Training Arbitrarily Connected Neural Networks

with Second Order Algorithms

Except where reference is made to the work of others, the work described in this thesis is my own or was done in collaboration with my advisory committee. This thesis does not include proprietary or classified information.

Nicholas Jay Cotton

Certificate of Approval:

Thaddeus A. Roppel
Associate Professor
Electrical and Computer Engineering

Bogdan M. Wilamowski, Chair
Professor
Electrical and Computer Engineering

John Y. Hung
Professor
Electrical and Computer Engineering

George T. Flowers
Interim Dean, Graduate School

Training Arbitrarily Connected Neural Networks with Second Order Algorithms

Nicholas Jay Cotton

A Thesis

Submitted to

the Graduate Faculty of

Auburn University

in Partial Fulfillment of the

Requirements for the

Degree of

Master of Science

Auburn, Alabama
August 9, 2008

Training Arbitrarily Connected Neural Networks

with Second Order Algorithms

Nicholas Jay Cotton

_____

Signature of Author

_____

Date of Graduation

Thesis Abstract

Training Arbitrarily Connected Neural Networks

with Second Order Algorithms

Nicholas Jay Cotton

Master of Science, August 9, 2008
(B.E.E., Auburn University, 2006)

68 Typed Pages

Directed by Bogdan M. Wilamowski

Neural networks have been an active area of research and application for many years. Today they are gaining popularity with the growing processing power of modern computers. With neural networks gaining popularity comes a demand for a simple and reliable method of training all types of networks which is the focus of this paper. Neural Network Trainer is a training package that allows the user to create a simple netlist style network architecture in a text file and quickly begin training. Several algorithms are implemented including Error Back Propagation as well modified versions of the Levenberg-Marquardt algorithm. The software is demonstrated with results verifying the implemented algorithms as well as the trained neural networks.

Acknowledgments

This project would not have been possible without the support and guidance of the people around me. I would like to sincerely thank my advisor Dr. Bogdan M. Wilamowski for his guidance and inspiration through my educational career. I would also like to thank the members of my committee Dr. John Y. Hung and Dr. Thaddeus A. Roppel for their feedback. I would also like to thank all of the members of the Auburn University Electrical Engineering Department for their contribution to my overall education. Finally, I would like to thank my wife, parents, and grandparents for their support over the years throughout my education.

Style manual or journal used Journal of Approximation Theory (together with the style known as "aums"). Bibliography follows van Leunen's *A Handbook for Scholars*.

Computer software used The document preparation package TEX (specifically LATEX) together with the departmental style-file `aums.sty`.

Table of Contents

List of Figures

Chapter 1

Introduction

Neural Networks have been used for years in a wide range of Electrical Engineering disciplines. As computers' processing speeds increase software implemented neural networks can be used in many nontraditional places. Traditionally neural networks were very difficult to train and implement. The required processing power by training neural networks was too much to be handled for most computers, but this is not as true any longer. This evolution of computers is also allowing the neural network to evolve in unpredicted ways. This change not only includes the method by which neural networks are trained, but also where they can be implemented. Computers are used everywhere today from controlling production machines in factories to controlling the lights in the factory. Neural networks can be implemented on any of these computers with very little effort. The computation required is trivial for modern computers and the time required to do so is many cases negligible. This ease of implementation can be taken a step further by embedding neural networks.

Neural networks have become a growing area of research over the last few decades and have taken hold in many branches of industry. One example in the field of industrial electronics is motor drives according to [1, 2, 3]. Neural networks are also helping with power distribution problems such as harmonic distortion [4, 5, 6, 7]. These papers show how valuable neural networks are becoming in industry. This thesis offers a method of training fully connected networks to help solve some of these real world

problems. Fully connected networks are much more powerful and easier to train when using the Neural Network Trainer [8, 9]. Cross layer connections reduce the number of neurons by increasing the number of connections and corresponding weights. This in turn reduces the total number of calculations required for the network.

Unfortunately with all of the research being done, neural networks are often overlooked because they are widely misunderstood, difficult to train, or difficult to implement. Chapter 2 discusses pre-existing training algorithms that do not simplify these problems. In Chapter 3, a new training algorithm is presented that is demonstrated to be more effective and faster that those discussed in Chapter 2. Chapters 4 and 5 describe the Neural Network Trainer and how it was created to assist in the training process. Chapter 6 demonstrates the results of the various algorithms and training methods.

Chapter 2

Common Neural Network Training Algorithms

Various methods for neural network training have been developed ranging from random search through gradient-based methods. The best-known gradient method is the Error Back Propagation (EBP) [10, 11, 12]. This method is characterized by very poor convergence. Several improvements for EBP were developed such as the Quickprop algorithm, Resilient Error Back Propagation, Back Percolation, Delta-bar-Delta etc. Much better results can be obtained using second order methods such as Newton or Levenberg Marquet (LM) [12]. In the latter, not only the gradient but also the Hessian or Jacobian should be found. The EBP algorithms propagate errors layer by layer from outputs to the inputs. If a neural network has connections across layers the EBP algorithm becomes more complicated. This added complexity is even greater for second order algorithms such as the LM algorithm for such networks. Another method of calculating gradients is forward calculations, also known as the perturbation method. In this case small changes on neuron net values are introduced and the resultant changes on the outputs are recorded. The perturbation method is especially useful for VLSI implemented network structure, where instead of analytical computation of gradients only various signal gains are measured [13, 14][13, 14]. EBP can work best for feedforward neural networks. Similar to EBP, the LM algorithm was adapted only to feedforward networks organized in layers (without weights across layers).

2.1 Error Back Propagation

This section will demonstrate how EBP performs gradient descent to train neural networks with hidden layers. This method begins by calculating the error at each pattern where the error is defined as:

$$err = d_{out} - out \tag{2.1}$$

where d_{out} is equal to the desired output or the output of the training data and out is equal to the actual output. After the error has been calculated the derivative through the layers in reverse order is needed. The derivative is defined as:

$$f' = gain \cdot (1 - out^2) \tag{2.2}$$

Next the change in weights is calculated by:

$$\Delta = f' \cdot err \tag{2.3}$$

$$\Delta w = \alpha \cdot \Delta \cdot x \tag{2.4}$$

where α is a user defined constant used for adjusting the step size and x is equal to the value at that neuron connection. The new weight is then represented by:

$$w_{new} = \Delta w + w \tag{2.5}$$

This process is repeated for all output neurons and then for each layer starting at the output layer ending at the input layer. This entire process is then repeated for each training pattern. The total error for the network is then calculated using all the patterns and outputs as follows:

$$Total_{err} = \sum_{p=1}^{np} [d_p - o_p]^2 \tag{2.6}$$

This entire process is then repeated thousands of times in hopes of reaching the desired error. EBP will work in many cases but it has several drawbacks. The first being the algorithm is very slow to converge if it converges at all. The algorithm typically approaches the vicinity of the error very quickly but then drastically slows taking considerably longer to get closer to the final solution than when it began. This process continues to slow as it gets closer therefore making it very difficult to get a precise answer. However there are other algorithms that are able to get a much more precise answer very quickly such as Levenberg-Marquardt.

2.2 Levenberg-Marquardt

The Levenberg-Marquardt (LM) algorithm is a second order algorithm that many times is overlooked by those attempting to train neural networks possibly because it is more complex to implement than EBP. However, it definitely makes up for this

in superior performance. LM is similar to EBP in that it requires the calculation of the gradient vector, but in addition, LM also computes the Jacobian. The gradient vector is represented as:

$$g = \begin{pmatrix} \frac{\partial E}{\partial W_1} \\ \frac{\partial E}{\partial W_2} \\ \vdots \\ \frac{\partial E}{\partial W_n} \end{pmatrix} \tag{2.7}$$

where E is there error of the network for that pattern and W refers to the weights. The Jacobian is essentially every gradient for every training pattern and network output. The Jacobian is shown below.

$$J = \begin{bmatrix} \frac{\partial E_{11}}{\partial W_1} & \frac{\partial E_{11}}{\partial W_2} & \cdots & \frac{\partial E_{11}}{\partial W_n} \\ \frac{\partial E_{21}}{\partial W_1} & \frac{\partial E_{21}}{\partial W_2} & \cdots & \frac{\partial E_{21}}{\partial W_n} \\ \vdots & \vdots & & \vdots \\ \frac{\partial E_{M1}}{\partial W_1} & \frac{\partial E_{M1}}{\partial W_2} & \cdots & \frac{\partial E_{M1}}{\partial W_n} \\ \vdots & \vdots & & \vdots \\ \frac{\partial E_{12}}{\partial W_1} & \frac{\partial E_{12}}{\partial W_2} & \cdots & \frac{\partial E_{12}}{\partial W_n} \\ \frac{\partial E_{22}}{\partial W_1} & \frac{\partial E_{22}}{\partial W_2} & \cdots & \frac{\partial E_{22}}{\partial W_n} \\ \vdots & \vdots & & \vdots \\ \frac{\partial E_{MP}}{\partial W_1} & \frac{\partial E_{MP}}{\partial W_2} & \cdots & \frac{\partial E_{MP}}{\partial W_n} \end{bmatrix} \tag{2.8}$$

where N is the number of weights, M is the number of outputs, and P is the number of patterns. In other words the Jacobian will have as many columns as there are weights, and the number of rows will be equal to the product of M and P. Once the Jacobian is calculated, the LM algorithm can be represented by the following:

$$W_{k+1} = W_K - \left(J_k^T J_k + \mu I\right)^{-1} J_k^T E \tag{2.9}$$

where E is the total error for all patterns, I is the identity matrix, and μ is a learning parameter. The learning parameter μ is then adjusted several times each iteration and the result with the greatest reduction of error is selected. When the μ value is very large the LM algorithm becomes steepest decent or EBP, and when μ is equal to zero it is the Newton Method.

The entire process is then repeated until the error is reduced to the required value. This second order algorithm is significantly faster than EBP, as will be discussed in more detail in Chapter 6. However, as successful as the LM algorithm is it does require the inversion of the Jacobian matrix. For small networks with few training patterns this is not a major issue, but for networks with many training patterns it is very computationally intensive. This inversion will cause each training iteration for LM to take longer than an iteration for EBP. As long as there is enough memory to perform the inversion, the time required for training will still be far less than that of EBP, because the LM will require such few iterations.

Chapter 3

Modified Levenberg-Marquardt Algorithm

The LM algorithm was used as the basis for the following algorithms. This second order approach is modified to train any feed-forward architecture. Even though LM is a very powerful algorithm it is prone to stalling in local minima and becoming unable to proceed in the training process. This problem is addressed in the following sections.

3.1 Advantages of Arbitrarily Connected Neural Networks

Comparing FCN (Fully Connected Neurons) networks with MLP networks one may conclude that the latter ones require about twice as many neurons to perform a similar task[8]. For example Figures 3.1 and 3.2 show the minimum architectures required to solve parity-3 and parity-9 problems. It is relatively simple to design neural networks to solve parity-N problems [15]. However, to find a solution by training is much more difficult. For a three layer MLP network to solve a parity-N problem the required number of neurons is [15]:

$$N_{MLP} = N + 1 \tag{3.1}$$

while for three layer FCN networks the minimum number of neurons is :

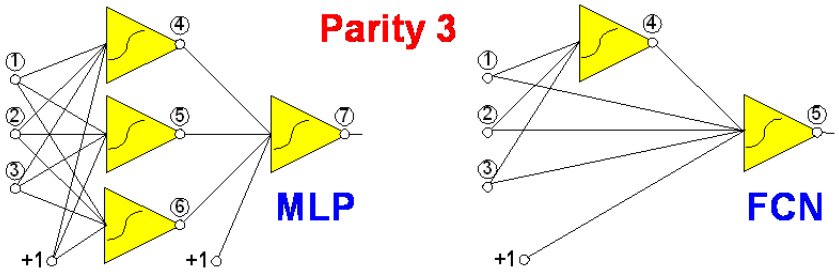

Figure 3.1: Network architectures to solve the parity-3 problem with 3 neurons in the hidden layer (MLP) and a network with 1 neuron in the hidden layer (FCN).

$$N_{FCN} = \begin{cases} \frac{N-1}{2} & \text{For odd number parity problems} \\ \frac{N}{2} & \text{For even number parity problems} \end{cases} \tag{3.2}$$

Another example in which an arbitrarily connected network outperforms a traditionally connected network is in a nonlinear control system. In Figure 3.1 a desired highly nonlinear control surface for two variables is shown. With the 3 layer, 8 neuron MLP network shown in Figure 3.1, it was not possible to reach the desired surface. However, with the 5 neuron FCN architecture 5 neurons shown in Figure 3.1, it was possible to find a satisfactory solution, the control surface obtained being very close to the required surface in Figure 3.1. One may notice that the FCN topology with 5 neurons produces significantly smaller error than the 8 neuron MLP topology with one hidden layer. The MSE (Mean Squared Error) is defined as:

$$MSE = \frac{1}{n_p n_o} \sum_{i=1}^{n_p} \sum_{i=1}^{n_o} e_{ij}^2 \tag{3.3}$$

where:

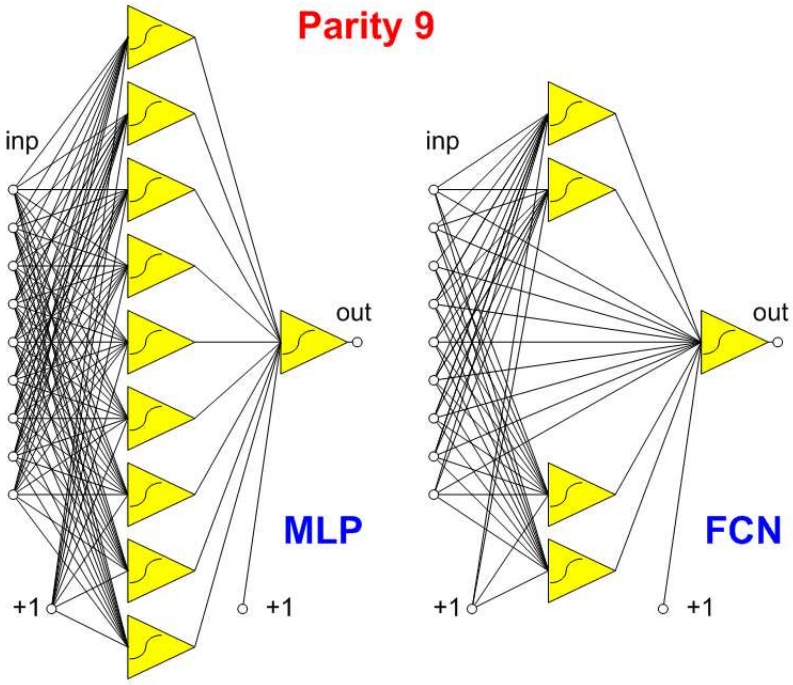

Figure 3.2: Network architectures to solve the parity- 9 problem with a 3 layer MLP network (left) and with a 3 layer FCN network (right.)

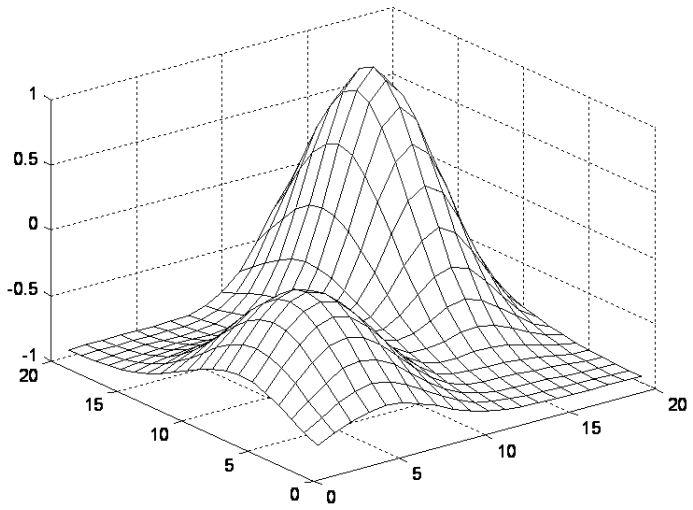

Figure 3.3: The desired nonlinear control surface to which the neural networks are trained.

$$e_{ij} = out_{ij} - dout_{ij} \tag{3.4}$$

dout is desired output, *out*is actual output, $n_p$is number of patterns, and $n_o$is number of outputs.

Comparing 3 layer networks, shown in Figures 3.1 and 3.2, one may also conclude that FCN networks are more transparent than MLP networks. With connections across layers in ACN networks, there are fewer neurons (nonlinear elements)g on the signal paths which results in the learning algorithms converging faster. Unfortunately, most of the neural network learning software, such as the popular MATLAB Neural Network Toolbox, is developed for MLP networks and are not able to handle FCN or ACN networks. It is also much easier to write computer software for regular architectures, organized layer by layer, in comparison to neural networks with arbitrarily

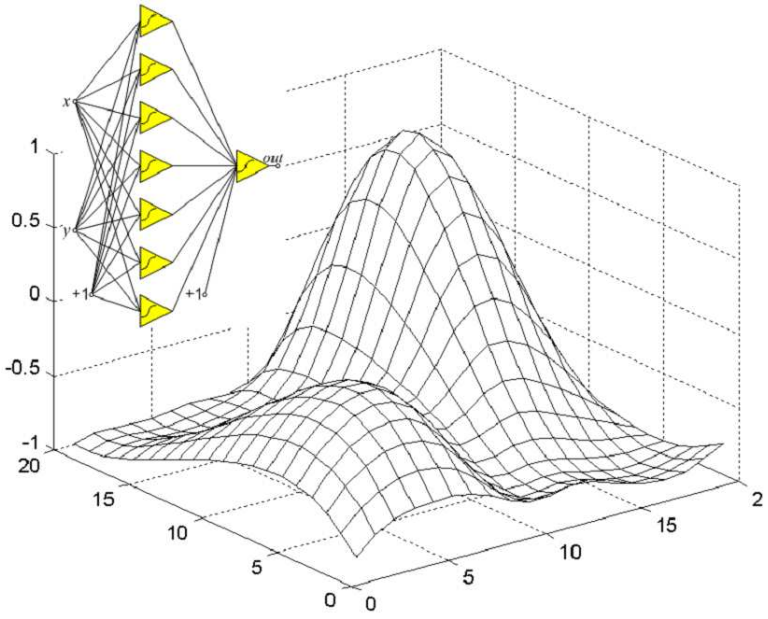

Figure 3.4: Resulting control surface obtained with an MLP architecture having 7 neurons in one hidden layer. The total MSE is 0.00234.

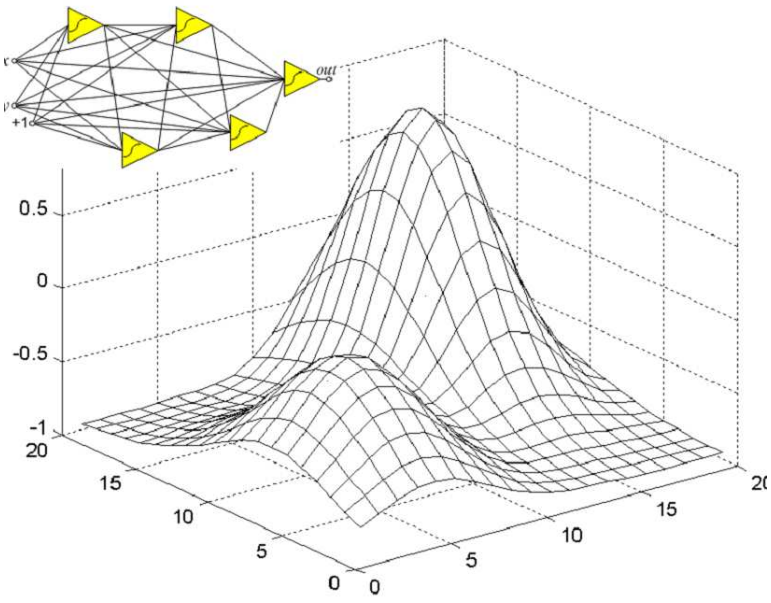

Figure 3.5: Resulting control surface obtained with an FCN architecture having 4 hidden neurons. The total MSE is 0.00014

connected neurons (ACN). Both MLP and FCN networks are of course, a subset of ACN networks.

3.2 Calculation of Gradient and Jacobian

The EBP algorithm requires only the computation of the error gradient. Second order algorithms, such as LM) algorithm or DFP (Davidon Fletcher Powel) [16] require the computation of the Jacobian. EBP follows the concept of the steepest descent optimization algorithm where global error is reduced by following the steepest descent path (moving in the opposite direction to the gradient g.) The weight updating rule is:

$$\Delta w = -\alpha g \tag{3.5}$$

where α is the experimentally selected "learning constant" and g is the gradient vector. For the LM algorithm the weight updating rule is:

$$\triangle w = -(J^T J + \mu I)^{-1} J^T e \tag{3.6}$$

where I is the identity matrix, e is error vector with elements given by equation 3.5, J is the Jacobian matrix, and μ is a learning parameter [8, 16]. If the Jacobian J is known then the gradient g can be found as:

$$g = 2J^T e \tag{3.7}$$

Therefore the updates on the weights $\triangle w$ can be found in both EBP and LM algorithms equations 3.6 and 3.5 if the error vectors e and the Jacobian matrix J are evaluated.

In the Jacobian matrix, each row corresponds to p-th input pattern and o-th network output, therefore the number of rows of the Jacobian is equal to the product $no \cdot np$, where np is number of training patterns and no is number of outputs. The number of columns is equal to the number of weights nw in the neural network. Every neuron has the number of elements in this row, which is equal to the number of inputs plus one. For the *p-th* pattern, *o-th* output, and *n-th* neuron with K inputs the fragment of Jacobian matrix has the form:

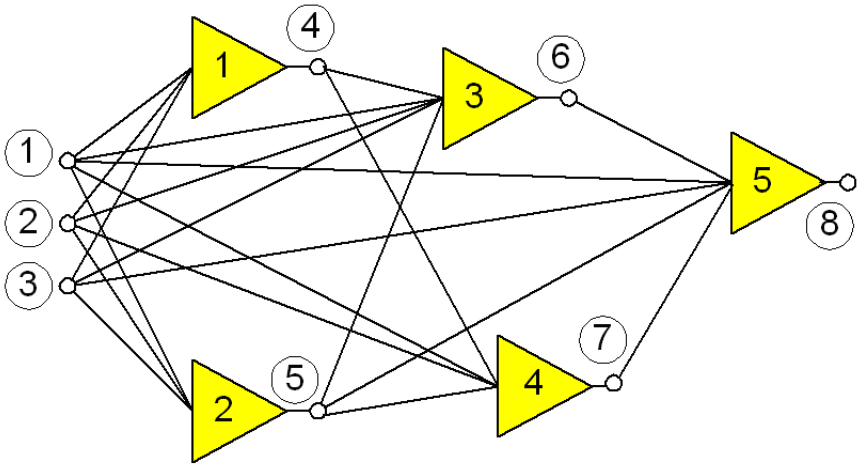

Figure 3.6: Example of an ACN network. The network has five neurons numbered from 1 to 5 and eight nodes 3 of which are input nodes from 1 to 3 and five neuron nodes from 4 to 8.

$$\cdots \frac{\partial e_{po}}{\partial w_{n0}} \frac{\partial e_{po}}{\partial w_{n1}} \frac{\partial e_{po}}{\partial w_{n2}} \frac{\partial e_{po}}{\partial w_{n3}} \cdots \frac{\partial e_{po}}{\partial w_{nK}} \cdots \tag{3.8}$$

Where the weight with index 0 is the biasing weight and *epo* is the error on the *o-th* network output. In this thesis, a new NBN (Neuron by Neuron) method for calculating the gradients and the Jacobians for arbitrarily connected feed forward neural networks is presented. The rest of the computations for weight updates follow the LM algorithm. In order to explain the computation algorithm, consider an arbitrarily connected neural network with one output as shown in Figure 3.6.

Row elements of the Jacobian matrix for a given pattern are being computed in three steps:

1. Forward computations
2. Backward computations

3. Calculation of Jacobian elements

3.2.1 Forward computation

Forward and backward calculations are done using neuron by neuron (NBN) calculations. In the forward calculation, the neurons connected to the network inputs are first processed so that their outputs can be used as inputs to the subsequent neurons. The following neurons are then processed when their input values become available. In other words, the selected computing sequence has to follow the concept of feedforward networks and the signal propagation. If a signal reaches the inputs of several neurons at the same time, then these neurons can be processed in any sequence. For the network in Figure 3.6 there are only four possible ways in which neurons can be processed in the forward direction: 12345; 21345; 12435; or 21435. When the forward pass is concluded, two temporary vectors are stored: vector o with the values of the signals on the neuron outputs and the second vector s with the values of slopes of the neuron activation functions, which are signal dependent.

3.2.2 Backward computation

The sequence of the backward computation is opposite to the forward computation sequence. The process starts with the last neuron and continues toward the input. In the case of the network of Figure 3.6 these are the possible sequences (backward signal paths): 54321; 54312; 53421; or 53412. To demonstrate the case let us select the sequence 54321. The attenuation vector (a) represents signal attenuation

from a network output to the outputs of all other neurons. The size of this vector is equal to the number of neurons.

The process starts with the value one assigned to the last element of the a vector and zeros to the remaining output neurons. During backward processing for each neuron the value of the delta of this neuron is multiplied by the slope of the neuron activation function (element of s vector calculated during forward computation) and then multiplied by neuron input weights. The results are added to the other elements of the a vector neurons which are not yet processed. The second step in the example updates only the elements of the a vector that are associated with neurons 3 and 4 because only these neurons are directly connected to inputs of neuron 5. In the next step neuron 4 is processed and elements of a vector associated with neurons 1 and 2 are updated. Next, the neuron 3 is processed and again elements of the a vector that correspond to neurons 1 and 2 are updated. There is no reason to continue the process beyond this point because there are no other neurons connected to inputs of neurons 1 or 2. As the result of backward processing elements of the a vector are obtained.

One may notice that the backward computation is done only for a limited number of neurons. For example in the case of the 4 topologies shown in Figures 3.1 and 3.2 only one output neuron is processed.

The size of the a vector is equal to the number of neurons while the number of Jacobian elements in one row is much larger and is equal to the number of weights in the network. In order to obtain all row elements of the Jacobian for p-th pattern and

o-th output a very simple formula can be used to obtain the element of the Jacobian matrix associated with the input k of the neuron n:

$$\frac{\partial e_{po}}{\partial w_{nk}} = d(n)_{po} \cdot s(n) \cdot node(k)_{po} \tag{3.9}$$

Where: *d(n)* is the element of a vector and *s(n)* is the slope calculated during forward computation, both are associated with neuron n; *node(k)* is the value on the *k-th* input of this neuron.

3.2.3 Calculation of Jacobian Elements

The process is repeated for every pattern and if a neural network has several outputs it is also repeated for every output. The process of gradient computation in arbitrarily connected neurons (ACN) network is exactly the same, but instead of storing values in the Jacobian matrix, they are being summed into one element of the gradient vector:

$$g(n,k) = \sum_{p=1}^{p} \sum_{o=1}^{O} \frac{\partial e_{po}}{\partial w_{nk}} e_{po} \tag{3.10}$$

If Jacobian is already computed then the gradient can also be calculated using equation 3.5. The latter approach, with Jacobian calculation, has similar computation complexity, but it requires much more memory to store the Jacobian matrix.

3.3 Self Aware Algorithm

The Self Aware algorithm (SA) is a modification of the NBN algorithm. It takes a very large step in automating the training process. The motivation for this algorithm is that many times when training problems that have deep local minima such as parity problems, the algorithms saturate and cannot escape this depression. This may happen after a few iterations or after tens of thousands of iterations and the algorithm must restart with different initial weights. Traditionally the user must manually restart the algorithm, but this means he must be present to observe the situation. This is far less than ideal because many times the training process may take several hours and consume the complete system resources of the computer on which it is operating.

The SA algorithm alleviates the need for a user to be present by self monitoring its progress. This task may seem trivial but actually many variables need to be taken into account. The mean squared error (MSE) will decrease but at drastically different rates during a successful run. The restart algorithm is shown in the following equation.

$$\frac{|MSE_{ite} - MSE_{ite-5}|}{MSE_{ite}} < MSE_{max} \tag{3.11}$$

where ite stands for the current iteration and MSE_{max} refers to the maximum error set by the user for training the network. This takes into account the rate at which the algorithm is reducing the error but relative to the total error. This is

important because if the total error is several orders of magnitude larger than the change in error then under most circumstances it will never converge and should be restarted. However, if the error is reducing by a very small number this is acceptable when the MSE is on the same order of magnitude. This method also takes into account the desired er for the problem at hand. If a rough solution is all that is desired the algorithm has a tendency to restart quicker to get to a rough solution as quickly as possible as where when a very precise solution is requested the algorithm is allowed to slow significantly without restarting.

3.4 Enhanced Self Aware Algorithm

The Enhanced Self Aware algorithm (ESA) incorporates both the NBN and SA algorithms in addition to a modification of the Jacobian matrix. A common problem with many training algorithms is that local minima often cause the weights to get pushed into the saturated portion of the tangent hyperbolic curve see (Figure 3.7.)

In the saturated region, the slope is nearly zero and therefore the derivative is nearly zero. This is a problem because even if a pattern has a large error it is lost when it is multiplied by the very small slope. This pattern no long causes the algorithm to adjust its weights to correct the error. This causes the algorithms to frequently become unable to move out of a local minimum. In order to prevent the patterns from saturating prematurely the Jacobian matrix is scaled by a constant. This allows the patterns to remain in the linear region longer in order for all of the patterns to be correctly classified. It does not create a problem for correctly classified patterns

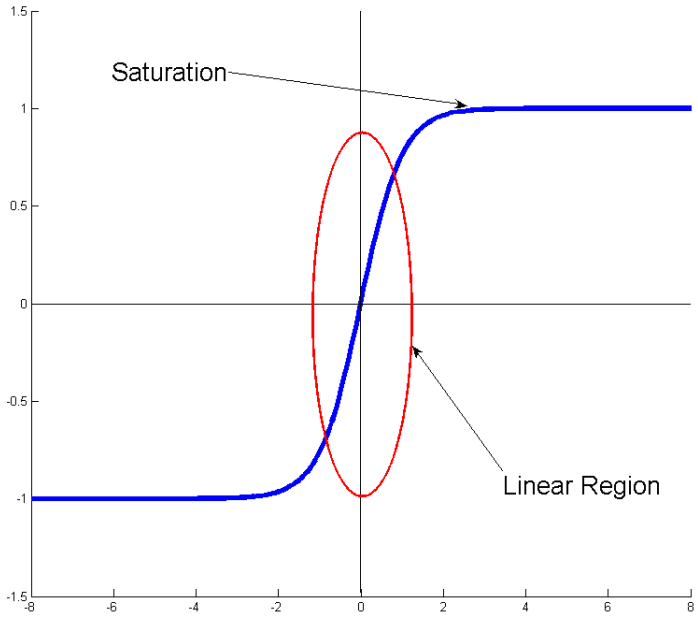

Figure 3.7: Typical hyperbolic tangent activation function.

because even though the derivative is larger their error is very small and they are still able to saturate. This method of delaying the saturation process does slow down the entire process slightly. When the Jacobian is modified in this manner it makes it more difficult to find the exact solution when the errors are small. To account for this, the algorithm monitors the errors for each pattern. When the errors for all patterns are small, the algorithm no longer scales the Jacobian Matrix. This allows the algorithm to converge to a very small error just as NBN does. This process may take slightly longer, but it is more likely to converge in difficult cases.

Chapter 4

Developed Software

Prior to the work reported herein, there was very little software available to train fully connected neural networks. As a significate contribution of this thesis research, a package with a graphical user interface has been developed in MATLAB for that purpose. This software allows the user to easily enter very complex architectures with initial weights, training parameters, data sets, and the choice of several powerful algorithms. The front end of the Neural Network Trainer (NNT) package is shown in Figure 5.1. The first section will discuss the internal workings of NNT followed by a tutorial on how to create and train a network. The final section demonstrates an addition to NNT that allows the user to verify networks used for nonlinear mapping.

The motivation behind NNT was to create a simple to use yet extremely powerful tool for training neural networks. NNT started out as a very rudimentary training method for neural networks that included modifying several mfiles and creating arrays of parameters in order to train a network, and since has evolved into an entire training package that is easy to use.

4.1 Graphical User Interface

The Graphical User Interface was built using the Matlab toolbox GUIDE. It allows the programmer to create series of buttons, textboxes, figures, and pulldown menus with relative ease. Figure 4.1 shows an early version of NNT in the GUIDE

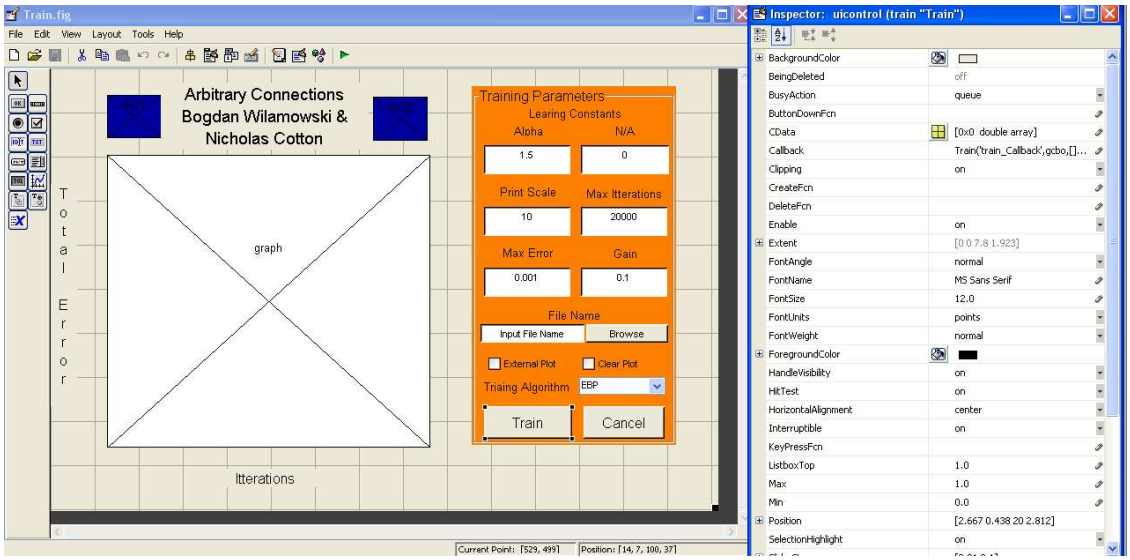

Figure 4.1: Early example of NNT in the GUIDE toolbox.

interface with the property inspector window open on the right side. This window is currently displaying the properties of the TRAIN button. It executes the TRAIN call back function that is at the top of the window. When this function is called it initiates the training process.

One important feature of NNT is that it allows the user to train the same network with multiple algorithms with the simple selection of a drop down menu. This creates a problem from the programmer's standpoint because the algorithms have different training parameters. A work around to the problem was implemented by manipulating typically constant text boxes in the interface using the set function. When the graphical user interface (GUI) is created it automatically generates a structure variable called handles and each text box has an entry in the handles structure. When the user selects a different algorithm the callback function is executed which contains

a switch statement with a case for every algorithm. This statement then sets all the parameter names and updates the default values of each. A snippet of the callback function is shown below.

```
switch handles.algorithm
    case 1
        set(handles.label1,'String','Alpha');
        set(handles.input1,'String','1.5');
        set(handles.label2,'String','N/A');
        set(handles.input2,'String','0');
        set(handles.input3,'String','10');
        set(handles.input4,'String','5000');
        set(handles.gain,'String','.1');
        handles.input1_usr=1.5;
        handles.input2_usr=0;
        handles.input3_usr=10;
        handles.input4_usr=5000;
        handles.gain_usr=0.1;


    case {2,3}
        set(handles.label1,'String','Mu');
        set(handles.input1,'String','0.01');
        set(handles.input2,'String','10');
        set(handles.label2,'String','Mu scale');
        set(handles.label3,'String','Print Scale');
        set(handles.input3,'String','1');
        set(handles.input4,'String','200');
        set(handles.gain,'String','.1');
        handles.input1_usr=.01;
        handles.input2_usr=10;
        handles.input3_usr=1;
        handles.input4_usr=200;
        handles.gain_usr=0.1;
```

This is just the first three cases, but it shows the pattern for the different cases. Notice that the last five lines of code are updating the variables to the corresponding values that are displayed. This is because changing the text box display using the set function is not the same as the user updating the value. This feature allows different default training variables for each algorithm to help assist the user in selecting reasonable values.

4.2 Continuous Training Control

After the user has entered all of the required data for the training process to begin, this data needs to be interpreted. The parameter data from the GUI is stored in a structure that easily passed to the training engine. However, the input file must be interpreted. The input file is read using a series of text document read commands standard in Matlab. These individual commands are fairly straightforward but very entangled. Once the data is input, several variables are created which the training engine uses directly. These details can be found in Chapter 3, which pertains to calculations and training algorithms. However, many features have been implemented into NNT other than simply sending training data to the engine. It also extracts information about the training process, monitors it, and gives the user as much feedback as possible. Once the training is finished it writes a text file with the weights that were the best for that set of runs. The following code block shows the handling of the self aware algorithms, successes, failures, saving data, and plotting.

```
RESTART=0;
total_ite=1;
total_TERR=0;
num_runs=1;
TERR_best=10000000000;

while RESTART==0 && CANCEL == 0
    [ww, nodes, TERR, ite] = LM_BMW(inp,dout,topo,ww, param,nmod);
    dlmwrite('wwtemp.ww',ww);
    if (external_plot)
        axes(handles.graph);
        figure(1);
        if RESTART~=1 || CANCEL==1
            semilogy(TERR,'r--');
            hold on
        else
            semilogy(TERR,'k');
            hold on
        end
    end
    axes(handles.graph);
    if RESTART~=1 || CANCEL==1
        semilogy(TERR,'r--');
        hold on
    else
        semilogy(TERR,'k');
        hold on
    end
    if TERR(end)<TERR_best(end)
        TERR_best=TERR;
        best_ww=ww;
        run_num=num_runs;
        num_ite=ite;
        dlmwrite('best_ww.ww',best_ww);
    end
    num_runs=num_runs+1;

 if total_ite>=max_ite; break; end;
```

```
end
```

This section of code is the link between the GUI and the training engine, as well as providing the user important usable data. The first thing to note is the presence of two global variables RESTART and CANCEL. Using global variables is always the last resort, but in this situation it is an absolute must especially in the case of the CANCEL variable. The reason for this issue is that the CANCEL variable is only set inside a function within the GUI, but it is needed after the GUI has already called the training engine, therefore making it impossible to pass it in the traditional manner. Another work-around that was needed to allow the cancel button to function was to place a .1ms pause inside of the training algorithm that allows the GUI to update. Without this pause even though the cancel button has been pressed, the call back function to set the CANCEL flag would not get called until the training process is complete, which would be too late to be useful. The other global variable is RESTART which is for the self aware algorithms. It allows any of the algorithms from within any function to cause a restart of the training session smoothly. Again, the reason for using a global variable is the complex set of nested functions that the restart command would need to propagate through is too extensive too be efficiently passed. These parameters are very important in controlling when to restart training and when to exit the training process.

The outer while loop controls this feature based on the RESTART and CANCEL flags. The loop continues to train the network until it is canceled by either the user or until certain conditions are met, which will be discussed shortly. The first line within

the loop sends the training data and parameters to the engine to be trained and it returns the weights, nodes, the error array, and the number of iterations performed. If either the RESTART or CANCEL flag is set high then the training process was a failure and is plotted with a dashed red line. The trainer then goes a step further to check the final value of the mean squared error and compare it to the previous best run. It then exports the best weights to a text file and stores the number of iterations required, the error vector, and the run number. If the process was not canceled by the user, another attempt is made until the RESTART flag returns a zero, indicating a successful run. The MSE is plotted with a solid black line, and the weights are written to a text file. After the training is finished, the weights are also printed to the command window along with the other data regarding the best run.

4.3 Nonlinear Mapping

When training neural networks for control surfaces it is important to be able to see the three dimensional shape of the network output, not simply the mean squared error. This was the motivation for creating a tool that works alongside NNT that allows the user to plot networks with two inputs and one output. The tool was created to aid in the process of analyzing and testing neural networks.

4.3.1 Interfacing

The software was created as a direct link to NNT it requires two files an input file and a weight file. The plotting software uses the same input file that was used

by NNT for training. The weight file is simply an array of weights in a text file, typically separated by a line return. This is further simplified for the user because NNT by default generates weight files when it finishes training called tempww.ww and best.ww which contain the weights for the most recent run and the best run respectively. The user needs only to select these two files, and set the appropriate gain used for training, and then simulate. The simulator can be seen in Figure 4.2.

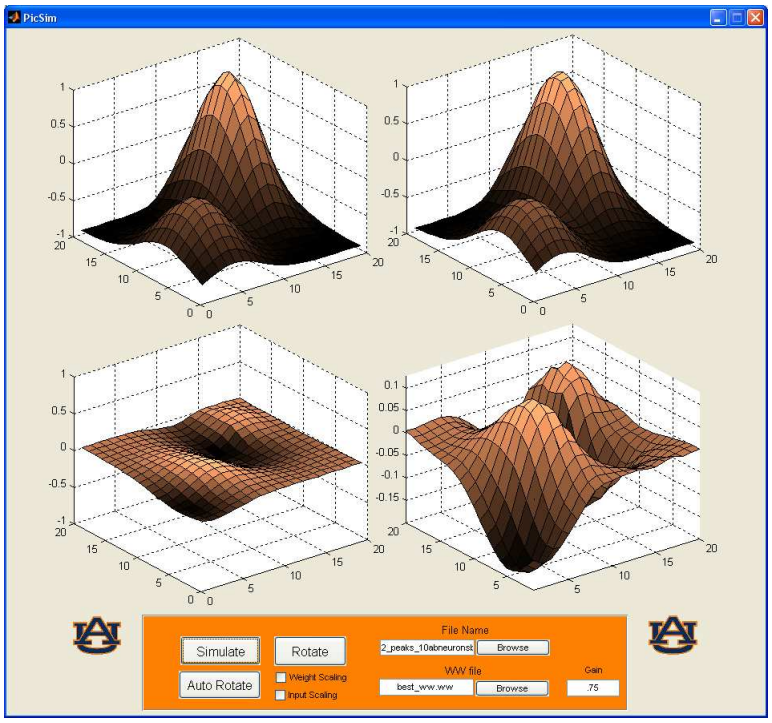

Figure 4.2: Neural network simulation environment for three dimensional surfaces.

As seen in Figure 4.2 there are four graphs. The graph in the top left is the desired surface that the network was trained from and the graph in the top right is the output of the trained network. The third graph is the difference between the previous two surfaces plotted on the axis with identical dimensions to give perspective

of the relative size of the error. The last plot is also a difference of the first two plots but the axes are a closer fit to allow the user to see the shape of the error more closely.

4.3.2 Synchronizing Plots

One of the key features of the software is that it not only allows the user to easily generate these plots but also to view them easily. When the user clicks the rotate button it allows the first figure to be manipulated to be seen from any angle. Once the user is satisfied with the current angle he must simply press the rotate button again which then aligns the remaining figures with an identical viewing angle. This task requires the use of the AXES and view commands in Matlab. The following block of code shows the process of matching the viewing angles.

```
function Rotate_Callback(hObject, eventdata, handles)
% handles    structure with handles and user data (see GUIDATA)
axes(handles.graph);
rotate3d

axes(handles.graph)
temp=view;
axes(handles.graph2)
view([temp]);

axes(handles.graph3)
view([temp]);
axes(handles.graph4)
view([temp]);
```

When the rotate button is pressed the callback function is executed which reads the new viewing angle of the first graph and then overwrites the view of the other three graphs. This step is then taken even further with the auto rotate button. This button does a similar task but instead of the user rotating the view manually it slowly rotates automatically one rotation or until stoped by the user. This function for rotating is shown below.

```
function auto_rotate(handles)
global STOP_ROTATE
deg=360;
[az el]=view;
rotvec=0:deg/180:deg;

for i=1:length(rotvec)
    if STOP_ROTATE==-1; break; end

    axes(handles.graph)
    view([az+rotvec(i) el])
    axes(handles.graph2)
    view([az+rotvec(i) el])
    axes(handles.graph3)
    view([az+rotvec(i) el])
    axes(handles.graph4)
    view([az+rotvec(i) el])
    drawnow

    pause(.1)
end
```

The auto rotate function must again use a global variable in order to interrupt the process and allow the user to stop the rotation if so desired. The code creates a rotate vector between 0 and 360 degrees with a step size of 2 degrees. This allows

the graph to move at a reasonable pace and still be fluent. The rotations starts at the current azimuth of the plot not necessarily zero which is why the rotate vector is added to the original azimuth each rotation. At the end of the plot there is a drawnow function which is required to allow the current function to pause and draw anything that is in the buffer or else it would wait for the entire function to finish and would not be an animated rotation.

Chapter 5

Neural Network Trainer

The user will first notice there is an empty plot on the left side of the trainer where the iterations versus means squared error will be displayed as well as training parameters on the right hand side. In using NNT the user must follow a few simple steps before training a network. The user prepares an input file that contains the training data and an architecture file that describes the network connections, then sets the training parameters.

5.1 Training Data

The user must create a training file with all the data sets required to train the NN. This data may be created in various ways such as by hand, spreadsheet, or directly through Matlab. A simple parity-3 problem will be used for demonstration purposes. This demonstration will use bipolar neurons so the extremes for data will

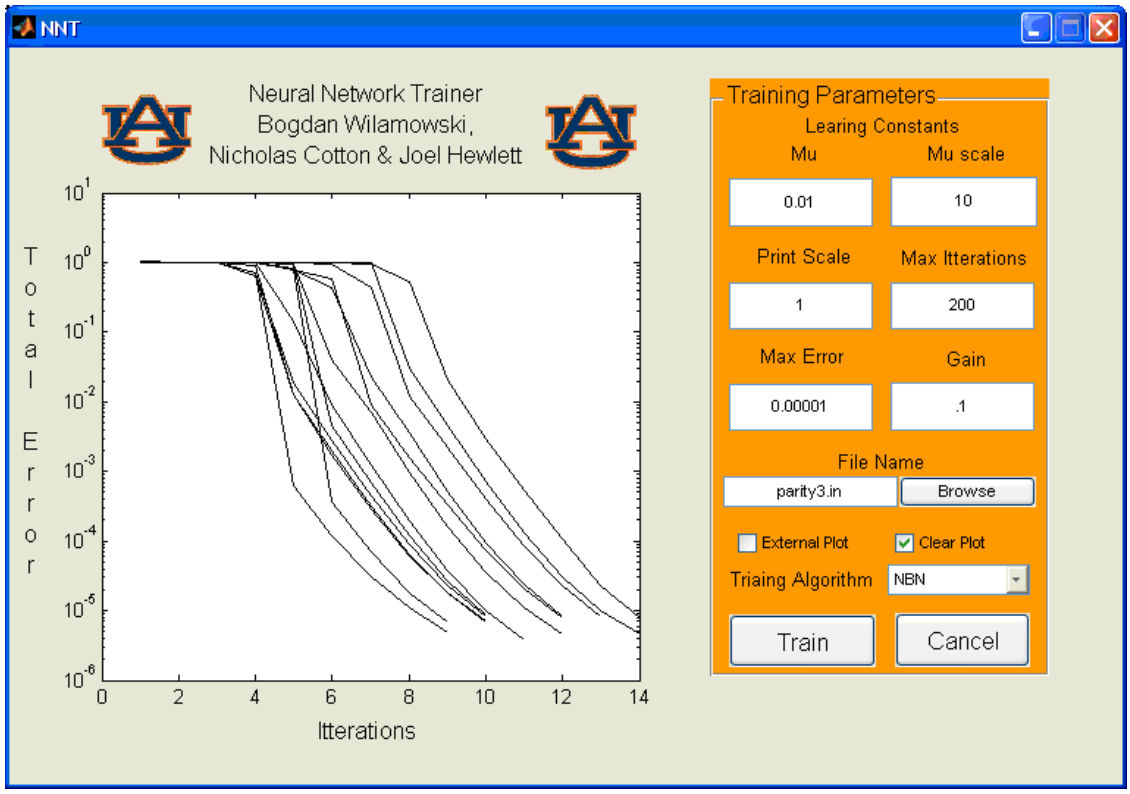

Figure 5.1: Front end of Neural Network Trainer (NNT)

be ±1. The training data for parity-3 is represented by the following matrix:

In_1	In_2	In_3	Out_1
−1	−1	−1	−1
−1	−1	1	1
−1	1	−1	−1
−1	1	1	1
1	−1	−1	−1
1	−1	1	1
1	1	−1	−1
1	1	1	1

As with any parity-n problem there are 2^n possible outcomes. As the top row indicates the first three columns are the inputs and the last column is the output for that row. The top row of the matrix is for demonstration purposes only but is not need in the actual data file. This data is then copied to a text file and saved with the file extension ".dat". Delimiters other than white space are not required. Once the data file is finished it can be referenced by numerous architecture input files.

5.2 Input File

The input file contains the network architecture, neuron models, data file reference, and optional initial weights. Each input file will be unique to each architecture but not necessarily to each data set. In other words, the same data set can be used for several different architectures simply by creating a new input file. The input file contains 3 sections: the architecture, model parameters, and data file definition. The following is an example of an input file for the parity-3 problem discussed in the previous section.

```
\\ Parity-3 input file (parity3.in)
n 4 mbip 1 2 3
n 5 mbip 1 2 3
n 6 mbip 1 2 3 4 5

W   5.17  20.08 -10.01   -4.23
W   1.0   10.81   2.20   19.84

.model mbip fun=bip, der=0.01
.model mu   fun=uni, der=0.01
```

```
.model mlin fun=lin, der=0.05

datafile=parity3.dat
```

The first line is a comment. Either a double backslash, as in C, or a percent sign, as in Matlab, is acceptable as a comment delimiter. After the comment comes the network architecture for a 3-neuron fully-connected network as shown in Figure 5.2

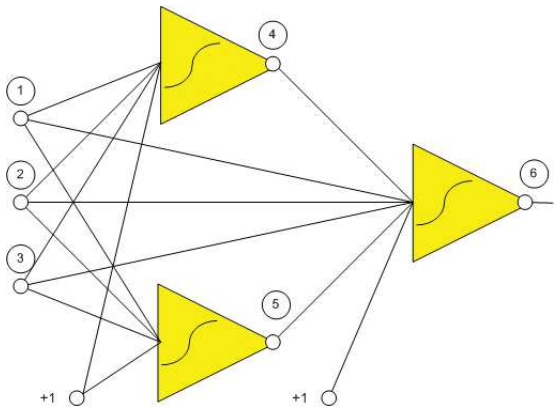

Figure 5.2: Three Neuron architechture for parity-3 problem.

The neurons are listed in a net list type of layout that is very similar to SPICE program. This way of listing the layout is node based. The first nodes are reserved for the input nodes. The first character of the line is an N to signify that this line describes a neuron. The N is followed by the neuron output node number. Looking at Figure 5.2, the first neuron is neuron 4 because it is the first available number after the three inputs, and it is connected to nodes 1, 2, and 3 which are inputs. The same is true for neuron 5 or the second neuron it is also connected to all three inputs. The output node is slightly different but it follows the same concept. It is connected to all three inputs as well as to the output of the first two neurons. Based on this it

should be straightforward to see the connection between the input file listed above and Figure 5.2.

Also one will notice on the line of each neuron is the model of the neuron which allows the user to specify a unique model for each neuron. This network that is setup to solve a parity-3 problem using three bipolar neurons. This is not the minimal architecture for this problem, but it serves as a good demonstration of the tool.

Following the architecture of the network are the optional starting weights. If no starting weights are given the trainer will choose random weights. The weights need to be listed in the same format as the architecture. Each line of weights starts with the capital letter W. The biasing weight goes in place of the output node of the neuron. In other words, the first weight listed for a particular neuron is the biasing weight followed by the remaining input weights in their respective order. See the input file for an example.

The user specifies a model for each neuron and these models are defined on a single line. The user has the ability to specify the activation function, and neuron type (unipolar, bipolar or linear) for each model. The user may include neurons with different activation functions in the same network.

The final line of the input file includes a reference to the data file. This line simply needs to read *datafile=* followed by the file name and in this example it would be *parity3.dat* which can be seen on the last line of the example input file.

5.3 Training Parameters

Once the network architecture has been decided and the input files created the next step is to select the training algorithm and parameters. When NNT is loaded there is an orange panel full of adjustable parameters on the right side of the window. These parameters change for each algorithm so they will be addressed accordingly in the following section. The algorithms themselves are explained in more detail in Chapter 3, but first the user should select the input file just created.

5.3.1 Implemented algorithms

The algorithm is chosen from the pull down menu in the training parameters. Four of the parameters are the same for all algorithms. They are: Print Scale, Max. Iterations, Max. Error, and Gain. The Print Scale refers to how often the mean squared error is printed to the Matlab command window. This can be important because in certain situations the longest calculation time is that of displaying the data, so increasing this number can significantly decrease training time. Max iterations is the number of times the algorithm will attempt to solve the problem before it is considered a failure. An iteration is defined as one adjustment of the weights, which includes calculating the error for every training pattern and adjusting at the end. The Max Error is the mean squared error that the user considers to be an acceptable value. When this number is reached the algorithm stops calculating and displays the final weights. The last common parameter is the Gain which is the factor by which

the sum of the inputs is multiplied for all the neurons before the activation function is applied see Equation (5.1.)

$$output = \tanh(gain \cdot net) \tag{5.1}$$

Error Back Propagation (EBP)

This algorithm is the traditional EBP with the ability to handle fully connected neural networks. The Alpha parameter is the learning constant. This value is a multiplier that acts as the numerical value of the step size in the direction of the gradient. If Alpha is too big the algorithm can oscillate instead of reducing the error. However, if alpha is too small the algorithm can move toward the solution too slowly and prematurely level off. This parameter should be adjusted by the user until an optimal value is found which has some oscillation that diminishes while the error continues to decrease.

Neuron By Neuron (INBN)

NBN is a modified Levenberg-Marquardt algorithm for arbitrarily connected neural networks. The NBN algorithm formally known as the BMW algorithm was briefly described in [8]. It has two training parameters, μ and μ Scale. The learning parameter of the LM algorithm is μ. Its use can be seen in Equation 5.2.

$$w_{k+1} = W_k - (J_k^T J_k + \mu I)^{-1} J_k^T e \tag{5.2}$$

If $\mu = 0$ then the algorithm becomes the Gauss-Newton method. For very large values of μ the algorithm becomes the steepest descent method or EBP. The μ parameter is automatically adjusted at each iteration to insure convergence. The amount it is adjusted each time is μ Scale which is the last parameter for the NBN algorithm.

Self Aware (SA)

The SA algorithm is a modification of NBN. It evaluates the progression of the algorithm's training and determines if the algorithm is failing to converge. If the algorithm begins to fail, the weights are reset and another trial is attempted. In this situation the program displays its progress to the user as dotted red line on the display and begins again. The algorithm continues to attempt to solve the problem until either it is successful or the user cancels the process. The SA algorithm uses the same training parameters as NBN.

Enhanced Self Aware algorithm (ESA)

ESA is also a modification of the NBN algorithm is used in order to increase chances for convergence. The modification was made to the Jacobian Matrix in order to allow the algorithm to be much more successful in solving very difficult problems with deep local minima. It also is aware of its current solving status and will reset when necessary.

The ESA algorithm uses a fixed value of 10 for the μ Scale parameter and instead allows the user to adjust the LM parameter. The LM parameter is essentially a scale

factor applied to the Jacobian matrix before it is used in calculating the weight adjustment. This scale factor is typically a positive number between 1 and 10 or possibly greater. The more local minima the problem has the larger the LM factor should be.

Forward-Enhanced Self Aware (F-ESA)

F-ESA is another modification of NBN algorithm where an alternative method for calculating the Jacobian matrix is used. The calculation of Jacobian is unique in the sense that only feed-forward calculations are needed. This approach is then paired with Enhanced Self-Aware LM algorithm. The F-ESA algorithm is included in the NNT but was written by Joel Hewlett. The F-ESA algorithm uses the same training parameters as the ESA algorithm.

Evolutionary Gradient

Evolutionary Gradient is newly developed algorithm, which evaluates gradients from randomly generated weight sets and uses gradient information to generate a new population of weights. This is a hybrid algorithm which combines the use of random populations with an approximated gradient approach. Like standard methods of evolutionary computation, the algorithm is better suited for avoiding local minima when compared to common gradient methods such as EBP. What sets the method apart is the use of an approximated gradient which is calculated with each population. By generating successive populations in the gradient direction, the algorithm is able to

converge much faster than other forms of evolutionary computation. This combination of gradient and evolutionary methods essentially offers the best of both worlds. The training parameters are very different than the LM based algorithms previously discussed. They include Alpha, Beta, Min. Radius, Max Radius, and Population. This algorithm was written by Joel Hewlett and detail regarding these parameters may be found in [17].

5.3.2 Training

Once the user selects the appropriate training algorithm the parameter boxes will change to the corresponding parameters and default values will fill the boxes. After the user sets the parameters, there are two other boxes that can be selected. The clear plot box when checked will overwrite any existing plot with the new one, but if it is left unchecked then the following plots will be drawn on the axis with all of the previous drawings. The last option is the external plot which draws the plots inside NNT and in a separate figure allowing for easy printing or modifying the plot. The train button begins the training process which prints the error to the Matlab Command Window as it is training. At any time the process can be halted and the results plotted by pressing the cancel button. Example plots will be shown in Chapter 6.

Chapter 6

Experimental Results

The Neural Network Trainer has been tested on several types of problems in order to show its success at training neural networks. Several tests were created to test both overall training ability as well as the individual algorithms. The following sections will compare the training speeds and success rates of the algorithms, demonstrate the trainers ability to train networks for nonlinear mapping, and networks with large number of inputs and deep local minima.

6.1 Fully Connected Networks

Most of the examples throughout the results sections will use fully connected networks because they produce better training results as measured by success rates and number of iterations required to converge. This is according to [8] as well as evidenced by the following results. The following experiment was done using the NBN algorithm on networks with different numbers of neurons in the hidden layer using both fully connected and traditional multilayer architectures. Figure 3.1 on the right side is an example of a network with one neuron in the hidden layer as where the left side has three neurons in the hidden layer. Parity-3 and parity-5 problems were tested and the results are shown in Tables 6.1 and 6.2. The results show that fully connected networks are converging with fewer iterations and in significantly more cases.

Hidden Neurons		1	2	3	4	5
MLP	SR	0%	80%	98%	100%	100%
	ANI	NA	19	10	9	8
FCN	SR	100%	100%	100%	100%	100%
	ANI	7.8	7.2	7.0	6.8	6.7

Table 6.1: Parity-3 SR-Success Rate, ANI- Average Number of Iterations using NBN algorithm.

Hidden Neurons		2	3	4	5	6	7
MLP	SR	0%	4.9%	42 %	74%	89 %	96 %
	ANI	NA	45	57.5	46.7	34.3	28.5
FCN	SR	24%	57%	69%	86%	95%	95%
	ANI	24	21	19	18	17.7	16.8

Table 6.2: Parity-5 SR-Success Rate, ANI- Average Number of Iterations using NBN algorithm.

6.2 EBP Compared to NBN

Error Back Propagation is used by many people around the world for training neural networks. Therefore it will be used as the benchmark for training speed as well as ability to converge. The following results show that EBP is outperformed by the NBN, SA, and ESA algorithms in all aspects. EBP was used to train a parity-3 problem and the results can be seen in Figure 6.1. EBP was also used in efforts to train parity-5 and parity-7 problems but was never able to converge (see Figure 6.2.) From examining Figure 6.1 it is apparent that EBP took several thousand iterations to reach an error of 10^{-6} where it was asymptotically approaching a limit.

SA was used to solve the same problem and the results are shown in Figure 6.3. From the graph it can be seen that the SA algorithm converged to an error of 10^{-6} in just tens of iterations and with no limit. Similar results were produced for parity-5

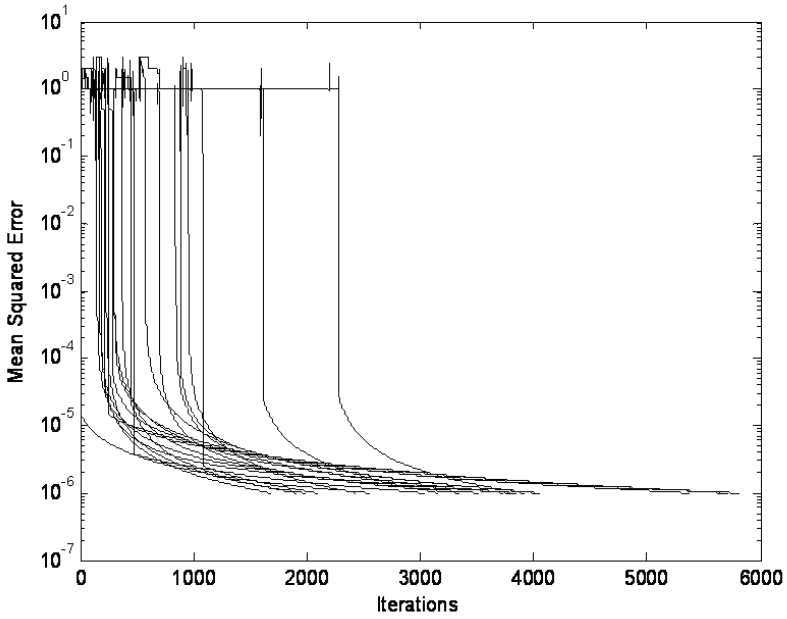

Figure 6.1: Results of EBP algorithm for a parity-3 problem using two neurons fully connected as in Figure 3.1

problems with comparable number of iterations. These results can be seen in Figure 6.4. However, with parity-5 NBN converges less frequently and rarely converges with parity-7. These situations are where ESA produces the best results.

6.3 ESA Compared to NBN

The ESA algorithm was compared with NBN algorithm. At the beginning of each training session random weights are generated as the starting weights for the calculation. These weights can play a significant role on whether or not the algorithm will converge. This makes it difficult to test whether the algorithm is actually solving the problem because it had a good starting point or because the algorithm is working

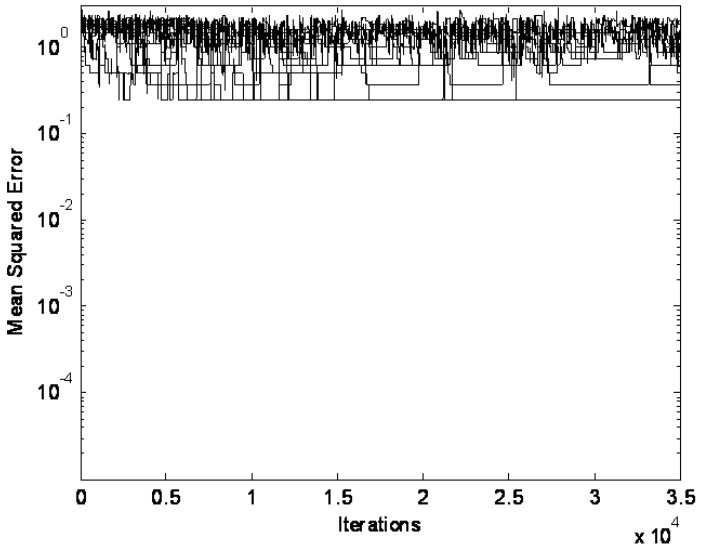

Figure 6.2: Results of EBP algorithm for a parity-5 problem using three neurons fully connected and EBP was never able to converge.

properly. In order to remove this variable a library of random weight sets were generated for each neural network and saved. Each algorithm was tested for each starting set and its convergence was recorded. Based on success rate of the ESA algorithm, it was better than that of the BMW algorithm on every training set compared. The ESA in many cases took more iterations but this cost is heavily outweighed by shear ability to converge with a reasonable success rate. Although the ESA algorithm was converging with better success rates, it was always converging more slowly than the NBN algorithm. Therefore, it should be used only for very difficult cases. Table 6.3 compares ESA and NBN algorithms for a variety of parity problems.

The ESA algorithm was used to solve parity-3 and parity-5 problems and the results are shown in Figures 6.5 and 6.6. These figures can be compared to the

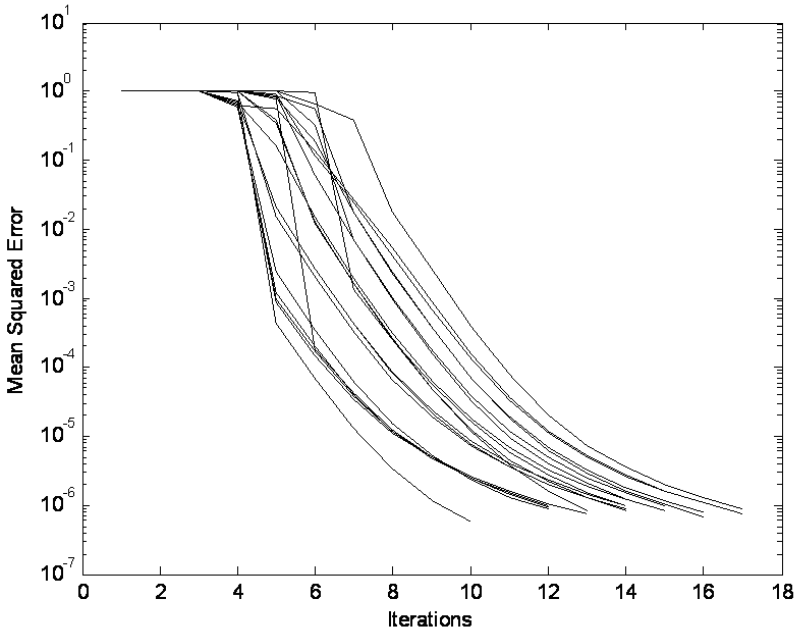

Figure 6.3: Results of SA algorithm for a parity-3 problem using two neurons fully connected as in Figure 3.1

results from the NBN algorithm in Figures 6.3 and 6.4. It can be seen that the ESA does converge more slowly but it is more reliable, as which will be seen in the next comparison.

The following figures show successful runs for very difficult parity problems. These problems are very difficult for several reasons. The number of inputs is increasing exponentially with each higher order problem, requiring more patterns to be classified. Also the Jacobian becomes large, therefore much more time is required at each step.

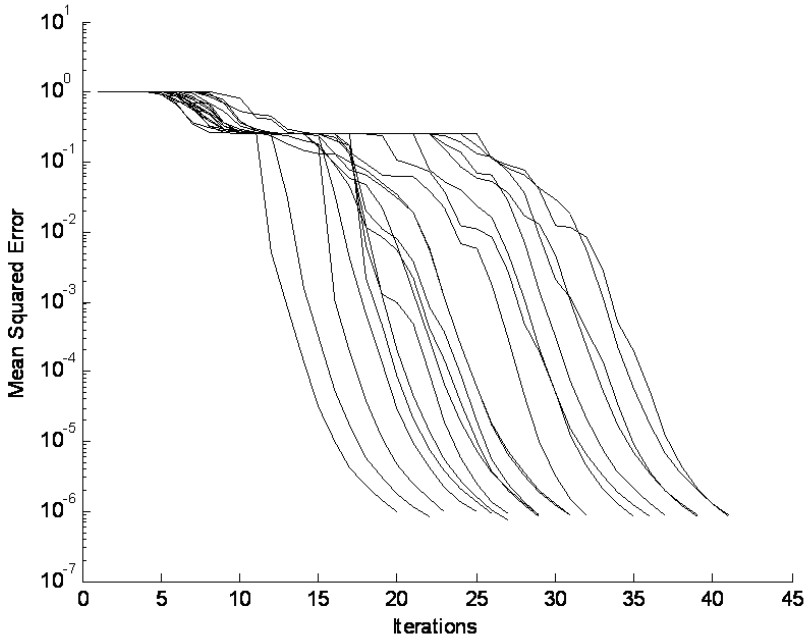

Figure 6.4: Results of SA algorithm for a parity-5 with a fully connected 3 neuron network.

6.4 Surfaces

Another network was trained to solve a nonlinear control problem. The surface was generated using the Matlab function *Peaks*. The network was then trained and the figures were created using the nonlinear mapping software described in Section 4.3. Figure 6.9 is the network that was trained to a mean squared error of 10^{-4} using NNT. The surface used for training is shown in Figure 6.10 and the output of the trained network is Figure 6.11. The difference between the two figures is shown in Figure 6.12 on the same axis to help put the error in perspective. Figure 6.13 shows

	Algorithm	Success Rate	Scale
Parity-3	NBN	79%	-
	ESA	94%	2.9
Parity-5	NBN	21%	-
	ESA	70%	2.5
Parity-7	NBN	<1%	-
	ESA	43%	4
Parity-9	NBN	17%	-
	ESA	59%	4

Table 6.3: Comparison of NBN and ESA algorithms

the error on a much smaller scale to illustrate the locations of the largest error which is typically at the tips of the peaks and depressions.

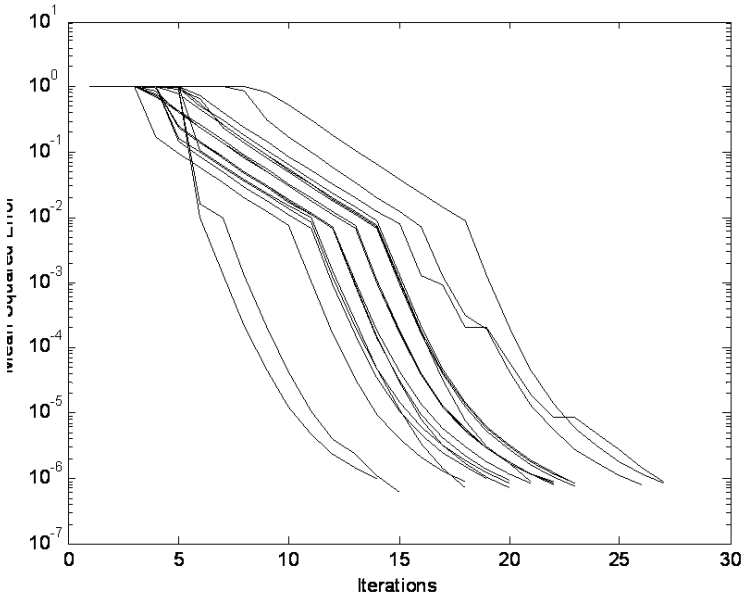

Figure 6.5: Results of ESA algorithm for a parity-3 with two neurons fully connected.

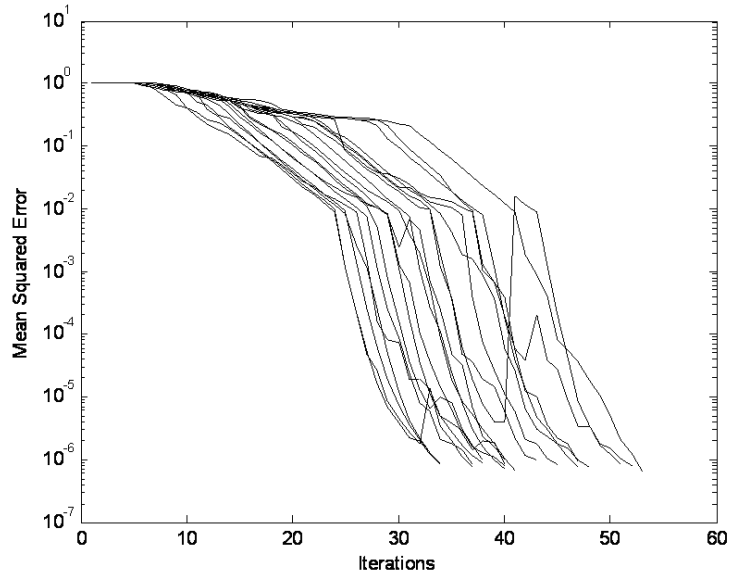

Figure 6.6: Results of ESA algorithm for a parity-5 with a fully connected four neuron network.

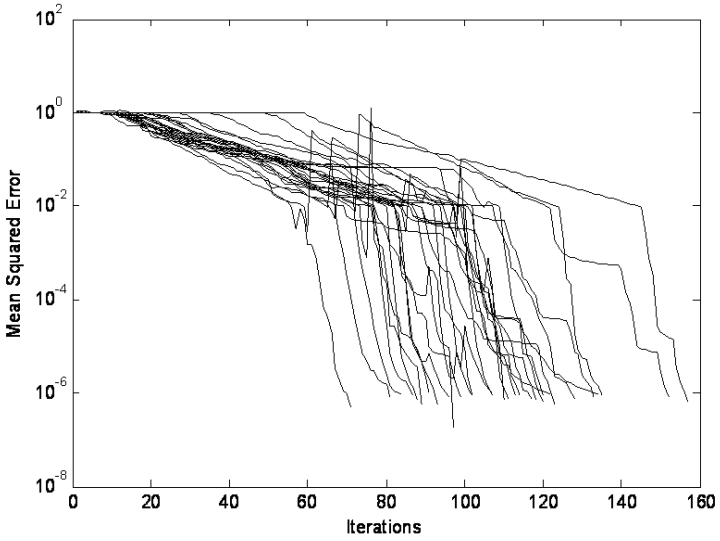

Figure 6.7: Results of ESA algorithm for a parity-7 with a fully connected 4 neuron network.

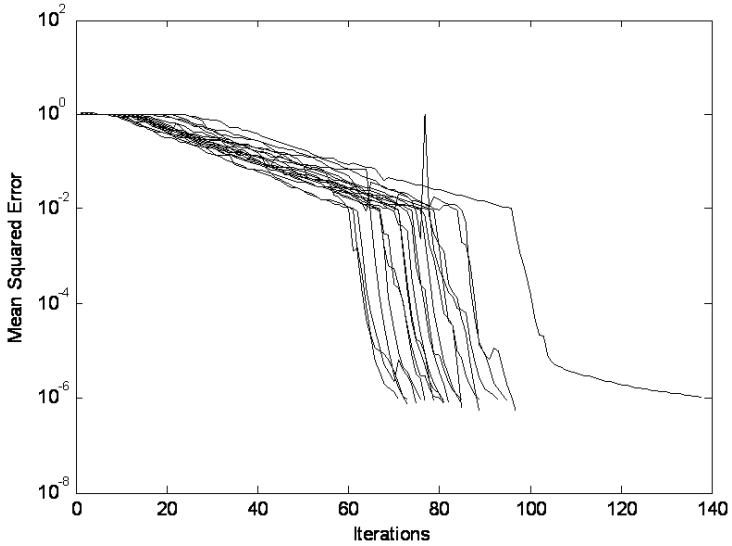

Figure 6.8: Results of ESA algorithm for a parity-9 with a fully connected 4 neuron network.

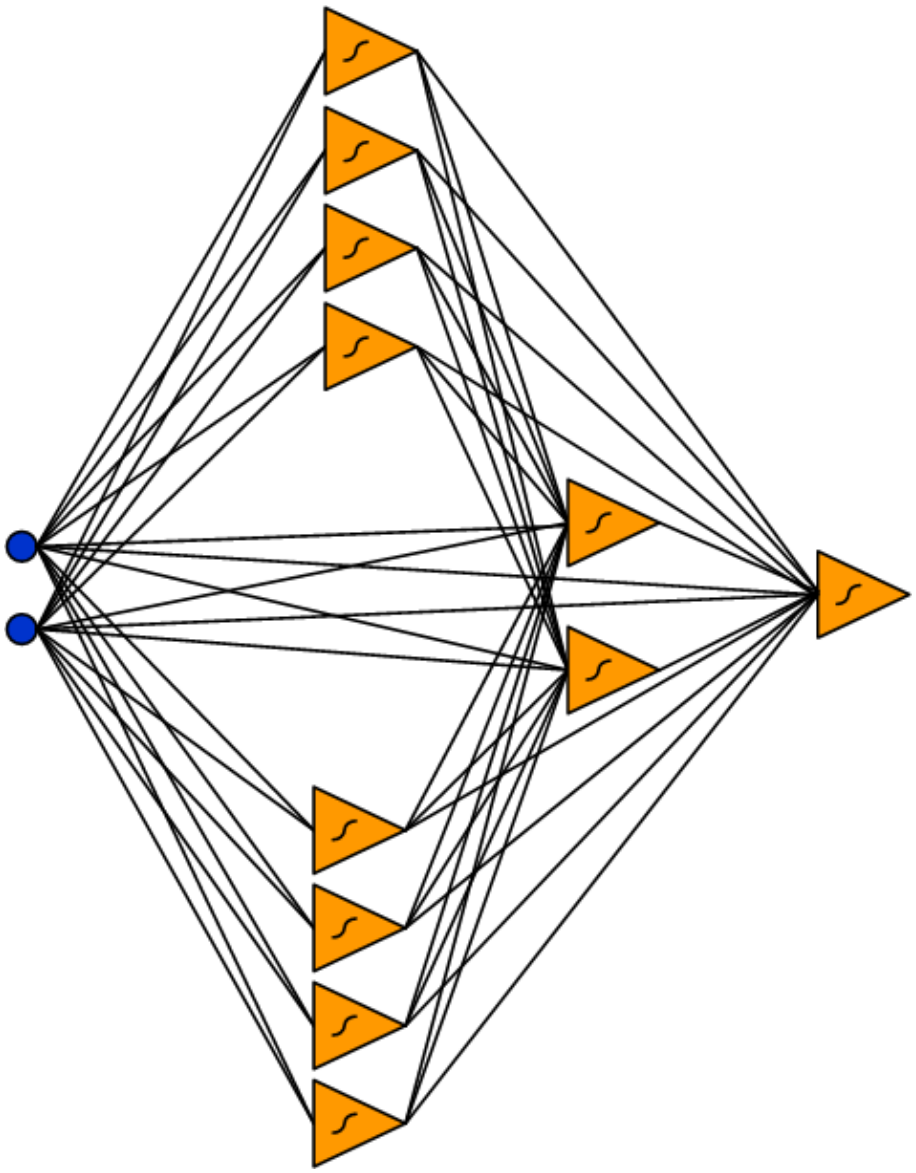

Figure 6.9: The fully connected architecture used for the models in Figures 6.10. Note that this architecture does not show each neuron's biasing weight in order to simplify the drawing.

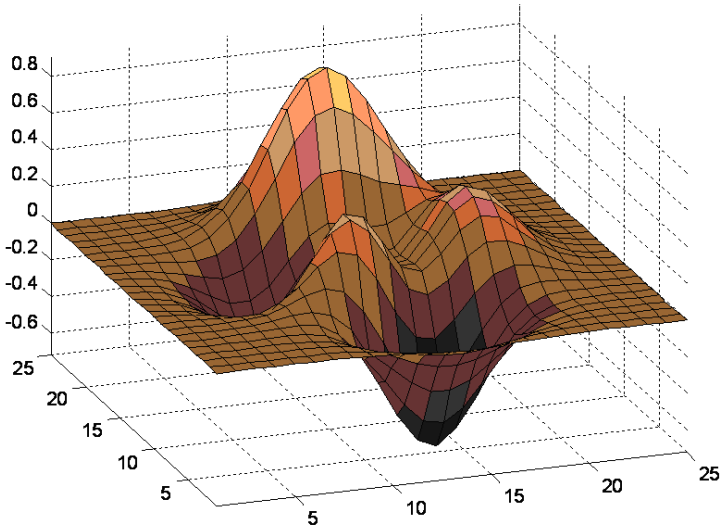

Figure 6.10: The desired surface used for training.

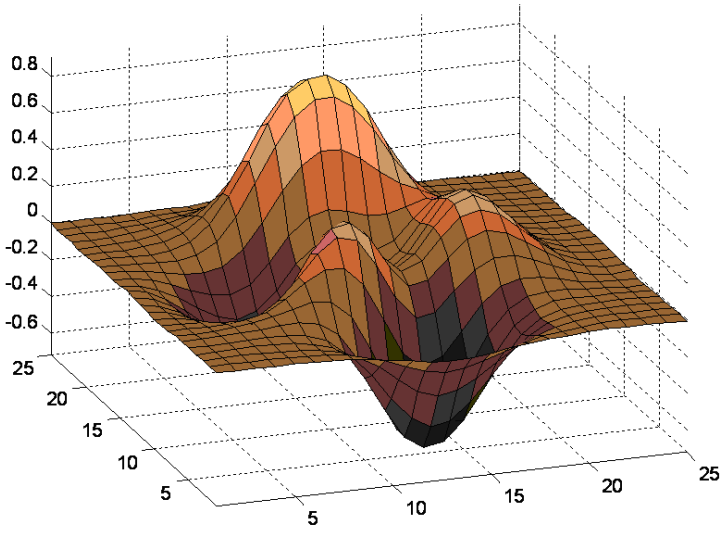

Figure 6.11: The surface obtained from the output of the successfully trained network in Figure 6.9.

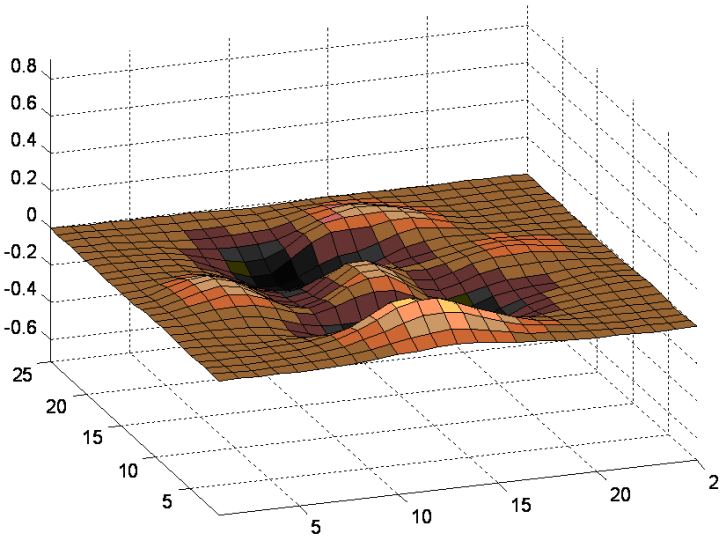

Figure 6.12: The difference between Figures 6.10 and 6.11 plotted on the same scale as the original figures.

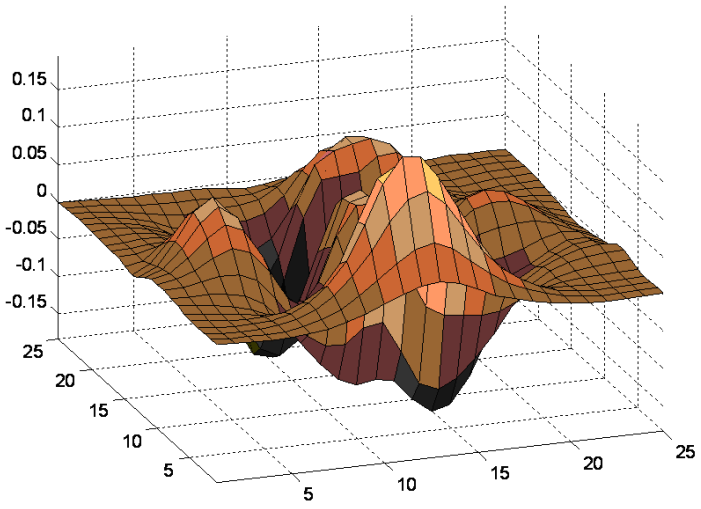

Figure 6.13: The difference between Figures 6.10 and 6.11 plotted on a much smaller axis to show details.

Chapter 7

Conclusion

This Thesis presents a method that eases the process of creating, training, and testing neural networks. Common training algorithms are compared such as Error Back Propagation as well as Levenberg-Marquardt. It is shown that EBP is easily implemented to train networks, but is not powerful enough for more difficult problems. The LM algorithm is much more powerful but it is more difficult to implement. Currently there is very little software available to train fully connected networks. Fully connected networks are shown to converge more often and faster than traditionally connected networks. A solution for training fully connected neural networks with the LM algorithm is presented.

The Neural Network Trainer is a training package that allows the user to easily create different topologies for the same problem ,train them, and make the appropriate adjustments. NNT also allows the user to train networks with a variety of algorithms and easily compare the results.

Included as an additional tool for NNT is software to map nonlinear surfaces. This software is ideal for testing three dimensional nonlinear control problems. It allows the user to easily compare the ideal surface with the actual surface produced. It simplifies this task by allow the surfaces to be rotated simultaneously for easier viewing.

The included algorithms were tested and the results compared to show the advantages and disadvantages of each. The training results of NNT are shown solving several different problems to demonstrate the versatility of the trainer.

Bibliography

[1] J. Martins, P. Santos, A. Pires, L. da Silva, and R. Mendes, "Entropy-based choice of a neural network drive model," vol. 54, no. 1, pp. 110–116, Feb. 2007.

[2] H. Zhuang, K.-S. Low, and W.-Y. Yau, "A pulsed neural network with on-chip learning and its practical applications," vol. 54, no. 1, pp. 34–42, Feb. 2007.

[3] B. K. Bose, "Neural network applications in power electronics and motor drives an introduction and perspective," *Industrial Electronics, IEEE Transactions on*, vol. 54, no. 1, pp. 14–33, February 2007.

[4] H. C. Lin, "Intelligent neural network-based fast power system harmonic detection," vol. 54, no. 1, pp. 43–52, Feb. 2007.

[5] B. Singh, V. Verma, and J. Solanki, "Neural network-based selective compensation of current quality problems in distribution system," vol. 54, no. 1, pp. 53–60, Feb. 2007.

[6] W. Qiao and R. Harley, "Indirect adaptive external neuro-control for a series capacitive reactance compensator based on a voltage source pwm converter in damping power oscillations," vol. 54, no. 1, pp. 77–85, Feb. 2007.

[7] S. Chakraborty, M. Weiss, and M. Simoes, "Distributed intelligent energy management system for a single-phase high-frequency ac microgrid," vol. 54, no. 1, pp. 97–109, Feb. 2007.

[8] B. Wilamowski, N. Cotton, O. Kaynak, and G. Dundar, "Method of computing gradient vector and jacobean matrix in arbitrarily connected neural networks," in *Proc. IEEE International Symposium on Industrial Electronics ISIE 2007*, 4–7 June 2007, pp. 3298–3303.

[9] B. M. Wilamowski, N. Cotton, J. Hewlett, and O. Kaynak, "Neural network trainer with second order learning algorithms," in *Proc. th International Conference on Intelligent Engineering Systems*, June 29 2007–July 1 2007, pp. 127–132.

[10] L. Da Silva, L. Da Silva, B. Bose, and J. Pinto, "Recurrent-neural-network-based implementation of a programmable cascaded low-pass filter used in stator flux synthesis of vector-controlled induction motor drive," vol. 46, no. 3, pp. 662–665, 1999.

[11] M. Embrechts, M. Embrechts, and S. Benedek, "Hybrid identification of nuclear power plant transients with artificial neural networks," vol. 51, no. 3, pp. 686–693, 2004.

[12] M. Hagan, M. Hagan, and M. Menhaj, "Training feedforward networks with the marquardt algorithm," vol. 5, no. 6, pp. 989–993, 1994.

[13] T. J. Andersen and B. Wilamowski, "A modified regression algorithm for fast one layer neural network training," in *World Congress of Neural Networks*, vol. 1. World Congress of Neural Networks, July 1995, pp. 687–690.

[14] R. Sandige and B. Wilamowski, "Methods of removing single variable static hazards in boolean functions," vol. 38, no. 3, pp. 274–278, Aug. 1995.

[15] B. Wilamowski, D. Hunter, and A. Malinowski, "Solving parity-n problems with feedforward neural networks," in *Proc. International Joint Conference on Neural Networks*, vol. 4, 2003, pp. 2546–2551 vol.4.

[16] B. Wilamowski, "Neural network architectures and learning," in *Proc. IEEE International Conference on Industrial Technology*, vol. 1, 10–12 Dec. 2003, pp. TU1–T12.

[17] J. Hewlett, B. Wilamowski, and G. Dundar, "Merge of evolutionary computation with gradient based method for optimization problems," in *Proc. IEEE International Symposium on Industrial Electronics ISIE 2007*, 4–7 June 2007, pp. 3304–3309.