measured before and after the network generation. After this, the system was restarted and a new CNM network instance was created and moved to a remote machine before the network generation. In this second case the remote machine memory should be used. The memory of the local machine was again measured before and after the network generation on the remote machine. The results of Test 1 can be seen in Table 5.9.

- Test 2, the same test done before was again performed but now with 3 instances of the CNM network at the same time. The 3 instances were generated on the local machine and its memory was measured before and after the ANN generation. Later, the CANN simulation environment was restarted and another 3 CNM networks where locally created, but in this case, they where moved to three different remote machines before generating network architectures. The memory of the local machine was measured twice: The first measure was after creating the networks, but before moving them to the remote location and generating the network architecture; The second measurement was done after generating the ANN structures in the remote machines. The results of this test are shown in Table 5.10.

The first test shows that less memory was allocated on the local machine when the CNM neural network was generated on the remote machine. When the network was remotely generated, the local machine allocated most of the memory (about 1 MBytes) during the process of transferring the CNM neural network instance to the remote machine. This indicates that the ORB allocated good part of that memory to be used on the handling of the distribution. Not much memory was allocated on the local machine during the process of allocating the memory on the remote machine. The remote machine allocated 4616 MBytes of memory when the remote CNM network was generated.

Table 5.9 – Test 1 – Generating one CNM instance

CNM Credit	Local Machine Before	Local Machine After
Local generation	72148	75392
Remote generation	72140	73580

The second test shows similar results as Test 1. When distributing the neural networks, the necessary memory to be allocated on the local machine is significantly less than the necessary to run the ANN´s locally. Running the 3 CNM instances locally, 9336 Mbytes of RAM are necessary to generate the 3 CNM neural structures, while running the 3 instances remotely, the necessary local memory was only 3476 Mbytes. Of this amount, 1404 Mbytes are allocated when transferring the CNM instances to the local machine, that is, memory allocated by the ORB to manage the remote instances. The average of memory allocated on each of the remote machines was 4799 Mbytes.

Table 5.10 – Test 2 – Generating 3 CNM instances

CNM Credit	Local Machine Before	Local Machine After
Local generation	73168	82504
Remote generation	73116	76592

Those tests show that it is useful to distribute the ANN generation and learning processes in terms of the memory footprint. When the limitation for generation of an ANN is memory, it is feasible to use the ORB to distribute the ANN structure. The ORB does not demand much memory footprint on the local machine and the memory the ANN allocates on the remote machine is freed on the local machine.

5.2.2.4 Measuring communication time

The performance results taken so far do not encourage the use of the distributed solution. It is necessary to clearly understand where is the performance bottleneck of the implemented solution is located. The first attitude before performing new tests was to upgrade the JDK and Voyager versions assuming that new releases improve performance.

The whole solution was migrated to the up-to-date Voyager ORB version (in that case the 3.3 version), which demands the whole software migration to the Java JDK1.3 (using Java HotSpot Client Virtual Machine (build 1.3.0-C, mixed mode)). By simply doing this, the learning time of the ANN was about 33% better for the remote machine 1 (Ether) as can be seen in Table 5.11. The learning on the local machine had the same performance.

Table 5.11 – Learning time for JDK1.3 and Voyager 3.3

CNM Credit	**Learn**
Local	00.31 min
Remote 1	23.06 min

One important aspect to evaluate is the distribution behavior of other ANN models implemented in the CANN environment. Tests were performed then using the Backpropagation solution and the ART1 solution. Both implementations are quite straightforward based on the same assumptions done to implement the CNM solution. The usage of an implementation interface to be used as a proxy to the ANN instance was reproduced for the two models.

Table 5.12 shows the performance average for the two models running on local and remote machines. The results for both models are not satisfactory either. The Backpropagation model performed about 50 times slower than on the local machine and the ART less than 2 times slower. The learning characteristics of both models are quite different, the first applies the learning cases many times while the second only once.

The results show that the performance bottleneck quite likely happens on the communication among the local and remote machines for feeding learning examples. So it is important to evaluate, especially in the CNM model, how the communication has been implemented in order to find out a possible performance bottleneck.

Table 5.12 – Learning performance for BP and ART1

		Learn
BP	Local	471
	Remote 1	24886
ART1	Local	1752
	Remote 1	2623

Another attempt to understand the performance bottleneck is to re-evaluate the differences between the first experiment with the Kohonen agent and the CANN distribution solution. There is one main architectural difference: the first migrates the whole application and does not have to fetch for learning information on the local machine.

Having this in mind, the next step is to isolate the data access by avoiding fetching learning cases in a distributed way. It means that the source code of the ANN is modified in order to fetch cases in its own program, not calling remotely to get the necessary data to learn. The modification is done on the level of the ANN neurons. Instead of fetching the data from the proxy relations to the Domain on the local machine (see Figure 5.3), the data are generated at the place the ANN is running. The proxy relations are still built, but the data are not fetch through them. The data are directly fetched to the input neurons at the particular node being a fast process.

With this modification the learning time is reduced more than 50%. The result is shown on Table 5.13.

Table 5.13 – Learning time fetching the learning data locally

CNM Credit	Learn
Local	00.34 min
Remote 1	09.13 min

The next modification is not to build the proxy relations to the Domain on the local program so that neither the data were fetched remotely nor the remote relations were kept. With this modification the learning time dropped down drastically. Table 5.14 shows the execution of the CNM learning process at the local and remote machines. One important aspect of this modification is that while running the learning on the remote machine, its CPU reached 100 percent of usage while the local machine was stable with nearly no usage. The communication between the two machines is nearly zero and the whole process has no dependence to the local machine as happened before. This proves that it is not only important to avoid fetching the data via the ORB but also not to build many proxy structures that can cause overhead to the ORB to manage.

Table 5.14 – Learning time with no proxies to the local program

CNM Credit	Learn
Local	00.34 min
Remote 1	02.16 min

The results show that the ORB communication is the biggest problem of the solution implemented. It is necessary to limit the communication between the local and remote machines to the absolutely necessary information. The learning and test data shall be available on the remote machine in order to avoid communication to fetch learning and test data. One possible solution to this problem is to use an internal database offered by the Voyager ORB. The database can be migrated to the remote machine together with the agent. Another solution may be to fetch data in bigger chunks, which means, to transfer more cases at once and then do the learning or testing locally.

Two other experiments are executed still having no proxy references to test the performance when 2 ANN's are learned at the same time. Table 5.15 shows the results when the first ANN is kept learning at the local machine and the second ANN is sent to the remote machine to perform the learning. The results are similar to the ones when the ANN's are learned separately.

Table 5.15 – Learning time with 2 ANN's at the same time, one at the local machine and the other at the remote machine

CNM Credit	Learn
Local	00.34 min
Remote 1	02.22 min

The second experiment shows the learning of 2 ANN's at the same time at the remote machine (Table 5.16). This experiment is important to evaluate the solution when more ANN's are in the remote machine and have to perform any task. It evaluates the influence of the ORB in such situation. The result is that both learning processes are done twice slower than before. As the two ANN's are sharing CPU, it is absolutely normal that the performance dropped to the half. There is no evidence that the ORB has an extra influence when more than one ANN is learning at the same time in a remote machine.

Table 5.16 – Learning time with 2 ANN's at the same time at the remote machine

CNM Credit	Learn
Remote 1 (1st ANN)	05.10 min
Remote 1 (2nd ANN)	05.10 min

With the elimination of the ORB communication for fetching data, the only remaining ORB communication is the ANN learning management. The class *CNMManager* that always runs on the local machine still keeps track of the learning process, commanding the fetching of the learning data and the start of the learning process. It commands the remote ANN to start the process of learning for each learning case and takes care of getting back its results. This communication represents most of the remaining time difference between running the ANN locally and remotely. It could be minimized if the coordination of the learning process also migrates with the ANN or if it were already installed on the remote machine being able to simply plug in different ANN's and coordinate the processing on the particular machine.

5.3 Testing the Voyager communication mechanism

The goal of this section is to analyze in detail the Voyager communication implementation in order to verify if there are external aspects that could be influencing the ANN performance results of the sections before. By analyzing this, it is possible to conclude how communication is influencing on the overall distribution performance. So a small Java application is defined in order to have a well-controlled communication application.

The implemented Java application tests the communication performance among remote objects. This test simulates the communication of the CANN framework where input neuron objects call a method in a remote object in order to fetch its input data. These data are typically a primitive Java data type such as a double value.

In this experiment, an agent object (consumer) is created and has a reference to an object (producer) that provides it with randomly generated double numbers. The agent is moved to a remote program and the number generator object keeps running on the local program. The agent creates an array of double values and populates it by calling a method of the number generator object. The method returns one double number so that it has to be called as many times as necessary for populating the complete array. The algorithm is explained below:

```
- A program is started on machine 1
- The program creates two objects:
      - A random generator (producer) that runs on the machine 1
      - An agent (consumer) that is moved to the machine 2
- When the agent is requested to run (doCommunication) it requests from
the random generator a double number. So that there is a communication
among the objects located in different machines.
      - The random object method getNumber simply returns a randomly
generated double value
      - The agent request as many double values as the size of a pre-
defined array.
```

The consumer object is a Voyager agent that runs on the remote machine 2. Its method *doCommunication(int)* (see Code 5.3), is called when the communication between the agent and the producer object called *RandomGenerator* shall happen. The producer object is located on machine 1 and when called executes the method *getNumber()* that can be seen on Code 5.4.

Code 5.3 - The agent *doCommunication* method

```
public void doCommunication(int arraySize) {
      theDataArray = new double[arraySize];
      for (int i=0; i<theDataArray.length; i++)  {
            theDataArray[i] = rGenerator.getNumber();
      }
}
```

Code 5.4 - The *RandomGenerator* object *getNumber* method

```
public double getNumber() {
      return Math.random();
}
```

5.3.1 Measuring the TCP traffic

The TCP/IP network traffic is measured on the net where this communication test program and the performance experiments were performed. The Windump tool was used to check for TCP traffic.

The subnet where the machines are located is not isolated but the tests are always performed in time frames when there is not much competing traffic. At the moment of the tests there is no significant number of other applications packages. For a total of 10000 numbers exchanged between the remote and local machines, about 20000 application packages are created. Only about 100 are other packages not related to the application. So there are no external packages that could influence the overall experiment performance.

The communication packages generated by Voyager are collected and analyzed. They are TCP/IP packages + level 2 header of 77 bytes (FastEthernet standard has minimal packages of 64 bytes). This package has 23 bytes of data. The application send packages with only one Java double value = 8 bytes. As the packages are very small in size, the Voyager is not adding data overhead to the overall packages.

From this measurement it is possible to conclude that the Voyager communication engine and the network infrastructure do not represent an overhead at the level of TCP/IP communication.

5.3.2 Performance results

The communication experiment runs in two different ways. In the first, there are two Voyager servers running on the same machine. In that case, the producer and the consumer objects are located on the same machine though running on different Voyager servers and Java virtual machines. In the second, two machines are used where two instances of the Voyager ORB server are running. The producer and consumer objects both run on a different machine.

For each experiment the number of communication steps (number of time the consumer calls the producer in order to get the generated double number), are incremented from 10000 to 50000. For each size, three runs are performed and the average is shown in Figure 5.4 for running on the same machine and in Figure 5.5 when running on the separate machines.

Table 5.17 below shows the number of steps of communication, the average results in milliseconds that the program take to perform all the communication steps for the local and the remote experiment, and the number of times the remote running is slower that the local.

Table 5.17 – Performance of the communication experiment

Communication Steps	***Remote Average (ms)***	***Local Average (ms)***	***Times slower***
10000	4373	13	328
20000	8281	30	276
30000	12458	33	374
40000	16251	44	372
50000	20189	50	404

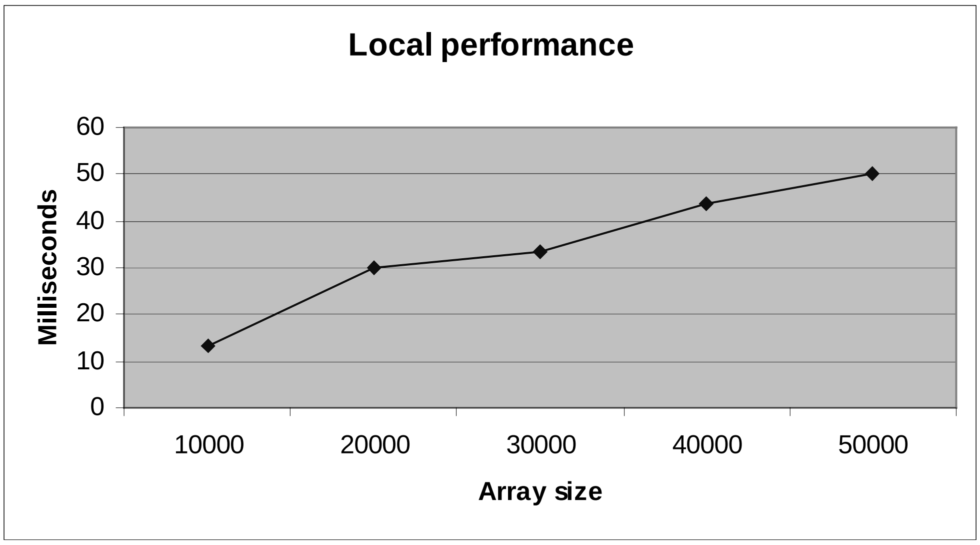


Figure 5.4 – Performance running locally

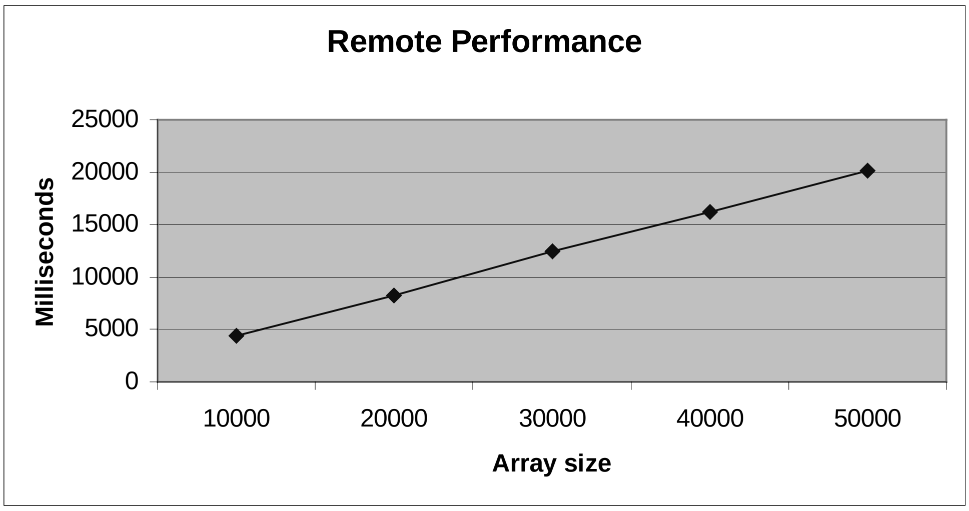


Figure 5.5 – Performance running remotely

The test is performed using two Pentium III - 500 MHz machines. The used network has a velocity of 100 Mbps. It shows that the ORB communication can be, on average, 350 times slower when performing remotely than locally. This proves that it is necessary to avoid as much as possible the communication among the distributed objects. Data that are necessary for the remote object execution shall be provided to this object using other mechanisms such as migrating the data together with the agent or accessing them in a distributed way by special distributed databases, etc.

5.4 Re-implementing the CNM framework

Summarizing, the identified problem with the CNM architecture so far is that it has an ORB intensive communication from the input neurons on the remote machine to the CANN simulation environment on the local machine to fetch the data for learning or testing. While learning, for each CNM combination, its associated input neurons have to fetch data on the local machine. As the combinations are done over the same input neurons, the same input neuron is unnecessarily fetching many times the same data on the local machine for different combinations it belongs to. The goal is then to effectively make the software architectural changes to minimize this communication, to find a way to have the input neurons populated only once for each learning case. Two solutions are considered:

5.4.1 A timestamp control for fetching the learning data

Each new learning case represents a new timestamp. For each timestamp each input neuron has to fetch data only once. If it is requested to run, it checks the timestamp. It has to fetch data on the local machine only if it is one step before the timestamp of the learning case. If it is on the same timestamp it simply uses the data already fetched to run and generate its output. The performance using the timestamp is shown in Table 5.18 and is the same as the one from Table 5.13 where the learning is done fetching the data locally. This is the expected result, the ORB references to the local machine are kept but adequately used, and the data is fetched in the necessary frequency.

The difficulty of implementing the timestamp solution is to adequately keep and disseminate the timestamp information along the framework. To implement the timestamp, the input neuron *compute* method is extended to evaluate the timestamp before performing its computation. The manager class has the control of the learning timestamp while the input neurons know their own timestamp. By comparing its timestamp with the learning timestamp the input neuron knows if it has to fetch data or not.

The problem is that on the CNM architecture there is no direct communication between the input neurons and the *CNMManager* class. The CNMManager knows only the hypothesis neurons, which have communication to the combinatorial neurons through the upper synapses and so on. The learning time stamp would have to be passed as parameter along the ANN to the input neurons or to be kept as a class variable at the manager class in order to be accessed by any class of the framework. The first solution is elegant, but changes the interface of the framework methods such as *compute*. It is not clear whether this change would be useful for the other ANN models. The second solution is implemented to evaluate the performance of the timestamp solution, but it is not adequate because it compromises the parallelism engine once the class variable can only keep the information for the timestamp of one CNM network at a time. So the main disadvantage of the timestamp solution is that for an adequate solution it changes the CANN architecture. Perhaps the necessary changes could be incorporated to the overall framework to cope with such concurrent situations as the fetching of input data, but its necessity is not evident at this moment for other ANN models.

Table 5.18 – Learning time using time stamp control

CNM Credit	Learn
Local	00.31 min
Remote 1	09.48 min

5.4.2 Controlling the learning data fetching

The learning data fetching for the input neurons has to be exclusively controlled by the manager class. The framework architecture had to be extended to implement such a solution. The idea is to separate the input neuron computation form its data fetching that was both performed inside the same method *compute*. The input neuron *compute* method that is responsible for fetching the data whenever called, lost this responsibility. Another method called *fetchData* data is implemented to make this task. The compute method is the responsible only for the computational evaluation of the input value. When the new method *fetchData* is called, it accesses the associated Domain attribute and fetches the data.

The *CNMImplementation* class is modified to have a new method called *fetchDataToInputNeurons* responsible for calling all the CNM input neurons whenever they must fetch new learning or testing data. When the *CNMManager* class knows that a new learning step must be performed, it calls the appropriate domain attributes to fetch a new case and then calls the CNMImplementation new method *fetchDataToInputNeurons* to fetch this data to the input neurons. Only after this, the CNMManager calls the CNMImplementation to do the learning of the case. At this moment the input neuron compute method is called without fetching data. The input neurons can be called many times in a learning step that they will not fetch the data again and again.

The results of the tests performed using this solution are in Table 5.19. The overall performance is better even when the ANN is running locally because it does not lose time accessing the Domain unnecessarily. When running remotely the performance is similar to the timestamp solution that was already expected.

Table 5.19 – Learning time with the change on the input neuron functionality

CNM Credit	Learn
Local	00.29 min
Remote 1	09.17 min

The range of experiments done in this section shows that the main situations that generate ORB communication are:

1. The learning or testing data fetching through the ORB

2. The control of the learning or testing process trough the ORB.

Eliminating the first reason, the performance can be considered satisfactory. The ideal is to eliminate both, giving to the agent total autonomy regarding its learning process.

The solution shown above solves the first communication bottleneck. The second one, i.e. the maintaining of the proxies references among the remote and local machines, is still an issue to be considered. The simple existence of such references decreases the performance much more than when they do not exist. Other architectural improvements could be considered here to minimize this situation.

5.5 Future implementation possibilities

The distributed solution can be improved in many aspects such as the synchronization among the remote and local objects, controlling the learning process in a distributed way and breaking the ANN structure to run in different machines. Below those possible implementations are explained in detail.

5.5.1 Synchronization aspects

The solution presented here introduces the possibility of having the trained ANN as Agents. The ANN agents are able to move to different machines on the network and act independently. It is possible then to create solutions to problems where multiple machines shall be verified or controlled at the same time to reach a certain result. Problems like monitoring certain distributed processes can be implemented in a straightforward manner.

The implementation done so far does not have such a functionality of generating various agents from one trained ANN and automatically distribute them to pre-defined machines. In the solution so far the user controls each generated ANN and has to manually send them to each machine for learning or testing proposes.

Besides using the *Agent* facet, it is possible to move objects using Voyager *Mobility* facet. In this case, the object to move must implement the interface *IMobility*. In most cases the use of this facet is enough to implement distributed computation. The *Agent* facet is useful when it is necessary to give autonomy to the remote object. In the case of the ANN simulator this facet is used because, in the future, it is planned to have the ANN learning and testing performed independently of the local program. The actual solution is not completely independent because the remote ANN relies on the local simulation program to provide it with the learning and testing cases from the database.

The existence of proxy references from the remote ANN to the local domain model generates two problems:

- First, the extra work for taking care of the creation and elimination of proxy references to the domain model.
- Second, the extra processing time lost accessing the domain in the local program to get the next data to perform the learning or test process.

To avoid these two problems it would make sense to copy the domain to the remote program together with the databases for learning and testing. This solution is feasible once the domain objects are small and Voyager offers a special database service that allows small databases to move with an object. Such a solution leads to decisions regarding the size of the moved database and its content. If the learning or testing database were large, it would be too much overhead to move it entirely. A database service would be necessary to communicate with the different remote ANN instances sending parts of the database as requested. Such a solution is certainly necessary to make the remote ANN instance more independent of the local program, being able to run most of its activities in the remote program, and consequently improving its efficiency.

5.5.2 Controlling the distributed learning process

The simulator is able to have several independent ANN instances trying to find the solution to a specific problem. The user should be able to synchronously stop the learning process of all those ANN instances. This could happen whenever the user wants or when one ANN instance reaches the problem solution. Such kind of synchronization has not been implemented yet.

5.5.3 Dividing and distributing one ANN model

As ANNs are essentially parallel processes, it would make sense to use the same distribution capabilities to divide and distribute one ANN model. This implies:

- Dividing the ANN structure.
- Controlling the learning in a parallel and distributed way.

The breaking of the ANN structure depends very much on its inherent architecture. Neural networks such as the CNM have a modular architecture being possible to divide its ANN structure. However, neural networks such as the Backpropagation are complicated regarding breaking their structure, because their neurons are fully connected from one ANN layer to the other.

5.6 Conclusion

In the experiments running the whole Kohonen application on different machines, it is verified that it runs in an acceptable time and velocity. Although the results achieved in this research cannot be generalized for every distributed application using the Java language and the Voyager environment, they can be useful as guidelines regarding the distributed implementations of neural network models under Voyager. The results encourage a more detailed implementation of distribution facilities under the CANN framework.

Adding mobility to the ANN implementations in CANN is a straightforward task basically because of the use of the Java language and the object architecture of the CANN framework. The Java language has several characteristics that are useful in this implementation:

- It offers the capability of executing the generated code on any platform, which together with the use of Voyager makes the distribution of code possible. Furthermore, the facilities the language offers such as dynamic binding and polymorphism permitted the implementation of important requirements such as adding new ANN models at runtime. For instance, the user is able to develop and add ANN models at runtime to create several ANN instances, and to try different solutions of the problem at hand. The user is able to execute these ANN tasks in a distributed way making use of the computer network infrastructure. Those aspects are considered as very important and, in many cases, compensate possible performance losses of using the Java language.

- Java implement concurrent programming. Skillicorn and Talia (1998) evaluate languages for concurrent computation. They argue that it shall be easy to program, it should have a software development methodology, it should be architecture-independent, it should be easy to understand, it should guarantee performance and it should provide accurate information about the cost of programs. Java covers quite well at least the first four items. Regarding performance, Java is clearly not reaching the point. It is necessary to consider the performance losses of the language when running the CANN framework. The experience is that the possibility of using multiple machines to test different ANN implementations and CPU's supported by the parallel CNM implementation may also compensate this drawback.

- Java has explicit programming for concurrent implementation (Skillicorn and Talia (1998)). That means the software developers shall take care with all details of the concurrent implementation. It can be difficult to achieve the best possible correctness and performance when programming with such a language. To compensate this, the Voyager library offers a brokering system in Java. It allows the implementation of mobile objects in a clear and simple way without much coding effort and without influencing the application objects implementations.

The software architecture of the CANN facilitates the implementation of mobile components. This architecture is carefully designed to allow flexible ANN implementations. The clear definition and separation of the objects that form the architecture, help to choose where and how to implement the distribution without changing the implementation done so far.

The presented solution for distribution of ANN objects opens several application possibilities. It is possible to develop applications where it is necessary to migrate the ANN code to perform artificial intelligent tasks on remote machines such as classification, forecasting and clustering. Imagine having an ANN in each computer where customer classification shall be accomplished in real-time. A previously trained ANN could be sent to each machine whenever necessary and be autonomously evaluating the input data. Other tasks in different application areas such as computer networks management could also be implemented like: making intelligent routing of messages or intelligent control of the network resources.

However, the performance results of the CANN distribution solution do not encourage its usage in a production environment. The time for performing the ANN learning and testing on the remote machines were significantly worse than the time spent when running on the local machine, which makes the distribution environment inefficient and inapplicable at this stage.

Additional tests should be conducted to verify the behavior of distributed applications when using some other technology. An evaluation of the achieved results in this experiment also depends on the application in question, which could be considered appropriate for one application but unreliable for another application. Furthermore, this experiment is concerned with providing an idea about the time, memory and computer resources someone could expect to use when building a system in a similar environment, and not to make a judgment about the appropriateness of time, memory or processing capacity. Such a judgment is beyond the scope of this experiment and is relative to each particular application.

Some parallel systems permit the load balancing among the participating CPUs (for example Cray systems). This is certainly an important feature to be implemented by the ORB when being used to implement CPU critical applications such as distributed ANN learning. With such a feature the CPU usage could be better tuned for each hardware involved in the processing and better results could be reached.

Another point to discuss is the number of agents running simultaneously on the same machine. Tests with more than two agents running simultaneously on the same machine were not performed because the scope of this work is to investigate the performance of the distributed application among several computers, and not to investigate the performance of the computers when running several applications simultaneously.

Additional topics that could be considered in future work include:

- Implementation of different distribution solutions for each of the considered ANN models.
- Adding knowledge representation and communication features to the object agents using for example knowledge communication languages, such as DAML (The DARPA Agent Markup Language - www.daml.org). Agent implementation and communication is not explored here, even though the mobile ANN components are implemented as mobile agents.
- Experiment with other distributed Java development environments such as *Aglets* (Aglets SDK, IBM 2000) or *Sumatra* (The Sumatra Project, Arizona University).

6 Optimizations of the Combinatorial Neural Model

The CNM model is explained in detail in Chapter 2. Along this work, complementary information about its implementation on the CANN framework is provided. The implementation of the CANN CNM component core construction principles is explained in Chapter 3 and a parallel solution is presented in Chapter 4. Besides implementing parallelism specific to the CNM model, this work dedicated to introduce and implement significant optimizations for its algorithm. This chapter presents the optimizations and empiric results of its application. The most important optimization aims at taming combinatorial explosion, which is the main problem inherent to this model.

6.1 CNM Optimizations

The main limitation of CNM is the possibility of combinatorial explosion, since the intermediate layer grows exponentially. The combinatorial explosion problem is critical because of memory and processing restrictions that computers have. It is not possible to previously generate all domain problem hypotheses (represented by CNM combinatorial neurons) and subsequently evaluate which one must remain or not. Because of this restriction, until now the CNM model is applied to few areas with a maximum combination order of 3. Some effort has been made trying to avoid these restrictions by using genetic algorithms to increase the maximum combination order (Denis and Machado, 1991; Machado and Rocha, 1992). The next section shows the contribution of this work to this problem.

6.1.1 Separation of Evidences by Hypotheses

The CNM model is essentially based on the knowledge graphs defined by Leão and Rocha (1990). During the knowledge graphs construction, the domain expert defines which evidences and findings have to be considered. He/she can also determine which evidences/findings relate to each problem hypothesis. This means that for some hypotheses, a smaller number of evidences/findings can be considered. As the CNM neural network structure generation is independent for each defined hypothesis, some of them can have its combinatorial explosion reduced. This happens in reality, for example, in a credit analysis problem: The expert determined that the evidence sex is important for evaluating bad customers but not for evaluating good ones. So, considering a distinct set of relevant findings for each hypothesis may significantly reduce the search space.

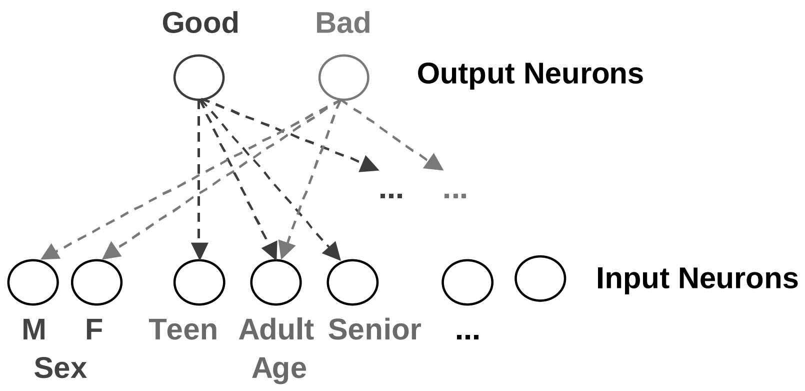


Figure 6.1 – Separation of evidences by hypothesis

Figure 6.1 shows how this example would affect the generation of the combinations of the CNM network. At the time the CNM is generated, the evidence *Sex* is not considered by the hypothesis *Good*; but the evidence *Age* is considered and all its findings (*Teen*, *Adult* and *Senior*), are considered in the combinations generation. For the hypothesis *Bad* the evidence *Sex* is considered so that combinations are generated with its findings (*Male* and *Female*). It may also happen that only one finding of an evidence shall be considered for a given hypothesis. This is the case of the finding *Adult* for the hypothesis *Bad* in Figure 6.1. The other findings of the evidence *Age* are not considered for generating combinations for the hypothesis *Bad*.

6.1.2 Avoiding nonsense combinations

There are some combinations that do not correspond to reality. We call it here *nonsense* combination and the CNM algorithm shall avoid its generation. For instance, it does not make sense to generate combinations of findings of the same evidence. For example, for the evidence Sex (Figure 6.2), nonsense combinations are those where more than one sex is considered. A person can't have two sexes so such a combination shouldn't be considered. It is clear and logical, but the original CNM does not consider this kind of situation. This is done probably with the hope of simplifying the implementation algorithm but resulted in compromising the overall system's efficiency.

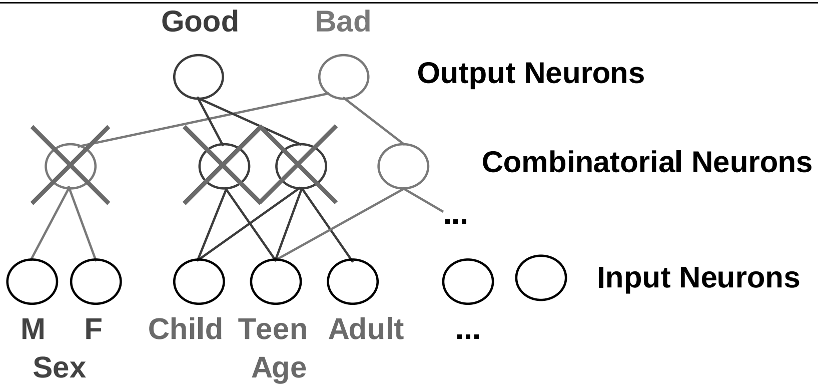


Figure 6.2 – Avoiding nonsense combinations

Even if the evidence is modeled as fuzzy, the combination of two of its findings (fuzzy sets) can't happen. The fact that the evidence is modeled of the fuzzy type warrants that more combinations including its findings will be kept after the pruning phase of the learning algorithm because more than one finding may be considered for a given evidence value. This behavior is not affected by the fact that combinations containing two findings of the same fuzzy evidence are not created before hand.

6.1.3 Optimization on the combination order definition and generation

Another approach in order to avoid creating unnecessary combinations is to take a look at what the others do in order to minimize the threat of the combinatorial explosion. So a property on which many association rule discovery algorithms (Agrawal et al. 1993) are based to cope with this problem is considered: Taking a set of selection criteria, the number of examples which pass such criteria cannot exceed the number of examples selected by any subset of this selection criteria. For example, if patients were selected from a database with the criteria: AGE > 30 AND SEX="FEMALE", the number of retrieved patients cannot be larger than the number of patients that would be selected by one of those criteria taken separately.

Based on such a property, the association rule discovery algorithm Apriori (Agrawal et al. 1996) first analyses individual items (which are equivalent to the concept of findings), so that only the ones supported by the examples used in training are combined generating 2-itemsets (combinations of 2 findings). From the 2-itemset combinations, only the ones, which are supported by the examples, are expanded generating 3-itemset combinations, and so on. With this as inspiration, the CNM algorithm for the topology generation has been optimized, resulting in a major search space reduction, especially for complex applications with a very large number of findings and where high order knowledge has to be discovered. The improved algorithm is shown in Algorithm 6.1.

Algorithm 6.1: New algorithm for CNM learning

```
Let Hk be a set of domain problem hypotheses;
Let N be an empty CNM network;
Let Fk be the set of findings that occur within the set of examples of Hk;
For each order Od from 2 up to N do
Begin
      Let Nkt be an empty temporary CNM network;
      For each hypothesis Hk do
            Add combinatorial neurons to Nkt by combining Fk with order Od;
      Train network Nkt;
      Prune non-rewarded combinations of network Nkt;
      Add to N network all remaining combinations from Nkt;
      For each hypothesis Hk do
            Let Fk be the set of findings that appear on rewarded combinations
            Nkt;
End;
Prune the N remaining networks by the original CNM algorithm readjusting the
weights;
```

In the first iteration of the main "for" loop, the findings that occur associated to each hypothesis will be considered to generate order-2 combinations. These combinations are stored in a temporary network, which is trained and pruned. By pruning, all combinations that are not validated by the examples will be deleted from the network. This pruning is a simplified version so that only the combinations that are never rewarded during learning (reward accumulator is zero) are pruned and the weights are not changed. After this pruning, the remaining combinations are transferred to the network.

As some parts of the network have been pruned, it is expected that some findings that does not occur in any combination that has been rewarded (which did not occur in the set of examples), will not be relevant in the next algorithm iteration. Since the complexity of combinatorial layer generation is exponential, even a small reduction of the number of findings to be combined has a significant impact on the size of the search space.

After doing the learning loop from the order 2 to the desired order, a final pruning process is applied over the remaining network. This pruning is the original CNM pruning algorithm, where combinations that received more punishments than rewards are pruned and the weights are modified.

The main advantage of this algorithm is the reduction of memory and time resources for the learning process without compromising accuracy, as no relevant findings are pruned.

Furthermore, it is possible to generate nets with higher orders than with the original (non-optimized) algorithm. This can be seen in the next section that discusses the performance of the non-optimized and optimized CNM algorithms.

The optimizations are applied using the original CNM reward, punishing and pruning calculations. If those calculi are changed, it is necessary to reevaluate the applicability of the optimization. Furthermore, other methods for optimizing the combination generation are not considered in this work and may be subject of further research and comparison to the presented optimizations.

6.2 Test Results

The domain for this test is that of credit analysis. Real customer data, provided by a company, have been used describing information about customers and what they bought. The task is to classify the customers as either "good" or "bad" ones. The company domain expert (credit analyst) defines the relevant evidences and findings. A set of 13 evidences are identified, e.g. age, sex, order value, type of customer, etc. From these 13 evidences, 32 findings are defined based on finding types such as fuzzy (e.g. age = teen, adolescent, adult, senior), numerical (e.g. type of customer = 1,2 or 3) or string (e.g. sex = M or F).

To better evaluate the improvements of the optimized algorithm, 3 tests were performed using 3 CNM networks with the following characteristics:

1. A CNM network generated using the normal CNM algorithm [9, 10] that is called here non-optimized network (Table 6.1). This network contained all combinations for the specified 32 findings from order 2 to order 4.

2. A CNM network generated without the non-sense combinations but still using the non-optimized algorithm (Table 6.2). This test is important for verifying the size of the remaining network without the non-sense combinations.

3. A CNM network generated by the optimized algorithm (Table 6.3). The findings are separated by hypotheses, non-sense combinations are eliminated, and the learning is done step by step by eliminating non-relevant findings based on Algorithm 6.1.

The Tables 6.1, 6.2 and 6.3 show the results of the 3 proposed tests. The table's structure is as follows:

- The column "Comb. order" shows the combination order that the CNM network must generate for learning. In case of Tables 6.1 and 6.2, the CNM simultaneously generates all combinations from order 2 to the order specified in this column. In case of Table 6.3, the CNM generates combinations step by step for each combination order from 2 to 4 based on Algorithm 6.1.

- The column "Hypotheses" shows the two hypotheses considered in the testing problem domain: "good" and "bad". The division into good or bad is important for the evaluation of other columns that separately show the number of combinations for each hypothesis.

- The column "Number of generated combinations for order N" shows the number of generated combinations for each order N.

- The "Remaining rewarded combinations" column indicates the combinations that were not pruned because they received reward during the learning process (simplified pruning of Algorithm 6.1).

- The "Final number of combinations" column shows combinations which remain after performing the original CNM pruning (the last pruning of Algorithm 6.1).

- The "Findings number" is the number of findings considered for the combination order N. In these tests, the networks start considering all the 32 findings for each hypothesis.

- The column "Time" shows the time taken by the learning algorithm to generate the network and to perform the learning and pruning processes. A set of 44 cases are used (22 good and 22 bad) for the learning. The time is given in milliseconds and minutes.

- The column "Memo" shows the amount of memory (in Mbytes) used for doing the learning, i.e. the amount of memory used for allocating the CNM network.

- The column "Result" shows the number of correct responses for each hypothesis after testing another data set with 44 cases (22 good and 22 bad).

Table 6.1 - Non-optimized network for order 4

Comb. Order	Hypotheses	Number of generated combinations for order			Remaining rewarded combinations	Final number of combinations	Findings number	Time		Memo	Result
		2	3	4				µs	Min		
4	Good	496	4960	35960	7785	3306	32	495891	8:15	40936	15
	Bad	496	4960	35960	7636	3172	32				16

The columns "Remaining rewarded combinations" and "Final number of combinations" should have the same values for the three algorithm versions. The first test (Table 6.1) has different values due to remaining combinations among findings of the same fuzzy evidence. Those combinations should be eliminated since two different values of one evidence are forbidden. However, this sometimes occurs with fuzzy evidences because two different fuzzy values can be presented to the network at the same time in a single case (e.g. an age 45 can be considered 0.5 adult and 0.5 senior). Thus, there will inevitably be some non-sense combinations remaining at the end of the learning process. This difference is not encountered in the test types two and three because their nonsense combinations are not generated. It is important to realize that these remaining nonsense combinations are not strong enough to be activated, and do not influence the correct performance of the network.

Table 6.2 - Non-optimized network for order 4 – only eliminating non-sense combinations

Comb. Order	Hypotheses	Number of generated combinations for order			Remaining rewarded combinations	Final number of combinations	Findings number	Time		Memo	Result
		2	3	4				µs	Min		
4	Good	470	4196	25409	6968	2900	32	321328	5:21	28220	15
	Bad	470	4196	25409	6864	2809	32				16

The analysis of the findings, which are eliminated during the optimized learning, is equally important. In test type 3 (Table 6.3), after the learning and pruning of order 2, it is verified that some findings are eliminated, reducing to 28 findings for the "Good" hypothesis and to 26 findings for the "Bad" hypothesis. The next learning order only generates combinations for those remaining findings, greatly reducing the overall size of the generated network. In this problem domain, the number of findings does not reduce for the orders larger than 2.

Table 6.3 - Optimized network for order 4

Comb. Order	Hypotheses	Number of generated combinations for order			Remaining rewarded combinations	Final number of combinations	Findings number	Time		Memo	Result
		2	3	4				µs	Min		
2	Good	470	-	-	297	-	32	3938	0:03	1180	-
	Bad	470	-	-	271	-	32				-
3	Good	-	2768	-	1587	-	28	31750	0:31	3204	-
	Bad	-	2244	-	1514	-	26				-
4	Good	-	-	14441	5084	2900	28	118281	1:58	17760	15
	Bad	-	-	11001	5079	2809	26				16

The time consumed for the learning process shows a significant difference between the test type 1 and the test type 3 (non-optimized to the optimized). The optimized network spent 68.95% less time than the non-optimized network.

The tests also show the economic use of memory through the optimized learning algorithm. The optimized network (Table 6.3) use only 43.38% of the memory compared to the non-optimized (Table 6.1). Because of such memory savings during the learning, it is possible to generate the neural network up to order 5 using the optimization algorithm. It is not possible to generate the order 5 for the non-optimized algorithm because there is not enough memory to support all the combinations generated at the same time.

It is possible to verify this economy in memory during the CNM learning. With the non-optimized algorithm, the combinations (for all orders) are generated and maintained in memory simultaneously. For Tables 6.1 and 6.2 it is necessary to add the orders 2, 3 and 4 of the column **"Number of generated combinations for order"** for both **"good"** and **"bad"** hypotheses. For the optimized algorithm (Table 6.3), it is only necessary to add the **"Remaining rewarded combinations"** of previous orders, and the column **"Number of generated combinations for order"** for the order in learning process. Considering learning order 4, for the non-optimized network (Table 6.1), the total number of combinations is 82832 while for the optimized network (Table 6.3) it is 29111, which means a 64.86% reduction. It is important to remember that the number of generated combinations depends on the combination order and on the domain model. Thus, it may change very much from one application domain to another but the optimizations, in terms of the number of generated combinations, will always be relevant.

6.3 Conclusions

It is important to reinforce/summarize the optimizations results obtained by the algorithm proposed here:

1. About 1/3 of the time for learning.
2. About 1/2 of the memory used.
3. About 1/3 of the combinations generated.
4. Combination order up to 5.
5. Same classification quality.

The optimizations presented have significantly reduced the generation of the combinatorial layer of the CNM model. In the approach presented here, relevant findings are separated in a subset for each hypothesis (reducing the number of findings to be considered) and nonsense combinations are avoided. A major search space reduction has been achieved, as the generation of combinations is controlled in order to avoid the pre-generation of all possible combinations for a given combination order. A new algorithm with such optimizations is proposed, implemented and tested. The adequate software architecture makes it possible to consider each detail in the sense of best using the computational resources to make the CNM model applicable. Finally, the CANN CNM component implements the two algorithms, the original and the optimized.

7 ANN simulation

In this chapter the developed CANN simulation environment is explained in detail. Its functionality is shown and its strong and weak points are carefully analyzed. To better understand these aspects other simulation environments are also analyzed to form a background. So the goals of this chapter are twofold:

- To show the CANN simulator functionality and analyze it.
- To analyze other simulators and compare them to CANN.

To better understand what is going to be analyzed it is important to understand what kind of ANN simulation tools exist and what characteristic are taken into consideration in order to evaluate the simulators. The ANN development and simulation tools can be divided into three categories (Kock, 1996):

- **Menu based/graphic oriented systems** – Systems that allow the user to manipulate the ANN models via a graphic oriented interface. In general, the user can instantiate new ANN instances and manipulate them based on parameterization. Examples are: SNNS (Zell, 1995); NeuralWorks (NeuralWare, 1995); and ECANSE (SIEMENS AG, 1998).
- **Module libraries** – Software libraries programmed in a general purpose language such as C or Java. In general the ANN networks can be instantiated in a user program and appropriately accessed via parameterization. Sometimes the libraries offer extension facilities of the core library software. Examples are: Xerion (Camp, 1993); SESAME (Goddard et al. 1989); and ABLE (IBM, 2000).
- **Specific programming languages** – Systems that offer a specific programming language for creating ANN. They may include a library of ANN modules already implemented in that language. Examples are: Aspirin (Leighton, 1993) and CONNECT (Kock et al. 1994).

The simulators may belong to more than one category but usually one aspect is stronger, justifying its categorization. There are some criteria that can be taken into consideration when implementing and evaluating a simulation tool. The importance of one criterion may vary from user to user so that it is not important that a simulation tool implements all the criteria but properly identifies which ones are important for each application situation. Some accepted criteria are:

- **System handling** – The user interface should be easy to use and learn. The user may find proper information about the ANN abstractions offered by the tool and may have tools to understand the results of the performed simulations such as graphics, etc.
- **Flexibility** – The simulator should support a minimal set of pre-built relevant ANN models and it should be easy to play around with those models by changing topology, learning rules, etc. The flexibility has much to do with software design principles and reuse of the whole simulation software pieces.
- **System integration** – The tool should be able to integrate with other systems on the pre- and pos-processing of the ANN. For instance to have tools for fetching data and for facilitating the use of the learned ANN on other systems. The portability of the simulation tool is also an important aspect of integration.
- **Pragmatics** – The system scalability is an important aspect to be considered because of the nature of the ANN simulation that can quickly enlarge and take the environment resources.

Taking those criteria into consideration, some specific aspects that are relevant to this wok when developing and analyzing simulation environments are listed on the Table 7.1.

When the evaluating simulation tools it is important to consider, from this work point of view, the implemented software engineering principles. For instance, it is relevant to verify check whether the development is done taking care of OO principles, frameworks concepts, design patterns, and components technology. There are not many ANN simulation tools that can clearly be included in this group. In this work two of such tools will be analyzed: ECANSE (SIEMENS AG, 1998) and ABLE (IBM, 2000).

Table 7.1 – Criteria for analysing ANN Simulators

System handling	• GUI facilities o Data visualization o Visual programming o GUI framework o ANN visualization
Flexibility	• Components o Level of abstraction o Reusability o Deploy facilities o Testing facilities • Several ANN and domain at the same simulation environment • Including ANN components at runtime
Integration	• System and components portability • Data o Access – ASCII, Database o Manipulation/conversion facilities o Domain modeling
Pragmatics	• Scalability o Distribution facilities o Simulation and ANN parallelism

The three simulation environment characteristics are explained in detail along the rest of this chapter.

7.1 The CANN simulator

Taking the classifications for the ANN simulation tool introduced above, it is possible to classify CANN as a module library because its extensibility is done via software programming. As already explained before in Chapter 3, the CANN components implement, in general, *whitebox* frameworks that can be extended either by inheritance or by composition. However, it is also possible to consider CANN as a hybrid solution because a basic graphic simulation environment was already built. The ANN GUI framework generalizes some functionality management for any ANN model and there are also facilities for creating and editing domain models and data converters. The simulation environment forms a kernel for the simulation of ANN extended from the CANN components.

The main goals for building the CANN simulation environment in this work are:

- To have an environment where to perform ANN simulation using the CANN components.

- To get experience implementing a general GUI for ANN simulation.

The main characteristics of the CANN simulation environment are:

- Several ANN can be instantiated and simulated at the same time.

- Several domain can be modelled and used at the same time by different ANN models

- The ANN simulation can be distributed to the networked computers.

Anybody who wants to use the CANN only as a simulation environment through the already existing ANN components does not need to understand the inner programming characteristics of the CANN component frameworks. However, the user who would like to extend the CANN functionality does need to understand the inner programming characteristics. As CANN is a *whitebox* framework the whole source code is available to the programmer so that he is able to understand the core software details and change it. Along this section, the CANN functionality will be analyzed in detail. The CANN components that implement core functionalities may be freely cited without adding details. To fully understand the software engineering details of those components it is necessary to read Chapter 3.

7.1.1 The Project

The CANN simulation environment implements one instance of the CANN *Project* component. The simulator does not allow opening more than one project at the same time. A created project may contain any number of CANN Domain component instances and CANN ANN components instances. Figure 7.1 shows that the project called *twonets.prj* is already opened and the actions that can be applied to project are shown as well. The user can perform the following actions:

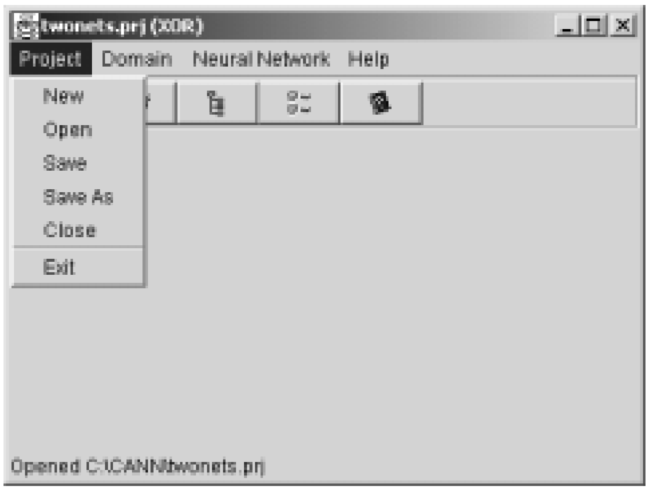
- New - Create a new project.

- Open - Open a project.

- Save - Save the opened project

- Save As - Save the opened project with a new name.
- Close - Close the opened project.
- Exit - Exit the CANN simulator.

At the bottom of the frame there is a status bar that shows important messages to the user, in general, relevant results of his/her actions or errors.



Opened C:\CANN\twonets.prj

Figure 7.1 – Actions over a CANN project

The creation of a new project presumes the creation of new domain and ANN instances. Next, the creations of those are explained in detail.

7.1.2 The Domain

Figure 7.2 shows the possible actions that can be executed over the Domain:

- Select – The user can create new domain instances and select one to manipulate its hypothesis and evidences definition.
- Evidences – Create and edit domain evidences.
- Hypotheses - Create and edit domain hypotheses.
- Data Sources – Actions over the data sources for learning and testing.

- Choose - Choose the type of data source, it is an instance of the Fetcher CANN component (e.g. ASCIIFetcher).
- Learn – Choose the data source for learning based on the selected type (e.g. a text file).
- Test – Choose the data source for testing based on the selected type.

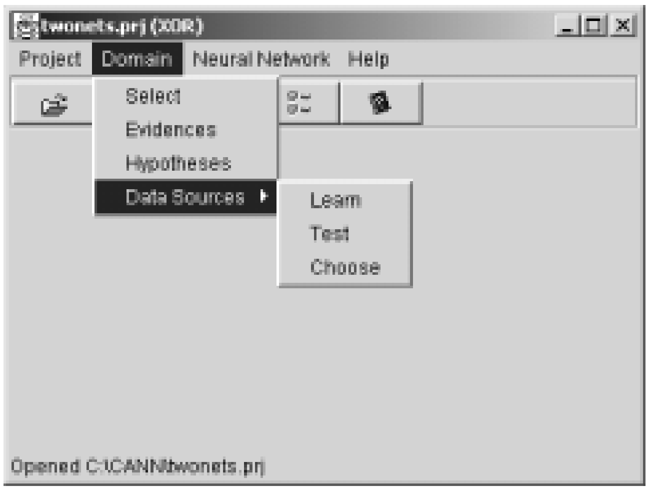


Figure 7.2 – The possible actions over the Domain

Figure 7.2 shows that the project *twonets.prj* is open. It shows also between parentheses the name of the selected domain, in that case *XOR*. After creating a new project the user must create the domain, which can be done by the *Domain-Select* menu. Figure 7.3 shows the Domain select dialog. Here the user can create a new domain simply by typing the name and clicking on the *Add* button. The created domain is then listed on the top dialog list of domain names where, at Figure 7.3, the *XOR* and *Bi-Dimensional* domains are listed. The user may also select between the two domains by using the *Select* button, or remove an existing domain from the list.

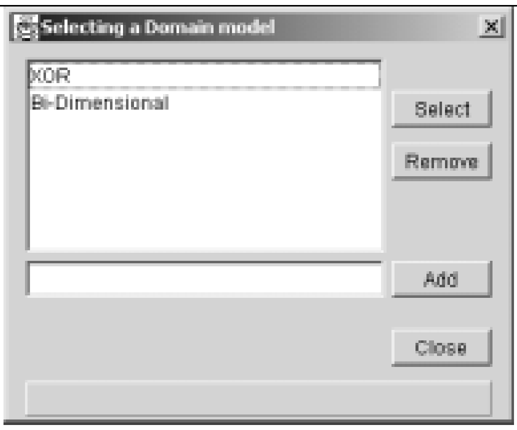


Figure 7.3 – Creating or selecting a domain model

After creating the *Domain* instance, it is possible to return to the main Domain menu and create the hypotheses and evidences of the selected domain.

7.1.2.1 The data converters

Before creating any evidence or hypothesis it is necessary to choose the type of data converter that will be applied for the given domain. The converters are subclasses of the *Fetcher* class and in the case of Figure 7.4 the *ASCIIFetcher* and *DatabaseFetcher* are available. The second is not fully implemented. At the moment, the CANN simulation environment does only work with ASCII file converters.

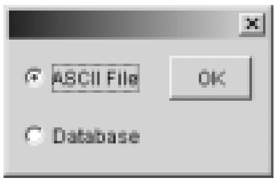
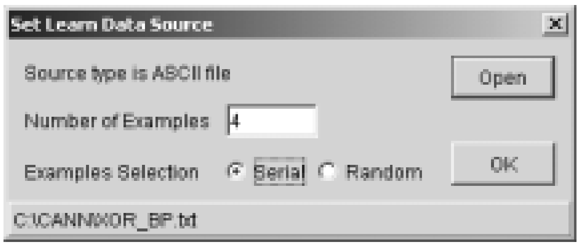


Figure 7.4 – Selecting the data converter

By setting the converter types the user is also automatically selecting the type of two implemented converters: the one that will be implemented for fetching the data for the domain, that means selecting a case from the case base; the other for the evidences and hypothesis, that means selecting the exact data inside the case that belongs to the given evidence or hypothesis. In this example, the domain will have associated an instance of the *ASCIIFetcher* component. In the same way, instances of the *EvidenceASCIIFetcher* will be automatically associated to the evidences and hypotheses at the time they are created. While creating these evidences and hypotheses the user will have to set the necessary values for the implemented *EvidenceASCIIFetcher*.

After the selection of the converter type, it is possible to select the data sources based on the converter type. For the ASCII file converter, the dialog of the Figure 7.5 appear to the user, where he/she can browse the file resources and select a given ASCII file (*Open* button). In the example of Figure 7.5 the file *C:\CANN\XOR_BP.txt* is selected. The user can also set the number of examples (cases) that must be considered from this file as learning examples. He/she may also set whether the examples shall be selected serially from the first example or randomly among the given examples inside the file. The dialog for setting the testing data file is very similar.


Set Learn Data Source

Source type is ASCII file

Number of Examples 4

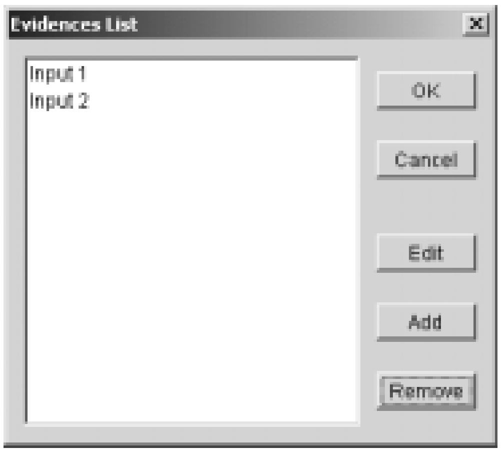
Examples Selection [✓ Serial] Random

C:\CANN\XOR_BP.txt

Figure 7.5 – Setting the learn data source

7.1.2.2 The Evidences

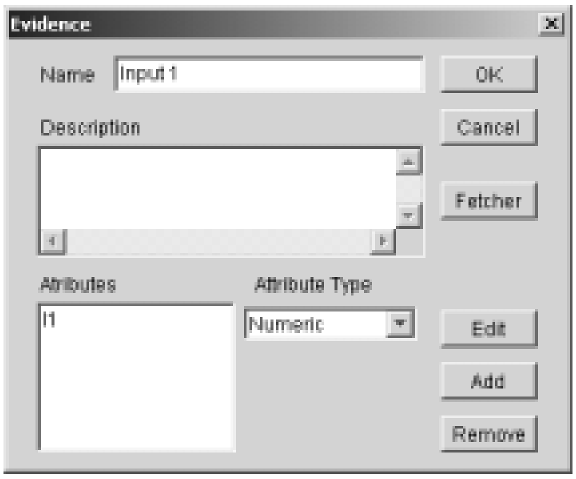
The evidences defined for the *XOR* problem can be seen in Figure 7.6. In this dialog, the user can Edit, Add or Remove evidences to and from the domain model.


Evidences List

Input 1
Input 2

Figure 7.6 - List of Evidences

Experts use evidences to analyze the problem in order to come up with decisions. Evidences in the case of the XOR problem would be the values that represent the two variables analyzed in a XOR clause. In the example, the XOR problem has two input variables so that two evidences are created. The modeling of the evidences and hypothesis may not be the same for different ANN models. In general, the way the problem domain is modeled must consider the ANN model that it will be applied. For instance, modeling the XOR problem for the Backpropagation and CNM models are different. Figure 7.7 shows the dialog that is used to add or edit evidences. In the example the evidence *Input 1* has been edited.


Evidence

Name: Input 1
Description:
Attributes: I1
Attribute Type: Numeric

Figure 7.7 – Editing one Evidence

The evidence may have more than one attribute of different types. The *Input 1* evidence in Figure 7.7 has one attribute called *I1* of the *Numeric* type. The attributes of an evidence are responsible for preparing the learning or testing data for an input neuron of the ANN component. The attributes implement the necessary data conversion to turn the input data into something that the ANN input neuron is able to process. The attributes of the *Numeric* type simply warrant that the value be a numeric (float in that case). The *String* attribute implements a string that is used to compare with the given input data. If the strings are identical it returns the numeric value 1, otherwise it returns the value 0. The *Fuzzy* attribute applies a fuzzy function to the given input and returns the result of the function, a numeric value from 0 to 1. Finally, the *Range* attribute is a specialization of the *Numeric* attribute that returns 1 if the given input value is inside its defined numeric interval, otherwise it returns 0.

The button Fetcher on Figure 7.7 is used to configure the evidence converter. Figure 7.8 shows the implemented dialog for the *EvidenceASCIIFetcher*. In this case the user must define the columns where to take the date in a given case. The case inside an ASCII file must be organized as records. Each case is a record or a line on the file. Each record contains the data for all attributes of a given case. The attributes data shall be organized in determined positions inside this record. This is the most basic way of organizing an ASCII flat file. It is possible to implement other fetchers to implement coma-separated files, or even files based on XML definition.

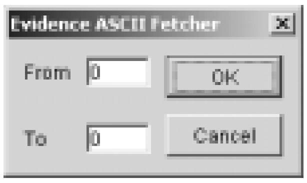
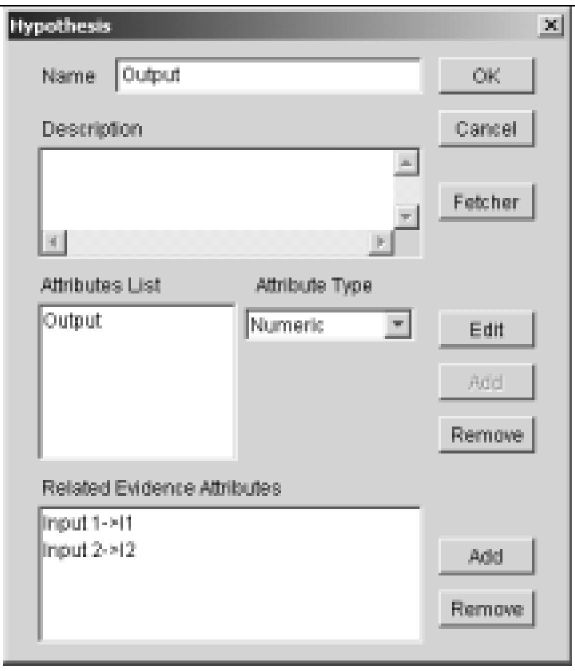


Figure 7.8 – Evidence Fetcher

In the example of Figure 7.8, it is defined that the evidence *Input 1* is located in the position from *0* to *0* in the file, which means the first column of the record. The *From* position starts on zero, so the first column of the file must be considered the column zero. The *To* position determines the last column that must be taken (*From* and *To* are "inclusive"). The attribute value is taken as a string and converted by the modeled evidence attributes based on its given type as already explained.

7.1.2.3 The Hypotheses

The hypotheses can be created and edited in a similar way as the evidences. There is a list of hypotheses that is similar to the list of evidences of the Figure 7.6. The dialog for creating/editing a hypothesis can be seen in Figure 7.9 and is very similar to the evidence dialog as well. This dialog has the same functionalities as creating the evidence fetcher and the evidence attributes. Besides this, there is a list of the related evidence attributes. This list was specially created for the CNM optimized model introduced in Chapter 6, but turned out to be important for all the CANN components. This list defined which evidence attributes shall be considered in the creation of the ANN topology, given a certain hypothesis. Usually all the modeled evidence attributes are associated to all hypothesis. In the example, the evidence *Output* is associated to all modeled evidence attributes. Because of this association it is necessary to create first the evidences and its attributes and then create the hypotheses.



Name: Output

Description:

Attributes List: Output

Attribute Type: Numeric

Related Evidence Attributes:
Input 1->I1
Input 2->I2

Figure 7.9 – Editing one Hypothesis

In the example, the *XOR* problem has only one output so that one hypothesis was modeled and called *Output*. It has only one numeric attribute called *Output* as well. In the case of the Backpropagation ANN component this attribute will be associated to the output neuron and during the supervised learning phase it will provide the learning cases results to be compared with the ANN calculated output. An attribute is always associated to one ANN input/output neuron. The attributes defined for the evidences and hypothesis are used to automatically build the ANN topology.

7.1.3 The ANN simulation

Figure 7.10 shows the menu alternatives for adding a new ANN component to the CANN simulation and for managing the simulations of the ANN instances.

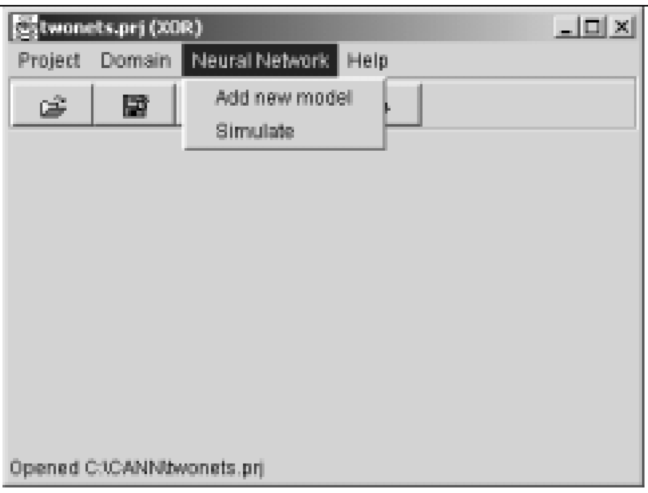


Figure 7.10 – Neural Network menu

7.1.3.1 Adding ANN components at runtime

The user is able to add at runtime any ANN component through the menu *Neural Network - Add New Model*. Figure 7.11 shows the dialog used for including new ANN components. In the example, the components for the Backpropagation and SOM ANN models are already included. To include new models the user must simply type the name of the component and the simulator will find it on the appropriate CANN path. The ANN components are the concrete classes of the *NetManager*. In the example the CNM component name was typed, it is necessary simply to press the button *Add* to have it plugged to the simulator.

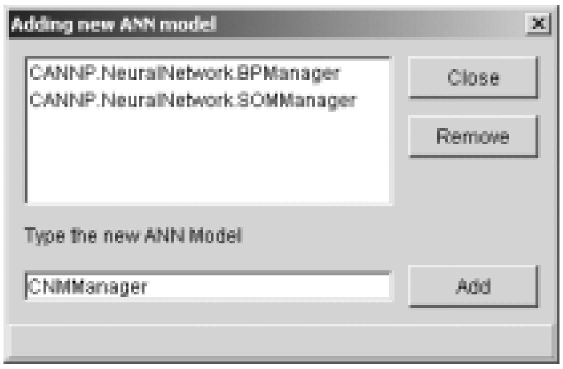
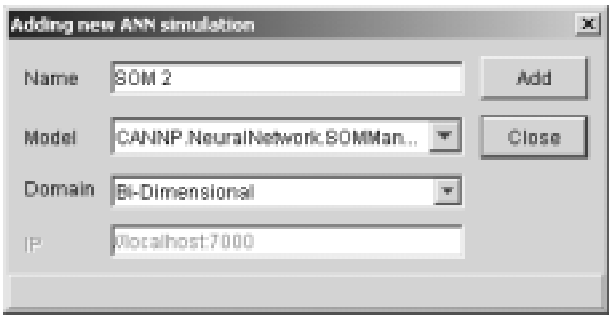


Figure 7.11 – Plugging a new ANN component at runtime

7.1.3.2 Creating ANN instances

The creation of an ANN instance based on the added components is done in the dialog shown on Figure 7.12. The user gives a name for the ANN instance he/she is creating, for instance *SOM 2*. The ANN component can be selected using the *Model* combo box. This combo shows the models added before on Figure 7.11 dialog. The ANN instance will also have an associated domain instance that is selected from the already created domains using the *Domain* combo box.


Adding new ANN simulation

Name: SOM 2
Model: CANNP.NeuralNetwork.SOMMan...
Domain: Bi-Dimensional
IP: localhost:7000

Figure 7.12 – Creating a new ANN instance

The variable *IP* is disabled because it is only informative. It means that the ANN instance is created and run on the IP of the local host at port 7000, that is, the address where the Voyager server is running. The ANN instance starts running on the Voyager default IP and port and can move to another host under the user's choice. This aspect is explained further in Section 7.2.3.3.

The dialog of Figure 7.12 is called when pressing the button *Add* on the *Simulating ANN's* dialog of Figure 7.13.

7.1.3.3 Simulating the ANN instances

Figure 7.13 shows, on its upper list, three ANN instances. The first two (*BP 1* and *SOM 1)* were previously created and were persisted within the project *twonets.prj*. A third ANN instance was created when creating the Figure 7.12 example. The *Status* list below reports the user's actions and occasional errors. The printed message of the example is reporting that the *SOM 2* instance was successfully created.

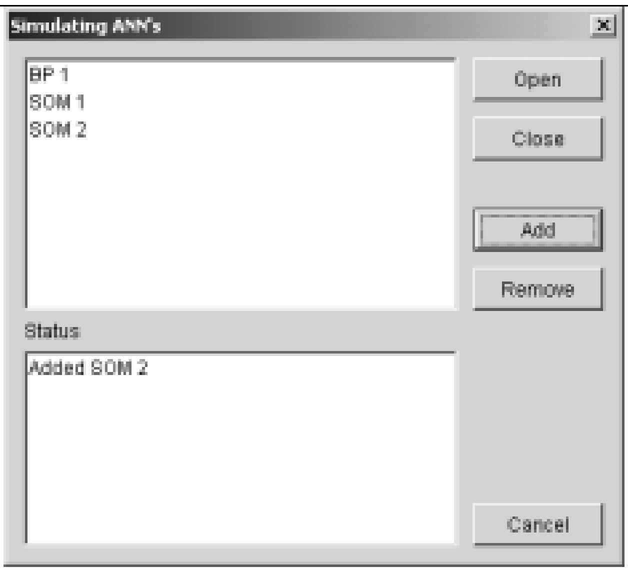


Figure 7.13 – Managing the ANN simulation

In Figure 7.13, the buttons *Open* and *Close* are used to open or close the ANN instance simulation GUI that is shown on Figure 7.14.

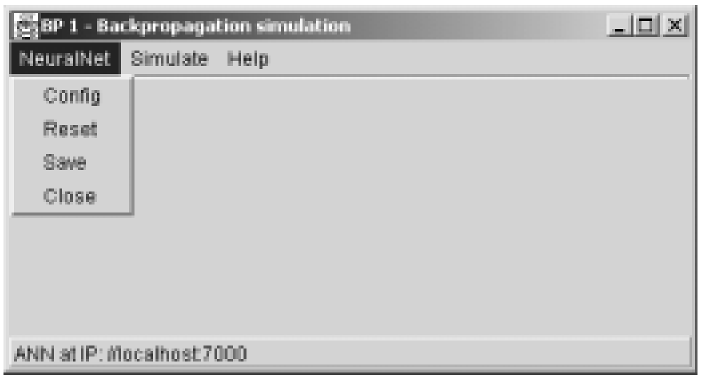
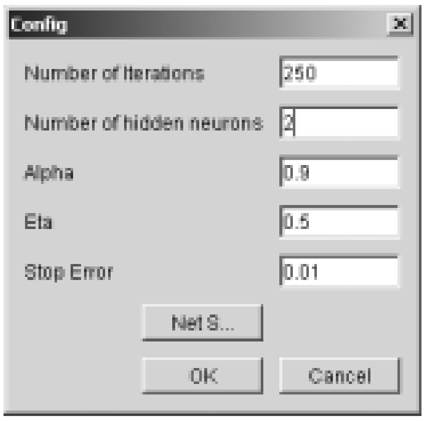

BP 1 - Backpropagation simulation
NeuralNet
Simulate
Help
Config
Reset
Save
Close
ANN at IP: //localhost:7000

Figure 7.14 – ANN simulation frame

For simulating an ANN a GUI framework is created, which is formed by a set of generic Java classes based on *Frame* and *Dialog* classes that are used by any ANN instance. The details of this implementation are already explained in Chapter 3. In the case of Figure 7.14 it is running the Backpropagation instance called *BP 1*. The menu *Neural Net* offers the possibilities to configure the inner ANN component, reset - meaning creating a new network structure with new weights, save the neural network instance and close the frame.

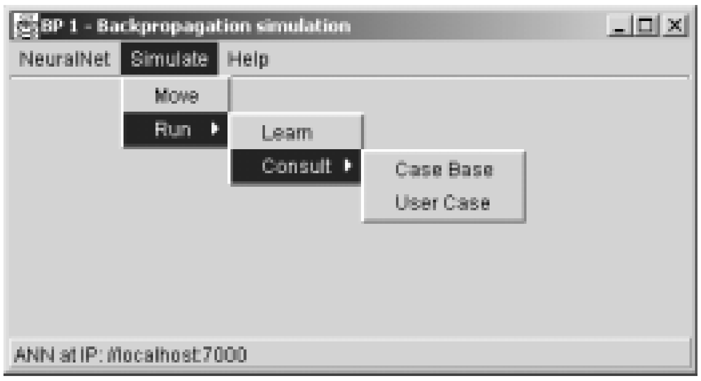
Figure 7.15 shows the dialog where the user can choose the appropriate learning parameters for the Backpropagation ANN instance. As already explained before, this dialog must be implemented for each different ANN model (see Chapter 3).


Config

Number of Iterations 250
Number of hidden neurons 2
Alpha 0.9
Eta 0.5
Stop Error 0.01

Figure 7.15 –Backpropagation configuration

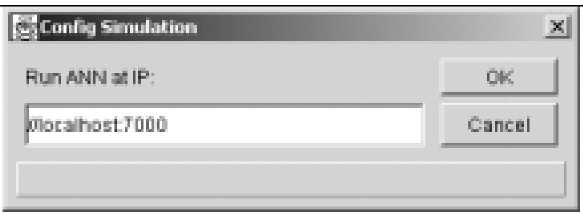
The menu *Simulate* shown in Figure 7.16 offers the alternatives to execute the ANN moving, learning and testing (called here *Consult*). The Help menu is not implemented yet. At the bottom of the dialog there is a status bar to print information and errors to the users.



ANN at IP: //localhost:7000

Figure 7.16 – The Simulate menu

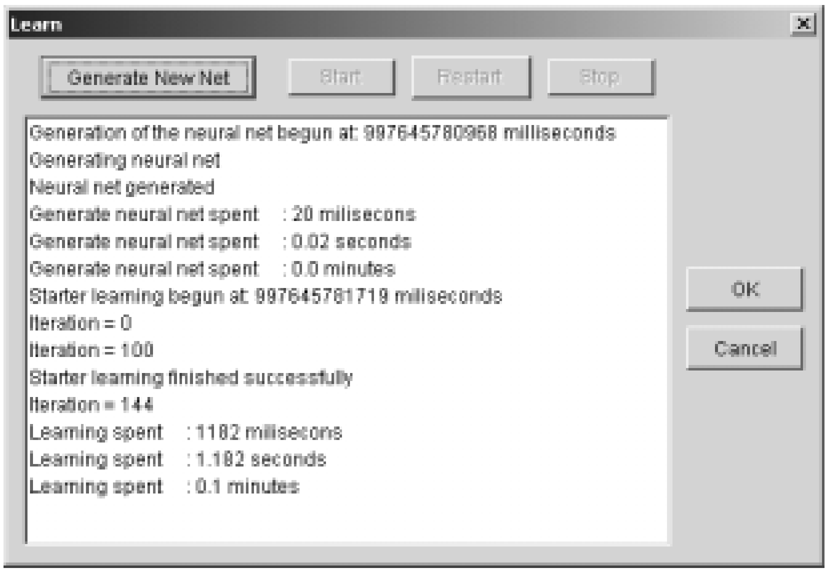
Figure 7.17 shows the dialog called by the *Move* menu. In this dialog the user can specify a host IP and port where to move the ANN instance, making use of the mobility characteristic of the ANN components. The mobility is explained in detail in Chapter 5.



```
localhost:7000
```

Figure 7.17 – Moving the ANN component to run in a remote machine

Figure 7.18 shows the learning dialog. In this example the Backpropagation network is generated for the given XOR domain problem in 20 milliseconds. The learning was performed for the XOR problem as well, and succeeded at 144 epochs taking 1182 milliseconds. In this dialog the user can generate new nets, start, stop and restart the learning. The learning is performed based on an ASCII file that was already defined by the implemented *Fetcher* at the ANN associated *Domain* instance.



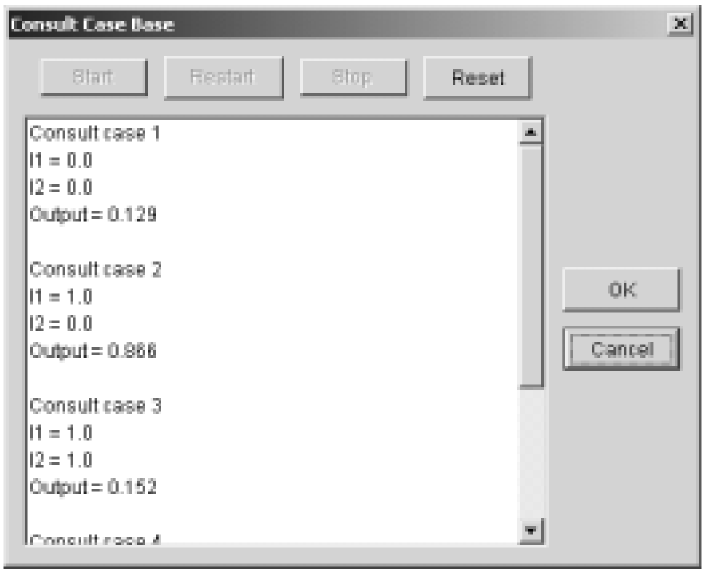
```
Generation of the neural net begun at: 997645780968 milliseconds
Generating neural net
Neural net generated
Generate neural net spent    : 20 milisecons
Generate neural net spent    : 0.02 seconds
Generate neural net spent    : 0.0 minutes
Starter learning begun at: 997645781719 miliseconds
Iteration = 0
Iteration = 100
Starter learning finished successfully
Iteration = 144
Learning spent    : 1182 milisecons
Learning spent    : 1.182 seconds
Learning spent    : 0.1 minutes
```

Figure 7.18 – Backpropagation learning the XOR problem

Figure 7.19 shows the dialog for testing a case base. It is showing a case base formed by the XOR problem being tested by the Backpropagation learned ANN. The test is performed based on an ASCII file that was already defined by the implemented *Fetcher* at the *Domain* instance. The cases show the input values for *I1* and *I2* (*Input 1* and *Input 2* evidences and *I1* and *I2* attributes of these evidences), having *0* for false and *1* for true. The ANN output result is a numeric value between 0 (false) and 1 (true). The Figure shows the network performing properly for the first three cases. In this dialog it is possible to start, stop, restart and reset the testing of the case base.

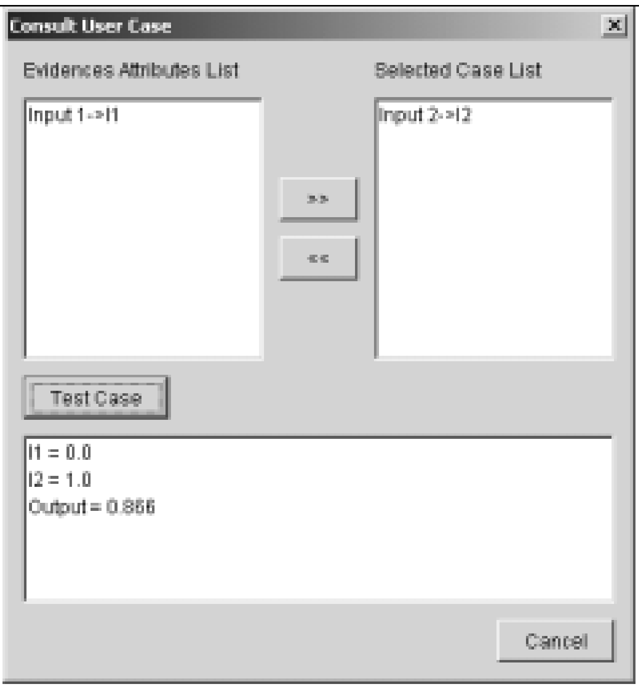
Figure 7.20 shows the testing tool/dialog where the user has the chance of building at runtime a case to be presented for the ANN. In the case, the user did not select the Input 1 meaning that this input evidence must have value zero (false) as activation. The selected Input 2 will have activation 1 (true). The evaluation of this case to the XOR learned Backpropagation network gave the output result of 0.866, that is, a value next to value one, meaning (true).

CANN does not have a dedicated component for graphical data visualization. However, it is possible to integrate visualization graphics. It enables the user to better understand the ANN data input or produced output, or even the ANN weight structure, as can be seen in the Figure 7.21, where it is possible to see the SOM weights when learning the bi-dimensional domain problem.



```
Consult case 1
I1 = 0.0
I2 = 0.0
Output = 0.129

Consult case 2
I1 = 1.0
I2 = 0.0
Output = 0.866

Consult case 3
I1 = 1.0
I2 = 1.0
Output = 0.152

Consult case 4
```

Figure 7.19 – Backpropagation testing the XOR problem



```
I1 = 0.0
I2 = 1.0
Output = 0.866
```

Figure 7.20 - Performing the testing of a user case

The CANN functionality and implementation aspects shown so far provide an overview of what this simulation tool is capable. The system is not complete in the sense of GUI facilities but already incorporates some important aspects, such as the ANN GUI framework that makes the simulation operation unified for any ANN model. Data and ANN visualization facilities appear in very simple forms and or are built programmatically.

The strongest part of the CANN framework is its flexibility. The component frameworks are tied together on the tool in a way that the programmer can easily create new ANN models and include them to the environment. The component framework offers different levels of reusability and strong deploy and portability capabilities, due to its standard implementation as JavaBeans. The simulation environment is able to manage several problem domains running different ANN models. It is also possible to include new ANN components at runtime. There is an easy way of accessing learning and testing data via specific user configured components. The data can also be converted by the ready-made converter components or by extended converter components. There is no database access component implemented. The simulation environment scales well, it is able to run ANN's that allocate huge amounts of memory or CPU processing. Thanks to the Java portability, memory management and threads implementation, the system is able to properly allocate the machine resources in order to support big ANN structure and long learning simulations.

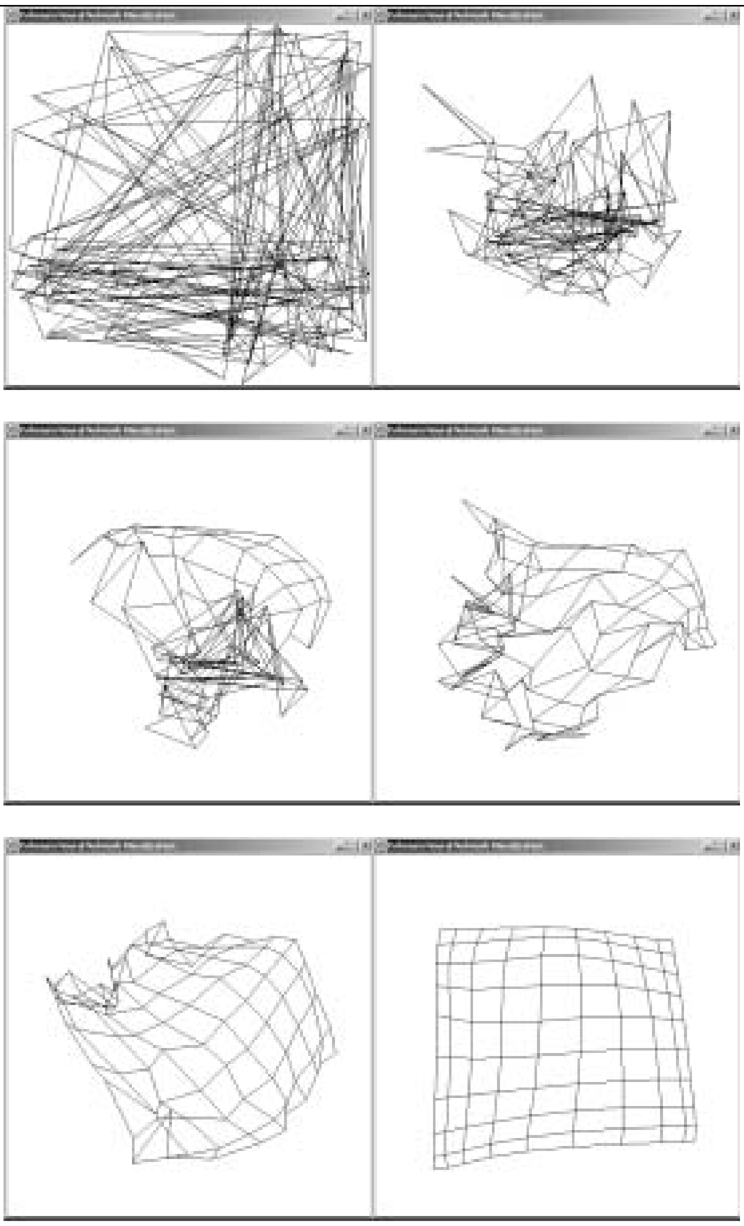


Figure 7.21 – SOM learning graphics (6 snapshots)

The CANN simulation environment is not a finished system, having much to evolve. However, its core characteristics presented here show that it already supports some of the most desirable characteristics for ANN simulation. Next, two other simulation tools are analyzed in order to compare with the CANN and give hints on what can be done in its future developments. Table 7.2 resumes the CANN simulator characteristics.

Table 7.2 – Resuming CANN characteristics

Characteristic	CANN
Data visualization	✓
Visual programming	
GUI framework	✓
High level of abstraction	✓
Reusability	✓
Deploy facilities	✓
Testing facilities	
Several ANN and domain	✓
ANN components at runtime	✓
System and components portability	✓
Data Access – ASCII, Database	✓
Manipulation/conversion facilities	✓
Domain knowledge modeling	✓
Distribution facilities	✓
Simulation and ANN parallelism	✓

7.2 Analysis of ANN simulators

There are some well-known ANN simulation environments available academically and commercially (see Section 7.1). The intention here is not to be extensive in analyzing as many tools as possible, but to pick from these well-known ones some that are up-to-date and that implement software engineering characteristics that make them similar/competitive to the CANN principles and ideas. In that way, it is important to understand in which aspects they differ, when one or the other makes better or worse implementation choices, and what CANN can learn from those tools. The first analyzed tool is the ECANSE (Environment for Computer Aided Neural Software Engineering), which is developed by SIEMENS and is commercially available. The second is the ABLE (Agent Building & Learning Environment), a research project at IBM.

7.2.1 ECANSE (Environment for Computer Aided Neural Software Engineering)

The ECANSE demo version 2.02.2 is analyzed. It is a visual development, simulation and testing tool where the user can create mathematical simulations including fuzzy sets, ANN and genetic algorithms. The ready-made ANN models are SOM, Backpropagation, Hopfield and RBF (Radial Basis Functions).

The ECANSE objects are represented visually on the simulation environment as can be seen in Figure 7.22. A component has input and output blocks that permit the connection among them using the connector objects. Data flowing among objects can be two types: the C basic data types or pre-defined derived vector types. A configuration is a combination of the objects in a data flow as exemplified in Figure 7.22. The data flowing through the connectors can suffer the necessary transformations to be properly treated by the next object on the flow.

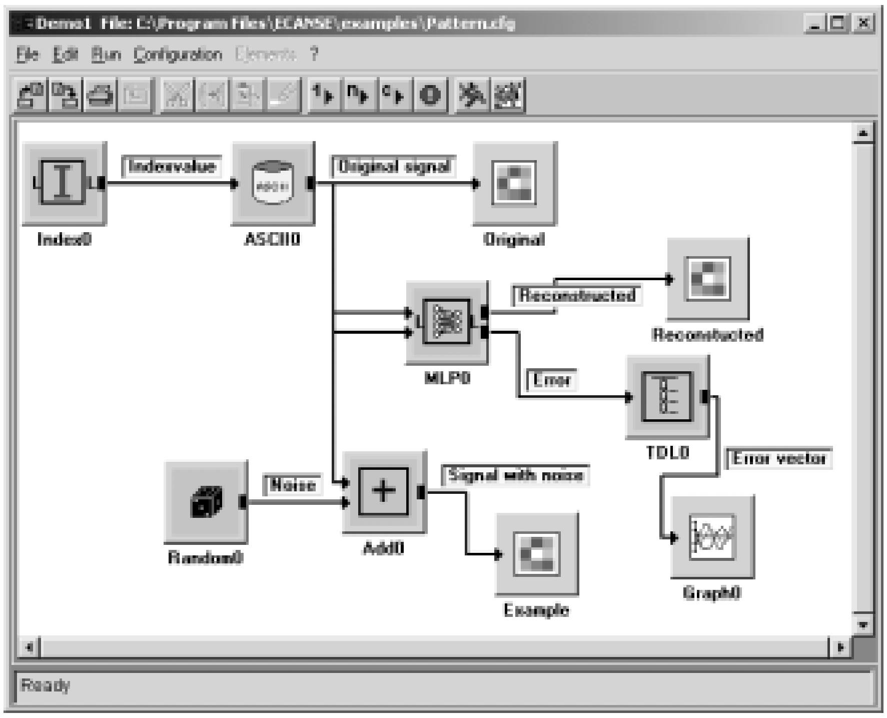


Figure 7.22 - ECANSE visual simulation environment

Each object can be parameterized when created. For example, the learning parameters can be set while instantiating an ANN component. There are parameters that may change in runtime. The overall functionality is based on time discretization. Each element on a configuration has its time control (time step). Learning and testing phases can be defined and separate data for each phase can be defined as well.

ECANSE was developed in C++, taking care of OO concepts and reusability. The programmer’s version allows to extend the system. Batch programming is available to automate already finished work, which means, to make simulations as a batch job. The system is available for Windows and Unix machines.

ECANSE is based on a system kernel that provides object-oriented mechanisms to derive classes from the kernel classes. It has two main classes from which the others shall derive: the *AnyObject* and the *EcanseObject*. The *AnyObject* implement the ECANSE mechanisms that are required for simulation (so called *static* mechanisms), such as load, save, copy, paste, parameterization and visualization. The *EcanseObject* derives from *AnyObject* and takes care of the dynamic mechanisms required for simulation such as the definition of object input and output, connecting outputs to inputs and executing an algorithm.

Those static and dynamic mechanisms of ECANSE are generic, working for all objects derived from *AnyObject* or *EcanseObject*. Such derived class describes its attributes and methods based on those generic descriptions. As a consequence, it easily integrates into the ECANSE environment. This mechanism ECANSE implements is similar to the concepts of interface and abstract classes on the Java Language that are extensively used to implement the generic solution of the CANN Framelets.

ECANSE implements a complete framework for building the visualization of its components. It provides generic means for implementing the visual control of object parameters and methods execution. By implementing specific interfaces any object can be included and manipulated on the visual simulation environment.

Derived *EcanseObjects* have to implement two specific methods that are responsible for the object simulation capabilities. Those methods are: *algorithm()* and *reset()*. The *algorithm()* methods defines what the object performs during a simulation step, typically the outputs are calculated from the inputs, the internal values or state variables and the parameters. The *EcanseObject* implements a synchronization mechanism that warrants the appropriate access to the input and output values in a simulation flow. The programmer also has to be aware of what variables and parameters he/she can or cannot modify inside the *algorithm()* method.

The *reset()* method resets the EcanseObject internal variables. The user must implement this method in order to initialize any user-controlled variable. Besides this, resetting the object results in the initialization of the synchronization for data transfer and the setting of time counter to zero.

Neural Networks ECANSE objects also shall implement some methods that are useful for controlling its learning and testing phases such as: *is_learning()*, *is_end_epoche()*; *is_begin_epoche()*; *set_learning()* and *set_testing()*. The programmers manual provides a template for adaptive objects where the programmer can easily add its own code for creating ANN objects by implementing the methods to override.

The ECANSE open API does provide objects at a high level of abstraction, in general math functions or complete ANN models. There are no objects that could help with the creation of new ANN models. The available components have a black box reusability nature. The user/developer can simply use them on the visual programming environment only by understanding its public interface, not being necessary to understand the inner code.

There are no facilities to deploy the ECANSE components in other systems or applications. The components were built to run only inside its simulation environment. This aspect compromises the applicability of the resulting learned ANN's very much.

ECANSE implements objects to perform tasks such as selecting data from a file and transforming it to be accessed by the ANN. The manipulation of ASCII file is very rich but there are no alternatives for direct access to databases. The problem domain modeling is implemented using those data access objects. The data are organized by the data access component and the ANN component input is configured in a way to be prepared for receiving this modeled data. ECANSE approach for implementing data fetching and domain modeling is data centered, being a very simply approach. Nevertheless, it is very complete and provides very good reusability. It forms a good example for CANN evolution of its equivalent components.

The ECANSE environment GUI facilities are absolutely satisfactory. Its visual programming environment is very easy to use and intuitive. It includes components for visualizing data as 2D and 3D graphics, besides enabling the visualization of the ANN inner data such as the neurons weights.

ECANSE offers parallelism at the session level the same way CANN does. In ECANSE more than one ANN can run in parallel by creating two *configurations*, which is how a visual program is called inside the simulation environment. The two configurations may contain instances of completely independent objects that can run simultaneously.

There is no information about how the ANN models were implemented, whether they have implemented parallelism inside the ANN structure. There are also no ways for running the simulations in a distributed way such as the ones implemented in CANN. Another concept that is not implemented on the simulation environment is the possibility of including new ANN components at runtime. Table 7.3 resumes the ECANSE simulator characteristics.

Table 7.3 – Resuming ECANSE characteristics

Characteristic	ECANSE
Data visualization	✓
Visual programming	✓
GUI framework	✓
High level of abstraction	✓
Reusability	✓
Deploy facilities	
Testing facilities	
Several ANN and domain	
ANN components at runtime	
System and components portability	✓
Data Access – ASCII, Database	✓
Manipulation/conversion facilities	✓
Domain knowledge modeling	
Distribution facilities	
Simulation and ANN parallelism	✓

7.2.2 ABLE (Agent Building and Learning Environment)

ABLE (IBM, 2000) is a research project at the IBM T.J. Watson Research Center (http://www.watson.ibm.com/). The evaluated ABLE version is the 1.2a (December 7, 2000). The main ABLE goal is to build hybrid intelligent agents (*AbleAgent*) that include both reasoning and learning. It provides a framework for constructing components (*AbleBeans*) that implement intelligence coming from specific Artificial Intelligence algorithms including ANN. It also includes an IDE (the *AbleEditor*) for building the agents. The core *AbleBeans* includes beans for reading and writing data from text files, for data transformation and scaling. ABLE is developed in Java and its components are standard JavaBeans.

The *AbleAgent* is an encapsulated application program that is built using *AbleBeans* and aggregates data, property, and event connections. Therefore, an *AbleAgent* is said to be a container of *AbleComponents*. There are some predefined ANN components such as Backpropagation, SOM and RBF. The ANN components can be plugged to data components on the *AbleEditor* in order to perform simulations. There are specific beans to import and export data from/to text files and perform the necessary data transformations to apply to the ANN components. *AbleAgents* can be serialized, so that the user can save them using the *AbleEditor*.

ABLE is a toolkit for developing and deploying hybrid intelligent agents and agent applications. The agents are considered hybrid because they can combine different methodologies such as fuzzy sets, ANN's, genetic algorithms and rule-based systems.

AbleAgents are said to be autonomous software components because an *AbleAgent* can run on its own thread of control or can be called synchronously by another agent or process either through a direct method call or by sending an event. *AbleAgents* are situated in their environment through the use of sensors and effectors, which provide a generic mechanism for linking them to Java applications.

AbleAgents can be accessed remotely using Java RMI. However they have no mobility capabilities, so they can't be distributed at runtime in order to use the machine resources available on the network. As part of the ABLE project, there is a plan to build a FIPA-compliant agent platform in Java. FIPA, the Foundation for Intelligent Physical Agents (http://www.fipa.org), is an international standards body working toward agent interoperability.

The ABLE user can extend the system core classes like text file I/O, data transformation, neural networks, and fuzzy and Boolean reasoning. By packaging the extended classes as beans in a Java Archive (JAR) file, it can be plugged into the *AbleEditor* to build and debug an application consisting of multiple beans and connections.

The user is able to extend the framework by extending the *AbleObject* or *AbleDefaultAgent* base classes. The beans integration is quite flexible having possibilities for tight or loose integration. In the first case, using method calls and running on the application's thread of control, and in the second case, using event passing and some or all of the beans could have their own thread of control. Data can be shared between beans by accessing bean properties held in the container agent, or data can be passed between beans using the data flow (buffer) connections.

The major design aspects that developers have to take care of while extending the ABLE agents are:

- Threading – The user shall decide if the bean will have its own threads.
- Data flow – Deciding if the data will be passed by properties or global data, via notification or action events.
- Processing flow – The agent processing can be controlled by the default wiring of data flow or can be hard-coded inside the extended agent.

ABLE implements parallelism at the session level. In the implemented architecture, the agent component can handle its own thread of control so that the simulation of the ANN is independent from the simulation environment control. As the thread control can be completely implemented by the extended *AbleAgent* it is probably possible to implement parallelism inside the ANN model architecture as well. However, there are no references in the ABLE manuals about already implemented parallelism on the ready-made ANN components.

The ABLE ANN components are defined at the abstraction level of ANN model. There are no lower level ANN components to be used as basic building blocks to facilitate the construction of other ANN models. Therefore, the reusability at the level of ANN structure and algorithm construction is minimal when it is necessary to build a new ANN component. ABLE has a very complete tutorial on how to extend its components and good quality of components documentation.

The GUI (*AbleEditor* – Figure 7.23) offers the opportunity to plug together the *AbleBeans* in order to create *AbleAgents* for simulation. The editor is not very user friendly as a visual programming environment. The gluing of the components is not intuitive because the ways to connect the components are not visually explicit; it is necessary to navigate in menus to find the connection possibilities. The components have visual inspections that include graphics for visualizing the flow of data in different kinds of charts and network graphic representation.

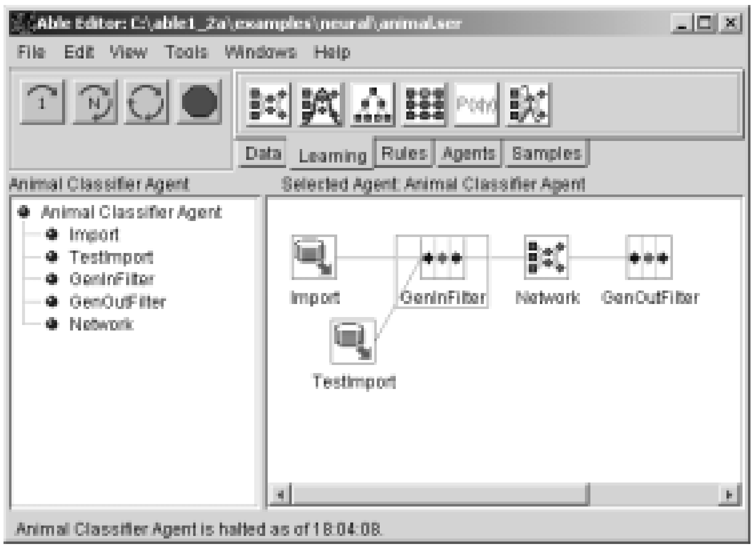


Figure 7.23 – Able Editor

New components can be added to the *AbleEditor* at runtime as *.jar* files including ANN components. However, it is not possible to have more than one running simulation (agent) simultaneously. The mechanisms for data access and transformation are based on components that import or export data from text files and components (filters) that permit the domain problem mapping and the transformation of the input data into appropriate values for the ANN. Those components have a pretty similar behavior to the correspondent CANN components. Table 7.4 resumes the ABLE simulator characteristics.

Table 7.4 – Resuming ABLE characteristics

Characteristic	**ABLE**
Data visualization	✓
Visual programming	✓
GUI framework	✓
High level of abstraction	✓
Reusability	✓
Deploy facilities	✓
Testing facilities	
Several ANN and domain	
ANN components at runtime	
System and components portability	✓
Data Access – ASCII, Database	✓
Manipulation/conversion facilities	✓
Domain knowledge modeling	
Distribution facilities	
Simulation and ANN parallelism	✓

7.3 Conclusion

By evaluating the two component and simulation environments, it is possible to clearly define the unique characteristics of CANN and the characteristics that the analyzed tools have also implemented. The Table 7.5 schematically shows the comparison of the CANN and the two simulation environments. The other tools were clearly developed taking into consideration the object-oriented approach, having design and reusability issues as primary goals, like CANN did. It is possible to consider that the main characteristics that CANN includes are:

- The component frameworks for building ANN models and simulation.
- The domain and ANN integration plus the possibility of running several combinations of the two.

- The parallelism and distribution facilities.

In general, when a project is focused on developing a simulation environment performance is not the major design goal, but the software engineering aspects. In the development of CANN the performance was not neglected because parallelism and distribution design aspects were studied and implemented. The construction of the CANN simulation tool reflects the existence of the two aspects, for instance it includes facilities such as plugging new ANN components at runtime, executing different ANN models at the same time and distributing the ANN components to run on the networked computers.

Table 7.5 – Software characteristics and the analyzed related work

Characteristic	CANN	ECANSE	ABLE
Data visualization	✓	✓	✓
Visual programming		✓	✓
GUI framework	✓	✓	✓
High level of abstraction	✓	✓	✓
Reusability	✓	✓	✓
Deploy facilities	✓		✓
Testing facilities			
Several ANN and domain	✓		
ANN components at runtime	✓		
System and components portability	✓	✓	✓
Data Access – ASCII, Database	✓	✓	✓
Manipulation/conversion facilities	✓	✓	✓
Domain knowledge modeling	✓		
Distribution facilities	✓		
Simulation and ANN parallelism	✓	✓	✓

The two analyzed tools have implemented characteristics that are not implemented or considered in the CANN such as:

- Visual programming environment – A plug and play environment where the ANN models and the simulations can be developed by visually plugging the available components together.

- Data visualization – Components for visualizing data as text or graphics such as the ANN input and output values, weight values, error signals, etc.

As a result, the future research and developments related to the CANN simulation environment could be concentrated on improving the CANN best capabilities and what is still missing or appears to be very useful by the evaluation of the other two tools. The two main areas would be:

- Evolving the ANN components to be as complete as possible on its extension and integration capabilities specially by polishing the ANN and simulation frameworks. This effort should be complemented by the creation of a visual programming environment where those components should be plugged together at runtime as it was already done by ECANSE and ABLE.

- Evolving the distribution capabilities towards building an intelligent agents simulation environment where concurrent and distributed agent applications could be developed. The intelligent agents can include not only ANN based agents but also agents whose inner intelligence can be implemented by other algorithms.

Furthermore, there are some implementation issues that could be added to help with some important activities improving CANN functionality. The use of XML files could happen in many parts of the system such as:

- XML based knowledge description of the domain model (Olson and Kent, 1997) – The domain could be created either via the GUI interface or a XML file that have a textual description of the problem domain at hand. The simulator should be able to import or export domain models from and to such an XML file. This could make the construction of the domain knowledge very intuitive. Persisting the domain as an XML file allows for easy communication of the represented knowledge, facilitating its reuse by any intelligent system.

- XML based learning and testing data – Text files is the most used form of fetching data to the ANN simulators. However, the organization of those data, positionally or using any separator, is very poor because the data can't be easily interpreted by the user or the system, and is very error prone because any missing information or wrong position can lead to completely wrong results. The use of an XML organization for these data can help with its handling for the computer systems and for the humans.

- XML files for deploying the ANN – The ANN learned structure (ANN model, number of layers, neurons, learning parameters, weight values, etc), could be also persisted in XML files. This would permit to rebuild the ANN model in any ANN simulation environment simply by correctly interpreting the ANN information persisted on the XML file. The CANN should be able to import and export such ANN representation.

- XML for deploying the ANN results – The results of the ANN simulation sometimes have to be analyzed by other systems, organized in reports, graphics, etc. The data evaluated by the ANN, especially when testing from a case base, could be persisted in an XML file in order to be easily imported to other systems or programs.

- XML script programming for batch running – The batch learning may be very useful when it is necessary to make simulations during nonworking time or in background (example of XML based scripting is the ANT (), a scripting language for Java).

There are some additional aspects that could also be implemented by demand:

- Web interface for the simulator - The simulator can run associated to a web server and be accessed via web. The distributed simulations could be coordinated via this interface too.

- Component to monitor one ANN instance working - Useful when an ANN component is deployed as a standalone component in a separate system or application. This component could be connected to it in order to allow its monitoring independent of the system flow.

CANN has brought some important engineering aspects to the simulation of ANN. Many other aspects can still be explored. The frameworks implemented so far and the applied technologies show the proper path for continuing the CANN evolution.

8 Conclusions and Future Work

This chapter summarizes the work. Complementarily, it exposes the author's vision about the possible future work.

8.1 Conclusions

This work has shown that the application of framework technology leads to the construction of ANN software with appropriate flexibility. The implementation of the CANN framework based on an object-oriented design of neural network components delivered the expected software benefits. The CANN components have been used for different purposes in different systems as expected.

As already explained in Chapter 1, the CNM ANN component was applied to perform credit rating in a retail company. The optimized CNM component was the base for the implementation of a commercial data-mining toll, the AIRA (http://www.godigital.com.br) that has been largely applied in the area of personalization of web sites. The CANN simulation environment has been applied to weather forecast. In the work described in (da Rosa et. Al, 2001a and 2001b), it is applied to rare event weather forecasting at airport terminals. The CANN simulation environment also has been used as a simulation tool for implementing a VMI solution (Vendor Managed Inventory) for an E-Business company (http://www.mercador.com).

The framework design has given flexibility and reliability to those cited systems and applications. The CANN framework components goals expanded from a classificatory system with only one learning algorithm to the possibility of implementing many different learning algorithms. The design permits the straightforward application of the different ANN models to different ANN domain problems. Different data sources are easily coupled to the domain problem at hand and applied to the ANN learning and testing processes. The design also allowed the framework to add other implementation facilities such as parallelization and distribution.

Implementing parallelism requires complete control over the ANN software architecture in order to reproduce specific parallel software structures and control mechanisms. The CANN framework offered the proper structures and mechanisms to implement a generic solution to simulate ANN in parallel. The first implementation of parallelism at the level of each synapses (Weight parallelism) proved to be too fine-grained, leading to performance problems. The second solution implementing parallelism at the level of *Training session parallelism* was appropriate for the given architecture. In such parallelism, different ANN instances run in parallel sharing the CPU resources without degrading its performance.

The architecture of each ANN model defines its possible parallel solutions. As the architectures differ very much from model to model, it is very difficult to have a generic parallel solution. However, the CANN framework facilitates the parallel implementation by giving exact entry points for implementing thread control. There is a clear separation of ANN architectural parts such as neurons, synapses and management components. Given those facilities, it was straightforward to implement a parallel solution for the CNM model. This implementation is unique; there are no other parallel implementations for the CNM model so far. A positive consequence of having a specific parallel solution for the CNM model is that it performs properly, leading to performance improvements. The CNM parallel solution can accommodate the usage of the available hardware resources leading to a better hardware usage during the learning and testing processes. The tests proved that it is possible to create an appropriate number of threads in order to get the best results from the number of available CPU's.

Adding mobility to the ANN implementations in CANN was quite a straightforward task basically because of the use of the Java language and the object architecture of the CANN framework. The clear definition and separation of the objects that form the architecture helped to choose where and how to implement the distribution without changing the implementation done so far.

Unfortunately, the performance of the CANN distribution solution still is not in an adequate level. The time for performing the ANN learning and testing on the remote machines are significantly worse than the time spent when running on the local machine. Further improvements of the distributed solution shall be done specially concentrated on avoiding as much communication between the local and remote components as possible.

Even with performance limitations, the presented solution for the distribution of ANN objects opens several application possibilities. It is possible to develop applications where it is necessary to migrate the ANN code to perform artificial intelligent tasks in remote machines such as classification, forecasting and clustering.

Another important contribution of this work was the proposition and implementation of improvements on the CNM algorithm. The optimizations presented have significantly reduced the generation of the combinatorial layer of the CNM model. In the approach presented here, relevant findings are separated in a subset for each hypothesis, reducing the number of findings to be considered, and nonsense combinations are avoided. A major search space reduction has been achieved, as the generation of combinations is controlled in order to avoid the pre-generation of all possible combinations for a given combination order. The adequate software architecture of the CANN framework makes it possible to consider each detail in the sense of best using the computational resources to make the CNM model applicable.

Finally, the CANN CNM component implements the two algorithms, the original and the optimized. It is important to reinforce/summarize the results obtained by the optimized algorithm proposed here when compared with the original one:

6. About 1/3 of the time for learning.
7. About 1/2 of the memory used.
8. About 1/3 of the combinations generated.
9. Combination order up to 5.
10. Same classification quality.

Based on the construction of the CANN framework, a complete ANN simulation environment was built. The CANN simulation environment was used to perform the tests of each implemented ANN component. It was also used for evaluating the parallelism and distribution solutions besides the CNM optimized algorithm. It has also been applied to different areas from weather forecasting to web sites personalization.

In general, when a project is focused on developing a simulation environment, the performance is not the major design goal, but the software engineering aspects. In the development of CANN it did not happened, the performance was not neglected because parallelism and distribution design aspects were studied and implemented. The construction of the CANN simulation tool also includes facilities such as plugging new ANN components at runtime, executing different ANN models at the same time and distributing the ANN components to run on the networked computers.

The main characteristics that CANN includes are:

- The component frameworks for building ANN models and simulation.
- The domain and ANN integration plus the possibility of running several combinations of the two.
- The parallelism and distribution facilities.

CANN has brought some important engineering aspects to the simulation of ANN. Many other aspects can still be explored. The frameworks implemented so far and the applied technologies show the proper path for continuing the CANN evolution.

As a corollary, based on the above conclusions it is possible to say that this work has achieved its goals, which are:

- Come up with a flexible and efficient design for ANN implementation.
- Give hints on how to better develop ANN software.
- Come up with contributions on how to implement ANN parallelism in software and code mobility for ANN architectures in order to provide ANN execution in a distributed system.
- Promote contributions to ANN models improvements.

8.2 Future Work

The current design and implementation of CANN may be considered as a generic decision-making system based on neural networks. An ambitious goal would be to enhance the framework further, so that other decision-support problems can be supported. Also ambitious would be to allow the implementation of other learning mechanisms that do not rely only on neural networks, such as machine learning algorithms.

The parallel implementation of CNM is intrinsically implemented in its architecture and cannot be extended to other ANN models. It would be an important future implementation experiment trying to add specific parallel solutions to other ANN models inside the CANN framework such as the BackPropagation or SOM.

The distribution solution should be improved in order to achieve better performance results. Additional topics that could be considered in future work on distribution include:

- Implement different distribution solutions for each of the considered ANN models.

- Add knowledge representation and communication features to the object agents using a knowledge communication language.

The future research and developments related to the CANN simulation environment could be concentrated on improving the CANN best capabilities and what is still missing or appears to be very useful by the evaluation of the other two tools. The two main areas would be:

- Evolving the ANN components to be as complete as possible on its extension and integration capabilities specially by polishing the ANN and simulation frameworks. This effort should be complemented by the creation of a visual programming environment where those components should be plugged together at runtime such as already done by ECANSE and ABLE.

- Evolving the distribution capabilities towards of building an intelligent agents simulation environment where concurrent and distributed agent applications could be developed. The intelligent agents can include not only ANN based agents but also agents whose inner intelligence can be implemented by other algorithms.

The CANN simulation environment could also improve by having some user facilities, such as:

- Visual programming environment – A plug and play environment where the ANN models and the simulations can be developed by visually plugging the available components together.

- Data visualization – Components for visualizing data as text or graphics such as the ANN input and output values, weight values, error signals, etc.

Furthermore, there are some implementation issues that could also be added to help some important activities improve CANN functionality. The use of XML files could happen in many parts of the system such as:

- The domain description of the problem.
- The description and persistence of the ANN structure.
- The persistence of the learning and testing data.

There are some additional aspects that could also be implemented by demand:

- Web interface for the simulator.
- Component to monitor one ANN instance working.

9 References

ABLE HTML Documentation. http://www.alphaworks.ibm.com/tech/able. IBM, (2000).

Aglets Software Development Kit. http://www.trl.ibm.com/aglets/ (IBM, 2000)

Agrawal, R.; Imielinski, T. & Swami, A. (1993). Mining association rules between sets of items in large databases. In: Proceedings of the *ACM SIGMOD Conference on Management of Data*, 207-216. Washington, DC.

Agrawal, R.; Mannila, H.; Srikant, R.; Toivonen, H. & Verkamo, A. I. (1996). Fast discovery of association rules. *Advances in knowledge discovery and data mining*. Cambridge: AAAI Press/The MIT Press.

Anderson, J., (1995). *An Introduction to Neural Networks.* MIT Press.

Beckenkamp, F.G. and Pree, W., (1999). Neural Network Framework Components. Book chapter in Fayad M., Schmidt D.C. and Johnson R. editors, Object-Oriented Application Framework: Applications and Experiences, John Wiley.

Beckenkamp, F.G., and Pree, W., (1999). Neural Network Framework Components. Fayad M., Schmidt D.C. and Johnson R. editors, *Object-Oriented Application Framework: Applications and Experiences,* Volume 2, John Wiley.

Beckenkamp, F.G., and Pree, W., (2000). Neural Networks Components. *Neural Computation 2000*, NC'2000. May 2000, Berlin, Germany. http://www.icsc-naiso.org/

Beckenkamp, F.G.; Pree, W. and Feldens, M. A. (1998). *Optimizations of the Combinatorial Neural Model.* 5th Brazilian Symposium on Neural Networks (SBRN'98). Belo Horizonte, Dezember 1998, IEEE.

Blurock, Edward S., (1998). ANALYSIS++: Object-Oriented Framework for Multi-Strategy Machine Learning Methods. *ESPRIT Project 22897*, UNI-SOFTWARE PLUS, A-4232 Hagenberg, Austria. ftp://ftp.risc.uni-linz.ac.at/pub/techreports/1998/98-12.ps.gz.

Camp, D. van. (1993). *A UserGuide for the Xerion Neural Network Simulator*, Version 3.1, Department of Computer Science, University of Toronto.

Coad and Jourdon (1991). *Object-Oriented Design.* Yourdon Press Computing Series. ISBN: 0136300707

Da Rosa, S.I.V., Beckenkamp, F.G., and Hoppen, N. (1997). The Application of Fuzzy Logic to Model Semantic Variables in a Hybrid Model for Classification Expert Systems. Proceedings of the *Second International ICSC Symposium on Fuzzy Logic and Applications* (ISFL'97). Zurich, Switzerland

Da Rosa, S.I.V., Leão, B.F., and Hoppen, N. (1995). Hybrid Model for Classification Expert System**.** Proceedings of the *XXI Latin American Conference on Computer Science.* Canela, Brazil.

Da Rosa, S.I.V; Burnstein, F and Beckenkamp, F.G. (2000). An empirical study of distribution based on Voyager: A performance analysis. HICS'2000.

Dawson, C.K. O'Reilly, R.C. and McClelland**,** J.L. (1997). *The PDP++ Sofware Users Manual*, Version 1.2. Technical report, Carnegie Mellon University.

Denis, F.A.R.M. and Machado, R.J. (1991). *O Modelo Conexionista Evolutivo.* Rio de Janeiro: IBM – Rio Scientific Center (Technical Report CCR - 128).

ECANSE – User's Manual. SIEMENS AG, (1998).

Feldens, M.A. and Castilho, J. M. V. (1997). Data mining with the combinatorial rule model: an application in a health-care relational database. In: XXIII CLEI. Valparaíso, Chile.

Fontoura, M., Pree, W. and Rumpe, B. (2000). UML-F: A Modeling Language for Object-Oriented Frameworks. *European Conference on Object-Oriented Programming* (ECOOP'2000), Sophia Antipolis and Cannes, France.

Fontoura, M., Pree, W. and Rumpe, B. (2001). *The UML-F Profile for Framework Architectures.* To be published by Addison-Wesley/Pearson Education in fall 2001.

Freeman, J. A. and Skapura, D. M. (1992). *Neural Networks: Algorithms, Applications, and Programming Techniques.* Addison-Wesley.

Fuggetta, A., Picco G. P. and Vigna G. (1998). *Understanding Code Mobility.* IEEE Transactions on Software Engineering, Vol. 24.

Gamma, E., Helm R., Johnson R. and Vlissides J. (1995). *Design Patterns—Elements of Reusable Object-Oriented Software.* Reading, Massachusetts: Addison-Wesley

Goddard, N.; Lynne, K.; Mintzk T. and Bukys, L. (1989). *Rochester Connectionist Simulator*, Technical Report TR 233 (revised), Dep. of Comp. Sci., University of Rochester.

Grossberg,. S. (1987). Competitive Learning: From Interactive Activation to Adaptive Resonance. Cognitive Science. Vol. 11, p. 23-63.

Guazelli, A. and Leão, B.F. (1994). Incorporating semantics to ART. IEEE International Conference on Neural Networks. vol.3, p.1726-1731 : il. Piscataway 1994.

Hammerstrom, D. (1990). A VLSI architecture for High-Performance, Low-Cost, On-Chip Learning, *In Proc. International Joint Conference on Neural Networks*, pages II--537--543.

Haykin, S., (1994). *Neural Networks A Comprehensive Foundation.* Upper Saddle River, NJ, Prentice-Hall

Hecht-Nielsen, R. (1989). *Neurocomputing.* Addison-Wesley.

Hopfield, J.J. (1982). Neural networks and phisical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences of the U.S.A.* 79, 2554-2558.

Hopp, H. and Prechelt, L. (1997). CuPit-2: A Portable Parallel Programming Language for Artificial Neural Networks, *Proc. 15th IMACS World Congress on Scientific Computation*, Modelling, and Applied Mathematics, Berlin, Germany.

Hwang, Kai and Xu, Zhiwei. (1998). *Scalable Parallel Computing.* MacGraw-Hill.

Jabri, M. Tinker, E. and Leerink, L. (1994). Mume: A multi-net multi-architecture neural simulation environment. *Neural Network Simulation Environments.* Editor Josef Skrzypek,, pages 229-- 247. Kluwer, Norwell:MA.

Kock, Gerd and Serbdzija, Nikola B. (1994). Artificial Neural Networks: From Compact Descriptions to C++, *Proc. of the Int. Conference on Artificial Neural Networks* (ICANN'94). Pp. 1372-1375.

Kock, Gerd and Serbdzija, Nikola B. (1996). *Simulation of Artificial Neural Networks.* SAMS, Vol. 27, pp. 15-59.

Kohonen, T. (1982). Self-organized formation of topologically correct feature maps. *Biological Cybernetics* 43, 59-69.

Kosko, B. (1988). Bidirectional associative memories. *IEEE Transactions on Systems, Man, and Cybernetics.* vol. 18, pp. 49-60, Jan/Feb.

Kosko, B. (1992). *Neural Networks and Fuzzy Systems.* Englewood Cliffs, NJ: Prentice-Hall

Lawrence D. (1991). *The Handbook of Genetic Algorithms.* New York: Van Nostrand Reinhold

Lea, Doug. (1999). *Concurent Programming in Java.* 2nd Edition, Addison-Wesley.

Leao, B. F. and Reategui, E. (1993). Hycones: a hybrid connectionist expert system. Proceedings of the *Seventeenth Annual Symposium on Computer Applications in Medical Care - SCAMC*, IEEE Computer Society, Maryland.

Leão, B. F. and Reátegui, E. (1993a). A hybrid connectionist expert system to solve classificational problems. Proceedings of *Computers in Cardiology* , IEEE Computer, IEEE Computer Society, London.

Leão, B. F. and Rocha, A. F. (1990). Proposed Methodology for Knowledge Acquisition: A Study on Congenital Heart Disease Diagnosis. *Methods of Information in Medicine*. 29(1), p. 30-40.

Linden, A. Sudbrak, Th. Tietz, Ch. and Weber, F. (1993). An Object-Oriented Framework for the Simulation of Neural Nets. *In Advances in Neural Information Processing Systems 5*, pages 797--804, San Mateo, Ca. Morgan Kaufmann Publishers.

Log4J project. http://jakarta.apache.org/log4j/docs/index.html (Apache Software Foundation)

Machado, R. J. and Rocha, A. F. (1990). The combinatorial neural network: a connectionist model for knowledge based systems. In B. Bouchon-Meunier, R. R. Yager, and L. A. Zadeh, editors, *Uncertainty in knowledge bases*. Springer Verlag.

Machado, R. J., Barbosa V. C. and Neves P.A. (1998). Learning in the Combinatorial Neural Model. *IEEE Transactions on Neural Networks*. Vol. 9, No. 5.

Machado, R.J. and Rocha, A.F. (1989). *Handling Knowledge in High Order Neural Networks: The Combinatorial Neural Model*. Rio de Janeiro: IBM Rio Scientific Center (Technical Report CCR076).

Machado, R.J. and Rocha, A.F. (1991). The combinatorial neural network: a connectionist model for knowledge based systems. In B. Bouchon-Meunier, R. R. Yager, and L.A. Zadeh, editors, *Uncertainty in Knowledge Bases*. Springer Verlag.

Machado, R.J. and Rocha, A.F. (1992). Evolutive fuzzy neural networks. Proceedings of the *IEEE International Conference on Fuzzy Systems*.

Masters, T. (1993). *Practical Neural Networks Recipes in C++*. Academic Press.

Medsker, L. R. and Bailey, D. L., (1992). Models and Guideliness for Integratig Expert Systems and Neural Networks. In: Kandel A. & Langholz G. *Hybrid Architectures for Intelligent Systems*, CRC Press.

NeuralWorks Reference Guide, NeuralWare, Inc. (1995).

Olson, D. and Kent, R.E. (1997). *Conceptual knowledge markup language*, an XML application. Unpublished presentation, given at the XML Developers Day, August 21, 1997, Montreal Canada.

Prechelt, Lutz. (1994). CuPit – A Parallel language for Neural Algorithms: Language Reference and Tutorial. *Technical Report 4/94*, Fakultät für Informatik, Universität Karlsruhe, Germany.

Pree, W. (1991). *Object-Oriented Versus Conventional Construction of User Interface Prototyping Tools* (PhD thesis), Publisher: Verband der wissenschaftlichen Gesellschaften Österreichs (VWGÖ), Vienna.

Pree, W. (1995). *Design Patterns for Object-Oriented Software Development*. Reading, MA: Addison-Wesley/ACM Press.

Pree, W. (1997). Object-Oriented Design Patterns and Hot Spot Cards. *IEEE International Conference on the Engineering of Complex Computer Systems* (ICECCS'97), Como, Italy.

Pree, W. (2000). Hot-Spot-Driven Framework Development. *Building Application Frameworks: Object-Oriented Foundations of Framework Design* (ed.: M. Fayad, D. Schmidt, R. Johnson), Wiley & Sons, New York City.

Pree, W. and Koskimies, K. (1999). Framelets—Small and Loosely Coupled Frameworks. *ACM Computing Survey Symposium on Application Frameworks* (Ed.: M. Fayad).

Pree, W. and Koskimies, K. (2000). Framelets—Small is Beautiful. *Building Application Frameworks: Object-Oriented Foundations of Framework Design* (ed.: M. Fayad, D. Schmidt, R. Johnson), Wiley & Sons, New York City.

Pree, W., Althammer, E. and Sikora, H. (1998). Framelets als handliche Architekturbausteine. *Softwaretechnik '98*. Paderborn, Germany.

Pree, W., Beckenkamp, F. and da Rosa, S.I.V. (1997). *Object-Oriented Design & Implementation of a Flexible Software Architecture for Decision Support Systems.* 9th International Conference on Software Engineering and Knowledge Engineering - SEKE'97. Madrid, June 1997.

Ramacher, U. (1992). SYNAPSE - A Neurocomputer that Synthesizes Neural Algorithms on a Parallel Systolic Engine. *Journal of Parallel and Distributed Computing*, 14:306--318.

Reátegui, E. and Campbell, J. (1994). A classification system for credit card transactions. In Haton, J-P., Keane, M., Manago, M. (eds). *Advances in Case-Based Reasoning. Second European Workshop EWCBR-94.* Chantilly, France. Springer Verlag

Reátegui, E., Torres, R., Campbell, J. (2001). Personalizing with Recommendation Frames. *2001 ACM SIGIR Workshop on Recommender Systems*, New Orleans, EUA

Rogers, J. (1997). *Object-Oriented Neural Networks in C++*. Academic Press.

Rojas, R., (1996). Neural Networks: A Systematic Introduction. Springer-Verlag.

Rumelhart, D.E., and McClelland, J.L. (1986). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Vol. 1. Cambridge, Ma: MIT Press.

Simpson, P. K. (1992). Foundations of Neural Networks. In Sánches-Sinencio, E. and Lau, C. Artificial Neural Networks: Paradigms, Applications, and Hardware Implementations. IEEE Press.

Skapura, D. M. (1996). Building Neural Networks. Addison-Wesley.

Skillkorn, D. B. and Talia, D. (1998). Models and Languages for Parallel Computation. *ACM Computing Surveys.* June 1998, Volume 30, Number 2.

Strey, A. (1999). *EpsiloNN --- A Tool for the Abstract Specification and Parallel Simulation of Neural Networks.* Systems Analysis Modelling Simulation (SAMS), Gordon & Breach.

Szyperski, C. (1998). *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley.

The Sumatra Project. http://www.cs.arizona.edu/sumatra/ (Arizona University)

Viademonte, S., Burstein, F. (2001b). An Intelligent Decision Support Model for Aviation Weather Forecasting. The 4th International Conference on Intelligent Data Analysis (IDA 2001). Conference proceedings "Advances in Intelligent Data Analysis", LNCS 2189, Pages 278-288. Cascais, Portugal.

Viademonte, S., Burstein, F., Dahni, R., Willians, S. (2001a). Discovering Knowledge from Meteorological Databases: A Meteorological Aviation Forecast Study. Third International Conference on Data Warehousing and Knowledge Discovery (DaWaK 2001). Conference proceedings, LNCS 2114, Pages 61-70. Munich, Germany.

Vondrák, I. (1994). Object-Oriented Design of Artificial Neural Networks. *Proceedings of the IDG Czechoslovakia.* VSP International Science Publishers, Netherlands.

Vondrák, I. (1994a). Object-Oriented Neural Networks. *AI Expert*, Vol. 9(6), pg.20-25.

Weinand A., Gamma E. and Marty R. (1989). Design and Implementation of ET++, a Seamless Object-Oriented Application Framework. *Structured Programming*, 10(2), Springer Verlag.

Wirfs-Brock, R. and Johnson, R. (1990). Surveying Current Research in Object-Oriented Design. *Communications of the ACM*, 33(9).

Wooldridge, Michael, Nicholas, Jennings (1995) *Inteligent Agents: Theory and Practice.* Inteligent Agents, Springer-Verlag, Berlin.

Zell, A. et. Al. (1995). *SNNS – Stuttgart Neural Network Simulator*. User Manual, Version 4.1. Report No. 6/95.