Appendix

B

MATLAB Aided Control System Design: Intelligent Control

An introduction to knowledge-based tools—neural networks, fuzzy logic, genetic algorithms, for control system design was given in Chapters 11 and 12. This appendix is written to familiarize the reader with the use of **Neural Network Toolbox**, **Fuzzy Logic Toolbox**, and **Genetic Algorithm and Direct Search Toolbox** associated with MATLAB.

NEURAL NETWORK TOOLBOX

The Neural Network Toolbox (Version 4.0.3) is contained in a directory called **nnet**. Type **help nnet** for a listing of help topics.

The Neural Network Toolbox is a collection of functions built on the MATLAB numeric computing environment. It provides tools to create and edit neural network models, within the framework of MATLAB; or, if we prefer, we can integrate our network models into simulations with Simulink. This toolbox relies heavily on graphical user interface (**GUI**) tools to help us accomplish our work, although we can work entirely from the command line if we prefer.

The toolbox provides three categories of tools:

- Command line functions
- Graphical interactive tools
- Simulink blocks

The first category of tools is made up of functions (**M-files**) that we can call from the command line. These functions are used to create, initialize, train and simulate neural networks. We can also extend the toolbox by adding our own M-files.

Creating a Network

The function **newff** creates a feedforward backpropagation network. It requires four inputs and returns the network object **net**. The first input is an $n \times 2$ matrix of maximum and minimum values for each of the n elements of the input vector. The second input is an array containing the sizes of each layer. The third input is a cell array containing the names of the transfer functions to be used in each layer, e.g., **logsig, tansig, purelin,** ..... The final input contains the name of the training function to be used, e.g.,

traingd : Gradient descent backpropagation

traingda : Gradient descent with adaptive learning rate backpropagation

traingdm : Gradient descent with momentum backpropagation

traingdx : Gradient descent with momentum and adaptive learning rate backpropagation

⋮

For example, the following command creates a two-layer network. There is one input vector with two elements. The values for the first element of the input vector range between –1 and 2, and the values of the second element range between 0 and 5. There are three neurons in the first layer, and one neuron in the second (output) layer. The transfer function in the first layer is tan-sigmoid, and the output layer transfer function is linear. The training function is gradient descent backpropagation.

net = newff ([–1 2; 0 5], [3, 1], {'tansig', 'purelin'}, 'traingd');

Initializing Weights

The **newff** command will automatically initialize the weights and biases. Re-initialization may be done using the command **init**.

The network's weights and biases are referenced as follows to see how they have been initialized.

net.IW{*i*, *j*}: The weight matrix for the weights going to the *i*th layer (destination), from the *j*th input vector (source). The $i = j = 1$ situation will be frequently used in our applications.

net.LW{*i*,*j*}: The weight matrix for the weights going to the *i*th layer (destination), from the *j*th layer (source).

net.b{*i*}: The bias vector for the *i*th layer.

Training

The batch steepest descent training function is **traingd**. There are several training parameters associated with **traingd**. If we want to use the default values of these parameters, no additional commands are necessary. We might want to modify some of the default training parameters. This is done as is shown below by an example.

net.trainParam.show = 50;

net.trainParam.lr = 0.005;

net.trainParam.epochs = 300;

net.trainParam.goal = 1e–5;

The training status is displayed for every **show** iteration of the algorithm. **lr** is the specified learning rate. The other parameters determine when the training stops. The training stops if the number of iterations exceeds **epochs**, or if the performance function drops below **goal**. The default performance function for the feedforward networks is mean square error **mse**—the average squared error between the network outputs and the target outputs.

Suppose that the network training set data consists of 3 concurrent 2-dimensional input vectors:

$$\mathbf{x}^{(1)} = \begin{pmatrix} x_1^{(1)} \\ x_2^{(1)} \end{pmatrix}; \mathbf{x}^{(2)} = \begin{pmatrix} x_1^{(2)} \\ x_2^{(2)} \end{pmatrix}; \mathbf{x}^{(3)} = \begin{pmatrix} x_1^{(3)} \\ x_2^{(3)} \end{pmatrix}$$

and the corresponding targets are

$$y^{(1)}, y^{(2)}, y^{(3)}$$

The concurrent input vectors are presented to the network as a single 2 × 3 matrix; *i*th row of this matrix is made up of *i*th element of each vector.

$$\mathbf{X} : \left[x_1^{(1)} x_1^{(2)} x_1^{(3)}; x_2^{(1)} x_2^{(2)} x_2^{(3)}\right]$$

The concurrent targets are presented to the network as a single 1 × 3 vector.

$$\mathbf{y}: [y^{(1)} \quad y^{(2)} \quad y^{(3)}]$$

To perform batch training using the function **train**, the following command may be used:

[net, tr] = train(net, X, y)

The training record **tr** contains information about the progress of training.

The function **sim** simulates a network. It takes the network input **X** and the network object **net**, and returns the network outputs $\hat{\mathbf{y}}$.

$$\hat{\mathbf{y}} = \mathbf{sim(net, X)}$$

A single matrix of concurrent vectors is presented to the network, and the network produces a single matrix of concurrent vectors as output.

In the *batch mode*, the weights and biases of the network are updated only after the entire training set has been applied to the network. In the *incremental mode*, weights and biases are updated after each input is applied to the network. In this case, we use the function **adapt** and we present the inputs and targets as sequences.

The concurrent inputs $\mathbf{x}^{(1)}$, $\mathbf{x}^{(2)}$, $\mathbf{x}^{(3)}$ and targets $y^{(1)}$, $y^{(2)}$, $y^{(3)}$ are presented as a cell array of sequential vectors.

$$\mathbf{x}_s\text{: } \left\{\left[x_1^{(1)}; x_2^{(2)}\right]\left[x_1^{(2)}; x_2^{(2)}\right]\left[x_1^{(3)}; x_2^{(3)}\right]\right\}$$
$$\mathbf{y}_s\text{: } \{y^{(1)}\ y^{(2)}\ y^{(3)}\}$$

[net, netOutput, netError] = adapt(net, $\mathbf{x}_s$, $\mathbf{y}_s$)

Graphical User Interface

The second category of tools is made up of a number of interactive tools that let us access many of the **M-files** through a **GUI**. This interface allows us to

- Create networks
- Enter data into the **GUI**
- Initialize, train and simulate networks
- Export the training results from the **GUI** to the command line workspace
- Import data from the command line workspace to the **GUI**

To open the **Network/Data Manager** window, type **nntool**. Click on **Help** to get started on a new problem and to see descriptions of the buttons and lists.

Simulink Simulation

The third category of tools is a set of blocks for use with the Simulink simulation software. Bring up the Neural Network Toolbox blockset with the command **neural**. The result is a window that contains four blocks: Transfer Functions, Net Input Functions, Weight Functions, and Control Systems.

Each of the blocks of the **Transfer Functions** takes a net input vector and generates a corresponding output vector whose dimensions are the same as the input vector (Double-click on the **Transfer Functions** block). Each of the blocks of **Net Input Functions** takes any number of weighted input vectors, weighted layer output vectors, and bias vectors and returns a net-input vector. Each of the blocks of **Weight Functions** takes a neuron's weight vector and applies it to an input vector (or a layer output vector), to get a weighted input value for a neuron.

We can extend the blockset by adding our own S-functions. The blockset is used to build neural networks in Simulink. The function **gensim** generates the Simulink version of any network we have created in MATLAB. The call

gensim(net, T)

results in a screen that contains a Simulink system consisting of the network object **net**, connected to an input block (a standard '**Constant**' block from the Simulink blockset: '**Sources**') and a **Scope**. ***T*** is the discrete sample time. ***T*** = –1 tells **gensim** to generate a network with continuous sampling. To build a feedback control system with **net** as one of the components, we can replace the '**Constant**' input block with a signal generator from the

Simulink '**Sources**' blockset, and use other blocks from the Simulink blockset/Neural Network Toolbox blockset/user-created S-function blocks.

Double-click on the **Control Systems** block in the **neural** window. This brings up the following three blocks:

- **Model Reference Controller**
- **NARMA-L2 Controller**
- **NN Predictive Controller**

The neural model reference control architecture uses two neural networks: a controller network and a plant model network. The plant model is identified first, and then the **Model Reference Controller** is trained so that the plant output follows the reference model output.

The neurocontroller, called NARMA-L2 Controller, is also described by the name, **Feedback Linearization Controller**. The central idea of this type of control is to transform nonlinear system dynamics into linear dynamics by cancelling the nonlinearities.

The **NN Predictive Controller** uses a neural network model of a nonlinear plant to predict future plant performance. The controller then calculates the control input that will optimize plant performance over a specified future time horizon.

These three controllers are implemented as Simulink blocks. A controller block from the Neural Network Toolbox blockset is copied to Simulink model window. Double-clicking on the controller block brings up a window for designing the control. Performance of a closed-loop control system is evaluated by running its Simulink model.

FUZZY LOGIC TOOLBOX

The Fuzzy Logic Toolbox (Version 2.1.3) is contained in a directory called **fuzzy**. Type **help fuzzy** for a listing of help topics.

The Fuzzy Logic Toolbox provides tools to create and edit fuzzy inference system (**FIS**) within the framework of MATLAB. We can integrate our fuzzy systems into simulations with Simulink.

Similar to the Neural Network Toolbox, the Fuzzy Logic Toolbox also provides three categories of tools:

- Command line functions
- Graphical interactive tools
- Simulink blocks

Building a Fuzzy Inference System

To build a system entirely from the command line, the commands **newfis**, **addvar**, **addmf** and **addrule** would be used.

The function **newfis** creates new **FIS** structures. It has upto seven input arguments, and the output argument is a **FIS** structure. The seven input arguments are as follows:

- **fisName** is the string name of **FIS** structure; **fisName**.**fis** you create.
- **fisType** is the type of FIS; Mamdani type is default.
- **andMethod, orMethod, impMethod, aggMethod,** and **defuzzMethod**, respectively, provide the methods for AND, OR, implication, aggregation, and defuzzification. The defaults are, respectively, min, max, min, max and centroid (centre of area).

The function **addvar** adds a variable to a **FIS**. It has four arguments in this order:

- The name of the **FIS** structure in the MATLAB workspace.
- The string representing the type of the variable we want to add ('**input**' or '**output**').
- The string representing the name of the variable we want to add.
- The vector describing the limiting range values (universe of discourse), for the variable we want to add.

```
a = newfis('servo')
a = addvar(a, 'varType', 'varName', varBounds)
```

Indices are applied to variables in the order in which they are added; so the first input variable added to a system will always be known as input variable number one for that system. Input and output variables are numbered independently.

The Fuzzy Logic Toolbox includes many membership function types. The simplest membership functions are formed using straight lines. Of these, the simplest is the *triangular* membership function, and it has the function name **trimf**. The triangular curve is a function, of three scalar parameters a, b, and c; the parameters a and c locate the "feet" of the triangle and the parameter b locates the peak. The *trapezoidal* membership function, **trapmf**, has a flat top. The trapezoidal curve depends on four scalar parameters a, b, c, and d; the parameters a and d locate the "feet" of the trapezoid and the parameters b and c locate the "shoulders".

Each input/output variable existing in MATLAB workspace (**FIS** variables added by the function **addvar**), is resolved into a number of different fuzzy linguistic sets. The input/output variables must be fuzzified according to each of these linguistic sets.

A membership function can only be added to a variable in an existing MATLAB workspace **FIS**. Indices are assigned to membership functions in the order in which they are added; so the first membership function added to a variable, will always be known as membership function number one for that variable. The function **addmf** adds a membership function to **FIS**. The function requires six input arguments in this order:

- A MATLAB variable name of a FIS structure in the workspace.
- A string representing the type of variable we want to add the membership function to ('**input**' or '**output**').
- The index of the variable you want to add the membership function to (We cannot add a membership function to input variable number two of a system if only one input has been defined).
- A string representing the name of the new membership function.
- A string representing the type of the new membership function.
- The vector of parameters that specify the membership function.

```
a = newfis ('servo')
a = addvar (a, 'varType', 'varName', varBounds)
a = addmf (a, 'varType', varIndex, 'mfName', 'mfType', mfParams)
```

Probably the trickiest part of the process of building a Fuzzy System, is learning the 'short hand' that the fuzzy inference systems use for building rules. This is accomplished using the command line function **addrule**.

Each variable, input or output, has an index number, and each membership function has an index number. The rules are built from statements like this:

IF input 1 is MF1 **and** input 2 is MF3 THEN Output is MF2

This rule is turned into a structure according to the following logic. If there are p inputs to a system and q outputs, then the first p vector entries of the rule structure correspond to inputs 1 through p. The entry in column 1 is the index number for the membership function associated with input 1. The entry in column 2 is the index number for the membership function associated with input 2, and so on. The next q columns work the same way for the outputs. Column $p + q + 1$ is the weight associated with the rule (typically 1; all the rules have equal weightage), and column $p + q + 2$ specifies the connective used (where **and** = 1 and **or** = 2). The structure associated with the rule shown above, is

1 3 2 1 1

The function **addrule** has two arguments. The first argument is the MATLAB workspace variable **FIS** name. The second argument is a matrix of one or more rows, each of which represents a given rule. The rule-list matrix takes the very specific format defined above.

```
ruleList = [ 1   1   1   1   1
             1   2   2   1   1 ];
a = addrule(a, ruleList);
```

If the above system has two inputs and one output, the first rule can be interpreted as

"IF input 1 is MF1 **and** input 2 is MF1 THEN output 1 is MF1".

To evaluate the output of a fuzzy system for a given input, we use the function **evalfis**. It has the following arguments:

- A number or a matrix specifying the input values. If input is a $P \times p$ matrix, where p is the number of input variables, then **evalfis** takes each of the P rows of input as an input vector and returns the $P \times q$ matrix to the variable, output, where each row is an output vector and q is the number of output variables.
- The name of the **FIS** structure to be evaluated.

Gaphical User Interface

It is possible to use the Fuzzy Logic Toolbox by working strictly from the command line. However, in general, it is much easier to build a system graphically. There are five primary **GUI** tools for building, editing and observing fuzzy inference systems in the Fuzzy Logic Toolbox: the Fuzzy Inference System or FIS Editor, the Membership Function Editor, the Rule Editor, the Rule Viewer and the Surface Viewer.

The **FIS Editor** handles the high level issues for the system: How many input and output variables? What are their names?...

The **Membership Function Editor** is used to define the shapes of all the membership functions associated with each variable.

The **Rule Editor** is for editing the list of rules that defines the behaviour of the system.

The **Rule Viewer** is a display of the fuzzy inference diagram.

The **Surface Viewer** is used to display the dependency of one of the outputs on any one or two of the inputs.

The five primary GUIs can all interact and exchange information. For any fuzzy inference system, any or all of these five GUIs may be open. If more than one of these editors is open for a single system, the various GUI windows are aware of the existence of others and will, if necessary, update related windows. Thus, if the names of the membership functions are changed using the Membership Function Editor, these changes are reflected in the rules shown in the Rule Editor.

To start building a fuzzy inference system, type **fuzzy** at the MATLAB prompt. The generic untitled **FIS Editor** opens. At the top is a diagram of the system, with input and output clearly labelled. By double-clicking on the input or output boxes, you can bring up the **Membership Function Editor**. Double-clicking on the **fuzzy rule box** in the centre of the diagram will bring up the **Rule Editor**.

Just below the diagram is a text field that displays the name of the current **FIS**. Lower left of window has a series of pop up menus, and the lower right has fields that provide information about the current variable.

GUI Editors:

fuzzy – Basic FIS Editor

mfedit – Membership Function Editor

ruleedit – Rule Editor

ruleview – Rule Viewer

surfview – Output Surface Viewer

When you save your fuzzy system to the MATLAB workspace, you are creating a variable (whose name you choose) that will act as a MATLAB structure for the FIS system.

Simulink Simulation

Once you have created your fuzzy system entirely from the command line, or using the **GUI** tools, you are ready to embed your system directly into Simulink and test it out in a simulation environment. The **Fuzzy Logic Toolbox** in the Simulink library contains the **Fuzzy Logic Controller**, and the Fuzzy Logic Controller with

Rule Viewer blocks. It also includes a **Membership Functions** sub-library that contains Simulink blocks for the built-in membership functions. The Fuzzy Logic Controller with Rule Viewer block is an extension of the Fuzzy Logic Controller block. It allows you to visualize how rules are fired during simulation.

To start building a Simulink Fuzzy Model, drag the **Fuzzy Logic Controller block** (with or without the **Rule Viewer**) from the Simulink library to the simulation window (This can also be done by typing **fuzblock** at the MATLAB prompt). To initiate the Fuzzy Logic Controller block, double-click on the block and enter the name of the structure variable describing your **FIS**. This variable must be located in the MATLAB workspace. In most cases, the Fuzzy Logic Controller block automatically generates a hierarchical block diagram representation of your **FIS**. The block diagram representation only uses built-in Simulink blocks. This automatic, model-generation ability is called the **Fuzzy Wizard**. In cases where Fuzzy Wizard cannot handle **FIS**, the Fuzzy Logic Controller block uses the S-function **sffis** to-simulate the **FIS**.

Genetic Algorithm and Direct Search Toolbox (Version 2.0.2) will be used in Fuzzy-Genetic solutions.

Problems

Each problem covers an important aspect of neural network/fuzzy logic/genetic algorithm/neuro-fuzzy system applications to function approximation, system identification, and control (motion control, and process control). MATLAB commands are given at the **URL: http://www.mhhe.com/gopal/dc3e** as help to these problems. Open these files in MATLAB environment. The description of the MATLAB functions used in the script files can easily be accessed from the **help file** using **help** command. Comments to the special features of commands are included in the script files to enhance the learning experience. To accelerate the learning process, brief descriptions are included with some problems.

Simulink files are included as help to some problems. Download the Simulink files from the URL. Open these files in MATLAB environment. Double-click and study the properties of each block.

Following each problem, one or more *what-if's* may be posed to examine the effects of variations of the key parameters.

Function Approximation(NN)

B.1 It is desired to design a 2-layer feedforward NN to approximate the function $y = f(x)$:

x	–1:0.1:1
y	–0.960, –0.577, –0.073, 0.377, 0.641, 0.660, 0.461, 0.134, –0.201, –0.434, –0.500, –0.393, –0.165, 0.099, 0.307, 0.396, 0.345, 0.182, –0.031, –0.219, –0.320

The hidden layer with five neurons has hyperbolic activation functions, and the output layer is linear.

(a) Using batch gradient descent with momentum, train the network so that mean square error (mse) < 0.005. The learning rate $\eta = 0.01$. Repeat training three times with different initial weights. What is the conclusion of the experiment?

(b) Train the network using (i) batch gradient descent (ii) batch gradient descent with momentum, (iii) batch gradient descent with adaptive learning, and (iv) batch gradient descent with momentum and adaptive learning, and compare the convergence rates.

(c) Choose the fastest of the tested training algorithms. Train a network with 20 hidden units and compare its performance with one having 5 hidden units.

B.2 One of the most common applications of the multilayer neural networks trained with backpropagation is the approximation of nonlinear functional mappings. Write a MATLAB program and design a neural network with one hidden layer, and train by backpropagation to perform the following mapping:

$$y = e^{-x} \sin(3x)$$

in the interval [0, 4].

Do the following :

(a) Generate two independent sets of input patterns.
Training set: 21 patterns; sample the interval [0,4] with 21 points separated by 0.2.
Testing set: 401 patterns; sample the interval [0,4] with 401 points separated by 0.01.
For each of the sets, generate the target values using the analytical expression for the function to be approximated.

(b) Experiment with 50 neurons in hidden layer. Use MATLAB routine trainlm to perform the network training with hyperbolic tangent activation functions, and a target mean square error of 0.01 over the entire training data set. Check if the desired and actual network outputs for the training data set, are in agreement.

(c) Test the network's generalization capability using testing data set. If the response of the network does not show good agreement with the function you are trying to approximate, the reason could be *overfitting* of the training data. Try using considerably smaller number of neurons in hidden layer to see if the network would perform approximation task in a better way.

B.3 An RBF network is to be used to approximate the nonlinear function

$$y = e^{-x} \sin(3x)$$

in the interval [0,4].

Sample the interval with 21 points separated by 0.2 to generate training data set. Use the MATLAB function **newrbe** to train the network. The number of hidden units is 21, and the centres correspond to the input training vectors. Set the spread parameter of the Gaussian RBF used in **newrbe** to 0.2. Plot the response of the trained network to the training patterns.

To test the generalization capability of the network, sample the interval [0,4] with 401 points separated by 0.01 and generate the test data set. Plot the network response to this test data set and comment on the performance of the network

System Identification(NN)

B.4 The nonlinear system to be identified is expressed by

$$y(k+1) = \frac{y(k)[y(k-1)+2][y(k)+2.5]}{8.5 + [y(k)]^2 + [y(k-1)]^2} + u(k)$$

where $y(k)$ is the output of the system at the kth time step and $u(k)$ is the input, which is a uniformly bounded function of time. The system is stable at $u(k) \in [-2, 2]$. Let the identification model be of the form

$$\hat{y}(k+1) = f(y(k), y(k-1)) + u(k)$$

where $f(y(k), y(k-1))$ represents the backpropagation network. The goal here is to train the network.

(a) Use the randomly created data from the input space [–2, 2] to obtain training patterns.

(b) Build the initial backpropagation network with 10 nodes **(tansig)** in the hidden layer and one **(linear)** output node.

(c) With batch training procedure, train the network to achieve mean square error (mse) < 0.005. The learning rate $\eta = 0.1$.

(d) After the backpropagation network is trained, test its prediction power for the input

$$u(k) = 2\cos(2\pi k/100),\ k \le 200$$

and

$$u(k) = 1.2\sin(2\pi k/20),\ 200 < k \le 500$$

(e) Iterate on number of hidden layers, number of neurons in each layer, activation functions, mse goal, learning rate, and training algorithm, if the learned network could not predict the nonlinear output quite well.

B.5 Consider a nonlinear system given by the following state variable model:

$$x_1(k+1) = \frac{x_2(k)}{x_1(k)+4}$$

$$x_2(k+1) = \tanh(x_1(k) + \{1 + x_2(k)\}\, u(k))$$

$$y(k) = 2x_2(k)$$

where $y(k)$ is the output of the system at the kth time step, $u(k)$ is the input (uniformly bounded function of time), and $\{x_1(k), x_2(k)\}$ are state variables. The system is assumed to be BIBO stable.

Let the identification model be of the form

$$\hat{y}(k) = f(y(k-1), ..., y(k-4)), u(k-1)..., u(k-4))$$

Design a neural network identifier with two hidden layers and backpropagation training.

Do the following:

(a) Use uniformly distributed noise sequence in the interval [–0.5,0.5] to obtain training patterns. Take 2 neurons in first hidden layer and 1 neuron in the second, and train the network.

(b) Plot the outputs of the system and the neural network model, when tested with a square wave input of peak amplitude 0.2. Comment on the test result.

(c) The input in (b) is within the range of the random noise used for training of the network. Take a peak amplitude of the input waveform that lies outside the range of input patterns used in the development of the neural network model (say, 2). Test the performance of the network and comment on the result.

B.6 Consider the same dynamic system given in Problem **B.5**. Use an RBF network to accomplish the task of nonlinear mapping in the identifier. Take 30 neurons in the hidden layer, and *Gaussian radial basis function* with spread parameter set to 1. Test the identifier performance using the square wave inputs of peak amplitudes 0.2 and 2, respectively. Comment on the test results.

B.7 Consider the identification of the following linear SISO second-order dynamic system:

$$Y(s) = \frac{3}{s^2 + s + 3}U(s)$$

With the sampling rate $T = 0.25$ sec, we obtain the discrete-time equation

$$y(k) = 1.615\, y(k-1) - 0.7799\, y(k-2) + 0.08508\, u(k-1) + 0.07824\, u(k-2)$$

Design a linear neuron for estimating the parameters a_i and b_i of the following model for the given system:

$$y(k) = a_1\, y(k-1) + a_2\, y(k-2) + b_1 u(k-1) + b_2 u(k-2) + \varepsilon(k)$$

where $\varepsilon(k)$ is the additive white noise.

The training input u is a pseudo-random-binary-signal (PRBS) and the output of the system is corrupted by Gaussian white noise with SNR (signal-to-noise ratio) 25 dB.

(a) Using a set of 50 input-output data pairs, estimate the parameters a_i and b_i. Test the network on previously unseen input (say, a unit-step).

(b) Use a larger data set, say 500 input-output data pairs. Test the network on unit-step input. Comment on the effect of increasing the training data set on the performance of the network.

B.8 *AdaptiveIdentifcation* Given the input signal

$$u(k) = 0.6 \sin(2\pi k/10) + 1.2 \cos(2\pi k/10);\ k \in \{1, 2, \ldots\}$$

This input is given to the linear system

$$y(k) = u(k) + 0.5\ u(k-1) - 1.5\ u(k-2)$$

Design a neural network that predicts the system output y, given the current and the previous two input signals u.

Parameters of the linear system are not robust. To take care of this problem, we go for adaptive identification. Train the network incrementally, assuming that the measurements are y_m:

randn('state', 0)

y_m = y + 0.1*randn(size (y))

('**randn**' is a MATLAB command for random number generation).

Control(NN)

B.9 Consider the speed control system of Fig. 11.33 (Review Example 11.1).

(a) Using randomly generated inputs and the corresponding targets, train the NN to learn inverse dynamics of the plant. Evaluate the performance of the trained NN identifier, by executing the dc motor model with the voltage

$$v_a(k) = 50 \sin(2\pi T/7) + 45 \sin(2\pi kT/3);\ \forall kT \in [0, 20]$$

and comparing it with the estimated values given by NN identifier. What is the maximum prediction error?

(b) The tracking capability of the NN-based controller of Fig. 11.33, can be investigated for different arbitrarily specified trajectories. Simulate the performance of the controller in the configuration of Fig. 11.33, for an arbitrarily selected speed track

$$\omega_r(k) = 10 \sin(2\pi kT/4) + 16 \sin(2\pi kT/7);\ \forall kT \in [0, 20]$$

(c) Using MATLAB's Simulink, simulate the feedback system of Fig. 11.33 and obtain the response for the given command.

B.10 Consider the temperature control system of Fig. 11.34 (Review Example 11.2).

(a) Using a train of pulses as the input and the corresponding targets, train the NN to learn inverse dynamics of the plant. Evaluate the performance of the trained NN identifier on the training data set.

(b) From the initial condition $y(0) = Y_0$, the target is to follow a control reference, set to 35°C for $0 \le t \le 30$ samples, 55°C for $30 < t \le 60$ samples, and 75° C for $60 < t \le 90$ samples. Simulate the performance of the controller in the configuration of Fig. 11.33 for this temperature track.

(c) Using MATLAB's Simulink, simulate the feedback system in the configuration of Fig. 11.33 and obtain the response for the given command.

B.11 Consider a discrete-time nonlinear system specified by the following difference equation:

$$y(k+1) = K_5\ u(k) + K_1\ y(k) + K_2\ y(k-1) + K_3 \text{ sign } (y(k))\ y^2(k) + K_4 \text{ sign } (y(k))\ y^2(k-1) + K_6$$

where the constants are

$K_1 = 0.34366$; $K_2 = -0.1534069$; $K_3 = -2.286928\text{e}{-3}$; $K_4 = 3.5193358\text{e}{-4}$; $K_5 = 0.2280595$; $K_6 = -0.105185$.

The sampling interval $T = 0.25$ sec.

A reference model specifying the desired behaviour of the system, is given by the transfer function

$$Y_m(s)/R(s) = 42/(s^2 + 11\ s + 30)$$

where y_m is the reference model output and r is the input.

We need to design a neural network controller so that the overall behaviour of the plant can be described by the discrete-time reference model equation, obtained by discretization of the given reference model transfer function with sampling $T = 0.25$ sec. The structure of the neural network controller is given in Fig.11.25.

Design the controller using MLP neural networks.

Fuzzy/Fuzzy-Genetic Models

B.12 Write a set of functions built on the MATLAB numerical computing environment to implement the simple Genetic Algorithm described in Chapter 12. Using these MATLAB functions, solve Problem 12.22.

Help: The code in Script PB.12 implements a simple Genetic Algorithm, constructed with the sole intent of clarifying the issues involved; generality, thus, is not the objective.

The MATLAB function **genetic** extremizes variables (within the limits imposed by min and max) as per fitness measure provided by **objfunc.m.**

The following notes will be helpful in appreciating the development of the program in Script PB. 12.

Note 1: Each population string encodes *N* parameters of the objective function. Number of bits for coding *j*th parameter is *nbit* (*j*). The decimal value of this parameter varies between 0 and 2^*nbit* (*j*) – 1. *inivar* (*i*, *j*) is a random decimal integer value between these limits. This way, we create a *popsize* × *N* initial-variable matrix whose elements are random decimal integers.

Note 2: The following example illustrates the conversion process from decimal to binary. Suppose the decimal value is 13.

2	13	
2	6	1
2	3	0
2	1	1
	0	1

Binary bits: 1101

Note 3: Actual parameter values are calculated as follows.

$$\text{Parameter value} = \min + \frac{\max - \min}{2^{nbit} - 1} \text{ (decimal value)}$$

Note 4: *randperm (popsize)* will return the shuffling of numbers from 1 to *popsize* and store them in *mplocation*. For example, if *popsize* = 4, we may get

mplocation = [1 4 3 2]

rand is a random number between 0 and 1. It is similar to a value we will get after a spin of wheel. Say

csum = [0.125 0.25 0.6 0.9 1], and *rand* = 0.5.

rand < = *csum*(3); the selected *j*th string (3rd in the example) is placed at random location in *mpool*.

Note 5: Since the mating pool is generated randomly, we pick the pairs of strings from the top of the list.

Note 6: Consider the *j*th parameter. Number of bits = *nbit*(*j*). Consider string *i*. There are *lchrom* bits in this string. Suppose bits *b*l to *bn* on the string correspond to this parameter.

A bit at *n*th location has the decimal weight 2^{n-1}.

B.13 A set of functions built on the MATLAB numerical computing environment to implement the simple Genetic Algorithm are described in **Genetic Algorithm and Direct Search Toolbox**. Using these MATLAB functions, solve Problem 12.22.

B.14 (a) Using Fuzzy Logic Toolbox (working strictly from the command line), realize the Fuzzy System specified in Problem 12.13. Determine the inferred value by the COA defuzzification method for the inputs $x_0 = 4$ and $y_0 = 8$.
(b) Build the fuzzy inference system (FIS) using GUI tools. Embed your system directly into Simulink and test it out in a simulation environment.

B.15 (a) Using Fuzzy Logic Toolbox (working strictly from the command line), realize the Fuzzy System specified in Review Example 12.1.

From the initial condition $y(0) = 0$, the target is to follow a control reference, set to 20° for $0 \le t \le 250$ samples, and 23° for $250 \le t \le 500$ samples. Simulate the performance of the Fuzzy Controller in configuration of Fig. 12.36 for this position track. Choose the scaling constants *GE*, *GV*, and *GU′* by trial-and-error.
(b) Build the fuzzy inference system (FIS) using GUI tools. Embed your system directly into Simulink and test it out in a simulation environment.

B.16 Using Fuzzy Logic Toolbox, realize the Fuzzy System specified in Review Example 12.2.

From the initial condition $y(0) = Y_0$, the target is to follow a control reference, set to 35° C for $0 \le t \le 360$ samples. Simulate the performance of the Fuzzy Controller for this temperature track.

(a) Choose the scaling constants *GE*, *GV* and *GU*' using genetic algorithm developed in Problem B.12.
(b) Choose the scaling constants *GE*, *GV* and *GU′* using genetic algorithm described in **Genetic Algorithm and Direct Search Toolbox.**

Neuro-Fuzzy Models

B.17 Consider the modelling Problem P12.19. From the given input range, generate 121 training data points. Use **anfis** with 16 rules (four membership functions assigned to each input variable), to determine the 24 premise and 48 consequent parameters. Plot the root mean square error curve. Compare your **anfis** with 2-18-1 backpropagation MPL.

B.18 Consider the modelling Problem P12.20. From the given ranges, generate 216 training data and 125 testing data. Use **anfis** with eight rules (two membership functions assigned to each input variable), to determine the premise and consequent parameters. Plot the root mean square error curve.

B.19 Consider the temperature control system of Fig. 11.34 (Review Example 11.2). For this system, implement the **anfis** controller using the inverse model of the system as a controller in the configuration of Fig. 11.33. Obtain the training data by imposing random input voltages to the water bath system, and recording the corresponding temperatures. The **anfis** is then trained to identify the inverse model of the water bath system, using the gathered training data. To start the **anfis** training, we need a FIS matrix that specifies the structure and initial parameters of the FIS for learning. You can use the command **genfis1** to generate a FIS matrix from training data, using a grid-type partition according to the given number and types of membership functions.

Embed your controller in the feedback system and carry out simulation using MATLAB's Simulink.