Development of A New Recurrent Neural Network Toolbox (RNN-Tool) *

A Course Project Report on Training Recurrent Multilayer Perceptron and Echo State Network

Le Yang, Yanbo Xue
Email address: yangl7@psychology.mcmaster.ca; yxue@soma.mcmaster.ca

Abstract

In this report, we developed a new recurrent neural network toolbox, including the recurrent multilayer perceptron structure and its companying extended Kalman filter based training algorithms: BPTT-GEKF and BPTT-DEKF. Besides, we also constructed programs for designing echo state network with single reservoir, together with the offline linear regression based training algorithm. We name this toolbox as the RNN-Tool. Within the toolbox, we implement the RMLP and ESN as MATLAB structures, which are used throughout the processes of network generation, training and testing. Finally we study a predictive modeling case of a phase-modulated sinusoidal function to test this toolbox. Simulation results show that ESN can outperform the BPTT-GEKF and BTPP-DEKF methods both on computational load and prediction accuracy.

Keywords: Recurrent neural network, recurrent multilayer perceptron, backpropagation through time, extended Kalman filter, echo state network, RNN-Tool

*Both authors contribute equally to this report.

July 10, 2006

Contents

Figures

1 Introduction

Over the past decades, the researches and applications of neural network can be exemplified by a potato model, as shown in Jaeger [1], that most efforts has been put on the feedforward structures, while the recurrent neural network (RNN) was allocated only a small portion. However, motivated by the increasing interests in practical processes with memory, RNNs started to sprout, as more RNN models were developed, such as the recurrent multilayer perceptron (RMLP) [2] and echo state network/liquid state network [3, 4, 5], as well as efficient training algorithms, such as the backpropatation through time (BPTT) and extended Kalman filter (EKF) based methods.

In this report, we developed a recurrent neural network toolbox, including the RMLP structure and its companying training algorithms: BPTT-GEKF and BPTT-DEKF. Besides, we also constructed programs for ESN with single reservoir, together with the offline linear regression based training algorithm. We name this toolbox as the RNN-Tool. The rest of the report is organized as follows. In section 2, we briefly revisit the mathematical formulations of the RMLP structure along with its training and weight update algorithms: BPTT-GEKF/DEKF, and the ESN structure. Section 3 describes the structure and built-in functions of the RNN-Tool toolbox. As an illustrative example of applying the toolbox in practical tasks, we present the results of solving a predictive modeling problem with RNN-Tool. Finally, we conclude this report in section 5. The source codes of the newly developed toolbox are attached in Appendix I, II, and III.

2 Fundamentals of Recurrent Neural Network

2.1 Recurrent Multilayer Perceptron

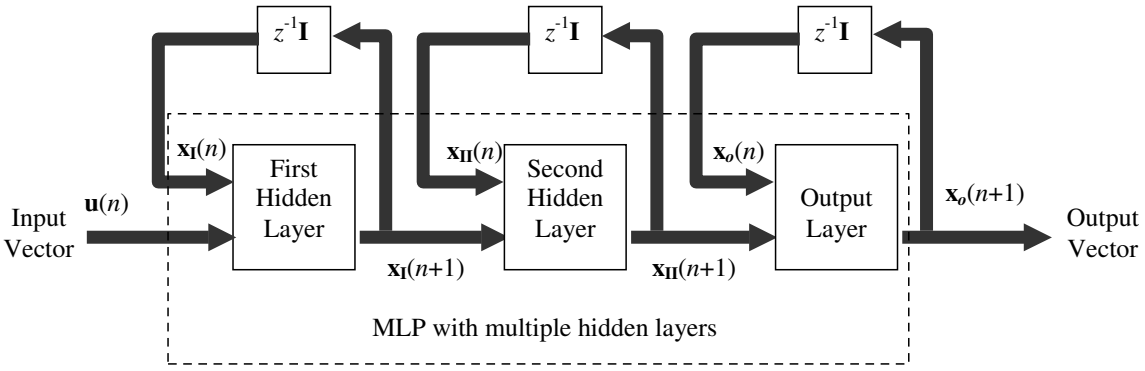

Fig. 1: Exemplary structure of RMLP

The recurrent multilayer perceptron (RMLP), due to Puskorius [2], is first proposed for controlling nonlinear dynamical systems, where RMLP serves directly as a controller. A schematic 3-layer diagram for RMLP is shown in Fig. 1, where RMLP can be considered as a feedforward network augmented by recurrent synapses. Generally, a RMLP has one or more recurrent layers, i.e., there exist layers that receive recurrent inputs from themselves and/or other layers. In this exemplary RMLP, recurrent inputs to a certain layer are solely composed of the action potentials of that layer at the previous time step. Let $\mathbf{x}_I(n)$ and $\mathbf{x}_{II}(n)$ denote the output of the first and second hidden layer, respectively, and $\mathbf{x}_o(n)$ be the output of the output layer and

$\mathbf{u}(n)$ denotes the input vector. Then, the operational principles of the RMLP given in Fig. 1 can be mathematically expressed by the following coupled equations:

$$\begin{aligned} \mathbf{x}_I(n+1) &= \boldsymbol{\varphi}_I(\mathbf{w}_I \cdot \begin{bmatrix} \mathbf{x}_I(n) \\ u(n) \end{bmatrix}) \\ \mathbf{x}_{II}(n+1) &= \boldsymbol{\varphi}_{II}(\mathbf{w}_{II} \cdot \begin{bmatrix} \mathbf{x}_{II}(n) \\ \mathbf{x}_I(n) \end{bmatrix}) \\ \mathbf{x}_o(n+1) &= \boldsymbol{\varphi}_o(\mathbf{w}_o \cdot \begin{bmatrix} \mathbf{x}_o(n) \\ \mathbf{x}_{II}(n) \end{bmatrix}) \end{aligned} \tag{1}$$

where $\boldsymbol{\varphi}_I(\cdot,\cdot)$, $\boldsymbol{\varphi}_{II}(\cdot,\cdot)$ and $\boldsymbol{\varphi}_o(\cdot,\cdot)$ are the activation functions of the first hidden layer, second hidden layer, and output layer, respectively; $\mathbf{w}_I$, $\mathbf{w}_{II}$ and $\mathbf{w}_o$ denote the weight matrices of the first hidden layer, second hidden layer, and output layer, respectively.

2.2 Extended Kalman Filter

To briefly present the extended Kalman filter based training approach for RMLP, we again use the 3-layer exemplary network given in Fig. 1. We first re-write the operational principles of the RMLP given in Eqn. (1) in a state-space fashion:

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \boldsymbol{\omega}(n) \tag{2}$$

$$\mathbf{x}_o(n) = \mathbf{c}(\mathbf{w}(n), \mathbf{u}(n), \mathbf{v}(n)) + \boldsymbol{\nu}(n) \tag{3}$$

where $\mathbf{w}(n)$ denote the synaptic weights of the entire network, i.e., $\mathbf{w}(n)$ is the aggregation of weight vectors $\mathbf{w}_I$, $\mathbf{w}_{II}$ and $\mathbf{w}_o$, and $\mathbf{x}_o(n)$ is the network output. $\mathbf{v}(n)$ is the vector containing all recurrent inputs and $\boldsymbol{\nu}(n)$ denote the measurement noise vector. $\mathbf{u}(n)$ is the network input vector, the same as the one defined in (1). $\boldsymbol{\omega}(n)$ denotes the artificial process noise, the variance of which is zero after the training process ends, or a small value during the training process to circumvent the divergence problem [6]. $\mathbf{c}(\cdot)$ is the network transfer function, which is generally highly nonlinear and very hard, if not possible, to be expressed in a closed form. Due to the nonlinearity in the transfer function, the classical Kalman filter cannot be directly applied to calculate the weight vector update.

Through linearizing the measurement equation, the extended Kalman filter (EKF) approach provides us with a tool to tackle this problem. To implement this idea, we expand Eqn. (3) using Taylor series, and if only the linear factor is considered, we obtain:

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \boldsymbol{\omega}(n) \tag{4}$$

$$\hat{d}(n) = \mathbf{C}(n)\mathbf{w}(n) + \boldsymbol{\nu}(n), \tag{5}$$

where $\hat{d}(n)$ is the first-order approximation of $\mathbf{x}_o(n)$, and $\mathbf{C}(n)$ is the p-by-W measurement matrix of the linearized model, given by

$$\mathbf{C}(n) = \begin{bmatrix} \frac{\partial c_1}{\partial w_1} & \frac{\partial c_1}{\partial w_2} & \cdots & \frac{\partial c_1}{\partial w_W} \\ \frac{\partial c_2}{\partial w_1} & \frac{\partial c_2}{\partial w_2} & \cdots & \frac{\partial c_2}{\partial w_W} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial c_p}{\partial w_1} & \frac{\partial c_p}{\partial w_2} & \cdots & \frac{\partial c_p}{\partial w_W} \end{bmatrix}, \tag{6}$$

with p and W denoting the number of output neurons and synaptic weights, respectively, and $c_i(\cdot)$ is the activation function of the ith output neuron, $i = 1, 2, ..., p$. Applying the classic Kalman filter theory to the linearized state space model, we could the derive the weight updating equations:

$$\begin{aligned}
\mathbf{\Gamma}(n) &= \left[\sum_{i=1}^{g} \mathbf{C}_i(n)\mathbf{K}_i(n, n-1)\mathbf{C}_i^T(n) + \mathbf{R}(n)\right]^{-1} && (7)\\
\mathbf{G}_i(n) &= \mathbf{K}_i(n, n-1)\mathbf{C}_i^T(n)\mathbf{\Gamma}(n) && (8)\\
\boldsymbol{\alpha}(n) &= \mathbf{d}(n) - \hat{\mathbf{d}}(n) && (9)\\
\mathbf{w}_i(n+1) &= \mathbf{w}_i(n) + \mathbf{G}_i(n)\boldsymbol{\alpha}(n) && (10)\\
\mathbf{K}_i(n+1, n) &= \mathbf{K}_i(n, n-1) - \mathbf{G}_i(n)\mathbf{C}_i(n)\mathbf{K}_i(n, n-1) + \mathbf{Q}_i(n) && (11)
\end{aligned}$$

where g is the number of blocks, each of which contains the weights of all synapses targeted on a neuron. $\mathbf{\Gamma}(n) \in \mathbb{R}^{p\times p}$ and $\boldsymbol{\alpha}(n) \in \mathbb{R}^{p\times 1}$ denote respectively the global conversion factor of the RMLP network and the innovations. $\mathbf{G}_i(n) \in \mathbb{R}^{W_i\times p}$, $\mathbf{w}_i(n) \in \mathbb{R}^{W\times 1}$, and $\mathbf{K}_i(n, n-1) \in \mathbb{R}^{k_i\times k_i}$ denote the Kalman gain, the weight vector $\mathbf{w}_i(n)$ at time n, and error covariance matrix, for the ith group of neurons, respectively. $\mathbf{d}(n)$ is the desired output provided by the teacher forcing signal.

This algorithm is termed the decoupled extended Kalman filter (DEKF). Note that if we set $g = 1$, i.e., the weights for all neurons are grouped into a single block, the DEKF algorithm will change into the global extended Kalman filter (GEKF) algorithm. The $\mathbf{R}(n)$ is the diagonal covariance matrix for the measurement noise vector $\boldsymbol{\nu}(n)$ and $\mathbf{Q}(n)$ is the diagonal covariance matrix for the artificial process noise vector $\boldsymbol{\omega}(n)$. From the implementation perspective, the non-zero entries in $\mathbf{R}(n)$ and $\mathbf{Q}(n)$ should be annealed as the training goes on, so that at the beginning of the training process, we are able to search a larger space for an optimal solution, instead of being trapped in a local minima quickly, and toward the end of the training, the weight vector is stabilized. Besides, it is not hard to notice that GEFK has much larger computational complexity, compared with DEKF, since the calculation involving a high-dimension error covariance matrix $\mathbf{K}(n+1, n)$ is more expensive than handling g lower-dimension ones. To avoid the over-training problem, we introduce a learning rate $\eta(n)$ into the weight updating Eqn. (10) to obtain

$$\mathbf{w}_i(n+1) = \mathbf{w}_i(n) + \eta(n) \cdot \mathbf{G}_i(n)\boldsymbol{\alpha}(n). \tag{12}$$

As the training process goes on, we gradually reduce $\eta(n)$ to a small value to fix the weights, so that overtraining effect can be partially mitigated.

2.2.1 Backpropagation Through Time

As can see from (5) that we need to calculate the gradient $\mathbf{C}(n)$ in order to linearize the nonlinear transfer function of the RMLP. Usually, it is very hard to derive a closed-form expression of $\mathbf{C}(n)$, largely due to the difficulty in obtaining $c(\cdot)$ in a tractable form. To address this problem, an efficient gradient calculation algorithm, termed as the backpropagation through time (BPTT), is proposed by Werbos (first appeared in his Ph.D. thesis [7] and then described in [8]). BPTT is indeed an extension of the standard backpropagation method: it unfolds the recurrent network temporally into a multilayer feedforward network so that the derivative of the output error of the recurrent network with respect to previous inputs and network states, i.e., the action potentials stored in the recurrent tap delay lines, can be computed using the backpropagation algorithm. In principle, the unfolding operation of BPTT will generate a network with infinite

number of layers. However, for real-time implementations, the truncated version of the original BPTT [9] must be used. The idea of truncation is motivated by the so-called vanishing gradient effect and the finite memory assumption. Specifically, although the recurrent network may have infinite memory, i.e., its current output is in fact dependent on all its previous inputs, we always assume that due to the effects of forgetness, the correlation between the current output and the inputs far beyond are very small. Moreover, for the backpropagation algorithm, the error can not propagate efficiently through infinite layers, and usually, it completely dies out after several layers, which further reduces the necessities of using an infinite-layered network. The number of unfolding operations conducted on the recurrent connections is termed as the truncation depth h, which normally takes the value of three to five.

2.3 Echo State Network

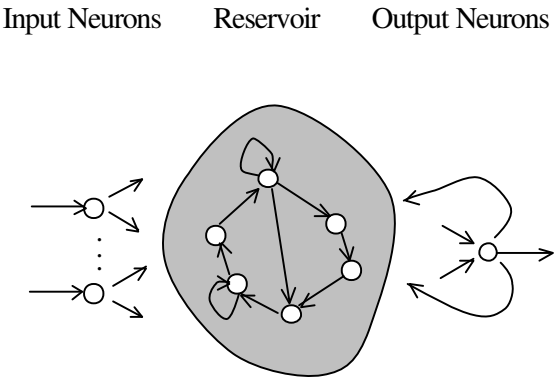

Fig. 2: The basic structure of an echo state network.

Echo state network (ESN) [3, 4] and its spiking counterpart, liquid state network (LSN)[5] are new recurrent neural network (RNN) structures. Both were motivated by recent neurophysiological experiments (see [4] and the references therein). The kernel part of ESN is a single reservoir with a large number of neurons that are randomly inter-connected and/or self-connected. The reservoir itself is fixed, once it is chosen. Moreover, during the training process of ESN, only the output connections are changed through offline linear regression or online methods, such as the recursive least square (RLS) [3, 4, 10]. The ESN has been successfully applied in chaotic and nonlinear dynamic systems modeling, identification and control [4, 11, 12].

The general structure of an ESN with an N-neuron reservoir is shown in Fig. 2. The update of the reservoir state is expressed as

$$\mathbf{x}(n) = \boldsymbol{\varphi}(\mathbf{w}^{in}\mathbf{u}(n) + \mathbf{w}^{DR}\mathbf{x}(n-1) + \mathbf{w}^{back}d(n-1)), \tag{13}$$

where $\boldsymbol{\varphi}(\cdot) = (\varphi_1, ..., \varphi_N)^T$ are sigmoidal activation functions, T denotes the matrix transpose, $\mathbf{x}(n)$ is the internal state of the reservoir at time step n, $d(n-1)$ is the action potential of the output neuron at the previous time step, and $\mathbf{u}(n)$ is the current input vector. $\mathbf{w}^{in}$, $\mathbf{w}^{DR}$ and $\mathbf{w}^{back}$ are weight matrices for input connections, the reservoir, and feedback connections, respectively. The output of the ESN is typically given by

$$d(n) = \boldsymbol{\varphi}^{out}\left(\mathbf{w}^{out} \cdot \begin{bmatrix} \mathbf{x}(n) \\ d(n-1) \end{bmatrix}\right), \tag{14}$$

where φ^{out} can be either linear or sigmoidal, depending on the complexity of the task, and $\mathcal{W}^{out}$ denotes the weight matrix of output connections, which is determined through either online or offline training. With these configurations, the weight matrix dimensions for an M-neuron input, N-neuron reservoir and single-neuron output ESN are respectively $\mathbf{w}^{in} \in \mathbb{R}^{N\times M}$, $\mathbf{w}^{DR} \in \mathbb{R}^{N\times N}$, $\mathbf{w}^{back} \in \mathbb{R}^{N\times 1}$, and $\mathbf{w}^{out} \in \mathbb{R}^{1\times N}$.

3 Toolbox Description

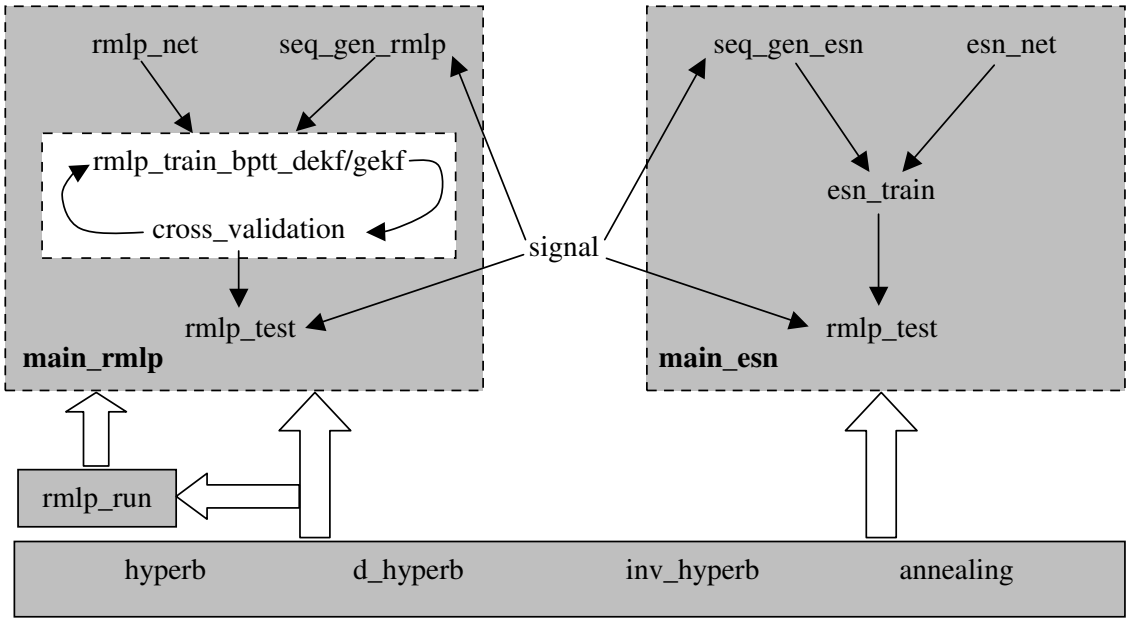

Fig. 3: RNN-Tool structure and flow chart of functions

The Recurrent Neural Network Toolbox (RNN-Tool) is a collection of MATLAB M-files that implement two recurrent neural network structures and corresponding training algorithms. The Recurrent Multilayer Perceptron (RMLP) is trained by BPTT-GEKF and BPTT-DEKF. And echo state network (ESN) is trained using an offline linear regression. The toolbox can be freely downloaded from `http://soma.mcmaster.ca/~yxue/publications.html`.

Fig. 3 shows the toolbox structure and flow chart of the functions of RNN-Tool. We can see that the RNN-Tool structure includes three parts: the RMLP training and testing routines, the ESN training and testing routines, and the necessary sub-routines that provide basic functions for RMLP and ESN.

The RMLP part includes the following functions, where the source codes of each function are given in Appendix I,

1. main_rmlp.m is the main function for RMLP network training and testing;
2. rmlp_train_bptt_dekf.m realizes BPTT-DEKF training algorithm
3. rmlp_train_bptt_gekf.m implements BPTT-GEKF training algorithm;
4. rmlp_net.m generates a RMLP network for training;
5. rmlp_run.m runs the RMLP network;

6. rmlp_test.m tests the trained RMLP network using data points not exposed to the network;
7. seq_gen_rmlp.m generates the training data sets for RMLP;
8. cross_validation.m cross-validates the training RMLP network;
9. signal.m is a function designed to generate the target signals for both RMLP and ESN;

The ESN part includes the following functions, where the source codes of each function are given in Appendix II,

1. main_esn.m is the main function of ESN training and testing;
2. esn_net.m generates an ESN for training;
3. esn_train.m trains the ESN;
4. esn_test.m tests the ESN;
5. seq_gen_esn.m generates the training data sets for ESN;

The private part includes the following functions, where the source codes of each function are given in Appendix III,

1. annealing.m implements the annealing function for $\mathbf{Q}(n)$ and $\mathbf{R}(n)$, currently only linear annealing is available;
2. hyperb.m is the hyperbolic function;
3. d_hyperb.m calculates the first-order derivative of hyperbolic function;
4. inv_hyperb.m computes the inverse hyperbolic function.

4 A Case Study: Predictive Modeling with RNN-Tool

In this section, we present the results for testing the toolbox using the following phase-modulated sinusoidal function

$$x(n) = \sin(n + \sin(n^2)) \quad n = 0, 1, 2, ... \tag{15}$$

The task is of a predictive nature, i.e., the recurrent multilayer perceptron or the echo state network is trained to predict the sample value of the time series given in (15) at the time step $n + 1$ given the signal history up to n. Mathematically, the prediction of the network $\hat{x}(n + 1)$ is given by

$$\hat{x}(n + 1) = f(x(n), x(n - 1), x(n - 2), ..., x(n - m + 1), s(n)), \tag{16}$$

where m is the length of signal history, that is, the prediction order, $f(\cdot)$ is the transfer function of the network, and $s(n)$ denotes the current state of the network. For all testing experiments conducted in this report, we use the first 4000 samples of the time series for training. Then, all the adjustable connection weights are fixed and the trained networks undergo exploitation using the next 3000 samples. Mean square error (MSE) is used as the benchmark for quantizing the prediction accuracy.

4.1 RMLP Training With BPTT-GEKF and BPTT-DEKF

The recurrent multilayer perceptron has four layers, two of which are hidden layers. The input layer has 99 neurons, i.e., the prediction order of the RMLP is 99. Both the first and the second hidden layer has ten neurons, which have sigmoidal activation function. For accommodating the predictive modeling task, the output layer contains only one linear neuron. The recurrent connections are indeed first-order tap delay lines, which store the current output action potentials of the first hidden layer and feed them back to it as part of the input for calculating the layer output for the next time step. The neurons belonging to any neighboring layers, such as the input layer and the first hidden layer, are fully connected. Moreover, the recurrent connections are composed of 100 synapses so that each neuron in the first hidden layer receives feedback from all the neurons in the same layer. In the initialization stage, the connection weights in the network are randomly selected from the range $[-0.1, 0.1]$. The number of weight blocks g are 21 for BPTT-DEKF.

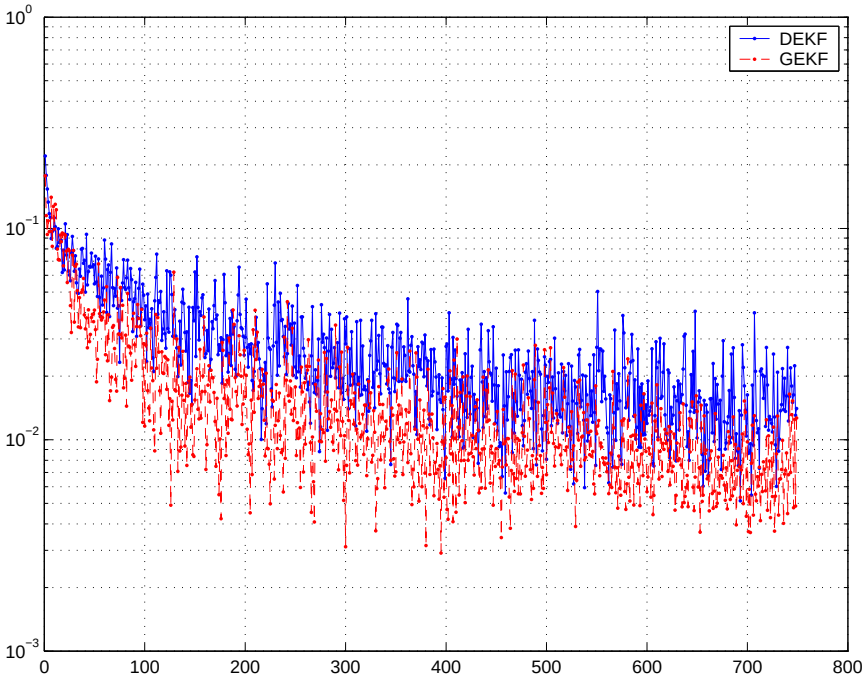

Fig. 4: The training process for BPTT-GEKF and BPTT-DEKF algorithms, where x-axis denotes the epoch number and y-axis denotes the MSE.

The training process has 750 epoches. Before each epoch starts, the network is reset by setting the values stored in the recurrent connections to zero and we re-initialize $\mathbf{K}(1,0)$ to be $\mathbf{K}(1,0) = \delta^{-1}\mathbf{I}$ with $\delta = 1E-2$. Then, we randomly choose an integer k in the range $[1, 3700]$ and then take the data points from the time step $k+1$ to time step $k+300$ in the training set to form a subset of 200 elements, where each element contains 99 inputs and one desired output. During the training period, the variance of the artificial noise $\mathbf{Q}(n)$ is linearly annealed from 10^{-2} to 10^{-6}, while the diagonal entry of the covariance matrix of the measurement noise decreases linearly from 100 to 5. The learning rate $\eta(n)$ is initially set to 1 and linearly reduced to 10^{-4} to avoid over-training. In calculating the derivative matrix $\mathbf{C}(n)$ using BPTT, a truncation depth of three is used, since we do not observe dramatic performance improvement for either DEKF or GEKF with truncation depth more than three.

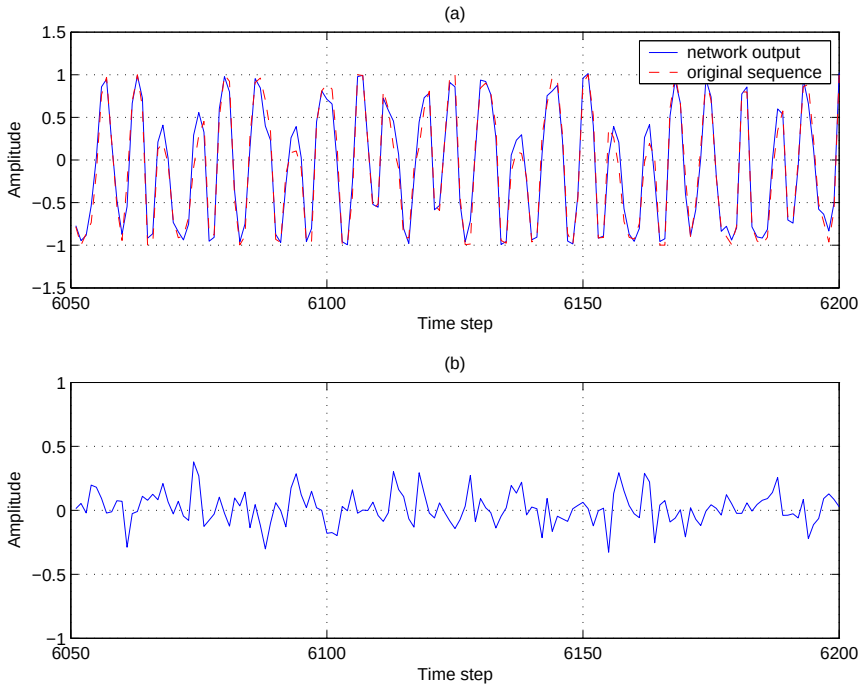

Fig. 5: The network prediction and the corresponding error curve for the RMLP trained with BPTT-GEKF algorithm

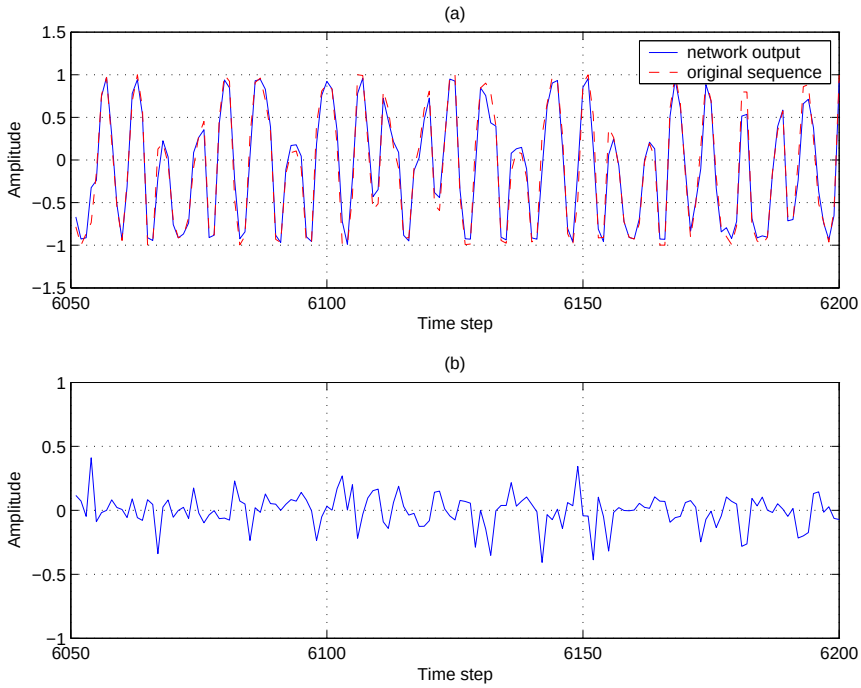

Fig. 6: The network prediction and the corresponding error curve for the RMLP trained with BPTT-DEKF algorithm

In Fig. 4, we plot the training MSE of each epoch for BPTT-GEKF and BPTT-DEKF. As we could see from the figure, BPTT-GEKF converges faster to an MSE of about 0.006, compared with BPTT-DEKF to a value of 0.011. And we illustrate the the prediction performance of trained networks in the testing stage in Fig. 5 for BPTT-GEKF and in Fig. 6 BPTT-DEKF, respectively.In both figures, sub-figure (a) shows the predicted time series generated by trained RMLPs, compared with the target signal, while sub-figure (b) demonstrates the error curve. For the sake of clarity, we only show the prediction performance over a selected region starting from the time step 6051 to 6200. It could be seen that both algorithms are able to train RMLPs to closely track the temporal evoluation of the phase-modulated sinusoidal signal, and BPTT-GEFK can render a network with higher prediction accuracy. More specifically, over the 3000 testing data, the RMLP trained by BPTT-GEKF has the prediction MSE of 0.0575, compared with a value of 0.0711 for the BPTT-DEKF trained RMLP. However, the better performance of BPTT-GEKF comes at the cost of higher computational complexity for the training process. For example, on a laptop with 1024MB memory and 1.8GHz AMD Turion processor, the average CPU running time for each iteration of BPTT-GEKF is around 2.5 seconds, compared with a value of 0.5 second for BPTT-DEKF.

4.2 ESN for Prediction Task

The ESN has a reservoir of 600 neurons with sigmoidal activation function. The reservoir weight matrix has a sparsity of 30%, i.e., 30% of the matrix elements are non-zeros. The non-zero elements are chosen randomly from the range $[-0.5, 0.5]$. The prediction order is 15. The input connection weights are randomly selected from the range $[-0.25, 0.25]$. Due to the predictive nature of the task, we do not use any feedback connections from the output neuron back to the reservoir. The activation function of the output neuron is also sigmoidal. The spectral radius of the reservoir is set to 0.3. The training is based on the signal from time step 1 to time step 4000 with washout time 1000 points.

In Fig. 7, we demonstrate the network performance in the predictive modeling task. In particular, in the subplot (a), we show the prediction results together with the target signal, while in the subplot (b), the error curve is given. Compared with Fig. 5 and Fig. 6, we can observe that powered by a large reservoir of 600 neurons, ESN can predict the nonlinear target signal with the highest accuracy. Of course, the performance we obtained is generated by one successful design of ESN after many trials. The prediction MSE of ESN over the 3000 testing data is around 0.0172.

5 Conclusion

Feedforward neural networks have been widely studied and used in many applications, while the recurrent neural networks are still not fully exploited because of its tedious training and complex system structure. We have developed a new recurrent neural network toolbox, namely the RNN-Tool, based on two RNN structures, the recurrent multilayer perceptron and echo state network. In RNN-Tool, we have implemented the RMLP and ESN as MATLAB structures and processes, including network generation, extended Kalman filter based training, linear regression and network testing. And we studied an exemplary predictive task as an example to show the validity of the toolbox. Simulation results have shown that: BPTT-GEKF algorithm can generate RMLP networks with higher prediction accuracy that that obtained via BPTT-DEKF at the cost of heavier computation induced by the training process; and the ESN approach has

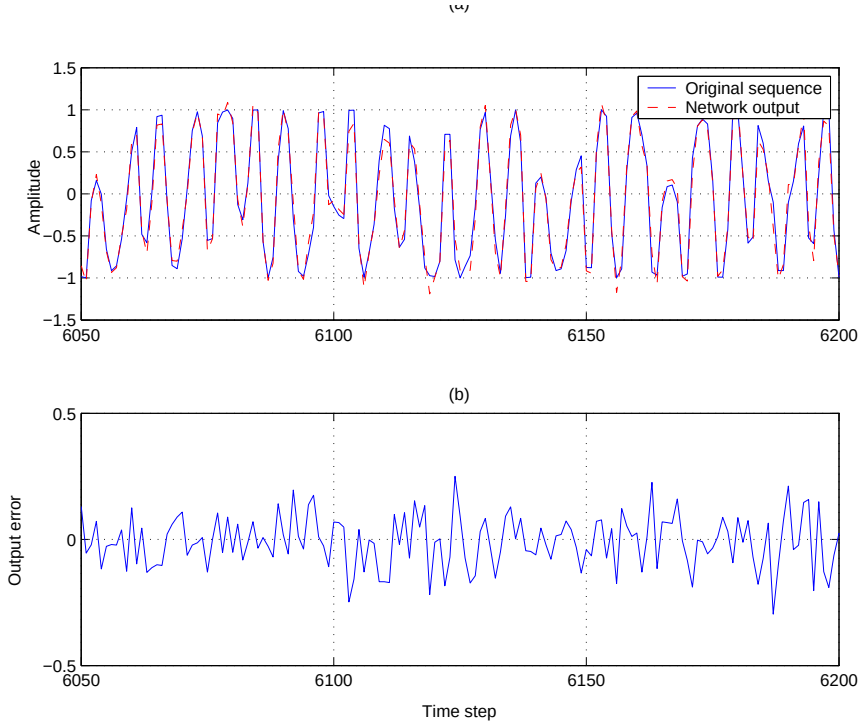

Fig. 7: The network prediction and the corresponding error curve for the echo state network

presented the highest performance, at the cost of introducing some heuristics in the initialization stage of the echo state network structure.

Acknowledgement

The authors would like to thank Dr. Simon Haykin for his valuable suggestions for the further development of ESN to its decoupled version, termed DESN [13]. The discussions with Mathangi Ganapathy and Jerome Vincent of Adaptive Systems Laboratory (ASL) during the MATLAB codes development are also appreciated.

References

[1] H. Jaeger, The echo state approach to recurrent neural networks, 2004. seminar slides, [Online]. Available: http://www.faculty.iu-brumen.de/hjaeger/courses/SeminarSpring04/ESNStandardslides.pdf

[2] G. V. Puskorius, L. A. Feldkamp, and L. I. Davis, "Dynamic neural network methods applied to on-vehicle idle speed control," in *Proceedings of the IEEE*, vol. 84, pp. 1407–1419, Oct. 1996.

[3] H. Jaeger, "The echo state approach to analysing and training recurrent neural networks," Fraunhofer Institute for Autonomous Intelligent Systems (AIS), GMD Report 148, Dec. 2001.

[4] H. Jaeger, "Harnessing nonlinearity: predicting chaotic systems and saving energy in wireless communication," *Science*, vol. 304, pp. 18-80, April 2004.

[5] W. Maass, T. Natschläger, and H. Markram, "Real-time computing without stable states: a new framework for neural computation based on perturbations," *Neural Computation*, vol. 14, no. 11, pp. 2531-2560, Nov. 2002.

[6] S. Haykin, *Neural Networks: A Comprehensive Foundation*. 2nd edition, Upper Saddle River, NJ: Prentice Hall PTR, 2004.

[7] P. J. Werbos, "Beyond regression: new tools for prediction and analysis in the behavioral sciences," PhD thesis, Harvard University, 1974.

[8] P. J. Werbos, "Backpropagation through time: What it does and how to do it," *Proceedings of IEEE*, vol. 78, no. 10, Oct., 1990, pp.1550-1560.

[9] R. J. Williams and J. Peng, "An effcient gradient-based algorithm for on-line training of recurrent network trajectories," *Neural Computation*, vol. 2, pp. 490-501, 1990.

[10] H. Jaeger, "Adaptive nonlinear system identification with echo state networks," in *Advances in Neural Information Processing Systems 15*, D. Derickson, Ed. Cambridge, MA: MIT Press, pp. 593-600, 2003

[11] P. G. Plöger, A. Arghir, T. Günther, and R. Hosseiny, "Echo state networks for mobile robot modeling and control," in *Proc. RoboCup*, pp. 157-168, 2003.

[12] K. Ishii, T. van der Zant, V. Becanovic, and P. Plöger, "Identificatin of motion with echo state network," in *Proc. OCEANS'04*, pp. 1205-1210, Nov. 2004.

[13] Y. Xue, L. Yang, and S. Haykin, "Decoupled echo state network with lateral inhibition," submitted to *IEEE Trans. on Neural Networks*.

APPENDIX

I Program Source Codes of RMLP

I.1 An exemplary main function of RMLP network training and testing

```
% Main function of RMLP Training and Testing;

%%%% Author: Yanbo Xue & Le Yang
%%%% ECE, McMaster University
%%%% yxue@grads.mcmaster.ca; yangl7@psychology.mcmaster.ca
%%%% May 11, 2006
%%%% This is a joint work by Yanbo and Le
%%%% For Project of Course of Dr. Haykin: Neural Network

% Globalize some variables
clear;
clc;
global NUM_EPOCH
global NUM_SUBSET
global LEN_SEQ
NUM_EPOCH    = 750;                % number of epochs
NUM_SUBSET   = 200;                % number of subsets in training data
LEN_SEQ      = 4000;               % length of sequence for training

% Generate RMLP network for training
net = rmlp_net(99,10,10,1);

%====================== BPTT-DEKF ====================================
[dekf_net_trained, dekf_mse, dekf_mse_cross] = rmlp_train_bptt_dekf(net);
% Plot training result
figure;
subplot(211);
plot(1:NUM_EPOCH, dekf_mse,'r.-');
hold on;
grid on;
set(gca,'YScale','log');
legend('Output RMSE for training Data');
xlabel('Number of epoch');
ylabel('RMSE');
subplot(212);
plot(1:NUM_EPOCH, dekf_mse_cross,'bx-');
set(gca,'YScale','log');
legend('RMSE of cross validated data');
xlabel('Number of epoch');
ylabel('RMSE');
hold on;
grid on;
```

```
% Test trained RMLP network
[dekf_original_out,dekf_net_out,dekf_error]  = rmlp_test(dekf_net_trained,'N');

%====================== BPTT-GEKF ====================================
[gekf_net_trained, gekf_mse, gekf_mse_cross] = rmlp_train_bptt_gekf(net);
% Plot training result
figure;
subplot(211);
plot(1:NUM_EPOCH, gekf_mse,'r.-');
hold on;
grid on;
set(gca,'YScale','log');
legend('Output RMSE for training Data');
xlabel('Number of epoch');
ylabel('RMSE');
subplot(212);
plot(1:NUM_EPOCH, gekf_mse_cross,'bx-');
set(gca,'YScale','log');
legend('RMSE of cross validated data');
xlabel('Number of epoch');
ylabel('RMSE');
hold on;
grid on;
% Test trained RMLP network
[gekf_original_out,gekf_net_out,gekf_error]  = rmlp_test(gekf_net_trained,'N');
```

I.2 BPTT-DEKF function

```
function [net_trained, mse, mse_cross] = rmlp_train_bptt_dekf(net)
% RMLP_train_bptt_dekf - Train the RMLP using BPTT-DEKF
% where the first hidden layer is recurrent and the second one is not.
% Bias input is not considered.
% ============================================
% net = rmlp_train_bptt_dekf(net, I_data, O_data)
% net         - network structure being trained
% net_trained - trained network
% mse         - RMSE of trained network
% mse_cross   - RMSE of cross-validated data

%%%% Author: Yanbo Xue & Le Yang
%%%% ECE, McMaster University
%%%% yxue@grads.mcmaster.ca; yangl7@psychology.mcmaster.ca
%%%% May 11, 2006
%%%% This is a joint work by Yanbo and Le
%%%% For Project of Course of Dr. Haykin: Neural Network

% Globalize some variables
global NUM_EPOCH
```

```
global NUM_SUBSET
global LEN_SEQ

%>>>>>>>>> Obtain parameters from RMLP net <<<<<<<<<<<<<<<<<<<<
ANC  = net.numAllNeurons;
IUC  = net.numInputUnits;
OUC  = net.numOutputUnits;
HUC1 = net.numHiddenLayer1;
HUC2 = net.numHiddenLayer2;
num_weights   = net.numWeights;
num_groups    = ANC;
len_subset    = IUC + OUC;           % length of subset
weights_all   = [net.weights.value]; % get weights value
weights_group = [net.weights.dest];  % define the group that the weights belong to
% Divide the weights of RMLP net into group from #1 to #ANC
for i = (1:num_groups),
    weights(i).value  = weights_all(min(find(weights_group == i)) : ...
                        max(find(weights_group == i)));
    weights(i).length = length(find(weights_group == i));
end;
%>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

%>>>>>>>>>>>>>>>> Initialization of Training <<<<<<<<<<<<<<<<<<
num_Epoch  = NUM_EPOCH;             % number of epochs
num_subset = NUM_SUBSET;            % number of subsets in training data
len_seq    = LEN_SEQ;               % length of sequence for training
R = annealing(100,5,num_Epoch);     % anneal R from 100 to 5
Q = annealing(1E-2,1E-6,num_Epoch); % anneal Q from 1E-2 to 1E-6
learning_rate = annealing(1,1E-5,num_Epoch); % learning_rate;
n = 1;                              % a counter for plotting
m = 1;                              % a counter for cross-validation plotting
timeflag = cputime;                 % a timer for saving the training time
start_point = ceil((len_seq-num_subset-len_subset+2)*rand(1,num_Epoch));
                                    %starting point of training data
%>>>>>>>>>>>>>>> End of training initialization >>>>>>>>>>>>>>>>

%>>>>>>>>> Main loop - Decoupled Extended Kalman Filter: DEKF <<<<<<<<<<<<<<
for k = (1:num_Epoch),
    %>>>>>>>>>>>>>>>>>>> Generate training data <<<<<<<<<<<<<<<<<<<<
    [I_data, T_data]  = seq_gen_rmlp(len_seq,len_subset,num_subset,start_point(k));
    [inpSize, inpNum] = size(I_data');
    [tarSize, tarNum] = size(T_data');
    if inpSize ~= IUC,
        error ('Number of input units and input pattern size do not match.');
    end;
    if tarSize ~= OUC,
        error ('Number of output units and target pattern size do not match.');
    end;
```

```
if inpNum ~= tarNum,
    error ('Number of input and output patterns are different.');
end;
%>>>>>>>>>>>> Set the waitbar - Initialization <<<<<<<<<<<<<<<<<
wb1 = waitbar(0, 'RMLP Neural Network Training (BPTT-DEKF) in Progress...');
set(wb1,'name',['Epoch = ' sprintf('%2.1f',k)]);
%>>>>>>>>>>>> Initialize some variables <<<<<<<<<<<<<<<<<<<<<<<
X1_0 = zeros(1,HUC1);
% Ricatti equation initialization
for i = (1:num_groups),
    K(i).value = 0.01^(-1)*eye(weights(i).length);
end;
weights0 = zeros(HUC1,HUC1+IUC);
%>>>>>>>>>>>>>>>> End of initialization >>>>>>>>>>>>>>>>>>>>>>>>

%>>>>>>>>>>> Initialization of recurrent layer states <<<<<<<<<<<
[X1_1 X2 out(1)] = rmlp_run(net,I_data(1,:),X1_0);
[X1_2 X2 out(2)] = rmlp_run(net,I_data(2,:),X1_1);
%>>>>>>>>>>>>>>> End of twice RMLP runnings >>>>>>>>>>>>>>>>>>>>

for j = (3:inpNum), % number of datasets
        %>>>>>>>>>>>>>>>>> Display Waitbar <<<<<<<<<<<<<<<<<<<<<
        waitbar(j/inpNum,wb1)
        set(wb1,'name',['Epoch = ' sprintf('%2.1f', k) ', Progress = '...
            sprintf('%2.1f',j/inpNum*100) '%']);
        %>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

        %>>>>>>>>>>>>>>>>>>> Initialization <<<<<<<<<<<<<<<<<<<<<<
        temp1    = 0;  % a temporatory variable in Gamma
        AA       = []; % a temporatory variable for re-grouping weights
        weights1 = []; % weights from input to first hidden layer
        weights2 = []; % weights from first to second hidden layer
        weights3 = []; % weights from second hidden layer to output layer

        % weights matrix between input layer and first hidden layer
        for i = (1:HUC1),
            weights1 = [weights1; weights(i).value];
        end;

        % weights matrix between first and second hidden layer
        for i = (HUC1+1:HUC1+HUC2),
            weights2 = [weights2; weights(i).value];
        end;

        % weights matrix between second hidden layer and output layer
        for i = (HUC1+HUC2+1:ANC),
            weights3 = [weights3; weights(i).value];
        end;
```

```
%>>>>>>>>>>>>>>> End of Initialization >>>>>>>>>>>>>>>>>>>>

%>>>>>>>>> Forward running of RMLP network <<<<<<<<<<<<<<<
[X1_3 X2 out(j)] = rmlp_run(net,I_data(j,:),X1_2);
%>>>>>>>>>>>>>>> End of Forward running >>>>>>>>>>>>>>>>>>>

%>>>>>>>>> Backward Propagation of Error <<<<<<<<<<<<<<<<<<
%>>>>>>>>>> Jacobian Matrix C Calculation - BPTT <<<<<<<<<<
% X2 (row vector): output of second hidden layer
% output neuron is supposed to be linear
for i = (HUC1+HUC2+1 : ANC),
    C(i).value = X2;
end;
% X1_3 (row vector): output of first hidden layer at time t
%D1 =  diag(weights3)*d_hyperb(weights2*X1_3')*X1_3;
D1 =  (weights3*diag(d_hyperb(weights2*X1_3')))'*X1_3;
for i = (HUC1+1 : HUC1+HUC2),
    C(i).value = D1(i-HUC1,:);
end;
% X1_2 (row vector): output of first hidden layer at time t-1
% [X1_2 I_data(t,:)]: input of first hidden layer at time t
% X1_1 (row vector): output of first hidden layer at time t-2
% [X1_1 I_data(t-1,:)]: input of first hidden layer at time t-1
D2 = (weights3*diag(d_hyperb(weights2*X1_3'))*...
     weights2*diag(d_hyperb(weights1*[X1_2 I_data(j,:)]')))'*...
     [X1_2 I_data(j,:)];
D2 = D2 + (weights3*diag(d_hyperb(weights2*X1_3')) * ...
     weights2*diag(d_hyperb(weights1*[X1_2 I_data(j,:)]'))* ...
     weights1(:,1:HUC1)*diag(d_hyperb(weights0*...
     [X1_1 I_data(j-1,:)])))'*[X1_1 I_data(j-1,:)];
for i = (1 : HUC1),
    C(i).value = D2(i,:);
end;
%>>>>>>>>>>>>>>>>>>>> End of Jacobian >>>>>>>>>>>>>>>>>>>>

%>>>>>>>> Decoupled Extended Kalman Filter <<<<<<<<<<<<<<<<<<
alpha = T_data(j) - out(j);  % innovation of output
for m = (1:num_groups),
    temp1 = C(m).value*K(m).value*C(m).value' + temp1;
end;
Gamma =  inv(temp1+R(k));
for i = (1:num_groups), % number of groups
    G(i).value = K(i).value*C(i).value'*Gamma;
    % Update the weights only if the innovation is larger than
    % a thereshould
    if abs(alpha) > 5E-2,
        weights(i).value = weights(i).value + ...
                           learning_rate(k)*(G(i).value*alpha)';
```

```
                end;
                % Re-calculte the Ricatti equation
                K(i).value = K(i).value - G(i).value*C(i).value*K(i).value + Q(k);
            end;
            %>>>>>>>>>>>>>>>>>>>> End of DEKF >>>>>>>>>>>>>>>>>>>>>>>>>

            %>>>>>>>>> Update the weights of the RMLP net <<<<<<<<<<<<<<<
            for i = (1:num_groups),
                AA = [AA, weights(i).value];
            end;
            for i = (1:num_weights),
                net.weights(i).value = AA(i);  % update weights of RMLP
            end;
            %>>>>>>>>>>>>>> End of weights updating <<<<<<<<<<<<<<<<<<<<<<

            % Recurrent states replacement
            X1_1 = X1_2;
            X1_2 = X1_3;
            % First layer weights replacement
            weights0 = weights1;
    end;
    %>>>>>>>>>>>>>>>>>>> End of One Epoch <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
    close(wb1);              % close waitbar.

    %>>>>>>>>>>>>>>>>>>>> Calculate RMSE  <<<<<<<<<<<<<<<<<<<<<<<<<<<
    mse(k) = sqrt(mean((out(1:end) - T_data(1:end)').^2));
    if mse(k) < 1E-2, break; end;
    mse_cross (k) = cross_validation(net);
    n = n+1;
    fprintf('Epoch: %d, Output RMSE: %f,...
             Cross-validated RMSE: %f\n', k, mse(k), mse_cross(k));
end;
%>>>>>>>>>>>>>>>>>>>>>>> End of Main Loop <<<<<<<<<<<<<<<<<<<<<<<<<<<<<
net_trained = net;
timeflag    = cputime - timeflag;
fprintf('Training accomplished! Total time is %2.2f hours',timeflag/3600);
```

I.3 BPTT-GEKF function

```
function [net_trained, mse, mse_cross] = rmlp_train_bptt_gekf(net)
% RMLP_train_bptt_gekf - Train the RMLP using BPTT-GEKF
% where the first hidden layer is recurrent and the second one is not.
% Bias input is not considered.
% ============================================
% net = rmlp_train_bptt_gekf(net, I_data, O_data)
% net         - network structure being trained
% net_trained - trained network
% mse         - RMSE of trained network
```

```
% mse_cross   - RMSE of cross-validated data

%%%% Author: Yanbo Xue & Le Yang
%%%% ECE, McMaster University
%%%% yxue@grads.mcmaster.ca; yangl7@psychology.mcmaster.ca
%%%% May 18, 2006
%%%% This is a joint work by Yanbo and Le
%%%% For Project of Course of Dr. Haykin: Neural Network

% Globalize some variables
global NUM_EPOCH
global NUM_SUBSET
global LEN_SEQ

%>>>>>>>>> Obtain parameters from RMLP net <<<<<<<<<<<<<<<<<<<
ANC  = net.numAllNeurons;
IUC  = net.numInputUnits;
OUC  = net.numOutputUnits;
HUC1 = net.numHiddenLayer1;
HUC2 = net.numHiddenLayer2;
num_weights   = net.numWeights;
num_groups    = ANC;
len_subset    = IUC + OUC;           % length of subset
weights_all   = [net.weights.value]; % get weights value
weights_group = [net.weights.dest];  % define the group that the weights belong to
% Divide the weights of RMLP net into group from #1 to #ANC
for i = (1:num_groups),
    weights(i).value  = weights_all(min(find(weights_group == i)) :...
                        max(find(weights_group == i)));
    weights(i).length = length(find(weights_group == i));
end;
%>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

%>>>>>>>>>>>>>>>> Initialization of Training <<<<<<<<<<<<<<<<<<
num_Epoch  = NUM_EPOCH;              % number of epochs
num_subset = NUM_SUBSET;             % number of subsets in training data
len_seq    = LEN_SEQ;                % length of sequence for training
R = annealing(100,5,num_Epoch);      % anneal R from 100 to 5
Q = annealing(1E-2,1E-6,num_Epoch); % anneal Q from 1E-2 to 1E-6
learning_rate = annealing(1,1E-5,num_Epoch); % learning_rate;
n = 1;                               % a counter for plotting
m = 1;                               % a counter for cross-validation plotting
timeflag = cputime;                  % a timer for saving the training time
start_point = ceil((len_seq-num_subset-len_subset+2)*rand(1,num_Epoch));
                                     %starting point of training data
%>>>>>>>>>>>>>>> End of training initialization >>>>>>>>>>>>>>>>

%>>>>>>>>> Main loop - Decoupled Extended Kalman Filter: DEKF <<<<<<<<<<<<<<
```

```
for k = (1:num_Epoch),
   %>>>>>>>>>>>>>>>>>>> Generate training data <<<<<<<<<<<<<<<<<<<<
    [I_data, T_data]  = seq_gen_rmlp(len_seq,len_subset,num_subset,start_point(k));
    [inpSize, inpNum] = size(I_data');
    [tarSize, tarNum] = size(T_data');
    if inpSize ~= IUC,
        error ('Number of input units and input pattern size do not match.');
    end;
    if tarSize ~= OUC,
        error ('Number of output units and target pattern size do not match.');
    end;
    if inpNum ~= tarNum,
        error ('Number of input and output patterns are different.');
    end;
    %>>>>>>>>>>>> Set the waitbar - Initialization <<<<<<<<<<<<<<<<<<
    wb1 = waitbar(0, 'RMLP Neural Network Training (BPTT-GEKF) in Progress...');
    set(wb1,'name',['Epoch = ' sprintf('%2.1f',k)]);
    %>>>>>>>>>>>> Initialize some variables <<<<<<<<<<<<<<<<<<<<<<<<
    X1_0 = zeros(1,HUC1);
    K_global = 0.01^(-1)*eye(num_weights);
    weights0 = zeros(HUC1,HUC1+IUC);
    %>>>>>>>>>>>>>>>> End of initialization >>>>>>>>>>>>>>>>>>>>>>>>

    %>>>>>>>>>>> Initialization of recurrent layer states <<<<<<<<<<<
    %Run the RMLP network for two times to get the initial recurrent
    %network states X1_1 and X1_2
    [X1_1 X2 out(1)] = rmlp_run(net,I_data(1,:),X1_0);
    [X1_2 X2 out(2)] = rmlp_run(net,I_data(2,:),X1_1);
    %>>>>>>>>>>>>>>> End of twice RMLP runnings >>>>>>>>>>>>>>>>>>>>

    for j = (3:inpNum), % number of datasets
            %>>>>>>>>>>>>>>>>>> Display Waitbar <<<<<<<<<<<<<<<<<<<<<<
            waitbar(j/inpNum,wb1)
            set(wb1,'name',['Epoch = ' sprintf('%2.1f', k) ',Progress = '...
                sprintf('%2.1f',j/inpNum*100) '%']);
            %>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

            %>>>>>>>>>>>>>>>>>>> Initialization <<<<<<<<<<<<<<<<<<<<<<
            temp1 = 0;     % a temporatory variable in Gamma
            weights1 = []; % weights from input to first hidden layer
            weights2 = []; % weights from first to second hidden layer
            weights3 = []; % weights from second hidden layer to output layer
            C_global = []; % C for Global EKF

            % weights matrix between input layer and first hidden layer
            for i = (1:HUC1),
                weights1 = [weights1; weights(i).value];
            end;
```

```
% weights matrix between first and second hidden layer
for i = (HUC1+1:HUC1+HUC2),
    weights2 = [weights2; weights(i).value];
end;

% weights matrix between second hidden layer and output layer
for i = (HUC1+HUC2+1:ANC),
    weights3 = [weights3; weights(i).value];
end;
%>>>>>>>>>>>>>> End of Initialization >>>>>>>>>>>>>>>>>>>

%>>>>>>>>> Forward running of RMLP network <<<<<<<<<<<<<<<
[X1_3 X2 out(j)] = rmlp_run(net,I_data(j,:),X1_2);
%>>>>>>>>>>>>>> End of Forward running >>>>>>>>>>>>>>>>>>>

%>>>>>>>>> Backward Propagation of Error <<<<<<<<<<<<<<<<<<
%>>>>>>>>>>>>>> Jacobian Matrix C Calculation <<<<<<<<<<<<<
% X2 (row vector): output of second hidden layer
% output neuron is supposed to be linear
C_global = [reshape(X2,1,OUC*HUC2), C_global];
% X1_3 (row vector): output of first hidden layer at time t
D1 = (weights3*diag(d_hyperb(weights2*X1_3')))'*X1_3;
C_global = [reshape(D1',1,HUC1*HUC2),C_global];
% X1_2 (row vector): output of first hidden layer at time t-1
% [X1_2 I_data(t,:)]: input of first hidden layer at time t
% X1_1 (row vector): output of first hidden layer at time t-2
% [X1_1 I_data(t-1,:)]: input of first hidden layer at time t-1
D2 = (weights3*diag(d_hyperb(weights2*X1_3'))*...
     weights2*diag(d_hyperb(weights1*[X1_2 I_data(j,:)]')))'*...
     [X1_2 I_data(j,:)];
D2 = D2 + (weights3*diag(d_hyperb(weights2*X1_3')) * ...
     weights2*diag(d_hyperb(weights1*[X1_2 I_data(j,:)]'))* ...
     weights1(:,1:HUC1)*diag(d_hyperb(weights0*...
     [X1_1 I_data(j-1,:)])))' *[X1_1 I_data(j-1,:)];
C_global = [reshape(D2',1,(HUC1+IUC)*HUC2),C_global];
%>>>>>>>>>>>>>>>>>>>> End of Jacobian >>>>>>>>>>>>>>>>>>>>


%>>>>>>>> Decoupled Extended Kalman Filter <<<<<<<<<<<<<<<<<<
alpha = T_data(j) - out(j);  % innovation of output
temp1 = C_global*K_global*C_global'; % about 0.2 seconds for this step
Gamma =  inv(temp1+R(k));
G_global = K_global*C_global'*Gamma;
% Update the weights only if the innovation is larger than
% a thereshould
if abs(alpha) > 5E-2,
    weights_all = weights_all + learning_rate(k)*(G_global*alpha)';
```

```
            end;
            % Re-calculte the Ricatti equation - Most time-consuming step
            K_global = K_global - G_global*C_global*K_global + Q(k)*eye(num_weights);
            %>>>>>>>>>>>>>>>>>>>> End of DEKF >>>>>>>>>>>>>>>>>>>>>>>>>

            %>>>>>>>>> Update the weights of the RMLP net <<<<<<<<<<<<<<<
            for i = (1:num_weights),
                net.weights(i).value = weights_all(i);
            end;
            %>>>>>>>>>>>>>> End of weights updating <<<<<<<<<<<<<<<<<<<<<

            %>>>>>>>>>> Redivide the weights into group <<<<<<<<<<<<<<<<<
            for i = (1:num_groups),
                weights(i).value  = weights_all(min(find(weights_group == i)) :...
                max(find(weights_group == i)));
            end;
            %>>>>>>>>>>>>>>>> End of weights division <<<<<<<<<<<<<<<<<<<

            % Recurrent states replacement
            X1_1 = X1_2;
            X1_2 = X1_3;
            weights0 = weights1;
    end;
    %>>>>>>>>>>>>>>>>>>> End of One Epoch <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
    close(wb1);              % close the waitbar

    %>>>>>>>>>>>>>>>>>>> Calculate RMSE  <<<<<<<<<<<<<<<<<<<<<<<<<
    mse(k) = sqrt(mean((out(1:end) - T_data(1:end)').^2));
    if mse(k) < 1E-5, break; end;
    fprintf('Epoch: %d, RMSE: %f\n', k, mse(k));
    mse_cross(k) = cross_validation(net);
    n = n+1;
    fprintf('Epoch: %d, Output RMSE: %f,...
             Cross-validated RMSE: %f\n', k, mse(k), mse_cross(k));
end;
%>>>>>>>>>>>>>>>>>>>>>>> End of Main Loop <<<<<<<<<<<<<<<<<<<<<<<<<<<<
net_trained = net;
timeflag = cputime - timeflag;
fprintf('Training accomplished! Total time is %2.2f hours',timeflag/3600);
```

I.4 Generation function of the RMLP network for training

```
function net = rmlp_net(IUC, HUC1, HUC2, OUC)
% RMLP_net - setup recurrent multilayer perceptron network
% where the first hidden layer is recurrent and the second
% one is not.
% Bias input is not considered.
% net = rmlp_net(IUC, HUC1, HUC2, OUC)
```

```
% where
% ======================================================
% Outputs include:
% net - new network structure
% ======================================================
% Inputs include:
% IUC  - number of input units
% HUC1 - number of hidden units for the first hidden layer
% HUC2 - number of hidden units for the second hidden layer
% OUC  - number of output units

%%%% Author: Yanbo Xue & Le Yang
%%%% ECE, McMaster University
%%%% yxue@grads.mcmaster.ca; yangl7@psychology.mcmaster.ca
%%%% May 10, 2006
%%%% This is a joint work by Yanbo and Le
%%%% For Project of Course of Dr. Haykin: Neural Network

% set number of all units
AUC = IUC + 2*HUC1 + HUC2 + OUC;

% set number of all neurons
ANC = HUC1 + HUC2 + OUC;

% set numbers of units
net.numInputUnits    = IUC;
net.numOutputUnits   = OUC;
net.numHiddenLayers  = 2;
net.numHiddenLayer1  = HUC1;
net.numHiddenLayer2  = HUC2;
net.numAllUnits      = AUC;
net.numAllNeurons    = ANC;

% set neuron masks
net.maskInputUnits   = [zeros(HUC1,1); ones(IUC, 1); zeros(AUC-IUC-HUC1, 1)];
net.maskOutputUnits  = [zeros(AUC-OUC, 1); ones(OUC, 1)];
net.indexInputUnits  = find(net.maskInputUnits);
net.indexOutputUnits = find(net.maskOutputUnits);

% number of weights: initialization
n=1;
% set weights
weight = struct('dest',0,'source',0,'layer',0,'delay',0,...
                'value',0,'const',false,'act',1,'wtype',1);

% weights for input layer to first hidden layer
for i = (1 : HUC1),
    % recurrent weights
```

```
    for j = (1 : HUC1),
        net.weights(n) = weight;
        net.weights(n).dest   = i;
        net.weights(n).source = j;
        net.weights(n).layer  = 1;
        net.weights(n).delay  = 1;
        n = n+1;
    end;
    % weights for input to first hidden layer
    for j = (HUC1+1 : IUC+HUC1),
        net.weights(n) = weight;
        net.weights(n).dest   = i;
        net.weights(n).source = j;
        net.weights(n).layer  = 1;
        n = n+1;
    end;
end;

% weights for first hidden layer to second hidden layer
for i = (HUC1+1 : HUC1+HUC2),
    for j = (1 : HUC1),
        net.weights(n) = weight;
        net.weights(n).dest   = i;
        net.weights(n).source = j;
        net.weights(n).layer  = 2;
        n = n+1;
    end;
end;

% weights for first hidden layer to second hidden layer
for i = (HUC1+HUC2+1 : HUC1+HUC2+OUC),
    for j = (1 : HUC2),
        net.weights(n) = weight;
        net.weights(n).dest   = i;
        net.weights(n).source = j;
        net.weights(n).layer  = 3;
        n = n+1;
    end;
end;

% set number of weights
net.numWeights = n-1;

% initialize weight matrices from [-0.25, 0.25]
for i=(1:net.numWeights),
    net.weights(i).value = rand ./ 2  - 0.25;
end;
```

I.5 Running function of the RMLP network

```
function [X1, X2, out] = rmlp_run(net,I_data,R_data)
% RMLP_run - Run RMLP
% where the first hidden layer is recurrent and the second
% one is not.
% Bias input is not considered.
% [X1,X2,out] = rmlp_run(net, I_data, R_data)
% Input:
% net    - trained RMLP network
% I_data - input data
% R_data - recurrent data of last state
% Output:
% X1     - output of the first hidden layer
% X2     - output of the second hidden layer
% out    - network output

%%%% Author: Yanbo Xue & Le Yang
%%%% ECE, McMaster University
%%%% yxue@grads.mcmaster.ca; yangl7@psychology.mcmaster.ca
%%%% May 12, 2006
%%%% This is a joint work by Yanbo and Le
%%%% For Project of Course of Dr. Haykin: Neural Network

% Fetch parameters from net
ANC  = net.numAllNeurons;
IUC  = net.numInputUnits;
OUC  = net.numOutputUnits;
HUC1 = net.numHiddenLayer1;
HUC2 = net.numHiddenLayer2;
weights_all   = [net.weights.value]; % get weights value
weights_group = [net.weights.dest]; % define the group that the weights belong to

% divide the weights of the net into group from #1 to #ANC
for i = (1:ANC),
    weights(i).value  = weights_all(min(find(weights_group == i)) :...
                        max(find(weights_group == i)));
    weights(i).length = length(find(weights_group == i));
end;

% parameter checking
[inpSize, inpNum] = size(I_data');
[recSize, recNum] = size(R_data');
if inpSize ~= IUC,
    error ('Number of input units and input pattern size do not match.');
end;
if recSize ~= HUC1,
    error ('Number of recurrent units do not match.');
```

```
end;

X1 = []; % output row vector of first hidden layer
X2 = []; % output row vector of second hidden layer
out= []; % network output row vector
% output of first hidden layer
for i = (1:HUC1),
    x(i).value = hyperb(weights(i).value*[R_data,I_data]');
    X1 = [X1, x(i).value];
end;

% output of second hidden layer
for i = (HUC1+1:HUC1+HUC2),
    x(i).value = hyperb(weights(i).value*X1');
    X2 = [X2, x(i).value];
end;

% output of the network - linear neuron
for i = (HUC1+HUC2+1:ANC),
    x(i).value = weights(i).value*X2';
    out = [out, x(i).value];
end;
```

I.6 Testing function of the trained RMLP network

```
function [original_out,net_out,error] = rmlp_test(net_trained,RT_plot)
% [original_out,net_out,error] = rmlp_test(net_trained,RT_plot)
% Testing the performance of trained RMLP network.
% where:
% ========================================================
% Inputs include:
% net_trained  -    the trained network used for testing
% RT_plot      -    real-time plot option: 'Y' or 'N'
% ========================================================
% Outputs include:
% original_out -    original output of the test sequence
% net_out      -    network output
% error        -    error of the testing data (= net_out - original_out)

%%%% Author: Yanbo Xue & Le Yang
%%%% ECE, McMaster University
%%%% yxue@grads.mcmaster.ca; yangl7@psychology.mcmaster.ca
%%%% May 18, 2006
%%%% This is a joint work by Yanbo and Le
%%%% For Project of Course of Dr. Haykin: Neural Network


%>>>>>>>>>>>>>>>>>>>>>> Initialization <<<<<<<<<<<<<<<<<<<<<<<<
```

```
HUC1 = net_trained.numHiddenLayer1; % Number of first hidden layer neurons
IUC  = net_trained.numInputUnits;   % Number of input layer neurons (linear)
OUC  = net_trained.numOutputUnits;  % Number of output layer neurons
X1   = zeros(1,HUC1);               % Initial state of the recurrent input
len_subset   = IUC+OUC;             % Length of the subset data
num_testdata = 2000;                % Number of testing data
S_point      = 6000;   %Starting point of the testing data generation
                       %The first testing data should be at S_point+len_subset-1
TS_point= S_point + len_subset -1;  % Starting point of testing data output
TE_point= TS_point + num_testdata -1;%Ending point of testing data
t = [S_point : S_point+len_subset-1];% Vector t for generating testing data

%>>>>>> Set up waitbar to show the process of tesing <<<<<<<<<
wb2 = waitbar(0, 'RMLP Neural Network Testing in Progress...');

%>>>>>>>>>>>>>>>>>> Main testing loop <<<<<<<<<<<<<<<<<<<<<<<<
for i = (TS_point : TE_point),
    y = signal(t); % Testing function
    [X11, X2, out] = rmlp_run(net_trained,y(1:len_subset-1),X1);

    if (RT_plot == 'Y'),    % 'YES' for real-time plotting
        if (i>=1+TS_point), % Draw a line to former data when i>=2
            % Draw first subfigure for original output vs network output
            subplot(211);
            plot(i,y(len_subset),'b',i,out,'r'); %Draw output data pixels
            line([i-1,i],[y0,y(len_subset)],'Color','b');
            line([i-1,i],[out0,out],'Color','r');
            hold on; grid on;
            legend('Original sequence','Network output');
            xlabel('Time'); ylabel('Amplitude');
            % Rolling display windows when the data number > 100
            if (i>100+TS_point),
                xlim([i-100,i]);
            end;

            % Draw second subfigure for original output vs network output
            subplot(212);
            plot(i,out-y(len_subset),'b'); hold on; grid on;
            line([i-1,i],[error0,out-y(len_subset)],'Color','b');
            xlabel('Time'); ylabel('Output error');
            % Rolling display windows when the data number > 100
            if (i>100+TS_point),
                xlim([i-100,i]);
            end;
        end;
    end;

    y0     = y(len_subset);    %Save as former original output
```

```
    out0   = out;              %Save as former network output
    error0 = out-y(len_subset);%Save as former error

    % For final plotting with option of 'Non-real-time drawing'
    original_out(i-TS_point+1) = y0;
    net_out(i-TS_point+1)      = out0;
    error(i-TS_point+1)        = error0;

    t = t + 1;  % Move one-step forward
    X1 = X11;   % Save state for next recurrent network input

    %>>>>>>>>>>>>>> Update Waitbar <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
    waitbar((i-TS_point)/(TE_point-TS_point),wb2);
    set(wb2,'name',['Progress = ' sprintf('%2.1f',...
        (i-TS_point)/(TE_point-TS_point)*100) '%']);
end;
close(wb2);     % Close waitbar

%>>>>>>>>>>> Final Plotting For Overall Data <<<<<<<<<<<<<<<<<<<<<<<
if (RT_plot ~= 'Y'),
    figure;
    subplot(211);
    plot([TS_point:TE_point],original_out,'b',[TS_point:TE_point],net_out,'r');
    hold on; grid on;
    legend('Original sequence','Network output');
    xlabel('time'); ylabel('Amplitude');
    subplot(212);
    plot([TS_point:TE_point],error,'b');
    hold on; grid on;
    xlabel('Time'); ylabel('Output error');
end;

%>>>>>>> Harness the might-happening warnings <<<<<<<<<<<<<<<<<<<<<
warning off MATLAB:colon:operandsNotRealScalar;
```

I.7 Generation function of the training data sets

```
function [I_data, T_data] = seq_gen_rmlp(len_seq,len_subset,num_subset,start_point)
% function: [I_data, T_data] = seq_gen_rmlp(len_subset)
% generate training sequence for RMLP network.
% randomly choose a starting point and generate overlapped training data
% where I_data    - Input data of training sequence
%       T_data    - Target data of training sequence
%       len_subset- subset length = I_data + T_data
%       len_seq   - length of sequence: select data from 0 : leq_seq-1
%       num_subset- number of subsets
%       start_point - starting point of the subset
% See also: seq_gen_esn
```

```
%%%% Author: Yanbo Xue & Le Yang
%%%% ECE, McMaster University
%%%% yxue@grads.mcmaster.ca; yangl7@psychology.mcmaster.ca
%%%% July 6, 2006
%%%% This is a joint work by Yanbo and Le
%%%% For Project of Course of Dr. Haykin: Neural Network

t = 0 : len_seq-1;
y = signal(t);
for i = (1:num_subset),
    I_data(i,:) = y(start_point + (i-1) : start_point + len_subset + i - 3);
    T_data(i,:) = y(start_point + len_subset + i - 2);
end;
```

I.8 Cross-validation function of the trained RMLP network

```
function mse = cross_validation(net)
% [mse1,mse2] = cross_validation(net, time_index)
% Cross-validation function for using in training
% where:
% ================================================
% Inputs include
% net        - the RMLP network for cross-validation
% ================================================
% Outputs include
% mse        - RMSE of the data

%%%% Author: Yanbo Xue & Le Yang
%%%% ECE, McMaster University
%%%% yxue@grads.mcmaster.ca; yangl7@psychology.mcmaster.ca
%%%% May 18, 2006
%%%% This is a joint work by Yanbo and Le
%%%% For Project of Course of Dr. Haykin: Neural Network

IUC  = net.numInputUnits;
OUC  = net.numOutputUnits;
HUC1 = net.numHiddenLayer1;
t    = 6000:6299;
y    = signal(t);
len_subset    = IUC + OUC;           % length of subset
X1   = rand(1,HUC1);

for i = (1:length(t)-(len_subset-1)),
    I_data(i,:) = y(i: i+(len_subset-2));
    T_data(i,:) = y(i+(len_subset-1));
end;
```

```
for i = 1:length(t)-(len_subset-1),
    [X11, X2, out(i)] = rmlp_run(net,I_data(i,:),X1);
    X1 = X11;
end;
mse = sqrt(mean((out(1:end) - T_data(1:end)').^2));
```

I.9 Generation function of target signals for both RMLP and ESN

```
function x = signal(t)
% function: x = signal(t)
% where t  - time_interval
%       x  - signal output

%%%% Author: Yanbo Xue & Le Yang
%%%% ECE, McMaster University
%%%% yxue@grads.mcmaster.ca; yangl7@psychology.mcmaster.ca
%%%% June 6, 2006
%%%% This is a joint work by Yanbo and Le
%%%% For Project of Course of Dr. Haykin: Neural Network

x = sin(t+sin(t.^2));
```

II Program Source Codes of ESN

II.1 An exemplary main function of ESN training and testing

```
% Main function of ESN Training and Testing;

%%%% Author: Yanbo Xue & Le Yang
%%%% ECE, McMaster University
%%%% yxue@grads.mcmaster.ca; yangl7@psychology.mcmaster.ca
%%%% May 11, 2006
%%%% This is a joint work by Yanbo and Le
%%%% For Project of Course of Dr. Haykin: Neural Network

clear;
clc;
% Generate ESN for training
net = esn_net(25, 600, 1);

% Generate training data
[I_data, T_data] = seq_gen_esn(26);

% Train ESN
net_trained = esn_train(net,I_data,T_data);

% Test ESN
[original_out,net_out,error] = esn_test(net_trained);
```

II.2 Generation of the ESN for training

```
function net = esn_net(IUC, HUC, OUC)
% esn_net - setup Echo State Network
% net = esn_net(IUC, HUC, OUC)
% where
% ======================================================
% Outputs include:
% net  : new network structure
% ======================================================
% Inputs include:
% IUC  : number of input units
% HUC  : number of hidden units in the dynamic reservoir
% OUC  : number of output units
% See also: esn_train, esn_test, seq_gen_esn, rmlp_net

%%%% Author: Yanbo Xue & Le Yang
%%%% ECE, McMaster University
%%%% yxue@grads.mcmaster.ca; yangl7@psychology.mcmaster.ca
%%%% May 19, 2006
%%%% This is a joint work by Yanbo and Le
%%%% For Project of Course of Dr. Haykin: Neural Network

% set number of all units
AUC = IUC + HUC + OUC;
net.bl_out = 1;  % 1: the output is linear neuron, 0: not
net.int_bk = 0; % intensity of feedback weights
net.attenu = 2/3; % attenuation ratio for the signal

% set numbers of units
net.numInputUnits    = IUC;
net.numOutputUnits   = OUC;
net.numHiddenLayer   = HUC;
net.numAllUnits      = AUC;

% set neuron masks
net.maskInputUnits   = [ones(IUC, 1); zeros(AUC-IUC, 1)];
net.maskOutputUnits  = [zeros(AUC-OUC, 1); ones(OUC, 1)];
net.indexInputUnits  = find(net.maskInputUnits);
net.indexOutputUnits = find(net.maskOutputUnits);

% weights matrix initialization
dr_sp_den = 0.3;          % sparse density of reservoir weights matrix
alpha     = 0.7;          % spectral radius to scale reservoir weights
W0        = 2*rand(HUC) - 1;           % element of W0 in [-1,1];
W0        = W0.*(rand(HUC)<dr_sp_den); % let W0 be sparse with density dr_sp_den
net.reservoirWeights = alpha*W0/max(abs(eig(W0))); % dynamic reservoir weights matrix
net.inputWeights     = rand(HUC,IUC)-0.5;          % input weights
```

```
net.backWeights      = (rand(HUC,OUC)-0.5);        % backward weights from output layer
net.outputWeights    = zeros(OUC,HUC);             % output weights matrix
```

II.3 Training function of the ESN

```
function net_trained = esn_train(net,I_data,T_data)
% esn_train - train Echo State Network
% net_trained = esn_train(net)
% where
% ====================================================
% Inputs include:
% net    - ESN to be trained
% I_data - Input data of training sequence
% T_data - Target data of training sequence
% ====================================================
% Outputs include:
% net_trained  - ESN after training
% See also: seq_gen_esn, esn_net, esn_test

%%%% Author: Yanbo Xue & Le Yang
%%%% ECE, McMaster University
%%%% yxue@grads.mcmaster.ca; yangl7@psychology.mcmaster.ca
%%%% May 19, 2006
%%%% This is a joint work by Yanbo and Le
%%%% For Project of Course of Dr. Haykin: Neural Network

%>>>>>>>>> Obtain parameters from RMLP net <<<<<<<<<<<<<<<<<<
AUC  = net.numAllUnits;           % number of all units
IUC  = net.numInputUnits;         % number of input units
OUC  = net.numOutputUnits;        % number of output units
HUC  = net.numHiddenLayer;        % number of hidden units
drWeights = net.reservoirWeights; % dynamic reservoir weights matrix
inWeights = net.inputWeights;     % input matrix
bkWeights = net.backWeights;      % backward weights from output layer
ouWeights = net.outputWeights;    % output weights matrix
bl_out    = net.bl_out;           % type of output neuron
int_bk    = net.int_bk;           % intensity of feedback
attenu    = net.attenu;           % attenuation ratio for the signal

%>>>>>>>>>>>>>>>>> Parameter Check <<<<<<<<<<<<<<<<<<<<<<<<<
[inpSize, inpNum] = size(I_data');
[tarSize, tarNum] = size(T_data');
if inpSize ~= IUC,
    error ('Number of input units and input pattern size do not match.');
end;
if tarSize ~= OUC,
    error ('Number of output units and target pattern size do not match.');
end;
```

```
if inpNum ~= tarNum,
    error ('Number of input and output patterns are different.');
end;

%>>>>>>>>>>>  Initialization of Training <<<<<<<<<<<<<<<<<<<<
I_data  = attenu*I_data;
T_data  = attenu*T_data;
X(1,:)  = zeros(1,HUC);               % initial reservoir state
I1_data = [zeros(1,inpSize); I_data]; % add zero to initial input
T1_data = [zeros(1,tarSize); T_data]; % add zero to initial output
timeflag= cputime;                    % a timer to save the training time
wb = waitbar(0, 'Echo State Network Training in Progress...');
T0 = 1000;                            % washout time
fprintf('\nThe echo state network training is in process...\n');

%>>>>>>>>>>>>>>> Main Loop of ESN Training <<<<<<<<<<<<<<<<<<
for i = (1:inpNum),
    waitbar(i/inpNum,wb)
    set(wb,'name',['Progress = ' sprintf('%2.1f',i/inpNum*100) '%']);
    X(i+1,:) = hyperb((inWeights*I1_data(i+1,:)' + drWeights*X(i,:)' + ...
                int_bk*bkWeights*T1_data(i,:)' + 0.001*(rand(1,HUC)-0.5)')');
end;
close(wb);
fprintf('Please wait for another while...\n');

%>>>>>> Calculate output weights and update ESN <<<<<<<<<<<<<
if (bl_out == 1),
    ouWeights = (pinv(X(T0+2:end,:))*(T_data(T0+1:end,:)))'; % linear output
else
    ouWeights = (pinv(X(T0+2:end,:))*(inv_hyperb(T_data(T0+1:end,:))))';
end;
net.outputWeights = ouWeights;
net_trained       = net;
timeflag          = cputime - timeflag;
fprintf('Training accomplished! Total time is %2.2f hours.\n',timeflag/3600);
```

II.4 Testing function of the ESN

```
function [original_out,net_out,error] = esn_test(net)
% esn_test - test Echo State Network
% [original_out,net_out,error] = esn_test(net)
% where
% ======================================================
% Inputs include:
% net           - ESN to be tested
% ======================================================
% Outputs include:
% original_out  - original output of testing data
```

```
% net_out       - ESN output
% error         - error of output
% See also: esn_train, esn_net, seq_gen_esn

%%%% Author: Yanbo Xue & Le Yang
%%%% ECE, McMaster University
%%%% yxue@grads.mcmaster.ca; yangl7@psychology.mcmaster.ca
%%%% May 20, 2006
%%%% This is a joint work by Yanbo and Le
%%%% For Project of Course of Dr. Haykin: Neural Network

%>>>>>>>>> Obtain parameters from RMLP net <<<<<<<<<<<<<<<<<<
AUC  = net.numAllUnits;           % number of all units
IUC  = net.numInputUnits;         % number of input units
OUC  = net.numOutputUnits;        % number of output units
HUC  = net.numHiddenLayer;        % number of hidden units
drWeights = net.reservoirWeights; % dynamic reservoir weights matrix
inWeights = net.inputWeights;     % input matrix
bkWeights = net.backWeights;      % backward weights from output layer
ouWeights = net.outputWeights;    % output weights matrix
len_subset= IUC + OUC;            % subset length
bl_out    = net.bl_out;           % type of output neuron
int_bk    = net.int_bk;           % intensity of feedback
attenu    = net.attenu;           % attenuation ratio for the signal

%>>>>>>>>>>>> Testing Parameters Setting <<<<<<<<<<<<<<<<<<<
S_point = 4000;            % starting point of testing data
testNum = 3000;            % number of testing data
X(1,:)  = zeros(1,HUC);    % initial reservoir state
t       = [S_point : S_point+len_subset-1];
y0      = rand(1,OUC)-0.5; % initial output

%>>>>>>>>>>>>>>>>>>>>>>>> Check parameter <<<<<<<<<<<<<<<<<<
if length(t) ~= len_subset,
    error('Length of testing data subset and the network structure do not match');
end;

%>>>>>>>>>>>>>>>> Testing Main Routine <<<<<<<<<<<<<<<<<<<<<<
wb = waitbar(0, 'Echo State Network Testing in Progress...');
for i = (1:testNum),
    waitbar(i/testNum,wb)
    set(wb,'name',['Progress = ' sprintf('%2.1f',i/testNum*100) '%']);
    y = attenu*signal(t);                          % generate testing data
    X(i+1,:) = hyperb((inWeights*y(1:end-OUC)' + ...
               drWeights*X(i,:)' + int_bk*bkWeights*y0')');
    if (bl_out == 1),
        Y(i+1,:) = ouWeights*X(i+1,:)';            % linear output
    else
```

```
        Y(i+1,:) = hyperb(ouWeights*X(i+1,:)');    % nonlinear output
    end;

    % update state for next iteration and output
    original_out(i) = (1/attenu)*y(end-OUC+1:end); % original output
    net_out(i)      = (1/attenu)*Y(i+1,:);         % network output
    error(i)        = net_out(i) - original_out(i);% errors
    y0 = Y(i+1,:);                                 % save the output for next step
    t  = t + 1;                                    % Move one-step forward
end;
close(wb);

%>>>>>>>>>>>>>>>>>> Plotting <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
subplot(211);
plot([S_point+1:S_point+testNum],original_out,'b',...
     [S_point+1:S_point+testNum],net_out,'r');
hold on; grid on;
legend('Original sequence','Network output');
xlabel('time'); ylabel('Amplitude');
subplot(212);
plot([S_point+1:S_point+testNum],error,'b');
hold on; grid on;
xlabel('Time'); ylabel('Output error');
RMSE = sqrt(mean((net_out(1:end) - original_out(1:end)).^2))
```

II.5 Generation function of the training data sets for ESN

```
function [I_data, T_data] = seq_gen_esn(len_subset)
% seq_gen_esn: generate training sequence for ESN
% [I_data, T_data] = seq_gen_esn(len_subset)
% where
% len_subset - length of subset = IUC + OUC
% I_data     - Input data of the network
% T_data     - Target data of the network

%%%% Author: Yanbo Xue & Le Yang
%%%% ECE, McMaster University
%%%% yxue@grads.mcmaster.ca; yangl7@psychology.mcmaster.ca
%%%% May 20, 2006
%%%% This is a joint work by Yanbo and Le
%%%% For Project of Course of Dr. Haykin: Neural Network

%>>>>>>>>>>>>> Initilization <<<<<<<<<<<<<<<<<<<<<
t = 0 : 3999;                     % training sequence time interval
y = signal(t);                    % generate training sequence
len = length(y);
incom_data = (rand(1,len)>0.00);  % incomplete data ratio
y = y.*incom_data;
```

```
num_subset = len - len_subset + 1; % number of subset
fprintf('Training sequence generation is in process, please wait...\n')
%>>>>>>>>>>>>>>>>> Main Loop <<<<<<<<<<<<<<<<<<<<
for i = (1:num_subset),
    I_data(i,:) = y(i:len_subset-2+i);
    T_data(i,:) = y(len_subset-1+i);
end;
fprintf('Training sequence is generated.\n');
```

III Program Source Codes of Private Functions

III.1 Annealing function, currently only linear annealing available

```
function out = annealing(start_data,end_data,num)
% annealing - anneal data from 'start' to 'end' with number 'num'
% out = annealing(start,end,num)
% start_data - starting point
% end_data   - ending point
% num        - number of annealing point
% out        - annealed data sequence

%%%% Author: Yanbo Xue & Le Yang
%%%% ECE, McMaster University
%%%% yxue@grads.mcmaster.ca; yangl7@psychology.mcmaster.ca
%%%% May 12, 2006
%%%% This is a joint work by Yanbo and Le
%%%% For Project of Course of Dr. Haykin: Neural Network

% Check input parameters
if start_data == end_data,
   error ('Starting point and ending point is the same.');
end;
if num <= 2,
   error('Number of annealed data point should > = 2.');
end;

% Linear annealing
step = (end_data - start_data)/(num-1);
out = [start_data:step:end_data];
```

III.2 Hyperbolic function

```
function y = hyperb(x)
% y = hyperb (x)
% hyperbolic function
% x - input data
% y - output data
```

```
%%%% Author: Yanbo Xue & Le Yang
%%%% ECE, McMaster University
%%%% yxue@grads.mcmaster.ca; yangl7@psychology.mcmaster.ca
%%%% May 11, 2006
%%%% This is a joint work by Yanbo and Le
%%%% For Project of Course of Dr. Haykin: Neural Network

y = (exp(2*x)-1)./(exp(2*x)+1);
```

III.3 Differentiation of hyperbolic function

```
function y = d_hyperb(x)
% y = d_hyperb (x)
% differentiation of hyperbolic function
% x - input data
% y - output data

%%%% Author: Yanbo Xue & Le Yang
%%%% ECE, McMaster University
%%%% yxue@grads.mcmaster.ca; yangl7@psychology.mcmaster.ca
%%%% May 11, 2006
%%%% This is a joint work by Yanbo and Le
%%%% For Project of Course of Dr. Haykin: Neural Network

y = (4*exp(2*x))./((1 + exp(2*x)).^2);
```

III.4 Inverse of hyperbolic function

```
function y = inv_hyperb(x)
% y = inv_hyperb (x)
% inverse of hyperbolic function
% x - input data
% y - output data

%%%% Author: Yanbo Xue & Le Yang
%%%% ECE, McMaster University
%%%% yxue@grads.mcmaster.ca; yangl7@psychology.mcmaster.ca
%%%% May 20, 2006
%%%% This is a joint work by Yanbo and Le
%%%% For Project of Course of Dr. Haykin: Neural Network

y = 0.5*log((1+x)./(1-x));
```