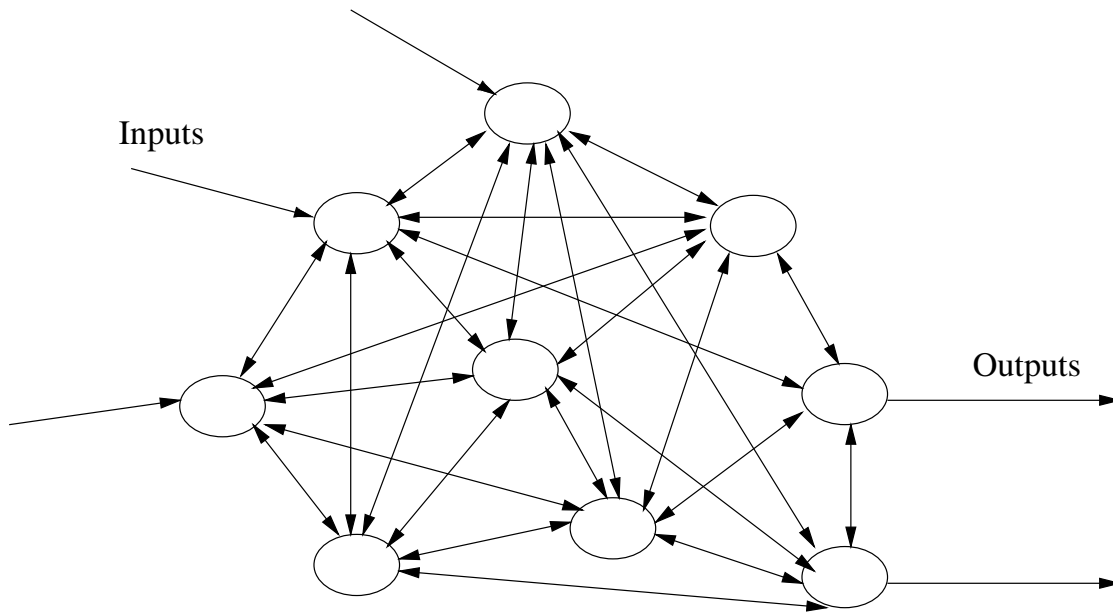


RANDOM NEURAL NETWORK SIMULATOR (RNNSIM v.2)

For Use with MATLAB

Hossam Eldin Abdelbaki



Technical Report
School of Computer Science
University of Central Florida

September 1999

Contents

1	Introduction	1
1.1	Release Notes	1
1.2	Installing the Simulator	3
1.3	RNNSIM v.2 Functions and Script Files	5
1.4	Contact Information	6
1.5	Copyright and Disclaimer	6
2	The Random Neural Network Model	8
2.1	Basics of the Random Neural Network Model	8
2.2	Learning in the Supervised Recurrent Random Neural Network Model	10
2.3	Applications of the Random Neural Network Model [4]	13
2.3.1	Optimization	14
2.3.2	Texture Image Generation	14
2.3.3	Image and Video Compression	15
3	Using the RNNSIM v.2	16
3.1	Example 1: Feed-Forward Structure	18
3.2	Example 2: Recurrent Structure	29
3.3	Conclusion	34
A	RNNSIM v.2 Functions	35
	Bibliography	43

Chapter 1

Introduction

This Chapter contains important information on how the present set of tools is to be installed and the conditions under which it may be used. The MATLAB m-files in this software distribution implement the recurrent learning algorithm of the random neural network (RNN) model [1, 2, 3]. Please read it carefully before use.

1.1 Release Notes

The simulator is provided in two versions, **RNNSIM v.1** and **RNNSIM v.2**. Both versions have been tested using MATLAB v.5 (student and professional editions) under Windows 95, Windows NT and Unix operating systems. The entire simulator is implemented as ordinary m-files and thus it should work equally well on all hardware platforms. The functions in this package work completely independent of the the MATLAB neural network toolbox. This technical report discusses the usage of the second version **RNNSIM v.2**.

RNNSIM v.1 program was designed to deal with three layers feed-forward random neural network structures. Although **RNNSIM v.1** has a friendly graphical use interface, it lacks the flexibility of choosing the network structure, exploring the intermediate values of the weights, testing the trained network with patterns that change with time, and dealing with image processing applications. Figure 1.1 shows the main window of **RNNSIM v.1**. The program has been posted into the public directory of MathWorks company in August 1998 and it can be downloaded from the following links as separate m-files

- <ftp://ftp.mathworks.com/pub/contrib/v5/nnet/rnnsim>
- <http://www.cs.ucf.edu/~ahossam/rnnsim/>

or as a compressed zip file form

- <ftp://ftp.mathworks.com/pub/contrib/v5/nnet/rnnsim.zip>
- <http://www.cs.ucf.edu/~ahossam/rnnsim/rnnsim.zip>

The program has a detailed help included into the interface.

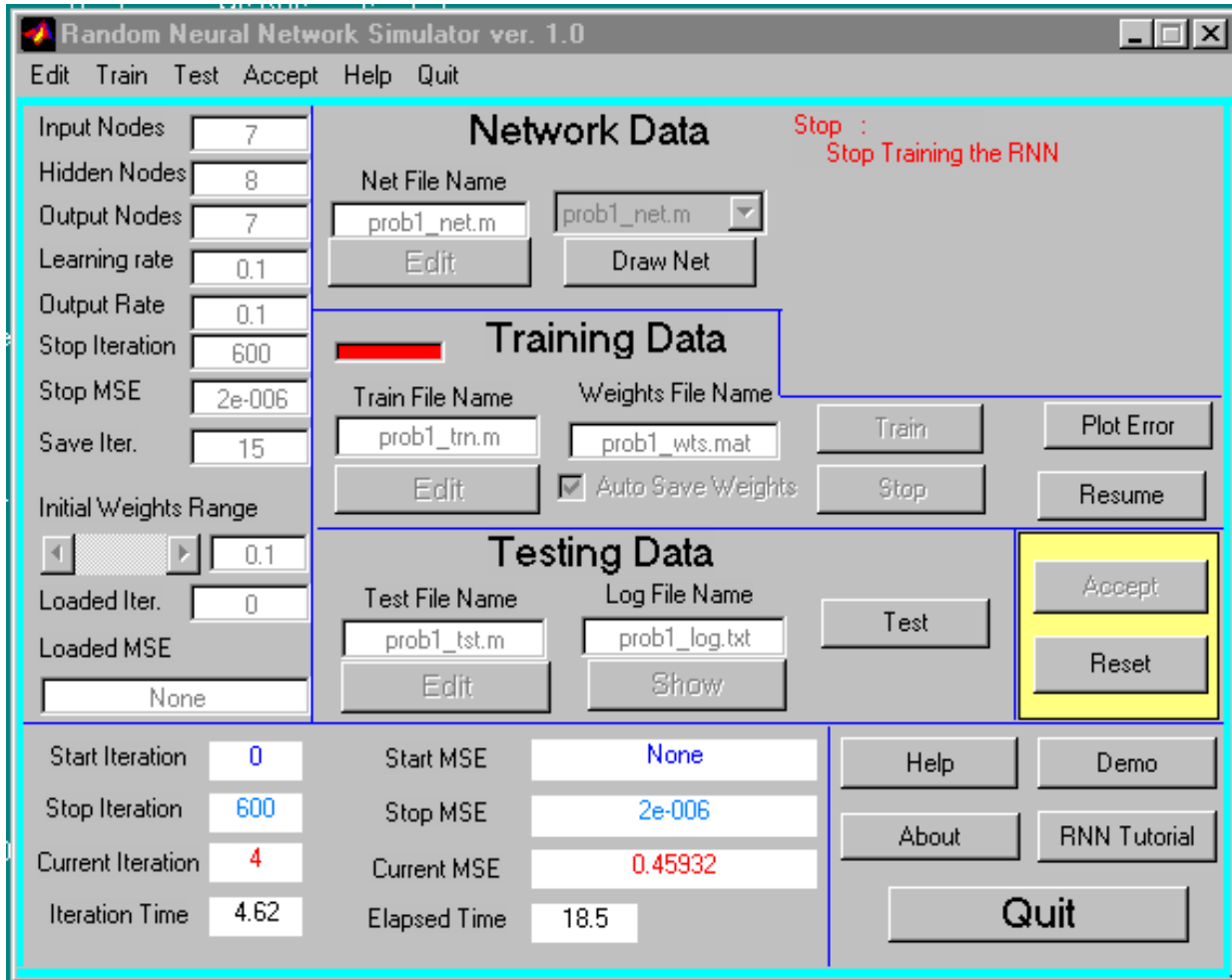


Figure 1.1: RNNSIM v.1

RNNSIM v.2 does not have a graphical user interface, on the other hand, it is very flexible and it overcomes all the drawbacks of **RNNSIM v.1**. In our illustrative examples, we will concentrate on using **RNNSIM v.2**. In Chapter 2, the basics of the random neural network model [1, 2] are presented and the steps of the recurrent learning algorithm [3] are introduced. Chapter 3 illustrates the usage of the package by two detailed examples. The first example shows

a simple three layers feed-forward RNN model and compares its performance with that of the conventional connectionist model trained using back-propagation algorithm. The second example is a fully recurrent RNN model. Both examples are very simple so as to emphasize on the RNN design and training process.

RNNSIM v.2 can be downloaded as as separate files from

- <http://www.cs.ucf.edu/~ahossam/rnnsimv2/>

or as a compressed zip file from

- <http://www.cs.ucf.edu/~ahossam/rnnsimv2/rnnsimv2.zip>

1.2 Installing the Simulator

To install **RNNSIM v.2** on a Unix system, follow the following steps:

- Assuming your current directory is */home/yourname/*, download the compressed file *rnnsimv2.zip* and put it in your current directory.
- Unzip the file using the Unix command *unzip -d rnnsimv2.zip*.
- Now the directory */home/yourname/rnnsimv2* will be created. Inside this directory, you will find the following two directories: */home/yourname/rnnsimv2/example1* and */home/yourname/rnnsimv2/example2*.

To install **RNNSIM v.2** on Windows 95 system, follow the following steps:

- Assuming your current directory is *c:/*, download the compressed file *rnnsimv2.zip* and put it in your current directory.
- Unzip the file using the WinZip program (or any other program that can extract a zip file)
- Now the directory *c:/rnnsimv2* will be created. Inside this directory, you will find the following two directories: *c:/rnnsimv2/example1* and *c:/rnnsimv2/example2*.

Your MATLAB path must include the directory *rnnsimv2* which contains the main functions used by the program. To use these functions into the current MATLAB session, update the MATLAB path using the “path” command as follows:

```
>> path(path, 'c:/rnnsimv2/')      (Windows)
```

```
>> path(path, '/home/yourname/rnnsimv2/')  (Unix)
```

Another way to update the current path is to use the change directory command (assuming that the current directory is *c:* (Windows) or */home/yourname/* (Unix))

```
>> cd rnnsimv2      (Windows & Unix)
```

If the tools are going to be used on a regular basis, it is recommended that the path statements are included in ones personal *startup.m* file or added permanently to the MATLAB path (see the MATLAB manual for more details).

1.3 RNNSIM v.2 Functions and Script Files

After successful installation of **RNNSIM v.2** you should have the following files into the current directory (*/rnnsimv2/*).

A: Main files (problem independent)

File	Type	Short description
train_rnn_gen.m	function m file	trains the RNN
test_rnn_gen.m	function m file	tests the RNN
rnn_gen_test_exact.m	function m file	tests the RNN in batch mode by sloving the nonlinear equations of the model
rnn_gen_test_iterative.m	function m file	tests the RNN in batch mode without sloving the nonlinear equations of the model

C: General files (problem independent)

File	Type	Short description
vardef.m	script m file	declares the global variables.
italize.m	script m file	initializes the RNN.
read_connection_matrix.m	function m file	reads the connection file.
load_net_file.m	function m file	loads the network file.
load_trn_file.m	script m file	loads the training data file.
load_tst_file.m	function m file	loads the testing data file.
initialize_weights.m	function m file	initializes the network weights.
extract_name.m	function m file	extracts a file name without its extension.

D: Documentation and Examples

File	Type	Short description
rnnsimv2.pdf	PDF file	technical report that describes how to use the RNNSIM v.2.
readme.txt	ASCII file	readme file.
contents.m	script m file	contains a list of the package files.
example1	directory	contains files of example 1.
example2	directory	contains files of example 2.

There are also two subdirectories, *example1* and *example2*, containing two applied examples for using the RNNSIM v.2.

D: Contents of the directory /rnnsimv2/example1)

File	Type	Short description
use_rnn_gen1.m	script m file	main file for example 1
rnn_gen_net1.m	script m file	defines the learning parameters.
rnn_gen_con1.dat	ASCII data file	contains the network connections.
rnn_gen_wts1.mat	Mat file	contains the weights.
rnn_gen_trn1.m	script m file	contains the training data.
rnn_gen_tst1.m	script m file	contains the testing data.
rnn_gen_log1.txt	ASCII data file	results of testing the trained network.
rnn_gen_log1.m	script m file	results of testing the trained network.

E: Contents of the directory /rnnsimv2/example2)

File	Type	Short description
use_rnn_gen2.m	script m file	main file for example 2
rnn_gen_net2.m	script m file	defines the learning parameters.
rnn_gen_con2.dat	ASCII data file	contains the network connections.
rnn_gen_wts2.mat	Mat file	contains the weights.
rnn_gen_trn2.m	script m file	contains the training data.
rnn_gen_tst2.m	script m file	contains the testing data.
rnn_gen_log2.txt	ASCII data file	results of testing the trained network.
rnn_gen_log2.m	script m file	results of testing the trained network.

1.4 Contact Information

Hossam Abdelbaki

School of Computer Science

University of Central Florida

Orlando, Florida.

E-mail: ahossam@cs.ucf.edu Web: www.cs.ucf.edu/~ahossam/

Phone: (407) 823-1061 FAX: (407) 823-5419

1.5 Copyright and Disclaimer

This software is copyrighted by Hossam Abdelbaki (Copyright (c) 1998-1999 Hossam Abdelbaki). The following terms apply to all files associated with the software unless explicitly disclaimed in individual files. The author hereby grant permission to use, copy, and distribute this software and its documentation for any purpose, provided that existing copyright notices are retained in all copies and that this notice is included verbatim in any distributions. Additionally, the author grant permission to modify this software and its documentation for any purpose, provided that

such modifications are not distributed without the explicit consent of the author and that existing copyright notices are retained in all copies.

IN NO EVENT SHALL THE AUTHOR OR DISTRIBUTORS BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE, ITS DOCUMENTATION, OR ANY DERIVATIVES THEREOF, EVEN IF THE AUTHOR HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE AUTHOR AND DISTRIBUTORS SPECIFICALLY DISCLAIM ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT. THIS SOFTWARE IS PROVIDED ON AN "AS IS" BASIS, AND THE AUTHOR AND DISTRIBUTORS HAVE NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

- MATLAB is a trademark of the MathWorks, Inc.
- MS-Windows is a trademark of Microsoft Corporation.
- Unix is a trademark of AT&T.

Chapter 2

The Random Neural Network Model

2.1 Basics of the Random Neural Network Model

In the random neural network model (RNN) [1, 2, 3] signals in the form of spikes of unit amplitude circulate among the neurons. Positive signals represent excitation and negative signals represent inhibition. Each neuron's state is a non-negative integer called its potential, which increases when an excitation signal arrives to it, and decreases when an inhibition signal arrives. Figure 2.1 shows a single neuron representation in the RNN.

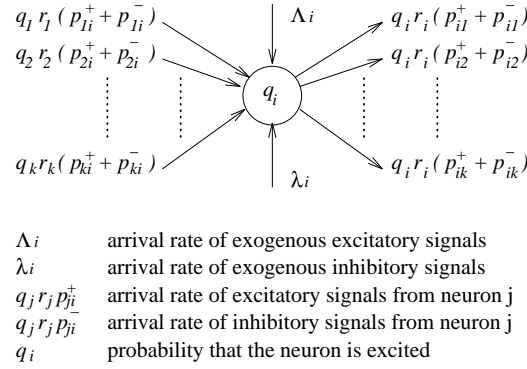


Figure 2.1: Representation of a neuron in the RNN.

An excitatory spike is interpreted as a “+1” signal at a receiving neuron, while an inhibitory spike is interpreted as a “−1” signal. A neuron i , shown in Figure 2.1, emitting a spike whether it is an excitation or an inhibition, will lose potential of one unit, going from some state whose value is k_i to the state of value $k_i - 1$. The state of the n -neuron network at time t , is represented by the vector of non-negative integers $\mathbf{k}(t) = (k_1(t), \dots, k_n(t))$, where $k_i(t)$ is the potential or integer state of neuron i . We will denote by k and k_i arbitrary values of the state vector and of the i -th neuron's state. Neuron i will “fire” (i.e. become excited) if its potential is *positive*. The spikes will then be sent out at a rate r_i , with independent, identically and exponentially distributed interspike intervals. Spikes will go out to some neuron j with probability p_{ij}^+ as excitatory signals, or with probability p_{ij}^- as inhibitory signals. A neuron may also send signals out of the network with

probability d_i , and $d_i + \sum_{j=1}^n [p_{ij}^+ + p_{ij}^-] = 1$. Let $w_{ij}^+ = r_i p_{ij}^+$, and $w_{ij}^- = r_i p_{ij}^-$. Here the “ w ’s” play a role similar to that of the synaptic weights in connectionist models. Although they specifically represent rates of excitatory and inhibitory spike emission, they are non-negative. Exogenous excitatory and inhibitory signals arrive to neuron i at rates Λ_i , λ_i , respectively. This is a “recurrent network” model which may have feedback loops of arbitrary topology. Computations are based on the probability distribution of network state $p(k, t) = \Pr[\mathbf{k}(t) = \mathbf{k}]$, or with the marginal probability that neuron i is excited $q_i(t) = \Pr[k_i(t) > 0]$. As a consequence, the time-dependent behavior of the model is described by an infinite system of *Chapman-Kolmogorov* equations for discrete state-space continuous time Markov systems. Information in this model is carried by the *frequency* at which spikes travel. Thus, neuron i , if it is excited, will send spikes to neuron j at a frequency $w_{ij} = w_{ij}^+ + w_{ij}^-$. These spikes will be emitted at exponentially distributed random intervals. In turn, each neuron behaves as a non-linear *frequency demodulator* since it transforms the incoming excitatory and inhibitory spike trains’ rates into an “amplitude”. Let $q_i(t)$ be the probability that neuron i is excited at time t . The stationary probability distribution associated with the model is given by:

$$p(k) = \lim_{t \rightarrow \infty} p(k, t), \quad q_i = \lim_{t \rightarrow \infty} q_i(t), \quad i = 1, \dots, n. \quad (2.1)$$

Theorem ([1, 3]) Let

$$q_i = \frac{\lambda_i^+}{r_i + \lambda_i^-} \quad (2.2)$$

where the λ_i^+ and λ_i^- , for $i = 1, \dots, n$ satisfy the system of non-linear simultaneous equations:

$$\lambda_i^+ = \Lambda_i + \sum_{j=1}^n q_j w_{ji}^+, \quad \lambda_i^- = \lambda_i + \sum_{j=1}^n q_j w_{ji}^-, \quad (2.3)$$

$$r_i = \sum_{j=1}^n q_j w_{ji}^+ + q_j w_{ji}^- \quad (2.4)$$

If a unique non-negative solution λ_i^+ , λ_i^- exist to Equations 2.2 and 2.3 such that each $q_i < 1$ then,

$$p(K) = \prod_{i=1}^n (1 - q_i) q_i^{k_i} \quad (2.5)$$

This means that whenever the $0 < q_i < 1, i = 1, \dots, n$ can be found from 2.2 and 2.3, the network is stable in the sense that all the moments of the neural network state can be found from the above formula, and all moments are finite. For example, the average potential at a neuron i is simply $q_i/(1 - q_i)$. If for some neuron, we have $q_i = \lambda_i^+/(r_i + \lambda_i^-) > 1$, we say that the neuron is unstable or saturated. This implies that it is constantly excited in steady state. Its rate of spike emission, r_i , to another neuron j of the network appears as a constant source of positive or negative signals of rates $r_i p_{ij}^+$ and $r_i p_{ij}^-$.

2.2 Learning in the Supervised Recurrent Random Neural Network Model

[3]

Let w_{ij}^+ is the rate at which neuron i sends “excitation spikes” to neuron j when neuron i is excited and w_{ij}^- is the rate at which neuron i sends “inhibition spikes” to neuron j when neuron i is excited. For an n neuron network, the network parameters are these $n \times n$ “weight matrices” $\mathbf{W}^+ = w_{ij}^+$ and $\mathbf{W}^- = w_{ij}^-$ which need to be learned from the input data.

The algorithm chooses the set of network parameters \mathbf{W}^+ and \mathbf{W}^- from the training set of K input-output pairs (\mathbf{X}, \mathbf{Y}) where the set of successive inputs is denoted $\mathbf{X} = (\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(K)})$. $\mathbf{x}^{(k)} = (\Lambda^{(k)}, \lambda^{(k)})$ are pairs of excitation and inhibition signal flow rates entering each neuron from outside of the network.

$$\Lambda^{(k)} = (\Lambda_1^{(k)}, \dots, \Lambda_n^{(k)}), \quad \lambda^{(k)} = (\lambda_1^{(k)}, \dots, \lambda_n^{(k)}), \quad (2.6)$$

The successive desired outputs are the vectors $\mathbf{Y} = (\mathbf{y}^{(1)}, \dots, \mathbf{y}^{(K)})$, where each vector $\mathbf{y}^{(k)} = (y_1^{(k)}, \dots, y_n^{(k)})$ whose elements $y_i^{(k)} \in [0, 1]$ correspond to the desired output values for each neuron. The network adjusts its parameters to produce the set of desired output vectors in a manner that minimizes a cost function $E^{(k)}$

$$E^{(k)} = \frac{1}{2} \sum_{i=1}^n a_i (q_i^{(k)} - y_i^{(k)})^2, \quad a_i \geq 0. \quad (2.7)$$

In this network all neurons are potentially output neurons, therefore if a neuron j is to be removed from the network output it suffices to set $a_j = 0$ in the error function. For each successive desired input-output pair, indexed by k , the $n \times n$ weight matrices $\mathbf{W}_k^+ = \{w_k^+(i, j)\}$ and $\mathbf{W}_k^- = \{w_k^-(i, j)\}$ must be adjusted after each input is presented, by computing for each input $\mathbf{x}^{(k)} = (\Lambda^{(k)}, \lambda^{(k)})$, a new value \mathbf{W}_k^+ and \mathbf{W}_k^- of the weight matrices, using gradient decent. Since the matrices represent rates, only nonnegative values of the matrices are valid. Let the generic term w_{uv} denotes any weight term, which would be either $w_{uv} \equiv w_{uv}^+$ or $w_{uv} \equiv w_{uv}^-$. The rule for the weight update may be written as

$$w_{uv}^{(k)} = w_{uv}^{(k-1)} - \eta \sum_{i=1}^n a_i (q_i^{(k)} - y_i^{(k)}) [\partial q_i / \partial w_{uv}]^{(k)} \quad (2.8)$$

Where $\eta > 0$ is the learning rate, and

- $q_i^{(k)}$ is calculated using the input $\mathbf{x}^{(k)}$ and $w_{uv} = w_{uv}^{(k-1)}$, in Equations 2.2 and 2.3.
- $[\partial q_i / \partial w_{uv}]^{(k)}$ is evaluated at the values $q_i = q_i^{(k)}$, $w_{uv} = w_{u,v}^{(k-1)}$.

To compute $[\partial q_i / \partial w_{uv}]^{(k)}$ the following equation is derived

$$\begin{aligned} \partial q_i / \partial w_{uv} &= \sum_j \partial q_j / \partial w_{uv} [w_{ji}^+ - w_{ji}^- q_i] / \lambda_i^- \\ &\quad - \mathbf{1}[u \equiv i] q_i / \lambda_i^- \\ &\quad + \mathbf{1}[w_{uv} \equiv w_{ui}^+] q_u / \lambda_i^- \\ &\quad - \mathbf{1}[w_{uv} \equiv w_{ui}^-] q_u q_i / \lambda_i^- \end{aligned} \quad (2.9)$$

where

$$\mathbf{1}[x] = \begin{cases} 1 & \text{if } x \text{ is true,} \\ 0 & \text{otherwise.} \end{cases}$$

Let $\mathbf{q} = (q_1, \dots, q_n)$, and define the $n \times n$ matrix

$$\mathbf{W} = \{[w_{ij}^+ - w_{ij}^- q_j]/\lambda_j^-\} \quad i, j = 1, \dots, n \quad (2.10)$$

The vector equations can now be written as

$$\begin{aligned} \partial \mathbf{q} / \partial w_{uv}^+ &= \partial q_i / \partial w_{uv}^+ \mathbf{W} + \gamma^+(u, v) q_u \\ \partial \mathbf{q} / \partial w_{uv}^- &= \partial q_i / \partial w_{uv}^- \mathbf{W} + \gamma^-(u, v) q_u \end{aligned} \quad (2.11)$$

Where the elements of the n -vectors

$$\gamma^+(u, v) = [\gamma_1^+(u, v), \dots, \gamma_n^+(u, v)]$$

and

$$\gamma^-(u, v) = [\gamma_1^-(u, v), \dots, \gamma_n^-(u, v)]$$

are

$$\gamma_i^+(u, v) = \begin{cases} -1/\lambda_i^- & \text{if } u = i, v \neq i \\ +1/\lambda_i^- & \text{if } u \neq i, v \neq i \\ 0 & \text{otherwise.} \end{cases} \quad \gamma_i^-(u, v) = \begin{cases} -(1 + q_i)/\lambda_i^- & \text{if } u = i, v = i \\ -1/\lambda_i^- & \text{if } u = i, v \neq i \\ -q_i/\lambda_i^- & \text{if } u \neq i, v = i \\ 0 & \text{otherwise.} \end{cases} \quad (2.12)$$

Notice that

$$\begin{aligned} \partial \mathbf{q} / \partial w_{uv}^+ &= \gamma^+(u, v) q_u [\mathbf{I} - \mathbf{W}]^{-1} \\ \partial \mathbf{q} / \partial w_{uv}^- &= \gamma^-(u, v) q_u [\mathbf{I} - \mathbf{W}]^{-1} \end{aligned} \quad (2.13)$$

where \mathbf{I} denotes the $n \times n$ identity matrix. Hence the main computational effort in this algorithm is to obtain $[\mathbf{I} - \mathbf{W}]^{-1}$. This is of time complexity $O(n^3)$ or $O(mn^2)$ if m -step relaxation method is used.

From the above, the learning algorithm for the network can be derived. First initialize the matrices \mathbf{W}_0^+ and \mathbf{W}_0^- in some appropriate manner. This initiation can be made at random if no better method can be determined. Choose a value of η , and then for each successive value of k , starting with $k = 1$ proceed as follows,

1. Set the input values to $\mathbf{x}^{(k)} = (\mathbf{\Lambda}^{(k)}, \lambda^{(k)})$.
2. Solve the system of nonlinear Equations in 2.2 and 2.3 with these values.
3. Solve the system of linear equations in Equation 2.13 with the results of step 2.
4. Using Equation 2.8 and the results of steps 2) and 3), update the matrices \mathbf{W}_k^+ and \mathbf{W}_k^- . Since the “best” matrices (in terms of gradient decent of the quadratic cost function) which satisfy the nonnegativity constraint are sought, in any step k of the algorithm, if the iteration yields a negative value of a term, there are two alternatives:
 - set the term to zero, and stop the iteration for this term in this step k ; in the next step, $k + 1$, iterate on this term with the same rule starting from its current zero value;
 - go back to the previous value of the term and iterate with a smaller value of η .

In our implementation we have used the first technique. This general scheme can be specialized to feed-forward networks by noting that the matrix $[\mathbf{I} - \mathbf{W}]^{-1}$ will be triangular, yielding a computational complexity of $O(n^2)$, rather than $O(n^3)$, for each gradient iteration. Furthermore, in a feed-forward network, Equations 2.2 and 2.2 are simplified in that q_i is only dependent upon q_j for $j < i$. This reduces the computational effort required for finding the solution.

After the network has been trained and the optimum weights are obtained, the network can be used in the normal operation mode. The only needed computations to obtain the output are the addition, multiplication and division as noticed from Equations 2.2 and 2.2.

2.3 Applications of the Random Neural Network Model [4]

By appropriately mapping external signals and neuron states into certain physical quantities, the RNN has been successfully applied to several engineering problems including solving NP-complete optimization problems [5, 6], texture image generation [7, 10] and image and video compression [8, 9]. The empirically observed success of the network can be justified using the theoretical results reported in [11], where the authors showed that for any continuous multivariate function f on a compact set, it is possible to construct a RNN model with a predefined structure that can implement a mapping close enough to f in some precise sense to a given degree of accuracy.

2.3.1 Optimization

The RNN model has been successfully applied to solve some well-known NP-complete problems such as the traveling salesman problem (TSP) [5]. For the TSP, the random neural network is applied by Gelenbe et al. to obtain approximate solutions. The basic idea is to assume a state dependent dynamic behavior of the input parameters to the RNN. Neuron inputs are adjusted by time-delayed feedback, while the external inputs to the neurons are varied according to a penalty function that is to be optimized. The network parameters are derived for the TSP, and tested on a large number of 10 and 20 city examples chosen randomly. Comparisons are conducted with exact solution and with well-known greedy algorithm, as well as with some other neural network approaches. Experiments show that the RNN provides better results than the nearest neighbor heuristic and the Hopfield network approach given by Abe [12].

2.3.2 Texture Image Generation

Modeling image textures have been of great interest because it is useful in analyzing images such as satellite images or medical images. The conventional methods such as the Markov Random Fields (MRF) require rather complicated computation. Fast image texture generation algorithms have been proposed for texture modeling, as well as for other applications such as computer graphics and video game background generations. Artificial texture images can be generated using locally interconnected RNN [7, 10]. Because different artificial texture images have different local granularity and directional dependencies, the setting of positive and negative weights connecting neighboring neurons provides an effective control mechanism for texture characteristics the RNN generates. A RNN can be mapped to an image, with each neurons stable state solution corresponding to the image pixel intensity. By connecting each neuron to its nearest neighbors, and setting different connection weights towards different directions, the texture image characteristics can be directly controlled. Experimental results by Atalay et. el. [7, 10]. show a variety of image textures generated using the RNN with relatively small computational requirement. Further more, the RNN is intrinsically parallel, so that it can be easily sped up using parallel computers.

2.3.3 Image and Video Compression

The RNN arranged into three layer feed-forward structure is used for gray-level still image compression [13]. The network has 64 input and 64 output neurons, while the number of hidden neurons, m , is variable so that different compression ratio can be achieved. At the encoder end, the image is segmented into 8 by 8 blocks, i.e. 64 pixels. Each pixel value is scaled linearly to $[0, 1]$ and fed into corresponding neuron in the input layer. The neuron values at the hidden neurons are then quantized and transmitted as compressed data. With the received hidden neuron values, the decoder can compute output neuron values, and scale it to get the reconstructed image. The results of [9] and [13] show comparable results as those using connectionist models, but inferior to transform type compression algorithms, such as JPEG and wavelet. However, the RNN compression scheme can be adapted to video compression and achieve satisfactory results. Because the neural codec is trained beforehand, it is fast and can be used in real-time video compression. With a simplified motion compensation technique and combination of multiple codecs operating at different compression ratio, low bit rate compression with flexible bit rate control is demonstrated [9]. The rate distortion performance is comparable to the low bit rate video compression standard H.261 at normal compression ratio, and is superior to H.261 at high compression ratios.

Chapter 3

Using the RNNSIM v.2

In this Chapter, the RNN design process is illustrated through two detailed examples. A comparison has also been made between the RNN model and some of the conventional neural network models. All the files discussed in this chapter can be found within the **RNNSIM v.2** package.

Before going through the examples, we will do some steps to make sure that the program is installed successfully and also to get familiar with invoking the functions. We will assume that you followed the installation procedure discussed in Section 1.2. Any line begins with `>>` is a MATLAB command that you should write inside the MATLAB command window (you need not to write the comments that begin with `%`).

For Windows operating system,

```
>> pwd % print the current working directory
c:/
>> path(path,'c:/rnnsimv2/') % add rnnsimv2 directory to the current path
>> cd c:/rnnsimv2/example1/ % change the current directory
>> pwd % again print the current working directory
c:/rnnsimv2/example1
```

For Unix operating system,

```
>> pwd % print the current working directory
/home/yourname
>> path(path,'/home/yourname/rnnsimv2/'); % add rnnsimv2 directory to the current path
>> cd /home/yourname/rnnsimv2/example1/ % change the current directory
>> pwd % again print the current working directory
/home/yourname/rnnsimv2/example1
```

From this point, all the MATLAB scripts that will be written will work equally on Windows or Unix systems.

```
>> load rnn_gen_wts1 %load the weights of the trained network of example 1
```

```

>> qqq = test_rnn_gen('1',[-1 1 -1 -1 1 1 1]) % test the trained network
qqq
=
    1.0000    0.0281
>> qqq = test_rnn_gen('1',[-1 1 -1 -1 1 1 -1]*0.5)
qqq
=
    0.8475    0.0326

```

If you changed the current directory to the directory of the second example, *exmaple2*, you can try the following.

```

>> load rnn_gen_wts2          %load the weights of the trained network of example 2
>> qqq = test_rnn_gen('2',[.1 .3 .5 .9 .1 .3 .5 .9 1]) % test the trained network
qqq
=
    Columns 1 through 7
    0.1210    0.3233    0.5023    0.8695    0.1013    0.3068    0.5253
    Columns 8 through 9
    0.9151    0.4463
>> rnn_gen_test_exact([.1 .3 .5 .9 .1 .3 .5 .9 1], 100 ,0)
ans
=
    Columns 1 through 7
    0.1013    0.3009    0.5014    0.9029    0.1007    0.3008    0.5015
    Columns 8 through 9
    0.9026    0.1002

```

Do not worry about the arguments passed to the shown functions, they will be discussed later. If you tried the above code segments and they work as shown, then you can go to the next sections to design, train, and evaluate the RNN. All the simulations in the following sections have been carried out on a PC with Pentium 233 MHz processor.

3.1 Example 1: Feed-Forward Structure

In this example, we assume a three layers random neural network with 7 neurons in the input layer, 5 neurons in the hidden layer, and 2 neurons in the output layer (see Figure 3.1). The task to be carried out using this network is to classify the patterns $[-1 \ 1 \ -1 \ -1 \ 1 \ 1 \ 1]$ and $[1 \ -1 \ 1 \ 1 \ -1 \ -1 \ -1]$ by associating the output $[1 \ 0]$ to the first pattern and the output $[0 \ 1]$ to the second one. This seems to be trivial problem but in fact it can lead to some good applications in decoding data coded with special sequences. This problem can be applied directly to **RNNSIM v.1** since it has a traditional feed-forward structure. The files for this examples can be found in the directory */rnnsimv2/example1*.

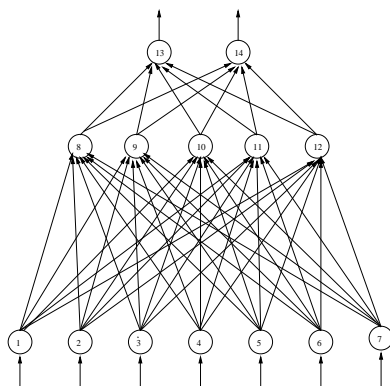


Figure 3.1: The Feed-forward RNN discussed in example 1.

First prepare the connection data file (**rnn_gen_con1.dat**) as shown in the following listing.

```

% File Name           : rnn_gen_con1.dat
% 3 Layers Feed Forward RNN Model
% -----
% Number of Neurons in the Input Layer   : 7 neurons
% Number of Neurons in the Hidden Layer  : 5 neurons
% Number of Neurons in the Output Layer  : 2 neurons
% Total Number of Neurons
14
% Connection Matrix for the Positive Weights ( wplus )
1 5 8 9 10 11 12
2 5 8 9 10 11 12
3 5 8 9 10 11 12
4 5 8 9 10 11 12
5 5 8 9 10 11 12
6 5 8 9 10 11 12
7 5 8 9 10 11 12
8 2 13 14
9 2 13 14
10 2 13 14
11 2 13 14
12 2 13 14
13 0
14 0
% Connection Matrix for the Negative Weights ( wminus )
1 5 8 9 10 11 12
2 5 8 9 10 11 12
3 5 8 9 10 11 12
4 5 8 9 10 11 12
5 5 8 9 10 11 12
6 5 8 9 10 11 12
7 5 8 9 10 11 12
8 2 13 14
9 2 13 14
10 2 13 14
11 2 13 14
12 2 13 14
13 0
14 0
% Number of Input Neurons
7 1 2 3 4 5 6 7
% Number of Output Neurons
2 13 14

```

This file contains the indexes of the neurons that are connected together. For example, in the file, the line 1 5 8 9 10 11 12 indicates that neuron 1 is connected to 5 neurons. The indexes of these neurons are 8, 9, 10, 11, and 12. In other words, we can say that $W(1, 8)$, $W(1, 9)$, $W(1, 10)$, $W(1, 11)$, and $W(1, 12)$ exist. If a neuron is an output neuron and is not connected to any of the neurons in the network, then 0 is put as the number of neurons connected to it. Any line begins with % is considered as a comment for the program.

The second step it to prepare the work data file (**rnn_gen_net.m**) as shown below.

```
%% rnn_gen_net1.m
%% ##### Network Parameter Initialization #####
Mse_Threshold = 0.0005;           %Required stop mean square error value
Eta = 0.1;                        %Learning rate
N_Iterations = 600;              %Maximum number of iterations
R_Out = 0.1;                     %Firing rate of the output Neurons
RAND_RANGE = 0.2;               %Range of weights initialization
FIX_RIN = 0;                    %DO NOT CHANGE (Related to RNN function approximation)
R_IN = 1;                        %DO NOT CHANGE (Related to RNN function approximation)
SAVE_Weights = 1;               %Flag to indicate if the weight will be automatically saved
N_Saved_Iterations = 10;        %Number of iterations after which weights will be saved automatically
Net_File_Name = 'rnn_gen_net.m'; %Network file name (this file)
Connection_File_Name = 'rnn_gen_con1.dat'; %Connections file name
Weights_File_Name = 'rnn_gen_wts1.mat'; %Weights file name
Trn_File_Name = 'rnn_gen_trn1.m'; %Train data file name
Tst_File_Name = 'rnn_gen_tst1.m'; %Testing data file name
Log_File_Name = 'rnn_gen_log1.txt'; %results file name (ASCII format)
Temp_Log_File_Name = 'rnn_gen_log1.m'; %Results file name (MATLAB format)
SAVE_TEXT_LOG = 1;              %Flag to indicate if the results will be saved in ASCII format
SAVE_MFILE_LOG = 1;            %Flag to indicate if the results will be saved in MATLAB format
Res = 0.0001;                  %Resolution of solving the non linear equations of the model
AUTO_MAPPING = 0;              %Flag = 0 for FF networks and 1 for fully recurrent networks
```

We predefine the learning rate and the stopping mean squared error to be equal to 0.1 and 0.0005 respectively. The weights will be automatically saved to the file **rnn_gen_wts1.mat** after each 10 iterations. When the trained network is tested, the response will be saved to the file **rnn_gen_log1.txt** as normal text and to the file **rnn_gen_log1.m** as a MATLAB m-file (to be used for plotting or comparisons).

The training patterns are saved in the file **rnn_gen_trn1.m** as shown. In the main file (**use_rnn_gen1.m**), these patterns can be loaded either by inserting them inside the file or by loading this training data file.

```

%%%%%%%%%%%% Training file: rnn_gen_trn1.m %%%%%%%%%%%%%
N_Patterns = 2;           %Number of Training Patterns
##### Input Training Patterns #####
TRAIN_INPUT = [
        -1    1   -1   -1    1    1    1
         1   -1    1    1   -1   -1   -1 ];
##### Desired Output Patterns #####
TARGET = [
        1 0
        0 1 ];
%Order of applying the patterns during the training process
Z = 1:2;

```

Also the patterns that will be used for testing can be saved in the test data file **rnn_gen_tst1.m**. Again, In the main file, these test patterns can be loaded by writing them directly or by loading the test data file.

```

%%%%%%%%%%%% Testing file: rnn_gen_tst1.m %%%%%%%%%%%%%
N_Patterns = 2;           %Number of test patterns
##### Input Test Patterns #####
TEST_INPUT = [ -1    1   -1   -1    1    1    1
               1   -1    1    1   -1   -1   -1 ];

```

The training and test data files are optional and it is recommended to use them when there are very large training and testing sets. Finally, update the input-output patterns inside the main file (**use_rnn_gen1.m**).

```

%%%%%% The training pairs can be loaded from an external data file %%%%%
% Trn_File_Name = 'rnn_gen_trn.m';
% load_trn_file(Trn_File_Name);
% X = TRAIN_INPUT;
% Y = TARGET;

```

```

%%%%%%%% Or they can be supplied directly %%%%%%%%%
%%%%%%%% Input Patterns %%%%%%%%%
X = [ -1  1  -1  -1  1  1  1
      1  -1  1  1  -1  -1  -1];
%%%%%%%% Output Patterns %%%%%%%%%
Y = [ 1  0
      0  1];
%%%%%%%% Training pairs are applied with the order saved in Z %%%%%%%%%
ss = size(X);  Z = 1:ss(1);
%%%%%%%% Train the feed-forward RNN %%%%%%%%%
err_result = train_rnn_gen('1',X,Y);
%%%%%%%% or using multiple epochs %%%%%%%%%
% err_result = train_rnn_gen('1',X,Y,4);

```

The first argument of the function **train_rnn_gen** defines the training mode of the network, '1' for feed-forward mode and '2' for recurrent mode. The function also accepts the input and output patterns needed for training and gives the current error as an output argument. The function can take an optional argument that represents the number of times each training pair will be applied to the network.

IMPORTANT NOTE: Notice before running the main file (**use_rnn_gen1.m**) that the directory *example1* contains the file **rnn_gen_wts1.mat** which represents the weights of the network of example 1 after being successfully trained. So it is better to have a backup copy of this file because all the discussions in the following sections assume the existence of the original copy of the file **rnn_gen_wts1.mat** distributed with the simulator (the same will apply for the weights file of example 2).

When you run the file **use_rnn_gen1.m**, the following message will appear.

```
Do you want to start from beginning, load from disk or Test (S/L/T)?
```

Now enter 'S' from the keyboard to start training. The following lines will appear.

```
Reading the network parameters ....( rnn_gen_net1.m )
Reading the connection matrix ....( rnn_gen_con1.dat )
```

```
Network      : loaded from file ( rnn_gen_net1.m )
Input/Output : 7 / 2
Weights      : Initialized
Iteration    : 0
MSE         : None
Save after   : 1 Iterations
Mode        : Start Training
```

press any key to continue

After pressing any key, you will see the progress in training as below. Suppose that you stopped the training, by pressing **CTRL C** keys, after the 13th iteration. At this point it is possible to plot the error, or print the weights.

1	1.121977831	0.490000	0.490000
2	0.653439555	0.440000	0.930000
3	0.376987274	0.440000	1.370000
4	0.106726622	0.490000	1.860000
5	0.036940461	0.500000	2.360000
6	0.025955774	0.490000	2.850000
7	0.020786295	0.490000	3.340000
8	0.017176270	0.500000	3.840000
9	0.014388401	0.490000	4.330000
10	0.012278605	0.500000	4.830000

Weights are saved to file (rnn_gen_wts1.mat)

11	0.010625446	0.440000	5.270000
12	0.009249951	0.500000	5.770000
13	0.008093234	0.490000	6.260000

Each line indicates the iteration number, the average mean squared error, the time of each iteration (in sec.), and the total time elapsed (in sec.) since the beginning of the training process. Notice that the weights were saved after the 10th iteration.

To continue training the network beginning from the point at which the training was stopped, run the file **train_rnn_gen1** again and then type 'L' to load the previous network. The following lines will appear.

```

Network      : loaded from file ( rnn_gen_net1.m )
Input/Output : 7 / 2
Weights      : loaded from file ( rnn_gen_wts1.mat )
Iteration    : 10
MSE          : 0.012278605
Save after   : 10 Iterations
Mode         : Continue Training

```

press any key to continue

When you press any key, the training will proceed and will stop automatically and the weights will be saved when the goal MSE is reached or the maximum number of training epochs are exceeded. This procedure facilitates the training when the network structure is complex or the training data set is so large. For the network in our example, the goal MSE is reached after 48 epochs (23.19 sec.) and the training is stopped as shown below.

```

46      0.000517621    0.440000    22.250000
47      0.000500650    0.440000    22.690000
48      0.000484094    0.500000    23.190000
Weights are saved to file ( rnn_gen_wts1.mat )
      STOP TRAINING
      Minimum Mean Squared Error is reached

```

The RNN can then be tested by invoking the function `test_rnn_gen1` which accepts the type of testing routine, feed-forward or recurrent, and returns the output vector calculated using the network parameters (weights).

```

% Test the feed-forward RNN
>> qqq = test_rnn_gen('1',X);

```

The result will be

```

>> qqq
      =
      1.0000    0.0281
      0.0121    1.0000

```

The function `test_rnn_gen1` can accept some other optional arguments such as the weights file name and the output log file name, we did not use these arguments in the discussion because we are assuming that the testing follows the training in the same MATLAB session. Note that all the network parameters are defined as global variable so that they can be used directly from the MATLAB command window. For example to display the values of \mathbf{W}^+ , you can just write

```
>> wplus
```

and you can plot the MSE by writing

```
>> plot(err)
```

There is another methods to test the network and save the results to external files. This is done by running the file `use_rnn_gen1` again and passing the choice 'T'. The following message will appear.

```
-----  
Network      : loaded from file ( rnn_gen_net1.m )  
Input/Output : 7 / 2  
Weights      : loaded from file ( rnn_gen_wts1.mat )  
Iteration    : 48  
MSE          : 0.000484094  
Save after   : 10 Iterations  
Mode         : Start Testing  
-----
```

```
press any key to continue
```

After pressing any key, you will find the files `rnn_gen_log1.m` and `rnn_gen_log1.txt` saved to the current directory. The contents of the these files are shown below.

```
%File rnn_gen_log1.m  
%Iteration No. 48      MSE = 0.000484094  
RESPONSE = [  
1.000000000 0.028097642  
0.012097887 1.000000000  
];
```

```

%File rnn_gen_log1.txt
Iteration No. 48      MSE = 0.000484094
-----
pattern No. 1:
-1.00  1.00 -1.00 -1.00  1.00  1.00  1.00
Response:
1.000000000  0.028097642
pattern No. 2:
1.00 -1.00  1.00  1.00 -1.00  -1.00 -1.00
Response:
0.012097887  1.000000000

```

Now, return to the example, the ratio between the first and the second outputs can be used as the classification criteria. For example, for the first pattern, $[-1 \ 1 \ -1 \ -1 \ 1 \ 1 \ 1]$, the output ratio is $1.0000/0.0281 = 35.5872$ and for the second pattern, $[1 \ -1 \ 1 \ 1 \ -1 \ -1 \ -1]$, the ratio is $0.0121/1.0000 = 0.0121$. So we can classify the input pattern to belong to one of the two classes if the output ratio is greater than one and belong to the other class otherwise (assuming that we used training pattern from different classes).

As an assessment for the performance of the RNN gradient algorithm, we applied the same classification problem to a conventional feed- forward neural networks with the same structure $(7 - 5 - 2)$. We tried three different variations of the back-propagation (BP) algorithm, namely, the basic gradient descent (GD), the conjugate gradient (CG) algorithm, and the Levenberg-Marquardt (LM) algorithm. We set the learning rate and the stopping MSE to the same values used for the RNN (0.1 and 0.0005). The learning curves for the different feed-forward variations are shown in Figure 3.2.

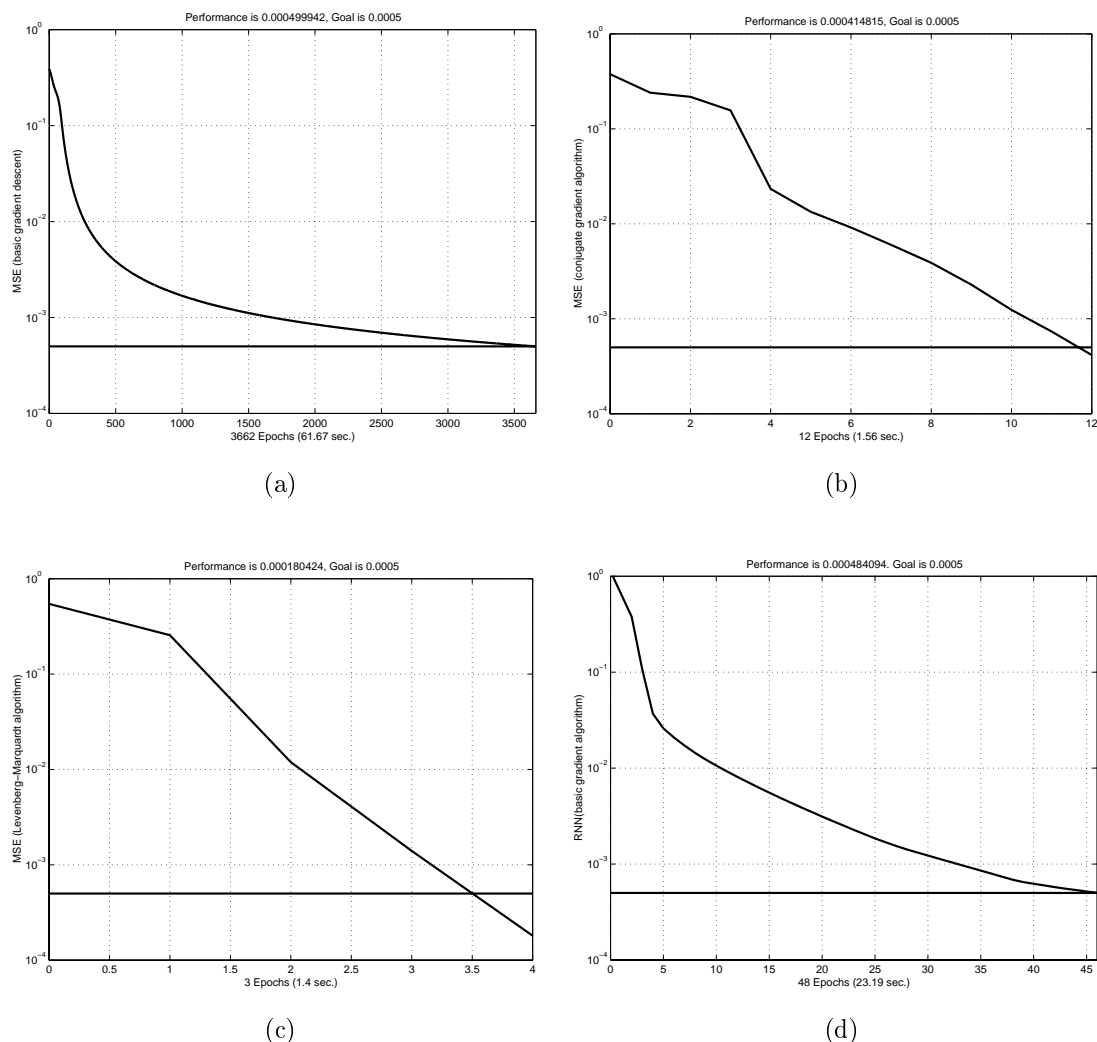


Figure 3.2: Learning curves for 7 – 5 – 2 feed-forward structure trained using (a) BP with basic gradient descent algorithm iteration (b) BP with conjugate gradient algorithm iteration (c) BP with Levenberg-Marquardt algorithm (d)RNN gradient descent training.

From the curves, it is clear that the LM algorithm is the fastest and the basic GD algorithm is the slowest. But in fact, the more important issue is not the speed of learning but the robustness of the trained network to patterns which have not been seen before. By testing any of the previous networks with the patterns used in the training, all the networks give the exact output, but when we apply some different patterns, the behavior changes from network to network. For example, if we apply the pattern $[0.01 \ -0.01 \ 0.01 \ 0.01 \ -0.01 \ -0.01 \ -0.01]$, which is the second pattern used in the training but multiplied by 0.01, the output of the RNN is only correct output among all the trained networks. To test the RNN with this pattern we invoke the following command

```
% Test the feed-forward RNN
>> qqg = test_rnn_gen('1',[1 -1 1 1 -1 -1 -1]*0.01);
```

The output will be

```
>> qqg =
    0.0010    0.0372
```

the output ratio of the RNN is $0.0010/0.0372 = 0.0269 < 1$. This means that the pattern used for testing is most probably the second pattern used in the training. On the other hand, the output ratios for the gradient descent, conjugate gradient, and Levenberg-Marquardt algorithms are 1.6727, 1.1080, and 13.1881. Since all the ratios are greater than one, the decision is wrong.

If we think that the training patterns are codes and the neural network is used to decode these codes, then most probably, due to noise, the received code will be affected. So to test the trained networks with some versions of the perturbed codes, we used the the following codes

- C1: [1 -1 1 1 -1 -1 1] to simulate a signal with one bit error (last bit).
- C2: [0.01 -0.01 0.01 0.01 -0.01 -0.01 -0.01] to simulate a very weak received signal.
- C3: [0.01 -0.01 0.01 0.01 -0.01 -0.01 0.01] to simulate weak signal with one bit in error.

The output ratios are recorded for all the network as listed in Table 3.1. From the results shown in the Table, it can be noticed that the RNN is the only neural network that gives correct decision (output ratios < 1) for all the applied perturbed codes.

	GD	CG	LM	RNN
C1	0.73	0.02	0.03	0.34
C2	1.67	13.18	1.10	0.03
C3	1.73	13.16	1.13	0.41

Table 3.1: Output ratio for the different training algorithms

3.2 Example 2: Recurrent Structure

In this example, we illustrate a single layer fully recurrent RNN model. The model has 9 neurons as shown in Figure 3.3. The task to be carried out using this network is to store the analog pattern, $[0.1 \ 0.3 \ 0.5 \ 0.9 \ 0.1 \ 0.3 \ 0.5 \ 0.9 \ 0.1]$, as a fixed attractor for the network. It should be mentioned here that since the training pattern is not binary, Hopfield network can not be used in these kind of problems. This training pattern may be for example, the normalized gray levels of a certain texture image that we want the network to recognize and classify. The files for this example can be found in the directory */rnnsimv2/example2*.

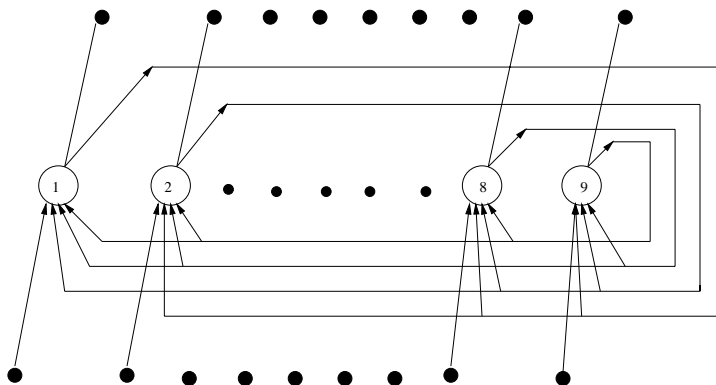


Figure 3.3: The recurrent RNN discussed in example 2.

As Example 1, the first step is to prepare the connection data file (*rnn_gen_con2.dat*). The contents of this file should be as shown.

```
% File Name           : rnn_gen_con2.dat
% Single Layer Recurrent RNN Model ( Association )
% -----
% Number of Neurons in the Input Layer   : 9 neurons
% Number of Neurons in the Output Layer  : 9 neurons

% Total Number of Neurons
9

% Connection Matrix for the Positive Weights ( wplus )
1 8 2 3 4 5 6 7 8 9
2 8 1 3 4 5 6 7 8 9
3 8 1 2 4 5 6 7 8 9
4 8 1 2 3 5 6 7 8 9
```

```

5 8 1 2 3 4 6 7 8 9
6 8 1 2 3 4 5 7 8 9
7 8 1 2 3 4 5 6 8 9
8 8 1 2 3 4 5 6 7 9
9 8 1 2 3 4 5 6 7 8

```

```
% Connection Matrix for the Negative Weights ( wminus )
```

```

1 8 2 3 4 5 6 7 8 9
2 8 1 3 4 5 6 7 8 9
3 8 1 2 4 5 6 7 8 9
4 8 1 2 3 5 6 7 8 9
5 8 1 2 3 4 6 7 8 9
6 8 1 2 3 4 5 7 8 9
7 8 1 2 3 4 5 6 8 9
8 8 1 2 3 4 5 6 7 9
9 8 1 2 3 4 5 6 7 8

```

```
% Number of Input Neurons
```

```
9 1 2 3 4 5 6 7 8 9
```

```
% Number of Output Neurons
```

```
9 1 2 3 4 5 6 7 8 9
```

The network parameters are defined in the network data file (**rnn_gen_net2.m**).

```
%% rnn_gen_net2.m
```

```
%% ##### Network Parameter Initialization #####
```

```

Mse_Threshold = 4e-7;           %Required stop mean square error value
Eta = 0.1;                      %Learning rate
N_Iterations = 600;            %Maximum number of iterations
R_Out = 0.1;                   %Firing rate of the output neurons
RAND_RANGE = 0.2;             %Range of weights initialization
FIX_RIN = 0;                   %DO NOT CHANGE (Related to RNN function approximation)
R_IN = 1;                      %DO NOT CHANGE (Related to RNN function approximation)
SAVE_Weights = 1;             %Flag to indicate if the weight will be automatically saved
N_Saved_Iterations = 10;      %Number of iterations after which weights will be saved automatically
Net_File_Name = 'rnn_gen_net2.m'; %Network file name (this file)
Connection_File_Name = 'rnn_gen_con2.dat'; %Connections file name
Weights_File_Name = 'rnn_gen_wts2.mat'; %Weights file name
Trn_File_Name = 'rnn_gen_trn2.m'; %Train data file name
Tst_File_Name = 'rnn_gen_tst2.m'; %Testing data file name
Log_File_Name = 'rnn_gen_log2.txt'; %results file name (ASCII format)
Temp_Log_File_Name = 'rnn_gen_log2.m'; %Results file name (MATLAB format)

```

```

SAVE_TEXT_LOG = 1;           %Flag to indicate if the results will be saved in ASCII format
SAVE_MFILE_LOG = 1;        %Flag to indicate if the results will be saved in MATLAB format
Res = 0.0001;              %Resolution of solving the non linear equations of the model
AUTO_MAPPING = 1;          %Flag = 0 for FF networks and 1 for fully recurrent networks

```

Note here that since we have the input and output neurons are the same, it is very important to set the parameter **AUTO_MAPPING** to the value 1. We predefine the the learning rate and the stopping MSE to 0.1 and 4×10^{-7} respectively.

finally, update the input-output patterns inside the main file (**use_rnn_gen2.m**).

```

% Input Patterns
X = [0.1 0.3 0.5 0.9 0.1 0.3 0.5 0.9 0.1];
% Output Patterns
Y = X;
% Train the recurrent RNN
err_result = train_rnn_gen('2',X,Y);

```

Now run the main file, by writing **use_rnn_gen2** from inside the MATLAB command window. After 255 epochs (54.82 sec.), the goal MSE is reached and the training is stopped.

The RNN is then tested, to check that the target equilibrium point vector is indeed contained in the network, by invoking the following function.

```

% Test the recurrent RNN
>> qq = test_rnn_gen('2', [0.1 0.3 0.5 0.9 0.1 0.3 0.5 0.9 0.1]);

```

The result will be

```

>> qq =
    0.10    0.30    0.50    0.90    0.10    0.30    0.50    0.90    0.10

```

The network is guaranteed to have a stable equilibrium point at the target vector since there is only one training pattern. On the other hand, if the network is trained on many different input patterns, there may be other spurious equilibrium points as well. We should mention here that using RNN in this auto-association mode can have many applications such as texture classification and automatic target recognition and these kind of applications need continuous processing of the images under consideration.

Once the network has been trained, it can be tested with one or more input vectors. Hopefully those input vectors close to target equilibrium point will find their target. We might try another input pattern that is not a design point, such as: [1 0.3 0.5 0.9 0.1 0.3 0.5 0.9 1]. This point is reasonably close to the design point, so one might anticipate that the network would converge to its attractor. To see if this happens, we will run the following code.

```
% Test the recurrent RNN
>> qqq = test_rnn_gen('2', [1 0.3 0.5 0.9 0.1 0.3 0.5 0.9 1]);
```

The result will be

```
>> qqq
    =
    0.4719  0.3556  0.5218  0.8926  0.1176  0.3127  0.5398  0.9479  0.4607
```

Note that we run the network test procedure for only one step and this might not be enough to achieve convergence. To proceed with another step, we may execute the network test function but we use the last output as the current input.

```
% Test the recurrent RNN
>> qqq = test_rnn_gen('2', qqq);
```

The result will be

```
>> qqq
    =
    0.2583  0.3519  0.5244  0.8935  0.1182  0.3157  0.5449  0.9565  0.2486
```

We notice that the second output is closer to the attractor vector than the first output. Instead of repeating this procedure until convergence is reached, the function `rnn_gen_test_exact` has been written to feed back the previous output vectors as inputs. These output vectors can be compared to the target vector to see how the solution is proceeding. The ability to run batches of trial input vectors quickly allows you to check the design in a relatively short time.

```
% Test the recurrent RNN iteratively
>> Probe = [0.1  0.3  0.5  0.9  0.1  0.3  0.5  0.9  0.1];
>> Num_Iter = 100;
>> Display = 0;
>> Distance = 0.001
>> [qqq i qqq_array] = rnn_gen_test_exact(Probe, Num_Iter, Display, Distance);
```

The result will be

```
>> qqq
      =
      0.1015  0.3013  0.5022  0.9044  0.1008  0.3015  0.5033  0.9060  0.1005
>> i_out =
      16
```

Here **Probe** is the input vector used for testing, **Num_Iter** is the input maximum number of iteration, **Display** is an input flag used to control displaying the intermediate outputs from the network, and **Distance** is a small value used for measuring the distance between the current output and the next output and hence The function returns three arguments, **qqq** is the final vector obtained after convergence, **i** is the number of iterations needed for convergence, and **qqq_array** is a (16×9) matrix with each row representing the current output from the network.

Testing the recurrent RNN requires solving the non linear equations of the model. This is a difficult task if one wants to implement the RNN in a dedicated hardware. We found out through extensive simulations of the model that the testing mode of the recurrent model can be carried out using the feed-forward routine (although the training mode uses the recurrent algorithm). This results in an easy hardware implementation of the model even in the case of recurrent structure. The function **rnn_gen_test_iterative** can be used for the iterative testing of the recurrent model without solving the non linear equations, it can be used as shown below.

```
% Test the recurrent RNN iteratively
>> Probe = [1  0.3  0.5  0.9  0.1  0.3  0.5  0.9  1];
>> Initial = [0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0];
>> Num_Iter = 100; Display_in = 0;
>> Display = 0;
>> Distance = 0.001
>> rnn_gen_test_iterative(Probe, Initial, Num_Iter, Display_in, Distance);
```

The result will be

```
>> qqq
      =
      0.1014  0.3010  0.5017  0.9029  0.1007  0.3011  0.5026  0.9046  0.1004
>> i
      =
      18
```

The input and output arguments for the function `rnn_gen_test_iterative` are the same as for the function `rnn_gen_test_exact` except that there is an extra input argument, **Initial**, which defines the initial state of the network before applying the external input (this input can be set to zeros but it is made variable to facilitate testing the performance of the network for various initial states). More details on using the training and testing functions are found in Appendix A.

3.3 Conclusion

In this Chapter, the **RNNSIM v.2** package is illustrated via two detailed examples. The examples were chosen to be very simple so that one can emphasize on the network design and training steps. Although the two given examples were illustrating feed-forward and fully recurrent structures, the RNN in general can have any type of connection depending on the application. For example for character recognition, the network can be defined to be fully recurrent and in texture classification, it is sufficient to just connect each neuron to its surrounding neighbors. The RNN recurrent training algorithm implemented in MATLAB is time consuming. It can be made much faster if the training and testing functions are implemented as executable MEX functions, in this case, we can use MATLAB as a frame work to invoke the MEX functions and visualize or save the network response.

Appendix A

RNNSIM v.2 Functions

1- train_rnn_gen

Function train_rnn_gen

Purpose train a recurrent random neural network.

Syntax MSE = train_rnn_gen(kind, INPUT, TARGET, num_iter)

To get help Type help train_rnn_gen

Description train_rnn_gen trains a RNN according to the parameters read from the file rnn_gen_net.m.

train_rnn_gen(kind, INPUT, TARGET, num_iter) takes,

kind - Character flag. When equals to '1', the training procedure will assume a feed-forward structure and when equals to '2', a recurrent structure will be assumed.

INPUT - Network inputs.

TARGET - Network targets.

num_iter - Number of iterations (optional).

and returns,

MSE - The average mean squared error.

The number of rows of INPUT and TARGET represent the number of training patterns. The number of columns of INPUT and TARGET represent the number of input and output neurons respectively.

Examples

```
>> X = [-1  1 -1 -1  1  1  1
        1 -1  1  1 -1 -1 -1];
```

```
>> Y = [1 0  
        0 1];
```

```
>> MSE = train_rnn_gen('1', X, Y, 1);
```

This example assumes a feed-forward RNN with 7 input neurons, 5 hidden neurons, and 2 output neurons. The number of neurons and the connection between them are defined in the connection data file (see example 1).

```
>> X = [0.1 0.3 0.5 0.9 0.1 0.3 0.5 0.9 0.1];
```

```
>> Y = X;
```

```
>> err_result = train_rnn_gen('2',X,Y);
```

This example assumes a 9 neuron recurrent RNN (see example 2).

It is recommended to perform the training process from inside the main file (use_rnn_gen.m) so that the error and weights can be automatically saved after each iteration and the stopping conditions can be continuously checked.

2- test_rnn_gen

Function test_rnn_gen

Purpose test a recurrent random neural network

Syntax RESULT = test_rnn_gen(kind, TEST_INPUT, r, Weights_File, TEST_File,
M_File)

To get help Type help test_rnn_gen

Description test_rnn_gen tests a RNN according to the parameters passed to it.

test_rnn_gen(kind, TEST_INPUT, r, Weights_File, TEXT_File, M_File)
takes

- kind - Character flag. When equals to '1', the testing procedure will assume a feed-forward structure and when equals to '2', a recurrent structure will be assumed.
- TEST_INPUT - Test patterns.
- r - Firing rate vector. This parameter is optional, if not passed to the function, it will be calculated.
- Weights_File - Weight file name. This file is assumed to contain the weights of the trained network. This parameter is optional, if not passed to the function, the current global network parameters will be used in testing.
- TEXT_File - Text file name. The results of testing the network will be saved in this file (optional).
- M_File - MATLAB file name. The results of testing the network will be saved in this file in MATLAB readable format (optional).

and returns,

- RESULT - The actual output vector from the network.

The number of rows of TEST_INPUT and RESULT represent the number of testing patterns. The number of columns of TEST_INPUT and RESULT represent the number of input and output neurons respectively.

Examples

```
>> X = [-1  1 -1 -1  1  1  1
        1 -1  1  1 -1 -1 -1];
```

```
>> Q = test_rnn_gen('1', X);
```

For a successfully trained feed-forward network this may give

```
>> Q
    =
    1.0000    0.0281
    0.0121    1.0000
```

This example tests a feed-forward RNN with 7 input neurons, 5 hidden neurons, and 2 output neurons. The number of neurons and the connection between them are defined in the connection data file (see example 1).

```
>> qq = test_rnn_gen('2', [0.1 0.3 0.5 0.9 0.1 0.3 0.5 0.9 0.1]);
```

For a successfully trained recurrent network this may give

```
>> qq
    =
    0.1  0.3  0.5  0.9  0.1  0.3  0.5  0.9  0.1
```

This example tests a 9 neuron fully recurrent RNN (see example 2).

The shown examples illustrate the simplest form of the function. The testing can be performed from inside the main file (use_rnn_gen.m) or from the MATLAB command window.

3- test_rnn_gen_iterative

Function `rnn_gen_test_iterative`

Purpose Test a recurrent RNN iteratively without solving the non linear equation of the model. Here the output at the current iteration is fed as the input of the next iteration in batch mode.

Syntax `[qqq, i, qqq_array] = rnn_gen_test_iterative(Probe, Initial, Num_Iter, Display, dist)`

To get help Type `help rnn_gen_test_iterative`

Description `test_rnn_gen_iterative` takes,

`Probe` - The external input vector used for testing.
`Initial` - which defines the initial state of the network before applying the external input.
`Num_Iter` - Maximum number of iterations.
`Display` - A flag when set to 1, the intermediate results will be displayed
`dist` - Used for measuring the distance between the current output and the next output and hence indicating the convergence progress.

and returns,

`qqq` - The final output vector obtained after convergence.
`i` - The number of iterations needed for convergence
`qqq_array` - A matrix in which each row represents the current output from the network.

Example

```
>> Probe = [1 0.3 0.5 0.9 0.1 0.3 0.5 0.9 1];
>> Initial = [0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0];
>> [qqq i qqq_array] = rnn_gen_test_iterative(Probe, Initial, ...
```

```
100, 0 ,0.001);
```

This will give

```
>> qqg
      =
0.1014  0.3010  0.5017  0.9029  0.1007  0.3011  0.5026  0.9046  0.1004

>> i
      = 18

>> qqg_array
      =
0.4712  0.1915  0.3361  0.8955  0.0570  0.1797  0.3405  0.8900  0.4467
0.2531  0.2887  0.4220  0.8883  0.0796  0.2256  0.4321  0.9480  0.2482
0.1622  0.3111  0.4667  0.8908  0.0909  0.2589  0.4742  0.9615  0.1631
0.1253  0.3118  0.4888  0.8945  0.0963  0.2795  0.4941  0.9588  0.1271
0.1107  0.3083  0.4992  0.8976  0.0987  0.2912  0.5034  0.9511  0.1121
0.1051  0.3053  0.5036  0.8997  0.0999  0.2975  0.5074  0.9427  0.1059
0.1030  0.3034  0.5051  0.9011  0.1004  0.3005  0.5088  0.9351  0.1033
0.1023  0.3024  0.5054  0.9020  0.1006  0.3019  0.5089  0.9287  0.1022
0.1020  0.3018  0.5050  0.9026  0.1008  0.3024  0.5084  0.9234  0.1016
0.1019  0.3016  0.5045  0.9029  0.1008  0.3025  0.5076  0.9191  0.1013
0.1018  0.3015  0.5040  0.9032  0.1008  0.3024  0.5067  0.9156  0.1011
0.1017  0.3014  0.5035  0.9033  0.1008  0.3022  0.5059  0.9129  0.1010
0.1016  0.3013  0.5031  0.9033  0.1008  0.3019  0.5052  0.9106  0.1008
0.1016  0.3012  0.5027  0.9032  0.1008  0.3017  0.5045  0.9088  0.1007
0.1015  0.3012  0.5024  0.9032  0.1008  0.3015  0.5039  0.9074  0.1006
0.1015  0.3011  0.5021  0.9031  0.1008  0.3014  0.5034  0.9062  0.1005
0.1014  0.3010  0.5019  0.9030  0.1008  0.3012  0.5030  0.9053  0.1005
0.1014  0.3010  0.5017  0.9029  0.1007  0.3011  0.5026  0.9046  0.1004
```

If the function is invoked without its output arguments as shown below,

```
>> rnn_gen_test_iterative(Probe, Initial, 100, 0, 0.001);
```

the patterns will be applied for the whole 100 iterations and the function will not check the convergence internally.

4- test_rnn_gen_exact

Function `rnn_gen_test_exact`

Purpose Test a recurrent RNN iteratively by solving the non linear equation of the model. Here the output at the current iteration is fed as the input of the next iteration in batch mode.

Syntax `[qqq, i, qqq_array] = rnn_gen_test_exact(Probe, Num_Iter, Display, dist)`

To get help Type `help rnn_gen_test_exact`

Description `test_rnn_gen_exact` takes,

`Probe` - The external input probe vector.

`Num_Iter` - Max Number of Iterations.

`Display` - A flag when set to 1, the intermediate results will be displayed.

`dist` - Used for measuring the distance between the current output and the next output and hence indicating the convergence progress.

and returns,

`qqq` - The final output vector obtained after convergence.

`i` - The number of iterations needed for convergence

`qqq_array` - A matrix in which each row represents the current output from the network.

Example

```
>> Probe = [1 0.3 0.5 0.9 0.1 0.3 0.5 0.9 1];
```

```
>> [qqq i qqq_array] = rnn_gen_test_exact(Probe, 100 , 0, 0.001);
```

This will give

```
>> qqq
```

```
=
```

```
0.1015 0.3013 0.5022 0.9044 0.1008 0.3015 0.5033 0.9060 0.1005
```

```

>> i
    = 16

>> qqq_array
    =
0.4719  0.3556  0.5218  0.8926  0.1176  0.3127  0.5397  0.9479  0.4607
0.2585  0.3522  0.5250  0.8943  0.1183  0.3160  0.5452  0.9572  0.2488
0.1698  0.3377  0.5221  0.8976  0.1146  0.3151  0.5402  0.9537  0.1633
0.1322  0.3249  0.5179  0.9005  0.1105  0.3127  0.5328  0.9464  0.1282
0.1158  0.3160  0.5140  0.9026  0.1073  0.3102  0.5260  0.9389  0.1133
0.1086  0.3104  0.5109  0.9040  0.1051  0.3080  0.5204  0.9322  0.1069
0.1052  0.3069  0.5086  0.9048  0.1035  0.3062  0.5160  0.9265  0.1040
0.1036  0.3048  0.5069  0.9053  0.1026  0.3049  0.5127  0.9218  0.1025
0.1027  0.3036  0.5056  0.9055  0.1019  0.3039  0.5102  0.9181  0.1018
0.1022  0.3028  0.5047  0.9055  0.1015  0.3032  0.5083  0.9150  0.1013
0.1020  0.3023  0.5040  0.9053  0.1013  0.3027  0.5069  0.9126  0.1010
0.1018  0.3020  0.5034  0.9052  0.1011  0.3023  0.5058  0.9106  0.1009
0.1017  0.3017  0.5030  0.9050  0.1010  0.3020  0.5049  0.9091  0.1007
0.1016  0.3016  0.5027  0.9048  0.1009  0.3018  0.5042  0.9078  0.1006
0.1015  0.3014  0.5024  0.9046  0.1009  0.3016  0.5037  0.9068  0.1005
0.1015  0.3013  0.5022  0.9044  0.1008  0.3015  0.5033  0.9060  0.1005

```

If the function is invoked without its output arguments as shown below,

```

>> rnm_gen_test_exact(Probe, 100, 0 ,0.001);

```

the patterns will be applied for the whole 100 iterations and the function will not check the convergence internally.

Bibliography

- [1] E. Gelenbe, "Random neural networks with negative and positive signals and product form solution," *Neural Computation*, vol. 1, no. 4, pp. 502-511, 1989.
- [2] E. Gelenbe, "Stability of the random neural network model," *Neural Computation*, vol. 2, no. 2, pp. 239-247, 1990.
- [3] E. Gelenbe, "Learning in the recurrent random neural network," *Neural Computation*, vol. 5, no. 1, pp. 154-164, 1993.
- [4] Y. Feng, Adaptive Processing of Multimedia: Image Understanding, Video Compression and Communication, *Ph.D Thesis, Duke University*, 1997.
- [5] E. Gelenbe, V. Koubi, and F. Pekergin, "Dynamical random neural network approach to the travelling sales man problem," *Elektrik*, vol. 2, no. 1, pp. 1-9, April 1994.
- [6] Anoop Ghanawani, "A Qualitative comparison of neural network models applied to the vertex covering problem," *Elektrik*, vol. 2, no. 1, pp. 11-18, 1994.
- [7] V. Atalay and E. Gelenbe, and N. Yalabik, "The random neural network model for texture generation," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 6, no. 1, pp. 131-141, 1992.
- [8] E. Gelenbe, T. Feng, K. R. R. Krishnan, "Neural Network methods for volumetric magnetic resonance imaging of the human brain," *Proceedings of the IEEE*, vol. 84, no. 10, pp. 1488-1496, October 1996.
- [9] C. Cramer, E. Gelenbe, and H. Bakircioglu, "Low bit rate video compression with neural networks and temporal sampling," *Proceedings of the IEEE*, vol. 84, no. 10, pp. 1529-1543, October 1996.
- [10] V. Atalay and E. Gelenbe, "Parallel algorithm for color texture generation using the random neural network model," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 6, no. 2, pp. 437-446, 1992.
- [11] E. Gelenbe, Z. H. Mao, and Y. D. Li, "Function approximation with spiked random networks," *IEEE Trans. on Neural Networks*, vol. 10, no. 1, pp. 1-9, 1999.
- [12] S. Abe, "Global convergence performance and suppression of spurious states of the Hopfield neural network," *IEEE Transaction on Circuits and Systems*, vol. 40, no. 4, pp. 245-257, 1990.

- [13] E. Gelenbe, and M. Sungur, "Random network learning and image compression," *Proc. of the Int. Conf. on Artificial Neural Networks*, pp. 3996-3999, 1994.