Applications of Recurrent Neural Network on Financial Time Series

by

Yufan Wang (CID: 00820668)

Department of Mathematics
Imperial College London
London SW7 2AZ
United Kingdom

Thesis submitted as part of the requirements for the award of the MSc in Mathematics and Finance, Imperial College London, 2016-2017

Declaration

The work contained in this thesis is my own work unless otherwise stated.

Signature and date:

Acknowledgements

I would like to express gratitude to Dr. Daniel Bloch who provides me with this unique opportunity to work on recurrent neural network and its applications on financial time series. Daniel is very keen on the topic and he is patient in explaining insightful concepts to me. It has been a precious experience for me to exchange ideas with him and try to crack the challenge step by step.

I also deeply appreciate Dr. Giuseppe Di Graziano's guidance throughout the summer. As an approachable machine learning lecturer, he arouses my great interest of the subject. His enlightening suggestions and detailed feedbacks always impress and motivate me to work harder. It has been a wonderful time to have meaningful discussions with him.

I am very grateful to all the lecturers since they bring me interesting stories about mathematical finance.

Finally I would like to hug my parents who financially support my whole year study at Imperial college.

1

INTRODUCTION

Financial time series prediction is always appealing due to its potential profits. However, it is also challenging since financial market is a dynamical system which is highly sensitive and noisy. Numerous factors that can affect stock prices include intrinsic value of the company, national economy, prospects of investors, speculators and political reasons.

During the past ten years, various mathematical and computational researches are taken, attempting to get higher accuracy on financial time series prediction. Among them, neural networks, especially recurrent neural networks, stand out as one of the emerging models that are capable of capturing non-linearity of a system and cope with chaotic, non-stationary time series.

This paper introduces two types of recurrent neural networks: Echo State Network(ESN) and Recurrent Radial Basis Function Network. The biggest feature of recurrent neural network is that it has a sparsely connected hidden layer called reservoir, which enables RNNs to have short term 'memory' that captures information about what has been calculated so far. Usually, this recurrent part of the structure is essential in learning complex patterns.

The main purpose of the thesis is to verify performance of RRNs on denoised financial time series. The modelling is under the assumption that historical price series contains all necessary information to be used to predict next day prices and returns.

In the context of test error, the empirical results imply that the proposed recurrent neural networks, especially ESN, successfully manage to reduce great percentages of the test errors made by linear regression under the same condition.

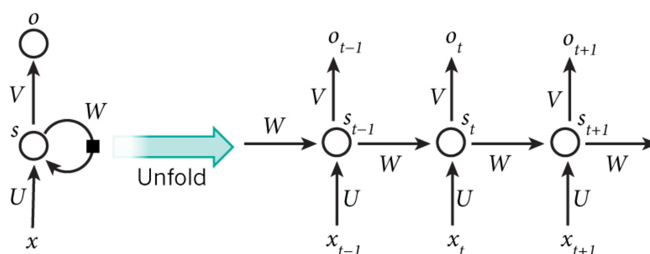


Figure 1: Recurrent Neural Network

2

BASICS

2.1 DATA PREPROCESSING

2.1.1 *Data Normalization*

Data normalization is the process of transforming raw data into some standard form of data. This process is usually important for complex machine learning models. Some models even require this procedure before data are fed into the models, since models internally use distances or feature variances and thus without normalization the results would be heavily affected by the feature with the largest variance or scale. Normalizing inputs could also help numerical optimization method (such as Gradient Descent) converge much faster and accurately.
There are different ways of data normalization:

- Linearly transform all the data to a certain range $[a, b]$, where the minimum of the dataset is mapped to a and the largest value is mapped to b.
- Calculate the mean and standard deviation of the dataset, subtract each sample by sample mean, and then divide each sample by sample standard deviation.

Consider method one: In fact, various normalization methods were tested to improve the network training (Demuth and Beale, 2002; Chaturvedi et al., 1996; Sola and Sevilla, 1997), including "the normalized data in the range of [0, 1] by using the following equation:"

$$\mathbf{x}_{norm} = \frac{\mathbf{x} - \mathbf{x}_{min}}{\mathbf{x}_{max} - \mathbf{x}_{min}} \tag{1}$$

Generally, given a time series vector $\mathbf{x}$, if desired target range is set to be $[l, r]$, then if $m = max(\mathbf{x})$ and $n = min(\mathbf{x})$, solve the equation

$$an + b = l \tag{2}$$

$$am + b = r \tag{3}$$

will give gradient a and intercept b respectively:

$$a = \frac{r-l}{m-n} \tag{4}$$

$$b = r - am = \frac{lm - rn}{m-n} \tag{5}$$

Thus $y = ax + b$ will give the mapped data. Below is an experiment interpreting the effectiveness of this method:
For price series, one usually take its return

$$rt(n+1) = \frac{p(n+1) - p(n)}{p(n)}$$

or log return

$$log_rt = log(1 + rt(n+1)) = log(\frac{p(n+1)}{p(n)})$$

as the variable to be fed into the model, the reasons being that the log return is usually the variable of interest instead of raw price, and to some extent, log returns can be assumed to be nearly stationary while price cannot.
Suppose log returns are imported into a certain machine learning model. Other parameters kept all the same, with and without data normalization gives the following comparison on prediction performance:

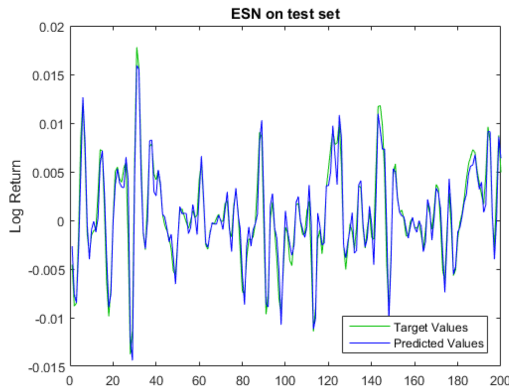


Figure 2: With Normalization

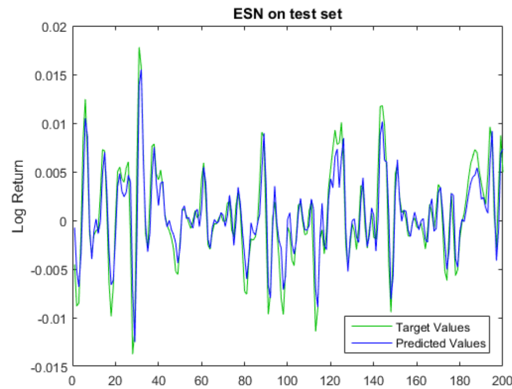


Figure 3: Without Normalization

Comparing two figures, one could visually see that the task performance of the first one surpass that of the latter one, which in a way proves the utility of data normalization.
Now, consider method 2, suppose $\mathbf{x}$ is a dataset, with $\bar{\mathbf{x}}$ being sample mean of the

dataset and s being the sample standard deviation, then the normalized dataset $\mathbf{x}$ has the following expression:

$$\mathbf{x}' = \frac{\mathbf{x} - \bar{\mathbf{x}}}{s} \tag{6}$$

Sometimes population mean μ instead of $\bar{\mathbf{x}}$ is used. For instance, one may assume the population mean of log return is zero. The choice varies from case to case generally.

In summary, data normalization is an important part of the model which affects its prediction performance. The main aim of normalization is to make data more tractable so that when the normalized data are further processed by the model, things become more controllable.

2.1.2 *Data Denoising*

Usually, financial time series is noisy. Thus, it is filled with irrelevant information that may confuse a certain model and reduce its task performance. Therefore, denoising is an integral and significant part of a financial series prediction system.
External information(e.g. news) may or may not be used for denoising. At the end of the prediction system, noise needs to be added back.
Therefore, as a whole, denoising is an very important and interesting area that worth delving. However, it is beyond the scope of this paper. Hence, in the next few sections, the built-in denoising package of Matlab is used. Please find below a comparison between the original noisy time series and the denoised one:

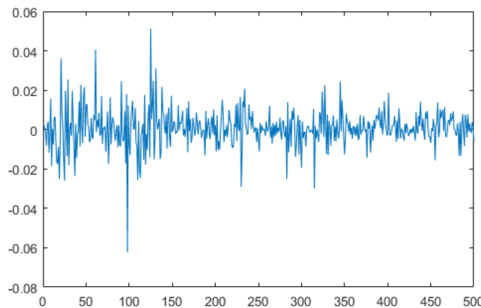


Figure 4: Raw Data

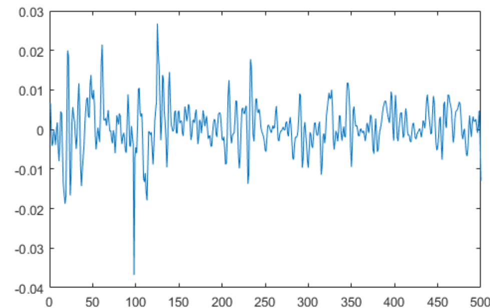


Figure 5: Denoised Data

2.2 NOTATIONS AND COMMON SETTINGS

In this paper there are many notations and common settings that need to be clarified. First of all, there are two datasets used in this paper: one is the classic chaotic time

series-Mackey Series, the other one is daily close price data of 100 stocks from S&P 500. In the next few sections, whenever these datasets are imported, the following numbers are used:

	size of training set	size of test set
Mackey Series	2000	500
Stock Data	1200	200

In this thesis, $\mathbf{u}(n)$ stands for input vector at time n and has size N_u, while $\mathbf{y}(n)$ represents the corresponding output vector and has size N_y.
Generally, matrix $[u(n_0), ..., u(n-2), u(n-1), u(n)]$ is used to predict $y(n) = u(n+1)$. In further sections, uTr stands for input dataset in training set, yTr stands for output dataset in training set. Similarly, uTest and yTest represent input dataset and output dataset in test set respectively.
Below are typical steps that are followed in this paper to apply machine learning models on datasets. Note the largest iteration is over stocks.

For each stock, do the following:

1. - Initialize input weight matrix $\mathbf{W}_{in}$ and reservoir weight matrix $\mathbf{W}$.
   - Assign values to various parameters.

   Note this step could be taken outside the loop and placed before the loop if all stocks use the same common parameter setting.

2. Raw data is transformed to log return (or other quantities).

3. Log return is denoised.

4. Map the denoised log return into a predefined range(e.g.$[-1, 1]$), record the gradient and intercept.

5. Calculate the volatility of the whole series and divide it by square root of the time length of the series to get daily log return volatility of the stock.

6. Split the whole set into training set and test set. For training set, uTr=data(1 : 1200), yTr=data(2 : 1201); For test set, uTest=data(1201 : 1400), yTest=data(1202 : 1401);

7. Adjust both sets according to the value of input size N_u. For example, suppose uTr=$[1, 2, 3, 4, 5]$, yTr=$[2, 3, 4, 5, 6]$, if $N_u = 2$, uTr is changed to

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 5 \end{bmatrix}$$

   and yTr is changed to $[3, 4, 5, 6]$. uTest and yTest are similarly transformed. In particular, if $N_u = 1$, this step does nothing.

8. Apply specific machine leaning model(e.g.Linear Regression, Echo State Network etc.). For each model, do the following:
   a) Train the model using training set, get output weight matrix $\mathbf{W}_{out}$. For recurrent neural network, the first 100 samples are used to develop reservoir states hence will not be used for training(For consistency this is also done with linear regression).
   b) Do predictions on test set.
   c) Linearly map the target data $\mathbf{y}$ and predicted data $\hat{\mathbf{y}}$ back to orignial scale using the recorded gradient and intercept. Then, calculate the corresponding root mean square errors for training and test set.
9. Draw corresponding figures and error tables.

Kindly note that:

- For Mackey series, it is equivalent to the case of one stock. However, the difference is that the Mackey series is neither denoised nor transformed to log return before further processing.
- For stocks, daily close price instead of daily adjusted close price is used since adjust close value is affected by cash dividends and stock dividends, thus it is not the real close value on that day.
- This paper focus on fitting the exact amount of log returns. Therefore, the cost function to be minimized is as follows:

$$E = \frac{1}{n}\sum_{t=1}^{n}(\hat{\mathbf{y}}(t) - \mathbf{y}(t))^2 \tag{7}$$

- Training and test errors are comparable over different normalization ranges since these errors are calculated after data are mapped back to original scale.
- Random initializations of $\mathbf{W}_{in}$ and $\mathbf{W}$ can be made reproducible. For instance, in Matlab, one could use command: rand('seed',42);
- The common parameters that will appear in this paper are:

Input Size: N_u	Reservoir Size: N_x	Target Size: N_y
Input Weight Matrix: $\mathbf{W}_{in}$	Reservoir Weight Matrix: $\mathbf{W}$	Reservoir Density
Leak Rate: γ	Spectral Radius	Regularization term: λ

2.3 LINEAR REGRESSION

Linear regression is one of the simplest regression models which are still widely used in industry. This most straight forward method works as a benchmark or criterion when competed against a more complex model. Typically, the root mean square error is calculated as follows:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(h_\theta(\mathbf{X}^{(i)}) - \mathbf{y}^{(i)})^2} \quad (8)$$

Note $h_\theta(\mathbf{X}^{(i)})$ is linear hypothesis of the form $\theta_0 + \theta_1\mathbf{x}_1 + \theta_2\mathbf{x}_2 + \theta_3\mathbf{x}_3 + ... + \theta_m\mathbf{x}_m$. For consistency with further discussions about recurrent neural network, $\mathbf{x} = \mathbf{X}^{(i)}$ is set to be the *ith* column instead of *ith* row of the design matrix $\mathbf{X} \in \mathbb{R}^{m\times n}$. Then, assuming $\mathbf{y}$ is the vector of targets, in the sense of normal equation, the solution has the form

$$\hat{\boldsymbol{\theta}} = \mathbf{y}\mathbf{X}^T(\mathbf{X}\mathbf{X}^T)^{-1} \quad (9)$$

In order to mitigate potential overfitting, a regularization term λ is introduced so that it becomes *ridge regression* and the solution turns into

$$\hat{\boldsymbol{\theta}} = \mathbf{y}\mathbf{X}^T(\mathbf{X}\mathbf{X}^T + \lambda\mathbf{I})^{-1} \quad (10)$$

where $\mathbf{I}$ is identity matrix. The following is a pseudo code for training linear regression:

Algorithm 1 LinearRegression Training

```
1: function linearTrain(u, y, modelInputs)          ▷ modelInputs ← N_u, N_y etc.
2:     m ← nbPointsToIgnore + 1;
3:     S = [1; u(:, m : end)];
4:     D = y(:, m : end);
5:     W_out = DS^T pinv(SS^T + λI);      ▷ Ridge Regression, 'pinv' is pseudo inverse
6: end function
```

Note that in the above algorithm,

- At line 2, *nbPointsToIgnore* equals to 100 in this paper
- At line 3, the semicolon indicates a vertical concatenation.
- At line 5, superscript T stands for transpose of a matrix.

In the following sections, the task performance of linear regression will be mainly used as a threshold. If a complex model cannot surpass linear regression, then in terms of efficiency no one would use this complex model and abandon the simple linear regression.

3

RECURRENT NEURAL NETWORK

3.1 ECHO STATE NETWORK

Echo State Network(ESN) are a certain type of recurrent neural network(RRN) that is built to solve non-linear tasks. The main difference of it from feedforward neural network is that there is an internal reservoir in ESN represented by a sparse matrix which holds connections between reservoir neurons. The basic idea of ESN is that it projects input vector $\mathbf{u}(n)$ into a high dimensional vector $\mathbf{x}(n)$ in order to capture non-linearity of the system, where n being certain discrete time point. Unlike Principal Component Analysis(PCA) which attempts to reduce dimensions, ESN expands dimensions of inputs to simplify the problem, which is crucial for reservoir computing method[2].
ESN has advantages compared to traditional feed-forward neural networks mainly due to its feature of short-term memory, which retains and reflects recent history pretty well and gradually fades as time goes on. This property shares similarity with stock market dynamics, which renders it possible, even suitable for ESN to predict financial time series.
The general update equations for ESN are :

$$\widetilde{\mathbf{x}}(n) = f(\mathbf{W}_{in}[1;\mathbf{u}(n)] + \mathbf{W}\mathbf{x}(n-1)) \tag{11}$$

$$\mathbf{x}(n) = (1-\gamma)\mathbf{x}(n-1) + \gamma\widetilde{\mathbf{x}}(n) \tag{12}$$

$$\hat{\mathbf{y}}(n) = \mathbf{W}_{out}[1;\mathbf{u}(n);\mathbf{x}(n)] \tag{13}$$

where $\mathbf{u}(n) \in \mathbb{R}^{N_u}$ and $\mathbf{x}(n) \in \mathbb{R}^{N_x}$ are vectors of input and reservoir neuron activations respectively, $\mathbf{y}(n) \in \mathbb{R}^{N_y}$ is the output vector at time n. $\mathbf{W}_{in}$ is the input weight matrix, $\mathbf{W}$ stands for the reservoir weight matrix; At time step n, f can be either *tanh* or *sigmoid* and is applied element-wise; $\gamma \in (0,1]$ is called leak rate. If $\gamma = 1$, then it is standard ESN. Note that sigmoid function has the following expression:

$$f(x) = \frac{1}{1+e^{-x}}$$

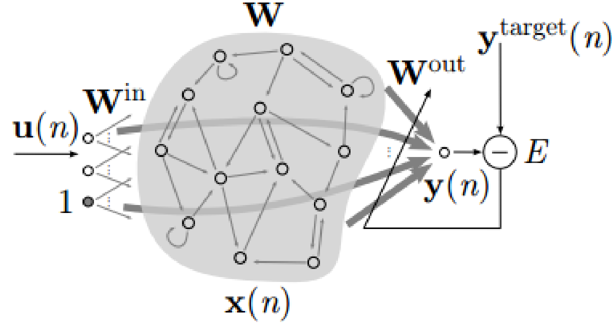


Figure 6: Echo State Network Architecture(Lukosevicius, 2012, S. 3)

Basic steps to construct an ESN are as follows:

- $\mathbf{W}_{in}$ and $\mathbf{W}$ are randomly initialized, an appropriate γ is selected.
- For each n,
  - Import input vector $\mathbf{u}(n)$ of training set to the system and evolve the corresponding reservoir state $\mathbf{x}(n)$. Note that applying function f is a way of data normalization since it keeps $\mathbf{x}(n)$ bounded so that it does not have unexpected 'weird' behaviour.
- A supervised learning algorithm is built on pairs of $\mathbf{x}(n)$ and $\mathbf{y}^{target}(n)$. Typically, for off-line optimization, ridge regression is used to yield matrix $\mathbf{W}_{out}$.
- Foe each n,
  - Apply trained $\mathbf{W}_{out}$ on input vector $\mathbf{u}(n)$ of test set to compute predictions $\mathbf{y}(n)$.

Here $\mathbf{W}_{out}$ is computed using ridge regression:

$$\mathbf{W}_{out} = \mathbf{y}\mathbf{X}^T(\mathbf{X}\mathbf{X}^T + \lambda\mathbf{I})^{-1}; \tag{14}$$

where $\mathbf{X}$ is the design matrix collecting evolutions of $\mathbf{x}(n)$:

$$\mathbf{X} = \begin{bmatrix} | & | & & | \\ x(1) & x(2) & \dots & x(n) \\ | & | & & | \end{bmatrix}$$

Initial state $\mathbf{x}(0)$ is arbitrary. Typically, $\mathbf{x}(0) = \mathbf{0}$[9]. Usually initial 'warm-up' states of $\mathbf{x}(n)$ are discarded since they are affected by artificial initial setting and should not be used in training $\mathbf{W}_{out}$.
Moreover, it is useful to monitor elements of $\mathbf{W}_{out}$: large entries means that a tiny

change in $\mathbf{x}(n)$ maybe be unnecessarily amplified. Thus, even if task performance is pretty well at the training stage, slight deviation on input conditions may lead to poor prediction performance on test set. In other words, being unstable, the system is very sensitive to inputs and is much likely to be under risk of overfitting. That is why regularization term λ is needed.

3.1.1 *Pseudo Code*

The following is the pseudo code for training ESN:

Algorithm 2 ESN Training Algorithm

1: **function** $esnTrain(\mathbf{u}, \mathbf{y}, modelInputs)$ ▷ $modelInputs \leftarrow \gamma, \mathbf{W}_{in}, \mathbf{W}, \mathbf{X}_0^{trn}$ *etc.*
2: $\mathbf{X} \in \mathbb{R}^{N_x \times nbDataPoints} \leftarrow$ *Design Matrix*
3: **for** $i = 1$ to $nbDataPoints$ **do**
4: $\mathbf{X}(:, i) = (1 - \gamma) * \mathbf{X}(:, i-1) + \gamma * tanh(\mathbf{W}\mathbf{X}(:, i-1) + \mathbf{W}_{in}[1; \mathbf{u}(:, i)]);$
5: $\mathbf{z}(:, i) = [1; \mathbf{u}(:, i); \mathbf{X}(:, i)];$ ▷ *Vertical Concatenation*
6: **end for**
7: $m \leftarrow nbPointsToIgnore + 1;$
8: $S = \mathbf{z}(:, m : end);$
9: $D = \mathbf{y}(:, m : end);$
10: $\mathbf{W}_{out} = DS^T pinv(SS^T + \lambda I);$ ▷ *Ridge Regression, 'pinv' is pseudo inverse*
11: **end function**

The following is the pseudo code for testing ESN:

Algorithm 3 ESN Predicting Algorithm

1: **function** $esnPredict(\mathbf{uTest}, \mathbf{yTest}, modelOutput)$ ▷ $modelOutput \leftarrow \mathbf{W}_{out}, \mathbf{X}_0^{prd}$ *etc.*
2: $\mathbf{X} \in \mathbb{R}^{N_x \times nbDataPoints} \leftarrow$ *Design Matrix*
3: $\mathbf{X}(:, 1) = (1 - \gamma) * \mathbf{X}_0^{prd} + \gamma * tanh(\mathbf{W}\mathbf{X}_0^{prd} + \mathbf{W}_{in}[1; \mathbf{uTest}(:, 1)]);$
4: $\mathbf{z}(:, 1) = [1; \mathbf{uTest}(:, 1); \mathbf{X}_0^{prd}];$ ▷ *Vertical Concatenation*
5: **for** $i = 2$ to $nbDataPoints$ **do**
6: $\mathbf{X}(:, i) = (1 - \gamma) * \mathbf{X}(:, i-1) + \gamma * tanh(\mathbf{W}\mathbf{X}(:, i-1) + \mathbf{W}_{in}[1; \mathbf{uTest}(:, i)]);$
7: $\mathbf{z}(:, i) = [1; \mathbf{uTest}(:, i); \mathbf{X}(:, i)];$ ▷ *Vertical Concatenation*
8: **end for**
9: $\mathbf{yPredict} = (modelOutput.\mathbf{W}_{out}) * \mathbf{z}$
10: **end function**

Please note that in the predicting algorithm:

- No reservoir states $\mathbf{x}(n)$ are discarded since the first reservoir state $\mathbf{X}_0^{prd}$ is equal to the final reservoir states in the training algorithm.
- Sometimes the term $\mathbf{uTest}(:,i)$ can be replaced by $\mathbf{yPredict}(:,i-1)$, since $\mathbf{yPredict}(:,i-1)$ is expected to be equal to $\mathbf{uTest}(:,i)$. In this case, only the very first value of dataset **uTest** is used and predictions for several days ahead can be made.

3.1.2 *Parameters*

Choosing the right parameters can be more beneficial than choosing the right model itself. However, it is never trivial to tune parameters of ESN. Experience will play an important role here. It is inevitable and necessary to use manual selection to get a sense of which parameters affect results more than others do, and these manually selected values could be used as the initial values for further automated selection. During the process, please ensure that *only* one parameter is changed at a time. The quality of a certain parameter setting is evaluated using root mean square error of training set and test set. Note that all the parameters ought to be optimized under a condition where adequate regularization measure is taken to alleviate overfitting.
Practically, the input scaling, leak rate and spectral radius of reservoir weight matrix are the three most important parameters of ESN.

Input Scaling

Input weight matrix $\mathbf{W}_{in}$ is usually dense without zero elements. The entries of the matrix usually follow a uniform distribution in a symmetric range $[-a, a]$ or classic range $[0, 1]$. Often candidate values for a is 1, 0.1, 0.01. The distribution can be Gaussian or a customized one as well, whereas the boundness is not guaranteed.
The first column of the input weight matrix is usually the bias column, which corresponds to bias element $'1'$ in input vector $\mathbf{u}(n)$. Hence it does not share the same status as the rest of the columns. Thus, to reduce degree of freedom, all columns except the first column of the input weight matrix are randomized using one parameter a, and the first column uses another parameter b.
Small values of a will map input to a small range around zero, where $tanh()$ is virtually linear, hence this setting is suitable for solving linear systems. On the other hand, large values of a will map $\mathbf{u}(n)$ to extreme values of $tanh()$, namely -1 or 1, creating a non-linear, binary manner. Overall, input scaling controls non linearity of $\mathbf{x}(n)$.

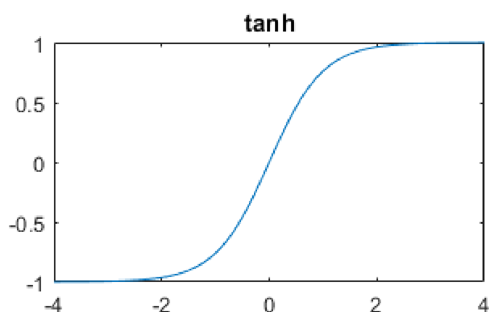


Figure 7: activation function tanh

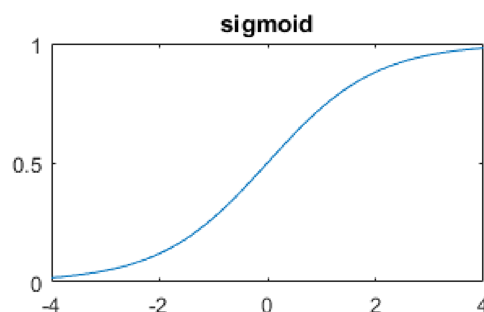


Figure 8: activation function sigmoid

The reservoir weight matrix **W** represents a structure of recurrently connected units. Similar to the case of $\mathbf{W}_{in}$, non-zero elements of **W** usually have uniform distribution in a symmetric range. However, unlike $\mathbf{W}_{in}$, **W** is sparse and has other distinct properties.

Size of the Reservoir

Intuitively, larger the reservoir, the better the attainable performance. the size can be as big as 20, 50, 100, 1000, even 10000 is not uncommon. There is few condition when reservoir may be oversized: one is that the task itself is pretty much trivial, and the other could be that the size approaches meaningful fraction of number of data points, or in other words, lack of data points. Normally, if sample size is n, it is required that [3]

$$N_x <= n/2; \tag{15}$$

Typically, it also holds that $1 + N_u + N_x << n$, this constraint also reduces the risk of overfitting. At the same time, the size of reservoir should at least be the number of independent time steps that are supposed to be memorized to address the problem.

Sparsity

Though the reservoir is big, the reservoir weight matrix is always set to be *sparse* in many publications. In my experience, higher density tends to give higher prediction performance. However, at the same time, lower sparsity means more recurrent connections, and large non-zero matrix operations increase computational costs. Generally, above a certain threshold, connectivity barely affects the final performance. Hence I would recommend 30% connectivity, which is robust enough to be a balance between efficiency and efficacy. Compared to others, sparsity has relatively low priority to be optimized. In Matlab, there is a special representation for sparse matrix that could speed up computations.

Spectral Radius

Being the central parameter of ESN, spectral radius is the maximum of absolute eigenvalues of reservoir matrix **W**. Intuitively, it determines the width of the distribution of non-zero entries. If it is set too high, then the reservoir may hold periodic, even chaotic spontaneous attractor modes which deactivates echo state property. It has been proved that, in most cases, $r(\mathbf{W}) < 1$ ensures the echo state property[7]. However, it is also true that **W** with spectral radius greater than 1 retains echo state property. As a rule of thumb, if the task requires extensive input history, then it is recommended to set the spectral radius higher. The following steps are usually taken when specific spectral radius is assigned to reservoir weight matrix:

1. Initialize a reservoir weight matrix $\mathbf{W}_0$
2. Normalize $\mathbf{W}_0$ to $\mathbf{W}_1$ using $\mathbf{W}_1 = \mathbf{W}_0/r_1$, where r_1 is the spectral radius of $\mathbf{W}_0$. Now that $\mathbf{W}_1$ has unitary spectral radius.
3. Multiply $\mathbf{W}_1$ by r element-wise so that it has spectral radius r.

Leak Rate

Leak rate is a real value between 0 and 1. Echo state network without leak rate γ is called standard echo state network, the one with leak rate or leaky integrator is called LI-ESN. In this latter case, γ can either be placed before application of $f()$ or after. Introduction of leak rate makes it possible to learn very slow dynamic systems and replay them at various speeds(Jaeger, Lukosevicius, Popovici, and Siewert,2010,S. 335).
Small values of leak rate may induce slow dynamics of $\mathbf{x}(n)$ and extend short term memory[4]. Virtually, the prediction performance is sensitive to leak rate since a small change in leak rate may lead to a big change in final result. Hence, this parameter also has relatively high priority. Some low-prioritized parameters could be assigned default values for convenience.

3.1.3 *Empirical Results*

ESN is expert in fitting and predicting chaotic time series, this could be verified by its performance on classic Mackey-Glass series:

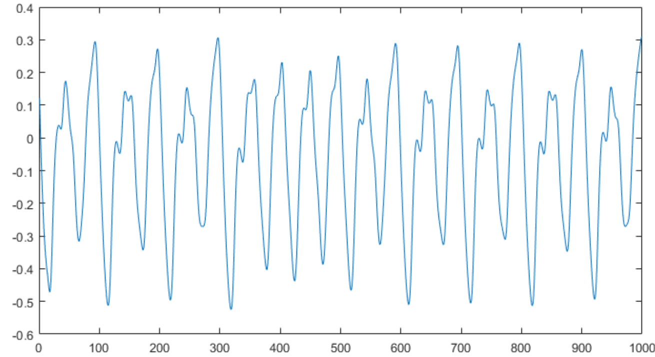


Figure 9: Samples of Mackey-Glass Chaotic Time Series

In a publication by Jaeger, it is claimed that advanced ESN could yield root mean square error(RMSE) of order 10^{-7} on predicting Mackey series[5].
Now, in my experiment, 2000 data points are used for training, with the first one hundred reservoir states $\mathbf{x}(n)$ being discarded due to initial transient effect, and 500 data points are used for testing. Both linear regression and ESN will be implemented. For ESN, parameters are specified as follows:

Leak Rate γ	0.9	Spectral Radius	1.25
Input Weight Matrix $\mathbf{W}_{in}$	$[-0.5, 0.5]$	Reservoir Weight Matrix $\mathbf{W}$	$[-0.5, 0.5]$
Reservoir Size $\mathbf{N}_x$	400	Normalization Range	$[-1, 1]$
Reservoir Density	0.3	Regularization term λ	10^{-8}

After plugging all the parameters into ESN, following errors are summarized:

	Training error(RMS)	Testing error(RMS)
Linear Regression	0.0315	0.032
ESN	7.9×10^{-6}	8.0×10^{-6}

Apparently ESN produces negligible errors compared to that of linear regression. In fact, even if reservoir size N_x is reduced to 35, training and testing errors are still less than 0.001, which is less than 3% of the errors made by linear regression.
For illustration, the predicting performance of ESN with $N_x = 35$, along with that of linear regression, are displayed below:

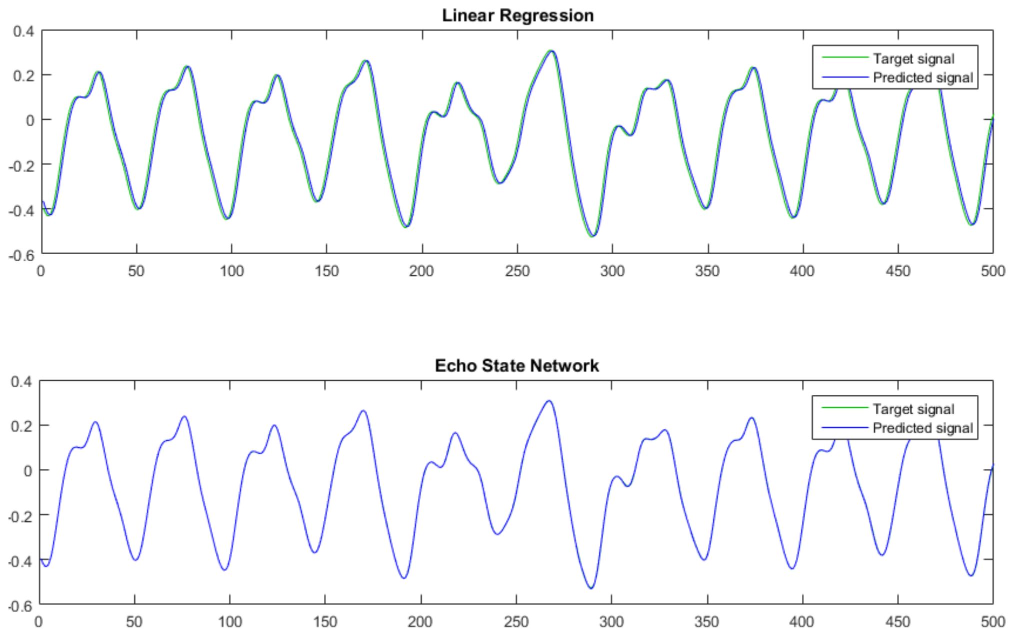
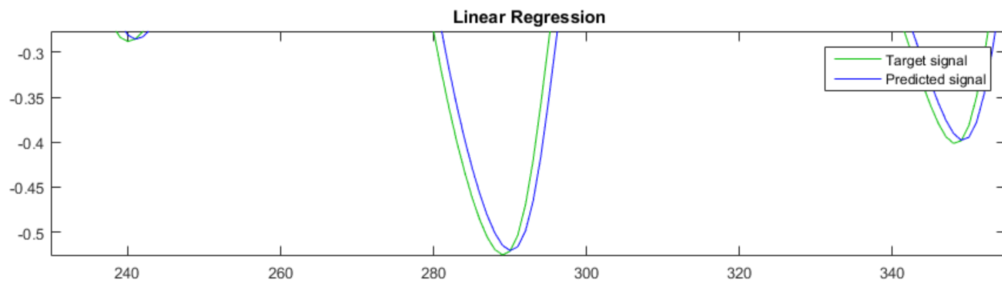


Figure 10: Mackey Series test set

In both figures, one could not virtually discern between predicted values and actual values, since two curves nearly coincide with each other most of the time. Judging from the plot, it seems that linear regression has reasonably well fit even if it has 'huge' test error compared to ESN. However, if details of two figures are studied more carefully, difference could be realized:



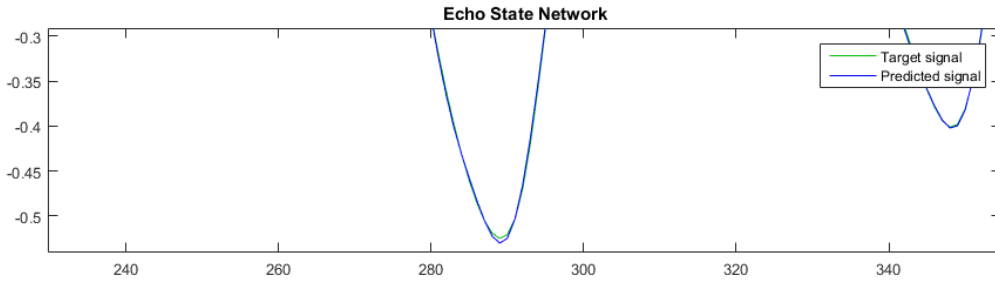


Figure 11: Zoomed-in Figure Comparison

One may notice that prediction curve of linear regression is just a forward shift of the target curves. In other words, the prediction curve is just a *time-lagged* copy of the actual target curve if horizontal axis is discrete time. Therefore, in this case, the prediction capability of linear regression is relatively poor. On the other hand, ESN prediction curve tends to move *in phase* with the target curve. Even at the turning point around $n = 290$, where two curves can visually be distinguished, the trends of prediction values keep pace with that of target curves. Hence, statistically, ESN has much better test error than linear regression. Below are some samples of reservoir states $\mathbf{x}(n)$:

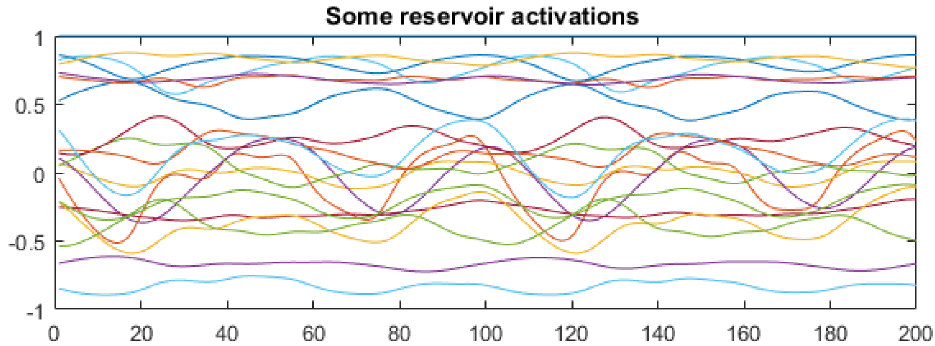


Figure 12: Evolutions of reservoir activations

In the figure above, reservoir activations actually means a vertical concatenation of bias factor, input vector and actual reservoir state, i.e. $(1, \mathbf{u}(n), \mathbf{x}(n))^T$. It is a vector of size $1 + N_u + N_x$, since N_x is large, the first 20 entries of the vector are plotted. Hence in the graph there are totally 20 curves, with each curve being the evolutions of a fixed position within the vector $(1, \mathbf{u}(n), \mathbf{x}(n))^T$. At the same time, x-axis indicates $n = 1$ to 200, and the number of evolutions is 200. In the figure, the reservoir state is not saturated since not all curves converges to the same number(e.g. 1 or -1). Note also that the upper most curve, the bias factor, is a straight line which always equals to one.
The following is a bar plot of output weight matrix $\mathbf{W}_{out}$:

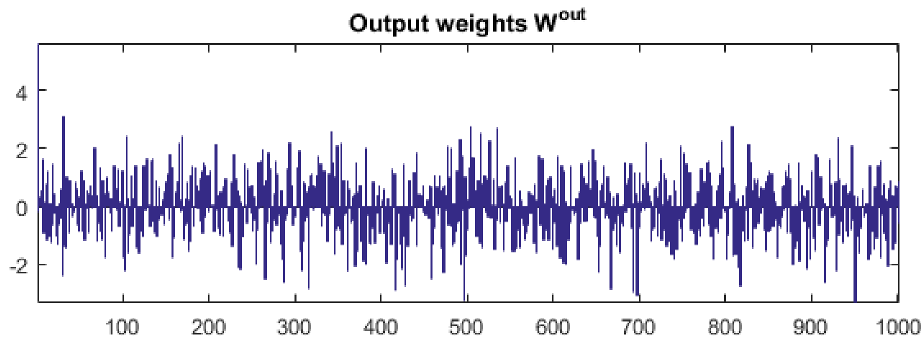


Figure 13: Evolutions of reservoir states

Assuming $N_y = 1$, $\mathbf{W}_{out}$ is a vector of size $N_x \times 1$. In the figure above, $N_x = 1000$, and almost all the elements of $\mathbf{W}_{out}$ are within the range $[-3, 3]$. Hence, there are no extreme large values in $\mathbf{W}_{out}$ and the model is less likely to suffer from overfitting.

Recall that the word 'chaotic' means that a time series consists of irregular patterns, so that visually no obvious trends could be easily perceived, while 'noisy' emphasizes on stochastic or irrelevant factors that may interfere or even obscure useful inputs. A time series could be both chaotic and smooth at the same time. e.g. Mackey Series, while noisy time series is usually filled with *wavelets*.
After adding some Gaussian noise to the standard Mackey series, the following figure is plotted:

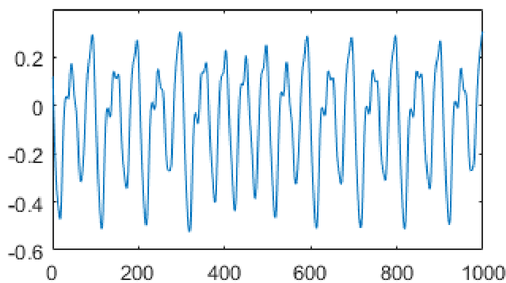


Figure 14: Mackey series

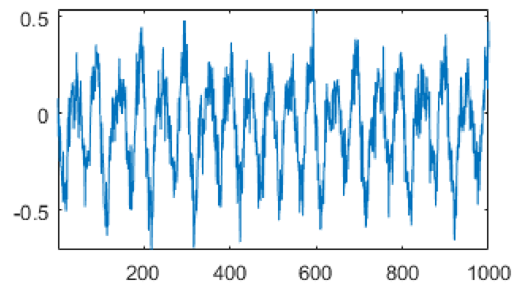


Figure 15: noisy Mackey series

As in figure 15, the noisy curves are usually oscillating more frequently and therefore are much more difficult for any models to track its path. As a matter of fact, it could be seen that ESN possesses advantages in fitting and predicting chaotic time series. However, experiments show that the performance of ESN will be discounted coping with noisy time series directly.
Unfortunately, financial time series is often chaotic and noisy. In addition, great caution is required when financial data are utilized for backtesting. There is no doubt that

flawed data can lead to misleading results. In worst cases, historical returns may be overestimated. Following are several facts about financial data:

- Dataset with high resolution offers more details, at the expense of being noisier (Aamodt, 2015, S. 50)
- Stock prices need to be adjusted after share split or dividend pay-out.
- Most importantly, many accessible datasets have survisorship bias[1] since historical database does not hold information about companies which disappeared due to delistings, bankruptcies or mergers.

Along with the first two bullet points, the size of dataset also matters in a machine learning model, while the third bullet point, as well as transaction cost, is crucial in terms of backtesting.
Due to limited public access to all types of financial datasets, the focus is on modelling stock returns. Hence, regardless of survivorship bias, only survived stocks, especially those from S&P 500, are chosen for experiments. Generally, these stocks have advantages over other stocks in terms of market cap and liquidity. The index S&P 500 itself is often used as the weather report of financial industry.
In my experiment, 100 constituents of S&P 500 are imported. For each stock, the daily closing price data are downloaded. 1200 data points are used for training(the first 100 points are used to evolve reservoir states $\mathbf{x}(n)$ and will be discarded), while another 200 points are left out for testing.
Applying linear regression to the raw log returns of symbol MMM gives the following figures:

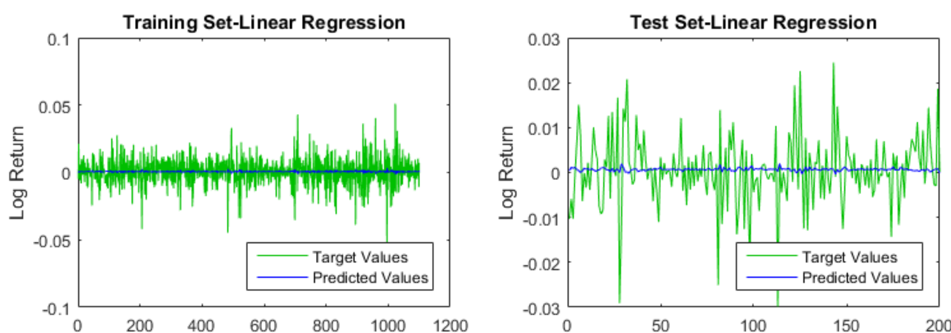


Figure 16: Linear Regression on raw stock returns

The figure shows that the log return time series is so intractable and noisy such that only approximately a straight line around zero would minimize the least square error for linear regression, where the regularization term λ is the only parameter of ridge regression. However, regardless of what λ is, it always gives similar results. Here, the ratio of predicted values volatility to target values volatility would be close to zero,

which is undesirable.
Now, the same raw data are fed into ESN:

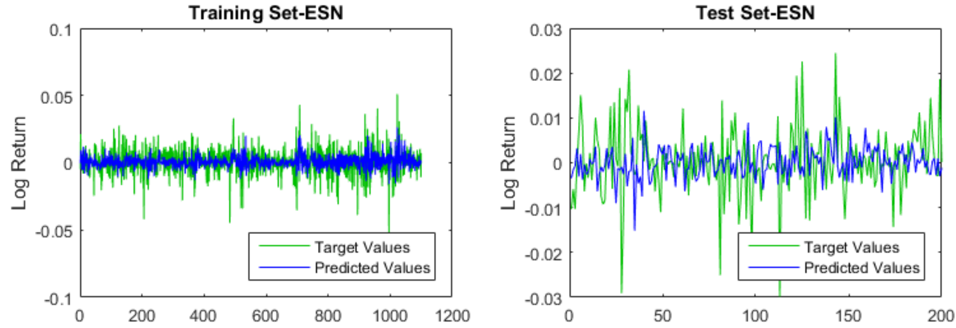


Figure 17: ESN on raw stock returns

Statistically, ESN has test error of only 5% lower than that of linear regression, despite the fact that visually their figures differ a lot. Moreover, in dealing with unprocessed data, adjustment of parameter values does not improve performance of ESN a lot.

Now, the log return series of MMM is denoised before fed into linear regression and ESN. With regularization term $\lambda = 10^{-8}$, linear regression gives the following error:

	Training error(RMS)	Test error(RMS)
Linear Regression	0.00472	0.0036

At the same time, ESN is equipped with common parameter settings below,

Leak Rate γ	1	Spectral Radius	0.8
Input Weight Matrix $\mathbf{W}_{in}$	$[-2, 2]$	Reservoir Weight Matrix $\mathbf{W}$	$[-0.5, 0.5]$
Reservoir Density	0.5	Regularization Term λ	10^{-8}

In Matlab, the same random initializations for $\mathbf{W}_{in}$ and $\mathbf{W}$ are reproducible by setting values for seed. For example, rand('seed', 34);
Now, fix these parameters, and only vary reservoir size N_x since it can sensitively affect performance of ESN. The results are displayed in triplets, which stands for N_x, training error and test error respectively.

ESN	f=tanh	f=sigmoid
No normalization	$850, 0.00227, 0.00157$	$200, 0.00235, 0.00204$
Normalized to $[-1, 1]$	$170, 0.00209, 0.00150$	$600, 0.00228, 0.00140$
Normalized to $[-2, 2]$	$140, 0.00234, 0.00166$	$80, 0.00232, 0.00139$
Normalized to $[0, 1]$	$190, 0.00193, 0.00136$	$650, 0.00250, 0.00155$

From the table, it could be seen that:

- Normalization generally improves task performance of ESN(note normalization does not virtually change performance of linear regression).
- Compared to tanh, sigmoid function works better in symmetric ranges. Especially with the range $[-2, 2]$, ESN with sigmoid activation function uses small reservoir size $N_x = 80$ and still perform better than that of tanh with a larger reservoir size $N_x = 140$.
- The best test error comes from ESN with normalization range $[0, 1]$, it reduces 62% of the test error made by linear regression. Moreover, it is also coupled with best training error in the table.

Cautiously note that these are observations from the test on stock MMM. Further experiments on large set of stocks are required to check if the above observations universally apply.
The corresponding figures for ESN with normalization range $[0, 1]$ and linear regression are displayed below:

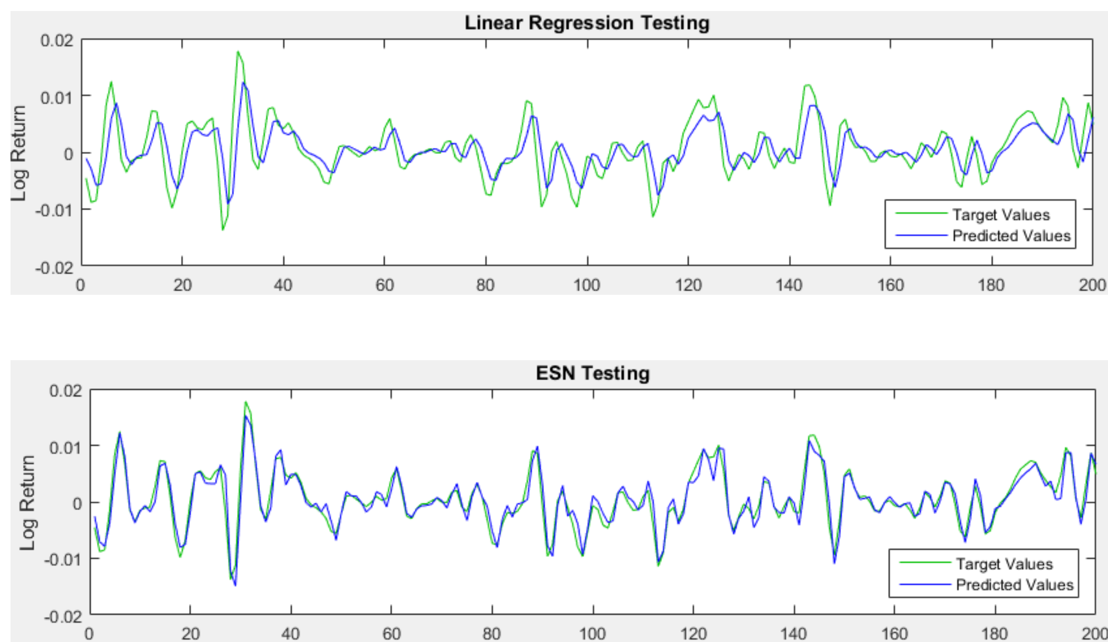


Figure 18: ESN on denoised stock returns test set

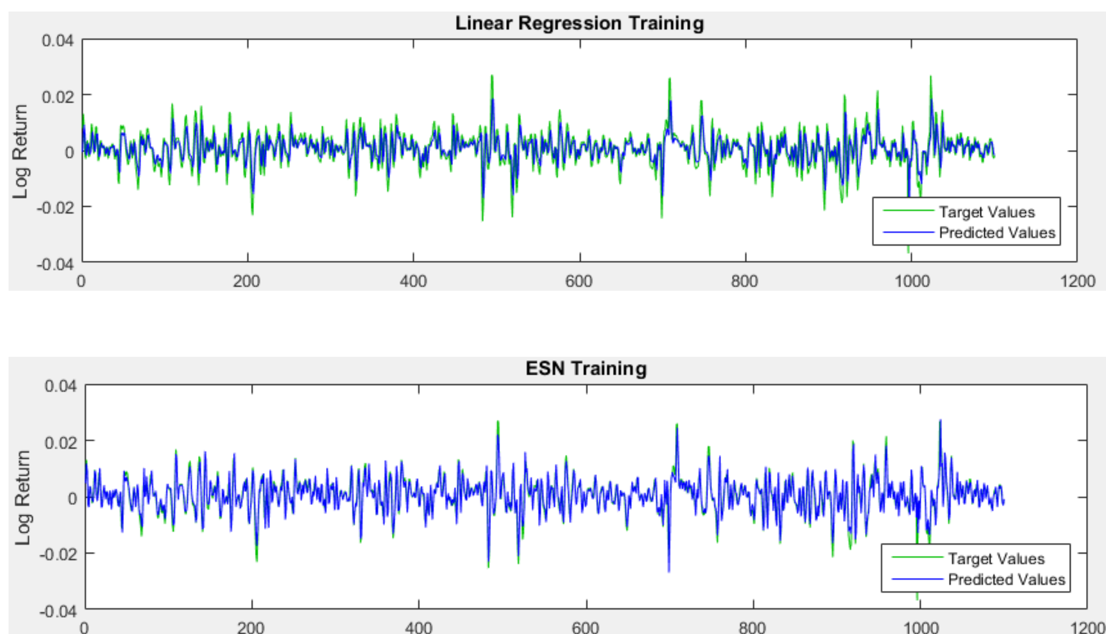


Figure 19: ESN on denoised stock returns training set

From the figures it could be seen that ESN restores some of its prediction capability when dealing with denoised chaotic time series instead of raw time series. ESN apparently has better fittings in comparison of linear regression. Statistically, with normalization, ESN reduces at least 57% of the test error made by linear regression.
Moreover, note that parameters are manually tuned one at a time. Therefore, if systematic way of tuning(i.e. grid search, genetic algorithm) is used, the errors are likely to be further reduced by a meaningful percentage. Moreover, optimal parameters settings are specific and unique for each stock. If the same setting is used for various stocks, the corresponding errors would be inevitably increased. Nevertheless, an experiment is done with the following parameter setting:

Leak Rate γ	1	Spectral Radius	0.8
Input Weight Matrix $\mathbf{W}_{in}$	$[-2, 2]$	Reservoir Weight Matrix $\mathbf{W}$	$[-0.5, 0.5]$
Reservoir Density	0.5	Reservoir Size N_x	350

Meanwhile, a relative large regularization term λ=0.01 is chosen to reduce chance of overfitting. Also, sigmoid function with normalization range $[-2, 2]$ is used. ESN with this parameter setting is applied on 100 stocks from S&P 500 and then gives the error table as follows:

	Average Training error(RMS)	Average Test error(RMS)
Linear Regression	0.00809	0.00799
ESN	0.00498	0.00515

The table shows that this time ESN reduces only 36% instead of 57% of the test error made by linear regression. This is due to the use of common parameter setting for all 100 stocks for convenience. Nevertheless, 36% reduction still reveals competitive prediction capability of ESN across stocks with a common parameter setting.

3.1.4 *Batch Intrinsic Plasticity*

So far there are no generalized unsupervised learning algorithms that are powerful enough to be able to fully train ESN reservoirs, although several attempts[8] were made. However, sometimes pre-training methods in simple machine learning models could be enlightening and useful as well in complex models such as ESN.

Extreme Learning Machine(ELM) is a simple type of feedforward neural network. It has only one hidden layer and no recurrent part. Similar to ESN, its task performance greatly depends on random initialization of the input weight matrix $\mathbf{W}_{in}$. Without additional tuning strategies, in some cases random generations can lead to saturations of reservoir states $\mathbf{x}(n)$, where most of the entries are equal to -1 or 1, or early convergence where $\mathbf{x}(n) = \mathbf{x}(n+1)$ when n is small.
Batch Intrinsic Plasticity[10] is introduced to adapt activation functions so that desired distribution for $\mathbf{x}(n)$ can be realized.
Suppose ELM has the following update equations:

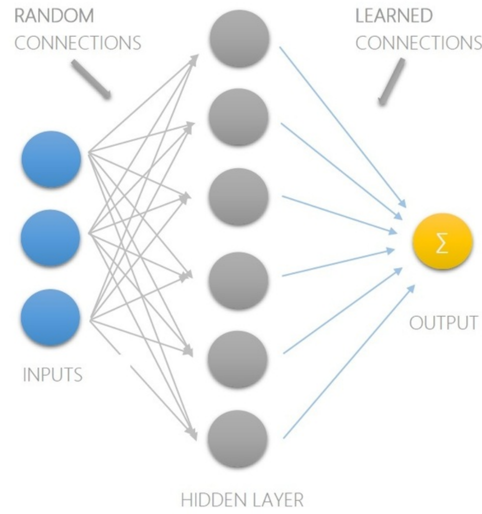


Figure 20: Extreme Learning Machines

$$\mathbf{x}_i = f(a_i \mathbf{W}^i_{in} \mathbf{u} + b_i) \tag{16}$$

$$\hat{\mathbf{y}} = \mathbf{W}_{out} \mathbf{x} \tag{17}$$

where $\mathbf{u} = (u(1), u(2), ..., u(nbDataPoints)) \in \mathbb{R}^{N_u \times nbDataPoints}$ is input vector, $\mathbf{W}^i_{in}$ is the *ith* row of input weight matrix $\mathbf{W}_{in} \in \mathbb{R}^{N_x \times N_u}$, and $\mathbf{a} \in \mathbb{R}^{1 \times N_x}$ and $\mathbf{b} \in \mathbb{R}^{1 \times N_x}$ are scalar vectors to be determined by BIP.
Consider $f = tanh$, then $x = [-1, 1]$ is an important range where $y = tanh(x)$ is virtually linear. Outside this range, especially when $|x| > 2$, y will be very close to -1 or 1, which can cause saturations of $\mathbf{x}(n)$ if $\mathbf{x}_i(n) = f(\mathbf{W}^i_{in} \mathbf{u}(n))$. Now with scaler vectors $\mathbf{a}$ and $\mathbf{b}$, this situation may be mitigated.

Suppose $f = tanh$, $\mathbf{s}_0 \in \mathbb{R}^{N_x \times nbDataPoints} = \mathbf{W}_{in}\mathbf{u}$ and $\mathbf{x} \in \mathbb{R}^{1 \times N_x}$ has N_x positions for elements. Then, for *ith* position, BIP will do the following:

1. Uniform samples are drawn from $[-1, 1]$ and sorted ascendingly, the result is collected by target vector $\mathbf{t} \in \mathbb{R}^{1 \times nbDataPoints}$.

2. The *ith* row of $\mathbf{s}_0$ is sorted ascendingly and collected by vector $\mathbf{s}$.

3. Take $\mathbf{s}$ as input and $\mathbf{t}$ as output, implement a linear regression to get coefficients a_i and b_i.

Finally, iterations of i from 1 to N_x will give vectors $\mathbf{a}$ and $\mathbf{b}$.
The corresponding pseudo code can be found below:

Algorithm 4 BIP Training Algorithm

Require: : Import input vectors $\mathbf{u} = (u(1), u(2), ..., u(nbDataPoints))$
1: $\mathbf{s}_0 = \mathbf{W}_{in}\mathbf{u}$
2: **for** $i = 1$ to N_x **do**
3: $\mathbf{s} \leftarrow$ sorted *ith* row of $\mathbf{s}_0$
4: Construct concatenation $\phi = [\mathbf{s}; (1, 1, ..., 1)] \in \mathbb{R}^{2 \times nbDataPoints}$
5: Draw targets $\mathbf{t} = (t(1), t(2), ..., t(nbDataPoints))$ from desired distribution
6: $\mathbf{t} \leftarrow$ sorted $\mathbf{t}$
7: $v_i = [a_i, b_i] = \mathbf{t} * pinv(\phi)$
8: **end for**
9: **Return v**

Some bullet points are given here:

- For f=sigmoid function, the range for uniform sampling could extend to $[-2, 2]$ due to the shape of sigmoid function.

- The input and output vectors are sorted before linear regression due to the fact that the function $\mathbf{y} = a_i\mathbf{x} + b_i$ is set to be monotonic increasing.

- Apart from uniform distributions, the above procedure also applies to other bounded distributions. For unbounded distributions such as Gaussian, things need to be changed a little bit. Drawing normally distributed samples $\mathbf{t}$ and applying *tanh* on it clearly no longer works since the samples are not bounded between $[-1, 1]$. At this stage, one possibility to get round would be that the normal distribution is applied on $f(\mathbf{t})$, i.e. the reservoir state $\mathbf{x}$, instead of $\mathbf{t}$. Hence, when $\mathbf{x}$ is set to be normally distributed, f^{-1} needs to be computed. The result works as new output, say, $\mathbf{t}_2$, and linear regression can then be constructed between $\mathbf{s}$ and $\mathbf{t}_2$.

Now, this algorithm could also be used on ESN as well. Consider standard ESN with leak rate $\gamma = 1$, and the following update equation:

$$\mathbf{x}(n) = f(\mathbf{W}_{in}[1; \mathbf{u}(n)] + \mathbf{W}\mathbf{x}(n-1)) \tag{18}$$

This time, suppose design matrix $\mathbf{X} = (x(1), x(2), ..., x(nbDataPoints)) \in \mathbb{R}^{N_x \times nbDataPoints}$, there are three ways of setting $\mathbf{s}_0$.

- $\mathbf{s}_0 = f^{-1}(\mathbf{X})$
- $\mathbf{s}_0 = \mathbf{X}$
- $\mathbf{s}_0 = \mathbf{W}_{in}\mathbf{u}$

Experimentally these three settings can all improve ESN. In particular, the third one will be illustrated in detail since it generally performs better than the first two. Partly, it is because the last setting has smallest root mean square error in desired target fitting.

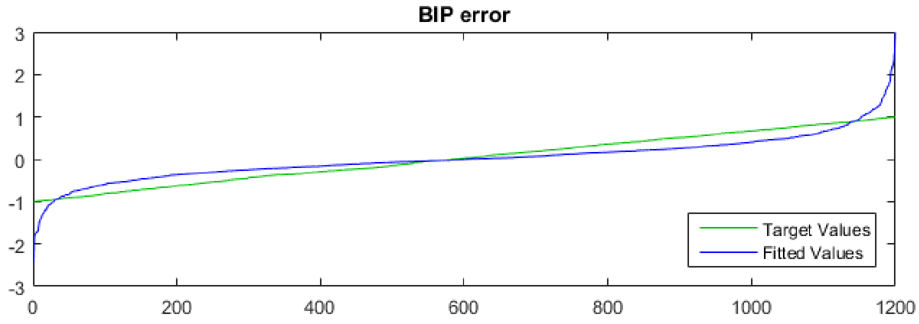


Figure 21: desired target fitting

Now, suppose $\mathbf{s}_0 = \mathbf{W}_{in}\mathbf{u}$, other settings kept the same as ELM, the ESN reservoir state has the following update equation:

$$\mathbf{x}(n) = f(\mathbf{a}. * \mathbf{W}_{in}\mathbf{u}(n) + \mathbf{b} + \mathbf{W}\mathbf{x}(n-1)) \tag{19}$$

Note $\mathbf{a} \in \mathbb{R}^{N_x \times 1}$ and $\mathbf{W}_{in}\mathbf{u}(n) \in \mathbb{R}^{N_x \times 1}$, so $.*$ here means element wise multiplication. For example,

$$\begin{pmatrix} a \\ b \end{pmatrix} .* \begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} ac \\ bd \end{pmatrix}$$

Empirically, unlike case with original ESN, normalization barely changes task performance of BIP-attached ESN, while desired target range matters more. Hence, experiments are done on the stock MMM using different target ranges:

	f=tanh	f=sigmoid
Uniform Range $[-2, 2]$	$90, 0.00247, 0.00148$	$170, 0.00198, 0.00130$
Uniform Range $[-1, 1]$	$120, 0.00234, 0.00147$	$150, 0.00199, 0.00109$
Uniform Range $[-0.5, 0.5]$	$130, 0.00213, 0.00145$	$440, 0.00206, 0.00122$
Uniform Range $[0, 1]$	$110, 0.00207, 0.00134$	$450, 0.00206, 0.00125$

Similar to the previous experiment on ESN, the triplet in the table still means N_x, training error and test error respectively. Seen from the table,

- Sigmoid in general surpasses tanh on task performance.
- For f=sigmoid, that the range $[-1, 1]$ works better than $[-2, 2]$ may be due to specific setting of $\mathbf{s}_0 = \mathbf{W}_{in}\mathbf{u}$: Since $\mathbf{x}(n) = f(\mathbf{a}. * \mathbf{W}_{in}\mathbf{u}(n) + \mathbf{b} + \mathbf{W}\mathbf{x}(n-1)) = \mathbf{a}. * \mathbf{s}_0 + \mathbf{b} + \mathbf{W}\mathbf{x}(n-1))$, $\mathbf{a}. * \mathbf{s}_0 + \mathbf{b}$ only partly contributes to calculation of $\mathbf{x}(n)$, unlike in case of ELM where $\mathbf{x}(n) = f(\mathbf{a}. * \mathbf{W}_{in}\mathbf{u}(n) + \mathbf{b})$ and $\mathbf{a}. * \mathbf{s}_0 + \mathbf{b}$ fully determines $\mathbf{x}(n)$. Moreover, $x = [-1, 1]$ is a range where f(x) is steepest, hence elements of $\mathbf{x}(n)$ will be more spread out there.
- The best test error 0.00109 comes from sigmoid with uniform range $[-1, 1]$, it reduces 69% of the test error made by linear regression and 20% of the smallest test error made by original ESN. The error comparisons are displayed below:

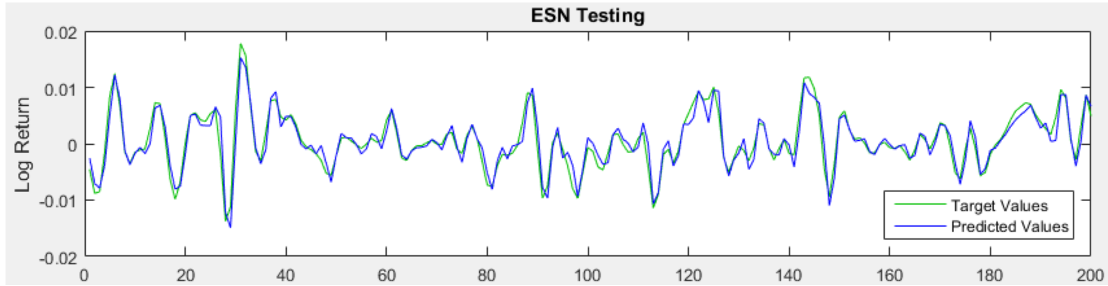


Figure 22: ESN on denoised stock returns test set

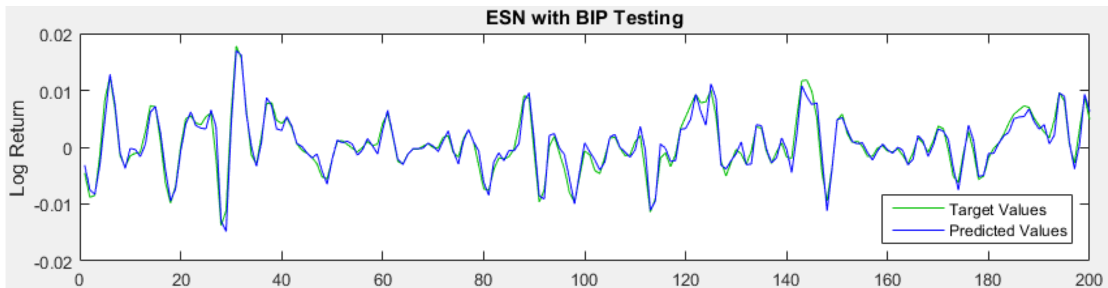


Figure 23: ESN with BIP on denoised stock returns test set

Also, general performance of BIP-attached ESN over 100 stocks is assessed using the following common parameter setting:

Leak Rate γ	1	Spectral Radius	0.65
Reservoir size N_x	150	Activation function	sigmoid
Reservoir Size $\mathbf{W}_{in}$	$[-2, 2]$	Reservoir Weight Matrix $\mathbf{W}$	$[-0.5, 0.5]$
Reservoir Density	0.5	Regularization term λ	10^{-2}

Again, the desired target range used is $[-1, 1]$, which gives the following error table:

Errors over stocks	Average Training error(RMS)	Average Test error(RMS)
Linear Regression	0.00809	0.00799
ESN	0.00498	0.00515
ESN with BIP	0.00397	0.00459

Simple calculations show that BIP-attached ESN reduces on average 20% of training error and 11% of test error made by plain ESN. This observation underpins BIP as a promising tool to improve ESN.

3.2 RADIAL BASIS FUNCTION(RBF) AND GRADIENT DESCENT

Radial Basis Function(RBF) are a certain type of functions whose value only depend on the distance from some center $\mathbf{c}$. Any function ϕ satisfying $\phi(\mathbf{x}, \mathbf{c}) = \phi(||\mathbf{x} - \mathbf{c}||)$ is called a radial basis function. The norm here is usually taken as Euclidean norm. As an important branch of neural networks, RBF networks have numerous applications including system identification and non-linear chaotic time series forecasting. It has rather distinct properties from ESN(e.g. only require small reservoir size). In this section, the most commonly used Gaussian kernel is adopted:

$$\mathbf{x}_i = \Phi(||\mathbf{u} - \mathbf{c}_i||) = exp\{-||\mathbf{u} - \mathbf{c}_i||^2/(2\beta_i^2)\} \tag{20}$$

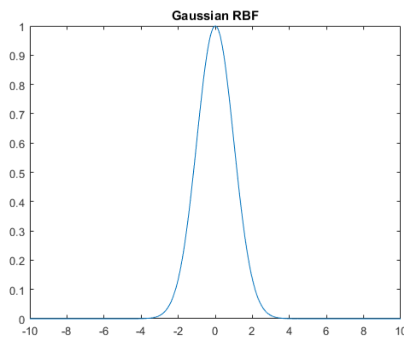


Figure 24: Gaussian kernel

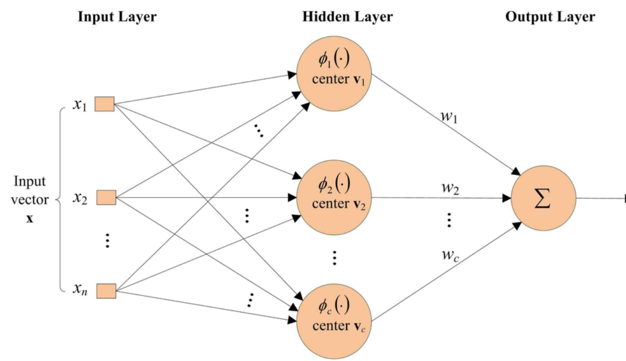


Figure 25: RBF

Basically, each time a new input $\mathbf{u}$ is imported, it is transformed into $\mathbf{x}$ using radial basis function 20. Then multiplication of $\mathbf{x}$ with output weight matrix $\mathbf{W}_{out}$ gives the output. This time, the output weight matrix $\mathbf{W}_{out}$ is no longer trained using a supervised ridge regression. Instead, it is tuned along with other parameters. Generally, with $\mathbf{w}$ standing for $\mathbf{w}_{out}$, the triplet $\{\mathbf{c}, \beta, \mathbf{w}\}$ [11] needs to be tuned together using a learning algorithm called *gradient descent*, which iteratively takes steps proportional to the negative of the current gradient of the function. The following are update equations:

$$w_i(n+1) = w_i(n) - \eta_1\epsilon(n)C(n)\mathbf{x}_i(n) \tag{21}$$

$$\mathbf{c}_i(n+1) = \mathbf{c}_i(n) - \eta_2\epsilon(n)w_i(n)C(n)\frac{\mathbf{x}_i(n)}{\beta_i(n)^2}(\mathbf{u}(n) - \mathbf{c}_i(n)) \tag{22}$$

$$\beta_i(n+1) = \beta_i(n) - \eta_3\epsilon(n)w_i(n)C(n)\frac{\mathbf{x}_i(n)}{\beta_i(n)^3}||\mathbf{u}(n) - \mathbf{c}_i(n)|| \tag{23}$$

Where η_1, η_2, η_3 are learning rates which control the speed of the process. $\epsilon(n)$ is the difference between target value and predicted value at time n i.e.

$$\epsilon(n) = \hat{y}(n) - y(n) = \mathbf{w}(n)\mathbf{x}(n) - y(n) \tag{24}$$

$C(n)$ is an additional feature called stochastic data-time effective function:

$$C(n) = \frac{1}{\tau} exp(\int_{t_0}^{n} \mu(t)dt + \int_{t_0}^{n} \sigma(t)dW_t) \tag{25}$$

The focus is on gradient descent algorithm itself, so for simplicity $\mu(t)$ and $\sigma(t)$ are both set to be zero. Below is the pseudo code for training RBF using gradient descent:

Algorithm 5 RBF Training Algorithm

1: **function** $modelOutput = rbfTrain(\mathbf{u}, \mathbf{y}, modelInputs)$ ▷ $modelInputs \leftarrow N_x, \eta, \tau$ ...
2: $\mathbf{c} \in \mathbb{R}^{N_u \times N_x}, \boldsymbol{\beta} \in \mathbb{R}^{N_x \times 1}, \mathbf{W}_{out} \in \mathbb{R}^{1 \times N_x}, \mathbf{yHat} \in \mathbb{R}^{1 \times nbDataPoints}$
3: $k = 1$, $error = 10^5$, $maxIteration = 500$, $tolerance = 0.008$ ← *Initializations*
4: **while** ($error >$ tolerance **and** $k < maxIteration$ **do**)
5: **for** $j = 1$ to $nbDataPoints$ **do**
6: **for** $i = 1$ to N_x **do**
7: $\boldsymbol{\phi}(i) = exp(-\frac{||\mathbf{u}(:,j) - \mathbf{c}(:,i)||^2}{2\boldsymbol{\beta}(i)^2})$
8: **end for**
9: $\epsilon = \mathbf{W}_{out}\boldsymbol{\phi} - \mathbf{y}(j)$
10: $C = 1/\tau$
11: **for** $i = 1$ to N_x **do**
12: $w_i = \mathbf{W}_{out}(i)$
13: $\mathbf{c}_i = \mathbf{c}(:, i)$
14: $b_i = \boldsymbol{\beta}(i)$
15: $\mathbf{W}_{out}(i) = \mathbf{W}_{out}(i) - \eta\epsilon C\boldsymbol{\phi}(i)$
16: $\mathbf{c}(:, i) = \mathbf{c}(:, i) - \eta\epsilon C w_i \boldsymbol{\phi}(i)(\mathbf{u}(:, j) - \mathbf{c}_i)/b_i^2$ ▷ $\mathbf{c}(i)$ *is a vector*
17: $\boldsymbol{\beta}(i) = \boldsymbol{\beta}(i) - \eta\epsilon C w_i \boldsymbol{\phi}(i)||\mathbf{u}(:, j) - \mathbf{c}_i||/b_i^3$
18: **end for**
19: **end for**
20: **for** $j = 1$ to $nbDataPoints$ **do**
21: **for** $i = 1$ to N_x **do**
22: $\boldsymbol{\phi}(i) = exp(-\frac{||\mathbf{u}(:,j) - \mathbf{c}(:,i)||^2}{2\boldsymbol{\beta}(i)^2})$
23: **end for**
24: $\mathbf{yHat}(j) = \mathbf{W}_{out}\boldsymbol{\phi}$
25: **end for**
26: $m \leftarrow nbPointsToIgnore + 1$;
27: $\mathbf{y}_1 = (\mathbf{yHat(:,m:end)} - b)/a$ ▷ *the first m points are discarded*
28: $\mathbf{y}_2 = (\mathbf{y(:,m:end)} - b)/a$ ▷ *the y values are mapped back before calculating errors*
29: $error = \sqrt{\frac{1}{nbDataPoints} \sum_{n=1}^{nbDataPoints} (\mathbf{y}_1(n) - \mathbf{y}_2(n))^2}$
30: $k = k + 1$
31: **end while**
32: Store updated $\mathbf{c}$, $\boldsymbol{\beta}$ and $\mathbf{W}_{out}$
33: **end function**

Note in the above algorithm,

- Suppose $N_y = 1$, then $\mathbf{c}$ is a matrix of size $N_u \times N_x$, whenever a new input $\mathbf{u}(n)$ is imported, every column of $\mathbf{c}$ is updated using $\mathbf{c}(:,i) = \mathbf{c}(:,i) - k(\mathbf{u}(:,j) - \mathbf{c}_i)$ where k is some scalar. Similarly, $\boldsymbol{\beta}$ is a vector of size $N_x \times 1$ and $\mathbf{w}$ is a vector of size $1 \times N_x$. They will be updated accordingly.

- The algorithm will terminate if either the training error is less than the predefined tolerance, or the maximum number of iterations are hit.

With the following parameter settings:

Reservoir Size N_x	15	Input size N_u	1
Normalization Range	$[0,1]$	Learning Rate η	0.001
Data-time Effective Function C	10	Maximum Iterations	500

and these initializations using uniform distributions on certain ranges:

$\mathbf{c}$: $[0,1]$	β: $[0.1, 0.3]$	$\mathbf{w}$: $[-0.1, 0.1]$

Setting the tolerance to 0.008 would give the following error table and the corresponding prediction performance:

	Training error(RMS)	Testing error(RMS)
Linear Regression	0.0315	0.032
RBF	0.008	0.00771

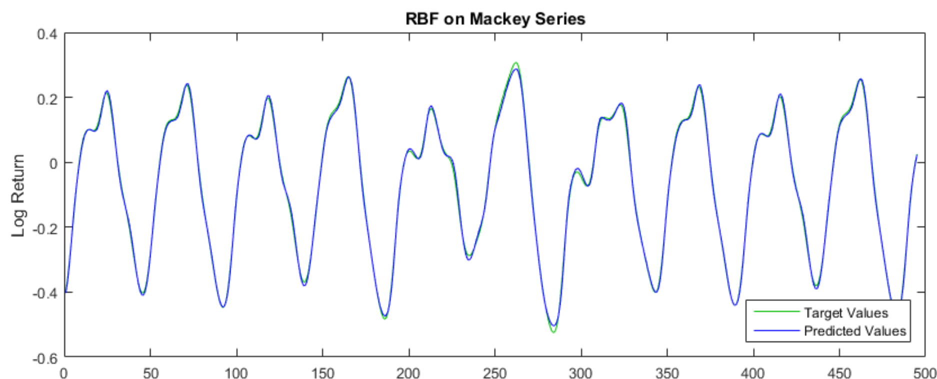


Figure 26: RBF on Mackey Series test set

This first experiment costs 33 seconds, and the evolution of training errors are plotted:

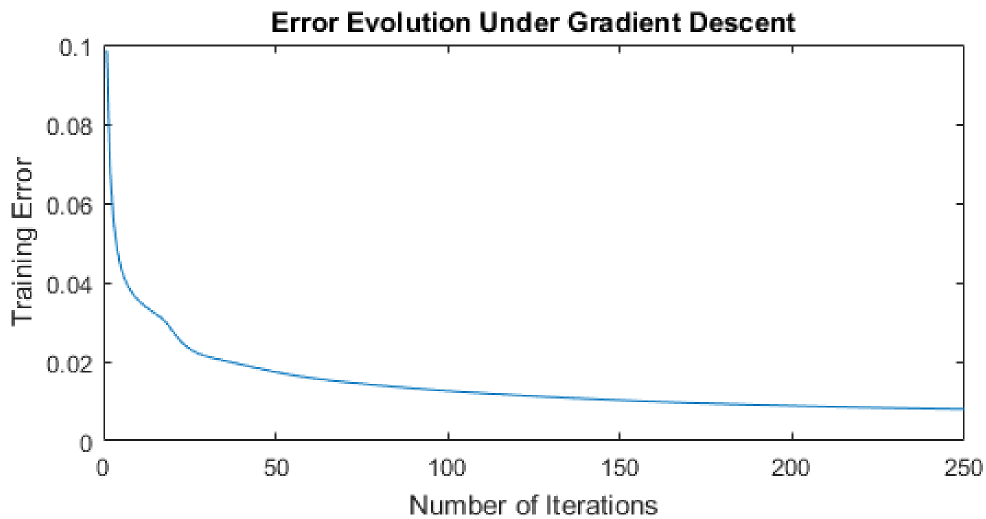


Figure 27: Error Evolutions

Now if the tolerance is set to be 0.007, i.e. require the training error to be no more than 0.007, then it takes 53 seconds to reach this error target:

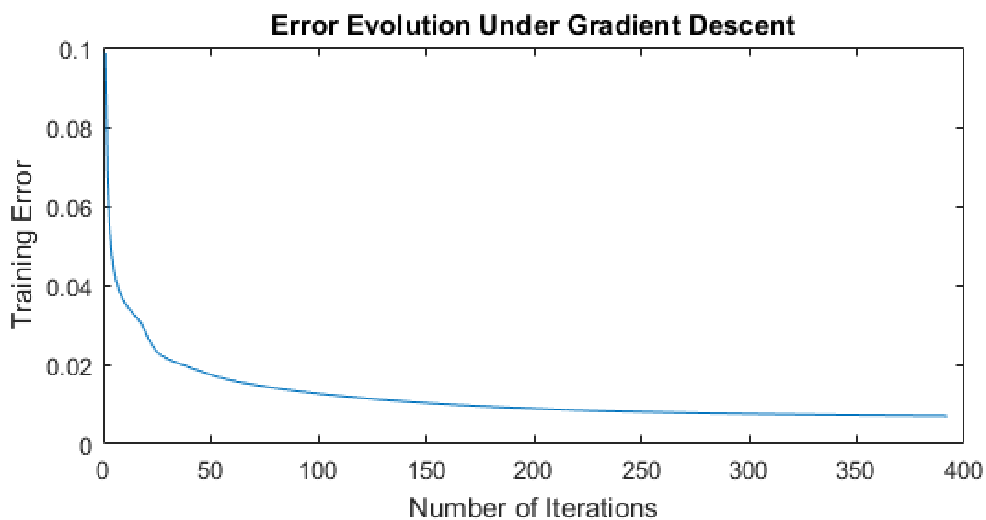


Figure 28: Error Evolutions

Therefore, technically it takes 20 seconds for the algorithm to get rid of the last 0.001 training error. That means, it consumes additional 60% of the previous running time to reduce 12% of the error. Seen from the table, gradient descent has fast convergence rate at the beginning. However, after 200 iterations it struggles to just improve a little bit. Now, other parameters kept the same(the same random initializations are reproducible), more experiments are done using various learning rates:

Learning Rate η	Running Time(seconds)
0.001	53
0.002	43
0.005	63
0.0005	66

In the table, only a learning rate of 0.002 speeds up the process by 10 seconds. However, further increase of the learning rate($\eta = 0.005$) no longer helps.
In additional to learning rate, the evolvements of $\mathbf{w}$, $\boldsymbol{\beta}$ and $\mathbf{c}$ are also of interest. In the original case where tolerance=0.008 and learning rate=0.001, the final form of $\boldsymbol{\beta}$ is as follows:

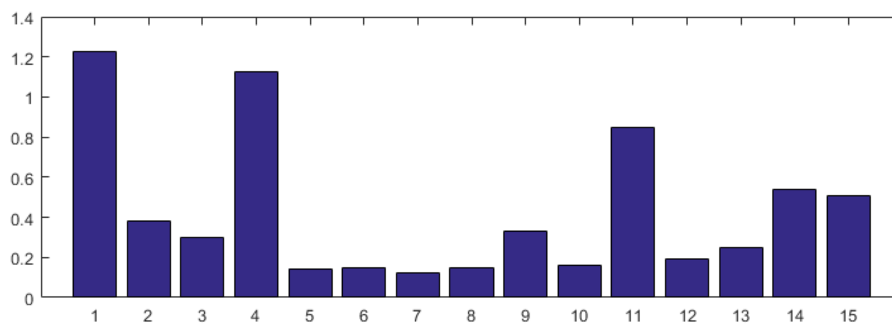


Figure 29: bar plot of $\boldsymbol{\beta}$

Since $N_x = 15$, each bar represents each element of $\boldsymbol{\beta}$. Note that $\boldsymbol{\beta}$ is initialized to be within range $[0.1, 0.3]$. But seen from the plot, about $1/3$ of the elements go far beyond its upper bound 0.3, in particular, the highest value is 1.23, which is as 4 times as 0.3, and the lowest value is 0.12. Hence, if the initialization range for $\boldsymbol{\beta}$ is expanded to $[0, 1]$ or $[0.12, 1.23]$, it is possible that the running time will be further reduced. Empirically however, neither case manages to decrease the original running time. Nevertheless, if the learning rate is adjusted to 0.002, all three cases achieve reduction of 10 seconds on average. On the other hand, similar experimental results apply to the case of $\mathbf{w}$ and $\mathbf{c}$. Generally, the learning rate is a delicate parameter in machine learning, if it is set small, it will take too much time to converge. However, if it is greater than needed, it may never converge and even diverge. See the following figure for intuition:

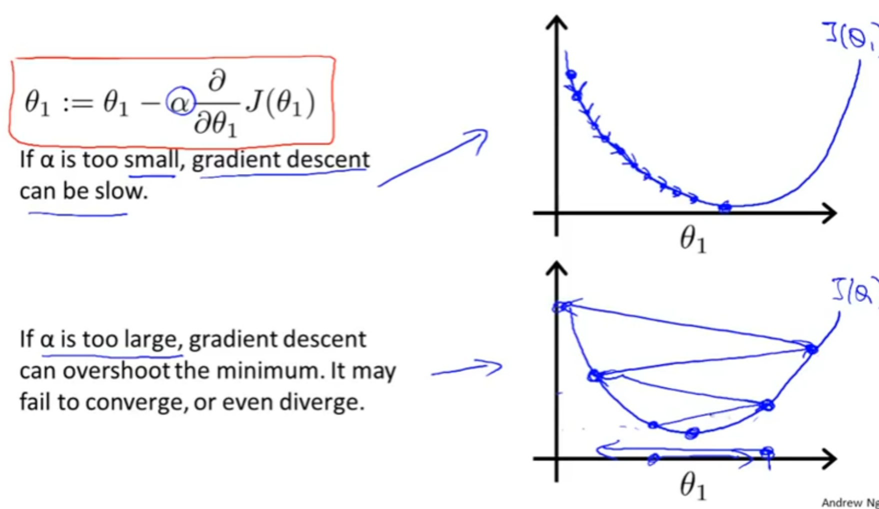


Figure 30: Gradient Descent-Andrew Ng

At the same time, although for gradient descent, initializations are not required to be close-to-optimal, it has to be within a certain range for the algorithm to reach the target error within reasonable time. Ideally, gradient descent works best on convex functions. If the function has multiple peaks or several local minima, given different initializations, gradient descent may end up finding the global minimum or being trapped in a local minimum.

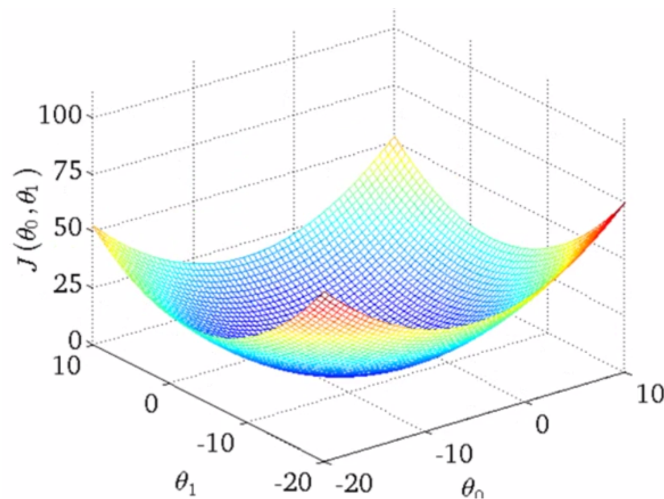


Figure 31: Convex Functions

In summary, this section introduces a new parameter optimizing method. The main advantage of gradient descent is that one can specify the desired error level beforehand

so that task running time could be allocated more efficiently according to different accuracy requirements. At the same time, however, it has several limitations:

- The setting of learning rate largely depends on experience. Inappropriate setting can make it impossible to reach the target error level within reasonable time.
- Gradient descent is relatively slow close to the minimum: technically, its asymptotic rate of convergence is inferior to many other methods[WikiPedia].
- Gradient descent fails confronting non-differentiable functions. Thus these functions may need to be bounded by a smooth function for gradient descent to work.

3.3 RECURRENT RBF NETWORK

3.3.1 *Standard RRBFN*

Recall the RBF model has the following form:

$$\mathbf{x}_i = \Phi(||\mathbf{u} - \mathbf{c}_i||) = exp\{-||\mathbf{u} - \mathbf{c}_i||^2/(2\beta_i^2)\} \tag{26}$$

Now, this model is featured with recurrent structure so that it turns into an advanced structure called Recurrent Radial Basis Function Network(RRBFN), which has the form:

$$\mathbf{x}_i(n) = \Phi(||\mathbf{u}(n) - \mathbf{c}_1^i||, ||\mathbf{x}^i(n-1) - \mathbf{c}_2^i||) \tag{27}$$

$$= exp\{-(||\mathbf{u}(n) - \mathbf{c}_1^i||^2 + ||\mathbf{x}_i(n-1) - \mathbf{c}_2^i||^2)/(2\beta_i^2)\} \tag{28}$$

This time, the function values depend on two inputs instead of one. Two 'distances' from the two center vectors determine the output value. More specifically, its update equation is:

$$\widetilde{\mathbf{x}}_i(n) = exp(-\alpha||\mathbf{W}_{in}^i - \mathbf{u}(n)||^2 - \beta||\mathbf{W}^i - \mathbf{x}(n-1)||^2), \quad i = 1, ..., N_x \tag{29}$$

where $\widetilde{\mathbf{x}}_i(n)$ is the *ith* element of vector $\widetilde{\mathbf{x}}(n) \in \mathbb{R}^{N_x}$; $\mathbf{W}_{in}^i$ and $\mathbf{W}^i$ are *ith* row of $\mathbf{W}_{in}$ and $\mathbf{W}$ respectively, α and β are scalar parameter which absorb the information of β_i in equation 26

Pseudo Code

The following is the pseudo code for training RRBFN:

Algorithm 6 RRBFN Training Algorithm

1: **function** $rrbfTrain(\mathbf{u}, \mathbf{y}, modelInputs)$ ▷ $modelInputs \leftarrow N_x, \mathbf{W}_{in}, \mathbf{W}, \mathbf{X}_0$ *etc.*
2: $\mathbf{X} \in \mathbb{R}^{N_x \times nbDataPoints} \leftarrow$ *Design Matrix*
3: **for** $i = 1$ to $nbDataPoints$ **do**
4: **for** $j = 1$ to N_x **do**
5: $\mathbf{X}(j,i) = exp(-\alpha||[\mathbf{W}_{in}(j,:)]^T - \mathbf{u}(:,i)||^2 - \beta||[\mathbf{W}(j,:)]^T - \mathbf{X}(:,i-1)||^2);$
6: **end for**
7: $\mathbf{z}(:,i) = [1; \mathbf{u}(:,i); \mathbf{X}(:,i)];$ ▷ *Vertical Concatenation*
8: **end for**
9: $m \leftarrow nbPointsToIgnore + 1;$
10: $S = \mathbf{z}(:, m : end);$
11: $D = \mathbf{y}(:, m : end);$
12: $\mathbf{W}_{out} = DS^T pinv(SS^T + \lambda I);$ ▷ *Ridge Regression, 'pinv' is pseudo inverse*
13: **end function**

Parameter Tuning

As a benchmark chaotic time series, Mackey series is again used to tune parameters of RRBFN. Compared to other chaotic even noisy time series, it is relatively smooth and tractable. Therefore, if a machine learning model cannot succeed on Mackey series, then it is unlikely to have strong potential in predicting chaotic time series. Most importantly, Mackey Series could be used to find priority of parameters. Generally,

- An increase in N_u will monotonically decrease both training error and test error to a certain level. However it is not recommended to set a high value for N_u since it increases the chance of overfitting.
- W barely changes results.
- Other parameters give no obvious trends and thus need to be tuned one at a time.

It is worth mentioning that α and β have similar positions in the model, they both are scalar factors of a certain 'distance'.

- α is the distance between *ith* row of $\mathbf{W}_{in}$ and input vector $\mathbf{u}(n)$.
- β is the distance between *ith* row of $\mathbf{W}$ and reservoir state $\mathbf{x}(n-1)$.

Therefore, they should be treated as a pair. First of all, assume α and β have equal importance. Start from $\alpha = \beta = 0.1$, a monotonic increase in both values until $\alpha = \beta = 0.9$ gives a general decrease in errors. In particular, $\alpha = \beta = 0.6$ is chosen to be a robust set since there is a huge drop in errors when $\alpha = \beta = 0.5$ is changed to $\alpha = \beta = 0.6$, and also this is the first time when test error is less than 10^{-3}. Nevertheless,

$\alpha = \beta = 0.9$ is robust too, for it produces smallest test error when α and β are set equal.
Secondly, it remains to be seen which one has more impact on final result. Fixing other parameters, the following statistics are obtained:

α	0.6	0.6	1
β	0.6	1	0.6
Test error(RMSE)	0.000737	0.000684	0.000852

From this table, it could be observed that 66% change in β leads to 7% decrease in the test error, while 66% change in α results in 15.6% increase in the test error, which indicates that generally α is more influential on the error term. Then fixing $\alpha = 0.6$, careful tuning on β gives the final pair:

α	0.6
β	0.9
Test error(RMSE)	0.000669

Similarly, it is possible that some other parameters are also strongly inter-connected and should be tuned together. However, identification of this requires additional expertise and deep analysis on the behaviour of parameters.
Finally, with cautiously tuned parameters below,

[α,β]	[0.6,0.9]	Spectral Radius	1.25
Reservoir size N_x	10	Input Size N_u	1
Reservoir Size $\mathbf{W}_{in}$	$[-1, 1]$	Regularization Term	10^{-8}

Without using data normalization, the errors are as displayed:

	Training error(RMS)	Test error(RMS)
Linear Regression	0.0315	0.032
RRBF	0.000705	0.000648

From now on it is assumed that 0.000648 is the smallest test error that can be achieved by standard RRBFN with $N_u = 1$. This important result will be cited later for further comparison and evaluation with new methods.

3.3.2 *Randomly Parametrized RRBFN*

Based on the conjecture that each entry of input vector $\mathbf{u}(n)$ or reservoir states $\mathbf{x}(n)$ can contribute differently to the new reservoir state $\mathbf{x}(n+1)$, modifications on distance computation are made hoping to take advantage of this hypothesis. For example, if $\mathbf{u}(n) = (u_1, u_2, u_3, u_4, u_5)^T$ is a vector representing the log returns of the past 5 days, then u_1 could have more impact on $\mathbf{u}(n+1)$ then u_5 does. Therefore, when computing $\mathbf{W}_{in}^i - \mathbf{u}(n)$, the resultant vector $[0.1, 0, 0, 0, 0]^T$ should be distinguished from

$[0,0,0,0,0.1]$. If one favours small values of the norm $||\mathbf{W}^i_{in} - \mathbf{u}(n)||$, then the latter vector is more desirable than the first one since u_1 matters more than u_5. However, in the sense of previous Euclidean norm, the two vectors have exactly the same norm. Therefore, *weights* may be assigned to different positions within a vector to unearth and make use of the priority of importance.
However, usually the order of importance within $\mathbf{u}(n)$ is unknown and it may be constantly changing. If a specific order list is presumed and widely used over different time periods and stocks, generality of the model may be impaired. Hence, N_u being the length of vector $\mathbf{u}(n)$, a randomly assigned weight distribution is generated following the steps below:

- Draw N_u samples from standard uniform distribution $\mathrm{U}(0,1)$.
- Calculate the sum of N_u samples, call it s;
- Divide each sample by s to get a probability distribution $\mathbf{p}$;
- Multiply each element of $\mathbf{p}$ by N_u to get final weight vector $\mathbf{w}_1$;

In the situation of equal weight distribution, all elements of $\mathbf{p}$ are the same and equal to $1/N_u$, and the weight vector $\mathbf{w}_1 = N_u * \mathbf{p}$ consists of all ones. This special case is equivalent to the previous standard RRBFN, where distances are measured in Euclidean norm.
The above justifications and calculations apply similarly to the norm $||\mathbf{W}^i - \mathbf{x}(n-1)||$, with N_u changed into N_x.

Fixing the optimal parameter settings for standard RRBF, iterations over different weight vector $\mathbf{w}_1$ show that random parametrized RRBF is relatively unpredictable in improving standard RRBF. Sometimes, it could reduce the test error by 20%; On the other hand, with some extreme distribution of $\mathbf{w}_1$, this could be the other way round. Therefore, its performance largely depends on initialization of $\mathbf{w}_1$ and is relatively unstable.
Below is a case where $\mathbf{w}_1$ is randomly initialized using rand('seed',9). In this case, randomly parametrized RRBF reduces 17% of the test error made by standard RRBF:

	Training error(RMS)	Test error(RMS)
Linear Regression	0.0315	0.032
Standard RRBF	0.000705	0.000648
Randomly Parametrized RRBF	0.000591	0.000539

3.3.3 *Directional Parametrized RRBFN*

Due to the fact that the performance of randomly parametrized RRBFN greatly relies on random generation of weight vector $\mathbf{w}_1$, one may want to alleviate or get rid of

the random factor by generating $\mathbf{w}_1$ using non-random rules. This type of network is called directional parametrized RRBFN. It is paid attention since in the previous experiment, some distributions of $\mathbf{w}_1$ do lead parametrized RRBFN to a better performance than that of standard RRBFN.
In my trial, elements of $\mathbf{w}_1$ are set to be proportional to magnitudes of elements in vector $\mathbf{W}^i_{in} - \mathbf{u}(n)$. For instance, if $\mathbf{W}^i_{in} - \mathbf{u}(n) = (1,3)^T$, then the corresponding $\mathbf{p}$ and $\mathbf{w}_1$ would be $(0.25, 0.75)^T$, $(0.5, 1.5)^T$ respectively. Note that only the magnitude matters. So if $\mathbf{W}^i_{in} - \mathbf{u}(n) = (-1,3)$, the corresponding $\mathbf{w}_1$ would still be $(0.25, 0.75)^T$. Under this rule, the vector difference between $\mathbf{W}^i_{in}$ and $\mathbf{u}(n)$ is amplified. For example, if $\mathbf{W}^i_{in} - \mathbf{u}(n) = (1,3)^T$ holds,

- Euclidean norm would give $\sqrt{1^2 + 3^2} = \sqrt{10}$
- Under new rule, the norm is $\sqrt{0.5 * 1^2 + 1.5 * 3^2} = \sqrt{14}$

Empirically, with proper parameter settings:

- α=0.2, β=0.8
- Spectral Radius=0.9

Directional parametrized RRBF is able to improve the best standard RRBF by a meaningful amount, please refer to the following table for details:

	Training error(RMS)	Test error(RMS)
Linear Regression	0.0315	0.032
Standard RRBF	0.000705	0.000648
Randomly Parametrized RRBF	0.000591	0.000539
Directional Parametrized RRBF	0.000548	0.000479

From the table, it could be seen that directional parametrized RRBF successfully reduces 26% of the test error made by standard RRBF. This observation is encouraging since it shows that directional parametrized RRBF may have potential to *steadily* improve RRBF with general stock data.

3.3.4 *Empirical Results*

Firstly, the test is on a specific stock. For consistency, stock 'MMM' is again used. After plugging in appropriate parameters, the errors are displayed below:

	Training error(RMS)	Test error(RMS)
Linear Regression	0.00472	0.0036
Standard RRBF	0.00290	0.00173
Directional Parametrized RRBF	0.00269	0.00153

Again, the above errors in the table are smallest errors(especially test error) attainable using corresponding methods. This time, directional parametrized RRBF decreases 12% of the test error made by standard RRBF, which is a good start.
Now, with a common parameter setting, data are collected across 100 stocks:

	Average Training error(RMS)	Average Test error(RMS)
Linear Regression	0.00809	0.00799
Standard RRBF	0.00531	0.00598
Directional Parametrized RRBF	0.00519	0.00532

Several comments are given below:

- To some extent, it is proved that directional parametrized RRBF is capable of improving standard RRBF in terms of both training and test error.
- On average, directional parametrized RRBF decreases 11% of the test error made by standard RRBF. However, it takes around 1 minute to run the program, while standard RRBF costs 4 seconds. Nevertheless, the time is still acceptable.
- In general, an significant increase in N_x does not effectively reduce test error. This might be one of the main differences between ESN and RRBF: For RRBF, N_x is usually chosen to be less then 15, while in ESN, N_x can be in hundreds, thousands, even in ten thousands. This structural difference may be part of the reason why RRBF has average test error which is 17% higher than that of ESN.

3.3.5 *Unsupervised Learning for RRBFN*

There are many unsupervised learning algorithm sfor training RBF type neural networks. One of them is called self organizing map(SOM) [6]

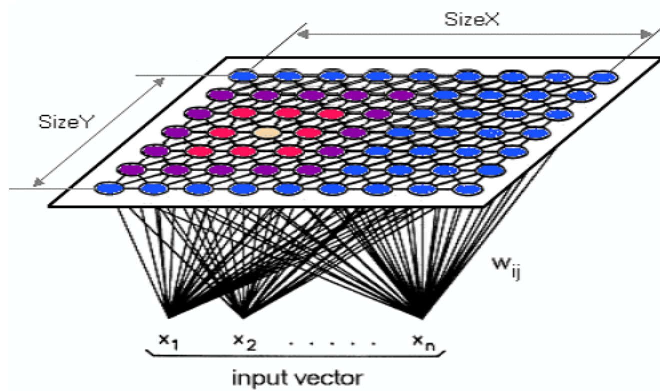


Figure 32: Self-Organizing Maps

Its update equations are:

$$\mathbf{W}_{in}^{i}(n+1) = \mathbf{W}_{in}^{i}(n) + \eta(n)h(i,n)(\mathbf{u}(n) - \mathbf{W}_{in}^{i}(n)) \tag{30}$$

$$\mathbf{W}^{i}(n+1) = \mathbf{W}^{i}(n) + \eta(n)h(i,n)(\mathbf{x}(n) - \mathbf{W}^{i}(n)) \tag{31}$$

where $\eta(n)$ is learning rate and $h(i,n)$ is called learning gradient distribution function:

$$h(i,n) = exp(-d_h(i, bmu(n))^2 / b_h(n)^2) \tag{32}$$

In the above equation, bmu(n)=argmax$_i$($\mathbf{x}_i$(n)) is the index of the 'best matching unit'(BMU), $d_h(i,j)$ is the distance between units with indices i and j in the additionally defined topology for reservoir units. In our case, $d_h(i,j)$ is:

- either the Euclidean distance between $\mathbf{W}_{in}^{i}$and $\mathbf{W}_{in}^{bmu(n)}$.
- or the Euclidean distance between $\mathbf{W}_i$ and $\mathbf{W}_{bmu(n)}$.

Practically, every time a new input $\mathbf{u}(n)$ is imported, firstly the reservoir state $\mathbf{x}(n)$ is updated using $\mathbf{u}(n)$ and $\mathbf{x}(n-1)$:

$$\widetilde{\mathbf{x}}_i(n) = exp(-\alpha||\mathbf{W}_{in}^{i} - \mathbf{u}(n)||^2 - \beta||\mathbf{W}^{i} - \mathbf{x}(n-1)||^2), \quad i = 1, ..., N_x \tag{33}$$

Then, equations 30 and 31 are used to update rows of $\mathbf{W}_{in}$ and $\mathbf{W}$ adaptively. Meanwhile, this updating effect is controlled by a scaler called *learning rate* η. Empirical values for learning rate are $0.1, 0.01, 0.001$. In some cases, under certain criteria, the value of learning rate itself can dynamically change during the training process to better fit the dataset. But for simplicity fixed value for η is used here.

Take equation 30 for instance: consider input weight matrix $\mathbf{W}_{in} \in \mathbb{R}^{N_x \times N_u}$, with every row being a *center* vector to the new input $\mathbf{u}(n)$, then there are in total N_x center vectors. Hence, N_x is supposed to be the estimated number of *independent* center vectors. Ideally,

- These center vectors should be far from each other in Euclidean distance
- Every new input $\mathbf{u}(n) \in \mathbb{R}^{N_u}$ should close to all of the center vectors, i.e. all the rows of $\mathbf{W}_{in}$

However, due to the fact that $\mathbf{W}_{in}$ and $\mathbf{W}$ are randomly initialized, their rows may not function as suitable center vectors that perfectly match input dataset. Therefore, they need to be adaptively updated during the training process.

Whenever a new input $\mathbf{u}(n)$ is imported, its distances to every rows of $\mathbf{W}_{in}$ are computed. Among all N_x distances, the best matching unit(BMU) is the row index of $\mathbf{W}_{in}$ which is closest to $\mathbf{u}(n)$. Then, assuming $bmu = k$, the distances from the kth row to

every other rows of $\mathbf{W}_{in}$ are calculated and the results are summarized in the term $h(i,n)$, where i means ith row and n indicates the time point. Specifically, it has the following expression:

$$h(i,n) = exp(-||\mathbf{W}_{in}^{i}(n) - \mathbf{W}_{in}^{k}(n)||^2/(2\sigma^2)) \tag{34}$$

Here the setting of σ varies from case to case, usually it is the volatility of the variable of interest.

Now, look back to equation 30: if $\mathbf{u}(n)$ is closest to a center $\mathbf{W}_{in}^{k}(n)$, then $\mathbf{W}_{in}^{k}(n)$ becomes $\mathbf{W}_{in}^{bmu}(n)$. If $\mathbf{W}_{in}^{bmu}(n)$ is far from other center vectors, then the value of $h(i,n)$ should be small since the norm $||\mathbf{W}_{in}^{i}(n) - \mathbf{W}_{in}^{bmu}(n)||$ would be large. At the same time, if $\mathbf{u}(n)$ does not deviate too much from all the center vectors, $\mathbf{u}(n) - \mathbf{W}_{in}^{i}(n)$ would be relatively small for all i. Hence, the term $\eta(n)h(i,n)(\mathbf{u}(n) - \mathbf{W}_{in}^{i}(n))$ as a whole would be small if the learning rate $\eta(n)$ is selected and fixed. As a matter of fact, this is a desirable situation and $\mathbf{W}_{in}$ does not need to be modified much.

Moreover, it holds true that for $i = 1...N_x$,

$$||\mathbf{u}(n) - \mathbf{W}_{in}^{bmu}(n)|| <= ||\mathbf{u}(n) - \mathbf{W}_{in}^{i}(n)|| \tag{35}$$

$$1 = h(bmu(n), n) >= h(i,n) > 0 \tag{36}$$

However, it is not clear if $h(bmu(n),n)(\mathbf{u}(n) - \mathbf{W}_{in}^{bmu}(n)) >= h(i,n)(\mathbf{u}(n) - \mathbf{W}_{in}^{i}(n))$. i.e. we are not aware if $\mathbf{W}_{in}^{bmu}$ moves greater distance towards $\mathbf{u}(n)$ than other rows do. Practically, it turns out that most of the time $\mathbf{W}_{in}^{bmu}$ tends to move faster due to the exponential decay of $h(i,n)$. Nevertheless, there exist other models such as Neural Gas(NG) [6] that require $h(bmu(n),n) = 0$. The above explanation similarly applies to equation 31. Only the 'distance' involved changes.

Furthermore, due to the fact that the rows of $\mathbf{W}_{in}$ is always compared to $\mathbf{u}(n)$, in some cases the first N_x training inputs $\mathbf{u}(1)$, $\mathbf{u}(2)$,...,$\mathbf{u}(N_x)$ are used to initialize $\mathbf{W}_{in}$:

$$\mathbf{W}_{in} = \begin{bmatrix} — & u(1)^T & — \\ — & u(2)^T & — \\ & \vdots & \\ — & u(N_x)^T & — \end{bmatrix} \in \mathbb{R}^{N_x \times N_u}$$

Pseudo Code

The following is the pseudo code for training RRBFN with SOM:

Algorithm 7 RRBFN with SOM Training Algorithm

1: **function** $rrbfSOMTrain(\mathbf{u}, \mathbf{y}, modelInputs)$ ▷ $modelInputs \leftarrow N_x, \mathbf{W}_{in}, \mathbf{W}, \mathbf{X}_0$ *etc.*
2: $\mathbf{X} \in \mathbb{R}^{N_x \times nbDataPoints} \leftarrow$ *Design Matrix*
3: **for** $i = 1$ to $nbDataPoints$ **do**
4: **for** $j = 1$ to N_x **do** ▷ *Finally update* $\mathbf{x}(n)$
5: $\mathbf{X}(j,i) = exp(-\alpha||[\mathbf{W}_{in}(j,:)]^T - \mathbf{u}(:,i)||^2 - \beta||[\mathbf{W}(j,:)]^T - \mathbf{X}(:,i-1)||^2);$
6: **end for**
7: $\mathbf{z}(:,i) = [1; \mathbf{u}(:,i); \mathbf{X}(:,i)];$ ▷ *Vertical Concatenation*
8: $BMU_1 = ||\mathbf{W}_{in}(1,:)^T - \mathbf{u}(:,i)||;$
9: $Index_1 = 1;$
10: **for** $j = 2$ to N_x **do** ▷ *To find* BMU_1
11: $Test = ||\mathbf{W}_{in}(j,:)^T - \mathbf{u}(:,i)||;$
12: **if** $Test < BMU_1$ **then**
13: $BMU_1 = Test;$
14: $Index_1 = j;$
15: **end if**
16: **end for**
17: $BMU_2 = ||\mathbf{W}(1,:)^T - \mathbf{X}(:,i)||;$
18: $Index_2 = 1;$
19: **for** $j = 2$ to N_x **do** ▷ *To find* BMU_2
20: $Test = ||\mathbf{W}(j,:)^T - \mathbf{X}(:,i)||;$
21: **if** $Test < BMU_2$ **then**
22: $BMU_2 = Test;$
23: $Index_2 = j;$
24: **end if**
25: **end for**
26: **for** $k = 1$ to N_x **do** ▷ *Update* $\mathbf{W}_{in}$ *and* $\mathbf{W}$
27: $\mathbf{W}_{in}(k,:) = \mathbf{W}_{in}(k,:) + \eta([\mathbf{u}(:,i)]^T - \mathbf{W}_{in}(k,:))exp(-||\mathbf{W}_{in}(k,:) - \mathbf{W}_{in}(Index_1,:)||^2/(2\sigma^2));$
28: $\mathbf{W}(k,:) = \mathbf{W}(k,:) + \eta([\mathbf{x}(:,i)]^T - \mathbf{W}(k,:))exp(-||\mathbf{W}(k,:) - \mathbf{W}(Index_2,:)||^2/(2\sigma^2));$
29: **end for**
30: **end for**
31: $m \leftarrow nbPointsToIgnore + 1;$
32: $S = \mathbf{z}(:, m : end);$
33: $D = \mathbf{y}(:, m : end);$
34: $\mathbf{W}_{out} = DS^T pinv(SS^T + \lambda I);$ ▷ *Ridge Regression, pinv is pseudo inverse*
35: **end function**

Empirical Results

Fix the optimal parameter setting for standard RRBF on stock 'MMM',

$[\alpha,\beta]$	[0.6,1]	Spectral Radius	1.25
Reservoir size N_x	12	Normalization Range	$[0,1]$
Reservoir Size $\mathbf{W}_{in}$	$[-1,1]$	Regularization Term	10^{-8}

and only add unsupervised update equation for $\mathbf{W}_{in}$ and $\mathbf{W}$(equation 30,31) with learning rate $\eta = 0.01$ to the algorithm, the corresponding errors are shown below:

	Training error(RMS)	Test error(RMS)
Linear Regression	0.00472	0.0036
Standard RRBF	0.00290	0.00173
Standard RRBF with SOM	0.00258	0.00163

Seen from the table, with adjustment of $\mathbf{W}_{in}$ and $\mathbf{W}$, the SOM algorithm manages to reduce 6% of the test error made by standard RRBF. Please see the evolution of input weight matrix $\mathbf{W}_{in}$ below:

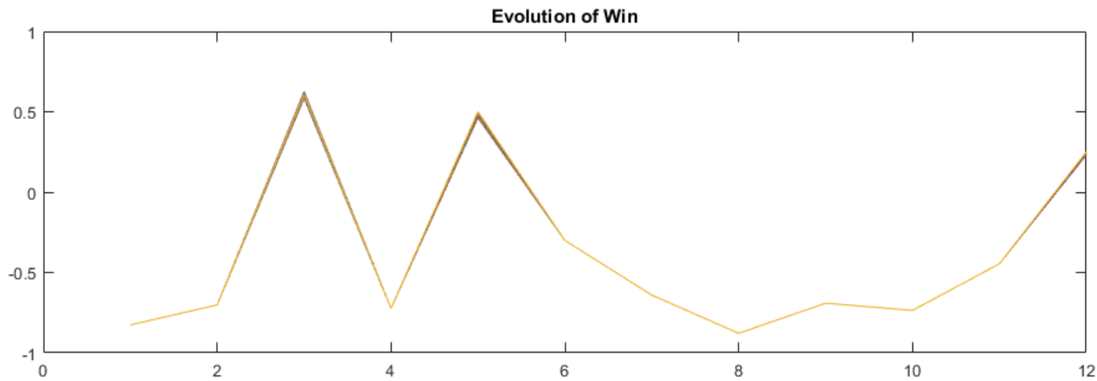


Figure 33: Evolution of $\mathbf{W}_{in}$ during training

Note that $\mathbf{W}_{in}$ has dimension $N_x \times N_u = 12 \times 1$. In the above figure, since $N_x = 12$, the x-axis has 12 discrete points representing 12 positions in the vector $\mathbf{W}_{in}$. For each position, the corresponding y-value is the element value on that position. In theory, there supposed to be 1200 curves, with each curve shown in unique color. However, most of the curves coincide with each other and one can hardly tell one from another. This figure indicates that $\mathbf{W}_{in}$ has not changed a lot during the process. With a closer look at the figure, the 3th, 5th and 12th element may be found to evolve a bit during the training process.
Kindly note that it is only in the training process that SOM is applied, the test set is always left untouched. At the end of the training process, the evolved $\mathbf{W}_{in}$ and $\mathbf{W}$ are used on test set. In order to reduce chance of overfitting, no further evolution is

undertaken at test stage.

Similar to $\mathbf{W}_{in}$, the evolution of $\mathbf{W}$ is also of interest. However, since $\mathbf{W}$ has $2D$ dimension of $N_x \times N_x$, it is difficult to visualize its adaptation process. Nevertheless, the initial and final formations of the reservoir weight matrix $\mathbf{W}$ are plotted:

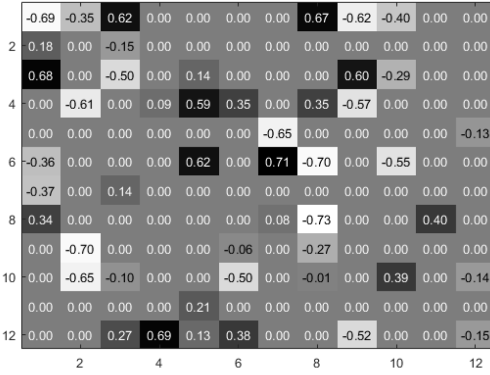


Figure 34: Initial W

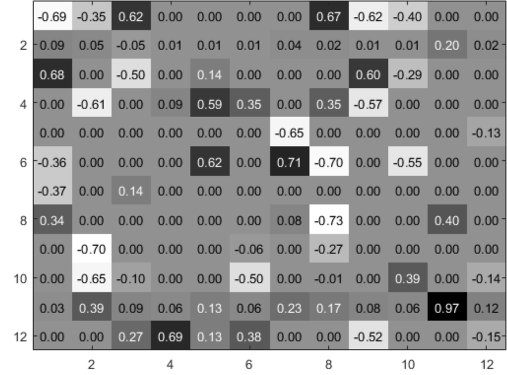


Figure 35: Evolved W

Note that the deeper the color, the higher the value. In the initial $\mathbf{W}$, the matrix is supposed to be sparse so that only 40% of them are non-zero. Comparing two figures, it could be seen that only 2nd and 11th row changes by a meaningful amount. This is partly because originally $\mathbf{W}$ is already relative close to optimal setting.

Now, other parameters kept fixed, if range of $\mathbf{W}_{in}$ changes to $[-0.5, 0.5]$, then the errors generated are as follows:

	Training error(RMS)	Test error(RMS)
Linear Regression	0.00472	0.0036
Standard RRBF	0.00332	0.00213
Standard RRBF with SOM	0.00319	0.00189

The corresponding evolution of $\mathbf{W}_{in}$ is also displayed:

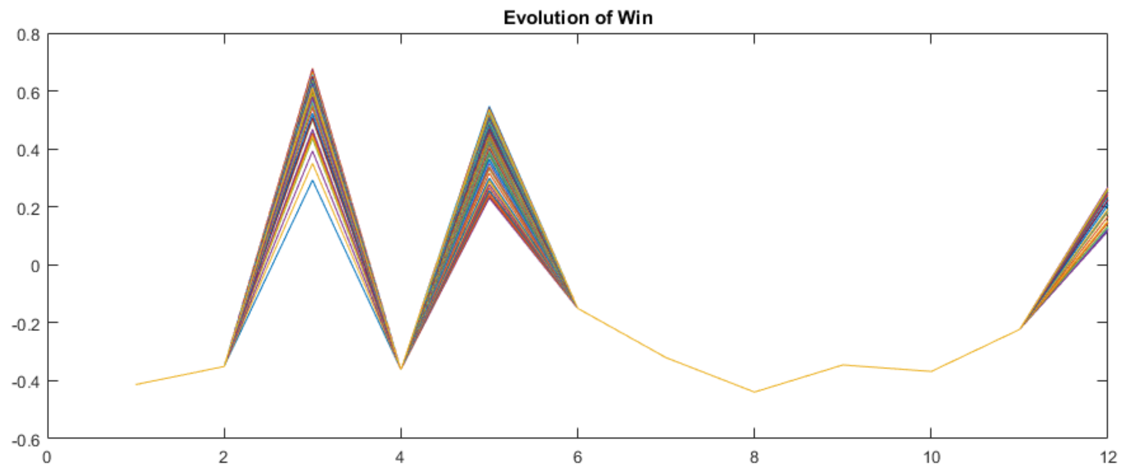


Figure 36: Evolution of $\mathbf{W}_{in}$

Several comments are given here:

- Seen from the table, SOM successfully reduces 11% of the error made by standard RRBF. The error reduction doubles compared to the previous case.
- The figure shows a relative large evolution on the 3th, 5th and 12th rows. It looks as if SOM tries to 'rectify' these rows of W_{in} and lead them to the right track. Generally, in those much more complex tasks, where optimal parameter settings are hard to find, SOM can guide the system adaptively in the right direction.

Now, the general performance of SOM is tested on 100 stocks. The parameter settings are kept exactly the same as the previous one used for standard RRBF tests across 100 stocks. The resultant errors are as follows:

	Average Training error(RMS)	Average Test error(RMS)
Linear Regression	0.00809	0.00799
Standard RRBF	0.00531	0.00598
Standard RRBF with SOM	0.00509	0.00571

Seen from the table, on average SOM reduces test errors by 5%, which is half of the previous 11% reduction on a specific stock. Moreover, the error percentage reduction distribution across 100 stocks is also displayed:

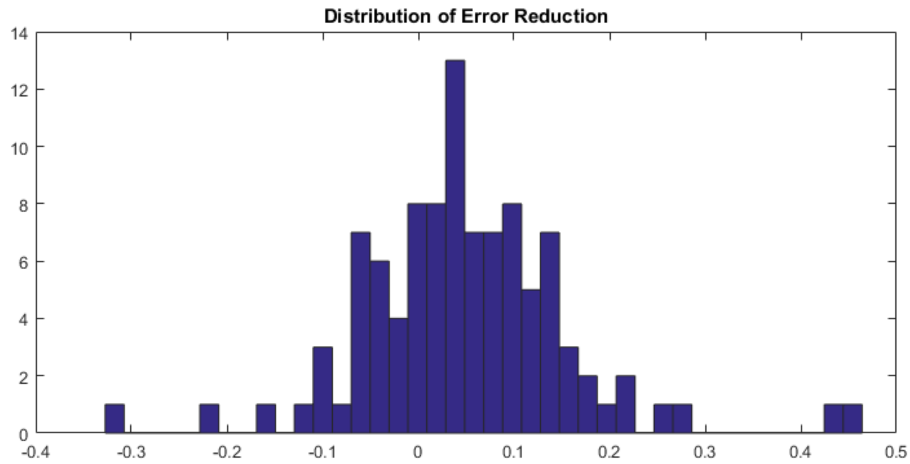


Figure 37: Histogram of Error Reduction

Since a positive number means that SOM does reduce test errors, statistically it turns out that 72% of the time SOM successfully decreases test errors made by standard RRBF, but in the rest 28% of the time, SOM undesirably increases the errors. The failure of SOM to decrease test errors can be partly due to a common setting of the learning rate η. Throughout the process only one single η is used. It is fixed during the training process, and over stocks. Therefore, if a changeable η is used, it is likely that SOM will remain effective more frequently. The following is a prediction comparison on a sample stock:

Figure 38: RRBF on test set

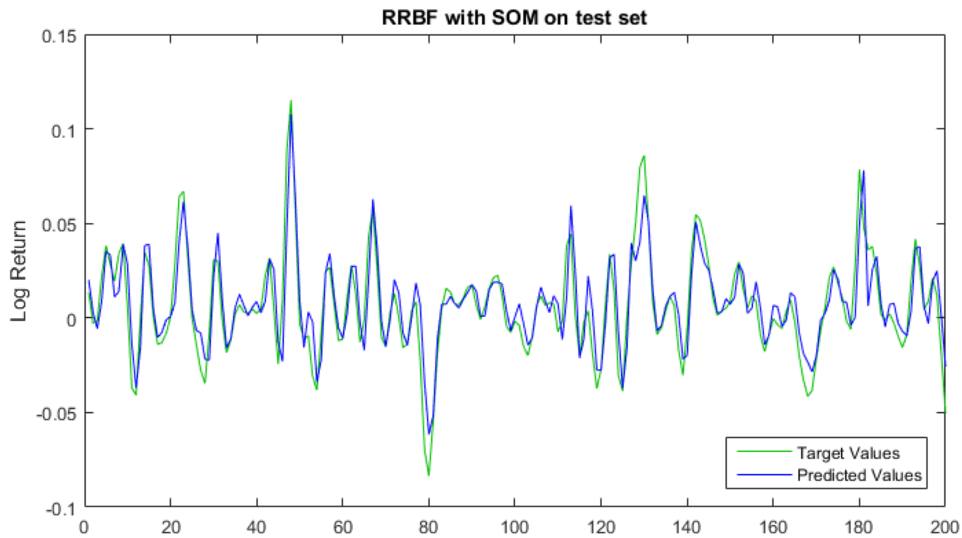


Figure 39: RRBF with SOM on test set

As for directional parametrized RRBF, the same experiments are done on the same 100 stocks using a common parameter setting. The results show that despite that on average training errors decrease by 6%, only 44% of the time SOM manages to reduce the test error made by directional parametrized RRBF. Moreover, the training process takes much more time than previous experiments. The incompatibility of SOM to directional parametrized may be due to that directional parametrized RRBF itself has already scaled element-wise the distance between $\mathbf{u}(n)$ and $\mathbf{W}_{in}^{i}$, hence a further modification on $\mathbf{W}_{in}$ might create adverse effects.

Overall, the SOM model is promising and useful on standard RRBF in the sense that most of the time it is able to adaptively lead rows of $\mathbf{W}_{in}$ and $\mathbf{W}$ to the right positions when the original settings are not suitable for input dataset. Moreover, if the learning rate could be made flexible, or another $h(i,n)$ is utilized, chances are there for SOM to be further developed.

4

CONCLUSION

This thesis displays mainly two types of Recurrent Neural Network: ESN and RRBFN, as well as associated unsupervised learning techniques for improving them. Empirically, the results show that RRN, especially ESN, could successfully be applied on stock log returns. In terms of test error, the predictions ESN made prevail greatly over that of linear regression, slightly better than that of RRBFN. For details, please refer to the following tables.
For performance on the stock 'MMM':

Models	Training Error	Test Error
Linear Regression	0.00472	0.0036
ESN	0.00193	0.00136
ESN with BIP	0.00199	0.00109
RRBF	0.00290	0.00173
RRBF with SOM	0.00258	0.00163
Directional RRBF	0.00269	0.00153

For performance over 100 stocks:

Models	Training Error	Test Error
Linear Regression	0.00809	0.00799
ESN	0.00498	0.00515
ESN with BIP	0.00397	0.00459
RRBF	0.00531	0.00598
RRBF with SOM	0.00509	0.00571
Directional RRBF	0.00519	0.00532

Values in the table prove the strong capability of ESN for predicting chaotic time series. Also, all the above methods, except directional RRBF, are efficient in terms of running time. For the stock 'MMM', the error results come out almost instantly. For iterations over 100 stocks, it still takes less than 10 seconds. Moreover, 'for loop' in the previous pseudo codes could be replaced by 'vectorization' technique in Matlab to further improve efficiency.

Although in general, RRN works pretty well on denoised series, great care needs to be taken before it is actually used for trading:

- 'All models are wrong, some models are useful.' Typically, a machine learning model only works under suitable conditions and assumptions. Then it attempts to find some patterns to learn. However, unlike human behaviour, financial time series involves large proportions of noise, and hence are difficult to predict.
- The possibility of overfitting always exists. Careful tuning of parameters can only decrease chance of overestimation of returns in backtesting, but cannot get rid of it.
- The result of denoising is essential since it is supposed to retain useful information and discard irrelevant noise. However, it is often technically hard to distinguish the useful part from the rest.

At the same time, there are several points that could be done to improve the model if time allows:

- More dissections on denoising technique to increase the chance of predicting actual price instead of noise.
- More relevant inputs or features may be added(e.g. Trading Volumes).
- Discover more unsupervised adaptive algorithms such as Maximal Entropy(Lukosevicius & Jaeger, 2009, S. 137) to further improve a model.
- Utilize systematic ways of tuning parameters such as Grid Research(available in Python), or Genetic Algorithm, which is robust in heuristic search. Moreover, some parameters could be made changeable adaptively during the training process(e.g. Learning Rate in Gradient Descent or SOM). A very simple example would be using different learning rates for different dataset sizes.
- Use cross-validation to improve reliability of the results.
- Financial data used should be survivorship bias free and transaction costs should be considered when calculating log returns.

Overall, as a state of the art prediction method, RRN works pretty well on the denoised financial time series in terms of test error. It takes a big leap from linear regression and it is still fast. The above bullet points also reveal great potential of the model. No matter how many of the above points will effectively refine the model, more possibilities are always there to be exploited.

BIBLIOGRAPHY

[1] Ernie Chan. *Quantitative trading: how to build your own algorithmic trading business*, volume 430. John Wiley & Sons, 2009.

[2] Christian Emmerich, René Felix Reinhart, and Jochen Jakob Steil. Recurrence enhances the spatial encoding of static inputs in reservoir networks. In *International Conference on Artificial Neural Networks*, pages 148–153. Springer, 2010.

[3] Herbert Jaeger. *Tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the" echo state network" approach*, volume 5. GMD-Forschungszentrum Informationstechnik, 2002.

[4] Herbert Jaeger. Long short-term memory in echo state networks: Details of a simulation study. Technical report, Jacobs University Bremen, 2012.

[5] Herbert Jaeger and Harald Haas. Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication. *science*, 304(5667):78–80, 2004.

[6] Mantas Lukoševicius. On self-organizing reservoirs and their hierarchies. *Jacobs University Bremen, Tech. Rep*, (25), 2010.

[7] Mantas Lukoševičius. A practical guide to applying echo state networks. In *Neural networks: tricks of the trade*, pages 659–686. Springer, 2012.

[8] Mantas Lukoševicius. *Reservoir computing and self-organized neural hierarchies*. PhD thesis, Jacobs University Bremen, 2012.

[9] Mantas Lukoševičius and Herbert Jaeger. Reservoir computing approaches to recurrent neural network training. *Computer Science Review*, 3(3):127–149, 2009.

[10] Klaus Neumann and Jochen J Steil. Batch intrinsic plasticity for extreme learning machines. In *International Conference on Artificial Neural Networks*, pages 339–346. Springer, 2011.

[11] Hongli Niu and Jun Wang. Financial time series prediction by a random data-time effective rbf neural network. *Soft Computing*, 18(3):497–508, 2014.