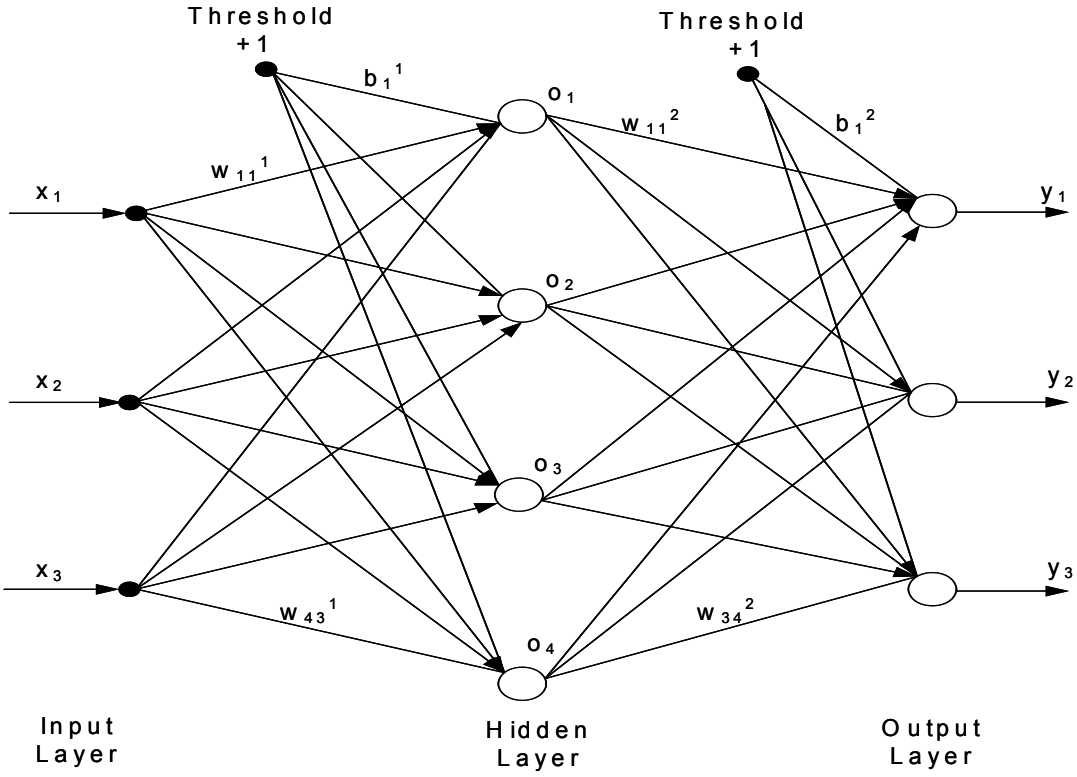
School of Electrical, Electronic and Computer Engineering

EEE 8005 – Student Directed Learning (SDL)

Industrial Automation – Artificial Neural networks

Written by: Shady Gadoue



Module Leader: Dr. Damian Giaouris

Damian.Giaouris@ncl.ac.uk

Artificial Neural Networks

Introduction

Artificial Neural Networks (ANN) are a branch of the field known as "Artificial Intelligence" (AI) which may also consists of Fuzzy logic (FL) and Genetic Algorithms (GA). ANN are based on the basic model of the human brain with capability of generalization and learning. The purpose of this simulation to the simple model of human neural cell is to acquire the *intelligent* features of these cells. The term "artificial" means that neural nets are implemented in computer programs that are able to handle the large number of necessary calculations during the learning process.

ANN have gain a lot of interest over the last few years as a powerful technique to solve many real world problems. Compared to conventional programming, they own the capability of solving problems that do not have algorithmic solution and are therefore found suitable to tackle problems that people are good to solve such as pattern recognition. They have been therefore successfully applied in various application areas such as finance, medicine (clinical diagnosis and image analysis), engineering and physics. Moreover, ANN have been introduced in solving a lot of problems related to prediction, classification, control and identification. This is due to their high ability to learn from experience in order to improve their performance and to adapt themselves to changes in the environment in addition to their ability to deal with incomplete information or noisy data and can be very effective especially in situations where it is not possible to define the rules or steps that lead to the solution of a problem.

The basic computing element in the biological system is the neuron which receives electrochemical signals from different sources and generates electric impulses to be transmitted to other neurons. The human nervous system consists of about 10^{10} to 10^{12} neurons which are capable of storing numerous bits of information. Each neural cell works like a simple processor and only the massive interaction between all cells and their parallel processing makes the brain's abilities possible. About 10% of the neurons are input and output whereas the remaining are interconnected with other neurons performing storage of information and transformation of the signals being propagated through the

network. As shown in Fig. 1 a neuron is composed of a nucleus, a cell body, numerous dendritic links which provide input connections from other neurons through synapses and an axon trunk which carries the output action to other neurons through synapses and terminal links. The connections between the neurons are *adaptive*, what means that the connection structure is changing dynamically. It is commonly acknowledged that the learning ability of the human brain is based on this adaptation.

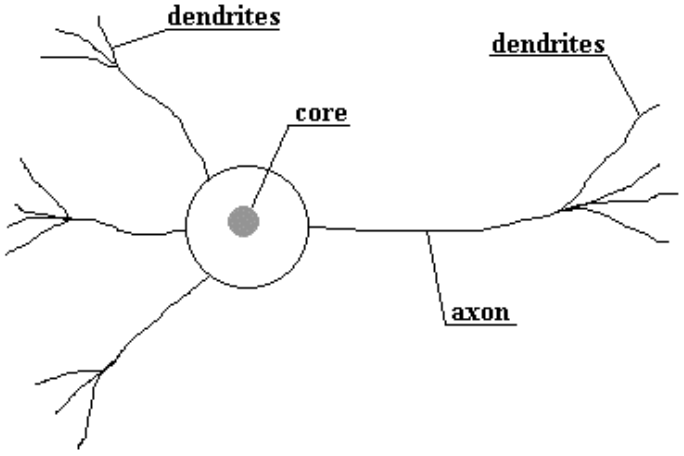


Fig.1 Structure of a neural cell in the human brain

Stimulated by the structure of the brain, an ANN consists of a set of highly interconnected processing units, called *nodes* or *units*. Each unit is designed to mimic its biological counterpart, the neuron. Each accepts a weighted set of inputs and responds with an output. ANN resembles the biological neuron in acquiring knowledge by learning from examples and storing these informations within inter-neuron connection strengths called *weights*.

Application areas of Artificial Neural Networks

Application areas of ANN can be technically divided into the following categories:

Classification and diagnostic: ANN have been applied in the field of diagnosis in medicine, engineering and manufacturing by correct association between input patterns representing some forms of abnormal behaviour with the corresponding disease or fault type. An example is fault diagnosis of electrical motors

Pattern recognition: ANN have been successfully applied in recognition of complex patterns such as: speech recognition, handwritten character recognition and a lot of other application in the area of image processing.

Modelling: A neural network is a powerful data modelling tool that is able to capture and represent complex input/output relationships. The true power and advantage of neural networks lies in their ability to represent both linear and non-linear relationships and in their ability to learn these relationships directly from the data being modelled. The purpose of the neural network is to create a model that correctly maps the input to the output using historical data so that the model can then be used to produce the output when the desired output is unknown.

Forecasting and prediction: ANN have shown high efficiency as predictive tool by looking at the present informations and predict what is going to happen.

Estimation and Control: ANN have been powerfully applied in the field of automatic control in system identification, adaptive control, parameter estimation and optimization and a lot of other applications in this field.

Structure of Artificial Neural Networks

Similar to the biological neural cell, the unit of structure of ANN is the neuron which consists basically of a summer and an activation function as shown in Fig. 2.

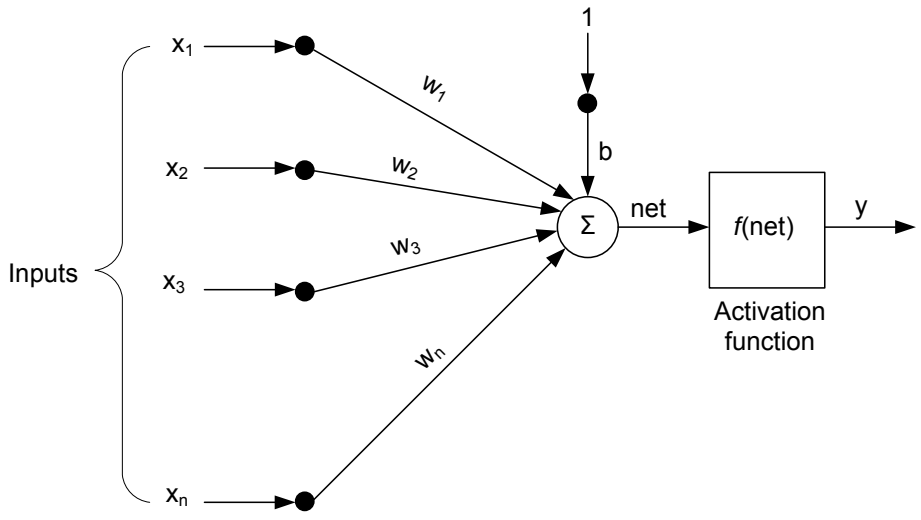


Fig. 2 Structure of the artificial neuron

where x_1 ,x_2,x_3,…,x_n are the inputs to the neuron with corresponding weights w_1 ,w_2,w_3,…,w_n which model the synaptic neural connections in biological nets and act in such a way as to increase or decrease the input signals to the neuron. Sometimes a threshold term b is added to the inputs. Generally, inputs, weights, thresholds and neuron

output could be real value or binary or bipolar. All inputs are multiplied by their corresponding weights and added together to form the net input to the neuron called net. The mathematical expression for net can be simply written as:

$$net = \sum_{i=1}^{n} w_i x_i + b = w_1 x_1 + w_2 x_2 + w_3 x_3 + ... + w_n x_n + b$$

The neuron behaves as an activation or mapping function *f*(net) to produce an output *y* which can be expressed as:

$$y = f(net) = f\left(\sum_{i=1}^{n} w_i x_i + b \right)$$

where *f* is called the neuron activation function or the neuron transfer function. Some examples of the neuron activation functions are:

(1) Linear activation function

In this case *f* =1, the neuron transfer function is shown in Fig.3 where:

$$y = f(net) = \sum_{i=1}^{n} w_i x_i + b = net$$

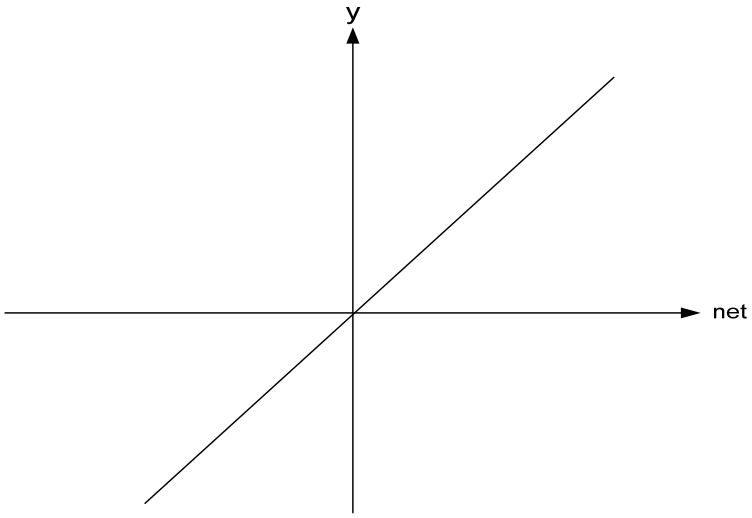


Fig.3 Linear transfer function

(2) Threshold activation function:

In this case, the output is hard limited to two values +1 and -1 (sometimes 0) depending on the sign of net as shown in Fig.4. The expression of the output *y* in this case can be written as:

$$y = \begin{cases} +1 & if\ net > 0 \\ -1 & if\ net < 0 \end{cases}$$

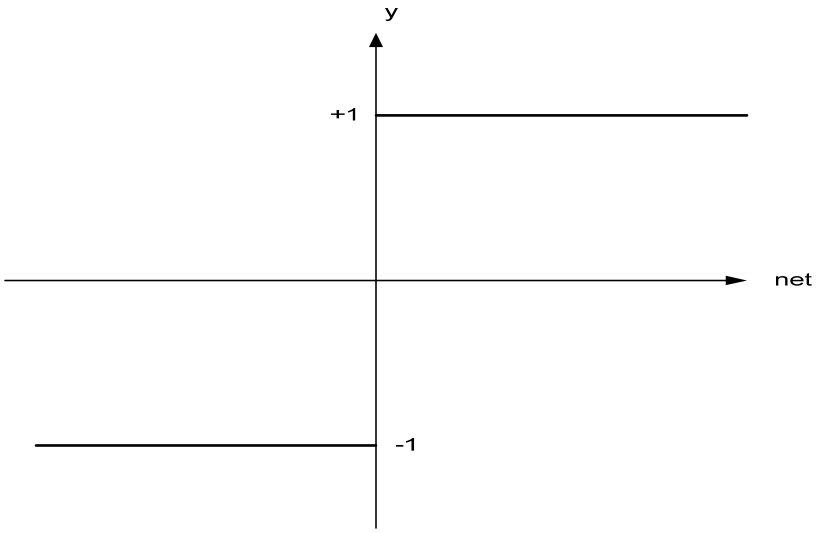


Fig.4 Threshold transfer function

(3) Sigmoid function:

In this case the net neuron input is mapped into values between +1 and 0.The neuron transfer function is shown in Fig.5 and is given by:

$$y = \frac{1}{1 + \exp\left(-\frac{net}{T}\right)}$$

where T is a constant

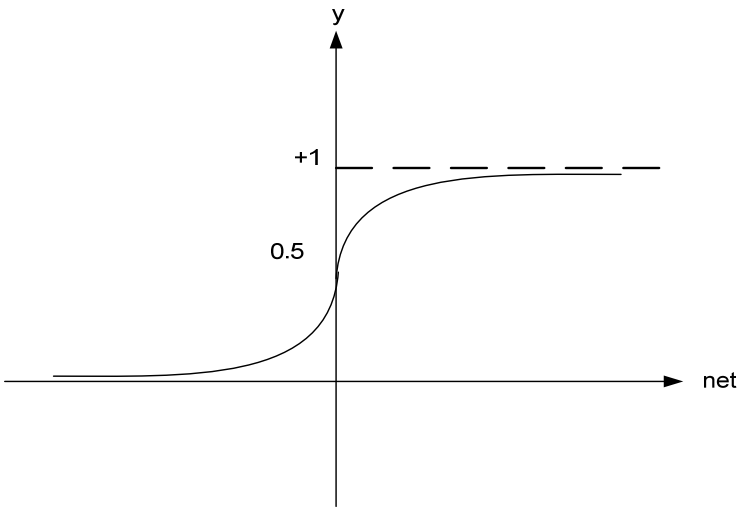


Fig.5 Log-Sigmoid transfer function

(4) Tansigmoid function:

In this case the net neuron input is mapped into values between +1 and -1. The neuron transfer function is shown in Fig.6 and is given by:

$$y = \tanh(net) = \frac{1-\exp(-net)}{1+\exp(-net)}$$

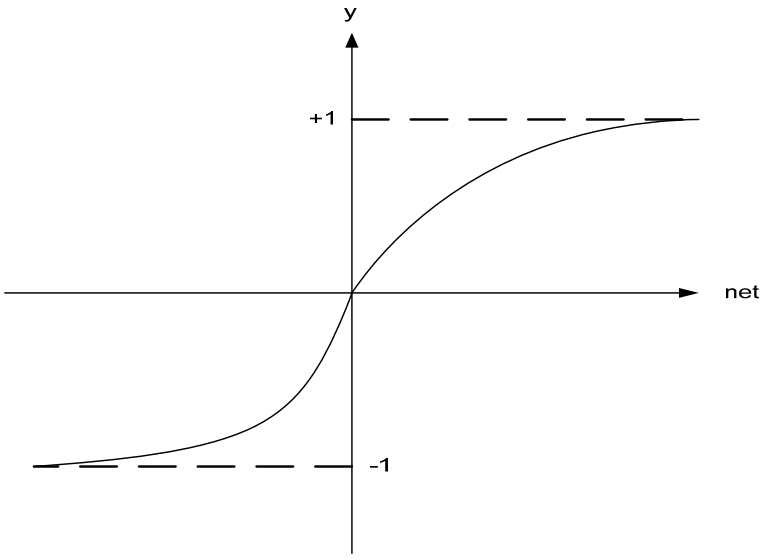


Fig.6 Tan-Sigmoid transfer function

Since ANN are frequently used as nonlinear function approximators, the activation function *f* is usually a nonlinear function. The most common type of ANN is the multi layer feedforward neural network which consists of group of interconnected neurons

organised in layers: input layer, hidden layer and output layer where each layer consists of a group of neurons as shown in Fig.7. It is feedforward because signals propagate only in a forward direction from the input nodes to the output nodes and no signals are allowed to be fed-back among the neurons. The number of hidden layers, number of neurons in each layer totally depends on the complexity of the problem being solved by the network. This structure is commonly used in system identification and nonlinear function approximation applications. The shown network has 3 inputs **x**, 4 outputs from the hidden layer **o** and 3 outputs **y** at the output layer that can be written as:

$$\mathbf{x} = [x_1 \quad x_2 \quad x_3]^T, \quad \mathbf{o} = [o_1 \quad o_2 \quad o_3 \quad o_4]^T \; and \quad \mathbf{y} = [y_1 \quad y_2 \quad y_3]^T$$

In the first layer, the weight and the bias matrices can be written as:

$$\mathbf{w^1} = \begin{bmatrix} w_{11}^1 & w_{12}^1 & w_{13}^1 \\ w_{21}^1 & w_{22}^1 & w_{23}^1 \\ w_{31}^1 & w_{32}^1 & w_{33}^1 \\ w_{41}^1 & w_{42}^1 & w_{43}^1 \end{bmatrix} \qquad \mathbf{b^1} = \begin{bmatrix} b_1^1 \\ b_2^1 \\ b_3^1 \\ b_4^1 \end{bmatrix}$$

where the superscript *1* assigns for 1st layer. The output of the hidden layer **o** can be written in matrix form as:

$$\mathbf{o} = f^1 \left\{ \left[\mathbf{w^1 x} + \mathbf{b^1} \right] \right\}$$

where f^1 is the activation function of the first layer. The output of the last layer (the network output) can be also calculated in similar way. For example the first output of the hidden layer o_1 and the first output from the output layer (first network output) y_1 could be calculated as:

$$o_1 = f^1\left(w_{11}^1 x_1 + w_{12}^1 x_2 + w_{13}^1 x_3 + b_1^1\right)$$

$$y_1 = f^2\left(w_{11}^2 o_1 + w_{12}^2 o_2 + w_{13}^2 o_3 + w_{14}^2 o_4 + b_1^2\right)$$

where f^2 is the activation function of the output layer.

Another architecture of ANN commonly used in control applications is the recurrent neural networks (RNN) which differs from the feedforward structure by having feedback connections which propagate the outputs of some neurons back to the inputs of other neurons in order to perform repeated computations on the signal as shown in Fig.8.

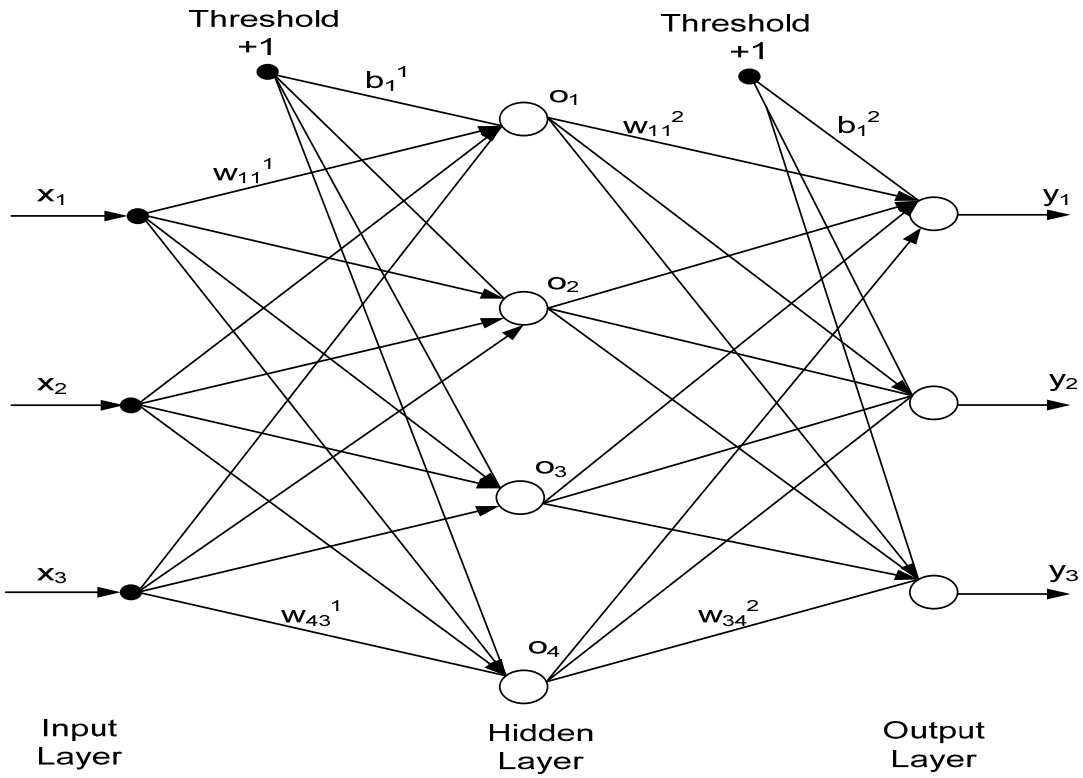


Fig.7 Architecture of multilayer feedforward neural network

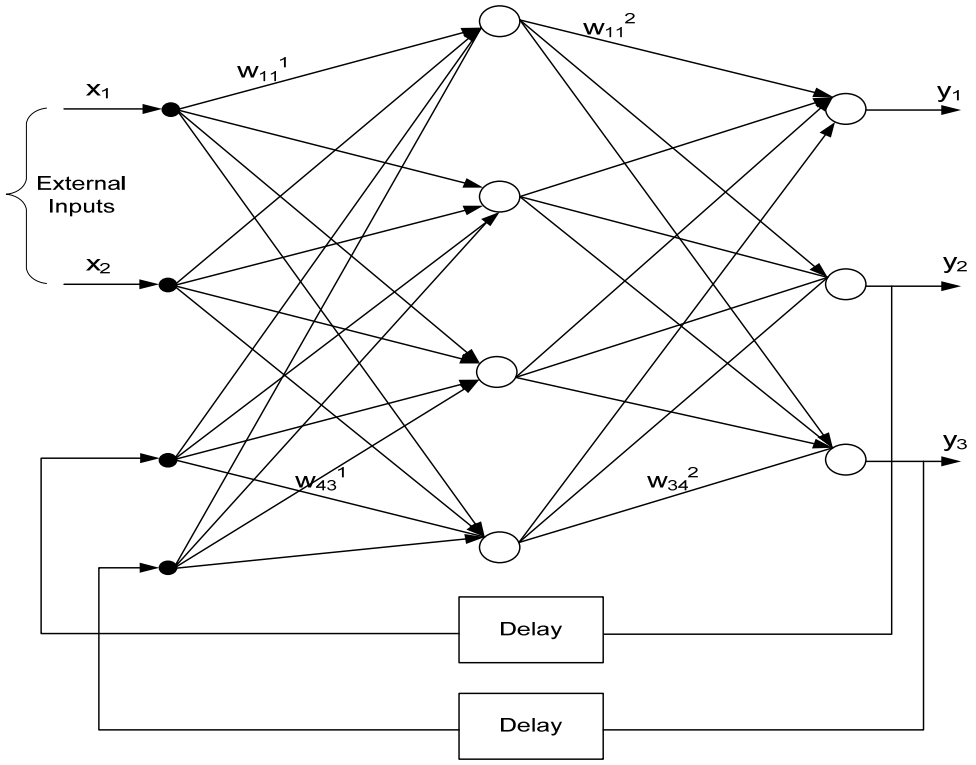


Fig.8 Structure of recurrent neural network

Classification of ANN

As discussed before, ANN resemble the human brain in learning through training and data storage. Based on learning strategy three main categories of ANN can be described: Supervised, reinforcement or unsupervised learning. Supervised and unsupervised will be considered here due to their popularity. However, all interest will be given to the supervised type of learning since it is frequently used in the majority of ANN applications.

Supervised Learning: In this type of learning a teacher is present during the learning process and the ANN is trained through a given input/ target data training pattern which includes input pattern associated with the corresponding target or desired pattern This training pattern will form a pool of examples used to train the ANN in order to learn a specific behaviour and the presence of desired output(s) for each input in the training pattern makes this type of learning supervised. During the learning process, the neural network output is compared with the target value and a network weight correction via a learning algorithm is performed in such a way to minimize an error function between the two values. This is an optimization problem in which the learning algorithm is searching for the optimal weights that can represent the solution to the approximation problem.

A commonly used error function is the mean-squared error (MSE) which tries to minimize the average error between the network's output and the target value over all the example pairs and a commonly used weight correction algorithm is a gradient descent algorithm called Back Propagation. This algorithm is used frequently to train multi layer feedforward ANN either online or off-line.

Unsupervised Learning: In this type of learning, no desired or target is available to the network and only the input pattern is present, i.e. there is no teacher to learn the network. The system must learn by discovering and adapting to structured features in the input pattern. This is done by adapting to statistical regularities or clustering of patterns from the input training samples.

Back propagation training of ANN

As discussed before, supervised learning is frequently used to train multi layer feedforward ANN in a lot of applications. Usually, back propagation learning algorithm

is used to update the network weights during training in order to improve the network performance. The block diagram of the training process is shown in Fig.9.

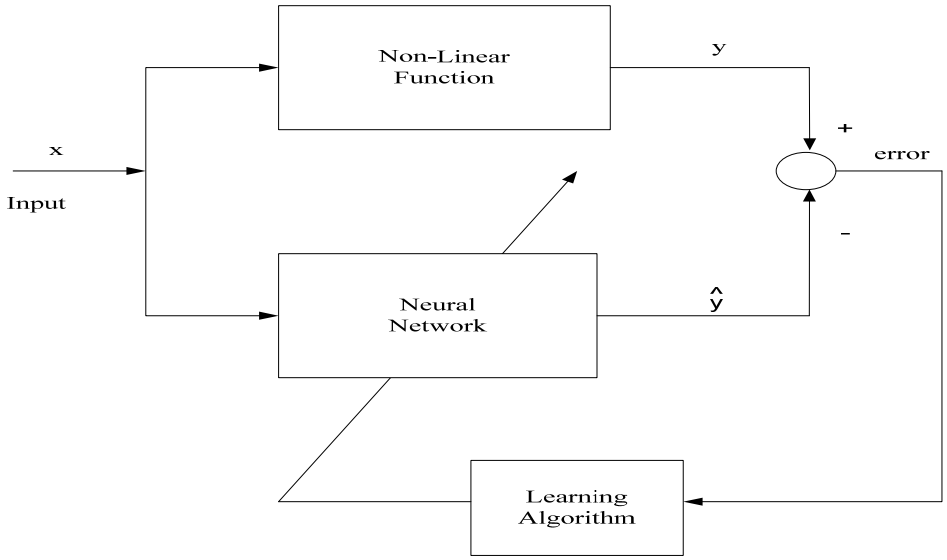


Fig.9 Block diagram of training neural network as function approximator using supervised learning

Back propagation (BP) is one of the gradient descent algorithms used to reduce the performance function ***E*** through updating the neural network weights by moving them along the negative of the gradient of the performance function. The term *backpropagation* refers to the manner in which the gradient is computed for nonlinear multilayer networks. This method requires that activation functions *f* are differentiable as the weight update rule is based on the gradient of the error function which is defined in terms of the weights and activation functions. The general rule used to update the weight can be written as:

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}}$$

where wij is the weight on the connection between node i and j and η is the learning rate which is multiplied by the negative of the gradient to determine the changes to the weights and biases. The larger the learning rate, the bigger the step. If the learning rate is made too large, the algorithm becomes unstable. If the learning rate is set too small, the algorithm takes a long time to converge. Therefore careful choice of η is vital to increase

the convergence time without affecting algorithm stability. It is not practical to determine the optimal setting for the learning rate before training, and, in fact, the optimal learning rate changes during the training process, as the algorithm moves across the performance surface.

The new weight can be updated as follow:

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

In order to accelerate the convergence of the network, some algorithms may introduce the previous weight change into the updating equation as:

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

where α is called the momentum rate which can be any number between 0 and 1. When the momentum constant is 0, a weight change is based only on the gradient. When the momentum constant is 1, the new weight change is set to equal the last weight change and the gradient is simply ignored.

To train an ANN using BP to solve a specific problem there are generally four main steps in the training process:

1- Assemble the suitable training data
2- Create the network object
3- Train the network
4- Simulate the network response to new inputs

Using Matlab to simulate neural networks

*** Create a neural network (*newff*):**

After collecting the suitable training data, a network is to be created. In Matlab, the command *newff* is used to create a multilayer feedforward ANN called *net*. It requires four inputs: The first input is an R by 2 matrix of min and max values for R input elements, the second input is an array containing the size of each layer starting from the second one, the third input is a cell array describing the activation functions used for each layer and the final input contains the name of the training function to be used.

Example 1: Typing the command:

net= newff ([-4 3; -5 5], [4,1], {'tansig','purelin'},'trainlm' **)**

Creates a 3 layer feedforward ANN: input, hidden and output layers. The program identifies the number of input nodes from the defined ranges. The input vector contains 2 elements (2 nodes) where the first input values range from -4 to 3 and the second input ones range from -5 to 5. The 2nd layer (hidden layer) consists of 4 neurons while the output layer consists of only one neuron (one output). The activation functions used in the second layer is *tansig* (Tansigmoid transfer function) and for the output layer is purelin (Linear transfer function). Finally, the training function is *trainlm*. The structure of the created ANN is shown in Fig. 10.

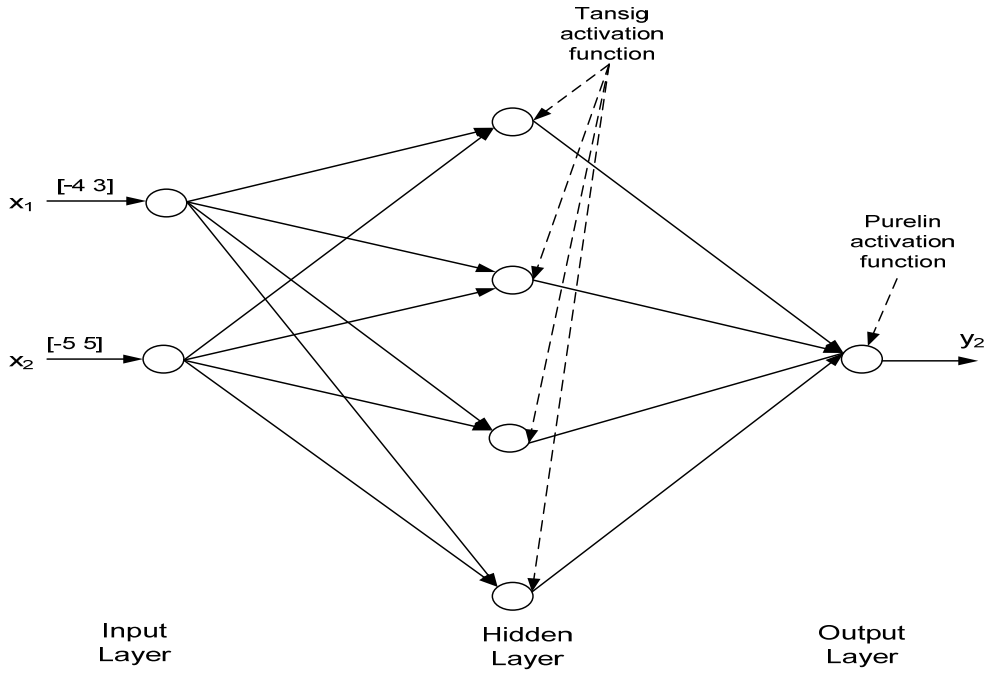


Fig. 10 Schematic of example 1 ANN

Using the command *newff* not only creates the neural network object but also randomly initializes all weights and biases for the network which make the created ANN ready for training. If anyone would like to reinitialize them, the command *init* can be used on the form *net= init(net);*

*** Network training (*train*):**

After the network has been created and all weights and biases have been randomly initialized, the network becomes ready to be trained using the pre-collected training data (set of examples of the proper network performance). During the training, the weights

and biases of the network are iteratively updated to minimize an error function between the desired (target) output(s) and the network output(s). This error function is defined in Matlab as *net.performFcn*. The default one is the mean square error *MSE* which represents the average squared error between the network outputs and the target values. Several high performance training algorithms providing faster training than the conventional back propagation algorithm are already defined in Matlab based on variable learning rate (traingda, traingdx) or other numerical optimization techniques such as quasi Newton methods (trainbfg) and Levenberg-Marquardt (trainlm). All these training algorithms work in the batch mode where the weights and biases of the network are updated only after the entire training set has been applied to the network (one iteration). The gradients calculated at each training example are added together to determine the change in the weights and biases. Some other training algorithms work in the incremental mode where the gradient is computed and the weights are updated after each input is applied to the network (one iteration).

The most common training functions used in Matlab are:

traingd, traingdm, traingda, traingdx, trainlm

Training parameters:

net.trainParam.show = 100; % the training result is shown after every 100 iterations (epochs)

net.trainParam.lr = 0.05; %Defining the network learning rate

net.trainParam.epochs = 1000; % Defining the max number of iterations

net.trainParam.goal = 1e-4; % Defining the training stopping criterion

After defining the training function, the training parameters, the training pattern (p,t) where p is the input and t is the target, the network is ready to be trained. Then using the Matlab command *train* will start training the network. *[net,tr]=train(net,p,t)*

*** Network simulation (*sim*):**

After the network has been trained, a simulation stage is performed to check the network output *y* corresponding to a given input p_1 using the Matlab command *sim*. knowing the target values t_1 corresponding to the input p_1 the error between the ANN output and the true output can be calculated. This error should be very small if the

training was successfully performed. The general form of the *sim* command can be written as: y_1=sim(net,p_1)

Example 2: Function approximation

In this example ANN will be used as a nonlinear function approximator. Consider the function:

$$y = f(x) = 0.03x^3 - 0.2x^2 + 6x + 5$$

Our task is to design a ANN to model this function where x= [0, 20].

Step 1: Generate the training data

```
>> x=0:0.25:20
>> y=(0.03*x.^3)-(0.2*x.^2)+(6*x)+5
>> p=x;
>> t=y;
```

Now we have a training pattern (p,t) that will be used to train the ANN

Step 2: Creating the neural network

```
>> net=newff(minmax(p),[10,1],{'tansig','purelin'},'trainlm');
>> net.trainParam.goal=1e-5;
>> net.trainParam.epochs=500;
```

Step 3: Training the neural network *net*

```
>> [net,tr]=train(net,p,t);
TRAINLM, Epoch 0/500, MSE 16119.4/1e-005, Gradient 21303.2/1e-010
TRAINLM, Epoch 25/500, MSE 6.2698/1e-005, Gradient 4619.86/1e-010
TRAINLM, Epoch 50/500, MSE 0.868411/1e-005, Gradient 32923.3/1e-010
TRAINLM, Epoch 75/500, MSE 0.0333011/1e-005, Gradient 42.0376/1e-010
TRAINLM, Epoch 100/500, MSE 0.0152236/1e-005, Gradient 9.68433/1e-010
TRAINLM, Epoch 125/500, MSE 0.0103669/1e-005, Gradient 4.12834/1e-010
TRAINLM, Epoch 150/500, MSE 0.00515693/1e-005, Gradient 382.348/1e-010
TRAINLM, Epoch 175/500, MSE 0.00253803/1e-005, Gradient 109.02/1e-010
TRAINLM, Epoch 200/500, MSE 0.00071966/1e-005, Gradient 133.928/1e-010
TRAINLM, Epoch 225/500, MSE 0.000603385/1e-005, Gradient 2.051/1e-010
```

TRAINLM, Epoch 250/500, MSE 0.000540297/1e-005, Gradient 4.93452/1e-010

TRAINLM, Epoch 275/500, MSE 0.000383632/1e-005, Gradient 31.178/1e-010

TRAINLM, Epoch 300/500, MSE 0.00015986/1e-005, Gradient 55.3771/1e-010

TRAINLM, Epoch 325/500, MSE 9.98549e-005/1e-005, Gradient 15.7561/1e-010

TRAINLM, Epoch 350/500, MSE 4.29489e-005/1e-005, Gradient 195.24/1e-010

TRAINLM, Epoch 375/500, MSE 2.2137e-005/1e-005, Gradient 33.7242/1e-010

TRAINLM, Epoch 400/500, MSE 1.59442e-005/1e-005, Gradient 13.9289/1e-010

TRAINLM, Epoch 425/500, MSE 1.27898e-005/1e-005, Gradient 7.69581/1e-010

TRAINLM, Epoch 450/500, MSE 1.08291e-005/1e-005, Gradient 4.9221/1e-010

TRAINLM, Epoch 465/500, MSE 9.96279e-006/1e-005, Gradient 3.936/1e-010

TRAINLM, Performance goal met.

During the training, the following Figure appears. It represents the network performance (in blue) versus the number of epochs. The network performance starts by a large value at the first epochs and due to training the weights are adjusted to minimize this function which makes it decreasing. Moreover, a black constant line is plotted representing the training goal (after which value of the network performance we can stop training). The training stops when the blue line (network performance) intersects with the black line (training goal). The performance function of the network is shown in Fig.11.

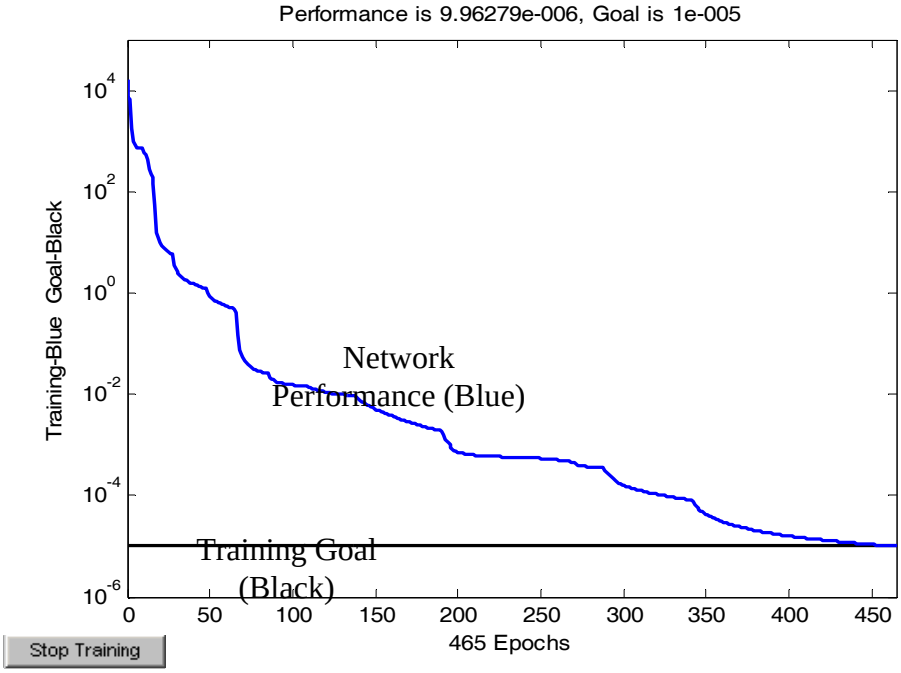


Fig.11 Performance function of the network during training

Step 4: Simulate the trained ANN

Let's simulate the ANN with the input vector p=x and then compare the network output a with the correct output y.

>> a=sim(net,p)

To compare *a* and *y* let's plot both on the same graph:

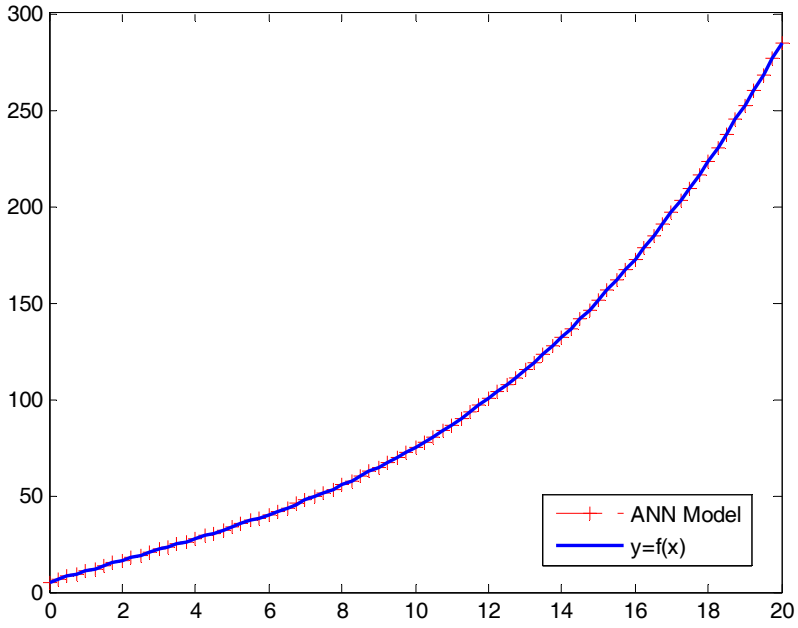


Fig.12 Real function and NN model

It is clear that the ANN model can approximate the function y=f(x) with very good accuracy as shown in Fig.12.

Example 3:

Assuming that you have the following training set (p,t) where p is the input vector and t is the target vector: $p=[x_1,x_2]$ in table 1.

Table 1

Training data

x_1	x_2	t
-1	0	-1

-1	5	-1
2	0	1
2	5	1

Create a multilayer feedforward ANN to learn the relation between p and t and then calculate the error in the approximation for the training data.

The training data could be written as:

```
>> p=[-1 -1 2 2;0 5 0 5]

p =

   -1    -1     2     2
    0     5     0     5

>> t=[-1 -1 1 1]

t =

   -1    -1     1     1
```

Let's start by creating a new network and let's choose the structure as 2-3-1 (3 layer network with 2 neurons in the input layer, 3 neurons in the hidden layer and 1 neuron in the output) and choosing the activation functions in both layer to be tansigmoid and linear and the training function to be traingd.

Note: Usually the selection of the number of hidden layers, number of neurons in each hidden layer, type of activation functions and the training algorithm is done by trial error technique and no general guide lines are assigned to choose the network parameters.

```
>> net=newff(minmax(p),[3,1],{'tansig','purelin'},'traingd');
>>
```

To find out the training parameters set by default for the training type:

```
>> net.trainParam
```

```
ans =

       epochs: 100
         goal: 0
     max_fail: 5
    mem_reduc: 1
     min_grad: 1.0000e-010
           mu: 1.0000e-003
       mu_dec: 0.1000
       mu_inc: 10
       mu_max: 1.0000e+010
         show: 25
         time: Inf
>> net.performFcn

ans =

mse
```

To modify the default training parameters type:

```
>> net.trainParam.epochs=300;
>> net.trainParam.goal=1e-5;
>> net.trainParam.show=50 ;
>> net.trainParam.lr=0.05;
```

To train the network *net* we use the command *train:*

```
[net,tr]=train(net,p,t);
TRAINGD, Epoch 0/300, MSE 3.6913/1e-005, Gradient 4.54729/1e-010
TRAINGD, Epoch 50/300, MSE 0.00437253/1e-005, Gradient 0.0907065/1e-010
TRAINGD, Epoch 100/300, MSE 3.93547e-005/1e-005, Gradient 0.00848131/1e-010
TRAINGD, Epoch 115/300, MSE 9.67565e-006/1e-005, Gradient 0.00420503/1e-010
```

TRAINGD, Performance goal met.

The network performance (in blue) versus the number of epochs appears during training. In our example, the training goal is set to 1e-5, the performance function (defined as MSE) starts from 3.6913. After 115 epochs, the training goal was met and the training stops. Note that you can stop the training anytime using "stop training" button appearing on the down left of the Fig. 13.

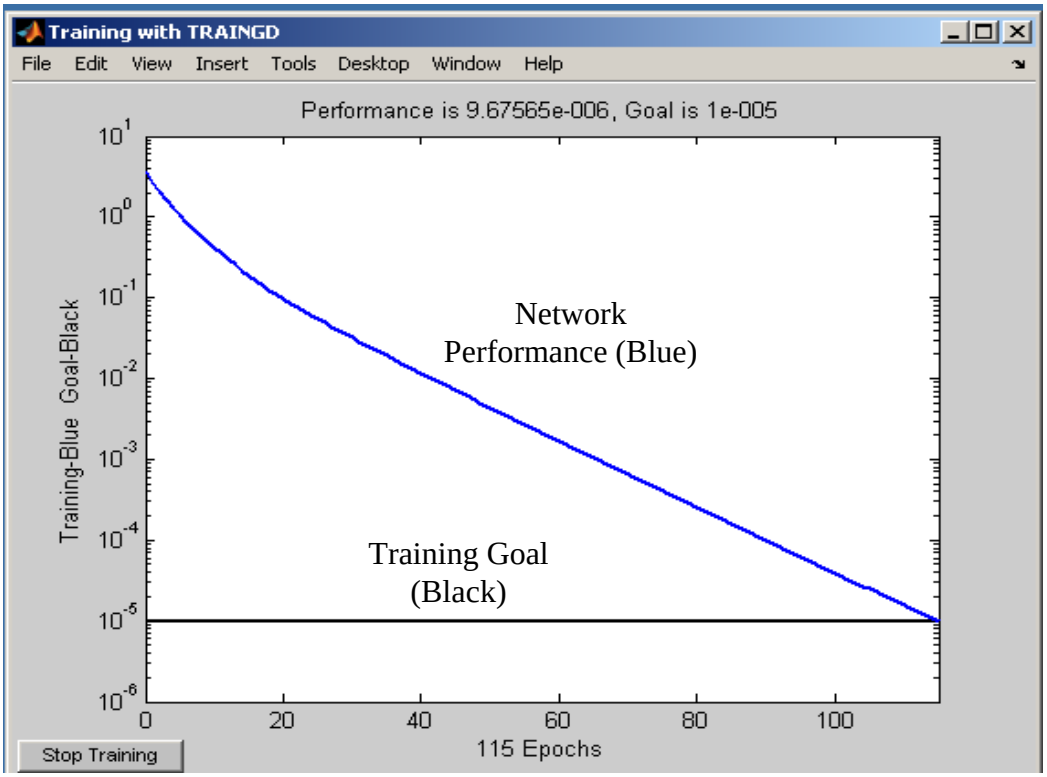


Fig.13 Convergence characteristics of the network during training

After the network has been trained, it can be simulated to check its response using the command *sim.* Let's check the network output corresponding to the training input vector p.

```
>> a=sim(net,p)
a =
   -1.0025   -0.9952    0.9970    0.9999
```

Note that the correct outputs corresponding to p are the target values t:

```
t =
    -1    -1     1     1
```

Comparing the outputs obtained from this well-trained network (let's call it network1) and the desired (correct) outputs we can notice that the network output a is very near to the correct answer.

Sometimes when you type the *train* command after trying all epochs the performance is still greater than the training goal.

[net,tr]=train(net,p,t);

TRAINGD, Epoch 0/300, MSE 0.711565/1e-005, Gradient 1.7024/1e-010

TRAINGD, Epoch 50/300, MSE 0.00944188/1e-005, Gradient 0.0936902/1e-010

TRAINGD, Epoch 100/300, MSE 0.00186081/1e-005, Gradient 0.0290989/1e-010

TRAINGD, Epoch 150/300, MSE 0.000726989/1e-005, Gradient 0.015155/1e-010

TRAINGD, Epoch 200/300, MSE 0.000355603/1e-005, Gradient 0.00962905/1e-010

TRAINGD, Epoch 250/300, MSE 0.000191929/1e-005, Gradient 0.00669697/1e-010

TRAINGD, Epoch 300/300, MSE 0.000109026/1e-005, Gradient 0.0048828/1e-010

TRAINGD, Maximum epoch reached, performance goal was not met.

At that time, the convergence figure will appear as shown in Fig. 14.

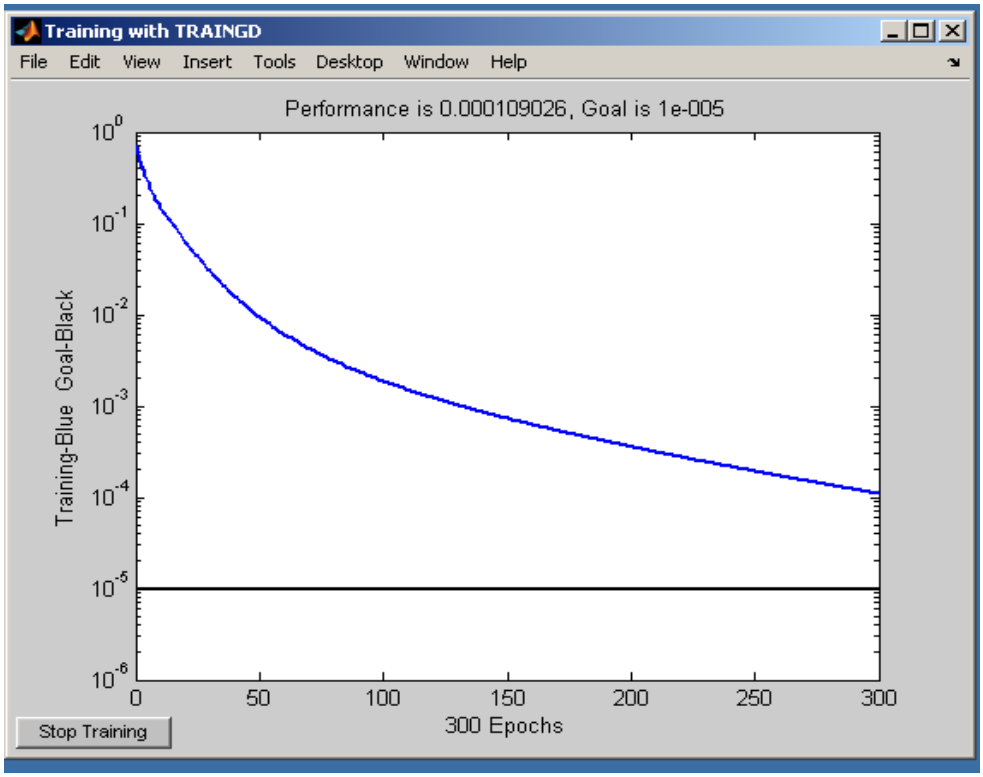


Fig.14 Performance function of the network during training

Note that every time you try to build the same network again use the same commands you may get different training results. This is normal because *newff randomly* initializes the network weights each time that makes its performance difference.

Now let's simulate this not well-trained network (let's call it Network2) using the training inputs p.

```
>> a=sim(net,p)

a =

   -1.0047   -0.9961    0.9852    1.0135
t =

    -1    -1     1     1
```

Note that the approximation accuracy is still Ok but is less accurate than network1 as shown in Table 2.

Table 2

Desired and output values

x_1	x_2	t	Output Network1	Output Network2
-1	0	-1	-1.0025	-1.0047
-1	5	-1	-0.9952	-0.9961
2	0	1	0.9970	0.9852
2	5	1	0.999	1.0135

Exercise: Try to repeat the above example for different number of neurons in the hidden layer, with different activation functions and with different training functions and note the difference in the obtained results.

Example 4: Plant identification using neural networks:

This example illustrates a very relevant application of neural networks in control. It is the problem of system identification. Fig.15 shows a control system which consists of a

controller to control a plant. In some applications, the plant model is not available. Neural networks can be trained to learn the unknown model of the plant using input/output data obtain experimentally from the plant. In this example we assume the plant model is known and is expressed as:

$$G(s) = \frac{5}{s+10}$$

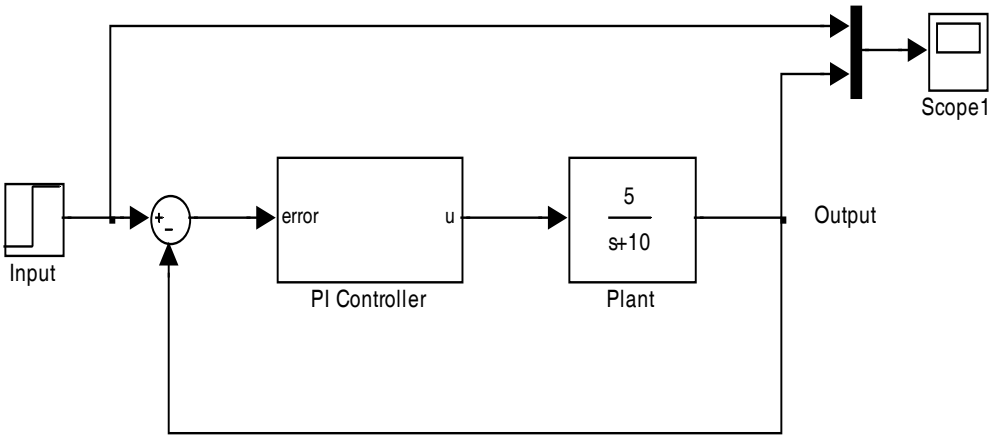


Fig.15 Control system architecture

Step 1: Generating the training data

This will be done by subjecting the plant to a sequence of input *p* and obtaining the corresponding output *t*. The simulink model is shown in Fig.16. Adjust the simulation parameters as in Fig.17 and adjust the ZOH sampling time to 1e-3. The training input is Band-limited white noise which is chosen because it generates normally distributed random input. Adjust power to 0.1, sampling time to 0.1 and seed to 23341 as shown in Fig.18 Use the block *To workspace* to save the training data in array format. Name the input *p* and the output *t*. Set the simulation time to 10 and start simulation. After the end of simulation, if you look at the workspace you should find the arrays representing *p* and *t* or you can open the scopes representing the input and output as shown in Fig.19.

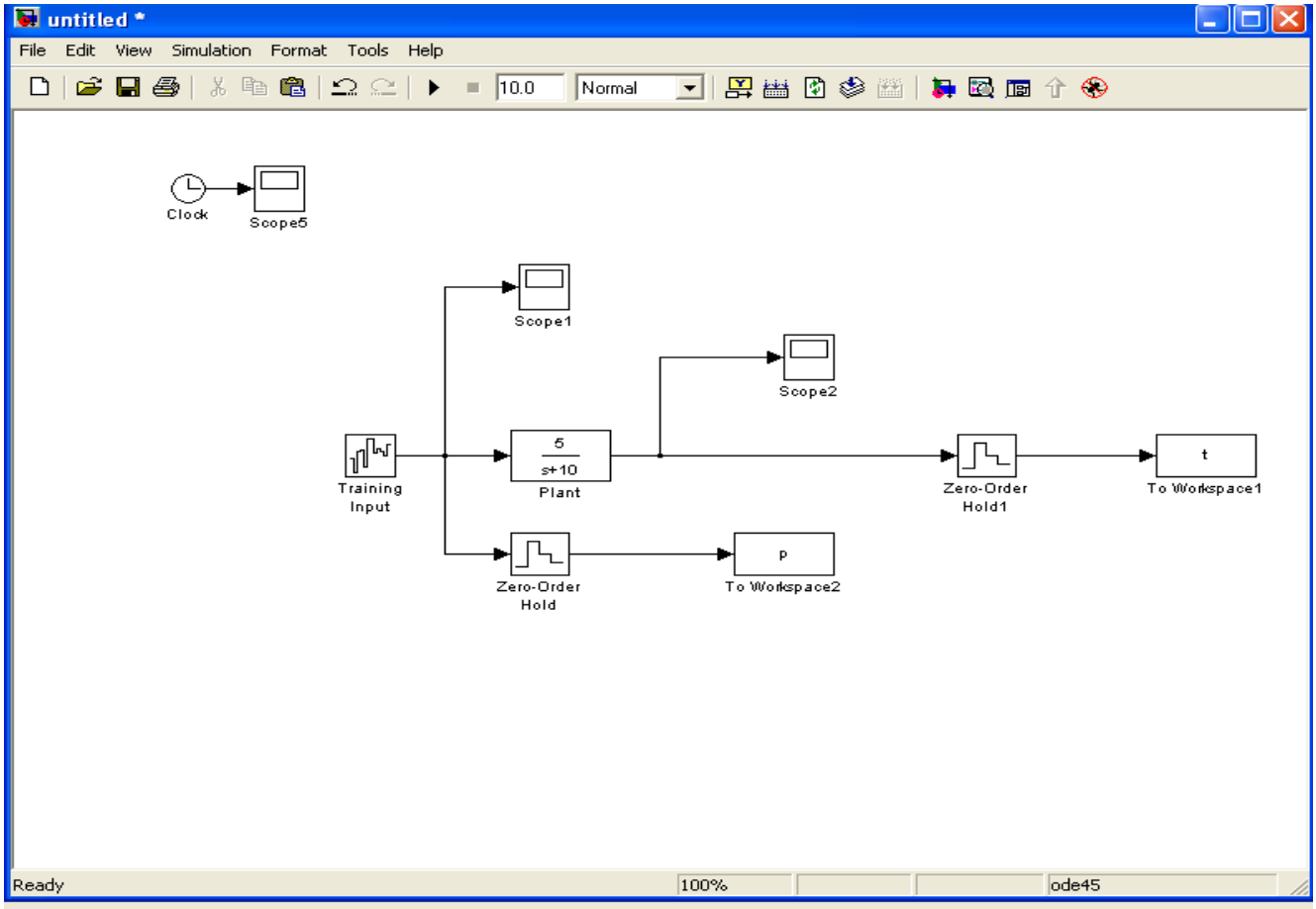


Fig.16 Simulink model used to generate training data

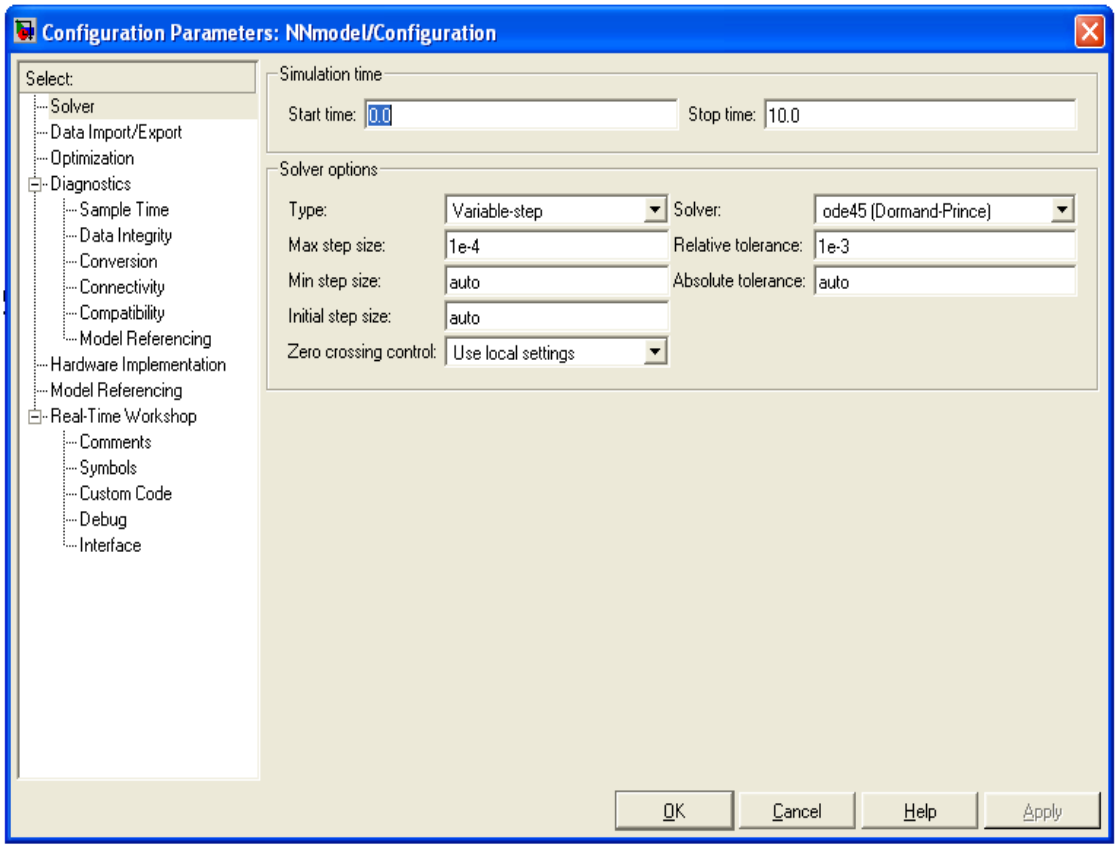


Fig.17 Simulation parameters configuration

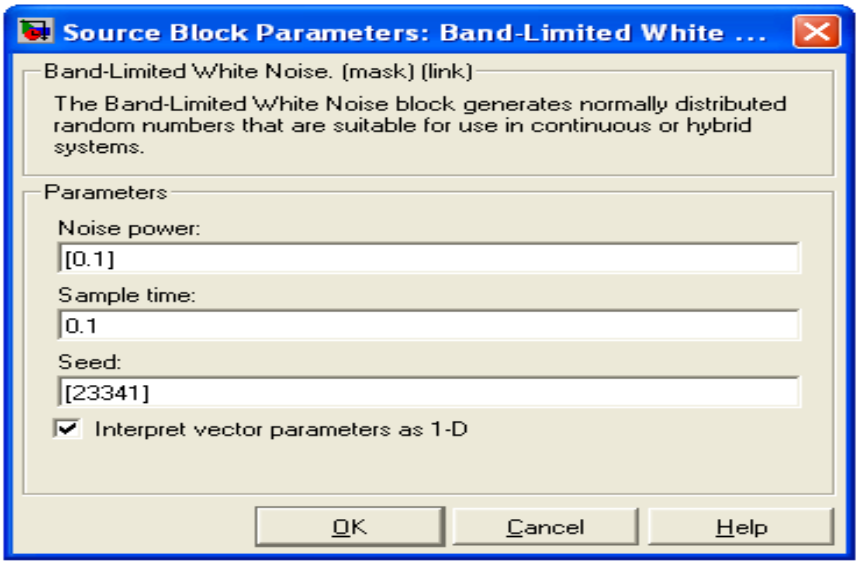
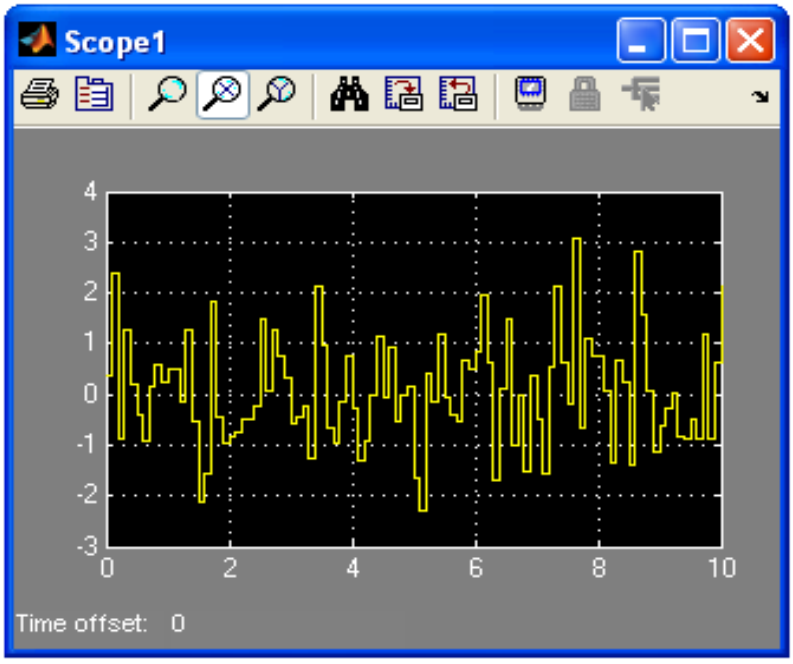
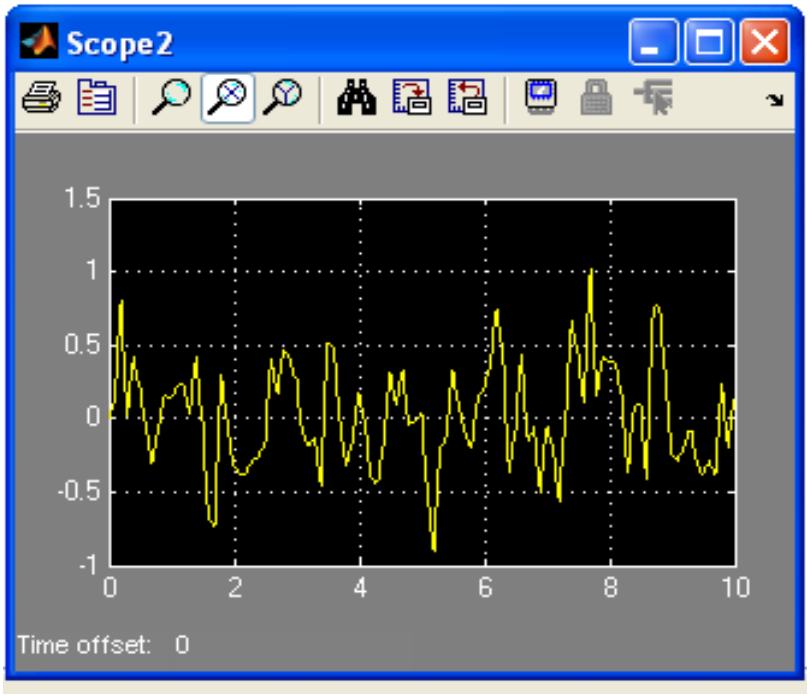


Fig.18 Input block parameters



Input



Output

Fig.19 Input/output training pattern

Step 2: Creating and training the neural network

After generating the training data, go to Matlab command window and start typing commands to create and train the neural network as follow:

```
>> net=newff(minmax(p'),[30,1],{'tansig','purelin'},'trainlm');
>> net.trainParam.epochs=1000;
>> [net,tr]=train(net,p',t');
TRAINLM, Epoch 0/1000, MSE 2.85796/0, Gradient 30464.9/1e-010
TRAINLM, Epoch 25/1000, MSE 0.0384398/0, Gradient 31.4321/1e-010
TRAINLM, Epoch 50/1000, MSE 0.0345469/0, Gradient 24.9989/1e-010
TRAINLM, Epoch 75/1000, MSE 0.0322424/0, Gradient 133.248/1e-010
TRAINLM, Epoch 100/1000, MSE 0.0279006/0, Gradient 9.16506/1e-010
TRAINLM, Epoch 125/1000, MSE 0.0267224/0, Gradient 43.5687/1e-010
TRAINLM, Epoch 150/1000, MSE 0.0250702/0, Gradient 3.73641/1e-010
TRAINLM, Epoch 175/1000, MSE 0.025039/0, Gradient 1.54462/1e-010
TRAINLM, Epoch 200/1000, MSE 0.0250024/0, Gradient 1.79969/1e-010
TRAINLM, Epoch 225/1000, MSE 0.0249883/0, Gradient 8.63699/1e-010
TRAINLM, Epoch 250/1000, MSE 0.0249872/0, Gradient 0.152471/1e-010
TRAINLM, Epoch 275/1000, MSE 0.024987/0, Gradient 0.00954264/1e-010
TRAINLM, Epoch 300/1000, MSE 0.024987/0, Gradient 0.0324412/1e-010
TRAINLM, Epoch 325/1000, MSE 0.024987/0, Gradient 0.0860265/1e-010
TRAINLM, Epoch 350/1000, MSE 0.0249869/0, Gradient 0.0895968/1e-010
TRAINLM, Epoch 375/1000, MSE 0.0249869/0, Gradient 0.0292589/1e-010
TRAINLM, Epoch 400/1000, MSE 0.0249868/0, Gradient 1.25556/1e-010
TRAINLM, Epoch 425/1000, MSE 0.0249865/0, Gradient 1.01611/1e-010
TRAINLM, Epoch 450/1000, MSE 0.0249862/0, Gradient 0.994071/1e-010
TRAINLM, Epoch 475/1000, MSE 0.024986/0, Gradient 1.49094/1e-010
TRAINLM, Epoch 500/1000, MSE 0.0249858/0, Gradient 5.64823/1e-010
TRAINLM, Epoch 525/1000, MSE 0.0249854/0, Gradient 8.91029/1e-010
TRAINLM, Epoch 550/1000, MSE 0.0249851/0, Gradient 3.24994/1e-010
TRAINLM, Epoch 575/1000, MSE 0.0249848/0, Gradient 1.1172/1e-010
```

TRAINLM, Epoch 600/1000, MSE 0.0249846/0, Gradient 0.409848/1e-010
TRAINLM, Epoch 625/1000, MSE 0.0249844/0, Gradient 0.163045/1e-010
TRAINLM, Epoch 650/1000, MSE 0.0249838/0, Gradient 0.266647/1e-010
TRAINLM, Epoch 675/1000, MSE 0.0249837/0, Gradient 0.475013/1e-010
TRAINLM, Epoch 700/1000, MSE 0.0249836/0, Gradient 0.60567/1e-010
TRAINLM, Epoch 725/1000, MSE 0.0249835/0, Gradient 0.3998/1e-010
TRAINLM, Epoch 750/1000, MSE 0.0249833/0, Gradient 0.486509/1e-010
TRAINLM, Epoch 775/1000, MSE 0.024983/0, Gradient 2.04466/1e-010
TRAINLM, Epoch 800/1000, MSE 0.0249827/0, Gradient 1.41733/1e-010
TRAINLM, Epoch 825/1000, MSE 0.0249825/0, Gradient 0.378752/1e-010
TRAINLM, Epoch 850/1000, MSE 0.0249824/0, Gradient 1.17317/1e-010
TRAINLM, Epoch 875/1000, MSE 0.0249823/0, Gradient 0.859761/1e-010
TRAINLM, Epoch 900/1000, MSE 0.0249822/0, Gradient 0.0968464/1e-010
TRAINLM, Epoch 925/1000, MSE 0.024982/0, Gradient 10.6725/1e-010
TRAINLM, Epoch 950/1000, MSE 0.0249819/0, Gradient 0.036869/1e-010
TRAINLM, Epoch 975/1000, MSE 0.0249818/0, Gradient 2.52703/1e-010
TRAINLM, Epoch 1000/1000, MSE 0.0249818/0, Gradient 3.94388/1e-010
TRAINLM, Maximum epoch reached, performance goal was not met.

During training the network performance will be shown as in Fig.20.

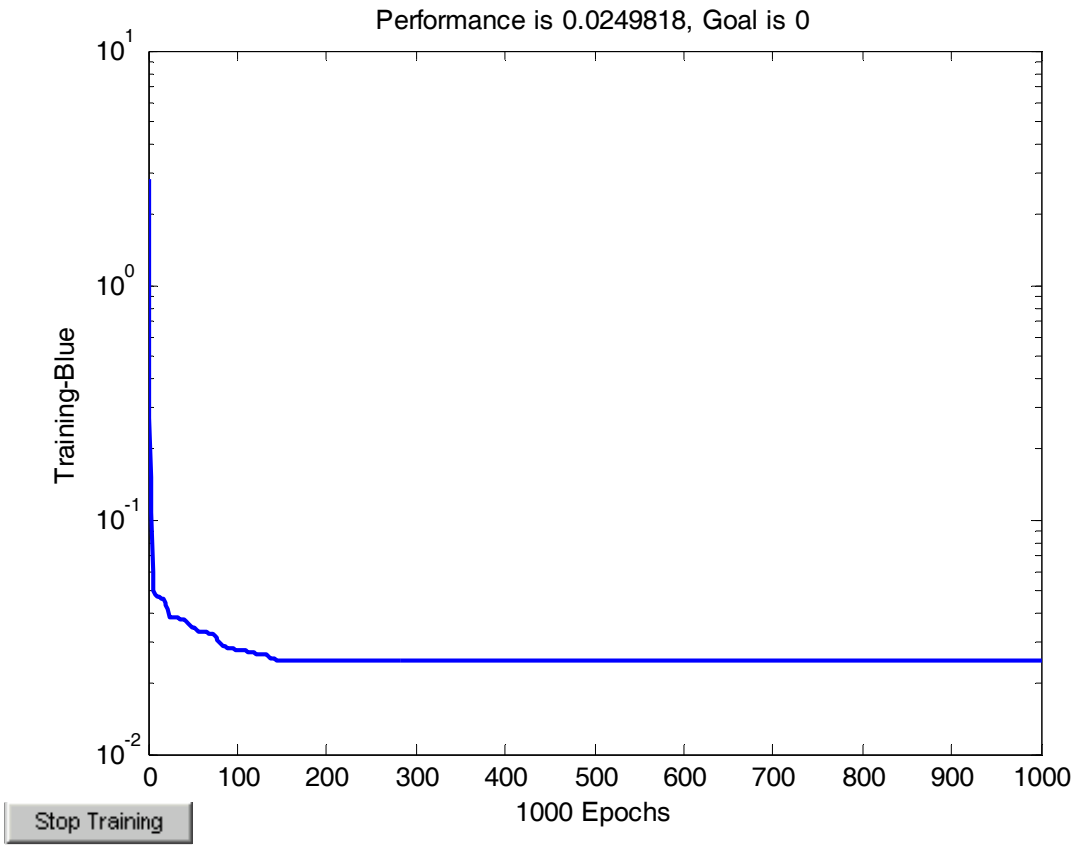


Fig.20 Network performance function during training

To generate a simulink model of the trained neural network, use the command *gensim* which is written as *gensim(network name, sampling time)*, choose a sample time of 1e-3 for good accuracy as follow:

>> gensim(net,1e-3)

You will get the following Fig.21. Copy and paste the blue NN model into a new simulink model to test it as a third step.

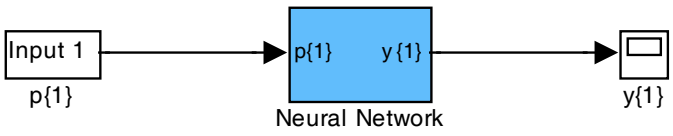


Fig.21 Generating simulink model of ANN

Step 3: Testing the trained neural network

In this step, the trained neural network will be tested using a testing input. Build a new Simulink model consisting of both plant model and NN model subjected to the same input as shown in Fig.22. You can use the training input as the test input or you can change its parameters. In this example, we will use the first option. Observe the plant output and the NN model output on the same scope. If your training is good enough you should find the two outputs very similar as shown in Fig.23.

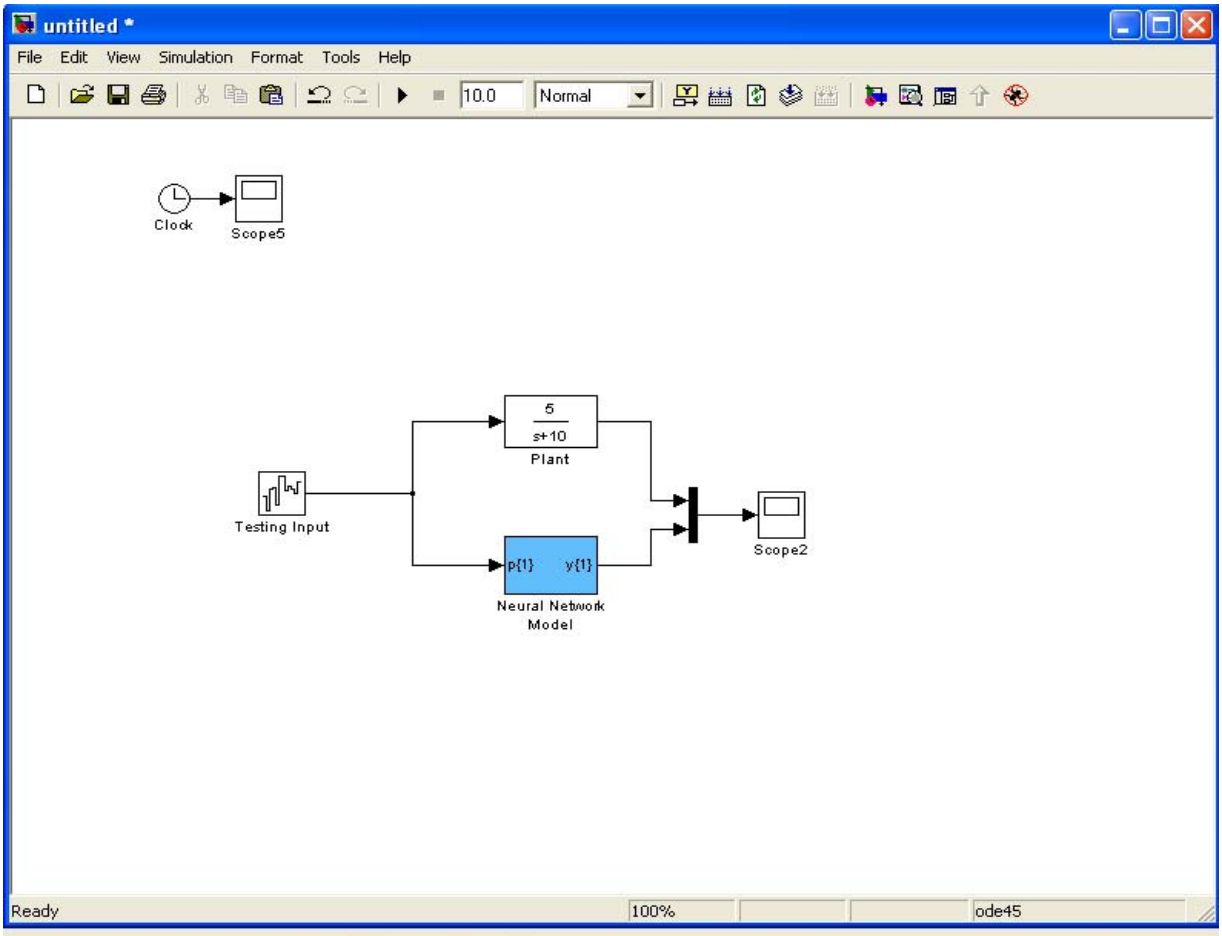


Fig.22 Simulink model of NN model testing

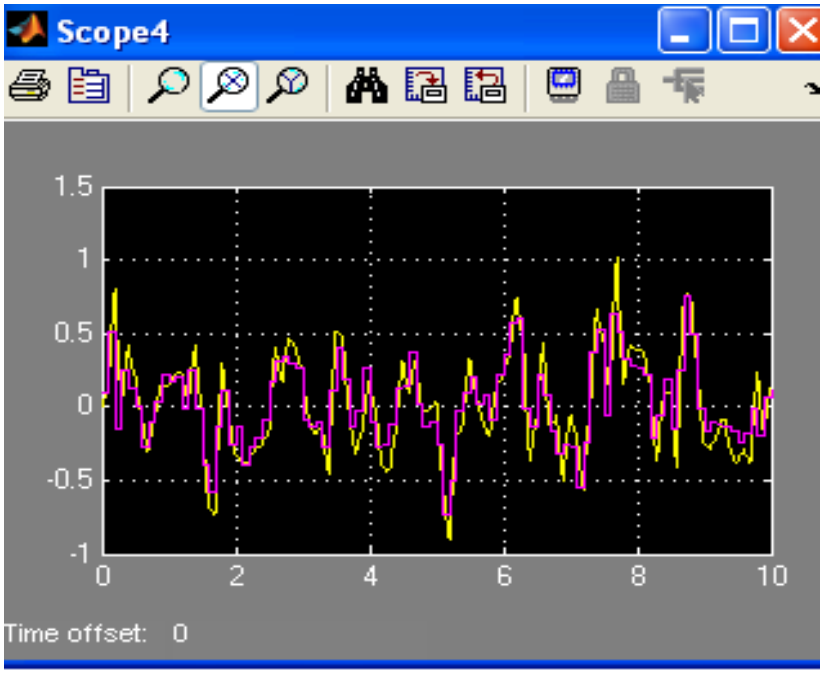


Fig.23 Actual plant and NN model outputs

Now, let's operate the plant with its normal inputs (usually step inputs) in presence of the controller. In this case we assume that the controller is already designed but normally controller design stage starts after obtaining the accurate NN model of the plant and it will be designed based on certain response specifications. Build another simulink model as shown in Fig.24. The model consists of 2 identical control systems one controls the actual plant and the other controls the plant NN model. Set the PI controller gains to: k_p= 10; k_i=50 in both schemes. Now subject both control systems to a bi-directional step input which starts with 0 and steps to 1 after 1sec then steps to -1 after 2 sec. The input will be *a repeating sequence stair* block. You should configure the simulation parameters as in the beginning of the example and change the simulation time to 3sec. The block parameters of the input and the PI controller are shown in Figs.25-26. After you run your simulation open the scope with three inputs: input, real system output, NN model system output and as expected the two outputs are quite similar as shown in Fig.27. Better accuracy can be obtained by fitting a more accurate NN model depending on the choice of number of hidden layers, number of neurons in each hidden layer, type of activation function. The network with the least performance function after the end of training will be the best choice.

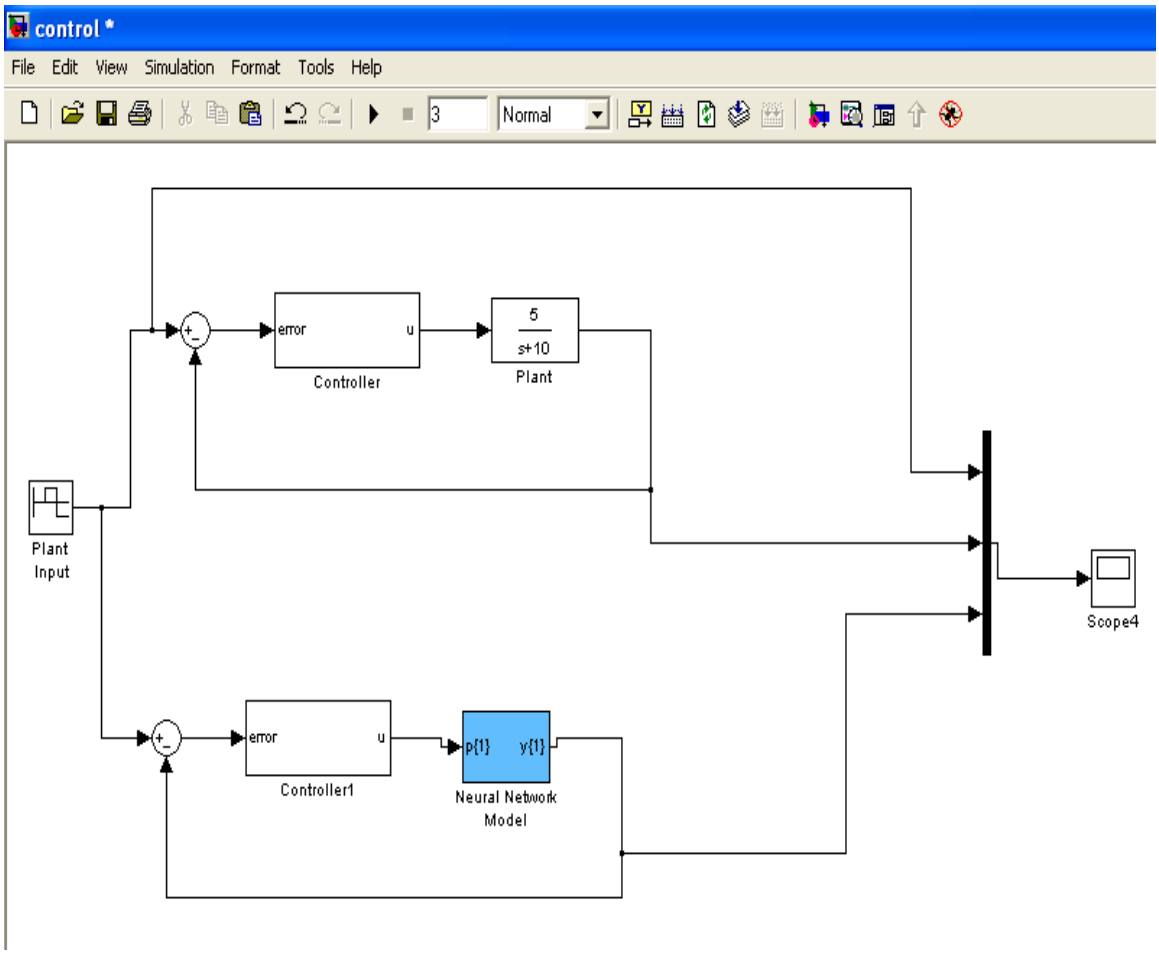


Fig.24 Simulink model of control system

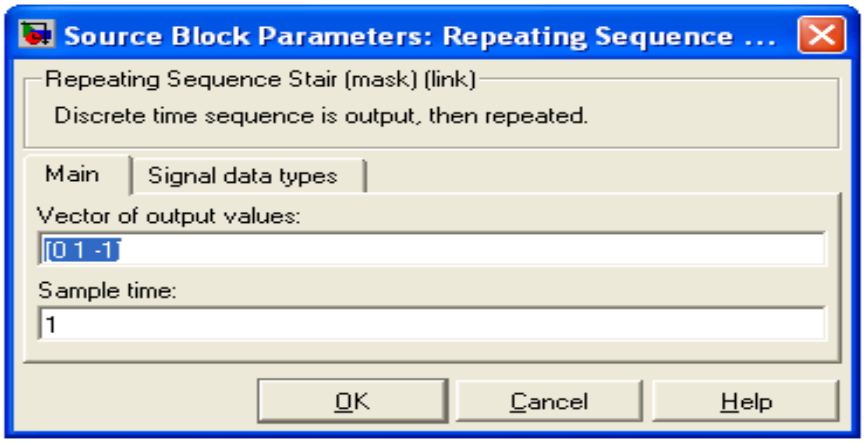


Fig.25 Input block parameters

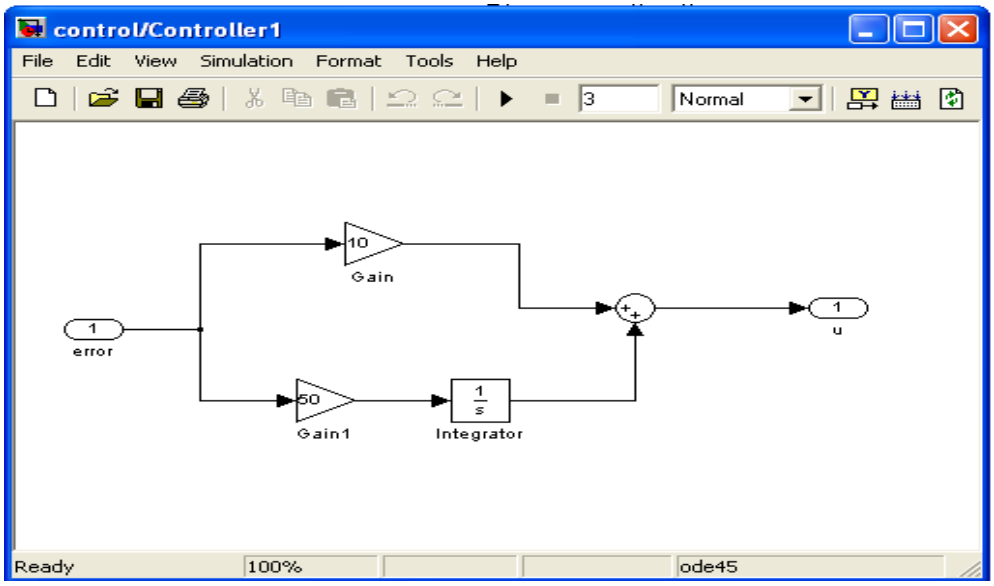


Fig.26 PI controller Simulink model

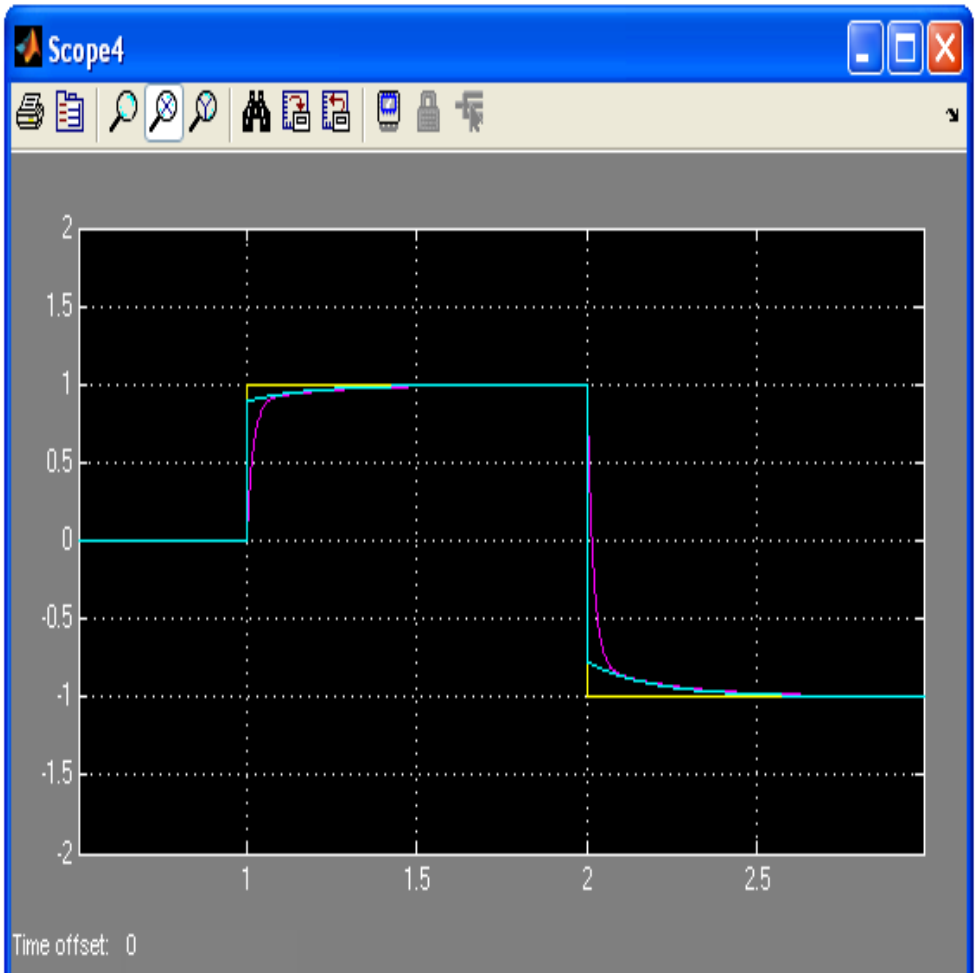


Fig.27 Actual plant and NN model response to step input

References:

[1] Dan W. Patterson (1995) *Artificial Neural networks Theory and Applications*. Singapore; New York: Prentice Hall.

[2] Kishan Mehrotra, Chilukuri K. Mohan, Sanjay Ranka (1997) *Elements of artificial neural networks*. Cambridge, Mass.: MIT Press.

[3] The Mathworks Neural Network Toolbox user guide. Available on line on:
http://www.mathworks.com/access/helpdesk/help/pdf_doc/nnet/nnet.pdf