Algorithms for Optimal Construction and Training of Radial Basis Function Neural Networks

by

Philip David Reiner

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
May 10, 2015

Keywords: Machine Learning, Radial Basis Functions, Artificial Neural Networks, Optimization, Support Vector Machines

Abstract

Machine Learning and Computational Intelligence are rapidly growing fields of research in both academia and industry. Artificial neural networks are at the heart of much of this research. Efficiently constructing and training artificial neural networks is of utmost importance to advancing the field. It has been shown that compact architectures show better generalization performance to networks containing many computational nodes. Furthermore, special neurons consisting of a Radial Basis Function can be used to improve local performance of ANNs. Many algorithms such as Support Vector Regression, Error Backpropagation, and Extreme Learning Machines can be used to train networks once an architecture is chosen. Other algorithms such as RAN, MRAN, and GGAP can train networks as they are constructed. However, many of these algorithms have limitations that lead to an excessive network size. Two new RBF network construction algorithms are introduced with the aim of increasing error convergence rates with fewer computational nodes. The first method is introduced in Chapter 3 and expands on the popular Incremental Extreme Learning Machine algorithms by adding a Nelder-Mead simplex optimization to the process. The second algorithm, described in Chapter 4, uses a Levenberg-Marquardt algorithm to optimize the positions and heights of RBF units as they are added to a network. These algorithms are compared to many state of the art algorithms on difficult benchmarks and real-world problems. The results demonstrate that more compact networks with superior error performance are created.

Acknowledgements

I must begin by thanking my advisor and mentor for the past several years, Prof. Bogdan Wilamowski for his guidance and motivation during my studies. It has been an honor and a privilege to work with him.

I would like to thank each of my committee members for their time and guidance. It has been a joy to learn from each of you during my studies at Auburn.

I would also like to thank my wife Heather Reiner for standing by my side throughout the entirety of my graduate career. Her patience has allowed me the opportunity to be the best that I can be.

Table of Contents

List of Tables

List of Figures

Chapter 1 Introduction

In our modern society, computers are everywhere. From statistical modelling of complex systems to turning on and off the lights, computers are used to solve problems in every aspect of our lives. As technology becomes more advanced, the number of problems that can feasibly be handled by software increases. However, there are some complex real-world problems that cannot effectively be solved by traditional approaches such as first principles modeling or explicit statistical modeling. Many of these problems are not considered to be mathematically well-posed problems. However, nature often provides many examples of biological systems exhibiting incredibly complex functions. For instance, the human body has 244 degrees of freedom being controlled by 630 muscles [1], yet humans have little trouble executing target movements. Furthermore, these controls must be able to be executed in the presence of uncertainty, noise, and an ever-changing context.

The attempt to address complex real-world problems using nature-inspired computational methodologies is often known as Computational Intelligence (CI). The characteristic of "intelligence" is usually attributed to humans, but the field of CI attempts to use software to imitate the abilities of humans to perform reasoning and decision making. For example, Fuzzy Logic was introduced by Zadeh in 1965 as a tool to formalize and represent the reasoning process. Fuzzy logic systems possess many characteristics attributed to intelligence by dealing effectively with uncertainty that is common for human reasoning, perception, and inference, while maintaining the formal mathematical backbone needed for computation [2]. Evolutionary computation mimics the population based evolution through reproduction of generations and genetics in so called genetic algorithms [3].

Another attribute of intelligence that CI attempts to mimic is learning, or the ability for a system to change with respect to the data it receives rather than follow explicitly programmed instructions. This field is called Machine Learning (ML) and shares its roots, along with CI, in computer science and statistics. ML is also closely tied with optimization. In fact, many of the learning algorithms can be thought of as optimizing a system relative to the problem to be learned. As with other CI subfields, machine learning is employed in computing tasks where designing and programming explicit rule-based algorithms is infeasible. ML is often used in real-world tasks such as, spam filtering, optical character recognition (OCR), search engines, pattern recognition, data mining, and computer vision [4].

ML tasks can be broken into several categories such as: supervised learning, unsupervised learning, semi-supervised learning, and reinforcement learning. Supervised learning algorithms are trained on labelled examples, data where the desired output is known for a specific input. These algorithms attempt to generalize a function that maps inputs to outputs so that a previously unseen input will generate an output. Unsupervised learning algorithms are trained on unlabeled examples, where the data is examined to find an underlying structure. Semi-supervised learning combines both labeled and unlabeled data to generate an appropriate function or mapping. Reinforcement learning is concerned with how intelligent agents should act in an environment to maximize some notion of reward or minimize a notion of cost. The agent performs a set of actions which cause the observable state of the environment to change. By examining the environment after each action, the agent attempts to gather knowledge about how the environment responds to its actions. The agent then attempts to perform actions that maximize a reward or minimize a cost [5].

There are several approaches to solving an ML problem. These approaches include algorithms such as: decision tree learning, association rule learning, artificial neural networks, inductive logic programming, support vector machines, clustering, bayesian networks, reinforcement learning, representation learning, similarity and metric learning, and sparse dictionary learning. The focus here will be on developing Artificial Neural Networks (ANNs); their architectures, feature mapping, and training algorithms. ANNs will be used to solve supervised problems consisting of both classification and regression.

1.1 Artificial Neural Networks

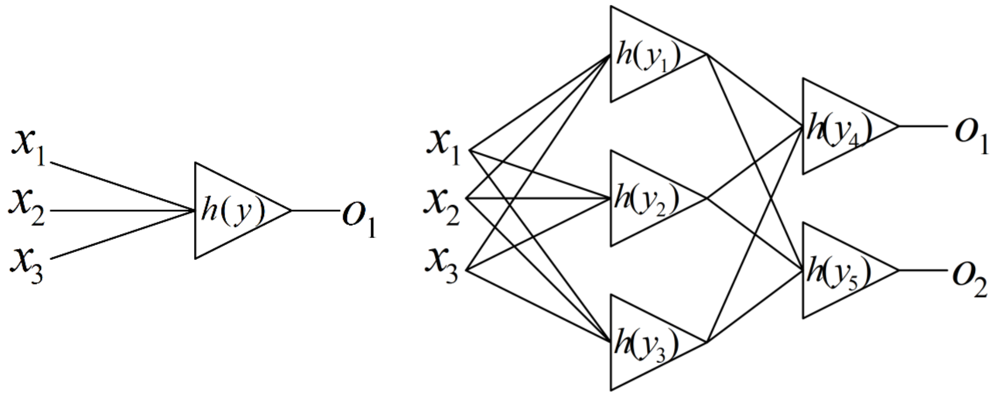
In the field of machine learning, Artificial Neural Networks (ANNs) are computational models designed to replicate the function of an animal's central nervous system (CNS) to bring a sense of intelligence to a software module. Computations in an ANN are structured in terms of an interconnected group of artificial neurons, called a network. Modern neural networks are non-linear statistical data modeling tools. These networks are designed to do popular tasks in machine learning such as classification, regression, and pattern recognition. The idea of intelligence in an ANN comes from the ability of the system to be changed by the data that is fed through it. This ability is referred to as "learning." There are several methods by which an ANN learns. For instance, in unsupervised cases, the distance between separations in the data created by the neural network is maximized, or in supervised cases, the difference between the ANN output and the target output is minimized. Learning methods will be discussed in great detail later.

There is no single formal definition of what an artificial neural network is. However, a class of computational models may be called “Neural” in current research literature if they possess the following characteristics:

1. They consist of sets of adaptive (tunable) weights that affect the relative strength of various inputs, and these weights are adjusted by a learning algorithm.
2. They are capable of approximating non-linear mappings of their inputs to an output.

The adaptive weights are conceptually connection strengths between neurons which are activated as data is passed through the network for both training and prediction. In modern neural networks, the similarities between ANNs and their biological counterparts is found mostly in the implementation of the artificial neurons and their arrangement in parallel and local processing paradigms and in their ability to adapt with new data.

The neurons used in ANNs are modelled after neurons found in biological systems. Each neuron has a set of input connections (representing dendrites) and an output (representing the neuron’s axon). The inputs are usually combined by a weighted sum and operated on by a nonlinear function, known as the neuron’s activation function, to produce an output. This activation function is the basis of how an ANN provides nonlinear mappings from inputs to outputs. Traditionally, the activation function of a neuron is a simple threshold function causing the output of each neuron to be one or zero. This corresponds to the all or nothing firing of a biological neuron. However, continuous functions such as the sigmoid or logistic function are used to provide a differentiable output while still providing a nearly all or nothing output. This differentiability is important for many modern learning methods that will be discussed later. In special cases, other activation functions are used for neurons to have specific effects on the mapping of inputs to outputs. **Figure 1.1-1**An example of an artificial neuron, (a) and a collection of neurons, (b), known as an artificial neural network. Here x_i is an input, h is the activation function, and y_n is equal to the weighted sum of neuron inputs: $y_n = x_1 w_{1,n} + x_2 w_{2,n} \dots + x_i w_{i,n}$.Figure 1.1-1 depicts a single neuron and a collection of neurons in a neural network.



(a) (b)

Figure 1.1-1 An example of an artificial neuron, (a) and a collection of neurons, (b), known as an artificial neural network. Here $\boldsymbol{x_i}$ is an input, $\boldsymbol{h}$ is the activation function, and $\boldsymbol{y_n}$ is equal to the weighted sum of neuron inputs: $\boldsymbol{y_n = x_1 w_{1,n} + x_2 w_{2,n} \dots + x_i w_{i,n}}$.

1.1.1 Learning

The most interesting attribute of ANNs is the possibility of learning. Given a specific task to solve, i. e. a set of data that needs to be classified or a function to be approximated, learning is the ability of the ANN to solve the given task in some optimal sense. The optimality is defined as minimizing a given cost function. For instance, given a set of data and the task to find a function that approximates the characteristics of the data, the cost function would be some measure of total error between the network output and the actual data.

The cost function is an important concept in learning, as it is the mechanism by which the success of the network is measured. The cost function, C, is defined such that for the optimal solution, f^*, $C(f^*) \leq C(f) \forall f \in F$. Basically, no solution has a cost less than the cost of the optimal solution. A particular learning algorithm is defined by the methods it uses to search through the solution space to find a function that has the smallest possible cost. For problems where the solution is dependent on some data, the cost must be a function of the observations. In this case, the solution is an approximation of a statistic of the data. For instance, the problem of finding a function, f, which minimizes $C = \frac{1}{N}\sum_{i=1}^{N}(f(x_i) - y_i)^2$ for N data pairs (x, y) drawn from some distribution, $\mathcal{D}$. In this case, the cost is minimized over a sample of the data in hopes that the solution will adequately represent the entire data set. Online learning methods attempt to address this problem when $N \to \infty$. In online learning, a portion of the cost is minimized as each sample is shown to the network. In the end, the cost function will be chosen either based on its desirable properties, such as convexity, or because it arises naturally from the problem to be solved.

There are three major paradigms associated with learning tasks (not including the combination of supervised and unsupervised learning, semi-supervised learning). These are supervised learning, unsupervised learning, and reinforcement learning.

In supervised learning, a set of training data is given such that each sample is a $(x, y), x \in X, y \in Y$ pair, and the goal of the learning is to find a function $f: X \to Y$ in the set of functions that matches the examples. In other words, we wish to infer the mapping implied by the data. The cost function will then be a function of the errors between our mapping of the data and the data itself. A commonly used cost for this type of problem is the mean-squared error which is the squared difference between the network's output, $f(x)$, and the training target, y, over all of the training data. Gradient descent algorithms are commonly used to minimize this cost. Tasks that fall under the umbrella of supervised learning are pattern recognition (classification) and regression (function approximation). In some cases, sequential data can be applied to the supervised learning paradigm, i.e. speech and gesture recognition. In these problems, a function representing feedback on the quality of the solutions obtained is given.

In unsupervised learning, some data, x, is given. The cost function to be minimized can then be any function of the data and the network output, f. The cost function will be dependent on the desired model and the a priori assumptions of the data. For example, if the model is simply a constant, $f(x) = a$, and the cost is the mean-squared error discussed earlier, minimizing the cost will yield an a equal to the mean of the data. Of course, the cost function is typically much more complicated than that. For example, it could be related to the mutual information between x and $f(x)$, or it could be related to the posterior probability of the model given the data. Machine learning tasks that fall under the unsupervised learning paradigm include, clustering, statistical distribution modelling, compression, and filtering.

Semi-supervised learning is a combination of both supervised and unsupervised paradigms. Typically this paradigm is created as a series combination of supervised and unsupervised networks. In some cases, the raw data itself may not be suitable for a supervised learning approach, so an unsupervised approach is used to model some properties of the raw data. These modelled properties are then treated as data and fed into a supervised learning structure. For example, a set of data may be clustered by an unsupervised algorithm to determine a set of categories for the data. Then that information is added to the data and passed to a function approximation network that will treat a training sample differently based on the class to which it belongs. This combination of paradigms is used often in the industry for many specific applications such as, medical diagnosis, image processing, control systems, and many others.

Reinforcement learning is the paradigm with which most people associate the idea of "artificial intelligence." In reinforcement learning, the data are usually not given explicitly, but generated by an agent's interactions with the environment. At each point in time, t, the agent performs an action, y_t, and the interaction with the environment generates an observation, x_t, and an instantaneous cost, c_t. The goal is then to discover a rule base for selecting actions that minimizes a measure of the expected cumulative cost. The environment is often modelled as a Markov Decision Process (MDP) with states $s_1, \dots, s_n \in S$, and actions, $a_1, \dots, a_m \in A$, with the probability distributions pertaining to the instantaneous cost distribution, $P(c_t|s_t)$, the observation distribution, $P(x_t|s_t)$, and the transition distribution, $P(s_{t+1}|s_t, a_t)$. Then the rule base, or policy, is defined as the conditional distribution over actions given the observations. Taken together, the MDP and the policy are defined as a Markov Chain (MC). The goal is to discover the MC for which the cost is minimal. Many times ANNs are used as a block in the overall reinforcement learning algorithm, being coupled with other ideas such as Dynamic programming or fuzzy systems. Specific problems that can be solved in this paradigm are, intelligent vehicle routing, resource management, controls, and other sequential decision making tasks.

There are many algorithms for learning. Most of them can be thought of as an optimization algorithm that adjusts the relative strength of connections in a neural network in order to minimize a cost function. This optimization of neural networks is the key concept that allows them to be useful in many modern day applications where the problem is not easily solved using direct analytical methods. Specifically, the ability to adjust neurons in a local sense is very important in solving many problems that seem to be radially based such as, image processing, clustering, and classification.

1.1.2 Modelling Artificial Neural Networks

The term model in the context of an ANN can be used to describe a particular arrangement of neurons in a network or a certain activation function for each neuron. These models are referring to a series of mathematical models that define a mapping, $f: X \rightarrow Y$, or a distribution over X and Y. Sometimes, models are also closely associated with a specific learning rule or training algorithm.

For instance, many of today's neural networks are built on sigmoidal or bipolar neurons. These neurons are modelled as threshold neurons like the ones found in biology consisting of an activation function that is either all the way on or all the way off such as: $f(x) = \frac{1}{1+e^{-x}}$. Another similar activation function is the bipolar activation function which has the same shape as a sigmoidal function. This function however operates in the range (-1,1). Many times this function is modelled as the tangent hyperbolic function: $f(x) = \tanh(x)$. These types of neurons are generally thought of as having a global impact with relation to the input space. Algorithms such as Error Backpropoagation (EBP) are generally designed to handle architectures consisting of these neurons. In fact, the most commonly used network paradigms today consist of sigmoidal neurons trained by the EBP algorithms. However, it will be investigated in subsequent sections whether or not this is the most effective training paradigm. In some cases, the global nature of the sigmoidal neurons can actually be a disadvantage to some systems. If a more local neuron structure is needed, sigmoidal networks will require a far larger number of neurons than a local paradigm such as Gaussian based neurons.

Neurons with locally tuned response characteristics can be found in many parts of biological nervous systems, such as visual systems. These neurons are selective within a finite range of the input space. The local characteristics of these neurons makes them suitable for problems in which there are strong spatial relationships. For example, the cochlear stereocilia cells of a biological ear have a locally tuned response to the frequency of sound being sensed. Additionally, much of the data obtained in the visual field has strong spatial relations, so it is no surprise that neurons associated with the biological eye model signals locally. These characteristics allow artificial locally tuned neural networks to be well suited to solving signal processing and computer vision problems.

The activation functions of locally tuned neurons are often referred to as Radial Basis Functions (RBF), and networks made out of locally tuned neurons are called RBF networks. Most RBF network algorithms use a simple three layered architecture where only one hidden layer of parallel units exists. This has the benefit of making the training problem much simpler as well as eliminating the complexity of choosing an architecture (however more architectural considerations for RBF networks will be discussed in Chapter 2). Radial Basis Functions also have low sensitivity to noise in the data. This allows the adjustable parameters to converge to a stable minimum during the training process. All of the aforementioned advantages of RBF networks allow them to be tuned and examined to have good generalization performance. The research proposed in this work will demonstrate the usefulness of RBF networks in solving real-world data problems.

1.1.3 Radial Basis Function Networks

RBF networks can be made up of a variety of activation functions. The only requirement is that the function value depends only on the distance from the inputs to an origin or center. This allows the neurons to be trained so that inputs in different areas of the input space will have different effects on the outputs. Given a center, c and radius, σ, there are several functions that are considered RBFs. A few popular examples are shown below:

Gaussian:

$$\boldsymbol{h}(\boldsymbol{x}) = \boldsymbol{e}^{-\frac{\|x-c\|^2}{\sigma^2}} \tag{1.1-1}$$

Multiquadric:

$$h(x) = \sqrt{1 + \frac{\|x-c\|^2}{\sigma^2}} \tag{1.1-2}$$

Inverse quadratic:

$$h(x) = \frac{1}{1 + \frac{\|x-c\|^2}{\sigma^2}} \tag{1.1-3}$$

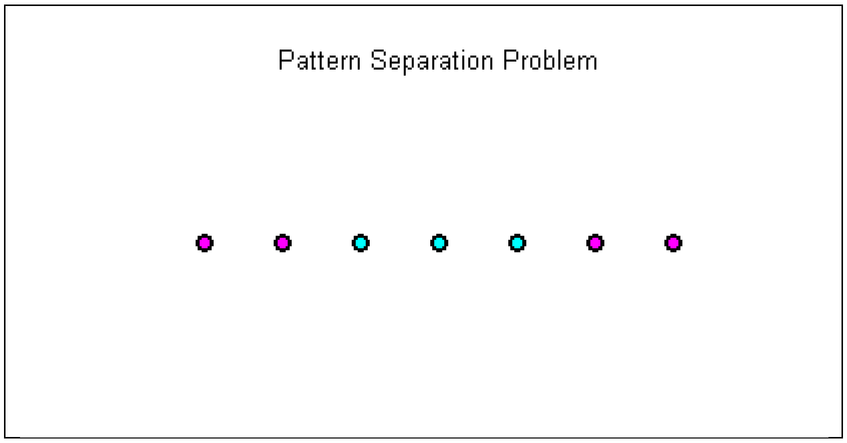
Inverse multiquadratic:

$$h(x) = \frac{1}{\sqrt{1 + \frac{\|x-c\|^2}{\sigma^2}}} \tag{1.1-4}$$

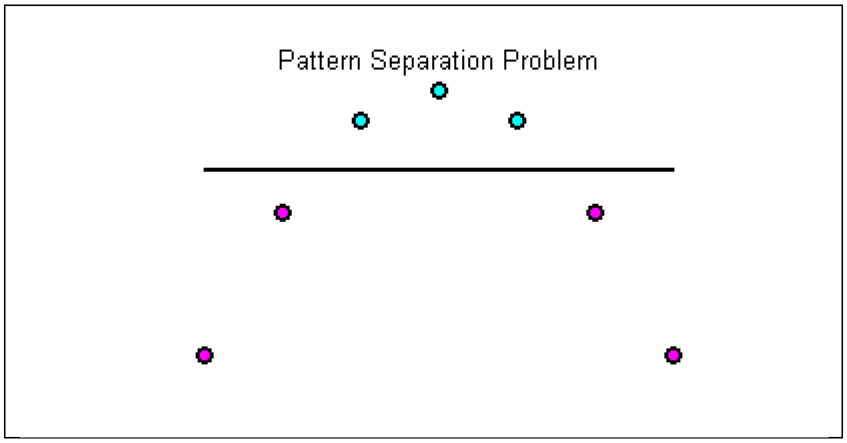
In a learning paradigm these functions will be optimized by adjusting the centers, radii, and heights of each RBF.

The first use of RBF functions for mapping is found in T. M. Cover's work in 1965. Cover's theorem on the pattern-separating capacity of hyperplanes asserts that a complex classification problem is more likely to be linearly solvable if it is mapped nonlinearly into a high-dimensional space [6]. This concept is also the motivation for the use of nonlinear kernel functions in Support Vector Machines (SVM). Figure 1.1-2 depicts this concept.

Most often, RBF networks consist of a single hidden layer with many RBF neurons in parallel and an output layer that is a weighted sum of the hidden neurons. This is arrangement is known as the Single-Layer Feedforward Network (SLFN). J. Moody and C. J. Darken first proposed a network in the form of a SLFN with locally tuned processing units in 1988 [7]. An example of a SLFN is shown in Figure 1.1-3. This network architecture is used extensively in constructing RBF networks.



(a)



(b)

Figure 1.1-2 An illustration of Cover's theorem. (a) A set of data that is not linearly separable in 1 dimension. (b) By mapping this data to 2-dimensional space using the nonlinear function: $(x) = e^{-\|x-0.5\|^2}$, the data is made linearly separable.

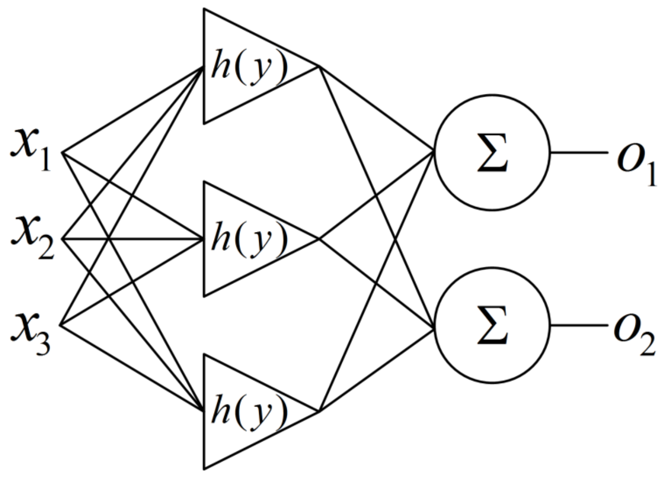


Figure 1.1-3 An SLFN network like the one used by Moody and Darken. Here, y is a function of x and the network parameters: input weights, centers, widths, etc.

Many of the early implementations of the SLFN type of network set the network parameters by first randomly choosing training data points as RBF centers then using singular-value decomposition to solve for the weights (or heights) of the RBF neurons. T. Poggio and F. Girosi created a method of selecting RBF centers using a gradient descent training approach. They called their algorithm Generalized Radial Basis Function (GRBF) networks [8]. S. Chen, C. F. N. Cowan, and P. M. Grant demonstrated that an orthogonal least squares learning method can be used to select RBF centers in such a way that each RBF unit maximizes the variance of a desired output. They demonstrated this method's effectiveness on two signal processing applications [9]. Shortly afterwards, in 1992, D. Wettschereck and T. Dietterich demonstrated an effective application of GRBF networks to the task of language pronunciation [10].

Selecting the appropriate number of RBF neurons to solve a given problem is a task as important as selecting the proper RBF parameters. A resource allocating network (RAN) was proposed in which a network learns by adjusting the parameters of existing neurons, then adding new neurons to compensate for poor performance on certain input patterns [11]. An improved version of the RAN algorithm was proposed in which an extended Kalman filter (RANEKF) was used, instead of a least-mean square algorithm, for updating the network parameters [12]. Further improvement was made on the RAN algorithm by Yingwei *et. al.* [13]. This algorithm creates a minimally sized network and is known as MRAN. It is very often used in current literature in real-world applications [14]. A growing radial basis function network algorithm was proposed in which, at first, a small number of RBF neurons are trained. During the training process, there is a period called the "growing cycle" in which a neuron satisfying two splitting criteria is split into two new neurons. The learning scheme provided a framework for incorporating existing supervised and unsupervised training algorithms into the growing RBF network [15]. An algorithm for growing and pruning RBF (GAP-RBF) networks was introduced by P. Saratchandran and N. Sundararajan [16]. This algorithm evaluates the "significance" of each neuron based on its contribution to the network output averaged over all the input data. After it is evaluated, the neuron is either kept in the network or discarded. This process allows problems to be solved with a greatly reduced network size and training time.

Two algorithms that further optimize the network construction process with the aim of increasing error convergence rates of highly compact networks are introduced in this work. The first method, Nelder-Mead Enhanced Extreme Learning Machine (NME-ELM) is introduced in Chapter 3 and expands on the popular Incremental Extreme Learning Machine algorithms by adding a Nelder-Mead simplex optimization to the process. The Error Correction (ErrCor) algorithm, described in Chapter 4, uses a Levenberg-Marquardt algorithm to optimize the positions and heights of RBF units as they are added to a network. These algorithms are compared to many state of the art algorithms on difficult benchmarks and real-world problems.

Currently RBF networks are used in many different areas of industry. For example, Sue Inn Ch’ng *et. a.l* used an adaptive momentum Levenberg-Marquardt RBF for face recognition in [17]. Like traditional ANNs RBF networks have also been developed to handle fault diagnosis problems [19], adaptive control problems [20]–[24], image processing [25], [26], approximation and interpolation [27], [28], and classification [18].

Chapter 2 Neural Network Training and Construction

Once a network paradigm is chosen, the construction and training of an ANN is a non-trivial process. There are many ways a network can be implemented. In general, two major considerations must be made:

(1) What is the architecture of the network, and how many neurons will be in that architecture?

(2) Which algorithm can be used to train the given network architecture to a desirable error level?

These considerations are typically dependent on each other and on the knowledge of the creator of the network. Many architectures and training algorithms were studied in the comparative work [29]. These architectures and algorithms have different advantages and disadvantages for each situation. They will be discussed more in depth in subsequent sections of this work.

2.1 Neural Network Architectures

One of the major difficulties facing researchers using ANNS is the decision of how many neurons must be used to solve a given problem, and in which topology should these neurons be arranged. Unfortunately, there is a nearly infinite number of combinations of networks that could possibly solve a given problem. There are three major architectures that are used in the research to solve many problems. These architectures are depicted in Figure 2.1-1, 2.1-2, 2.1-3, and 2.1-4 while their advantages and disadvantages are discussed in depth in the following paragraphs.

The most common architectures are examined and compared in [30]. The problem on which they are compared is the parity-N problem. This problem is essentially a mapping defined by 2^N binary vectors that indicates whether the sum of the N elements of every binary vector is odd or even. In this problem, any pattern with the same sum as another pattern can be omitted from training as it will have the exact same answer as another pattern. Therefore, a simplified set of the original 2^N patterns can be obtained which contains only $N + 1$ patterns. This problem was shown to be a suitable benchmark for comparison in [31]. Table 1 below shows the full parity-3 problem and table 2 depicts the reduced set of patterns.

Table 1 Parity-3 problem inputs and outputs.

Input	Sum of Inputs	Output
000	0	0
001	1	1
010	1	1
011	2	0
100	1	1
101	2	0
110	2	0
111	3	1

Table 2 Parity-3 problem using sum of inputs as simplified inputs.

Simplified Inputs	Output
0	0
1	1
2	0
3	1

The most popular and simplest of the studied neural network architectures is the Multilayer Perceptron architecture. This architecture can have any number of hidden layers with any number of neurons, but the connections cannot go across layers. In a MLP network with a single hidden layer consisting of n hidden neurons, the largest possible $parity - N$ problem that can be solved is:

$$N = n. \qquad (2.1\text{-}1)$$

Figure 2.1-1 depicts a typical single layer MLP. If a MLP network is restricted to a single hidden layer, it is called a Single Layer Feedforward Network (SLFN).

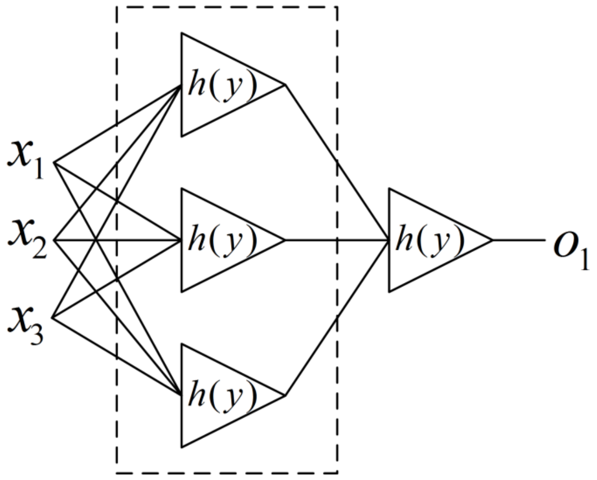


Figure 2.1-1 A simple MLP architecture with a single hidden layer. The hidden neurons are the neurons contained in the dashed rectangle. For this network, $n = 3$ and the largest parity problem that can be solved is parity-3.

If a MLP network is allowed to have connections across layers, then the network is called a Bridged Multilayer Perceptron (BMLP). These networks have been shown to be more powerful than traditional MLP networks [30], [31]. For a single layer BMLP network, like the one shown in Figure 2.1-2, consisting of n neurons the largest possible $parity - N$ problem that can be solved is:

$$N = 2n - 1. \qquad (2.1\text{-}2)$$

Of course, most designers of neural networks would like to have more than a single hidden layer. Given a BMLP network with two hidden layers (like the one in Figure 2.1-3) where the number of neurons in the first layer is m and the number of neurons in the second layer is n, the largest possible $parity - N$ problem that can be solved is:

$$N = 2(m+1)(n+1) - 1 \tag{2.1-3}$$

If this pattern is carried out further so that a BMLP network has k hidden layers with each layer containing a number of neurons $n_i, i = 1, \ldots, k$, then the largest $parity - N$ problem that can be solved is:

$$N = 2\prod_{i=1}^{k}(n_i + 1) - 1 \tag{2.1-4}$$

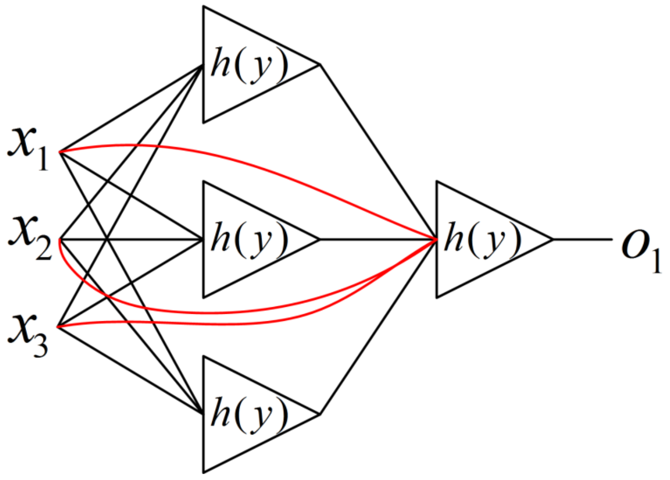


Figure 2.1-2 A simple BMLP architecture. The “bridged” connections that distinguish this network from the one shown in Figure 2.1-1 are highlighted in red.

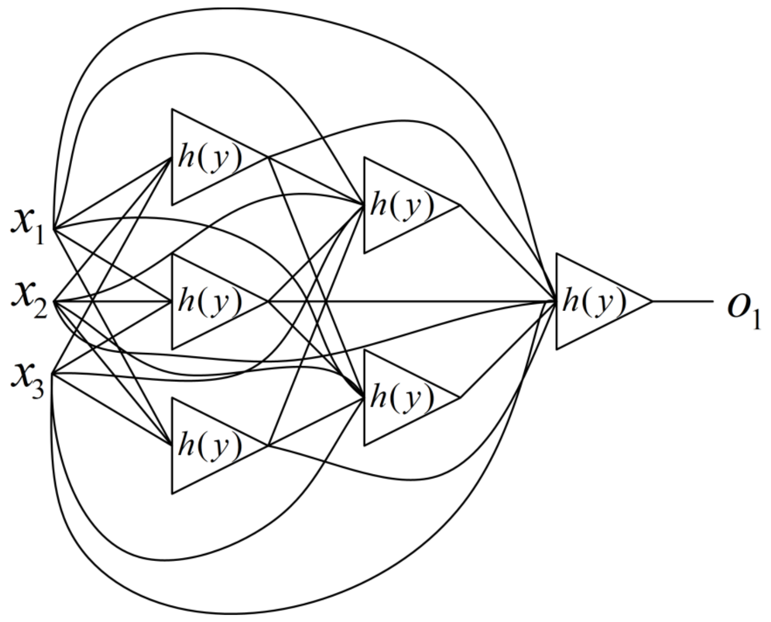


Figure 2.1-3 A fully bridged BMLP architecture with two hidden layers, $n_1 = 3, n_2 = 2$. This network is capable of solving a parity-23 problem.

A fully connected cascade (FCC) network is the third type of network investigated. This network is a BMLP network created with the following constraints:

(1) All connections are bridged and connected to all subsequent layers.

(2) All hidden layers are restricted to having a single neuron apiece.

This network architecture is depicted in Figure 2.1-4. Intuitively, this architecture requires as many hidden layers as there are neurons. This arrangement allows for a very powerful network to be created without using a large number of neurons. Many networks called "deep networks" are similar to the FCC network. The problem with these networks is that not many algorithms can train them. However, recent research has been focused on solving this problem. The largest $parity - N$ problem that can be solved by a FCC network containing n hidden neurons is:

$$N = 2^n - 1 \qquad (2.1\text{-}5)$$

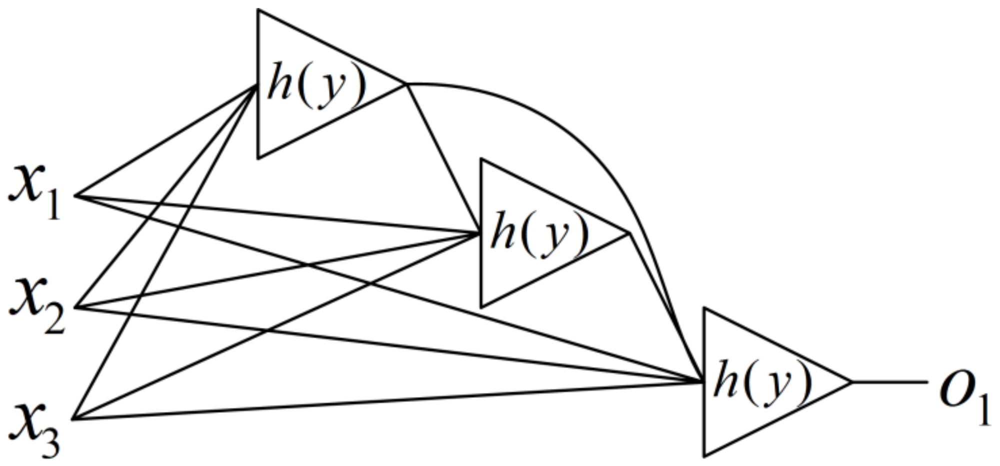


Figure 2.1-4 A FCC network with 3 neurons. This network is capable of solving the parity-7 problem.

It may seem obvious that the FCC architecture should be used to solve for every problem, but the task of constructing an optimized network is not so simple. In most cases, a network is designed with some predetermined architecture and then the weights on the connections between neurons are optimized in an attempt to obtain error convergence. In some networks, the error will never converge for a particular problem. One must still answer the question, “How will an optimal network be built, so that the error for a particular problem will always converge?”

These neural network architectures offer an interesting set of solutions for solving a given problem with ANNS. For RBF networks, the SLFN structure is commonly used. This structure is similar to the MLP shown above. However, the output neuron of a RBF network is typically a linear, or summing, neuron.

2.2 Supervised Neural Network Training

The ability for a network to change its parameters to solve a problem in some optimal sense is what makes ANNS appealing. The algorithms used to optimize neural networks are collectively known as training algorithms. As is always the case with optimization, a cost function must be defined. The cost function used in neural networks is known as the error, meaning the difference between a desired output and the current network output for a given pattern. This error function is a function of the data presented to the neural network and the neural network parameters. This will allow the error to be minimized by tuning the parameters in the neural network.

The data used for training neural networks in this document can be thought of as a set of input, target pairs:

$$\aleph = \{(x_i, t_i) | x_i \in \mathbb{R}^d, t_i \in \mathbb{R}^m, i = 1, \ldots, N\} \tag{2.2-1}$$

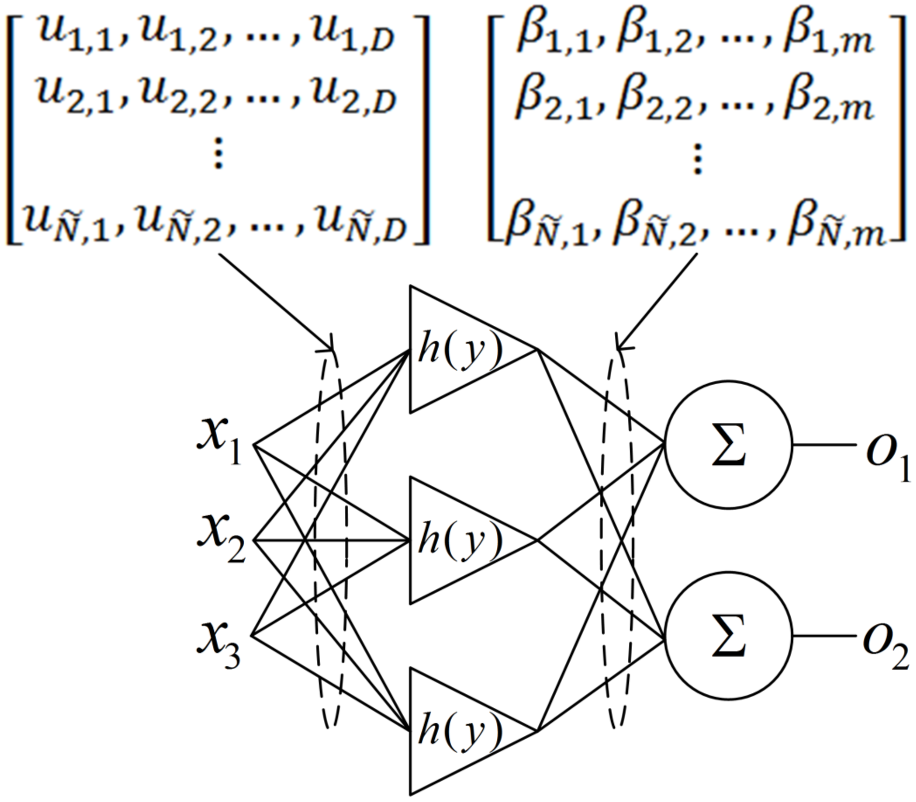
Where N is the total number of patterns in the training set. For each input, the network will have some output, o_i. The error for each pattern is then defined as the difference between the target value and the network output for a specific pattern

$$e_i = t_i - o_i \tag{2.2-2}$$

In order to have a single value to examine that gives a measure of the overall error of the network, the sum square error (SSE) value is used.

$$SSE = \sum_{i=1}^{N} (e_i)^2 \tag{2.2-3}$$

The relative strength of network connections can be adjusted to minimize the error. These connections are collectively known as weights. However, in the case of RBF networks, the term weights can also refer to the RBF parameters center and radius. In this document, the input weights of an RBF unit are denoted as $u_{j,i}$ where i is the index of the input dimension and j is the index of the neuron in the hidden layer. The output weights will be denoted as $\beta_{j,k}$ where j is again the index of the neuron in the hidden layer and k is the index of the output. The RBF parameters, center and radius, will be denoted as c_j and σ_j respectively. Again, j is the index of the neuron in the hidden layer. Figure 2.2-1 illustrates this notation.



$\begin{bmatrix} u_{1,1}, u_{1,2}, \dots, u_{1,D} \\ u_{2,1}, u_{2,2}, \dots, u_{2,D} \\ \vdots \\ u_{\tilde{N},1}, u_{\tilde{N},2}, \dots, u_{\tilde{N},D} \end{bmatrix}$
$\begin{bmatrix} \beta_{1,1}, \beta_{1,2}, \dots, \beta_{1,m} \\ \beta_{2,1}, \beta_{2,2}, \dots, \beta_{2,m} \\ \vdots \\ \beta_{\tilde{N},1}, \beta_{\tilde{N},2}, \dots, \beta_{\tilde{N},m} \end{bmatrix}$
x_1
x_2
x_3
$h(y)$
$h(y)$
$h(y)$
Σ
Σ
o_1
o_2

Figure 2.2-1 A typical RBF network with input weights designated by their corresponding neuron and input dimension, $u_{j,i}$ and output weights designated by their corresponding neuron and output, $\beta_{j,k}$.

2.2.1 Error Back Propagation and Gradient Descent

The most popular training algorithm is Error Back Propagation (EBP). This algorithm is based on a gradient descent technique, and has been very well used and well researched. Since the original EBP algorithm was published, many improvements have been made [32]–[34]. These improvements include: the notion of momentum [35], flat spot elimination [36], a stochastic learning rate [37], the RPROP algorithm [38], and the QUICKPROP variation of EBP [39].

The EBP algorithm is an integral part of the field of neural networks today. The algorithm provides very stable training convergence, and provides the foundation on which many optimization algorithms can be applied to neural networks. The original EBP algorithm uses a first order steepest descent approach to minimizing the error of the network. Let us use the notation presented previously to describe the EBP algorithm. In order to follow the gradient of the error, the derivative of the error with respect to the network parameters must be found. During each iteration, the vector of input parameters (input weights, biases, and output weights) in a neural network, for iteration q is denoted as Δ_q. For the purpose of simplification, the parameters to be optimized may be referred to individually as $\Delta_q = [\Delta_{q,1}, \Delta_{q,2}, \dots, \Delta_{q,P}]$, where P is the number of parameters being optimized. The reader can assume that if only one index is used, Δ_q refers to the vector of all input parameters at the k^{th} iteration. Likewise, two indexes are used to refer to a single parameter, $\Delta_{q,i}$. The squared error term is used as the cost function.

$$\boldsymbol{E} = \frac{1}{2}\sum_{i=1}^{N}\sum_{k=1}^{m}\left(t_{i,k} - o_{i,k}\right)^2 = \frac{1}{2}\sum_{i=1}^{N}\sum_{k=1}^{m}\left(t_{i,k}^2 - 2t_{i,k}o_{i,k} + o_{i,k}^2\right) \qquad (2.2\text{-}4)$$

Where $o_{i,k}$ is a function of the input and the network parameters.

Then the gradient of the errors with respect to the network parameters is calculated.

$$\boldsymbol{g} = \frac{\partial E(x,\Delta_q)}{\partial \Delta_q} = \left[\frac{\partial E}{\partial \Delta_{q,1}}, \frac{\partial E}{\partial \Delta_{q,2}}, \dots, \frac{\partial E}{\partial \Delta_{q,P}}\right] \qquad (2.2\text{-}5)$$

The parameters are then updated according to the rule of steepest descent:

$$\Delta_{q+1} = \Delta_q - \alpha \boldsymbol{g}_q \tag{2.2-6}$$

Where α is the learning constant, or step size. Assuming the network being trained is the one shown in Figure 2.2-1, the first step is to calculate the outputs:

$$o_k = \sum_{j=1}^{\tilde{N}} \beta_{j,k} h_j(y_j)\,, k = 1, \ldots, m \tag{2.2-7}$$

Here, y is the net function, this is a function of the inputs and any adjustable parameters for neuron j. Once the output and errors are calculated, the partial derivatives of the errors will be calculated for each network parameter. The process of calculating the derivatives of parameters backwards through the network is known as back-propagation. First the derivatives of the errors are found for the output weights, β.

$$\frac{\partial E_i}{\partial \beta_{j,k}} = -(t_{i,k} - o_{i,k}) \frac{\partial o_{i,k}}{\partial \beta_{j,k}} = -e_i \frac{\partial o_{i,k}}{\partial \beta_{j,k}} \tag{2.2-8}$$

$$\frac{\partial o_{i,k}}{\partial \beta_{j,k}} = h_j(y_j) \tag{2.2-9}$$

$$\frac{\partial E_{i,k}}{\partial \beta_{j,k}} = -e_{i,k} h_j(y_j) \tag{2.2-10}$$

Now the parameters for the previous layer can be calculated. Define a variable for the derivatives of the errors with respect to the output weights.

$$\delta_k = e_{i,k} h_j(y_j) \tag{2.2-11}$$

$$\frac{\partial E_i}{\partial y_j} = -\left[\sum_{k=1}^{m} \delta_k \beta_{j,k}\right] \frac{\partial h_j}{\partial y_j} \tag{2.2-12}$$

$$\frac{\partial E_i}{\partial \Delta_{q,j}} = -\left[\sum_{k=1}^{m} \delta_k \beta_{j,k}\right] \frac{\partial h_j}{\partial y_j} \frac{\partial y_j}{\partial \Delta_{q,j}} \tag{2.2-13}$$

Where $y_j(x_i, \Delta_{q,j})$ is a function of the inputs and network parameters. This step is where the term backpropagation comes into play. The errors and the output weights are propagated back to the previous layers. Of course in a software environment, this can be done as a matrix computation so that the gradients of all output weights are found at once. This leads to the formation of a gradient matrix of first derivatives for each parameter with respect to the error. For the sake of brevity, the next several algorithms are described using a single parameter matrix, Δ_q.

The EBP algorithms will usually lead to small training error values, but these algorithms have several drawbacks. The first drawback is that these algorithms still only use the first order gradient and can therefore be trapped in local minima. Much better results can be obtained by using a second-order computation to aid the search process. Second, the algorithms are only able to handle MLP type of architectures [29]. Adding bridged connections to the network will cause the gradient computations to change, and the algorithm will fail. Finally, the EBP algorithm requires both a backwards and forwards pass through the network during each iteration. This means that computation can become very expensive for a network of substantial size [40]. Many of the currently used RBF paradigms are trained with a gradient descent algorithm like EBP.

2.2.2 Newton's Algorithm

Let us consider each individual component of the gradient vector as a function of the network parameters:

$$\begin{cases} g_1 = F_1(\Delta_{q,1}, \Delta_{q,2}, \dots, \Delta_{q,P}) \\ g_2 = F_2(\Delta_{q,1}, \Delta_{q,2}, \dots, \Delta_{q,P}) \\ \qquad\vdots \\ g_P = F_p(\Delta_{q,1}, \Delta_{q,2}, \dots, \Delta_{q,P}) \end{cases} \tag{2.2-14}$$

Finding the minimum of the error surface can be posed as finding the roots of the error derivatives. Assuming the network parameters are linearly independent, Newton’s algorithm can be used to find these roots. First, the gradients are set to zero:

$$\begin{cases} g_1 = 0 = F_1\left(\Delta_{q,1}, \Delta_{q,2}, \dots, \Delta_{q,P}\right) \\ g_2 = 0 = F_2\left(\Delta_{q,1}, \Delta_{q,2}, \dots, \Delta_{q,P}\right) \\ \qquad\qquad \vdots \\ g_P = 0 = F_p\left(\Delta_{q,1}, \Delta_{q,2}, \dots, \Delta_{q,P}\right) \end{cases} \tag{2.2-15}$$

Then the gradient functions can be approximated using the first two terms of a Taylor expansion:

$$\begin{cases} g_1 = 0 \approx g_{1,0} + \frac{\partial g_1}{\partial \Delta_{q,1}}\left(\Delta_{q+1,1} - \Delta_{q,1}\right) + \frac{\partial g_1}{\partial \Delta_{q,2}}\left(\Delta_{q+1,2} - \Delta_{q,2}\right) + \cdots + \frac{\partial g_1}{\partial \Delta_{q,2}}\left(\Delta_{q+1,P} - \Delta_{q,P}\right) \\ g_2 = 0 \approx g_{2,0} + \frac{\partial g_2}{\partial \Delta_{q,2}}\left(\Delta_{q+1,1} - \Delta_{q,1}\right) + \frac{\partial g_2}{\partial \Delta_{q,2}}\left(\Delta_{q+1,2} - \Delta_{q,2}\right) + \cdots + \frac{\partial g_2}{\partial \Delta_{q,2}}\left(\Delta_{q+1,P} - \Delta_{q,P}\right) \\ \qquad\qquad \vdots \\ g_P = 0 \approx g_{P,0} + \frac{\partial g_P}{\partial \Delta_{q,P}}\left(\Delta_{q+1,1} - \Delta_{q,1}\right) + \frac{\partial g_P}{\partial \Delta_{q,P}}\left(\Delta_{q+1,2} - \Delta_{q,2}\right) + \cdots + \frac{\partial g_P}{\partial \Delta_{q,P}}\left(\Delta_{q+1,P} - \Delta_{q,P}\right) \end{cases} \tag{2.2-16}$$

Using equation (2.2-5), the derivative of the gradient can be found:

$$\frac{\partial g_i}{\partial \Delta_{q,j}} = \frac{\partial\left(\frac{\partial E}{\partial \Delta_{q,j}}\right)}{\partial \Delta_{q,i}} = \frac{\partial^2 E}{\partial \Delta_{q,i} \partial \Delta_{q,j}} \tag{2.2-17}$$

Substituting equation (2.2-17) into the taylor series expansion (2.2-16) yields the following:

$$\begin{cases} 0 \approx \frac{\partial E}{\partial \Delta_{q,1}} + \frac{\partial^2 E}{\partial \Delta_{q,1}^2}\left(\Delta_{q+1,1} - \Delta_{q,1}\right) + \frac{\partial^2 E}{\partial \Delta_{q,1} \partial \Delta_{q,2}}\left(\Delta_{q+1,2} - \Delta_{q,2}\right) + \cdots + \frac{\partial^2 E}{\partial \Delta_{q,1} \partial \Delta_{q,P}}\left(\Delta_{q+1,P} - \Delta_{q,P}\right) \\ 0 \approx \frac{\partial E}{\partial \Delta_{q,2}} + \frac{\partial^2 E}{\partial \Delta_{q,2} \partial \Delta_{q,1}}\left(\Delta_{q+1,1} - \Delta_{q,1}\right) + \frac{\partial^2 E}{\partial \Delta_{q,2}^2}\left(\Delta_{q+1,2} - \Delta_{q,2}\right) + \cdots + \frac{\partial^2 E}{\partial \Delta_{q,2} \partial \Delta_{q,P}}\left(\Delta_{q+1,P} - \Delta_{q,P}\right) \\ \qquad\qquad \vdots \\ 0 \approx \frac{\partial E}{\partial \Delta_{q,P}} + \frac{\partial^2 E}{\partial \Delta_{q,P} \partial \Delta_{q,1}}\left(\Delta_{q+1,1} - \Delta_{q,1}\right) + \frac{\partial^2 E}{\partial \Delta_{q,P} \partial \Delta_{q,2}}\left(\Delta_{q+1,2} - \Delta_{q,2}\right) + \cdots + \frac{\partial^2 E}{\partial \Delta_{q,P}^2}\left(\Delta_{q+1,P} - \Delta_{q,P}\right) \end{cases} \tag{2.2-18}$$

Then the system of equations can then be re-written as:

$$\begin{cases} -\frac{\partial E}{\partial \Delta_{q,1}} \approx \frac{\partial^2 E}{\partial \Delta_{q,1}^2}\left(\Delta_{q+1,1} - \Delta_{q,1}\right) + \frac{\partial^2 E}{\partial \Delta_{q,1} \partial \Delta_{q,2}}\left(\Delta_{q+1,2} - \Delta_{q,2}\right) + \cdots + \frac{\partial^2 E}{\partial \Delta_{q,1} \partial \Delta_{q,P}}\left(\Delta_{q+1,P} - \Delta_{q,P}\right) \\ -\frac{\partial E}{\partial \Delta_{q,2}} \approx \frac{\partial^2 E}{\partial \Delta_{q,2} \partial \Delta_{q,1}}\left(\Delta_{q+1,1} - \Delta_{q,1}\right) + \frac{\partial^2 E}{\partial \Delta_{q,2}^2}\left(\Delta_{q+1,2} - \Delta_{q,2}\right) + \cdots + \frac{\partial^2 E}{\partial \Delta_{q,2} \partial \Delta_{q,P}}\left(\Delta_{q+1,P} - \Delta_{q,P}\right) \\ \qquad\qquad \vdots \\ -\frac{\partial E}{\partial \Delta_{q,P}} \approx \frac{\partial^2 E}{\partial \Delta_{q,P} \partial \Delta_{q,1}}\left(\Delta_{q+1,1} - \Delta_{q,1}\right) + \frac{\partial^2 E}{\partial \Delta_{q,P} \partial \Delta_{q,2}}\left(\Delta_{q+1,2} - \Delta_{q,2}\right) + \cdots + \frac{\partial^2 E}{\partial \Delta_{q,P}^2}\left(\Delta_{q+1,P} - \Delta_{q,P}\right) \end{cases} \tag{2.2-19}$$

Notice that now it is possible to write the system of P equations and P unkowns as a solvable set of matrix equations.

$$\begin{bmatrix} -g_1 \\ -g_2 \\ \vdots \\ -g_P \end{bmatrix} = \begin{bmatrix} \frac{\partial^2 E}{\partial \Delta_{q,1}^2} & \frac{\partial^2 E}{\partial \Delta_{q,1} \partial \Delta_{q,2}} & \cdots & \frac{\partial^2 E}{\partial \Delta_{q,1} \partial \Delta_{q,P}} \\ \frac{\partial^2 E}{\partial \Delta_{q,2} \partial \Delta_{q,1}} & \frac{\partial^2 E}{\partial \Delta_{q,2}^2} & \cdots & \frac{\partial^2 E}{\partial \Delta_{q,2} \partial \Delta_{q,P}} \\ & \vdots & & \\ \frac{\partial^2 E}{\partial \Delta_{q,P} \partial \Delta_{q,1}} & \frac{\partial^2 E}{\partial \Delta_{q,P} \partial \Delta_{q,2}} & \cdots & \frac{\partial^2 E}{\partial \Delta_{q,P}^2} \end{bmatrix} \times \begin{bmatrix} \left(\Delta_{q+1,1} - \Delta_{q,1}\right) \\ \left(\Delta_{q+1,2} - \Delta_{q,2}\right) \\ \vdots \\ \left(\Delta_{q+1,P} - \Delta_{q,P}\right) \end{bmatrix} \tag{2.2-20}$$

The square matrix of second derivatives is called the Hessian matrix:

$$H = \begin{bmatrix} \frac{\partial^2 E}{\partial \Delta_{q,1}^2} & \frac{\partial^2 E}{\partial \Delta_{q,1} \partial \Delta_{q,2}} & \cdots & \frac{\partial^2 E}{\partial \Delta_{q,1} \partial \Delta_{q,P}} \\ \frac{\partial^2 E}{\partial \Delta_{q,2} \partial \Delta_{q,1}} & \frac{\partial^2 E}{\partial \Delta_{q,2}^2} & \cdots & \frac{\partial^2 E}{\partial \Delta_{q,2} \partial \Delta_{q,P}} \\ & \vdots & & \\ \frac{\partial^2 E}{\partial \Delta_{q,P} \partial \Delta_{q,1}} & \frac{\partial^2 E}{\partial \Delta_{q,P} \partial \Delta_{q,2}} & \cdots & \frac{\partial^2 E}{\partial \Delta_{q,P}^2} \end{bmatrix} \tag{2.2-21}$$

Assuming the Hessian is invertible, the equations above can be re-written in the form of the update rule for the Newton algorithm:

$$\Delta_{q+1} = \Delta_q - H_q^{-1} g_k \tag{2.2-22}$$

It can be noticed from equations (2.2-6) and (2.2-22) that the Hessian matrix gives a good approximation of the step size.

2.2.3 Gauss-Newton Algorithm

Let us again examine the equations pertaining to the error gradient. Combining equations (2.2-4) and (2.2-5) gives:

$$g_i = \frac{\partial E}{\partial \Delta_{q,i}} = \sum_{j=1}^{N} \sum_{k=1}^{m} \frac{\partial e_{j,k}}{\partial \Delta_{q,i}} e_{j,k} \tag{2.2-23}$$

Let us define a matrix of derivatives of each error component with respect to each network parameter called the Jacobian:

$$\boldsymbol{J} = \begin{bmatrix} \frac{\partial e_{1,1}}{\partial \Delta_{q,1}}, \frac{\partial e_{1,1}}{\partial \Delta_{q,2}}, \dots, \frac{\partial e_{1,1}}{\partial \Delta_{q,P}} \\ \frac{\partial e_{1,2}}{\partial \Delta_{q,1}}, \frac{\partial e_{1,2}}{\partial \Delta_{q,2}}, \dots, \frac{\partial e_{1,2}}{\partial \Delta_{q,P}} \\ \vdots \\ \frac{\partial e_{1,m}}{\partial \Delta_{q,1}}, \frac{\partial e_{1,m}}{\partial \Delta_{q,2}}, \dots, \frac{\partial e_{1,m}}{\partial \Delta_{q,P}} \\ \vdots \\ \frac{\partial e_{N,1}}{\partial \Delta_{q,1}}, \frac{\partial e_{N,1}}{\partial \Delta_{q,2}}, \dots, \frac{\partial e_{N,1}}{\partial \Delta_{q,P}} \\ \frac{\partial e_{N,2}}{\partial \Delta_{q,1}}, \frac{\partial e_{N,2}}{\partial \Delta_{q,2}}, \dots, \frac{\partial e_{N,2}}{\partial \Delta_{q,P}} \\ \vdots \\ \frac{\partial e_{N,m}}{\partial \Delta_{q,1}}, \frac{\partial e_{N,m}}{\partial \Delta_{q,2}}, \dots, \frac{\partial e_{N,m}}{\partial \Delta_{q,P}} \end{bmatrix} \qquad (2.2\text{-}24)$$

Let us arrange a matrix of individual errors:

$$e = \begin{bmatrix} e_{1,1} \\ e_{1,2} \\ \vdots \\ e_{1,m} \\ e_{2,1} \\ e_{2,2} \\ \vdots \\ e_{2,m} \\ \vdots \\ e_{P,1} \\ e_{P,2} \\ \vdots \\ e_{P,m} \end{bmatrix} \qquad (2.2\text{-}25)$$

The gradient can then be written:

$$g = Je \qquad (2.2\text{-}26)$$

Similarly, inserting equation (2.2-4) into equation (2.2-21) the element of the i^{th} row and j^{th} column of the Hessian matrix can be calculated:

$$H_{i,j} = \sum_{p=1}^{P} \sum_{k=1}^{m} \left(\frac{\partial e_{p,k}}{\partial \Delta_{q,i}} \frac{\partial e_{p,k}}{\partial \Delta_{q,j}} + \frac{\partial^2 e_{p,k}}{\partial \Delta_{q,i} \partial \Delta_{q,j}} e_{p,k} \right) \tag{2.2-27}$$

Ignoring the second derivative term, an approximation for the Hessian matrix can be written as:

$$H \approx J^T J \tag{2.2-28}$$

Finally, the update rule for Newton's algorithm can be re-written as the rule for the Gauss-Newton algorithm:

$$\Delta_{q+1} = \Delta_q - \left(J_q^T J_q\right)^{-1} g_q \tag{2.2-29}$$

This algorithm has the advantage of not needing to directly calculate the second derivatives of the error function. However, there are still issues where the Hessian approximation is not invertible.

2.2.4 Levenberg-Marquardt Algorithm

In order to ensure that the Hessian matrix is invertible, the Levenberg-Marquardt (LM) algorithm introduces yet another modification to the Hessian approximation:

$$H \approx J^T J + \mu I \tag{2.2-30}$$

Where μ is an always positive parameter called the combination coefficient and $\boldsymbol{I}$ is the identity matrix. Substituting this approximation into equation (2.2-29) gives the update rule for the LM algorithm:

$$\Delta_{q+1} = \Delta_q + \left(\boldsymbol{H} + \boldsymbol{\mu}_q \boldsymbol{I}\right)^{-1} \boldsymbol{J}_q^T \boldsymbol{e}_q \tag{2.2-31}$$

The combination coefficient, μ, is modified during each iteration. When an iteration results in a decrease in sum squared error:

$$\mu = \frac{\mu}{\gamma} \tag{2.2-32}$$

When an iteration results in an increase in SSE:

$$\mu = \gamma\mu \tag{2.2-33}$$

Typically $\mu = 0.01$ and $\gamma = 10$ is the starting point for the algorithm. Notice that when μ is large, the parameters are adjusted according to the steepest descent algorithm. When μ is small, the parameters are adjusted according to the Gauss-Newton algorithm. For this reason, the Levenberg-Marquardt algorithm can be considered a trust region modification of the Gauss-Newton method [41]. Below is a description of training with the Levenberg-Marquardt Algorithm.

Levenberg-Marquardt Algorithm

Given the notations described above, a single iteration of the Levenberg-Marquardt algorithm executes as follows.

Step 1. Initialization: Calculate the error E_1. Set an initial value for the combination coefficient, μ. Set the tuning parameter, γ. Choose an acceptable error threshold, ε. Choose a maximum number of iterations, Q. Initialize the input parameters, Δ_1. Set the iteration number, $Q = 1$.

Step 2. Optimization: While $E_q < \varepsilon$ or $q > Q$

(a) Calculate the Jacobian matrix, J, according to equation (2.2-24).

(b) Calculate the quasi-Hessian matrix according to equation (2.2-28).

(c) Adjust the parameters according to the LM update rule (2.2-29).

(d) Increment q.

(e) Calculate new error E_q.

(f) If $E_q < E_{q+1}$

```
            i. Adjust $\mu$ according to equation (2.2-32).
    (g) Else
            i. Adjust $\mu$ according to equation (2.2-33).
        End If
        End While
```

Several second–order algorithms have been adopted for use in the neural network training process. The most efficient of these is the Levenberg-Marquardt algorithm [42]. This algorithm uses a Hessian matrix computation to gain information about the shape of the error surface, and apply it to find the best search direction. This algorithm was shown to be very fast and efficient for relatively small problems.

2.2.5 Improved Hessian Computation

It should also be noted that the computation of the Jacobian matrix is very expensive and often leads to problems when the data set is very large. However, this problem can be mostly eliminated by changing the way that the matrices are multiplied so that a large Jacobian matrix (proportional in size to the size of the training data) is never computed and stored [43].

Assuming the same notation as in the previous section, this modification is done by changing the method of matrix multiplication when calculating the quasi-hessian matrix, $\boldsymbol{H}$. Instead of multiplying the Jacobian by its transpose in the traditional way, each row is multiplied by itself to form a part of the Hessian matrix denoted as $\boldsymbol{h}$. Then each $\boldsymbol{h}$, is added to create the full Hessian. The gradient can then also be computed in the same way. **Figure**

2.2-2 shows the difference between the two multiplication approaches. More formally, the specific computations required are shown below.

$$\boldsymbol{j}_{i,k} = \left[\frac{\partial e_{i,k}}{\partial \Delta_{q,1}}, \frac{\partial e_{i,k}}{\partial \Delta_{q,2}}, \dots, \frac{\partial e_{i,k}}{\partial \Delta_{q,P}}\right] \qquad (2.2\text{-}34)$$

$$\boldsymbol{h}_{i,j} = \boldsymbol{j}_{i,j}^{T}\boldsymbol{j}_{i,j} \qquad (2.2\text{-}35)$$

$$H = \sum_{i=1}^{N}\sum_{j=1}^{m} h_{i,j} \qquad (2.2\text{-}36)$$

$$\boldsymbol{g} = \boldsymbol{j}_{i,j}e_{i,j} \qquad (2.2\text{-}37)$$

The H calculated here is identical to the previously calculated matrix, but the Jacobian need not be calculated and stored. Only a single row at a time is used.

The Hessian in the improved LM-Method is then calculated using the following algorithm.

Improved Hessian Computation

```
Step 1.   Initialization: $\boldsymbol{H} = 0, \boldsymbol{g} = 0$.
Step 2.   Multiplication: For $i = 1,2,\dots,N$
            (a) For $j = 1,2,\dots,m$
                  i.   Calculate $\boldsymbol{j}_{i,m}$ using equation (2.2-34).
                  ii.  Calculate $\boldsymbol{h}_{i,m}$ using equation (2.2-35).
                  iii. Calculate $\boldsymbol{g}_{i,m}$ using equation (2.2-37).
                  iv.  $\boldsymbol{H} = \boldsymbol{H} + \boldsymbol{h}_{i,m}$
                  v.   $\boldsymbol{g} = \boldsymbol{g} + \boldsymbol{g}_{i,m}$
            (b) End for
Step 3.   End For
```

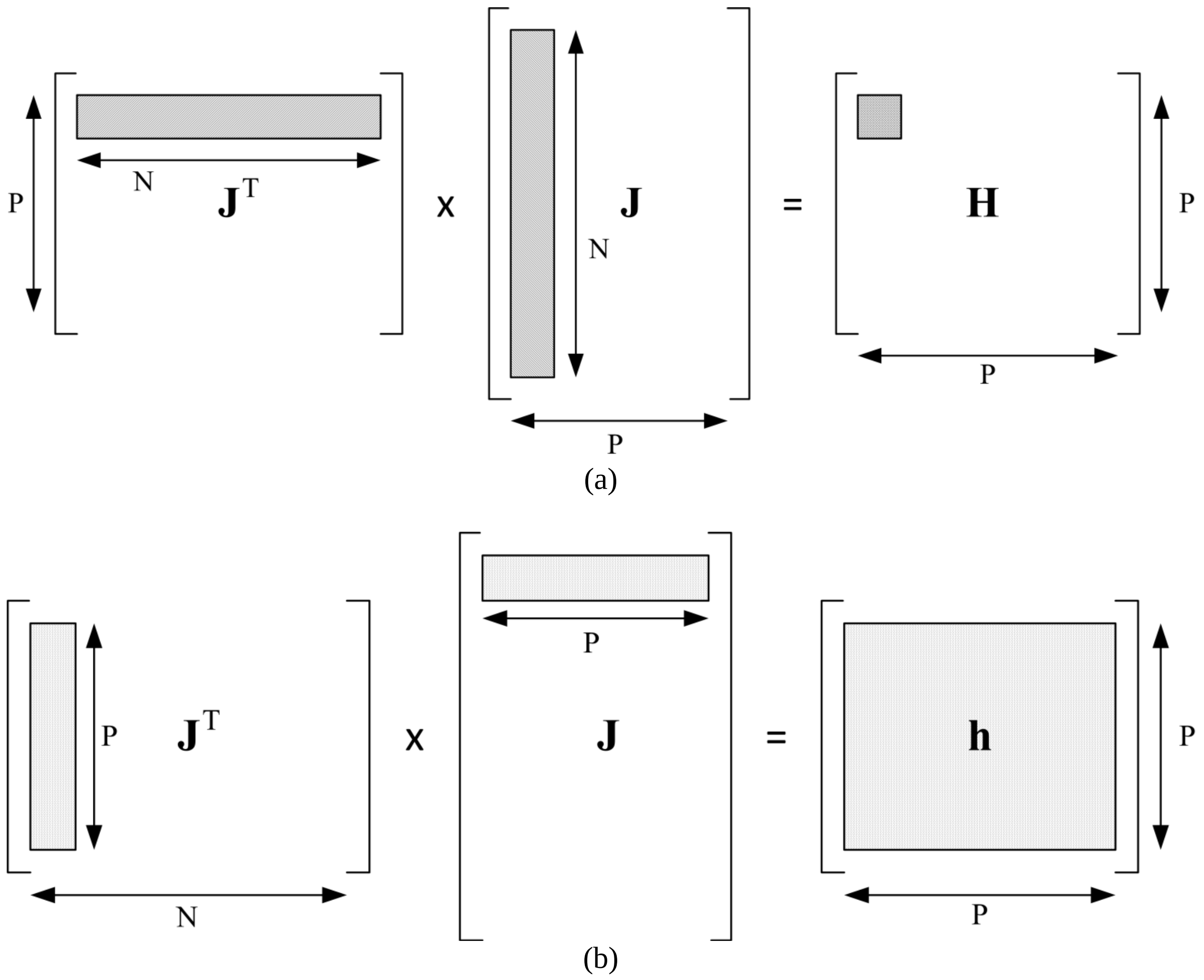


Figure 2.2-2 Illustration of the two ways of matrix multiplication. (a) The typically used method. (b) The method requiring only one row to be computed.

This allows the Hessian to be computed without storing the entire Jacobian matrix. This improved second-order (ISO) training method is the basis for the radial basis function training algorithms described in Chapter 4.

2.3 Optimal Construction and Training

All of the algorithms discussed in section 2.2 are viable and commonly used in the industry to train artificial neural networks. However, the ramifications of selecting different network architectures is largely un-addressed by these algorithms. Most of them assume a fixed architecture that is chosen before the training process begins. In order to be as efficient as possible, it is proposed that the neural networks be constructed in an optimal sense. Much of the research is dedicated to training neural networks to reach very small errors on the training data set. However, it is often the case that this is not the best measure for the effectiveness of the neural network. For practical applications, the generalization ability of the network is far more important than the training errors. For this reason many of the current publications are focusing on minimizing the so called “testing errors”, errors of the network on patterns that were not seen in training.

It can be readily seen that a neural network can be easily made to converge to nearly zero training error with an excessive number of neurons and weights. This is commonly known as overtraining and is akin to using a polynomial approximation with too many terms. The comparative works [29] and [30] show that architectures with the smallest numbers of neurons and weights often show the best performance on the testing sets. So, the motivation of new algorithms is to find a way to construct a neural network architecture in such a way that the number of neurons (and weights) can be minimized.

As discussed previously, RBF networks are often constructed in a simple three layer architecture containing, an input layer, an output layer, and a single hidden layer of RBF neurons. With this in mind, the problem of creating the smallest network possible is simplified to minimizing the number of RBF neurons in the hidden layer. This means that an optimal construction and training algorithm is one that allows each RBF neuron to have as much of an effect as possible on the reduction of errors as possible while still providing good generalization abilities. This is the guiding concept behind the algorithms presented in this work.

There are many algorithms that attempt to find the best initializations of RBF neurons and optimal sizes of the RBF networks [44]–[47]. The RAN algorithm adjusts parameters of an existing network and adds new neurons to compensate for poor performance on certain input data [11]. The RANEKF algorithm uses an extended kalman filter (EKF) procedure rather than an LMS procedure to update the network parameters [12]. In this sequential learning method, the network is initialized as a blank slate, no neurons have been allocated to store any input patterns. Let us introduce the term δ_i to denote the distance between the nearest RBF center and the i^{th} pattern.

$$\delta = \|x_i - c_{nearest}\| \tag{2.3-1}$$

where $c_{nearest}$ is the center of the nearest existing neuron.

2.3.1 Resource Allocating Network (RAN and RANEKF) Algorithm

Given a training set as described in equation (2.2-1), an activation function, $h(x)$, a maximum number of hidden neurons $\widetilde{N}$, a required error threshold, ε, a minimum nearest distance, d, and RBF parameters center, $c,$ radius σ, and height, β.

Step 1. For each input, target pair, (x_i, t_i).

(a) Evaluate the network output, o_i.

(b) Calculate the magnitude of the error $|e_i|$.

(c) Calculate the distance between the new pattern and the nearest existing RBF center, δ.

(d) If $\|e_i\| > \varepsilon$ AND $\delta > d$,

i. Create a new neuron with center at:

$$c_i = x_i. \tag{2.3-2}$$

```
            ii.  Set the output weight,

                 $\beta_i = t_i - o_i$                                    (2.3-3)

            iii. Set the radius of the unit proportionally to the distance between the
                 new center and the nearest existing center:

                 $\sigma_i = k\|c_i - c_{nearest}\|.$
                                                                          (2.3-4)

        (e) Else, adjust the existing network parameters using the new pattern and the
            LMS (or EKF) method.
        End If
    End For
```

Further improvement was made on the RAN algorithm by Yingwei *et. al.* [13]. This algorithm creates a minimally sized network and is known as MRAN. The MRAN algorithm removes neurons that have little significance on the error thus reducing the size of the network. At the end of each training iteration of the RAN algorithm, the RMS value of the network error is evaluated over a window for each neuron. If the neuron contributes to the reduction of the error in a window centered on the neuron, it is kept in the network. Otherwise, it is discarded.

Huang *et al.* proposed a generalized growing and pruning (GGAP) algorithm to find the proper sizes of RBF networks [16], [48]. Based on a measure of so called "significance" RBF units are added one at a time with specified initial conditions. A detailed description of the GGAP algorithm is given below.

2.3.2 Generalized Growing and Pruning Algorithm

Given a training set as described in equation (2.2-1), an activation function, $h(x)$, a maximum number of hidden neurons $\widetilde{N}$, a minimum error threshold, ε, a minimum distance between new data and existing centers, d, and RBF parameters center, c, radius σ, and height, β.

```
Step 2.   Initialization: Create a RBF SLFN with $n$ hidden neurons. Typically randomly
          chosen neurons are used, but the network can be initialized in any way.
Step 3.   Learning: For each input pattern and target pair, $(x_i, t_i)$
    (a) Evaluate the network output, $o_i$.
    (b) Calculate the magnitude of the error, $|e_i|$
    (c) Calculate $\delta$ according to equation (2.3-1).
    (d) Calculate the SSE according to equation (2.2-3).
    (e) If the parameters calculated in b, c, and d are greater than the pre-set threshold
        values, create a new RBF neuron centered at $x_i$.
    (f) Else, adjust the network parameters using the EKF method.
    (g) For each neuron,
        i.   evaluate the pruning criterion, significance:
                 $s_i = \left|\frac{1.8\sigma_i\beta_i}{r}\right|$        (2.3-5)
        ii.  If $s_i < S$, prune the neuron.
    End For
End For
```

Neurons that make little contribution to network performance (low significance) are eliminated from the network and ignored. However, the GGAP algorithm has trouble with problems with complex probability distributions and high dimensional data. However, the Gaussian Mixture Model (GMM) was introduced to approximate the GGAP performance evaluation formula [49]. This modification allows the GGAP-GMM algorithm to design even more compact networks for the same tasks on which the original GGAP algorithm failed.

2.3.3 Support Vector Machines

Support vector machine (SVM) learning attempts to minimize the number of computational nodes required to solve a particular problem. This is done by selecting certain patterns to be used as training data while the rest of the data is ignored. SVMs can use nonlinear kernel functions to cast inputs into higher dimensional spaces. This concept was introduced by Vladimir Vapnik in his book [50]. A description of SVMs used for the purpose of regression, Support Vector Regression (SVR), is given below.

Given a training set as described in equation (2.2-1), we introduce the approximation:

$$y_i = w \cdot x_i + b \tag{2.3-6}$$

Where y_i is the predicted output of the SVR.

The SVR will use a more sophisticated cost function than SSE. There will be no penalty associated with predicted values that are within some maximum distance, ϵ, of their associated target values. Furthermore, two slack variables, γ^+ and γ^-, are assigned to vary the penalty associated with predicted values that lie outside ϵ. The conditions required are then written:

$$\begin{cases} (t_i - y_i) \leq \epsilon + \gamma^+, & (t_i - y_i) > 0 \\ (t_i - y_i) \geq -\epsilon - \gamma^-, & (t_i - y_i) < 0 \end{cases} \tag{2.3-7}$$

This leads to the cost function for SVR:

$E = C\sum_{i=1}^{N}(\gamma^{+} + \gamma^{-}) + \frac{1}{2}\|w\|^2$ (2.3-8**)**

Where the constant $C > 0$ determines the trade-off between the flatness of the approximation and tolerance of errors larger than ϵ.

Thus the SVR algorithm seeks to solve the minimization problem:

$$
\begin{aligned}
&\mathit{minimize,} \qquad C\sum_{i=1}^{N}(\gamma^{+} + \gamma^{-}) + \frac{1}{2}\|w\|^2 \\
&\mathit{subject\ to,} \qquad \begin{cases} (t_i - y_i) \leq \epsilon + \gamma^{+} \\ (t_i - y_i) \geq -\epsilon - \gamma^{-} \\ \qquad \gamma^{+} \geq 0 \\ \qquad \gamma^{-} \leq 0 \end{cases}
\end{aligned}
\tag{2.3-9}
$$

Let us now find a dual set of variables by constructing a Lagrange function and the corresponding constraints from the objective function **(**2.3-9). Let us introduce the Lagrange multipliers, $\mu_i^{+}, \mu_i^{-}, \alpha_i^{+}, \alpha_i^{-}$. The Lagrange function is then:

$$
\begin{aligned}
L = \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{N}(\gamma^{+} + \gamma^{-}) - \sum_{i=1}^{N}(\mu_i^{+}\gamma^{+} + \mu_i^{-}\gamma^{-}) - \sum_{i=1}^{N}\alpha_i^{+}(\epsilon + \gamma^{+} + y_i - t_i) \\
- \sum_{i=1}^{N}\alpha_i^{-}(\epsilon + \gamma^{-} - y_i + t_i)
\end{aligned}
\tag{2.3-10}
$$

Where the Lagrange multipliers are positive values:

$\mu_i^{+}, \mu_i^{-}, \alpha_i^{+}, \alpha_i^{-} \geq 0$ (2.3-11**)**

Because the formulation of the problem is quadratic, the min or max will be located where the partial derivatives of the Lagrange function equal zero.

$$\frac{\partial L}{\partial b} = \sum_{i=1}^{N}(\alpha_i^{+} - \alpha_i^{-}) = 0 \tag{2.3-12}$$

$$\frac{\partial L}{\partial w} = w - \sum_{i=1}^{N}(\alpha_i^{+} - \alpha_i^{-})x_i = 0 \tag{2.3-13}$$

$$\frac{\partial L}{\partial \gamma^+} = C - (\alpha_i^+ + \mu_i^+) \tag{2.3-14}$$

$$\frac{\partial L}{\partial \gamma^-} = C - (\alpha_i^- + \mu_i^-) \tag{2.3-15}$$

Re-writing these equations and substituting them into the Lagrange function leads to the formulation of the Lagrangian dual:

$$\begin{aligned} L_D = \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}(\alpha_i^+ - \alpha_i^-)(\alpha_j^+ - \alpha_j^-)(\boldsymbol{x}_i \cdot \boldsymbol{x}_j) + \epsilon\sum_{i=1}^{N}(\alpha_i^+ - \alpha_i^-) \\ - \sum_{i=1}^{N}(\alpha_i^+ - \alpha_i^-)t_i \end{aligned} \tag{2.3-16}$$

Notice that the dual variables μ_i^+ and μ_i^- were eliminated by solving equations (2.3-14) and (2.3-15) for α_i^+ and α_i^-. Now, the problem in (2.3-9) can be re-formulated as maximizing the negative of the Lagrange dual problem:

$$\begin{aligned} &maximize, \qquad -L_D \\ &subject\ to, \qquad 0 \le \alpha_i^+ \le C, 0 \le \alpha_i^- \le C, and \sum_{i=1}^{N}(\alpha_i^+ - \alpha_i^-) = 0 \end{aligned} \tag{2.3-17}$$

This is done using a quadratic programming algorithm. Predictions may then be made by substituting (2.3-13) into (2.3-6).

$$y_j = \sum_{i=1}^{N}(\alpha_i^+ - \alpha_i^-)(\boldsymbol{x}_i \cdot \boldsymbol{x}_j) + b \tag{2.3-18}$$

A set of support vectors S can be created by finding the inputs that satisfy the following:

$$x_s \in S \text{ iff } 0 < \alpha_i^{\pm} < C \text{ and } \gamma_i^{\pm} = 0 \tag{2.3-19}$$

The number of support vectors is then defined as N_s. This allows the bias to be calculated:

$$b = \frac{1}{N_s}\sum_{s=1}^{N_s}\left[t_s - \epsilon - \sum_{k=1}^{N_s}(\alpha_k^+ - \alpha_k^-)\boldsymbol{x}_k \cdot \boldsymbol{x}_s\right] \tag{2.3-20}$$

In order to move to higher dimensional space, a kernel function $k(\boldsymbol{x}_i, \boldsymbol{x}_j)$ is used. The kernel function must be a function of the inner product between $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$. The kernel must also satisfy several other criteria explained in great detail in Alex Smola's tutorial [51]. Popular kernels include, polynomial kernels, RBF kernels, and sigmoidal kernels. For the purpose of this work, the Gaussian kernel will be used in comparisons with other algorithms. Inserting a kernel into the formulas derived above is as easy as substituting $k(\boldsymbol{x}_i, \boldsymbol{x}_j)$ in place of $\boldsymbol{x}_i \cdot \boldsymbol{x}_j$.

Given all of these considerations, let us outline the SVR algorithm:

Support Vector Regression Algorithm: Given a training set as described in equation (2.2-1), a kernel function $k(x_i, x_j)$, an error significance C, and an error insensitivity parameter ϵ. For simplicity, let us denote the alpha parameters in the following way:

$$\alpha_i = (\alpha_i^+ - \alpha_i^-) \tag{2.3-21}$$

Let us also denote the matrix of kernel function values in the following way:

$$H = \begin{bmatrix} H_{1,1}, H_{1,2}, \dots, H_{1,N} \\ \vdots \\ H_{N,1}, H_{N,2}, \dots, H_{N,N} \end{bmatrix} = \begin{bmatrix} k(x_1, x_1), k(x_1, x_2), \dots k(x_1, x_N) \\ \vdots \\ k(x_N, x_1), k(x_N, x_2), \dots k(x_N, x_N) \end{bmatrix} \tag{2.3-22}$$

Step 1. Calculate $\boldsymbol{H}$ according to equation (2.3-22).

Step 2. Us a Quadratic Programming algorithm to solve the dual problem:

$$\begin{gathered} maximize, \quad \sum_{i=1}^{N} \alpha_i (t_i - \epsilon) - \frac{1}{2} \alpha^T H \alpha \\ subject\ to, -C \le \alpha_i \le C\ \forall_i\ and\ \sum_{i=1}^{N} \alpha_i = 0 \end{gathered} \tag{2.3-23}$$

Step 3. Calculate weights according to equation (2.3-13).

Step 4. Determine the set of support vectors as in equation (2.3-19).

Step 5. Calculate the bias with equation (2.3-20).

Step 6. New data, $\boldsymbol{x}'$ is processed using:

$$y' = \sum_{i=1}^{N} \alpha_i k(x_i, x') + b \tag{2.3-24}$$

In terms of neural networks, SVMs can be viewed as a single hidden layer network with a bias weight and activation functions equal to the kernel function. For instance, a sigmoidal kernel will lead to a single layer MLP network.

2.4 Extreme Learning Machines

From the perspective of mathematics, the approximation capabilities of feed-forward neural networks has been largely dedicated to two problems: universal approximation on small input sets, and universal approximation over a finite set of training patterns. In fact, much research has been done on the capabilities of a typical multilayer feed-forward network. Hornik, [52], showed that neural networks can approximate continuous mappings over compact input sets if the activation function is continuous, bounded and non-constant. Further work on the subject has been done by Leshno, [53], who proved that feed-forward networks with a non-polynomial activation function can approximate any continuous function. Huang and Babri, [54] showed mathematical proof that a single-hidden layer feed-forward neural network (SLFN) with *N* or fewer nodes can learn *N* distinct training patterns.

In all of these previous theoretical works, the network parameters, weights and biases, need to be adjusted iteratively in order for the networks to learn the data. In most cases, gradient descent-based optimization is the core of the learning algorithm. However, it has been postulated that gradient descent-based algorithms have issues that reduce their efficiency and desirability as learning algorithms. For instance, the size of the learning step must be chosen carefully or else the algorithm will converge slowly or converge to a local minimum. Furthermore, it is often the case that many attempts need to be made in order to obtain satisfactory learning performance. Huang showed in [55] that an SLFN with N hidden nodes and randomly chosen input weights and biases can exactly learn N distinct training patterns. This goes against the traditional thinking on the subject because it demonstrates that the input weights may not always need to be adjusted. Actually, Huang *et. Al*, did some simulations on artificial and real-world applications in [56], and found that learning with random input weights and biases makes learning extremely fast and also produces good generalization performance. The Extreme Learning Machine (ELM) algorithm was spawned from this idea and was proposed in [57].

In [57] it is rigorously proven that the input weights and hidden layer biases of SLFNs can be randomly assigned if the activation functions of neurons in the hidden layer are infinitely differentiable. In other words:

Given a training set as described in equation (2.2-1), a SLFN with $\widetilde{N}$ hidden nodes and an activation function, $h(x)$ is mathematically modeled as:

$$\sum_{i=1}^{\widetilde{N}} \beta_i h_i(x_j) = \sum_{i=1}^{\widetilde{N}} \beta_i h_i(w_i \cdot x_j + b_i) = o_j, j = 1, \dots, N \tag{2.4-1}$$

$$w_i = [w_{i,1}, w_{i,2}, \dots, w_{i,n}] \in \mathbb{R}^n \tag{2.4-2}$$

$$\beta_i = [\beta_{i,1}, \beta_{i,2}, \dots, \beta_{i,m}] \in \mathbb{R}^m \tag{2.4-3}$$

Where: w_i is the weight vector that weights the connections between the i^{th} hidden node and the inputs; β_i is the weight vector that connects the outputs and the i^{th} hidden node; and b_i is the bias of the i^{th} hidden node. The term, $w_i \cdot x_j$, is the inner product of w_i and x_j.

Let us say that we wish for a standard SLFN with $\widetilde{N}$ hidden nodes with activation function $h(x)$ to approximate the N training samples with zero error. In other words:

$$\sum_{j=1}^{\tilde{N}} \| o_j - t_j \| = 0, \tag{2.4-4}$$

This requires that $\exists \beta_i, w_i$, and b_i such that:

$$\sum_{i=1}^{\tilde{N}} \beta_i h_i(w_i \cdot x_j + b_i) = t_j, j = 1, \dots, N \tag{2.4-5}$$

To use a more compact notation, the above N equations can be re-written as:

$$H\beta = T \tag{2.4-6}$$

$$H(w_1, \dots, w_{\tilde{N}}, b_1, \dots, b_{\tilde{N}}, x_1, \dots, x_N) =$$

$$\begin{bmatrix} h(w_1 \cdot x_1 + b_1) & \cdots & h(w_{\tilde{N}} \cdot x_1 + b_{\tilde{N}}) \\ \vdots & \cdots & \vdots \\ h(w_1 \cdot x_N + b_1) & \cdots & h(w_{\tilde{N}} \cdot x_N + b_{\tilde{N}}) \end{bmatrix} \tag{2.4-7}$$

$$\beta = \begin{bmatrix} \beta_1 \\ \vdots \\ \beta_{\tilde{N}} \end{bmatrix} \tag{2.4-8}$$

$$T = \begin{bmatrix} t_1 \\ \vdots \\ t_N \end{bmatrix} \tag{2.4-9}$$

Where the size of H is $(N \times \tilde{N})$, the size of β is $(\tilde{N} \times m)$, and the size of T is (Nxm). From here forward, H is called the hidden layer output matrix of the SLFN. The rows of H correspond to the inputs while the columns of H correspond to the hidden layer neurons such that H_{ij} is equal to the output of the j^{th} neuron given the i^{th} input. Given this network, it is proven in [57] that the required number of nodes $\tilde{N} \leq N$.

Notice that in a conventional neural network, H is found by iteratively searching for the minimum of $\|H\beta - T\|$ using a gradient-based learning algorithm. Recall from section 2.2 that the weights and biases are typically adjusted by the following formula:

$$w_k = w_{k-1} - \alpha \frac{\delta E(w)}{\delta w} \tag{2.4-10}$$

This learning rule is generally implemented in the form of the back propagation (BP) algorithm. Huang states that there are four problems with BP algorithms:

(1) When the learning rate α is too large, the algorithm becomes unstable and diverges. However, when α is too small, the algorithm converges very slowly and faces greater risk of getting stuck in a local minimum.

(2) The presence of local minima can greatly affect performance of the learning algorithm. Local minima are often traps where the learning algorithm will stop before it reaches a global minima.

(3) The BP algorithms can lead to overtraining thereby hurting generalization performance of the network. This gives rise to the necessity for complicated stopping methods in the cost minimization algorithm.

(4) In most applications, gradient-based learning is very time consuming.

The ELM algorithm then proposes that the input weights and biases are allowed to be random and unchanged and that a least squares solution, $\hat{\beta}$, of the linear system from equation (2.4-6) will yield a suitable approximation:

$$\left\|H(w_1, \dots, w_{\tilde{N}}, b_1, \dots, b_{\tilde{N}})\hat{\beta} - T\right\| = \text{MIN}_{\beta}\|H(w_1, \dots, w_{\tilde{N}}, b_1, \dots, b_{\tilde{N}})\beta - T\| \qquad (2.4\text{-}11)$$

Which is equivalent to minimizing the cost function:

$$E = \sum_{j=1}^{N}\left(\sum_{i=1}^{\tilde{N}} \beta_i h_i\left(w_i \cdot x_j + b_i\right) - t_j\right)^2 \qquad (2.4\text{-}12)$$

If the number of hidden neurons is equal to the number of training samples, then the matrix H is square and invertible. The SLFN will approximate these training samples with zero error. However, most of the time the number of neurons is far fewer than the number of training samples and there may not exist a set of parameters such that equation (2.4-6) is satisfied. In these cases, we must find the smallest norm least-squares solution by:

$$\hat{\beta} = H^{\dagger}T \qquad (2.4\text{-}13)$$

Where $H^{\dagger}$ is the Moore-Penrose generalized inverse of H.

This gives the algorithm the following properties:

(1) *Minimum Training Error*: The special solution $\hat{\beta} = H^{\dagger}T$ is a least squares solution of the general linear system $H\beta = T$. This means that the smallest training error can be reached by the special solution:

$$\left\|H\hat{\beta} - T\right\| = \left\|HH^{\dagger}T - T\right\| = min_{\beta}\|H\beta - T\| \qquad (2.4\text{-}14)$$

Of course all learning algorithms attempt to reach the minimum training error, however, many of them cannot reach it because of the problems discussed previously or because it would require an infinite number of training iterations.

(2) *Smallest Norm of Weights*. The special solution $\hat{\beta} = H^{\dagger}T$ has the smallest norm of all the least-squares solutions of $H\beta = T$.

$$\left\|\hat{\beta}\right\| = \left\|H^{\dagger}T\right\| \leq \|\beta\|, \forall\beta \in \left\{\beta : \|H\beta - T\| \leq \|Hz - T\|, \forall z \in \mathbb{R}^{\tilde{N}\times N}\right\} \qquad (2.4\text{-}15)$$

According to Bartlett, [58], the set of parameters with the smallest norm will provide the best generalization performance.

(3) The minimum norm least-squares solution of $H\beta = T$ is unique.

Thus the ELM algorithm can be summarized as follows:

ELM Algorithm: Recall a training set as described in equation (2.2-1) has the form:

$$\aleph = \{(x_i, t_i) | x_i \in \mathbb{R}^n, t_i \in \mathbb{R}^m, i = 1, \dots, N\} \qquad (2.4\text{-}16)$$

Step 1. Randomly assign input weight, w_i, and bias, b_i, for $i = 1, \dots, \tilde{N}$.

Step 2. Calculate the hidden layer output matrix, H.

Step 3. Calculate the output weights, β, using equation (2.4-13).

This algorithm was tested on several benchmarks and real-world problems in [57]. The algorithm was also compared with two state of the art learning algorithms in the Levenberg-Mardquardt BP algorithm and the Support Vector Machine algorithm. The simulations for the BP and ELM algorithms were carried out in MATLAB while the SVM algorithms were run using the compiled C-coded package, LIBSVM. Though the ELM can be used on any infinitely differentiable activation function, the simple sigmoidal function was used for both the ELM and BP trials. The kernel function used for the SVM trials was a Gaussian radial basis function. The inputs of the test data sets were normalized into the range, $[0,1]$, while the outputs were normalized into the range, $[-1,1]$. Table 1 summarizes the results found by the experiments. It can be seen that the ELM algorithm performs very well in terms of training time. It is also efficient in terms of network size and root mean square error (RMSE) performance. The RMSE is defined as:

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N} e_i^2} \tag{2.4-17}$$

Table 3 ELM, SVR, and BP results on real world data sets.

Data Set	BP			SVR			ELM		
	Train RMSE	Test RMSE	Train Time	Train RMSE	Test RMSE	Train Time	Train RMSE	Test RMSE	Train Time
Abalone	0.0785	0.0874	1.7562	**0.0759**	**0.0784**	1.6123	0.0803	0.0824	**0.0125**
Auto Price	**0.0443**	0.1157	0.2456	0.0652	**0.0937**	0.0042	0.0754	0.0994	**0.0016**
Cal Housing	0.1046	0.1285	6.532	0.1089	**0.118**	74.184	0.1217	0.1267	**1.1177**
Delta Ailerons	**0.0409**	0.0481	2.7525	0.0418	**0.0429**	0.6726	0.0423	0.0431	**0.0591**
Delta Elevators	0.0544	0.0592	1.1938	**0.0534**	**0.054**	1.121	0.0545	0.0568	**0.2812**
Machine CPU	0.0352	0.0826	0.2354	0.0574	0.0811	0.0018	**0.0332**	**0.0539**	**0.0015**

2.4.1 Incremental Extreme Learning Machine

Following the work done on the Extreme Learning Machine in [57] Huang *et. al* in [59] extended those ideas to prove that given any bounded non-constant piecewise continuous activation function, the network output, f_n, can converge to any continuous target function, f, by only adjusting the output weights that connect the hidden layer to the output neurons and fixing the hidden layer parameters to random values. This research showed that one could also choose special neural network activation functions such as RBFs and still assign input parameters, centers and widths in the case of RBF, randomly while maintaining good approximation characteristics. Furthermore, they proposed a constructive algorithm known as the Incremental Extreme Learning Machine (I-ELM) which is a very important algorithm related to the work being done here. Like the ELM the I-ELM algorithm focuses on constructing SLFNs, but is proven to work for Two Hidden Layer Feedforward Networks (TLFNs) as well. The I-ELM algorithm works in the same way as the ELM algorithm with the exception that random neurons are added to the network one at a time. Then the output weights of these neurons are adjusted using the least squares solution with the current residual error as a target. Finally the newly added neuron's function is subtracted from the current residual error and the process repeats until a desired level of total error or a maximum number of neurons is reached. A diagram detailing the flow of this algorithm is shown in Figure 2.4-1 and a detailed description of the algorithm is below:

I-ELM Algorithm: Given a training set as described in equation (2.4-16), an activation function, $h(x)$, a maximum number of hidden neurons $\widetilde{N}$, and a required learning accuracy, ε.

Step 1. Initialization: Let the number of hidden neurons and the residual error be $n = 0$ and $E = T$, where T is the target vector.

Step 2. Learning: while $n < \widetilde{N}$ and $\|E\| < \varepsilon$

(h) Create a new neuron with random input parameters and increment n.

(i) Calculate the output weight,

$$\beta_n = \frac{E \cdot H_n^T}{H_n \cdot H_n^T} \tag{2.4-18}$$

(j) Calculate the residual error after adding the new hidden node n.

$$E = E - \beta_n \cdot H_n \tag{2.4-19}$$

End While.

The main differences to notice between I-ELM and the previously examined ELM are:

(1) The network is constructed one node at a time with each new node being created with random input parameters, weights and biases or centers and widths.

(2) The calculation for the output weights β changes slightly. Since only one β needs to be found at a time, the least squares solution goes from (2.4-13), to what is shown in equation (2.4-18).

The authors of the I-ELM algorithm also studied various ways of improving the algorithm. These algorithms known as the Convex Incremental Extreme Learning Machine (CI-ELM) and the Enhanced Random Search Based Incremental Extreme Learning Machine (EI-ELM) were published in [60] and [61] respectively. Together these algorithms are known as extreme learning machines. These algorithms are all built on the mathematics discussed in section 3.1. They are used in comparison with the algorithms proposed in this work because they all follow similar processes with regards to the incremental construction of RBF networks. These algorithms will have very fast training times since there is only one calculation made per iteration and most environments allow the calculation of β in matrix form to be very fast. The I-ELM algorithm was compared on a range of real world data sets and is compared with the algorithms proposed in this work in detail in section 3.3. The drawback to these algorithms is that only one of three possible input parameters is optimized. This intuitively leaves room for some improvements that will still maintain the integrity of the network being developed.

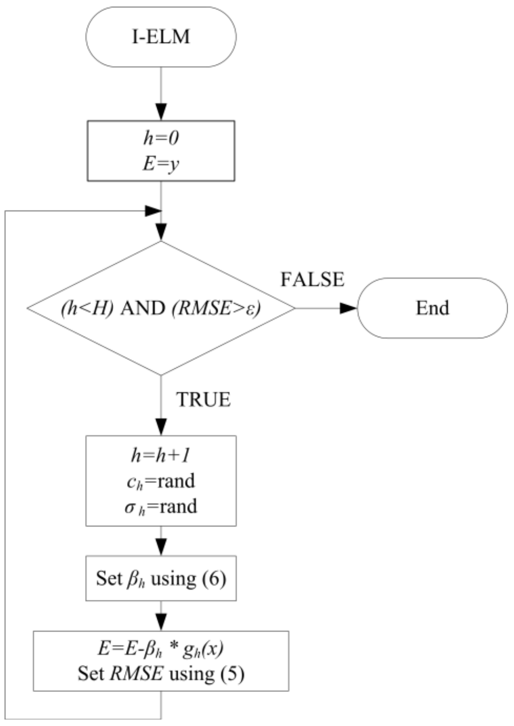


Figure 2.4-1 The I-ELM algorithm given a training set $\{(x_p, y_p)|x_p \in R^D, y_p \in R, p = [1 \dots P]\}$, an activation function g(x), a maximum node number H, and an expected learning accuracy ε.

2.4.2 Convex Incremental Extreme Learning Machine

The CI-ELM algorithm is an attempt to improve upon the methods proposed in [57] and [59]. This algorithm functions almost identically to the I-ELM. The only difference is that the solutions for β_n and β are found using Barron's convex optimization learning method [62]. This method first estimates the output weight of the newly added neuron using:

$$\beta_n = \frac{E \cdot [E-(F-H_n)]^T}{[E-(F-H_n)] \cdot [E-(F-H_n)]^T} \qquad (2.4\text{-}20)$$

Where $H_n$is the vector containing the output of the new node for each input pattern, E is the residual error vector before the new node was added, and F is the target vector containing each target.

Then the algorithm adjusts all of the existing output weights as well using:

$$\beta_{i,new} = (1-\beta_n)\beta_i, \quad i = 1, \dots, n-1 \qquad (2.4\text{-}21)$$

Finally, the residual error is calculated again with the new node in the network using:

$$E = (1-\beta_n)E + \beta_n(F-H_n) \qquad (2.4\text{-}22)$$

By using these equations to find β, the authors of the CI-ELM hoped to make use of more information and therefore find a better approximation of the targets. The full description of the CI-ELM algorithm is given below:

CI-ELM Algorithm: Given a training set as described in equation (2.4-16), an activation function, $h(x)$, a maximum number of hidden neurons $\widetilde{N}$, and a required learning accuracy, ε.

Step 1. Initialization: Let the number of hidden neurons and the residual error be $n = 0$ and $E = T$, where T is the target vector.

Step 2. Learning: while $n < \widetilde{N}$ and $\|E\| < \varepsilon$

(k) Create a new neuron with random input parameters and increment n.

(l) Estimate the output weight, β_n, for the newly added hidden node according to equation (2.4-20).

(m) If $n > 1$, recalculate the output weight vectors for all previously existing hidden neurons according to equation (2.4-21).

(n) Re-calculate the residual error after the addition of the new hidden neuron according to equation (2.4-22).

End While.

The CI-ELM was compared on a range of real world data sets and is compared with the algorithms proposed in this work in detail in section 3.3.

2.4.3 Enhanced Random Search Incremental Extreme Learning Machine

The EI-ELM algorithm proposed in, [61] is an attempted improvement to the I-ELM algorithm. This algorithm differs from I-ELM in that at each iteration, instead of a single random node being created and added into the network, an array of random neurons of length k is created. Then the output weight for each neuron is calculated using the same equation used in I-ELM equation (2.4-18). Then a vector of errors is calculated for each of the k neurons. The neuron which results in the smallest error value is then added to the network. A detailed description of the algorithm is given below.

EI-ELM Algorithm: Given our typical training set from equation (2.4-16), an activation function, $h(x)$, a maximum number of hidden neurons $\widetilde{N}$, a maximum number of trials, k, and a required learning accuracy, ε.

Step 1. Initialization: Let the number of hidden neurons and the residual error be $n = 0$ and $E = T$, where T is the target vector.

Step 2. Learning: While $n < \widetilde{N}$ and $\|E\| < \varepsilon$

(a) Increment n.

(b) For $i = 1\ to\ k$

i. Create a new neuron, n_i, with random input parameters.

```
        ii.  Calculate the output weight using equation (2.4-18).
        iii. Calculate the residual error after adding the new hidden neuron, $n$, using
             equation (2.4-19).
             End For
    (c) Let:
          $i^* = \{i| \min_{1\leq i\leq k} \|E_i\|\}$                                  (2.4-23)
    (d) Set neuron, $n = n_{i^*}$, output weight, $\beta_n = \beta_{i^*}$, and residual error, $E = E_{i^*}$.
    End While
```

The EI-ELM was compared on a range of real world data sets and is compared with the algorithms proposed in this work in detail in section 3.3.

Chapter 3 Nelder-Mead Enhanced Extreme Learning Machine

3.1 Nelder-Mead Simplex Method

The Nelder-Mead algorithm is an optimization algorithm that uses a quasi-gradient descent method to find the minimum of a real valued function. It was originally published in [63]. Since then it has been widely used in a myriad of applications. Its popularity stems from the fact that it is unconstrained and does not require the computation of derivatives of the function to be optimized. However, many studies such as what is presented in [64] show that the Nelder-Mead algorithm has many inefficiencies. Some of these deficiencies were recently corrected in [65]. This algorithm will be used as a fast optimization method to improve the performance of the I-ELM algorithm discussed in section 3.2.1.

The Nelder-Mead algorithm was proposed as a method for minimizing a real-valued function $f(x)$ for $x \in R^n$. According to [63], four scalar parameters must be specified to define a complete Nelder-Mead method: coefficients of *reflection* (ρ), *expansion* (χ), *contraction* (γ), and *shrinkage (α)*. According to the original publication, these parameters should satisfy:

$$\rho > 0, \chi > 1, \chi > \rho, 0 < \gamma < 1, and\ 0 < \alpha < 1 \qquad (3.1\text{-}1)$$

In almost all cases (and in experiments conducted in section 3.3) these parameters are chosen to be:

$$\rho = 1, \quad \chi = 2, \quad \gamma = \frac{1}{2}, and\ \alpha = \frac{1}{2} \qquad (3.1\text{-}2)$$

Nelder-Mead Simplex Algorithm

At the beginning of the k^{th} iteration, $k \geq 0$, a non-degenerate simplex Δ_k is given, as well as its $n+1$ vertices, each of which is a point in R^n. It is always assumed that iteration k begins by ordering and labeling these vertices as $x_1^k, \dots, x_{n+1}^k$, such that $f_1^k \leq f_2^k \leq \cdots \leq f_{n+1}^k$. Where f_1^k denotes $f\left(x_1^k\right)$. The k^{th} iteration generates a set of $n+1$ vertices that define a different simplex for the next iteration. In terms of minimizing f, we refer to x_1^k as the best vertex and to x_{n+1}^k as the worst vertex.

The result of each iteration is one of two cases:

(1) A single new vertex – the accepted point – replaces the vertex, x_{n+1} in the set of vertices for the next iteration.

(2) A shrink is performed and a set of *n* new points is generated that, together with x_1, form the simplex at the next iteration.

For explanation purposes in this work, an iteration of the Nelder-Mead algorithm will be described and the superscript k will be omitted to avoid confusion. The algorithm explanation shown in this work was extracted from the explanatory publication [63].

The steps of a single iteration of the Nelder-Mead Algorithm are:

Step 1. Order the n+1 vertices so that $f(x_1) \leq f(x_2) \leq \cdots \leq f(x_{n+1})$.

Step 2. Reflection:

(a) Compute the reflection point x_r from:

$$x_r = (1+\rho)\hat{x} - \rho x_{n+1} \tag{3.1-3}$$

where $\hat{x} = \sum_{i=1}^{n} x_i / n$ is the centroid of the n best points.

(b) Evaluate:

$$f_r = f(x_r) \tag{3.1-4}$$

(c) If $f_1 \leq f_r < f_n$, accept the point and terminate the iteration.

Step 3. Expansion:

(a) If $f_r < f_n$, calculate the expansion point x_e,

$$x_e = \hat{x} + \chi(x_r - \hat{x}) \tag{3.1-5}$$

i. Evaluate $f(x_e)$

ii. If $f_e < f_r$, accept x_e and terminate the iteration.

iii. Otherwise, accept x_r and terminate the iteration.

(b) But if $f_r \geq f_n$, move to step 4.

Step 4. Contraction: Perform either and outside or inside contraction.

(a) If $f_n \leq f_r < f_n + 1$, perform an outside contraction.

i. Calculate:

$$x_c = \gamma(x_r - \hat{x}) + \hat{x} \tag{3.1-6}$$

ii. Evaluate $f(x_c)$.

iii. If $f(x_c) < f_r$ accept x_c and terminate the iteration.

iv. Otherwise, go to step 5.

(b) If $f_r \geq f_n + 1$, perform and inside contraction.

i. Calculate:

$$x_{cc} = \gamma(x_{n+1} - \hat{x}) + \hat{x} \tag{3.1-7}$$

ii. Evaluate $f(x_{cc})$.

iii. If $f(x_{cc}) < f_n + 1$, accept x_{cc} and terminate the iteration.

iv. Otherwise, go to step five.

Step 5. Shrink: Evaluate f at the n points:

$$v_i = x_1 + \alpha(x_i - x_1), i = 2, \ldots, n+1 \qquad (3.1\text{-}8)$$

The vertices of the simplex at the next iteration will consist of $x_1, v_2, \ldots, v_n + 1$. Terminate the iteration.

Typically this process only needs to be repeated 5-10 times for it to provide a very large improvement over the starting point. Figure 3.1-1 Figure 3.1-2 depict the effects of each step in two dimensions, where the simplex is a triangle. Both figures assume the values for the simplex parameters to be equal to those given in equation (3.1-2).

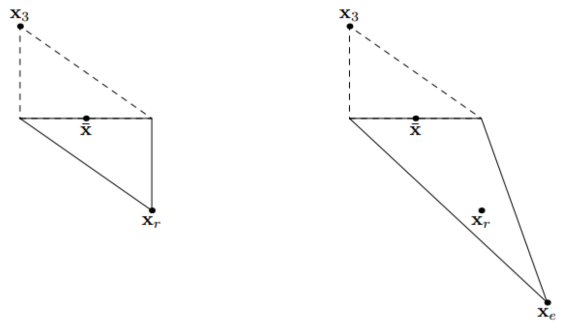


Figure 3.1-1. Nelder-Mead simplices after a reflection (left) and an expansion (right). The original simplex is shown with a dashed line.

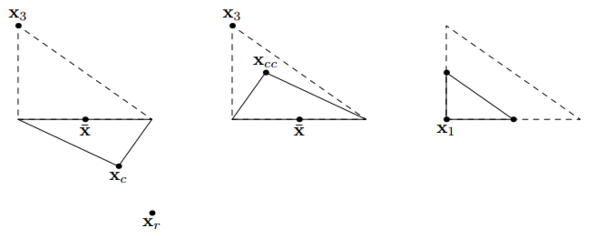


Figure 3.1-2. Nelder-Mead simplices after an outside contraction (left), an inside contraction (middle), and a shrink (right). The original simplex is shown with a dashed line.

3.1.1 Improved Nelder-Mead Simplex Method

While widely considered simple and effective, the Nelder-Mead Simplex algorithm also has its inefficiencies as studied in [64]. This study suggests that the NM algorithm fails to converge on certain problems due to the fact that the search direction becomes increasingly orthogonal to the steepest decent direction. The authors propose what they call a “Direct Search Algorithm” to replace the NM algorithm. Fuchang Gao and Lixing Han proposed an improvement to the NM algorithm in which the expansion, contraction, and shrinking parameters are adaptive with respect to the dimensionality of the problem [66]. Finally, Nam Pham and Bogdan Wilamowski improved the NM algorithm by adding a quasi-gradient calculation and selecting the new vertex point in the direction of the estimated gradient [65]. This improvement showed much improvement in convergence speeds and success rates of the algorithm. This improved algorithm is as follows:

Improved Nelder-Mead Simplex Algorithm: At the beginning of the k^{th} iteration, $k \geq 0$, a non-degenerate simplex Δ_k is given, as well as its $n+1$ vertices, each of which is a point in R^n. It is always assumed that iteration k begins by ordering and labeling these vertices as $x_1^k, \ldots, x_{n+1}^k$, such that $f_1^k \leq f_2^k \leq \cdots \leq f_{n+1}^k$. Where f_1^k denotes $f\left(x_1^k\right)$. The k^{th} iteration generates a set of $n+1$ vertices that define a different simplex for the next iteration. In terms of minimizing f, we refer to x_1^k as the best vertex and to x_{n+1}^k as the worst vertex.

The result of each iteration is one of two cases:

(1) A single new vertex – the accepted point – replaces the vertex, x_{n+1} in the set of vertices for the next iteration.

(2) A shrink is performed and a set of *n* new points is generated that, together with x_1, form the simplex at the next iteration.

As before, the description of the algorithm below is a description of a single iteration. The algorithm was extracted from the paper [65].

The steps of a single Improved Nelder-Mead Simplex algorithm iteration are:

Step 1. Order the n+1 vertices so that $f(\boldsymbol{x}_1) \leq f(\boldsymbol{x}_2) \leq \cdots \leq f(\boldsymbol{x}_{n+1})$.

Step 2. Create an extra point x_s with it's coordinates composed of n vertices in the simplex such that the coordinates are each from a different vertex in the simplex.

For example, select x_s such that:

$$x_s = diag(\begin{bmatrix} x_{1,1}, x_{1,2}, \dots, x_{1,n} \\ x_{2,1}, x_{2,2}, \dots, x_{2,n} \\ \vdots \\ x_{n,1}, x_{n,2}, \dots, x_{n,n} \end{bmatrix}) \tag{3.1-9}$$

Step 3. Calculate quasi-gradients,

$$\boldsymbol{g} = [g_1, g_2, \dots, g_n], \text{ based on the extra point, } x_s. \tag{3.1-10}$$

Step 4. For $i = 1\ to\ n$:

(a) If $i\ mod\ 2 = 0$,

$$g_i = \frac{\partial f}{\partial x_i} = \frac{f(x_{i-1,i}) - f(x_{si})}{x_{i-1,i} - x_{si}} \tag{3.1-11}$$

(b) Otherwise,

$$g_i = \frac{\partial f}{\partial x_i} = \frac{f(x_{i+1,i}) - f(x_{si})}{x_{i+1,i} - x_{si}} \tag{3.1-12}$$

(c) End Iteration

Step 5. Calculate the reflection point:

$$x_r = x_1 - \rho * g, \text{ where } \rho \text{ is a reflection coefficient.} \tag{3.1-13}$$

Step 6. Expansion:

a. If $f_r < f_n$, calculate the expansion point:

$$x_e = (1 - \chi) * x_1 + \chi * x_r \tag{3.1-14}$$

i. Evaluate $f(x_e)$.

ii. If $f_e < f_r$, accept x_e and terminate the iteration.

iii. Otherwise, accept x_r and terminate the iteration.

End Iteration.

One may notice that this version of the NM algorithm does not require a contraction or shrink step. This is because the quasi-gradient is calculated, so the so called reflected point, x_r, is going to be in the correct direction. This algorithm does have an increased cost in each iteration due to the calculation of the quasi-gradient, but the algorithm saves in the number of iterations needed to converge [65].

3.2 Nelder-Mead Enhanced Extreme Learning Machine

The ELM family of algorithms boast very good training times and acceptable generalization performance, but the network size is always very large. This is likely due to the fact that out of three possible tunable parameters, the ELMs elect to only optimize one parameter. Let us think in terms of a network of RBF neurons. Each neuron has 3 tunable parameters, the center, the width, and the height. The height of each neuron is optimized by the ELM algorithms, but the center, and width are not. However, the ELM algorithms omit these parameters in the interest of training time. The goal of the algorithm presented here is to provide similarly fast training times and errors, but to also provide a more compact network with better generalization properties. This is done by adjusting the I-ELM algorithm so that it now optimizes the radii and the weights of the RBF neurons while choosing the centers in a greedy fashion. These design choices will help us to construct as compact a network as possible. This work has been very recently published in the Journal of Neurocomputing [67].

Keeping this in mind, the Nelder-Mead algorithm was chosen for the task of optimizing the radius of each newly added node for the following reasons: the Nelder-Mead algorithm tends to produce significant improvement over the first few iterations, the Nelder-Mead algorithm does not require many calculations of derivatives only a few function values at each iteration, and finally, it is easy to understand and explain [68]. All of these properties allow the algorithm to be used to very quickly change the radius of each node so that the error is improved. Furthermore, the improved version of the Nelder-Mead algorithm published by Pham in [65] was used in the final version of this algorithm. In the initial iteration of the algorithm proposed here, the original nelder-mead simplex algorithm as described in [63] was used, but better performance was achieved using the improved version.

NME-ELM Algorithm: Given a training set as described in equation (2.4-16), an activation function, $h(x)$, a maximum number of hidden neurons $\widetilde{N}$, a maximum number of Nelder-Mead iterations, k, and a required learning accuracy, ε.

Step 1. Initialize**:** Let the number of nodes, $n = 0$ and the error, $E = t$.

Step 2. Learning: While $n < \widetilde{N}$ and $\|E\| < \varepsilon$

(a) Increment n.

(b) Find the index,

$$i^* = \{i \mid \max_{1 \le i \le k} \|e_i\|\}. \quad (3.2\text{-}1)$$

(c) Assign the center c_n, of the new node to be the input pattern x_{i^*} .

(d) Assume the initial value of β_n to be equal to e_{i^*}.

(e) Initialize the Nelder-Mead Simplex [63] algorithm:

i. Set the Simplex parameters according to the parameters defined Chapter 2:

$$\rho = 1, \ \chi = 2, \ \gamma = \frac{1}{2}, \ and\, \alpha = \frac{1}{2} \tag{3.2-2}$$

ii. Choose some initial values for σ_n.

iii. Calculate:

$$SSE(\sigma_{n,j}) = \sum_{i=1}^{N} (e_i - \beta_n * g_n(x_i))^2 \tag{3.2-3}$$

For each $\sigma_{n,j}$.

The vector of σ values becomes the vector of vertices for a simplex. The vector of SSE values becomes the vector of function values for the simplex.

iv. Perform k iterations of the Simplex algorithm (k is typically 5-10). This step results in an optimal σ_n value.

(f) Re-calculate β_n using equation (2.4-18).

(g) Calculate the residual error as in equation (2.4-19).

End While Loop.

A flow chart of this algorithm is shown in Figure 3.2-1.

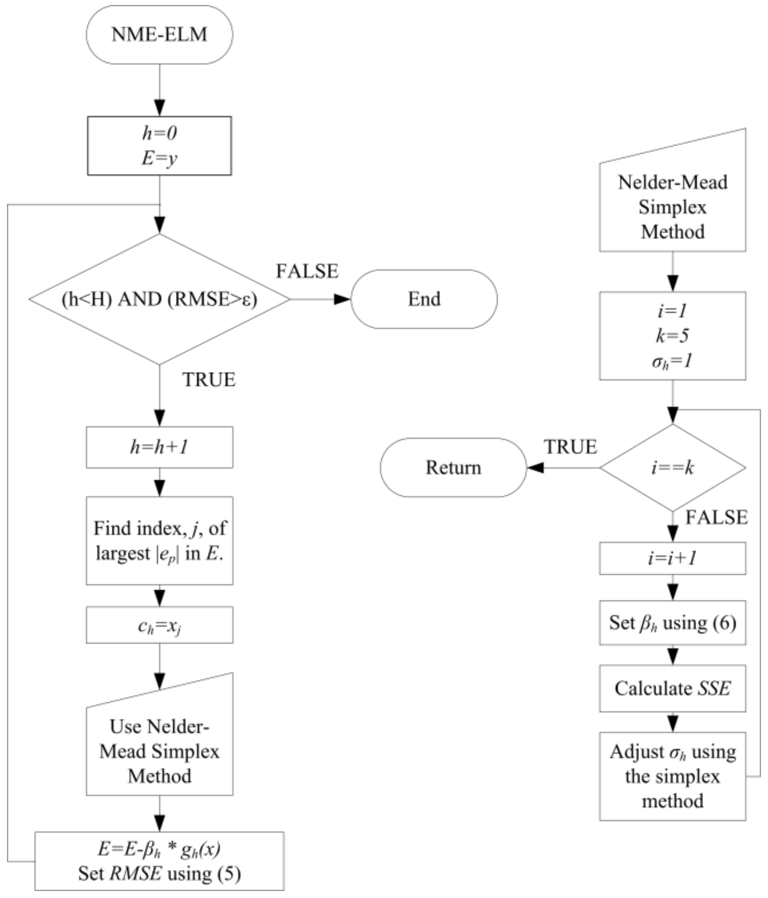
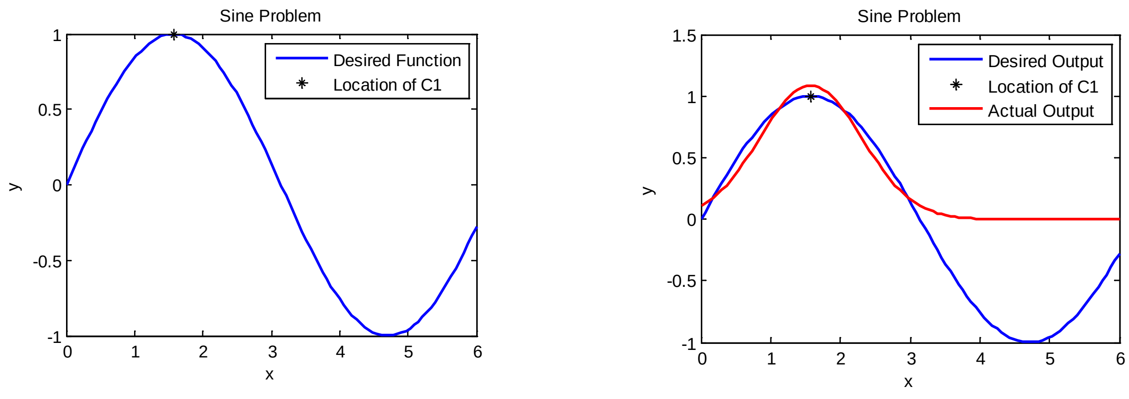


Figure 3.2-1 The NME2-ELM algorithm given a training set $\{(x_p, y_p) | x_p \in R^D, y_p \in R, p = [1 \ldots P]\}$, an activation function *g(x)*, a maximum node number H, and an expected learning accuracy ε.

In order to illustrate the intuitive process of this algorithm, the step by step network construction process will be demonstrated on a simple problem. The network will be constructed to approximate one period of a simple sine wave. The desired function is shown in Figure 3.2-2a.

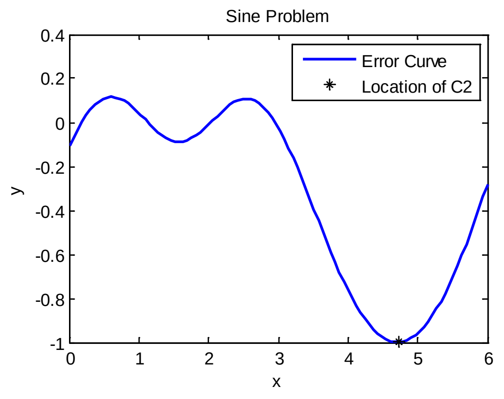


(a) (b)

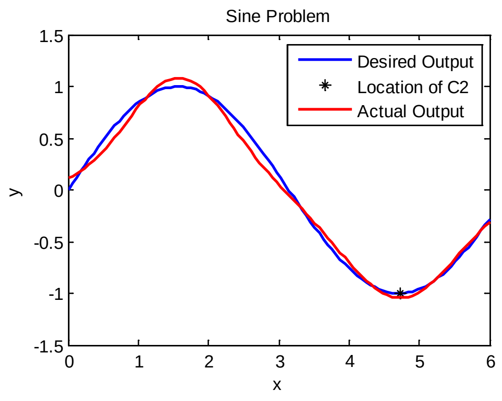
Figure 3.2-2 An illustration of the first few iterations of the NME2-ELM algorithm. (a) Desired sinusoidal function with the first center c_1 shown by the black asterisk. (b) The output of one node using NME2-ELM.

From the inputs and desired outputs, a new neuron is created and its center is chosen. For this problem, the center $c_1 = 1.5678$, and is shown in Figure 3.2-2 as the black asterisk. Then the radius σ_1 and the height β_1 are optimized using the Nelder-Mead algorithm described previously. This yields $\sigma_1 = 1.1429$ and $\beta_1 = 1.0714$. Figure 6b shows the results of this process.

The residual error is calculated using equation (2.4-19) and is used in the next iteration as a second neuron is added. Figure 3.2-3a depicts the error surface (new desired curve) and the second added center c_2.



(a)

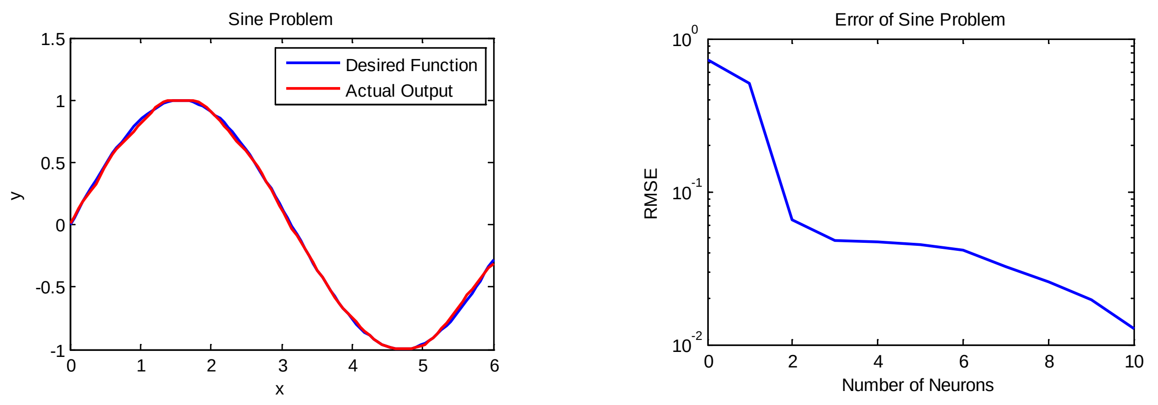


(b)

Figure 3.2-3 Step by step illustration for the second neuron. (a) The desired curve for the second node and the center c_2. (b) Desired surface and NME2-ELM output after 2 RBF units have been added to the network. RMSE = 0.0627.

Continuing the algorithm to add a second neuron to the network, the resulting parameters are: $c_2 = 4.7035$, $\sigma_2 = 0.6797$, and $\beta_2 = 1.0260$. Figure 3.2-3b shows that after two RBF neurons are added to the network, the NME2-ELM algorithm reaches a *RMSE* value of 0.0627. The RMSE can be further reduced by adding more neurons to the network. If the algorithm is allowed to continue to a total of five RBF units, the *RMSE* value is as low as 0.0162. The results after 10 neurons are added to the network are shown in Figure 3.2-4.

Intuitively, it can be seen that this algorithm makes choices that allow a network to reduce error very significantly with the addition of each neuron. This algorithm was tested on several bench mark problems including some real world problems from the UCI Machine Learning Repository [69]. The next section presents the results from these tests and comparisons with the ELM algorithms and SVR.



(a) (b)

Figure 3.2-4 Summary of results after ten neurons are used. (a) The sine problem after 10 RBF neurons are added to the network. The RMSE = 0.0162. (b) The RMSE with respect to number of neurons in the network.

3.3 Testing and Comparisons

In this section, the performance of the proposed NME-ELM algorithm is evaluated and compared with other popular algorithms used to construct RBF networks. Algorithms such as the various incremental ELM algorithms and the popular SVR algorithm are compared to the proposed algorithm on several problems. The testing environment used is MATLAB with the exception of SVR that was tested using the LIBSVM package [70]. The hardware consists of an Intel i7-2600 CPU @ 3.4GHz with 8 GB of RAM on a 64 bit operating system.

The results seem to indicate that the NME-ELM algorithm performs as it is expected to perform. The errors converge with very few neurons required, and generalization performance is good. Another important thing to note is that there is no randomness in the NME-ELM approach. This means that fewer trials are needed to reach an acceptable solution. In the case of the other ELMs, a poor random selection of the input parameters can lead to the network never converging to acceptable error levels. This requires that many trials be run and then the top few networks selected for further analysis.

3.3.1 Highly Nonlinear Benchmark: Peaks Problem

In this section the NME-ELM algorithm is tested and compared on a highly nonlinear problem called the peaks problem. This problem is designed to be used as a benchmark in testing the robustness of the learning algorithm. The problem has 2-dimensional inputs from the range [-1,1] and an output in the range [-1,1] can be described by the following equation:

$$\begin{aligned} z(x,y) = &-\frac{1}{30}\exp(-1-6x-9x^2-9y^2) \\ &-(0.6x-27x^3-243y^5)\exp(-9x^2-9y^2) \\ &+(0.3-1.8x+2.7x^2)\exp(-1-6y-9x^2-9y^2) \end{aligned} \tag{3.3-1}$$

In order to make this problem more “real” 500 training vectors with random coordinates (x,y) were chosen in the range [-1,1]. Then the desired outputs were calculated using equation (3.3-1). Then a Gaussian noise distribution with a standard deviation of 0.1 was added to the target outputs to simulate imperfect data. For testing, an evenly spaced grid of x and y coordinates consisting of 900 patterns was used. The parameters for each algorithm were optimized with respect to testing RMSE using a grid search run 20 times (to help eliminate the randomness that comes with ELM). For this particular problem the impact factor for the ELM algorithms was set to 2.7583, and centers were chosen randomly from the input space. The radius of the RBF kernel for SVR was set to 0.5 and the C parameter was varied between 2^0 and 2^{10} (this is how we have multiple points to plot for the SVR errors).

The desired testing surface of the peaks problem is shown in Figure 3.3-1. Figure 3.3-2a shows the training errors of different algorithms with respect to the number of neurons (or support vectors) required. Figure 8b shows the corresponding testing errors for each algorithm with respect to the number of neurons required. Each algorithm was run 20 times and the average errors are presented. From these results, it can be seen that the proposed algorithm converges to lower error with fewer neurons than the other algorithms and far outperforms the other incrementally constructive algorithms of the ELM family. It should also be noticed that once the algorithm reaches a certain training error (which is far below the level of the other algorithms), the testing error does not improve. This is believed to be caused by overtraining or training to noise rather than trends in the data.

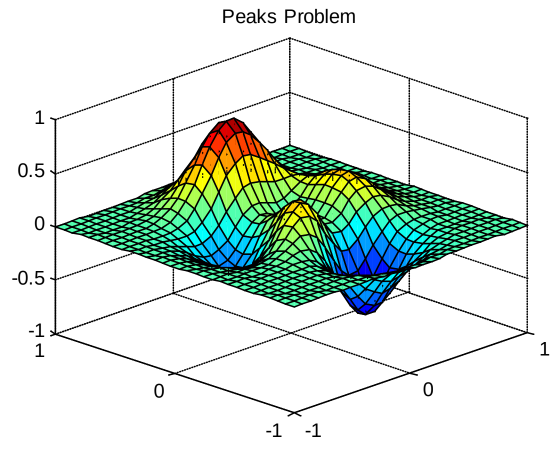
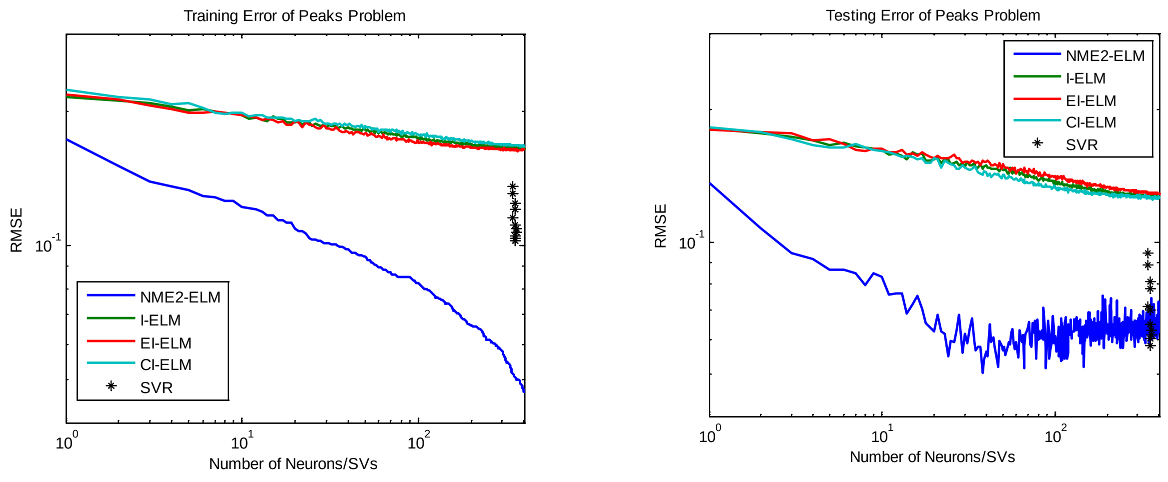


Figure 3.3-1 Desired surface for the peaks problem.



(a) (b)

Figure 3.3-2 Testing and training errors of various algorithms on the peaks problem. (a) Training results on the peaks problem with different algorithms. It can be seen that the NME2-ELM algorithm takes a very quick path to lower error. (b) Testing results on the peaks problem with different algorithms. It can be seen that for NME2-ELM the network starts training to noise in the data after ~40 neurons are added.

3.3.2 Real World Data

The proposed algorithm's robustness is further demonstrated by testing it on high dimensional real world data. The datasets used are taken from the UCI Repository of Machine Learning Databases [69]. The testing errors obtained for the data sets and the generated network sizes are then compared to other popular RBF algorithms such as the ELMs and SVR.

All of the inputs used in for the real world data sets were normalized to the range [-1, 1], and the outputs were normalized to the range [0, 1]. Each of the data sets were randomly divided into training data and testing data. In most cases, this was roughly %50 of the patterns for both sets. For each data set, the training and testing sets were randomly generated and run 20 times. The results presented are the averages of these trials. Table 4 shows the data sets used and the way they were split apart for testing and training as well as the number of input dimensions. Notice that for the ELM family of algorithms a somewhat arbitrary choice must be made to select the number of hidden units, because the errors decrease slowly as the network size increases (Figure 3.3-3b-d). However, it can be seen that in the case of NME2-ELM, an error saturation is reached very early.

Table 4 Dataset Information

Problem	#Training Data	#Testing Data	# Attributes
Abalone	2000	2177	8
Auto Price	80	79	15
Boston Housing	250	256	13
Cal Housing	8000	12640	8
Delta Ailerons	3000	4129	5
Delta Elevators	4000	5517	6
Machine CPU	100	109	6

Figure 3.3-3a shows the testing results for the NME-ELM algorithm on several real world datasets. It can be seen that the testing *RMSE* converges within 50 RBF units for each problem. This *RMSE* is comparable to the *RMSE* obtained by the ELM algorithms, but in this case only 50 RBF nodes are used. Furthermore, Figure 3.3-3b-d shows a comparison between the ELM algorithms, the proposed algorithm, and SVR. It can be seen that the proposed algorithm converges to its minimum testing error very quickly, especially when compared to the other Incremental ELMs.

The data presented in Table 5 shows that the NME-ELM algorithm performs very well in terms of testing *RMSE*. Again, notice that the NME-ELM algorithm's testing error is comparable or better than the other algorithms despite using far fewer neurons. Table 6 Table 7 give a comparison of training times and network size respectively. It can be seen that the NME2-ELM performs well as it constructs a smaller network than the Incremental ELMs, and is often able to train faster as well.

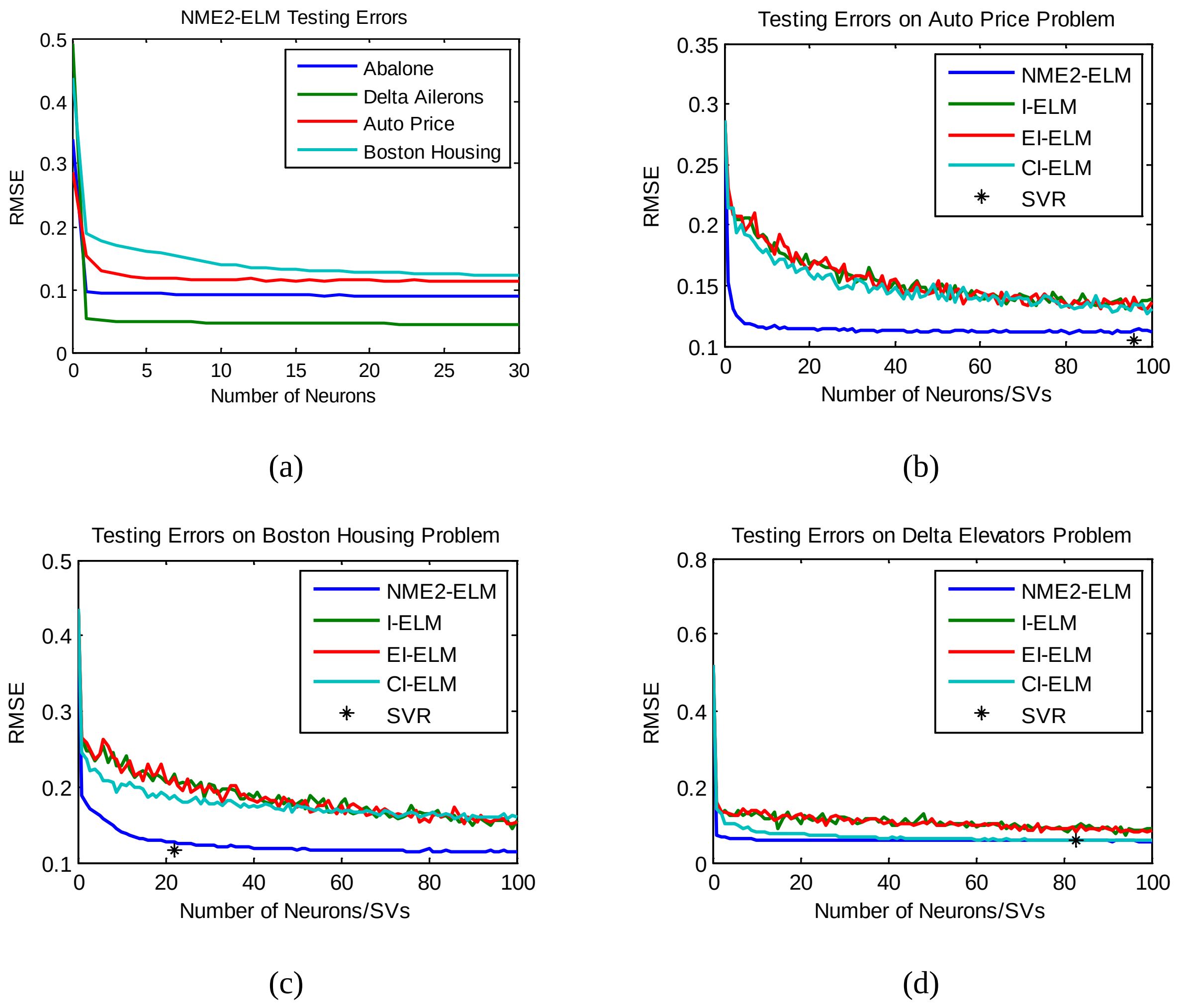


Figure 3.3-3 (a) The NME2-ELM on various Real World Problems. (b) A comparison between various algorithms on the Auto Price problem. (c) A comparison between various algorithms on the Boston Housing problem. (d) A comparison between various algorithms and the Delta Elevators problem.

Table 5 Average Testing Errors For Real World Problems

Problems	NME2-ELM	I-ELM	EI-ELM	CI-ELM	SVR	
	RMSE	RMSE	RMSE	RMSE	(C,γ)	RMSE
Abalone	0.0849	0.0938	0.0829	0.0858	$(2^4, 2^{-6})$	0.0846
Auto Price	0.1104	0.1222	0.1139	0.1197	$(2^8, 2^{-5})$	0.1052
Boston Housing	0.1124	0.1261	0.1281	0.1423	$(2^4, 2^{-3})$	0.1155
Cal Housing	0.1642	0.1691	0.1503	0.1756	$(2^3, 2^1)$	0.1311
Delta Ailerons	0.0413	0.0513	0.0448	0.0416	$(2^3, 2^{-3})$	0.0467
Delta Elevators	0.0557	0.0632	0.0575	0.0566	$(2^0, 2^{-2})$	0.0603
Machine CPU	0.0791	0.0674	0.0554	0.0675	$(2^6, 2^{-4})$	0.0620

Table 6 Training Times For Real World Problems

Problems	NME2-ELM	I-ELM	EI-ELM	CI-ELM	SVR	
	Time (s)	Time (s)	Time (s)	Time (s)	(C,γ)	Time (s)
Abalone	0.0944	0.5990	5.9603	0.6098	$(2^4, 2^{-6})$	0.2659
Auto Price	0.0064	0.0203	0.1777	0.0204	$(2^8, 2^{-5})$	0.0294
Boston Housing	0.0243	0.0822	0.6910	0.2237	$(2^4, 2^{-3})$	0.0461
Cal Housing	0.3746	1.8631	14.554	1.4608	$(2^3, 2^1)$	6.0684
Delta Ailerons	0.1587	0.5511	5.3020	0.5453	$(2^3, 2^{-3})$	0.2499
Delta Elevators	0.2399	0.8651	8.5861	0.8789	$(2^0, 2^{-2})$	1.0234
Machine CPU	0.0099	0.2549	0.3078	0.0353	$(2^6, 2^{-4})$	0.0265

Table 7 Network Size For Real World Problems

Problems	NME2-ELM	I-ELM	EI-ELM	CI-ELM	SVR	
	# Neurons	# Neurons	# Neurons	# Neurons	(C,γ)	# Neurons
Abalone	100	200	200	200	$(2^4, 2^{-6})$	310
Auto Price	82	200	200	200	$(2^8, 2^{-5})$	96
Boston Housing	94	200	200	200	$(2^4, 2^{-3})$	22
Cal Housing	195	200	200	200	$(2^3, 2^1)$	47
Delta Ailerons	182	200	200	200	$(2^3, 2^{-3})$	2189
Delta Elevators	140	200	200	200	$(2^0, 2^{-2})$	83
Machine CPU	28	200	200	200	$(2^6, 2^{-4})$	261

Chapter 4 Error Correction Algorithm

In this work it has been posited that being able to adjust more parameters leads to more compact learning networks. In the case of classical neural networks, this is the way they have always been constructed and trained. However, RBF neural networks are often constructed in a different manner. Typically this is done by selecting the centers using a clustering algorithm such as kohonen training. Then RBF neurons are created with fixed widths and the heights are adjusted using any number of popular training techniques. The ability to move RBF network centers to minimize the errors should allow for very compact networks with good generalization abilities.

4.1 Levenberg-Marquardt Training for RBF Networks

It was proposed in [71] that a second order method be used to train RBF networks. The method proposed is an adapted version of the Neuron by Neuron algorithm published in [72]. Furthermore, the Levenberg-Marquardt method was improved for computations with large data sets in [43]. This allows for the Jacobian matrix needed for the second order approximation to be efficiently stored for large networks and large data sets. A training algorithm using this LM method was proposed in [71] that shows very good results; however, there are some deficiencies that are addressed in the following section.

4.1.1 ISO Deficiencies

The Improved Second-Order (ISO) algorithm for training radial basis function networks was introduced in [71]. This algorithm uses the improved version of the Levenberg-Marquardt algorithm to optimally adjust the parameters of a given RBF network. This algorithm requires first that you have a network of RBF units with some initial parameters and an initial error state. Then it uses the LM method as described earlier to adjust the parameters to find a minimum in the error surface. The requirement for an initial state of a network is a dilemma however, because one must still consider the non-trivial problem of choosing an appropriately sized network for the problem. This coupled with the fact that the neurons are often initialized with random parameters leads to the need for several trials before an optimum solution is found. The ErrCor algorithm presented in the following section attempts to address these issues. In a single trial, a network is both constructed and trained to a minimum error with zero random parameters and no initial network. Figure 4.1-1 depicts the difference in error convergence over many trials with ISO and a single trial of the new ErrCor algorithm.

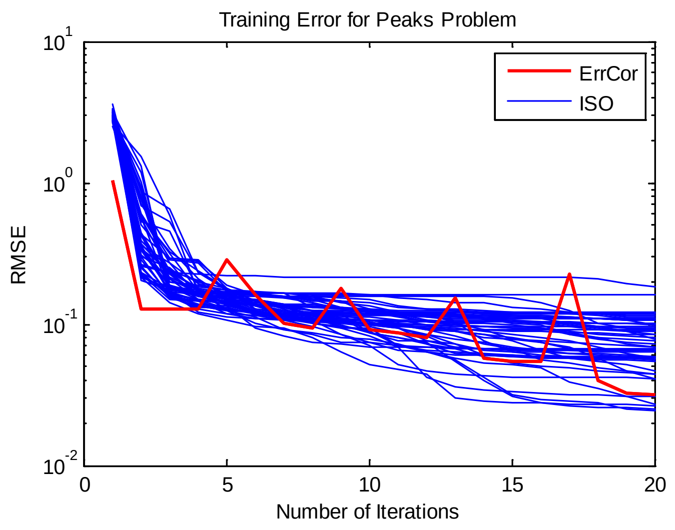


Figure 4.1-1 The ISO algorithm and the ErrCor algorithm on the Peaks problem using 5 RBF units. Notice that the ISO algorithm errors vary greatly due to the random start points, while ErrCor reaches small error with a single try.

4.2 Error Correction Algorithm

The Error Correction (ErrCor) algorithm described here is a greedy incremental network construction algorithm [73]. This means that the algorithm starts from scratch and places each RBF neuron into the network based on a heuristic measure. Basically the ErrCor algorithm attempts to reduce the error as much as possible during each training step by adding an RBF neuron located at the place with the highest peak in the error surface. Then the RBF parameters are further optimized using the LM method described earlier. This algorithm has been shown to have good training and generalization characteristics on benchmark problems and real world datasets. A detailed description of the algorithm is given in this section, and experimental results and comparisons are given in section 4.3.

Error Correction Algorithm: Recall that a typical training set is of the form:

$$\aleph = \{(x_i, t_i) | x_i \in \mathbb{R}^n, t_i \in \mathbb{R}^m, i = 1, \dots, N\} \qquad (4.2\text{-}1)$$

Given the set described above, activation function, $h(x)$, a maximum number of hidden neurons $\widetilde{N}$, and a required learning accuracy, ε.

Step 1. **Initialization:**

(a) Declare network output, $o = 0$.

(b) Set LM training parameters the maximum iterations, $\mathrm{M_{iter}}$, the combination coefficient, μ, and the minimum error difference, e_{diff}.

Step 2. **Learning:**

(a) While $n < \widetilde{N}$ and $E > \varepsilon$

i. Increment n.

ii. calculate errors:

$$\boldsymbol{E} = |\boldsymbol{T} - \boldsymbol{o}| = [e_1, e_2, \dots e_N] \qquad (4.2\text{-}2)$$

iii. Find the index according to:

$$i^* = \{i |\ \max_{1 \le i \le k} \|e_i\|\} \qquad (4.2\text{-}3)$$

iv. Create a new RBF unit with center equal to x_{i^*}.

v. Set output weight and width of new RBF unit to 1.

vi. if $n > 1$,Initialize existing network to the training results of step $n - 1$.

vii. Evaluate RMSE using equation (2.4-17):

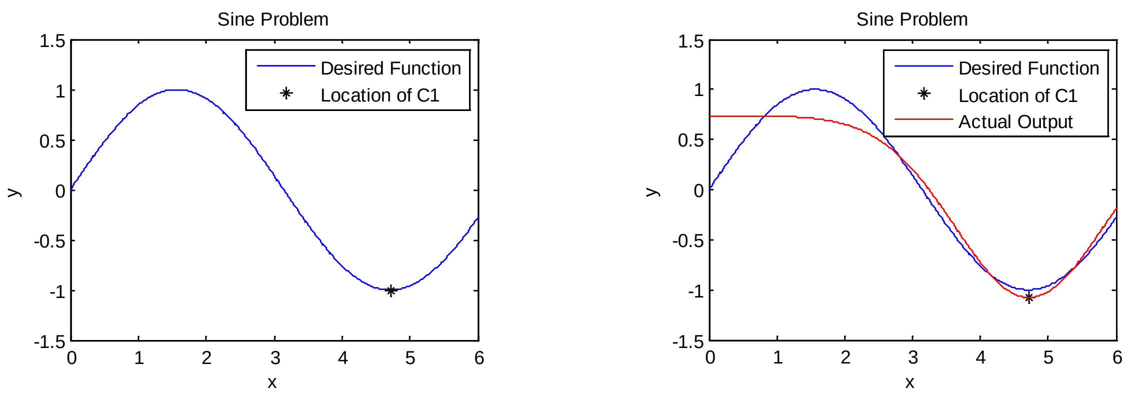
$$RMSE_1 = \sqrt{\sum_{i=1}^{N} e_i^2} \qquad (4.2\text{-}4)$$

viii. Set the number of LM iterations, $k = 1$.

ix. While $k < M_{iter} - 1$ and $e_{diff} < (RMSE_k - RMSE_{k+1})$

```
        1.  Calculate quasi-Hessian matrix, $\boldsymbol{Q_k}$ and gradient vector, $\boldsymbol{g_k}$.
        2.  Update network parameters using the ISO update rule:
                $\Delta_{k+1} = \Delta_k - (\boldsymbol{Q}_k + \mu_k I)^{-1}\boldsymbol{g_k}$          (4.2-5)
        3.  Compute output of network $\boldsymbol{o}$;
        4.  Evaluate, $RMSE_{k+1}$
        5.  Increment $k$.
        End While.
    x.  Calculate $\boldsymbol{E}$.
End While.
```

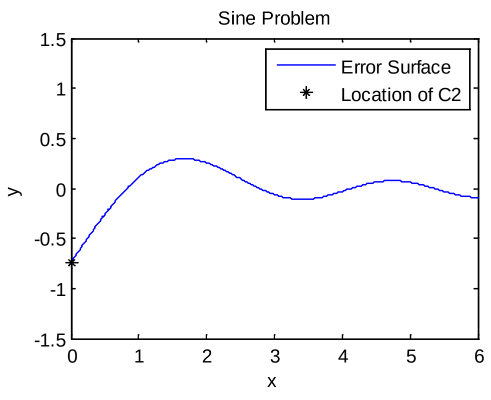
In order to illustrate the intuitive process of this algorithm, the step by step network construction process will be demonstrated on a simple problem. The network will be constructed to approximate one period of a simple sine wave. The desired function is shown in Figure 4.2-1a.



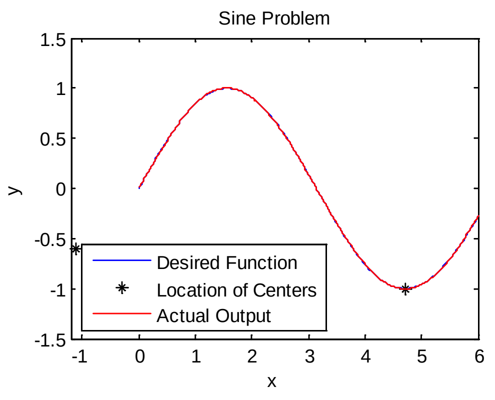
(a) (b)

Figure 4.2-1 An illustration of the first iteration of the Error Correction algorithm. (a) Desired sinusoidal function with the first center c_1 shown by the black asterisk. (b) The output of the network created using ErrCor after one iteration.

By going through the data of the curve in Figure 4.2-1a, the center location $c_1 = 4.7$ and the error height $\beta_1 = -1$ corresponding to the highest magnitude point in the error surface are chosen as the initial parameters of the first neuron. Then the neuron is trained by applying the LM algorithm (section II.B) for parameter adjustment. The resulting network (a single neuron) output is shown in Figure 4.2-1b. The RBF parameters after the LM training are: $c_1 = 4.722$, $\beta_1 = -1.076$, $\sigma_1 = 1.560$. Based on the training results, the outputs of the RBF network are visualized in Figure 4.2-1b, and new error curve (Figure 4.2-2a) is obtained as the difference between Figure 4.2-1a and Figure 4.2-1b. Comparing the error curves in Figure 4.2-1a and Fig. 14a, one may notice that, the lowest valley (marked as an asterisk) in Figure 4.2-1a has been eliminated in Figure 4.2-2a. This results in an RMSE of 0.2000. This residual error is then used to find the initial location of the next RBF neuron (see Figure 4.2-2).



(a)



(b)

Figure 4.2-2 Illustration of the ErrCor algorithm during the second iteration. (a) The desired curve for the second node and the center c_2. (b) Desired surface and network output after 2 RBF units have been added to the network. RMSE = 0.0025.

Continuing the algorithm to add a second neuron to the network, the resulting parameters are: $c_2 = -1.1$, $\beta_2 = -0.5958$, and $\sigma_2 = 2.079$. Figure 14b shows that after two RBF neurons are added to the network, the ErrCor algorithm reaches a *RMSE* value of 0.025. The RMSE can be further reduced by adding more neurons to the network. If the algorithm is allowed to continue to a total of five RBF units, the *RMSE* value is as low as 0.000025. The results after 5 neurons are added to the network are shown in Figure 4.2-3.

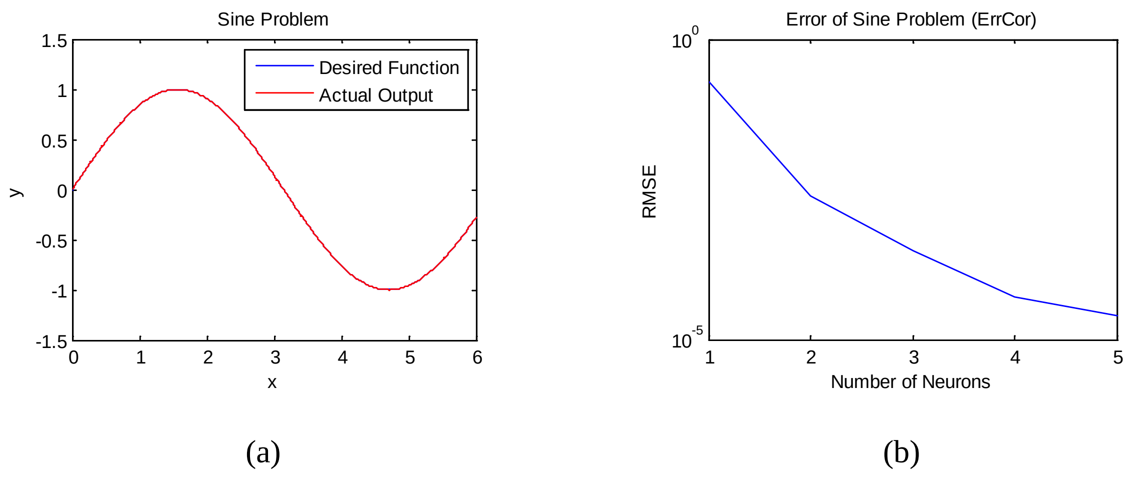
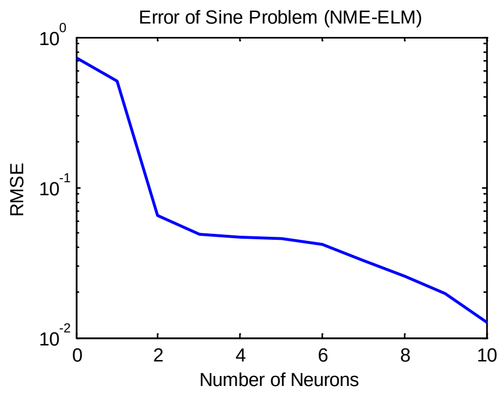
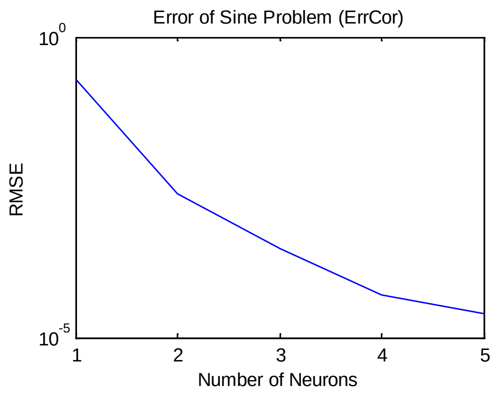


Figure 4.2-3 Summary of results of the ErrCor algorithm after five neurons are used. (a) The sine problem after 5 RBF neurons are added to the network. The RMSE is effectively 0. (b) The RMSE with respect to number of neurons in the network.

It is worth comparing the error convergence of this algorithm and the previously discussed Nelder-Mead Enhanced Extreme Learning machine. Intuitively, it can be seen that this algorithm makes choices that allow a network to reduce error very significantly with the addition of each neuron even more so that the NME-ELM. This improvement is due to the fact that the centers are not fixed once they are guessed to be the highest peak in the error surface. Figure 4.2-4 shows the error rate for each added neuron for the two algorithms. Notice that ErrCor reaches a lower error in fewer neurons. This algorithm was tested on several bench mark problems including some real world problems from the UCI Machine Learning Repository [69]. The next section presents the results from these tests and comparisons with the ELM algorithms and SVR.





(a) (b)

Figure 4.2-4 Summary of results of the NME-ELM algorithm and the ErrCor algorithm after several neurons are used. (a) The RMSE of the NME-ELM as it adds up to 10 neurons to the network. (b) The RMSE of the ErrCor algorithm as it adds up to 5 neurons to the network.

4.3 Testing and Comparisons

In this section, the performance of the proposed ErrCor algorithm is evaluated and compared with other popular algorithms used to construct RBF networks. These algorithms include the popular GGAP, GGAP-GMM, RAN, MRAN, and RANEKF algorithms. These algorithms are primarily designed as online algorithms while the algorithms proposed here are offline algorithms. However, the aforementioned algorithms also work well as offline algorithms; therefore, they are also compared with the proposed ErrCor algorithm. It is important to note that the training time is a very important aspect of online algorithms because they are constantly being retrained as new information is seen. However, offline algorithms such as the various incremental ELM algorithms, the popular SVR algorithm, the new NME-ELM algorithm, and the proposed ErrCor algorithm focus on execution times and generalization abilities. This keeps the compactness of the networks and the testing RMSE at the forefront of the evaluation process.

The testing environment used is MATLAB with the exception of SVR that was tested using the LIBSVM package [70]. The hardware consists of an Intel i7-2600 CPU @ 3.4GHz with 8 GB of RAM on a 64 bit operating system. The results seem to indicate that the ErrCor algorithm performs as it is expected to perform. The errors converge with very few neurons required, and generalization performance is superior to all of the algorithms against which it is compared. Another important thing to note is that there is no randomness in the ErrCor approach, not even in the initialization process. This means that only a single trial is needed to reach an acceptable solution. In the case of the other ELMs, a poor random selection of the input parameters can lead to the network never converging to acceptable error levels. This requires that many trials be run and then the top few networks selected for further analysis.

4.3.1 Highly Nonlinear Benchmarks

In the presented study, it was found that many of the real-world data sets are not highly nonlinear and good results can be obtained with very few RBF neurons (see Table 11). Therefore, in this section, the ErrCor algorithm is applied so some well-known nonlinear bench tests to demonstrate in an easily visible manner the power and robustness of the algorithm. These benchmark tests are organized as follow, Rapidly Changing Function, Peaks Problem, and Two Spiral Problem.

4.3.1.1 Rapidly Changing Function

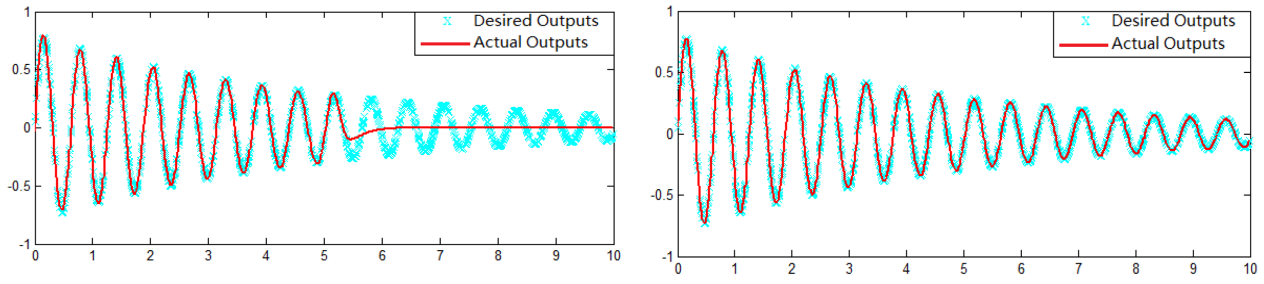
In this experiment, the proposed algorithm is applied to design RBF networks to approximate the following rapidly changing function this is the same function used to test many popular algorithms such as the GGAP-RBF algorithm shown in [16].

The formula for this benchmark problem is the following:

$$y(x) = 0.8e^{-0.2x}\sin(10x)$$

In this problem, there are 3000 training patterns with x-coordinates uniformly distributed in range [0, 10]. The validation data set consists of 1500 patterns with x-coordinates randomly generated in the same range [0, 10].

Figure 4.3-1 shows the testing results of the proposed ErrCor algorithm, with the number of RBF units equal to 10 and 20 respectively. Figure 4.3-2 shows the training results of proposed ErrCor algorithm and several other algorithms. One may notice that the proposed ErrCor algorithm can reach a similar training/testing error level with a 3 to 30 times smaller network.



(a) (b)

Figure 4.3-1 Testing results of the ErrCor algorithm on the rapidly changing function. (a) The results after 10 neurons are added to the network. The training and testing mean square errors are 7.846×10^{-3} and 7.516×10^{-3} respectively. (b) The results after 20 neurons are added to the network. The training and testing mean square errors are 5.428×10^{-6} and 5.347×10^{-6} respectively.

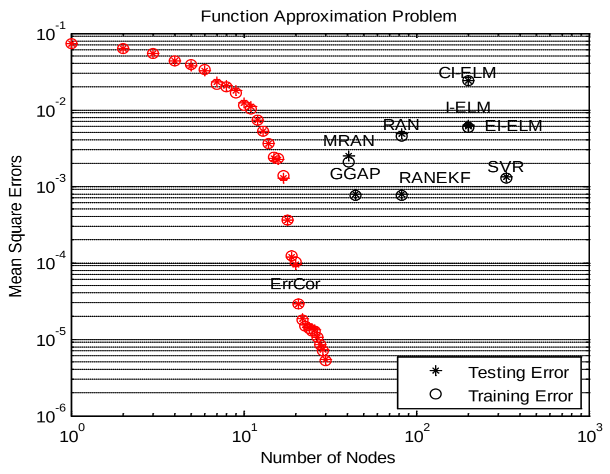


Figure 4.3-2 Function approximation problem: training/testing average sum square errors vs. average number of RBF units.

Table 8 presents the comparison of average training time, training errors, testing time, and testing error for each algorithm. For the proposed ErrCor algorithm, the computation time is counted until the RBF network with 20 units (with smaller training/testing errors than other algorithms) gets trained. For the ELM algorithms, the centers were generated from the input range [0,10] while impact factors were from the range (0,0.5]. For GAP-RBF the parameters are fixed at $\epsilon_{max} = 1.15, \epsilon_{min} = 0.04, \kappa = 0.10, and\ \gamma = 0.999$. For the MRAN algorithm, the threshold for growing and pruning was set as $e_{min}^{MR} = 0.06$, and the appropriate size of the sliding window was chosen as $M = 100$. The parameters for GGAP were $e_{min}^{GG} = 0.00001$. For SVR, the parameter C was tuned to 1000 while γ was set at 1.

In order to provide a measure independent of physical CPU power, a normalized computation time was used to determine the efficiency of the constructed networks. The normalization was done by first testing two different data sets on networks of different sizes twenty times each. The average computation time per RBF unit per testing input was 1.195μs.

Table 8 Comparison of training times/errors and validation times per pattern/errors for the rapidly changing function problem

Algorithm	Train Time (s)	Train RMSE	Test Time (μs)	Test RMSE
GGAP	24.808	0.0265	54.16	0.0265
MRAN	78.572	0.0458	52.15	0.0490
RANEKF	105.72	0.0265	106.8	0.0265
RAN	45.514	0.0671	112.2	0.0686
SVR	0.2552	0.0346	2496	0.0361
I-ELM	0.5509	0.0831	239.0	0.0843
CI-ELM	0.5597	0.1356	239.0	0.1378
EI-ELM	5.3991	0.0728	239.0	0.0755
NME-ELM	0.1725	0.0238	119.5	0.0303
ErrCor	48.530	0.0141	23.90	0.0141

4.3.1.2 Peaks Problem

The peaks problem is a problem with a two dimensional input that yields an output with many peaks and valleys; the peaks problem provides a way to easily visualize the training process of the various algorithms. In this experiment, the peaks problem consists of 2000 randomly generated patterns in the range (-1,+1) for both *x* and *y* directions using the formula described in chapter 3 (3.3-1).

Once again the described function is shown in Figure 4.3-3. Another 1000 randomly generated patterns were used for the validation.

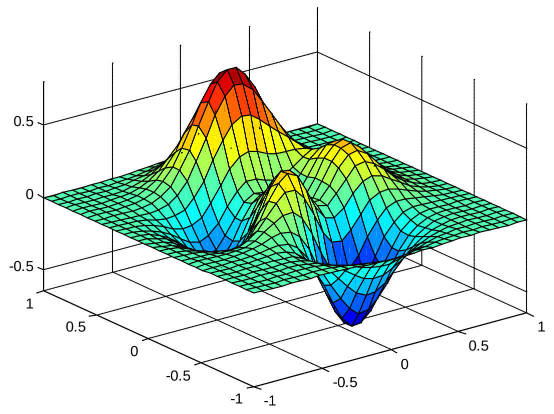


Figure 4.3-3 The desired output for the peaks problem.

As can be seen in Figure 4.3-4a, the major peaks and valleys of the desired output are targeted by the ErrCor algorithm with only five RBF units. This compact network achieves a validation RMSE of 0.031. As training continues, the error decreases steadily as units are added until the RMSE reaches about 0.0003 with 20 units. As was expected, after five RBF units were added to the network, the centers of the RBF units in the trained network are located approximately in the centers of the highest peaks and valleys. What is interesting however, is that after twenty RBF units were added, the centers had moved to completely different locations.

In comparison to the other algorithms, ErrCor was able to reach a much smaller RMSE with much fewer RBF units. This demonstrates that the ErrCor algorithm is very efficient when choosing heights, widths, and centers of the RBF units. The ELM family of algorithms was tested on this problem and was able to achieve an RMSE of about 0.03 with one thousand RBF units (See Figure 4.3-4d, Figure 4.3-5, and Figure 4.3-6). This error is still 100 times larger than the error obtained with only 20 RBF units using the ErrCor algorithm (RMSE = 0.0003). The SVR algorithm used thirty-six support vectors to achieve an RMSE of 0.031 (See Figure 4.3-7). Still, this requires about seven times more units than ErrCor for the same error.

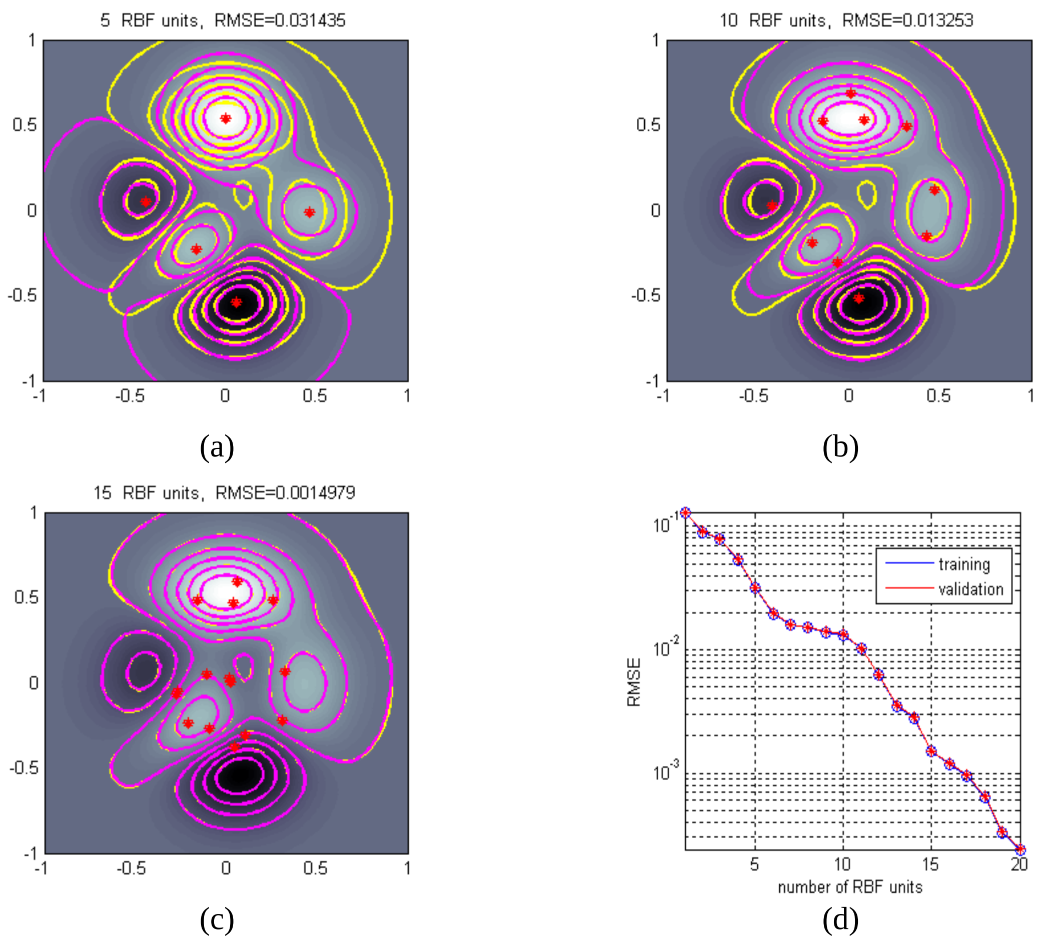


Figure 4.3-4 ErrCor output for the peaks problem. The yellow contour depicts the desired surface, the purple contour depicts the network output, and the red asterisks show where the centers of the RBF units are located.

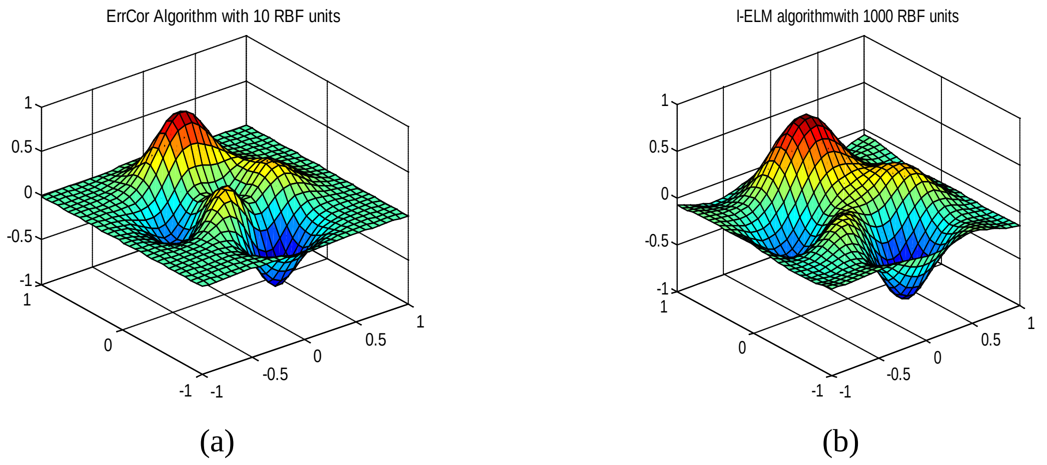


Figure 4.3-5 ErrCorr output using 10 nodes, (a) compared to ELM output using 1000 nodes, (b).

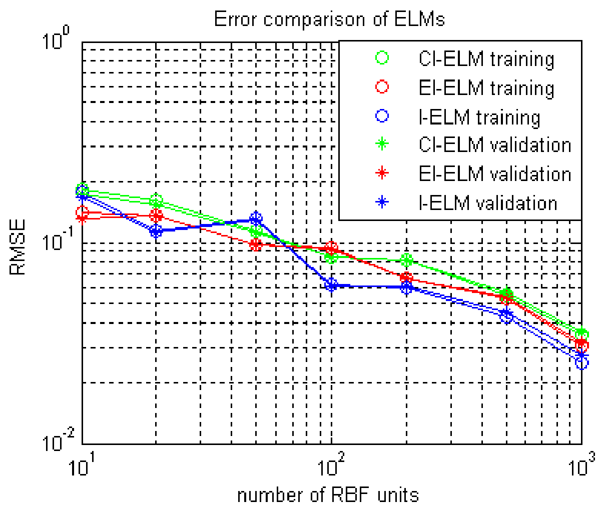


Figure 4.3-6 Comparison of the three ELM algorithms on the peaks problem. All three attain similar errors. The random centers for the ELM algorithm were generated in the range of inputs [-1,1] while the impact factors were in the range (0,0.5].

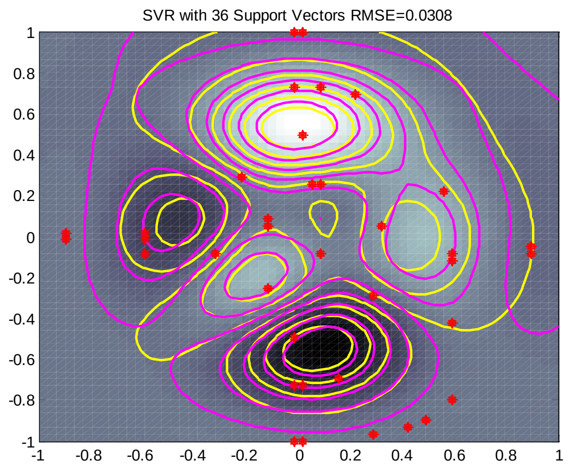


Figure 4.3-7 SVR output for the peaks problem. The yellow contour depicts the desired surface, the purple contour depicts the algorithm output, and the red asterisks show where the support vectors are located. The SVR parameters used were: G=0.3, Epsilon = 0.001, and C=10.

4.3.1.3 Two Spiral Problem

The two-spiral problem is primarily used as a benchmark for pattern classification. It can also be used as an approximation problem where patterns on one spiral should produce +1 outputs, while patterns on the other spiral should produce -1 outputs.

This problem is widely used as a challenging benchmark to evaluate the efficiency of learning algorithms and their network architectures. For the purpose of approximation the two-spiral data set needs to be better defined, so in this work 388 patterns were used instead of the typical 194 patterns.

The RBF-MLP networks proposed in [74] required at least 74 RBF units to solve the two-spiral problem. It was reported in [75] that the two-spiral problem was solved using 70 hidden RBF units. Using the ortho-normalization procedure in [76], the two-spiral problem can be solved with at least 64 RBF kernel functions.

Applying the ErrCor algorithm, Figure 4.3-8 shows several steps in the training process. One may notice that, each newly added RBF unit contributes the error reduction during the training process. The ErrCorr algorithm constructs the network by adding one RBF unit at a time, and with 22 RBF units the training error drops below 0.003 (Figure 4.3-9). The SVR algorithm was tested using the LIBSVM package in [70]. SVR was trained to the two spiral problem using the parameters, C=1, G=0.5, and epsilon = 0.01. This output can be seen in Figure 4.3-10.

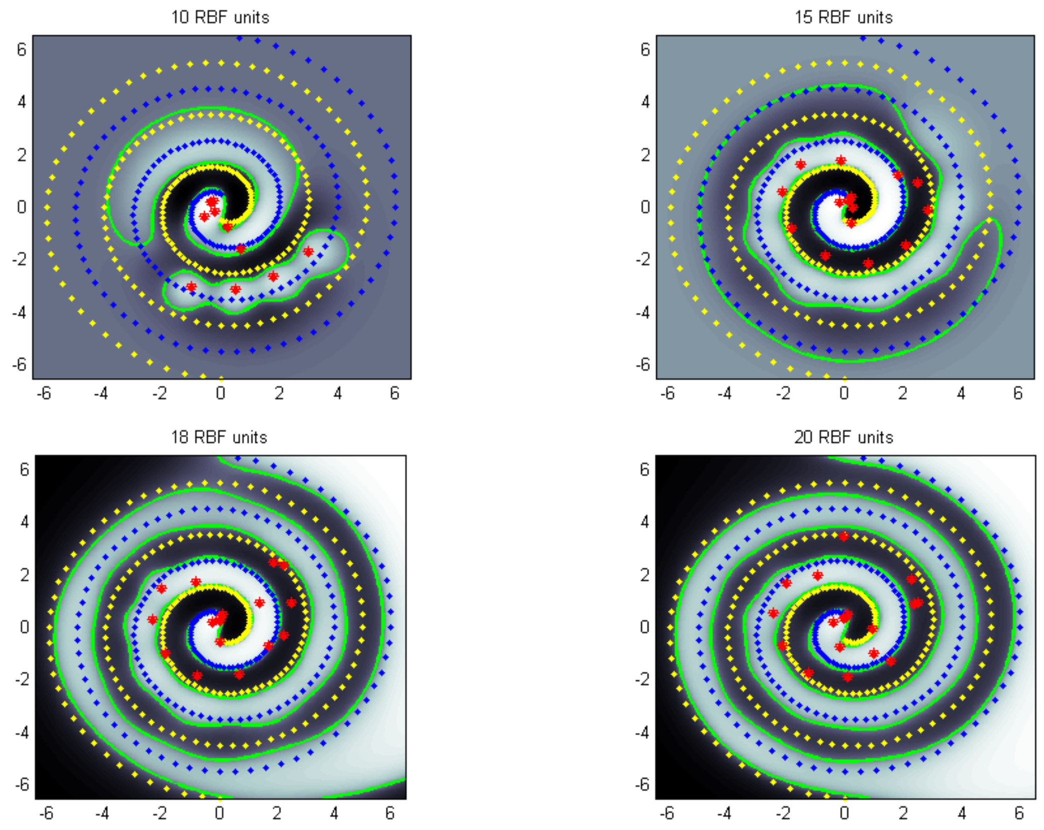


Figure 4.3-8 The ErrCor algorithm incrementally solves the two spiral problem. The two classes of patterns are shown as blue and yellow asterisks, while the green contour shows the network output. The red asterisks are the locations of the RBF centers.

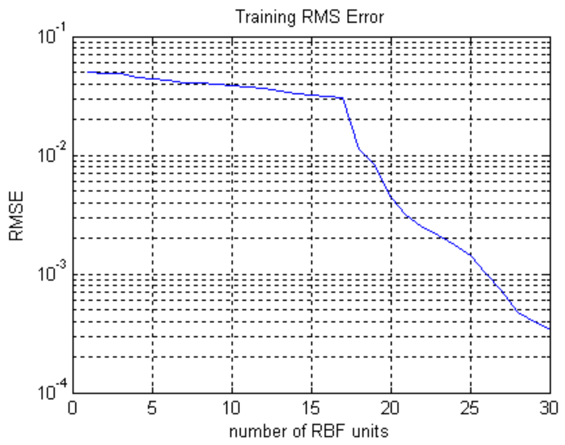


Figure 4.3-9 The RMSE as the ErrCor algorithm adds neurons to solve the two spiral problem.

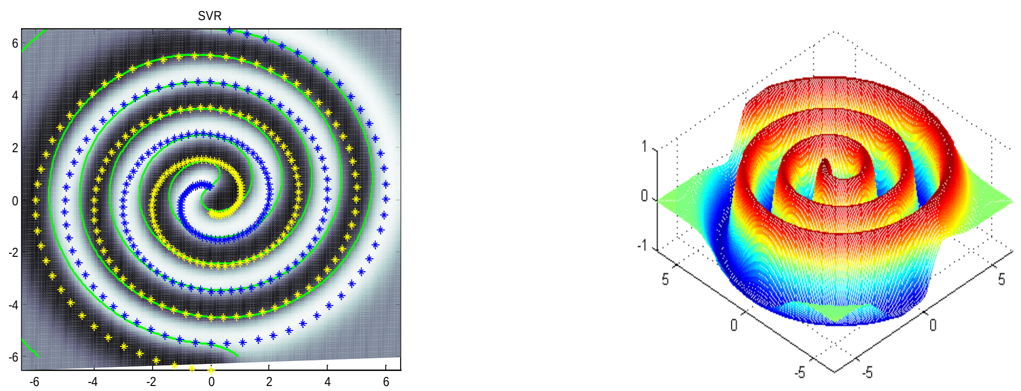


Figure 4.3-10 The SVR algorithm solves the two spiral problem. 297 patterns were used as support vectors to reach an RMSE of 0.003.

4.3.2 Real-World Data

This section compares ErrCor with well-known algorithms on traditional benchmarks from various repositories, [69]. These are real life problems with many dimensions and with number of patterns from hundreds to thousands. Table 9 depicts the specifications of the benchmark data sets. In our experiments, all of the inputs have been normalized into the range [-1,1] while the outputs have been normalized into [0,1].

Table 9 Real-World Dataset Information

Problem	# Training Data	# Testing Data	#Attributes
Abalone	2000	2177	8
Auto-MPG	320	78	7
Auto-Price	80	79	15
Bos Housing	250	256	13
Cal Housing	8000	12640	8
Delta-Ailerons	3000	4129	5
Delta-Elevators	4000	5517	6
Machine CPU	100	109	6

In each benchmark samples are randomly divided into two categories: training samples and validation samples. These experiments are repeated with 20 different random selections so the average and standard deviation results can be evaluated. Table 10 and Table **11** and Figure 4.3-11and Figure **4.3-12** present more detailed comparisons on the Abalone and Auto-MPG datasets. These comparisons are given to compare the behavior of the ErrCor algorithm with other popular algorithms. Table 12 presents a comparison of validation errors and Table 13 presents a comparison of units required to reach the desired errors by currently popular algorithms on all of the datasets.

The proposed algorithm was compared with other algorithms such as: GAP [16], GGAP [48], GGAP-GMM [49], SVR [50], [51], I-ELM [59], CI-ELM [60], EI-ELM [61], MRAN [14], RAN-EKF [12], RAN [11]. The parameters for these algorithms were set based on the data presented in the aforementioned papers. For all data sets, the ELM algorithm parameters were centers in the range of inputs, [-1,1], and impact factors in the range (0, 0.5]. For GAP-RBF the parameters are fixed at $\epsilon_{max} = 1.15, \epsilon_{min} = 0.04, \kappa = 0.10,\ and\ \gamma = 0.999$. For the MRAN algorithm, the threshold for growing and pruning was set as $e_{min}^{MR} = 0.0001$, and the appropriate size of the sliding window was chosen as $M = 50$. The parameters for GGAP were $e_{min}^{GG} = 0.00008$ and $e_{min}^{GG} = 0.00007$for Abalone and Auto-MPG respectively. For GGAP-GMM the parameters for the significance threshold are $e_{min}^{gmm} = 0.08, \eta = 0.1$ for Abalone and $e_{min}^{gmm} = 0.11, \eta = 0.06$ for Auto-MPG. The DRNN algorithm used a parameter of A=2000 and A=40 for the abalone and fuel consumption datasets respectively. The parameters for SVR are mentioned in Table 12.

As before, the testing environment of the proposed algorithm consists of a Windows 7 64-bit operating system, an Intel Core i7-2600 CPU @ 3.4 GHz processor, and 8GB RAM.

It can be noticed from Figure 4.3-11 and Figure **4.3-12** that, the proposed ErrCor algorithm reaches smaller training/testing errors with a more compact RBF architecture than the other algorithms. Longer training with more than four RBF units leads to smaller training errors, but greater validation errors due to over-fitting. One may notice that other offline algorithms such as ELM, SVR, or DRNN give much worse results. DRNN was omitted from these figures because the best case yielded a validation error of RMSE=0.34.

A comparison of training times for different algorithms on both the Abalone and the Fuel Consumption data sets can be seen in Table 10. Again, the proposed ErrCor algorithm has a larger training time than the SVR, I-ELM, and CI-ELM algorithms, but a faster training time than the GGAP, MRAN, RANEKF, RAN, and EI-ELM algorithms. Notice that the SVR algorithm may show a lower training error than ErrCor because ErrCor training was stopped when a very small validation error was reached.

A more important comparison for the purpose at hand is that of validation times. This comparison answers the question, "How efficient is the network once it has been trained?" In general, for RBF networks, this will be determined by how many units are in the network. As in section 4.3.1, a normalized computation time for RBF calculation was used to calculate the testing time for each algorithm. A comparison of computation time for testing patterns is shown in Table 11.

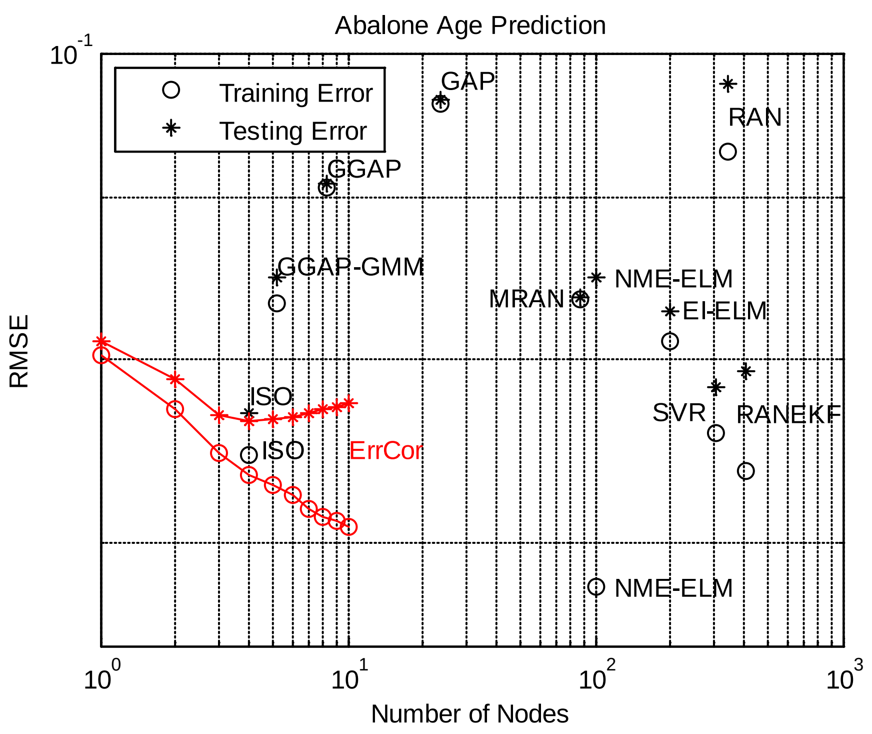


Figure 4.3-11 Abalone age prediction problem: training/testing average sum square errors vs. average number of RBF units.

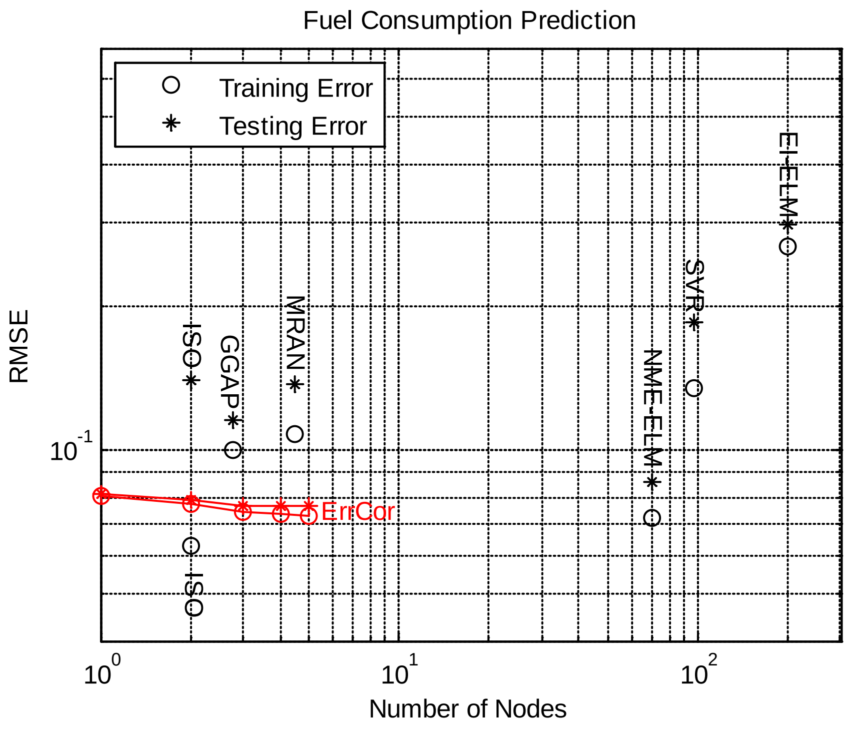


Figure 4.3-12 Fuel consumption prediction problem: training/testing average sum square errors vs. average number of RBF units.

Table 10 comparison between training times and training errors for abalone and fuel consumption problem

	Abalone		Fuel Consumption	
Algorithm	Time(s)	RMSE	Time(s)	RMSE
GAP	14.28	0.0963	0.4524	0.1144
MRAN	255.8	0.0836	1.4644	0.1086
RANEKF	15480	0.0738	1.0103	0.1088
RAN	105.17	0.0931	0.8042	0.2923
SVR	0.4446	0.0706	0.0210	0.0465
I-ELM	0.5990	0.0920	0.0593	0.0949
CI-ELM	0.6635	0.0827	0.0612	0.0929
EI-ELM	5.732	0.0811	0.5638	0.0930
NME-ELM	0.0944	0.0597	0.0184	0.0724
DRNN	9.404	0.0820	0.0837	0.3506
ISO	8.497	0.0747	0.6657	0.0724
ErrCor	4.808	0.0758	0.5030	0.0671

Table 11 comparison between validation times per pattern and validation errors for abalone and fuel consumption problem

Algorithm	Abalone		Fuel Consumption	
	Time(s)	RMSE	Time(s)	RMSE
GAP	2.82e-5	0.0966	3.73e-6	0.1404
MRAN	1.05e-4	0.0837	5.33e-6	0.1376
RANEKF	4.89e-4	0.0794	6.14e-6	0.1387
RAN	4.13e-4	0.0978	5.31e-6	0.3081
SVR	6.75e-4	0.0846	1.15e-4	0.0785
I-ELM	2.39e-4	0.0938	2.39e-4	0.0970
CI-ELM	2.39e-4	0.0857	2.39e-4	0.1105
EI-ELM	2.39e-4	0.0829	2.39e-4	0.0892
NME-ELM	1.20e-4	0.0849	8.37e-5	0.0861
DRNN	2.39e-3	0.3361	3.82e-4	0.3098
ISO	4.78e-6	0.0770	2.39e-6	0.1445
ErrCor	3.59e-6	0.0765	3.59e-6	0.0792

Table 12 comparison of most current algorithms in terms of testing RMSE on several real-world benchmark problems

Real World Problem	I-ELM	CI-ELM	EI-ELM	SVR		NME-ELM	ErrCor
	RMSE (Test)			RMSE	C, γ	RMSE	RMSE
Abalone	0.0938	0.0827	0.0829	0.0846	$(2^4, 2^{-6})$	0.0849	0.0765
Auto-MPG	0.0970	0.0929	0.0892	0.0785	$(2^0, 2^0)$	0.0861	0.0792
Auto-Price	0.1261	0.1196	0.1139	0.1052	$(2^8, 2^{-5})$	0.1104	0.0909
Boston Housing	0.1320	0.1455	0.1077	0.1155	$(2^4, 2^{-3})$	0.1124	0.0989
California Housing	0.1731	0.1660	0.1503	0.1311	$(2^3, 2^1)$	0.1642	0.1223
Delta-Ailerons	0.0632	0.0494	0.0448	0.0467	$(2^3, 2^{-3})$	0.0413	0.0394
Delta-Elevators	0.0790	0.0622	0.0575	0.0603	$(2^0, 2^{-2})$	0.0557	0.0532
Machine CPU	0.0674	0.0589	0.0829	0.0846	$(2^6, 2^{-4})$	0.0791	0.0765

Table 13 comparison of most current algorithms in terms of network size on several real-world benchmark problems

Real World Problem	ELMS	SVR	NME-ELM	ErrCor
Abalone	200	310	100	4
Auto-MPG	200	96	73	3
Auto-Price	200	22	82	2
Boston Housing	200	47	94	4
California Housing	200	2189	195	10
Delta-Ailerons	200	83	182	3
Delta-Elevators	200	261	140	3
Machine CPU	200	8	28	1

Chapter 5 Conclusions

Much of the appeal to modern day computing comes from the ability to solve difficult problems without the use of a human. However, there are some complex real-world problems that cannot effectively be solved by traditional approaches such as first principles modeling or explicit statistical modeling. Many of these problems are not considered to be mathematically well-posed problems. In many cases, nature is able to handle incredibly difficult problems in an ever-changing context. In an attempt to imitate nature, a computational unit called a neuron is used to provide a mapping from input data to an output. The construction of networks of these computational units for the task of solving problems is a widely researched topic in the field of computer engineering.

Artificial neural networks are used extensively in industry to solve important problems such as, fault detection, adaptive control, and computer vision. However, many of the currently used methods for obtaining a suitable ANN for a given problem could be improved. In some cases, it may make more sense to use locally tuned units instead of the global units that are typically used. It has been shown that locally tuned neurons have good performance in areas where spatial relationships are important such as, computer vision and signal processing. Once the type of neuron is chosen, there are still training and construction considerations that must be made. Specifically addressing the questions: “How should I layout my neural network?” and “How can I train this neural network to achieve acceptable performance?” is paramount to developing optimally designed networks.

Several attempts have been made at answering the questions posed above when building RBF network systems. The RAN, RANEKF, and MRAN algorithms attempt to construct RBF networks that use a minimal number of neurons to represent a large amount of data. The GGAP and GGAP-GMM algorithms attempt to further minimize network sizes by pruning neurons that do not adequately impact network outputs. The ELM family of algorithms addresses the question of training. These algorithms focus on constructing networks to achieve acceptable error performance as quickly as possible.

The algorithms presented in this work attempt to both construct minimally sized networks and reach desirable error performance. The NME-ELM algorithm uses the concept of allocating neurons to compensate for the largest value in the error function. Then it adjusts the radius of the neurons using the Nelder-Mead Simplex method. This algorithm was shown to generate better error performance and more compact networks than the ELM algorithms while still maintaining a fast training time. The Error Correction algorithm also minimizes network size by allocating neurons to compensate for large errors, then reaches incredibly low error levels by training the neurons with a second-order training method. This algorithm was demonstrated to reach very good error performance with extremely compact networks. These algorithms were compared with many of the other state of the art approaches to constructing networks on several benchmark tests and real-world data sets. The experimental results demonstrate effective construction of compact and robust networks.

References

[1] V. Zatsiorsky and B. Prilutsky, *Biomechanics of Skeletal Muscles*. Human Kinetics.

[2] L. A. Zadeh, “Fuzzy sets,” *Inf. Control*, vol. 8, no. 3, pp. 338–353, Jun. 1965.

[3] L. Fogel, A. Owens, and M. Walsh, *Artificial Intelligence through Simulated Evolution*. John Wiley, 1966.

[4] T. Mitchell, *Machine Learning*. Burr Ridge, IL: Mcgraw Hill, 1997.

[5] M. Mohri, A. Rostamizadeh, and A. Talwalkar, *Foundations of Machine Learning*. MIT Press, 2012.

[6] T. M. Cover, “Geometrical and Statistical Properties of Systems of Linear Inequalities with Applications in Pattern Recognition,” *IEEE Trans. Electron. Comput.*, vol. EC-14, no. 3, pp. 326–334, Jun. 1965.

[7] J. Moody and C. Darken, “Learning with localized receptive fields.” Yale Univ., Department of Computer Science, 1988.

[8] T. Poggio and F. Girosi, “A theory of networks for approximation and learning.” Massachusetts Inst of Tech Cambridge Artificial Intelligence Lab, 1989.

[9] S. Chen, C. F. N. Cowan, and P. M. Grant, “Orthogonal least squares learning algorithm for radial basis function networks,” *IEEE Trans. Neural Netw.*, vol. 2, no. 2, pp. 302–309, Mar. 1991.

[10] D. Wettschereck and T. Dietterich, “Improving the performance of radial basis function networks by learning center locations,” *Adv. Neural Inf. Process. Syst.*, vol. 4, pp. 1133–1140, 1992.

[11] J. Platt, “A Resource-allocating Network for Function Interpolation,” *Neural Comput*, vol. 3, no. 2, pp. 213–225, Jun. 1991.

[12] V. Kadirkamanathan and M. Niranjan, “A Function Estimation Approach to Sequential Learning with Neural Networks,” *Neural Comput*, vol. 5, no. 6, pp. 954–975, Nov. 1993.

[13] L. Yingwei, N. Sundararajan, and P. Saratchandran, “A Sequential Learning Scheme for Function Approximation Using Minimal Radial Basis Function Neural Networks,” *Neural Comput.*, vol. 9, no. 2, pp. 461–478, Feb. 1997.

[14] N. Sundararajan and P. Saratchandran, “Radial Basis Function Neural Networks With Sequential Learning: MRAN and Its Applications,” Singapore: World Scientific, 1999.

[15] N. B. Karayiannis and G. W. Mi, “Growing radial basis neural networks: merging supervised and unsupervised learning with network growth techniques,” *IEEE Trans. Neural Netw.*, vol. 8, no. 6, pp. 1492–1506, Nov. 1997.

[16] G.-B. Huang, P. Saratchandran, and N. Sundararajan, “An efficient sequential learning algorithm for growing and pruning RBF (GAP-RBF) networks,” *IEEE Trans. Syst. Man Cybern. Part B Cybern.*, vol. 34, no. 6, pp. 2284–2292, Dec. 2004.

[17] S. I. Ch’ng, K. P. Seng, and L.-M. Ang, “Adaptive momentum Levenberg-Marquardt RBF for face recognition,” in *2012 IEEE International Conference on Circuits and Systems (ICCAS)*, 2012, pp. 126–131.

[18] K. Meng, Z.-Y. Dong, D. H. Wang, and K. P. Wong, “A Self-Adaptive RBF Neural Network Classifier for Transformer Fault Analysis,” *IEEE Trans. Power Syst.*, vol. 25, no. 3, pp. 1350–1360, Aug. 2010.

[19] S. Huang and K. K. Tan, “Fault Detection and Diagnosis Based on Modeling and Estimation Methods,” *IEEE Trans. Neural Netw.*, vol. 20, no. 5, pp. 872–881, May 2009.

[20] L. Cai, A. B. Rad, and W.-L. Chan, “An Intelligent Longitudinal Controller for Application in Semiautonomous Vehicles,” *IEEE Trans. Ind. Electron.*, vol. 57, no. 4, pp. 1487–1497, Apr. 2010.

[21] M. L. Corradini, V. Fossi, A. Giantomassi, G. Ippoliti, S. Longhi, and G. Orlando, “Minimal Resource Allocating Networks for Discrete Time Sliding Mode Control of Robotic Manipulators,” *IEEE Trans. Ind. Inform.*, vol. 8, no. 4, pp. 733–745, Nov. 2012.

[22] S.-L. Dai, C. Wang, and F. Luo, “Identification and Learning Control of Ocean Surface Ship Using Neural Networks,” *IEEE Trans. Ind. Inform.*, vol. 8, no. 4, pp. 801–810, Nov. 2012.

[23] L. Guo and L. Parsa, “Model Reference Adaptive Control of Five-Phase IPM Motors Based on Neural Network,” *IEEE Trans. Ind. Electron.*, vol. 59, no. 3, pp. 1500–1508, Mar. 2012.

[24] F. F. M. El-Sousy, “Adaptive Dynamic Sliding-Mode Control System Using Recurrent RBFN for High-Performance Induction Motor Servo Drive,” *IEEE Trans. Ind. Inform.*, vol. 9, no. 4, pp. 1922–1936, Nov. 2013.

[25] Y. J. Lee and J. Yoon, “Nonlinear Image Upsampling Method Based on Radial Basis Function Interpolation,” *IEEE Trans. Image Process.*, vol. 19, no. 10, pp. 2682–2692, Oct. 2010.

[26] H. Zhuang, K.-S. Low, and W.-Y. Yau, “Multichannel Pulse-Coupled-Neural-Network-Based Color Image Segmentation for Object Detection,” *IEEE Trans. Ind. Electron.*, vol. 59, no. 8, pp. 3299–3308, Aug. 2012.

[27] J. Park and I. W. Sandberg, “Universal approximation using radial-basis-function networks,” *Neural Comput*, vol. 3, no. 2, pp. 246–257, Jun. 1991.

[28] M. J. D. Powell, “Algorithms for Approximation,” J. C. Mason and M. G. Cox, Eds. New York, NY, USA: Clarendon Press, 1987, pp. 143–167.

[29] B. M. Wilamowski, “Neural network architectures and learning algorithms,” *IEEE Ind. Electron. Mag.*, vol. 3, no. 4, pp. 56–63, 2009.

[30] D. Hunter, H. Yu, M. S. Pukish, J. Kolbusz, and B. M. Wilamowski, “Selection of Proper Neural Network Sizes and Architectures #x2014;A Comparative Study,” *IEEE Trans. Ind. Inform.*, vol. 8, no. 2, pp. 228–240, May 2012.

[31] B. M. Wilamowski, H. Yu, and K. T. Chung, “Parity-N problems as a vehicle to compare efficiency of neural network architectures,” in *Industrial Electronics Handbook, 2nd Edition*, vol. 5, CRC Press, 2011, pp. 10–1 to 10–8.

[32] S. Ferrari and M. Jensenius, “A Constrained Optimization Approach to Preserving Prior Knowledge During Incremental Training,” *IEEE Trans. Neural Netw.*, vol. 19, no. 6, pp. 996–1009, Jun. 2008.

[33] Q. Song, J. C. Spall, Y. C. Soh, and J. Ni, “Robust Neural Network Tracking Controller Using Simultaneous Perturbation Stochastic Approximation,” *IEEE Trans. Neural Netw.*, vol. 19, no. 5, pp. 817–835, May 2008.

[34] A. Slowik, “Application of an Adaptive Differential Evolution Algorithm With Multiple Trial Vectors to Artificial Neural Network Training,” *IEEE Trans. Ind. Electron.*, vol. 58, no. 8, pp. 3160–3167, Aug. 2011.

[35] V. V. Phansalkar and P. S. Sastry, “Analysis of the back-propagation algorithm with momentum,” *IEEE Trans. Neural Netw.*, vol. 5, no. 3, pp. 505–506, May 1994.

[36] L. Torvik and B. Wilamowski, “Modification of gradient computation in the back-propagation algorithm,” in *ANNIE ’93 - Artificial Neural Networks in Engineering, (Intelligent Engineering Systems Through Artificial Neural Networks)*, vol. 3, New York: ASME PRESS, 1993, pp. 175–180.

[37] A. Salvettit and B. Wilamowski, “Introducing stochastic processes within the backpropagation algorithm for improved convergence,” in *ANNIE’94 - Artificial Neural Networks in Engineering, (Intelligent Engineering Systems Through Artificial Neural Networks)*, vol. 4, New York: ASME PRESS, 1994, pp. 205–209.

[38] M. Riedmiller and H. Braun, “A direct adaptive method for faster backpropagation learning: the RPROP algorithm,” in *, IEEE International Conference on Neural Networks, 1993*, 1993, pp. 586–591 vol.1.

[39] S. E. Fahlman, “An Empirical Study of Learning Speed in Backpropagation Networks.” 1988.

[40] B. M. Wilamowski, “Challenges in applications of computational intelligence in industrial electronics,” in *2010 IEEE International Symposium on Industrial Electronics (ISIE)*, 2010, pp. 15–22.

[41] R. Battiti, “First- and Second-order Methods for Learning: Between Steepest Descent and Newton’s Method,” *Neural Comput*, vol. 4, no. 2, pp. 141–166, Mar. 1992.

[42] D. Marquardt, “An algorithm for least-squares estimation of nonlinear parameters,” *J. Soc. Ind. Appl. Math.*, vol. 11, no. 2, pp. 431–441, 1963.

[43] B. M. Wilamowski and H. Yu, “Improved Computation for Levenberg #x2013;Marquardt Training,” *IEEE Trans. Neural Netw.*, vol. 21, no. 6, pp. 930–937, Jun. 2010.

[44] B. Fritzke, “Fast learning with incremental RBF Networks,” 1994, pp. 2–5.

[45] C. Constantinopoulos and A. Likas, “An incremental training method for the probabilistic RBF network,” *IEEE Trans. Neural Netw.*, vol. 17, no. 4, pp. 966–974, July.

[46] L. H. Shenq Chen, “Symmetric complex-valued RBF receiver for multiple-antenna-aided wireless systems.,” *IEEE Trans. Neural Netw. Publ. IEEE Neural Netw. Counc.*, vol. 19, no. 9, pp. 1659–65, 2008.

[47] M. L. Kothari, S. Madnani, and R. Segal, “Orthogonal least squares learning algorithm based radial basis function (RBF) network adaptive power system stabilizer,” presented at the , 1997 IEEE International Conference on Systems, Man, and Cybernetics, 1997. Computational Cybernetics and Simulation, Oct, vol. 1, pp. 542–547 vol.1.

[48] G.-B. Huang, P. Saratchandran, and N. Sundararajan, “A generalized growing and pruning RBF (GGAP-RBF) neural network for function approximation,” *IEEE Trans. Neural Netw.*, vol. 16, no. 1, pp. 57–67, Jan. 2005.

[49] M. Bortman and M. Aladjem, “A Growing and Pruning Method for Radial Basis Function Networks,” *IEEE Trans. Neural Netw.*, vol. 20, no. 6, pp. 1039–1045, Jun. 2009.

[50] V. N. Vapnik, *Statistical Learning Theory*, 1st ed. Wiley-Interscience, 1998.

[51] A. J. Smola and B. Schölkopf, “A tutorial on support vector regression,” *Stat. Comput.*, vol. 14, no. 3, pp. 199–222, 2004.

[52] K. Hornik, “Approximation capabilities of multilayer feedforward networks,” *Neural Netw.*, vol. 4, no. 2, pp. 251–257, 1991.

[53] M. Leshno and S. Schocken, “Multilayer feedforward networks with a nonpolynomial activation function can approximate any function,” *Neural Netw.*, vol. 6, pp. 861–867, 1993.

[54] G.-B. Huang and H. A. Babri, “Upper bounds on the number of hidden neurons in feedforward networks with arbitrary bounded nonlinear activation functions,” *IEEE Trans. Neural Netw.*, vol. 9, no. 1, pp. 224–229, Jan. 1998.

[55] G.-B. Huang, “Learning capability and storage capacity of two-hidden-layer feedforward networks,” *IEEE Trans. Neural Netw.*, vol. 14, no. 2, pp. 274–281, Mar. 2003.

[56] G.-B. Huang, Q.-Y. Zhu, and C.-K. Siew, “Real-time learning capability of neural networks,” *IEEE Trans. Neural Netw.*, vol. 17, no. 4, pp. 863–878, Jul. 2006.

[57] G.-B. Huang, Q.-Y. Zhu, and C.-K. Siew, “Extreme learning machine: Theory and applications,” *Neurocomputing*, vol. 70, no. 1–3, pp. 489–501, Dec. 2006.

[58] P. L. Bartlett, “The sample complexity of pattern classification with neural networks: the size of the weights is more important than the size of the network,” *IEEE Trans. Inf. Theory*, vol. 44, no. 2, pp. 525–536, Mar. 1998.

[59] G.-B. Huang, L. Chen, and C.-K. Siew, “Universal approximation using incremental constructive feedforward networks with random hidden nodes,” *IEEE Trans. Neural Netw.*, vol. 17, no. 4, pp. 879 – 892, Jul. 2006.

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

[61] 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.

[62] A. R. Barron, “Universal approximation bounds for superpositions of a sigmoidal function,” *IEEE Trans. Inf. Theory*, vol. 39, no. 3, pp. 930–945, May 1993.

[63] J. A. Nelder and R. Mead, “A Simplex Method for Function Minimization,” *Comput. J.*, vol. 7, no. 4, pp. 308–313, Jan. 1965.

[64] E. Boyd, K. W. Kennedy, R. A. Tapia, V. J. Torczon, and V. J. Torczon, “Multi-Directional Search: A Direct Search Algorithm for Parallel Machines,” Rice University, 1989.

[65] N. Pham and B. M. Wilamowski, “Improved Nedler Mead’s Simplex Method and Applications,” *J. Comput.*, vol. 3, no. 3, pp. 55–63, Mar. 2011.

[66] F. Gao and L. Han, “Implementing the Nelder-Mead simplex algorithm with adaptive parameters,” *Comput. Optim. Appl.*, vol. 51, no. 1, pp. 259–277, Jan. 2012.

[67] P. Reiner and B. M. Wilamowski, “Efficient incremental construction of RBF networks using quasi-gradient method,” *Neurocomputing*, vol. 150, Part B, pp. 349–356, Feb. 2015.

[68] J. C. Lagarias, J. A. Reeds, M. H. Wright, and P. E. Wright, “Convergence Properties of the Nelder-Mead Simplex Method in Low Dimensions,” *SIAM J. Optim.*, vol. 9, pp. 112–147, 1998.

[69] A. Asuncion and A. Frank, “UCI Machine Learning Repository.” University of California, Irvine, School of Information and Computer Sciences, 2010.

[70] C.-C. Chang and C.-J. Lin, “LIBSVM: a library for support vector machines.” ACM Transactions on Intelligent Systems and Technology, 2011.

[71] T. Xie, H. Yu, J. Hewlett, P. Rozycki, and B. Wilamowski, “Fast and Efficient Second-Order Method for Training Radial Basis Function Networks,” *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 23, no. 4, pp. 609–619, 2012.

[72] B. M. Wilamowski and H. Yu, “Neural Network Learning Without Backpropagation,” *IEEE Trans. Neural Netw.*, vol. 21, no. 11, pp. 1793–1803, Nov. 2010.

[73] H. Yu, P. D. Reiner, T. Xie, T. Bartczak, and B. M. Wilamowski, “An Incremental Design of Radial Basis Function Networks,” *IEEE Trans. Neural Netw. Learn. Syst.*, vol. Early Access Online, 2014.

[74] N. Chaiyaratana and A. M. S. Zalzala, “Evolving hybrid RBF-MLP networks using combined genetic/unsupervised/supervised learning,” in *Control ’98. UKACC International Conference on (Conf. Publ. No. 455)*, 1998, vol. 1, pp. 330–335 vol.1.

[75] R. Neruda and P. Kudová, “Learning methods for radial basis function networks,” *Future Gener. Comput. Syst.*, vol. 21, no. 7, pp. 1131–1142, Jul. 2005.

[76] W. Kaminski and P. Strumillo, “Kernel orthonormalization in radial basis function neural networks,” *IEEE Trans. Neural Netw.*, vol. 8, no. 5, pp. 1177–1183, Sep. 1997.

Appendix

Main Files:

Peaks_Test

```
%Benchmark Peaks Test for ELM, I-ELM, EI-ELM, CI-ELM, NME-ELM, ErrCor and SVR
format compact; clear all;
%create evenly spaced validation data
x=linspace(-1,1,30);
g=1;
for i=1:30
    for j=1:30
        in(g,:)=[x(i) x(j)];
        g=g+1;
    end
end
nts=size(in,1);
% use the peaks equation
y =(0.3-1.8*in(:,1)+2.7*in(:,1).^2).*exp(-1-6*in(:,2)-9*in(:,1).^2-9*in(:,2).^2) ...
    - (0.6*in(:,1)-27*in(:,1).^3-243*in(:,2).^5).*exp(-9*in(:,1).^2-9*in(:,2).^2) ...
    - 1/30*exp(-1-6*in(:,1)-9*in(:,1).^2-9*in(:,2).^2);
yn=awgn(y,15);
x1=reshape(in(:,1),30,30);y1=reshape(in(:,2),30,30);z1=reshape(y,30,30);zn=reshape(yn,30,30);
figure(2);clf;surf(x1,y1,zn);

% load randomly spaced training data
X=load('peaks500.dat');
[np,nd]=size(X);
trainx=X(:,1:nd-1);targets=X(:,nd);
targets=awgn(targets,20);
trIn=trainx; trOut=targets;
trSize=size(trIn,1);
tstIn=in;tstOut=y;
RMSEVN(1)=sqrt((y'*y)/900);
RMSEVE(1,:)=ones(1,20)*sqrt((y'*y)/900);
RMSEVI(1,:)=ones(1,20)*sqrt((y'*y)/900);
RMSEVEI(1,:)=ones(1,20)*sqrt((y'*y)/900);
RMSEVCI(1,:)=ones(1,20)*sqrt((y'*y)/900);
RMSEVSV(1,:)=sqrt((y'*y)/900);
rangesb=[-10 -5 -2 -1 0];
rangest=[0 1 2 5 10];
C=[2^0 2^1 2^2 2^3 2^4 2^5 2^6 2^7 2^8 2^9 2^10];
G=[0.1 0.5 1 1.5 2];
GE=linspace(0.1,3,25);
for j=1:100
    t=mod(j-1,5)+1;
    v=ceil(j/5);
    nodes=j;
    tic;
    [cent, weights, radius, mseTr]=NME_ELM3(trainx,targets,nodes);
    time1(j)=toc
    % E(3)=mseTr(nodes+1)
    figure(3);clf;plot(0:nodes,mseTr);
    O=calc_ELM_Out1(in,weights,cent,radius,nodes);
    er=y-O;SSEV=er'*er;RMSEVN(j+1)=sqrt(SSEV/nts)
    x1=reshape(in(:,1),30,30);y1=reshape(in(:,2),30,30);z1=reshape(O,30,30);
    figure(4);clf;surf(x1,y1,z1);
    wRange=[rangesb(t),rangest(v)];bRange=wRange;
    for i=1:20
       %Run Original ELM
        %train
        [inw outw bias O error]=ELMR(trainx,targets,2,GE(j),nodes);
        %verify
        O=calc_ELM_Out(tstIn,outw,inw,bias,nodes);
        er=y-O;SSEV=er'*er;RMSEVE(j+1,i)=sqrt(SSEV/nts);
       %run I-ELMS
        %train
        tic;
        [cent, weights, radius, mseTr]=I_ELM(trIn,trOut,nodes,GE(j),2);
        timeE(j,i)=toc;
        %verify
        O=calc_ELM_Out(tstIn,weights,cent,radius,nodes);
        ver=tstOut-O;SSEV=ver'*ver;RMSEVI(j+1,i)=sqrt(SSEV/nts);
        %train
        tic;
        [cent, weights, radius, mseTr]=CI_ELM(trIn,trOut,nodes,GE(j),2);
        timeCI(j,i)=toc;
        %verify
        O=calc_ELM_Out(tstIn,weights,cent,radius,nodes);
        ver=tstOut-O;SSEV=ver'*ver;RMSEVCI(j+1,i)=sqrt(SSEV/nts);
        %train
        tic;
        [cent, weights, radius, mseTr]=EI_ELM(trIn,trOut,nodes,GE(j),2);
        timeEI(j,i)=toc;
        %verify
        O=calc_ELM_Out(tstIn,weights,cent,radius,nodes);
        ver=tstOut-O;SSEV=ver'*ver;RMSEVEI(j+1,i)=sqrt(SSEV/nts);


        tic;
           [weights_output, widths, weights_input,
centers]=ISO_RBF(nodes,trIn,trOut,stop,maximum_iteration)
        Time(4,i)=toc;
        O=verification(weights_input, weights_output, widths, centers, tstIn)
        ver=tstOut-O;SSEV=ver'*ver;RMSEV4(k)=sqrt(SSEV/nts);

    end

        tic;
           [weights_output, widths, cebters,
weights_input]=ErrCor(nodes,trIn,trOut,stop,maximum_iteration)
        Time(5,i)=toc;
        O=verification(weights_input, weights_output, widths, centers, tstIn)
        ver=tstOut-O;SSEV=ver'*ver;RMSEV5(k)=sqrt(SSEV/nts);

    %      run SVR
    d=data(trIn,trOut);
    tic;
    [tr,a]=train(svr({kernel('rbf',0.5),'optimizer = "libsvm"',['C='
num2str(C(j))],'epsilon=0.1'}),d);
    timeSV(i)=toc;
    svs(j) = size(a.Xsv.X,1);
    sseTr = sum((trOut-tr.X).^2);
    mseTr = sseTr/trSize;
    rmseTr(i) = sqrt(mseTr);
    %test the svm
    d = data(tstIn);
    cost=test(a,d);
    sseTst = sum((tstOut-cost.X).^2);
    mseTst = sseTst/nts;
    rmseTst(j)= sqrt(mseTst);
end
RMSEVE=mean(RMSEVE,2);
RMSEVI=mean(RMSEVI,2);
RMSEVEI=mean(RMSEVEI,2);
RMSEVCI=mean(RMSEVCI,2);
RMSEVSV=[RMSEVSV;rmseTst'];
figure(1);clf;
plot(0:j,RMSEVN,'LineWidth',1.5);hold all;
plot(0:j,RMSEVE,'LineWidth',1.5);plot(0:j,RMSEVI,'LineWidth',1.5);plot(0:j,RMSEVEI,'LineWidth',1.
5);plot(0:j,RMSEVCI,'LineWidth',1.5);
plot(0:j,RMSEVSV,'LineWidth',1.5);
[sm,I1]=min(RMSEVE)
t=mod(I1-2,5)+1
v=ceil((I1-1)/5)
```

```
[sm,I2]=min(RMSEVI)
t=mod(I2-1,5)+1
v=ceil(I2/5)
[sm,I3]=min(RMSEVEI)
t=mod(I3-1,5)+1
v=ceil(I3/5)
[sm,I4]=min(RMSEVCI)
t=mod(I4-1,5)+1
v=ceil(I4/5)
[sm,I5]=min(RMSEVSV)
t=mod(I5-2,5)+1
v=ceil((I5-1)/5)
```

Rapidly_Changing_Function_Test

```
format compact; clear all; clc; close all;
% Benchmark rapidly changing function Test for ELM, I-ELM, EI-ELM, CI-ELM, NME-ELM, ErrCor and
SVR

trIn=linspace(0,10,3000)';
trOut=0.8*exp(-0.2*inputs).*sin(10*inputs);
ver_inputs = 10*rand(1500,1);
nts=1500;
tstIn = sort(ver_inputs);
tstOut = 0.8*exp(-0.2*ver_inputs).*sin(10*ver_inputs);
RMSEVN(1)=sqrt((y'*y)/nts);
RMSEVE(1,:)=ones(1,20)*sqrt((y'*y)/nts);
RMSEVI(1,:)=ones(1,20)*sqrt((y'*y)/nts);
RMSEVEI(1,:)=ones(1,20)*sqrt((y'*y)/nts);
RMSEVCI(1,:)=ones(1,20)*sqrt((y'*y)/nts);
RMSEVSV(1,:)=sqrt((y'*y)/900);
rangesb=[-10 -5 -2 -1 0];
rangest=[0 1 2 5 10];
C=[2^0 2^1 2^2 2^3 2^4 2^5 2^6 2^7 2^8 2^9 2^10];
G=[0.1 0.5 1 1.5 2];
GE=linspace(0.1,3,25);
for j=1:100
    t=mod(j-1,5)+1;
    v=ceil(j/5);
    nodes=j;
    tic;
    [cent, weights, radius, mseTr]=NME_ELM3(trainx,targets,nodes);
    time1(j)=toc
    % E(3)=mseTr(nodes+1)
    figure(3);clf;plot(0:nodes,mseTr);
    O=calc_ELM_Out1(in,weights,cent,radius,nodes);
    er=y-O;SSEV=er'*er;RMSEVN(j+1)=sqrt(SSEV/nts)
    x1=reshape(in(:,1),30,30);y1=reshape(in(:,2),30,30);z1=reshape(O,30,30);
    figure(4);clf;surf(x1,y1,z1);
    wRange=[rangesb(t),rangest(v)];bRange=wRange;
    for i=1:20
       %Run Original ELM
        %train
        [inw outw bias O error]=ELMR(trainx,targets,2,GE(j),nodes);
        %verify
        O=calc_ELM_Out(tstIn,outw,inw,bias,nodes);
        er=y-O;SSEV=er'*er;RMSEVE(j+1,i)=sqrt(SSEV/nts);
       %run I-ELMS
        %train
        tic;
        [cent, weights, radius, mseTr]=I_ELM(trIn,trOut,nodes,GE(j),2);
        timeE(j,i)=toc;
        %verify
        O=calc_ELM_Out(tstIn,weights,cent,radius,nodes);
        ver=tstOut-O;SSEV=ver'*ver;RMSEVI(j+1,i)=sqrt(SSEV/nts);
        %train
        tic;
        [cent, weights, radius, mseTr]=CI_ELM(trIn,trOut,nodes,GE(j),2);
        timeCI(j,i)=toc;
        %verify
        O=calc_ELM_Out(tstIn,weights,cent,radius,nodes);
        ver=tstOut-O;SSEV=ver'*ver;RMSEVCI(j+1,i)=sqrt(SSEV/nts);
        %train
        tic;
        [cent, weights, radius, mseTr]=EI_ELM(trIn,trOut,nodes,GE(j),2);
        timeEI(j,i)=toc;
        %verify
        O=calc_ELM_Out(tstIn,weights,cent,radius,nodes);
        ver=tstOut-O;SSEV=ver'*ver;RMSEVEI(j+1,i)=sqrt(SSEV/nts);


        tic;
           [weights_output, widths, weights_input,
centers]=ISO_RBF(nodes,trIn,trOut,stop,maximum_iteration)
        Time(4,i)=toc;
        O=verification(weights_input, weights_output, widths, centers, tstIn)
        ver=tstOut-O;SSEV=ver'*ver;RMSEV4(k)=sqrt(SSEV/nts);

    end

        tic;
           [weights_output, widths, cebters,
weights_input]=ErrCor(nodes,trIn,trOut,stop,maximum_iteration)
        Time(5,i)=toc;
        O=verification(weights_input, weights_output, widths, centers, tstIn)
        ver=tstOut-O;SSEV=ver'*ver;RMSEV5(k)=sqrt(SSEV/nts);

    %      run SVR
    d=data(trIn,trOut);
    tic;
    [tr,a]=train(svr({kernel('rbf',0.5),'optimizer = "libsvm"',['C='
num2str(C(j))],'epsilon=0.1'}),d);
    timeSV(i)=toc;
    svs(j) = size(a.Xsv.X,1);
    sseTr = sum((trOut-tr.X).^2);
    mseTr = sseTr/trSize;
    rmseTr(i) = sqrt(mseTr);
    %test the svm
    d = data(tstIn);
    cost=test(a,d);
    sseTst = sum((tstOut-cost.X).^2);
    mseTst = sseTst/nts;
    rmseTst(j)= sqrt(mseTst);
end
RMSEVE=mean(RMSEVE,2);
RMSEVI=mean(RMSEVI,2);
RMSEVEI=mean(RMSEVEI,2);
RMSEVCI=mean(RMSEVCI,2);
RMSEVSV=[RMSEVSV;rmseTst'];
figure(1);clf;
plot(0:j,RMSEVN,'LineWidth',1.5);hold all;
plot(0:j,RMSEVE,'LineWidth',1.5);plot(0:j,RMSEVI,'LineWidth',1.5);plot(0:j,RMSEVEI,'LineWidth',1.
5);plot(0:j,RMSEVCI,'LineWidth',1.5);
plot(0:j,RMSEVSV,'LineWidth',1.5);
[sm,I1]=min(RMSEVE)
t=mod(I1-2,5)+1
v=ceil((I1-1)/5)
[sm,I2]=min(RMSEVI)
t=mod(I2-1,5)+1
v=ceil(I2/5)
[sm,I3]=min(RMSEVEI)
t=mod(I3-1,5)+1
v=ceil(I3/5)
[sm,I4]=min(RMSEVCI)
t=mod(I4-1,5)+1
v=ceil(I4/5)
[sm,I5]=min(RMSEVSV)
t=mod(I5-2,5)+1
v=ceil((I5-1)/5)
```

Real_Data_Test

```
%Test the ELMs, SVR, ErrCor, and the NME_ELM on real-world data
format compact; clear all;
dat_name=('Abalone_Norm');
trSize=2000; C=2^4; G=2^-6;

% dat_name=('auto_MPG_N');
% trSize=320; C=2^0; G=2^0;
%
% dat_name=('Auto_Price_Norm');
% trSize=80; C=2^8; G=2^-5;
%
% dat_name=('Boston_Norm');
% trSize=250; C=2^4; G=2^-3;
%
% dat_name=('Cal_Norm');
% trSize=8000; C=2^3; G=2^1;
%
% dat_name=('Delta_Ailerons_Norm');
% trSize=3000; C=2^3; G=2^-3;
%
% dat_name=('Delta_Elevators_Norm');
% trSize=4000; C=2^0; G=2^-2;
%
% dat_name=('MachineCPU_Norm');
% trSize=100; C=2^6; G=2^-4;

X=load([dat_name,'.dat']);
[np,nd]=size(X); nts=np-trSize;
stop=0.001;

for j=1:200
    nodes=j;
    for i=1:20
        %Shuffle Data
        [trIn trOut tstIn tstOut]=RandomizeData(trSize,X);

        %NME-ELM
        tic;
        [cent, weights, radius, mseTr]=NME_ELM3(trIn,trOut,nodes);
        time(1,i)=toc;
        E(1)=mseTr(nodes+1)
        O=calc_ELM_Out(tstIn,weights,cent,radius,nodes);
        ver=tstOut-O;SSEV=ver'*ver;RMSEV(1,i)=sqrt(SSEV/nts);

        for k=1:20
            tic;
            [cent, weights, radius, mseTr]=I_ELM(trIn,trOut,nodes);
            time(2,i)=toc;
            E(2)=mseTr(nodes+1)
            O=calc_ELM_Out(tstIn,weights,cent,radius,nodes);
            ver=tstOut-O;SSEV=ver'*ver;RMSEV1(k)=sqrt(SSEV/nts);
            tic;
            [cent, weights, radius, mseTr]=EI_ELM(trIn,trOut,nodes);
            time(3,i)=toc;
            O=calc_ELM_Out(tstIn,weights,cent,radius,nodes);
            ver=tstOut-O;SSEV=ver'*ver;RMSEV2(k)=sqrt(SSEV/nts);
            E(3)=mseTr(nodes+1)

            tic;
            [cent, weights, radius, mseTr]=CI_ELM(trIn,trOut,nodes);
            time(3,i)=toc;
            O=calc_ELM_Out(tstIn,weights,cent,radius,nodes);
            ver=tstOut-O;SSEV=ver'*ver;RMSEV3(k)=sqrt(SSEV/nts);
            E(4)=mseTr(nodes+1)

           tic;
           [weights_output, widths, weights_input,
centers]=ISO_RBF(nodes,trIn,trOut,stop,maximum_iteration)
        Time(4,i)=toc;
        O=verification(weights_input, weights_output, widths, centers, tstIn)
        ver=tstOut-O;SSEV=ver'*ver;RMSEV4(k)=sqrt(SSEV/nts);
        end

        tic;
           [weights_output, widths, cebters,
weights_input]=ErrCor(nodes,trIn,trOut,stop,maximum_iteration)
        Time(5,i)=toc;
        O=verification(weights_input, weights_output, widths, centers, tstIn)
        ver=tstOut-O;SSEV=ver'*ver;RMSEV5(k)=sqrt(SSEV/nts);

        RMSEV(2,i)=mean(RMSEV1,2);
        RMSEV(3,i)=mean(RMSEV2,2);
        RMSEV(4,i)=mean(RMSEV3,2);
        RMSEV(5,i)=mean(RMSEV4,2);
        RMSEV(6,i)=RMSEV5;

        %SVR
        d=data(trIn,trOut);
        tic;
        [tr,a]=train(svr({kernel('rbf',G),'optimizer = "libsvm"',['C='
num2str(C)],'epsilon=0.1'}),d);
        time(4,i)=toc;
        svs(j) = size(a.Xsv.X,1);
        sseTr = sum((trOut-tr.X).^2);
        mseTr = sseTr/trSize;
        rmseTr(i) = sqrt(mseTr);
        %test the svm
        d = data(tstIn);
        cost=test(a,d);
        sseTst = sum((tstOut-cost.X).^2);
        mseTst = sseTst/nts;
        RMSEV(5,i)= sqrt(mseTst);
    end
    RMSE(j,:)=mean(RMSEV,2)';
    SDEV(j,:)=std(RMSEV,0,2)';
    tavg(j,:)=mean(time,2)';
end
xlswrite(['ELMS_SVR_',dat_name,'.xls'],[RMSE SDEV tavg]);
```

Two_Spiral_Test

```
format compact; clear all; clc; close all;
% Benchmark Two-Spiral Test for ELM, I-ELM, EI-ELM, CI-ELM, NME-ELM, ErrCor and SVR
X=load('spiral4.dat');
[np,nd]=size(X);
inputs = X(1:361,1:nd-1);
outputs = X(1:361,nd);
trIn=inputs;
trOut=outputs;
tstIn=inputs;
tstOut=outputs;
nts=np;

RMSEVN(1)=sqrt((y'*y)/nts);
RMSEVE(1,:)=ones(1,20)*sqrt((y'*y)/nts);
RMSEVI(1,:)=ones(1,20)*sqrt((y'*y)/nts);
RMSEVEI(1,:)=ones(1,20)*sqrt((y'*y)/nts);
RMSEVCI(1,:)=ones(1,20)*sqrt((y'*y)/nts);
RMSEVSV(1,:)=sqrt((y'*y)/900);
rangesb=[-10 -5 -2 -1 0];
rangest=[0 1 2 5 10];
C=[2^0 2^1 2^2 2^3 2^4 2^5 2^6 2^7 2^8 2^9 2^10];
G=[0.1 0.5 1 1.5 2];
GE=linspace(0.1,3,25);
for j=1:100
    t=mod(j-1,5)+1;
    v=ceil(j/5);
    nodes=j;
    tic;
    [cent, weights, radius, mseTr]=NME_ELM3(trainx,targets,nodes);
    time1(j)=toc
    % E(3)=mseTr(nodes+1)
    figure(3);clf;plot(0:nodes,mseTr);
    O=calc_ELM_Out1(in,weights,cent,radius,nodes);
    er=y-O;SSEV=er'*er;RMSEVN(j+1)=sqrt(SSEV/nts)
    x1=reshape(in(:,1),30,30);y1=reshape(in(:,2),30,30);z1=reshape(O,30,30);
    figure(4);clf;surf(x1,y1,z1);
    wRange=[rangesb(t),rangest(v)];bRange=wRange;
    for i=1:20
       %Run Original ELM
        %train
        [inw outw bias O error]=ELMR(trainx,targets,2,GE(j),nodes);
        %verify
        O=calc_ELM_Out(tstIn,outw,inw,bias,nodes);
        er=y-O;SSEV=er'*er;RMSEVE(j+1,i)=sqrt(SSEV/nts);
       %run I-ELMS
        %train
        tic;
        [cent, weights, radius, mseTr]=I_ELM(trIn,trOut,nodes,GE(j),2);
        timeE(j,i)=toc;
        %verify
        O=calc_ELM_Out(tstIn,weights,cent,radius,nodes);
        ver=tstOut-O;SSEV=ver'*ver;RMSEVI(j+1,i)=sqrt(SSEV/nts);
        %train
        tic;
        [cent, weights, radius, mseTr]=CI_ELM(trIn,trOut,nodes,GE(j),2);
        timeCI(j,i)=toc;
        %verify
        O=calc_ELM_Out(tstIn,weights,cent,radius,nodes);
        ver=tstOut-O;SSEV=ver'*ver;RMSEVCI(j+1,i)=sqrt(SSEV/nts);
        %train
        tic;
        [cent, weights, radius, mseTr]=EI_ELM(trIn,trOut,nodes,GE(j),2);
        timeEI(j,i)=toc;
        %verify
        O=calc_ELM_Out(tstIn,weights,cent,radius,nodes);
        ver=tstOut-O;SSEV=ver'*ver;RMSEVEI(j+1,i)=sqrt(SSEV/nts);


        tic;
           [weights_output, widths, weights_input,
centers]=ISO_RBF(nodes,trIn,trOut,stop,maximum_iteration)
        Time(4,i)=toc;
        O=verification(weights_input, weights_output, widths, centers, tstIn)
        ver=tstOut-O;SSEV=ver'*ver;RMSEV4(k)=sqrt(SSEV/nts);

    end

        tic;
           [weights_output, widths, cebters,
weights_input]=ErrCor(nodes,trIn,trOut,stop,maximum_iteration)
        Time(5,i)=toc;
        O=verification(weights_input, weights_output, widths, centers, tstIn)
        ver=tstOut-O;SSEV=ver'*ver;RMSEV5(k)=sqrt(SSEV/nts);

    %      run SVR
    d=data(trIn,trOut);
    tic;
    [tr,a]=train(svr({kernel('rbf',0.5),'optimizer = "libsvm"',['C='
num2str(C(j))],'epsilon=0.1'}),d);
    timeSV(i)=toc;
    svs(j) = size(a.Xsv.X,1);
    sseTr = sum((trOut-tr.X).^2);
    mseTr = sseTr/trSize;
    rmseTr(i) = sqrt(mseTr);
    %test the svm
    d = data(tstIn);
    cost=test(a,d);
    sseTst = sum((tstOut-cost.X).^2);
    mseTst = sseTst/nts;
    rmseTst(j)= sqrt(mseTst);
end
```

```
RMSEVE=mean(RMSEVE,2);
RMSEVI=mean(RMSEVI,2);
RMSEVEI=mean(RMSEVEI,2);
RMSEVCI=mean(RMSEVCI,2);
RMSEVSV=[RMSEVSV;rmseTst'];
figure(1);clf;
plot(0:j,RMSEVN,'LineWidth',1.5);hold all;
plot(0:j,RMSEVE,'LineWidth',1.5);plot(0:j,RMSEVI,'LineWidth',1.5);plot(0:j,RMSEVEI,'LineWidth',1.
5);plot(0:j,RMSEVCI,'LineWidth',1.5);
plot(0:j,RMSEVSV,'LineWidth',1.5);
[sm,I1]=min(RMSEVE)
t=mod(I1-2,5)+1
v=ceil((I1-1)/5)
[sm,I2]=min(RMSEVI)
t=mod(I2-1,5)+1
v=ceil(I2/5)
[sm,I3]=min(RMSEVEI)
t=mod(I3-1,5)+1
v=ceil(I3/5)
[sm,I4]=min(RMSEVCI)
t=mod(I4-1,5)+1
v=ceil(I4/5)
[sm,I5]=min(RMSEVSV)
t=mod(I5-2,5)+1
v=ceil((I5-1)/5)
```

Building Blocks:

Calculate_gradient

```
%-------------------------------------------------------------------------
%*                          METHOD - calculate_gradient
%-------------------------------------------------------------------------
%* Method calculates the gradients for an RBF network and returns the
%* gradient and quasi-hessian.
%*
%* INPUTS:
%*  ww - the input weights
%*  weights - the output weights of the network
%*  widths - the widths of the neurons in the network
%*  centers - the centers of the neurons in the network
%*  inputs - the training inputs
%*  outputs - the desired outputs
%*
%* OUTPUTS:
%*  gradient - the gradient matrix used for parameter updates
%*  hessian - the quasi-hessian matrix used for second order methods
function [gradient, hessian] = calculate_gradient(ww, weights, widths, centers, inputs, outputs)
[p1,p2] = size(weights);
[p3,p4] = size(centers);
[p5,p6] = size(widths);
[p7,p8] = size(ww);
g_weight = zeros(p1,p2);
g_center = zeros(p3,p4);
g_width = zeros(p5,p6);
g_ww = zeros(p7,p8);
gradient = zeros(1,p1*p2+p3*p4+p5*p6+p7*p8);
hessian = zeros(p1*p2+p3*p4+p5*p6+p7*p8,p1*p2+p3*p4+p5*p6+p7*p8);
%     gradient = zeros(1,p1*p2+p3*p4+p7*p8);
%     hessian = zeros(p1*p2+p3*p4+p7*p8,p1*p2+p3*p4+p7*p8);
[m,n] = size(inputs);
for i = 1:m
    net = weights(1);
    for j = 1:p3
        node(j) = exp(-sum((ww(j,:).*inputs(i,:)-centers(j,:)).^2)/widths(j));
        net = net + node(j)*weights(j+1);
    end;
    % for g_weight
    out = net;
    de = 1;
    err = outputs(i,1) - out;
    J_weight(1) = -de;
    for j = 2:p2
        J_weight(j) = J_weight(1)*node(j-1);
    end;
    % for g_center
    for j = 1:p3
        J_center(j,:) = (-1)*weights(j+1)*node(j)*2*(ww(j,:).*inputs(i,:)-
centers(j,:))./widths(j);
        J_width(j) = (-1)*weights(j+1)*node(j)*sum((ww(j,:).*inputs(i,:)-
centers(j,:)).^2)/widths(j)^2;
        for k = 1:n
            J_ww(j,k) = (-1)*weights(j+1)*node(j)*(-1)/widths(j)*2*(ww(j,k)*inputs(i,k)-
centers(j,k))*inputs(i,k);
        end;
    end;
    J = parameter_combination(J_weight, J_width, J_ww, J_center);
    gradient = gradient + err*J;
    hessian = hessian + J'*J;
end;
```

Calculate_SSE

```
%-------------------------------------------------------------------------
%*                         METHOD - calculate_SSE
%-------------------------------------------------------------------------
%* Method calculates the SSE for a network
%*
%* INPUTS:
%*  ww - the input weights
%*  weights - the output weights of the network
%*  widths - the widths of the neurons in the network
%*  centers - the centers of the neurons in the network
%*  inputs - the training inputs
%*  outputs - the desired outputs
%*  eps - the parameter for error forgiveness
%*
%* OUTPUTS:
%*  SSE - the SSE for the network
function [SSE] = calculate_SSE(ww, weights,widths,centers,inputs,outputs,eps)
[m,n] = size(inputs);
[p,q] = size(centers);
SSE = 0;
for i = 1:m
    count = weights(1);
    for j = 1:p
        count = count + weights(j+1)*exp(-sum((ww(j,:).*inputs(i,:)-centers(j,:)).^2)/widths(j));
    end;
        SSE = SSE + (count - outputs(i,1))^2;
end;
```

Calc_ELM_Out1

```
% ******************************************************************
% Method: calc_ELM_Out1 - calculates output of any SLFN RBF network

% --------OUTPUTS-------
% output = the predicted output of the algorithm

% -------INPUTS--------
% inputs = the inputs to the datasat you would like to predict
% weights = the output weights produced by the training algorithm
% centers = the centers generated by the training algorithm
% radii = the random radii generated by the training algorithm
% nodes = the number of nodes used to train
%*************Copyright - Dr. Philip Reiner - 2015**********************

function [output]=calc_ELM_Out1(inputs,weights,centers,radii,nodes)
[ni,nd]=size(inputs);
for i=1:ni
    for j=1:nodes
```

```
        h(j) = weights(j).*exp(-1*(1/radii(j)^2)*norm(inputs(i,:)-centers(j,:))^2);
    end
    output(i) = sum(h);
end
output = output';
end
```

CI-ELM

```
% ************************ Method - CI_ELM ************************
% This function is the CI_ELM function that randomly generates centers
% Radius for RBF functions and then weights the output of the functions.

% --------OUTPUTS-------
% centers = the generated centers for all of the nodes
% weights = the output weights generated for the nodes
% radii = the radii of all the rbf nodes
% rmse = the root mean square error after each generated node

% -------INPUTS--------
% inputs = the inputs to the datasat you would like to train
% outputs = the training outputs corresponding to the inputs
% nodes = the maximum number of nodes you would like your net to have
% 1/G = the maximum value allowed for your radius
% range = the maximum range of the inputs
%*************Copyright - Dr. Philip Reiner - 2015**********************

function [centers, weights, radii, mseTr]=CI_ELM(inputs,outputs,nodes,G,range)
[ni,nd] = size(inputs);
Er = outputs';
Do = outputs';
n=0;
mse = sum(Er.^2)/ni;
mseTr(1)=mse;
while (n < nodes )&&(mse > 10e-5)
    n = n+1;
    %assign random center and impact factor
    centers(n,:) = range.*rand(1,nd)-range/2;
    radii(n) = G*rand(1);
    %calculate output weights
    %   weight B(N) = sum(error(p)*H(p))/sum(H(p)^2) for all p
    for i=1:ni,
        H(i) = exp(-1*radii(n)*norm(inputs(i,:)-centers(n,:))^2);
    end
   Beta = (Er*(Er-(Do-H))')/((Er-(Do-H))*(Er-(Do-H))');
            if n > 1
                weights = (1-Beta).*weights;
            end
            weights(n) = Beta;
            Er = (1-weights(n)).*Er+weights(n).*(Do-H);       % remaining error surface
            mse = sum(Er.^2)/ni;
    mseTr(n+1) = mse;
    if sum(H)==0
        n=n-1;
    end
end
end
```

EffELM

```
%% Original ELM (not incremental ELM)
% Inputs ********************
%  x are the training vectors
%  y are the targets
%  wRange is a 1x2 matrix containing the lower and upper bounds for the
%  range of the input weights
%  bRange is the same as wRange but pertaining to the input bias
%  nodes is the number of nodes in the network
%  output = sum of outw*g(inw*x+bias)
% Outputs ********************
%  inw - input weights
%  outw - output weights
%  bias - bias parameters
%  outputs - network outputs
%  error - errors
%*************Copyright - Dr. Philip Reiner - 2015**********************

function [inw outw bias outputs error]=EffELM(x,y,wRange,bRange,nodes)
[np,nd]=size(x);
inw=(wRange(2)-wRange(1))*rand(nodes,nd)+wRange(1);
bias=(bRange(2)-bRange(1))*rand(nodes,1)+bRange(1);
%Sort By affine Transformations
tempx=x;
for j=1:nd
    w(1,j)=1/max(abs(x(1:nodes,j)));
    tempx(1:nodes,j)=w(1,j)*tempx(1:nodes,j);
    for i=1:nodes-1
        tempy(i,j)=abs(tempx(i+1,j)-tempx(i,j));
    end
    del=log10(nd)+log10(2);
    n(j)=-log10(min(tempy(:,j)))+del;
    w(2,j)=w(1,j)*10*sum(n);
end
inw=w(2,:);
M=1; xo=0; alp=max([sqrt(abs(2*log(nodes))),1])+1;
dist=max([alp-xo,alp+xo]);
for i=2:nodes-1
    k=(2*dist)/(min([inw(1,:)*tempx(i+1,:)'-inw(1,:)*tempx(i,:)',inw(1,:)*tempx(i,:)'-
inw(1,:)*tempx(i-1,:)']));
    inw(i,:)=k*inw(1,:);
end
inw(1,:)=inw(2,:); inw(nodes,:)=inw(nodes-1,:);
for i=1:nodes
    bias(i)=xo-inw(i,:)*tempx(i,:)';
end

for i=1:nodes
    for j=1:np
        H(j,i)=1/(1+exp(-(inw(i,:)*tempx(j,:)'+bias(i))));
    end
end
%Calculate Moore-Penrose generalized inverse of H
Ht=pinv(H);
%Calculate output weights
outw=Ht*y;
outputs=(outw'*H')';
error=y-outputs;
```

EI_ELM

```
% *****************************************************************
% This function is the EI_ELM function that randomly generates centers
% Radius for RBF functions and then weights the output of the functions.

% --------OUTPUTS-------
% centers = the generated centers for all of the nodes
% weights = the output weights generated for the nodes
% radii = the radii of all the rbf nodes
% rmse = the root mean square error after each generated node

% -------INPUTS--------
% inputs = the inputs to the datasat you would like to train
% outputs = the training outputs corresponding to the inputs
% nodes = the maximum number of nodes you would like your net to have
% 1/G = the maximum value allowed for your radius
% range = the maximum range of the inputs
%*************Copyright - Dr. Philip Reiner - 2015**********************
function[centers, weights, radii, mseTr] = EI_ELM(inputs, outputs, nodes,G,range)
[ni,nd] = size(inputs);
Er = outputs';
n = 0;
```

```
mse = sum(Er.^2)/ni;
mseTr(1) = mse;
while (n < nodes )&&(mse > 10e-5)
        n = n+1;
        for z=1:10
            %assign random center and impact factor
            a(z,:) = range.*rand(1,nd)-range/2;
            b(z) = G*rand(1);
            %calculate output weights
            %   weight B(N) = sum(error(p)*H(p))/sum(H(p)^2) for all p
            for i=1:ni,
                H(i) = exp(-1*b(z)*norm(inputs(i,:)-a(z,:))^2);
            end
            B(z) = (Er*H')/(H*H');
            O = B(z).*H;
            IEr(z,:) = Er-O;       % remaining error surface
            imse(z) = sum(Er.^2)/ni;
        end
        %choose best node
        [Y,I]=sort(imse);
        Er = IEr(I(1),:);
        centers(n,:) = a(I(1),:);
        radii(n) = b(I(1));
        weights(n) = B(I(1));
        mse = Y(1);
        mseTr(n+1) = mse;
end
 end
```

ErrCor

```
%-------------------------------------------------------------------------
%*                         METHOD - ErrCor
%-------------------------------------------------------------------------
%* Method constructs and trains an RBF network incrementally
%*
%* INPUTS:
%*  nodes - the number of neurons in the network
%*  inputs - training inputs
%*  outputs - training outputs
%*  stop - error stopping criterion
%*  maximum_iteration - loop stopping criterion
%*
%* OUTPUTS:
%*  weights_output - trained output weights of the network
%*  widths - trained widths of the RBF neurons
%*  weights_input - trained input weights of the network
%*  centers - trained network centers

function [weights_output, widths, centers, weights_input] =
ErrCor(nodes,inputs,outputs,stop,maximum_iteration)
 Nmax=nodes;
[m,n] = size(inputs);
[np,nd]=size(inputs);

actual_output_ = zeros(size(outputs));
centers = [];
weights_input = [];
weights_output = 1;
widths = [];
number_of_hidden_unit = 0;
 tic;
 g=1;
    maximum_error = stop;
    mu = 100;
for kkk = 1:Nmax,
    [maxi_, index_1] = max(abs(outputs-actual_output_));
    number_of_hidden_unit = number_of_hidden_unit + 1;
    centers = [centers; inputs(index_1,:)];
    weights_input = [weights_input; ones(1,n)];
    weights_output = [weights_output, 1];
    widths = [widths, 1];
    para_cur = parameter_combination(weights_output, widths, weights_input, centers);
    % para_cur = weights_output;
    I = eye(length(para_cur));
    % other parameters

    % training process
    [SSE(1)] = calculate_SSE(weights_input, weights_output,widths,centers,inputs,outputs);
    SSE2(g)=SSE(1);
    g=g+1;
    fprintf('Number of RBF units = %d, iteration = 1, SSE = %6.10f\n',kkk,SSE(1));
    for iter = 2:maximum_iteration
        jw = 0;
        [gradient, hessian] = calculate_gradient(weights_input, weights_output, widths, centers, 
inputs, outputs );
        para_back = para_cur;
        while 1
            para_cur = para_back - (inv(hessian+mu*I)*gradient')';
            [weights_output, widths, wieghts_input, centers] = 
parameter_divison(para_cur,number_of_hidden_unit,inputs);
            [SSE(iter)] = calculate_SSE(weights_input, 
weights_output,widths,centers,inputs,outputs);
            SSE2(g)=SSE(iter);
            g=g+1;
            if SSE(iter) <= SSE(iter-1)
                if mu > 10^-20
                    mu = mu/10;
                end;
                break;
            end;
            if mu < 10^20
                mu = mu*10;
            end;
            jw = jw + 1;
            if jw > 20
                break;
            end;
        end;

    end;

end;
```

evalEr

```
%% Error Evaluation (SSE)
% inputs
%   data - two column set where the first column is desired outputs and the
%          second column is the current output
% outputs
%   Er - the sum squared error Er = sum((data(:,1)-data(:,2)).^2)
%**************Copyright - Dr. Philip Reiner - 2015***********************
function SSE=evalEr(rad)
global inputs prevOut newCen w desired H;
H=funct(inputs,rad,newCen);
Er=desired-prevOut;
w=(H'*Er)/(H'*H);
Er=Er-w*H;
SSE=Er'*Er;
```

funct

```
%*********************** Method - funct ***********************************
%*  method calculates the output of a RBF unit
%* INPUTS:
%*  x - input data
%*  radi - width of RBF unit
%*  xo - center of RBF unit
%* OUTPUTS:
%*  y - output for each data input
```

```
%*************Copyright - Dr. Philip Reiner - 2015**********************

function y=funct(x,radi,xo)
[m,n]=size(x);
for q=1:m
    y(q)=exp(-sum((x(q,:)-xo).^2,2)/radi^2);
end
y=y';
return;
```

I-ELM

```
% *****************************************************************
% This function is the I_ELM function that randomly generates centers
% Radius for RBF functions and then weights the output of the functions.

% --------OUTPUTS-------
% centers = the generated centers for all of the nodes
% weights = the output weights generated for the nodes
% radii = the radii of all the rbf nodes
% rmse = the root mean square error after each generated node

% -------INPUTS--------
% inputs = the inputs to the datasat you would like to train
% outputs = the training outputs corresponding to the inputs
% nodes = the maximum number of nodes you would like your net to have
% 1/G = the maximum value allowed for your radius
% range = the maximum range of the inputs
%*************Copyright - Dr. Philip Reiner - 2015**********************

function [centers, weights, radii, mseTr]=I_ELM(inputs,outputs,nodes,G,range)
[ni,nd] = size(inputs);
Er = outputs';
n=0;
mse = sum(Er.^2)/ni;
mseTr(1) = mse;
while (n < nodes )&&(mse > 10e-5)
    n = n+1;
    %assign random center and impact factor
    centers(n,:) = range.*rand(1,nd)-range/2;
    radii(n) = G*rand(1);
    %calculate output weights
    %   weight B(N) = sum(error(p)*H(p))/sum(H(p)^2) for all p
    for i=1:ni,
        H(i) = exp(-1*radii(n)*norm(inputs(i,:)-centers(n,:))^2);
    end
    weights(n) = (Er*H')/(H*H');
    O = weights(n).*H;
%     figure(1); clf; mesh(reshape(O,30,30));
    Er = Er-O;       % remaining error surface
%     figure(2); clf; mesh(reshape(Er,30,30));
    mse = sum(Er.^2)/ni;
    mseTr(n+1) = mse;
    if sum(H)==0
        n=n-1;
    end
end
end
```

ISO_RBF

```
%-------------------------------------------------------------------------
%*                         METHOD - ISO_RBF
%-------------------------------------------------------------------------
%* Method creates a randomly initialized RBF network, and then trains it
%* using the Improved Second order training method.
%*
%* INPUTS:
%*  nodes - the number of neurons in the network
%*  trIn - training inputs
%*  trOut - training outputs
%*  stop - error stopping criterion
%*  maximum_iteration - loop stopping criterion
%*
%* OUTPUTS:
%*  weights_output - trained output weights of the network
%*  widths - trained widths of the RBF neurons
%*  weights_input - trained input weights of the network
%*  centers - trained network centers

function [weights_output, widths, weights_input,
centers]=ISO_RBF(nodes,trIn,trOut,stop,maximum_iteration)
 %% set the number of RBF units
    number_of_hidden_unit = nodes;
    %% initial parameter generation
    weights_input = rand(nodes,size(trIn,2)) ;
    weights_output = rand(nodes+1,size(trOut,2));
    widths = rand(nodes) ;
    centers = rand(nodes,size(trIn,2));
    binputs=trIn;
    boutputs=trOut;
    %% Run algorithm on the entire data set as a control
    [m,n] = size(binputs);
    %% combination of parameters
    para_cur = parameter_combination(weights_output, widths, weights_input, centers);
    %% other parameters
    I = eye(length(para_cur));
%     maximum_iteration = 30;
    maximum_error = stop;
    mu = 1;
    %% training process
    [SSE(1)] = calculate_SSE(weights_input, weights_output,widths,centers,binputs,boutputs,eps);
    RMSE(1)=sqrt(SSE(1)/m);
    fprintf('iteration = 1, SSE = %6.10f\n',SSE(1));
    tic
    for iter = 2:maximum_iteration
        jw = 0;
        [gradient, hessian] = calculate_gradient(weights_input, weights_output, widths, centers,
binputs, boutputs );
        para_back = para_cur;
        while 1
            para_cur = para_back - ((hessian+mu*I)\gradient')';
            del(iter-1,:)=((hessian+mu*I)\gradient')';
            [weights_output, widths, weights_input, centers] =
parameter_divison(para_cur,number_of_hidden_unit,binputs);
            [SSE(iter)] = calculate_SSE(weights_input,
weights_output,widths,centers,binputs,boutputs,eps);
            if SSE(iter) <= SSE(iter-1)
                if mu > 10^-20;
                    mu = mu/10;
                end;
                break;
            end;
            if mu < 10^20
                mu = mu*10;
            end;
            jw = jw + 1;
            if jw > 10
                break;
            end;
        end;
        RMSE(iter)=sqrt(SSE(iter)/m);
        fprintf('iteration = %d, RMSE = %6.10f\n',iter, RMSE(iter));
        if SSE(iter) < maximum_error
            break;
        end;
    end;
```

nelder_mead_ndmd2

```
function [f_BEST,BEST]=nelder_mead_ndmd2(obj,x0,d_SIM,df_min,ite_max,times)
```

```
%   INPUT ARGUMENTS:
%   nelder_mead_ndmd2(@testf1,[100,100],1,1e-4,2e2,100)
%   obj         - Handle of objective function.
%   x0          - Initial starting point.
%   d_SIM    - Size of initial simplex.
%   df_min   - Minimum improvement required for termination.
%   ite_max - Desired number of iterations.

%  OUTPUT ARGUMENTS:
%   BEST     - Location of baest solution.
%   f_BEST   - Best value of the objective found.
%   SIMPLEX - Matrix conatining final simplex.
%   f        - Objective values for each point in the simplex.

format long;
tavg_ite=0;
tsecond=0;
second=0;
succ_time=0;
avg_ite=0;
avg_time=0;
avg_error=0;
average_min=0;
% Initialize parameters and create simplex
for itee=1:times,   %training timesa=1;
    tic;
    alpha=1;
    a=1;
    b=2;
    c=0.5;
    n=length(x0);
    mo=zeros(1,n);
    mu=0.1;
    X0=ones(n,1)*x0;
    SIMPLEX=[X0+diag(d_SIM*(rand(1,n)));x0]; % create simplex vertices
    f(n+1)=0;
    f_mid(n)=0;
    mid=zeros(n);

    for init=1:n+1
        f(init)=feval(obj,SIMPLEX(init,:));
    end
    init=0;
    SIMPLEX(:,end+1)=f';
    SIMPLEX=sortrows(SIMPLEX,n+1);    %sort row depending of value of f in ascending order;

    f=SIMPLEX(:,end)';
    SIMPLEX(:,end)=[];

    % Simplex Code
    for ite=1:ite_max,

         Pb=sum(SIMPLEX(1:n,:))/n;       %calculate the centroid P_ of points with i#h
         Ps=(1+a)*Pb-a*SIMPLEX(end,:);    %calculate reflection point of Ph:Ps
         f_Ps=feval(obj,Ps);
         Pss=(1-b)*Pb+b*Ps;    %calculate P** by expansion
         f_Pss=feval(obj,Pss);

         if f_Ps>f(1)
             % using hyper plane equation
             I=SIMPLEX(:,1:n);
             A=ones(1,n+1)';
             A(:,2:n+1)=I;
             B=f';
             P=pinv(A)*B;
             grad=P';
             Gs=SIMPLEX(1,:)-alpha*grad(1,2:n+1);
             % Calculate reflected point
             P3=(1+a)*SIMPLEX(1,:)-SIMPLEX(end,:);
             P1=SIMPLEX(1,:);
             P2=Gs;
```

```
              PP=(P3-P1).*(P2-P1);
              u=sum(PP)/sum((P2-P1).^2);
              Gs=P1+u*(P2-P1);
              f_Gs=feval(obj,Gs);

              if f_Gs<f_Ps
                  Ps=Gs;             %new reflected point
                  f_Ps=f_Gs;
                  Pb=SIMPLEX(1,:);
                  Pss=(1-b)*SIMPLEX(1,:)+b*Ps;   %calculate P** by expansion
                  f_Pss=feval(obj,Pss);
              end

          end

          if f_Ps<f(1)    %f(P*)<f(l)
                if f_Pss<f(1)    %f(P**)<f(l)
                    SIMPLEX(end,:)=Pss;      %replace Ph by P**
                    f(end)=f_Pss;
                else
                   SIMPLEX(end,:)=Ps;        %replace Ph by P*
                   f(end)=f_Ps;
                end
          else
              check=0;
              for i=1:n,
                  if f_Ps>f(i)      % f_P*>f_i and i#h
                      check=1;
                      break;
                  end
              end
              if check==0
                  SIMPLEX(end,:)=Ps;        %replace Ph by P*
                  f(end)=f_Ps;
              else
                  if f_Ps>f(end)    %f_P*>f_h
                      Pss=c*SIMPLEX(end,:)+(1-c)*Pb; %calculate P** by expansion
                      f_Pss=feval(obj,Pss);
                      if f_Pss>f(end)    %f(P**)>f(h)
                          for i=1:n+1
                             SIMPLEX(i,:)=(SIMPLEX(i,:)+SIMPLEX(1,:))/2; %replace all Pi' by
(Pi+Pl)/2

                             f(i)=feval(obj,SIMPLEX(i,:));
                          end
                      else
                         SIMPLEX(end,:)=Pss;      %replace Ph by P**
                         f(end)=f_Pss;
                      end
                  else
                     SIMPLEX(end,:)=Ps;        %replace Ph by P*
                     f(end)=f_Ps;
                  end
              end
          end

        % reorder and display iteration output
        SIMPLEX(:,end+1)=f';
        SIMPLEX=sortrows(SIMPLEX,n+1);

        f=SIMPLEX(:,end)';
        SIMPLEX(:,end)=[];
        error(ite)=f(1);
        t(ite)=ite;

        % terminate condition3 for neural network training
        if f(1)<df_min,
            succ_time=succ_time+1;
            avg_ite=avg_ite+ite;
            avg_time=avg_time+1;
            avg_error=avg_error+f(1);
            second=second+toc;
            break;
        end
     end;
    % display the result
    succ_rate=succ_time/times;
    BEST=SIMPLEX(1,:);
    f_BEST=f(1);
%     average_min=average_min+f_BEST;
%     tavg_ite=tavg_ite+ite;
%     tsecond=tsecond+toc;
%     disp(' ');
%     disp(['Minimum value of f = ',num2str(f_BEST),])
%     disp(['located at x = [',num2str(BEST),'].'])
%     disp(['Success rate = [',num2str(succ_rate),'].'])
%     % plot
%     semilogy(t,error,'b');
%     xlabel('Iterations')
%     ylabel('Error')
%     hold on;
end
%     avg_iteration=avg_ite/avg_time;
%     avg_errors=avg_error/avg_time;
%     avg_second=second/avg_time;
%     tavg_iteration=tavg_ite/times;
%     avg_minimum=average_min/times;
%     avg_tsecond=tsecond/times;
%     disp(['Average Iteration = ',num2str(avg_iteration),])
%     disp(['Average Error = ',num2str(avg_errors),])
%     disp(['Average second = ',num2str(avg_second),])
%     disp(['tAverage Iteration = ',num2str(tavg_iteration),])
%     disp(['tAverage Minimum = ',num2str(avg_minimum),])
%     disp(['tAverage second = ',num2str(avg_tsecond),])
return
```

NME-ELM

```
%************************** METHOD - NME-ELM ******************************
%* INPUTS:
%*  in - the input pairs for training NxD
%*  outputs - training target values Nxm
%*  Nodes - the maximum number of neurons
%*  eps - the error criterion
%* OUTPUTS:
%*  cent - the resulting network centers N~xD
%*  weights - the resulting network output weights
%*  radius - the widths of the resulting network
%*  mseTr - the mean squared error for each added neuron
%*************Copyright - Dr. Philip Reiner - 2015***********************

function [cent, weights, radius, mseTr]=NME_ELM3(in,outputs,Nodes,eps)
%alpha, beta, and gamma are simplex parameters
% figure(6);clf;
global inputs prevOut newCen w desired np nd H;
inputs=in;
[np,nd]=size(inputs);
desired = outputs;
prevOut=zeros(np,1);
Er=outputs;
mseTr(1)=(Er'*Er)/np;
j=1;
while j<=Nodes
    %% Initialize each node
    [big I]=max(abs(Er));
    cent(j,:)=inputs(I(1),:);
    newCen=cent(j,:);
    % find the radiusssssssssssssssssssssssss
    figure(1);clf;
    [f_BEST,BEST]=nelder_mead_ndmd2(@evalEr,1,4,1e-5,10,1);
    %% Calculate Final weight
    weights(j)=w;
    Er=Er-weights(j)*H;
    radius(j)=BEST;
    mseTr(j+1)=(Er'*Er)/np;
    prevOut=prevOut+w*H;
    %     x1=reshape(in(:,1),30,30);y1=reshape(in(:,2),30,30);z1=reshape(prevOut,30,30);
    %     figure(4);clf;surf(x1,y1,z1);
    % %     title('Desired Curve');xlabel('x');ylabel('y');
    %     plot(inputs,w(j)*H(:,j),'r','LineWidth',2.5);
    %     legend('Desired','C1','NME-ELM Out');
    j=j+1;
end
```

Parameter_combination

```
%-------------------------------------------------------------------------
%*                         METHOD - parameter_combination
%-------------------------------------------------------------------------
%* Method calculates the SSE for a network
%*
%* INPUTS:
%*  weights_input - the input weights
%*  weights_output - the output weights of the network
%*  widths - the widths of the neurons in the network
%*  centers - the centers of the neurons in the network
%*
%* OUTPUTS:
%*  vector - a vector of all the network parameters in a single row
function [vector] = parameter_combination(weights_output, widths, weights_input, centers)
[p1,p2] = size(weights_input);
[p3,p4] = size(centers);
vector = [weights_output widths reshape(weights_input',1,p1*p2) reshape(centers',1,p3*p4)];
```

Parameter_division

```
%-------------------------------------------------------------------------
%*                         METHOD - parameter_division
%-------------------------------------------------------------------------
%* Method calculates the SSE for a network
%*
%* INPUTS:
%*  vector - a vector of all the network parameters in a single row
%*  num - number of nodes in the network
%*  data - the input data for training
%*
%* OUTPUTS:
%*  weights_input - the input weights
%*  weights_output - the output weights of the network
%*  widths - the widths of the neurons in the network
%*  centers - the centers of the neurons in the network

function [weights_output, widths, weights_input, centers] = parameter_divison(vector, num, data)
[row, col] = size(data);
for i = 1:(num+1)
    weights_output(1,i) = vector(1,i);
end;
for i = 1:num
    widths(1,i) = vector(1, num+1+i);
end;
for i = 1:num
    for j = 1:col
        weights_input(i,j) = vector(1,2*num+1+(i-1)*col+j);
        %             weights_input(i,j) = vector(1,num+1+(i-1)*col+j);
    end;
end;
for i = 1:num
    for j = 1:col
        centers(i,j) = vector(1,2*num+1+num*col+(i-1)*col+j);
        %             centers(i,j) = vector(1,num+1+num*col+(i-1)*col+j);
    end;
end;
```

verification

```
%------------------------------------------------------------------------
%*                         METHOD - Verification
%------------------------------------------------------------------------
%* Method calculates outputs generated by a network
%*
%* INPUTS:
%*  weights_input - the input weights
%*  weights_output - the output weights of the network
%*  widths - the widths of the neurons in the network
%*  centers - the centers of the neurons in the network
%*  testing_input - the test data for the network to process
%*
%* OUTPUTS:
%*  output - network outputs for each entry in the testing inputs

function [output] = verification(weights_input, weights_output, widths, centers, testing_input)
%% verification process
[m,n] = size(testing_input);
[p,q] = size(centers);
for i = 1:m
    count = weights_output(1);
    for j = 1:p
        count = count + weights_output(j+1)*exp(-sum((weights_input(j,:).*testing_input(i,:)-
centers(j,:)).^2)/widths(j));
    end;
    output(i,1) = count;
end;
```