Building Neural Network components

Fábio Ghignatti Beckenkamp and Wolfgang Pree
Software Research Lab
University of Constance
D-78457 Constance, Germany
E-Mail: *lastname*@acm.org

Abstract

The paper discusses the implementation of Artificial Neural Networks (ANN) components. In particular the proposal of an ANN component interface necessary for these components to be deployed by third parties. The paper starts analyzing different approaches to the software implementation of ANN including structured and object-oriented programming paradigms. Following the paper introduces the ANN component interface the authors consider minimal for the implementation of ANN in a component based software environment. The proposed interface offers the means for the object to implement basic ANN operations such as generating the neural network architecture or testing and learning data cases. The authors' goal, in proposing such an interface, is to offer a means for different ANN software programmers and users to easily share their ANN software solutions by using the proposed interface and by following a component standard. The paper finishes commenting the importance of having ANN component software and exemplifies some applications the already implemented ANN components have been used.

1 Introduction

The main bibliography in Artificial Neural networks generally explains the different ANN models using mathematical representation and has no code examples on how to implement in software the mathematical definitions. When the authors do some sort of code example, they are compelled to do some assumptions or reductions due to the chosen programming language or programming representation method. Those assumptions do not mean that the implementations are wrong or are not equivalent to the mathematical representation. However it may mean that, when represented as programming code, some important aspects to software construction has to be considered such as the code efficiency, the code facility in communicating the ANN method and the code usability. Furthermore when somebody wants to map the mathematical representation to computer software, the concepts can be implemented in different ways.

The author that chooses to have code examples shell assume that some readers are not familiar to the language he chooses, or that some readers does not like to work with that language or even that he/she cannot work with it. The author's programming stile has to be very clear in order not to turn the code complicated to read and consequently to understand. Finally, the reader may do not agree with some programming choices made by the author such as the definition of certain data structures or functions. These aspects turn difficult the communication among the author and the reader. One example of authors that choose to have code example is (Freeman and Skapura, 1992). They develop a complete system

infrastructure (simulation environment) to facilitate the development of different ANN. The code is done in Pascal using structured paradigm. The coding is very didactic, the style is clear and precise. The most complex programming structures used are linked lists for implementing the core ANN structures such as layers of neurons and synapses connections. But the programming paradigm they used is structured and nowadays people are interested in object-oriented solutions.

There are also few authors that directly approach the problem of constructing programming code to ANN. Some relevant attempts are (Rogers, 1997; and Masters, 1993). Masters attempts to give tips on how to implement ANN in C++. His coding is not properly object-oriented. He used classes to define ANN models but does not have more fine-grained classes to implement more diverse ANN aspects. The several aspects are implemented as methods much likely as in a structured or functional programming style. Rogers goes more deeply in the object-oriented approach, he defines ANN classes such as neurons and synapses and has the preoccupation of appropriately apply and benefit of the object-oriented characteristics such as polymorphism, inheritance and code reusability. The problem of the object-oriented approaches is the code efficiency. Object-oriented software is complex and sometimes its efficiency is dependent of appropriate design choices.

Why there is not much published research/work directly approaching the development of software for ANN? Is there no real preoccupation regarding the appropriateness of programming methods for the construction of ANN software among the ANN research community? Is it such a preoccupation a task for the ANN community? The aim of this work is not to answer these questions. They arose in this work because it focuses exactly in developing appropriate ANN code.

The construction of ANN in computer is clearly and naturally a reduction of the original model, as the definition of ANN is a reduction of the biological neural networks. For example the inherent parallelism of the ANN models is rarely implemented or considered by most of the ANN programmers. Of course this is part of the reductionism that people assumes that exists, and does not care much about it.

This reductionism reveals that communicating the characteristics of an ANN model using programming code is pretty dangerous. It is possible to induce the reader to misunderstand important aspects of ANN. This work also does not propose a new way of communicating the details of an ANN model to readers. This work, in fact, just points the problem because it is analyzing several programming solutions for ANN in order to propose a way of implementing them using the most recent programming paradigm, the components. However one important characteristic of component software is to nicely communicate what a software code is and is not able to do via interfaces. Perhaps component software is an efficient way of explaining and exploring the ANN characteristics.

The Szyperski (1998) definition of components explains how they act through interfaces: “A software component is a unit of composition with contractually

specified interfaces". Components have interfaces to communicate to other components. The interfaces are contracts among the components that allow them to communicate to each other. It is necessary that the component implement a certain contractual interface to be able to be compound by third parties. One nice characteristic is that components may be implemented in any computer language and not only in object-oriented languages as some people presume. The component interface must be respected and implemented independently of the language of choice. This means that a programmer can use his/her language of preference to create the component. It also means that there could exist different implementations for the same component. This is an important characteristic because it is possible to develop different versions of a component with different levels of complexity or efficiency. Perhaps a component may be implemented in a language that contemplates efficient memory allocation or in another that contemplates efficient parallel code execution. The user may choose to use any of the components based on his/her necessities once they properly implement the same public interface.

Nowadays there are competing components "wiring" standards such as Object Management Group (OMG) Common Object Request Broker Architecture (CORBA), Sun's JavaBeans and Microsoft's Component Object Model (COM). At the moment, it is not able to forecast which one is going to be the standard "de facto" or if there will be only one actual standard. CORBA essential definition is worried with the fundamental problem of wiring components that is how components implemented on different languages and running on different platforms interact. Its goal is to enable open interconnection of a wide variety of languages, definitions and platform. Such a very open approach do not permit the interoperation based on efficient binary level leading the definition to depend on high-level protocols that are resource expansive. The enormous consortium that forms the CORBA effort shows that this standard has an important contribution on the component usage solidification.

The Sun´s JavaBeans standard is concentrated on the Java language specification. It supports classes and interfaces where the interfaces are, in fact, fully abstract classes. The Java interfaces took out implementation inheritance, which favors object composition and message-forwarding techniques. The Java distributed model restricts remote access to interfaces and relies on RMI (Remote Method Invocation) service. Although implemented to be the Java language component standard, JavaBeans has been designed to enable the integration of a bean into container environments outside Java such as ActiveX and OpenDoc.

The COM components foundation was defined to the Microsoft's platform, but its is also available to other platforms by third parties. COM does not follow CORBA standard and contrast very much form it because it is binary standard. It means that COM can work more efficiently in the implemented platform. COM defines only interfaces making the standard completely independent of programming language. The COM distributed standard DCOM is based on the creation of client side proxy objects and server side stub objects. Microsoft's COM+ standard combines COM

with lightweight object models. COM+ is similar in scope to CORBA and even more concrete than JavaBeans because it allows the binary compatibility of components, it is a virtual-machine architecture to support dynamic languages as scripting languages. It also provides true garbage collection, and safe in-process cooperation between components.

The user shell distinguishes the different approaches, and makes his/her choices. The tendency is not to converge to one universal standard but to continue the existence of few competing standards and few bridge products among them. In this work the used standard is JavaBeans because Java is the language of choice. Anyway it is easy to use tools that automatically generate the CORBA interfaces and stubs for the JavaBeans components what makes the implementation in Java worth while. The use of the Java language restricts the solution efficiency because its performance restrictions. Anyway the goal of this work is not to have high performance ANN components but to test the generality of the proposed ANN components interface. Furthermore the component ability of mobility and multi-platform running are requirements that Java language covers with its virtual machine and RMI facilities.

2 The CANN (Components for Artificial Neural Networks) solution

The CANN is a study that implement ideas on how neural networks can be constructed on a component basis. The study evolves: the creation of basic ANN components to build any ANN model at hand; the creation of ANN components that can be reused by third parties; the construction of a simulation infra-structure that allows to plug the several ANN components and use/test them. Furthermore there is a component created to support the problem domain modeling and the data sources access to the ANN learning and testing process.

The CANN components are object-oriented designed. The core parts are done as small frameworks to improve implementation reusability and flexibility. The details on the object design is described on (Beckenkamp and Pree, 1999).

In the realm of CANN development, because of time constrains, it is not possible to approach many existent ANN models. Few models were picked up among the most important ones, based on 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 the most of the ANN models without having to implement many models.

The first chosen model was the Backpropagation (Rumelhart et al., 1986). This model was chosen because it implements a supervised learning based on error correction learning algorithm, and its architecture is feedforward multilayer and it is fully connected. 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 implemented it in this work

were twofold: the profound knowledge of the author on this model; and the special interest on the application of this model on the credit scoring problem by some European companies. The third ANN is a typical unsupervised competitive learning model, the Self-Organizing Feature Maps (SOM) (Kohonen, 1982). The SOM architecture is based in a lattice two-dimensional map. Finally, another unsupervised competitive learning model was chosen, the Adaptive Resonance Theory (ART) (Carpenter and 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.

2.1 The ANN component interfaces

Each ANN component relies in a general interface that allows them to communicate to other components of the CANN environment. One interface was created to act as the ANN components core communication: *INetImplementation*. This interface defines a set of methods that an ANN component shell implement to be able to be deployed in a third part software solution. It defines methods for general ANN tasks such as learning or testing a case, generating the ANN structure, etc. It also defines essential methods that an ANN component must implement to be able to work on the CANN distribution facility. The ANN component also implements the interface *java.io.Serializable*, which allows the component to be persisted. The Figure XX shows a schematic design of an ANN component with its two “wiring” interfaces.

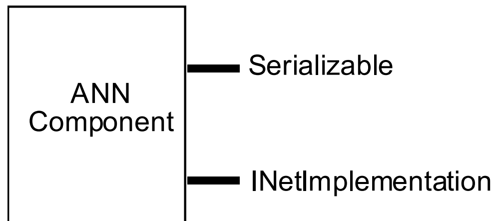

Figure 1 – ANN Component.

The code Example 1 shows the *INetImplementation* interface definition in Java.

Example 1 – *INetImplementation* interface.

```
public interface INetImplementation {
  public void setLearningParameters(Vector parameters);
  public Vector getLearningParameters();
  public int generateNet(IDomainAttributes domain)
                     throws java.rmi.RemoteException;
  public int getNetSize(IDomainAttributes domain);
  public void learnCase();
  public Vector testCase();
  public Vector getOutputActivations();
  public boolean getStopLearning();
  public void setProxies(IDomainAttributes domain)
                     throws java.rmi.RemoteException;
  public boolean restoreObjectsReferences();
}
```

The INetImplementation interface defines the basic messages an ANN component shell be required to implement. Those messages are, in fact, tasks that most ANN implementations shell do such as setting learning parameters or learning a case.

The first methods defined are *setLearningParameters(Vector parameters)* and *getLearningParameters()*. An ANN component implements these methods accordingly to its necessities. For example the parameters for a Backpropagation neural network may include the learning-rate (*eta)* and the momentum (*alpha*). The parameters are informed to the component via a java.util.*Vector* parameter of the *setLearningParameters* method and its actual state can be retrieved from the component using the *getLearningParameters* method. The *parameters* vector shell be populated with Java's *java.lang.Number* subclasses instances such as *java.lang.Integer* or *java.lang.Float*. The order of how the vector shell be populated is free to the ANN component developer. For example he/she is implementing a Backpropagation component, the Vector parameter can be generated in the sequence as the code Example 2.

Example 2 – Populating the vector *parameters*.

```
parameters = new Vector();
parameters.addElement(new Integer(250)); // number of iterations
parameters.addElement(new Integer(2));   // number of hidden neurons
parameters.addElement(new Float(0.9f));  // alpha
parameters.addElement(new Float(0.5f));  // eta
parameters.addElement(new Float(0.01f)); // stop error
setLearningParameters(parameters);
```

The ANN component developer defines the order the vector is populated and must then implement *setLearningParameters* and *getLearningParameters* methods respecting this order. The order shell be documented in a way that the component users can properly use those methods.

Following, the user can require the ANN component to generate the ANN architecture using the method *generateNet(IDomainAttributes domain)*. This method throws *java.rmi.RemoteException* because the neural network can be generated in a remote program using the distribution facility that is going to be explained later. The method returns an integer number where zero means that the generation did not succeed, any other number means that the generation succeed. This returning value can be the number of generated neurons or synapses or any other relevant information; indeed, it is free to the ANN component developer to use this returning value as he needs. The parameter *domain* must implement the interface *IDomainAttributes*. This interface can be seen in the code Example 3 below.

Example 3 – *IDomainAttributes* interface.

```
public interface IDomainAttributes extends
               java.io.Serializable, com.objectspace.voyager.IRemote {
  java.util.Vector getInputAttributes();
  java.util.Vector getOutputAttributes();
```

```
    void setInputAttributes( java.util.Vector ia );
    void setOutputAttributes( java.util.Vector oa );
  }
```

The *IDomainAttributes* is an interface created to facilitate the generation of the problem domain model. This interface defines methods to access the structures that define the learning and testing data. The data attributes at the data sources like ASCII files or databases have to be preprocessed to be useful by the neural network. Similar to the learning parameters getter and setter process previously seen, this interface has methods to populate a vector with objects of the class *Attribute*. Each attribute object in the vector will be associated to its correspondent input or output neuron on the ANN structure that shell be created by the ANN component. If the user creates 10 input attributes and 5 output attributes, the neural network will be created with 10 input neurons and 5 output neurons and the attributes will be associated to them. Attribute objects shell prepare the data from external sources so that the data can be directly fed to the input or output neurons of the ANN. Using the *Attribute* method *setActivation()*, the user can feed data into the attribute. Always the network needs data for learning or testing, it will ask its associated *Attribute* for the data via a method called *getActivation()*. So that the ANN component user shell take care of creating appropriate instances of *Attribute* and populate vectors that shell be handled by the object that implements the *IDomainAttributes* interface. The reference to this object is the way the ANN component user has to feed data to the network for learning or testing processes. Furthermore, the user also shell takes care of accessing the appropriate data files and takes care of setting the *Attributes* activation. A detailed explanation of the *Domain* framework of CANN is to long for the goal of this paper, complementary information can be found on (Beckenkamp and Pree, wiley).

The forth method on the *INetImplementation* interface is the *getNetSize(IDomain domain)*. This method simply calculates the memory footprint of a neural network based on its parameters and on the modeled domain.

Following the INetImplementation interface appears the methods learnCase() and testCase(). As the name says the methods shell be called when the user wants the neural network to learn or test a case. The case shell has been already fed into the ANN component by setting the attributes activation as explained before. Then the user just call the appropriate learn or test method that the neural network will start an interaction processing the data fed from the input neurons via its associated attributes. For instance when the neural model implements supervised learning, also the output neurons target activations shell be fed. The testCase() method returns a vector that contains a list of java.lang.String objects with the detailed result explanation. This vector also may return relevant result data, it is up to the ANN component developer to decide what is more important to return.

The method getOutputActivations() was developed to provide a means for the user to get the ANN output neurons activations at any time, before or after a learning or testing cycle. Finally the method getStopLearning() was defined to provide to the ANN component user a way to check it the learning process was already finished.

When the ANN component implements parallel learning procedure then it is necessary to check a flag that indicates that the learning has got to the end and that the user can go for another learning case or verify the ANN results.

The CANN simulation environment should be able to manage various ANN models while testing the solution of a specific problem in a distributed way. Several neural models can then run at the same time. As explained before, each ANN component that implements *INetImplementation* creates its own ANN structure and is able to handle it. Each created ANN structure can allocate significant amounts of memory and its processing can take significant CPU time. 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. The ANN components that implement the *INetImplementation* interface can be created either locally or remotely. In both cases, the instance can be moved later.

The distribution implementation is based on the Objectspace's Voyager library (www.objectspace.com). The library is written in Java and allows the creation of mobile Java code. Voyager uses Java interfaces to simplify the distribution of objects. A special proxy object that implements the same interface as the local object represents the remote object. Therefore, a variable whose static type is an interface may either refer to an instance of the actual object or a proxy object. The ANN components then are referred using the interface *INetImplementation* that allows the object to be either on the local or remote program.

The ANN component user will be responsible for implementing the Voyager mobility using its facilities such as requesting a proxy to the ANN component, moving the component back and forth, etc. Though the component moving is responsibility of the application. What the *INetImplementation* interface offers is the necessary methods to maintain the internal object references of the moved component to the rest of the application. When the ANN component is moved, Voyager makes, in fact, a copy of the object and all the objects it refers internally and move them to the remote program. Sometimes a copy of objects referred internally is not enough to maintain the proper component functionality. It is the case of the references the input and output neurons has to the domain attributes. It is important that the component located on the remote program refer to the local appropriate application attributes. Appropriate object proxies must restore those references. This will allow the moved ANN component to continue being properly controlled by the main application that is running on the local program.

The method *setProxies(IDomainAttributes domain)* shell be implemented to restore the references the ANN component has to the domain attributes. As explained before, when the ANN component is moved, the attributes references that the input and output neurons maintain are lost. Those references are restored be calling the remote ANN component method *setProxies* and passing as parameter the *domain* variable. The *domain* variable implements the interface *IDomainAttributes* that extends the voyager interface *com.objectspace.voyager.IRemote*. This interface extends the Java interface java.rmi.Remote that serves to identify all remote

objects. The methods specified in a *Remote* interface are available remotely. So that the application is able to call those methods implemented by this interface and get from the local program the proper proxies to the attribute objects. The attribute objects also implement the *Remote* interface to allow its methods to be called.

The method *restoreObjectsReferences()* does the opposite job. When an ANN component that was located on the remote program is returned to the local program its proxy references to the application objects are restored to the original objects. The ANN component developer also must implement those two methods. The ANN component user must only take care of properly calling them after moving the ANN component to a remote program or after moving it back to the local program.

3 Conclusion

The ANN interface shown here is a first draft definition of a possible ANN components interface. We invite the ANN community to analyze and construtively criticize this proposition. Our goal is to incite the ANN community to worry about the possibilities of having a wiring standard for ANN components. Such a standard would certainly facilitate ANN use and study to the public in general.

Software developers goal is to improve the applicability and the reuse of his/her software. The proper use of object-oriented and component technologies deserves these characteristics. The definition of a wiring standard for ANN software components is the initial condition in having ANN software components available to be used in the enormous variety of situations the neural networks technology can be applied. The CANN components and simulation environment have been used by different researchers and companies, here some examples:

- The ANN components were applied in a credit analysis problem for an Austrian retail company.
- The ANN components were included in a data mining tool developed by a Brazilian software company.
- The CANN simulation environment has been used in a Brazilian academic research to evaluate the applicability of the ANN models in the area of text subject recognition for the web.
- The CANN simulation environment has been considered in a research project on weather forecast in Australia.

References

Beckenkamp F. 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.

Carpenter, G. and Grossberg, S. 1987. A massively parallel architecture for a self-organizing neural pattern recognition machine. *Computer Vision, Graphics, and Image Understanding*, vol. 37, p. 54-115.

Freeman, J. A. and Skapura, D. M., 1992. *Neural Networks: Algorithms, Applications, and Programming Techniques.* Addison-Wesley.

Kohonen, T., 1982. Self-organized formation of topologically correct feature maps. Biological Cybernetics 43, 59-69.

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.

Masters, T. 1993. *Practical Neural Networks Recipes in C++*. Academic Press.

Rogers, J. 1997. *Object-Oriented Neural Networks in C++*. Academic Press.

Rumelhart, D.E., and McClelland, J.L., 1986. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Vol. 1. Cambridge, Ma: MIT Press.

Szyperski, C. 1998. Component Software: Beyond Object-Oriented Programming. Addison-Wesley.