A NEW DEVS-BASED GENERIC ARTFICIAL NEURAL NETWORK MODELING APPROACH

S. TOMA[(a)], L. CAPOCCHI[(b)], D. FEDERICI[(c)]

[(a)(b)(c)] SPE UMR CNRS 6134 Laboratory, University of Corsica, Quartier Grimaldi, 20250, Corte, France

[(a)]toma@univ-corse.fr, [(b)]capocchi@univ-corse.fr, [(c)]federici@univ-corse.fr

ABSTRACT

The Artificial Neural Network (ANN) is a black box model capable of resolving paradigms that linear computing cannot. Therefore, the configuration of ANN is a hard task for modeler since it depends on the application complexity. The Discrete EVent system Specification (DEVS) is a formalism to describe discrete event system in a hierarchical and modular way. DEVS is mainly used to defragment a system or a model in an easy way allowing the interaction with the architecture and behavior of the system. This paper presents a new artificial neural network modeling approach using DEVS formalism in order to facilitate the network configuration by introducing a new scheme of the training phase. We validate our approach with a simple not linearly separable data set example provided by two-dimensional XOR problem.

Keywords: artificial intelligence, discrete event systems, artificial neural networks, learning systems, modeling, simulation.

1. INTRODUCTION

Throughout the years, the computational changes have brought growth to new technologies. Such is the case of ANNs; they have given various solutions to the industry. Designing and implementing intelligent systems has become a crucial factor for the innovation and development of better products for society. Such is the case of the implementation of artificial life as well as giving solution to interrogatives that linear systems are not able resolve (Bishop 1995, Mas and Flores 2008, Agatonovic-Kustrin and Beresford 2000). In the world of engineering, neural networks have two main functions: Pattern classifiers and non linear adaptive filters (Bishop 1995). As its biological predecessor, an ANN is an adaptive system where each parameter is changed during its operation and it is deployed for solving the problem in matter (Drew and Monson 2003).

ANN is a system capable of resolving paradigms that linear computing cannot. It is a system based on the operation of biological neural networks, in other words, it is an emulation of biological neural system. Another aspect of the ANN is that there are different architectures, which consequently requires different types of algorithms, so it might look like a complex system (Agatonovic-Kustrin and Beresford 2000). Always said that the ANN is a black box system and we can never interact with its structure. Sometimes depending on the architecture or the algorithm used some parameters must be initialized. Some of these parameters are a function of the complexity of the system that we will try to solve. For certain types of problem try and error are able to get the best network configuration, but it will be better to find some algorithms to automate this process (Bishop 1995, Agatonovic-Kustrin and Beresford 2000).

DEVS is a formalism which allows the behavior modeling of a non linear system (Zeigler and Praehofer and kim 2000). This formalism provides a model (atomic) in order to define the behavior of a system (Concepcion and Zeigler 1988). DEVS and ANN are two concepts that are able to simulate complex systems and problems. Combining DEVS and ANNs could make a perfect match because of the nature of each of these concepts. In (Choi and Kim 2002) we can see this combination was the extraction of the DEVS from a trained ANN. An another interesting approach has been presented in (Filippi and Bisgambilia and Delhom 2001) where the ANN behavior is encapsulated into only one atomic DEVS model making a hybrid system that offers a better simulation.

In order to go much further with hybrid systems this paper presents a new modeling approach of the ANN using DEVS aspects. This new model will concern presenting an ANN into certain number of atomic and coupled models. This approach will be able to facilitate the network configuration that depends a lot on the application. In other words the new model will be able to give the space to implement algorithms and plug-ins to automate the network configuration as the network efficiency.

The remainder of the paper is organized as follows. In section 1, we introduce the DEVS formalism and the ANN concepts, showing their usage, their structure and how could they be implemented. Section 2 describes the new hybrid system that transforms the ANN into several DEVS models. Section 3 describes a test comparison of the new model using an XOR problem in order to present our new design. Finally, we conclude and present future works.

2. BACKGROUND

2.1. DEVS Formalism

DEVS is a formalism introduced by Zeigler (1976) to describe discrete event system in a hierarchical and modular manner. A manner way means that the system has input and output ports that allow it to interact with the external environment (Concepcion and Zeigler 1988). This formalism is distincter between the simulation approach and the modeling one. The DEVS modeling approach captures dynamic behavior with atomic models. The simulation approach is responsible for the automatic generation of the simulation algorithms. DEVS collected the simulated system behaviors into two models: atomic and coupled model. Each model type could be considered like a black box with some behaviors and interactions with the external environment through input and output ports. The atomic models can be linked together in a well-defined way to produce more complex coupled models whose behaviors are described by their atomic models and a set of a relation between those models. Any real system can be modeled using DEVS into a collection of coupled and atomic models (Barros and Zeigler and Fishwick 1998).

2.1.1. Atomic and Coupled Models

An atomic model is a model for a system that has a set of inputs, outputs, states, transition functions, a time advance function and an output function. Any atomic model can be defined by the following structure.

$$M_{atomic} = < X, Y, S, \delta_{ext}, \delta_{int}, \lambda, t_a >$$

- X is a set of inputs
- Y is a set of outputs
- S is a set of states
- $\delta_{int} : S \rightarrow S$ is the internal transition function.
- $\delta_{ext} : Q \times X \rightarrow S$ is the external transition function, Where $Q = \{(s, e) \mid s \in S, 0 \leq e \leq t_a(s)\}$ is the set of total states, e is the time elapsed since the last transition.
- $\lambda : S \rightarrow Y$ is the output function
- $t_a : S \rightarrow R^{+}_{0,\infty}$ is the time advance function.

The internal transition function describes state changes that occur in the absence of input over time. The external transition function responds to an input with a certain state change. Based on the current state the output function produces and output event. The time advance function calculates the amount of time before the next internal state transition takes place (assuming no inputs arrive in the interim).

A coupled model is a confirmation on the hierarchical notation of DEVS. It is consisted of a set of sub-models. Sub-models could be either atomic or coupled models. The behavior of such models is defined by the behavior of its models components and the relations between them.

The coupled models consist of a set of inputs, outputs, states, a set of sub-models with the influences between them and three types of coupling between models (Figure 1).

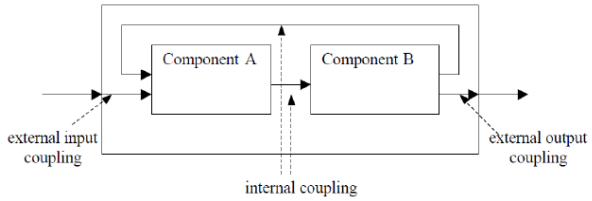

Figure 1: DEVS coupling in coupled models

The coupled model has inputs and outputs defined the same way as the atomic model. The couplings between the sub-models are defined under three categories: (i) External input coupling, (ii) Internal coupling, (iii) External output coupling. With the atomic and the coupled models it is easy to model and simulate discrete event systems.

2.1.2. DEVS Softwares

Nowadays the number of tools implementing the DEVS formalism is growing too quickly. In (Lara and Vangheluwe 2002) the authors present a General User Interface (GUI) allowing multi-paradigm (including DEVS) modeling. PowerDEVS (Kofman and Lapadula and Pagliero 2003) is an excellent GUI for the DEVS modeling and simulation focused on hybrid systems. Another interesting tool is described in (Baati and Frydman and Giambiasi 2007) that is often used for pedagogical aspects. In the case of ANN modeling we need a GUI only based on DEVS formalism allowing us to design and implement any architecture and ANN algorithm.

The main benefit using this kind of software is the simplicity of modeling the ANN algorithms and the possibility of creating libraries of reusable and "viewable" components. The GUI software allows creating, deleting, handling or switching the models in a simply way using toolbar or shortcut. Moreover, the simulation process is automatic and performed by clicking on a simple button.

DEVSimPy (Python Simulator for DEVS models) (Capocchi and Santucci and Poggi And Nicolai 2011) is a user-friendly interface for collaborative modeling and simulation of DEVS systems implemented in Python. Python is a programming language known for its simple syntax and its capacity to allow modelers to implement quickly their ideas (Sanner 1999). The DEVSimPy project uses the python language and provides a GUI based on PyDEVS (Bolduc and Vangheluwe 2001) Application Program Interface (API) in order to facilitate both the coupling and the reusability of PyDEVS models. This API is used in the excellent multi-modeling GUI software named ATOM3 (Lara and Vangheluwe 2002) which allows the usage of several formalisms without focusing on DEVS. DEVSimPy is an open source project under GPL V3 license and its development is supported by the SPE

research laboratory team. It uses the wxPython graphic library and it can be downloaded from http://code.google.com/p/devsimpy/.

The main goal of this environment is to facilitate the modeling of DEVS systems using the GUI dynamic library and the drag and drop functionality. With DEVSimPy, models can be stored in a dynamic library in order to be reused and shared. The creation of dynamic libraries composed by DEVS components is easy since the user is coached by dialogs and wizard during the building process. We propose in this paper the DEVS modeling of ANNs algorithm through DEVSimPy in order to implement a generic ANN library. Thereby, the DEVSimPy developer will be able to use this library when the ANN is needed in the modeling of complex systems at all times.

2.2. Artificial Neural Network

Basically, an ANN is a system that receives input, process the data, and provides an output. This system can be used for two main functions: Pattern classifiers and as non linear adaptive system. By adaptive, it means that the system parameters can be changed during operation to solve the faced problem. This is called the training phase. During this phase a classifier or a non linear adaptive system ANN tries to adapt its parameters to solve the problem in matter. The ANN has many different architectures and for everyone there is some personalized algorithms. In this section the Feed-Forward Neural architecture with Backpropagation algorithm is presented.

2.2.1. Feed-Forward Neural Architecture

Any ANN has a certain number of entities called neuron. The power of the network comes from the weighted connections between different neurons. When neuron receives weighted inputs it calculates the sum and then passes the data through a transfer function. The transfer function is the element that introduces the non-linearity aspect into the neural network. Many types of function can be used: hyperbolic, threshold, piecewise-linear, and the sigmoid functions.

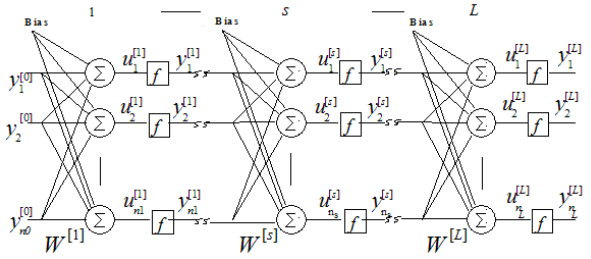

Figure 2: Artificial Neural Network Architecture.

The multilayer perceptrons (MPLs) is the most used class of ANN in all applied fields. A MPL consists of a set of input units (input layer), one or more computation layers (hidden layers), and one output layer (computation/output). In feed-forward architecture (Figure 2) a neuron on layer *s* always connects to a neuron on layer *s+1*. A fully connect network means that all neurons on layer s are connected to each neuron on layer *s+1*. Every neural network must be trained before we can use it. So a training algorithm is chosen to adapt and change the connection weights between each layer.

2.2.2. Backpropagation Learning Algorithm

The training algorithm that is used to adjust the network weights is a principal factor of the network accuracy and performance. Two major types of algorithms can be found to train an ANN: supervised and unsupervised (Omatu and Khalid and Yusof 1996, Bishop 1995). Unsupervised learning is used when no output is desired for the ANN. Supervised training is used when we have a well defined desired output. First the input propagate forwardly throw layer and an output is calculated. The supervised algorithm calculates the error between desired and calculated output. The layer connection weights are modified trying to minimize this error. This cycle is repeated many times until the network is trained (Agatonovic-Kustrin and Beresford 2000).

$$\mu_{pj}^{[s]} = \sum_{i=1}^{n_s-1} w_{ij}^{s}\, y_{pi}^{[s-1]} \tag{1}$$

$$y_{pj}^{[s]} = f\left(\mu_{pj}^{[s]}\right) \tag{2}$$

$$\delta_{pj}^{[s]} = (d_{pj} - y_{pj}^{[L]}) f'(u_{pj}^{[L]}) \tag{3}$$

$$\delta_{pj}^{[s]} = f'\left(u_{pj}^{[L]}\right) \sum_{r=1}^{n_s+1} \left(\delta_{pj}^{[s]} w_{rj}^{s+1}\right) \quad s = L-1, \dots, 1 \tag{4}$$

$$w_{ji}^{[s]}(t) = w_{ji}^{[s]}(t) + N\left(\delta_{pj}^{[s]}(t) y_{pji}^{[s-1]}\right) + M(\delta_{pj}^{[s]}(t-1) y_{pji}^{[s-1]}) \tag{5}$$

$$E = \frac{1}{N} \sum_{p=1}^{m} \sum_{k=1}^{n_L} (d_{pk} - y_{pk}^{[L]})^2 \tag{6}$$

The backpropagation (BP) is the most common supervised learning algorithm to a feed-forward neural network used as classifiers (Figure 2). A BP network learns by example; in other word it learns by training sets. At the beginning of the training phase, all weights are initialized by random values - say between -1 and +1. Next the input patterns (p) propagate throw the network layers (s) calculating the output value (Eq.1, Eq.2). After the first propagation of data the calculated output normally is different than the desired output. At that point the BP algorithm comes to calculate the error of each output neuron. After that a reverse procedure to the forward propagation takes place trying to calculate new weights as a function of the calculated error. This back propagation of values is calculated using the derivative of the activation function, the inputs, outputs of each layer, a momentum factor (M) and a learning factor (N). Equation 3 is used for the output layer error calculation and for all other layer the equation 4 is applied. Equations 5 and 6 are used for the weight adaption and to calculate the quadratic error.

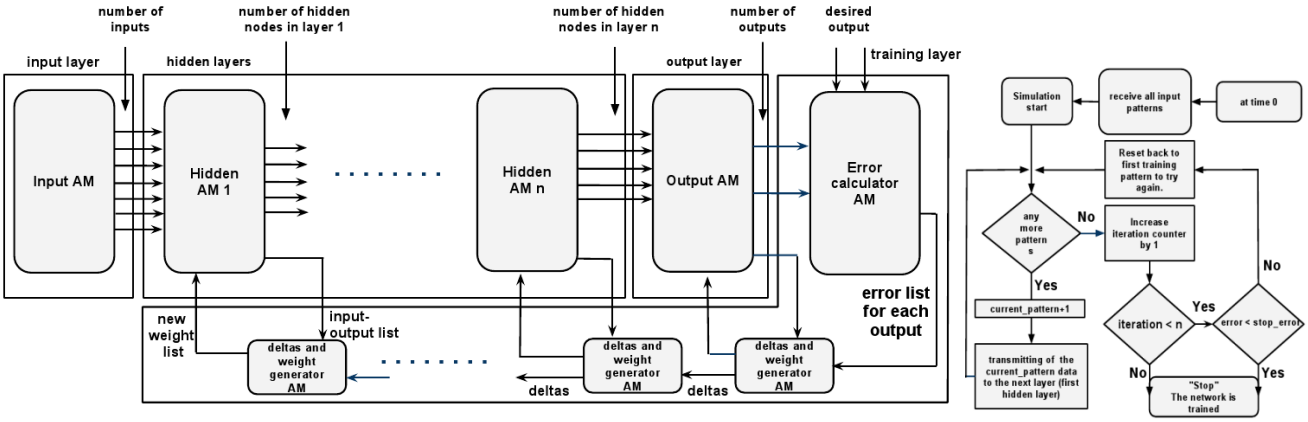

Figure 3: (a) ANN/DEVS model transformation (b) Input Atomic Model block diagram.

3. PROPOSED ANN MODELING APPROACH

3.1. ANN/DEVS Compatibility

Since its birth the ANN is inspired from the human brain neurons and known as a black box. When an ANN is build some configuration must take place. First the number of layers that we are going to use and this depends on the complexity of the problem that we are trying to solve. Usually we use a two layer neural network (two calculation layer + input layer) that is sufficient to solve most non-linear problems. Second the number of neuron in each layer: it is fixed for the input and output layers by the input and output data length; for the hidden layer it is one of the hardest choices. The choice of the number of neuron in the hidden layer is different for each application but some recommendations may take place; taking half the number of input neurons plus one is one of them. Also the bias value in each layer can be a difficulty too. Third the algorithms that can learn or train the network can have a great effect on the network performance. The BP algorithm is one of the most common ones. As a result for the BP choice, learning and momentum factors (N,M) must be chosen. Forth the stop condition for the training phase. Too much training could be considered as over-training, which means that the network will learn the data noise and not the desired pattern. Too short training could lead to an untrained network. So an iteration number depending on the application complexity must be chosen or define a stop condition.

As shown in the previous paragraph too many parameters must be calibrated before using the neural network. Creating a generic ANN library that can be automatic configured and even new aspect and algorithms can be plug-in directly to the network without re-building it. Trying to break the idea that the ANNs are always black boxes and they are not easy to calibrate, a DEVS model is proposed to simulate an ANNs. The ANNs are by default using discrete event; the network is always waiting to an input event to generate an output one. Even inside the network itself, every layer is waiting to receive data from the previous layer to start calculations. Also during the BP algorithm the recalculation of weights starts when an output is calculated; than the error is calculated; than modification of weight is done layer after layer waiting the changes in the previous ones. So ANNs have a natural compatibility with DEVS formalism, which makes it obvious to create the generic neural library using this formalism.

3.2. ANN/DEVS Mapping Approach

Usually when we represent the neural network we see it as in Figure 2, but what we can see in Figure 3 (a) could be a little bit different. In this paper we propose a DEVS model per layer, which means that for the input, hidden and output layers in the neural network will be presented as a standalone atomic model. And a new training layer will be presented into multiple atomic models.

The input atomic model (input layer) could be considered as the leader of the network. It is called leader because it controls the data propagation throw the network. Controlling the data propagation means that it controls when to propagate the learning, testing, or the real data patterns. First this input model receives with all pattern types and the stop learning condition (iteration number). After initialization it starts to push learning patterns into the network to start calculations (Figure 3 (b)). An iteration number can be determined but also a minimum error to reach to not to get the over training problem. In this design the hidden and the output models are almost the same model. The unique difference is the number of neurons in the output model is fixed by the number of outputs as it is the rule of any neural network. Then the calculations made in the hidden model are the same as in the output model. Both of them are multiplying the inputs by the weight list for each neuron inside this model and then go through the transfer function that must be chosen before the calculation starts (Eq.1, Eq.2).

This is the first time to see something called training layer. All models that help only to train the network will be considered as the training layer (Figure 3 (a)). The idea of having a neural DEVS network came

while trying to enhance and automate the training of any neural network and make it as generic as possible. In this layer the learning algorithm takes place, so any learning algorithm can implemented with its own design. One of these algorithms is the BP shown in the previous section. So the training layer will be composed of an error calculator and deltas and weights generators. The error calculator model has two function; First is to calculate the error of each output, which means the difference between the calculated output and the desired one for each single output; Second is the calculate the global quadratic error (Eq.6). The deltas and weight generator is the model where the learning algorithm appears. Algorithm 1 shows the external function of this generator model and how the BP algorithm can be implemented.

Algorithm 1:

```
errors = msg_received
for i in range(len(outputs)):
 deltas[i] = self.dactivation(outputs[i]) * errors[i]
 for j in range(len(inputs)):
   for k in range(len(outputs)):
     change=deltas[k]*nputs[j]
     weights[j][k] =weights[j][k] + N*change + M * C[j][k]
     C[j][k] = change
for i in range(len(inputs)):
 for j in range(len(outputs)):
   GError[i] = GError[i] + (deltas[j] * weights[i][j])
```

As shown in Figure 3(a) the first deltas and weights generator receives the output error list from the error generator and then starts to calculate the deltas and the new weights for the output model, then the next deltas and weights generator does the same for all hidden models. Trying to prevent the overtraining phenomenon an error stop condition will be implemented. After the training phase ends either by a fixed iteration number or another stop condition the test phase begins automatically by the input model. During the test phase the error generator still works but only to calculate the global error and send nothing to the deltas and weight generator so no more weight modification could be done during this phase.

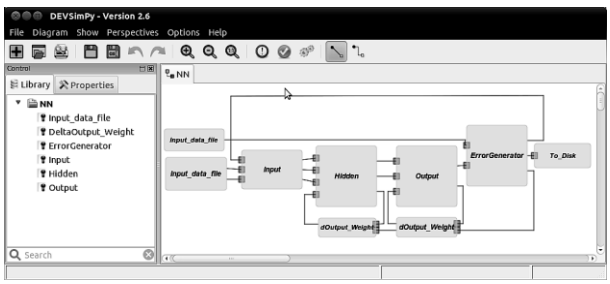

Figure 4: DEVSimPy Dynamic library.

Also sometimes we need to use some validation patterns during the training phase. Using DEVS and so DEVSimPy we can send during the training phase in parallel to the training patterns and the validation patterns too; which can help to have at the end of the training and the validation error data after each iteration. The validation pattern can help us to see if the network is going thought an over training or not.

To implement this modeling approach into DEVSimPy an ANN library is created that contains six atomic models that will help to construct a full ANN: input, output, hidden, error calculator, deltas and weights generator, input file (Figure 4). The input file is a model that extracts data from a file and gives it to the network in form of patterns. Some validation tests using this library will be presented in the next section.

4. VALIDATION AND ANALYSIS

The ANN can solve many non-linear problems and depending on the problem complexity the configuration of the network changes. The complexity of a network depends on two parameters: First the problem dimension; second is if the problem data is linearly separable or not. A simple example of a data set which is not linearly separable is provided by two-dimensional X-OR problem (Mas and Flores 2008) In this example we can show the problem of learning to classify a given data set, where each input vector has been labeled as belonging to one of two classes C1 and C2. The input vectors x = (0,0) and (1,1) belongs to class C1,while the input vectors (0,1) and (1,0) belongs to class C2. It is clear that there is no linear decision boundary which can classify all four point correctly. The neural network using the bias value and the weights values in each layer which is a linear discriminant that will lead to perfect classification. A comparison is made between the designed DEVSimPy neural network models (Figure 4)

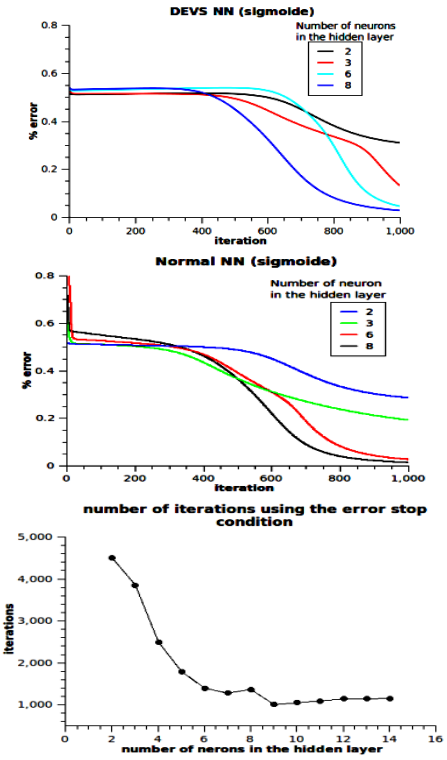

Figure 5: Test and Validation

and a mono-thread ordinary neural network written with python programming language. The tests represent the effect of the incrementation of neurons number inside the hidden layer; also the implementation of the stop learning condition is presented with the different number of neurons to show the effect of increasing neurons number and the iteration number.

Figure 5 represent the comparison between the proposed neural DEVS network and the ordinary neural network when the transfer function is sigmoid with different number of neurons in the hidden layer. These results shows that is the Neural DEVS network is capable to solve the same problem as the normal ANN but using DEVS has a big facility to add or remove hidden layers using the Drag and drop ability that DEVSimPy offer. Also using DEVS formalism gives more automatic configuration that can be added and many plug-ins and enhancements could be developed too. One of those implemented enhancements is the automization of the stop condition. In other word the learning phase stops only when the network has been learned with minimum accepted error. Figure 5 shows the number of iteration as a function of number of neurons in the hidden layer when an XOR problem is solved using sigmoid transfer function.

5. CONCLUSION

In this paper a new DEVS-model for artificial neural network is presented. This model shows a technique during the training phase by implementing an additional layer named as the training layer. The training layer is composed of several small atomic DEVS models that control the weights adaption independently for each layer. This approach was tested and compared to the standard network. With this technique a great facility to change the training algorithm or to add additional algorithms to make it more efficient is available. The stop error condition is added to the training layer as a test of additional algorithms that can be added as a new model and that could be removed at any time. With the same idea more algorithms to enhance the network performance could simply be added. Moreover there is a work in progress to implement the pruning algorithm. The pruning is a very interesting algorithm that minimizes the number of neurons in the hidden layers to get the smallest network to solve the problem in question. With this defragmentation of the training layer into several models the pruning algorithm can intervenes as an atomic DEVS model just before the entry of the new weights into the hidden layer to make its elimination decision. On the other hand the decision of using DEVS formalism to implement this defragmentation of the neural network opens a new dimension to implement new individual small models or plug-ins to enhance the performance of the network.

REFERENCES

Agatonovic-Kustrin, S., Beresford, R., 2000. Basic Concepts of Artificial Neural Network (ANN) Modeling and its Application in Pharmaceutical Research. *Journal of Pharmaceutical and Biomedical Analysis*, vol. 22, no. 5, pp. 717-727.

Baati, L., Frydman, C., Giambiasi, N., 2007. LSIS DME M&S Environment Extended by Dynamic Hierarchical Structure DEVS Modeling Approach, in *Proceedings of the 2007 spring simulation multiconference*, Vol. 2, pp. 227-234. San Diego, CA, USA.

Barros, F. J., Zeigler, B. P., Fishwick, P. A., 1998. Multimodels and Dynamic Structure Models: an Integration of DSDE/DEVS and OPMM, *Proceedings of the 30th conference on Winter simulation*, pp. 413-420. Los Alamitos, CA, USA.

Bishop, C. M., 1995. Neural Networks for Pattern Recognition, *1st ed. Oxford University Press*, USA.

Bolduc, J. S., Vangheluwe, H., 2001. The Modelling and Simulation Package PythonDEVS for Classical Hierarchical DEVS, *MSDL Technical Report MSDL-TR--01*. Montreal, Quebec, Canada.

Capocchi, L., Santucci, J.F., Poggi, B., Nicolai, C., 2011. DEVSimPy: A Collaborative Python Software for Modeling and Simulation of DEVS Systems, Accepted in 20th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, Paris.

Choi , S. J., Kim, T. G., 2002. Identification of Discrete Event Systems Using the Compound Recurrent Neural Network: Extracting DEVS from Trained Network, *Simulation*, vol. 78, no. 2, p. 90.

Concepcion, A.I., Zeigler, B.P., 1988. DEVS Formalism: A Framework for Hierarchical Model Development, *IEEE Transactions on Software Engineering*, vol. 14, no. 2, pp. 228-241.

Drew, P. J. J., Monson, R. T., 2003. Artificial Neural Networks, *Surgery*, vol. 127, no. 1, pp. 3-11, jan.

Kofman, E., Lapadula, M., Pagliero, E., 2003. PowerDEVS: A DEVS-based Environment for Hybrid System Modeling and Simulation, Tech. Rep., Rosario National University.

Filippi. J., Bisgambiglia, P., Delhom, M., 2001. Neuro-DEVS, an Hybrid Methodology to Describe Complex Systems, *in Actes of SCS ESS 2001 conference on simulation in industry*, vol. 1, pp. 647-652.

Lara, J., Vangheluwe, H., 2002. AToM 3: A Tool for Multi-Formalism and Meta-Modelling, *Fundamental Approaches to Software Engineering*, pp. 174-188.

Mas, J. F., Flores,J. J., 2008. The application of Artificial Neural Networks to the Analysis of Remotely Sensed Data, *International Journal of Remote Sensing*, vol. 29, no. 3, p. 617.

Omatu , S., Khalid, M., Yusof,R., 1996. Neuro-Control and its Applications, *Springer*.

Sanner, M. F., 1999. Python: A Programming Language For Software Integration and Development, *J. Mol. Graphics Mod*, vol. 17, pp. 57-61.

Zeigler, B.P., Praehofer, H., Kim, T.G., 2000. Theory of Modeling and Simulation, *Academic press*, 2nd Edition.