A DEVS-BASED ANN TRAINING AND PREDICTION PLATFORM

A Thesis Presented to the Department of

Computer Science,

African University of Science and Technology, Abuja

In Partial Fulfillment of the Requirements

For The Degree of

MASTER OF SCIENCE

By

David Ifeoluwa Adelani

Abuja, Nigeria

December, 2014

A DEVS-BASED ANN TRAINING AND PREDICTION PLATFORM

By

David Ifeoluwa Adelani

A THESIS APPROVED BY THE COMPUTER SCIENCE DEPARTMENT

RECOMMENDED: **……**

Supervisor, Professor Mamadou Kaba Traore

……

Head, Department of Computer Science

APPROVED: **……..**

Chief Academic Officer

……

Date

ABSTRACT

The artificial intelligence (AI) domain grows every day with new algorithms and architectures. Artificial Neural Networks (ANNs), a branch of AI has become a very interesting domain since the eighties when the back-propagation learning algorithm and the feed-forward architecture were first introduced. As time passed, ANNs were able to solve non-linear problems, and were being used in classification, prediction, and representation of complex systems. However, ANN uses a black box learning approach – which makes it impossible to interpret the relationship between the input and the output. Discrete Event System Specification (DEVS) is a mathematical well-defined formalism that can be used to model dynamic systems in a hierarchical and modular manner; it can automatically generate simulators for the described DEVS models. Combining ANN and DEVS, we can model the complex configuration of ANNs and express its internal workings. In this thesis, we are extending the DEVS-Based ANN proposed by Toma et al [1] for comparing multiple configuration parameters and learning algorithms. The DEVS model is described using a visual modeling language known as High Level Language Specification (HiLLS) for a clear understanding. This approach will help users and algorithm developers to test and compare different algorithm implementations and parameter configurations of ANN.

ACKNOWLEDGEMENT

First, I want to appreciate God – the source of my provision and wisdom. Thank you for sparing my life till today.

I would like to express my deepest appreciation to my Supervisor, Head of Department and Acting President of AUST for his scholarly assistance, advice and strong motivation in doing research. He really assisted in strengthening my weak foundation in Computer Science. I am also grateful to Doyin, Hajara, Aliyu and Seun for their assistance in this research work.

To the sponsor of my 18 month Master's program, the Nelson Mandela Institute, I say a big thank you. I am also grateful to Prof Wole Soboyejo, Prof Kunle Olukotun, Prof Lehel Csato, Prof Hamada and other lecturers.

I say a big thank to my lovely parents Pastor and Mrs Adelani and my uncle, Mr Sunday Adelani for their prayers, encouragements and financial support; and also to my siblings: Blessing, Johanah and Nehemiah.

Also, my appreciation goes to the Director (IT) of Federal Ministry of Communication Technology, Mrs Moni Udoh for the permission to pursue my Master's program. I'm also grateful to the members of Deeper Life Campus Fellowship for my spiritual upkeep: Pastor Gilbert and his wife, Prof Igbokoyi, Bro Peter, Toyin, Esewi, Sunday, Bro Tunde, Pastor Ben and others.

Finally, special thanks should be given to my colleagues for their assistance in one way or the other: Seun, Rolland, Fisayo, Esewi, Michele, Walu, Sam, Amanze, John, Saheed and Rasheed; PhD students: Tabot, Ignace, Yoro and the entire AUST community. May the Almighty God richly bless you in all your endeavours.

DEDICATION

I dedicate this thesis to the Almighty God, the Master Planner, and the source of wisdom and understanding.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

1.0 INTRODUCTION

1.1. Context

Modeling and Simulation (M&S), the third pillar of science is a paradigm that provides a way of obtaining the behavior of the representation of an object in real life without doing physical experiments. As introduced by the theory of Modeling and simulation [2], there are four major important concepts of M&S. The concepts are defined below:

a) **System**: is a well-defined object in the real world under specific conditions that we are interested in modeling.
b) **Experimental Frame (EF)**: is a specification of the conditions within which the system is observed or experimented. It is realized as a system (with generators, acceptors and transducers) that interacts with the source system to obtain data of interest under specified conditions.
c) **Model**: is an abstract representation of the structure and properties of a system at some particular point in time or space intended to promote understanding of the real system.
d) **Simulation**: is the execution of a model over time in order to get the information about the changes in the behavior of the system during executions.

Modeling complex systems requires a robust formalism. The Discrete Event System Specification (DEVS) formalism [3] that was introduced in the early 70's is a theoretically well-defined formalism for modeling discrete event systems in a hierarchical and modular manner. It allows the behavior modeling of complex systems.

Artificial Neural Networks (ANN) is a branch of artificial intelligence that became popular in the eighties when the back-propagation algorithm [4] for multilayer feed-forward architectures was introduced. It is widely known that classical neural networks, even with one hidden layer, are universal function approximators [5]. ANNs became widely applicable for real applications when it had the capabilities to solve non-linear problems. It is used for modeling of complex optimization problems such as classification, prediction and pattern recognition.

Artificial neural network is capable of modeling complex non-linear systems using adaptive learning mechanism to derive meaning from complicated or imprecise data with a high degree of accuracy. However, ANN uses a black box learning approach – when the general architecture is defined, you almost don't have an idea of how the output is produced. To overcome this, DEVS is combined with ANN to express the relationship between the input and output. Combining DEVS and ANN is possible because ANNs are by default using discrete events i.e. the network is always waiting to an input event to generate an output one. Toma et al [1] proposed an approach for the describing the structure of ANN with DEVS known as DEVS-Based ANN. This approach was said to be able to facilitate the network configuration that depends a lot on ANN.

We propose to extend the work in [1] for comparing multiple configuration parameters and learning algorithms. The configuration parameters for ANNs are number of hidden layers, output neurons for each layer and stopping condition (minimum error or maximum number of iterations) to avoid over-training. This will help users and developers test and compare different algorithm implementations and parameter configurations. A new visual modeling language, High Level for System Specification [6] (HiLLS) which is an extension of DEVS Driven Modeling Language (DDML) [7] will be used to describe the approach for clear understanding.

1.2. Research Objectives

The objective of this work is to achieve a DEVS-based Modeling and Simulation (M&S) platform that will allow the specification of ANNs and their configuration (training) and use for prediction as well. We aim to build a generic systems dynamic-based model for ANNs specification and training with HiLLS – a graphical representation of DEVS and implement the DEVS models with the SimStudio M&S package for ANN learning algorithms comparisons. The built DEVS-Based ANN platform will be benchmarked with study cases.

1.3. Related Works

Neural networks have been used extensively to determine the behavior of discrete event systems because it can learn from empirical data but little research is focused on using discrete event modeling to specify the structure and components of a neural network.

1.3.1. Abstraction of Continuous System to Discrete Event System Using Neural Network

Sung Hoon Jung and Tag Gon Kim in [8] explored the use of Artificial Neural Networks in modeling of a hybrid system which consists of continuous systems and discrete event systems. The continuous system cannot directly communicate with the discrete event system. Therefore, they proposed neural networks as an interface for communication between the two disjoint systems. To achieve this, the continuous system was first represented by a timed state transition model and the model is then mapped into a neural network by learning capability of the network.

The timed state transition model was obtained through 3 steps - output partitioning, input sampling and measuring of delay. From the information gotten they specified a 6 tuple model based on DEVS [9] model.

$$M = < X, Y, S, T_d, \lambda, s_0 >$$

X is the input events set obtained from input sampling

Y is the output events set obtained from output partitioning

S is the states set

$T_d: S \times X \times S \rightarrow \mathbb{R}^{+}_{0,\infty}$ is the time delay function

$\lambda: S \rightarrow Y$ is the output function

s_0 is the initial state in S

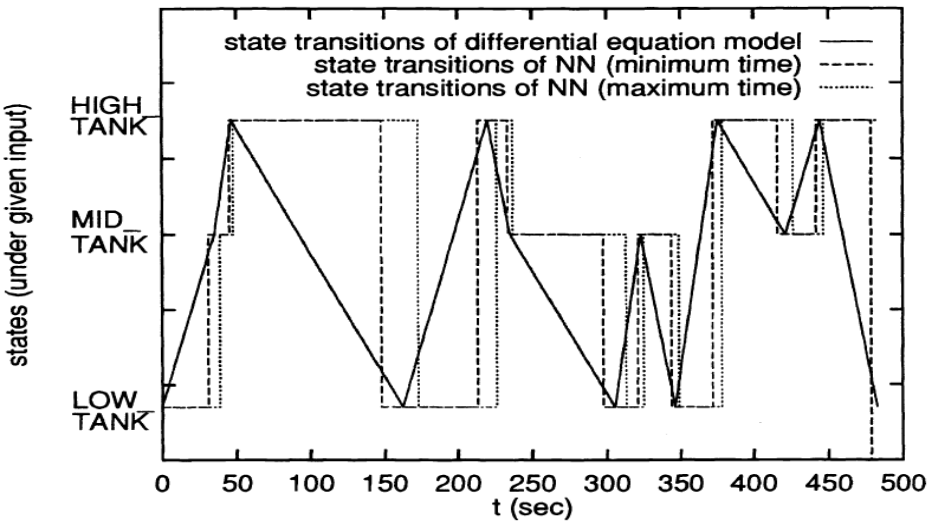

Figure 1.1: State Trajectories of Differential Equation Model and Neural Network Model [8]

The table function T_d is a tedious and difficult one, hence, they trained a neural network to obtain the state transition rules using state variables, inputs and outputs as the neural networks input and time delay (since it is a real number) as the neural network output. The approach was applied to Water Tank Continuous System and the results of the table function were compared to that of differential equations which shows an impressing result. (See figure 1.1).

1.3.2. Identification of Discrete Event Systems: Using the Compound Recurrent Neural Network: Extracting DEVS from Trained Network

Si Jong Choi and Tag Gon Kim [10] identified Discrete Event Systems (DES) from a compound recurrent neural network (CRNN) by 2 major steps: behavior learning using a specially signed neural network and extraction of a DEVS model from the neural network. The neural network was trained using observed input/ output events of an unknown DES.

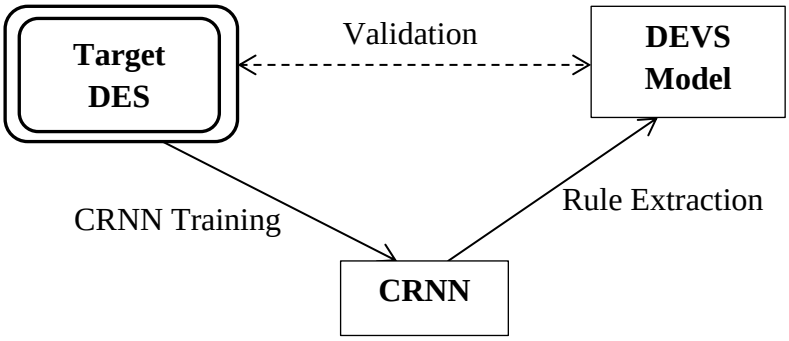

Figure 1.2: Identification of Discrete Event

After training of CRNN, it is behaviorally equivalent to the explored DEVS model. The extracted DEVS model is validated against the timed I/O event sequences of the target DES. As a result of identification process, an unknown target DES is identified as a single DEVS model or equivalent DEVS model

1.3.3. Neuro-DEVS, an Hybrid Methodology to describe Complex Systems

Jean-Baptiste Filipi, Paul Bisgambigia and Marielle Delhom [11] presented an Object Oriented Modeling and Simulation (OOMS) of Complex Systems based on DEVS and neural networks. For OOMS to be implemented efficiently, the model behavior has to be well known. Neural networks can learn from empirical data to obtain a systems behavior. Therefore, they extended

the OOMS environment with neural network objects known as adaptive models or neural net sub-models.

They proposed Neuro-DEVS extended the DEVS atomic model with some neural networks essential functions like activation function, learning function and connection links. The OOMS coupled model now has Neuro-DEVS has one of its atomic models. The hybrid system offers better simulation in the following ways:

a) Concurrent simulation: useful for comparing neural network outputs with the output of a simple model. This helps to validate the results of neural networks.
b) Adaptive models can be used to modify the neural network runtime according to an error feedback [12]. The error is the difference between the model's forecast and the real world data collected afterwards.
c) ANN as a sub-component can be used if Neural Networks provides better results for only a piece of the whole system.

1.3.4. Dynamic Neuronal Ensembles (DNE): Neurobiologically Inspired Discrete Event Neural Networks

S. Vahle [13] made use of discrete event modeling formalism to model the neurobiological components of the neural network in order to increase the computational power, adaptability and dynamic response of ANNs. He proposed a DEVS model called Dynamic Neuronal Ensembles (DNE). The DNE is a coupled model of components that are themselves coupled models called dynamic neurons (DN).

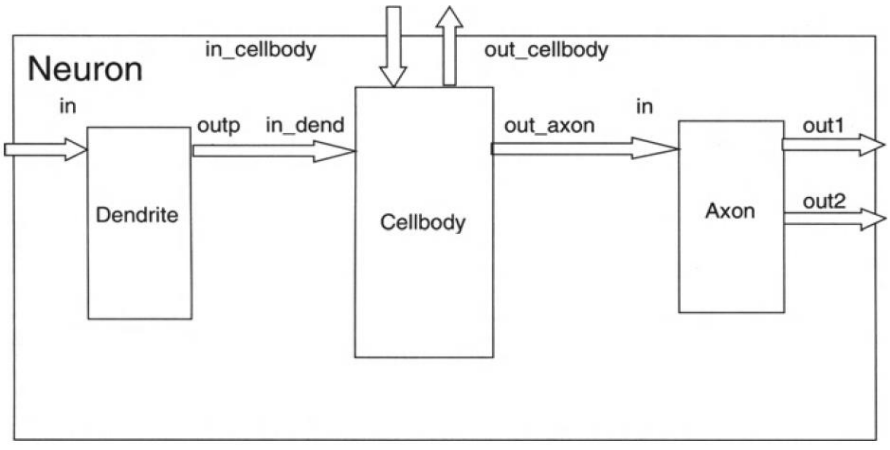

Figure 1.3: The Dynamic Neuron [13]

The proposed dynamic neuron coupled model is composed of three basic atomic models: the dendrite, the cell body and the axon. The dendrite receives the inputs from axonal output connections, adjusts each according to its weighting requirements and requests connection modification. The cell body adds the input signals received from its attached dendrites to its decayed potential level. A fire signal is sent to the cell body's attached axons if the potential level meets or exceeds the thresholds. The axon produces an output when it receives a fire signal.

1.3.5. A New DEVS-Based Generic Artificial Neural Network Modeling Approach

S. Toma, L. Capocchi, and D. Federici [1] described the structure of ANN with DEVS atomic and coupled models. They pointed out that ANNs are by default using discrete event; the network is always waiting to an input event to generate an output one. S. Toma [14] described three possible ways by which message can be transferred in ANN/DEVS mapping. In figure below, we have (a) neuron architecture level, (b) layer architecture level and (c) and reduced-layer architecture level. In the neuron architecture, each neuron is an atomic model – this offers good visual representation of ANN but the simulation will be slow because of large number of messages sent between neurons. On the other hand, in layer architecture, each layer is an atomic model with less visual representation of the neuron connections; it clearly shows the number of outputs from one layer to another and lesser messages is transferred during simulation. The reduced-layer architecture has the least number of messages transferred between atomic models which ensure fast simulation but no graphical representation for the number of neurons in the each layer. The layer architecture level is preferred.

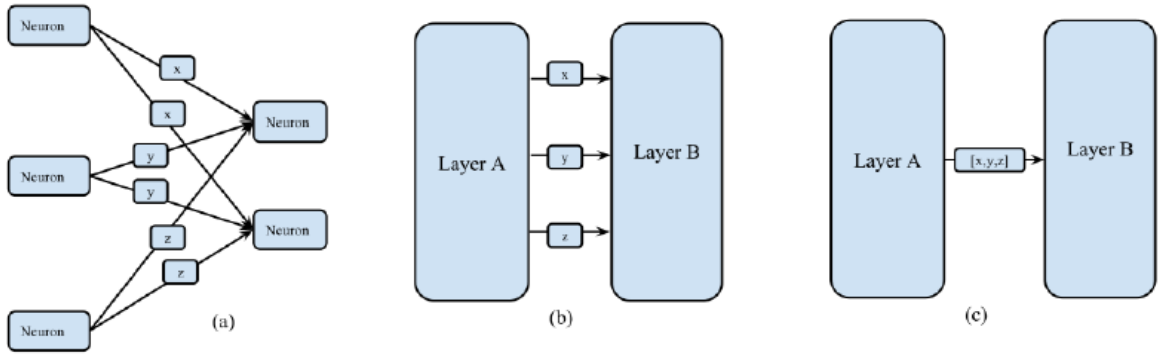

Figure 1.4: The Dynamic Neuron [14]

They used four atomic models to describe a multilayer ANN: non-calculation layer (input layer), calculation layer (hidden and output layers), error generator and delta weight. The non-calculation layer forwards inputs to the first hidden layer through linear activation function $f(x) = x$. The calculation layer will compute the neuron output using non-linear activation functions. The calculated output is compared with the target output in the error generator model, and the error difference is forwarded to the delta weight model for error correction. 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.

1.4. Approach Adopted

In this work, we reviewed Artificial Neural Networks learning algorithms based on gradient descent such as Back propagation, Back propagation with momentum, Silva & Almeida, Delta-Bar-Delta, Quickprop and Resilient Propagation. Common activation functions that are continuous, non-linear and differentiable such as Binary sigmoid function, Bipolar sigmoid function, Hyperbolic tangent function and Gaussian functions were also reviewed.

HiLLS visual modeling language is then used to describe the DEVS-Based ANN model taking into consideration the parameter configurations such as number of hidden layers, output neurons for each layer and stopping condition (minimum error or maximum number of iterations).

The DEVS model is then implemented with SimStudio [15] package and Java programing language. The Graphical DEVS-Based ANN platform built is able to compare different learning algorithms and activation functions. It will also be benchmarked with case studies.

1.5. Organization of Work

This work is organized as follows: Chapter 2 introduces the Artificial Neural Networks architecture and learning algorithms; DEVS formalism; and HiLLS visual modeling language. Chapter 3 presents the DEVS-Based ANN approach, HiLLS specification and SimStudio implementation. Chapter 4 presents the case studies, data extraction and results. Chapter 5 provides the summary of the work done, pros and cons and future works.

CHAPTER 2

2.0 STATE OF THE ART

2.1. Artificial Neural Networks (ANN)

ANN is a branch of Artificial Intelligence that models the way biological neurons process information. ANN is commonly referred to as Neural Networks. It is composed of a large number of highly interconnected processing elements (neurons) working in unison to solve specific problems [16]. Neural networks learn by example. They are particularly useful for solving problems that cannot be expressed as a series of steps such as recognizing patterns, classifying into groups, series prediction and data mining [17].

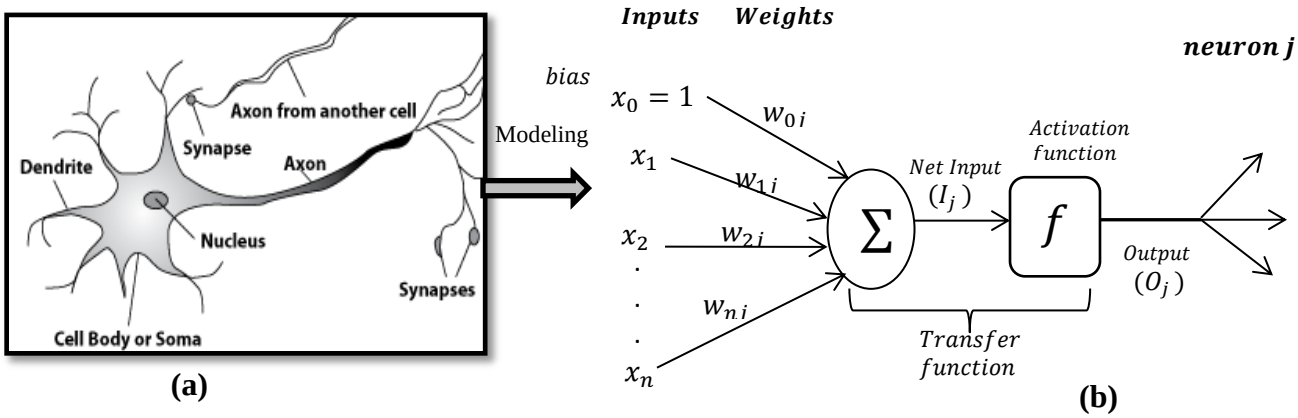

Figure 2.1: Biological Neuron and Artificial Neuron

A biological neuron (in figure 2.1a) has four major of components used in modeling artificial neural networks (shown in figure 2.1b): dendrites, synapse, soma, and axon. The many dendrites receive signals from other neurons (**inputs**). The signals are electric impulses that are transmitted across a synaptic gap by means of a chemical process in a manner similar to the action of the **weights** in an artificial neural network[18]. The soma, or cell body, sums the incoming signals and converts it into output activations (**transfer function**). Axons act as transmission lines to send activation (**outputs**) to other neurons.

2.1.1. History

Artificial neural networks have been the subject of an active field of research that has matured greatly over the past 40 years. McCulloch and Pitts [19] in 1943 designed what are generally

regarded as the first neural networks. They recognized that combining many simple neurons into neural systems was the source of increased computational power. The weights on a McCulloch-Pitts neuron are set so that the neuron performs a particular simple logic function, with different neurons performing different functions. The idea of a threshold that if the net input to a neuron is greater than the threshold then the unit fires is one feature of a McCulloch-Pitts neuron that is used in many artificial neurons today [18] .However, McCulloch-Pitts neurons are used most widely as logic circuits [20].

Donald Hebb [21] in 1949 designed the first learning rule for artificial neural networks. His premise was that if two neurons were active simultaneously, then the strength of the connection between them should be increased.

In the 1950 and 60's, many researchers (Block, Minsky, Papert and Frank Rosenblatt [22]) introduced and developed a large class of artificial neural networks called perceptrons. The most typical perceptron consisted of an input layer connected by paths with fixed weights to associated neurons; the weights on the connection paths were adjustable. The perceptron learning rule uses an iterative weight adjustment that is more powerful than the Hebb rule. Perceptron learning can be proved to converge to the correct weights if there are weights that will solve the problem at hand (i.e., allow the net to reproduce correctly all of the training input and target output pairs).

In 1969, Minsky and Papert [23] showed that perceptron could not learn functions which are not linearly separable like XOR (see figure 2.2). The neural network researched declined throughout the 1970 and until mid80's because the perceptron could not learn certain important functions.

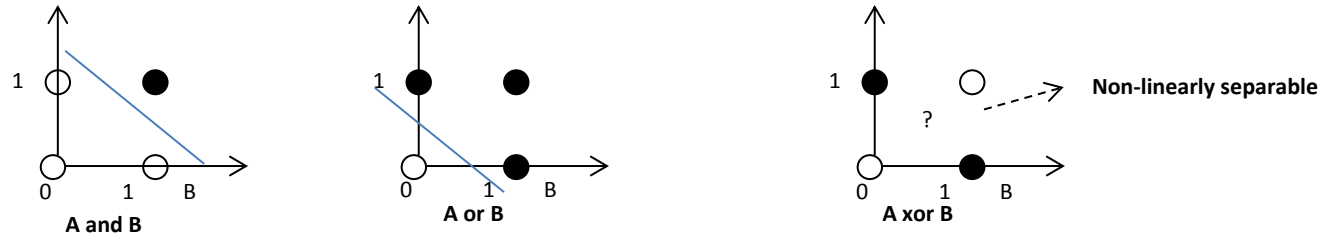

Figure 2.2: Linear and Non-Linear Separable Functions

The capabilities of neural networks were expanded from linear to nonlinear domains in 1974 by Werbos [24], with the intention of developing a method as general as linear regression, but with nonlinear capabilities. These multilayered perceptrons (MLPs) were trained via gradient descent

methods, and the original algorithm became known as “back-error propagation” [25]. Parker and LeCun discovered it independently before it became widely known [18], [26], [27]. Artificial neural networks with back-propagation were popularized by Rumelhart and McClelland.

2.1.2. Architecture

The Artificial Neural Network (ANN) is a black box model capable of resolving paradigms that linear computing cannot [1]. It receives input, process the data and provides output. Neural networks are composed of neurons or unit (Figure 1) connected by directed links. A link with numeric weight w_{ij} from unit i to unit j serves to propagate the activation O_j. The weight determines the strength and sign of the connection. In most algorithms, each neuron has a bias input $x_0 = 1$ with an associated weight w_{0j}. Each unit j first computes a weighted sum of its inputs (equation 1.1) and then applies an activation function f to derive the output (equation 1.2). Typically, the activation function is a non-linear one.

$$I_j = \sum_{i=0}^{n} w_{ij}\, x_i \qquad (2.1)$$

$$O_j = f(I_j) \qquad (2.2)$$

There are two major neural networks architecture: Feed-Forward Networks and Recurrent Networks. A **feed-forward neural network (FFN)** has connections only in one direction - that is, it forms a directed acyclic graph. Every node receives input from “upstream” nodes and delivers output to “downstream” nodes; there are no loops. A feed-forward network represents a function of its current input; thus, it has no internal state other than the weights themselves. A **recurrent neural network (RNN)**, on the other hand, feeds its outputs back into its own inputs. This means that the activation levels of the network form a dynamical system that may reach a stable state or exhibit oscillations or even chaotic behavior [28].

The Feed-Forward Network is usually arranged in layers, such that each unit receives input only from units in the immediately preceding layer. It can be either Single-Layer or Multi-Layer Network.

1) **Single-Layer Feed-Forward Neural Network**

A Single-layer Network has one layer of connection weights. The synaptic link carrying weights connect every input unit to every output unit. In the typical single-layer network (see figure 2.3), the input units are fully connected to output units but are not connected to other input units, and the output units are not connected to other output units.

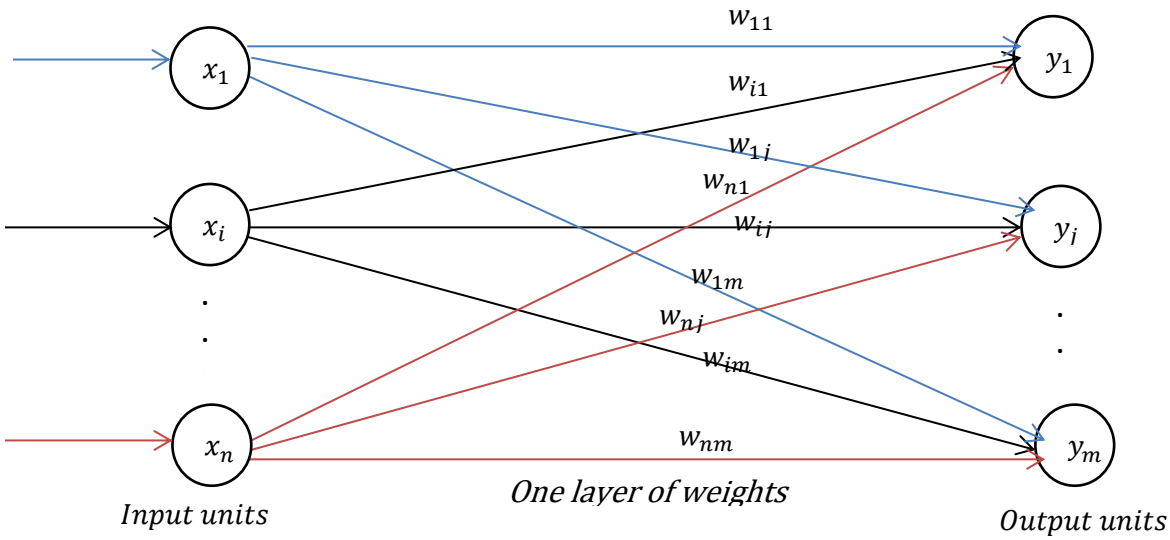

Figure 2.3: Single Layer Feed-Forward Network

2) **Multi-Layer Feed-Forward Neural Network**

A multilayer network has one or more layers called hidden layer between the input layer and the output layer (figure 2.4). The computational units of the hidden layer are known as hidden neurons. Multilayer networks can solve more complicated problems than can single-layer nets, but training may be more difficult.

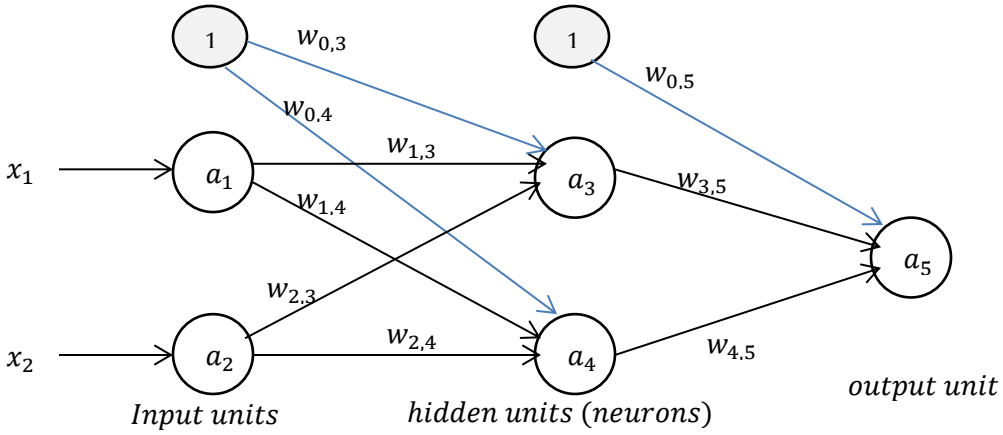

Figure 2.4: Example of a Multilayer Feed-Forward Network

Given an input $\boldsymbol{x} = (x_1, x_2)$, the activations of the input units are set to $(a_1, a_2) = (x_1, x_2)$. The output at unit 5 (expressed as a function of inputs and weights) is given by (see figure 2.4):

$$\begin{aligned} a_5 &= f(w_{0,5} + w_{3,5}a_3 + w_{4,5}a_4) \\ &= f\left(w_{0,5} + w_{3,5}\, f\left(w_{0,3} + w_{1,3}a_1 + w_{2,3}a_2\right) + w_{4,5}\, f\left(w_{0,4} + w_{1,4}a_1 + w_{2,4}a_2\right)\right) \\ &= f\left(w_{0,5} + w_{3,5}\, f\left(w_{0,3} + w_{1,3}x_1 + w_{2,3}x_2\right) + w_{4,5}\, f\left(w_{0,4} + w_{1,4}x_1 + w_{2,4}x_2\right)\right) \end{aligned} \quad (2.3)$$

2.1.3. Activation Functions

As mentioned earlier, the basic operation of an artificial neuron involves summing its weighted input signal an activation function to produce a calculated output. For the input layer, the function is an identity function (Equation 1.3). For hidden and output layers, a non-linear activation function is used. The activation function is expected to be continuous, differentiable and monotonically non-decreasing. For computational efficiency, it is desirable that its derivative be easy to compute [18].

1. **Identity function**

$$f(x) = x \quad \forall x \quad (2.4)$$

2. **Binary step function (with threshold θ):**

$$f(x) = \begin{cases} 1 & if\ x \geq \theta \\ 0 & if\ x \leq \theta \end{cases} \quad (2.5)$$

Single-layer nets is often used by a step function to convert the net input, which is a continuously valued variable, to an output unit that is a binary (1 or 0) or bipolar (1 or -1) signal (see Figure 2.5). A threshold value θ could be adopted as shown in (equation 1.4).

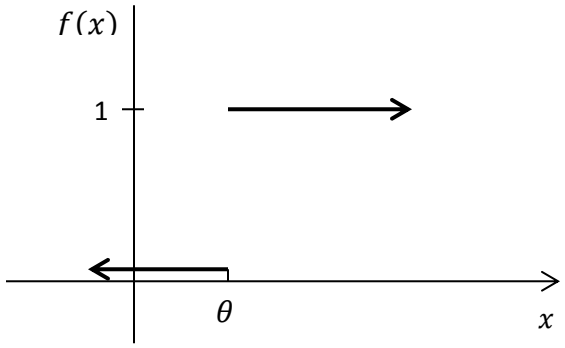

Figure 2.5: Binary step function

3. **Binary Sigmoid Function (S- shaped curve):** range of values of the function is between 0 and 1 (see figure 2.6). The function is continuous and easily differentiable. Hence, it is commonly used for multilayer perceptron.

$$f(x) = \frac{1}{1 + e^{-x}} \qquad (2.6)$$

$$f'(x) = f(x)[1 - f(x)] \qquad (2.7)$$

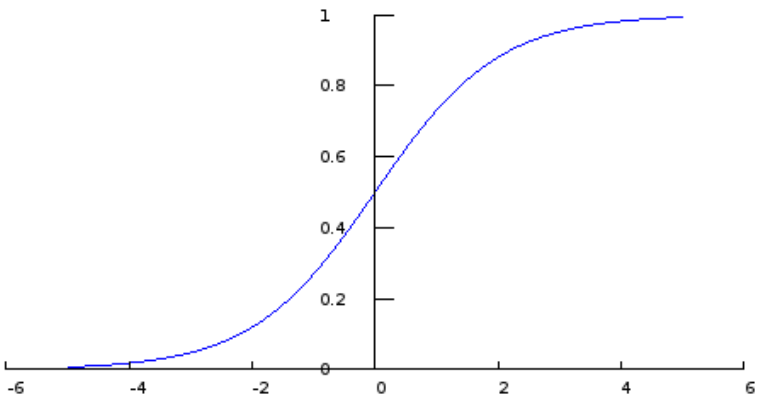

Figure 2.6: Binary Sigmoid function

4. **Bipolar Sigmoid Function:** The range of values of is between -1 and 1. The function is also continuous and differentiable like binary sigmoid.

$$f(x) = \frac{2}{1 + e^{-x}} - 1 \qquad (2.8)$$

$$f'(x) = \frac{1}{2}[1 + f(x)][1 - f(x)] \qquad (2.9)$$

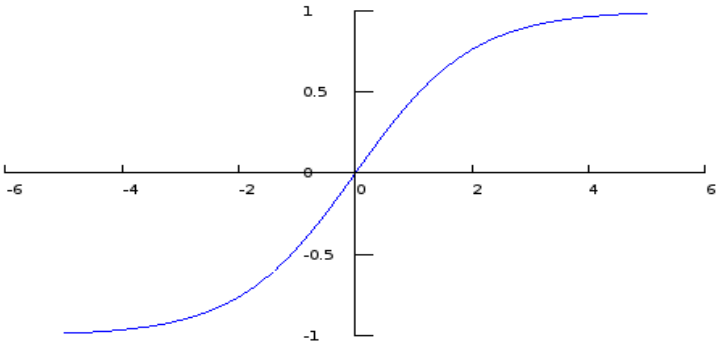

Figure 2.7: Bipolar Sigmoid function

Hyperbolic tangent function: The function is closely related to bipolar sigmoid function with desirable range of output values between -1 and 1.

$$f(x) = \tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \qquad (2.10)$$

$$f'(x) = sech^{2}(x) = 1 - (f(x))^{2} \qquad (2.11)$$

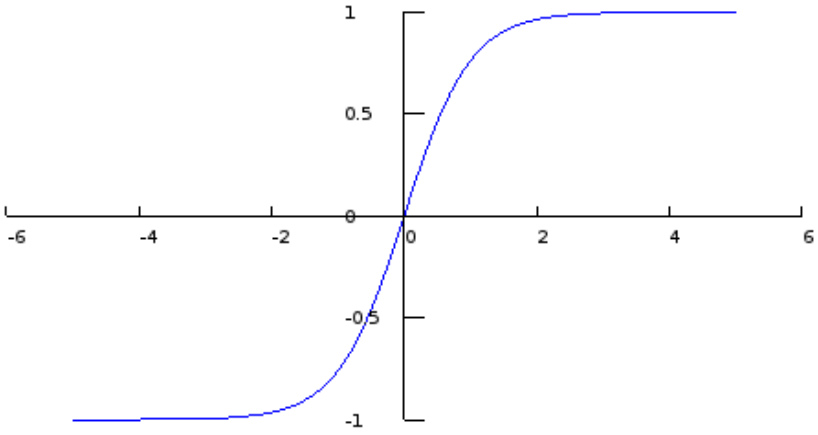

Figure 2.8: Hyperbolic Tangent function

5. **Gaussian Activation function:** The function is a bell-shaped curve that is continuous. Unlike others, it is not an increasing function; it maps high values into low ones and mid-range values into high ones. The output range is between 0 and 1.

$$f(x) = e^{-x^{2}} \qquad (2.12)$$

$$f'(x) = -2xe^{-x^{2}} \qquad (2.13)$$

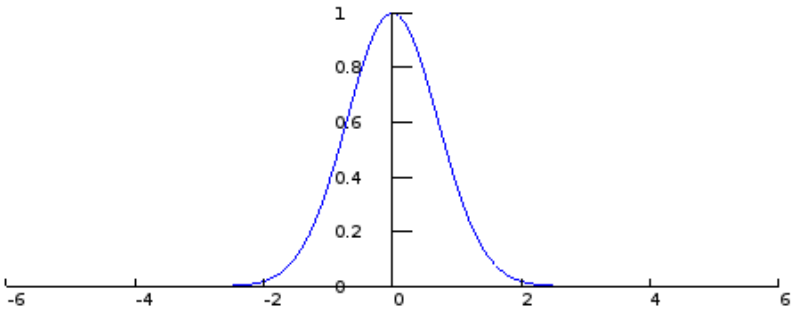

Figure 2.9: Gaussian function

For this thesis, we will be making use of binary sigmoid, bipolar sigmoid, hyperbolic tangent and Gaussian activation functions because they are continuous and differentiable.

2.1.4. Learning Algorithms

As a fact, neural networks cannot be predefined - they have to go through a learning process so they can be used. Depending on the learning algorithm used to train them, they can learn either very fast or very slow. The performance of the learning process for a given architecture can depend on multiple factors [14], [29]

- The learning algorithm.
- The data quality used to train the network. The learning data has to be chosen very carefully, it has to include the most information possible with least number of inputs.
- The configuration parameters.

The power of ANN comes from the learning process. It needs to be trained to map an input vector to the targeted output. Through the learning phase, the ANN adapts its transfer function to deliver the desired output [14]. There are two major types of learning algorithms – supervised and unsupervised learning.

1. Unsupervised learning

Unsupervised learning algorithms are used group similar input vectors together to identify hidden patterns when no specific output is requested of the network. This type of learning is used for data clustering, feature extraction and similarity detection [14]. The lack of direction for the learning algorithm in unsupervised learning can sometime be advantageous, since it lets the algorithm to look back for patterns that have not been previously considered. It is also referred to self-organizing neural networks. A good example is Kohonen self-organizing map (SOM) [30]

2. Supervised learning

Supervised learning is used for training data that has both the input and target vectors for each pattern. The learning process is usually a repetitive calculation using the activation functions to get the output pattern (feed-forward calculation). The difference between the desired output and calculated one (error) is used to correct the network behavior by adjusting the weights to give a

more appropriate output (error correction). Typically, supervised learning uses gradient descent for the minimization of errors between the desired output and calculated output.

Gradient Descent Learning

Gradient descent method is a way to find a local minimum of a function. This is based on the minimization of error E_p by taking the gradient of the function in terms of weights and activation function. The process is repeated in the negative direction of the gradient until it eventually converges to zero (which corresponds to a local minimum). The activation function of the network is required to be differentiable because the update of the weight is dependent on the gradient of the error E_p [31].The error function for a pattern p is expressed as:

$$E_p = \frac{1}{2}\sum_{i=0}^{n}(y_i - t_i)^2 \qquad (2.14)$$

$$where\ \ y_i\ is\ the\ calculated\ output\ and\ t_i\ is\ target\ output$$

If ΔW_{ij} is the weight update of the link connecting the ith and the jth neuron of the two neighboring layers, then ΔW_{ij} is defined as

$$\Delta W_{ij} = -\eta \frac{\partial E}{\partial W_{ij}} \qquad (2.15)$$

where η is the learning rate (usually $\eta > 0$) parameter and $\frac{\partial E}{\partial W_{ij}}$ is error gradient with reference to the weight W_{ij}

All the learning algorithms considered in this thesis are gradient descent based.

2.1.4.1. Standard Back Propagation (BP) Algorithm

The BP is a type of supervised learning with error correction methodology. It uses a gradient-based technique to minimize the error function equivalent to the MSE (equation 2.12) between the desired and actual network outputs. The BP algorithm propagates backward the error between the desired signal and the network output through the network. After providing an input

pattern, the output of the network is then compared with a given target pattern and the error of each output unit calculated. This error signal is propagated backward, and a closed-loop control system is thus established [32].

Back propagation Algorithm MLP with one hidden layer

Step 0: Initialize randomly weights for all layers preferably in range -0.5 and 0.5

Step 1: While stopping condition is false, do step 2-10

Step 2: For each training pair do steps 3-9

Feed-Forward:

***Step* 3**: $I_j = \sum_{i=0}^{n} w_{ij} x_i$

Step 4: $o_j = f(I_j)$ $\quad where\ f\ is\ the\ activation\ function$

***Step* 5**: $n_k = \sum_{j=0}^{m} w_{jk} o_j$

Step 6: $y_k = f(n_k)$

Error Calculation

Step 7: $\delta_k = f'(n_k)(y_k - t_k)$ $\quad where\ \ t_k\ is\ the\ targeted\ output\ for\ neuron\ k$

$$\Delta w_{jk} = \eta \frac{\partial E}{\partial w_{jk}} = \eta \delta_k o_j \qquad (2.16)$$

Step 8: $\delta_j = f'(I_j)\left(\sum_{k=0}^{p} w_{jk}\delta_k\right)$ $\quad p\ is\ the\ size\ of\ output\ neurons$

$$\Delta w_{ij} = \eta \frac{\partial E}{\partial w_{ij}} = \eta \delta_j x_i \qquad (2.17)$$

Weight Update

Step 9: $w_{jk}(t+1) = \ w_{jk}(t) + \Delta w_{jk}$ $\quad where\ t\ is\ the\ iteration\ step$

$$w_{ij}(t+1) = \ w_{ij}(t) + \Delta w_{ij}$$

Step 10: Testing stopping condition. This is usually the mean square error for all patterns.

$$\text{For all patterns, Mean Square Error (MSE)} = \sum_{p \varepsilon P}(E_p) = \frac{1}{2}\sum_{p \varepsilon P}\left(\sum_{i=0}^{n}(y_i - t_i)^2\right) \qquad (2.18)$$

One of the drawbacks of using the error gradient function to calculate the error is being trapped in a local minima and never getting the global minima. Despite the disadvantage, the BP learning algorithm is still the most popular learning rule for performing supervised learning tasks [33], [34] It is not only used to train FNNs such as the MLP, it has also been adapted to RNNs [35]

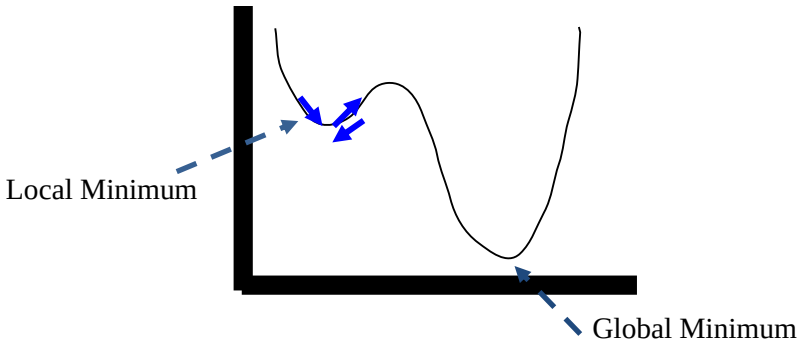

Figure 2.10: Local Minimum

In order to implement the BP algorithm, a continuous, nonlinear, monotonically increasing, differentiable activation function is required.

2.1.4.2. Back Propagation with Momentum Algorithm

A momentum term was introduced in the BP algorithm[4] by Rumelhart. The weight change is in a direction that is a combination of the current gradient and the previous gradient. This is a modification of gradient descent whose advantages arise chiefly when some training data are very different from the majority of the data (and possibly even incorrect)[18]. It is used to solve the problem of local minimum experienced in the standard BP algorithm as shown in figure 2.9

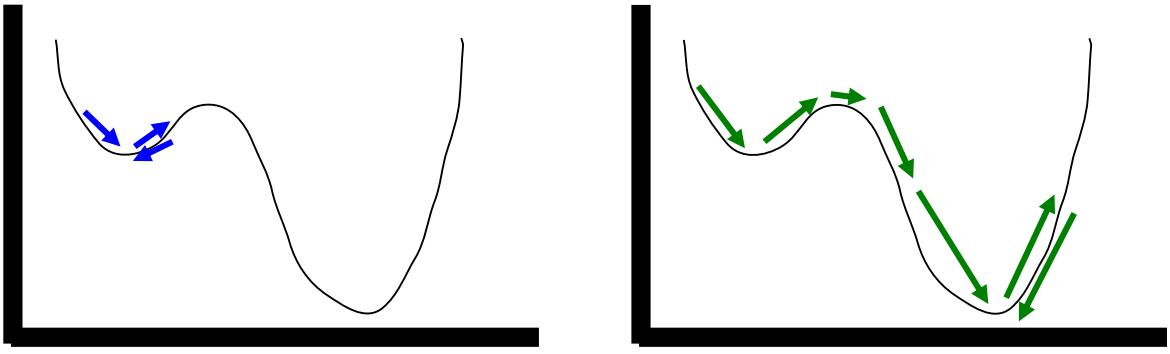

Figure 2.11: Effects of Momentum to Overcome Local Minimum

Equations (2.16) and (2.17) in Back propagation Algorithms are modified to give

$$\Delta w_{jk}(t) = \eta \frac{\partial E}{\partial w_{jk}} + \alpha \Delta w_{jk}(t-1)$$

$$\Delta w_{ij}(t) = \eta \frac{\partial E}{\partial w_{ij}} + \alpha \Delta w_{ij}(t-1) \qquad (2.19)$$

Limitations to the effectiveness of momentum include the fact that the learning rate places an upper limit on the amount by which a weight can be changed and the fact that momentum can cause the weight to be changed in a direction that would increase the error [36]

2.1.4.3. Silva and Almeida (SA) Algorithm

F. M. Silva and L. B. Almeida[37] are using separate learning rates for each connection instead of a single one used in BP. The adaptation of these learning rates is done by observing the signs of the last two gradients. As long as no change in sign is detected the corresponding learning rate is increased by η^+ . If the sign changes the learning rate is decreased by η^-. The weight update $\Delta w_{ij}(or\ \Delta w_{jk})$ in equation (2.16 and 2.17) of BP algorithm is achieved by the following:

a) Choose some small initial value for every $\eta_{ij}(0)$

b) Adapt the learning rates:

$$\eta_{ij}(t) = \begin{cases} \eta_{ij}(t-1) * \eta^+ & ,if\ \frac{\partial E}{\partial w_{ij}}(t) * \frac{\partial E}{\partial w_{ij}}(t-1) \geq 0 \\ \eta_{ij}(t-1) * \eta^- & ,else \end{cases} \qquad (2.20)$$

c) Update the connections:

$$\Delta w_{ij}(t) = sign\left(\eta_{ij}(t)\right) * \left(\frac{\partial E}{\partial w_{ij}} + \alpha * \Delta w_{ij}(t-1)\right) \qquad (2.21)$$

Note: $sign(n) = \begin{cases} 1\ ,if\ n > 0 \\ -1\ ,\ n < 0 \\ 0\ ,if\ n = 0 \end{cases}$ (2.22)

The choice of the parameter is important for the efficiency of the learning algorithm. The recommended increase factor $\eta^+ > 1$ and decrease factor $\eta^- < 1$. Also, small initial values for $\eta_0(0)$ has to be chosen. In addition to this update rule, Silva and Almeida use a global

backtracking strategy which restarts an update step if the total error increases. In this case, both acceleration and deceleration learning rates are halved.

2.1.4.4. Delta-Bar-Delta Algorithm

Robert A. Jacobs proposed another local learning rate adaptation algorithm [36] that is similar to Silva and Almeida's technique. It differs from the former approaches through the control of learning rates by observing the sign changes of an exponential average gradient (equation 2.23). Based on this, the learning rate makes linear acceleration or exponential deceleration. The steps are:

1. Choose some small initial value for every $\eta_{ij}(0)$.
2. Compute the learning rate deltas $\Delta\eta_{ij}(t)$ required for adapting the learning rates

$$\Delta\eta_{ij}(t) = \begin{cases} \eta^{+}, & ,if\ \frac{\partial E}{\partial w_{ij}}(t) * \bar{\delta}(t-1) > 0 \\ -\eta^{-} * \eta_{ij}(t-1), & if\ \frac{\partial E}{\partial w_{ij}}(t) * \bar{\delta}(t-1) < 0 \\ 0 & ,otherwise \end{cases} \qquad (2.23)$$

$\bar{\delta}(t)$ denotes the exponential average gradient

3. Adapt the learning rates

$$\eta_{ij}(t) = \eta_{ij}(t-1) + \Delta\eta_{ij}(t) \qquad (2.24)$$

4. Compute the exponential average gradient $\bar{\delta}(t)$

$$\bar{\delta}(t) = (1-\phi)\frac{\partial E}{\partial w_{ij}}(t) + \phi\bar{\delta}(t-1) \qquad (2.25)$$

ϕ is known as the delta-bar constant

5. Update the deltas and weights.

$$\Delta w_{ij}(t) = \eta_{ij}(t) * \frac{\partial E}{\partial w_{ij}}(t) + \alpha\Delta w_{ij}(t-1) \qquad (2.26)$$

Usually, $\Delta w_{ij}(t)$ is computed without the momentum, α because it may sometimes cause the delta-bar-delta algorithm to diverge.

$$w_{ij}(t) = w_{ij}(t-1) + \Delta w_{ij}(t)$$

The values of η^{+}, η^{-} and ϕ are positive numbers specified by the user. Jacob recommends ϕ =0.7. All $\eta_{ij}(t)$ are initialized with small values.

2.1.4.5. Quickprop Algorithm

This algorithm is a collection of different heuristics for optimizing back propagation [38]. It is a second-order method, based heuristically on Newton's technique. Basically, this algorithm relies on a second-order approximation of the optimal weight step. In real situations, the update rule makes algorithm particularly unstable. Some control is then required.

Step 1: Compute the optimal weight step as shown below

$$\tilde{\alpha}_{ij}(t) = \frac{\frac{\partial E}{\partial w_{ij}}(t)}{\frac{\partial E}{\partial w_{ij}}(t-1) - \frac{\partial E}{\partial w_{ij}}(t)} \qquad (2.27)$$

Step 2: If the step computed by this formula is too large, infinite or uphill on the current gradient, this momentum has to be limited. Schiffmann et al [39] proposes to control the momentum term as follows:

$$\alpha_{ij}(t) = \alpha_{max}, \quad \textbf{if } \tilde{\alpha}_{ij}(t) \; is \; infinite$$
$$\vee \; \tilde{\alpha}_{ij}(t) > \alpha_{max}$$
$$\vee \; \tilde{\alpha}_{ij}(t) * \Delta w_{ij}(t-1) * \frac{\partial E}{\partial w_{ij}}(t-1) > 0 \qquad (2.28)$$
$$\alpha_{ij}(t) = \tilde{\alpha}_{ij}(t), \quad else$$

Step 3: A learning rate is also necessary to start the training or restart it after a gradient sign change.

$$\eta_{ij}(t) = \eta_0, \quad if \; \frac{\partial E}{\partial w_{ij}}(t) * \Delta w_{ij}(t-1) < 0.0$$
$$\vee \; \Delta w_{ij}(t-1) = 0.0$$
$$\eta_{ij}(t) = 0, \quad else$$

Step 4: Hence, The weight change is given as

$$\Delta w_{ij}(t) = sign\left(\eta_{ij}(t)\right) * \frac{\partial E}{\partial w_{ij}} + \alpha_{ij}(t) * \Delta w_{ij}(t-1)) \qquad (2.29)$$

Benchmark experiments show that the Quickprop typically performs very reliably and converges very fast (Rojas R, 1994). Recommend values for the parameters are: $\alpha_{max} = 1.75, \eta_0 = 0.55$

2.1.4.6. Resilient Back propagation

A variant of Silva and Almeida's algorithm is RPROP, first proposed by Riedmiller and Braun [40]. To overcome the inherent disadvantages of pure gradient-descent, RPROP tries to update the network weights using just the learning rate and the sign of the partial derivative of the error function with respect to each weight. This accelerates learning mainly in flat regions of the error function as well as when the iteration has arrived close to a local minimum. To avoid accelerating or decelerating too much, a minimum value for the learning rates Δmin and a maximum value Δmax are enforced [32].

To compute the new weights $w_{ij}(t+1)$, update-value Δ_{ij} was introduced to determine the size of the weight update Δw_{ij}. This adaptive update-value evolves during the learning process based on its local sight on the error-function E**,** according to the following learning-rule

$$\Delta_{ij}(t) = \begin{cases} \eta^+ * \Delta_{ij}(t-1)\ ,\ if\ \frac{\partial E}{\partial w_{ij}}(t) * \frac{\partial E}{\partial w_{ij}}(t-1) > 0 \\ \eta^- * \Delta_{ij}(t-1)\ ,\ if\ \frac{\partial E}{\partial w_{ij}}(t) * \frac{\partial E}{\partial w_{ij}}(t-1) < 0 \\ \Delta_{ij}(t-1) \qquad\qquad\qquad\qquad ,\ else \end{cases} \tag{2.30}$$

$$where\ 0 < \eta^- < 1 < \eta^+$$

Once the update-value for each weight is adapted, the weight-update follows the simple rule:

$$\Delta w_{ij}(t) = \begin{cases} -\Delta_{ij}(t)\ ,\ if\ \frac{\partial E}{\partial w_{ij}}(t) > 0 \\ \Delta_{ij}(t)\ ,\ if\ \frac{\partial E}{\partial w_{ij}}(t) < 0 \\ 0, \qquad\qquad ,\ else \end{cases} \tag{2.31}$$

$$w_{ij}(t+1) = \quad w_{ij}(t) + \Delta w_{ij}$$

However, there is one exception: If the partial derivative changes sign, i.e. the previous step was too large and the minimum was missed, the previous weight-update is reverted:

$$\Delta w_{ij}(t) = -\Delta w_{ij}(t-1)\ ,\ if\ \frac{\partial E}{\partial w_{ij}}(t) * \frac{\partial E}{\partial w_{ij}}(t-1) < 0 \tag{2.32}$$

The parameters Δmax and Δmin are introduced to cap the minimum and maximum step size taken, values suggested by the authors are 0.000001 and 50, and an initial Δ_{ij}value of 0.1. The recommended values for η^- and η^+ is 0.5 and 1.2 respectively.

Research [39] shows that Resilient Propagation out performs other propagation methods such as Back Propagation, SA and Delta-Bar-Delta both on constructed test cases and in some real world data tests.

The RPROP algorithm for Error Correction

$\forall i,j: \frac{\partial E}{\partial w_{ij}}(t-1) = 0$

for all weights and biases{

if $\left(\frac{\partial E}{\partial w_{ij}}(t) * \frac{\partial E}{\partial w_{ij}}(t-1) > 0\right) > 0$ ***then***{

$\Delta_{ij}(t) = minimum(\Delta_{ij}(t-1) * \eta^+, \Delta\max)$

$\Delta w_{ij}(t) = -sign\left(\frac{\partial E}{\partial w_{ij}}(t)\right) * \Delta_{ij}(t)$

$\frac{\partial E}{\partial w_{ij}}(t-1) = \frac{\partial E}{\partial w_{ij}}(t)$

}

else if $\left(\frac{\partial E}{\partial w_{ij}}(t) * \frac{\partial E}{\partial w_{ij}}(t-1) > 0\right) < 0$ ***then***{

$\Delta_{ij}(t) = maximum(\Delta_{ij}(t-1) * \eta^-, \Delta\min)$

$\Delta w_{ij}(t) = -\Delta w_{ij}(t-1)$

$\frac{\partial E}{\partial w_{ij}}(t-1) = 0$

}

else if $\left(\frac{\partial E}{\partial w_{ij}}(t) * \frac{\partial E}{\partial w_{ij}}(t-1)\right) = 0$ ***then***{

$\Delta w_{ij}(t) = -sign\left(\frac{\partial E}{\partial w_{ij}}(t)\right) * \Delta_{ij}(t)$

$\frac{\partial E}{\partial w_{ij}}(t-1) = \frac{\partial E}{\partial w_{ij}}(t)$

}

$w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}$

}

2.1.5. Applications of ANN

a) Clustering

A clustering algorithm explores the similarity between patterns and places similar patterns in a cluster. Clustering is particularly useful in data analysis and machine learning, including pattern classification, system modeling, data mining, document retrieval, and image segmentation [32]. Many clustering neural networks and competitive learning-based algorithms such as SOM [41] and ART [31] models are also widely used for clustering

b) Classification and Pattern recognition

ANN has been used extensively for pattern recognition. One specific area in which many neural network applications have been developed is the automatic recognition of handwritten characters (digits or letters) [18]. Neural networks are also used for speech recognition, speech synthesis, and classification of radar / sonar signals, remote sensing and image classification.

c) Function approximation

The task of function approximation is to find an estimate of the unknown function subject to noise. Most applications of neural networks utilize the function approximation capability of neural networks. These include modeling and system identification, regression and prediction, control, signal processing and associative memory. Image restoration is also a function approximation problem.

d) Prediction Systems:

The task is to forecast some future values of a time-sequenced data. Prediction has a significant impact on decision support systems. There are quite a number of real-world applications that makes use of ANN for prediction and forecasting. Examples are Hospital patient stay length prediction, Jury summoning prediction, Natural gas price prediction, Financial Time series forecasting and Stock market prediction.

2.2. Discrete Event System Specification (DEVS)

The Discrete event system specification (DEVS) is a formalism introduced by Zeigler in 1976 to describe discrete event system in a hierarchical and modular manner. It is modular because the system has input and output ports that allows it to interact with the external environment [9]. The DEVS formalism describes a system as a mathematical expression using set theory. It is a theoretically well-defined system formalism [42]. The DEVS models are seen as black boxes with input and output ports used to describe system structure as well as system behavior. As a result, DEVS offers a platform for modeling and simulation (M&S) of complex systems in different domains. It provides a mechanism to mix different formalisms as well as a generic mechanism for M&S [43]

The DEVS formalism distinguishes between the modeling and simulation approaches. The DEVS modeling uses the concept of hierarchy by offering two types of models: atomic models and coupled models. On the other hand, the simulation approach is responsible for the automatic generation of the simulation algorithms for the DEVS models. 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 [1], [44].

The original DEVS that was proposed called Classic DEVS (C-DEVS) had constraints that originated with the sequential operation of early computers and hindered the exploitation of parallelism. The Parallel DEVS (P-DEVS) [45] equips bags to accommodate multiple input messages and confluent function to handle simultaneous internal and external events [2]. The following important concepts of DEVS formalism are explained below.

2.2.1. The DEVS Modeling

A system under study can be modeled in DEVS by two groups of models: atomic models and coupled models. The atomic model describes the system behavior using the inputs, output, states of the model and the relationship between them. On the other hand, the coupled model is used to describe the structure of the system under study. The coupled model is composed of a group of sub-models (atomic and/or coupled) that are interconnected in a manner that delivers the global behavior of the system.

2.2.1.1. Atomic Classic DEVS Model

A atomic C-DEVS model is defined as a 7-tuple

$$\boldsymbol{AM} = < \boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{S}, \boldsymbol{\delta_{int}}, \boldsymbol{\delta_{ext}}, \boldsymbol{\lambda}, \boldsymbol{ta} >$$

Where

$X = \{(p, v) | \, p \in InPorts, v \in X_p\}$ is the set of input events where $InPorts$ is the set of input ports and X_p is the set of input values.

$Y = \{(p, v) | \, p \in OutPorts, v \in Y_p\}$ is the set of output events where $OutPorts$ is the set of output ports and Y_p is the set of output values.

S is the set of state variables

$\delta_{int}: S \to S$ is the internal transition function

$ta: S \to \mathbb{R}^{+}_{0,+\infty}$ is the time advance function

$\delta_{ext}: Q \times X \to S$ is the external transition function

where $Q = \{(s, e) | s \in S, e \in [0, ta(s)]\}$ is the set of total states and e is the time elapsed since the last transition

$\lambda: S \to Y$ is the output function

A DEVS model is always in a state $s \in S$ at a given time. The model can transit from one state to another using transition functions δ_{int} and δ_{ext}. In the absence of external events, it remains in the state for a lifetime of $ta(s)$. When ta(s) is reached, the model outputs value $y \in Y$ through its port (this is the only possibility for the model to generate output) using the output function $\lambda(s)$, then it changes to a new state defined by $\delta_{int}(s)$. A transition that is as a result of the expiration of $ta(s)$ is known as internal transition. In the case of an external events triggered by external inputs, the external transition function determines the new state given by $\delta_{ext}(s, e, x)$, where s is the current state, e is the time elapsed since the last transition, and $x \in X$ is the external event received.

Time-advance function for a state $ta(s)$ can take a real value between 0 and ∞. If $ta(s) = 0$, the model will trigger an immediate transition, the state is called a transient state. In contrast, if $ta(s) = \infty$, the model will remain in this state until an external event occurs. This is called a passive state.

Figure 2.12 shows an example of a state trajectory of a DEVS model with an initial state s_1.

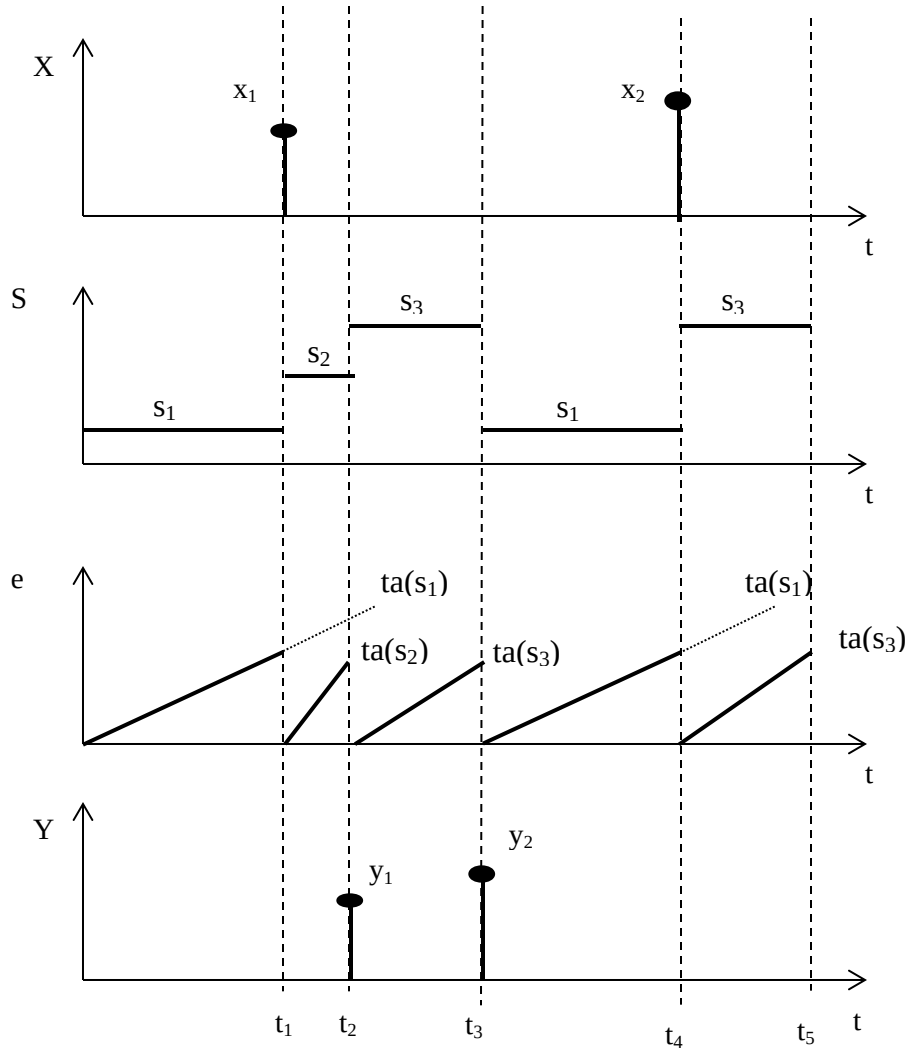

Figure 2.12: Atomic Model State Trajectory

In the figure above, the system receives an external event at t_1 before the expiration of $t(s_1)$. The new state that the system will transit to is defined by $\delta_{ext}(s_1, e, x_1) = s_2$ where e is the elapsed time that increases from 0 to the point of transition the state s_1 and x is the external input. Internal events occurred at t_2 and t_3 as a result of the expiration of $t(s_2)$ and $t(s_3)$ respectively leading to the output of y_1 and y_2. Another external event received at t_4 causes the system to change to state s_3 . In this case, $\lambda(s_1)$ cannot produce output because $e < ta(s_1)$.

2.2.1.2. Coupled Classic DEVS Model

A DEVS coupled model is composed of several atomic or coupled sub-models. It is formally defined by

$$CM = < X, Y, D, \{M_d \mid d \in D\}, EIC, EOC, IC, select >$$

Where

$X = \{(p, v) | \, p \in InPorts, v \in X_p\}$ is the set of input ports and values.

$Y = \{(p, v) | \, p \in OutPorts, v \in Y_p\}$ is the set of output ports and values.

D is the set of the component or model names.

M_d is a DEVS model $\forall \, d \in D$

EIC is the set of external input couplings which connects external inputs to components inputs i.e. $EIC \subseteq \left\{\left((Self, in_{self}), (j, in_j)\right) | \, in_{self} \in InPorts, j \in D, in_j \in InPorts_j\right\}$

EOC is the set of external output couplings that connects components outputs to the external outputs. $EOC \subseteq \left\{\left((Self, out_{self}), (i, out_i)\right) | \, out_{self} \in OutPorts, \; i \in D, in_i \in InPorts_i\right\}$

IC is the set of internal couplings which connects component outputs to component inputs. $IC \subseteq \left\{\left((i, out_i), (j, in_j)\right) | \, i, j \in D, out_i \in OutPorts_i, in_j \in InPorts_j\right\}$

$Select: 2^D \backslash \emptyset \rightarrow D$ is the tiebreaker function used to select a model to execute in case of simultaneous internal events.

Coupled models group several DEVS into a composite model that can be regarded, due to the coupling property, as a new DEVS model. The closure property guarantees that the coupling of several class instances results in a model of the same class, allowing hierarchical construction[46].

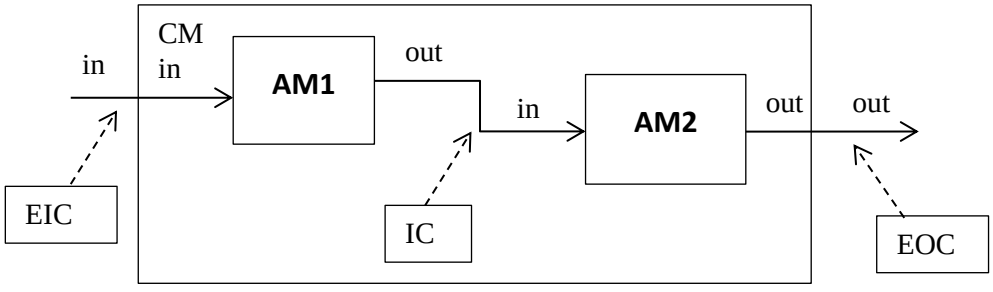

Figure 2.13: Example of a Coupled Model

In Classic DEVS, multiple subcomponents can be scheduled for an internal transition at the same time, ambiguity could arise. In figure 2.13, if AM1 executes its internal transition first, producing an output that maps into an external event for AM2 (which is also scheduled for an

internal transition at the same time), then it is not clear which transition AM2 should execute first. There are two alternatives for this:

- to execute the external transition first and then the internal transition, with $e = ta(s)$; or
- to execute the internal transition first, followed by the external transition, with $e = 0$.

The select function provides a simple way to solve this ambiguity. The function defines a priority for the components of a coupled model in case of simultaneous internal events. The one with the highest priority is chosen.

2.2.2. The DEVS Simulation

One of the most important DEVS properties is that it can automatically generate a dedicated simulator for each model. A generic simulation framework has been proposed [2] that mirrors the hierarchical structure of DEVS model being simulated. There is a DEVS simulator corresponding to each atomic model, and a DEVS coordinator corresponding to each coupled model. At the top of the hierarchy, a special coordinator called the root coordinator is in charge of controlling the progress of the simulation. Figure 2.14 illustrates the simulation hierarchy.

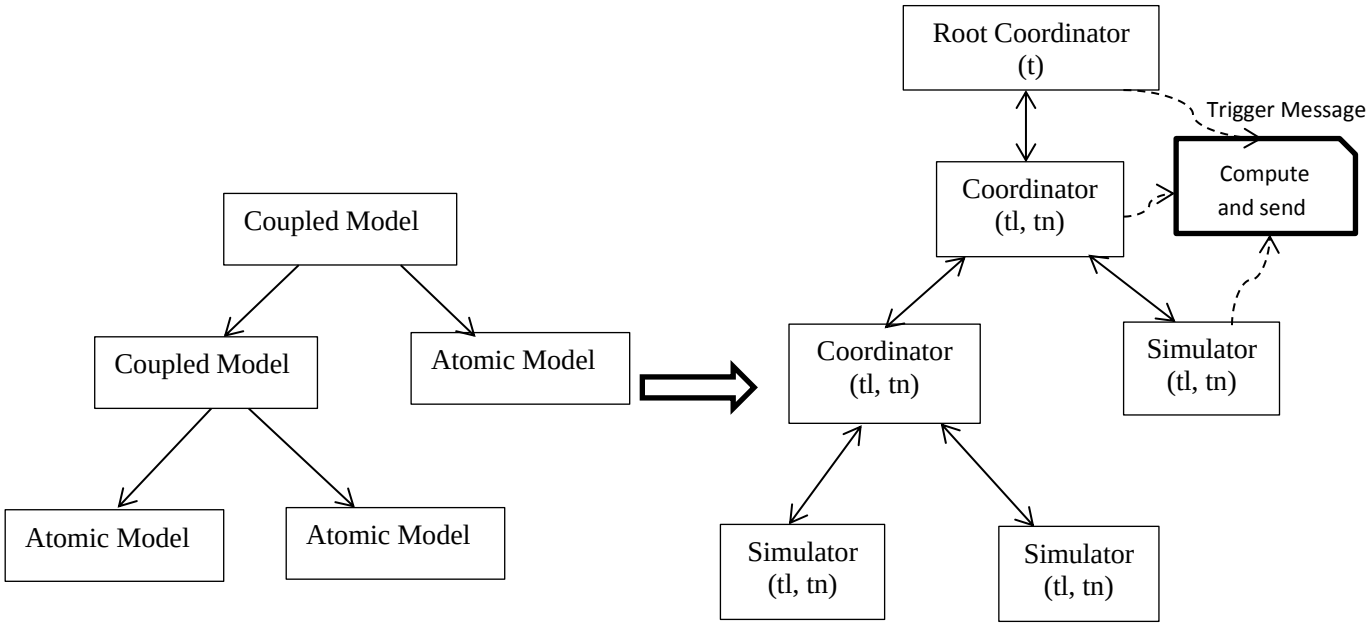

Figure 2.14: DEVS Simulation Strategy

The DEVS simulator has no event calendar. Instead, it uses two simulation time variables, tl for time of the last event occurred, and tn for the time of the next scheduled (internal) event. The simulator transfers the value of tn its parent coordinator.

The coordinator maintains an event calendar which is a list of pairs (d, t) where d is the name of a subordinate component and t is the time of its next internal event. It is sorted by the event time t and the priority of conflicting models provided by the Select function.

The root coordinator maintains a tn variable containing the time of the next scheduled event within the entire model.

Before beginning the simulation, the root coordinator sends an initialization message (i-message) which is propagated down the tree. Each atomic simulator initializes its state, calculates the tn value, and makes this value available to the parent coordinator. The process continues bottom-up till the root coordinator receives the minimum tn. After the initialization, the simulation proceeds in iterations following the following steps:

i. The root coordinator sends a message (*-message) with its tn value which is propagated down the tree. At each tree level, it is routed based on its event calendar to the atomic model with the minimum timestamp that is equal to the root's tn value.
ii. The selected atomic model invokes its output function λ, and computes its new state using the internal transition function δ_{int}.
iii. If there is an output message (y-message) generated by the λ function immediately before the internal transition, this message propagates through the tree. In each coordinator which it passes on the way, it is translated by means of the EIC functions of the corresponding coupled model in form of an external input (x-message) to some atomic models.
iv. All atomic models which underwent a state transition update their tl and tn times and the new values are propagated up the tree. The coordinators update their calendars using these values and eventually the root coordinator gets the new global minimum timestamp. This completes the simulation iteration.

2.2.3. DEVS SimStudio Simulation Package

SimStudio[15] is an architecture built upon the DEVS formalism that aims at integrating tools for M&S, analysis and collaboration through Model-Driven Engineering (MDE) features such as

code generation [47]. SimStudio Simulation Package (a component of SimStudio) is an Object-Oriented implementation of simulation algorithm for C-DEVS and P-DEVS written in Java. The tool provides an efficient and easy-to-use implementation for DEVS.

The figures below show the class diagrams for the SimStudio implementation. The AbstractSimulator class keeps track of the simulation times and events of a model, it has two sub-classes, Simulator and Coordinator for managing the simulations of Atomic and Coupled DEVS models respectively. Every Simulator or Coordinator collaborates with its parent in order to ensure the interaction, mainly message passing between the model it simulates and other models in the system. The RootCoordinator manages the simulation of the entire system model.

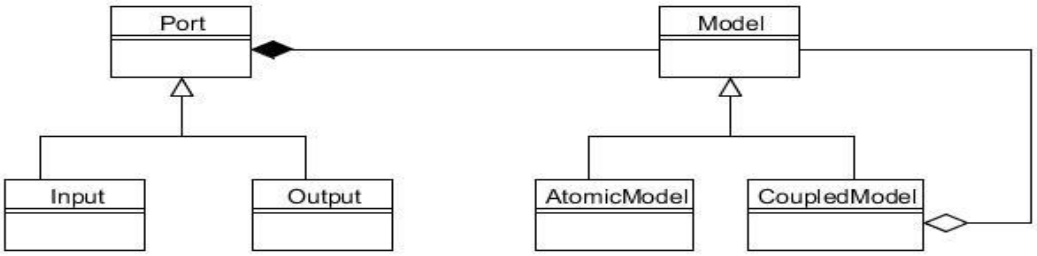

Figure 2.15: Model Class Diagram

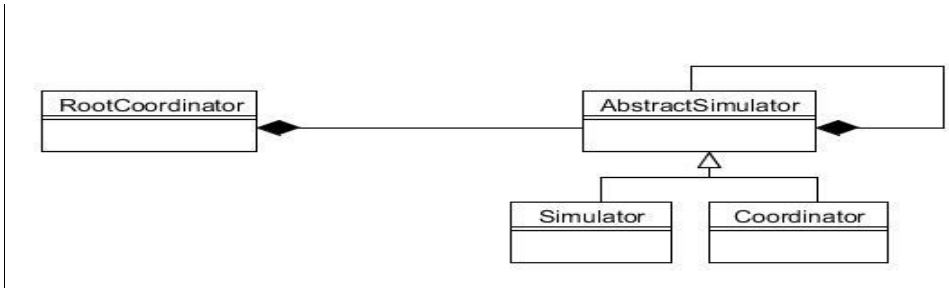

Figure 2.16: Simulator Class Diagram

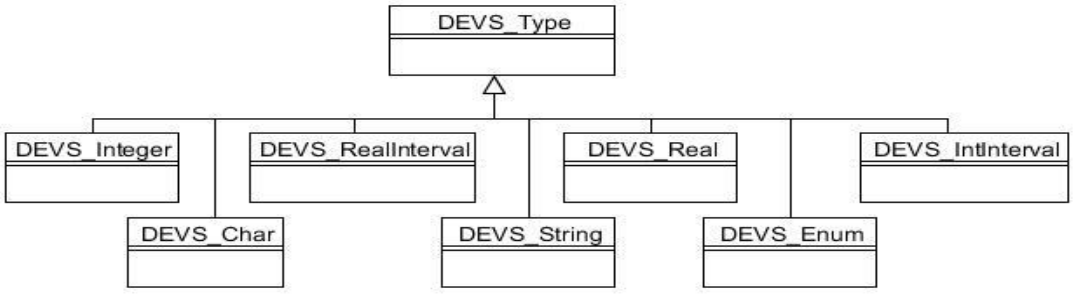

Figure 2.17: DEVS Data Type Class Diagram

2.3. High Level Language for System Specification (HiLLS)

2.3.1. HiLLS Architecture

HiLLS [6] is a visual modeling language that has its roots in system theory and software engineering. It is a composition of formalisms used in scientific techniques such as simulation, prototyping and formal analyses for investigating the properties and behaviors of complex systems. Thus, it promises to be a many-in-one specification language for exhaustive study of systems.

HiLLS evolve from the DEVS-Driven Modeling Language (DDML)[48]. DDML is a visual simulation modeling language originally meant to combine modeling concepts from DEVS and Unified Modeling Language (UML) [49] to facilitate the simulation modeling process by enhancing the communication between domain and simulation experts.

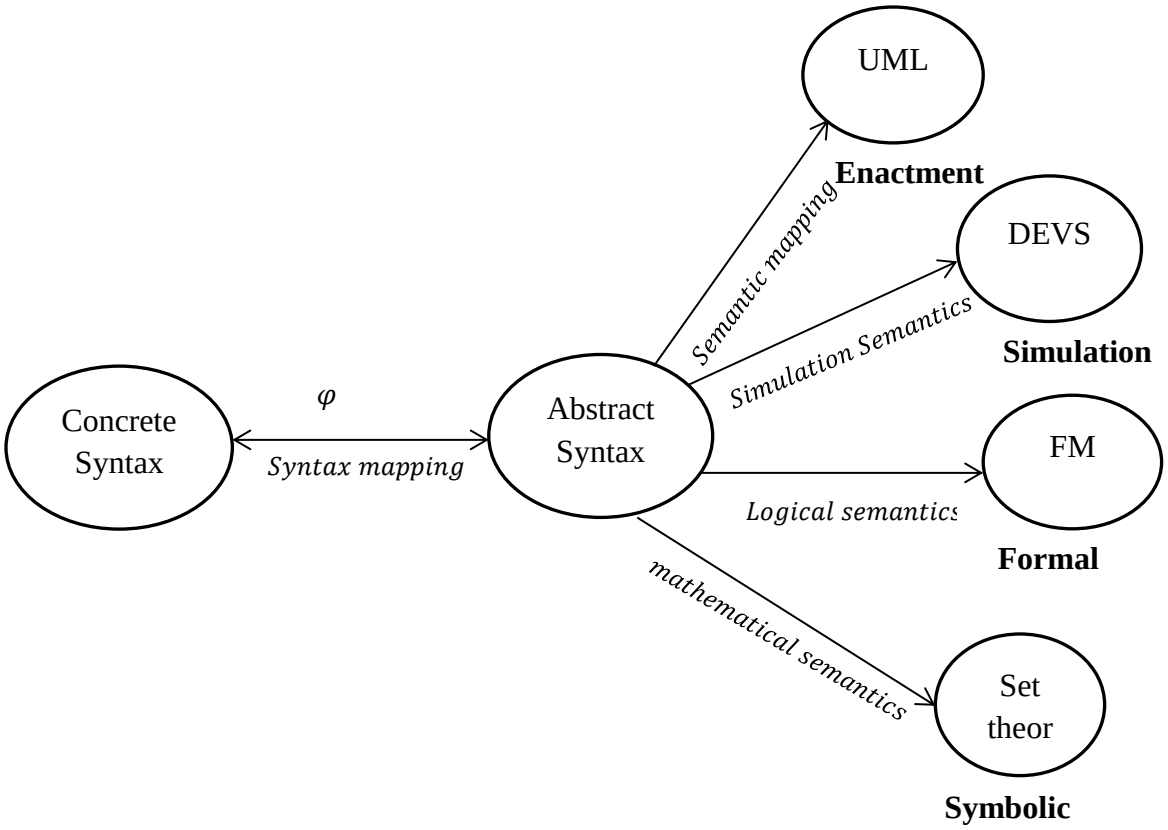

Figure 2.18: Components of HiLLS Specification

HiLLS combine the system-theoretic abstractions of DEVS with Software Engineering concepts from the UML and Z-system specification language (Z) [50] in one coherent whole. The

language inherits and extends some universal graphical notations from the UML family to render the structure and behavior of systems. This is further strengthened with Z-schema to foster complete specifications of functional and behavioral properties; this would also enhance the complete syntheses of executable program.

2.3.2. Concrete Syntax of HiLLS

The configurations in HiLLS are classified according to the time advance function in DEVS.

- $ta \in \mathbb{R}^+ - \{0\}$ defines a configuration that has a time span (Finite Configuration which is represented with four compartment; Label, Declarations, Operations, Sub Configuration)
- $ta = 0$ defines a configuration that has no time attached to it, like it is an instant passage to other configuration (Transient Configuration which is represented with an Oval shape to show its Zero timing)
- $ta = \infty$ defines a configuration that has an indefinite time span attached to it (Passive Configuration which is represented with four compartment; Label, Declarations, Operations, Sub Configuration but with a strip at the side to show its infinity).

The transitions are the same as those defined in DEVS, but with modifications in their graphical representations.

- All Internal transitions must go into a configuration through the left hand side and exit through the right hand side and it is represented with a Hard Line and an arrow filled head.
- All External transition must exit a configuration through its centre top or its centre bottom and it is represented with Dashed Line and an arrow filled head.
- All Confluent transition exits through the rightmost top or bottom and it is represented with a Dotted Line and an arrow filled head.

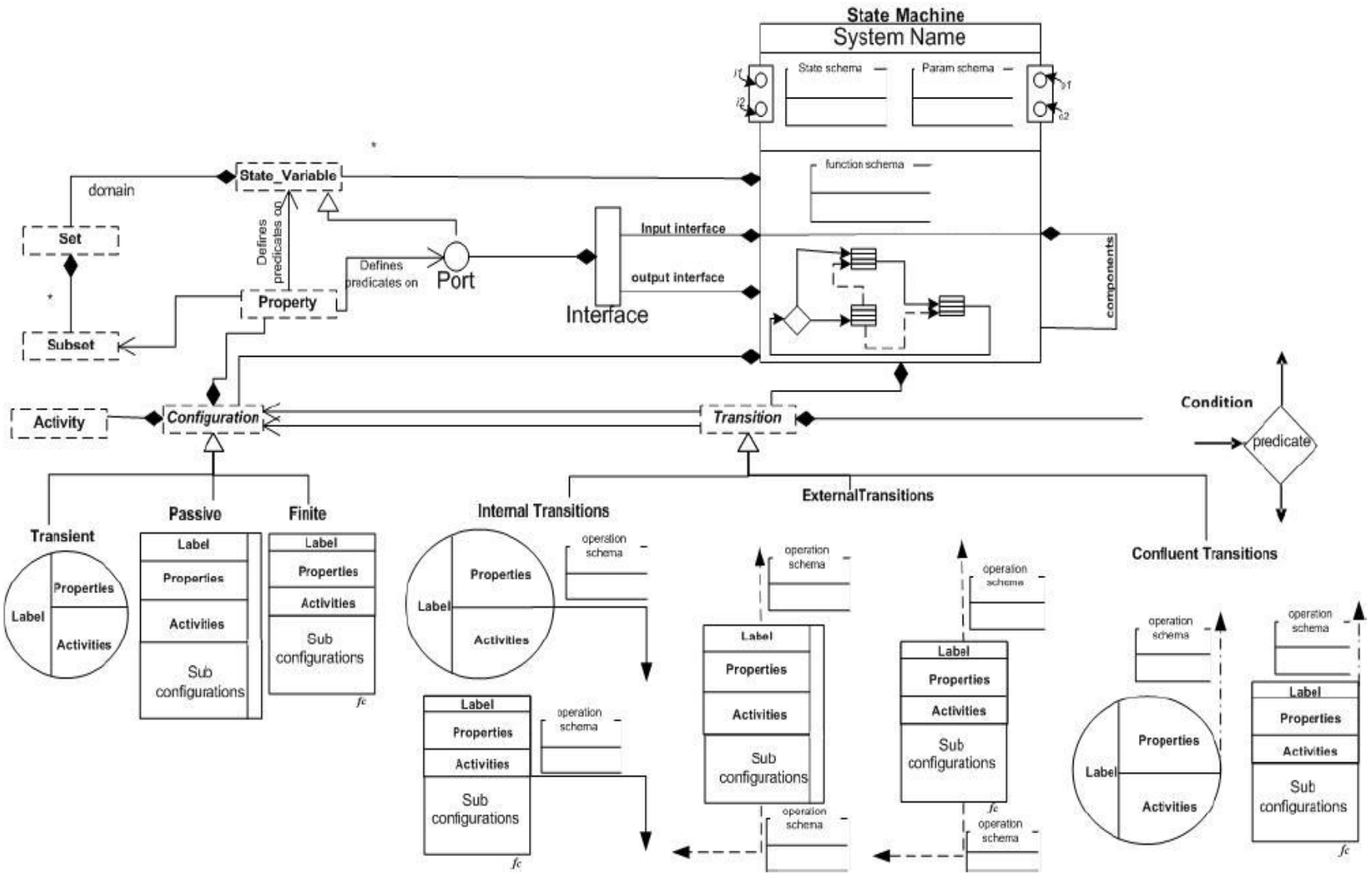

Figure 2.19: Concrete Syntax of HiLLS

The representation of HiLLS model is shown in figure 2.20 below:

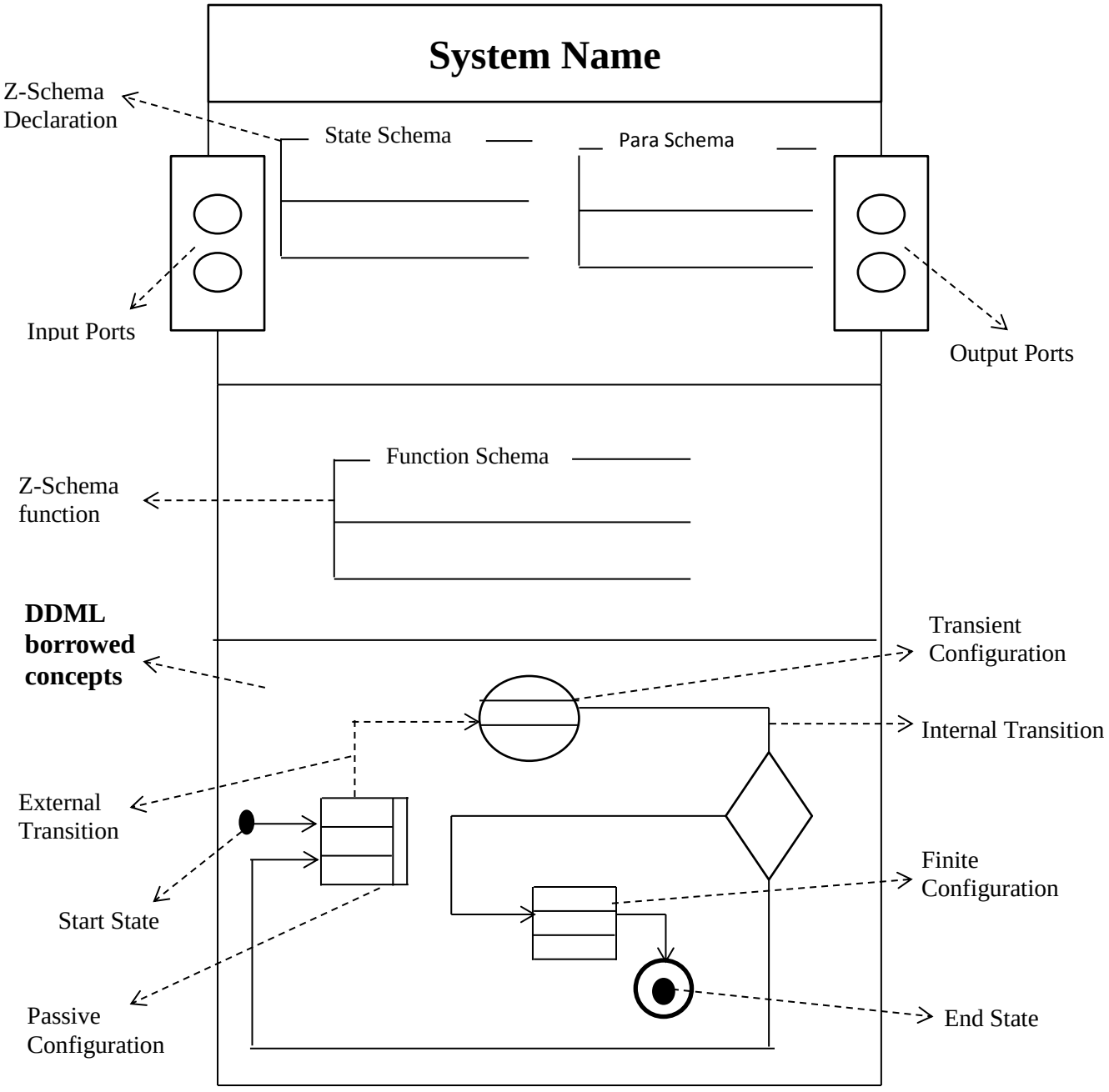

Figure 2.20: HiLLS Model

2.3.3. HiLLS Example: Single Lane Road Model

Consider a Single Lane Road System that accepts a car and moves it from one cell position to another. A car in a cell is represented by 1 and no car is 0. (See figure 2.21).

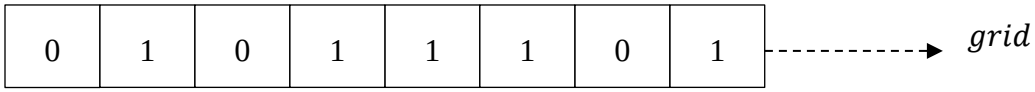

Figure 2.21: Single Lane Road

The number of possible states is 2^n, where n is the number of cells. However, we can categorize the states into: empty state (no car in the system), unblocked state (at least a car can move to the next cell), blocked state (no car can move to the next cell). Also, an accept state is added to handle car received external event.

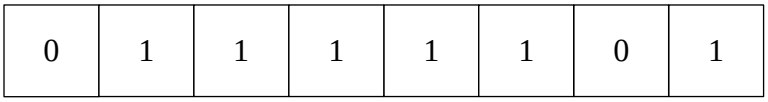

Figure 2.22: Example of an Unblocked State

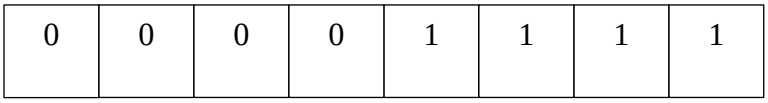

Figure 2.23: Example of a Blocked state

The system is starts with an empty state, when a car is received; it transits to an unblocked state. After the expiration of $2seconds$, the car moves to the next cell and the grid is outputted; if no car can move in the lane, it transits to blocked state else it returns back to the unblocked state (internal transition). When an external event occurs, the system transits to the accept state with $ta = 0$; if there is no car in the first cell and there is possibility of car movement, it transits to unblocked state else it moves to blocked state.

The HiLLS diagram is given below.

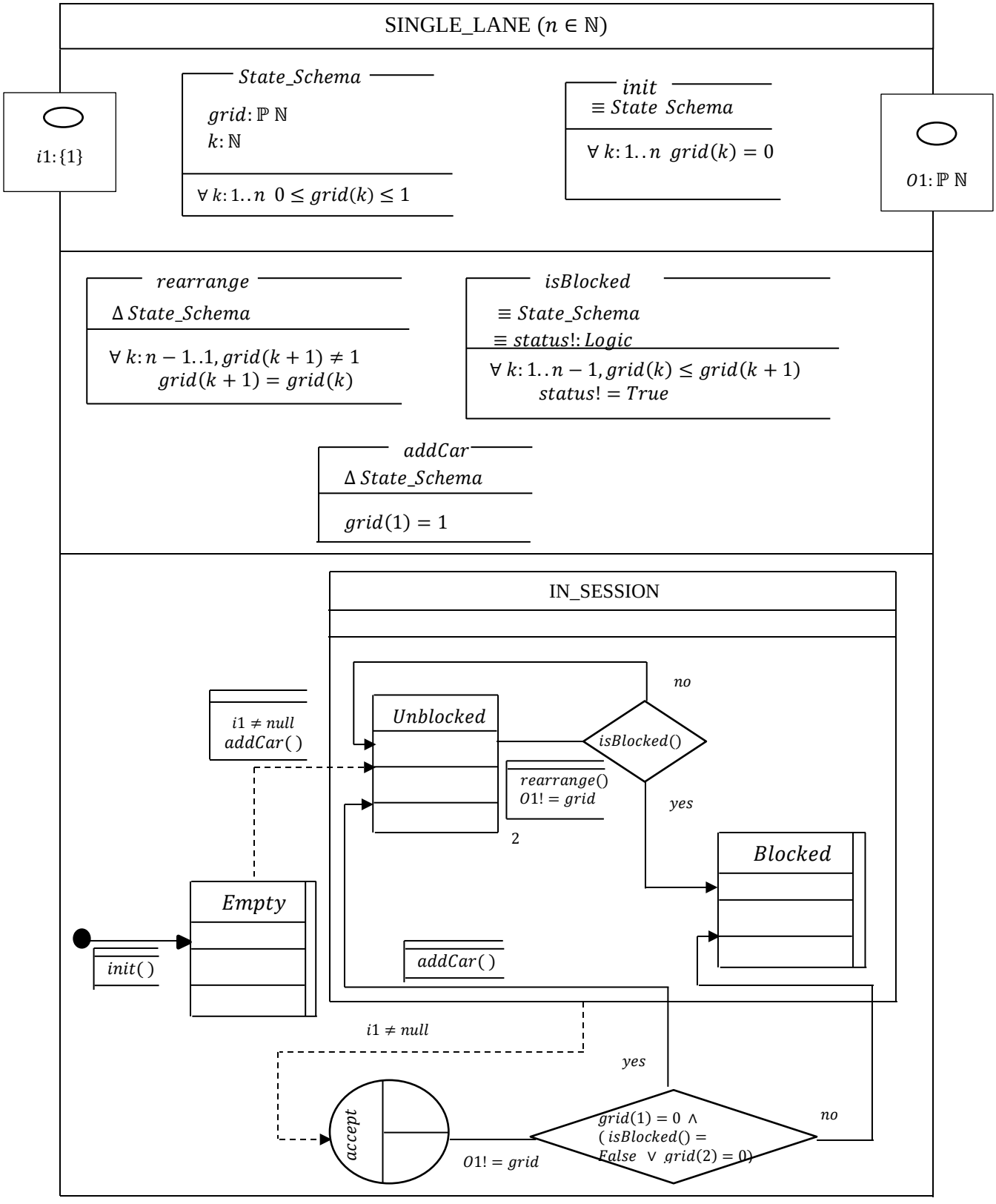

Figure 2.24: Single Lane HiLLS Model

CHAPTER 3

3.0 DEVS- BASED ANN

3.1. DEVS-Based ANN Approach

In this section, we will describe the four atomic models in the DEVS-Based ANN [14] using a multilayer feed-forward neural networks and some gradient descent algorithms . The first model is the non-calculation (or input) layer that forwards inputs to the first calculation (hidden) layer. The calculation layer model describes the structure of the hidden and output layers. The error-generator model and delta-weight models are used for obtaining network result and error correction respectively.

3.1.1. DEVS-Based ANN Design

Figure 3.1 shows the DEVS-based neural network modeling design. There are four blocks representing the ANN layers and the separate learning models. The input layer is considered as the *non-calculation layer*. Hidden and Output layers are the *calculation layers*. *Error-Generator* and *Delta-Weight* are the two models that are used by the ANN learning phase. After the neural network has been trained, one can do prediction by simply removing the learning models.

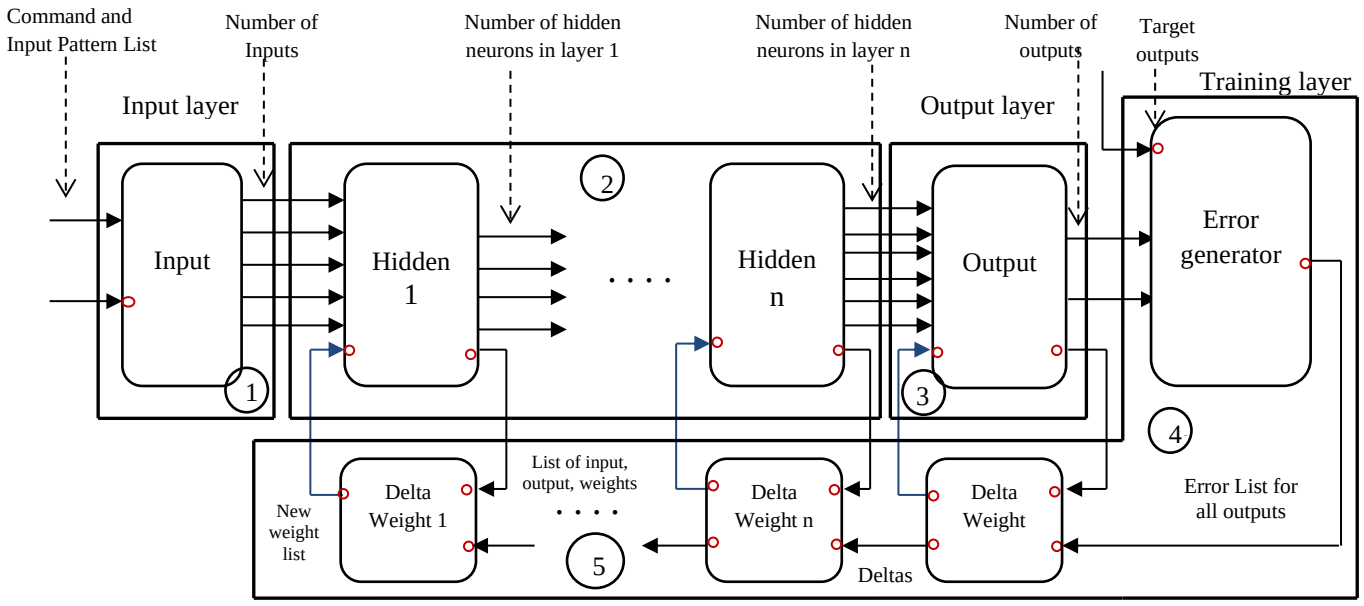

Figure 3.1: DEVS-Based Neural Networks Architecture

All the ports with circle are ports used only in training phase. As shown in the Figure 3.1, the connections between the training layer and the calculation models are basically for learning purposes.

The Input atomic model forwards data to the hidden layer (shown in step 1 in the figure), and then each hidden layer (including the Output layer) computes the weighted sum and activation functions. In steps 2 and 3, each hidden or output layer sends output data to the next layer and learning data (inputs, outputs and weights) to the delta-weights. As soon as the output layer sends its output (ANN calculated outputs) to the *error-generator* model, it sends error (difference between the target output and calculated output) to the *Delta-Weight* models (step 4). The Delta-Weight model calculates the new error (delta) from the learning data and the weight change from any of the gradient descent algorithms specified. After that, the delta-weight model will back-propagate error to another delta-weight model if it exist and weight list will be sent to the corresponding calculation layer (step 5).

The entire cycle explained above is considered as one learning iteration (or epoch) that is repeated (steps from 1 to 5) depending on the stopping condition.

3.1.2. Feed-Forward Calculations Model Set

The feed-forward calculation consists of non-calculation (input) layer and calculation layer (at least one hidden layer and one output layer).

3.1.2.1. Non-Calculation Layer Atomic Model

This model has the responsibility of receiving all training input patterns and forwards the patterns sequentially to the first hidden layer. The non-calculation layer has two input ports. The first port is to signal that all patterns have been received while the second port is responsible for receiving the input patterns.

The DEVS atomic model is in a passive state with $ta = +\infty$. When an event is received from port 2 (input pattern), the input pattern is stored and a counter N is incremented. However, on receiving a '1' message from the first port, the model computes $dt = 1/N$, and transits to an active state. The model remains active with a life time of $1/N$, where N is the number of patterns used for training. After each dt, it forwards each value in the input pattern list through different

output ports to the calculation layer; a counter p is incremented when all the values are forwarded. The number of output ports is equal to the number of values inside each input pattern. When $p = N$, this means that all patterns have been sent for training; p resets back to 0 for retraining purpose.

3.1.2.2. Calculation Layer Atomic Model

The calculation layer model represents any hidden or output layer, as both layers have the same behavior. This model has the responsibility of calculating weighted sum and activation function which will be sent to another calculation model or error generator (in case of output layer).The calculation layer has two special ports reserved for learning: one among the input ports (port a) and the other among the output ports (port b).

The model is in a passive state with $ta = +\infty$. When messages are received through the input ports (except port a), it transits to an active state with $ta = 0$. Immediately, it computes the calculated outputs through the activation function; after which it sends the calculated outputs through its output ports and the learning data through port b before returning to passive state. If message is received from port a, the weights of the layer are updated.

3.1.3. Back-Propagation Learning Model Set

This section consists of error-generator model and delta-weight atomic models for error backward propagation and weight adjustment.

3.1.3.1. Error-Generator Atomic Model

The model receives calculated output from the last calculation layer (output layer) and also target outputs. It compares the two outputs by computing the difference known as *error*. In an ideal situation, the error between the target output and calculated output should be zero. Unfortunately, there is always a percentage error. The error list is forwarded to the delta-weight atomic model of the corresponding output layer to re-calibrate the network for better performance.

The model has one special input port for target output list (port a). The model is in a passive state with $ta = +\infty$. When a message is received from port a, the model stores the output pattern and increment a variable ($N = N + 1$); but it remains in the passive state. However, if there are inputs from other ports, the model transit to an active state with $ta = 0$ where the error E_k is computed.

$$E_k = t_k - y_k \ \ \forall k \in [0, N] \qquad (3.1)$$

$$E_{Total} = \frac{1}{2}\left(\sum_{k=0}^{N} (E_k)^2\right) \qquad (3.2)$$

Where N is the number of neurons in the output layer.

If a minimum error ($minError$) is provided, training is stopped (error is not forwarded to the delta-weight model) as soon as $minError < E_{Total}$.

3.1.3.2. Delta-Weight Atomic Model

The *Delta-Weight* model has two input ports and two output ports. It receives learning data from the calculation layer through the first input port. The learning data consist of the input, output and weight list of the corresponding calculation layer. On the other hand, the second port receives the error list from another delta-weight model or error generator (for the last delta-weight).

The Delta-Weight model has two states (passive and active). The model is in a passive state with $ta = +\infty$. When it receives a message from the first port, it stores the learning data required for weight update and remains in the passive state. On receiving a message from the second port, it transits to active state with $ta = 0$; then, it computes the new error and the new weights. The new error is sent to another delta-weight model if available while the new weights are sent to the corresponding calculation model.

3.2. HiLLS Description of DEVS-Based ANN

In this session, we will describe the DEVS-Based ANN in section 3.1 with HiLLS. The expressive power of HiLLS makes it easy to understand the architecture of ANNs. We are able to express all the arithmetic and logical expressions because of the rich mathematical language used (Z-Schema). We will add two models that are components of the Experimental Frame (input and target generators). They supply data used for training the neural networks.

3.2.1. Input Generator (IGEN)

IPattern: is a list of input or output in a single pattern (usually many patterns are used for training a neural network). Example {0.35, 0.9}

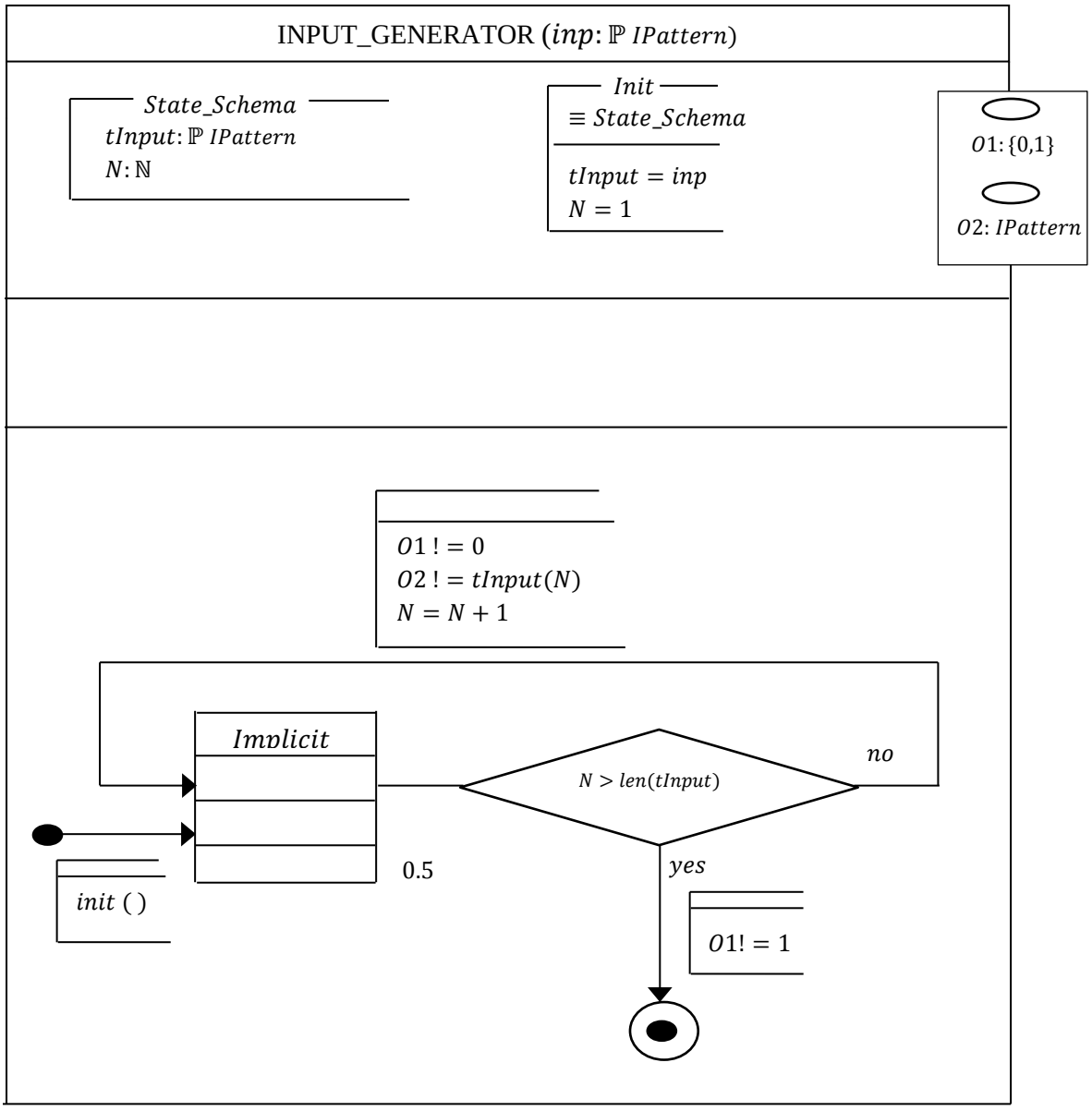

Figure 3.2: Input Generator HiLLS Model

3.2.2. Non-Calculation Layer Atomic Model (NC)

tInput: is the list of all training patterns

$port\ i1$ is added to signal to the NC model that all training patterns have been sent

m is the number of inputs

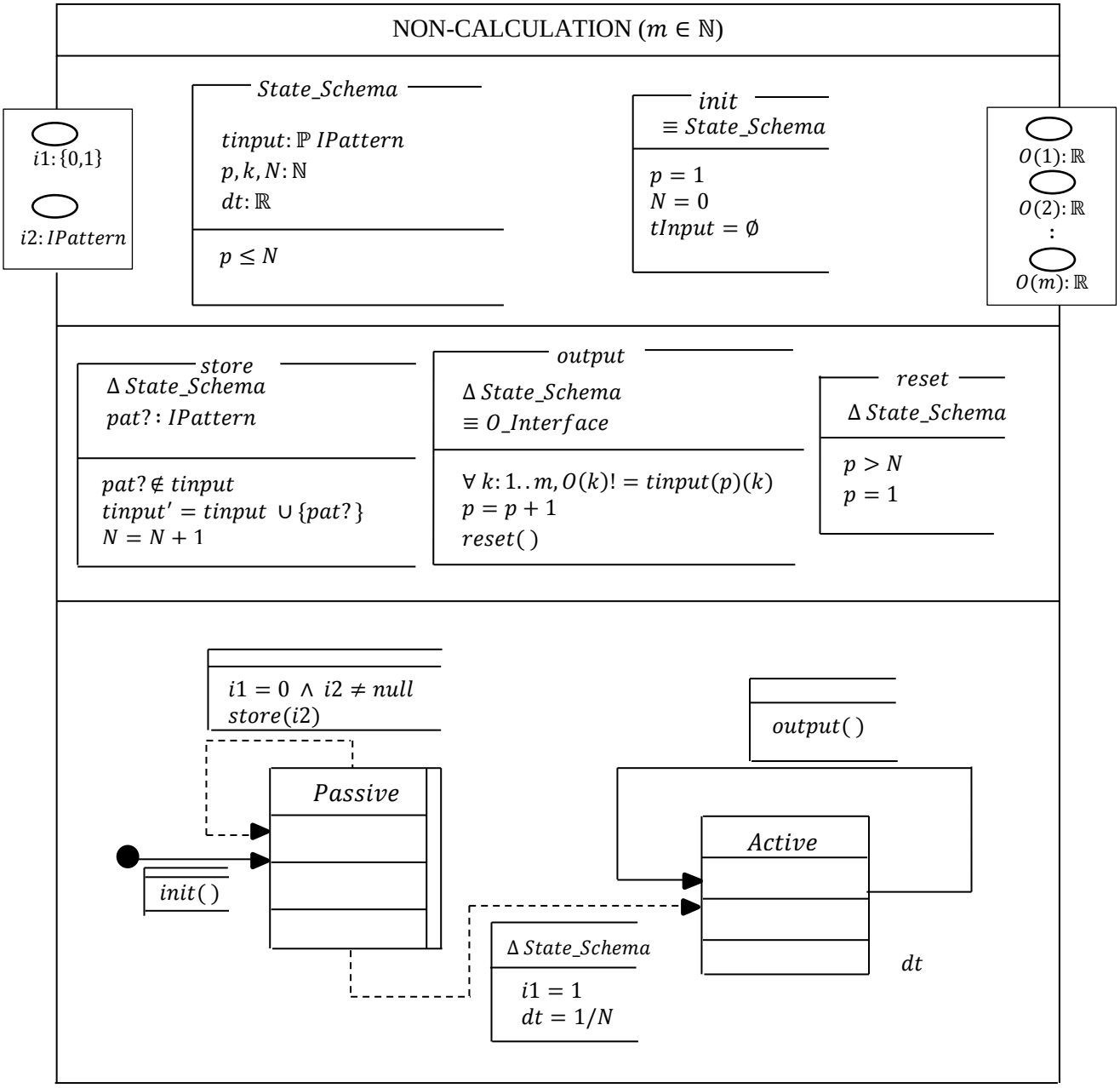

Figure 3.3: Non-Calculation Layer HiLLS Model

3.2.3. Calculation Layer Atomic Model (CAL)

Weight: is a data type expressed as w_{ij} to shows connection between i input and j output.

LData: is expressed as a record that has weights, input and output. The values are needed by the DELTA-WEIGHT model. ***m*** is the number of input neurons and **n** is the number of output neurons.

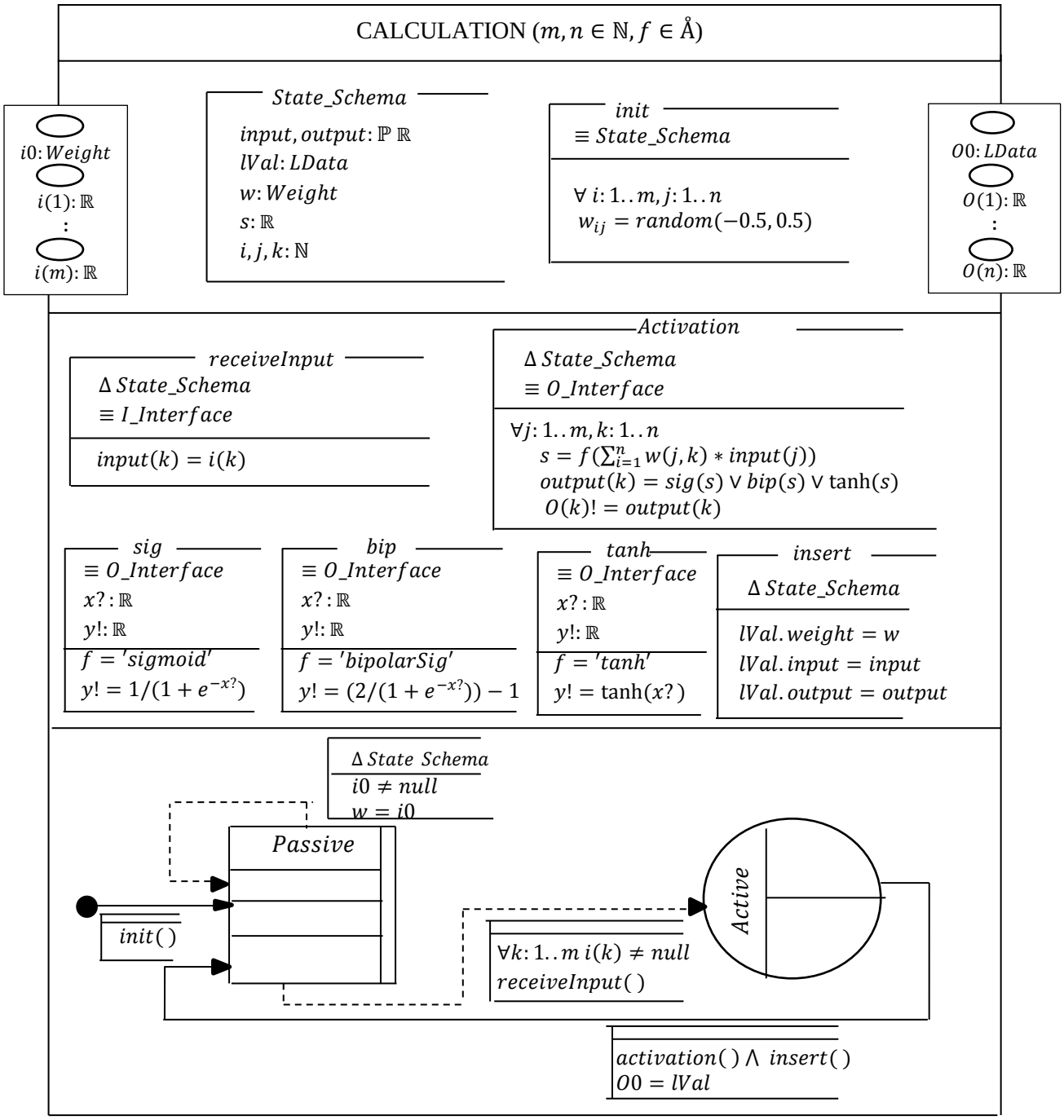

Figure 3.4: Calculation Layer HiLLS Model

3.2.4. Target Generator (TGEN)

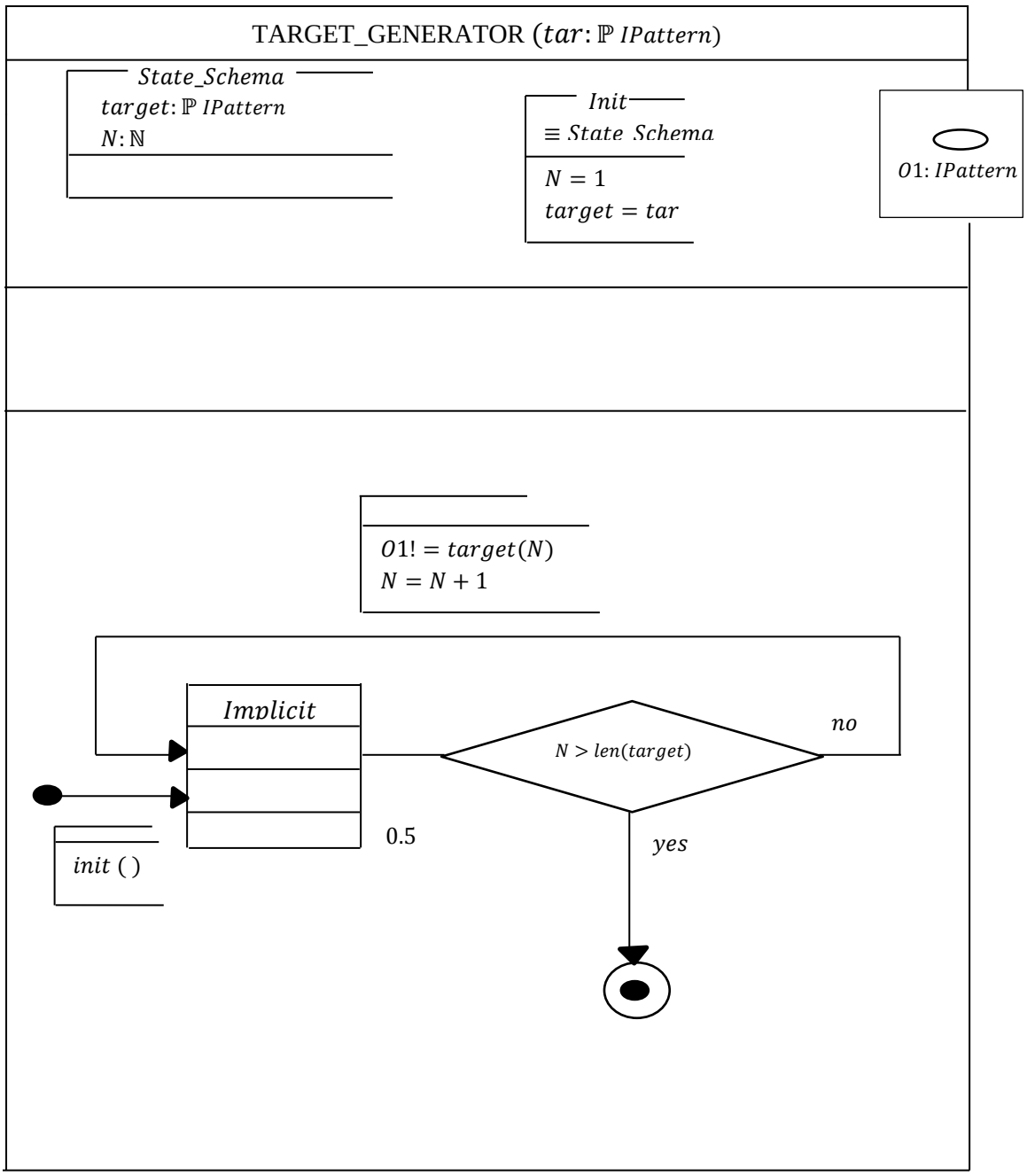

Figure 3.5: Target Generator HiLLS Model

3.2.5. Error Generator (ERR)

$\boldsymbol{m}$ is the number of calculated outputs. $\boldsymbol{minE}$ is the minimum error.

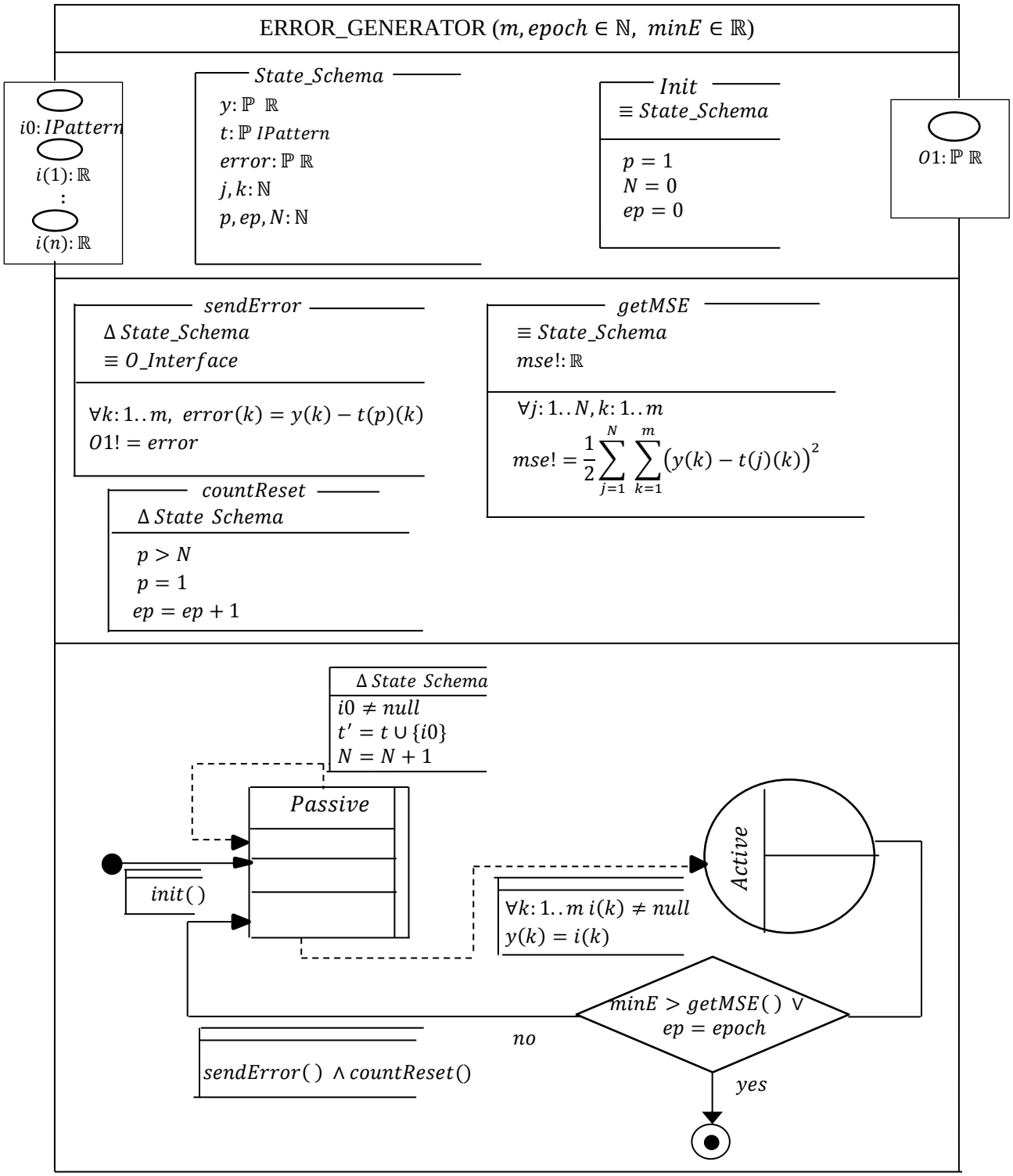

Figure 3.6: Error Generator HiLLS Model

3.2.6. Delta-Weight Atomic Model (DW)

m is the length of input and ***n*** is the length of output. The algorithms described in section 2.1.4.1 to 2.1.4.6 are used. BP (algo) returns delta for standard back propagation (BP). EBP (algo) returns delta for Momentum BP. SA (algo) returns delta for Silva and Almeida. DB (algo) return delta for Delta-Bar-Delta. QP (algo) returns delta for Quickprop, and RP(algo) for RPROP.

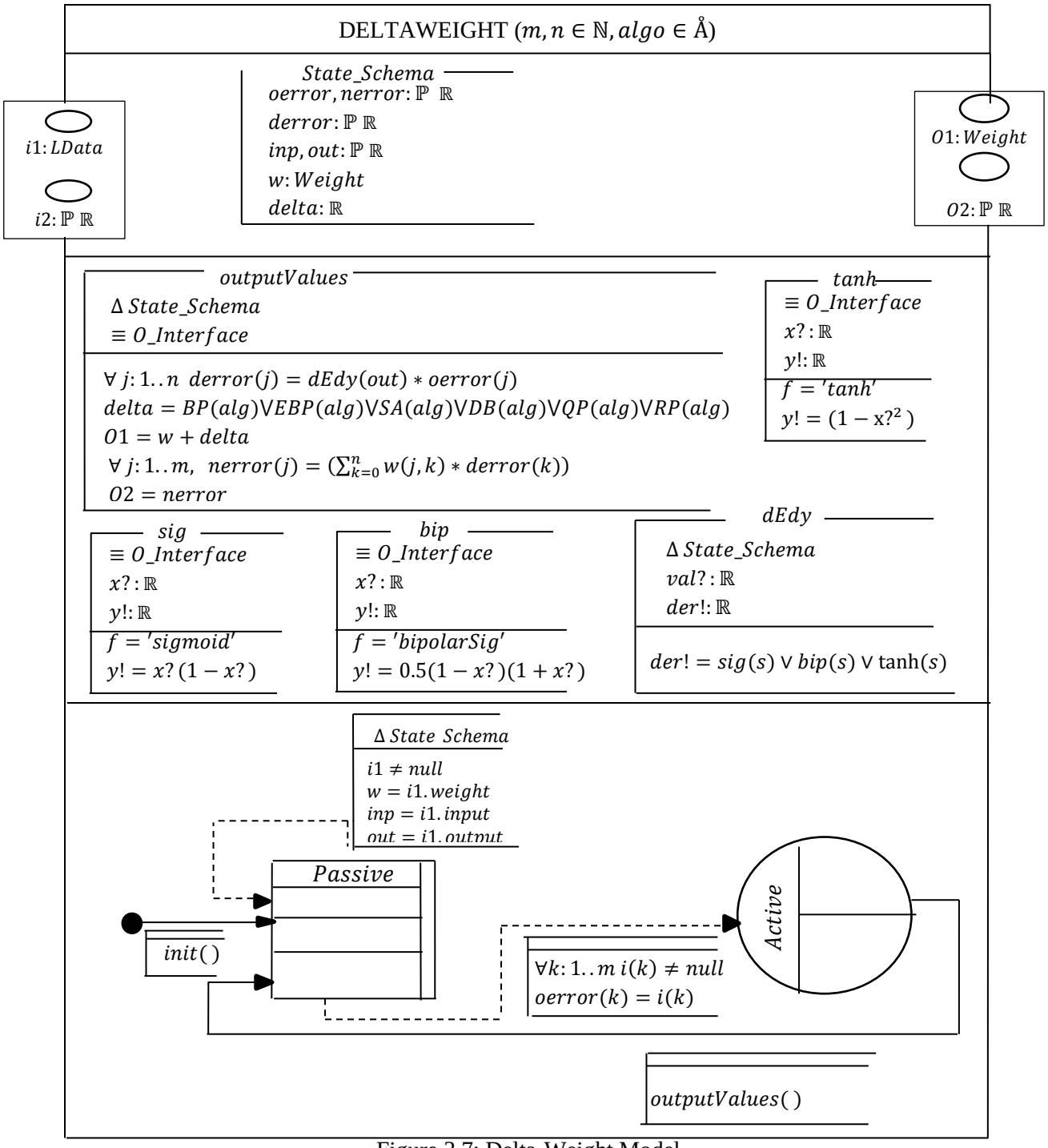

Figure 3.7: Delta-Weight Model

3.2.7. DEVS-Based ANN Coupled Model

$\boldsymbol{n}$ is the number of layers (hidden and output) and $\boldsymbol{m}$ is the number of neurons in each model

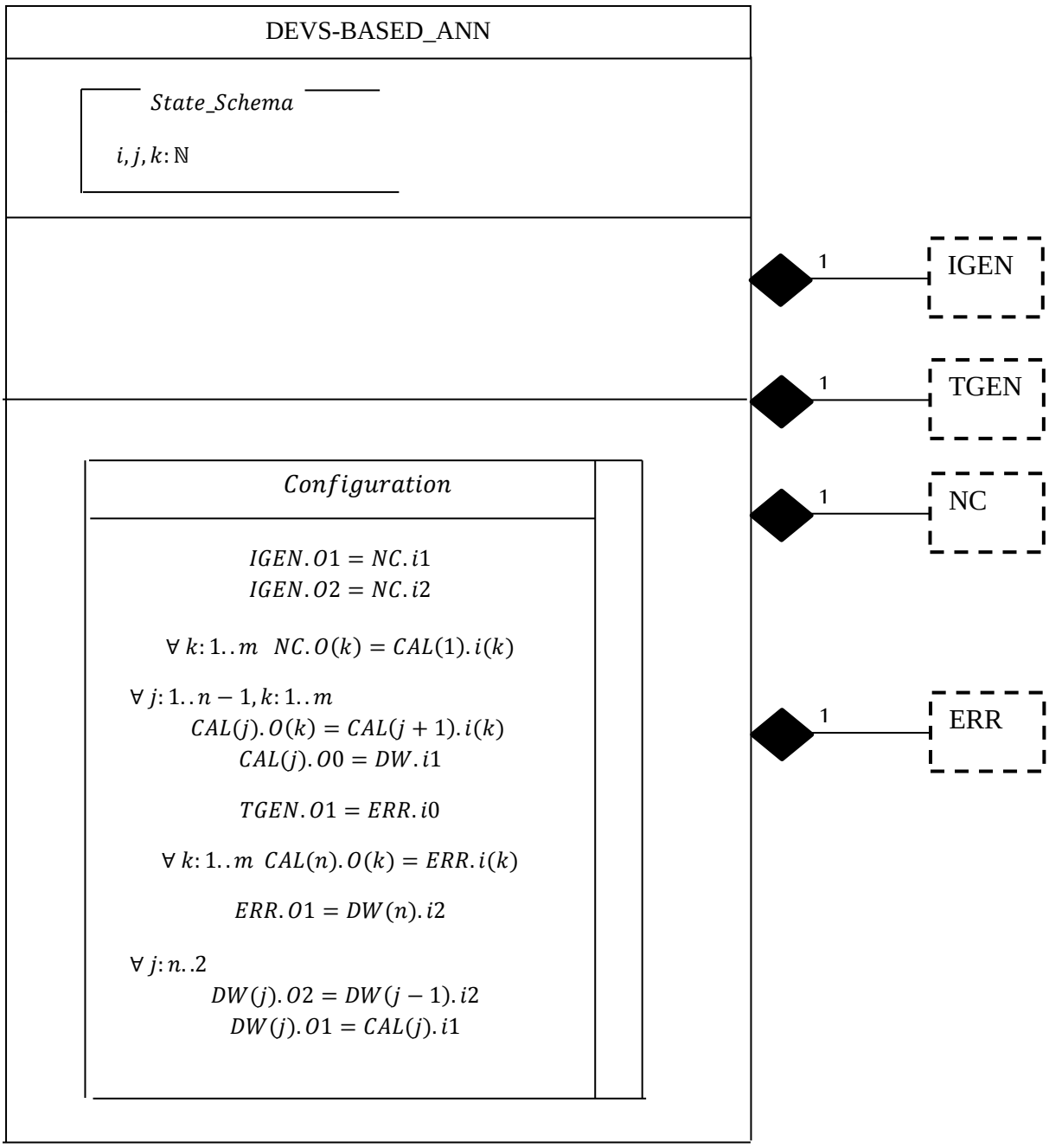

Figure 3.8: DEVS-Based ANN HiLLS Coupled Model

3.3. SimStudio Implementation

The DEVS model described in section 3.1 and 3.2 is implemented with the SimStudio C-DEVS framework in JAVA programming language. It is a coupled model with 6 atomic models namely: *InputGenerator, TargetGenerator, InputLayer, CalculationLayer, ErrorGenerator and DeltaWeight*. DEVS requires fixed atomic models and ports. However, two properties are dynamic in the ANN: (1) the number of hidden layers and (2) the number of hidden neurons. It is important to know the number of hidden layers, the number of inputs, hidden neurons and outputs before creating the coupled model for a DEVS-Based ANN. Also, the coupled model constructor provides the training input patterns, training output patterns, activation function and learning algorithms to InputGenerator and TargetGenerator, CalculationLayer and DeltaWeight respectively before simulation starts.

C-DEVS requires the order of priority to handle simultaneous internal event in the select function. The order of priority for the sub-component atomic model is given below;

$$InputLayer > CalculationLayer > ErrorGenerator > DeltaWeight > InputGenerator > TargetGenerator$$

Consider a DEVS-Based ANN coupled model with 5 inputs, 2 outputs and 2 hidden layers. The first hidden layer has 4 output neurons and second hidden layer has 6 output neurons. The coupled model produced is shown below (See figure 3.9)

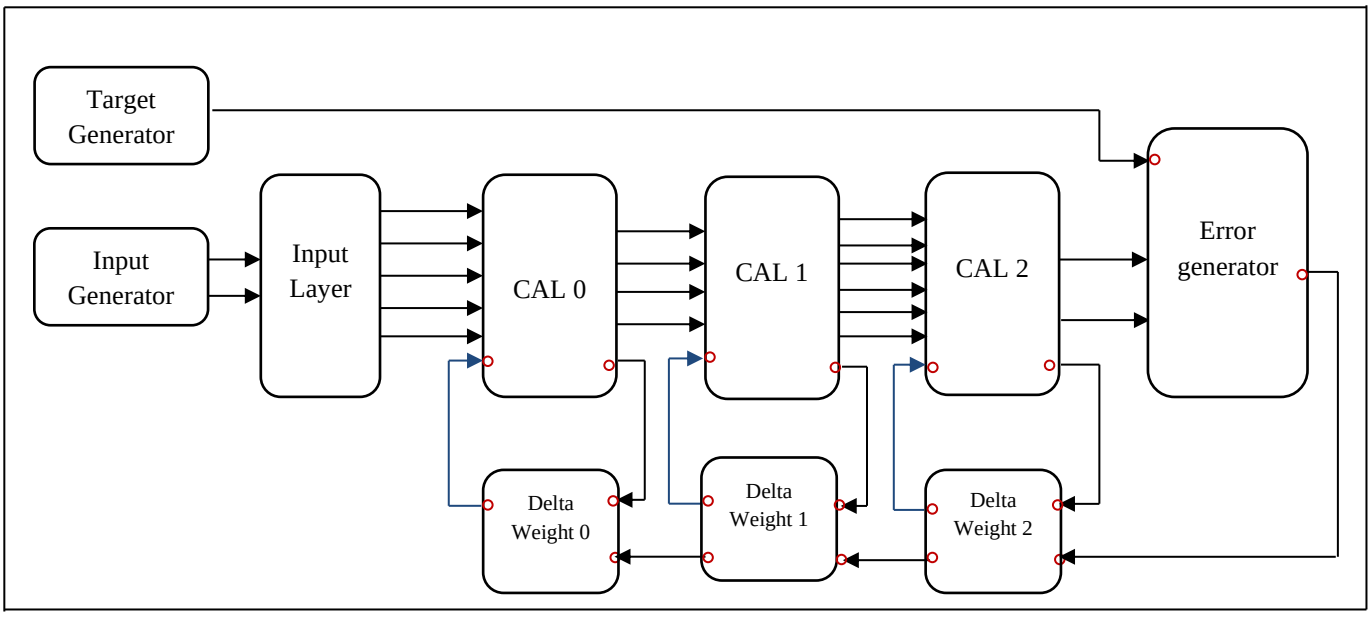

Figure 3.9: DEVS-Based ANN Coupled Model

The learning algorithms implemented are:

- Standard Back propagation algorithm
- Back propagation with momentum
- Silva and Almeida Algorithm
- Delta-Bar-Delta Algorithm
- Quickprop
- Resilient Propagation.

The activation functions implemented are;

- Binary Sigmoid function [0,1]
- Bipolar Sigmoid function [-1,1]
- Hyperbolic Tangent function [-1,1]
- Gaussian function [-1, 1] with peak at 0.

For each run, we need to specify the *learning rate, momentum, activation function, learning algorithm, training input pattern list, training target pattern list, number of hidden neurons for each hidden layer and the minimum error*.

The error generator computes the mean square error over all patterns (equation 2.18) when all pattern calculated outputs have been received. It compares the mean square error (MSE) with the minimum error (minError). If $mse < minError$, no error list is sent to the delta-model and no weight update; this signifies that the neural networks understand the training pattern to a high percentage.

3.4. User Interface

A Graphical user friendly platform has been developed to facilitate the modeling and simulation approach of the DEVS-Based ANN system. The platform has the ability to build a neural network model with several activation functions and learning algorithms for a multi-layer neural network. The GUI has 3 sections as shown in figure 3.10.

1) Parameter configuration and Execution: This section is sub-divided into three sections. The first section is for *model parameters* such as number of hidden layers, number of hidden neurons, number of inputs, and number of outputs neuron, activation function and learning algorithm. You can also upload training input and output pattern from this section. The second section is for *execution parameters*: the learning rate, momentum, minimum error and simulation time are specified here before training the ANN. The last section is for *comparison*; here, you can compare many algorithms with one activation function or many activation functions with one learning algorithm.
2) DEVS model design view: generates the model structure similar to the diagram in figure 3.9 for clear picture of DEVS-Based ANN design.
3) Graph Result: After training, the mean square error is plotted against the number of iterations (epochs). This gives us a clear picture of the rate of error reduction in a specified learning algorithm and activation function. This is the section that shows the comparison result of the learning algorithms and activation functions. The Graph makes use of JFreeChart [51] library.

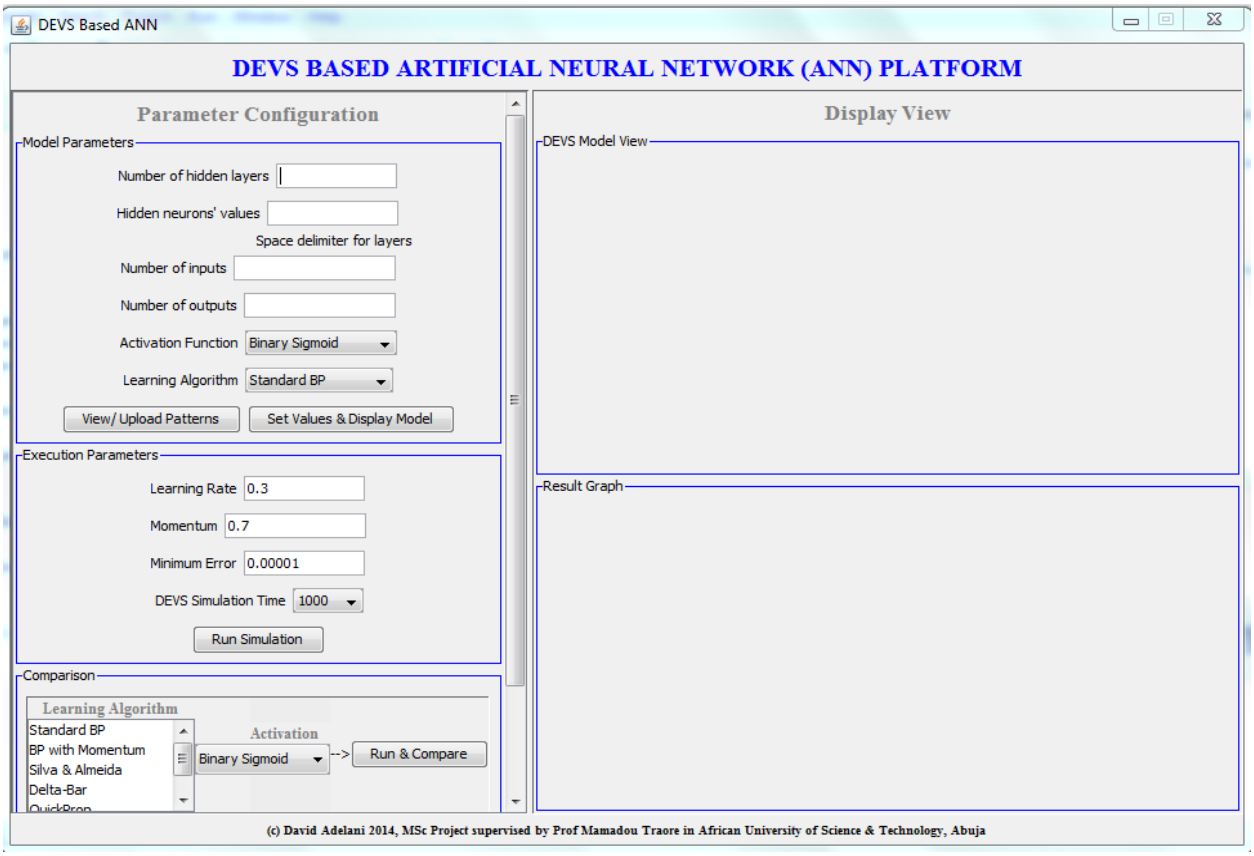

Figure 3.9: DEVS BASED ANN PLATFORM

We will illustrate this platform with a non-linearly separable function (XOR).

X_1	X_2	Y
0	0	0
0	1	1
1	0	1
1	1	0

Table 3-1: XOR function

Figure 3.11 shows the result of XOR function with 1 hidden layer (3 neurons), hyperbolic tangent activation function and Back propagation with momentum learning algorithm

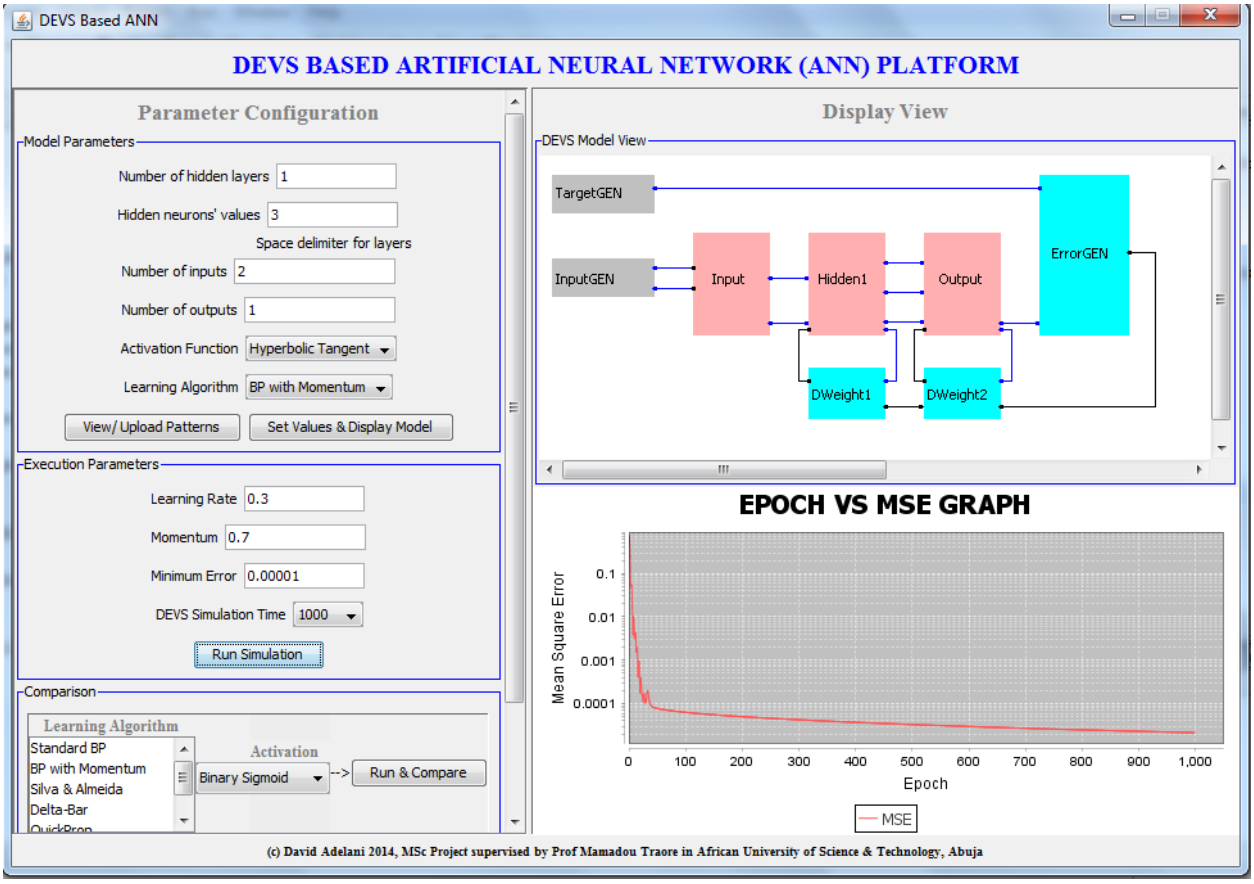

Figure 3.10: Simulation Result for Back propagation with Momentum

Figure 3.12 shows an XOR function result with multiple learning algorithms (Standard Back propagation, momentum back propagation, Silva and Almeida and Delta-Bar-Delta) and Binary sigmoid. The learning rate of 0.3 and momentum of 0.7 was used. Also, a minimum error of 0.00001 was used to compare the algorithms.

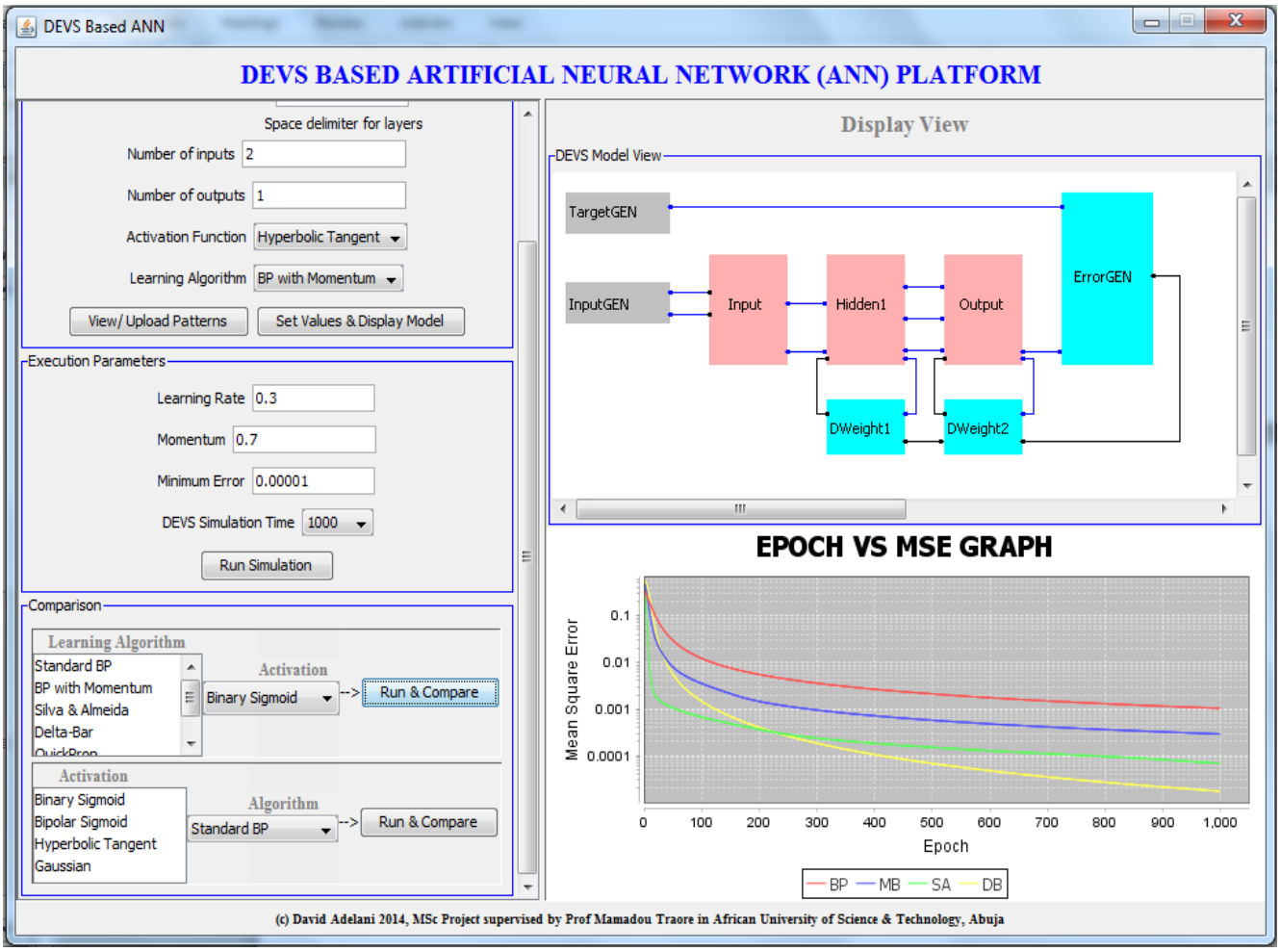

Figure 3.11: DEVS Based ANN for Multiple Algorithms

Figure 3.12 shows an XOR function result with multiple activation functions (Binary sigmoid, Bipolar sigmoid, Hyperbolic tangent and Gaussian) and Resilient propagation algorithm. The learning rate of 0.3 and momentum of 0.7 was used. Also, a minimum error of 0.00001 was used to compare the activation functions.

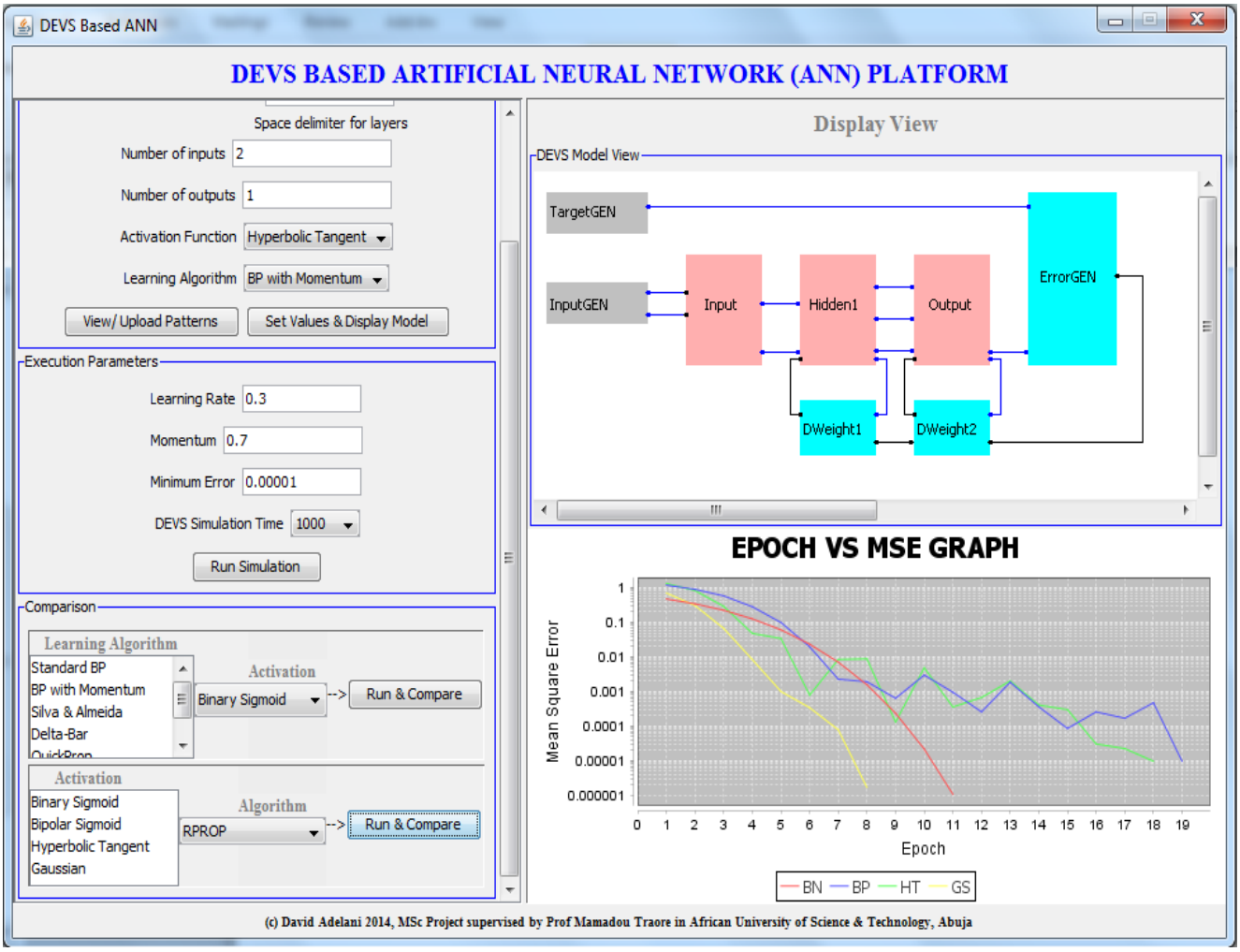

Figure 3.12: DEVS Based ANN for Multiple Activation Functions

CHAPTER 4

4.0 APPLICATION OF DEVS-BASED ANN

4.1. Presentation of the Case Studies

In this section, we will solve a pattern recognition problem using neural networks. Neural networks are very good for pattern recognition once they are trained. We will build a neural network to classify wines from three wineries. The data we are using are the results of a chemical analysis of wines grown in the same region in Italy but derived from three different cultivars. The analysis determined the quantities of 13 constituents found in each of the three types of wines. The attributes are:

1. Alcohol
2. Malic Acid
3. Ash
4. Ash Alcalinity
5. Magnesium
6. Total Phenols
7. Flavanoids
8. Nonflavanoid Phenols
9. Proanthocyanins
10. Color Intensity
11. Hue
12. OD280/OD315 of dedulted wines
13. Proline

The thirteen neighborhood attributes will act as inputs to a neural network, and the respective target for each will be a 3-element row vector with a 1 in the position of the associated winery (1 0 0, 0 1 0 and 0 0 1 for #1, #2 and #3 respectively). The data set consists of 178 instances (or input pattern) and each one is described with 13 characteristics given above. The dataset can be found in the UCI Machine Learning Repository [52].

However, we will be using 30 input patterns for the training of the neural networks. Each winery has 10 input patterns.

4.2. Data Extraction

4.2.1. Raw Data Presentation

The raw data gotten through analysis cannot be used directly for training a neural network. It needs to be normalized to speed up the training time of the neural network. Another important reason for data normalization is that the activation functions used produces output that range between [0, 1] or [-1, 1]. The raw data is shown in the table 3.2 below

1	2	3	4	5	6	7	8	9	10	11	12	13	Wine Type
14.23	1.71	2.43	15.60	127.00	2.80	3.06	0.28	2.29	5.64	1.04	3.92	1065.00	#1
13.20	1.78	2.14	11.20	100.00	2.65	2.76	0.26	1.28	4.38	1.05	3.40	1050.00	#1
13.16	2.36	2.67	18.60	101.00	2.80	3.24	0.30	2.81	5.68	1.03	3.17	1185.00	#1
14.37	1.95	2.50	16.80	113.00	3.85	3.49	0.24	2.18	7.80	0.86	3.45	1480.00	#1
13.24	2.59	2.87	21.00	118.00	2.80	2.69	0.39	1.82	4.32	1.04	2.93	735.00	#1
14.20	1.76	2.45	15.20	112.00	3.27	3.39	0.34	1.97	6.75	1.05	2.85	1450.00	#1
14.39	1.87	2.45	14.60	96.00	2.50	2.52	0.30	1.98	5.25	1.02	3.58	1290.00	#1
14.06	2.15	2.61	17.60	121.00	2.60	2.51	0.31	1.25	5.05	1.06	3.58	1295.00	#1
14.83	1.64	2.17	14.00	97.00	2.80	2.98	0.29	1.98	5.20	1.08	2.85	1045.00	#1
13.86	1.35	2.27	16.00	98.00	2.98	3.15	0.22	1.85	7.22	1.01	3.55	1045.00	#1
12.37	0.94	1.36	10.60	88.00	1.98	0.57	0.28	0.42	1.95	1.05	1.82	520.00	#2
12.33	1.10	2.28	16.00	101.00	2.05	1.09	0.63	0.41	3.27	1.25	1.67	680.00	#2
12.64	1.36	2.02	16.80	100.00	2.02	1.41	0.53	0.62	5.75	0.98	1.59	450.00	#2
13.67	1.25	1.92	18.00	94.00	2.10	1.79	0.32	0.73	3.80	1.23	2.46	630.00	#2
12.37	1.13	2.16	19.00	87.00	3.50	3.10	0.19	1.87	4.45	1.22	2.87	420.00	#2
12.17	1.45	2.53	19.00	104.00	1.89	1.75	0.45	1.03	2.95	1.45	2.23	355.00	#2
12.37	1.21	2.56	18.10	98.00	2.42	2.65	0.37	2.08	4.60	1.19	2.30	678.00	#2
13.11	1.01	1.70	15.00	78.00	2.98	3.18	0.26	2.28	5.30	1.12	3.18	502.00	#2
12.37	1.17	1.92	19.60	78.00	2.11	2.00	0.27	1.04	4.68	1.12	3.48	510.00	#2
13.34	0.94	2.36	17.00	110.00	2.53	1.30	0.55	0.42	3.17	1.02	1.93	750.00	#2
12.86	1.35	2.32	18.00	122.00	1.51	1.25	0.21	0.94	4.10	0.76	1.29	630.00	#3
12.88	2.99	2.40	20.00	104.00	1.30	1.22	0.24	0.83	5.40	0.74	1.42	530.00	#3
12.81	2.31	2.40	24.00	98.00	1.15	1.09	0.27	0.83	5.70	0.66	1.36	560.00	#3
12.70	3.55	2.36	21.50	106.00	1.70	1.20	0.17	0.84	5.00	0.78	1.29	600.00	#3
12.51	1.24	2.25	17.50	85.00	2.00	0.58	0.60	1.25	5.45	0.75	1.51	650.00	#3
12.60	2.46	2.20	18.50	94.00	1.62	0.66	0.63	0.94	7.10	0.73	1.58	695.00	#3
12.25	4.72	2.54	21.00	89.00	1.38	0.47	0.53	0.80	3.85	0.75	1.27	720.00	#3
12.53	5.51	2.64	25.00	96.00	1.79	0.60	0.63	1.10	5.00	0.82	1.69	515.00	#3
13.49	3.59	2.19	19.50	88.00	1.62	0.48	0.58	0.88	5.70	0.81	1.82	580.00	#3
12.84	2.96	2.61	24.00	101.00	2.32	0.60	0.53	0.81	4.92	0.89	2.15	590.00	#3

Table 4-1: Raw Wine Sample Data

4.2.2. Data Normalization

One of the common tools used to obtain better results in ANN training is data normalization. Ideally a system designer wants the same range of values for each input feature in order to minimize bias within the neural network for one feature over another [53] Data normalization can speed up the training time and performance when the data for each feature are within the same scale. Therefore, data normalization can be useful for modeling applications whose inputs are generally on different scales. Two commonly used normalization techniques are discussed below:

4.2.2.1. Statistical or Z-Score Normalization

This normalization technique uses the mean (μ) and standard deviation (σ) for each attribute (or feature) across a set of training data to normalize each input attribute vector. The mean and standard deviation are computed for each input attribute, and then the transformation given in equation (3.1) is made to each input attribute vector as it is presented. This produces data where each attribute has a zero mean and a unit variance. The statistical norm for an input is given by;

$$x_i' = \left[\frac{(x_i - \mu)}{\sigma}\right] \qquad (3.1)$$

4.2.2.2. Min-Max Normalization

Min-Max normalization is used when a neural network designer wants to constrain the range of each input feature or output of a neural network. This is done by rescaling the features from one range of values to a new range of values. The most common range is [0, 1] or [-1, 1]. The data rescaling is more required if different variables have typical values which differ significantly. The linear interpolation formula for an input is given in equation 3.2

$$x_i' = \left(max_{target} - min_{target}\right) \times \left[\frac{(x_i - x_{min})}{(x_{max} - x_{min})}\right] + min_{target} \qquad (3.2)$$

$$where\ (x_{max} - x_{min}) \neq 0.$$

As seen in equation 3.2, the minimum and maximum values (x_{min}, x_{max}) for each feature in the data are calculated and the data are linearly transformed to lie within the range of values ($min_{target}, max_{target}$).

We will be applying Min-Max normalization approach to the raw data in table 3.2 using target range (0, 1). Since $min_{target} = 0$ and $max_{target} = 1$, equation 3.2 will be transformed to equation 3.3 for the raw data. The data was generated using a MATLAB code.

$$x_i' = \frac{x_i - x_{min}}{x_{max} - x_{min}} \qquad (3.3)$$

(T1, T2, T3) represents the output. (1,0,0), (0,1,0), and (0,0,1) are for wine types #1, #2 and #3.

1	2	3	4	5	6	7	8	9	10	11	12	13	T1	T2	T3
0.774	0.168	0.709	0.347	1.000	0.611	0.858	0.239	0.783	0.631	0.481	1.000	0.631	1	0	0
0.387	0.184	0.517	0.042	0.449	0.556	0.758	0.196	0.363	0.415	0.494	0.804	0.618	1	0	0
0.372	0.311	0.868	0.556	0.469	0.611	0.917	0.283	1.000	0.638	0.468	0.717	0.738	1	0	0
0.827	0.221	0.755	0.431	0.714	1.000	1.000	0.152	0.738	1.000	0.253	0.823	1.000	1	0	0
0.402	0.361	1.000	0.722	0.816	0.611	0.735	0.478	0.588	0.405	0.481	0.626	0.338	1	0	0
0.763	0.179	0.722	0.319	0.694	0.785	0.967	0.370	0.650	0.821	0.494	0.596	0.973	1	0	0
0.835	0.204	0.722	0.278	0.367	0.500	0.679	0.283	0.654	0.564	0.456	0.872	0.831	1	0	0
0.711	0.265	0.828	0.486	0.878	0.537	0.675	0.304	0.350	0.530	0.506	0.872	0.836	1	0	0
1.000	0.153	0.536	0.236	0.388	0.611	0.831	0.261	0.654	0.556	0.532	0.596	0.613	1	0	0
0.635	0.090	0.603	0.375	0.408	0.678	0.887	0.109	0.600	0.901	0.443	0.860	0.613	0	1	0
0.075	0.000	0.000	0.000	0.204	0.307	0.033	0.239	0.004	0.000	0.494	0.208	0.147	0	1	0
0.060	0.035	0.609	0.375	0.469	0.333	0.205	1.000	0.000	0.226	0.747	0.151	0.289	0	1	0
0.177	0.092	0.437	0.431	0.449	0.322	0.311	0.783	0.088	0.650	0.405	0.121	0.084	0	1	0
0.564	0.068	0.371	0.514	0.327	0.352	0.437	0.326	0.133	0.316	0.722	0.449	0.244	0	1	0
0.075	0.042	0.530	0.583	0.184	0.870	0.871	0.043	0.608	0.427	0.709	0.604	0.058	0	1	0
0.000	0.112	0.775	0.583	0.531	0.274	0.424	0.609	0.258	0.171	1.000	0.362	0.000	0	1	0
0.075	0.059	0.795	0.521	0.408	0.470	0.722	0.435	0.696	0.453	0.671	0.389	0.287	0	1	0
0.353	0.015	0.225	0.306	0.000	0.678	0.897	0.196	0.779	0.573	0.582	0.721	0.131	0	1	0
0.075	0.050	0.371	0.625	0.000	0.356	0.507	0.217	0.263	0.467	0.582	0.834	0.138	0	1	0
0.440	0.000	0.662	0.444	0.653	0.511	0.275	0.826	0.004	0.209	0.456	0.249	0.351	0	0	1
0.259	0.090	0.636	0.514	0.898	0.133	0.258	0.087	0.221	0.368	0.127	0.008	0.244	0	0	1
0.267	0.449	0.689	0.653	0.531	0.056	0.248	0.152	0.175	0.590	0.101	0.057	0.156	0	0	1
0.241	0.300	0.689	0.931	0.408	0.000	0.205	0.217	0.175	0.641	0.000	0.034	0.182	0	0	1
0.199	0.571	0.662	0.757	0.571	0.204	0.242	0.000	0.179	0.521	0.152	0.008	0.218	0	0	1
0.128	0.066	0.589	0.479	0.143	0.315	0.036	0.935	0.350	0.598	0.114	0.091	0.262	0	0	1
0.162	0.333	0.556	0.549	0.327	0.174	0.063	1.000	0.221	0.880	0.089	0.117	0.302	0	0	1
0.030	0.827	0.781	0.722	0.224	0.085	0.000	0.783	0.163	0.325	0.114	0.000	0.324	0	0	1
0.135	1.000	0.848	1.000	0.367	0.237	0.043	1.000	0.288	0.521	0.203	0.158	0.142	0	0	1
0.496	0.580	0.550	0.618	0.204	0.174	0.003	0.891	0.196	0.641	0.190	0.208	0.200	0	0	1
0.252	0.442	0.828	0.931	0.469	0.433	0.043	0.783	0.167	0.508	0.291	0.332	0.209	0	0	1

Table 4-2: Normalized Wine Data

4.3. Results

In this section, we will show the result of our simulation with different algorithms and activation functions.

Figure 4.1 shows the training result for QuickProp algorithm and Binary Sigmoid activation function.

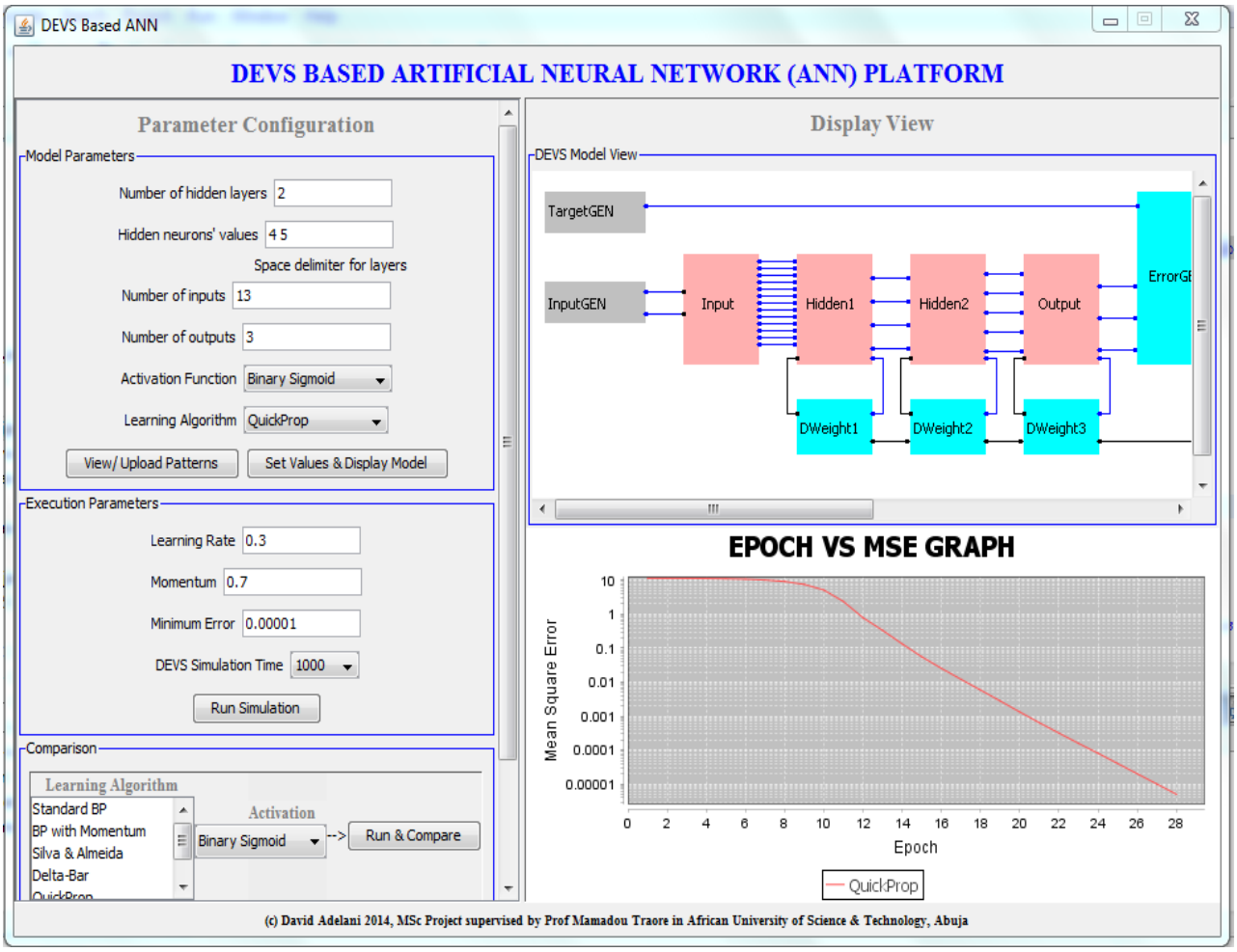

Figure 4.1: Simulation Result for QuickProp Algorithm with Binary Sigmoid

Figure 4.2 shows the training result for multiple algorithm (Silva and Almeida, Delta-Bar-Delta, Quickprop, RPROP) and Bipolar Sigmoid activation function. The learning rate of 0.3 and momentum of 0.7 was used. Also, a minimum error of 0.00001 was used to compare the algorithms

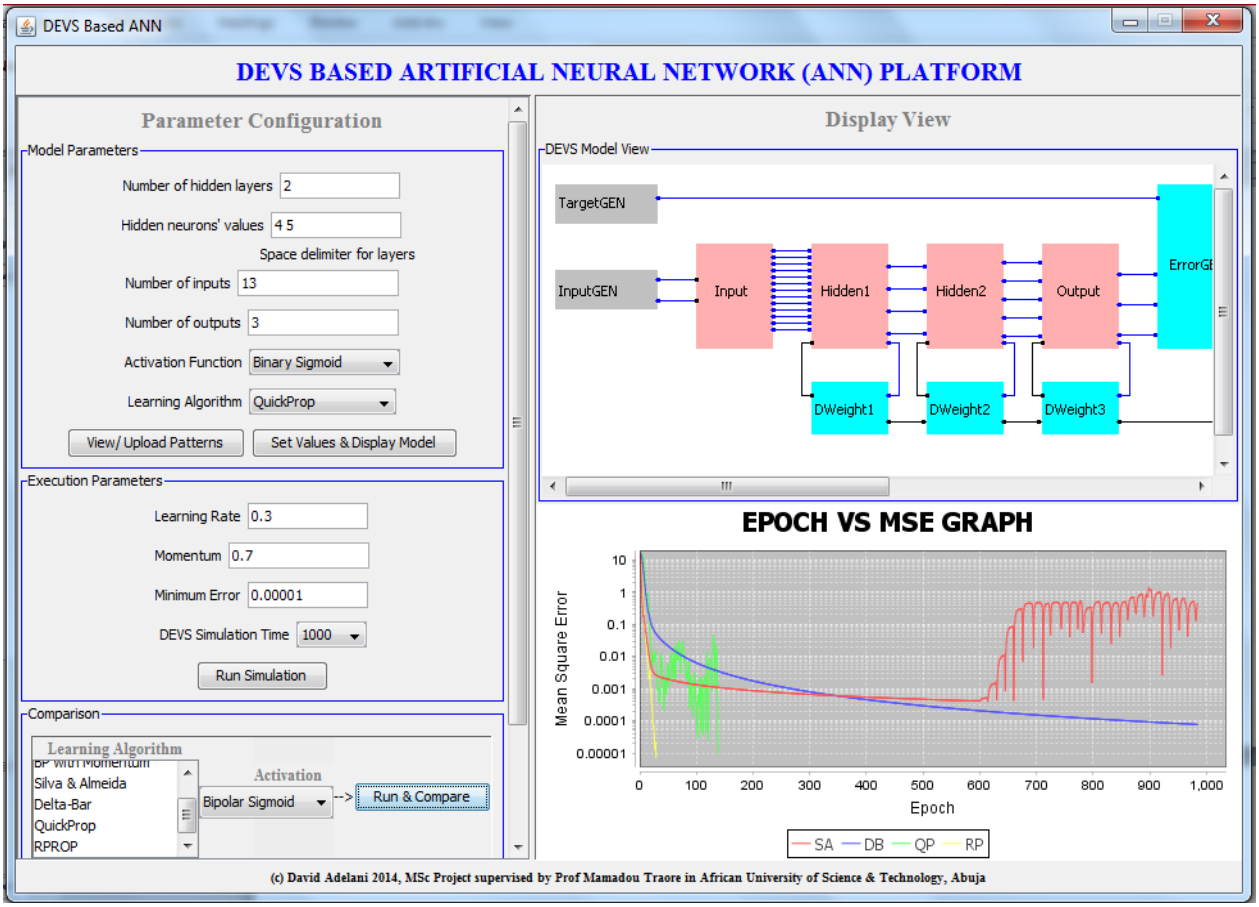

Figure 4.2: Simulation Result for Multiple Algorithms

Figure 4.3 shows the training result with multiple activation functions (Binary sigmoid, Bipolar sigmoid, Hyperbolic tangent and Gaussian) and Delta-Bar-Delta algorithm. The learning rate of 0.3 and momentum of 0.7 was used. Also, a minimum error of 0.00001 was used to compare the activation functions.

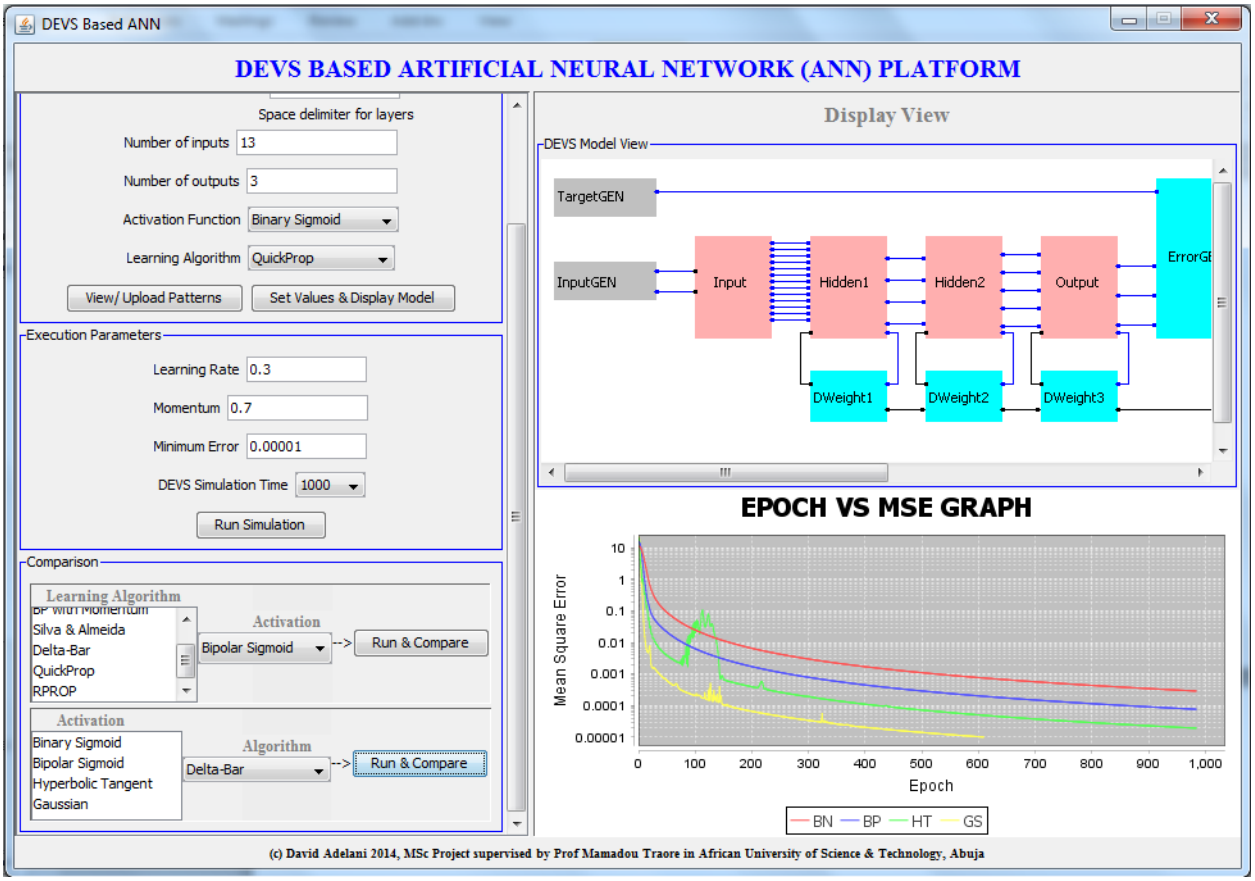

Figure 4.3: Simulation Result for Many Activation function

CHAPTER 5

5.0 CONCLUSION

5.1. Summary of Work Done

DEVS and ANN are two concepts that are able to simulate complex systems. DEVS is a formalism used to describe discrete event system in a hierarchical and modular manner. It allows the behavior modeling of a non- linear system. On the other hand, ANNs has wide applicability in real applications because of its capability to solve complex problems. The modular property of DEVS system coupled with the fact that ANNs operates in a discrete-event way (the network is always waiting to an input event to generate an output) has made it possible to build a DEVS-Based ANN system.

DEVS-Based ANN is composed of two groups of models: feed-forward calculations (non-calculation and calculation atomic models) and learning models (error-generator and delta-weight). This separation makes it easy to do only prediction through the feed-forward calculations (with optimized weights) immediately after the training process.

The DEVS-Based ANN was specified with HiLLS visual modeling language for clear understanding; this is because DEVS does not have concrete syntax. This approach will help users and algorithm developers to test and compare different algorithm implementations and parameter configurations of ANN

5.2. Pros and Cons

Toma et al [1] presented a DEVS-Based Approach to facilitate the network configuration of ANN using back propagation with Momentum. The approach provided an efficient way of doing prediction because of the clear separation of feed-forward calculation models and learning models. We extended the DEVS-Based ANN to test several learning algorithms (Standard Back propagation, Back propagation with Momentum, Silva & Almeida, Delta-Bar-Delta, Quickprop and RPROP) and architectures (with multiple hidden layers and activation functions – Binary Sigmoid, Bipolar Sigmoid, Hyperbolic Tangent and Gaussian). Our contributions are;

- The extended DEV-Based ANN approach will help users and algorithm developers to test and compare different algorithm implementations and parameter configurations of ANN
- The approach shows the power of DEVS formalism in modeling adaptive systems such as ANNs.
- The HiLLS specification used in describing DEVS-Based ANN models enables us to express all the arithmetic and logical expressions because of the rich mathematical language of Z-Schema. HiLLS can be used to provide concrete syntax when describing complex systems.

However, the simulation speed is largely dependent on the number of patterns used for training and the number of hidden layers. As the number of patterns used for training increases, the simulation is slower. This is because of the large number of messages being passed from one model to another during simulation as discussed in section 1.3.5.

5.3. Future Works

The DEVS-Based ANN platform built is using the classic DEVS and gradient descent based algorithms. Further work might include using different DEVS extension like PDEVS. Also, other supervised learning algorithm approach that is not gradient-descent based could be considered. It will also be interesting to build a DEVS-Based model that can handle unsupervised learning.

REFERENCES

[1] S. Toma, L. Capocchi, D. Federici, and others, "A New DEVS-Based Generic Artificial Neural Network Modeling Approach," *Proceeding EMSS 2011*, 2011.

[2] B. P. Zeigler, H. Praehofer, and T. G. Kim, *Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems*. Academic press, 2000.

[3] B. P. Zeigler, *Theory of modeling and simulation*. New York: Wiley-Interscience, 1976.

[4] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533 – 536, 1986.

[5] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Netw.*, vol. 2, no. 5, pp. 359–366, 1989.

[6] H. O. Aliyu, O. Maïga, H. I. Abdulwahab, and M. K. Traoré, "High Level Language for System Specification."

[7] M. K. Traoré, "A graphical notation for DEVS," in *Proceedings of the 2009 Spring Simulation Multiconference*, 2009, p. 162.

[8] S. H. Jung and T. G. Kim, "Abstraction of continuous system to discrete event system using neural network," in *AeroSense'97*, 1997, pp. 42–51.

[9] A. I. Concepcion and B. F. Zeigler, "DEVS formalism: A framework for hierarchical model development," *IEEE Trans. Softw. Eng.*, vol. 14, no. 2, pp. 228–241, 1988.

[10] S. J. Choi and T. G. Kim, "Identification of discrete event systems using the compound recurrent neural network: Extracting DEVS from trained network," *Simulation*, vol. 78, no. 2, pp. 90–104, 2002.

[11] J. Filippi, P. Bisgambiglia, and M. Delhom, "Neuro-devs, an hybrid methodology to describe complex systems," in *Proceedings of the SCS ESS 2002 conference on simulation in industry*, 2002, vol. 1, p. 647.

[12] O. Gérard, Je.-N. Patillon, and F. D'Alché-Buc, "Discharge prediction of rechargeable batteries with neural networks," *Integr. Comput.-Aided Eng.*, vol. 6, no. 1, pp. 41–52, 1999.

[13] S. Vahie, "Dynamic neuronal ensembles: neurobiologically inspired discrete event neural networks," in *Discrete Event Modeling and Simulation Technologies*, Springer, 2001, pp. 229–262.

[14] S. Toma, *Detection and identification methodology for multiple faults in complex systems using discrete events and neural networks: applied to the wind turbines diagnosis*. Université Pascale Paoli, Corsica, 2014.

[15] M. K. Traoré, "SimStudio: a next generation modeling and simulation framework," in *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, 2008, p. 67.

[16] A. K. Gupta and Y. P. Singh, "Analysis of Back Propagation of Neural Network Method in the String Recognition," *Int. J. Comput. Theory Eng.*, vol. 3, no. 5, 2011.

[17] J. Heaton, *Introduction to neural networks with Java*. Heaton Research, Inc., 2008.

[18] L. V. Fausett, *Fundamentals of neural networks*. Prentice-Hall, 1994.

[19] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *Bull. Math. Biophys.*, vol. 5, no. 4, pp. 115–133, 1943.

[20] J. A. Anderson and E. Rosenfeld, *Neurocomputing*, vol. 2. MIT press, 1993.

[21] D. O. Hebb, *The organization of behavior: Aneuropsychological theory*. Wiley, 1949.

[22] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain.," *Psychol. Rev.*, vol. 65, no. 6, p. 386, 1958.

[23] M. Minsky and S. Papert, "Perceptron: an introduction to computational geometry," *MIT Press Camb. Expand. Ed.*, vol. 19, p. 88, 1969.

[24] P. Werbos, "Beyond regression: New tools for prediction and analysis in the behavioral sciences," 1974.

[25] J. E. Dayhoff and J. M. DeLeo, "Artificial neural networks," *Cancer*, vol. 91, no. S8, pp. 1615–1635, 2001.

[26] D. B. Parker, "Learning logic," 1985.

[27] Y. LeCun and others, "Generalization and network design strategies," *Connect. Perspect. N.-Holl. Amst.*, pp. 143–55, 1989.

[28] S. Russell, "Artificial Intelligence: A Modern Approach Author: Stuart Russell, Peter Norvig, Publisher: Prentice Hall Pa," 2009.

[29] B. Yegnanarayana, *Artificial neural networks*. PHI Learning Pvt. Ltd., 2009.

[30] T. Kohonen, E. Oja, O. Simula, A. Visa, and J. Kangas, "Engineering applications of the self-organizing map," *Proc. IEEE*, vol. 84, no. 10, pp. 1358–1384, 1996.

[31] R. C. Chakraborty, "Fundamentals of Neural Networks," *Soft Comput.*, pp. 7–14, 2010.

[32] K.-L. Du and M. N. Swamy, *Neural networks in a softcomputing framework*. Springer, 2006.

[33] P. J. Werbos, "Consistency of HDP applied to a simple reinforcement learning problem," *Neural Netw.*, vol. 3, no. 2, pp. 179–189, 1990.

[34] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Cogn. Model.*, 1988.

[35] F. J. Pineda, "Generalization of back-propagation to recurrent neural networks," *Phys. Rev. Lett.*, vol. 59, no. 19, p. 2229, 1987.

[36] R. A. Jacobs, "Increased rates of convergence through learning rate adaptation," *Neural Netw.*, vol. 1, no. 4, pp. 295–307, 1988.

[37] F. M. Silva, "Speeding up backpropagation," *Adv. Neural Comput.*, pp. 151–160, 1990.

[38] S. E. Fahlman, "Faster-learning variations on back-propagation: An empirical study," 1988.

[39] W. Schiffmann, M. Joost, and R. Werner, *Optimization of the backpropagation algorithm for training multilayer perceptrons*. Univ., Inst. of Physics, 1993.

[40] M. Riedmiller and H. Braun, "A direct adaptive method for faster backpropagation learning: The RPROP algorithm," in *Neural Networks, 1993., IEEE International Conference on*, 1993, pp. 586–591.

[41] T. Kohonen, "Self-organized formation of topologically correct feature maps," *Biol. Cybern.*, vol. 43, no. 1, pp. 59–69, 1982.

[42] C. Seo and B. P. Zeigler, "Interoperability between DEVS simulators using service oriented architecture and DEVS namespace," in *Proceedings of the 2009 Spring Simulation Multiconference*, 2009, p. 157.

[43] A. Adegoke, H. Togo, and M. K. Traoré, "A unifying framework for specifying DEVS parallel and distributed simulation architectures," *Simulation*, p. 0037549713504983, 2013.

[44] F. J. Barros, B. P. Zeigler, and P. A. Fishwick, "Multimodels and dynamic structure models: an integration of DSDE/DEVS and OOPM," in *Proceedings of the 30th conference on Winter simulation*, 1998, pp. 413–420.

[45] A. C. H. Chow and B. P. Zeigler, "Parallel DEVS: A parallel, hierarchical, modular, modeling formalism," in *Proceedings of the 26th conference on Winter simulation*, 1994, pp. 716–722.

[46] G. A. Wainer, *Discrete-event modeling and simulation: a practitioner's approach*. CRC Press, 2008.

[47] R. Franceschini, P.-A. Bisgambiglia, L. Touraille, P. Bisgambiglia, D. Hill, R. Neykova, and N. Ng, "A survey of modelling and simulation software frameworks using Discrete Event System Specification," in *2014 Imperial College Computing Student Workshop*, 2014, p. 40.

[48] U. B. Ighoroje, O. Maïga, and M. K. Traoré, "The DEVS-driven modeling language: syntax and semantics definition by meta-modeling and graph transformation," in *Proceedings of the 2012 Symposium on Theory of Modeling and Simulation-DEVS Integrative M&S Symposium*, 2012, p. 49.

[49] OMG, "UML 2.3," May-2010. [Online]. Available: http://www.omg.org/spec/UML/2.3/. [Accessed: 24-Nov-2014].

[50] J. M. Spivey and J. R. Abrial, *The Z notation*. Prentice Hall Hemel Hempstead, 1992.

[51] "JFreeChart." [Online]. Available: http://www.jfree.org/jfreechart/. [Accessed: 28-Nov-2014].

[52] "UCI Machine Learning Repository: Wine Data Set." [Online]. Available: http://archive.ics.uci.edu/ml/datasets/Wine. [Accessed: 28-Nov-2014].

[53] K. L. Priddy and P. E. Keller, *Artificial neural networks: an introduction*, vol. 68. SPIE Press, 2005.