International Journal of Forecasting 17 (2001) 305–322

www.elsevier.com/locate/ijforecast

Software reviews

The aim of this section is to provide objective information to guide choices in both scientific and business applications. Anyone wishing to contribute a review should write to the editor at the address below. The suggested outline for a software review includes the following topics: identification of the target user, equipment requirements, data entry and editing capability, evaluation of graphics and other data analysis features, a synopsis of modelling options, the validity of computations, quality of documentation and ease of learning and use. Standard *IJF* instructions to authors apply.

Editor: B.D. McCullough
Dept. of Decision Sciences
LeBow College of Business
Drexel University
Philadelphia, PA 19104
USA

1. Environment

Neural Network Toolbox 3.0 for use with MATLAB™

The Mathworks, Inc., 24 Prime Park Way, Natick, MA 01760-1500, USA. Tel.: +1-508-647-7000. Fax: +1-548-647-7001. Sales, pricing, and general information: info@mathworks.com. http://www.mathworks.com. Microsoft Windows, UNIX, and Macintosh versions are available. Neural Network Toolbox 3.0 requires MATLAB.

Neural Network Toolbox authors

Professor Emeritus Howard Demuth, University of Idaho, and Mark Beale, President of MHB, Inc.

Related products

Neural Network Toolbox User's Guide, Howard Demuth and Mark Beale. Fifth Printing. Version 3. Mathworks, MA, 1998; info@mathworks.com.

Neural Network Design, Martin T. Hagan, Howard B. Demuth, and Mark Beale. PWS Publishing Company, 1996; info@pws.com

2. Introduction

There is a variety of numerical techniques for modelling and prediction of nonlinear time series such as the threshold model, exponential model, nearest neighbors regression and neural network models. In addition, the Taylor series expansion, radial basis function and nonparametric kernel regression are also used for nonlinear prediction. These techniques essentially involve interpolating or approximating unknown functions from scattered data points. Among these techniques, artificial neural networks is one of the most recent techniques used in nonlinear modelling and prediction. A recent survey of this literature is presented in Kuan and White (1994).

In feedforward networks, signals flow in only one direction, without feedback. Applications in forecasting, signal processing and control require explicit treatment of dynamics. Feedforward networks can accommodate dynamics by including past input and target values in an augmented set of inputs. Gençay and Dechert (1992) and Gençay (1996) used feedforward networks in estimating the Lyapunov exponents of an unknown system. Gençay (1994, 1999) used feedforward networks in predicting noisy time series and in foreign exchange predictions. Among many applications, Dougherty and Cobbett (1997) model inter-urban traffic forecasts using neural networks; Callen, Clarence, Patrick and Yufei (1996) model quarterly accounting earnings; Hill, Marquez, O'Connor and Remus (1994) focus on forecasting and decision making; Kim and Se (1998) construct probabilistic networks to model stock market activity; Kirby, Watson and Dougherty (1997) use neural networks for short term traffic forecasting; Swanson and White (1997) study modelling and forecasting economic time series; and Refenes (1994) comments on the field of neural networks.

Our interest in this review is confined to the function approximation and filtering capabilities of various neural network models.

The Neural Network Toolbox (NNT) is one of several toolboxes the Mathworks offers. The company states that the NNT is a comprehensive environment for neural network research, design and simulation with the MATLAB. For the NN experts, the key features of the NNT are classified as follows:

- Supervised network paradigms: perceptron, linear network, backpropagation, Levenberg–Marquardt (LM) and reduced LM algorithms, conjugate gradient, radial basis, Elman, Hopfield, learning vector quantization (LVQ), probabilistic network, generalized regression, and quasi-Newton algorithm.
- Unsupervised network paradigms: Hebb, Kohonen, competitive, feature maps and self-organizing maps.
- Unlimited number of sets of inputs and network interconnections.
- Customizable architecture and network functions.
- Modular network representation.
- Automatic network regularization.
- Competitive, limit, linear and sigmoid transfer functions.

In this paper, we review the numerical accuracy and the robustness of some Matlab NNT commands. The sections are organized such that each section corresponds to a NNT chapter. In this review, we study Chapters 2, 3 and 4 with specific examples which are comparable to the examples given in the Matlab NNT.[1] We conclude afterwards.

3. Chapter 2

Chapter 2 of the user's guide contains basic material about the network architectures. The chapter has seven examples. Each example states a problem, shows the network used to solve the problem, and presents the results. We replicate six of these examples with slightly different input sets.

3.1. Example 1

The static network is the simplest case among all classes of network simulation as there are no feedbacks or delays in the system. Given a set of weights for each input and assuming zero

[1]Matlab codes of the examples are available at www.bilkent.edu.tr/~faruk.

bias, a well designed NN should result in a weighted sum of the input set. The first example of Chapter 2 on page 2-15 (concurrent inputs in a static network) demonstrates this property. In the example, four concurrent vectors are presented into the static network created with *newlin* command. Given preset weights and bias, the *sim* command simulates the network and produces an output. However, when we utilized the same network with a different input set, the reported results on the screen were completely wrong, although the results in the program memory were correct. After some experiments, we found that the NNT gives the correct answer on the screen if one uses a different format for the output, a peculiar solution to avoid a misleading result in a high caliber program like MATLAB.

The problem of misleading results on the screen is not specific to a concurrent input static network. The example of incremental training with static networks given on page 2-20 also produces misleading results if the default format is not changed to ‘format long e’. The lesson of our very first two experiments with the NNT is a clear one: *to avoid misleading results, change the default format ‘short’ to ‘long e’ before you start using the NNT.*

3.1.1. Concurrent inputs in a static network (page 2-15)

This example creates a two-element input linear layer with one neuron. The ranges of inputs are [−100000 100000] and [−100000 100000]. The input weights are 0.00001 and 100000. The bias is 0.

Input

$$P_1 = \begin{pmatrix}0.00001\\100000\end{pmatrix},\quad P_2 = \begin{pmatrix}0.00001\\0.00001\end{pmatrix},$$

$$P_3 = \begin{pmatrix}100000\\0.00001\end{pmatrix},\quad P_4 = \begin{pmatrix}100000\\100000\end{pmatrix}.$$

Output

$$\begin{aligned} A_1 &= WP_1 + b \\ &= [0.00001\ 100000]\begin{pmatrix}0.00001\\100000\end{pmatrix} + 0 \\ &= (1.0e + 010) + (1.0e - 010) \end{aligned}$$

$$\begin{aligned} A_2 &= WP_2 + b \\ &= [0.00001\ 100000]\begin{pmatrix}0.00001\\0.00001\end{pmatrix} + 0 \\ &= (1.0e + 000) + (1.0e - 010) \end{aligned}$$

$$\begin{aligned} A_3 &= WP_3 + b \\ &= [0.00001\ 100000]\begin{pmatrix}100000\\0.00001\end{pmatrix} + 0 \\ &= 2.0e + 000 \end{aligned}$$

$$\begin{aligned} A_4 &= WP_4 + b \\ &= [0.00001\ 100000]\begin{pmatrix}100000\\100000\end{pmatrix} + 0 \\ &= (1.0e + 010) + 1. \end{aligned}$$

The code

```
net=newlin([-100000 100000;-100000 100000],1);
net.IW{1,1}=[0.00001 100000];
net.b{1}=0;
P=[0.00001 0.00001 100000 100000; 100000 0.00001 0.00001 100000];
A=sim(net,P)
```

Results

```
A =1.0e+010 *
1.0000 0.0000 0.0000 1.0000
```

Comments

By hand calculation, the correct answer is

$$(1.0e + 010) + (1.0e - 010), (1.0e + 000) + (1.0e - 010), 2.0e + 000, (1.0e + 010) + 1.$$

With the default format, the results on the screen are misleading. To have the correct answer displayed, the format must be set to ‘format long e’.

3.2. Example 2

The training methods of a network can be classified into two groups: *incremental* training and *batch* training. In incremental training, the weights and biases of the network are updated each time an input is presented to the network. On the other hand, the batch training updates all the weights and biases *after* the entire input set is presented.

An example on page 2-20 presents a case for incremental training with static networks. First, the learning rate of the system is set to 0.0 to show that ‘‘if you do not ask the network to learn, it will not learn’’. As expected, the network outputs are zero since there is no ‘learning’, e.g. the weights and biases in the network are not updated at all. Later, the learning rate is set to 0.1 to show that the system, in fact, learns. Now, the first output from the network is zero (since there is no updating with the first input). After the second input is presented, the second output differs from zero, although the error is large. With the third and fourth inputs, the errors get smaller and smaller giving the impression that the system is in fact ‘learning’. According to the example, the weights continue to be modified as each error is computed. The authors claim that ‘‘if the network is capable and the learning rate is set correctly, the error will eventually be driven to zero’’.

In order to check the validity of this claim, we presented the same input and the same target values several times to the same network in the example. We expected that after a reasonable number of inputs, the network would learn and the errors ‘‘would eventually be driven to zero’’. Although there were some improvements in terms of the mean squared error, the results were far from satisfactory. When we input the *same set of numbers 24 times*, the mean squared error was still different from zero.[2] We think that the NNT manual should give detailed explanations and warnings regarding the importance of the choice of the learning rate.

[2]We even tried the same set of numbers 48 times. The mean squared error was still different from zero.

3.2.1. Incremental training with static networks (page 2-20)

This example creates one two-element input linear layer with one neuron. The ranges of inputs are: [1 3] and [1 3]. The input delay is 0. The learning rate is 0.1. We train the network to create a linear function

$$t = 2p_1 + p_2$$

where p_1 and p_2 refer to inputs.

The inputs are

$$P_1 = \begin{pmatrix} p_1 \\ p_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}, \quad P_2 = \begin{pmatrix} p_1 \\ p_2 \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \end{pmatrix},$$
$$P_3 = \begin{pmatrix} p_1 \\ p_2 \end{pmatrix} = \begin{pmatrix} 2 \\ 3 \end{pmatrix}, \quad P_4 = \begin{pmatrix} p_1 \\ p_2 \end{pmatrix} = \begin{pmatrix} 3 \\ 1 \end{pmatrix}.$$

The target is

$$t_1 = 2*1 + 2 = 4, \quad t_2 = 2*2 + 1 = 5,$$
$$t_3 = 2*2 + 3 = 7, \quad t_4 = 2*3 + 1 = 7.$$

The code

```
1.
net=newlin([1 3;1 3],1,0,0.1);
net.IW{1,1}=[0 0];
net.b{1}=0;
P={[1;2][2;1][2;3][3;1]};
T={4 5 7 7};
[net,a,e,pf]=adapt(net,P,T)
2.
net=newlin([1 3;1 3], 1 , 0, 0.1);
net.IW{1,1}=[0 0];
net.b{1}=0;
P={[1;2] [2;1] [2;3] [3;1] [1;2] [2;1] [2;3] [3;1]};
T={4 5 7 7 4 5 7 7};
[net,a,e,pf]=adapt(net,P,T)
3.
net=newlin([1 3;1 3], 1 , 0, 0.1);
net.IW{1,1}=[0 0];
net.b{1}=0;
P={[1;2] [2;1] [2;3] [3;1] [1;2] [2;1] [2;3] [3;1][1;2] [2;1] [2;3] [3;1]};
T={4 5 7 7 4 5 7 7 4 5 7 7};
[net,a,e,pf]=adapt(net,P,T)
4.
net=newlin([1 3;1 3], 1 , 0, 0.1);
net.IW{1,1}=[0 0];
net.b{1}=0;
P={[1;2] [2;1] [2;3] [3;1] [1;2] [2;1] [2;3] [3;1][1;2] [2;1] [2;3] [3;1][1;2]
[2;1] [2;3] [3;1] [1;2] [2;1] [2;3] [3;1] [1;2] [2;1] [2;3] [3;1][1;2] [2;1]
[2;3] [3;1] [1;2] [2;1] [2;3] [3;1][1;2] [2;1] [2;3] [3;1][1;2] [2;1] [2;3]
[3;1] [1;2] [2;1] [2;3] [3;1][1;2] [2;1] [2;3] [3;1][1;2] [2;1] [2;3] [3;1]
[1;2] [2;1] [2;3] [3;1][1;2] [2;1] [2;3] [3;1][1;2] [2;1] [2;3] [3;1] [1;2]
[2;1] [2;3] [3;1][1;2] [2;1] [2;3] [3;1][1;2] [2;1] [2;3] [3;1] [1;2] [2;1]
[2;3] [3;1][1;2] [2;1] [2;3] [3;1][1;2] [2;1] [2;3] [3;1] [1;2] [2;1] [2;3]
[3;1][1;2] [2;1] [2;3] [3;1]};
T={4 5 7 7 4 5 7 7 4 5 7 7 4 5 7 7 4 5 7 7 4 5 7 7 4 5 7 7 4 5 7 7 4 5 7 7
4 5 7 7 4 5 7 7 4 5 7 7 4 5 7 7 4 5 7 7 4 5 7 7 4 5 7 7 4 5 7 7 4 5 7 7 4 5
7 7 4 5 7 7 4 5 7 7 4 5 7 7 4 5 7 7 4 5 7 7};
[net,a,e,pf]=adapt(net,P,T);
mse(e)
```

Results

```
1.
a = [0] [2] [6.0000] [5.8000]
e = [4] [3] [1.0000] [1.2000]
mse=6.86
2.
a = [0] [2] [6.0000] [5.8000] [5.5200] [4.8000] [7.3920] [5.9760]
e = [4] [3] [1.0000] [1.2000] [-1.5200] [0.2000] [-0.3920] [1.0240]
mse=3.8741
3.
a = [0] [2] [6.0000] [5.8000] [5.5200] [4.8000]
[7.3920] [5.9760] [4.9696] [4.9408] [7.0419] [6.5261]
e = [4] [3] [1.0000] [1.2000] [-1.5200] [0.2000]
[-0.3920] [1.0240] [-0.9696] [0.0592] [-0.0419] [0.4739]
mse=2.6802
4.
mse=0.3756
```

Comments

From the result of a and e, there is no clear-cut evidence that, when using the adapt function, the output will close to the target and the error will eventually be driven to zero. The MSE gets smaller when the same set of inputs are supplied a greater number of times. This is some evidence of improvement. However, when we use the same numbers 96 times (fourth case), the mean squared error is still different from zero.

3.3. Example 3

Unlike incremental training in which weights and biases are updated each time, the batch training updates the weights and biases after all the inputs are presented to the system. This training can also be used in static and dynamic networks. An example on page 2-23 shows an application of batch training with static networks.

We adopted the same approach with slightly modified input and output settings. Particularly, we defined the target values from the following linear function:

$t = 3p + 8$

where p is the input. In the first two cases, we set the input weights and bias all to be zero and we obtained the network outputs to be all zero, because the weights are not updated until all of the training set is presented. This result is not unexpected. After presenting the entire data set in the second case, we expected that the resulting input weight should be 3 and the bias should be 8 since the linear function used to train the network is $t = 3p + 8$. The updated input weights and bias are far from these values even if we present the same set of numbers more and more times. In the third case, we set the input weight to 3 and bias to 8. This time the output is equal to the target and the input weights and bias are all correct. Fine. In the last case, we make a small change in input weights and bias as compared to the third experiment and set them to 2 and 6. The updated input weights and bias are completely off from what we would expect. The experiment shows that if one gives the correct input weights and bias to the system, the network does not diverge from these correct values. However, the network does not converge to the correct set if given input weights and biases are slightly different than the true set. The researcher may obtain an incorrect answer and not know it. Here, we expect that the Matlab NNT would provide robustness and stability benchmarks to the researchers.

3.3.1. Batch training with static networks (page 2-23)

This example creates a single input linear layer with one neuron. The range of input is [−500000 500000]. The input delays are 0 and 1. The learning rate is 0.1. The function trained is

$t = 3p + 8$

where p refers to inputs. The inputs are

$P_1 = 1, \quad P_2 = 0.0001, \quad P_3 = 10000,$
$P_4 = -500, \quad P_5 = 3000000,$
$P_6 = -0.00003.$

The target is

$t_1 = 3*1 + 8 = 11,$

$t_2 = 3*0.0001 + 8 = 8.0003,$

$t_3 = 3*10000 + 8 = 30008$

$t_4 = 3*(-500) + 8 = -1492,$

$t_5 = 3*3000000 + 8 = 9000008,$

$t_6 = 3*(-0.00003) + 8 = 7.99991.$

The code

```
1.
format long e
net=newlin([-1000 3000000],1,0,0.1);
net.IW{1,1}=[0];
net.b{1}=0;
P=[1 0.0001 10000 -500 3000000 -0.00003];
T=[11 8.0003 30008 -1492 9000008 7.99991];
[net,a,e,pf]=adapt(net,P,T)
2.
format long e
net=newlin([-1000 3000000],1,0,0.1);
net.IW{1,1}=[0];
net.b{1}=0;
P=[1 0.0001 10000 -500 3000000 -0.00003 1 0.0001 10000 -500 3000000 -0.00003
1 0.0001 10000 -500 3000000 -0.00003 1 0.0001 10000 -500 3000000 -0.00003
1 0.0001 10000 -500 3000000 -0.00003 1 0.0001 10000 -500 3000000 -0.00003];
T=[11 8.0003 30008 -1492 9000008 7.99991 11 8.0003 30008 -1492 9000008 7.9999
11 8.0003 30008 -1492 9000008 7.99991 11 8.0003 30008 -1492 9000008 7.99991
11 8.0003 30008 -1492 9000008 7.99991 11 8.0003 30008 -1492 9000008 7.99991];
[net,a,e,pf]=adapt(net,P,T)
3.
format long e
net=newlin([-1000 3000000],1,0,0.1);
net.IW{1,1}=[3];
net.b{1}=8;
P=[1 0.0001 10000 -500 3000000 -0.00003];
T=[11 8.0003 30008 -1492 9000008 7.99991];
[net,a,e,pf]=adapt(net,P,T)
4.
format long e
net=newlin([-1000 3000000],1,0,0.1);
net.IW{1,1}=[2];
net.b{1}=6;
P=[1 0.0001 10000 -500 3000000 -0.00003];
T=[11 8.0003 30008 -1492 9000008 7.99991];
[net,a,e,pf]=adapt(net,P,T)
```

Results

```
1.0
a =0 0 0 0 0 0
e =Columns 1 through 3
1.100000000000000e+001 8.000299999999999e+000 3.000800000000000e+004
Columns 4 through 6
-1.492000000000000e+003 9.000008000000000e+006 7.999910000000000e+000
net.IW{1,1}=2.700032482601100e+012 net.b{1}=9.028551000210000e+005
2.
a =Columns 1 through 12
0 0 0 0 0 0 0 0 0 0 0 0
Columns 13 through 24
0 0 0 0 0 0 0 0 0 0 0 0
Columns 25 through 36
0 0 0 0 0 0 0 0 0 0 0 0
net.IW{1,1}=1.620019489560660e+013
net.b{1}=5.417130600126000e+006
3.
a =Columns 1 through 3
1.100000000000000e+001 8.000299999999999e+000 3.000800000000000e+004
Columns 4 through 6
-1.492000000000000e+003 9.000008000000000e+006 7.999910000000000e+000
e = 0 0 0 0 0 0
net.IW{1,1}=3
net.b{1}=8
4.
a =Columns 1 through 3
8.000000000000000e+000 6.000200000000000e+000 2.000600000000000e+004
Columns 4 through 6
-9.940000000000000e+002 6.000006000000000e+006 5.999940000000000e+000
e =Columns 1 through 3
3.000000000000000e+000 2.000099999999999e+000 1.000200000000000e+004
Columns 4 through 6
-4.980000000000000e+002 3.000002000000000e+006 1.999970000000000e+000
net.IW{1,1}=9.000106269023001e+011
net.b{1}=3.009573000070000e+005
```

Comments

This example once again demonstrates that the estimated network weights are highly unstable if the starting network values are not chosen to be their actual values. In real data applications, the underlying function and its parameters are unknown so that this instability has to be addressed.

3.4. Example 4

The batch training with the dynamic networks example on page 2-25 uses a linear network with a delay. We adopted the same example with a different setting. Specifically, the linear function training the network is defined as

$$t_{m+1} = 1 - 100000t_m.$$

Therefore, we expect that the input weights from the network should be -100000 and 1 and the resulting bias should be 1. Again, with a learning rate of 0.02 as in the original example, the results of the network are far from being satisfactory. When we change the learning rate to 0.000000000000001, the results are closer to what they should be. Note that a researcher normally does not know the training function. As a result, setting the correct learning rate may not be obvious in practice. The NNT does not provide any guidance on this matter.

3.4.1. Batch training with dynamic networks (page 2-25)

This example creates a single input linear layer with one neuron. The range of input is $[-100\ 100000000]$. The input delays are 0 and 1. The learning rate is 0.02. The linear function training this network is

$$t_{m+1} = 1 - 100000t_m.$$

The inputs are

$$P_1 = 0.01, \quad P_2 = -999, \quad P_3 = 99900001.$$

When $P = t_0 = 0.01$

$$t_1 = 1 - 100000*0.01 = -999,$$
$$t_2 = 1 - 100000*(-999) = 99900001,$$
$$t_3 = 1 - 100000*(99900001)$$
$$= -999000009999.$$

The code

```
1.
net=newlin([-1000 100000000],1,[0 1],0.02);
net.IW{1,1}=[0 0];
net.biasConnect=0;
net.trainParam.epochs=1;
Pi={1};
P={0.01 -999 99900001};
T={-999 99900001 -999000009999};
net=train(net,P,T,Pi)
2.
net=newlin([-1000 10000],1,[0 1],0.000000000000001);
net.IW{1,1}=[0 0];
net.biasConnect=0;
net.trainParam.epochs=1;
Pi={1};
P={-0.01 -999 99900001};
T={-999 99900001 -999000009999};
net=train(net,P,T,Pi)
```

Results

```
1.
net.IW{1,1}=-1.9960e+018 1.9960e+013
net.b{1}=[]
2.
net.IW{1,1}=1.0e+004 *
-9.9800 0.0001
net.b{1}=[]
```

Comments

The linear function training this network is $t_{m+1} = 1 - 100000t_m$, so the IW should be -100000 and 1, and b should be 1. In the first case, when we use a learning rate of 0.02 for the training, the results of input weights and bias are far from it. In the second case, when we use a learning rate of 0.000000000000001, the results are closer to what they should be. We conclude that if the network is capable and the learning rate is set correctly, it gets the correct output. In most cases, we do not know the training function and setting the appropriate learning rate may not be obvious. Therefore, we question whether this procedure produces accurate answers in real situations.

4. Chapter 3

A single layer network with a hard limit transfer function is called a perceptron. Chapter 3 introduces perceptrons and shows the advantages and limitations of them in solving different problems. After creating a perceptron and setting its initial weights and biases, one can check whether the network responds as expected or not. This is done in the NNT with the *sim* command. After checking the integrity of a perceptron with *sim*, it can be trained with a desired learning rule.

In general, a *learning rule* or *a training algorithm* is a procedure for modifying the weights and biases of a network. The NNT provides learning rules which can be classified into two groups: supervised learning and unsupervised learning.

In *supervised learning*, the learning rule is introduced to the network with a training set. In this algorithm, as the inputs are introduced to the system the output of the network is compared to the targets in the training set. The learning rule is then used to adjust the weights and biases of the network. A learning rule might be ‘minimum error’, ‘minimum mean squared error’, ‘minimum absolute error’, or some other criterion depending on the problem in hand.

In *unsupervised learning*, the weights and biases are adjusted only as a response to inputs and there are no targets.

The perceptron learning rule in the NNT, *learnp*, is a supervised learning rule. It has an objective of minimizing the error between the input and the target. If simulation *sim* and perceptron learning rule *learnp* are used repeatedly, the perceptron will eventually find input weight and bias values which solve the problem. Each presentation of input and targets to the system is called a ‘pass’. The NN toolbox provides another command, *adapt*, which performs these repetitive steps with a desired number of passes.

4.1. Example 1

First, we created a perceptron layer with one two-element input and one neuron. After defining our inputs and targets, we let the network adapt for one pass through sequence. The network performed successfully.

4.1.1. Adaptive training

This example creates a perceptron layer with one two-element input and one neuron. The ranges of inputs are [−10000 10000] and [−10000 10000]. Here we define a sequence of targets *t*, and then let the network adapt for one pass through the sequence.

The code

```
net=newp([-10000 10000;-10000 10000],1);
net.adaptParam.passes=1;
P={[-0.0001;10000][10000;-10000][-10000;0.0001][-0.0001;0.0001]};
t={0 1 1 1 };
[net,a,e]=adapt(net,P,t)
```

Results

```
a = [1] [1] [0] [1]
e = [-1] [0] [1] [0]
net.IW{1,1}=-9.999999900000001e+003 -9.999999900000001e+003
net.b{1}= 0
```

Comments

The network performs successfully.

4.2. Example 2

Now we create a perceptron layer with one three-element input and one neuron. First, we applied *adapt* for one pass through the sequence of all four input vectors and obtained the weights and bias. Another run with two passes resulted in correct answers. Our experiment is in accord with the claim in the handbook: *adapt* will converge in a finite number of steps unless the problem presented cannot be solved with a simple perceptron. However, it would be convenient for users with large data sets if *adapt* has an option which decides on the number of passes automatically.

4.2.1. Adaptive training

This example creates a perceptron layer with one three-element input and one neuron. The ranges of inputs are [−10000 10000], [−10000 10000] and [−10000 10000].

The code

```
1.
net=newp([-10000 10000;-10000 10000;-10000 10000],1);
net.adaptParam.passes=1;
P={[5;-10000;10000][-0.0001;10000;30][10000;0.0001;-5][10000;-10000;0.0001]};
t={1 0 0 1};
[net,a,e]=adapt(net,P,t)
2.
net=newp([-10000 10000;-10000 10000;-10000 10000],1);
net.adaptParam.passes=1;
P={[5;-10000;10000][-0.0001;10000;30][10000;0.0001;-5][10000;-10000;0.0001]};
t={1 0 0 1};
[net,a,e]=adapt(net,P,t);
A=sim(net,P)
3.
net=newp([-10000 10000;-10000 10000;-10000 10000],1);
net.adaptParam.passes=2;
P={[5;-10000;10000][-0.0001;10000;30][10000;0.0001;-5][10000;-10000;0.0001]};
t={1 0 0 1};
[net,a,e]=adapt(net,P,t)
4.
net=newp([-10000 10000;-10000 10000;-10000 10000],1);
net.adaptParam.passes=2;
P={[5;-10000;10000][-0.0001;10000;30][10000;0.0001;-5][10000;-10000;0.0001]};
t={1 0 0 1};
[net,a,e]=adapt(net,P,t);
A=sim(net,P)
```

Results

```
1.
a = [1] [1] [1] [0]
e = [0] [-1] [-1] [1]
net.IW{1,1}=9.999999929277692e-005 -2.000000010000000e+004 -2.499990000000000e+001
net.b{1}=-1
2.
A = [1] [0] [1] [1]
3.
a = [1] [0] [1] [1]
e = [0] [0] [-1] [0]
net.IW{1,1,1}=-9.999999900000001e+003 -2.000000020000000e+004 -1.999990000000000e+001
net.b{1}=-2
4.
A = [1] [0] [0] [1]
```

Comments

The network performs successfully *if* the number of passes is set correctly.

5. Chapter 4

Perceptrons introduced in Chapter 3 are very simple classification networks and they have very limited usage in practice. Adaptive Linear Neuron Networks (ADALINE) are different than perceptrons as they have a linear transfer function rather than a hard limiting function. The toolbox uses the Least Mean Squares learning rule for ADALINE. Particularly, the function *newlind* provides specific network values for weights and biases by minimizing the mean least squares. In other words, *newlind* designs a linear network given a set of inputs and corresponding outputs. The resulting network can be used for simulation purposes. Our experiments with *newlind* showed that it performs well even under some extreme situations.

5.1. Example 1

In this example, we design a network with *newlind* and check its performance. We found that the network performs successfully.

5.1.1. Linear system design (NEWLIND)

In this example, for given P and T, we use *newlind* to design a network and check its response.

The inputs are

$$P_1 = 0.00004, \quad P_2 = 100000,$$
$$P_3 = -30, \quad P_4 = 0.002, \quad P_5 = -50000.$$

When we train the network to create a linear function

$t = 0.001p + 5$

the outputs are

$$t_1 = 0.001{*}0.00004 + 5 = 5.00000004,$$
$$t_2 = 0.001{*}100000 + 5 = 105,$$
$$t_3 = 0.001{*}(-30) + 5 = 4.97,$$
$$t_4 = 0.001{*}0.002 + 5 = 5.000002,$$
$$t_5 = 0.001{*}(-50000) + 5 = -45.$$

When we train the network to create a linear function

$t = 1000p - 300$

the outputs are

$$t_1 = 1000{*}0.00004 - 300 = 299.96,$$
$$t_2 = 1000{*}100000 - 300 = 99999700,$$
$$t_3 = 1000{*}(-30) - 300 = -30300,$$
$$t_4 = 1000{*}0.002 - 300 = -298,$$
$$t_5 = 1000{*}(-50000) - 300 = -50000300.$$

The code

```
1.
format long e
P=[0.00004 100000 -30 0.002 -50000];
T=[5.00000004 105 4.97 5.000002 -45];
net=newlind(P,T);
Y=sim(net,P)
2.
format long e
P=[0.00004 100000 -30 0.002 -50000];
T=[-299.96 99999700 -30300 -298 -50000300];
net=newlind(P,T);
Y=sim(net,P)
```

Results

```
1.
Y =Columns 1 through 3
5.000000039999999e+000 1.050000000000000e+002 4.969999999999999e+000
Columns 4 through 5
5.000001999999999e+000 -4.500000000000000e+001
2.
Y =Columns 1 through 3
-2.999600000037561e+002 9.999969999999999e+007 -3.030000000000375e+004
Columns 4 through 5
-2.980000000037562e+002 -5.000030000000000e+007
```

Comments

The network performs successfully.

5.2. Example 2

The *train* function introduced earlier is explained for the ADALINE environment in Chapter 4. The function *train* takes each vector of a set of vectors and calculates the network weights and bias increments due to each of the inputs by utilizing the function *learnp*. The network is then adjusted with the sum of all these corrections. The *train* continues with the application of the inputs to the new network and calculates the outputs and mean squared errors. If the error goal is met or the number of preset echos is reached, the training is stopped. In our second example in this section, we utilized the same input and target values we used in our first example in the previous section. The utilized code is the same as the example on page 4-14. With our input and target set, the function *train* could not obtain a goal of 0.1. The results were reported as 'Not a Number' so we were not able to get the new weights and bias. The network simply stopped for no apparent reason. Note that the adaptive training with the same input and target set in Chapter 3 with perceptrons produced the correct results. Examples of these types of failures should be provided and reasons behind them be explained in the NNT manual.

5.2.1. Linear classification (TRAIN)

In this example, we use train to get the weights and biases for a network that produces the correct targets for each input vector. The initial weights and bias for the new network will be 0 by default. We set the error goal to 0.1 rather than accept its default of 0. The inputs and targets are the same as in Example 1 of Chapter 3.

The code

```
format long e
net=newlin([-10000 10000;-10000 10000],1);
P=[-0.0001 10000 -10000 -0.0001; 10000 -10000 0.0001 0.0001];
t=[0 1 1 1];
net.trainParam.goal=0.1;
[net,tr]=train(net,P,t)
```

Results

```
TRAINWB, Epoch 0/100, MSE 0.75/0.1.
TRAINWB, Epoch 25/100, MSE Inf/0.1.
TRAINWB, Epoch 50/100, MSE NaN/0.1.
TRAINWB, Epoch 75/100, MSE NaN/0.1.
TRAINWB, Epoch 100/100, MSE NaN/0.1.
TRAINWB, Maximum epoch reached.
net.IW{1,1}=NaN NaN
net.b{1}=NaN
```

Comments

The network cannot achieve the value of 0.1. The new weights and bias cannot be obtained either. The network cannot attain a numerical solution. Increasing the number of epochs does not change the non-numerical solution. At least in this case the user is not misled with an incorrect answer.

5.3. Example 3

Adaptive Filtering (ADAPT) is one of the major applications of ADALINE in practise. The output of an adaptive filtering is a simple weighted average of current and lagged (delayed) inputs. Therefore, the output of the filter is given by

$$\begin{aligned} a(k) &= purelin(Wp + b) \\ &= \sum_{i=1}^{R} W_{1,i}\, a(k - i + 1) + b. \end{aligned}$$

Our third example shows that the NNT performs well in this respect.

5.3.1. Adaptive filter

In this example, the input values have a range of −10000 to 10000. The delay line is connected to the network weight matrix through delays of 0, 1, 2, 3 and 4. The input weights are 0.07, −8000, 90, −6 and 0.4. The bias is 0. We define the initial values of the outputs of the delays as: pi={1 0.2 −100 50}.

The inputs are

$$P_1 = -30000, \quad P_2 = 0.0004,$$
$$P_3 = 500, \quad P_4 = 6.$$

The outputs are

$$\begin{aligned} a_1 &= 0.07*(-30000) + (-8000)*50 \\ &\quad + 90*(-100) + (-6)*0.2 + 1*0.4 \\ &= -411100.8 \end{aligned}$$

$$\begin{aligned} a_2 &= 0.07*0.0004 + (-8000)*(-30000) \\ &\quad + 90*50 + (-6)*(-100) + 1*0.2 \\ &= 240005100.2 \\ a_3 &= 0.07*500 + (-8000)*0.0004 \\ &\quad + 90*(-30000) + (-6)*50 + 1*(-100) \\ &= -2700368.2 \\ a_4 &= 0.07*6 + (-8000)*500 + 90*0.0004 \\ &\quad + (-6)*(-30000) + 1*50 \\ &= -38199495.544. \end{aligned}$$

The code

```
format long e
net=newlin([-10000 10000],1);
net.input weights{1,1}.delays=[0 1 2 3 4];
net.IW{1,1}=[0.07 -8000 90 -6 0.4];
net.b{1}=[0];
pi={1 0.2 -100 50};
p={-30000 0.0004 500 6};
[a,pf]=sim(net,p,pi)
```

Results

```
a = Columns 1 through 2
[-4.111008000000000e+005] [2.400051000800280e+008]
Columns 3 through 4
[-2.700308200000000e+006] [-3.819979544000000e+006]
pf = [-30000] [4.000000000000000e-004] [500] [6]
```

Comment

The network performs successfully.

5.4. Example 4

The network defined in Example 3 above can be trained with the function *adapt* to produce a particular output sequence. In our last example, we would like the network to produce the sequence of values 2999999995, 45, 50000005 and 600005. The network completely fails to produce the desired output, resulting in large errors even after 10 passes. This again demonstrates that NNT is not robust to the large input ranges and lacks numerical stability.

5.4.1. Adaptive filter

In this example, we would like the previous network to produce the sequence of values 2999999995, 45, 50000005 and 600005.

The code

```
format long e
net=newlin([-10000 10000],1);
net.input weights{1,1}.delays=[0 1 2 3 4];
net.IW{1,1}=[0.07 -8000 90 -6 0.4];
net.b{1}=[0];
pi={1 0.2 -100 50};
p={-30000 0.0004 500 6};
T={2999999995 45 50000005 600005};
net.adaptParam.passes=10;
[net,y,E,pf,af]=adapt(net,p,T,pi)
```

Results

```
y = Columns 1 through 2
[-1.458181921117053e+151] [-2.205426985469178e+155]
Columns 3 through 4
[-3.320985501570783e+159] [-4.990802594185386e+163]
E = Columns 1 through 2
[1.458181921117053e+151] [2.205426985469178e+155]
Columns 3 through 4
[3.320985501570783e+159] [4.990802594185386e+163]
pf = [-30000] [4.000000000000000e-004] [500] [6]
net.IW{1,1}=Columns 1 through 3
3.011086479645423e+162 2.495400635597510e+164 -9.960959081107002e+161
Columns 4 through 5
-1.497240612228395e+166 2.495069198586660e+163
net.b{1}=4.991134714791272e+161
```

Comment

The network errors are large and the network outputs are wrong.

6. Conclusions

The results on the screen as a result of the default format are very misleading. Also, our replications of simple examples from the NNT guide with slightly different input sets led to incorrect results and raised questions on the reliability of the toolbox.

In incremental training, setting the learning rate is a crucial step. The user guide should emphasize this point and should give detailed examples of the importance of the learning rate. The current version gives the impression that ‘some’ learning rate will be sufficient to obtain correct results. Our experiments with the example in the guide show that this is not the case, leaving an untrained user with the impression that the software is not functioning properly.

In batch training, we found that the network does not converge to the correct set if given input weights, and biases are slightly different than the true set. Since the true weight and bias cannot be known in practice, NNT should provide robustness and stability benchmarks to researchers.

A simple example utilizing the *train* function in the adaptive linear neuron network environment showed that the designed network can stop without any apparent reason. When we run an example with the function *adapt*, the network completely failed, indicating that NNT is not robust to large input ranges.

A study of the first three chapters of the NNT toolbox does not provide incentive for a trained researcher to utilize the neural network methodology by exploring its very rich capabilities in a simple, structured framework. After observing the capabilities of this toolbox in simple problems, we are not convinced of its numerical stability and robustness. We lost confidence in the toolbox after the first three chapters and did not proceed with more advanced topics.

Acknowledgements

We are grateful to B.D. McCullough for comments on an earlier draft and Jian Gao for research assistance. Ramazan Gençay gratefully acknowledges financial support from the Natural Sciences and Engineering Research Council of Canada and the Social Sciences and Humanities Research Council of Canada.

References

Callen, J. L., Clarence, K. C. Y., Patrick, C. Y., & Yufei, Y. (1996). Neural network forecasting of quarterly accounting earnings. *International Journal of Forecasting 12*, 475–482.

Dougherty, M. S., & Cobbett, M. R. (1997). Short-term inter-urban traffic forecasts using neural networks. *International Journal Forecasting 13*, 21–31.

Gençay, R. (1994). Nonlinear prediction of noisy time series with feedforward networks. *Physics Letters A 187*, 397–403.

Gençay, R. (1996). A statistical framework for testing chaotic dynamics via Lyapunov exponents. *Physica D 89*, 261–266.

Gençay, R. (1999). Linear, nonlinear and essential foreign exchange rate prediction with simple technical trading rules. *Journal of International Economics 47*, 91–107.

Gençay, R., & Dechert, W. D. (1992). An algorithm for the *n* Lyapunov exponents of an *n*-dimensional unknown dynamical system. *Physica D 59*, 142–157.

Hill, T., Marquez, L., O'Connor, M., & Remus, W. (1994). Artificial neural network models for forecasting and decision making. *International Journal of Forecasting 10*, 5–15.

Kuan, C. -M., & White, H. (1994). Artificial neural networks: an econometric perspective. *Econometric Reviews 13*, 1–91.

Kim, S. H., & Se, H. C. (1998). Graded forecasting using an array of bipolar predictions: application of probabilistic neural networks to a stock market index. *International Journal of Forecasting 14*, 323–337.

Kirby, H. R., Watson, S. M., & Dougherty, M. S. (1997). Should we use neural networks or statistical models for short-term motorway traffic forecasting? *International Journal of Forecasting 13*, 43–50.

Refenes, A. N. (1994). Comments on neural networks: 'forecasting breakthrough or passing fad' by C. Chatfield. *International Journal of Forecasting 10*, 43–46.

Swanson, N. R., & White, W. H. (1997). Forecasting economic time series using flexible versus fixed specification and linear versus nonlinear econometric models. *International Journal of Forecasting 13*, 439–461.

Ramazan Gençay[a,*]
Faruk Selçuk[b]
[a]*Department of Economics*
University of Windsor
401 Sunset, Windsor
Ontario N9B 3P4
Canada

[b]*Department of Economics*
Bilkent University, Bilkent
Ankara 06533
Turkey

*Corresponding author. Tel.: +1-519-253-3000, extn. 2382; fax: +1-519-973-7096.
E-mail address: gencay@uwindsor.ca (R. Gençay).

PII: S0169-2070(01)00084-X

DecisionTime 1.0 and WhatIf? 1.0: SPSS, Inc., Marketing Department, 233 South Wacker Drive, 11th Floor, Chicago, IL 60606-6307. Tel.: +1-312-651-3000; fax: +1-312-651-3668; http://www.spss.com. List price: Single user license for DecisionTime and WhatIf? US$ 1,999, additional single user license of WhatIf? US$ 399 (North America only); System requirements: Windows 95, Windows 98, or Windows NT 4.0, 32 MB RAM, 486DX or higher processor, 30 MB disk space, SVGA Monitor, Math co-processor, CD-ROM drive.

1. Background and methods

Founded in 1968, SPSS has since been a major player in data analysis software. The base program, now in its tenth edition, supplies a comprehensive menu of statistical techniques with tabulation, graphing and reporting capabilities. Specialty add-on modules extend the base program capability in data collection, modeling and presentation. The *Trends* module, introduced in 1994 to serve practitioners of time series forecasting, expanded the SPSS-family modeling capability to include autoregression, ARIMA and some exponential smoothing techniques. SPSS Trends is one of 15 forecasting programs evaluated in the Tashman and Hoover (2001) (TH) chapter of the new *Principles of Forecasting Handbook* from Kluwer Academic Publishers (2001). The authors found SPSS Trends to be far less effective in implementing principles of forecasting than is the time series module offered in SAS/ETS. You can view the chapter and summary tables at the Principles of Forecasting website – hops.wharton.upenn.edu/forecast.

With the emergence of DecisionTime and What-If?, SPSS is attempting to enter the mainstream market of dedicated business-forecasting software. DecisionTime is a standalone product that does not require the SPSS base program. In this market, it joins established players such as Autobox, Forecast Pro, Smartforecasts and tsMetrix. These programs offer in varying degrees exponential smoothing, ARIMA, regression, intervention/event modeling, and, importantly, an ‘‘expert system’’ for automatic forecasting of a time series. Automatic forecasting is in fact the central mission of DecisionTime as the forecaster is given little direction in understanding how to build or choose models on his own.

The methodological mix in DecisionTime offers several interesting twists. These are: (a) the incorporation of ARIMA with explanatory variables (ARIMAX) into the automatic forecasting system; (b) a remarkably simple but effective procedure for modeling special events, including outliers; and (c) through its What-If? companion, the opportunity to explore the effects of alternate assumptions about future values of the predictor variables on the forecasts of the dependent variable. The first feature is presently available in Autobox but without the full exponential smoothing component. The WhatIf? functionality is really a macro that improves the packaging and presentation of scenarios that could be directly examined, albeit more crudely, in a spreadsheet.

DecisionTime includes a batch processing capability, called a *production job*, which makes