Journal of Modern Applied Statistical Methods

Volume 19 | Issue 1 Article 19

9-28-2021

JMASM 55: MATLAB Algorithms and Source Codes of 'cbnet' Function for Univariate Time Series Modeling with Neural Networks (MATLAB)

Cagatay Bal
Muğla Sitki Kocman University, Turkey, cagataybal@mu.edu.tr

Serdar Demir
Muğla Sitki Kocman University, Turkey, serdardemir@mu.edu.tr

Journal of Modern Applied Statistical Methods
May 2020, Vol. 19, No. 1, eP2928
doi: 10.22237/jmasm/1608553080

ISSN 1538 − 9472

ALGORITHMS & CODE

JMASM 55: MATLAB Algorithms and Source Codes of 'cbnet' Function for Univariate Time Series Modeling with Neural Networks (MATLAB)

Cagatay Bal
Mugla Sitki Kocman Univ.
Muğla, Turkey

Serdar Demir
Mugla Sitki Kocman Univ.
Muğla, Turkey

Artificial Neural Networks (ANN) can be designed as a nonparametric tool for time series modeling. MATLAB serves as a powerful environment for ANN modeling. Although Neural Network Time Series Tool (`ntstool`) is useful for modeling time series, more detailed functions could be more useful in order to get more detailed and comprehensive analysis results. For these purposes, `cbnet` function with properties such as input lag generator, step-ahead forecaster, trial-error based network selection strategy, alternative network selection with various performance measure and global repetition feature to obtain more alternative network has been developed, and MATLAB algorithms and source codes has been introduced. A detailed comparison with the `ntstool` is carried out, showing that the `cbnet` function covers the shortcomings of `ntstool`.

Keywords: Artificial neural networks, MATLAB algorithms and codes, time series modeling

Introduction

Time series is data ordered through a time-dependent structure which has unique characteristics within time lags. An assumption is autocorrelation, which assumes that time series model contains the correlation between any given lag observations or intervals [1]. Therefore, at the modeling aspect, dependent variables can be created from the lags of the time series which is shown at Table 1 below.

doi: 10.22237/jmasm/1608553080 | Accepted: Aug. 24, 2018; Published: Sep 28, 2021.
Correspondence: Cagatay Bal, cagataybal@mu.edu.tr

Table 1. Data representation

***Y*(*t*)**	*Y*1	*Y*2	…	…	*Y*(*n*−*p*−1)	***Y*(*n*−*p*)**
***Y*(*t*−1)**	*Y*2	*Y*3	…	…	*Y*(*n*−*p*)	***Y*(*n*−*p*+1)**
***Y*(*t*−2)**	*Y*3	*Y*4	…	…	*Y*(*n*−*p*+1)	***Y*(*n*−*p*+2)**
***Y*(*t*−3)**	*Y*4	*Y*5	…	…	*Y*(*n*−*p*+2)	***Y*(*n*−*p*+3)**
…	…	…	…	…	…	**…**
***Y*(*t*−*k*)**	*Y*(*k*+1)	*Y*(*k*+2)	…	…	*Y*(*n*−1)	***Y*(*n*)**

In Table 1, lags obtained from actual time series Y(t) will be used to generate input matrix. Generated input matrix with given lag intervals can be set as an input layer parameter for neural network. After utilizing the rest of the parameters and neural structure, network will be ready for training and modeling given forecasting task.

Neural networks can be described as nonlinear nonparametric method (Zhang, Patuwo, and Hu, 1998). Like as the most of methods that exist in the literature, neural networks have both advantages and disadvantages. No assumptions, alternative solutions, goal-driven characteristics and parameter tunability can be counted as its advantages. Poor generalizability, data-focused characteristics, unpromising optimal solution in every trial and expertise-based structures can be count as its disadvantages. These features restrained the efforts to develop intelligent strategies and trial-error method has been accepted widely for finding the best solution in the neural networks concept.

MATLAB is well-known and widely accepted software by engineers, researchers, students and companies from all around the world. MATLAB also serves as a useful environment for modelling and data processing tasks. Among its many toolboxes, `ntstool` has been developed for focusing time series analyses. We focus here on this particular toolbox and explain its advantages and disadvantages along with reasons to develop a specialized function `'cbnet'` for univariate time series analysis.

Neural Networks

Neural networks consist of three main components as architecture, learning algorithm and activation functions (Eğrioğlu, Aladağ, and Günay, 2008). Architecture of a neural network can be described as the layered visualization scheme (Figure 1) and should be resolved according to the task. Layer and neuron numbers, data preparations and data partitions are the parameters which could be considered within the architecture.

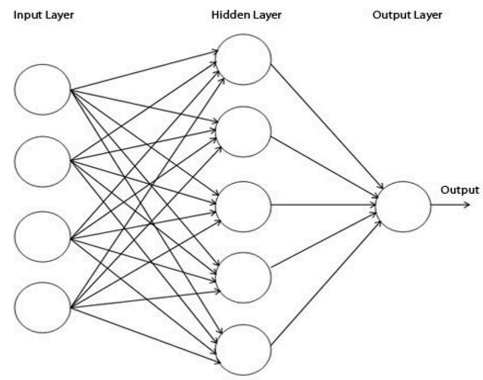


Figure 1. Artificial Neural Network Architecture

The Learning algorithm is the component of training process which makes neural networks learn from data lags in this case. Back propagation algorithms are the most widely used learning algorithms for neural networks. Learning algorithms of neural networks are basically backpropagation algorithms that uses error functions derivatives as gradients. General framework of backpropagation can be described as follows

$$E_p = \Sigma_k \left(d_k - x_k\right)^2 \tag{1}$$

An Error function E_p defines error between d_k target value and x_k output value of k^{th} neuron in network.

$$\overline{\in}_i = \frac{\partial^+ E}{\partial \overline{x}_i} \tag{2}$$

The Gradient vector of errors $\overline{\in}_i$ will be obtained for i^{th} neuron in network.

$$\overline{\in}_i = \begin{cases} -2\left(d_i - x_i\right)\dfrac{\partial x_i}{\partial \overline{x}_i} = -2\left(d_i - x_i\right)x_i\left(1 - x_i\right), \text{ if } i\, i^{\text{th}} \text{neuron is output neuron} \\ \dfrac{\partial x_i}{\partial \overline{x}_i} = \sum_{j,i<j} \dfrac{\partial^+ E}{\partial \overline{x}_j}\dfrac{\partial \overline{x}_j}{\partial x_i} = x_i\left(1 - x_i\right)\sum_{j,i<j} \overline{\in}_j \; w_{ij}, \text{ otherwise} \end{cases} \tag{3}$$

In equation (3), w_{ij} is weights between i^{th} and i^{th} neuron in network. If this value equals zero then that means there is no connection between i^{th} and i^{th} neuron in network.

$$\Delta w_{ki} = -\eta \frac{\partial^{+} E_p}{\partial w_{ki}} = -\eta \frac{\partial^{+} E_p}{\partial \overline{x}_i} \frac{\partial \overline{x}_i}{\partial w_{ki}} = -\eta \in_j x_k \tag{4}$$

In equation (4), η is described as learning ratio which affect convergence speed and the stability of the weights in the learning process. Bias can be update in the same way as equation (4). All weights are obtained after each iteration in training process.

$$\Delta w_{ki} = -\eta \frac{\partial^{+} E}{\partial w_{ki}} = -\eta \sum_p \frac{\partial^{+} E_p}{\partial w_{ki}} \tag{5}$$

$$\Delta w = -\eta \frac{\partial^{+} E}{\partial w} = -\eta \nabla_w E \tag{6}$$

In equation (5) and (6), weights updating through error gradients $E = \boldsymbol{\Sigma}_p E_p$ is described and the gradients will be calculated throughout the data set.

Data are then typically divided into two sets. The training process is mostly done with training set of the data and test set will remain for testing the networks performance for later steps of the evaluation. Another data partition approach is dividing data into three sets and the third set is called validation and the purpose of using validation set is to prevent over-fitting by stopping the training process when the conditions are satisfied. But using validation set might cause under-fitting (Prechelt, 1998).

Activation functions are linear and non-linear components of neural networks. They are responsible for mapping between input and target values. The S-shaped sigmoidal functions such as tangent-sigmoid and logistic-sigmoid functions are widely used as activation functions because of their non-linear mapping ability within the hidden layer of network. Output layer usually contains linear activation functions such as step and *purelin* functions.

Tangent and logistic sigmoidal functions are,

$$f_{\text{logistic-sigmoid}}\left(x\right) = \frac{1}{1+e^{x}} \tag{7}$$

$$f_{\text{tangent-sigmoid}}\left(x\right)=\frac{e^{x}-e^{-x}}{e^{x}+e^{-x}} \tag{8}$$

Purelin and step functions are,

$$f_{\text{purelin}}\left(x\right)=x \tag{9}$$

$$f_{\text{step}}\left(x\right)=\begin{cases}0 & \text{if } x\in \text{Output1}\\ 1 & \text{if } x\in \text{Output2}\end{cases} \tag{10}$$

Performance measures are used for model selection which is very important step after error-trial process with many alternative networks (Bal and Demir, 2017). Performance measures as MSE (Mean Square Error), RMSE (Root Mean Square Error), MAE (Mean Absolute Error) and MAPE (Mean Absolute Percentage Error) are given below,

$$MSE=\frac{1}{n}\sum_{j=1}^{n}\left(y_j-\hat{y}_j\right)^2 \tag{11}$$

$$RMSE=\sqrt{\frac{1}{n}\sum_{j=1}^{n}\left(y_j-\hat{y}_j\right)^2} \tag{12}$$

$$MAE=\frac{1}{n}\sum_{j=1}^{n}\left|y_j-\hat{y}_j\right|^2 \tag{13}$$

$$MAPE=\frac{1}{n}\sum_{j=1}^{n}\left|\frac{y_j-\hat{y}_j}{y_j}\right|^2 \tag{14}$$

There are many strategies for avoiding over-fitting which is the worst scenario for neural network. Performance measuring through the test set is a useful way to select the promising network among other alternatives.

Properties of `cbnet` Function

Data Partition

The data partition is an essential matter for ANN modeling. Basic approach is dividing the data into two sets. The first set is used for training and is generally called the training set and the second set is used for testing and is generally called the test set. Another approach is dividing the data into three sets and the third set, called the validation set, is used after training process for validating the training error. Both dividing procedures are essential in order to avoid over-fitting for model selection through test set performance.

MATLAB generates `dividerand` or `divideblock` functions for data partition in related neural network tool. `Dividerand` function is able to divide data perfectly through user's demand. However, it creates the vector indices randomly which is not quite accurate for time series. `Divideblock` function seems suitable for time series while it generates indices time-related but the lack of adjustability abilities can't fit the user's demand perfectly. For instance; users may or may not need the validation partition, since `dividerand` allows non-validation partition setup, `divideblock` doesn't. Therefore, modifying `dividerand` function with generating time-related indices instead of randomly will solve the problem completely.

Below the MATLAB codes and screen shots (Figure 2) are given for modified `dividerand` function and example of data partition are shown.

```
open dividerand % Opening the dividerand.m file from its original location.

allInd = 1:Q; % Modify the 105th row `allInd=randperm(Q);` of the dividerand.m file with this code.
```

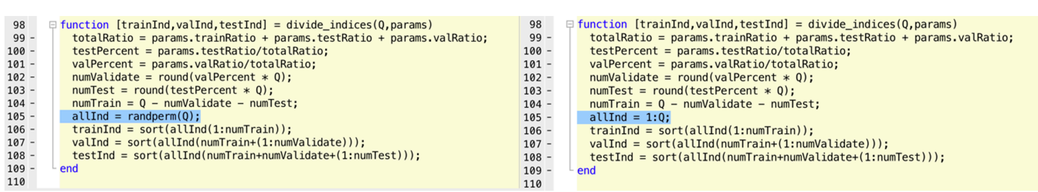


Figure 2. Screen shots for modified `dividerand` function.

3Generating Input Lag Matrixes

As long as the trial-error method requires alternative models to consider and compare, input matrixes for given lag intervals should be calculated before training process. Input matrixes and target vectors must be matched irreproachably for given time lag in order to achieve correct model construction. Below the MATLAB codes and screen shots (Figure 3) are given as a result of how input matrixes and target vectors are generating. Example of 10 lags and screenshots of 5 lagged results are shown.

```
function  lag(filename,imn)
%
% Function for creating input matrixes and target vectors.
%
% filename; name of the variable for time series vector in the same folder
%
% imn; input matrix number
%
datavector=cell2mat(struct2cell(load(filename)));
n=length(datavector);
        for p=1:imn
                for i=1:p
                        for j=1:i
                                inputvector=datavector(j:n-(p-(j-1)));
                                input{1,j}=inputvector;
                        end
                        input{2,p}(i,:)=input{1,j};
                end
                data{p,1}=input{2,p};
                data{p,2}=datavector(p+1:n);
        end
        save('lags.mat','data');
end
```

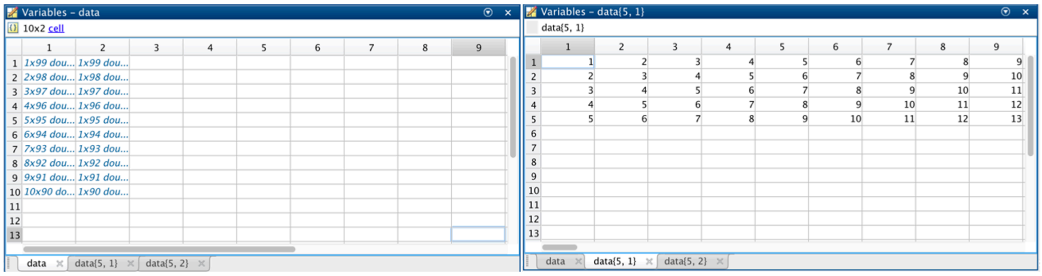


Figure 3. Screen shots of input matrixes and target vectors.

Forecasting

Forecasting the future values with trained ANN is important future in order to make networks as useful tools to benefit. The forecasting process algorithm is given below.

Step 1. Desired number of step-ahead forecasts; ***f***

Step 2. Obtaining number of neurons in input layer of trained network; ***n***

Step 3. Last ***n*** observations of target vector will be set as ***test vector***

Step 4. 1-step-ahead forecast will be obtained by using trained network as output of initialized ***test vector***

Step 5. 1-step-ahead forecast will save as first observation of ***forecast vector***

Step 6. ***Test vector*** will be updated by adding the 1-step-ahead at the end of ***test vector*** and removing the first observation. After updating process, ***test vector*** will remain same size and consist of forecast values.

Step 7. Step 4, 5 and 6 initialize ***f*** times to calculate forecast values.

Forecasting process codes for MATLAB are given below.

Results

Selection of the best neural network architecture can be based on different criteria that exist in literature. In this study, MSE, RMSE, MAE, and MAPE which are well-known criteria added to the codes in order to offer different results to the users. Results of each run collecting into `allResults` array for these four different criteria. These results are follows as input, hidden and output neuron number, test set error value of performance measure, vector of forecast values, input and target matrixes, test set vector, and trained network as MATLAB object for future use. After completion of every run `allResults` will contain the best architecture with minimum error for each run. The best architecture among all runs will be shown as the last step of the process. The codes for presentation of results will be given below.

`cbnet` Algorithm

In this study, a specialized ANN function named `cbnet` for MATLAB will be used. It has the ability to analyze univariate time series with the principle of error and trial method. Function `cbnet` has 11 parameters which are designed for data selection, number of lags for input matrix, maximum number of neuron numbers in hidden layer, training function, epoch number, activation function, training set ratio, validation set ratio, test set ratio, number of step ahead forecast and repetition number of the whole process respectively. For all parameters except the data selection, default values are predetermined for the `cbnet` function for practicality. The workflow of `cbnet` can be summarized as follows,

Step 1. Parameters are initializing according to given or default values.

Step 2. Input matrix is obtaining by using `lag` function.

Step 3. The first run starts with nested loops to generate total number of architectures with given parameter values or 100 possible architectures will be generated by the default values of 10 for `imn` and `maxhid` parameters.

Step 4. MATLAB function `feedforwardnet` is using to create network and after initializing the parameters, network training starts with the given inputs.

Step 5. After the training process, output of network calculated and test set partition separated from outputs for performance measuring.

Step 6. Performance measuring is carrying out via MSE, RMSE, MAE and MAPE. Errors of related measures will be checked with last achieved errors and the first assigned value is `1e100` for easy comparison.

Step 7. If current architecture is having less error than previous one, new FFNN is saving as the new best model. Otherwise previous FFNN will continue to be the best model until the process done.

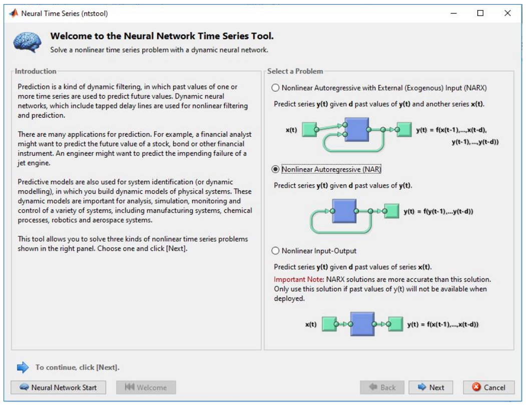
Step 8. Desired number of step ahead forecasts will be calculated by FFNN which is selected by each performance measure.

Step 9. If user demanded multi run for the evaluation, step 3 to 8 repeated with given repetition number.

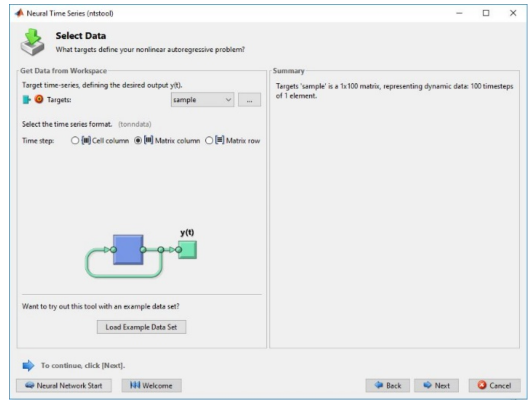
Step 10. Finally, all results will be saved in an array variable to workspace of MATLAB and summary of the best selected model among every run with important explanatories are given as table in command window.

Comparison of `ntstool` and `cbnet`

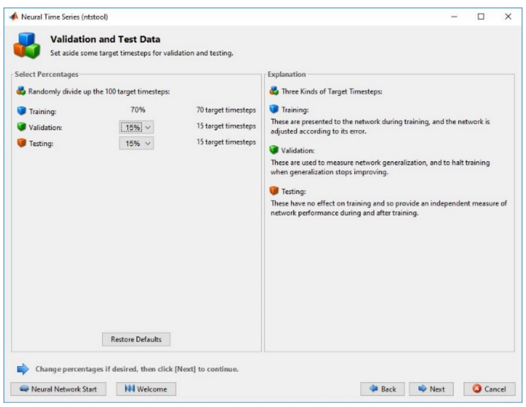
In this section, the workflow of `ntstool` with descriptions will be given and detailed comparisons with `cbnet` will be utilized with terms of how many steps needs to get the results, how much effort does it needs to end the analysis, pros and cons and satisfaction level of diversity of the results. This comparison is only to show detailed workflow of both `cbnet` and `ntstool` therefore dataset is not having a crucial role. The example dataset for this task is a linear vector values from 1 to 100 for simple utilization. Sample dataset is also recommended for tutorial of `cbnet` in the description of the `cbnet` function inside its .m file.



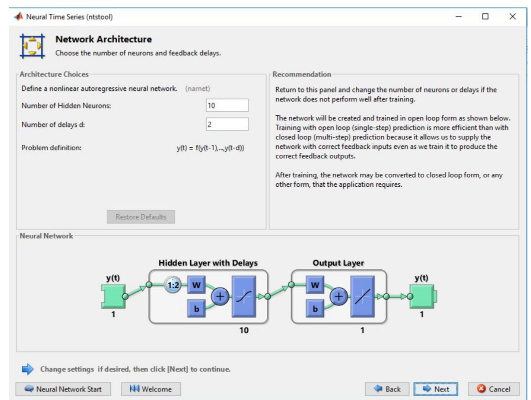
1. Choose NAR on `ntstool` first window



2. Select sample data set



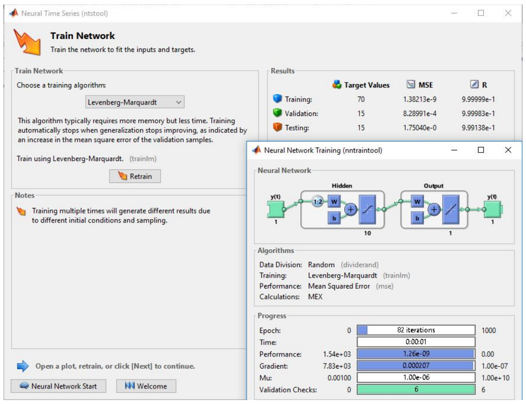
3. Set data partitions



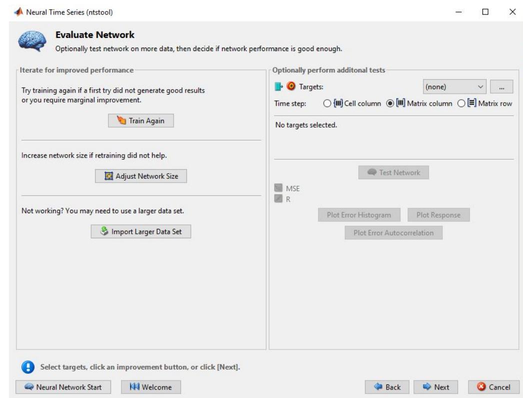
4. Define neuron number and number of delay

Figure 4. MATLAB `ntstool` workflow step by step (cont'd next page)

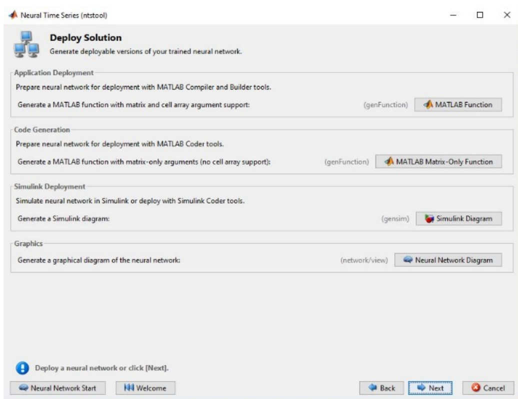
As noted in Figure 4, a total of 8 steps are required to train a network with given data set. `ntstool` has three choices for times series analysis: nonlinear autoregressive with external (exogenous) input (NARX), Nonlinear Autoregressive (NAR) and Nonlinear Input-Output. These options use MATLAB network functions `narxnet`, `narnet` and `timedelaynet` respectively. Each network functions are based on `feedforwardnet` MATLAB network function with their own modification to adapt the purpose of analyses for each case. In `ntstool`, these functions share the same default properties such as number of hidden neurons, number of delays, training function and the most importantly the data partition options. Levenberg-Marquadt Backpropagation is default training algorithm for `ntstool` along with 2 other function to select and for hidden layer activation



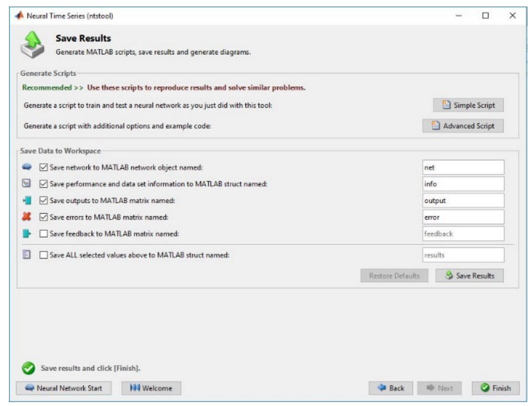
5. Train network and get MSE values



6. Retrain network or forward to next step



7. Generate various deployments of network



8. Save the results

Figure 4 ***(cont'd)*****.** MATLAB `ntstool` workflow step by step.

---

function, tangent-sigmoid function has been set as default (unfortunately there is not any other function offered) which can be seen at neural network diagram at Step 7. Data partition ratios are set default as 70% for training, 15% for validation and 15% for testing. Unfortunately, there is no option for eliminating the validation set partition which is required to offer users whether or not to use validation method as a stopping strategy for training process when network's generalization ability stops improving. This feature alone shows the `ntstool`'s inadequate preferences. Researchers may or may not need the usage of validation which is clearly dictated at Step 2 and also using validation partition will decrease the size of dataset for training and testing which could be a disadvantage such as increasing the variability of the estimates which could weaken the out-of-sample performance of network for multi-step ahead forecasting tasks (Faraway, 1992).

The data partition is made by using divider and function as default in `ntstool` which is randomly divide data into sets. The point to be noted is that the random partition is irrational in time series analysis because the data must remain sequential throughout the analysis. Also, random division may affect the network negatively and imbalanced input-target mapping will decrease the performance thus the step-ahead forecasting abilities of network. Below in Figure 5, it can be seen in another trial with the same sample data that the high testing error and poor output predictions proving the possible danger of random division of time series.

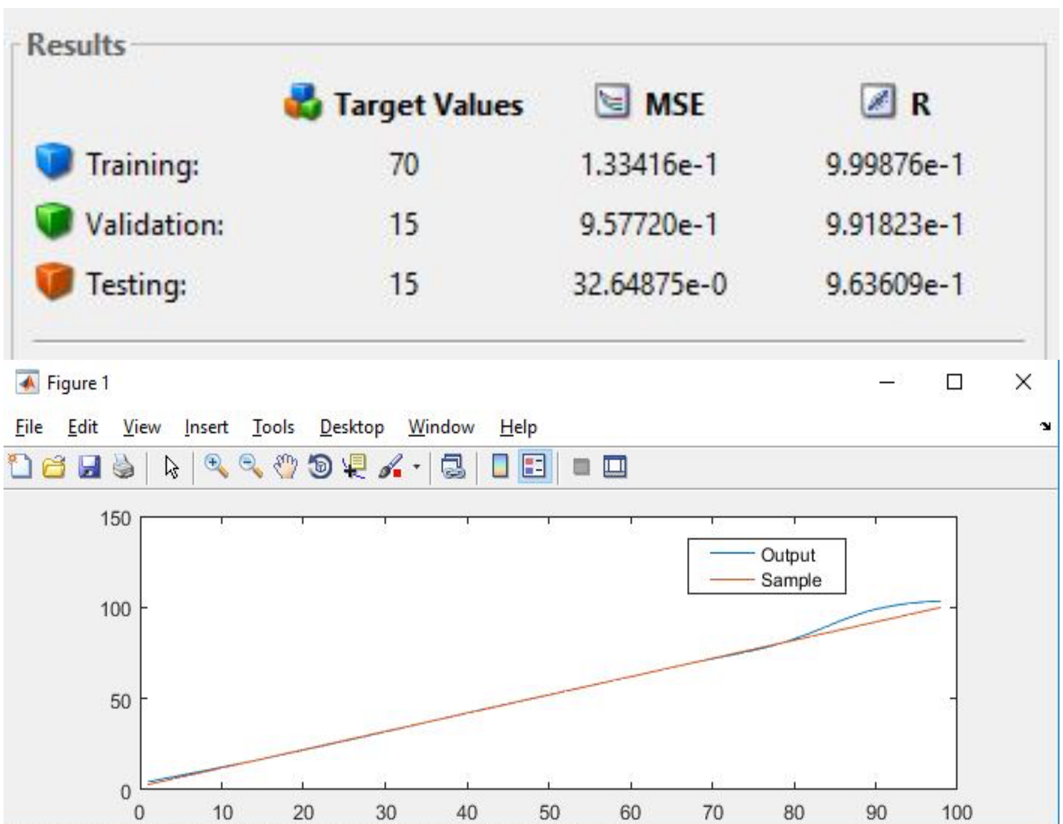


Figure 5. `ntstool` results.

Despite the effort to increase generalizability of a time series, the random partition of the data will produce unreliable results. To overcome this situation, a proposed solution is to self-edit the divider and function, which was already given in the previous section.

Network parameters as hidden layer neuron number and number of delay has been set by default to 10 neuron and 2 delays in `ntstool`. Finding the best parameter design is one of the biggest neural network problems in literature. There are numbers of approaches proposed to solve this problem such as early stopping (Haykin, 1998), noise injection (Holmstrom and Koistinen, 1992; Grandvalet, Canu, and Boucheron, 1997; Skurichina, Raudys, and Duin, 2000; Brown, Gedeon, and Groves, 2003; Seghouane, Moudden, and Fleury, 2004), error regularization (Reed, Marks, and Oh, 1995; Zur, Jiang, Pesce, and Drukker, 2009), weight decay (Poggio and Girosi, 1990; Haykin, 1998), optimized approximation algorithm (Liu, Starzyk, and Zhu, 2008), and trial-error which relies on finding many network and choosing the best performing one. Unfortunately, `ntstool` doesn't offer any strategy for parameter design other than assigning default values of the parameters and user have to know the best parameter design before the initialization or have to try all possible parameter designs manually by starting the `ntstool` all over in number of times. Also, `ntstool` doesn't offer some kind of trial-error based strategy for users so it might take very long time to find the best working parameter design by starting over the toolbox and go through 8 steps with changing the parameters in each time.

Another problematic aspect of `ntstool` is that there is not an option for selection of the best network among candidate networks by using performance measures/criteria via testing performance. This is highly related with the parameter designing situation mentioned below and must be considered altogether under trial-error strategy. The only performance indicator that `ntstool` uses is MSE which is either widely accepted but at the same time not reliable criterion and widely criticized (Hyndman and Koehler, 2006; Armstrong Collopy, 1992; Chatfield, 1988). It could be more useful to show the testing performance with various performance measure to deduce the results because every different measure calculates the network error with different aspects (Bal, 2016).

As noted in Figure 6, `cbnet` function with default settings can be utilized by typing '`cbnet('sample')`' in MATLAB command window and the data file must contain in current folder with `cbnet` in this case data set 'sample' is used same as `ntstool` examples above.

The proposed `cbnet` function uses the strategy of choosing the most proper network which have the best testing performance by using trial-error method and the network selection procedures are being made by four different criteria such as

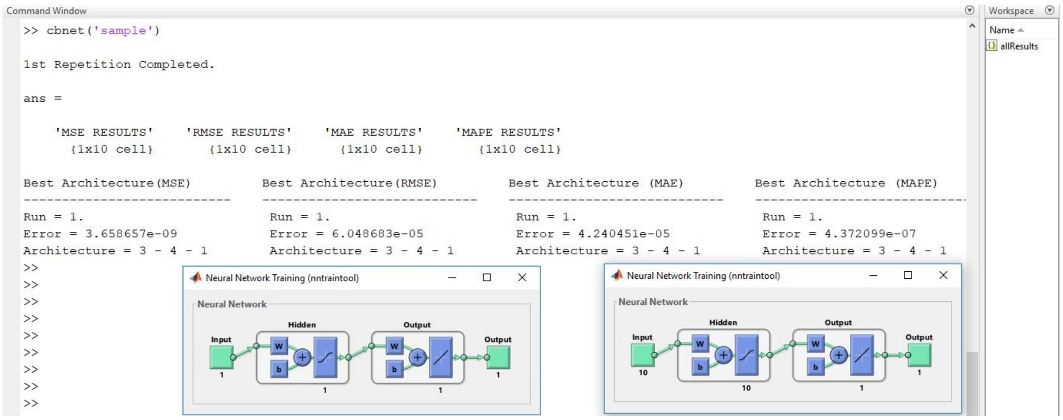


Figure 6. `cbnet` workflow.

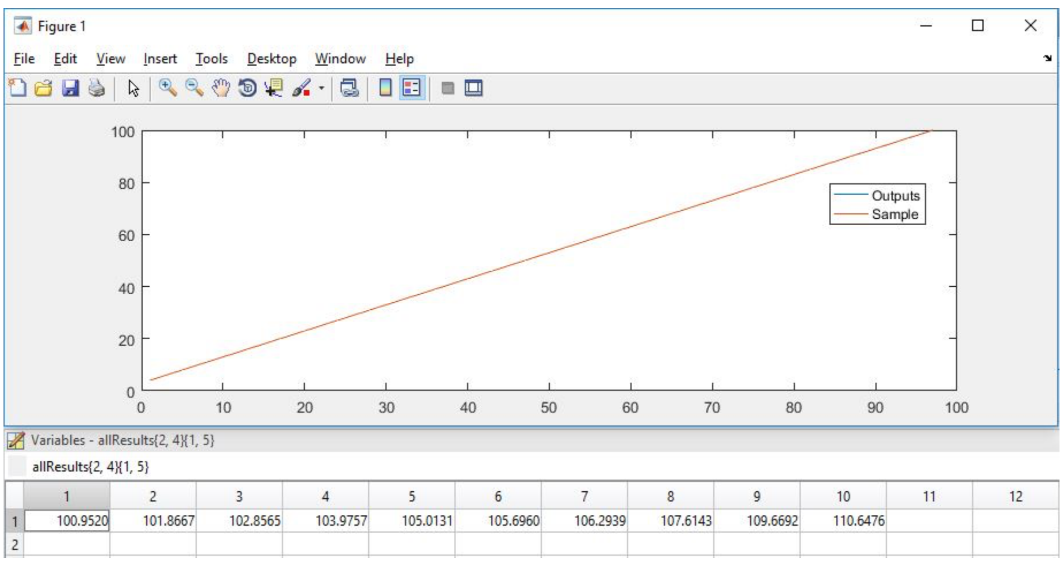


Figure 7. `cbnet` results.

MSE, RMSE, MAE and MAPE. In other words, `cbnet` uses 4 criteria to obtain many networks respectively and chooses the best performed one as a result for each criterion. In the future versions of `cbnet`, new performance criteria will be added for network selection to diversify the results. The 4 best networks for 4 criteria will be obtained. The default parameters of `cbnet` are set 1 to 10 neuron numbers for both input and hidden layer respectively and therefore total 100 candidate network architectures from 1-1-1 to 10-10-1 which can be seen in Figure 3, will be obtained in each repetition. Network with the best testing performance among these 100 networks is selected with MSE, RMSE, MAE and MAPE individually. This process also can be done more than 1 time with repetition parameter to obtain many best-chosen networks. In this example, all 4 performance measures selected the same network as their best choice. Detail workflow of `cbnet` has been given in section 3 above.

Table 2. Comparison chart of ntstool and cbnet

	`ntstool`	`cbnet`
GUI	Yes	No
Time Consumes for Utilization (Default)	8 steps	1 step
Network Obtained	1 network with given parameter design	Best network selection with given range of parameter design among candidate networks
Network Selection Strategy	No	Done by performance measures with testing performance
Data Partition	Randomly	Sequentially
Data Partition Ratio Eligibility	No	Yes
Step-Ahead Forecasting Function	No	Yes
Learning Function Variety	3 Backpropagation Function	8 Backpropagation Function
Activation Function Variety	Only tangent-sigmoid	tangent-sigmoid and log-sigmoid
Global Repetition Parameter for Obtaining More Selected Network in Single Run	No	Yes

In the Figure 7, the output performance of the network which is obtained with the `cbnet` function is shown. Along with the graphic, almost precise expected 10 step-ahead forecasts can also be seen in Figure 4. Forecasting function of `cbnet` allow users to calculate step-ahead forecasts with selected networks which will strengthen the results and test the network’s power further from testing performance. However, `ntstool` doesn’t offer such feature and therefore it is not quite possible to test the obtained network whether has strong forecasting ability or not. The advantages and disadvantages of both `ntstool` and `cbnet` has been truly given and proved the reason why `cbnet` has to be developed. Also, the next versions of `cbnet` will be included in the GUI, permitting easier usage along with various plot options for visual representation of results. Another update is planned is the addition of different neural networks and multi hidden layered architectures for deep time series analysis. All analyses and comparisons are made on MATLAB version 2016a.

Conclusion

A specialized cbnet function for univariate time series analysis with neural networks in MATLAB environment was introduced. It's simple and easy to use structure of the function will allow users to achieve more detailed results with very less effort for this particular type of analysis. The `cbnet` function has more advantageous properties than `ntstool` in order to analyze univariate time series in other words nonlinear autoregressive time series. The codes of `cbnet` is given in appendix. Also, it can be accessed at MATLAB's File Exchange platform (https://www.mathworks.com/matlabcentral/fileexchange/67628-cbnet).

References

Armstrong, B. J. S. and Collopy, F. (1992). Error Measures For Generalizing About Forecasting Methods: Empirical Comparisons. *International Journal of Forecasting, 8*(1), 69–80. https://doi.org/10.1016/0169-2070(92)90008-w

Bal, B. C. (2016). A Comparative Study of Artificial Neural Network Models for Forecasting EURO/USD Exchange Rates by Feed Forward Neural Network. *International Journal of Computing, Communication and Instrumentation Engineering, 3*(2). https://doi.org/10.15242/ijccie.u0616010

Bal, C. and Demir, S. (2017). Forecasting TRY/USD exchange rate with various artificial Neural Network Models. *TEM Journal, 6*(1), 11–16. https://doi.org/10.18421/TEM61-02

Box, G. E. P., Jenkins, G. M., and Reinsel, G. C. (1976). *Time Series Analysis, Forecasting and Control.* Third Edition. Holden-Day.

Brown, W. M., Gedeon, T. D., and Groves, D. I. (2003). Use of noise to augment training data: A neural network method of mineral-potential mapping in regions of limited known deposit examples. *Natural Resources Research, 12*(2), 141–152. https://doi.org/10.1023/a:1024218913435

Chatfield, C. (1988). Apples, oranges and mean square error. *International Journal of Forecasting, 4*(4), 515–518. https://doi.org/10.1016/0169-2070(88)90127-6

Eğrioğlu, E., Aladağ, Ç. H., and Günay, S. (2008). A new model selection strategy in artificial neural networks. *Applied Mathematics and Computation, 195*(2), 591–597. https://doi.org/10.1016/j.amc.2007.05.005

Faraway, J. J. (1992). On the cost of data analysis. *Journal of Computational and Graphical Statistics, 1*(3), 213–229. https://doi.org/10.1080/10618600.1992.10474582

Grandvalet, Y., Canu, S., and Boucheron, S. (1997). Noise Injection: Theoretical Prospects. *Neural Computation, 9*(5), 1093–1108. https://doi.org/10.1162/neco.1997.9.5.1093

Haykin, S. (1998). *Neural Networks: A Comprehensive Foundation*. 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall PTR.

Holmstrom, L. and Koistinen, P. (1992). Using additive noise in back-propagation training. *IEEE Transactions on Neural Networks, 3*(1) 24–38. https://doi.org/10.1109/72.105415

Hyndman, R. J. and Koehler, A. B. (2006). Another look at measures of forecast accuracy. *International Journal of Forecasting, 22*(4), 679–688. https://doi.org/10.1016/j.ijforecast.2006.03.001

Liu, Y., Starzyk, J. A., and Zhu, Z. (2008). Optimized approximation algorithm in neural networks without overfitting. *IEEE Transactions on Neural Networks, 19*(6), 983–995. https://doi.org/10.1109/tnn.2007.915114

Poggio, T. and Girosi, F. (1990). Networks for approximation and learning. *Proceedings of the IEEE, 78*(9), 1481–1497. https://doi.org/10.1109/5.58326

Prechelt, L. (1998). Automatic early stopping using cross validation: Quantifying the criteria. *Neural Networks*, *11*(4), 761–767. https://doi.org/10.1016/s0893-6080(98)00010-0

Reed, R., Marks, R. J., and Oh, S. (1995). Similarities of Error Regularization, Sigmoid Gain Scaling, Target Smoothing, and Training with Jitter. *IEEE Transactions on Neural Networks, 6*(3), 529–538. https://doi.org/10.1109/72.377960

Skurichina, M., Raudys, Š., and Duin, R. P. W. (2000). K-nearest neighbors directed noise injection in multilayer perceptron training. *IEEE Transactions on Neural Networks, 11*(2), 504–511. https://doi.org/10.1109/72.839019

Seghouane, A. K., Moudden, Y., and Fleury, G. (2004). Regularizing the effect of input noise injection in feedforward neural networks training. *Neural Computing and Applications, 13*(3), 248–254. https://doi.org/10.1007/s00521-004-0411-6

Zhang, G., Patuwo, B. E., and Hu, M. Y. (1998). Forecasting with artificial neural networks, *International Journal of Forecasting, 14*(1), 35-62. https://doi.org/10.1016/s0169-2070(97)00044-7

Zur, R. M., Jiang, Y., Pesce, L. L., and Drukker, K. (2009). Noise injection for training artificial neural networks: A comparison with weight decay and early stopping. *Medical Physics, 36*(10), 4810–4818. https://doi.org/10.1118/1.3213517

Appendix A: MATLAB function (cbnet.m)

```
function cbnet(filename,imn,maxhid,tf,ep,l1,trratio,valratio,teratio,fcast,
glorep, varargin)
%%
% CBNETfunction for univariate time series analysis with feed forward neural
network.
%
% filename; name of the variable for time series vector in the same folder.
% imn; input matrix number.
% maxhid; maximum neuron number of hidden layer.
% tf; training function of network.
% ep; maximum epoch number.
% l1; activation of hidden layer.
% trratio; training set ratio.
% valratio; validation set ratio.
% teratio; test set ratio.
% fcast; desired number of step ahead forecasts.
% glorep; repetition number of CBNETfunction.
%%          We recommend self-editing the 'dividerand.m'
% since 'divideblock.m' doesn`t allow not to choose validation set partition.
%
% Openning the dividerand.m file from its original location.
%    open dividerand
% Modify the 105th row `allInd=randperm(Q);` of the dividerand.m file with this
code.
%    allInd = 1:Q;
%%  Simple example of CBNETfunction;
%
%   sample=[1:100]; save('sample.mat','sample');
%   cbnet('sample')
%
%% Copyright (c) 2018, Cagatay BAL
%
%%
if nargin == 0 || isempty(filename), error('Error: data has not been chosen!'),
end
if nargin > 11 , error('Too many inputs!'), end
```

```
switch nargin
    case 1
        imn=10; maxhid=10; tf='trainlm'; ep=1000; l1='tansig';
        trratio=0.85; valratio=0; teratio=0.15; fcast=10; glorep=1;
    case 2
        maxhid=10; tf='trainlm'; ep=1000; l1='tansig'; trratio=0.85;
        valratio=0; teratio=0.15; fcast=10; glorep=1;
    case 3
        tf='trainlm'; ep=1000; l1='tansig'; trratio=0.85; valratio=0;
        teratio=0.15; fcast=10; glorep=1;
    case 4
        ep=1000; l1='tansig'; trratio=0.85; valratio=0; teratio=0.15;
        fcast=10; glorep=1;
    case 5
        l1='tansig'; trratio=0.85; valratio=0; teratio=0.15; fcast=10;
        glorep=1;
    case 6
        trratio=0.85; valratio=0; teratio=0.15; fcast=10; glorep=1;
    case 7
        valratio=0; teratio=0.15; fcast=10; glorep=1;
    case 8
        teratio=0.15; fcast=10; glorep=1;
    case 9
        fcast=10; glorep=1;
    case 10
        glorep=1;
end
function lag(filename,imn)

% Function for creating input matrixes and target vectors.

% filename; name of the variable for time series vector in the same folder

% imn; input matrix number

datavector=cell2mat(struct2cell(load(filename)));
n=length(datavector);

for p=1:imn

for i=1:p

    for j=1:i
    inputvector=datavector(j:n-(p-(j-1)));
    input{1,j}=inputvector;

    end
    input{2,p}(i,:)=input{1,j};

end

    data{p,1}=input{2,p}; %First column of array consist of input matrixes.
```

```
    data{p,2}=datavector(p+1:n); %Second column of array consist of target
vectors.

end

save('lags.mat','data'); %Saving the array to the related folder.
figure
plot(datavector)
end
function forecast(fcast)

% Function for calculation of given step ahead forecasts.

% fcast, desired number of step ahead forecasts.

%%
    for f=1:fcast
        fc=(netbest_MSE(MSE_testvector'));
        MSE_forecast(1,f)=fc;
        MSE_fcastvector=[MSE_testvector fc];
        MSE_fcastvector=MSE_fcastvector(2:end);
        MSE_testvector=MSE_fcastvector;
    end
    save([ 'MSE_forecast-' num2str(glr) '.mat'],'MSE_forecast')
%%
    for f=1:fcast
        fc=(netbest_RMSE(RMSE_testvector'));
        RMSE_forecast(1,f)=fc;
        RMSE_fcastvector=[RMSE_testvector fc];
        RMSE_fcastvector=RMSE_fcastvector(2:end);
        RMSE_testvector=RMSE_fcastvector;
    end
    save([ 'RMSE_forecast-' num2str(glr) '.mat'],'RMSE_forecast')
%%
    for f=1:fcast
        fc=(netbest_MAE(MAE_testvector'));
        MAE_forecast(1,f)=fc;
        MAE_fcastvector=[MAE_testvector fc];
        MAE_fcastvector=MAE_fcastvector(2:end);
        MAE_testvector=MAE_fcastvector;
    end
    save([ 'MAE_forecast-' num2str(glr) '.mat'],'MAE_forecast')
%%
    for f=1:fcast
        fc=(netbest_MAPE(MAPE_testvector'));
        MAPE_forecast(1,f)=fc;
        MAPE_fcastvector=[MAPE_testvector fc];
        MAPE_fcastvector=MAPE_fcastvector(2:end);
        MAPE_testvector=MAPE_fcastvector;
    end
    save([ 'MAPE_forecast-' num2str(glr) '.mat'],'MAPE_forecast')
end
function [allResults]=allResults(glorep)
```

```
% Function for gathering and saving the results.

MSE_minerr=1e100; RMSE_minerr=1e100; MAE_minerr=1e100; MAPE_minerr=1e100;

for ar=1:glorep
    load(['MSE-' num2str(ar) '.mat'])
    load(['MSE_forecast-' num2str(ar) '.mat'],'MSE_forecast')
    load(['RMSE-' num2str(ar) '.mat'])
    load(['RMSE_forecast-' num2str(ar) '.mat'],'RMSE_forecast')
    load(['MAE-' num2str(ar) '.mat'])
    load(['MAE_forecast-' num2str(ar) '.mat'],'MAE_forecast')
    load(['MAPE-' num2str(ar) '.mat'])
    load(['MAPE_forecast-' num2str(ar) '.mat'],'MAPE_forecast')

    allResults{1,1}='MSE RESULTS';
    allResults{2,1}{ar,1}=MSE_input;
    allResults{2,1}{ar,2}=MSE_hidden;
    allResults{2,1}{ar,3}=1;
    allResults{2,1}{ar,4}=MSE_performanceError;
    allResults{2,1}{ar,5}=MSE_forecast;
    allResults{2,1}{ar,6}=MSE_inputs;
    allResults{2,1}{ar,7}=MSE_targets;
    allResults{2,1}{ar,8}=MSE_testvector;
    allResults{2,1}{ar,9}=netbest_MSE;
    allResults{2,1}{ar,10}=ar;

    if MSE_performanceError <= MSE_minerr
        MSE_minerr=MSE_performanceError;
        MSE_minerr_order=ar;
        MSE_minerr_input=MSE_input;
        MSE_minerr_hidden=MSE_hidden;
    end

    allResults{1,2}='RMSE RESULTS';
    allResults{2,2}{ar,1}=RMSE_input;
    allResults{2,2}{ar,2}=RMSE_hidden;
    allResults{2,2}{ar,3}=1;
    allResults{2,2}{ar,4}=RMSE_performanceError;
    allResults{2,2}{ar,5}=RMSE_forecast;
    allResults{2,2}{ar,6}=RMSE_inputs;
    allResults{2,2}{ar,7}=RMSE_targets;
    allResults{2,2}{ar,8}=RMSE_testvector;
    allResults{2,2}{ar,9}=netbest_RMSE;
    allResults{2,2}{ar,10}=ar;

    if RMSE_performanceError <= RMSE_minerr
        RMSE_minerr=RMSE_performanceError;
        RMSE_minerr_order=ar;
        RMSE_minerr_input=RMSE_input;
        RMSE_minerr_hidden=RMSE_hidden;
    end

    allResults{1,3}='MAE RESULTS';
    allResults{2,3}{ar,1}=MAE_input;
```

```
    allResults{2,3}{ar,2}=MAE_hidden;
    allResults{2,3}{ar,3}=1;
    allResults{2,3}{ar,4}=MAE_performanceError;
    allResults{2,3}{ar,5}=MAE_forecast;
    allResults{2,3}{ar,6}=MAE_inputs;
    allResults{2,3}{ar,7}=MAE_targets;
    allResults{2,3}{ar,8}=MAE_testvector;
    allResults{2,3}{ar,9}=netbest_MAE;
    allResults{2,3}{ar,10}=ar;

    if MAE_performanceError <= MAE_minerr
        MAE_minerr=MAE_performanceError;
        MAE_minerr_order=ar;
        MAE_minerr_input=MAE_input;
        MAE_minerr_hidden=MAE_hidden;
    end

    allResults{1,4}='MAPE RESULTS';
    allResults{2,4}{ar,1}=MAPE_input;
    allResults{2,4}{ar,2}=MAPE_hidden;
    allResults{2,4}{ar,3}=1;
    allResults{2,4}{ar,4}=MAPE_performanceError;
    allResults{2,4}{ar,5}=MAPE_forecast;
    allResults{2,4}{ar,6}=MAPE_inputs;
    allResults{2,4}{ar,7}=MAPE_targets;
    allResults{2,4}{ar,8}=MAPE_testvector;
    allResults{2,4}{ar,9}=netbest_MAPE;
    allResults{2,4}{ar,10}=ar;

    if MAPE_performanceError <= MAPE_minerr
        MAPE_minerr=MAPE_performanceError;
        MAPE_minerr_order=ar;
        MAPE_minerr_input=MAPE_input;
        MAPE_minerr_hidden=MAPE_hidden;
    end

end
    save('allResults.mat','allResults')
    assignin('base','allResults',allResults)
end
%% Creating input matrix and target vector with 'lag' function.
lag(filename,imn)
%%
for glr = 1 : glorep
%% Assigning very big error values for performance measures.
global_error_MSE=1e100; global_error_RMSE=1e100;
global_error_MAE=1e100; global_error_MAPE=1e100;

%%
    for j=1:imn
    inputs=data{j,1};
    targets=data{j,2};
%%
    for i=1:maxhid
```

```
net = feedforwardnet(i);

net.trainFcn = tf;
net.trainParam.epochs = ep;
net.layers{1}.transferFcn = l1;
net.trainParam.showWindow = 1;

net.divideParam.trainRatio=trratio;
net.divideParam.valRatio=valratio;
net.divideParam.testRatio=teratio;

[net,tr] = train(net,inputs,targets);
outputs = net(inputs);
errors = gsubtract(targets,outputs);
testset=targets.* tr.testMask{1};
testset(isnan(testset))=[];
testerror = errors  .* tr.testMask{1};
testerror(isnan(testerror))=[];
%% Performance measures for calculating test set error.
MSE_testerror=mean(testerror.^2);
RMSE_testerror=sqrt(mean(testerror.^2));
MAE_testerror=mean(abs(testerror));
MAPE_testerror=mean(abs(testerror)./abs(testset));

%% Model selection algorithm for each performance measure.

if  MSE_testerror<global_error_MSE
    MSE_performanceError=MSE_testerror;
    global_error_MSE=MSE_testerror;
    MSE_input=j;
    MSE_hidden=i;
    MSE_inputs=inputs;
    MSE_targets=targets; MSE_tl=length(MSE_targets);
    MSE_outputs=outputs;
    MSE_testvector=MSE_targets(MSE_tl-MSE_input+1:MSE_tl);
    netbest_MSE=net;
    save(['MSE-' num2str(glr) '.mat'],'MSE_performanceError',...
        'global_error_MSE','MSE_input','MSE_hidden','MSE_inputs',...
        'MSE_targets','MSE_outputs','MSE_testvector','netbest_MSE')
end
if  RMSE_testerror<global_error_RMSE
    RMSE_performanceError=RMSE_testerror;
    global_error_RMSE=RMSE_testerror;
    RMSE_input=j;
    RMSE_hidden=i;
    RMSE_inputs=inputs;
    RMSE_targets=targets; RMSE_tl=length(RMSE_targets);
    RMSE_outputs=outputs;
    RMSE_testvector=RMSE_targets(RMSE_tl-RMSE_input+1:RMSE_tl);
    netbest_RMSE=net;
    save(['RMSE-' num2str(glr) '.mat'],'RMSE_performanceError',...
        'global_error_RMSE','RMSE_input','RMSE_hidden',...
        'RMSE_inputs','RMSE_targets','RMSE_outputs',...
        'RMSE_testvector','netbest_RMSE')
```

```
    end
    if  MAE_testerror<global_error_MAE
        MAE_performanceError=MAE_testerror;
        global_error_MAE=MAE_testerror;
        MAE_input=j;
        MAE_hidden=i;
        MAE_inputs=inputs;
        MAE_targets=targets; MAE_tl=length(MAE_targets);
        MAE_outputs=outputs;
        MAE_testvector=MAE_targets(MAE_tl-MAE_input+1:MAE_tl);
        netbest_MAE=net;
        save(['MAE-' num2str(glr) '.mat'],'MAE_performanceError',...
            'global_error_MAE','MAE_input','MAE_hidden','MAE_inputs',...
            'MAE_targets','MAE_outputs','MAE_testvector','netbest_MAE')
    end
    if  MAPE_testerror<global_error_MAPE
        MAPE_performanceError=MAPE_testerror;
        global_error_MAPE=MAPE_testerror;
        MAPE_input=j;
        MAPE_hidden=i;
        MAPE_inputs=inputs;
        MAPE_targets=targets; MAPE_tl=length(MAPE_targets);
        MAPE_outputs=outputs;
        MAPE_testvector=MAPE_targets(MAPE_tl-MAPE_input+1:MAPE_tl);
        netbest_MAPE=net;
        save(['MAPE-' num2str(glr) '.mat'],'MAPE_performanceError',...
            'global_error_MAPE','MAPE_input','MAPE_hidden',...
            'MAPE_inputs','MAPE_targets','MAPE_outputs',...
            'MAPE_testvector','netbest_MAPE')
    end
   end
end

forecast(fcast)     % Calculating the forecasts with 'forecast' function.

    if glr==1
       fprintf('\r%dst Repetition Completed.\r',glr)
    elseif glr==2
        fprintf('\r%dnd Repetition Completed.\r',glr)
    elseif glr==3
        fprintf('\r%drd Repetition Completed.\r',glr)
    else
        fprintf('\r%dth Repetition Completed.\r',glr)
    end
end

allResults(glorep)  % Obtaining results with 'allResults' function.

%%
disp('Best Architecture(MSE)         Best Architecture(RMSE)         Best
Architecture (MAE)         Best Architecture (MAPE)')
disp('---------------------------    ----------------------------    -----------
-----------------    -----------------------------')
fprintf('Run = %d.\t\t\tRun = %d.\t\t\tRun = %d.\t\t\tRun = %d.  \r',...
```

```
    MSE_minerr_order,RMSE_minerr_order,MAE_minerr_order,MAPE_minerr_order)
fprintf('Error = %e\t\tError = %e\t\tError = %e\t\tError = %e  \r',...
    MSE_minerr,RMSE_minerr,MAE_minerr,MAPE_minerr)
fprintf('Architecture = %d - %d - 1\tArchitecture = %d - %d - 1\tArchitecture =
%d - %d - 1\tArchitecture = %d - %d - 1  \r',...
    MSE_minerr_input,MSE_minerr_hidden,RMSE_minerr_input,...
    RMSE_minerr_hidden,MAE_minerr_input,MAE_minerr_hidden,...
    MAPE_minerr_input,MAPE_minerr_hidden)
end
```