Improved Remote Sensing Data Analysis

Using Neural Networks

by

Ambrose J. Slone

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Electrical [Computer] Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

February 1995

© 1995 Massachusetts Institute of Technology. All rights reserved.

Author

Department of Electrical Engineering and Computer Science

February 2, 1995

Certified by

David H. Staelin
Thesis Supervisor

Accepted by

F.R. Morgenthaler
Chairman, Department

Committee on Graduate Theses

AUG 10 1995
Improved Remote Sensing Data Analysis Using Neural Networks
by
Ambrose J. Slone

Submitted to the
Department of Electrical Engineering and Computer Science

February 1, 1995

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Computer [Electrical] Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

A method for utilizing neural networks in performing analysis and reduction of remote sensing data is presented. A software package for designing and testing extended neural network architectures is developed. An algorithm is discussed for reducing the memory requirements of the Levenberg-Marquardt training algorithm. Software is presented for generating data distributions in three dimensions for testing neural network performance. A definition of a neural network transform is presented, and several methods for implementing the transform are tested. A method for performing the neural network transform is developed which combines incremental network training and input and target data modification. This method is used to analyze artificially created data sets and a data set of atmospheric temperature profiles. Results show neural network performance which is significantly better than the performance of the Karhunen-Loeve transform. Neural networks can be expected to perform better than linear estimators for temperature profile data due to their ability to capture higher-order statistics in the data.
Acknowledgments

I would like to thank professor David H. Staelin for his guidance and patience over the course of my work. His ideas and suggestions were invaluable in completing this thesis.

I would also like to thank Carlos Cabrera for his contributions and comments to every section of this thesis. I also wish to thank the members of the MIT Research Laboratory of Electronics Remote Sensing Group Michael J. Schwartz, William J. Blackwell, Michelle J. Spina, Rizwan Koita, Mark Rawizza, and Dr. Philip W. Rosenkranz for their support and encouragement.

I also owe much to my friends and fraternity brothers Patrick Coady, Jeff Breedlove, Julian Sell, Jonathan Gass, and Benjamin Soule for many great times. Thanks to Eric Fitch, Chandler Harben, Timothy Michalak, and Chris Danielian for being a great fraternity family. I would like to thank all of my friends and fraternity brothers at Sigma Alpha Epsilon. Phi Alpha.

I would also like to thank my parents Lacy and Corenne Slone for unfailing support through all stages of my college experience, and through my entire life. I would like to thank my brother John for helping me keep perspective on things while away at school. I would also like to thank Rebecca Burton for always being there when I needed support.

Financial support from NASA contract NAS 5-31376 is acknowledged.
Contents

1 Introduction ......................................................... 9

2 Hardware and Software ........................................... 12
  2.1 Commercial Hardware and Software ......................... 12
  2.2 Customized Neural Network Software ....................... 13
    2.2.1 Neuron Data Type .................................. 14
    2.2.2 Layer Data Type .................................. 16
    2.2.3 Network Data Type ................................ 18
    2.2.4 Network Validation and Error Checking ............... 19
  2.3 Network Training ........................................... 22
  2.4 Display Functions .......................................... 25
  2.5 Test Data Generation ....................................... 26

3 The Neural Network Transform .................................. 31
  3.1 Initial Definition ........................................... 31
  3.2 Test Data .................................................... 32
  3.3 Unconstrained Neural Network .............................. 35
    3.3.1 Advantages ........................................ 35
    3.3.2 Limitations ........................................ 37
  3.4 Incrementally Trained Additive Networks ................. 39
    3.4.1 Limitations ........................................ 41
  3.5 Revised Definition ......................................... 43
  3.6 Incremental Networks With Prior Coefficients ............ 43
    3.6.1 Limitations ........................................ 45
  3.8 Non-Incremental Networks With Modified Inputs .......... 45
    3.8.1 Limitations ........................................ 49
    3.8.2 Improvements ...................................... 52
4 Final Implementation and Results

4.1 Incremental Network with Modified Inputs .............................................. 55

4.2 Performance Assessment ........................................................................... 58

4.2.1 Test Data Performance ......................................................................... 58

4.2.2 SATIGR Temperature Profile Data ....................................................... 63

5 Conclusions .................................................................................................. 69

5.1 Summary of Thesis .................................................................................... 69

5.2 Suggestions for Future Research ................................................................. 71

A Information From SATIGR Testing ............................................................... 73

B Parameters Used in Generating Test Data ...................................................... 83

C List of Selected Software .............................................................................. 85

References ........................................................................................................ 115
List of Figures

2.1 Neural network notation ................................................................. 15
2.2 Neural network data hierarchy ......................................................... 17
2.3 Example of extended neural network architecture ......................... 20
2.4 Illustration of classification of layers into levels ......................... 22
2.5 Algorithm for generating test data sets ........................................ 28
2.6 Illustration of mapping distribution in one dimension to curve $y=x^2$ ......................... 30
3.1 Mesh plots of generator surface used for test data set 1 .................. 33
3.2 Mesh plots of generator surface used for test data set 2 .................. 34
3.3 Unconstrained neural network reconstruction of test data set 1 ........ 36
3.4 Unconstrained neural network ...................................................... 37
3.5 Training and incremental additive network .................................... 38
3.6 Example of optimal one-coefficient neural network reconstruction which does not lie within optimal two-coefficient surface ............................................. 40
3.7 Dependence of residual error on first neural network coefficient for test data set 2 ......................... 42
3.8 Training an incremental network with prior coefficients ................. 44
3.9 Dependence of second neural network coefficient on first neural network coefficient for test data set 2 using incremental network with prior coefficients ......................... 46
3.10 Training non-incremental network with modified inputs ................ 48
3.11 Non-incremental neural network with modified inputs applied to test data set 1 ........ 50
3.12 Non-incremental neural network with modified inputs applied to test data set 2 ........ 51
3.13 Non-incremental neural network with modified inputs applied to test data set 1 illustrating sub-optimal second coefficient reconstruction ......................... 54
4.1 Training an incremental network with modified outputs

4.2 First neural network coefficient computed with non-incrementally trained network versus first neural network coefficient computed with incrementally trained network

4.3 Neural network reconstruction of test data set 1

4.4 Neural network reconstruction of test data set 2

4.5 Components of test data set 2 versus neural network coefficient 1

4.6 Percentage of data variance captured versus number of KLT coefficients

4.7 KLT coefficients plotted versus neural network coefficients

4.8 Percent variance capture versus number of coefficients used

A.1: Plots of percent variance capture versus number of coefficients used for different overall reconstructions

A.2: Plots of KLT coefficients 1-4 versus neural network coefficient 1

A.3: Plots of KLT coefficients 1-4 versus neural network coefficient 2

A.4: Plots of KLT coefficients 1-4 versus neural network coefficient 3

A.5: Plots of KLT coefficients 1-4 versus neural network coefficient 4

A.6: Plots of neural network coefficients versus other neural network coefficients

A.7: Plots showing neural network first-coefficient only reconstruction for four-coefficient overall reconstruction. Plots are overlaid with KLT points

A.8: Plots showing neural network first-coefficient only reconstruction for four-coefficient overall reconstruction. Plots are overlaid with KLT points

A.9 TIGR data information
List of Tables

4.1 Percentage of data variance captured for test data set 1........................................61
4.2 Percentage of variance captured for test data set 2.............................................61
4.3 Percentage of variance captured for SATIGR dat set............................................65
B.1 Data used to generate test data sets........................................................................84
Chapter 1

Introduction

Neural networks provide an interesting and powerful tool for performing data analysis, as their ability to capture nonlinear relationships provides a way to work with data which is difficult to analyze with linear methods. Neural networks are utilized in a wide range of areas, including control systems, signal processing and artificial intelligence.

This thesis represents an attempt to utilize the ability neural networks to capture nonlinear data relationships to perform data reduction and analysis similar to that accomplished with linear methods, most notably the Karhunen-Loeve transform. The Karhunen-Loeve transform performs optimally in a sum-squared sense for data with jointly gaussian statistics. The KLT performs sub-optimally, however for data with higher order statistics. By harnessing the nonlinear abilities of neural networks, the neural network transform is an attempt to provide improved performance over linear methods in dealing with nonlinear data.

The thesis is divided into four main sections. In chapter two, the tools used to investigate and develop the neural network transform are presented. Descriptions are
given of the commercial hardware and software platforms on which the work was done, and a detailed description is made of the custom software which was developed for this thesis. A software package providing an abstract data hierarchy for dealing with neural networks is described, along with software developed to generate data for testing the performance of different approaches to the neural network transform.

Chapter three introduces the concepts behind the neural network transform and provides the first attempt at establishing a definition of the transform. The results of several attempts at implementing a neural network transform are discussed, and the information learned from these first attempts lead to a revised definition of the transform. Using this revised definition, an approach is developed which performs in the desired manner.

Chapter four describes the final implementation of the neural network transform, developed by using a combination of the methods presented in chapter three. This final implementation is then tested on two test data sets, and its performance is compared with that of the Karhunen-Loeve transform. The neural network transform is then applied to a set of atmospheric temperature profiles and the neural network performance is again compared with that of the KLT.

Chapter five summarized the results presented and provides thoughts for further work in the area of the neural network transform. Appendix A provides a collection of data generated in performing the neural network transform on the atmospheric data. Appendix B provides specific information used in generating the test data sets used in the
thesis. Appendix C contains a listing of the functions written to implement the final neural network transform approach.
Chapter 2

Hardware and Software

A description of the commercial hardware and software packages used in this thesis is presented in this chapter. A customized software package for generating, training, and simulating neural networks is discussed. Also presented is a method for improving the memory performance of the Levenberg-Marquardt training algorithm. Software for generating data used for testing neural network behavior in performing compression and reconstruction is discussed.

2.1 Commercial Hardware and Software

The work with neural networks in this thesis was done on a pair of Hewlett-Packard 715/75 Apollo workstations. These workstations operate under HP-UX 9.05 and are equipped with sixty-four megabytes of RAM. Additional work was done on a Dell 450DE/2 personal computer running Microsoft Windows version 3.1.

The software used for modeling neural networks was created using The MathWorks' MATLAB software package for both UNIX and Windows. MATLAB
version 4.2 for UNIX and MATLAB version 4.2c for Windows were used. Additional software libraries were also utilized in developing the neural network transform. The MATLAB Neural Network Toolbox version 2.0 for UNIX and version 2.0a for Windows, as well as the MATLAB Statistics Toolbox version 1.0 for UNIX were used in writing code for neural network simulation.

The code provided in the MATLAB Neural Network Toolbox provides a solid core of functions for working with neural networks, but the functions provided fall short in some areas necessary for developing a neural network transform. The given functions only allow neural networks with up to two hidden layers. This proves too restrictive for doing data compression. In addition, the MATLAB neural network toolbox functions allow only simple network architectures in which each layer is fully connected to only one other layer after it in the network.

Another limitation of the neural network toolbox software is a lack of generality and modularity. For example, each neural network function requires each weight matrix, each bias matrix, and each transfer function as arguments. This approach provides no abstract representation of the network or its elements, or any data hierarchy for dealing with the networks.

2.2 Customized Neural Network Software

The first step in creating a software package for simulating neural networks is
creating a hierarchy of abstract data types. A three-level hierarchy is used, creating abstract types for networks, layers, and neurons. Each data type is comprised of a data structure storing the information relevant to that data type, and functions which allow access to and manipulation of this data. A partial listing of code is included in Appendix C.

In this thesis, the notation used in figures describing network topologies will be similar to that found in the MATLAB Neural Network Toolbox User’s Guide. Layers of neurons will be represented as a box containing a graphical representation of the activation function of the neurons of the layer. Multiple inputs to a layer will be represented by a line with a slash, along with a number reflecting the number of layer inputs present.

2.2.1 Neuron Data Type

The neuron data type is the simplest structure, containing three pieces of information. The neuron is comprised of a weight vector, a bias, and an identification (id) number. The weight vector and bias are the computational elements of a neuron, while the id number provides a tag to the neuron identifying it within its layer. This is included so that if a layer is modified by the removal or addition of new neurons, then the rest of the network interconnections will not have to be adjusted.

Without id tagging, if a layer is adjusted by removing a neuron or multiple neurons, then the number of outputs from the layer changes. As a result, the output numbers associated with each neuron also change. Throughout the entire network every
Figure 2.1: Neural network notation. (a) Explicit representation of all network elements. (b) Abbreviated notation.
connection to the modified layer must be updated to reflect the new relative neuron positioning. This is time-consuming and inefficient. If each neuron has a unique id tag, then layer modification requires only that those connections to the newly added or removed neuron be updated, which is a much simpler task. The neuron type has functions for creation, changing and accessing its data, as well as for adding and removing inputs.

2.2.2 Layer Data Type

The next abstract level in the neural network hierarchy is the layer data type. A layer is defined as a group of neurons which all share the same inputs. The layer type is comprised of a weight matrix, a bias vector, a transfer function, a connection matrix, a neuron id vector, and a level number. The weight matrix is simply the weight vectors of the layer neurons arranged as the columns of a matrix. The bias vector is likewise the individual biases of the layer's composite neurons arranged as a vector.

The remaining three elements require some explanation. The connection matrix stores the source of the inputs to the layer. Each input has a layer and neuron id associated with it describing its source. The network connections are done in this manner because, due to the architectural flexibility allowed, each output of a layer may be connected to multiple inputs throughout the network. Thus storing the output connections is made impractical due to the fact that the number of connections to a given output of a layer may vary. Each input, however, always has only one source.
Network
- Layers
- Output Connections
- Status Flags

Layer
- Neurons
- Input Connections
- Level Number
- ID Tag

Neuron
- Weights
- Bias
- ID Tag

Figure 2.2: Neural network data hierarchy
The neuron id vector lists the neurons contained in the layer. By using this information, the correct connections between layers can be made, and errors in constructing the network can be detected.

Lastly, the level number is an index which identifies where in the network the layer lies. The need for this indexing will become apparent when the tools for sorting and debugging a network are presented below in section 2.2.4.

Functions also exist for adding, removing and modifying neurons in a layer. In addition, functions exist for modifying the layer connections and for accessing and modifying the layer data. Access to and modification of layer weights and biases can be done directly in terms of matrices or can be done using neuron modification functions. Allowing direct modification of the data weakens the abstraction barrier, but makes using the functions easier.

### 2.2.3 Network Data Type

The last level of abstraction is the network data type. A network is composed of a group of layers, a list of network input and output connections, and a collection of status flags. The network also has several utility functions for data access and manipulation.

A network is created by first generating a group of layers, each with lists of network connections. These layers are then given as arguments to a function which
generates the network. The next step in creating a network is to set the input and output network connections. These connections establish the layers to which the inputs and outputs of the network are connected.

The standard MATLAB neural network utilities restrict the topology of the neural networks which can be constructed. The software developed for this thesis allows far more general networks to be constructed and tested. For example, the standard MATLAB package allows only three-layer networks, and allows only consecutive layers to be connected together. Using the programs developed for this thesis, networks with an arbitrary number of layers can be constructed. In addition, networks can be constructed which contain layers in parallel along with connections which skip layers.

2.2.4 Network Validation and Error Checking

Utility functions exist for checking the network for errors such as invalid connections or recurrent loops in the network. The first step in debugging a network architecture is to ensure that all of the network connections are valid. A connection is invalid if it references a layer output which does not exist, or if it references no layer output at all. A utility function exists which generates a display of the validity of the connections in the network. This function displays information for all of the layer inputs as well as for the network’s input and output connections.
Figure 2.3: Example of extended neural network architecture. (a) Most complex architecture allowed with standard MATLAB Neural Network Toolbox 2.0. (b) Example of extended architecture allowed with custom software package Network has skipped layers, parallel layers, and outputs from multiple layers.
The next step in validating a network is to check for recurrent loops in the architecture and then to sort the network into an ordered set of layers. A function exists which recursively searches through the network finding all loops in the architecture. If any loops are found, the program stops and displays them as a list of layers and outputs. If no loops are found, then the function sorts the layers of the network. The function ranks each layer with a level number indicating where it falls in the network. The rank is determined based on the data that a particular layer needs in order to compute its output. All layers which have the same level of data dependency fall in the same level.

The function begins by finding layers whose inputs come solely from the network inputs. It is clear that at least one such layer must exist. When these layers are found, they are assigned the level one. The network inputs have level zero. The function then searches the remaining layers for those which have inputs coming only from the network inputs and the level one layers. These layers are assigned a level of two. This process continues, with the function searching for layers whose inputs depend only upon layers with a level less than that of the current level. With the network stratified into layers, training and output computation can be performed. (See nt_sort in Appendix C)

The software presented represents an attempt to create a package which allows new and unusual network architectures to be tested. Due to the memory overhead associated with the generality of this code, it is much slower than customized functions
designed for a specific architecture. Thus, this software is intended for use primarily as an experimental tool. Once the validity of a certain architecture is established, then dedicated training functions can be written which perform better in terms of memory use and execution time.

### 2.3 Network Training

The algorithm used in this thesis for training neural networks is the Levenberg-Marquardt algorithm. The algorithm is of the form:
\[ \Delta W, B = [J^T J + \mu I]^{-1} J^T e \] (2.1)

where \( J \) is the Jacobian matrix containing the derivatives of the error in each output with respect to each weight and bias. The variable \( e \) is the error vector for the current network formed by taking the error vectors for all output vectors and appending them into a single vector. The variable \( \mu \) is a training parameter which serves to modify the learning rate of the training algorithm. It combines the effects of the learning rate and momentum variables in backpropagation training algorithms. When \( \mu \) is small, the algorithm approximates Newton's method. As \( \mu \) increases, the algorithm approaches standard gradient-descent backpropagation.

The problem with using this algorithm is that it requires large amounts of memory. This can become a problem with high-dimensional data sets and with large network architectures. The algorithm provides a difference in convergence properties such that it performs better than standard backpropagation when doing compression and reconstruction. Therefore, code was developed which uses less memory by taking advantage of the nature of the computations done in the algorithm. This allows the Levenberg-Marquardt algorithm to be used with data sets and network architectures which would otherwise require too much memory.

The main memory usage in the Levenberg-Marquardt algorithm is in the calculation of the Jacobian matrix of the derivatives of error. This matrix has \((V \times N \times (W+B))\) elements, where \( V \) is the number of output variables, \( N \) is the
number of training vectors, \( W \) is the number of weights in the network, and \( B \) is the number of biases in the network. Consider a two-layer network containing five nodes in its hidden layer and having five inputs. For an eight-variable set of eight hundred vectors, training would require a Jacobian matrix with \( 8 \times 800 \times (65+13) = 499,200 \) elements. Assuming that all elements are floating point numbers which require eight bytes of memory in MATLAB, then the Jacobian matrix alone would require 3,993,600 bytes of memory.

This memory requirement can be reduced by noticing that if the input and target vectors are divided into subgroups, or blocks, then the modified weights and biases can be calculated via:

\[
\Delta W, B = (J^T J + \mu I)^{-1} J^T e = \left( \sum_{i=1}^{N} J_i^T J_i \right)^{-1} \sum_{i=1}^{N} J_i^T e_i
\]

(2.2)

Where \( e_i \) is the \( i \)th block of error vectors, and \( J_i \) is the Jacobian matrix corresponding to \( e_i \). Using this approach, only the smaller sub-Jacobian matrices need to be calculated, and only one needs to be stored at a time. The other values \( J^T J \) and \( J^T e \) require only the current value and the partial sum to be stored. These values depend only on the size of the network, and not on the number of test vectors, and thus will not cause the same memory problems as does the full Jacobian matrix. This algorithm allows the use of the Levenberg-Marquardt training algorithm when otherwise memory constraints would
require the use of standard backpropagation algorithms. (See inc_tr8c in Appendix C.)

2.4 Display Functions

Modifications to the standard plotting functions were also implemented in order to better monitor and evaluate network training. One modification allows a superimposed plot of the mean-square errors for both a training and a test set of vectors. This makes the detection of overtraining easier. Another addition was the inclusion of plots of the target data overlaid with the network reconstruction of the target data. This allows better understanding of how the network is capturing the nature of the data. For three-dimensional data, full three-dimensional plots are used, while for higher-dimensional sets, a series of two dimensional projections is used. An additional plot displays a histogram of the values of the current neural network coefficient being trained.

In the course of working with data distributions in three dimensions it is necessary to be able to view the three-dimensional shapes from a variety of angles in order to determine the nature of the distribution and in order to determine the quality of the neural network reconstruction. The commands for doing this in MATLAB are cumbersome, and thus a function was written which provides a graphical user interface for viewing three-dimensional objects from different angles. This aids in error analysis and assessment of network performance. (See vmenu in Appendix C.)
2.5 Test Data Generation

In order to test the performance of the neural network in doing compression and reconstruction, a program was developed to generate three dimensional data sets based upon an underlying two dimensional shape. Knowledge of the two-dimensional surface used to generate the data allows a valid performance assessment since the network reconstruction can be compared with the generator surface. The test data sets were restricted to three dimensions in order to facilitate visualization of the data and of the network reconstruction.

The data generator creates a three dimensional object based on functions specified by the user. The data set follows a curve in three dimensions which is defined by the user as a set of three parametric equations in one variable. The range of this variable is determined by the user. This three dimensional parametric curve is referred to as the baseline. Along the baseline, the data is distributed according to a cross-section function. This function is input by the user as a function of a single variable, the range of which is also input by the user.

The user chooses the number of points to be generated along the baseline, and chooses the number of points to be generated along each cross-section. One point per cross-section yields data which is distributed uniformly, and this is what was used in all test cases here.
The test data function first generates a set of cross-sections, each with a given number of points along its curve. Next, the function moves the points away a random distance from the cross-section in the direction perpendicular to the curve at the given point. The variance of this distance can be varied through the use of an input parameter. The axes for the modified cross-section are then rotated such that the y-axis is mapped to \( \hat{r} \), where \( \hat{r} \) is the unit length direction vector associated with the current baseline point, and such that the x-axis is mapped to the direction of the cross-product of \( \hat{r} \) and \( d\hat{r} \) where \( d\hat{r} \) is the direction of the derivative of \( \hat{r} \) at the current baseline point. The origin for the cross-section function is then shifted to the current point along the baseline. This generates a three-dimensional shape which is based on the user-specified two-dimensional surface. To aid in the comparison of the network reconstruction and the original generator, the function produces a mesh grid showing the shape used to generate the data.

A problem with generating random points along an arbitrary parametric line is that a uniform distribution of points in the parametric variable often maps to a non-uniform distribution along the parametric curve. In order to overcome this, a function was written which maps an arbitrary distribution in one dimension to the correct values in the parametric variable such that the original distribution is mapped along the parametric curve.

This function first generates a set of values deterministically distributed within the user-defined range of the parametric variable. The input parameter space is thus quantized
Figure 2.5: Algorithm for generating test data sets. (a) Random distribution is mapped to cross-section function. (b) Points are moved away perpendicular to the cross-section curve. (c) Cross-section axes are rotated. (d) Cross-section is shifted to baseline.
into a series of equally spaced points. These points are then mapped via the parametric equations onto the curve. Next, the distance from each point to the next along the parametric curve is approximated by linear interpolation. These incremental distances are then summed to yield an absolute distance along the parametric curve for each of the points in the input space. The function then calculates the total length of the curve and scales the original distribution to make it be distributed over the same length. Next, the function takes the random distribution which is to be mapped on to the curve and calculates the distance in one dimension of each point in the random distribution from the minimum value in its range. The function then finds the value in the quantized parametric range which has a distance along the curve closest to the distance associated with the point in the original distribution. Each point in the original distribution is thus mapped to the quantized parametric value which yields the point along the curve which best approximates the mapping of the original distribution on to the curve. (See rand3d in Appendix(C.)
Figure 2.6: Illustration of mapping distribution in one dimension to curve $y=x^2$. (a) Original distribution. (b) Modified distribution. (c) Original distribution used as input to cross-section function. Notice spatial warping as derivative of curve increases. (d) Rotated cross-section is shifted to baseline.
Chapter 3

The Neural Network Transform

The neural network transform is defined in this chapter. Several methods for implementing the neural network transform are discussed, along with advantages and limitations to each approach. A revised definition of the neural network transform is developed based upon observation of the performance of the first attempts at implementing the transform. A discussion of the data used to test the performance of the network is also presented.

3.1 Initial Definition

The first step in developing a neural network transform is to establish the specific criteria that the transform should meet. First and most importantly, the neural transform should reduce the dimensionality of a data set in such a way that minimal information is lost. The performance of the neural network in capturing the information contained in the data is measured in terms of the sum-squared error between the neural network reconstruction and the original data. The sum-squared error criterion is chosen because
this is the criterion used by feedforward neural network training algorithms.

In addition to minimizing the reconstruction error, the neural network transform should behave in a manner similar to that of the Karhunen-Loeve transform, in that the neural network should maximize the signal energy of the reconstruction in its first coefficients. When reconstruction is performed with the first coefficient alone, then the reconstruction error should be the minimum that can be obtained from only one coefficient. Likewise, the reconstruction of the original data should be optimal for each successive coefficient.

By maximizing the signal energy of the data in successive coefficients, the transform will not only produce a reduction in dimensionality, but more importantly will provide information about the signal energy in the data and the relationships between the data variables. In essence, the neural network transform should behave as a non-linear Karhunen-Loeve transform, gaining improved performance over the KLT via its ability to capture non-linear relationships and higher order statistics.

The neural network transform coefficients should be computed using standard feedforward neural networks which have the original data as their input, while the data reconstruction should be done using feedforward neural networks which have only the neural transform coefficients as inputs.

3.2 Test Data

Two test data sets were used in evaluating the performance of the different
Figure 3.1: Mesh plots of generator surface used for test data set 1.
Figure 3.2: Mesh plots of generator surface used for test data set 2.
approaches to the neural network transform. These data sets were created using the software described in section 2.5. Each data set consists of nine hundred vectors. Six hundred vectors are used as a training set and three hundred vectors are used as a test set. These data sets were created to exhibit nonlinear variations along both the baseline and the cross-section. The first set has a fairly simple main line, while the second set exhibits a more complex baseline shape.

3.3 Unconstrained Neural Network

The first and most logical attempt at a neural network transform is to take a standard neural network and make the training data set the same as the input data set. Then, at some point within the network, reduce the number of connections between subsequent layers to a number less than the dimensionality of the input data. The network will thus be performing compression as it is trained to fit the original data.

Research has been done on this algorithm for performing compression. [Cottrell, et al., 1987; Bourland and Camp, 1988] Much of the research in this area has been focused on two-layer linear networks which have been shown to perform a mapping into the subspace spanned by the first $N$ principal components of the data. [Baldi and Hornik, 1989]

3.3.1 Advantages

This approach has several advantages. The first is that the network topology is
Figure 3.3: Unconstrained neural network reconstruction of test data set 1.
Figure 3.4 Unconstrained neural network.

very simple and the standard training algorithms for neural networks can be applied with little or no change. In terms of network size and memory usage, this approach scales well as the dimensionality of the input data set increases.

3.3.2 Limitations

This approach fails to meet the criteria established for the neural network transform. First, the network performs no ordering of signal energy among coefficients. In the three-dimensional case, the network reduces the data to at most two coefficients. By holding one coefficient constant and varying the other, we can obtain a two-dimensional grid illustrating how the coefficients map into three dimensions. This is displayed in figure 3.3. As is clear from the figure, the signal energy of the reconstruction is not maximized in either coefficient. The network is not constrained in any way to maximize signal energy in one coefficient, and thus there is no reason to expect this behavior. The first step in developing a neural network transform is therefore to force the network to maximize the signal energy of the input data set in its first coefficients.
Step 1: Train First Coefficient

Step 2: Train Second Coefficient

Step 3: Add Outputs

Figure 3.5: Training an incremental additive network.
3.4 Incrementally Trained Additive Networks

The first step in accomplishing this goal is to notice that a neural network is forced to maximize signal energy when it is reduced to only one coefficient using a sum-squared error criterion. Therefore, the first step in achieving signal energy maximization is to first train a neural network to reconstruct the original data using only one coefficient. The next step is to subtract this first neural network reconstruction from the original data, yielding a residual error. This residual error is then used as the target data set for a new network which has only one coefficient and which uses the original data as its input. The output from this second network is then added to the output from the first network in order to get the complete reconstruction. (See figure 3.5) This process is repeated for higher-dimensional data sets, with the reconstruction from each coefficient subtracted from the residual error, and with the successive reconstructions summed to produce the composite reconstruction.

3.4.1 Limitations

This approach reveals some unforeseen problems in performing a neural network transform. The first problem is that the best single coefficient fit is not necessarily contained within the data set which is being analyzed, as is illustrated in figure 3.6. The neural network is operating solely on a sum-squared error minimization criteria. Thus the network finds the best fit in a sum-squared sense, regardless of whether the fit lies within the data set.
One Coefficient Reconstruction Overlay with Data Generator

Figure 3.6: Example of optimal one coefficient neural network reconstruction which does not lie within optimal two-coefficient surface. Figure shows one coefficient incremental network reconstruction of test data set 1.
This illustrates that the best one-coefficient neural network fit to the data does not necessarily lie within the best two-dimensional, nonlinear neural network subspace. This inconsistency arises from our initial definition of the neural network transform. By forcing maximal compression within successive coefficients, then a sub-optimal overall fit may sometimes result. Thus, the definition of the neural network transform must be revised.

The second problem encountered in using this approach is more subtle, but likewise warrants a revision in our initial definition of the neural network transform. Looking at the generator surfaces for both test data set 1 and test data set 2 (figures 3.1 and 3.2, respectively), it is clear that the cross-section of each of the curves changes in orientation along its baseline.

Consider the first residual error, which is calculated by subtracting the first neural network reconstruction from the original data. It is clear that this residual error will exhibit a dependence upon the first neural network coefficient. This is illustrated in figure 3.7. This dependence arises from the relationship of cross-section orientation to baseline position. This illustrates the fact that, due to the non-linear nature of the neural network transform, subtracting a neural network reconstruction from the original data does not remove the dependence of the data on the coefficient used to create the reconstruction. Thus, when successive coefficients are trained using the residual errors resulting from previous coefficient reconstructions, the later coefficients must themselves depend upon previous coefficients in order to capture the energy in the residual error. As before, there
Figure 3.7: Dependence of residual error on first neural network coefficient for test data set 2.
is nothing in our initial definition of the neural network transform which characterizes this as undesirable. Thus our original definition must be revised.

3.5 Revised Definition

The new criteria for the transform must take into account the problems discussed in the last section. First, the optimal fit for a given number of coefficients is not necessarily the same as that for a different number of coefficients. Second, the issue of efficiency in the coefficient space must be addressed.

The first new criterion for the neural network transform states that the transform should find the optimal fit for a given number of coefficients, then order the signal energy within the subspace, maximizing signal energy in successive coefficients. This criterion implies that the neural network transform can be considered unique only for a given dimensionality in overall reconstruction.

The second new criterion for the neural network transform states that successive coefficients should not contain information already captured by previous coefficients. The transform coefficients should be efficient in the sense that no redundant information is contained in the coefficient set.

3.6 Incremental Networks With Prior Coefficients

Using this revised definition, the first approach to the neural network transform
Step 1: Train First Coefficient

Step 2: Train Second Coefficient

Step 3: Add Outputs

Figure 3.8: Training an incremental network with prior coefficients.
utilizes much the same methods as the previous approach, except that the second reconstruction network is now given the first coefficient as an additional input. (See figure 3.8) The first compression and reconstruction networks are held fixed, and only the second coefficient is trained. The idea is that in order to optimize reconstruction of the residual error the neural network will find a second coefficient space which has no dependence on the first coefficient, due to the fact that the first coefficient is already provided as an input to the second reconstruction level of the network.

3.6.1 Limitations

Despite this change in the architecture of the network, this method produces the same errors as when the first coefficient is not provided as an input. Figure 3.9 illustrates the strong dependence of the second coefficient on the first coefficient when a network is trained in this manner. It is clear that the network is given no reason to remove the redundant information in the second coefficient when trained using this method, thus this behavior is not to be expected. The solution to this problem is to modify the input and target data sets when training the network.

3.8 Non-Incremental Networks With Modified Inputs

To eliminate many of the problems encountered thus far, the neural network must be given the information that the signal energy of the data must be maximized in each coefficient. The way to accomplish this without modifying the basic training algorithm is
Figure 3.9: Dependence of second neural network coefficient on first neural network coefficient for test data set 2 using incremental network with prior coefficients.
to modify the input and target training sets. Consider the case of the unconstrained neural network presented in section 3.3. One reason that the neural network does not perform signal energy maximization in this case is that the network is trained in such a way that all of the inputs to the reconstruction level are changing with every vector. In three dimensions we desire that when the second coefficient is held constant, that the first coefficient alone can be used to perform the optimal reconstruction for one variable within the best two-dimensional surface. The network is not trained on data in which one coefficient is held constant, however. Therefore the network has no reason to converge to such a solution.

A solution to this problem is to expand the input and target training sets to reflect the case in which a limited number of coefficients is being used in reconstructing the input data. For simplicity we shall consider only the two-coefficient case. The first step in implementing this strategy is to separate the compression levels for each of the coefficients so that the inputs for each can be modified independently. This is done by having separate networks for calculating each coefficient. The next step is to create a data set in which the first coefficient is varying and the second is held constant. The way to accomplish this is to create an input set which consists of two duplicates of the original data as inputs to the first coefficient compression level, and which contains the original data and a set of zero vectors as inputs to the second coefficient compression level. The output data set consists of two copies of the original data.
Figure 3.10 Training non-incremental network with modified inputs.

This network is able to converge to the desired solution because it is given two goals in training. The first goal is to minimize the sum-squared error of the output. This results from the fact that the training algorithm is based on reducing the sum-squared error of the output. The second goal is to maximize the signal energy in the first coefficient. This goal is supplied to the network implicitly by the modification of the input and target sets. Because a constant input of zeros is given to the second coefficient branch of the network, then for half of the training period the neural network is reconstructing using only the first coefficient. Because the errors from the first and second coefficient reconstructions are summed, the network will maximize the signal energy in the first coefficient in order to minimize the overall sum-squared error.

By modifying the input and output training sets, the network is also given information implicitly which prevents the inefficiencies in the coefficients found in previous methods. Because the inputs to the compression networks for each coefficient
are separate, they can be modified independently. In this algorithm, the second coefficient
is held constant over a portion of the training set, independent of the first. This
independent modification of the coefficients propagates to the reconstruction network.
The reconstruction network is thus trained on a data set where the first and second
coefficients have independent properties due to the modified inputs. The second
coefficient is therefore determined such that it reflects the independence between the
inputs, resulting in an efficient coefficient space. (See figures 3.11 and 3.12).

3.8.1 Limitations

The problems with using this method concern the size of the network and the
memory required to train the network. As additional coefficients are added, additional
copies of the input and output data must be made, along with zero matrices for each case
where a coefficient is to be held constant. The size of the training data therefore grows
quickly as the number of coefficients is increased. This not only affects the memory
needed to store the actual input and output data sets, but as discussed in section 2.3
affects the memory required for training the network. The size of the Jacobian matrix
needed to implement the Levenberg-Marquardt training algorithm depends upon the
number of output variables, which in this case grows linearly with the number of
coefficients.

Due to the large memory requirements of the training algorithm, this method takes
a long time to converge. In addition, these large memory requirements makes using this
method difficult for large numbers of coefficients and large data. The algorithm.
Figure 3.11: Non-incremental neural network with modified inputs applied to test data set 1. Network here trained using reduced coefficient block to eliminate first coefficient bias. (a) coefficient distribution (b) Mesh plot.
Figure 3.12: Non-incremental neural network with modified inputs applied to test data set 2. Network here trained using reduced coefficient block to eliminate first coefficient bias. (a) coefficient distribution (b) Mesh plot.
discussed in section 2.3 for breaking the input and target sets into separate blocks when using the Levenberg-Marquardt training algorithm allows this method to be used more extensively, but the training times quickly become prohibitive.

This method also produces an error which results from the case where the optimal first coefficient reconstruction is not contained in the optimal two-coefficient surface. Because the size of the first coefficient training block is the same size the second coefficient training block, the errors resulting from the first coefficient block dominate the training. This results in a first coefficient reconstruction which is optimal, but a second coefficient which is sub-optimal. (See figure 3.13)

### 3.8.2 Improvements

The solution to this problem again lies in modifying the input and target data sets. The problem arises from the dominance of the first coefficient error in the training set. The solution is to reduce the size of the first coefficient training block. In the previous method the first coefficient training block is the same size as the original data. If we reduce the size of this block such that the errors associated with the second coefficient training block are more significant in training the network, then the network will find the best two-coefficient reconstruction and the first coefficient fit will be optimized within this subspace.

This approach also has the effect of reducing the memory requirements of the algorithm. Using this approach, only a subset of the input data needs to be presented for
the first-coefficient training. The first coefficient training block should be chosen such that the vectors contained in it span the data space reasonably well in order to ensure a good first coefficient reconstruction. This approach overcomes the biasing of the optimal one-coefficient reconstruction from the optimal two-coefficient surface and yields the desired solution.

The remaining problems with this method are the large memory requirement and the length of time needed to train the network. By a combination of this last method and previous methods, a solution can be found which improves performance in both of these areas.
Figure 3.13: Non-incremental neural network with modified inputs applied to test data set 1. Using full first coefficient training block leads to sub-optimal two-coefficient reconstruction.
Chapter 4

Final Implementation and Results

The final approach for performing the neural network transform is presented in this chapter. Results of applying this method to the test data sets are discussed. The neural transform is applied to a data set of atmospheric temperature profiles, and a comparison of the performance of the neural network to that of the Karhunen-Loève transform is presented.

4.1 Incremental Network with Modified Inputs

The final approach for implementing the neural network transform is a combination of the non-incremental style network with modified inputs and the incremental neural network. In the final approach, the first coefficient in the network is trained alone on the data as in the incremental neural network approach presented in section 3.4. Then, the second coefficient network is trained with the first coefficient fixed and given as an input to the reconstruction level. The second coefficient is not trained on the residual error from the first reconstruction, but instead on the original data. In addition, the input and output sets are modified as in section 3.8.3 in order to force the network to maximize the
Step 1: Train First Coefficient

Step 2: Train Second Coefficient and Reconstruction

Figure 4.1 Training an incremental network with modified outputs.

signal energy in the first coefficient. (See figure 4.1) This approach reduces the number of weights and biases which must be trained at one time, thereby reducing the training time and the memory requirements associated with the fully non-incremental approach presented in section 3.8

Incremental training of the neural network coefficients is based on the concept that several reconstructions can exist for the same neural network coefficient. When a
Figure 4.2: First neural network coefficient computed with a non-incrementally trained network versus first neural network coefficient computed with an incrementally trained network.
neural network which has only one coefficient is trained as in section 3.4, it sometimes produces a reconstruction which is not contained in the optimal two-coefficient surface. The coefficient set determined by the network, however, represents an ordering and classification of the input data which captures its signal energy. This coefficient set can be used to then reconstruct within the optimal two-coefficient surface. Figure 4.2 illustrates that the first coefficient determined by an incrementally trained network and the first coefficient determined by a non-incremental network with modified inputs are essentially identical.

4.2 Performance Assessment

This final approach to the neural network transform was applied to both test data sets to assess its performance versus that of the Karhunen-Loeve transform. The neural network transform was then applied to a set of atmospheric temperature profile vectors and compared with the Karhunen-Loeve transform.

4.2.1 Test Data Performance

Neural network transform analysis was performed upon both test data sets. The results of the testing are summarized in tables 4.1 and 4.2. As shown in the tables, the neural network outperforms the Karhunen-Loeve transform significantly in both cases. This results from the highly nonlinear shapes of both data sets which cannot be mapped well into linear components. As is shown in figures 4.3 and 4.4, the neural network
Figure 4.3: Neural network reconstruction of test data set 1.
Figure 4.4: Neural network reconstruction of test data set 2.
<table>
<thead>
<tr>
<th>Number of Coefficients</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>KLT</td>
<td>0.6260</td>
<td>0.9554</td>
</tr>
<tr>
<td>Neural Network (Training Set)</td>
<td>0.7605</td>
<td>0.9897</td>
</tr>
<tr>
<td>Neural Network (Test Set)</td>
<td>0.7659</td>
<td>0.9889</td>
</tr>
</tbody>
</table>

Table 4.1: Percentage of data variance captured for test data set 1

<table>
<thead>
<tr>
<th>Number of Coefficients</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>KLT</td>
<td>0.4473</td>
<td>0.7812</td>
</tr>
<tr>
<td>Neural Network (Training Set)</td>
<td>0.9748</td>
<td>0.9989</td>
</tr>
<tr>
<td>Neural Network (Test Set)</td>
<td>0.9809</td>
<td>0.9989</td>
</tr>
</tbody>
</table>

Table 4.2: Percentage of data variance captured for test data set 2

performs an efficient mapping in its coefficients.

Figure 4.5 displays a plot of the coordinate components of the second test data set versus the first neural network coefficient found for the set. The parametric equations which were used in creating the data set are clearly reflected in these plots. This illustrates that the neural network was able to find the underlying relationship in the data.
Figure 4.5: Components of test data set 2 versus neural network coefficient. 1. Relationships match functions used to generate data. (a) X component. Baseline equation: \( \cos(T), -3\pi/2 < T < 3\pi/2 \) (b) Y component. Baseline equation: \( \sin(T), -3\pi/2 < T < 3\pi/2 \) (c) Z component. Baseline equation \( T, -3\pi/2 < T < 3\pi/2 \).
and correctly map it to the first coefficient.

4.2.2 SATIGR Temperature Profile Data

Neural network analysis was also performed on a set of atmospheric temperature profiles collected in a data set called the SATellite Initial Guess Retrieval set (SATIGR) [Escobar, 1993]. This collection of temperature profiles is based on radiosonde data collected from latitudes ranging from polar to tropical. Many versions of this set exist, with various numbers of levels interpolated from the original radiosonde measurements. The set used here has 1761 vectors and consists of sixty-four levels of temperature values.

When performing the neural network transform on the SATIGR data, temperature profiles below twenty-five kilometers were used, resulting in a data set of 1761 observation vectors in fifty-two variables. Two thirds of this set was used as the training set and the remaining third of the data was used a test set.

In order to make the problem more tractable for the neural network, the data set was first reduced to eight variables using the Karhunen-Loeve transform. The eight most significant transform coefficients captured 99.51 percent of the signal energy of the data, and thus little or no generality of the data was lost by this reduction. (See Figure 4.6) The neural network was then trained on these eight KLT coefficients, which were normalized to have zero mean before the data set was separated into training and test sets. Each compression level contains two neurons in their hidden layer, and the final reconstruction level has eight neurons in its hidden layer. The second, third and fourth coefficients were
Figure 4.6: Percentage of data variance captured versus number of KLT coefficients used.
<table>
<thead>
<tr>
<th>Number of Coefficients</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>KLT</td>
<td>0.5769</td>
<td>0.8773</td>
<td>0.9414</td>
<td>0.9701</td>
</tr>
<tr>
<td>Neural Network (Training Set)</td>
<td>0.6241</td>
<td>0.8804</td>
<td>0.9428</td>
<td>0.9709</td>
</tr>
<tr>
<td>Neural Network (Test Set)</td>
<td>0.6340</td>
<td>0.8870</td>
<td>0.9448</td>
<td>0.9723</td>
</tr>
</tbody>
</table>

Table 4.3: Percent of variance captured for SATIGR data set.

trained with each having an additional block size of one hundred samples.

The results of applying the neural network transform on the SATIGR data are summarized in table 4.3. From the table we can see that the neural network transform offers better performance for all reconstructions. Figure 4.7 illustrates the relationship between each neural network coefficient and its corresponding KLT coefficient. This shows that the neural network coefficients are closely correlated with their corresponding KLT coefficients. This implies that the neural network is performing much the same mapping as the KLT.

Figure 4.8 shows the amount of signal variance captured by coefficient for the neural network and for the KLT. The neural network performs better for a on-coefficient reconstruction, but as more coefficients are added, the network performance
approaches that of the KLT. This occurs because the neural network can capture nonlinear relationships using one coefficient while the KLT must use multiple coefficients. For the SATIGR data set reconstructions using two or more coefficients, the KLT and the neural network appear to span the same space, but the neural network is able to maximize the signal energy of the data in the first coefficient more effectively.

Figure 4.8 also displays the variance captured by the neural network when reconstructing the data to find the optimal one-coefficient fit. The performance is significantly better that when the neural network is forced to find a higher-dimensional reconstruction. This is the same phenomenon seen in the first test data set, where the optimal one-coefficient fit does not lie within the optimal two-coefficient fit.
Figure 4.7 KLT coefficients plotted versus neural network coefficients. (a) KLT 1 versus neural network 1. (b) KLT 2 versus neural network 2. (c) KLT 3 versus neural network 3. (d) KLT 4 versus neural network 4.
Figure 4.8: Percent variance capture versus number of coefficients used.
Chapter 5

Conclusions

In this thesis an approach for utilizing feedforward neural networks in data reduction and analysis has been presented. A software package has been developed which creates a data hierarchy for dealing with neural networks. This software provides a flexible environment in which experimental network architectures can be designed and tested. An investigation into different approaches to implementing a neural network transform has produced results which reveal information about performing data reduction with neural networks. A definition of the neural network transform has been presented, and a method for implementing the transform has been demonstrated. Testing this implementation of the neural network transform has shown results superior to that of the Karhunen-Loeve transform in several instances.

5.1 Summary of Thesis

A software package for creating and testing experimental neural network architectures was presented in Chapter 2. This software creates a hierarchy of abstract
data types for working with neural networks. This software provides greatly expanded flexibility over the standard neural network functions found in the MATLAB Neural Network Toolbox 2.0. An algorithm was also demonstrated for reducing the memory requirements of the Levenberg-Marquardt training algorithm, allowing its use with large network architectures and large data sets. Software for generating test data sets for neural networks was also presented. The test data generation program uses user-defined parametric equations to allow complex three-dimensional data sets for use in testing the performance of neural networks.

In Chapter 3 a definition of the neural network transform was proposed. The use of unconstrained networks and incrementally trained additive networks for implementing the neural network transform were investigated. Unconstrained networks were found to lack the property of signal energy maximization, while incrementally trained additive networks demonstrated the dependence of residual errors on previous coefficients. This dependence led to the concept of efficiency in the neural coefficient space, meaning that no redundant information should be stored in the neural network coefficients. In addition, testing incremental networks demonstrated the fact that the optimal neural network reconstruction for a given number of coefficients need not lie within the optimal neural network space for a higher number of coefficients.

A revised definition of the neural network transform was presented, adding the constraint that the network determine an efficient coefficient space. The revised definition also states that the neural network transform should maximize signal energy within the
first coefficients of the optimal overall reconstruction subspace. Thus the neural network transform is considered unique only for a given overall reconstruction dimensionality. This revised definition led to the use of a non-incremental neural net with modified inputs to implement the neural network transform. It was demonstrated that by modification of the input and target data sets used to train the network, that a non-incrementally trained neural network can be made to perform the revised neural network transform.

In Chapter 4 an approach using an incremental neural network with modified inputs was presented. This method has better training time and memory usage properties than the non-incremental approach presented in Chapter 3. This method was tested on the two test data sets and then on the SATIGR data set. The neural network performance was shown to be superior to that of the Karhunen-Loeve transform for both test data sets and for the SATIGR data. In the SATIGR testing, the neural network performance was shown to approach that of the KLT as more coefficients were used. This implies that the neural network and the KLT are spanning the same reconstruction space, but that the neural network is better able to maximize signal energy in its first coefficient.

5.2 Suggestions for Future Research

Additional work with the neural network transform should seek to investigate methods for quantifying the number of vectors needed as training blocks (As in section 3.8.2) and relating this number to the input data set. A weight and bias initialization algorithm which speeds convergence of a single-coefficient network would prove useful.
and insightful. Additional testing of the neural network transform would also be useful. Applying the neural network transform to different nonlinear physical phenomena would provide a better idea of the ability of this method to capture nonlinear and statistically complex data relationships.
Appendix A

Information From SATIGR Testing

This appendix contains a collection of plots generated from performing neural network transform analysis on the SATIGR temperature profile data set.
Figure A.1: Plots of percent variance capture versus number of coefficients used for different overall reconstructions.
Figure A.2: Plots of KLT coefficients 1-4 versus neural network coefficient 1.
Figure A.3: Plots of KLT coefficients 1-4 versus neural network coefficient 2.
Figure A.4: Plots of KLT coefficients 1-4 versus neural network coefficient 3.
Figure A.5: Plots of KLT coefficients 1-4 versus neural network coefficient 5.
Figure A.7: Plots showing neural network first-coefficient only reconstruction for four-coefficient overall reconstruction. Plots are overlaid with KLT points.
Figure A.8 Plots showing neural network first-coefficient only reconstruction for four-coefficient overall reconstruction. Plots are overlaid with KLT points.
Figure A.6: Plots of neural network coefficients versus other neural network coefficients.
Description of the 1761 atmospheres of SATIGR on 40 pressure levels between surface and 0.05 hPa

This file has the following identification and characteristics:

RNOPO = identification of the atmosphere
from 1 to 322: tropical type atmospheres
from 323 to 710: mid latitude-1 type atmospheres
from 711 to 1064: mid latitude-2 type atmospheres
from 1065 to 1168: polar-1 type atmospheres
from 1169 to 1761: polar-2 type atmospheres

ALAT, ALON, ADAT= respectively latitude, longitude, date of the
nominal radiosounding

T2 = vector of the temperature in k from top to bottom at the
31 pressure levels (40 values) see below

TSO = surface temperature in k

H2O = same as above for h2o content in g/g, (40 values)

O2O = same as above for ozone content in g/g, (40 values)

DH2O = h2o content in one layer, in kg/cm2, (39 values)

SUMDN = cumulative h2o content in successive layers, upper bound
is the top of the atmosphere, (39 values)

The 40 pressure levels, as defined in the 4A (Automated Atmospheric
Absorption Atlas), are given below in hPa, from top to bottom:

level 1 : 0.05    level 2 : 0.09
level 3 : 0.17    level 4 : 0.30
level 5 : 0.55    level 6 : 1.00
level 7 : 1.50    level 8 : 2.23
level 9 : 3.33    level 10 : 4.98
level 11 : 7.43   level 12 : 11.11
level 13 : 16.60  level 14 : 24.79
level 15 : 37.04  level 16 : 45.73
level 17 : 56.46  level 18 : 69.71
level 19 : 86.07  level 20 : 106.27
level 21 : 131.2  level 22 : 161.99
level 23 : 200.    level 24 : 222.65
level 25 : 247.90 level 26 : 275.95
level 27 : 307.20 level 28 : 341.99
level 29 : 380.73 level 30 : 423.85
level 31 : 471.86 level 32 : 525.00
level 33 : 584.80 level 34 : 651.04
level 35 : 724.78 level 36 : 800.00
level 37 : 848.69 level 38 : 900.33

Figure A.9  TIGR data information.
Appendix B

Parameters Used in Generating Test Data

The exact parameters used to create the test data sets are listed in this appendix.
<table>
<thead>
<tr>
<th>Parametric Function in X</th>
<th>Test Data Set 1</th>
<th>Test Data Set 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>cos(T)</td>
<td></td>
<td>cos(T)</td>
</tr>
<tr>
<td>Parametric Function in Y</td>
<td>sin(T)</td>
<td>sin(T)</td>
</tr>
<tr>
<td>Parametric Function in Z</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>Range of Parametric Variable</td>
<td>$-\pi &lt; T &lt; \pi$</td>
<td>$-\frac{3\pi}{2} &lt; T &lt; \frac{3\pi}{2}$</td>
</tr>
<tr>
<td>Cross-section Function</td>
<td>$X^2$</td>
<td>$X^2$</td>
</tr>
<tr>
<td>Cross-section Range</td>
<td>$-2 &lt; X &lt; 2$</td>
<td>$-1.2 &lt; X &lt; 1.2$</td>
</tr>
<tr>
<td>Cross-section Scale Factor</td>
<td>0.5</td>
<td>0.3</td>
</tr>
<tr>
<td>Alpha</td>
<td>0.16</td>
<td>0.096</td>
</tr>
</tbody>
</table>

Table B.1: Data used to generate test data sets.
Appendix C

List of Selected Software

Selected functions developed and used in this thesis are listed.
function [net,outputs,inputs,linmat,loutmat,flag] = nt_chk(net)
%function [net outputs inputs linmat loutmat flag] = nt_chk(net)

flag = 0;
numin = nt_getin(net);
numouts = nt_getno(net);
numlayers = nt_getnl(net);
maxlin = 0;
maxlout = 0;
lid = nt_getlid(net);
lnuminn = zeros(1,numlayers);
lnumout = zeros(1,numlayers);

for p = 1:numlayers
    k = lid(p);
    cl = nt_getlr(net,k);
    lin = lr_getin(cl);
    lout = lr_getln(cl);
    lnuminn(1,p) = lin;
    lnumout(1,p) = lout;
    if lin > maxlin
        maxlen = lin;
    end
    if lout > maxlout
        maxlout = lout;
    end
end

linmat = zeros(maxlin+2,numlayers);
loutmat = zeros(maxlout+2,numlayers);
outputs = zeros(numouts,1);
inputs = zeros(numin,1);
linmat(1,:) = lid;
loutmat(1,:) = lid;
linmat(2,:) = lnuminn;
loutmat(2,:) = lnumout;
from = nt_getin(net,0);

for k = 1:numouts
    if from(k,1) == 0
        if from(k,2) <= numin & from(k,2) > 0
            inputs(from(k,2),1) = 1;
            outputs(k,1) = 1;
        else
            outputs(k,1) = -3;
            flag = 1;
        end
    else
        lrind = nt_idcklk(net,from(k,1));
        if lrind <= 0
            layer = nt_getlr(net,from(k,1));
            nrind = lr_idckk(layer,from(k,2));
            if nrind <= 0
                outputs(k,1) = 1;
                loutmat(nrind+2,lrind) = 1;
            else
                outputs(k,1) = -2;
                flag = 1;
            end
        end
    end
end
else
outputs(k,1) = -1;
flag = 1;
end
for m = 1:numlayers
    k = lid(m);
    cl = nt_getlr(net,k);
    from = lr_getfn(cl,0);
    clin = lr_getin(cl);
    for j = 1:clin
        if from(j,1) == 0
            if from(j,2) > 0 & from(j,2) <= numin
                inputs(from(j,2),1) = 1;
                linmat(j+2,m) = 1;
            else
                linmat(j+2,m) = -3;
                flag = 1;
            end
        else
            index = contains(lid,from(j,1));
            if index == 0
                nrind = lr_idchk(nt_getlr(net,lid),from(j,1));
                if nrind == 0
                    linmat(j+2,m) = 1;
                    loutmat(from(j,2)+2,index(1,2)) = 1;
                else
                    linmat(j+2,m) = -2;
                    flag = 1;
                end
            else
                linmat(index(1,2)+2,m) = -1;
                flag = 1;
            end
        end
    end
end
if flag == 0
    for k = 1:numin
        if inputs(k,1) == 0
            flag = 1;
        end
    end
end
if flag == 0
    for i = 1:numlayers
        numnr = loutmat(2,i);
        for j = 1:numnr+2
            if loutmat(j,i) == 0
                flag = 1;
            end
        end
    end
end
if flag == 0
    net(4, 1) = 1;
end

function eb_cbt(udata)

  maxd = get_ctrl(udata(:, 3), 'value');
  xfn = get(udata(1, 1), 'string');
  yfn = get(udata(2, 1), 'string');
  zfn = get(udata(3, 1), 'string');
  crosssection = get(udata(1, 2), 'string');
  p = get_ctrl(udata(:, 4), 'value');
  n = get_ctrl(udata(:, 5), 'value');
  csx_max = get_ctrl(udata(:, 7), 'value');
  cs_factor = get_ctrl(udata(:, 8), 'value');
  alpha = get_ctrl(udata(:, 6), 'value');
  del = .0001;

  % clear graphics window
  clf;
  view([-45, 45]);

  grd = 100;

  % the cross-sections for each point on the data line have a collection of p points
  % the 'x' values of these points is determined randomly here within the cross-section:
  % domain (csd) and normalized between -csx_max/2 and csx_max/2
  %csd = (rand(n, p).* csx_max) - (ones(n, p).* (csx_max/2));

  % this calculates the delta matrix for the cross-sectional function and the parametric time
  delcs = ones(1, p).* del;  % cross-sectional
  delta = ones(1, n).* del;  % parametric time
  delta2 = ones(1, grd).* del;

  % this calculates n random times for the parametric equations and distributes them between
  % maxd and -maxd
  T = rand(1, n);
  %T = T - (ones(1, n)/2);
  %T = T*maxd;

  T2 = -maxd.*abs(maxd)/30:maxd;
  T3 = T;

  %csd2 = ones(7, size(T2, 2));
  %csd2(1,:) = csd2(1,:).*(-abs(csx_max));
  %csd2(2,:) = csd2(2,:).*((-abs(csx_max)/3)*2);
  %csd2(3,:) = csd2(3,:).*(-abs(csx_max)/3);
  %csd2(4,:) = csd2(4,:).*0;
  %csd2(7,:) = csd2(7,:).*abs(csx_max);
  %csd2(6,:) = csd2(6,:).*abs(csx_max)/3*2;
  %csd2(5,:) = csd2(5,:).*abs(csx_max)/3;

  % here the base line points in x,y, and z are computed using the parametric equations
  [xpts, ypts, zpts, T] = rand3d(xfn, yfn, zfn, -maxd/2, maxd/2, n, 1000);
  xpts = eval(xfn);
  ypts = eval(yfn);
  zpts = eval(zfn);
\%T = T2;
[xpts2,ypts2,zpts2,T2] = gridln3d(xfn,yfn,zfn,-maxd/2,maxd/2,grd,1000);
\%xpts2 = eval(xfn);
\%ypts2 = eval(yfn);
\%zpts2 = eval(zfn);

\%T = T3;
\% here the cross-section points are calculated and normalized: note that the normalized
\% points are then scaled by cs_factor to make them the right size relative to the base
\% curve
\%X = csd;
\%cspts = eval(crosssection);
\%X3 = X;
\%X2 = csd2;
\%X = X2;
\%cspts2 = eval(crosssection);
\%X = X3;
[X,cspts] = rand2d(crosssection,-csx_max/2,csx_max/2,n,p,1000);
[X2b,cspts2b] = gridln2d(crosssection,-csx_max/2,csx_max/2,25,1000);
X2 = zeros(25,n);
cspts2 = zeros(25,n);
for i = 1:n
    X2(:,i) = X2b';
    cspts2(:,i) = cspts2b';
end
csd = X;

csd2 = X2;

\% plot the base line
\%plot3(xpts, ypts, zpts,'r');
\%hold on;
\%pause

\% here the derivative vectors for the base line are calculated at every point: this is
\% done by calculating points on + and - delta from the original points and then calculating
\% the derivative vector: these derivative vectors are then normalized to have magnitude
\% equal to one.

\% these are the points at the plus delta side of the original points
T = T + delta;
xplus = eval(xfn);
yplus = eval(yfn);
zplus = eval(zfn);
tempt = T;

T2 = T2 + delta2;
T = T2;
xplus2 = eval(xfn);
yplus2 = eval(yfn);
zplus2 = eval(zfn);
tempt2 = T;

\% these are the points on the minus delta side of the original points
T = tempt;
T = T - 2*delta;
xminus = eval(xfn);
yminus = eval(yfn);
zminus = eval(zfn);

T2 = tempt2;
T = T2 - 2*delta2;
xminus2 = eval(xfn);
yminus2 = eval(yfn);
zminus2 = eval(zfn);
T2 = T;
T = tempt;

% reset the T matrix
T = T + delta;
T2 = T2 + delta2;

% calculate the dr matrices
dr_x = xplus - xminus;
dr_y = yplus - yminus;
dr_z = zplus - zminus;

dr_x2 = xplus2 - xminus2;
dr_y2 = yplus2 - yminus2;
dr_z2 = zplus2 - zminus2;

% calculate the magnitudes of the derivatives and normalize
drmag = (dr_x.^2 + dr_y.^2 + dr_z.^2).^0.5;
dr_x = dr_x / drmag;
dr_y = dr_y / drmag;
dr_z = dr_z / drmag;

drmag2 = (dr_x2.^2 + dr_y2.^2 + dr_z2.^2).^0.5;
dr_x2 = dr_x2 / drmag2;
dr_y2 = dr_y2 / drmag2;
dr_z2 = dr_z2 / drmag2;

% this section shifts the derivative vectors to be centered at the origin and then
% moves them out to their corresponding points on the base line for plotting
%dr_x2 = [1,0] * (dr_x - (dr_x.* .5));
%dr_x2 = (dr_x2 + [(0,1] * (dr_x.* (-.5))));
%dr_x2 = dr_x2 + ([1,1] * xpts);
%dr_y2 = [1,0] * (dr_y - (dr_y.* .5));
%dr_y2 = (dr_y2 + [(0,1] * (dr_y.* (-.5))));
%dr_y2 = dr_y2 + ([1,1] * ypts);
%dr_z2 = [1,0] * (dr_z - (dr_z.* .5));
%dr_z2 = (dr_z2 + [(0,1] * (dr_z.* (-.5))));
%dr_z2 = dr_z2 + ([1,1] * zpts);

%plot3(dr_x2, dr_y2, dr_z2, '.');
%hold off;
%pause;

% here the r-hat vector for each data point on the base line is calculated
rmag = sqrt(xpts.^2 + ypts.^2 + zpts.^2);
r_x = xpts / rmag;
r_y = ypts / rmag;
r_z = zpts / rmag;
rmag2 = sqrt(xpts2.^2 + ypts2.^2 + zpts2.^2);
r_x2 = xpts2 / rmag2;
r_y2 = ypts2 / rmag2;
r_z2 = zpts2 / rmag2;
\% r X dr is calculated

\texttt{rcrossdr}_x = (r_y \cdot \texttt{dr}_z) - (\texttt{dr}_y \cdot r_z);
\texttt{rcrossdr}_y = (r_z \cdot \texttt{dr}_x) - (\texttt{dr}_z \cdot r_x);
\texttt{rcrossdr}_z = (r_x \cdot \texttt{dr}_y) - (\texttt{dr}_x \cdot r_y);
\texttt{rcrossdr}_x2 = (r_y2 \cdot \texttt{dr}_z2) - (\texttt{dr}_y2 \cdot r_z2);
\texttt{rcrossdr}_y2 = (r_z2 \cdot \texttt{dr}_x2) - (\texttt{dr}_z2 \cdot r_x2);
\texttt{rcrossdr}_z2 = (r_x2 \cdot \texttt{dr}_y2) - (\texttt{dr}_x2 \cdot r_y2);

\texttt{rxdrmag} = \sqrt{\texttt{rcrossdr}_x^2 + \texttt{rcrossdr}_y^2 + \texttt{rcrossdr}_z^2};
\texttt{rcrossdr}_x = \texttt{rcrossdr}_x / \texttt{rxdrmag};
\texttt{rcrossdr}_y = \texttt{rcrossdr}_y / \texttt{rxdrmag};
\texttt{rcrossdr}_z = \texttt{rcrossdr}_z / \texttt{rxdrmag};
\texttt{rxdrmag2} = \sqrt{\texttt{rcrossdr}_x^2 + \texttt{rcrossdr}_y^2 + \texttt{rcrossdr}_z^2};
\texttt{rcrossdr}_x2 = \texttt{rcrossdr}_x2 / \texttt{rxdrmag2};
\texttt{rcrossdr}_y2 = \texttt{rcrossdr}_y2 / \texttt{rxdrmag2};
\texttt{rcrossdr}_z2 = \texttt{rcrossdr}_z2 / \texttt{rxdrmag2};

\% dr X (r X dr) is calculated

\texttt{drxdrdr}_x = (\texttt{dr}_y \cdot \texttt{rcrossdr}_z) - (\texttt{rcrossdr}_y \cdot \texttt{dr}_z);
\texttt{drxdrdr}_y = (\texttt{dr}_z \cdot \texttt{rcrossdr}_x) - (\texttt{rcrossdr}_z \cdot \texttt{dr}_x);
\texttt{drxdrdr}_z = (\texttt{dr}_x \cdot \texttt{rcrossdr}_y) - (\texttt{rcrossdr}_x \cdot \texttt{dr}_y);
\texttt{drxdrdr}_x2 = (\texttt{dr}_y2 \cdot \texttt{rcrossdr}_z2) - (\texttt{rcrossdr}_y2 \cdot \texttt{dr}_z2);
\texttt{drxdrdr}_y2 = (\texttt{dr}_z2 \cdot \texttt{rcrossdr}_x2) - (\texttt{rcrossdr}_z2 \cdot \texttt{dr}_x2);
\texttt{drxdrdr}_z2 = (\texttt{dr}_x2 \cdot \texttt{rcrossdr}_y2) - (\texttt{rcrossdr}_x2 \cdot \texttt{dr}_y2);

\texttt{drxdrdrmagn} = \sqrt{\texttt{drxdrdr}_x^2 + \texttt{drxdrdr}_y^2 + \texttt{drxdrdr}_z^2};
\texttt{drxdrdr}_x = \texttt{drxdrdr}_x / \texttt{drxdrdrmagn};
\texttt{drxdrdr}_y = \texttt{drxdrdr}_y / \texttt{drxdrdrmagn};
\texttt{drxdrdr}_z = \texttt{drxdrdr}_z / \texttt{drxdrdrmagn};
\texttt{drxdrdrmagn2} = \sqrt{\texttt{drxdrdr}_x^2 + \texttt{drxdrdr}_y^2 + \texttt{drxdrdr}_z^2};
\texttt{drxdrdr}_x2 = \texttt{drxdrdr}_x2 / \texttt{drxdrdrmagn2};
\texttt{drxdrdr}_y2 = \texttt{drxdrdr}_y2 / \texttt{drxdrdrmagn2};
\texttt{drxdrdr}_z2 = \texttt{drxdrdr}_z2 / \texttt{drxdrdrmagn2};

\% here the derivative at every point in the cross-section is calculated using the same
\% algorithm as above

\texttt{csd} = \texttt{csd} + \texttt{deltacs};
\texttt{csxplus} = \texttt{csd};
\texttt{X} = \texttt{csd};
\texttt{csyplus} = \texttt{eval(crosssection)};

\texttt{csd} = \texttt{csd} - 2*\texttt{deltacs};
\texttt{csxminus} = \texttt{csd};
\texttt{X} = \texttt{csd};
\texttt{csyminus} = \texttt{eval(crosssection)};

\texttt{csd} = \texttt{csd} + \texttt{deltacs};

\texttt{csx} = \texttt{csxplus} - \texttt{csxminus};
\texttt{csy} = \texttt{csyplus} - \texttt{csyminus};

\% here the normal to the cross-section curve is calculated by determining the angle of
\% the derivative and then rotating it by plus or minus pi/2. The direction of rotation
\% is determined by the sign of a collection of normally distributed numbers. once the
\% normal has been determined, the data points along the cross-section are moved out
\% a distance beta normal to the cross-section curve. beta is determined by scaling
\% the spread factor alpha by a random number between 0 and 1.
\texttt{csx} = \texttt{ones(p,n)}; \quad \% cross-section x-points
csy = ones(p,n); % cross-section y-points

csx2 = csd2;
csy2 = cspts2;
alp = alpha * csx_max;

% unfortunately I had to use a loop here--slow, but I could find no other way.
for k = 1:n
    csx(:,k) = (csd(k,:) + (cos(theta(k,:))) * beta(k,:));
    csy(:,k) = (cspts(k,:) + (sin(theta(k,:))) * beta(k,:));
end

cxptsmax = max(max(abs(cxpts)));
cxmax = max(max(abs(csd)));
cxptsmax = max([cxmax; cxptsmax]);
cxpts = (cxpts/cxptsmax).* cs_factor;
csd = (csd/cxptsmax).* cs_factor;
csx = (csx/cxptsmax).* cs_factor;
csy = (csy/cxptsmax).* cs_factor;
cxpts2 = (cxpts2/cxptsmax).* cs_factor;
csd2 = (csd2/cxptsmax).* cs_factor;
csx2 = (csx2/cxptsmax).* cs_factor;
csy2 = (csy2/cxptsmax).* cs_factor;

% plot the last cross-section--this is useful to see the affect of alpha when using
% a decent sized p
%plot(csy(:,k), csy(:,k), '.'),
%hold on;
%plot(csd(k,:), cspts(k,:), 'o');
%pause;
%hold off;
%cyl;

shiftvect = ones(size(csx,1),1); % this is the vector used to shift the cs-points
% out to the base line

final_x = ones(size(csx,1),n);
for k = 1:n
    final_x(:,k) = csx(:,k) * rcrossdr_x(k) + (csy(:,k) * drxdrxdr_x(k)) + (shiftvect * xpts(k));
end
final_y = ones(size(csx,1),n);
for k = 1:n
    final_y(:,k) = csx(:,k) * rcrossdr_y(k) + (csy(:,k) * drxrdxdr_y(k)) + (shiftvec* ypts(k));
end

final_z = ones(size(csx,1),n);
for k = 1:n
    final_z(:,k) = csx(:,k) * rcrossdr_z(k) + (csy(:,k) * drxrdxdr_z(k)) + (shiftvec* zpts(k));
end

% this is the vector used to shift the cs-points % out to the base line

shiftvec2 = ones(size(csx2,1),1);

final_x2 = ones(25,size(T2,2));
%disp(size(csx2));
%disp(size(final_x2));
%disp(size(drxxrdx_y));
%disp(size(csy2));
%disp(size(xpts));
%disp(size(rcrossdr_x));
%pause;
for k = 1:size(T2,2)
    final_x2(:,k) = csx2(:,k) * rcrossdr_x2(k) + (csy2(:,k) * drxrdxrdx2(k)) + (shiftvec2* xpts2(k));
end

final_y2 = ones(size(csx2,1),size(T2,2));
for k = 1:size(T2,2)
    final_y2(:,k) = csx2(:,k) * rcrossdr_y2(k) + (csy2(:,k) * drxrdxrdx2(k)) + (shiftvec2* ypts2(k));
end

final_z2 = ones(size(csx2,1),size(T2,2));
for k = 1:size(T2,2)
    final_z2(:,k) = csx2(:,k) * rcrossdr_z2(k) + (csy2(:,k) * drxrdxrdx2(k)) + (shiftvec2* zpts2(k));
end

% here the output points are normalized to have magnitude <=1
% Carlos commented this out on 6/30/94. It is hard to keep track of
% these random factors.
%
% Note from Ambrose: This is far from random. This is just a final normalization.
final_xmax = max(max(abs(final_x)));
final_ymax = max(max(abs(final_y)));
final_zmax = max(max(abs(final_z)));
final_xyzmax = max([final_xmax; final_ymax; final_zmax]);
final_x = final_x/final_xyzmax;
final_y = final_y/final_xyzmax;
final_z = final_z/final_xyzmax;
xpts = xpts/final_xyzmax;
ypts = ypts/final_xyzmax;
zpts = zpts/final_xyzmax;
final_x2 = final_x2/final_xyzmax;
final_y2 = final_y2/final_xyzmax;
final_z2 = final_z2/final_xyzmax;

final_x2 = final_x2 - mean(final_x);
xpts = xpts - mean(final_x);
final_x = final_x - mean(final_x);
final_y2 = final_y2 - mean(final_y);
ypts = ypts - mean(final_y);
final_y = final_y - mean(final_y);
final_z2 = final_z2 - mean(final_z);
zpts = zpts - mean(final_z);
final_z = final_z - mean(final_z);

% plot the results
clf;
view(-45, 75);
plot3(final_x, final_y, final_z, 'r--');
hold on;
grid on;
plot3([-1 1], [0 0], [0 0], 'y-');
plot3([0 0], [-1 1], [0 0], 'y-');
plot3([0 0], [0 0], [-1 1], 'y-');
plot3(xpts, ypts, zpts, 'r--');
pause;
plot3(final_x2, final_y2, final_z2, 'g');
plot3(final_x2', final_y2', final_z2', 'b');
sse = sum(sqr(xpts-final_x)+sqr(ypts-final_y)+sqr(zpts-final_z),
p = zeros(3,n),
p(1,:) = final_x;
p(2,:) = final_y;
p(3,:) = final_z;

[fname fpath] = uiputfile('.mat', 'Save As');

fname = [fpath fname],
eval(['save ' fname ' final_x final_y final_z final_x2 final_y2 final_z2 xpts ypts zpts sse p xfn yfn zfn crossection alpha cs_factor csx_max maxd']);
vmenu(gcf, gca);

function [xpts,ypts] = gridln2d(function,min_x,max_x,numlines,numdiv)
%function [xpts,ypts] = gridln2d(function,min_x,max_x,numlines,numdiv)

%clf;
X = (0:numdiv).*((max_x - min_x)/numdiv)+(ones(1,numdiv+1)*min_x);
xdiffsq = ((max_x - min_x)/numdiv)*2;
Y = eval(function);
dists = diff(Y).^2;
dists = dists + ones(1,numdiv)*xdiffsq;
dists = sqrt(dists);
inc_dist = zeros(1,numdiv+1);
for i = 2:numdiv+1
    inc_dist(i) = inc_dist(i-1) + dists(i-1);
end
%subplot(311);
%plot(X,inc_dist);
%subplot(312);
tot_dist = dists*ones(numdiv,1);
%plot(X(1:1:numdiv),dists);
k = (0:numlines-1)/(numlines-1);
k = k*tot_dist;
k(1) = min_x;
k(numlines) = max_x;
for i = 2:numlines-1
    kdiff = abs(inc_dist-(ones(1,numdiv+1)*k(i)));
    kmatch = min(kdiff),
    k(i) = ((find(kdiff==kmatch)
umdiv)*(max_x - min_x)+min_x;
end
%subplot(313);
X2 = X;
X = k;
%disp(k);
%pause;
kpts = eval(function);
%plot(X,kpts,rx');
%hold on;
%plot(X2,Y);
xpts = X;
ypts = kpts;
function [xpts,ypts,zpts,T] = gridln3d(xfunction,yfunction,zfunction,min_t,max_t,numlines,numdiv)
%function [xpts,ypts,zpts,T] = gridln3d(xfunction,yfunction,zfunction,min_t,max_t,numlines,numdiv)

%clf;
T = ((1:numdiv).*(max_t - min_t)/numdiv)+(ones(1,numdiv+1)*min_t);
X = eval(xfunction);
Y = eval(yfunction);
Z = eval(zfunction);
dists = sqrt((diff(X).^2) + (diff(Y).^2) + (diff(Z).^2));
inc_dist = zeros(1,numdiv+1);
for i = 2:numdiv+1
    inc_dist(i) = inc_dist(i-1) + dists(i-1);
end
%subplot(311);
%plot(T,inc_dist);
%subplot(312);
tot_dist = dists*ones(numdiv,1);
%plot(T(1,1:numdiv),dists);
k = (0:numlines-1)./(numlines-1);
k = k*tot_dist;
k(1) = min_t;
k(numlines) = max_t;
for i = 2:numlines-1
    kdiff = abs(inc_dist-(ones(1,numdiv+1)*k(i)));
    kmatch = min(kdiff),
    k(i) = ((find(kdiff==kmatch)
umdiv)*(max_t - min_t)+min_t;
end
%subplot(313);
T2 = X;
T = k;
%disp(k);
%pause;
xpts = eval(xfunction);
ypts = eval(yfunction);
zpts = eval(zfunction);
%plot3(xpts,ypts,zpts,rx');
%hold on;
%plot3(X,Y,Z);
%grid on;
function [w1,b1,w2,b2,w3,b3,w4,b4,w5,b5,w6,b6,w7,b7,w8,b8,w9,b9,w10,b10,w11,w12,...
% TRAINING PARAMETERS
tp = mdef([25 1000 0.02 0.0001 0.001 10 0.1 1e10]),
df = tp(1);
me = tp(2);
eg = tp(3);
grad_min = tp(4);
mu_init = tp(5);
mu_inc = tp(6);
mu_dec = tp(7);
mu_max = tp(8);
df1 = feval(1,'delta');
df2 = feval(2,'delta');
df3 = feval(3,'delta');
df4 = feval(4,'delta');

% DEFINE SIZES
[s1,t] = size(w1);
[s2,s1] = size(w2);
[s3,s2] = size(w3);
[s4,s3] = size(w4);
w1_ind = 1:(s1*rt];
b1_ind = 1:s1 + w1_ind(length(w1_ind));
w2_ind = 1:(s1*s2) + b1_ind(length(b1_ind));
b2_ind = 1:s2 + w2_ind(length(w2_ind));
w3_ind = 1:s2*s3 + b2_ind(length(b2_ind));
b3_ind = 1:s3 + w3_ind(length(w3_ind));
w4_ind = 1:s3*s4 + b3_ind(length(b3_ind));
b4_ind = 1:s4 + w4_ind(length(w4_ind));
i = eye(b4_ind(length(b4_ind)));
dw1 = w1; db1 = b1;
dw2 = w2; db2 = b2;
dw3 = w3; db3 = b3;
dw4 = w4; db4 = b4;
el_p = size(p,1)*size(p,2);
el_t = size(t,1)*size(t,2);
sp = size(p,2);
half = fix(sp/2);
p1 = p(:,1:half);
p2 = p(:,half+1:sp);
ext_p1 = nncyp(p(:,1:half),s4);
ext_p2 = nncyp(p(:,half+1:sp),s4);

% PRESENTATION PHASE
[a1,a2,a3,a4] = simuf4(p,w1,b1,f1,w2,b2,f2,w3,b3,f3,w4,b4,f4);
e = pt-a4;
SSE = sumsqr(e);
MSE = SSE/el_p;
a1t,a2t,a3t,a4t = simuf4(t,w1,b1,f1,w2,b2,f2,w3,b3,f3,w4,b4,f4);
e_t = tt-a4t;
SSE_t = sumsqr(e_t);
MSE_t = SSE_t/el_t;

% TRAINING RECORD
tr = zeros(1,me+1);
tr(1) = MSE;
trt = zeros(1,me+1);
trt(1) = MSE_t;

% PLOTTING FLAG

96
plottype = (r==1) & (s2==1);

% PLOTTING
clg
subplot(321);
plot(p(1,:),pt(2,:),g');
hold on;
grid on;
plot(a4(1,:),a4(2,:),r');
hold off;
title('1-2');
subplot(322);
plot(p(1,:),pt(3,:),g');
hold on;
grid on;
plot(a4(1,:),a4(3,:),r');
hold off;
title('1-3');
subplot(323);
plot(p(2,:),pt(3,:),g');
hold on;
grid on;
plot(a4(2,:),a4(3,:),r');
hold off;
title('2-3');
subplot(324);
if size(a4,1) > 3
plot(p(1,:),pt(4,:),g');
hold on;
grid on;
plot(a4(1,:),a4(4,:),r');
title('1-4');
hold off;
end
subplot(325);
%title('2-4');
%plot(pt(2,:),pt(4,:),g');
%hold on;
%grid on;
%plot(a4(2,:),a4(4,:),r');
%hold off;
hist(a2,100);
title('Coefficient Histogram');
subplot(326);
message = sprintf('TRAINLIN: %%%g/%%g epochs, mu = %%%g, MSE = %%%g \n',me);
fprintf(message,0,mu_init,MSE)
if plottype
    h = plotfa(p,t,p,a3);
else
    [h,h2] = ploten2(tr(1),trt(1),eg);
end

mu = mu_init;
for i=1:me

% CHECK PHASE
if SSE < eg, i=i-1; break. end

% FIND JACOBIAN #1

97
ext_a1 = nncpyi(a1(:,1:half),s4);
ext_a2 = nncpyi(a2(:,1:half),s4);
ext_a3 = nncpyi(a3(:,1:half),s4);
d4 = feval(d4,a4(:,1:half));
ext_d4 = -nncpyd(d4);
ext_d3 = feval(d3,ext_a3,ext_d4,w4);
ext_d2 = feval(d2,ext_a2,ext_d3,w3);
ext_d1 = feval(d1,ext_a1,ext_d2,w2);
j1 = learnlm(ext_p1,ext_d1);
j2 = learnlm(ext_a1,ext_d2);
j3 = learnlm(ext_a2,ext_d3);
j4 = learnlm(ext_a3,ext_d4);
j1 = [j1, ext_d1', j2, ext_d2', j3, ext_d3', j4, ext_d4'];

% FIND JACOBIAN #2
ext_a1 = nncpyi(a1(:,half+1:sp),s4);
ext_a2 = nncpyi(a2(:,half+1:sp),s4);
ext_a3 = nncpyi(a3(:,half+1:sp),s4);
d4 = feval(d4,a4(:,half+1:sp));
ext_d4 = -nncpyd(d4);
ext_d3 = feval(d3,ext_a3,ext_d4,w4);
ext_d2 = feval(d2,ext_a2,ext_d3,w3);
ext_d1 = feval(d1,ext_a1,ext_d2,w2);
j1 = learnlm(ext_p2,ext_d1);
j2 = learnlm(ext_a1,ext_d2);
j3 = learnlm(ext_a2,ext_d3);
j4 = learnlm(ext_a3,ext_d4);
j2 = [j1, ext_d1', j2, ext_d2', j3, ext_d3', j4, ext_d4'];

% CHECK MAGNITUDE OF GRADIENT
e1 = e(:,1:half);
ej = j1*e1(:,);
e2 = e(:,half+1:sp);
je = je + (j2*e2(:,));
grad = norm(je);
if grad < grad_min, i = i-1; break, end

% INNER LOOP, INCREASE MU UNTIL THE ERRORS ARE REDUCED
jj = (t1^t1) + (t2^t2);

while (mu <= mu_max)
dx = -(jj + ii*mu) \ je;
dw1() = dwx(w1_in)
db1 = dbx(b1_in)
dw2() = dwx(w2_in)
db2 = dbx(b2_in)
dw3() = dwx(w3_in)
db3 = dbx(b3_in)
dw4() = dwx(w4_in)
db4 = dbx(b4_in)
new_w1 = w1 + dw1
new_b1 = b1 + db1
new_w2 = w2 + dw2
new_b2 = b2 + db2
new_w3 = w3 + dw3
new_b3 = b3 + db3
new_w4 = w4 + dw4
new_b4 = b4 + db4

% EVALUATE NEW NETWORK
[a1,a2,a3,a4] = simuflm(p,new_w1,new_b1,f1,new_w2,new_b2,f2,new_w3,new_b3,f3,new_w4,new_b4,f4);
new_e = pt-a4;
new_SSE = sumsq(new_e);
if (new_SSE < SSE), break, end
mu = mu * mu_inc;
end
if (mu > mu_max), i = i-1; break, end

98
\[ \mu = \mu \ast \mu_{\text{dec}}; \]

\% UPDATE NETWORK
\[ w_1 = \text{new}_w1; b_1 = \text{new}_b1; \]
\[ w_2 = \text{new}_w2; b_2 = \text{new}_b2; \]
\[ w_3 = \text{new}_w3; b_3 = \text{new}_b3; \]
\[ w_4 = \text{new}_w4; b_4 = \text{new}_b4; \]
\[ e = \text{new}_e; \text{SSE} = \text{new}_{\text{SSE}}; \text{MSE} = \text{SSE/1}_p; \]

\% TRAINING RECORD
\[ \text{tr}(i+1) = \text{SSE/1}_p; \]
\[ [a_1, a_2, a_3, a_4] = \text{simuff4}(1, w_1, b_1, f_1, w_2, b_2, f_2, w_3, b_3, f_3, w_4, b_4, f_4); \]
\[ e_t = ft-a_4; \]
\[ \text{SSE}_t = \text{sumsq}(e_t); \]
\[ \text{MSE}_t = \text{SSE}_t/\text{tr}(i+1); \]
\[ \text{tr}(i+1) = \text{MSE}_t; \]

\% PLOTTING
\[ \text{if rem}(i, df) == 0 \]
\[ clf \]
\[ \text{subplot}(321); \]
\[ \text{plot}(pt(1,:), pt(2,:), g.'); hold on; \]
\[ \text{grid on}; \]
\[ \text{plot}(a4(1,:), a4(2,:), y'); hold off; \]
\[ \text{title}('1-2'); \]
\[ \text{subplot}(322); \]
\[ \text{plot}(pt(1,:), pt(3,:), g.'); hold on; \]
\[ \text{grid on}; \]
\[ \text{plot}(a4(1,:), a4(3,:), y'); hold off; \]
\[ \text{title}('1-3'); \]
\[ \text{subplot}(323); \]
\[ \text{plot}(pt(2,:), pt(3,:), g.'); hold on; \]
\[ \text{grid on}; \]
\[ \text{plot}(a4(2,:), a4(3,:), y'); hold off; \]
\[ \text{title}('2-3'); \]
\[ \text{subplot}(324); \]
\[ \text{if size(a4, 1) > 3} \]
\[ \text{plot}(pt(1,:), pt(4,:), g.'); hold on; \]
\[ \text{grid on}; \]
\[ \text{plot}(a4(1,:), a4(4,:), y'); \]
\[ \text{title}('1-4'); \]
\[ \text{hold off}; \]
\[ \text{end} \]
\[ \text{subplot}(325); \]
\[ \text{title}('2-4'); \]
\[ \text{plot}(pt(2,:), pt(4,:), g.'); \]
\[ \text{hold on}; \]
\[ \text{grid on}; \]
\[ \text{plot}(a4(2,:), a4(4,:), y'); \]
\[ \text{hold off}; \]
\[ \text{hist}(a2, 100); \]
title('Coefficient Histogram'),
subplot(326);
fprintf(message,i,mu,MSE)
if plottype
delete(h); h = plot(p,a2,'m'); drawnow;
else
[h,h2] = ploterr2(tr(1:(i+1)),trt(1:(i+1)),eg,h,h2);
end
end

% TRAINING RECORD
tr = tr(1:(i+1));
trt = trt(1:(i+1));

% PLOTTING
if rem(i,df) == 0
subplot(321);
title('1-2');
plot(pt(1,:),pt(2,:), 'g');
hold on;
grid on;
plot(a4(1,:),a4(2,:), 'r');
hold off;
subplot(322);
title('1-3');
plot(pt(1,:),pt(3,:), 'g');
hold on;
grid on;
plot(a4(1,:),a4(3,:), 'r');
hold off;
subplot(323);
title('2-3');
plot(pt(2,:),pt(3,:), 'g');
hold on;
grid on;
plot(a4(2,:),a4(3,:), 'r');
hold off;
subplot(324);
%title('1-4');
%plot(pt(1,:),pt(4,:), 'g');
%hold on;
%grid on;
%plot(a4(1,:),a4(4,:), 'r');
%hold off;
subplot(325);
%title('2-4');
%plot(pt(2,:),pt(4,:), 'g');
%hold on;
%grid on;
%plot(a4(2,:),a4(4,:), 'r');
%hold off;
hist(a2,100);
subplot(326);
hold on;
grid on;
plot3(a4(1,:),a4(2,:),a4(3,:), 'r');
hold off;
subplot(122);
fprintf(message,i,mu,MSE)
if plotype
    delete(h);
    plot(p,a2,m');
    drawnow;
else
    [h,h2] = ploterr2(tr(1:(i+1)),trt(1:(i+1)),eg,h,h2);
end
end

% WARNINGS
if SSE > eg
    disp('')
    if (mu > mu_max)
        disp('TRAINLM: Error gradient is too small to continue learning.')
    else
        disp('TRAINLM: Network error did not reach the error goal.')
    end
    disp(' Further training may be necessary, or try different')
    disp(' initial weights and biases and/or more hidden neurons.')
    disp('')
end

%%%%%%
function [w1,b1,w2,b2,w3,b3,w4,b4,tr,trt] = inc_tr8d(w1,b1,f1,w2,b2,f2,w3,b3,f3,w4,f4,p,t,c,p,c,t,r,tart,tp)
%[w1,b1,w2,b2,w3,b3,w4,b4,tr,trt] = inc_tr8d(w1,b1,f1,w2,b2,f2,w3,b3,f3,w4,f4,p,t,c,p,c,t,r,tart,tp)

%TLM3 Train 3-layer feed-forward network w/Levenberg-Marquardt.
%
%    [W1,B1,W2,B2,W3,B3,TE/TR] = TLM3(W1,B1,F1,W2,B2,F2,W3,B3,F3,P,T)
%    Wi - Weight matrix of ith layer.
%    Bi - Bias vector of ith layer.
%    F - Transfer function (string) of ith layer.
%    P - RxQ matrix of input vectors.
%    T - S2xQ matrix of target vectors.
%    TP - Training parameters (optional).
%
% Returns:
%    Wi - new weights.
%    Bi - new biases.
%    TE - the actual number of epochs trained.
%    TR - training record: [row of errors]
%
% Training parameters are:
%    TP(1) - Epochs between updating display, default = 25.
%    TP(2) - Maximum number of epochs to train, default = 1000.
%    TP(3) - Sum-squared error goal, default = 0.02.
%    TP(4) - Minimum gradient, default = 0.0001.
%    TP(5) - Initial value for MU, default = 0.001.
%    TP(6) - Multiplier for increasing MU, default = 10.
%    TP(7) - Multiplier for decreasing MU, default = 0.1.
%    TP(8) - Maximum value for MU, default = 1e10.
%    Missing parameters and NaNs are replaced with defaults.
%
% TRAINING PARAMETERS
tp = ndde(tp,[25 1000 0.02 0.0001 0.001 10 0.1 1e10]);
df = tp(1);
me = tp(2);
eg = tp(3);
grad_min = tp(4);
mu_init = tp(5);
mu_inc = tp(6);
mu_dec = tp(7);
mu_max = tp(8);
df1 = feval(f1,'delta');
df2 = feval(f2,'delta');
df3 = feval(f3,'delta');
df4 = feval(f4,'delta');

% DEFINE SIZES
[s1,r1] = size(w1);
[s2,s1] = size(w2);
[s3,special] = size(w3);
[s4,s3] = size(w4);
w1_ind = [1:(s1*r1)];
b1_ind = [1:s1] + w1_ind(length(w1_ind));
w2_ind = [1:(s1*s2)] + b1_ind(length(b1_ind));
b2_ind = [1:s2] + w2_ind(length(w2_ind));
w3_ind = [1:(special*s3)] + b2_ind(length(b2_ind));
b3_ind = [1:s3] + w3_ind(length(w3_ind));
w4_ind = [1:(s3*s4)] + b3_ind(length(b3_ind));
b4_ind = [1:s4] + w4_ind(length(w4_ind));
ii = eye(b4_ind(length(b4_ind)));
dw1 = w1; db1 = b1;
dw2 = w2; db2 = b2;
dw3 = w3; db3 = b3;
dw4 = w4; db4 = b4;
c1_p = size(p,1)*size(p,2);
c1_t = size(t,1)*size(t,2);
sp = size(p,2);
half = fix(sp/2);
p1 = p(:,1:half);
p2 = p(:,half+1:sp);
ext_p1 = numpyi(p(:,1:half),s4);
ext_p2 = numpyi(p(:,half+1:sp),s4);
% PRESENTATION PHASE
[a1,a2,a3,a4] = simuff4x(y,cp,w1,b1,f1,w2,b2,f2,w3,b3,f3,w4,b4,f4);
e = tar-a4;
SSE = sumsqr(e);
MSE = SSE/c1_p;
[a1t,a2t,a3t,a4t] = simuff4x(t,ct,w1,b1,f1,w2,b2,f2,w3,b3,f3,w4,b4,f4);
et = tart-a4t;
SSE_t = sumsqr(e_t);
MSE_t = SSE/c1_t;
% TRAINING RECORD
tr = zeros(1,me+1);
tr(1) = MSE;
trt = zeros(1,me+1);
trt(1) = MSE_t;

% PLOTTING FLAG
plottype = (r==1) & (s2==1);

% PLOTTING
clf
subplot(321);
title('1-2');
plot(tar(1,:),tar(2,:),g.'),
hold on;
ggrid on;
plot(a4(1,:),a4(2,:),'.');
hold off;
subplot(322);
title('1-3');
plot(tar(1,:),tar(3,:),g.'),
hold on;
grid on;
plot(a4(1,:),a4(3,:),'.');
hold off;
subplot(323);
title('2-3');
plot(tar(2,:),tar(3,:),g.'),
hold on;
grid on;
plot(a4(2,:),a4(3,:),'.');
hold off;
subplot(324);
%title('1-4');
%plot(tar(1,:),tar(4,:),g.'),
%hold on;
%grid on;
%plot(a4(1,:),a4(4,:),'.');
%hold off;
subplot(325);
%title('2-4');
%plot(tar(2,:),tar(4,:),g.'),
%hold on;
%grid on;
%plot(a4(2,:),a4(4,:),'.');
%hold off;
s = size(cp,1)+1;

hist(a2(sc,:),100);
subplot(326);
message = sprintf('TRAINLM: %g epochs, mu = %g, MSE = %g',

fprintf(message,0,mu_init,MSE)
if plottype
    h = plotfa(p,t,p,a3);
else
    [h,h2] = ploterr2(tr(1),trt(1),eg);
end

mu = mu_init;
for i=1:me

% CHECK PHASE
if SSE < eg, i=i-1; break, end

% FIND JACOBIAN #1
    ext_a1 = nncpyi(a1(:,1:half),s4);
    ext_a2 = nncpyi(a2(:,1:half),s4);
    ext_a3 = nncpyi(a3(:,1:half),s4);
    d4 = feval(df4,a4(:,1:half));
    ext_d4 = -nncpyd(d4);
    ext_d3 = feval(df3,ext_a1,ext_d4,w4);
    d3_r = w3*ext_d3;
    ext_d2 = feval(df2,ext_a2,d3_r(sc,:));
    ext_d1 = feval(df1,(1)ext_a1,ext_d2,w2);
    j1 = learnlm(ext_p1,ext_d1);
j2 = learnlm(ext_a1,ext_d2);
  j3 = learnlm(ext_a2,ext_d3);
  j4 = learnlm(ext_a3,ext_d4);
  j1 = [j1, ext_d1'; j2, ext_d2'; j3, ext_d3'; j4, ext_d4'];

% FIND JACOBIAN #2
  ext_a1 = nncpyi(a1(:,half+1:sp),s4);
  ext_a2 = nncpyi(a2(:,half+1:sp),s4);
  ext_a3 = nncpyi(a3(:,half+1:sp),s4);
  d4 = feval(df4_a4(:,half+1:sp));
  ext_d4 = -nncpyd(d4);
  ext_d3 = feval(df3,ext_a3,ext_d4,w4);
  d3_r = w3*ext_d3;
  ext_d2 = feval(df2,ext_a2,d3_r(sc,:));
  ext_a1 = feval(df1,ext_a1,ext_d2,w2);
  j1 = learnlm(ext_p2,ext_d1);
  j2 = learnlm(ext_a1,ext_d2);
  j3 = learnlm(ext_a2,ext_d3);
  j4 = learnlm(ext_a3,ext_d4);
  j12 = [j1, ext_d1', j2, ext_d2', j3, ext_d3', j4, ext_d4'];

% CHECK MAGNITUDE OF GRADIENT
  e1 = e(:,1:half);
  je = j12*e1(:,);
  e2 = e(:,half+1:sp);
  je = je + (j12*e2(:,));
  grad = norm(je);
  if grad < grad_min, i=i-1; break, end

% INNER LOOP, INCREASE MU UNTIL THE ERRORS ARE REDUCED
  jj = (j11*j1) + (j12*j12);

  while (mu <= mu_max)
    dx = -(jj+j1*mu) \ je;
    dw1(:,)= dx(w1_ind); db1 = dx(b1_ind);
    dw2(:,)= dx(w2_ind); db2 = dx(b2_ind);
    dw3(:,)= dx(w3_ind); db3 = dx(b3_ind);
    dw4(:,)= dx(w4_ind); db4 = dx(b4_ind);
    new_w1 = w1 + dw1; new_b1 = b1 + db1;
    new_w2 = w2 + dw2; new_b2 = b2 + db2;
    new_w3 = w3 + dw3; new_b3 = b3 + db3;
    new_w4 = w4 + dw4; new_b4 = b4 + db4;

% EVALUATE NEW NETWORK
  [a1,a2,a3,a4] = simuf4x(p,cp,new_w1,new_b1,f1,new_w2,new_b2,f2,new_w3,new_b3,f3,new_w4,new_b4,f4);
  new_e = tar-a4;
  new_SSE = sumsqr(new_e);

  if (new_SSE < SSE), break, end
  mu = mu * mu_inc;
  else
    i = i-1; break, end
  end
  mu = mu * mu_dec;

% UPDATE NETWORK
  w1 = new_w1; b1 = new_b1;
  w2 = new_w2; b2 = new_b2;
  w3 = new_w3; b3 = new_b3;
  w4 = new_w4; b4 = new_b4;
  e = new_e; SSE = new_SSE; MSE = SSE/el_p;
% TRAINING RECORD
tr(i+1) = SSE/el_p;
[a1t,a2t,a3t,a4t] = simuff4x(t,ct,w1,b1,f1,w2,b2,f2,w3,b3,f3,w4,b4,f4);
e_t = tar-a4t;
SSE_t = sumsqr(e_t);
MSE_t = SSE_t/vel_t;
tr(i+1) = MSE_t;

% PLOTTING
if rem(i,df) == 0
subplot(321);
title('1-2');
plot(tar(1,:),tar(2,:),g');
hold on;
grid on;
plot(a4(1,:),a4(2,:),r');
hold off;
subplot(322);
title('1-3');
plot(tar(1,:),tar(3,:),g');
hold on;
grid on;
plot(a4(1,:),a4(3,:),r');
hold off;
subplot(323);
title('2-3');
plot(tar(2,:),tar(3,:),g');
hold on;
grid on;
plot(a4(2,:),a4(3,:),r');
hold off;
subplot(324);
%title('1-4');
%plot(tar(1,:),tar(4,:),g');
%hold on;
%grid on;
%plot(a4(1,:),a4(4,:),r');
%hold off;
subplot(325);
%title('2-4');
%plot(tar(2,:),tar(4,:),g');
%hold on;
%grid on;
%plot(a4(2,:),a4(4,:),r');
%hold off;
hist(a2(sc,:),100);
subplot(326);
fprintf(message,i,mu,MSE)
if plotype
    delete(h); h = plot(p,a2,'m'); drawnow;
else
    [h,h2] = ploterr2(tr(1:(i+1)),tr(1:(i+1)),eg,h,h2);
end
end
end

% TRAINING RECORD
tr = tr(1:(i+1));
trt = trt(1:(i+1));

% PLOTTING
i: r = m(i,df) = 0
subplot(321);
title('1-2');
plot(tar(1,:),tar(2,:),g,'');
hold on;
grid on;
plot(a4(1,:),a4(2,:),r,'');
hold off;
subplot(322);
title('1-3');
plot(tar(1,:),tar(3,:),g,'');
hold on;
grid on;
plot(a4(1,:),a4(3,:),r,'');
hold off;
subplot(323);
title('2-3');
plot(tar(2,:),tar(3,:),g,'');
hold on;
grid on;
plot(a4(2,:),a4(3,:),r,'');
hold off;
subplot(324);
%title('1-4');
%plot(tar(1,:),tar(4,:),g,'');
%hold on;
%grid on;
%plot(a4(1,:),a4(4,:),r,'');
%hold off;
subplot(325);
%title('2-4');
%plot(tar(2,:),tar(4,:),g,'');
%hold on;
%grid on;
%plot(a4(2,:),a4(4,:),r,'');
%hold off;
hist(a2(sc,:),100);
subplot(326);
hold on;
grid on;
plot3(a4(1,:),a4(2,:),a4(3,:),r,'');
hold off;
subplot(122);
fprintf(message,i,mu,MSE)
if plottype
delete(h);
plot(p,a2,'m');
drawnnow;
else
[h,h2] = ploterr2(tr(1:(i+1)),trt(1:(i+1)),eg,h,h2);
end
end

% WARNINGS
if SSE > eg
disp('')
if (mu > mu_max)
disp('TRAINLM: Error gradient is too small to continue learning.')
else
disp('TRAINLM: Network error did not reach the error goal.')
end
disp(' Further training may be necessary, or try different')
disp(' initial weights and biases and/or more hidden neurons.')
disp('')
end

function output = ntsimul(net,p)
% function output = nts_simul(net,p)

if nt_getsd(net) & nt_getck(net)
    numin = nt_getin(net);
    numlayers = nt_getnl(net);
    numouts = nt_getno(net);
    lr_info = nt_getli(net);
    [r,q] = size(p);
    if r == numin
        error('Input vector is of wrong size');
    else
        sum = numouts;
        for k = 1:numlayers
            sum = sum + lr_info(4,k);
        end
        Inth = zeros(sum,2);
        output = zeros(sum,q);
        cnumouts = numouts;
        for k = 1:numlayers
            cl = nt_getlr(net.lr_info(1,k));
            cwt = lr_getw(cl,0,0);
            cwt = cwt';
            cnumin = lr_info(3,k);
            cinp = zeros(cnumin,q);
            cfrom = lr_getfn(cl,0);
            cout = lr_getnl(cl);
            Intbl((cnumouts+1:cnumouts+cout),1) = ones(cout,1)*lr_info(1,k);
            Intbl((cnumouts+1:cnumouts+cout),2) = lr_getnd(cl);
            ctf = lr_gettl(cl);
            cbi = lr_getbl(cl,0);
            for m = 1:cnumin
                %grab all inputs--put in cinp
                if cfrom(m,1) == 0
                    cinp(m,:) = p(cfrom(m,2,:));
                else
                    orow = find(Intbl(:,1) == cfrom(m,1));
                    orowa = orow(1);
                    orow = find(Intbl(orowa,2) == cfrom(m,2));
                    orow = orow+orowa-1;
                    cinp(m,:) = output(orow,:);
                end
            end
            output((cnumouts+1:cnumouts+cout,:),:) = feval(ctl,cwt*cinp,cbi);
            cnumouts = cnumouts + cout;
        end
for m = 1:numouts
    cfrom = nt_getfn(net,0);
    if cfrom(m,1) == 0
        cfrom(m,1) = 0
    end
end

107
cinp(m,:) = p(cfom(m,2,:));
else
  orow = find(ival(:,1) == cfom(m,1));
  orowa = orow(1);
  orow = find(ival(orow,2) == cfom(m,2));
  orow = orow+orowa-1;
  output(m,:) = output(orow,:);
end
else
  error('Net must be sorted and checked first');
end

output2 = zeros(size(output,1),size(output,2)+2);
output2(:,[1,2]) = intb;
x = [3;size(output2,2)];
output2(:,x) = output;
output = output2;

function net = nt_sort(net)
function net = nt_sort(net)
if net(4,1) == 0
  error('Net has not been checked for bad connections');
end
numlayers = nt_getl(net);
if net(5,1) == 0
  [recurrmat doneloops pendlloops] = nt_recur(net);
  %disp(recurrmat);
  %disp(doneloops);
  %disp(pendlloops);
  %pause;
  if doneloops == [] & pendlloops == []
    prev_layers = [0];
    count = 0;
    lid = nt_getl(net);
    newlid = lid;
    j = 1;
    while j <= size(newlid,2)
      %disp(j);
      %disp(newlid);
      %disp(count);
      %pause;
      k = newlid(1,j);
      cl = nt_getl(net,k);
      from = lr_getfr(cl,0);
      clin = lr_getin(cl);
      if from(:,1) == zeros(clin,1)
        cl = lr_setl(cl,1);
        net = nt_setl(net,k,cl);
        newlid(j) = [1];
        count = count+1;
        prev_layers(1,size(prev_layers,2)+1) = k;
      else
        j = j+1;
      end
    end
  end
end
if count == 0
    error('There must be at least one layer connected solely to the net inputs');
else
    curlvl = 2;
    flag2 = 1;
    while size(newlid,2) > 0 & flag2 == 1
        flag2 = 0;
        numleft = size(newlid,2);
        j = 1;
        prev_layers2 = [0];
        while j <= size(newlid,2)
            cl = nl_getl(net,newlid(1,j));
            from = lr_getin(cl,0);
            from = from(:,1);
            lnumin = lr_getin(cl);
            flag = 0;
            %disp('prev_layers:');
            %disp(prev_layers);
            %pause;
            k = 1;
            while k <= lnumin & flag == 0
                if contains(prev_layers,from(k,1)) == [0 0]
                    flag = 1;
                    %disp('found');
                end
                k = k+1;
            end
            if flag == 0
                %disp('pl2:');
                cl = lr_setlv(cl,curlvl);
                net = nt_setl(net,newlid(1,:),cl);
                prev_layers2(1,size(prev_layers2,2)+1) = newlid(1,);
                %disp(prev_layers2);
                newlid() = [];
                numleft = numleft-1;
                flag2 = 1;
            else
                j = j+1;
            end
        end
    curlvl = curlvl+1;
    spl = size(prev_layers,2);
    prev_layers2(1) = [];
    %disp('pl2 after deleting 0');
    %disp(prev_layers2);
    prev_layers(1,[:spl+1:spl+size(prev_layers2,2)]) = prev_layers2;
end
if flag2 == 0
    error('Net not contiguous');
end
else
    error('There are feedback loops in the net');
end
net(:,[3:numlayers+2]) = sort2(net(:,[3:numlayers+2]),row,5);
net(5,1) = 1;
end

%%%%%
function [h2,h3]=ploterr2(e,e2,g,h,ht)

% PLOTERR Plot network sum-squared error vs epochs.
%
%   PLOTERR(E,G)
%   E - Row vector of error values.
%   G - Error goal.
%   Returns (optionally) handle to error curve in plot.
%
%   PLOTERR(E,G,H)
%   H - Handle returned by previous call to PLOTERR.
%   Deletes old error curve H, and plots new one.

if nargin < 1, error('Not enough arguments'), end

epochs = length(e)-1;
t = sprintf('Mean-Squared Network Error for %g Epochs',epochs);

% BACKWARD COMPATIBILITY FOR NNT 1.0
% Convert PLOTERR(E,T) -> PLOTERR(E)

nargin2 = nargin;
if nargin2 == 3
    if isstr(g)
        t = g;
        nargin2 = 1;
    end
end

if nargin2 < 4
    newplot;
    if nargin2 == 3
        plot([0 999999],[g g],',0.9,0.9,0.9,0.9,');
    end
    xlabel('Epoch')
    ylabel('Mean-Squared Error')
    title(t)
    set(gca,'box','on')
else
    delete(h);
    delete(ht);
end

hold on
e = e + eps;
H = plot([0:epochs,e]);
H2 = plot([0:epochs,e2,'r-']);
title(t)
hold off

set(gca,'xlim',[0 epochs+eps]);
set(gca,'yscale','log');
drawnow

if nargout == 1
    h2 = H;
    h3 = H2;
end
function [xpts,ypts] = gridln2d(function,min_x,max_x,numblpts,numpts,numdiv)

clf;
X = ((0:numdiv).*((max_x - min_x)/numdiv)+(ones(1,numdiv+1)*min_x));
xdiffsq = (((max_x - min_x)/numdiv).^2);
Y = eval(function);
dists = diff(Y).^2;
dists = dists + ones(1,numdiv).*xdiffsq;
dists = sqrt(dists);
inc_dist = zeros(1,numdiv+1);
for i = 2:numdiv+1
    inc_dist(i) = inc_dist(i-1) + dists(i-1);
end
subplot(311);
plot(X,inc_dist);
subplot(312);
tot_dist = dists*ones(numdiv,1);
plot(X(1,1:numdiv),dist);
%k = (0:numlines-1)/(numlines-1);
k = rand(numbl,numlines);
k = k*tot_dist;
for j = 1:numbl
    for i = 1:numlines
        kdiff = abs(inc_dist-(ones(1,numdiv+1)*k(j,i)));
kmatch = min(kdiff);
k(j,i) = ((find(kdiff==kmatch)/numdiv))*(max_x - min_x) + min_x;
    end
end
subplot(313);
X2 = X;
X = k;
pause;
kpts = eval(function);
plot(X,kpts,Y);
hold on;
plot(X2,Y);
xpts = X;
ypts = kpts;
function [xpts,ypts,zpts,T] = gridln3d(xfunction,yfunction,zfunction,min_t,max_t,numlines,numdiv)
function [xpts,ypts,zpts,T] = rand3d(xfunction,yfunction,zfunction,min_t,max_t,numpts,numdiv)

clf;
T = ((0:numdiv).*((max_t - min_t)/numdiv)+(ones(1,numdiv+1)*min_t));
%T = rand(1,numdiv+1);
X = eval(xfunction);
Y = eval(yfunction);
Z = eval(zfunction);
dists = sqrt((diff(X).*2) + (diff(Y).*2) + (diff(Z).*2));
inc_dist = zeros(1,numdiv+1);
for i = 2:numdiv+1
    inc_dist(i) = inc_dist(i-1) + dists(i-1);
end
subplot(311);
plot(T,inc_dist);
subplot(312);
tot_dist = dists*ones(numdiv,1);
plot(T(1,1:numdiv),dist);
%k = (0:numlines-1)/(numlines-1);
k = rand(1,numlines);
k = k*tot_dist;
%k(1) = min_t;
%k(numlines) = max_t;
for i = 1:numlines
    kdiff = abs(inc_dist-(ones(1,numdiv+1)*k(i)));
    kmatch = min(kdiff);
    k(i) = (((find(kdiff==kmatch)/numdiv)*(max_t - min_t))+min_t);
end
%subplot(313);
T2 = X;
T = k;
%disp(k);
%pause;
xpts = eval(xfunction);
ypts = eval(yfunction);
zpts = eval(zfunction);
%plot3(xpts,ypts,zpts,'r');
%hold on;
%plot3(X,Y,Z);
%grid on;
%
function [a1,a2,a3,a4] = simuff4(p,w1,b1,f1,w2,b2,f2,w3,b3,f3,w4,b4,f4)
% a1 = feval(f1,w1*b1);
% a2 = feval(f2,w2*a1,b2);
% a3 = feval(f3,w3*a2,b3);
% a4 = feval(f4,w4*a3,b4);
% a1 = feval(f1,w1*b1);
% a2 = feval(f2,w2*a1,b2);
% a3 = feval(f3,w3*a2,b3);
% a4 = feval(f4,w4*a3,b4);
% agnostictestpts4

load teapot;
fig = figure('position', [100 300 900 600]);
a = axes('visible', 'off', 'position', [0 0 1 1]);
xfn_txt = text('string', 'X-Function', 'position', [25 .5], 'fontname', 'times', ...
    'horizontalalignment', 'center');
yfn_txt = text('string', 'Y-Function', 'position', [.75 .5], 'fontname', 'times', ...
    'horizontalalignment', 'center');
zfn_txt = text('string', 'Z-Function', 'position', [25 .25], 'fontname', 'times', ...
    'horizontalalignment', 'center');
crosssection_txt = text('string', 'Crosssection-Function', 'position', [.75 .25], 'fontname', 'times', ...
    'horizontalalignment', 'center');
maxd_txt = text('string', 'Domain of t', 'position', [.4 .9], 'fontname', 'times');
pt_txt = text('string', 'CS points', 'position', [.4 .7], 'fontname', 'times');
n_txt = text('string', 'Baseline points', 'position', [.05 .9], 'fontname', 'times');
alph_txt = text('string', 'Alpha', 'position', [.7 .9], 'fontname', 'times');
csx_max_txt = text('string', 'CS domain', 'position', [.7 .7], 'fontname', 'times');
cs_factor_txt = text('string', 'CS Relative size', 'position', [.05 .7], 'fontname', 'times');

xfn_edit = uicontrol('style', 'edit', 'units', 'normal', 'max', 1, 'min', 1);
set(xfn_edit, 'position', [.15 .4 .2 .04], 'string', xfn_def, 'backgroundcolor', [0 0 1],...
foregroundcolor', [1 1 1]);
yfn_edit = uicontrol('style', 'edit', 'units', 'normal', 'max', 1, 'min', 1);
set(yfn_edit, 'position', [.65 .4 .2 .04], 'string', yfn_def, 'backgroundcolor', [0 0 1], ... 
'foregroundcolor', [1 1 1]);
znf_edit = uicontrol('style', 'edit', 'units', 'normal', 'max', 1, 'min', 1);
set(znf_edit, 'position', [.15 .15 .2 .04], 'string', znf_def, 'backgroundcolor', [0 0 1], ... 
'foregroundcolor', [1 1 1]);
crosssection_edit = uicontrol('style', 'edit', 'units', 'normal', 'max', 1, 'min', 1);
set(crosssection_edit, 'position', [.65 .15 .2 .04], 'string', crosssection_def, 'backgroundcolor', [0 0 1], ... 
'foregroundcolor', [1 1 1]);

maxd_box = ctrl_box('value', maxd_def, 'step_size', .1, 'editable', 1, 'minimum', 0, 'maximum', ... 1000, 'units', 'pixels', 'position', [485 500]);
alpha_box = ctrl_box('value', alpha_def, 'step_size', .1, 'editable', 1, 'minimum', 0, 'maximum', ... 1000, 'units', 'pixels', 'position', [745 500]);

n_box = ctrl_box('value', n_def, 'step_size', 1, 'editable', 1, 'minimum', 0, 'maximum', ... 1000000, 'units', 'pixels', 'position', [230 500], 'int', 1);
csx_max_box = ctrl_box('value', csx_max_def, 'step_size', .1, 'editable', 1, 'minimum', 0, 'maximum', ... 1000, 'units', 'pixels', 'position', [745 380]);

cs_factor_box = ctrl_box('value', cs_factor_def, 'step_size', .1, 'editable', 1, 'minimum', 0, 'maximum', ... 1000, 'units', 'pixels', 'position', [230 380]);
p_box = ctrl_box('value', p_def, 'step_size', 1, 'editable', 1, 'minimum', 0, 'maximum', ... 1000000, 'units', 'pixels', 'position', [485 380], 'int', 1);

show_cs_button = uicontrol('style', 'pushbutton', 'string', 'Show CS', 'units', 'pixels');
set(show_cs_button, 'position', [100 35 100 25], 'callback', 'csa_cbl(get(gcf, "userdata"));');

show_bl_button = uicontrol('style', 'pushbutton', 'string', 'Show BL', 'units', 'pixels');
set(show_bl_button, 'position', [400 35 100 25], 'callback', 'sbl_cbl(get(gcf, "userdata"));');

execute_button = uicontrol('style', 'pushbutton', 'string', 'Execute', 'units', 'pixels');
set(execute_button, 'position', [700 35 100 25], 'callback', 'eb_cbl(get(gcf, "userdata"));');
ake = [yfn_edit, yfn_edit, znf_edit];
fake2 = [crosssection_edit, crosssection_edit, crosssection_edit];
udata = [fake, fake2, maxd_box, p_box, n_box, alpha_box, csx_max_box, cs_factor_box];

set(fig, 'userdata', udata);

function [gridx, gridy, gridz] = thegrid(w1, b1, f1, w2, b2, f2, inp, ns)

gridx = zeros(ns, ns);
gridy = zeros(ns, ns);
gridz = zeros(ns, ns);

ntc1 = min1:((max1-min1)/(ns-1)):max1;
ntc2 = min2:((max2-min2)/(ns-1)):max2;
gridx = zeros(ns, ns);
gridy = zeros(ns, ns);
gridz = zeros(ns, ns);
%ntc1 = -2*sqrt(ntc1 var)+ntc1 mean+(4*sqrt(ntc1 var))/(ns-1):ntc1 mean+2*sqrt(ntc1 var),
%ntc2 = -2*sqrt(ntc2 var)+ntc2 mean+(4*sqrt(ntc2 var))/(ns-1):ntc2 mean+2*sqrt(ntc2 var),
for i = 1:ns
    gi = [ntc1,ntc2(i)*ones(1,ns)];
    [a1,a2] = simuflgi.w1,b1,f1,w2,b2,f2);
    gridx(i,:) = a2(1,:);
    gridy(i,:) = a2(2,:);
    gridz(i,:) = a2(3,:);
end
%end of listing
References


