Hybrid Learning of Feedforward Neural Networks for Regression Problems

by

Xing Wu

A dissertation submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
Dec 13, 2014

Keywords: Regression, Feedforward Neural Networks (FNN), Single Layer Feedforward Networks (SLFN), Fully Connected Cascade Networks (FCCN), Levenberg-Marquardt (LM) algorithm, Least Square (LS) method, Constructive algorithm, Orthogonal Least Square (OLS), Particle Swarm Optimization (PSO)

Copyright 2014 by Xing Wu

Approved by

Bogdan M. Wilamowski, Chair, Professor of Electrical and Computer Engineering
David A. Umphress, Professor of Compuater Science and Software Engineering
Michael E. Baginski, Professor of Electrical and Computer Engineering
Vishwani D. Agrawal, Professor of Electrical and Computer Engineering
Vitaly J. Vodyanoy, Professor of Anatomy, Physiology and Pharmacology

Abstract

Inspired by the structure and functional aspects of the biological neural networks, the Artificial Neural Network (ANN) is a very popular model in the machine learning fields to learn complex relationships in the data. The Feedforward Neural Networks (FNN) are the basic and most common type of ANN used in the supervised learning area. The research of the FNN consists of two major issues: architecture selection and learning.

The architecture of the FNN mainly includes Multilayer Perceptron (MLP) and Bridged Multilayer Perceptron (BMLP). As the simplest MLP with only one hidden layer, the Single Layer Feedforward Neural Networks (SLFN) had attracted much attention among the shallow models. When a BMLP has all the bridge connections, it becomes the special deep narrow architecture, called Fully Connected Cascade Networks (FCCN), which had been widely applied in different fields since it was proposed.

In this dissertation, we explored the learning of these two special types of FNN in details for regression problems. When applied to the regression problems, the output neuron of the FNN is usually set with linear activation function. With this character, the SLFN and FCCN architectures have much in common and their most learning algorithms could share to each other. The FCCN could be viewed as a SLFN with nest connections in the single hidden layer. Because the output neuron is linear, all the output parameters (weights) are linear related. Taking advantage of this relationship, a new hybrid learning algorithm is proposed for these two types of FNN by combining the efficient Levenberg-Marquardt (LM) algorithm and Least Square (LS) method.

In order to search the optimal network size, the hybrid algorithm is extended to the construction scheme. Two hybrid constructive algorithms are proposed for the SLFN and FCCN learning, namely HC1 and HC2 algorithm. The HC1 algorithm constructs the SLFN

or FCCN by adding randomly initialized hidden neuron one by one, each time the preceding hybrid algorithm is carried out to train the entire network. The HC2 algorithm can be considered as an enhanced version of HC1. Each time adding the new neuron, its initial parameters are picked in a more sophisticated way. Similar to the Orthogonal Least Square (OLS) algorithm, a contribution objective function of the new neuron is derived. The Particle Swarm Optimization (PSO) is cooperated to search the best set of parameters leading to the biggest contribution.

Both the HC1 and HC2 algorithms are practiced on several classical function approximation benchmarks for SLFN and FCCN construction. The experiment results illustrated the proposed hybrid constructive strategies can obtain more compact networks with good generalization ability compared with other popular learning algorithms.

Acknowledgments

First of all, I'd like to thank to Auburn University and the ECE department providing the good environment for study and research. I also appreciate all my committee members having interest to my research topics and their patience to give me suggestions. Most importantly, I want to express my sincere thanks to my advisor, Professor Wilamowski, who gave me the opportunity for the PhD study. It's him who took me into the magic realm of the neural networks and taught me how to solve problems independently. His persistence in the research has profound influence on me during my PhD study.

I also want to thank to my parents, who always support me when I have difficulties. I'd like to thank all my friends, who shared happiness and unhappiness with me and helped me a lot in my oversea life.

Table of Contents

Abstract ii

Acknowledgments iv

List of Figures viii

List of Tables x

1 Introduction 1

1.1 Artificial Neural Networks Background 1

1.2 Feedforward Neural Networks 3

1.3 Problem Formulation 6

1.4 Contribution 8

1.5 Organization 9

2 General Gradient Learning Algorithms 10

2.1 Error Backpropagation 12

2.2 Rprop 15

2.3 Conjugate gradient 16

2.4 Quickprop 17

2.5 Levenberg-Marquardt algorithm 17

2.6 Quasi-Newton Method (BFGS algorithm) 20

2.7 Neuron by Neuron algorithm 20

2.8 Forward only gradient calculation 22

3 Architecture Oriented Learning Algorithms 24

3.1 Linear Least Square Method 24

3.2 Kwok's methods 25

3.3 Extreme Learning Machine 27

3.3.1 Incremental ELM (I-ELM) 28
3.3.2 Enhanced Incremental ELM (EI-ELM) 29
3.3.3 Convex Incremental ELM (CI-ELM) 29
3.3.4 Error Minimized ELM (EM-ELM) 30
3.4 Support Vector Regression 31
3.5 Cascade Correlation Algorithm 35
3.6 Cascade2 Algorithm 36
3.7 Orthogonal Least Square Algorithm 36
3.8 Casper Algorithm 39
4 Hybrid Algorithm 42
4.1 Review Conventional LM Algorithm 43
4.2 Derive New Update Formula for SLFN 43
4.3 Regularization 48
4.4 New Update Formula for FCCN 50
5 Extend Hybrid Algorithm for Construction 53
5.1 When to add new neuron 53
5.2 How to add new neuron 54
5.2.1 Variables in Update Formula 55
5.2.2 Reformulate OLS algorithm 57
5.2.3 Particle Swarm Optimization 61
5.3 When to stop adding 65
6 Experiments 67
6.1 Experiments on the HC1 Algorithm for SLFN Construction 67
6.1.1 SinE function 67
6.1.2 Peaks Function Approximation 69
6.1.3 UCI real life problems 71
6.2 Experiments on the HC2 Algorithm for FCCN Construction 73

6.2.1 2D Function Approximation 74
6.2.2 Mackey-Glass Time Series Prediction 78
7 Conclusions and Discussion 83
Bibliography 85
A Publications of the Author 91
B MATLAB Programs for the Proposed Algorithms 93
B.1 File List 93
B.2 Matlab Codes 95

List of Figures

1.1 two types of Artificial Neural Networks 2

1.2 MLP and its special case SLFN 3

1.3 BMLP and its special case FCCN 4

1.4 sigmoid function and tanh function 5

1.5 1D example of RBF with different widths 6

2.1 notations for SLFN and FCCN architecture 11

2.2 propagation through single neuron 12

2.3 Pseudocode of the Rprop algorithm 15

2.4 Core idea behind NBN algorithm 21

2.5 Using forward only strategy on a FCCN with 4 neurons 22

3.1 SLFN construction 26

3.2 Explanation of the SVM and SVR 32

3.3 SARPROP 40

3.4 3 groups parameters in Casper 41

4.1 Pseudocode of Hybrid Algorithm For Fixed Sized SLFN 49

5.1 Construct sigmoid SLFN for 1-dimension approximation 62

5.2 Selection of the New Neuron with PSO 65

6.1 Training and testing data set for SinE function approximation 68

6.2 One example trial while using HC1 algorithm to approximate SinE function. (a) shows the decreasing error (RMSE) during the 1000 iterations, vertical lines are the moment to add new hidden neuron. (this trial ended with 17 hidden neurons and training RMSE=0.1302, testing RMSE=0.1234) (b) shows the approximation result of this trial. 68

6.3 Peaks function 70

6.4 meshplot of the five 2D functions for approximation 75

6.5 Averaged Testing FVU and Training Time While Approximating Function #1 . 78

6.6 Averaged Testing FVU and Training Time While Approximating Function #2 . 78

6.7 Averaged Testing FVU and Training Time While Approximating Function #3 . 79

6.8 Averaged Testing FVU and Training Time While Approximating Function #4 . 79

6.9 Averaged Testing FVU and Training Time While Approximating Function #5 . 80

6.10 Generalization performance for Mackey-Glass time series prediction 81

6.11 Best prediction results obtained by HCL among 20 trials: a FCCN with 16 hidden neurons, short term NRMS=0.0212, long term NRMS=0.0707. 81

List of Tables

2.1 δ table 23

4.1 vector version of the δ table 51

6.1 Experiment Results Comparisons for SinE Function Approximation 69

6.2 Experiment Results Comparisons for Peaks Function Approximation 71

6.3 UCI data sets specifications 72

6.4 Testing RMSE and the Corresponding Standard Deviation(std) Comparison While Approximating UCI data sets 73

6.5 Optimal Number of Hidden Neurons and the Average Training Time While Approximating UCI data sets 73

B.1 attributes of "param" in "construct.m" 95

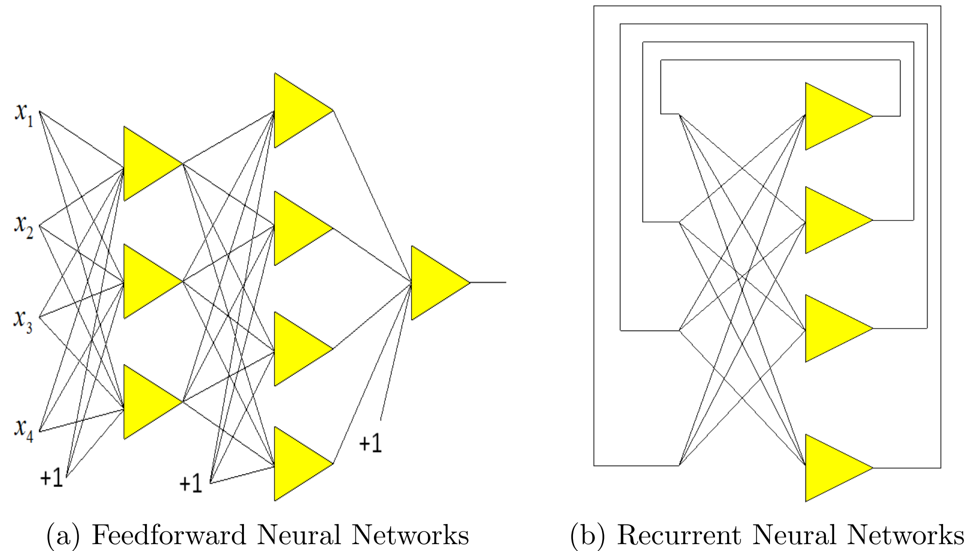
Chapter 1

Introduction

1.1 Artificial Neural Networks Background

Artificial neural networks (ANN) are computational models inspired by animals' central nervous systems (in particular the brain) which is capable of machine learning and pattern recognition. In the ANN, simple artificial neurons are connected together to form a network which mimics a biological neural networks. Modern neural networks are non-linear statistical data modeling tools. They are usually used to model complex relationships between inputs and outputs, to find patterns in data, or to capture the statistical structure in an unknown joint probability distribution between observed variables.

The first ANN model was created by Warren McCulloch and Walter Pitts with electronic circuits in 1943[1]. In 1949, Donald Hebb published the book *The Organization of Behavior*, which outlined a law for synaptic neuron learning[2]. This law, called Hebb's rule, is one of the simplest and most straight-forward learning rules for artificial neural networks. In 1959, Bernard Widrow and his PhD student Ted Hoff developed models called "ADALINE" (short for ADAptive LINear Elements) and "MADALINE" (short for Multiple ADALINE) in Stanford. They also invented the popular Least Mean Square (LMS) filter to adapt the model parameters. ADALINE was developed to recognize binary patterns so that if it was reading streaming bits from a phone line, it could predict the next bit. MADALINE was the first neural networks applied to the real world problem, using an adaptive filter that eliminates echoes on phone line. In 1969, Marvin Minsky and Seymour Papert published a precise mathematical analysis of the perceptron to show that the perceptron model was not capable of representing many important problems, like XOR problems[3], which lead to a silence period of the ANN research. The research was reconstructed slowly in 1970s and



(a) Feedforward Neural Networks (b) Recurrent Neural Networks

Figure 1.1: two types of Artificial Neural Networks

1980s. In 1982, Teuvo Kohonen looked into the mechanism involving self-organization in the brain and proposed the self-organizing maps (SOM)[4]. In 1974, Paul Werbos developed a learning procedure called *backpropagation of error*. But it was not until one decade later in 1986[8], the error backpropagation (EBP) algorithm became popular and important in the later ANN research. In the 1990s, neural networks were partially overtaken in popularity in machine learning by support vector machine (SVM) and other simpler methods. However, the renewed interest in ANN was sparked in 2000s by the success of deep learning[5]-[7].

The ANN consists of some interconnected "neurons", each of which can compute values from inputs, has adaptive weights (or parameters), and can represent nonlinear feature of the inputs. According to the connection topology, ANN can be divided into two main categories: Feedforward Neural Networks (FNN) and Recurrent Neural Networks (RNN), as shown in Figure 1.1. The RNN has some feedbacks to form directed cycles in the networks while the FNN doesn't. The RNN has temporal memories and is able to learn the sequences. However it's more difficult to train than the FNN.

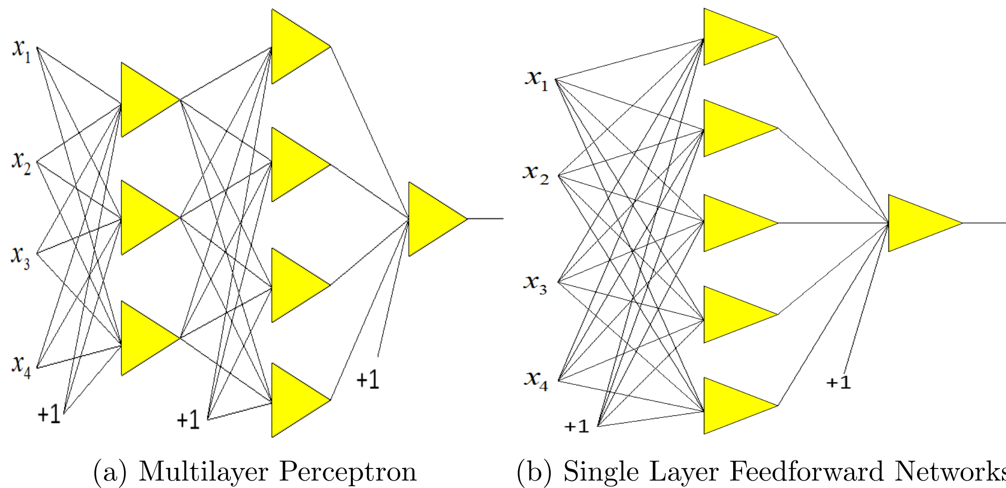
The ANNs are widely used in many real life applications. According to the learning target, the ANN learning can be divided into supervised learning, unsupervised learning and semi-supervised learning. The application fields of ANN learning include,

1. Regression, like interpolation, time series prediction, etc.
2. Classification, like pattern recognition, fault detection, decision making, etc.
3. Data processing, like filtering, clustering, feature selection, compression, etc.
4. System identification and control.

1.2 Feedforward Neural Networks

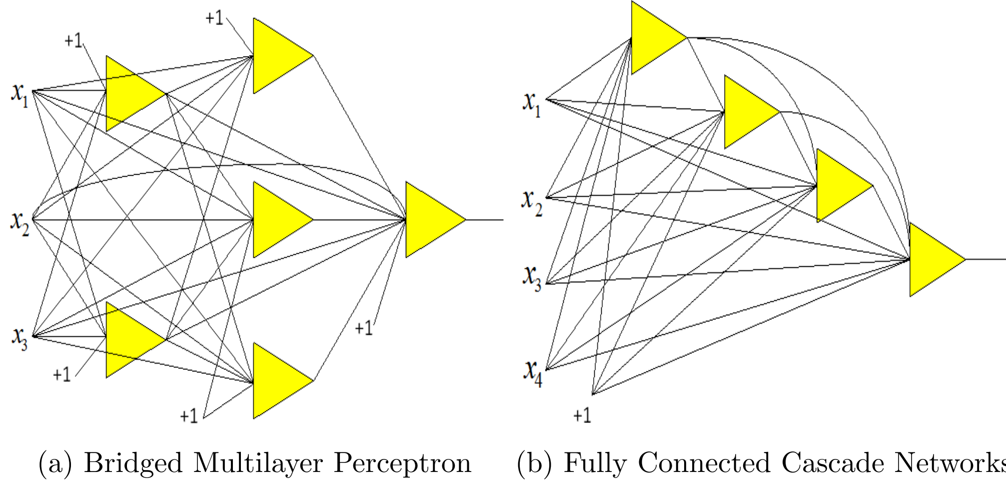
The FNN architecture has two main types: Multilayer Perceptron (MLP) and Bridged Multilayer Perceptron (BMLP), as shown in Figure 1.2 and Figure 1.3. The MLP architecture has several hidden layers between inputs and outputs, every two adjacent layers has forward connections and there's no connections between the neurons in the same layer. The Single Layer Feedforward Neural Networks (SLFN) is the simplest MLP, which has only one hidden layer. Different from the MLP, the BMLP has bridge connections across layers. The Fully Connected Cascade Networks (FCCN) is the special case of the BMLP, in which each layer has only one neuron.

While learning with the FNN model, one of the main task is to determine the network size. Observe the above architectures, a general MLP and BMLP have more structure



(a) Multilayer Perceptron (b) Single Layer Feedforward Networks

Figure 1.2: MLP and its special case SLFN



(a) Bridged Multilayer Perceptron (b) Fully Connected Cascade Networks

Figure 1.3: BMLP and its special case FCCN

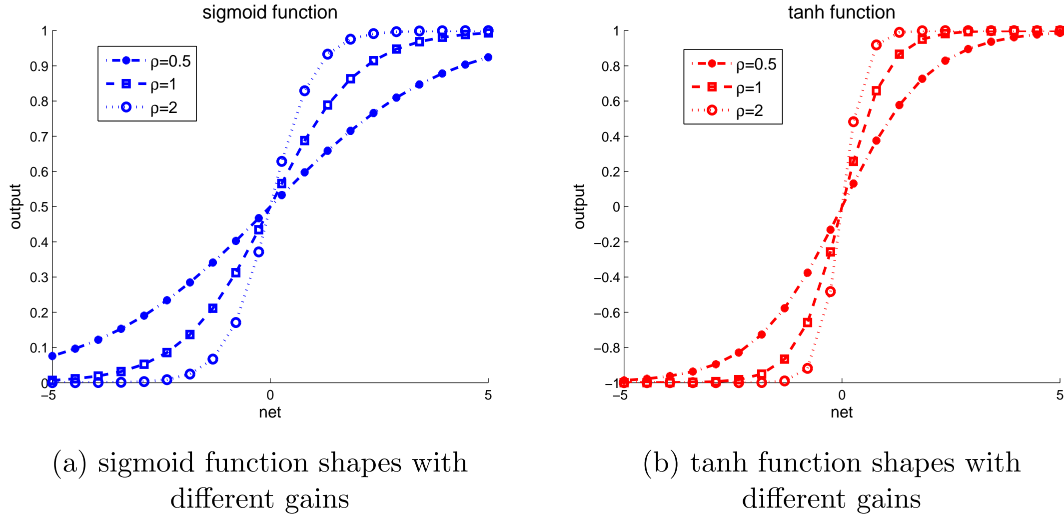
parameters. One has to determine the how many layers (the depth) and how many neurons in each layer (the width). It's quite difficult and complicated to find the optimal architecture parameters for them. Compared to the MLP and BMLP, the SLFN and FCCN are relatively easy to learn the structure. Since the SLFN has only one hidden layer, the depth is fixed as 1, one only needs to determine the width of this layer (the number of neurons). Similarly, the FCCN has only one hidden neuron in each layer, the width is fixed as 1, one only needs to determine the depth (the number of neurons). In this paper, we mainly focused on the learning algorithms of these two architectures.

The activation function of the neuron in the FNN could be linear or nonlinear. The neuron with linear activation function behaves like the summator, whose outputs the weighted summation of its inputs. The popularly used nonlinear activation functions include sigmoid functions and Radial Basis Functions (RBF). The output of a sigmoid neuron can be calculated as,

$$h(\mathbf{x}) = \frac{1}{1 + e^{-\rho \times \text{net}}} \tag{1.1}$$

in which, net is the weighted summation of the inputs,

$$\text{net} = w_0 + \sum_{d=1}^{D} x_d w_d \tag{1.2}$$



(a) sigmoid function shapes with different gains

(b) tanh function shapes with different gains

Figure 1.4: sigmoid function and tanh function

$\mathbf{x} = [x_1, x_2, ..., x_D]$ is the D-dimension input vector, $\mathbf{w} = [w_0, w_1, w_2, ..., w_D]$ are the corresponding weights (including bias w_0), ρ is gain of the neuron. The adaptive input weights $\mathbf{w}$ will be tuned during training. The gain ρ is the scale factor of the weights $\mathbf{w}$ and usually not considered for tuning. The output of the sigmoid function in (1.1) ranges in [0,1]. In some literature, its scaled format which ranges in [-1,1] is similarly used.

$$h(\mathbf{x}) = \tanh(\rho \times \text{net}) = \frac{2}{1 + e^{-2\rho \times \text{net}}} - 1 \tag{1.3}$$

Figure 1.4 shows some examples of the sigmoid function and tanh function shape respect to net value.

The RBF network uses the activation function whose output only depends on the distance from the input to the center. The most popularly used is the gaussian function,

$$h(\mathbf{x}) = \exp(-\frac{\|\mathbf{x} - \mathbf{c}\|^2}{\sigma^2}) \tag{1.4}$$

in which, $\|\bullet\|$ represents Euclidean distance. The tunable parameters are center $\mathbf{c}$ and width σ. Figure 1.5 shows 1D shape of the RBF with different widths.

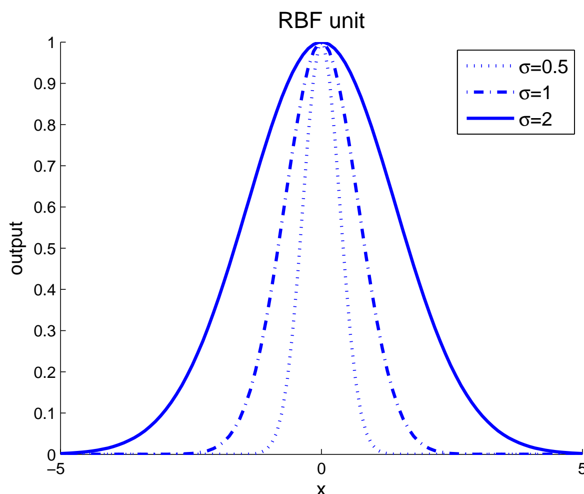


Figure 1.5: 1D example of RBF with different widths

1.3 Problem Formulation

The function approximation is one of the major branches in the supervised learning research. In general, a function approximation problem asks us to select a function among a well-defined class that closely matches an unknown target function. The problem can be formulated as following.

Given the training data set $\{(\mathbf{x}_p, y_p)|\mathbf{x}_p \in R^D, y_p \in R, p = 1, 2, ..., P\}$, in which there are P training patterns with D-dimension input and scalar output, $(\mathbf{x}_p, y_p)$ denotes the p_{th} input and output. For simplicity, we only consider a single function to be mapped from the multi-dimensional inputs (the desired output is a scalar). For the problem with multiple outputs, one can split it into several independent single output approximation problems. After selecting the FNN architecture as *prior*, one has to search the optimal structure (e.g. number of neurons) and the optimal set of the parameters (e.g. the weights) of the FNN, so that the obtained FNN could approximate those data to a desired accuracy. The usual procedure to train a FNN is to optimize those parameters to minimize some cost function.

According to the value of the desired outputs y_p, the function approximation can be divided into classification and regression. For the classification problems, the desired outputs are usually discrete values (e.g. labels). For the regression problems, the desired outputs

are continuous values. These two problems normally use different strategies during training. The FNN for classification usually uses neuron with sigmoid activation function as the last neuron. So the outputs are in the range [0,1] and they are considered as the probability of each pattern belonging to each category. The classification problems usually use the cross entropy function from the information theory as the cost function for optimization. For the regression problems, the FNN normally uses the linear neuron as the last neuron so that the outputs can be any continuous range. The outputs are interpreted directly as the outputs of the model instead of the probability. The cost function to be optimized in regression problems is simply the differences between the actual outputs and the desired outputs. In this dissertation, we mainly focused on the FNN learning for regression problems. A common used cost function in regression is the sum squared error (SSE),

$$C = \sum_{p=1}^{P} e_p^2 = \sum_{p=1}^{P} (y_p - \tilde{y_p})^2 \tag{1.5}$$

where, $\tilde{y_p}$, e_p are the actual output and the error with respect to the p_{th} training pattern.

$$e_p = y_p - \tilde{y}_p \tag{1.6}$$

For convenience, we also use the vectors $\tilde{\mathbf{y}} = [\tilde{y}_1, \tilde{y}_2, ..., \tilde{y}_P]^T$, $\mathbf{e} = [e_1, e_2, ..., e_P]^T$ in the rest of the paper. Except the SSE, some literatures also used its variants, like Mean Square Error(MSE), Root Mean Square Error(RMSE), etc. to evaluate the approximation quality.

Though the training procedure is to approximate those training data, the core objective of the learning task is to generalize from its experience. The generalization in this context is the ability of a learning machine to perform accurately on new, unseen examples or patterns after having experienced the training data set. To examine the generalization capability of the learned FNN, it's usually validated on another data set, called testing data set, which is different from the training set, given as $\{(\mathbf{x}_p^t, y_p^t) | \mathbf{x}_p^t \in R^D, y_p^t \in R, p = 1, 2, ..., P_t\}$.

1.4 Contribution

In this dissertation, we mainly focused on the FNN learning, SLFN and FCCN architectures in specific, for regression problems. Several current popularly used learning algorithms are explored and analyzed in details. Then a series of new hybrid learning algorithms for these two architectures are proposed. The dissertation has the following contributions:

1. A new hybrid algorithm for fixed size SLFN and FCCN is proposed, which combines the LM algorithm, one of the most efficient second order algorithm and the LS method. By converting the output weights to the dependent variables of the hidden parameters, the LM optimization is simplified.

2. The hybrid algorithm is extended to the constructive scheme to determine the network size simultaneously, namely HC1 algorithm. Each time when previous training entrapped into local minima, a new randomly initialized new neuron is added to the network. Then the entire network is trained again by the hybrid algorithm.

3. In order to enhance the HC1 algorithm to achieve more compact network, the HC2 algorithm is proposed. Instead of initializing each new neuron randomly, the HC2 picks the new neuron's parameters in a more sophisticated way. Similar to the Orthogonal Least Square (OLS) algorithm, a contribution objective function of the new neuron is derived. The Particle Swarm Optimization (PSO) is used to search the optimal parameters leading to the biggest contribution.

4. Both the HC1 and HC2 are practiced on some classic regression benchmarks and compared with other popularly used learning algorithms. The experiments demonstrated the HC1 and HC2 algorithms worked efficiently to obtain compact SLFN or FCCN with good generalization ability.

1.5 Organization

The rest part of the dissertation is organized as following.

In the chapter 2, some general gradient learning algorithms of the FNN are introduced. These algorithms are all based on the error backpropagation (EBP) and they can be used to train the fixed size FNN with different architectures. Most of them are still popularly used in SLFN and FCCN learning, like Quickprop, Rprop, Conjugate gradient, BFGS algorithm, LM algorithm, NBN algorithm, etc.

In the chapter 3, several architecture oriented learning algorithms of SLFN and FCCN are introduced. These algorithms are designed specifically for the SLFN or FCCN architecture. Since the SLFN and FCCN are similar in architecture, most of their architecture oriented learning algorithms can share to each other.

In the chapter 4, a hybrid algorithm for fixed size SLFN and FCCN learning is proposed in specific for the regression problem. So it's also architecture and problem oriented algorithm. By taking advantage of the linear relation of the output weights, the LS method is embedded into the LM algorithm to improve the efficiency.

In the chapter 5, the hybrid algorithm is extended to the construction scheme. Two hybrid constructive algorithms are proposed, namely HC1 and HC2 algorithm. The HC1 algorithm constructs the SLFN or FCCN by adding randomly initialized hidden neurons one by one. Each time after adding new neuron, the entire network is trained with the preceding hybrid algorithm. The HC2 combined the OLS and PSO algorithm to pick the optimal initial parameters of the new neuron sophistically.

In the chapter 6, several regression benchmarks are carried out to test the proposed hybrid algorithms for SLFN and FCCN construction. The experiment results are compared with other popular learning algorithms. The efficiency and advantages of the proposed algorithms are analyzed.

In the chapter 7, the conclusion is given and the future research is discussed.

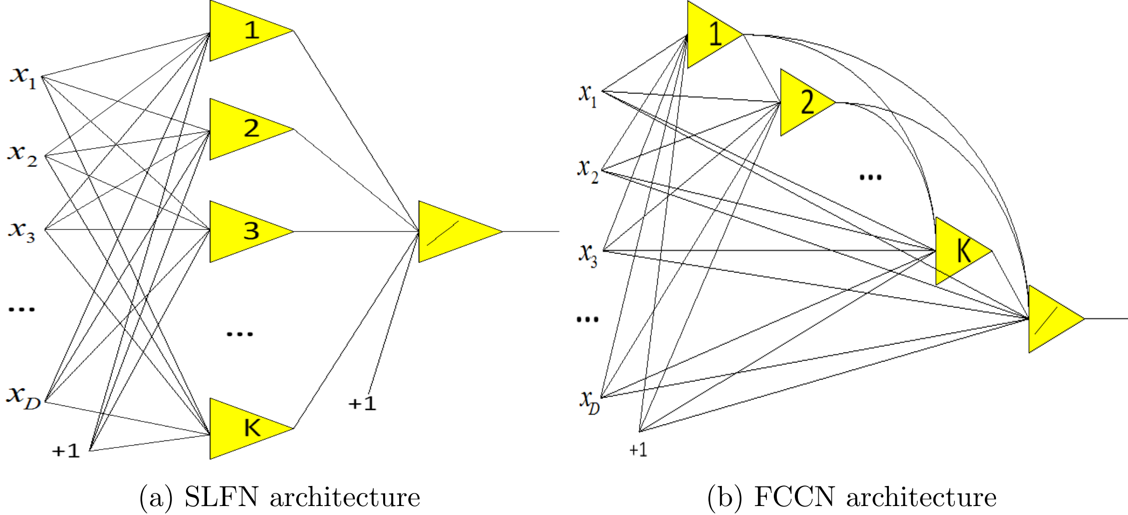
Chapter 2

General Gradient Learning Algorithms

The learning of the FNN consists of two main tasks: determining the optimal network size and tuning the network parameters (weights). In the neural networks research history, the trial and error approach was usually used to determine the network size. As a result, the second task, to tune the network parameters, had attracted much more attentions. In this chapter, we mainly discussed those popular parameters tuning algorithms for fixed size FNN.

Since the popularity of the Error Backpropagation (EBP), a lot of well-known parameters tuning algorithms for FNN had been proposed. Most of these algorithms are based on the EBP and used gradient optimization to tune the network parameters. These algorithms can be generally divided into two categories according to their update schedule: online learning and offline learning. The online learning, also called stochastic learning, is the group of algorithms that update the parameters every time passing one training pattern. In the contrast, the offline learning, also called batch learning, requires to pass through all the data set to calculate the averaged gradient each time before updating the parameters. While using the online learning, one doesn't need to prepare all the data set before training. As a result, many industrial applications preferred the online learning. Another advantages of the online learning is that it's more suitable for big data learning, since offline learning usually needs very large storage space and computing units for big data. However, because the online learning only used single pattern's information, the gradient is noisy and the parameters may not move precisely down along the gradient direction in each step.

In this dissertation, we mainly focused on the offline learning algorithms. In this chapter, we will analyze some popularly used offline learning algorithms, like the Quickprop[16],



(a) SLFN architecture (b) FCCN architecture

Figure 2.1: notations for SLFN and FCCN architecture

Rprop[17], Conjugate gradient (CG)[18], Levenberg-Marquardt (LM) algorithm[19, 20, 21], BFGS algorithm[22], NBN algorithm[23], etc. All these algorithms are general learning algorithms for the FNN and they could be used for most FNN architectures.

Before introducing these algorithms, we declared some common notations in the FNN. Because we mainly focused on the SLFN or FCCN, whose depth or width is fixed as 1, we don't specify the layer of each neuron in the FNN. Instead, we only index the hidden neurons as $k = 1, 2, ..., K$. The FNN has K hidden neurons and the output neuron is always a linear summator, as shown in Figure 2.1.

For the k_{th} hidden neuron, the outputs for the P patterns are $\mathbf{h}_k = [h_1, h_2, ..., h_P]^T$. The parameters of this neuron are denoted as $\mathbf{w}_k = [w_{k,1}, w_{k,2}, ..., w_{k,m}]$, where there are m parameters. For the neuron with sigmoid activation function, $\mathbf{w}_k$ are the input weights (including the bias); for the neuron with RBF, $\mathbf{w}_k$ are the center and width of this neuron. For the SLFN, the number of parameters (m) of each neuron is the same. For the FCCN, the number of parameters (m) increases as the depth of the neuron increases. For the k_{th} hidden neuron in the FCCN, its parameters will be $\mathbf{w}_k \in R^{D+k}$, where D is the number of inputs. For convenience, we denote the parameters of the output neuron (output weights) as $\mathbf{w}_{K+1}$. For the SLFN, $\mathbf{w}_{K+1} \in R^{K+1}$; For the FCCN, $\mathbf{w}_{K+1} \in R^{D+K+1}$. Denote $\boldsymbol{\psi}$ as a

vector including all the tunable parameters. Assume there are M parameters to be tuned in total so that the length of $\boldsymbol{\psi}$ is M. For the SLFN with K hidden neurons and each hidden neuron has m parameters, $M = Km + K + 1$; For the FCCN with K hidden neurons and D inputs, $M = (K+1)D + \frac{(K+1)(K+2)}{2}$. Because all the algorithms are iterative, we defined the iteration number is $t = 1, 2, ..., T$.

2.1 Error Backpropagation

While the sum squared error (SSE) in (1.5) is used as optimization cost function in the training procedure, the Error Backpropagation (EBP) uses the gradient descent method to tune the weights of the network iteratively. For a general MLP, the derivatives of the SSE respect to the weights are calculated as,

$$\frac{\partial C}{\partial w_{k,i}} = \frac{\partial \sum_{p=1}^{P}(y_p - \tilde{y}_p)^2}{\partial w_{k,i}} = 2\sum_{p=1}^{P} e_p \frac{\partial e_p}{\partial w_{k,i}} = -2\sum_{p=1}^{P} e_p \frac{\partial \tilde{y}_p}{\partial w_{k,i}} \tag{2.1}$$

so, one only needs to calculate the derivatives of the error or output respect to each parameter for every pattern. They can all be obtained through one forward and one backward propagation.

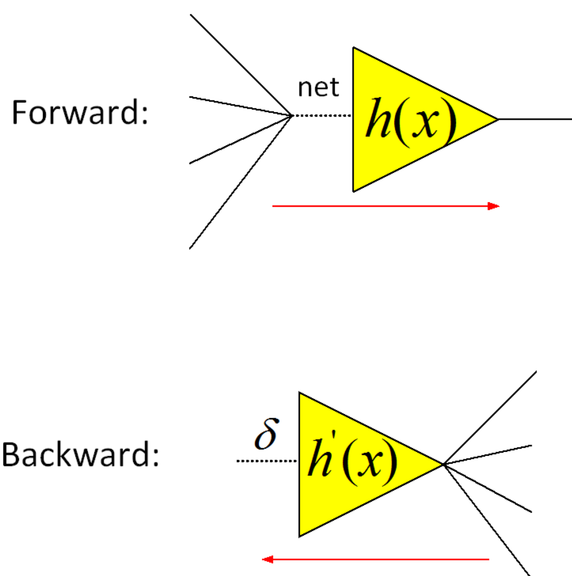


Figure 2.2: propagation through single neuron

For a single neuron, the propagation is shown in Figure 2.2. When calculating forward through the FNN, each neuron sums all the weighted inputs as net value and computes its output with the activation function $h(\mathbf{x})$ (sigmoid function or RBF). While propagating backward, the error propagated from output to input following the differential chain rule. We defined a delta value (δ) for each neuron, which is actually the derivative of the error respect to its net value.

$$\delta_{p,k} = \frac{\partial e_p}{\partial net_{p,k}} \tag{2.2}$$

where $\delta_{p,k}, net_{p,k}$ are delta value and net value of the k_{th} neuron for the p_{th} pattern. e_p is the network error for the p_{th} pattern.

During the backpropgation, the output neuron can calculate its delta value directly. For the internal hidden neuron k, as shown in Figure 2.2, it collects the weighted delta values from its fan-out neurons as the propagated error $e_{p,k}$, and multiplies its activation derivatives $h'(net_{p,k})$ to obtain its own delta value $\delta_{p,k}$, as shown below.

$$\delta_{p,k} = h'(net_{p,k}) \sum_{j=o_1}^{o_l} w_{k \to j} \delta p, j \tag{2.3}$$

in which, $j = o_1, o_2, ..., o_l$ are indices of this neuron's fan-out neurons, $w_{k \to j}$ is the weight connecting from the k_{th} neuron to the j_{th} neuron.

With the delta value $\delta_{p,k}$, the derivatives of the network error respect to its fan-in weights can be calculated as,

$$\frac{\partial e_p}{\partial w_{k,i}} = \frac{\partial e_p}{\partial net_{p,k}} \frac{\partial net_{p,k}}{\partial w_{k,i}} = \delta_{k,p} x_{p,i} \tag{2.4}$$

in which, $x_{p,i}$ is the i_{th} input (the network inputs or some previous neuron's output) of this neuron for the p_{th} pattern.

The error propagates backward as described above. By combining (2.1-2.4), the derivatives of the SSE respect to every weight of the network can be obtained. The EBP algorithm

update each weight with the following formula,

$$\Delta w_{k,i} = -\alpha \frac{\partial C}{\partial w_{k,i}} \tag{2.5}$$

where $\Delta w_{k,i}$ is the change of the weight $w_{k,i}$, each time $w_{k,i}$ will update by adding this change.

$$w_{k,i} = w_{k,i} + \Delta w_{k,i} \tag{2.6}$$

α is a positive constant value set by the user, called learning rate. The selection of the learning rate is important in EBP. If α is too small, the training will converge very slow; If α is too large, the training tends to oscillate. Rumelhart *et al.*[8] suggested to add a momentum term to the standard EBP method, and the change of each weight becomes,

$$\Delta w_{k,i}(t+1) = -\alpha \frac{\partial C}{\partial w_{k,i}}(t) + \eta \Delta w_{k,i}(t) \tag{2.7}$$

where the momentum parameter η is a positive constant value preset by the user. $\Delta w_{k,i}(t+1)$, $\Delta w_{k,i}(t)$ are the change of $w_{k,i}$ at the $(t+1)_{th}$ and t_{th} iteration. The addition of momentum can speed up the convergence and smooth the oscillation. Another modification of the EBP, called steepest descent, is to perform *line search* for the learning rate in each iteration. In each step, the parameters are searched as far as possible along the downhill gradient direction.

The EBP algorithm is the basis of almost all the other gradient methods for neural networks learning. The EBP worked slowly by using the constant learning rate, more sophisticated algorithms adapt the learning rate α while training or replace it with a matrix (like Hessian) to speed up the convergence.

```
1:  ∀k, i: Δ_{k,i}(0) = Δ_0, ∂C/∂w_{k,i}(0) = 0.
2:  for t ← 1 to T do                                              ▷ Iteration number
3:      for all the parameters (weights) do                        ▷ update each weight
4:          Calculate derivative of each weight ∂C/∂w_{k,i} like in EBP.
5:          if ∂C/∂w_{k,i}(t − 1) ∗ ∂C/∂w_{k,i}(t) > 0 then
6:              Δ_{k,i}(t) = min(Δ_{k,i}(t − 1) ∗ η^+, Δ_{max})
7:              Δw_{k,i}(t) = −sign(∂C/∂w_{k,i}(t)) ∗ Δ_{k,i}(t)
8:              w_{k,i}(t + 1) = w_{k,i}(t) + Δw_{k,i}(t)
9:          else if ∂C/∂w_{k,i}(t − 1) ∗ ∂C/∂w_{k,i}(t) < 0 then
10:             Δ_{k,i} = max(Δ_{k,i}(t − 1) ∗ η^−, Δ_{min})
11:             w_{k,i}(t + 1) = w_{k,i}(t) − Δw_{k,i}(t − 1)
12:             ∂C/∂w_{k,i} = 0
13:         else if ∂C/∂w_{k,i}(t − 1) ∗ ∂C/∂w_{k,i}(t) = 0 then
14:             Δw_{k,i}(t) = −sign(∂C/∂w_{k,i}(t)) ∗ Δ_{k,i}(t)
15:             w_{k,i}(t + 1) = w_{k,i}(t) + Δw_{k,i}(t)
16:         end if
17:     end for
18: end for
```

Figure 2.3: Pseudocode of the Rprop algorithm

2.2 Rprop

Resilient backpropagation (Rprop)[17] is a heuristic first order algorithm for FNN learning. It was created by Martin Riedmiller and Heinrich Braun in 1992. The Rprop algorithm only considers the sign of the derivatives of the cost function respect to each weight and update them independently. In the Rprop, the update step size (always nonnegative) of each weight is defined as $\Delta_{k,i}$, which has a upper bound (Δ_{max}) and lower bound (Δ_{min}). The $\Delta_{k,i}$ is adapted according to the derivative sign by multiplying two scale factors $\eta^+ > 1$ and $0 < \eta^- < 1$. Empirically, η^+ and η^- are set as 1.2 and 0.5 respectively. A typical procedure of the Rprop algorithm is shown in Figure 2.3.

The Rprop is one of the most efficient first order learning algorithms for FNN. Because only the signs of the derivatives are used for the parameters update, the Rprop is efficient respect to both time and storage consumption. It's also much less sensitive to the "gradient vanishing" problem of the deep architectures compared to the standard EBP algorithm.

2.3 Conjugate gradient

While optimizing the parameters, each update step would cost some computation. One doesn't want to destroy the preceding update during the next update step. According to the conjugate gradient method[18], every two successive update directions are "conjugate", which is defined as,

$$\mathbf{d}(t)\mathbf{H}\mathbf{d}(t+1) = 0 \tag{2.8}$$

where $\mathbf{d}(t), \mathbf{d}(t+1)$ are search direction in step t and $t+1$. $\mathbf{H}$ is the Hessian matrix, containing the second order derivatives information of the all the weights. In order to fullfill (2.8), each new step's search direction is set as following in the conjugate gradient,

$$\mathbf{d}(t+1) = -\boldsymbol{\nabla} C(t+1) + \beta * \mathbf{d}(t) \tag{2.9}$$

where $\boldsymbol{\nabla} C(t+1)$ is the gradient of the SSE with respect to all the weights in the $(t+1)_{th}$ iteration. The parameter β is computed according to one of the following rules,

1. Fletcher-Reeves: $\beta = \dfrac{\boldsymbol{\nabla}^T C(t+1)\boldsymbol{\nabla} C(t+1)}{\boldsymbol{\nabla}^T C(t)\boldsymbol{\nabla} C(t)}$

2. Polak-Ribiere: $\beta = \dfrac{\boldsymbol{\nabla}^T C(t+1)(\boldsymbol{\nabla} C(t+1) - \boldsymbol{\nabla} C(t))}{\boldsymbol{\nabla}^T C(t)\boldsymbol{\nabla} C(t)}$

3. Hestenes-Stiefel: $\beta = -\dfrac{\boldsymbol{\nabla}^T C(t+1)(\boldsymbol{\nabla} C(t+1) - \boldsymbol{\nabla} C(t))}{\mathbf{d}^T(t)(\boldsymbol{\nabla} C(t+1) - \boldsymbol{\nabla} C(t))}$

4. Dai-Yuan: $\beta = \dfrac{\boldsymbol{\nabla}^T C(t+1)\boldsymbol{\nabla} C(t+1)}{\mathbf{d}^T(t)(\boldsymbol{\nabla} C(t+1) - \boldsymbol{\nabla} C(t))}$

The conjugate gradient method searches along the steepest gradient direction in the first iteration, and searches along the conjugate direction in the rest steps. Following shows the optimization procedure in the t_{th} iteration,

1. Calculate the steepest gradient direction $\boldsymbol{\nabla} C(t)$.

2. Compute parameter β with one of the above 4 formulas.

3. Update the conjugate direction $\mathbf{d}(t)$ with (2.9).

4. Perform a line search: optimize $\alpha = \arg\min_{\alpha}\{\mathrm{SSE}(\boldsymbol{\psi}(t) + \alpha\mathbf{d}(t))\}$.

5. Update all the parameters: $\boldsymbol{\psi}(t+1) = \boldsymbol{\psi}(t) + \alpha\mathbf{d(t)}$

2.4 Quickprop

Quickprop algorithm[16] is a simple local adaptive second order learning algorithm invented by Fahlman. In Quickprop, there're two bold assumption: Firstly, each weight is independent to each other. Secondly, it assumes the error function respect to each weight is a parabola whose arms are opened upward. With these assumptions, each weight is updated locally with the following formula, which is same as the local Newton's method,

$$\Delta w_{k,i}(t) = \frac{\frac{\partial C}{\partial w_{k,i}}(t)}{\frac{\partial C}{\partial w_{k,i}}(t-1) - \frac{\partial C}{\partial w_{k,i}}(t)}\Delta w_{k,i}(t-1) \tag{2.10}$$

Since bold assumption was made while obtaining the simple update formula (2.10), the situations violating the assumption are necessary to be considered. For example, if the current slope is in the same direction as the previous slope and is the same size or large in magnitude, the parabola's arm is obviously not opened upward. When the successive two slopes are similar, the denominator will be small and the result step size will be too large. If Δw becomes 0 in some step, we have to find a way to wake the update again. To overcome these problems, Fahlman introduced a "maximum growth factor" μ (usually set as 1.75) to constrain the update step size. When the previous weight falls below a threshold, the conventional EBP update rule is taken to replace (2.10).

2.5 Levenberg-Marquardt algorithm

Due to the slow convergence of the first order gradient methods, most researchers investigated into second order algorithms. Different from the first order algorithms whose learning

rate is a scalar, the second order algorithms use a matrix to replace the scalar learning rate. Especially in Newton's method, the update formula is,

$$\boldsymbol{\Delta\psi} = \mathbf{H}^{-1}\boldsymbol{\nabla} C \tag{2.11}$$

where $\mathbf{H} \in R^{M\times M}$ (there are M parameters in total) is the hessian matrix, containing second order derivatives information,

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 C}{\partial w_{1,1}^2} & \frac{\partial^2 C}{\partial w_{1,1}\partial w_{1,2}} & \cdots & \frac{\partial^2 C}{\partial w_{1,1}\partial w_{K,m}} \\ \frac{\partial^2 C}{\partial w_{1,2}\partial w_{1,1}} & \frac{\partial^2 C}{\partial w_{1,2}^2} & \cdots & \frac{\partial^2 C}{\partial w_{1,2}\partial w_{K,m}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 C}{\partial w_{K,m}\partial w_{1,1}} & \frac{\partial^2 C}{\partial w_{K,m}\partial w_{1,2}} & \cdots & \frac{\partial^2 C}{\partial w_{K,m}^2} \end{bmatrix} \tag{2.12}$$

While it's very complex and time-consuming to calculate the exact Hessian matrix as Newton's method, Gauss-Newton method simplified it by approximating the Hessian matrix as $\mathbf{J}^T\mathbf{J}$, in which $\mathbf{J} \in R^{P\times M}$ is the Jacobian matrix. There're two types of Jacobian matrix in the literatures: one type uses the network errors as objective function, the other uses the network outputs as objective function. Following shows the Jacobian matrix using network outputs as objective function, similarly, one can get the other type by replacing the outputs $\tilde{\mathbf{y}} = [\tilde{y}_1, \tilde{y}_2, ..., \tilde{y}_P]^T$ in each numerator into $\mathbf{e} = [e_1, e_2, ..., e_P]^T$.

$$\mathbf{J} = \begin{bmatrix} \frac{\partial \tilde{y}_1}{\partial w_{1,1}} & \frac{\partial \tilde{y}_1}{\partial w_{1,2}} & \cdots & \frac{\partial \tilde{y}_1}{\partial w_{K,m}} \\ \frac{\partial \tilde{y}_2}{\partial w_{1,1}} & \frac{\partial \tilde{y}_2}{\partial w_{1,2}} & \cdots & \frac{\partial \tilde{y}_2}{\partial w_{K,m}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \tilde{y}_P}{\partial w_{1,1}} & \frac{\partial \tilde{y}_P}{\partial w_{1,2}} & \cdots & \frac{\partial \tilde{y}_P}{\partial w_{K,m}} \end{bmatrix} \tag{2.13}$$

Levenberg-Marquardt (LM) algorithm is considered a trust region modification to Gauss-Newton, which introduced a damping factor μ[19, 20]. By adjusting the damping factor, LM algorithm can switch between the Gauss-Newton algorithm and gradient descent method.

For the two different types of Jacobian matrix, the update formula of the LM algorithm has the opposite sign. For the Jacobian matrix using network errors as objective function, the update formula is,

$$\boldsymbol{\Delta\psi} = -(\mathbf{J}^T\mathbf{J} + \mu\mathbf{I})^{-1}\mathbf{J}^T\mathbf{e} \tag{2.14}$$

where $\mathbf{I}$ is the identity matrix; For the Jacobian matrix using network outputs as objective function, the update formula is,

$$\boldsymbol{\Delta\psi} = (\mathbf{J}^T\mathbf{J} + \mu\mathbf{I})^{-1}\mathbf{J}^T\mathbf{e} \tag{2.15}$$

each time, all the weights are updated as $\boldsymbol{\psi} = \boldsymbol{\psi} + \boldsymbol{\Delta\psi}$. In this paper, we use the second type with the network outputs as objective function, which is more straightforward.

The tuning of the damping factor μ plays an important role in the efficiency of LM algorithm. Given an initial guess μ_0 and scale factor $\beta > 1$, a common procedure for μ tuning can be described as:

1. compute forward and get SSE with (1.5).
2. calculate backward and get Jacobian matrix $\mathbf{J}$.
3. compute $\boldsymbol{\Delta\psi}$ with (2.15).
4. calculate forward with the parameters $\boldsymbol{\psi} + \Delta\boldsymbol{\psi}$ and get the new SSE. If the new SSE decreased, then reduce μ as μ/β, update the parameters $\boldsymbol{\psi} = \boldsymbol{\psi} + \Delta\boldsymbol{\psi}$ and go back to step 1). If the new SSE increased, then increase μ as $\beta\mu$ and go back to step 3).
5. The training stops when the norm of gradient is less than some preset value or SSE arrives the required value.

In this paper, the above tuning procedure and the suggested setting($\mu_0 = 0.01, \beta = 10$)[21] are used in the later experiments.

2.6 Quasi-Newton Method (BFGS algorithm)

Similar to the Gauss-Newton method and LM algorithm, the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm doesn't calculate the exact Hessian matrix[22]. Instead, it uses a positive definite matrix $\mathbf{B}$ to approximate the Hessian matrix and update this matrix iteratively.

Starting from the initial guess of all the weights and initial set of the matrix $\mathbf{B}$ (usually initialized as identity matrix $\mathbf{I}$). The BFGS algorithm optimize the weights iteratively, following shows the update procedure in the t_{th} iteration.

1. Obtain a direction $\mathbf{p}(t)$ by solving: $\mathbf{B}(t)\mathbf{p}(t) = -\boldsymbol{\nabla}C(t)$.

2. Perform a line search to find an acceptable step size α in the direction found in the first step, so the update step will be $\boldsymbol{\delta}(t) = \alpha\mathbf{p}(t)$, then update the weights: $\boldsymbol{\psi}(t+1) = \boldsymbol{\psi}(t) + \boldsymbol{\delta}(t)$.

3. Calculate the change of derivatives $\boldsymbol{\phi}(t) = \boldsymbol{\nabla}C(t+1) - \boldsymbol{\nabla}C(t)$.

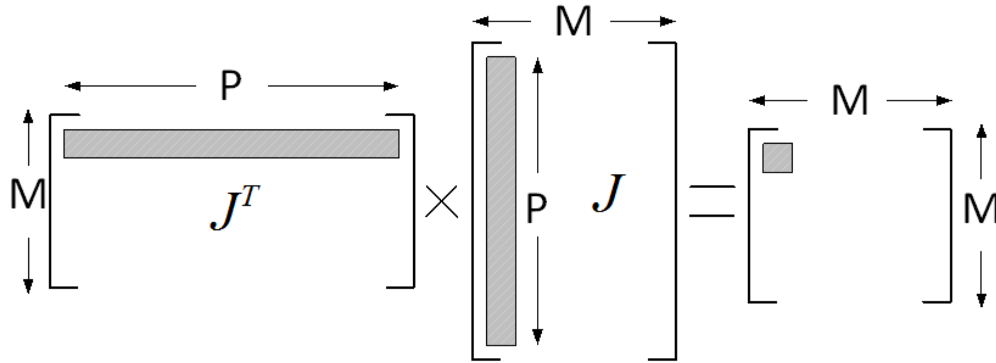
4. Update the matrix: $\mathbf{B}(t+1) = \mathbf{B}(t) + \dfrac{\boldsymbol{\phi}(t)\boldsymbol{\phi}^T(t)}{\boldsymbol{\phi}^T(t)\boldsymbol{\delta}(t)} - \dfrac{\mathbf{B}(t)\boldsymbol{\delta}(t)\boldsymbol{\delta}^T(t)\mathbf{B}(t)}{\boldsymbol{\delta}^T(t)\mathbf{B}(t)\boldsymbol{\delta}(t)}$

More conveniently, not only the Hessian matrix is approximated with matrix $\mathbf{B}$, one can also update the approximated inverse Hessian ($\mathbf{B}^{-1}$) by applying Sherman-Morrison formula. So the update formula in step 4 can also be written as,

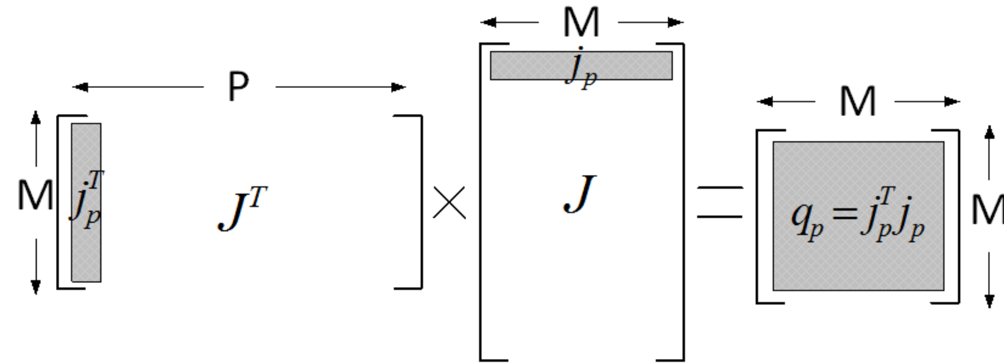
$$\mathbf{B}^{-1}(t+1) = \mathbf{B}^{-1}(t) + \left(1 + \frac{\boldsymbol{\phi}^T(t)\mathbf{B}^{-1}(t)\boldsymbol{\phi}(t)}{\boldsymbol{\delta}^T(t)\boldsymbol{\phi}}\right)\frac{\boldsymbol{\delta}(t)\boldsymbol{\delta}^T(t)}{\boldsymbol{\delta}^T(t)\boldsymbol{\phi}(t)} - \frac{\boldsymbol{\delta}(t)\boldsymbol{\phi}^T(t)\mathbf{B}^{-1}(t) + \mathbf{B}^{-1}(t)\boldsymbol{\phi}(t)\boldsymbol{\delta}^T(t)}{\boldsymbol{\delta}^T(t)\boldsymbol{\phi}(t)} \tag{2.16}$$

2.7 Neuron by Neuron algorithm

Although the LM algorithm works efficiently, one will have trouble to store the Jacobian matrix $\mathbf{J}$ while learning from a data set with large size. The Jacobian matrix is a $P \times M$ matrix, where P is the number of patterns and M is the number of tunable parameters.



(a) conventional way for Jacobian multiplication



(b) alternative way for Jacobian multiplication

Figure 2.4: Core idea behind NBN algorithm

Since modern machine learning problems usually have a large data set whose P is huge, a computer with limited memory could hardly handle the problem with the LM algorithm.

In order to overcome this deficiency, Wilamowski and Yu modified the computation convention of the matrix multiplication in $\mathbf{J}^T\mathbf{J}$ and proposed the Neuron by Neuron (NBN) algorithm[23]. Observe Figure 2.4, the conventional computation is to calculate the Jacobian matrix and stored as $\mathbf{J}$, then multiplied $\mathbf{J}^T$ and $\mathbf{J}$ to obtain the quasi Hessian matrix $\mathbf{Q}$ as shown in 2.4(a). With the NBN algorithm, instead of storing the entire Jacobian matrix $\mathbf{J}$, the multiplication is processed pattern by pattern separately. For each pattern, one stores a Jacobian vector $\mathbf{j}_p$ which is actually the p_{th} row of the Jacobian matrix. Then by multiplying each $\mathbf{j}_p^T\mathbf{j}$, we could obtain a small quasi Hessian matrix $\mathbf{q}_p$ for that pattern. Finally, the quasi Hessian matrix $\mathbf{Q} = \mathbf{J}^T\mathbf{J}$ will be the sum of all the small hessian matrices ($\sum_{p=1}^{P} \mathbf{q}_p$).

With the NBN algorithm, the space complexity is decreased. The LM algorithm could be utilized for any regression problem no matter how large the data set is.

2.8 Forward only gradient calculation

The EBP algorithm provided a strategy to calculate the derivatives of the cost function respect to every parameter in the FNN according to the differential chain rule. It works efficiently on the MLP architecture. However, for more complex architectures, like FCCN, or Arbitrary Connected Networks (ACN), the back propagation procedure becomes quite complicated. To simplify it, Wilamowski and Yu proposed a forward only strategy to calculate the gradients in ACN by using Dynamic Programming (DP)[39]. This strategy removed the backward propagation in EBP. Instead, all the required information are stored during the forward propagation. Following gives a FCCN example for the forward only computation.

Observe Figure 2.5, the FCCN has 4 neurons, whose activation functions are $h_1(\text{net})$, $h_2(\text{net})$, $h_3(\text{net})$, $h_4(\text{net})$ ($h(\bullet)$ is a function respect to each neuron's net value). In general, h_1, h_2 and h_3 are sigmoid functions, h_4 is linear function. For convenience, the weight connecting from the i_{th} neuron to the j_{th} neuron is noted as $w_{i\to j}$, as shown in Figure 2.5. All the other variables use the the preceding notation convention. Given the training pattern

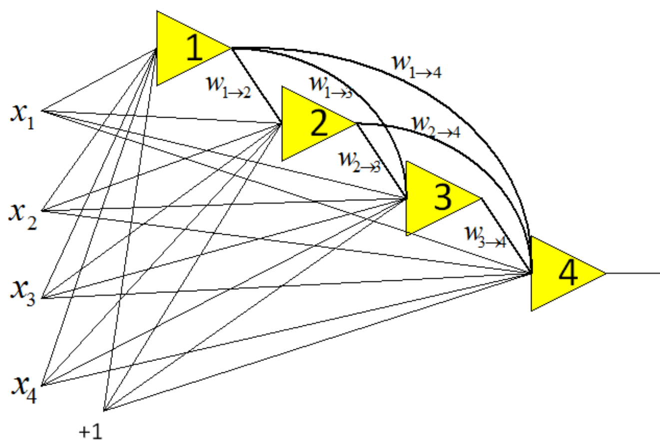


Figure 2.5: Using forward only strategy on a FCCN with 4 neurons

Table 2.1: δ table

neuron #	1	2	3	4
1	$\delta_{1,1}$			
2	$\delta_{2,1}$	$\delta_{2,2}$		
3	$\delta_{3,1}$	$\delta_{3,2}$	$\delta_{3,3}$	
4	$\delta_{4,1}$	$\delta_{4,2}$	$\delta_{4,3}$	$\delta_{4,4}$

$(\mathbf{x}_p, y_p)$, whose actual output is $\tilde{y}_p$, review (2.1), we need to calculate the derivative of the actual output respect to each parameter (weight) $\dfrac{\partial \tilde{y}_p}{\partial w_{k,i}}$.

The forward only strategy defined a general delta variable $\delta_{i,j}$, which is the derivative of the i_{th} neuron's output respect to the j_{th} neuron's net value.

$$\delta_{i,j} = \frac{\partial h_{p,i}}{\partial \text{net}_{p,j}} \qquad i \geq j \tag{2.17}$$

Then our target becomes,

$$\frac{\partial \tilde{y}_p}{\partial w_{k,i}} = \frac{\partial h_{p,4}}{\partial w_{k,i}} = \delta_{4,k} \frac{\partial \text{net}_{p,k}}{\partial w_{k,i}} \tag{2.18}$$

In the forward only strategy, a 4×4 (number of neuron) δ table is created, as shown in Table 2.1. One has to fill all the δ values in the table. The δs on the diagonal are actually the $h'(\text{net})$ of each neuron. Other δs can be obtained according the above δs in the column with the following rule,

$$\delta_{i,j} = \delta_{i,i} \sum_{m=j}^{i-1} \delta_{m,j} w_{m \to i} \tag{2.19}$$

After filling all the δs in the table, one can simply use the last row of the table to calculate $\dfrac{\partial \tilde{y}_p}{\partial w_{k,i}}$ by multiplying the δ with the corresponding input of the neuron, as in (2.18).

Chapter 3

Architecture Oriented Learning Algorithms

In Chapter 2, several well-known batch learning algorithms for the fixed size FNN are introduced. However, the other task of the FNN learning, to determine the optimal network size is still unsolved. The traditional strategy is to use trial and error approach. Try the FNN with different size, for each size, those algorithms in Chapter 2 are carried out to tune the parameters. The generalization performance is observed after each tuning. The process is repeated until the satisfactory results are obtained. Although some of those gradient learning algorithms work efficient, like LM, BFGS, etc, the trial and error approach costs much useless computation time.

Except those popular gradient learning algorithms, there are some other algorithms which are designed for some specific architectures. These algorithms take the advantage of some special characteristics of the architecture and improved the learning efficiency, we call them architecture oriented learning algorithms. Most of these algorithms not only tune the parameters, but also involve the search of the network size during the tuning process. In this chapter, we will review some of the popular algorithms specifically designed for the SLFN and FCCN. As analyzed previously, the SLFN and FCCN have much in common. So most learning algorithms of them could share to each other.

3.1 Linear Least Square Method

The least square model is a common model in statistics and optimization area. It can be generally described as,

$$\underset{\boldsymbol{\beta}}{\text{minimize}} \quad S = \sum_{i=1}^{n} (y_i - f(\mathbf{x}_i, \boldsymbol{\beta}))^2 \tag{3.1}$$

where the $\boldsymbol{\beta}$ are the parameters of the model. In fact, while solving the regression problems, the FNN is a nonlinear least square model, where $f(\bullet)$ is the output of the networks.

When the parameters of the least square model ($\boldsymbol{\beta}$) are linear related,

$$f(\mathbf{x}_i, \boldsymbol{\beta}) = \sum_{j=1}^{m} \beta_j x_{i,j} \tag{3.2}$$

it becomes the linear least square model. This becomes a typical convex optimization. The optima is unique and could be calculated in one step[26], as shown below,

$$\hat{\boldsymbol{\beta}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \tag{3.3}$$

where $\mathbf{X} = [\mathbf{x}_1; \mathbf{x}_2; ...; \mathbf{x}_n]$, $\mathbf{y} = [y_1, y_2, ..., y_n]^T$.

To our concern in this dissertation, since the SLFN and FCCN use linear activation function for the output neuron, they can be regarded as a linear least square model with respect to the output weights. As a result, the optimal output weights can always be determined as (3.3). Most of the following architecture oriented learning algorithms in this chapter are based on this LS method.

3.2 Kwok's methods

In the 1990s, Kwok and Yeung made a comprehensive review to the constructive algorithms to the FNN, especially the SLFN[24]. They also proposed a simple strategy for the SLFN construction[25]. The constructive algorithm is a very common strategy to search the network size. It starts from an empty SLFN or FCCN and then adds the hidden neurons one by one. Each time adding the new neuron, some parameters tuning is processed. Before introducing the constructive algorithm proposed by Kwok *et al*[25], we separate the parameters of the SLFN into 4 components, as shown in Figure 3.1. Assume the current SLFN has K hidden neurons, one is trying to add the $(K+1)_{th}$ hidden neuron. All the parameters while adding the new neurons could be divided into 4 parts: the hidden parameters of the

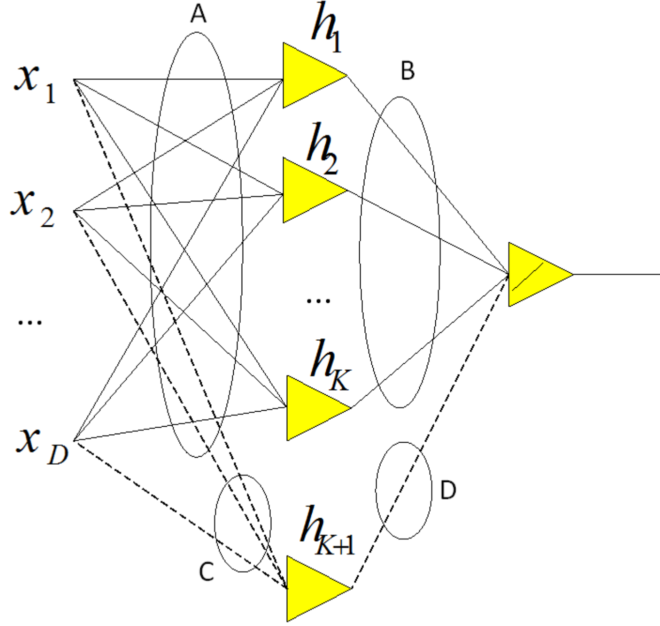


Figure 3.1: SLFN construction

previous SLFN **A**, the output weights of the previous SLFN **B**, the hidden parameters of the new added neuron **C** and the output weight corresponding to the new added neuron **D**. In the Kwok's algorithm, each time adding the new hidden neuron, the previous input parameters (**A**) are frozen. Several objective functions were proposed for optimization of the input parameters of the new neuron (**C**). Then with the optimized parameters **C**, all the output weights (**B** , **D**) are determined by pseudo inverse.

Following shows the several objective functions proposed by Kwok and Yeung [25] for optimizing hidden parameters of the new neuron (**C**). The cascade correlation objective function S_{cascor} is inspired from the cascade correlation algorithm for FCCN (will be introduced in section 3.5)[61].

1. $S_1 = \dfrac{(\mathbf{e}_K^T \mathbf{h}_{K+1})^2}{\mathbf{h}_{K+1}^T \mathbf{h}_{K+1}}$
2. $S_2 = (\mathbf{e}_K^T \mathbf{h}_{K+1})^2$
3. $S_3 = \dfrac{(\sum_{p=1}^{P}(e_{K,p} - \bar{e}_K)(h_{K+1,p} - \bar{h}_{K+1}))^2}{\sum_{p=1}^{P}(h_{K+1,p} - \bar{h}_{K+1})}$
4. $S_{cascor} = |\sum_{p=1}^{P}(e_{K,p} - \bar{e}_K)(h_{K+1,p} - \bar{h}_{K+1})|$

where $\mathbf{e}_K$ are the residual errors of the current SLFN (with K hidden neurons), $\mathbf{h}_{K+1}$ are the outputs of the new neuron, $\bar{e}_K$ and $\bar{h}_{K+1}$ are the average values of them over all the training patterns. In order to alleviate the plateau problem during the optimization, transformed version of these objective functions ($\sqrt{S_1}$, $\sqrt{S_2}$, $\sqrt{S_3}$) are also used.

The Kwok's algorithms can be summarized as following,

Start from a SLFN without hidden neurons, $K = 0$.

1. Add a randomly initialized hidden neuron, whose outputs are $\mathbf{h}_{K+1}$.

2. Using some algorithms to optimize the input parameters of this neuron ($\mathbf{C}$) with one of the above objective functions. (quickprop was used in [25])

3. With the new input parameters, recalculate the outputs of the new neuron $\mathbf{h}_{K+1}$. Combined with previous network, compute the optimal output weights by LS method: $\hat{\boldsymbol{\theta}} = (\mathbf{H}_{K+1}^T \mathbf{H}_{K+1})^{-1} \mathbf{H}_{K+1}^T \mathbf{y}$

4. $K = K + 1$. If the SSE of the new SLFN is acceptable, stop training; Otherwise, go back to step 1 to add another neuron.

3.3 Extreme Learning Machine

In 2000s, Huang *et al.*[27] proposed an Extreme Learning Machine (ELM) for SLFN learning, which attracted more and more attentions recently. The ELM is very simple to implement and the biggest advantage is the training speed is hundreds or thousands faster than other BP algorithms. The original ELM algorithm was proposed for training fixed size SLFN. In the ELM, all the input parameters are randomly generated (normally in the range [-1,1]) and there's no further tuning. One only determines the optimal output weights with the LS method in one step,

$$\boldsymbol{\theta} = \mathbf{H}^{\dagger}\mathbf{y} \tag{3.4}$$

where $\mathbf{H}^\dagger$ is the Moore-Penrose generalized inverse of the hidden matrix $\mathbf{H}$[26],

$$\mathbf{H}^\dagger = (\mathbf{H}^T\mathbf{H})^{-1}\mathbf{H}^T \tag{3.5}$$

Based on the ELM theory, many its variances were developed for SLFN construction.[28, 29, 30, 31, 32, 33, 34] Following introduces several popular constructive ELMs.

3.3.1 Incremental ELM (I-ELM)

In the Incremental ELM (I-ELM) algorithm[28], most parameters are frozen once they are generated. Observe the SLFN in Figure 3.1, each time adding the new neuron, the parameters **A** and **B** are frozen as previous values, parameters **C** are generated randomly and then frozen. Only the parameter **D** is determined with the following formula,

$$\theta_{K+1} = \frac{\mathbf{e}_K^T\mathbf{h}_{K+1}}{\mathbf{h}_{K+1}^T\mathbf{h}_{K+1}} \tag{3.6}$$

The learning procedure of the I-ELM algorithm can be summarized as following,

Starting from an empty SLFN without hidden neurons, $K = 0$, $\mathbf{e}_K = \mathbf{y}$.

1. Add a random hidden neuron, whose outputs are $\mathbf{h}_{K+1}$.

2. Calculate output weight of this new neuron with (3.6).

3. Update error $\mathbf{e}_{K+1} = \mathbf{e}_K - \theta_{K+1}\mathbf{h}_{K+1}$.

4. $K = K + 1$. If the SSE of the new SLFN is acceptable, stop training; Otherwise, go back to step 1 to add another neuron.

The convergence of the I-ELM had been proved mathematically[28]. However, since most of the parameters are not tuned once they were randomly generated, the I-ELM usually resulted in a SLFN much larger then required to arrive an acceptable error level.

3.3.2 Enhanced Incremental ELM (EI-ELM)

The Enhanced Incremental ELM (EI-ELM) improved based on the I-ELM by selecting the input parameters ($\mathbf{C}$) of each new neuron from a random candidate pool, instead of generating directly[29]. The procedure is shown below,

Set the pool size N_c, starting from an empty SLFN without hidden neurons, $K = 0$, $\mathbf{e}_K = \mathbf{y}$.

1. Randomly generate a candidate pool for the new neuron's input parameters, whose size is N_c. Then the outputs of each candidate are $\{\mathbf{h}_{K+1}^{(1)}, \mathbf{h}_{K+1}^{(2)}, ..., \mathbf{h}_{K+1}^{(N_c)}\}$.

2. For each candidate in the pool, calculate its corresponding optimal output weight with (3.6), then compute the SSE if adding this neuron. (virtually add each candidate).

3. Select the candidate which leads to maximum SSE reduction and add it to the SLFN.

4. $K = K + 1$. If the SSE of the new SLFN is acceptable, stop training; Otherwise, go back to step 1 to add another neuron.

The EI-ELM sacrifices the computation time to increase the network's compactness. While extra time cost on the selection of each neuron, the result SLFN could be more compact than I-ELM while obtaining similar approximation accuracy.

3.3.3 Convex Incremental ELM (CI-ELM)

The Convex Incremental ELM (CI-ELM) improved the I-ELM based on the convex optimization method by tuning more parameters[30]. Observe the SLFN architecture in Figure 3.1, instead of determining only parameters $\mathbf{D}$ as the I-ELM, the CI-ELM both calculates the optimal parameters $\mathbf{D}$ and rescales the previous output weights $\mathbf{B}$. The output weight of the new added neuron is determined as,

$$\theta_{K+1} = \frac{\mathbf{e}_K^T(\mathbf{e}_K - (\mathbf{y} - \mathbf{h}_{K+1}))}{(\mathbf{e}_K - (\mathbf{y} - \mathbf{h}_{K+1}))^T(\mathbf{e}_K - (\mathbf{y} - \mathbf{h}_{K+1}))} \tag{3.7}$$

all the previous output weights are rescaled as,

$$\theta_k = (1 - \theta_{K+1})\theta_k, \qquad k = 1, 2, ..., K \tag{3.8}$$

The training procedure of CI-ELM is shown below,

Starting from an empty SLFN without hidden neurons, $K = 0$, $\mathbf{e}_K = \mathbf{y}$.

1. Add a random hidden neuron, whose outputs are $\mathbf{h}_{K+1}$.

2. Compute optimal output weight of the new neuron with (3.7), rescale output weights of other hidden neurons as (3.8).

3. $K = K + 1$. If the SSE of the new SLFN is acceptable, stop training; Otherwise, go back to step 1 to add another neuron.

3.3.4 Error Minimized ELM (EM-ELM)

The Error Minimized ELM (EM-ELM) proposed by Feng *et al.*[31] can be considered to be the constructive version of the original ELM. Each time adding the random new hidden neurons, all the output weights ($\mathbf{B}$ and $\mathbf{D}$) are kept to be least square optimal. However, instead of doing the pseudo inverse repeatedly, the EM-ELM provides a recursive way to update those output weights, which works more efficient than the original ELM with trial and error.

The EM-ELM algorithm has two versions: Adding hidden neurons one by one; Adding hidden neurons batch by batch. For the second version, one doesn't know how big the batch should be and it's easy to make the network size large. Here, we only introduce the first version, adding hidden neurons one by one. In the EM-ELM algorithm, two intermediate matrices are defined,

$$\mathbf{D}_K = \frac{\mathbf{h}_{K+1}^T(\mathbf{I} - \mathbf{H}_K\mathbf{H}_K^\dagger)}{\mathbf{h}_{K+1}^T(\mathbf{I} - \mathbf{H}_K\mathbf{H}_K^\dagger)\mathbf{h}_{K+1}} \tag{3.9}$$

$$\mathbf{U}_K = \mathbf{H}_K^\dagger(\mathbf{I} - \mathbf{h}_{K+1}\mathbf{D}_K) \tag{3.10}$$

then all the output weights (including previous neurons and new neuron) are determined as,

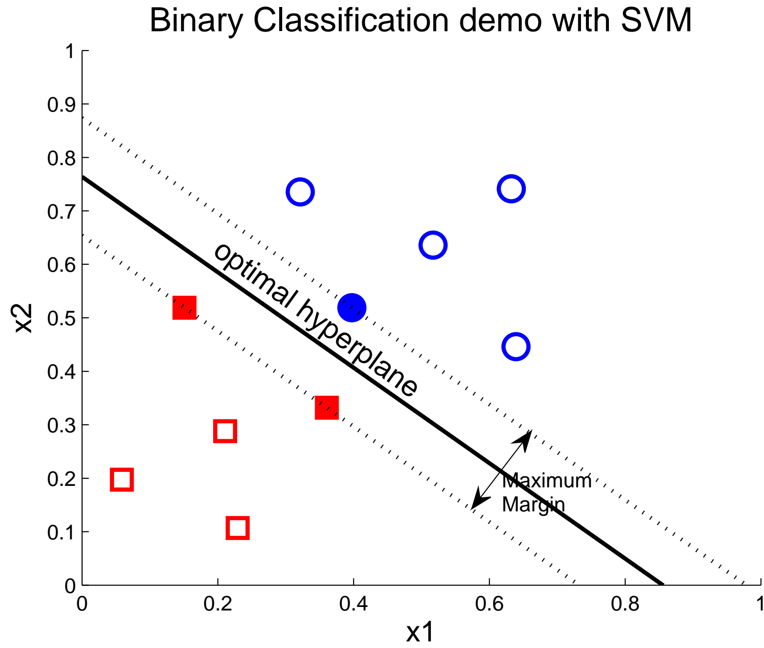
$$\boldsymbol{\theta}_{K+1} = \mathbf{H}_{K+1}^{\dagger}\mathbf{y} = \begin{bmatrix} \mathbf{U}_K \\ \mathbf{D}_K \end{bmatrix} \mathbf{y} \tag{3.11}$$

Same as the ELM, the EM-ELM always kept the output weights to be LS optimal. So it could result in much more compact SLFN than I-ELM, EI-ELM and CI-ELM while obtaining similar approximation accuracy. Because it provides a recursive way to update those optimal output weights, the EM-ELM works more efficiently than the direct ELM.

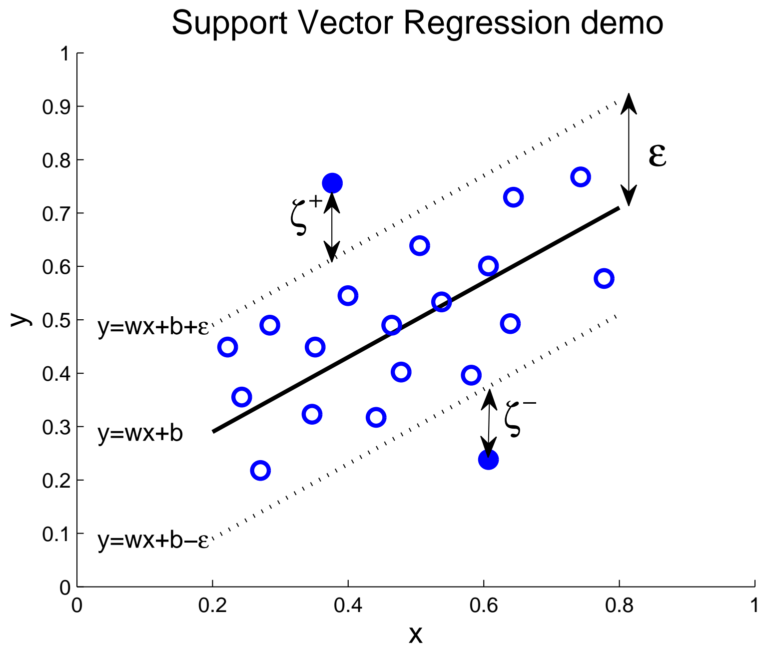
3.4 Support Vector Regression

Another special learning algorithm for the SLFN is the Support Vector Regression (SVR)[55, 56, 57]. Some literatures distinct the SVR (or SVM) from the ANN as different models. In fact, the SVR has the same architecture with SLFN and the kernels of the SVR can be regarded as the activation functions of the SLFN's hidden neurons. The SVR was inspired by the Support Vector Machine (SVM), which was popularly used for binary classification. The SVM was invented by Vladimir Vapnik and his co-workers in 1990s and the popularity overtook the interest of ANN in machine learning community in the following two decades until the appearance of deep learning.

The core idea of the SVM is to map the training data into a high or infinite dimensional space by using the kernel function and search an optimal hyperplane to separate the two classes in this space. The commonly used kernel functions include polynomial function, RBF and hyperbolic tangent (tanh) function. If the training patterns can be separated linearly in the high dimensional space, many different satisfactory hyperplanes exist. According to the SVM, a good separation is achieved by the hyperplane that has the largest distance to the nearest training pattern of any class (so-called functional margin). In general, the larger the margin is, a better generalization the classifier will have. The training patterns nearest to the hyperplane are called support vectors. Figure 3.2(a) shows a simple example of the



(a) Core idea of the SVM



(b) Core idea of the SVR

Figure 3.2: Explanation of the SVM and SVR

SVM classifier. The solid circle and squares are the support vectors. The line in the middle of the support vectors is the optimal separation solution.

The SVR has a similar idea. In order to approximate the training data set, the SVR maps them into a high dimensional space by using the kernel function and hopefully the data can be approximated with a hyperplane in that space. For the linear approximation task in the mapped high dimensional space, the SVR tries to build a hyper-tube, which can include all the training patterns. The center of the tube will be the desired hyperplane. A tube width ϵ needs to be set before training. Figure 3.2(b) shows a simple 2D example. The approximation allows to exist some outliers in the training data by introducing the slack variables ζ^+ and ζ^-. The penalty of the outliers will be added in the cost function.

The model of the SVM or SVR are described as,

$$f(\mathbf{x}, \boldsymbol{\theta}) = \sum_{j=1}^{K} \theta_j k(\mathbf{x}, \mathbf{x}_{s_j}) + b \tag{3.12}$$

in which, there are K kernels (similar to the hidden neurons). Each kernel function is $k(\mathbf{x}, \mathbf{x}_{s_j})$, which is similar to the activation function of the hidden neurons $h(\mathbf{x})$ in SLFN. $\{\mathbf{x}_{s_1}, ..., \mathbf{x}_{s_K}\} \in \{\mathbf{x}_1, ..., \mathbf{x}_P\}$ are the K support vectors selected from the training patters. They are regarded as tunable parameters, which is similar to the input weights of the sigmoid hidden neuron or the center of the RBF hidden neuron. $\boldsymbol{\theta} = [\theta_1, \theta_2, ..., \theta_K]^T$ are the coefficients of each kernel (like SLFN's output weights), b is the bias. The training procedure of the SVM or SVR is to find the support vectors from the training data set, and search the optimal parameters $\boldsymbol{\theta}$ and b in continuous space. An advantage of the SVR over the general gradient algorithms for SLFN is that the number of kernels K can be determined automatically according to the selected support vectors.

Different from the conventional loss function (SSE), the SVR uses ϵ-insensitive loss function $L_p(y_p, f(\mathbf{x}_p, \boldsymbol{\theta}))$ for each pattern.

$$L_p(y_p, f(\mathbf{x}_p, \boldsymbol{\theta})) = \begin{cases} 0 & \text{if } |y_p - f(\mathbf{x}_p, \boldsymbol{\theta})| \leq \epsilon \\ |y_p - f(\mathbf{x}_p, \boldsymbol{\theta})| - \epsilon & \text{otherwise} \end{cases} \tag{3.13}$$

So if the point is in the tube, the error is zero; if the pattern is outside of the tube, the error is linear related to the distance from the point to the tube. The entire loss function is,

$$L(\mathbf{y}, f(\mathbf{x}, \boldsymbol{\theta})) = \sum_{p=1}^{P} L_p(y_p, f(\mathbf{x}_p, \boldsymbol{\theta})) \tag{3.14}$$

At the same time, in order to reduce the complexity of the model, regularization is used by constraining the size of the coefficients $\boldsymbol{\theta}$. By introducing the nonnegative slack variables $\zeta_p^+, \zeta_p^- (p = 1, 2, ..., P)$ for each pattern, the SVR is robust to allow the existence of the outliers outside the tube. To obtain the tube with as many data points as possible inside it, the slack variables are added to the cost function so that the optimization makes the slack variables as sparse as possible. With all above considerations, the SVR is formulated as following constraint optimization problem,

$$\begin{aligned} \underset{\boldsymbol{\theta}}{\text{minimize}} \quad & \frac{1}{2}\|\boldsymbol{\theta}\|^2 + C\sum_{p=1}^{P}(\zeta_p^+ + \zeta_p^-) \\ \text{subject to} \quad & \begin{cases} y_p - f(\mathbf{x}_p, \boldsymbol{\theta}) \le \epsilon + \zeta_p^+ \\ f(\mathbf{x}_p, \boldsymbol{\theta}) - y_p \le \epsilon + \zeta_p^- \\ \zeta_p^+, \zeta_p^- \ge 0, p = 1, 2, ..., P \end{cases} \end{aligned}$$

The above optimization could be converted to quadratic programming problem. One has to preset the three hyperparameters before training: the tube width ϵ, the penalty C and the kernel parameters (e.g. gain for sigmoid kernel, width for RBF kernel, etc.). The result's generalization performance relies much on the selection of these hyperparameters. After the training, the SVR obtains several support vectors from the training data set. Then the number of the kernels will be the number of support vectors. Each support vector will be the parameters of the corresponding kernel (like input parameters of each hidden neuron in SLFN). The result $\boldsymbol{\theta}$ will be like the output weights of the SLFN. From the principle of the SVR, one can observe that the hidden parameters of the SLFN are not searched in the

continuous space. As a result, this strategy could hardly to find the optimal compact SLFN model.

3.5 Cascade Correlation Algorithm

The Fully Connected Cascade Networks (FCCN) topology was originally proposed by Fahlman and Lebiere in 1990 together with the Cascade Correlation (CasCor) algorithm[61]. Some literatures also called it as Cascade Correlation Neural Network. As shown in Figure 2.1(b), the FCCN topology has all the possible forward connections between every pair of neurons, where each neuron is a single layer of the network. Since each hidden neuron of the FCCN receives connections from all the inputs and all the previously installed hidden neurons, it's more powerful to represent high order nonlinear features. As a result, the FCCNs are widely used in different application fields.

The CasCor algorithm starts with a FCCN with no hidden neurons and constructs it by simply adding neurons one by one. Each time adding the new neuron, the CasCor algorithm has two steps: *Input training* and *Output training*. In the *Input training* step, several candidates of the hidden parameters of the new neuron are randomly generated. Each candidate is independently optimized by gradient ascent methods (the original paper used quickprop) to maximize the covariance between the outputs of this candidate neuron and the residual errors of the previous FCCN.

$$S_{cascor} = |\sum_{p=1}^{P}(e_{K,p} - \bar{e}_K)(h_{K+1,p} - \bar{h}_{K+1})| \tag{3.15}$$

The candidate with the maximum trained covariance is inserted to the FCCN. Then in the *Output training* step, all the output weights are tuned to minimize the SSE in (1.5). One can use some gradient methods or the LS method for the optimization task. The training procedure of the CasCor algorithm is exactly same as the previous mentioned Kwok's method, though one is training FCCN and the other is training the SLFN. As a result, all

the objective functions proposed by Kwok and the covariance of the CasCor could be shared for construction of both SLFN and FCCN.

3.6 Cascade2 Algorithm

While the CasCor algorithm worked well on many classification problems, like the *two-spiral* problem and the *parity-N* problem, it was argued that the covariance measurement for each hidden neuron's selection tended to make it saturate, which was not suitable for smooth regression problems[62, 63, 64]. For this reason, a second version of learning algorithm from the original author, the Cascade2 algorithm, had been investigated in several literatures[65].

The Cascade2 acts similar to the original CasCor algorithm, which has two steps: *Input training* and *Output training*. The *Output training* step is exactly same as the CasCor algorithm. In the *Input training* step, the Cascade2 doesn't use the covariance criteria to tune the input parameters of each candidate. Instead, it optimizes the input weights and output weight by minimizing the error between the weighted outputs of this neuron and the previous FCCN's residual error.

3.7 Orthogonal Least Square Algorithm

The Orthogonal Least Square (OLS) algorithm is a stepwise forward selection method[35, 36]. It was popularly used in RBF networks construction[37, 38]. Recently, Huang *et. al.*[60] utilized the OLS on the FCCN construction and proposed an OLSCN algorithm, which improved the learning performance a lot compared to the CasCor algorithm. In fact, the OLS algorithm is a constructive version of the LS method. By converting each component (neuron) to the orthogonal basis vector, each component's contribution to the squared error reduction can be considered independently. As a result, it's convenient to use the OLS algorithm in SLFN and FCCN construction. Following gives a simple introduction to the OLS algorithm.

In the SLFN or FCCN construction scheme, a candidate pool is initially generated for the $(K+1)_{th}$ neuron selection. Their input parameters are,

$$W_{K+1}^{pool} = \{\mathbf{w}_{K+1}^{(1)}, \mathbf{w}_{K+1}^{(2)}, ..., \mathbf{w}_{K+1}^{(N)}\} \tag{3.16}$$

the corresponding outputs are,

$$H_{K+1}^{pool} = \{\mathbf{h}_{K+1}^{(1)}, \mathbf{h}_{K+1}^{(2)}, ..., \mathbf{h}_{K+1}^{(N)}\} \tag{3.17}$$

The core idea of the OLS algorithm is to convert the components of the model (the columns of $\mathbf{H}_K$) into a set of orthogonal basis vectors by using QR decomposition.

$$\mathbf{H}_K = \mathbf{O}_K \mathbf{\Delta}_K \tag{3.18}$$

in which, $\mathbf{\Delta}_K$ is a $(K+1) \times (K+1)$ upper triangle matrix with 1s on the diagonal.

$$\mathbf{\Delta}_K = \begin{bmatrix} 1 & a_{01} & a_{02} & \cdots & a_{0K} \\ 0 & 1 & a_{12} & \cdots & a_{1K} \\ 0 & 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 1 & a_{K-1K} \\ 0 & \cdots & \cdots & 0 & 1 \end{bmatrix} \tag{3.19}$$

$\mathbf{O}_K$ is an $N \times (K+1)$ matrix with orthogonal columns ($\mathbf{o}_0$ are all 1s as bias),

$$\mathbf{O}_K = [\mathbf{o}_0, \mathbf{o}_1, ..., \mathbf{o}_K] \quad \mathbf{o}_i \mathbf{o}_j = 0, \text{ for all } i \neq j \tag{3.20}$$

With the storage of above decomposition for previous network, Gram-Schmidt process is carried out to convert each candidate in (3.17) to the new orthogonal basis vector. So

$\mathbf{O}_K$, $\boldsymbol{\Delta}_K$ are expanded to $\mathbf{O}_{K+1}$, $\boldsymbol{\Delta}_{K+1}$ by simply adding a new column. The detail of the process is described as below,

For i from 1 to N (every candidate in the pool),

1. Expanding new column to $\boldsymbol{\Delta}_K$.
   For j from 0 to K, $a_{jK+1}^{(i)} = \dfrac{\mathbf{o}_j^T \mathbf{h}_{K+1}^{(i)}}{\mathbf{o}_j^T \mathbf{o}_j}$
2. Expanding new column to $\mathbf{O}_K$.
   $\mathbf{o}_{K+1}^{(i)} = \mathbf{h}_{K+1}^{(i)} - \sum_{j=0}^{K} a_{jK+1}^{(i)} \mathbf{o}_j$

The advantage of the OLS algorithm is that by converting each component into an orthogonal vector, the cost function (SSE) of FCCN with K hidden neurons can be presented as,

$$C = \mathbf{y}^T \mathbf{y} - \sum_{j=0}^{K} \frac{(\mathbf{o}_j^T \mathbf{y})^2}{\mathbf{o}_j^T \mathbf{o}_j} \tag{3.21}$$

which means, every hidden neuron's contribution to the total error reduction can be described independent to each other.

As a result, for the $(K+1)_{th}$ neuron, one just select the candidate with the biggest contribution,

$$\arg\max_i \{[err]_{K+1}^{(i)} = \frac{(\mathbf{o}_{K+1}^{(i)}{}^T \mathbf{y})^2}{\mathbf{o}_{K+1}^{(i)}{}^T \mathbf{o}_{K+1}^{(i)}}\} \tag{3.22}$$

While one finished the selection process (assume K_m neurons are selected), the least square solutions of output weights can be achieved by solving the following equation,

$$\boldsymbol{\Delta}_{K_m} \boldsymbol{\theta} = \mathbf{g}_{K_m} \tag{3.23}$$

in which,

$$\mathbf{g}_{K_m} = [\frac{\mathbf{o}_1^T \mathbf{y}}{\mathbf{o}_1^T \mathbf{o}_1}, \frac{\mathbf{o}_2^T \mathbf{y}}{\mathbf{o}_2^T \mathbf{o}_2}, ..., \frac{\mathbf{o}_{K_m}^T \mathbf{y}}{\mathbf{o}_{K_m}^T \mathbf{o}_{K_m}}]^T \tag{3.24}$$

Since $\boldsymbol{\Delta}_{K_m}$ is a upper triangle matrix, it's easy to solve (3.23) by using back substitution.

Though the original OLS algorithm selects the best candidate from the pool as (3.22), it's not reasonable and necessary to search the parameters in the discrete space. With the objective function in (3.22), one can search the optimal parameters in continuous space by using some optimization methods. For example, in the OLSCN algorithm[60], a modified Newton's method was proposed to maximize the OLS objective function for searching each optimal new neuron in the FCCN construction.

3.8 Casper Algorithm

All the above algorithms use the freezing strategy during the construction. Though this strategy improves the learning efficiency, the result network is usually much larger than required. To overcome this problem, Treadgold and Gedeon dropped the freezing strategy and proposed a Casper algorithm[68, 69, 70] for the FCCN construction, which employed a Simulated Annealing Rprop (SARPROP) algorithm[67] to tune all the parameters in each stage.

The SARPROP algorithm was also proposed by Treadgold and Gedeon in 1998[67]. It modified the original Rprop algorithm by cooperating with the Simulated Annealing algorithm. With the aid of the Simulated Annealing algorithm, the SARPROP had more chances to arrive the global optima instead of the local minima during the training. The SARPROP also added the weight decay in the cost function in order to improve the generalization performance. The derivative of the cost function respect to each weight is shown as,

$$\frac{\partial C}{\partial w_{k,i}}^{SARPROP} = \frac{\partial C}{\partial w_{k,i}} - 0.01 * \frac{w_{k,i}}{1 + w_{k,i}^2} * SA \tag{3.25}$$

where $SA = 2^{-T*epoch}$ is the Simulated Annealing parameters decreasing exponentially as the iteration increases, T is the temperature, which is usually set in the range 0.01 to 0.05. The procedure of the SARPROP algorithm is similar to the Rprop algorithm. Figure 3.3 shows the pseudo code.

```
1: ∀k, i: Δ_{k,i}(0) = Δ_0, ∂C/∂w_{k,i}(0) = 0.
2: for t ← 1 to T do                                          ▷ Iteration number
3:     for all the parameters (weights) do                     ▷ update each weight
4:         Calculate derivative of each weight ∂C/∂w_{k,i} with (2.14).
5:         if ∂C/∂w_{k,i}(t − 1) ∗ ∂C/∂w_{k,i}(t) > 0 then
6:             Δ_{k,i}(t) = min(Δ_{k,i}(t − 1) ∗ η^+, Δ_max)
7:             Δw_{k,i}(t) = −sign(∂C/∂w_{k,i}(t)) ∗ Δ_{k,i}(t)
8:             w_{k,i}(t + 1) = w_{k,i}(t) + Δw_{k,i}(t)
9:         else if ∂C/∂w_{k,i}(t − 1) ∗ ∂C/∂w_{k,i}(t) < 0 then
10:            if Δ_{k,i}(t − 1) < 0.4 ∗ SA^2 then
11:                Δ_{k,i}(t) = Δ_{k,i}(t − 1) ∗ η^− + 0.8 ∗ r ∗ SA^2    ▷ r is random number in [0,1]
12:            else
13:                Δ_{k,i}(t) = Δ_{k,i}(t − 1) ∗ η^−
14:            end if
15:            Δ_{k,i}(t) = max(Δ_{k,i}(t), Δ_min)
16:            ∂C/∂w_{k,i}(t − 1) = 0
17:        else if ∂C/∂w_{k,i}(t − 1) ∗ ∂C/∂w_{k,i}(t) = 0 then
18:            Δw_{k,i}(t) = −sign(∂C/∂w_{k,i}(t)) ∗ Δ_{k,i}(t)
19:            w_{k,i}(t + 1) = w_{k,i}(t) + Δw_{k,i}(t)
20:        end if
21:    end for
22: end for
```

Figure 3.3: SARPROP

Different from the freezing strategy of other algorithms, the Casper algorithm tuned the entire FCCN with the SARPROP each time adding a new neuron. All the parameters setting of the SARPROP are same as the original Rprop algorithm except the initial learning rate Δ_0. The Casper algorithm divides the parameters of the network into three groups: L1, L2 and L3. Figure 3.4 shows a simple example when adding the second hidden neuron in the FCCN construction. The first group (L1) includes the weights connecting the inputs or previous neurons to the new added neuron. The second group (L2) is made up the weights connecting the new added neuron to the output neuron. The third group (L3) consists of all the rest weights. According to the analysis of Casper, the relation of the learning rates of the three groups should be L1 $\gg$ L2 $>$ L3. With this relation, the highest value of L1 allows the new neuron to learn previous network error. Similarly, the high value of L2 as compared

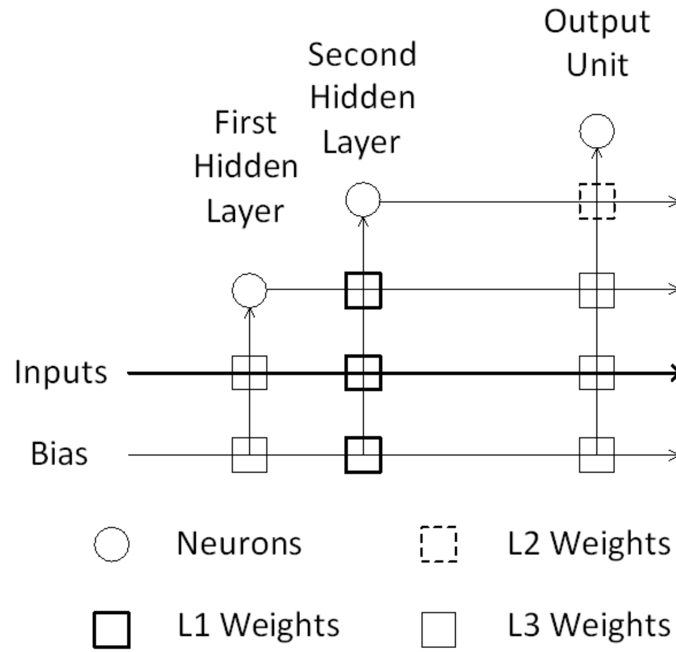


Figure 3.4: 3 groups parameters in Casper

to L3 allows the new neuron to cut down the error of network and avoids over interference from other weights.

Chapter 4

Hybrid Algorithm

Review those gradient learning algorithms in Chapter 2, all of them consider all the parameters of the FNN equally as nonlinear. However, as mentioned in Chapter 3, while solving the regression problems, the SLFN and FCCN have an important characteristic that the output neuron is linear. As a result, all the output weights are linear related and their optimal values can be determined in one step with the LS method.

Though many architecture oriented learning algorithms had taken this advantage, most of them used freezing strategy (like CasCor) or random generation mechanism (like ELMs). As a result, the training usually resulted in a network much larger than required. The large network would waste computing units and sometimes causes the computation delays, especially for the deep FCCN. On the other hand, the large network is more likely to overfit the data compared to the compact network. In order to achieve a compact network, the hidden parameters are necessary to be tuned.

The idea of combining the LS method and gradient algorithms to tune all the parameters is not new. McLoone *et al.*[42] proposed a hybrid Linear/Nonlinear training algorithm for SLFN training and used Full memory BFGS (FM) to optimize the nonlinear hidden parameters. Hui Peng *et al.*[43] combined LM algorithm and LS method for RBF networks training. Jian-Xun Peng *et al.*[44] proposed a new Jacobian matrix while using LM algorithm for nonlinear parameters optimization. Their experiments illustrated that the hybrid training outperformed the conventional optimization algorithms for SLFN training. However, the computation of the new Jacobian matrix by Jian-Xun Peng *et al.*[44] involved a large matrix manipulation (e.g. $\mathbf{R} \in R^{P\times P}$ in [44]). It's not practical for most of the modern regression problems with large data set. In this chapter, we derive a simpler update formula

for this idea and proposed the hybrid learning algorithm for the fixed size SLFN and FCCN. The hybrid algorithm also belongs to the architecture oriented learning algorithms defined in Chapter 3.

4.1 Review Conventional LM Algorithm

The conventional BP algorithm with LM optimization(LM-BP) treated all the parameters of SLFN or FCCN nonlinear and tuned them together. Define $\boldsymbol{\psi} = [\mathbf{W}_v^T; \boldsymbol{\theta}]$ as a vector including all the tunable parameters. The update formula of the conventional LM-BP algorithm was introduced in Chapter 2 (2.14)(2,15). The Jacobian matrix could use the network errors or outputs as objective function. Here, we use the direct outputs version as shown in (2.13).

The Jacobian matrix of the conventional LM-BP algorithm consists of two components —— derivatives over nonlinear parameters($\mathbf{W}_v$) $\mathbf{J}_n$ and derivatives over linear output weights($\boldsymbol{\theta}$) $\mathbf{J}_l$.

$$\mathbf{J} = [\mathbf{J}_n, \mathbf{J}_l] \tag{4.1}$$

The nonlinear part $\mathbf{J}_n$ is,

$$\begin{aligned} \mathbf{J}_n &= [\frac{\partial \tilde{\mathbf{y}}}{\partial w_{1,1}}, \frac{\partial \tilde{\mathbf{y}}}{\partial w_{1,2}}, ..., \frac{\partial \tilde{\mathbf{y}}}{\partial w_{K,m}}] \\ &= [\frac{\partial \mathbf{h}_1}{\partial w_{1,1}}\hat{\theta}_1, \frac{\partial \mathbf{h}_1}{\partial w_{1,2}}\hat{\theta}_1, ..., \frac{\partial \mathbf{h}_K}{\partial w_{K,m}}\hat{\theta}_K] \end{aligned} \tag{4.2}$$

The linear part is actually the hidden matrix $\mathbf{J}_l = \mathbf{H}$.

The detailed procedure of the conventional LM-BP algorithm is shown in Section 2.5.

4.2 Derive New Update Formula for SLFN

When combining with the LS method for the output weights, we only need to update the nonlinear part parameters. The linear part of the Jacobian matrix $\mathbf{J}_l$ should be dropped.

However, we can't use the nonlinear part of the Jacobian matrix $\mathbf{J}_n$ directly in the old update formula (2.13), the new Jacobian matrix $\mathbf{J}_{new}$ and the new update formula for only nonlinear parameters should be derived.

$$\mathbf{\Delta W_{\boldsymbol{v}}} = (\mathbf{J}_{new}^T \mathbf{J}_{new} + \mu \mathbf{I})^{-1} \mathbf{J}_{new}^T \mathbf{e} \tag{4.3}$$

While determining the optimal output weights with LS method, as shown in (4.4), the output weights become dependent variables of the nonlinear parameters.

$$\hat{\boldsymbol{\theta}} = (\mathbf{H}^T \mathbf{H})^{-1} \mathbf{H}^T \mathbf{y} \tag{4.4}$$

where the nonlinear parameters $\mathbf{W}$ are included in the hidden matrix $\mathbf{H}$. Then the actual outputs become,

$$\tilde{\mathbf{y}} = \mathbf{H}\hat{\boldsymbol{\theta}} = \mathbf{H}(\mathbf{H}^T \mathbf{H})^{-1} \mathbf{H}^T \mathbf{y} \tag{4.5}$$

So in the new Jacobian matrix, the column for the i_{th} parameter of the k_{th} hidden neuron($w_{k,i}$) is,

$$\frac{\partial \tilde{\mathbf{y}}}{\partial w_{k,i}} = \frac{\partial (\mathbf{H}(\mathbf{H}^T \mathbf{H})^{-1} \mathbf{H}^T)}{\partial w_{k,i}} \mathbf{y} \tag{4.6}$$

in which,

$$\begin{aligned} &\frac{\partial (\mathbf{H}(\mathbf{H}^T \mathbf{H})^{-1} \mathbf{H}^T)}{\partial w_{k,i}} \\ = &\frac{\partial \mathbf{H}}{\partial w_{k,i}} (\mathbf{H}^T \mathbf{H})^{-1} \mathbf{H}^T + \mathbf{H}(\mathbf{H}^T \mathbf{H})^{-1} \frac{\partial \mathbf{H}^T}{\partial w_{k,i}} \\ &- \mathbf{H}(\mathbf{H}^T \mathbf{H})^{-1} \frac{\partial \mathbf{H}^T}{\partial w_{k,i}} \mathbf{H}(\mathbf{H}^T \mathbf{H})^{-1} \mathbf{H}^T \\ &- \mathbf{H}(\mathbf{H}^T \mathbf{H})^{-1} \mathbf{H}^T \frac{\partial \mathbf{H}}{\partial w_{k,i}} (\mathbf{H}^T \mathbf{H})^{-1} \mathbf{H}^T \end{aligned} \tag{4.7}$$

Denote $\mathbf{R}_{k,i}$, $\mathbf{S}_{k,i}$ as,

$$\mathbf{R}_{k,i} = \frac{\partial \mathbf{H}}{\partial w_{k,i}} (\mathbf{H}^T \mathbf{H})^{-1} \mathbf{H}^T \tag{4.8}$$

$$\mathbf{S}_{k,i} = \mathbf{H}(\mathbf{H}^T\mathbf{H})^{-1}\frac{\partial \mathbf{H}^T}{\partial w_{k,i}}\mathbf{H}(\mathbf{H}^T\mathbf{H})^{-1}\mathbf{H}^T \tag{4.9}$$

Substitute into (4.6)(4.7), then,

$$\frac{\partial \tilde{\mathbf{y}}}{\partial w_{k,i}} = (\mathbf{R}_{k,i} - \mathbf{S}_{k,i}^T)\mathbf{y} + (\mathbf{R}_{k,i}^T - \mathbf{S}_{k,i})\mathbf{y} \tag{4.10}$$

Because,

$$\begin{aligned}
(\mathbf{R}_{k,i} - \mathbf{S}_{k,i}^T)\mathbf{y} &= \frac{\partial \mathbf{H}}{\partial w_{k,i}}(\mathbf{H}^T\mathbf{H})^{-1}\mathbf{H}^T\mathbf{y} \\
&\qquad - \mathbf{H}(\mathbf{H}^T\mathbf{H})^{-1}\mathbf{H}^T\frac{\partial \mathbf{H}}{\partial w_{k,i}}(\mathbf{H}^T\mathbf{H})^{-1}\mathbf{H}^T\mathbf{y} \\
&= \frac{\partial \mathbf{H}}{\partial w_{k,i}}\hat{\boldsymbol{\theta}} - \mathbf{H}(\mathbf{H}^T\mathbf{H})^{-1}\mathbf{H}^T\frac{\partial \mathbf{H}}{\partial w_{k,i}}\hat{\boldsymbol{\theta}} \\
&= (\mathbf{I} - \mathbf{H}(\mathbf{H}^T\mathbf{H})^{-1}\mathbf{H}^T)\frac{\partial \mathbf{h}_k}{\partial w_{k,i}}\hat{\theta}_k
\end{aligned} \tag{4.11}$$

$$\begin{aligned}
(\mathbf{R}_{k,i}^T - \mathbf{S}_{k,i})\mathbf{y} &= \mathbf{H}(\mathbf{H}^T\mathbf{H})^{-1}\frac{\partial \mathbf{H}^T}{\partial w_{k,i}}\mathbf{y} \\
&\qquad - \mathbf{H}(\mathbf{H}^T\mathbf{H})^{-1}\frac{\partial \mathbf{H}^T}{\partial w_{k,i}}\mathbf{H}(\mathbf{H}^T\mathbf{H})^{-1}\mathbf{H}^T\mathbf{y} \\
&= \mathbf{H}(\mathbf{H}^T\mathbf{H})^{-1}\frac{\partial \mathbf{H}^T}{\partial w_{k,i}}(\mathbf{y} - \mathbf{H}\hat{\boldsymbol{\theta}}) \\
&= \mathbf{H}(\mathbf{H}^T\mathbf{H})^{-1}\frac{\partial \mathbf{H}^T}{\partial w_{k,i}}\mathbf{e} \\
&= \mathbf{H}(\mathbf{H}^T\mathbf{H})^{-1}\begin{bmatrix} 0 \\ \vdots \\ \frac{\partial \mathbf{h}_k^T}{\partial w_{k,i}}\mathbf{e} \\ \vdots \\ 0 \end{bmatrix}
\end{aligned} \tag{4.12}$$

$$\mathbf{Q} = \begin{bmatrix} 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 & \cdots & \cdots & 0 & 0 & \cdots & 0 \\ \frac{\partial \mathbf{h}_1^T}{\partial w_{11}}\mathbf{e} & \frac{\partial \mathbf{h}_1^T}{\partial w_{12}}\mathbf{e} & \cdots & \frac{\partial \mathbf{h}_1^T}{\partial w_{1m}}\mathbf{e} & 0 & 0 & \cdots & 0 & \cdots & \cdots & 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & \frac{\partial \mathbf{h}_2^T}{\partial w_{21}}\mathbf{e} & \frac{\partial \mathbf{h}_2^T}{\partial w_{22}}\mathbf{e} & \cdots & \frac{\partial \mathbf{h}_2^T}{\partial w_{2m}}\mathbf{e} & \cdots & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 & \cdots & \cdots & \frac{\partial \mathbf{h}_K^T}{\partial w_{K1}}\mathbf{e} & \frac{\partial \mathbf{h}_K^T}{\partial w_{K2}}\mathbf{e} & \cdots & \frac{\partial \mathbf{h}_K^T}{\partial w_{Km}}\mathbf{e} \end{bmatrix} \quad (4.13)$$

So each column of the new Jacobian matrix can be divided into two parts. As a result, the whole Jacobian matrix can be divided into two components.

$$\mathbf{J}_{new} = \mathbf{U} + \mathbf{V} \quad (4.14)$$

in which, $\mathbf{U}$'s every column comes from (4.11); $\mathbf{V}$'s every column comes from (4.12).

One may notice that the last term ($\frac{\partial \mathbf{h}_k}{\partial w_{k,i}}\hat{\theta}_k$) in (4.11) is exactly the corresponding column in the conventional Jacobian matrix($\mathbf{J}_n$) in (4.2). So,

$$\mathbf{U} = (\mathbf{I} - \mathbf{H}(\mathbf{H}^T\mathbf{H})^{-1}\mathbf{H}^T)\mathbf{J}_n \quad (4.15)$$

For the second component, we define a sparse matrix $\mathbf{Q} \in R^{(K+1)\times(Km)}$ as shown in (4.13). Then,

$$\mathbf{V} = \mathbf{H}(\mathbf{H}^T\mathbf{H})^{-1}\mathbf{Q} \quad (4.16)$$

Substitute (4.14-4.16) into (4.3), one can get the update of the nonlinear parameters,

$$\Delta\mathbf{W}_v = (\mathbf{U}^T\mathbf{U} + \mathbf{U}^T\mathbf{V} + \mathbf{V}^T\mathbf{U} + \mathbf{V}^T\mathbf{V} + \mu\mathbf{I})^{-1}(\mathbf{U}^T\mathbf{e} + \mathbf{V}^T\mathbf{e}) \quad (4.17)$$

Notice that $(\mathbf{I} - \mathbf{H}(\mathbf{H}^T\mathbf{H})^{-1}\mathbf{H}^T)$ is symmetric and idempotent. So,

$$\begin{aligned} \mathbf{U}^T\mathbf{U} &= \mathbf{J}_n^T(\mathbf{I} - \mathbf{H}(\mathbf{H}^T\mathbf{H})^{-1}\mathbf{H}^T)^T(\mathbf{I} - \mathbf{H}(\mathbf{H}^T\mathbf{H})^{-1}\mathbf{H}^T)\mathbf{J}_n \\ &= \mathbf{J}_n^T\mathbf{J}_n - \mathbf{J}_n^T\mathbf{H}(\mathbf{H}^T\mathbf{H})^{-1}\mathbf{H}^T\mathbf{J}_n \end{aligned} \quad (4.18)$$

$$
\begin{aligned}
\mathbf{U}^T\mathbf{V} &= \mathbf{J}_n^T(\mathbf{I} - \mathbf{H}(\mathbf{H}^T\mathbf{H})^{-1}\mathbf{H}^T)\mathbf{H}(\mathbf{H}^T\mathbf{H})^{-1}\mathbf{Q} \\
&= \mathbf{J}_n^T\mathbf{H}(\mathbf{H}^T\mathbf{H})^{-1}\mathbf{Q} - \mathbf{J}_n^T\mathbf{H}(\mathbf{H}^T\mathbf{H})^{-1}\mathbf{H}^T\mathbf{H}(\mathbf{H}^T\mathbf{H})^{-1}\mathbf{Q} \\
&= \mathbf{J}_n^T\mathbf{H}(\mathbf{H}^T\mathbf{H})^{-1}\mathbf{Q} - \mathbf{J}_n^T\mathbf{H}(\mathbf{H}^T\mathbf{H})^{-1}\mathbf{Q} = \mathbf{0}
\end{aligned} \tag{4.19}
$$

$$
\mathbf{V}^T\mathbf{U} = (\mathbf{U}^T\mathbf{V})^T = \mathbf{0} \tag{4.20}
$$

$$
\begin{aligned}
\mathbf{V}^T\mathbf{V} &= \mathbf{Q}^T(\mathbf{H}^T\mathbf{H})^{-1}\mathbf{H}^T\mathbf{H}(\mathbf{H}^T\mathbf{H})^{-1}\mathbf{Q} \\
&= \mathbf{Q}^T(\mathbf{H}^T\mathbf{H})^{-1}\mathbf{Q}
\end{aligned} \tag{4.21}
$$

Notice that,

$$
\begin{aligned}
(\mathbf{H}^T\mathbf{H})^{-1}\mathbf{H}^T\mathbf{e} &= (\mathbf{H}^T\mathbf{H})^{-1}\mathbf{H}^T(\mathbf{y} - \tilde{\mathbf{y}}) \\
&= (\mathbf{H}^T\mathbf{H})^{-1}\mathbf{H}^T\mathbf{y} - (\mathbf{H}^T\mathbf{H})^{-1}\mathbf{H}^T\tilde{\mathbf{y}} \\
&= \hat{\boldsymbol{\theta}} - (\mathbf{H}^T\mathbf{H})^{-1}\mathbf{H}^T\mathbf{H}\hat{\boldsymbol{\theta}} = \mathbf{0}
\end{aligned} \tag{4.22}
$$

which is a zero vector, so,

$$
\begin{aligned}
\mathbf{U}^T\mathbf{e} &= \mathbf{J}_n^T(\mathbf{I} - \mathbf{H}(\mathbf{H}^T\mathbf{H})^{-1}\mathbf{H}^T)^T\mathbf{e} \\
&= \mathbf{J}_n^T\mathbf{e} - \mathbf{J}_n^T\mathbf{H}(\mathbf{H}^T\mathbf{H})^{-1}\mathbf{H}^T\mathbf{e} = \mathbf{J}_n^T\mathbf{e}
\end{aligned} \tag{4.23}
$$

$$
\mathbf{V}^T\mathbf{e} = \mathbf{Q}^T(\mathbf{H}^T\mathbf{H})^{-1}\mathbf{H}^T\mathbf{e} = \mathbf{0} \tag{4.24}
$$

Substitute (4.18-2.21)(4.23-4.24) into (4.17), one can get the following simple update formula(4.25) for the nonlinear parameters.

$$\Delta \mathbf{W}_v = (\mathbf{J}_n^T \mathbf{J}_n - \mathbf{J}_n^T \mathbf{H}(\mathbf{H}^T \mathbf{H})^{-1} \mathbf{H}^T \mathbf{J}_n + \mathbf{Q}^T (\mathbf{H}^T \mathbf{H})^{-1} \mathbf{Q} + \mu \mathbf{I})^{-1} \mathbf{J}_n^T \mathbf{e} \tag{4.25}$$

Observe the new update formula, the manipulating matrices include $\mathbf{J}_n^T \mathbf{J}_n \in R^{Km \times Km}$, $\{\mathbf{Q}^T, \mathbf{J}_n^T \mathbf{H}\} \in R^{Km \times (K+1)}$, $(\mathbf{H}^T \mathbf{H})^{-1} \in R^{(K+1) \times (K+1)}$. They are relatively cheaper computation compared to the new Jacobian matrix proposed by Jian-Xun Peng *et al.*[44], which involved a $P \times P$ matrix operation. Since general approximation problems have much more data than number of parameters, that $P \gg (Km + K + 1)$, the proposed hybrid algorithm is more suitable for approximation with large data sets.

4.3 Regularization

By using (4.25) to update the nonlinear hidden parameters and using (4.4) to determine the linear output weights in each iteration, the optimization process is simplified and sped up. However, one deficiency of the algorithm is that $\mathbf{H}^T \mathbf{H}$ is required to be nonsingular in every step. To avoid ill-conditioning case, it is suggested to add a regularizor λ to it. So the optimal linear weights in (4.4) becomes,

$$\hat{\boldsymbol{\theta}} = (\mathbf{H}^T \mathbf{H} + \lambda \mathbf{I})^{-1} \mathbf{H}^T \mathbf{y} \tag{4.26}$$

This is also called ridge regression, which is actually equivalent to,

$$\begin{aligned} &\underset{\boldsymbol{\theta}}{\text{minimize}} \quad C = (\mathbf{y} - \mathbf{H}\boldsymbol{\theta})^T (\mathbf{y} - \mathbf{H}\boldsymbol{\theta}) \\ &\text{subject to} \quad \|\boldsymbol{\theta}\| < \gamma, \ \text{for some } \gamma > 0. \end{aligned} \tag{4.27}$$

which constraints the search of the linear parameters in some trust region in each iteration. The constraint γ is related to the regularizor λ.

In fact, while one uses regularized LS to determine the linear weight in each iteration, the derived update formula for nonlinear parameters are much more complex than (4.25). However, experiments showed that adding regularizor to (4.25) directly, which is (4.28), still performed well in most problems.

$$\Delta \mathbf{W}_v = (\mathbf{J}_n^T\mathbf{J}_n - \mathbf{J}_n^T\mathbf{H}(\mathbf{H}^T\mathbf{H} + \lambda\mathbf{I})^{-1}\mathbf{H}^T\mathbf{J}_n + \mathbf{Q}^T(\mathbf{H}^T\mathbf{H} + \lambda\mathbf{I})^{-1}\mathbf{Q} + \mu\mathbf{I})^{-1}\mathbf{J}_n^T\mathbf{e} \tag{4.28}$$

Review the procedure of the conventional LM-BP algorithm, the proposed hybrid algorithm modified the step 2) to calculate the nonlinear Jacobian matrix $\mathbf{J}_n$ (4.2) and the sparse matrix $\mathbf{Q}$ (4.13). In step 3), instead of $\Delta\boldsymbol{\psi}$, $\Delta\mathbf{W}_v$ is calculated with (4.28). Another notation is that, each time update the nonlinear parameters, the linear parameters need to update immediately with (4.26), so that to keep the linear parameters to be the LS optimal

1: Assume the SLFN has K hidden neurons, all the nonlinear parameters included in a vector $\mathbf{W}_v$, the linear output weights are $\boldsymbol{\theta}$. Set damping factor $\mu = 0.01$, its upper bound μ_U and lower bound μ_L, scale factor $\beta = 10$, regularizor λ, maximum iteration T.
2: **for** $t \leftarrow 1$ to T **do**
3:     Calculate $\mathbf{J}_n$ with (4.2) and $\mathbf{Q}$ with (4.13);
4:     **for** $n \leftarrow 1$ to 10 **do**
5:         Calculate $\Delta\mathbf{W}_v$ with (4.28);
6:         Compute forward with parameters $\mathbf{W}_v + \Delta\mathbf{W}_v$, determine output weights as (4.26), calculate SSE(t);
7:         **if** $t \geq 2$ **then**
8:             **if** SSE(t) $\leq$ SSE($t-1$) **then**
9:                 $\mu \leftarrow \max\{\mu_L, \mu/\beta\}$
10:                 $\mathbf{W}_v \leftarrow \mathbf{W}_v + \Delta\mathbf{W}_v$
11:                 **break**
12:             **else**
13:                 $\mu \leftarrow \min\{\mu_U, \mu\beta\}$
14:             **end if**
15:         **end if**
16:     **end for**
17: **end for**

Figure 4.1: Pseudocode of Hybrid Algorithm For Fixed Sized SLFN

all the time. Figure 4.1 shows the pseudocode of the entire hybrid algorithm for training a fixed size SLFN.

4.4 New Update Formula for FCCN

Review the derivation of the hybrid update formula for SLFN, everything can be reused for the FCCN. The only difference between the SLFN and the FCCN is the calculation of the $\mathbf{J}_n$ and the $\mathbf{Q}$. Compared with the SLFN, it's more complex to calculate the Jacobian matrix for the FCCN, since every neuron collects the backpropagated signals from all the following neurons, as shown in Figure 2.1(b). Assume current FCCN has K hidden neurons, then there are $r = (D+1)+(D+2)+...+(D+K) = KD+\frac{K(K+1)}{2}$ nonlinear parameters. So the nonlinear Jacobian matrix $\mathbf{J}_n$ has r columns,

$$\mathbf{J}_n = [\frac{\partial \tilde{\mathbf{y}}}{\partial w_{10}}, ..., \frac{\partial \tilde{\mathbf{y}}}{\partial w_{1D}}, ..., ..., \frac{\partial \tilde{\mathbf{y}}}{\partial w_{k0}}, ..., \frac{\partial \tilde{\mathbf{y}}}{\partial w_{k(D+k-1)}}] \tag{4.29}$$

The sparse matrix $\mathbf{Q} \in R^{(k+D+1)\times r}$ becomes a blockwise lower triangular matrix with the first $D+1$ rows to be all zeros, as shown in (4.30).

Direct using of the backpropagation process according to the differential chain rule is quite complex in the FCCN architecture. In this dissertation, we use the forward only strategy by Wilamowski and Yu (shown in Section 2.8)[39] to calculate each components of the $\mathbf{J}_n$ and $\mathbf{Q}$.

The forward-only method removed the backpropagation process and created a lower triangular δ table to store every desired values through the forward computation. Here, we created a vector version of the δ table to store all desired values for calculating $\mathbf{J}_n$ and $\mathbf{Q}$, as shown in Table 4.1. The neuron number is from 1 to K in the order of the depth. "o" represents the linear output neuron. The vector in the cell (i,j) is the derivative of the i_{th} neuron's output over the j_{th} neuron's net value for all the training patterns, denote as $\boldsymbol{\delta}_{i,j}$

$$
\mathbf{Q}=\begin{bmatrix}
0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 & \cdots & \cdots & 0 & 0 & \cdots & 0\\
\vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \ddots & \vdots & \vdots & \ddots & \vdots\\
0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 & \cdots & \cdots & 0 & 0 & \cdots & 0\\
\frac{\partial \mathbf{h}_1^T}{\partial w_{10}}\mathbf{e} & \frac{\partial \mathbf{h}_1^T}{\partial w_{11}}\mathbf{e} & \cdots & \frac{\partial \mathbf{h}_1^T}{\partial w_{1D}}\mathbf{e} & 0 & 0 & \cdots & 0 & \cdots & \cdots & 0 & 0 & \cdots & 0\\
\frac{\partial \mathbf{h}_2^T}{\partial w_{10}}\mathbf{e} & \frac{\partial \mathbf{h}_2^T}{\partial w_{11}}\mathbf{e} & \cdots & \frac{\partial \mathbf{h}_2^T}{\partial w_{1D}}\mathbf{e} & \frac{\partial \mathbf{h}_2^T}{\partial w_{20}}\mathbf{e} & \frac{\partial \mathbf{h}_2^T}{\partial w_{21}}\mathbf{e} & \cdots & \frac{\partial \mathbf{h}_2^T}{\partial w_{2(D+1)}}\mathbf{e} & \cdots & \cdots & 0 & 0 & \cdots & 0\\
\vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \ddots & \vdots & \vdots & \ddots & \vdots\\
\frac{\partial \mathbf{h}_K^T}{\partial w_{10}}\mathbf{e} & \frac{\partial \mathbf{h}_K^T}{\partial w_{11}}\mathbf{e} & \cdots & \frac{\partial \mathbf{h}_K^T}{\partial w_{1D}}\mathbf{e} & \frac{\partial \mathbf{h}_K^T}{\partial w_{20}}\mathbf{e} & \frac{\partial \mathbf{h}_K^T}{\partial w_{21}}\mathbf{e} & \cdots & \frac{\partial \mathbf{h}_K^T}{\partial w_{2(D+1)}}\mathbf{e} & \cdots & \cdots & \frac{\partial \mathbf{h}_K^T}{\partial w_{K0}}\mathbf{e} & \frac{\partial \mathbf{h}_K^T}{\partial w_{K1}}\mathbf{e} & \cdots & \frac{\partial \mathbf{h}_K^T}{\partial w_{K(D+K-1)}}\mathbf{e}
\end{bmatrix} \tag{4.30}
$$

Table 4.1: vector version of the δ table

neuron #	1	2	3	$\cdots$	K	o
1	$\frac{\partial \mathbf{h}_1}{\partial \mathbf{net}_1}$					
2	$\frac{\partial \mathbf{h}_2}{\partial \mathbf{net}_1}$	$\frac{\partial \mathbf{h}_2}{\partial \mathbf{net}_2}$				
3	$\frac{\partial \mathbf{h}_3}{\partial \mathbf{net}_1}$	$\frac{\partial \mathbf{h}_3}{\partial \mathbf{net}_2}$	$\frac{\partial \mathbf{h}_3}{\partial \mathbf{net}_3}$			
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$		
K	$\frac{\partial \mathbf{h}_K}{\partial \mathbf{net}_1}$	$\frac{\partial \mathbf{h}_K}{\partial \mathbf{net}_2}$	$\frac{\partial \mathbf{h}_K}{\partial \mathbf{net}_3}$	$\cdots$	$\frac{\partial \mathbf{h}_K}{\partial \mathbf{net}_K}$	
o	$\frac{\partial \tilde{\mathbf{y}}}{\partial \mathbf{net}_1}$	$\frac{\partial \tilde{\mathbf{y}}}{\partial \mathbf{net}_2}$	$\frac{\partial \tilde{\mathbf{y}}}{\partial \mathbf{net}_3}$	$\cdots$	$\frac{\partial \tilde{\mathbf{y}}}{\partial \mathbf{net}_K}$	$\mathbf{1}$

* All the partial derivatives are pointwise.

(The partial derivatives shown in the table are all pointwise).

$$
\boldsymbol{\delta}_{i,j}=\frac{\partial \mathbf{h}_i}{\partial \mathbf{net}_j}=[\frac{\partial h_{1,i}}{\partial net_{1,j}},\frac{\partial h_{2,i}}{\partial net_{2,j}},...,\frac{\partial h_{P,i}}{\partial net_{P,j}}]^T \tag{4.31}
$$

The values on the diagonal are directly the slopes of each single neuron. The other values are computed according to all the above values in the same column.

$$
\boldsymbol{\delta}_{i,j}=\boldsymbol{\delta}_{i,i}\circ\sum_{m=j}^{i-1}\boldsymbol{\delta}_{m,j}w_{m\to i} \tag{4.32}
$$

where $w_{m\to i}$ is the weight connecting the m_{th} neuron to the i_{th} neuron, $\circ$ represents pointwise product of two vectors.

After going through the forward computation, with the vectors stored in the δ table, one can calculate the $\mathbf{Q}$ matrix with the first K rows and compute the $\mathbf{J}_n$ matrix with the last row. For example, with the vector $\frac{\partial \mathbf{h}_i}{\partial \mathbf{net}_j}(K \geq i \geq j \geq 1)$, one can calculate the corresponding differential term in matrix $\mathbf{Q}$ as,

$$\frac{\partial \mathbf{h}_i}{\partial w_{jm}} = \frac{\partial \mathbf{h}_i}{\partial \mathbf{net}_j} \circ \mathbf{H}(:, m) \quad m = 0, 1, ..., D + j - 1 \tag{4.16}$$

in which, $\mathbf{H}(:, m)$ is the m_{th} column of the *signal matrix* $\mathbf{H}$. Also given the vector $\frac{\partial \tilde{\mathbf{y}}}{\partial \mathbf{net}_i}$ in the last row, one can calculate the corresponding column in the nonlinear Jacobian matrix $\mathbf{J}_n$ as,

$$\frac{\partial \tilde{\mathbf{y}}}{\partial w_{im}} = \frac{\partial \tilde{\mathbf{y}}}{\partial \mathbf{net}_i} \circ \mathbf{H}(:, m) \quad m = 0, 1, ..., D + i - 1 \tag{4.17}$$

In fact, while using the forward only strategy for conventional LM algorithm, only the last row of the δ table is finally used and the rest values are dropped, like most dynamic programming problems. However, the hybrid algorithm used all the values in the table, which may be more suitable to employ the forward only strategy.

Same as in the SLFN, we introduce a regularizor λ to the hybrid algorithm for FCCN, which constrains the search of the output weights in a trust region. As a result, the optimal output weights are determined with (4.26), the update formula for the nonlinear parameters uses (4.28). This regularization avoids the risk of ill conditioning, it can also guarantee a better generalization for the training process.

Chapter 5

Extend Hybrid Algorithm for Construction

The hybrid learning algorithm reduces the LM optimization dimension by converting the linear output weights into dependent variables of the nonlinear parameters with LS method. It improves the training efficiency compared to the conventional backpropagation with LM optimization (LM-BP) algorithm. The hybrid algorithm works well while optimizing the parameters of SLFN and FCCN with fixed size. In order to train the network size simultaneously, based on the hybrid algorithm in Chapter 4, an incremental constructive scheme is proposed. We call the algorithm as Hybrid Constructive (HC) algorithm.

Since the SLFN and FCCN are similar in the construction, we don't distinct them in this chapter, call them FNN and introduce their construction together. Like other constructive algorithms, we start from an empty FNN and then add the hidden neurons one by one. However, the traditional freezing strategy is not used. Instead, each time adding a new neuron, the previous proposed hybrid algorithm is used to tune all the parameters. The result parameters of the previous training before adding the new neuron, are used as the initial parameters of the new hybrid training after adding this neuron. Because the linear output weights keep to be LS optimal in the hybrid algorithm, the convergence after adding new hidden neuron is obvious.

5.1 When to add new neuron

Though the LM algorithm is one of the most efficient optimization algorithm, one deficiency is that it can only find local minima, which means one have chance to fail during the training. A common scheme is to initialize random parameters, train and validate, restart with another random parameters if it fails. In fact, when the number of hidden neurons

is not enough, even global minima still cannot meet the required error. When the training entrapped into some local minima, no matter whether it is global minima, the proposed hybrid constructive(HC) algorithm can get out of the local minima by introducing more parameters(adding one more hidden neuron).

The criterion of evaluating the local minima is described as,

$$|\frac{C(t) - C(t-N)}{C(t)}| < \eta \tag{5.1}$$

in which, $C(t), C(t-N)$ are the cost function (SSE) at the t_{th} and the $(t-N)_{th}$ iteration. N is the iteration latency. η is the decreasing threshold. Both N and η are preset by user.

5.2 How to add new neuron

While the local minima is detected as in (5.1), a new neuron is added to the current FNN. One has two options to add the new neuron before the hybrid tuning: add a random neuron directly; select or tune the parameters of the new neuron. In this section, we describe both these two options. The first option is easy, one just generates random parameters for the new neuron and adds it the FNN, we call it HC1 algorithm. For the second option, one could use the objective functions by Kwok *et al.*[25] (Section 3.2), or the Cascade Correlation criteria[61] (Section 3.5), etc. In this dissertation, we propose a new criteria, which uses a similar criteria as the OLSCN algorithm[60] (Section 3.7). The Particle Swarm Optimization is used for the selection (or tuning) of the new neuron's parameters. We call the HC algorithm using this selection option as HC2 algorithm.

Since the hybrid algorithm will be used to tune all the parameters of the network after adding the new neuron, we need to update all the variables in the update formula of the hybrid algorithm (4.28) each time. However, one doesn't need to recalculate everything in the update formula. In Section 5.2.1, we derive a simpler method to update those variables in (4.28). Both the HC1 and HC2 algorithms could use this convenience for the variables

update. In Section 5.2.2 and 5.2.3, we will introduce the neuron selection details of the HC2 algorithm.

5.2.1 Variables in Update Formula

Review the new update formula (4.28) of the hybrid algorithm, the variables that need to update after adding new neuron include hidden matrix $\mathbf{H}$, nonlinear Jacobian matrix $\mathbf{J}_n$, the sparse matrix $\mathbf{Q}$, error vector $\mathbf{e}$ and output weights $\hat{\boldsymbol{\theta}}$. Denote them in current FNN (with K hidden neurons) as $\mathbf{H}_K$, $\mathbf{J}_K$, $\mathbf{Q}_K$, $\mathbf{e}_K$ and $\hat{\boldsymbol{\theta}}_K$. Their new values after adding a new neuron are $\mathbf{H}_{K+1}$, $\mathbf{J}_{K+1}$, $\mathbf{Q}_{K+1}$, $\mathbf{e}_{K+1}$ and $\hat{\boldsymbol{\theta}}_{K+1}$.

Assume the nonlinear parameters of the new neuron are $\mathbf{w}_{K+1} = [w_{K+1,1}, ..., w_{K+1,m}]$. In the HC1 algorithm, $\mathbf{w}_{K+1}$ are randomly generated directly (normally in the range [-1,1]); In the HC2 algorithm, $\mathbf{w}_{K+1}$ are the tuning results. Assume the outputs of the new neuron for all the patterns are $\mathbf{h}_{K+1}$. Then,

$$\mathbf{H}_{K+1} = [\mathbf{H}_K, \mathbf{h}_{K+1}] \tag{5.2}$$

Denote $\mathbf{M}_K = (\mathbf{H}_K^T \mathbf{H}_K + \lambda \mathbf{I})^{-1}$.

Now do a temporary linear LS regression to target $\mathbf{h}_{K+1}$ with current FNN (with same regularization λ). Denote the optimal output weights of this regression as $\hat{\boldsymbol{\theta}}_t$.

$$\hat{\boldsymbol{\theta}}_t = (\mathbf{H}_K^T \mathbf{H}_K + \lambda \mathbf{I})^{-1} \mathbf{H}_K^T \mathbf{h}_{K+1} = \mathbf{M}_K \mathbf{H}_K^T \mathbf{h}_{K+1} \tag{5.3}$$

and the error of this regression $\mathbf{e}_t$ is,

$$\mathbf{e}_t = \mathbf{h}_{K+1} - \mathbf{H}_K \hat{\boldsymbol{\theta}}_t \tag{5.4}$$

So,

$$\mathbf{M}_{K+1} = \left(\begin{bmatrix} \mathbf{H}_K^T \\ \mathbf{h}_{K+1}^T \end{bmatrix} \begin{bmatrix} \mathbf{H}_K & \mathbf{h}_{K+1} \end{bmatrix} + \lambda \mathbf{I} \right)^{-1}$$

$$= \begin{bmatrix} \mathbf{M}_K + \dfrac{\mathbf{M}_K \mathbf{H}_K^T \mathbf{h}_{K+1} \mathbf{h}_{K+1}^T \mathbf{H}_K \mathbf{M}_K}{a} & \dfrac{-\mathbf{M}_K \mathbf{H}_K^T \mathbf{h}_{K+1}}{a} \\ \dfrac{-\mathbf{h}_{K+1}^T \mathbf{H}_K \mathbf{M}_K}{a} & \dfrac{1}{a} \end{bmatrix}$$

$$= \begin{bmatrix} \mathbf{M}_K + \dfrac{\hat{\boldsymbol{\theta}}_t \hat{\boldsymbol{\theta}}_t^T}{a} & -\dfrac{\hat{\boldsymbol{\theta}}_t}{a} \\ -\dfrac{\hat{\boldsymbol{\theta}}_t^T}{a} & \dfrac{1}{a} \end{bmatrix} \tag{5.5}$$

in which, a is a scalar,

$$a = \mathbf{h}_{K+1}^T \mathbf{e}_t + \lambda \tag{5.6}$$

Then the new optimal output weights are,

$$\hat{\boldsymbol{\theta}}_{K+1} = \mathbf{M}_{K+1} \mathbf{H}_{K+1}^T \mathbf{y}$$

$$= \begin{bmatrix} \mathbf{M}_K \mathbf{H}_K^T \mathbf{y} + \dfrac{\mathbf{M}_K \mathbf{H}_K^T \mathbf{h}_{K+1} \mathbf{h}_{K+1}^T}{a} (\mathbf{H}_K \mathbf{M}_K \mathbf{H}_K^T \mathbf{y} - \mathbf{y}) \\ \dfrac{\mathbf{h}_{K+1}^T}{a} (\mathbf{y} - \mathbf{H}_K \mathbf{M}_K \mathbf{H}_K^T \mathbf{y}) \end{bmatrix}$$

$$= \begin{bmatrix} \hat{\boldsymbol{\theta}}_K - \dfrac{b}{a} \hat{\boldsymbol{\theta}}_t \\ \dfrac{b}{a} \end{bmatrix} \tag{5.7}$$

in which, scalar b is denoted as $b = \mathbf{h}_{K+1}^T \mathbf{e}_K$.

And the new errors are,

$$\mathbf{e}_{K+1} = \mathbf{y} - \mathbf{H}_{K+1} \hat{\boldsymbol{\theta}}_{K+1} = \mathbf{e}_K - \frac{b}{a} \mathbf{e}_t \tag{5.8}$$

So by introducing temporary LS regression to target $\mathbf{h}_{K+1}$, the update process after adding the new neuron is simplified a lot. Based on the above formulas (5.2-5.8), without matrix inversion, $(\mathbf{H}_{K+1}^T\mathbf{H}_{K+1} + \lambda\mathbf{I})^{-1}$, $\hat{\boldsymbol{\theta}}_{K+1}$ and $\mathbf{e}_{K+1}$ can be updated according to their previous values. For the nonlinear Jacobian matrix $\mathbf{J}_{K+1}$ and the sparse matrix $\mathbf{Q}_{K+1}$, according to their definitions (4.2)(4.13), their components $\{\frac{\partial \mathbf{h}_1^T}{\partial w_{11}}, \frac{\partial \mathbf{h}_1^T}{\partial w_{12}}, ..., \frac{\partial \mathbf{h}_K^T}{\partial w_{Km}}\}$ are exactly same as previous. So one only need to update them according to the updated output weights $\hat{\boldsymbol{\theta}}_{K+1}$ and errors $\mathbf{e}_{K+1}$ and then add the corresponding part for the new neuron.

5.2.2 Reformulate OLS algorithm

With the above formulas to update the variables in (4.28), the HC1 algorithm could be processed clearly. However, the random generation of the parameters doesn't guarantee the new neuron would contribute much to the FNN in the regression task. Though expanding the parameter dimension by adding a new neuron could help to drag the previous FNN out of the local minima, when the new neuron is not well initialized, the hybrid algorithm may still have trouble to escape the previous local minima. In this case, the new added neuron doesn't help to reduce the approximation errors (SSE). There's no need to add these useless neurons. In order to achieve a FNN as compact as possible, it's necessary to filter the initialization of each new neuron before real adding.

In Chapter 3, we introduced several architecture oriented learning algorithms. Different criteria were used in those algorithms for selecting each new added neuron, like the objective functions by Kwok *et al.*[25], the Cascade Correlation criterion[61], the Cascade2 criterion[65], the OLS criterion[60], etc. In this section, we propose a new criterion similar to the OLS algorithm. However, we reformulated the OLS in a recursive way so that it's more convenient to work with the hybrid algorithm.

Consider a FNN with K hidden neurons, whose hidden matrix is $\mathbf{H}_K$. While $\mathbf{H}_K^T\mathbf{H}_K$ is nonsingular, the global optimal solution for the output weights $\boldsymbol{\theta}_K$ can be simply computed

as,

$$\hat{\boldsymbol{\theta}}_K = \mathbf{H}_K^\dagger \mathbf{y} \tag{5.9}$$

where $\mathbf{H}_K^\dagger$ is the Moore-Penrose generalized inverse of the hidden matrix $\mathbf{H}_K$.

$$\mathbf{H}_K^\dagger = (\mathbf{H}_K^T \mathbf{H}_K)^{-1} \mathbf{H}_K^T \tag{5.10}$$

For a guess of the $(K+1)_{th}$ hidden neuron with hidden parameters $\mathbf{w}_{K+1}^{(i)}$ and outputs $\mathbf{h}_{K+1}^{(i)}$, similar as in 5.2.1, do a temporary linear regression to the target $\mathbf{h}_{K+1}^{(i)}$ with current FNN (with K hidden neurons). We can get the optimal solution $\hat{\boldsymbol{\theta}}_t^{(i)}$ and error $\mathbf{e}_t^{(i)}$ of this regression as shown in (5.3)(5.4).

Then let $\mathbf{M}_K = (\mathbf{H}_K^T \mathbf{H}_K)^{-1}$, similar to (5.5), we have, (here we use the ordinary OLS without regularization to filter the new neuron)

$$\begin{aligned}
\mathbf{M}_{K+1} &= (\mathbf{H}_{K+1}^T \mathbf{H}_{K+1})^{-1} \\
&= \begin{bmatrix} \mathbf{H}_K^T \mathbf{H}_K & \mathbf{H}_K^T \mathbf{h}_{K+1} \\ \mathbf{h}_{K+1}^T \mathbf{H}_K & \mathbf{h}_{K+1}^T \mathbf{h}_{K+1} \end{bmatrix}^{-1} \\
&= \begin{bmatrix} \mathbf{M}_K + \dfrac{\boldsymbol{\theta}_t \boldsymbol{\theta}_t^T}{\mathbf{h}_{K+1}^T \mathbf{e}_t} & -\dfrac{\boldsymbol{\theta}_t}{\mathbf{h}_{K+1}^T \mathbf{e}_t} \\ -\dfrac{\boldsymbol{\theta}_t^T}{\mathbf{h}_{K+1}^T \mathbf{e}_t} & \dfrac{1}{\mathbf{h}_{K+1}^T \mathbf{e}_t} \end{bmatrix}
\end{aligned} \tag{5.11}$$

The Moore-Penrose generalized inverse of the new hidden matrix $\mathbf{H}_{K+1}$ is,

$$\begin{aligned}
\mathbf{H}_{K+1}^{\dagger} &= \mathbf{M}_{K+1}\mathbf{H}_{K+1}^{T} \\
&= \begin{bmatrix} \mathbf{M}_K\mathbf{H}_K^T + \dfrac{\boldsymbol{\theta}_t}{\mathbf{h}_{K+1}^T\mathbf{e}_t}(\boldsymbol{\theta}_t^T\mathbf{H}_K^T - \mathbf{h}_{K+1}^T) \\ \dfrac{1}{\mathbf{h}_{K+1}^T\mathbf{e}_t}(\mathbf{h}_{K+1}^T - \boldsymbol{\theta}_t^T\mathbf{H}_K^T) \end{bmatrix} \\
&= \begin{bmatrix} \mathbf{H}_K^{\dagger} - \dfrac{\boldsymbol{\theta}_t\mathbf{e}_t^T}{\mathbf{h}_{K+1}^T\mathbf{e}_t} \\ \dfrac{\mathbf{e}_t^T}{\mathbf{h}_{K+1}^T\mathbf{e}_t} \end{bmatrix}
\end{aligned} \tag{5.12}$$

Denote the errors for current FNN as $\mathbf{e}_K = \mathbf{y} - \tilde{\mathbf{y}}_K$, and the new errors after adding the $(K+1)_{th}$ neuron as $\mathbf{e}_{K+1}$. Based on the above update formula for $\mathbf{H}_{K+1}^{\dagger}$, it's easy to derive (5.13)(5.14) for $\hat{\boldsymbol{\theta}}_{K+1}$ and $\mathbf{e}_{K+1}$.

$$\hat{\boldsymbol{\theta}}_{K+1} = \begin{bmatrix} \hat{\boldsymbol{\theta}}_K - \dfrac{\mathbf{h}_{K+1}^T\mathbf{e}_K}{\mathbf{h}_{K+1}^T\mathbf{e}_t}\hat{\boldsymbol{\theta}}_t \\ \dfrac{\mathbf{h}_{K+1}^T\mathbf{e}_K}{\mathbf{h}_{K+1}^T\mathbf{e}_t} \end{bmatrix} \tag{5.13}$$

$$\mathbf{e}_{K+1} = \mathbf{e}_K - \frac{\mathbf{h}_{K+1}^T\mathbf{e}_K}{\mathbf{h}_{K+1}^T\mathbf{e}_t}\mathbf{e}_t \tag{5.14}$$

After adding the $(K+1)_{th}$ hidden neuron, the total error reduction can be calculated as,

$$\begin{aligned}
[err]_{K+1} &= C_K - C_{K+1} \\
&= \mathbf{e}_K^T\mathbf{e}_K - \mathbf{e}_{K+1}^T\mathbf{e}_{K+1} \\
&= 2\frac{\mathbf{h}_{K+1}^T\mathbf{e}_K}{\mathbf{h}_{K+1}^T\mathbf{e}_t}\mathbf{e}_K^T\mathbf{e}_t - \frac{(\mathbf{h}_{K+1}^T\mathbf{e}_K)^2}{(\mathbf{h}_{K+1}^T\mathbf{e}_t)^2}\mathbf{e}_t^T\mathbf{e}_t
\end{aligned} \tag{5.15}$$

Since $(\mathbf{I} - \mathbf{H}(\mathbf{H}^T\mathbf{H})^{-1}\mathbf{H}^T)$ is symmetric and idempotent matrix,

$$\begin{aligned} & \mathbf{e}_K^T\mathbf{e}_t \\ & = (\mathbf{y} - \mathbf{H}_K(\mathbf{H}_K^T\mathbf{H}_K)^{-1}\mathbf{H}_K^T\mathbf{y})^T \times \\ & \quad (\mathbf{h}_{K+1} - \mathbf{H}_K(\mathbf{H}_K^T\mathbf{H}_K)^{-1}\mathbf{H}_K^T\mathbf{h}_{K+1}) \\ & = \mathbf{y}^T(\mathbf{I} - \mathbf{H}_K(\mathbf{H}_K^T\mathbf{H}_K)^{-1}\mathbf{H}_K^T)^T \times \\ & \quad (\mathbf{I} - \mathbf{H}_K(\mathbf{H}_K^T\mathbf{H}_K)^{-1}\mathbf{H}_K^T)\mathbf{h}_{K+1} \\ & = \mathbf{h}_{K+1}^T\mathbf{e}_K \end{aligned} \tag{5.16}$$

and,

$$\begin{aligned} & \mathbf{e}_t^T\mathbf{e}_t \\ & = \mathbf{h}_{K+1}^T(\mathbf{I} - \mathbf{H}_K(\mathbf{H}_K^T\mathbf{H}_K)^{-1}\mathbf{H}_K^T)^T \times \\ & \quad (\mathbf{I} - \mathbf{H}_K(\mathbf{H}_K^T\mathbf{H}_K)^{-1}\mathbf{H}_K^T)\mathbf{h}_{K+1} \\ & = \mathbf{h}_{K+1}^T\mathbf{e}_t \end{aligned} \tag{5.17}$$

Substitute (5.16)(5.17) into (5.15), one can get the contribution of the $(K+1)_{th}$ neuron as shown in (5.18).

$$[err]_{K+1}^{(i)} = \frac{(\mathbf{h}_{K+1}^{(i)}{}^T\mathbf{e}_K)^2}{\mathbf{h}_{K+1}^{(i)}{}^T\mathbf{e}_t^{(i)}} \tag{5.15}$$

which can replace (3.11) to evaluate the error reduction contribution of the new neuron with any possible hidden parameters.

Relation to the Conventional OLS Algorithm

While the conventional OLS algorithm used the orthogonal vectors as space basis vectors, each new neuron was decomposed as a representation of the previous basis vectors and

a new orthogonal vector,

$$\mathbf{h}_{K+1} = \sum_{j=0}^{K} a_{jK+1}\mathbf{o}_j + \mathbf{o}_{K+1} \tag{5.16}$$

Instead, the reformulated OLS algorithm regarded the orthogonal vectors as latent variables, and used the previous components as basis vectors directly.

$$\mathbf{h}_{K+1} = \sum_{j=0}^{K} \theta_{t,j}\mathbf{h}_j + \mathbf{e}_t \tag{5.17}$$

in which, $\theta_{t,j}$ is the j_{th} element of $\boldsymbol{\theta}_t$.

Since $\{\mathbf{o}_0, \mathbf{o}_1, ..., \mathbf{o}_K\}$ are the orthogonal basis vectors of $\{\mathbf{1}, \mathbf{h}_1, ..., \mathbf{h}_K\}$, we can conclude that the residual error of the temporary regression ($\mathbf{e}_t$) in (5.17) is actually the decomposed orthogonal vector of the new neuron ($\mathbf{o}_{K+1}$) in (5.16).

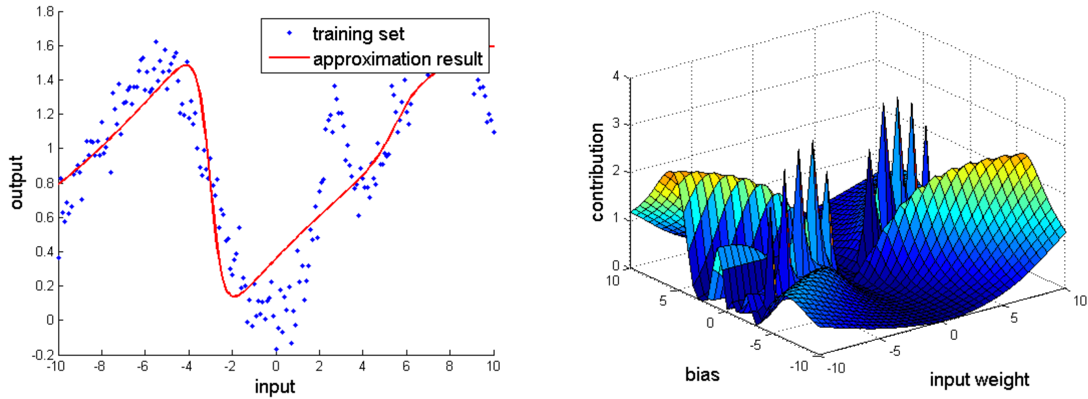
By reformulated the conventional OLS algorithm, all the parameters could be updated recursively. Therefore, the proposed reformulated OLS algorithm is more suitable for dynamic construction of FNN.

5.2.3 Particle Swarm Optimization

Motivation

The conventional OLS algorithm mainly worked as a selection method, which used the error reduction contribution criterion to pick the best component from a candidate pool. However, the parameters of the SLFN are not necessary limit in the discrete space, like the candidate pool. In this paper, the hidden parameters of the new added neuron are optimized in the continuous space to maximize the new derived contribution function (5.15).

For this optimization task, Huang *et al.*[60] proposed a modified Newton's method. Because of the local property of the gradient based algorithm, several candidates need to be trained to search the global optima. In fact, the error reduction contribution defined in the OLS algorithm is a complex multimodal function respect to the hidden parameters.



(a) Approximation by SLFN with 3 hidden neurons

(b) Contribution versus hidden parameters while adding the 4th neuron

Figure 5.1: Construct sigmoid SLFN for 1-dimension approximation

Figure 5.1 shows a 1-dimension approximation example while constructing a SLFN with sigmoid activation function. For the 1-dimensional case, each hidden neuron only has two parameters: one input weight and one bias. Therefore, the contribution function could be visualized as a 3D surface, as shown in Figure 5.1(b). One can observe that, even for a small-size SLFN, with which the approximation is not satisfied (as shown in Figure 5.1(a)), many local maxima exist. Considering the multimodal characteristic and the existence of the plateaus, in this paper, instead of the gradient based methods, Particle Swarm Optimization (PSO) is used to tune the new added neuron's hidden parameters[45].

Basis Concept of PSO

The PSO algorithm is a population based stochastic optimization technique developed by Kennedy and Eberhart in 1995, inspired by social behavior of bird flocking and fish schooling. Unlike genetic algorithm (GA), the PSO algorithm doesn't have the complicated operators, such as crossover and mutation[46], therefore it has less parameters to set. Due to the efficiency and simplicity, PSO is popular used for neural networks learning alternative to back propagation (BP) algorithm[47]-[50].

The PSO algorithm optimizes an objective function by having a population of candidate solutions, called particles, which are initialized randomly in the searching space. Each particle has a position and velocity. These particles are moved around the searching space to seek the global optima by updating their positions and velocities iteratively according to simple formula. The movement of each particle is influenced by its individual best known position (**pbest**) and the best known position in the swarm (**gbest**). Assume a population with N particles, for the i_{th} particle, denote its position and velocity are $\mathbf{w}^{(i)}, \mathbf{v}^{(i)} \in R^m$. Then each dimension of $\mathbf{w}^{(i)}$ and $\mathbf{v}^{(i)}$ are updated as,

$$\begin{aligned} v_j^{(i)} &= c_0 \times v_j^{(i)} + c_1 \times rand() \times (pbest_j^{(i)} - w_j^{(i)}) \\ &\quad + c_2 \times rand() \times (gbest_j - w_j^{(i)}) \end{aligned} \tag{5.18}$$

$$w_j^{(i)} = w_j^{(i)} + v_j^{(i)} \tag{5.19}$$

in which, $w_j^{(i)}, v_j^{(i)}$ are the j_{th} dimension of $\mathbf{w}^{(i)}$ and $\mathbf{v}^{(i)}$ $(j = 1, 2, ..., m)$. c_1, c_2 are the acceleration constants with positive values set by user. $rand()$ is randomly generated number in the range [0,1]. $pbest_j^{(i)}$ is the j_{th} dimension of the best known position ($\mathbf{pbest}^{(i)}$) in the i_{th} particle's searching history. $gbest_j$ is the j_{th} dimension of the best known position (**gbest**) in the entire swarm. c_0 is called inertia weight introduced by Shi and Eberhart[51], which plays a key role in balancing the exploration and exploitation process of the swarm. Many researches have been done on the parameters setting of the PSO algorithm[52, 53] In this paper, we will use the suggested setting in [53]: $c_1 = c_2 = 2$; a linearly decreasing inertia weight c_0 starts at 0.9 and ends at 0.4.

It has been shown that the neural networks with smaller size tend to produce smoother functions, which could generalize better[59]. Inspired by this, we have a bound for the searching space of those hidden parameters. Assume the maximum amplitude of the input data is $r > 0$, which means the input range $\subseteq [-r, r]$. Then the hidden parameters of each

neuron is bounded as 10 times of this range, which is $[-10r, 10r]$. In particular for the width σ of the RBF node, we bounded it as $\sigma \in (0, 10r]$. The velocity corresponding to each parameter is bounded in the same range ($[10r, 10r]$) to constraint the exploration of the swarm. As a result, in each iteration as the PSO proceeds, after the velocity is updated with (5.18), it's determined as,

$$v_j^{(i)} = \begin{cases} -10r & \text{if } v_j^{(i)} \leq -10r \\ v_j^{(i)} & \text{if } -10r < v_j^{(i)} < 10r \\ 10r & \text{if } v_j^{(i)} \geq 10r \end{cases} \tag{5.20}$$

after the parameters are updated with (5.19), for all the parameters of sigmoid neuron and centers of RBF neuron, they are determined as,

$$w_j^{(i)} = \begin{cases} -10r & \text{if } w_j^{(i)} \leq -10r \\ w_j^{(i)} & \text{if } -10r < w_j^{(i)} < 10r \\ 10r & \text{if } w_j^{(i)} \geq 10r \end{cases} \tag{5.21}$$

for the width of RBF neuron, they are determined as,

$$w_j^{(i)} = \begin{cases} \epsilon & \text{if } w_j^{(i)} \leq \epsilon \\ w_j^{(i)} & \text{if } \epsilon < w_j^{(i)} < 10r \\ 10r & \text{if } w_j^{(i)} \geq 10r \end{cases} \tag{5.22}$$

in which, ϵ is a small positive value set by user.

The pseudocode of the PSO procedure is shown in Figure 5.2. Each time after this procedure, the optimal parameters of the new hidden neuron is stored in **gbest**. The variables $\hat{\boldsymbol{\theta}}_t$, $\mathbf{e}_t$ corresponding to this best particle will be reused for the update of $\mathbf{H}_{K+1}^{\dagger}$, $\hat{\boldsymbol{\theta}}_{K+1}$ and $\mathbf{e}_{K+1}$ according to (5.12-5.14).

1: Assume current SLFN has K hidden neurons, whose hidden matrix is $\mathbf{H}_K$, the Moore-Penrose generalized inverse of the hidden matrix is $\mathbf{H}_K^{\dagger}$, the optimal output weights of current SLFN is $\hat{\boldsymbol{\theta}}_K$, the error is $\mathbf{e}_K$. One is trying to add the $(K+1)_{th}$ hidden neuron.
2: Initialize a population of N particles with random positions $\{\mathbf{w}_{K+1}^{(1)}, \mathbf{w}_{K+1}^{(2)}, ..., \mathbf{w}_{K+1}^{(N)}\}$ and random velocities $\{\mathbf{v}_{K+1}^{(1)}, \mathbf{v}_{K+1}^{(2)}, ..., \mathbf{v}_{K+1}^{(N)}\}$. Each position is initialized as $\mathbf{pbest}^{(i)}$, whose fitness calculated by (16-18) is f_i. Pick the maximum fitness as f_g and the corresponding particle position as **gbest**.
3: **for** $t \leftarrow 1$ to T **do** ▷ Iteration number
4:     **for** $i \leftarrow 1$ to N **do** ▷ Each particle
5:         **for** $d \leftarrow 1$ to D **do** ▷ Each dimension
6:             update each velocity with (24)(26)
7:             udpate each parameter with (25)(27)(28)
8:         **end for**
9:         Calculate fitness $[err]_{K+1}^{(i)}$ of current parameters using (16-18)
10:         **if** $[err]_{K+1}^{(i)} > f_i$ **then**
11:             $\mathbf{pbest}^{(i)} \leftarrow \mathbf{w}_{K+1}^{(i)}$
12:             $f_i \leftarrow [err]_{K+1}^{(i)}$
13:         **end if**
14:         **if** $[err]_{K+1}^{(i)} > f_g$ **then**
15:             $\mathbf{gbest} \leftarrow \mathbf{w}_{K+1}^{(i)}$
16:             Store $\hat{\boldsymbol{\theta}}_t$ and $\mathbf{e}_t$
17:             $f_g \leftarrow [err]_{K+1}^{(i)}$
18:         **end if**
19:     **end for**
20: **end for**

Figure 5.2: Selection of the New Neuron with PSO

5.3 When to stop adding

The construction process will stop when one of the following criterion is satisfied.

1. The cost function (SSE) meets the desired value ϵ, $C(t) < \epsilon$.

2. The number of iteration arrives the maximum number of iteration (T) set by the user.

3. The number of hidden neurons arrives the maximum number of hidden neurons (K_{max}) set by user.

4. When the training saturates so that adding more neurons does not help the minimization of SSE. Similar as the start adding criterion, denote $t(K)$ as the iteration number

to add the K_{th} hidden neuron. When the following condition satisfied, the construction can stop.

$$|\frac{C(t(K)-1)-C(t(K+L)-1)}{C(t(K)-1)}|<\tau \tag{5.23}$$

in which, L, τ are parameters set by user. L is the neuron number latency, τ is threshold. When the above condition meets, we'll remove the neurons from K to $K+L$ and use the parameters at the iteration $t(K)-1$.

Chapter 6

Experiments

6.1 Experiments on the HC1 Algorithm for SLFN Construction

In this section, the first kind of Hybrid Constructive Algorithm (HC1), which adds randomly initialized new neuron each time, is experimented on several practical regression problems by constructing the SLFN. All the experiments use the Root Mean Square Error (RMSE) to evaluate the approximation accuracy.

$$\text{RMSE} = \sqrt{\frac{C}{P}} \tag{6.1}$$

in which, C is SSE defined in (1.4) and P is the number of patterns.

The experiment environment consists of: Windows 7 Enterprise 64-bit operating system, Intel®Core™2 Quad CPU Q8400 2.67GHz process, 4.00GB RAM, MATLAB R2012a platform.

6.1.1 SinE function

In this example, a rapidly changing continuous SinE function (6.2) is given to test the efficiency of the HC1 algorithm for a sigmoid SLFN construction.

$$y(x) = 0.8 \exp(-0.2x) \sin(10x) \tag{6.2}$$

2000 points were randomly generated in the range [0,10] for training while 1000 points were picked in the same range randomly for testing. Both training and testing data were added gaussian noise with variance $Var = 0.001$, as shown in Figure 6.1. SLFN with sigmoid

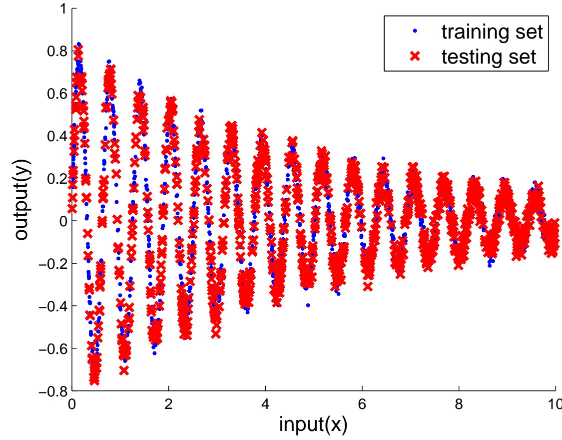
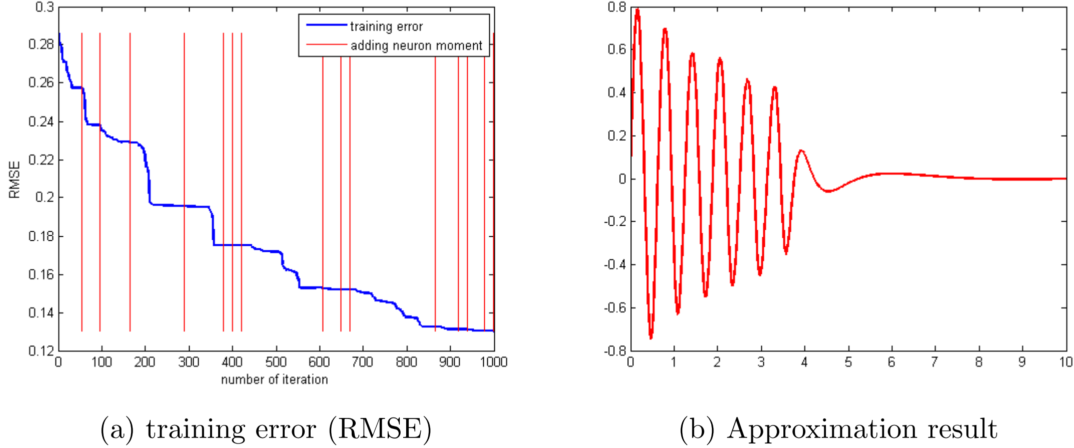


Figure 6.1: Training and testing data set
for SinE function approximation



(a) training error (RMSE) (b) Approximation result

Figure 6.2: One example trial while using HC1 algorithm to approximate SinE function. (a) shows the decreasing error (RMSE) during the 1000 iterations, vertical lines are the moment to add new hidden neuron. (this trial ended with 17 hidden neurons and training RMSE=0.1302, testing RMSE=0.1234) (b) shows the approximation result of this trial.

function in hidden layer were constructed with the proposed HC1 algorithm, conventional BP algorithm with LM optimization (LM-BP) and Extreme Learning Machine (ELM). Each algorithm was trained 100 trials from random initialization. The LM-BP and ELM trained SLFN with different size by trial and error. The maximum number of iteration of both HC1 and LM-BP were set as $T = 1000$. For the proposed HC1 algorithm, the maximum number of hidden neurons was $K_{max} = 100$, iteration latency and decreasing threshold for adding

Table 6.1: Experiment Results Comparisons for SinE Function Approximation

Algorithms	Number of hidden nodes	training RMSE		testing RMSE		Average training time (s)
		mean	std	mean	std	
LM-BP	24	0.1544	0.0079	0.1468	0.0091	12.47
LM-BP	26	0.1539	0.0072	0.1464	0.0085	16.03
LM-BP	28	0.1537	0.0081	0.1463	0.0097	17.39
ELM	100	0.1634	0.0433	0.1624	0.0615	0.02
ELM	200	0.1557	0.0988	0.1729	0.1462	0.04
ELM	300	0.1384	0.0828	0.1586	0.1244	0.06
HC1	24.6 (average)	0.1388	0.0204	0.1317	0.0213	8.8843

neuron criterion were $N = 10, \eta = 0.001$. The neuron number latency and threshold for stopping criterion were set as $L = 10, \tau = 0.001$. Regularizor in (10) is set as $\lambda = 0.001$.

Figure 6.2 shows one of the 100 trials while using HC1 algorithm to approximate the SinE function. One can observe that the training was dragged out of some local minima by adding more neurons. Since each training continued on the previous training results, the computation in the early stage was light. Therefore, it costs much less computation time than training the SLFN with the same size by using the LM-BP algorithm.

The LM-BP algorithm and ELM optimized SLFN with fixed size. Trial and error approach needs to be used to determine the optimal network size. Because of the rapid changing of the SinE function, as shown in Figure 6.1, many trials need to be considered. Table 6.1 shows the comparison results of these algorithms. Benefit from the random setting of all the nonlinear parameters, the ELM algorithm can train the SLFN hundreds of times faster than both LM-BP and HC algorithms to achieve an acceptable accuracy. However, because there's no further tuning for those random nonlinear parameters, the training always leads to a very large SLFN.

6.1.2 Peaks Function Approximation

The peaks function is a popular 2-dimension nonlinear benchmark for approximation algorithms. In this experiment, we used a normalized format of peaks function (6.3). Figure.

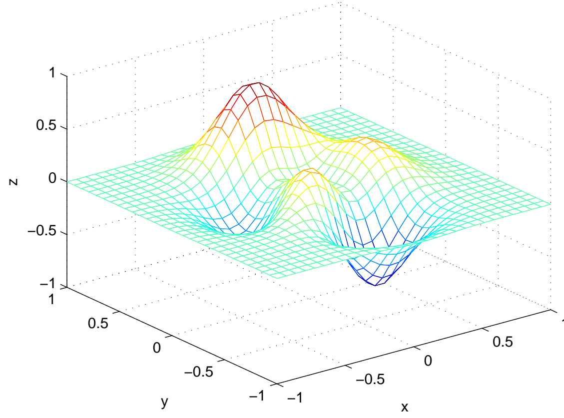


Figure 6.3: Peaks function

6.3 shows the mesh plot of the ideal peaks function.

$$
\begin{aligned}
z = & (0.3 + 1.8x + 2.7x^2) \exp(-1 - 6y - 9x^2 - 9y^2) - \\
& (0.6x - 27x^3 - 243y^5) \exp(-9x^2 - 9y^2) - \\
& \frac{1}{30} \exp(-1 - 6x - 9x^2 - 9y^2)
\end{aligned} \tag{6.3}
$$

In the experiment, 2000 points were generated randomly in the range [-1,1] as training sets, and another 1000 points were generated in the same way as testing sets. All the patterns added a guassian noise with variance $Var = 0.01$. A SLFN with RBF activation function was constructed with the proposed HC1 algorithm, LM-BP algorithm by trial and error, and ELM by trial and error. All the algorithms ran 100 trials from random initial parameters. The maximum number of iteration of the LM-BP algorithm and the HC1 algorithm are set as 100 and 150. For HC1 algorithm, all the other settings were the same as the previous experiment.

The experiment results are shown in Table 6.2. From the comparison, although ELM requires much less training time compared to those optimization methods, several times larger SLFN is required to achieve the same accuracy. While using LM-BP algorithm, one needs to try SLFN with 4, 5, 6, etc. hidden nodes, each of which cost much computation

Table 6.2: Experiment Results Comparisons for Peaks Function Approximation

Algorithms	Number of hidden nodes	training RMSE		testing RMSE		Average training time (s)
		mean	std	mean	std	
LM-BP	4	0.1460	0.0251	0.144773	0.0238	0.2918
LM-BP	5	0.1342	0.0234	0.1335	0.0221	0.3852
LM-BP	6	0.1311	0.0242	0.1304	0.0231	0.4811
LM-BP	7	0.1203	0.0197	0.1199	0.0187	0.5639
ELM	20	0.1517	0.0093	0.1527	0.0159	0.0054
ELM	30	0.1348	0.0079	0.1448	0.0673	0.0076
ELM	40	0.1228	0.0069	0.1352	0.0811	0.0102
ELM	50	0.1141	0.0056	0.1248	0.0445	0.0135
HC1	6.54 (average)	0.1157	0.0189	0.1160	0.0178	0.4405

time. With the proposed HC1 algorithm, the network size and the parameters can be trained simultaneously, and high accuracy can be achieved with a compact SLFN.

6.1.3 UCI real life problems

In this section, these algorithms were compared on several multivariate real life approximation problems from the UCI database[54]. The specifications of the data sets are shown in Table 6.3. All the data sets were approximated by the SLFN with sigmoid activation function. Support Vector Regression with sigmoid kernel was also experimented for comparison.

The maximum number of iterations for LM-BP and HC1 algorithms were both set as 100. With HC1 algorithm, the network size can be determined automatically during training. In this experiment, 100 trials had been run, the averaged network size and training time are shown in Table 6.5; the mean testing error (RMSE) and the corresponding standard deviation (std) are shown in Table 6.4. For the LM-BP and ELM algorithms, trial and error approach was used for determining network size. SLFN with different sizes were trained by LM-BP and ELM, the optimal network size is shown in Table 6.5. Then the SLFN with the optimal network size for each problem was trained by LM-BP and ELM for 100 trials. The averaged training time, mean testing error (RMSE) and corresponding standard deviation

Table 6.3: UCI data sets specifications

Data Sets	# Training Patterns	# Testing Patterns	Input Dimension
Abalone	2924	1253	8
Auto-MPG	274	118	7
Auto-Price	111	48	15
Boston Housing	354	152	13
California Housing	14448	6192	8
Delta-Ailerons	4990	2139	5
Delta-Elevators	6662	2855	6

are shown in Table 6.5 and Table 6.4. While training SVR, *LIBSVM*[58] was carried out and $\epsilon-$SVR algorithm was used. Since SVR is deterministic, that the training results only relied on setting of hyperparameters (penalty C, kernel parameter γ and ϵ in loss function), no statistics were recorded. Instead, grid search technique was used to determine the optimal setting of these hyperparameters[57]. The penalty C was searched from 1 to 10^4 with scale step $\sqrt[4]{10}$; The parameter of sigmoid kernel γ was searched from 10^{-4} to 1 with the same scale step; ϵ was searched in [0.1, 0.05, 0.02]. The optimal combination of these hyperparameters and the corresponding testing error for each problem are shown in Table 6.4. With the optimal combination of hyperparameters, the number of support vectors (SV) and training time are shown in Table 6.5.

From the comparison results shown in Table 6.4, 6.5, one can observe that by tuning all the parameters, LM-BP and HC1 algorithms can achieve much more compact SLFN compared to ELM and SVR while reaching similar accuracy. Because SVR only selects support vectors from discrete space (a training patterns pool), the size of the result SLFN is quite large. Because HC1 algorithm starts the training from a SLFN with 0 hidden neuron, the early stage of training is quite fast. Therefore, comparing with LM-BP for training the SLFN with same size, the proposed HC1 algorithm worked more efficiently and cost less computation time, as shown in Table 6.5. Since HC1 escaped from the previous local minimum by adding one more neuron (increasing the optimization space), it's more robust compared to the LM-BP for seeking the global optimal parameters.

Table 6.4: Testing RMSE and the Corresponding Standard Deviation(std) Comparison While Approximating UCI data sets

Algorithms	LM-BP		ELM		SVR		HC1	
	mean	std	mean	std	(C, γ, ϵ)	RMSE	mean	std
Abalone	0.0727	0.0015	0.0743	0.0007	(56.2, 10^{-3}, 0.05)	0.0791	0.0725	0.0018
Auto-MPG	0.0843	0.0098	0.0835	0.0044	(10^4, 0.0005, 0.1)	0.0849	0.0836	0.0050
Auto-Price	0.1362	0.0502	0.1365	0.0208	(17.7828, 0.0056, 0.05)	0.1146	0.1284	0.0247
Boston Housing	0.1042	0.0244	0.1181	0.0097	(316.2278, 0.0018, 0.1)	0.1037	0.1038	0.0096
California Housing	0.1255	0.0115	0.1265	0.0015	(316.2278, 0.0018, 0.1)	0.1410	0.1217	0.0022
Delta Ailerons	0.0389	0.0002	0.0388	0.0002	(10^2, 10^{-3}, 0.05)	0.0392	0.0385	0.0004
Delta Elevators	0.0540	0.0002	0.0544	0.0001	(316.2, 0.0003, 0.05)	0.0537	0.0539	0.0002

Table 6.5: Optimal Number of Hidden Neurons and the Average Training Time While Approximating UCI data sets

Algorithms	LM-BP		ELM		SVR*		HC1 (average)	
	# of hidden neurons	Training Time (s)	# of hidden neurons	Training Time (s)	# of SVs	Training Time (s)	# of hidden neurons	Training Time (s)
Abalone	4	0.70	40	0.01	1246	0.3545	3.97	0.50
Auto-MPG	4	0.09	40	0.001	72	0.0515	3.53	0.07
Auto-Price	3	0.03	7	0.0003	43	0.0012	2.31	0.02
Boston Housing	3	0.11	40	0.001	96	0.0040	3.34	0.12
California Housing	4	4.18	80	0.11	5644	17.03	4.02	2.66
Delta Ailerons	4	1.07	50	0.02	829	0.2994	3.75	0.59
Delta Elevators	4	1.33	60	0.04	2066	1.1855	4.36	1.11

* Core computation of LIBSVM is implemented in C, which is usually much faster than MATLAB.

6.2 Experiments on the HC2 Algorithm for FCCN Construction

In this section, the second kind of Hybrid Constructive Algorithm (HC2), which uses OLS and PSO to search the optimal initialization of the new neuron each time, is experimented on several classic 2D function approximation benchmarks[25][71] and the Mackey-Glass time series prediciton problem[72][73] by constructing FCCN. The results are compared with other architecture oriented learning algorithms described in Chapter 3: CasCor

algorithm[61], Cascade2 algorithm[65], OLSCN algorithm[60], Casper algorithm[68]. Since the convergence rate of different algorithms while adding hidden neurons is different, in the experiments, all the algorithms construct the FCCN from empty to the same maximum number of hidden neurons. The other stopping criteria described in Section 5.3 are not used.

All the experiments are carried out on Windows 7 Enterprise 64-bit operating system, Intel Core i5-2500K CPU @ 3.30GHz, 8.00GB RAM, MATLAB R2014a platform.

6.2.1 2D Function Approximation

In this experiment, five classic 2D function approximations[25][71] are given as the benchmark to compare the proposed HC2 algorithm and other learning algorithms. All the five functions are ranged in $[0,1]^2$. The description of the five functions are shown as below, their meshplots are shown in Figure 6.4.

1. Simple interaction function:

$$f^{(1)}(x_1,x_2) = 10.391((x_1-0.4)(x_2-0.6)+0.36)$$

2. Radial function:

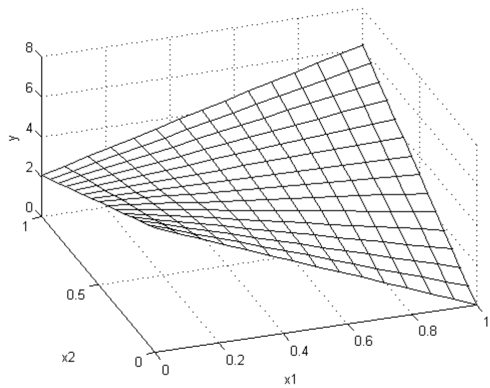
$$f^{(2)}(x_1,x_2) = 24.234(r^2(0.75-r^2))$$

in which,

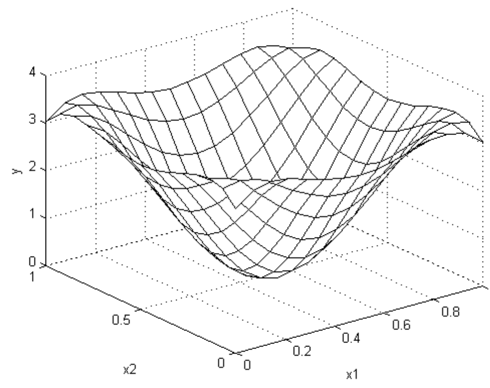
$$r^2 = (x_1-0.5)^2+(x_2-0.5)^2$$

3. Harmonic function:

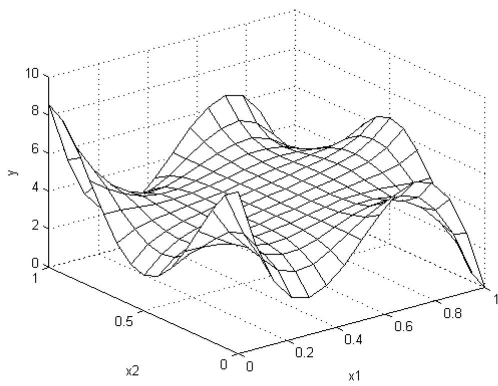
$$f^{(3)}(x_1,x_2) = 42.659(0.1+\tilde{x}_1(0.05+\tilde{x}_1^4-10\tilde{x}_1^2\tilde{x}_2^2+5\tilde{x}_2^4))$$



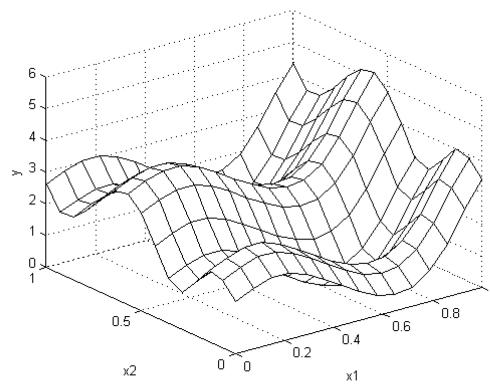
(a) function #1



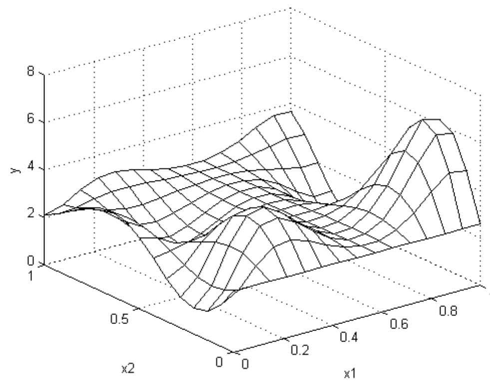
(b) function #2



(c) function #3



(d) function #4



(e) function #5

Figure 6.4: meshplot of the five 2D functions for approximation

in which,

$$\tilde{x}_1 = x_1 - 0.5, \quad \tilde{x}_2 = x_2 - 0.5$$

4. Additive function:

$$
\begin{aligned}
f^{(4)}(x_1, x_2) = 1.3356(1.5(1 - x_1) \\
+ e^{2x_1-1}\sin(3\pi(x_1 - 0.6)^2) \\
+ e^{3(x_2-0.5)}\sin(4\pi(x_2 - 0.9)^2))
\end{aligned}
$$

5. Complicated interaction function:

$$
\begin{aligned}
f^{(5)}(x_1, x_2) = 1.9(1.35 + e^{x_1}\sin(13(x_1 - 0.6)^2) \\
\times e^{-x_2}\sin(7x_2))
\end{aligned}
$$

The setup for training data and testing data are the same for all the five functions. The training data set has 225 patterns, which are generated randomly by the uniform distribution $U[0,1]^2$. All the training data are added independent and identically distributed (i.i.d.) Gaussian noise with mean zero and standard deviation 0.25. The testing data set consists 100×100 patterns generated from a regular spaced grid in the range $[0,1]^2$.

In this experiment, all the algorithms constructed the FCCN by adding hidden neurons from 0 to 20 while the other stopping criteria were not used. For the CasCor and Cascade2, quickprop was used to search the optimal input weights of each new hidden neuron while the maximum iteration was set as 100. In each stage, 8 candidates were trained independently to search the global optimal solution. For the OLSCN algorithm, in which a modified Newton's method was used to train each new neuron's input weights, the maximum iteration was set as 20. The number of candidates was also set as 8. For the Casper algorithm, the SARPROP algorithm[67] was used to tune all the parameters in each stage. All the parameter settings were same as in [68]. The maximum iteration was set as 1000 and the fully training stopped while the root meas squared error (RMSE) decreased less than 1% in 200 continuous iterations. For the HC2 algorithm, while selecting each new neuron, the

maximum iteration and the population size of the PSO were both set as 20; While tuning all the parameters with the hybrid algorithm, the maximum iteration were set as 200 and the parameters in (5.1) were set as $N = 70, \eta = 0.01$. The regularizor in (4.26)(4.28) was set as $\lambda = 0.001$. All the parameter settings were the same while approximating all the five functions.

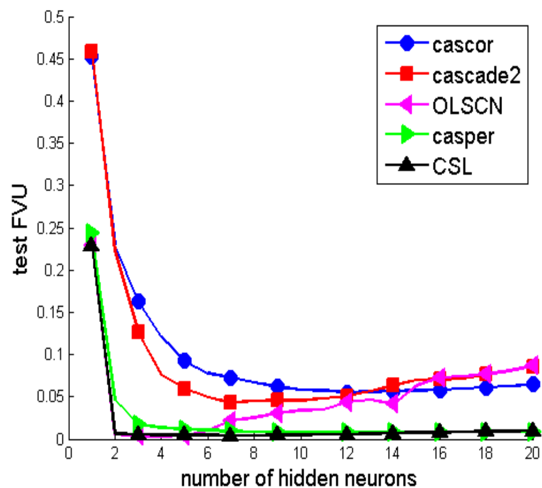
The generalization performance of each algorithm is evaluated by the fraction of variance unexplained (FVU)[71] on the testing data, which is actually proportional to the SSE defined in (1.4),

$$\text{FVU} = \frac{(\mathbf{y} - \tilde{\mathbf{y}})^T(\mathbf{y} - \tilde{\mathbf{y}})}{\sum_{p=1}^{P}(y_p - \bar{y})^2} \tag{6.4}$$

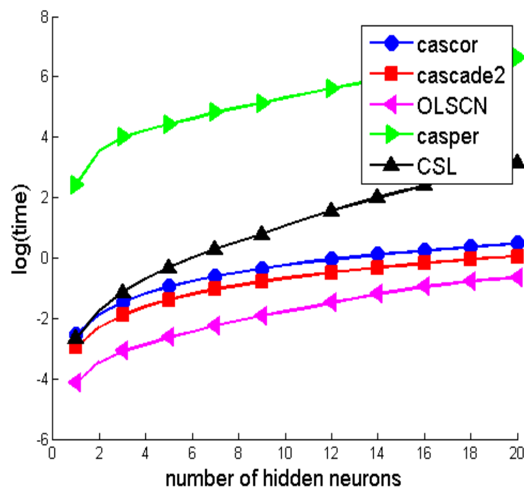
in which, $\mathbf{y} = [y_1, y_2, ..., y_P]^T \in R^P$ are the desired outputs for the testing patterns. $\tilde{\mathbf{y}}$ are the actual outputs with the trained FCCN. $\bar{y}$ is the average value of the desired outputs for all the testing patterns.

$$\bar{y} = \frac{1}{P}\sum_{p=1}^{P} y_p \tag{6.5}$$

Since all the algorithms started from randomly initialized parameters, the construction with each algorithm was repeated 20 times. The averaged testing FVU and training time of each algorithm while approximating the five functions were shown in Figure 6.5-6.9. From the testing FVU comparison, one can observe that the Casper algorithm and the proposed HC2 algorithm could always achieved better generalization than the CasCor and Cascade2 algorithms. Though the OLSCN converged fast, it tended to overfit the data as the number of hidden neuron increased. The HC2 algorithm performed similar to the Casper algorithm. However, benefit from the efficient second order hybrid algorithm while fully tuning all the parameters, which required much less iterations to converged, the HC2 saved much training time compared to the Casper algorithm.

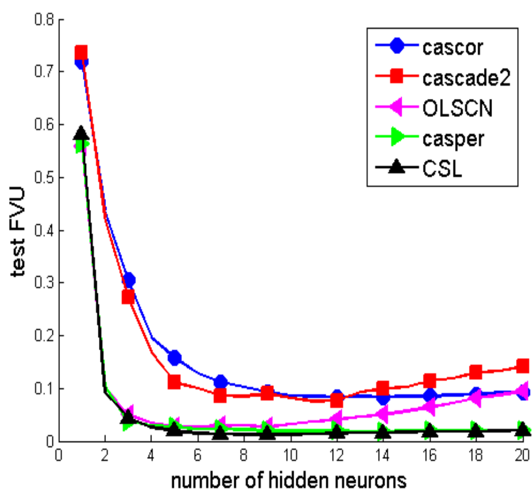


(a) Averaged Testing FVU

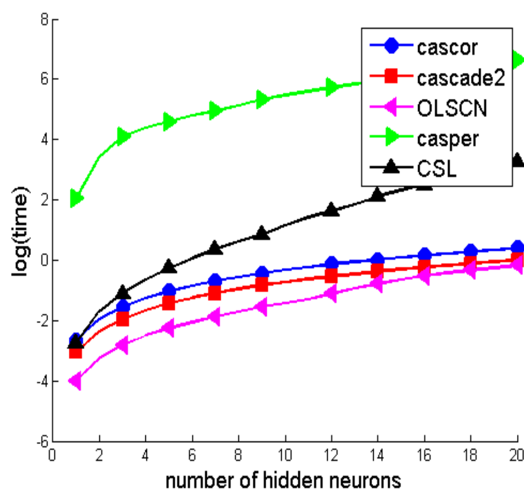


(b) Averaged Training Time

Figure 6.5: Averaged Testing FVU and Training Time While Approximating Function #1



(a) Averaged Testing FVU

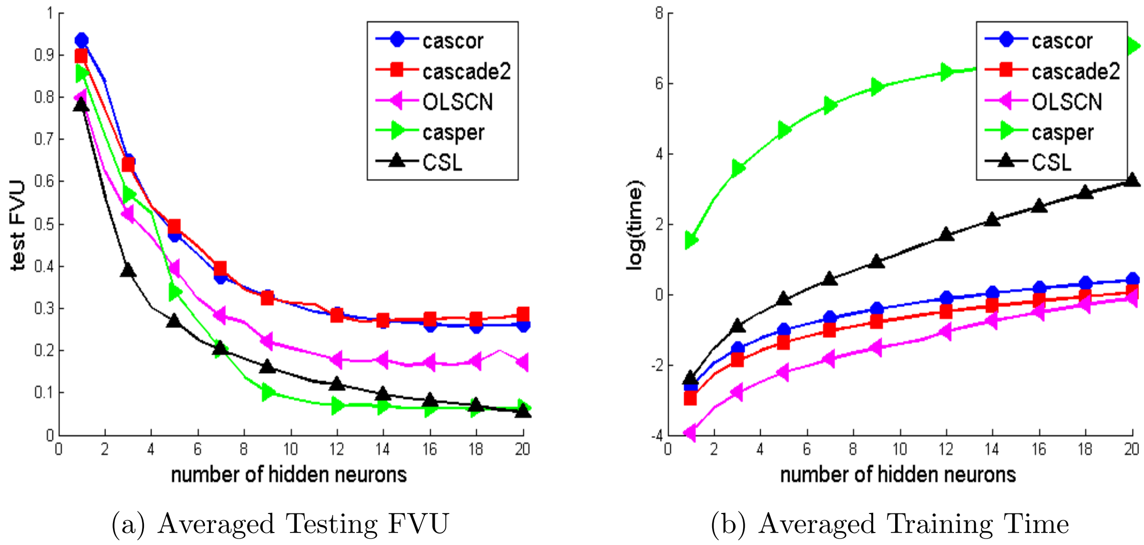


(b) Averaged Training Time

Figure 6.6: Averaged Testing FVU and Training Time While Approximating Function #2

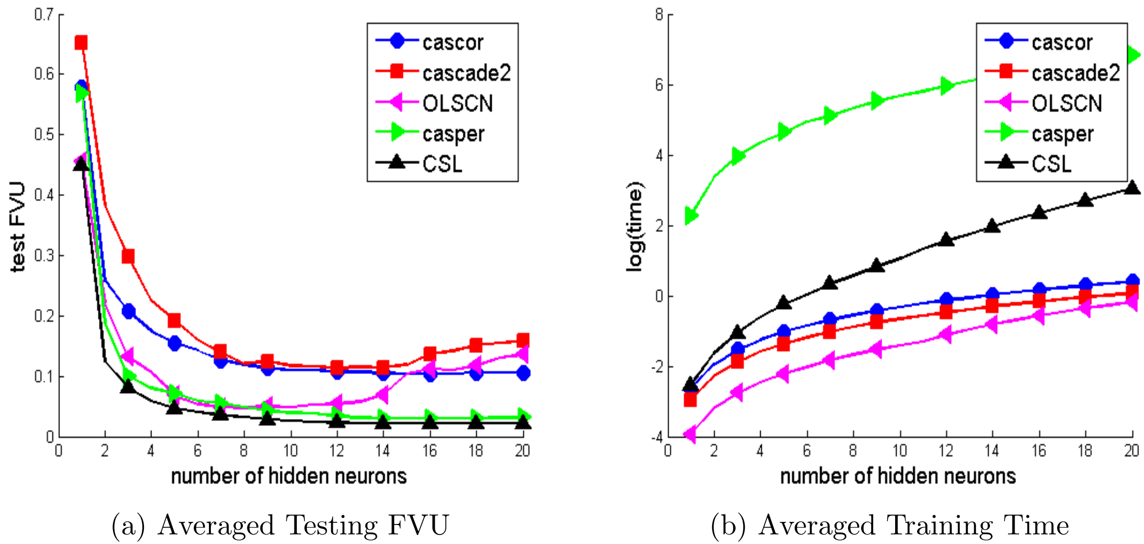
6.2.2 Mackey-Glass Time Series Prediction

In this experiment, the Mackey-Glass chaotic time series prediction problem was given to compare the FCCN construction algorithms[72]-[75]. The time series data set are generated



(a) Averaged Testing FVU (b) Averaged Training Time

Figure 6.7: Averaged Testing FVU and Training Time While Approximating Function #3



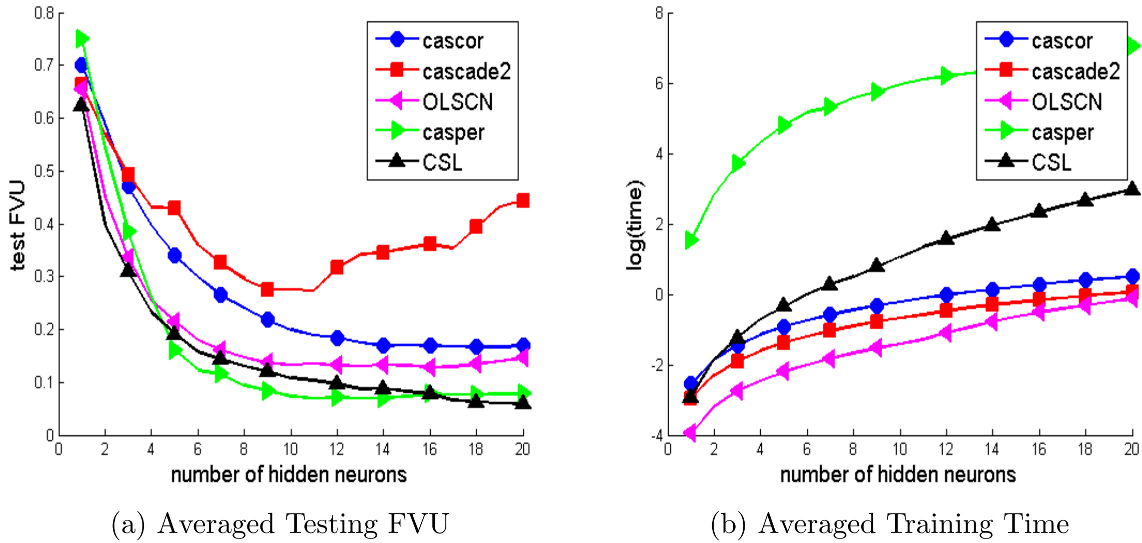
(a) Averaged Testing FVU (b) Averaged Training Time

Figure 6.8: Averaged Testing FVU and Training Time While Approximating Function #4

by following the following differential equation,

$$\dot{x}(t) = \frac{ax(t-\tau)}{1 + x(t-\tau)^{10}} - bx(t) \tag{6.6}$$

where $a = 0.2$, $b = 0.1$ and $\tau = 17$. x(t) is quasiperiodic and chaotic with a fractal attractor dimension 2.1 for the above parameters[74]. The prediction scheme of the Mackey-Glass



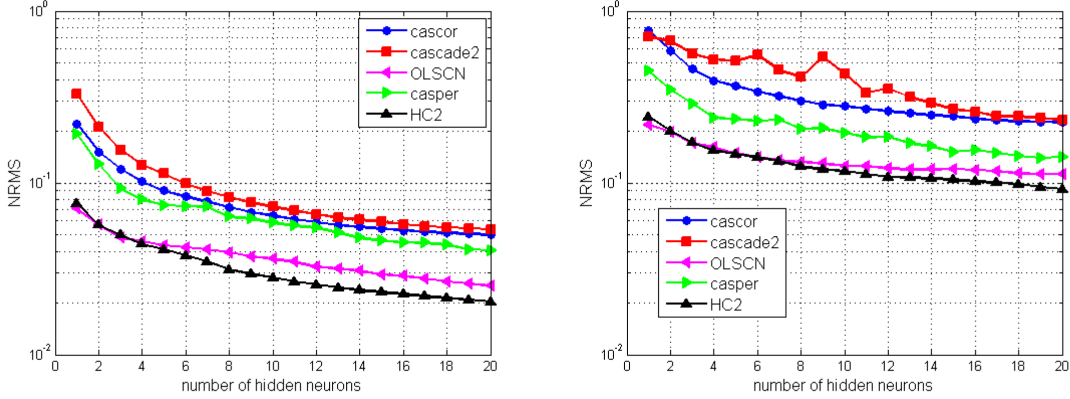
(a) Averaged Testing FVU (b) Averaged Training Time

Figure 6.9: Averaged Testing FVU and Training Time While Approximating Function #5

time series is usually performed by using $x(t-18)$, $x(t-12)$, $x(t-6)$ and $x(t)$ as inputs and $x(t+\Delta t)$ as output, where $\Delta t = 6$ is the basic short term prediction. In order to predict longer term, like $\Delta t = 90$, one has to start from $x(t-18)$, $x(t-12)$, $x(t-6)$ and $x(t)$ to predict $x(t+6)$, $x(t+12)$, ..., $x(t+84)$ iteratively to obtain $x(t+90)$. As a result, the longer the prediction term is, the lower generalization accuracy will be.

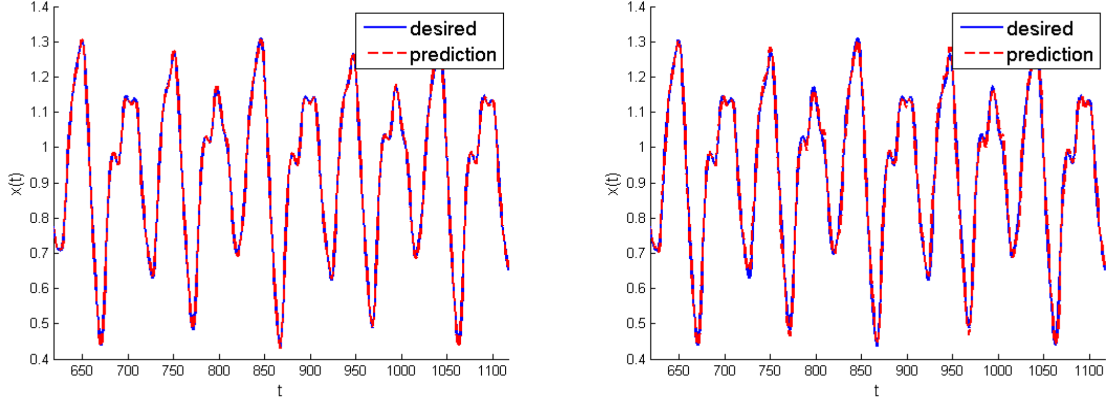
Both the training data and the testing data were obtained by applying the fourth-order Runge-Kutta method to (6.6) with initial condition $x(0) = 1.2$, $x(t-\tau) = 0$ for $0 \leq t < \tau$ and the time step is 1[74]. The state $x(t)$ at $t = 118 \sim 617$ were used as training targets (500 points) while the state at $t = 618 \sim 1117$ were used as testing target (500 points). For each training target, the corresponding $x(t-24)$, $x(t-18)$, $x(t-12)$ and $x(t-6)$ were used as inputs. After training, we performed two tests: short term ($\Delta t = 6$) and long term ($\Delta t = 90$), to evaluate the quality the FCCN for prediction.

The prediction accuracy performance was evaluated by computing the normalized root mean squared error (NRMS)[74][75] on the testing data, which is actually the squared root



(a) Short term prediction NRMS ($\Delta t = 6$) (b) Long term prediction NRMS ($\Delta t = 90$)

Figure 6.10: Generalization performance for Mackey-Glass time series prediction



(a) Short term prediction ($\Delta t = 6$) (b) Long term prediction ($\Delta t = 90$)

Figure 6.11: Best prediction results obtained by HCL among 20 trials: a FCCN with 16 hidden neurons, short term NRMS=0.0212, long term NRMS=0.0707.

of the FVU shown in (6.4),

$$\mathrm{NRMS} = \frac{\sqrt{(\mathbf{y} - \tilde{\mathbf{y}})^T(\mathbf{y} - \tilde{\mathbf{y}})/P}}{\sqrt{\sum_{p=1}^{P}(y_p - \bar{y})^2/P}} = \sqrt{\mathrm{FVU}} \tag{6.7}$$

All the parameters setting of each algorithm are same as in the previous experiment. Each algorithm constructed the FCCN by adding hidden neurons from 0 to 20. Each construction was repeated 20 times. The averaged NRMS for both short term testing and long term testing by each algorithm are shown in Figure 6.10. From the comparison, the HC2

and the OLSCN converged faster than other algorithms. The HC2 could achieve slightly better generalization performance than the OLSCN on both short term prediction and long term prediction. Figure 6.11 shows the prediction performance of the best FCCN trained by the HC2 among the 20 trials. One can observe that the FCCN could predict well even for the long term.

Chapter 7

Conclusions and Discussion

In this dissertation, we have a thorough review of the learning algorithms of two special architectures of the FNN, SLFN and FCCN, for regression problems. While solving regression problems, these two architectures have much in common and their most learning algorithms could share to each other. The current learning algorithms for the SLFN and FCCN could be generally divided into three categories:

1. Use trial and error approach to search the optimal network size. For each specific size, some general gradient learning algorithm is used to tune the parameters.
2. Tuning the parameters and altering network size simultaneously. Typical examples are those constructive algorithms in the Chapter 3.
3. Other algorithms, like SVR.

While using trial and error approach, though the parameters tuning algorithm worked efficiently, a lot of independent trials would slow down the training process. The second type, the constructive learning algorithms constructed the FNN efficiently. However, most of them used freezing strategy so that each time only part of the parameters are tuned during construction. As a result, the result network is usually much larger than required. For the SVR, since all the hidden parameters are searched in discrete space, the solution is not optimal.

In this dissertation, several new hybrid constructive learning algorithms for SLFN and FCCN are proposed. Firstly, a hybrid algorithm for fixed size SLFN or FCCN is proposed by combining the LS method and the LM algorithm. Then the hybrid algorithm is extended to the construction scheme. The training starts from an empty SLFN or FCCN and adds

hidden neurons one by one, each time the hybrid algorithm is used to tune all the parameters. Two construction versions are proposed together. One could generate random parameters for each added neuron before the hybrid tuning. One could also used a OLS-PSO algorithm to select a good initialization of the new added neuron. The experiments demonstrated the efficiency of the hybrid constructive learning algorithms.

Bibliography

[1] Warren McCulloch, Walter Pitts, "A Logical Calculus of Ideas Immanent in Nervous Activity", *Bulletin of Mathematical Biophysics*, 5(4), pp. 115133, 1943.

[2] Donald Hebb, The Organization of Behavior. emphNew York: Wiley, 1949.

[3] M. Minsky and S. Papert, *Perceptrons*. Oxford, England: M. I. T. Press, 1969.

[4] Teuvo Kohonen, "Self-Organized Formation of Topologically Correct Feature Maps", *Biological Cybernetics* 43 (1), pp. 5969, 1982.

[5] G. E. Hinton., "Learning multiple layers of representation," *Trends in Cognitive Sciences*, 11, pp. 428434, 2007.

[6] G. E. Hinton, S. Osindero, Y. Teh, "A fast learning algorithm for deep belief nets", *Neural Computation* 18 (7), pp. 15271554, 2006.

[7] D. C. Ciresan et al., "Deep Big Simple Neural Nets for Handwritten Digit Recognition," *Neural Computation*, 22, pp. 32073220, 2010.

[8] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, D. E. Rumelhart and J. L. McClelland, "Learning internal representations by error propagation", *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, vol. 1, 1986. MIT Press.

[9] K. Hornik, "Approximation capabilities of multilayer feedforward networks", *Neural Netw.*, vol. 4, pp.251 -257 1991.

[10] M. Leshno, V. Y. Lin, A. Pinkus and S. Schocken, "Multilayer feedforward networks with a nonpolynomial activation function can approximate any function", *Neural Netw.*, vol. 6, pp.861 -867 1993.

[11] J. Park and I. W. Sandberg, "Universal approximation using radial-basis-function networks", *Neural Comput.*, vol. 3, pp.246 -257 1991.

[12] B. Lin, B. Lin, F. Chong, and F. Lai, "Higher-order-statistics-based radial basis function networks for signal enhancement," *IEEE Trans. Neural Netw.*, vol. 18, no. 3, pp. 823832, May 2007.

[13] N. Xie and H. Leung, "Blind equalization using a predictive radial basis function neural network," *IEEE Trans. Neural Netw.*, vol. 16, pp. 709720, May 2005.

[14] H. Leung, T. Lo, and S. Wang, "Prediction of noisy chaotic time series using an optimal radial basis function neural network," *IEEE Trans. Neural Netw.*, vol. 12, no. 5, pp. 11631172, Sep. 2001.

[15] C. C. Min, D. Srinivasan and R. Cheu, "Neural networks for continuous online learning and control", *IEEE Trans. Neural Netw.*, vol. 17, no. 5, pp.1511 -1531 2006.

[16] S. E. Fahlman, D. S. Touretzky, G. Hinton, and T. Sejnowski, "Fast learning variations on backpropagation: An empirical study", *Proc. 1988 Connectionist Models Summer School*, pp.38 -51 1988.

[17] M. Riedmiller and H. Braun, "A direct adaptive method for faster backpropagation learning: The RPROP algorithm," In H. Ruspini, editor, *Proceeding of the IEEE International Conference on Neural Networks (ICNN)*, pp. 586-591, San Francisco, 1993.

[18] C. Charalambous, "Conjugate gradient algorithm for efficient training of artificial neural networks", *Proc. Inst. Elect. Eng.*, vol. 139, no. 3, pp.301-310, 1992.

[19] K. Levenberg, "A method for the solution of certain problems in least squares," *Quart. Appl. Math.*, vol. 54, pp. 164168, 1944.

[20] D. Marquardt, "An algorithm for least-squares estimation of nonlinear parameters," *SIAM J. Appl. Math.*, vol. 11, pp. 431441, 1963.

[21] M. T. Hagan and M. Menhaj, "Training feedforward networks with the marquardt algorithm," *IEEE Trans. Neural Networks*, vol. 5, pp. 989993, 1994.

[22] R. Battiti and F. Masulli, "BFGS optimization for faster automated supervised learning," in *Int. Neural-Network Conf.*, vol. 2, 1990, pp.757760.

[23] B. M. Wilamowski, H. Yu, "Improved Computation for Levenberg Marquardt Training," *IEEE Trans. on Neural Networks*, vol. 21, no. 6, pp. 930-937, June 2010.

[24] T. Y. Kwok and D. Y. Yeung, "Constructive algorithms for structure learning in feedforward neural networks for regression problems," *IEEE Trans. Neural Netw.*, vol. 8, no. 3, pp. 630645, May 1997.

[25] T. Y. Kwok and D. Y. Yeung, "Objective functions for training new hidden units in constructive neural networks," *IEEE Trans. Neural Netw.*, vol. 8, no. 5, pp. 11311148, Sep. 1997.

[26] C. R. Rao and S. K. Mitra, *Generalized Inverse of Matrices and its Applications.* New York: Wiley, 1971.

[27] G.-B. Huang, Q.-Y. Zhu, and C.-K. Siew, "Extreme Learning Machine: A New Learning Scheme of Feedforward Neural Networks," *2004 International Joint Conference on Neural Networks(IJCNN'2004)*, (Budapest, Hungary), July 25-29, 2004.

[28] G.-B. Huang, L. Chen and C.-K. Siew, "Universal Approximation Using Incremental Constructive Feedforward Networks with Random Hidden Nodes," *IEEE Transactions on Neural Networks*, vol. 17, no. 4, pp. 879-892, 2006.

[29] G.-B. Huang and L. Chen, "Enhanced random search based incremental extreme learning machine", *Neurocomputing*, vol. 71, no. 16-18, pp.3460-3468, Oct. 2008.

[30] G.-B. Huang and L. Chen, "Convex incremental extreme learning machine", *Neurocomputing*, vol. 70, no. 16-18, pp.3056-3062, Oct. 2007.

[31] G. Feng, G. B. Huang , Q. Lin and R. Gay, "Error minimized extreme learning machine with growth of hidden nodes and incremental learning", *IEEE Trans. Neural Netw.*, vol. 20, no. 8, pp.1352 -1357 2009.

[32] Rui Zhang, Yuan Lan, Guang-bin Huang, Zong-Ben Xu, "Universal Approximation of Extreme Learning Machine With Adaptive Growth of Hidden Nodes", *Trans. on Neural Networks and Learning Systems*, Vol. 23, Issue. 2, Jan 2012.

[33] N. Wang, M. J. Er, M. Han, "Generalized Single-Hidden Layer Feedforward Networks for Regression Problems", *Trans. on Neural Networks and Learning Systems*, Vol. PP, Issue 99, Jul 2014.

[34] N. Wang, M. J. Er, M. Han, "Parsimonious Extreme Learning Machine Using Recursive Orthogonal Least Squares", *Trans. on Neural Networks and Learning Systems*, Vol. PP, Issue. 99, Jan 2014.

[35] S. Chen, S. A. Billings and W. Luo, "Orthogonal least squares methods and their application to non-linear system identification", *Int. J. Control*, vol. 50, pp.1873 -1896 1989.

[36] S. Chen, C. Cowan, and P. M. Grant, "Orthogonal least squares learning algorithm for radial basis function networks," *IEEE Trans. Neural Netw.*, vol. 2, no. 2, pp. 302309, Mar. 1991.

[37] S. Chen, X. Hong, B. L. Luk and C. J. Harris, "Construction of tunable radial basis function networks using orthogonal forward selection", *IEEE Trans. Syst., Man, Cybern. B*, vol. 39, no. 2, pp.457-466, Apr. 2009.

[38] Long Zhang, Kang Li, Haibo He and Irwin, G. W., "A New Discrete-Continuous Algorithm for Radial Basis Function Networks Construction", *IEEE Trans. Neural Networks and Learning Systems*, vol. 24, no. 11, pp. 1785 - 1798, Jun. 2013.

[39] B. M. Wilamowski and H. Yu, "Neural Network Learning Without Backpropagation," *IEEE Trans. on Neural Networks*, vol. 21, no.11, pp1793-1803, Nov. 2010.

[40] T. Xie, H. Yu, J. Hewlett, P. Rozycki, B. Wilamowski, "Fast and Efficient Second Order Method for Training Radial Basis Function Networks", *IEEE Transactions on Neural Networks*, 2012, Vol. 24, Issue: 4, pp. 609-619.

[41] D. Hunter, Hao Yu, M.S. Pukish, J. Kolbusz, B. M. Wilamowski, "Selection of Proper Neural Network Sizes and ArchitecturesA Comparative Study", *Industrial Informatics, IEEE Transactions on*, pp. 228 - 240, vol 8, no. 2, May 2012.

[42] S. McLoone, M. D. Brown, G. W. Irwin, and G. Lightbody, "A hybrid linear/nonlinear training algorithm for feedforward neural networks," *IEEE Trans. Neural Netw.*, vol. 9, no. 4, pp. 669684, Jul. 1998.

[43] H. Peng , T. Ozaki , V. Haggan-Ozaki and Y. Toyoda, "A parameter optimization method for radial basis function type models", *IEEE Trans. Neural Netw.*, vol. 14, no. 2, pp.432 -438 2003.

[44] J. X. Peng, K. Li, and G. W. Irwin, "A new Jacobian matrix for optimal learning of single-layer neural networks," *IEEE Trans. Neural Netw.*, vol. 19, no. 1, pp. 119129, Jan. 2008.

[45] J. Kennedy, R.C. Eberhart, "Particle swarm optimization", *Proc. of IEEE Int. Conf. on Neural Network*, Perth, Australia, 1995, pp. 19421948.

[46] W.B. Langdon, Riccardo Poli, "Evolving problems to learn about particle swarm optimizers and other search algorithms", *IEEE Trans. Evol. Comput.* 11 (5) (2007) 561578.

[47] Zhang, C., Shao, H., and Li, Y. "Particle swam optimisation for evolving artificial neural network", *Proceedings of the IEEE lnternational Conference on Systems, Man, and Cybernetics*, 2000, pp. 2487-2490.

[48] Gudise, V.G., Venayagamoorthy, G.K., "Comparison of particle swarm optimization and backpropagation as training algorithms for neural networks", *Proceedings of the 2003 IEEE Swarm Intelligence Symposium,. SIS '03*, April 24-26, 2003, Page(s): 110-117.

[49] F. Han, H.-F. Yao, and Q.-H. Ling, "An improved extreme learning machine based on particle swarm optimization", in *Bio-Inspired Computing and Applications*, ser. Lecture Notes in Computer Science.

[50] R. Mendes, P. Cortez, M. Rocha, and J. Neves, "Particle swarms for feedforward neural network training," *Proc. Inl. Joint Conf. on Neural Networks*, pp. 1895-1899, 2002.

[51] Y. Shi, R.C. Eberhart, "A modied particle swarm optimizer", *Proceedings of IEEE World Conference on Computation Intelligence*, 1998, pp. 6973.

[52] R.C. Eberhart, Y. Shi, "Parameter selection in particle swarm optimization", *Evolutionary Programming VII, Lecture Notes in Computer Science*, Springer, 1998, pp. 591600.

[53] Y. Shi and R. C. Eberhart, "Empirical Study of Particle Swarm Optimization," *Proceedings of the 1999 Congress on Evolutionary Computation*, pp. 1945-1950, Piscataway, 1999.

[54] Bache, K. and Lichman, M. (2013). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.

[55] Hsu, C. W., Chang, C. C. and Lin C. J., "A Practical Guide to Support Vector Classification", *Technical report* , Department of Computer Science, National Taiwan University, 2003.

[56] V. N. Vapnik, *Statistical Learning Theory*, New York, Wiley, 1998.

[57] A. Smola and B. Schlkopf, "A tutorial on support vector regression", *Statistics and Computing*, vol. 14, pp. 199-222, 2004.

[58] C.-C. Chang and C.-J. Lin. *LIBSVM*: a library for support vector machines, 2001. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[59] P.L. Bartlett, "The sample complexity of pattern classication with neural networks: the size of the weights is more important than the size of the network", *IEEE Trans. Inf. Theory*, 1998, pp. 525536.

[60] G. Huang, S. Song, and C. Wu, "Orthogonal least squares algorithm for training cascade neural networks," *IEEE Trans. Circuits Syst. I, Reg.Papers*, vol. 59, no. 11, pp. 26292637, 2012.

[61] S. E. Fahlman and C. Lebiere, "The cascade-correlation learning architecture," in Advances in *Neural Information Processing Systems 2*, D. S. Touretzky, Ed. San Mateo, CA: Morgan Kaufmann, 1990, pp. 524532.

[62] T. K. Kwok and D. Y. Young, "Experimental analysis of input weight freezing in constructive neural networks," *in Proc. IEEE Int. Conf. Neural Networks*, San Francisco, CA, Mar. 1993, pp. 511516.

[63] Baluja, S., Fahlman, S., "Reducing Network Depth in the Cascade-Correlation Learning Architecture", *Tech. Rep.*, Pittsburgh, PA: Carnegie Mellon University.

[64] Sjogaard, S. "A Conceptual Approach to Generalisation in Dynamic Neural Networks", *PhD thesis*, Aarhus University, Aarhus, Danmark, 1991.

[65] L. Prechelt, "Investigating the cascor family of learning algorithms", *Neural Networks*, vol. 10, no. 5, pp.885-896, 1997.

[66] J. N. Hwang, S. S. You, S. R. Lay, and I. C. Jou, "The cascade-correlation learning: A projection pursuit learning perspective", *IEEE Trans. Neural Networks*, vol. 7, pp.278-289, 1996.

[67] N. K. Treadgold and T. D. Gedeon, "Simulated annealing and weigh decay in adaptive learning: The sarprop algorithm," *IEEE Trans. Neural Networks*, vol. 9, pp. 662668, 1998.

[68] N.K. Treadgold and T.D. Gedeon, "A Cascade Network Employing Progressive RPROP", *Int. Work Conf. on Artificial and Natural Neural Networks*, pp. 733-742, 1997.

[69] N.K. Treadgold and T.D. Gedeon, "Exploring constructive cascade networks," *IEEE Trans. Neural Networks*, vol. 10, pp. 1335-1350, Nov. 1999.

[70] N.K. Treadgold and T.D. Gedeon, "Extending casper: A regression survey," in *Proc. Int. Conf. Neural Inform. Processing*, Nov. 1997, pp. 310313.

[71] J. N. Hwang, S. R. Lay, M. Maechler, D. Martin, and J. Schimert, "Regression modeling in backpropagation and projection pursuit learning," *IEEE Trans. Neural Networks*, vol. 5, pp. 342-353, 1994.

[72] M. Mackey and L. Glass, "Oscillation and chaos in physiological control systems," *Science*, vol. 197, p. 287, 1977.

[73] R. S. Crowder, "Predicting the Mackey-Glass timeseries with cascade-correlation learning," in *Proc. Connectionist Models Summer School*, 1990, pp. 117-123.

[74] Y. Liu and X. Yao, "Simultaneous training of negatively correlated neural networks in an ensemble", *IEEE Trans. Syst., Man, Cybern., B*, vol. 29, no. 6, pp.716 -725, 1999.

[75] T. Y. Kwok and D. Y. Yeung, "Use of bias term in projection pursuit learning improves approximation and convergence properties", *IEEE Trans. Neural Networks*, vol. 7, pp.1168-1183, 1996.

Appendix A

Publications of the Author

- Xing Wu, Pawel Rozycki, Bogdan M. Wilamowski, "A Hybrid Constructive Algorithm for Single Layer Feedforward Networks Learning", *Trans. on Neural Networks and Learning Systems*, vol. PP, no. 99, pp. 1, 2014.

- Xing Wu, Bogdan M. Wilamowski, "A Greedy Incremental Algorithm for Universal Approximation with RBF Newtorks", *Advances in Soft Computing, Intelligent Robotics and Control, Springer*, 2014.

- Xing Wu and Bogdan M. Wilamowski, "Advantage analysis of Sigmoid based RBF Networks", *17-th IEEE Intelligent Engineering Systems Conference, INES 2013*, Costa Rica, June 19-21, 2013, pp. 243-248.

- Pradeep Dandamudi, Timothy A. Brown, Xing Wu, Marcin Jagiela, and Bogdan M. Wilamowski, "A Design Methodology of Lossy Transconductance Filters", *IEEE 38-th Annual Industrial Electronics Conference IECON 2012*, Montreal, Canada, Oct 25-28, 2012, pp. 6228-6233.

- Xing Wu, Hao Yu, Tiantian Xie, Michael S. Pukish, "Current Trends in Industrial Control", *IECON 2012*, Montral, Canada, October 25-28, 2012.

- Michael S. Pukish, Parameshwaran Gnanachchelvi, Xing Wu, "Recent Developments in Wireless Hardware Design, Modeling, and Analysis for Industrial Applications", *IECON 2012*, Montral, Canada, October 25-28, 2012.

- Michael S. Pukish, Philip Reiner, Xing Wu, "Recent Advances in the Application of Real-Time Computational Intelligence to Industrial Electronics", *IECON 2012*, Montral, Canada, October 25-28, 2012.

Appendix B

MATLAB Programs for the Proposed Algorithms

B.1 File List

1. **initialize.m** Initialize the network object by setting the network's type, activation function, number of hidden neurons, etc. Return a network object with following attributes:
   (a) *network.nd*(integer): number of inputs of the network.
   (b) *network.type*("SLFN" ‖ "FCCN"): network type.
   (c) *network.activation*("sigmoid" ‖ "tanh" ‖ "rbf"): activation function of each hidden neuron.
   (d) *network.K*(integer): number of hidden neurons.
   (e) *network.wi*(matrix or cell array):
      i. If network is SLFN and hidden neurons are "sigmoid" or "tanh", *wi* is (nd+1) × K matrix, first row are bias of each hidden neuron.
      ii. If network is SLFN and hidden neurons are "rbf", *wi* is (nd+1) × K matrix, first row is the width of each RBF unit, the rest of each column are the center of each RBF unit.
      iii. If network is FCCN (activation function is "sigmoid" or "tanh"), *wi* is a cell array with K cells, each cell stores the input weights of the corresponding hidden neuron.
   (f) *network.wo*(matrix): output weights of the network. For SLFN, *wo* is (K+1) × 1 matrix; for FCCN, *wo* is (nd+K+1) × 1 matrix.

One can set the initial *wi* and *wo* through arguments, or randomly initialize by default. The widths of the RBF networks are initialized as 2/rand(1); All the other parameters are initialized in the range [-1,1].

2. **forward.m** Calculate forward through the network with the given inputs "X". The hidden matrix "H" is returned. The first column are always 1s as bias. For the SLFN, "H" is an np $\times$ (K+1) matrix; For the FCCN, "H" is an np $\times$ (nd+K+1) matrix.

3. **JQmatrix.m** Calculate nonlinear Jacobian matrix in (4.2) and the Q matrix defined in (4.13) or (4.30).

4. **update.m** Update the nonlinear parameters of the network in each iteration of the training procedure.

5. **train.m** Train fixed-size network with the hybrid algorithm. "X" (np $\times$ nd) is the input matrix, "T" (np $\times$ 1) is the desired outputs vector. "maxite" is the maximum iteration. "lambda" is the regularization factor λ in (4.28). The function returns the "SSE" of each iteration and the network with the trained parameters.

6. **demo1.m** Simple demo to train fixed-size SLFN or FCCN with the proposed hybrid algorithm.

7. **construct.m** Construct SLFN or FCCN with the proposed HC1 or HC2 algorithm. "param" is the object including all the setting parameters as shown in Table B.1. The function returns the "SSE" of each iteration and the network with the trained parameters. It also returns the "tick", which means the iteration number when adding each hidden neuron.

8. **PSO.m** Pick the optimal parameters of each new added neuron with PSO algorithm. It's called when using the HC2 algorithm. "H" is the hidden matrix of the previous network. "X" is the input matrix of the new added neuron. "err" is the

Table B.1: attributes of "param" in "construct.m"

attributes	description
param.maxite	maximum training iteration
param.lambda	regularization factor λ in (4.28)
param.Kmax	maximum number of hidden neurons
param.begin_latency	latency determine when to add new neuron
param.begin_thd	threshold determine when to add new neuron
param.end_latency	latency determine when to stop adding
param.end_thd	threshold determine when to stop adding
param.algo	select algorithm 1 (HC1) or 2 (HC2)
param.PSO_maxite	maximum iteration of PSO
param.PSO_n	number of particles of PSO

error vector of previous network. "maxite" and "n" sets the maximum iteration and particles size for the PSO algorithm.

9. **demo2.m** Simple demo to construct SLFN or FCCN with the proposed HC1 and HC2 algorithm.

B.2 Matlab Codes

Listing B.1: initialize.m

```
function network = initialize(X, type, activation, K, wi, wo)
%% initialize network
if nargin<4
    error('Please provide input data, network type, activation and number
        of hidden neurons!');
```

```
end;
network.nd = size(X,2);
if strcmp(type, 'SLFN') || strcmp(type, 'FCCN')
    network.type = type;
else
    error('network.type should be ''SLFN'' or ''FCCN''!');
end;
if strcmp(type, 'SLFN')
    if strcmp(activation, 'sigmoid') || strcmp(activation, 'tanh') ||
       strcmp(activation, 'rbf')
        network.activation = activation;
    else
        error('SLFN Activation function (network.activation) should be ''
            sigmoid'', ''tanh'' or ''rbf''!');
    end;
else
    if strcmp(activation, 'sigmoid') || strcmp(activation, 'tanh')
        network.activation = activation;
    else
        error('FCCN Activation function (network.activation) should be ''
            sigmoid'' or ''tanh''!');
    end;
end;
network.K = K;
if nargin>=5
    network.wi = wi;
else
    if strcmp(type, 'SLFN')
        if strcmp(activation, 'sigmoid') || strcmp(activation, 'tanh')
            network.wi = 2*rand(network.nd+1, network.K)-1;
        else
            network.wi(1,:) = 2./rand(1,network.K);
```

```
            network.wi(2:network.nd+1,:) = 2*rand(network.nd, network.K)
                -1;
        end;
    else
        for k = 1:network.K
            network.wi{k} = 2*rand(network.nd+k,1)-1;
        end;
    end;
end;

if nargin>=6
    network.wo = wo;
else
    if strcmp(type, 'SLFN')
        network.wo = 2*rand(network.K+1,1)-1;
    else
        network.wo = 2*rand(network.nd+network.K+1,1)-1;
    end;
end;

return;
```

Listing B.2: forward.m

```
function H = forward(network, X)
%% calculate forward to get hidden matrix
np = size(X,1);
if strcmp(network.type, 'SLFN')
    if strcmp(network.activation, 'sigmoid')
        X = [ones(np,1), X];
        net = X*network.wi;
        H = [ones(np,1), 1./(1+exp(-net))];
```

```
    elseif strcmp(network.activation, 'tanh')
        X = [ones(np,1), X];
        net = X*network.wi;
        H = [ones(np,1), tanh(net)];
    elseif strcmp(network.activation, 'rbf')
        H = ones(np,1);
        for k = 1:network.K
            width = network.wi(1,k);
            center = network.wi(2:network.nd+1,k);
            dist = (X - ones(np,1)*center').^2*ones(network.nd,1);
            h = exp(-dist/width^2);
            H = [H, h];
        end;
    else
        error('SLFN Activation function (network.activation) should be ''
            sigmoid'', ''tanh'' or ''rbf''!');
    end;
elseif strcmp(network.type, 'FCCN')
    if strcmp(network.activation, 'sigmoid')
        H = [ones(np,1), X];
        for k = 1:network.K
            net = H*network.wi{k};
            h = 1./(1+exp(-net));
            H = [H, h];
        end;
    elseif strcmp(network.activation, 'tanh')
        H = [ones(np,1), X];
        for k = 1:network.K
            net = H*network.wi{k};
            h = tanh(net);
            H = [H, h];
        end;
    else
```

```
        error('FCCN Activation function (network.activation) should be ''
            sigmoid'' or ''tanh''!');
    end;
else
    error('network.type should be ''SLFN'' or ''FCCN''!');
end;
return;
```

Listing B.3: JQmatrix.m

```
function [J, Q] = JQmatrix(network, X, error)
%% calculate noninear Jacobian matrix and Q matrix
np = size(X,1);
if strcmp(network.type, 'SLFN')
    if strcmp(network.activation, 'sigmoid') || strcmp(network.activation
        , 'tanh')
        X = [ones(np,1), X];
        net = X*network.wi;
        if strcmp(network.activation, 'sigmoid')
            H = 1./(1+exp(-net));
            Hp = H.*(1-H);
        else
            H = tanh(net);
            Hp = (1-H.^2);
        end;
        J = [];
        Q = zeros(network.K+1, network.K*(network.nd+1));
        for k = 1:network.K
            tmp = Hp(:,k)*ones(1,network.nd+1).*X;
            J = [J, network.wo(k+1)*tmp];
            Q(k+1,(k-1)*(network.nd+1)+1:k*(network.nd+1)) = error'*tmp;
        end;
```

```
    elseif strcmp(network.activation, 'rbf')
        J = [];
        Q = zeros(network.K+1, network.K*(network.nd+1));
        for k = 1:network.K
            width = network.wi(1,k);
            center = network.wi(2:network.nd+1,k);
            dist = (X - ones(np,1)*center').^2*ones(network.nd,1);
            h = exp(-dist/width^2);
            dhc = 2/width^2*h*ones(1,network.nd).*(X - ones(np,1)*center
                '); % centers
            dhw = 2/width^3*h.*dist;          % width
            J = [J, network.wo(k+1)*dhw, network.wo(k+1)*dhc];
            Q(k+1,(k-1)*(network.nd+1)+1:k*(network.nd+1)) = error'*[dhw,
                dhc];
        end;
    else
        error('SLFN Activation function (network.activation) should be ''
            sigmoid'', ''tanh'' or ''rbf''!');
    end;
elseif strcmp(network.type, 'FCCN')
    if strcmp(network.activation, 'sigmoid') || strcmp(network.activation
        , 'tanh')
        H = [ones(np,1), X];
        delta = {};
        for k = 1:network.K
            net = H*network.wi{k};
            if strcmp(network.activation, 'sigmoid')
                h = 1./(1+exp(-net));
                delta{k,k} = h.*(1-h);
            else
                h = tanh(net);
                delta{k,k} = 1-h.^2;
            end;
```

```
            H = [H, h];
        end;
        nw = network.K*network.nd+network.K*(network.K+1)/2;
        J = [];  Q = zeros(network.K+network.nd+1,nw);
        for j = 1:network.K                % j_th column
            ni = network.nd + j;
            idx = (j-1)*network.nd+j*(j-1)/2;
            Q(j+network.nd+1,idx+1:idx+ni) = error'*(delta{j,j}*ones(1,ni
                ).*H(:,1:ni));
            for i = j+1:network.K          % i_th row
                tmp = zeros(np,1);
                for k = j:i-1      % all above values in the same column
                    tmp = tmp + network.wi{i}(network.nd+1+k)*delta{k,j};
                end;
                delta{i,j} = delta{i,i}.*tmp;
                Q(i+network.nd+1,idx+1:idx+ni) = error'*(delta{i,j}*ones
                    (1,ni).*H(:,1:ni));
            end;
            tmp = zeros(np,1);
            for i = j:network.K
                tmp = tmp + network.wo(i+1)*delta{i,j};
            end;
            delta{network.K+1,j} = tmp;
            J = [J, delta{network.K+1,j}*ones(1,ni).*H(:,1:ni)];
        end;
    else
        error('FCCN Activation function (network.activation) should be ''
            sigmoid'' or ''tanh''!');
    end;
else
    error('network.type should be ''SLFN'' or ''FCCN''!');
end;
```

Listing B.4: update.m

```
function network = update(network, wi, dWv)
%% update parameters
if strcmp(network.type, 'SLFN')
    network.wi = wi + reshape(dWv, network.nd+1, network.K);
elseif strcmp(network.type, 'FCCN')
    for k = 1:network.K
        network.wi{k} = wi{k} + dWv(1:network.nd+k);
        dWv(1:network.nd+k) = [];
    end;
else
    error('network.type should be ''SLFN'' or ''FCCN''!');
end;
return;
```

Listing B.5: train.m

```
function [SSE, network] = train(network, X, T, maxite, lambda)
%% train network with hybrid algorithm
%  network:
%           network.type = 'SLFN' or 'FCCN'
%           network.nd: number of input dimension
%           network.K:  number of hidden neurons
%           network.activation = 'sigmoid' or 'tanh' or 'rbf'
%                 if 'SLFN' && ('sigmoid'||'tanh'):
%                      network.wi (nd+1 X K)
%                      ------ input weights matrix (1st row for bias)
%                 if 'SLFN' && 'rbf':
%                      network.wi (nd+1 X K)
%                      ------ each column: 1st width, 2:nd+1 center
%                 if 'FCCN' && ('sigmoid'||'tanh'):
%                      network.wi{1,2,...,K}
```

```
%                     ------ input weights for each hidden neuron
%           network.wo:  output weights of the network (1st bias)
%  X (np X nd): input data
%  T (np X 1):  target data

network.nd = size(X,2);

mu = 0.01;
beta = 10;
muH = 1e15;
muL = 1e-15;

H = forward(network, X);
M = inv(H'*H+lambda*eye(size(H,2)));
network.wo = M*H'*T;
y = H*network.wo;
error = T - y;
SSE(1) = error'*error;
for ite = 2:maxite
    [J, Q] = JQmatrix(network, X, error);
    S = J'*H;
    A = J'*J - S*M*S';    B = Q'*M*Q;
    wi = network.wi;
    for i = 1:10
        dWv = (A + B + mu*eye(size(A,1)))\J'*error;
        network = update(network, wi, dWv);
        H = forward(network, X);
        M = inv(H'*H+lambda*eye(size(H,2)));
        network.wo = M*H'*T;
        y = H*network.wo;
        error = T - y;
        SSE(ite) = error'*error;
        if SSE(ite)<=SSE(ite-1)
```

```
            mu = max(muL, mu/beta);
            break;
        else
            mu = min(muH, mu*beta);
        end;
    end;
end;
```

Listing B.6: demo1.m

```
clear all;   format compact;
%% load training data and testing data
% training data:
%          input matrix Ti, output vector Td, number of patterns N
% testing data:
%          input matrix Tit, output vector Tdt, number of patterns Nt
load data

%% initialize the network
network = initialize(Ti, 'SLFN', 'sigmoid', 5);

%% train the network with training data
maxite = 100;
lambda = 0.001;
[SSE, network] = train(network, Ti, Td, maxite, lambda);

%% validate test data on the trained network
H = forward(network, Tit);
y = H*network.wo;
err = Tdt-y;
SSEt= err'*err;    % SSE of test data
```

Listing B.7: construct.m

```
function [SSE, network, tick] = construct(network, X, T, param)
%% construct the network
%   param.maxite         -------- maximum iteration
%   param.lambda         -------- regularization
%   param.Kmax           -------- maximum number of hidden neurons
%   param.begin_latency  -------- latency determine when to add new neuron
%   param.begin_thd      -------- threshold determine when to add new
    neuron
%   param.end_latency    -------- latency determine when to stop adding
%   param.end_thd        -------- threshold determine when to stop adding
%   param.algo           -------- 1 (HC1) or 2 (HC2)
%   param.PSO_maxite     -------- maximum iteration of PSO
%   param.PSO_n          -------- number of particles of PSO
mu = 0.01;  muL = 1e-15;  muH = 1e15;  beta = 10;  % LM parameters

[np, network.nd] = size(X);
%% clear network weights
if strcmp(network.type, 'SLFN')
    H = ones(np, 1);
    err = T - mean(T)*H;
    network.wi = [];
    if param.algo==1          % HC1
        if strcmp(network.activation, 'sigmoid') || strcmp(
           network.activation, 'tanh')
            network.wi = 2*rand(network.nd+1, 1)-1;
        elseif strcmp(network.activation, 'rbf')
            network.wi = [2/rand(1); 2*rand(network.nd, 1)-1];
        else
            error('SLFN Activation function (network.activation) should
                be ''sigmoid'', ''tanh'' or ''rbf''!');
        end;
```

```
    elseif param.algo==2      % HC2
        if strcmp(network.activation, 'sigmoid') || strcmp(
            network.activation, 'tanh')
             network.wi = PSO(network, H, [ones(np,1), X], err,
                 param.PSO_maxite, param.PSO_n);
        else
             network.wi = PSO(network, H, X, err, param.PSO_maxite,
                 param.PSO_n);
        end;
    else
        error('param.algo should be 1 (as HC1 algorithm) or 2 (as HC2
            algorithm)!');
    end;
elseif strcmp(network.type, 'FCCN')
    H = [ones(np,1), X];
    network.wo = pinv(H)*T;
    err = T - H*network.wo;
    network.wi = {};
    if param.algo==1
        if strcmp(network.activation, 'sigmoid') || strcmp(
            network.activation, 'tanh')
             network.wi{1} = 2*rand(network.nd+1, 1)-1;
        else
             error('FCCN Activation function (network.activation) should
                 be ''sigmoid'' or ''tanh''!');
        end;
    elseif param.algo==2
        if strcmp(network.activation, 'sigmoid') || strcmp(
            network.activation, 'tanh')
             network.wi{1} = PSO(network, H, H, err, param.PSO_maxite,
                 param.PSO_n);
        else
```

```
            error('FCCN Activation function (network.activation) should
                be ''sigmoid'' or ''tanh''!');
        end;
    else
        error('param.algo should be 1 (as HC1 algorithm) or 2 (as HC2
            algorithm)!');
    end;
else
    error('network.type should be ''SLFN'' or ''FCCN''!');
end;
network.K = 1;

%% construct
H = forward(network, X);
M = inv(H'*H+param.lambda*eye(size(H,2)));
network.wo = M*H'*T;
y = H*network.wo;
err = T - y;
SSE(1) = err'*err;
clk = 0;   tick = [];
for ite = 2:param.maxite
    clk = clk + 1;
    % start adding neuron criteria
    if clk > param.begin_latency && abs(SSE(ite-param.begin_latency)-SSE(
        ite-1))/SSE(ite-1) < param.begin_thd
        % stop adding neuron criteria
        if network.K > param.Kmax
            break;
        end;
        if network.K > param.end_latency && abs(SSE(tick(network.K-
            param.end_latency))-SSE(tick(network.K-1)))/SSE(tick(network.K
            -1)) < param.end_thd
            break;
```

```
end;
% adding new neuron
if strcmp(network.type, 'SLFN')
    if strcmp(network.activation, 'sigmoid') || strcmp(
       network.activation, 'tanh')
        if param.algo==1
            wi_new = 2*rand(network.nd+1,1)-1;
        else
            wi_new = PSO(network, H, [ones(np,1),X], err,
                param.PSO_maxite, param.PSO_n);
        end;
        network.wi = [network.wi, wi_new];
        net = [ones(np,1),X]*wi_new;
        if strcmp(network.activation, 'sigmoid')
            h = 1./(1+exp(-net));
        else    % tanh
            h = tanh(net);
        end;
    else    % rbf
        if param.algo==1
            width = 2/rand(1);
            center = 2*rand(network.nd,1)-1;
        else    % algo==2
            wi_new = PSO(network, H, X, err, param.PSO_maxite,
                param.PSO_n);
            width = wi_new(1);
            center = wi_new(2:end);
        end;
        network.wi = [network.wi, [width; center]];
        dist = (X-ones(np,1)*center').^2*ones(network.nd,1);
        h = exp(-dist/width^2);
    end;
else    % FCCN
```

```
        if param.algo==1
            network.wi{network.K+1} = 2*rand(network.nd+network.K
                +1,1)-1;
        else
            network.wi{network.K+1} = PSO(network, H, H, err,
                param.PSO_maxite, param.PSO_n);
        end;
        net = H*network.wi{network.K+1};
        if strcmp(network.activation, 'sigmoid')
            h = 1./(1+exp(-net));
        else  % tanh
            h = tanh(net);
        end;
    end;
    network.K = network.K + 1;
    woh = M*H'*h;      errh = h - H*woh;
    a = h'*err + param.lambda;     b = (h'*err)/a;
    M = [M + woh*woh'/a, -woh/a; -woh'/a, 1/a];
    network.wo = [network.wo - b*woh; b];
    err = err - b*errh;
    H = [H, h];

    clk = 0;
    mu = 0.01;  % set back initial mu
    tick = [tick, ite];
end;
[J, Q] = JQmatrix(network, X, err);
S = J'*H;
A = J'*J - S*M*S';    B = Q'*M*Q;
wi = network.wi;
for i = 1:20
    dWv = (A + B + mu*eye(size(A,1)))\J'*err;
    network = update(network, wi, dWv);
```

```
        H = forward(network, X);
        M = inv(H'*H+param.lambda*eye(size(H,2)));
        network.wo = M*H'*T;
        y = H*network.wo;
        err = T - y;
        SSE(ite) = err'*err;
        if SSE(ite)<=SSE(ite-1)
            mu = max(muL, mu/beta);
            break;
        else
            if mu==muH
                break;
            end;
            mu = min(muH, mu*beta);
        end;
    end;
end;
return;
```

Listing B.8: PSO.m

```
function gw = PSO(network, H, X, err, maxite, n)
%% PSO initialize parameters of each new neuron
c2 = 2;   c3 = 2;
c1s = 0.9;  c1e = 0.4;

np = size(H,1);
Hs = pinv(H);
if strcmp(network.activation, 'sigmoid') || strcmp(network.activation, '
    tanh')
    nd = size(X,2);
    blo = -10*ones(nd,1);
```

```
    bup = 10*ones(nd,1);
elseif strcmp(network.activation, 'rbf')
    nd = size(X,2)+1;
    blo = -10*ones(nd,1);   blo(1) = 0.001;
    bup = 10*ones(nd,1);
else
    error('network.activaiton should be ''sigmoid'', ''tanh'' or ''rbf''!
        ');
end;

ws = 2*rand(nd,n)-1;    vs = 2*rand(nd,n)-1;
pw = ws;       gw = zeros(nd,1);
pbest = zeros(1,n);     gbest = 0;
%% initialize particles
for i = 1:n
    if strcmp(network.activation, 'sigmoid')
        net = X*ws(:,i);
        h = 1./(1+exp(-net));
    elseif strcmp(network.activation, 'tanh')
        net = X*ws(:,i);
        h = tanh(net);
    else
        width = ws(1,i);
        center = ws(2:end,i);
        dist = (X-ones(np,1)*center').^2*ones(nd-1,1);
        h = exp(-dist/width^2);
    end;
    wh = Hs*h;    eh = h - H*wh;
    dSSE = (h'*err)^2/(h'*eh);
    pbest(n) = dSSE;
    if dSSE>gbest
        gbest = dSSE;
        gw = ws(:,i);
```

```
    end;
end;
%% optimization
for ite = 1:maxite
    c1 = c1e + ite/maxite*(c1s-c1e);
    for i = 1:n
        rp = rand(nd,1);   rg = rand(nd,1);
        % update velocity
        vs(:,i) = c1*vs(:,i) + c2*rp.*(pw(:,i)-ws(:,i)) + c3*rg.*(gw-ws
            (:,i));
        vs(:,i) = max(vs(:,i), blo);
        vs(:,i) = min(vs(:,i), bup);
        % update weight
        ws(:,i) = ws(:,i) + vs(:,i);
        ws(:,i) = max(ws(:,i), blo);
        ws(:,i) = min(ws(:,i), bup);
        % calculate constribution
        if strcmp(network.activation, 'sigmoid')
            net = X*ws(:,i);
            h = 1./(1+exp(-net));
        elseif strcmp(network.activation, 'tanh')
            net = X*ws(:,i);
            h = tanh(net);
        else
            width = ws(1,i);
            center = ws(2:end,i);
            dist = (X-ones(np,1)*center').^2*ones(nd-1,1);
            h = exp(-dist/width^2);
        end;
        wh = Hs*h;    eh = h - H*wh;
        dSSE = (h'*err)^2/(h'*eh);
        % update pbest and gbest
        if dSSE > pbest(i)
```

```
            pbest(i) = dSSE;
            pw(:,i) = ws(:,i);
        end;
        if dSSE > gbest
            gbest = dSSE;
            gw = ws(:,i);
        end;
    end;
end;
return;
```

Listing B.9: demo2.m

```
clear all;   format compact;
%% load training data and testing data
% training data:
%          input matrix Ti, output vector Td, number of patterns N
% testing data:
%          input matrix Tit, output vector Tdt, number of patterns Nt
load data

tic;
%% initialize the network
network = initialize(Ti, 'FCCN', 'tanh', 0);
%% set parameters
param.maxite = 1000;                %  maximum iteration
param.lambda = 0.0001;              %  regularization
param.Kmax = 10;                    %  maximum number of hidden neurons
param.begin_latency = 20;           %  latency determine when to add new
    neuron
param.begin_thd = 0.001;            %  threshold determine when to add new
    neuron
```

```
param.end_latency = 5;            %  latency determine when to stop adding
param.end_thd = 0.00001;          %  threshold determine when to stop
   adding
param.algo = 1;                   %  algorithm: 1 (HC1) or 2 (HC2)
param.PSO_maxite = 20;            %  maximum iteration of PSO
param.PSO_n = 10;                 %  number of particles of PSO

%% construct the network
[SSE, network, tick] = construct(network, Ti, Td, param);

%% validate on testing data
H = forward(network, Tit);
y = H*network.wo;
err = Tdt-y;
SSEt= err'*err;    % SSE of test data
```