Neural Network Library in Modelica

Fabio Codecà   Francesco Casella
Politecnico di Milano, Italy
Piazza Leonardo da Vinci 32, 20133 Milano

Abstract

The aim of this work is to present a library, developed in Modelica, which provides the neural network mathematical model. This library is developed to be used to simulate a non-linear system, previously identified through a specific neural network training system. The `NeuralNetwork` library is developed in Modelica 2.2 and it offers all the required capabilities to create and use different kinds of neural networks.
Currently, only the *feed-forward*, the *elman*[6] and the *radial basis* neural network can be modeled and simulated, but it is possible to develop other kinds of neural network models using the basic elements provided by the library.
Keywords: neural network, library, simulate, model

1 Introduction

The work described in this paper is motivated by the lack of publicly available Modelica libraries for neural networks. A neural network is a mathematical model, which is normally used to identify a non-linear system. Its benefit is the capability to identify a system also when its model structure is not defined. For such a characteristic, sometimes it is used to model complex non-linear system.
There are different kinds of neural networks in literature and all of them are characterized by a specific architecture or some other specific features. This library takes into consideration only three types of neural networks:

- the *feed-forward* neural network,
- the *elman*[6] neural network, which is a recurrent neural network,
- the *radial basis* neural network,

but the basic elements of the library make possible the construction of any other different neural network.

There are, in literature, different kinds of neural networks, many different algorithms to train them and many different softwares to do this task. For this reason, the library purposefully lacks any function to train a neural network; the training process has to be made by an external program. The MatLab[8] Neural Network toolbox was chosen, during the development and the tests, because it is commonly used and it is extremely powerful; however any other training software can be used.
The library was already used to develop and simulate the neural network model of an electro-hydraulic semi-active damper.
The paper is organized as follows. Section 2 presents the neural network mathematical model: a briefly description about the characteristics of each kind of network, implemented in the library, is provided. Section 3 describes the chosen library architecture and the reasons which guide its implementation. Section 4 shows an example of library use: the entire work process will be explained, from the neural network identification, with an external training software, through the network parameters exchange (from the training software environment to the Modelica one), to the validation of the Modelica model. Last section (5) shows some possibilities of future work, and draws some conclusions.

2 Neural Network model

The neural network mathematical model was born in the Artificial Intelligence (AI) research sector, in particular in the 'structural' one: the main idea is to reproduce the intelligence and the capability to learn from examples, simulating the brain neuronal structure on an calculator.
The first result was achieved by McCulloch and Pitts in 1943[1], when the first neural model was born. In 1962 Rosenblatt[2] proposed a new neuron model, called *perceptron*, which could be trained through examples. A *perceptron* makes the weighted sum of the inputs and, if the sum is greater then a bias value, it

sets its output as '1'. The training is the process used to tune the value of the bias and of the parameters which weight the inputs.
Some studies[3] underline the *perceptron* training limits. Next studies[4], otherwise, show that different basic neuron models, complex neuron networks architecture as suitable learning algorithms, ensure to go beyond the theoretical *perceptron* limits.
Three kinds of neural networks, which are described in the following paragraphs, were taken into consideration in the library: they differ in neuron model and network architecture.

2.1 Feed-forward neural network

The *feed-forward* neural network is the most used neural network architecture: it is based on the series connection of neuron layers, each one composed by a set of neurons connected in parallel. Examine the $i-th$ layer: the layer inputs, $u_1, u_2, \dots, u_n$, are used by r neurons, which produce r output signals, $y_1, y_2, \dots, y_r$. These signals are the inputs of the next layer, the $i+1-th$ layer.

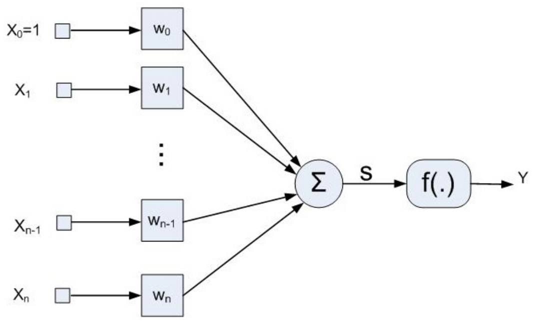

Figure 1: *Standard* neuron model

The neuron used in the *feed-forward* neural network is called *standard* neuron (figure 1). A *standard* neuron maps $\mathbb{R}^q$ into $\mathbb{R}$; it is characterized by n inputs, $u_1, u_2, \dots, u_n$, and one output, y. The first step, which is taken by the neuron, is to compute a weighted sum of the inputs: the result is called *activation signal*

$$s = w_0 u_0 + w_1 u_1 + w_2 u_2 + \dots + w_n u_n,$$

where $w_0, w_1, w_2, \dots, w_n$ are real parameters. w_0 is the neuron *bias* and $w_1, w_2, \dots, w_n$ are the neuron *weights*. The second step is to perform a non-linear elaboration of s, obtaining y. This is computed using a function $\sigma(\bullet)$ ($\mathbb{R} \rightarrow \mathbb{R}$), called *activation function*; it is usually a real function of real variable, monotonically increasing, with a lower and an upper asymptote:

$$s = \lim_{s \rightarrow -\infty} \sigma(\bullet) = \sigma_{inf},$$

$$s = \lim_{s \rightarrow -\infty} \sigma(\bullet) = \sigma_{inf}.$$

For this reasons, it is usually called *sigmoid*. Different functions can be used; the most used are:

- $\sigma(s) = tanh(s)$ (called in MatLab `tangsig`);
- $\sigma(s) = \frac{1}{1+\exp^{-n}}$ (called in MatLab `logsig`);

A linear function is also used as *activation function*: it is normally used for the neurons which compose the output layer ($\sigma(s) = s$ (called in MatLab `purelin`)).
The *feed-forward* architecture allows to build very complex neural networks: the only constraint is to connect the layer in series and the neurons of a layer in parallel, each of them with the same *activation function*. The first section of the network, which takes the inputs and passes them to the first layer without doing anything, is usually called *Input* layer. The last layer is called *Output* layer and the others are called *Hidden* layer[1].

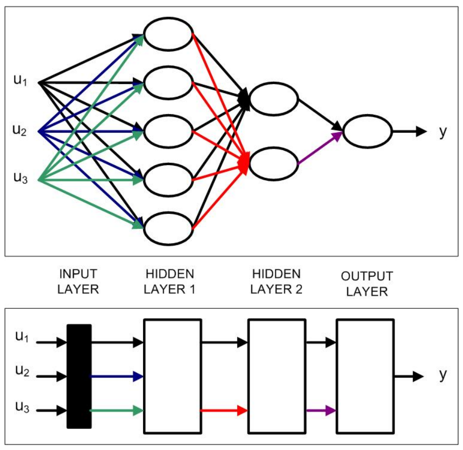

Figure 2: A *feed-forward* neural network structure

An important theoretical result, related to the *feed-forward* neural network, ensures to specify which is the non-linear function class that can be evaluated by a specific neural network. The result is applicable to different kinds of networks: in particular, it involves the *standard* neural network. This network is composed by only two layers: an *Hidden* layer, composed by m neurons[2], which processes n inputs, $u_1, u_2, \dots, u_n$, and an *Output* layer, composed by one neuron with a linear *activation function*.

[1]this is not an univocal nomenclature; for example, in MatLab, the first neuron layer is called *Input* layer and the others are simply called *layer*.

[2]all having the same *activation functions*

Theorem 1 (Universal Approximator[4]) *Take a standard neural network where $\sigma(\bullet)$ satisfies the following conditions:*

1. $\lim_{s\to\infty}\sigma(s) = 1$,
2. $\lim_{s\to-\infty}\sigma(s) = 0$,
3. $\sigma(\bullet)$ *is continuous.*

Taking a function $g(u) : \mathbb{R}^q \to \mathbb{R}$, continuous on a set I_u compact in $\mathbb{R}^q$, and an $\varepsilon > 0$, a standard neural network exists which achieves the transformation $y = f(u)$ so that

$$|g(u) - f(u)| < \varepsilon, \forall u \in I_u.$$

2.2 Recurrent neural network (Elman)

A particular type of neural network is the *recurrent* neural network. This network is a dynamical system, in which the output depends on the inputs and the internal state, which evolves with the network inputs. If the internal state is $Z(t)$, the network then agrees to the following relations:

$$\begin{cases} Z(t+1) &= F(Z(t), U(t)) \\ Y(t) &= G(Z(t), U(t)) \end{cases}$$

Recurrent networks are usually based on a feedback loop in the network architecture, but this is not the only way.
In the library, the *Elman*[6] neural network is considered: in this network, the feedback loop is between the output of the *Hidden* layer and the input of the layer itself. This allows the network to learn, recognize and create temporal and spatial models.
An *Elman* neural network is usually composed by two layer connected as shown in figure 3: there is an *Hidden* layer, which is the *recurrent* layer, composed by neurons with an hyperbolic tangent *activation function* ($\sigma(\bullet) = tanh(\bullet)$), and an output layer, characterized by a linear *activation function*.
As for the *feed-forward* neural network, the *universal approximator* theorem ensures that the *Elman* neural network is an universal approximator of a non-linear function. The only requirement is that the more the function to be estimated becomes complex, the more the number of the neurons, which compose the *Hidden* layer, increases.
The only difference between a *feed-forward* neural network and an *Elman* neural network is the recurrence: this allows the network to learn spatial and temporal models.

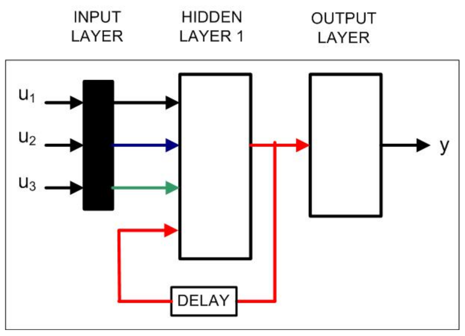

Figure 3: An *Elman* neural network

2.3 *Radial* basis neural network

The *Radial* basis neural network is used as an alternative to the *feed-forward* neural network. Like this one, it is based on the series connection of layers, each of them composed by a set of neurons connected in parallel. Two are the main differences:

- the number of layers is commonly fixed, with one *Hidden* layer and one *Output* layer;
- the basic neuron is not the *standard* neuron but it is called *radial* neuron.

A *radial* neuron maps $\mathbb{R}^q$ into $\mathbb{R}$; it is characterized by n inputs, $u_1, u_2, \ldots, u_n$, and one output, y. The first step took by the radial neuron is to compute an *activation signal*: it differs from the *standard* one because it is not a weighted sum of inputs but it is equal to:

$$s = dist(\{u_1, u_2, \ldots, u_n\}, \{\alpha_1, \alpha_2, \ldots, \alpha_n\})b,,$$

where $\alpha_1, \alpha_2, \ldots, \alpha_n$ are real parameters, regarding which distances of the inputs are calculated (they are called *centers* of the neuron), b is called neuron *amplitude* and the function $dist(\{x_1, x_2\}, \{a_1, a_2\})$ computes the euclidean distance between $\{x_1, x_2\}$ and $\{a_1, a_2\}$[3].
The following step is to perform a non-linear elaboration of s, obtaining y. This is made using the function $\sigma(\bullet) = \exp^{-(\bullet^2)}$ ($\mathbb{R} \to \mathbb{R}$) which is the *radial* neuron *activation function*; it is not a *sigmoid* function but a *bell-shaped* function (figure 4).
As previously remarked, the *radial basis* neural network architecture is commonly fixed: there is an *Hidden* layer, composed by *radial* neurons, and one *Output* layer, composed by a *standard* neuron with a linear activation function (*purelin*). Although the structure of this neural network is more limited, compared to the *feed-forward* one, this is not a limit for its approximator capability.

[3] $dist(\{x_1, x_2\}, \{a_1, a_2\}) = \sqrt{(x_1 - a_1)^2 + (x_2 - a_2)^2}$

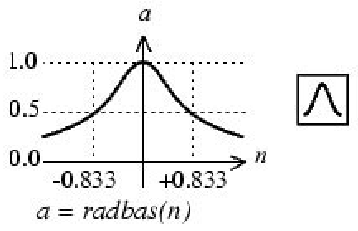

Figure 4: *Radial* neuron *activation function*

As for the *feed-forward* and the *Elman* neural networks, the universal approximator theorem ensures that this kind of neural network is an universal approximator of a non-linear function. The only requirement is that, the more the function to approximate becomes complex, the more the number of the neurons which compose the *Hidden* layer increases.

3 NeuralNetwork library

The reason of this library is the lack of an suitable Modelica library, able to simulate a neural network. The aim was to develop a library with the capabilities to create and to simulate such a mathematical model. There are already many different algorithms to train a neural network and many different softwares to do this task so no training algorithm was given. This requires that the training process must be performed by an external software. The MatLab[8] Neural Network toolbox was chosen, during the development and the tests, because it is used commonly and it is extremely powerful; however any other training software can be used. These elements affect some library architectural choices.

The first aim was to give to the users all the elements to create the previously presented neural networks: no constraints were put in for the user, who can create any kind of network architecture without limits. The user himself is directly responsible to use the basic blocks correctly and no checks are performed by the library blocks.

The basic element of the *NeuralNetwork* library was chosen to be a network layer. A layer in a neural network (`NeuralNetworkLayer`) is a set of neurons which are connected in parallel[5]. It is characterized by the following parameters:

- *numNeurons*: it is the number of neurons which compose the layer;
- *numInputs*: it is the number of inputs of the layer;
- *weightTable*: it is a matrix which collects the *weight* parameters (or the *centers* of neurons) used by every neuron of the layer to weight the inputs; its dimension is $[numNeurons \times numInputs]$;
- *biasTable*: it is a vector which collects the *biases* of neurons that compose the layer; its dimension is $[numNeurons \times 1]$;
- *NeuronActivationFunction*: it is the *activation function* used to compute the output by each neuron of the layer. The neurons, which compose a layer, can only have the same *activation function*.

Using a network layer as the basic element has the only limit that the *activation function* of each neuron in a layer must to be the same, but the neural network architectures previously presented don't need this property. Moreover this choice ensures to have an easier data exchange between the neural network training environment and the Modelica one.

This is particularly true when the MatLab Neural Network toolbox is used to train a neural network. As reported in the section 4, in the object used by MatLab to store a neural network, the *weights* (or the *centers* of neuron) and the *bias* of layer are collected in a matrix with the same property of the matrix used to initialize a `NeuralNetworkLayer`.

The library is organized in a tree-based fashion (Figure 5), and it is composed by five sub-packages:

- the package `BaseClasses`: it contains only one element, the `NeuralNetworkLayer`;
- the package `Networks`: it contains some neural networks based on the connection of many `NeuralNetworkLayer`;
- the package `Utilities`: it contains different functions and models used to define some library elements or used itself in the library;
- the package `Types`: it contains the constants used to specify the *activation functions* which characterize a `NeuralNetworkLayer`;
- the package `Examples`: it contains some examples which allow the user to explore the library capabilities.

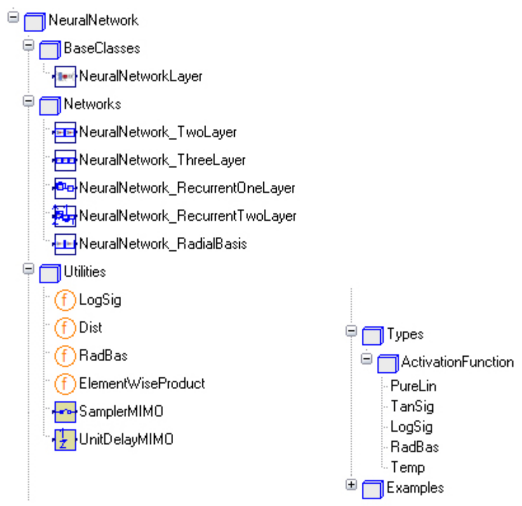

Figure 5: Library structure

3.1 BaseClasses - `NeuralNetworkLayer`

As previously described, there is only one element in the `BaseClasses` package, the `NeuralNetworkLayer`. This is a block with a MIMO interface, in which the number of inputs is specified through a parameter and the outputs number is the same to the neurons one.
The parameters of the `NeuralNetworkLayer` are:

- *numNeurons*: it is the number of neurons which compose the layer
- *numInputs*: it is the number of inputs to the layer
- *weightTable*: it is the table of the *weights*, if the layer is composed by *standard* neurons, or the table of the *centers* of the neuron, if the layer is composed by *radial* neurons
- *biasTable*: it is the *bias* matrix of the neurons which compose the layer
- *NeuronActivationFunction*: it is the *activation function* of the layer neurons

The *NeuronActivationFunction* characterizes the behavior of the neuron network layer. The parameter can be selected in the set, defined by `NeuralNetwork.Types.ActivationFunction`; the possible choices and behaviors are:

- `PureLin`: the block acts as a layer composed by *standard* linear neurons; the output is equal to the *activation signal s*, which is equal to

$$y = s = weightTable * u + biasTable[:,1]$$

- `TanSig`: the block acts as a layer composed by *standard* non linear neurons; the output is equal to the hyperbolic tangent of the *activation sign al*

$$y = Modelica.Math.tanh(s);$$

- `LogSig`: the block acts as a layer composed by *standard* non linear neurons; the output is equal to the value returned by the *LogSig* function:

$$y = NeuralNetwork.Utilities.LogSig(s);$$

- `RadBas`: the block acts as a layer composed by *radial* non linear neurons; the output is computed with the following steps:
  - the euclidean distance between the *centers* of layer neurons and the inputs is evaluated using the function `NeuralNetwork.Utilities.Dist()` with the following parameters: *weightTable*, *matrix(u)*
  - the element-wise product between the previous function output and the *bias* matrix is calculated using the `NeuralNetwork.Utilities.ElementWiseProduct` function: this value is the activation signal *s*
  - the output is then evaluated using the specific radial neuron *activation function* (`NeuralNetwork.Utilities.RadBas`);

3.2 Networks

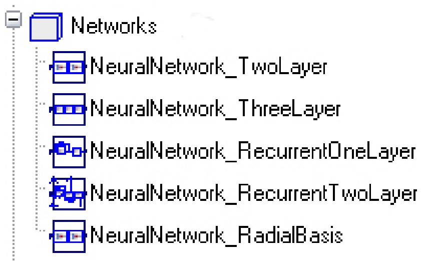
Networks
NeuralNetwork_TwoLayer
NeuralNetwork_ThreeLayer
NeuralNetwork_RecurrentOneLayer
NeuralNetwork_RecurrentTwoLayer
NeuralNetwork_RadialBasis

Figure 6: `Networks` package structure

This package (shown in figure 6) is composed by five blocks: each one represents a neural network. The *feed-forward* neural network and the *radial basis* neural network are easily composed using the `NeuralNetworkLayer` block.
The case of the *Elman* neural network (figure 7 shows the model in Dymola[7]), which in the library is called

NeuralNetwork_RecurrentOne(Two)Layer[4], is different. In figure 7 the NeuralNetwork_RecurrentOneLayer is shown: the delay block has been introduced to create the recurrence. The parameters of every layer and the parameter of the delay block, which is the *samplePeriod* of the recurrent layer, can be tuned. The *samplePeriod* has to be equal to the input signal sample rate, so that the network can work correctly.

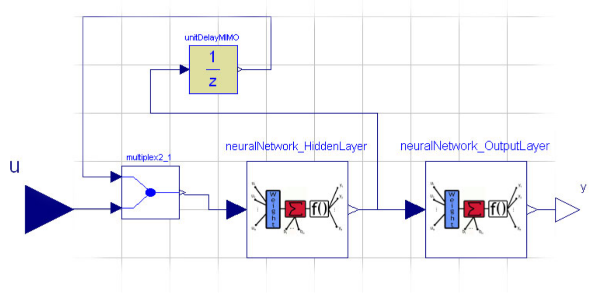

Figure 7: *Elman* neural network in Dymola

The NeuralNetwork.Utilities.UnitDelayMIMO was introduced to realize the layer feedback: it behaves as the Modelica.Blocks.Discrete.UnitDelay but it has a MIMO interface in place of the SISO one.

3.3 Utilities

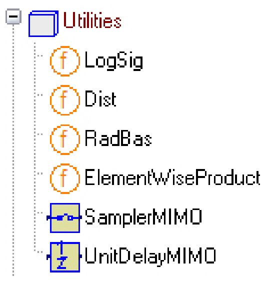

Figure 8: Utilities package

The Utilities package (shown in figure 8) is composed by some mathematical functions and blocks needed to the library to work. In the blocks there are the NeuralNetwork.Utilities.UnitDelayMIMO block, used to model an *Elman* neural network, and the NeuralNetwork.Utilities.SamplerMIMO, used to sample more signals at the same time and used to build the *Elman* neural network example. The mathematical functions instead are used to model a specific *activation function* (*LogSig* and *RadBas*) or to elaborate signals which are used by neurons to compute the *activation signal* (Dist and ElementWiseProduct are used by a layer composed by *radial*).

[4]they differ for the number of recurrent layer

4 An application example

The package Examples contains some instances which allow the user to explore the library capabilities. In this section, an example of how to use the NeuralNetwork library is shown: the entire work process will be explained, from the neural network identification, with MatLab Neural Network toolbox, through the network parameters exchange, to the validation on the model implementation in Modelica.

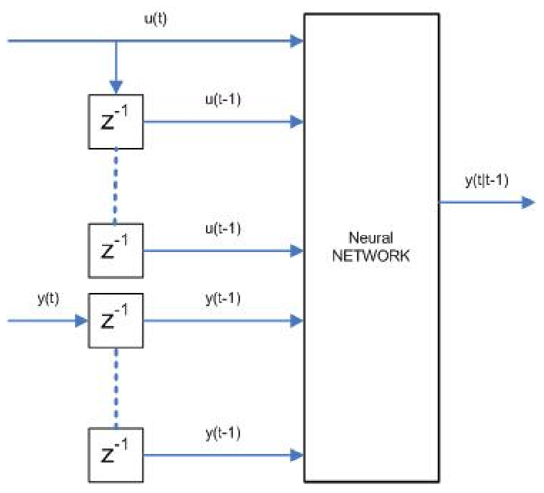

Figure 9: *NARX*: neural network with external dynamics

The example shown here (which is the model FeedForwardNeuralNetwork placed in Examples package) is about a *feed-forward* network with external dynamics. This neural network, shown in figure 9, is a *feed-forward* neural network in which the signals used as inputs are previously delayed. The *feed-forward* neural network with external dynamic, which is normally called *NARX*, performs the following function

$$y(t) = f(u(t)\dots u(t-n_a), y(t)\dots y(t-n_b)).$$

This example shows how to use the elements of the NeuralNetwork library to create a *feed-forward* neural network with external dynamic, where u is a vector composed by two elements, $n_a = 2$ and $n_b = 0$. First of all we have to create the model of the process which has to be identified by the network. We assume that the process is driven by the non-linear function

$$F(t) = (3x(t)x(t-1)x(t-2)) + (y(t)y(t-2)),$$

where x and y are the inputs of the system. Note that the process is dynamic because $F(t)$ uses the input values at t time, $t-1$ time and $t-2$. For this reason we choose to use a dynamical *feed-forward* network with 6 inputs:

$$y(t) = f(x(t), y(t), x(t-1), y(t-1), x(t-2), y(t-2)).$$

To train the network is mandatory to have some input signals and the correspondent outputs. The Matlab environment can be used: define the input signals with the following commands[5]

```
t=0:0.01:10;
x=sin(2*pi*t);
y=cos(5*pi*t);
```

and calculate the output signal of the process from the inputs previously defined.

```
for k=3:length(t)
   f(k)=(3*x(k)*x(k-1)*x(k-2));
   f(k)=f(k)+(y(k)*y(k-2));
end
```

After the input and output signals are created, the network has to be built. To construct a *feed-forward* neural network, the command `newff` has to be used. As parameters, the command requires the variances of the inputs, the dimension of the network and the layer *activation functions*. To do this use the following commands

```
var_x = [min(x) max(x)];
var_X = [var_x;var_x;var_x];
var_y = [min(y) max(y)];
var_Y = [var_y;var_y;var_y];
net = newff([var_X ; var_Y],[4 1],
               {'tansig','purelin'});
```

Note that `var_X` and `var_Y` are a 3×2 matrix, with one line for $x(t)$, one for $x(t-1)$ and one for $x(t-2)$. To train the network, the input signal matrices have to be created (they are `in_X` and `in_Y`). Some parameters, like the train method and the train epochs number, has to be set and then the function `train` can be used. This is done with the following commands:

```
in_X = [ x ; [0 x(1:end-1)] ;
             [0 0 x(1:end-2)] ];
in_Y = [ y ; [0 y(1:end-1)] ;
             [0 0 y(1:end-2)] ];
net.trainFcn = 'trainlm';
```

[5]when dealing with dynamic *feed-forward* networks it is very important that the sampling time during the simulation be the same as the one used for the network training, otherwise the model will not behave correctly.

```
net.trainParam.epochs = 100;
[net,tr]=train(net,[in_X;in_Y],f);
```

To see how the network has learned the non-linear system the command *sim* can be used

```
f_SIM = sim(net,[in_X;in_Y]);
```

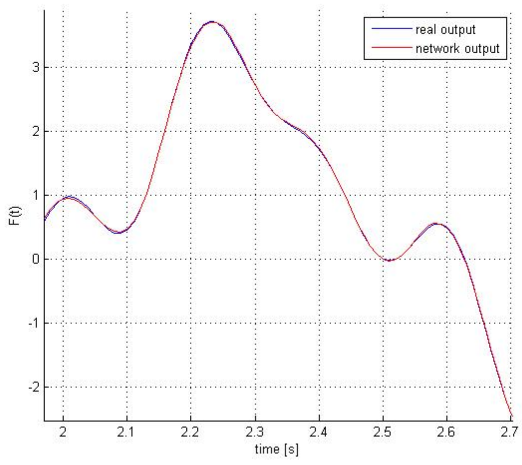

Figure 10: Real process and neural network output comparison

Plotting the real output and the network simulated output (figure 10), we can see that the network has identified the non-linear system very well. Two ways were taken into consideration in order to use the parameters coming from the MatLab environment:

- create a specific script for MatLab (called *extractData.m*) which collects the parameters from the environment and creates a text file containing all the information as the Modelica notation and the library requests;
- use the `DataFiles` library which provides some functions to read/write parameters from/into *.mat* files (saved using the -V4 option).

The `DataFiles` library is a particular implementation supplied by Dymola to manage *.mat* files: this approach was used in absence of a general solution in Modelica.
In this particular example the first way was used. At first, it has to be understood how the MatLab saves the *feed-forward* neural network parameters. Watching the figure 11, which shows how MatLab maps the *weights* and *bias* of the layer on the network object matrices and keeping in mind that the first hidden layer is called *InputLayer* and the others only *Layer*, can be asserted that:

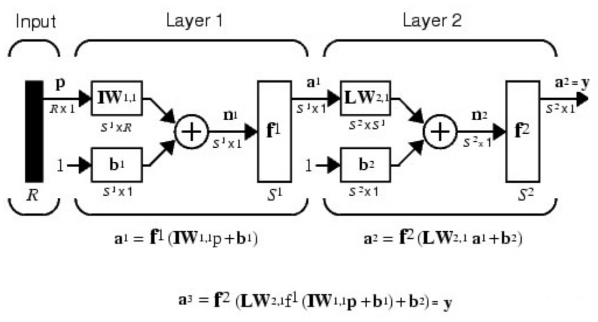

Figure 11: MatLab *weights* and *bias* matrices

- to access to the *weights* matrix of a layer has to be used the command `net.X{1,1}`[6], where *X=IW* for the first layer and *X=LW* for the others; the *weights* matrix is a $[S \times R]$, where S is the neuron number and R the layer inputs number
- to access to the *bias* matrix has to be used the command `net.b{1}`, $[S \times 1]$.

Using this information and the *extractData.m* script, two files, which contain the Modelica definition of the network layers that compose the neural network, were created:

```
extractData(’LW.txt’,’OutputLayer’,
             net.LW{2,1},net.b{2},’lin’)
extractData(’IW.txt’,’HiddenLayer’,
               net.IW{1},net.b{1},’tan’)
```

where ’LW.txt’ and ’IW.txt’ are the names of the file where the definition of the `Modelica neuralNetwork_TwoLayer` *OutputLayer* and *HiddenLayer* are stored. The other parameters of the command are the *weights* and the *bias* matrices and the layer *activation function*.
Now it’s possible to create this neural network using the Modelica language[7]. At first take a `neuralNetwork_TwoLayer` block and change its parameters using the results of the previous steps (located in ’IW.txt’ and ’LW.txt’). Then, since the neural network expects 6 inputs which have to be externally built, some unit delay blocks (with sample time sets to 0.01, which is the input signals sample time) and a multiplexer must be used.
As last step, build a *.mat* file enclosing the input signals used in MatLab to simulate the neural network. To compare the Modelica output to the MatLab one, enclose the output signals too.

[6]For the index selection please use the MatLab Neural Network toolbox help.

[7]The example model (figure 12) was created in Dymola[7]

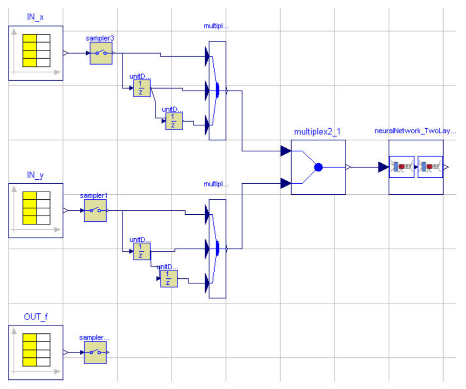

Figure 12: `FeedForwardNeuralNetwork` example

```
IN_x=[t’ , x’];
IN_y=[t’ , y’];
OUT_f=[t’ , f_SIM’];
save testData_FeedForwardNN.mat -V4
IN_x IN_y OUT_f
```

the figure 13 shows the output of the Modelica simulation and the output of MatLab: see that there is no difference between them.

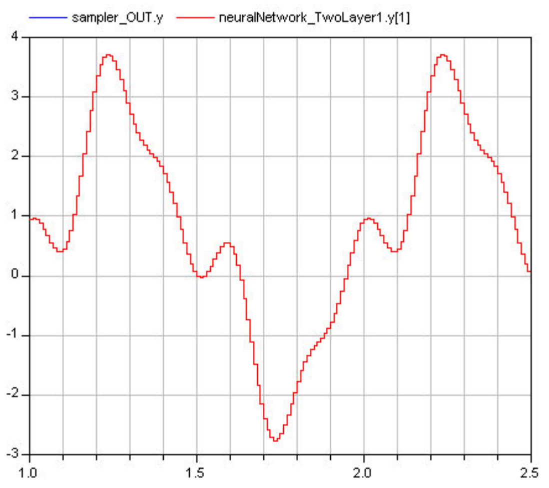

Figure 13: Matlab and Modelica simulation output comparison

Similar examples have been built for the other kinds of networks in the library. They are available in the Examples package, to check their results against the Matlab implementation.

5 Conclusion

A Modelica library, providing the neural network mathematical model is presented. This library is developed to be used to simulate a non-linear system,

previously identified through a specific neural network training system. The `NeuralNetwork` library is developed in Modelica 2.2 and it offers all the required capabilities to create and use different kinds of neural networks.
Currently, only the *feed-forward*, the *elman*[6] and the *radial basis* neural network can be modeled and simulated, but it is possible to build different network structures, by using the basic elements provided by the library. In section 4, a library extension example is shown: a dynamical neural network model is created using the library blocks. The entire work process is explained, from the neural network identification, with an external training software, through the network parameters exchange (from the training software environment to the Modelica one), to the validation of the Modelica model. This lead us to show that there is no difference between the Modelica simulation output and the MatLab one.
The library is publicly available under the Modelica License from the www.modelica.org website.

References

[1] McCulloch, W. S. and Pitts, W., A logical calculus of the ideas immanent in nervous activity. Bulletin of Mathematical Biophysics, 5, 115–133, 1943.

[2] Rosenblatt, F., The perceptron: A probabilistic model for information storage and organization in the brain, Psychological Reviw, 1958, 65, 386-408.

[3] Minsky, M.L. and Papert, S., Perceptrons: An Introduction to Computational Geometry. Cambridge, MA: MIT Press, 1969.

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

[5] Bittanti, S., Identificazione dei modelli e sistemi adattativi, Pitagora Editrice, 2002.

[6] Elman, J. L., Finding structure in time, Cognitive Science, 15, 1990, 179-211.

[7] Dymola, Dynamic Modeling Laboratory, Dynasim AB, Lund, Sweden.

[8] The Math Works Inc., MATLAB®- The language of Technical Computing, 1997.