Prediction of Wind Speeds with an Artificial Neural Network

A Senior Project

Presented to

The Faculty of the Electrical engineering Department

California Polytechnic State University, San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Bachelor of Sciene

By

Justin Tracy

March, 2010

Table of Contents

List of Tables and Figures

Tables

Figures

Abstract

This project involved designing and programming an artificial neural network, testing its function, and testing the resulting function against existing wind speed measurement data, in order to determine the ability of an artificial neural network to learn the relationship between measured data and future wind speed. The artificial neural network includes gradient momentum, batch training, incremental training, and a function to test the results of the trained neural network against an additional set of data without back propagation. It was first found that the artificial neural network was able perform unsupervised learning, and learn the model for a XOR gate. At a learning rate of .1 and a gradient momentum of .05, the network was able to learn a XOR gate function to an absolute error of .025 in 450 iterations of batch training and testing found an absolute error of .019.

When testing the ability of the neural network to learn in a wind speed prediction application, it was found that an artificial neural network can produce a good prediction of the wind speed. The prediction of the wind speed in the next 1 minute interval was found during the testing portion of the run to reach an average absolute difference of .032. For the use of logarithmic scaling function, an average absolute difference of .0336 was obtained.

Comparing this to a non-linear regression technique for determining the coefficients in a wind speed model, the absolute error of .0234 from the traditional method was less than the ANN results.

Acknowledgements

I would first like to acknowledge and thank my senior project advisor, Helen Yu, for her help in completing this project. I would also thank the sources I consulted during the design, listed after the bibliography, especially the author of "Neural Networks: A Systematic Introduction", Raul Rojas; this book was a primary source of consultation for the design and testing of this project.

Chapter 1 Introduction

The ability to predict wind speed can serve a number of purposes; one of the most exciting is the use of wind prediction in order to manage the generation of power from wind farms. By predicting the production of power from wind farms, the produced power could be better distributed and managed. Foreknowledge of a sudden spike in the amount of power produced could allow a utility to better manage the overall distribution of its power, and help manage the amount of power produced at other sources to match the demand.

This would provide an additional feature for incorporation into a Smart Grid for electrical distribution, providing a method for estimation of the supply available. This would provide the Smart Grid with a means of predicting the supply available for distribution into the future from wind resources, providing the grid with the information it needs to shape demand to fit the supply available reliably.

There are a number of traditional methods for wind speed prediction, most of which involve the statistical analysis of wind speed data from the past and the creation of a static model from a mathematical method, such as least squares curve fitting or non-linear regression. The use of these techniques runs into a serious problem; these techniques produce a model which is tied to past data and unable to learn and adapt to changing conditions. This carries a potential problem in a system which is subject to varying conditions, due to interactions between variables within the atmospheric system, as well as interactions between the atmospheric system and

outside systems such as the oceans and the part evaporating ocean water plays in influencing atmospheric conditions.

A promising method for adaptive large scale wind speed prediction is the use of Artificial Neural Networks. Devising a model for every wind farm would be time consuming, energy intensive, and any significant change in the conditions assumed would require a new model. An artificial neural network could potentially learn the model for a for the wind speed by taking a set of data over time, possibly requiring only a limited amount of input information. If there was a change in conditions, an artificial neural network could learn the change over time, and adjust the model used for prediction.

Chapter 2 Background

Artificial neural networks are one of the hot topics in learning algorithm research today. Due to the ability of a generalized neural network to develop a model for a specific application, artificial neural networks hold great promise for a number of applications. Applications have been researched in industrial process and finance regions[1].

Another use that has been researched for neural networks is the use of neural networks for the prediction of wind speed. This is often carried out to provide benefit in the prediction of the power generation of wind farms. A notable example of this sort of research is a study in Spain, in which existing wind power generation models were improved by the inclusion of a neural network into the existing model [2]. Other researchers have looked into the prediction of wind speed directly from a neural network. "Comparison of Feedforward and Feedback Neural Network Architectures for Short Term Wind Speed Prediction" found short term wind speeds could be predicted to a maximum mean absolute relative error of .3892 during training and .4354 with a recursive neural network [3].

The meteorological data used comes from the National Wind Technology Center. The National Wind Technology Center is a renewable energy laboratory in Boulder, Colorado that records and provides meteorological readings minute by minute. Readings of average wind speed and direction are taken at several heights (2m, 5m, 10m, 20m, 50m and 80m) on the tower by sampling the sensors at 1microsecond and averaging the results over a minute. The peak and standard

deviation of the wind speed measurements are also recorded, as well as the direction at the peak wind speed.

Temperature is measured at three points on the tower (2m, 50m and 80m) once per minute and recorded. Pressure is measured at ground level once per minute, and the shortwave horizontal irradiance is measured once per minute several feet above ground level.

From these measurements, an accumulated irradiance, dew point temperature, relative and specific humidity, wind chill, and wind shear are calculated once per minute.

Chapter 3 Requirements

The requirement for this project is to develop an Artificial Neural Network (ANN) and test its accuracy in predicting wind speeds into the future based on current and past data measurements.

The biggest requirement of this ANN is the ability to demonstrate its ability to learn to match input conditions to output conditions. This will be demonstrated by training the ANN to learn the input to output relationship for an XOR function.

To define success in the training of the ANN, the ANN should produce an output with an average error $E = \sum_{r=1}^{N}(T_r - y_r)^2$ of less than 1% for the last 100 on line runs of each case.

The measurement of the success of the ANN in wind prediction will be its ability to, given current and past data for a specified time, output a value from the outer-most layer which corresponds closely with the actual data found at the next time interval, and its ability to learn to track the actual wind speed at the next future time interval.

As the future wind speed is likely difficult to predict given current and past data, the error considered acceptable for this test will be much greater than in the more predictable cases used for testing. Achieving an average error of less than 30% for the last 100 training run could be considered success in predicting wind speed, as this is the value of error reported previously in “Comparison of Feedforward and Feedback Neural Network Architectures for Short Term Wind Speed Prediction”.

Chapter 4 Design

The basic design of the ANN is based on a design described in Neural Networks: A Systematic Introduction [1], shown below in Figure 1. This uses an input layer of n input sites, with a 1 to serve as a bias value. The output of each input site and the bias value is added as a weighted sum to the input of each node in the next layer. Each hidden layer has k nodes, along with a 1 to use as a bias value. The weight sum value received at the left side of each node is passed through the activation function. Each of these nodes and the bias value is once again added as a weighted sum to the input of each node in the next layer. The weighted sum value passed to each of the m output nodes is passed through a final activation function and the resulting value is one of the outputs of the neural network.

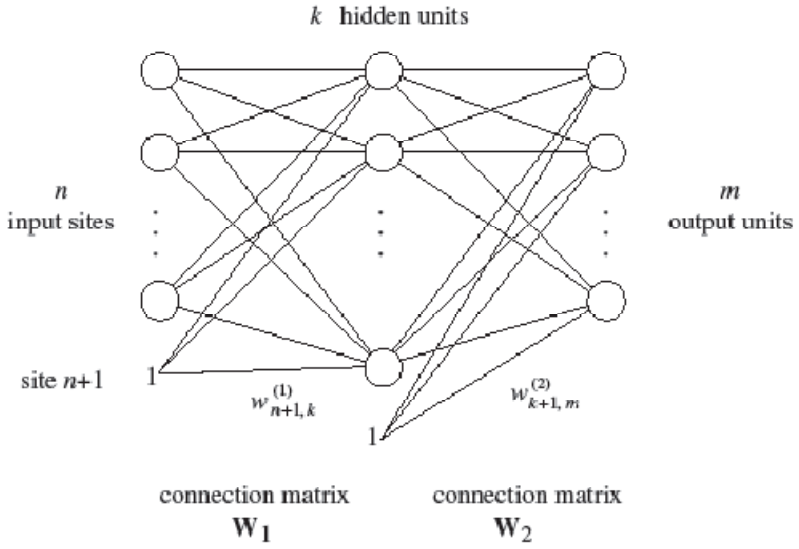


Figure 1: Basic design principle of the neural network

[4] “7.17 Notation for the Three Layered Network”, pg 165.

The training and testing is a three stage process: a batch training process to initially train a set of weight matrices to a minimum specified value for a full training set, an incremental training process to continue learning across a different training set, and an incremental feed forward system without back propagation to test the results of the training against a third set of data points.

The batch training portion is intended to train the function down to as small a value as possible for a representative data set. The batch training set is fed in and the weights are trained against the whole data set at one time. The value of error that the batch training will run to can be set by the user in the matrix, as can the maximum number of times the batch training can run in its attempts to reach this minimum error.

However, if there are a significant number of inputs and/or significant previous time values of inputs or previous outputs are used as new inputs, this batch training can become time consuming. It also raises the possibility of overtraining the weight matrix against a noisy or not fully representative data set used for batch training [5]. For this reason, the incremental training process is next run. This process feeds forward and back propagates once for only a single data point or a small batch, then repeats for a new data point.

The incremental learning process can be used to train against a large data set separate from that used in batch training in a reasonable amount of time, as the single data point training results in smaller, faster calculations. It trains more slowly but, since the batch training should have already trained the weights to reasonable

accuracy for the data set, the slower learning is offset by preventing the system from bogging down on a large training set like the batch training process. This process can also be used to verify the input to output relationship of the weight matrix and check for overtraining while still training, making it a useful way to check results. This incremental learning portion can also, by setting the Incremental_Runs_Size parameter to a value greater than one, be used as a bath training process which trains against a bath of data a single time, then captures a new set of data and moves to the next batch of data.

This portion of training randomizes the order in which it gathers the training sets to be used, which is intended to prevent a situation, likely in a system in which is time varying, in which the neural network learns different behaves toward the beginning and end of the training; if that occurred, by the end of the training the relationship learned at the beginning of the training could be forgotten. This would produce a system which is only learning the system it is provided at the time it is training against, and not learning the system in general. By scrambling the inputs during training, the learning is general, rather than specific to recent time in the training set.

Finally, the incremental feed forward for a separate data set is used to check the results of the output and error and verify that the training as produced weight which accurately describe the input to output relationship.

The first step in this program is to define the constants to be used, and read in the variables for use. Then the three processes described above can being. Each of

these processes is composed of a combination of a set of common steps. For the batch training process, the steps are needed are: Set up initial network cell arrays, until a low limit of error is reached feed forward the inputs and perform back propagation, re-randomize the weights and repeat again, and finally store the best weight matrix found and information on the error. For the incremental training process, the steps needed are: Reset the network cell array with a new input dataset, feed forward and back propagate, and store the error. For the testing, the need is to repeatedly feed in a single input dataset, feed forward, and store the error.

4.1 Definitions of constants defined before run

Learning_Rate: The learning rate to be used during back propagation.

Gradient_Momentum: The gradient momentum coefficient, or the amount of the last change to the weights to add during the change in weights during back propagation.

Max_RMSE: The maximum value of root mean square error during batch training that will stop batch training.

Max_Runs: The number of epochs that batch training can perform before batch training is stopped, whether or not the Max_RMSE value of root mean square error has been reached.

Batch_Size: The number of data inputs and outputs to read in, for use in batch training.

Incremental Run Size: The number of data inputs and outputs to read in, during the incremental run portion of the training process. If set to 1, the incremental run protion

of the process will be a true incremental training process. If this value is set to a value greater than 1, it will train one single time against a b

Incremental Runs: The number of times the incremental training portion will gather another set of data points. If Data_Points is set to 1, this is the number of points the neural network will incrementally train against. If Data_Points is greater than 1, this is the number of batches that will be gathered and trained.

Testing_Points: The number of data inputs and outputs to read in, for use during the testing process.

First_Input_Row: The first row of the tab delimited text file, from which to begin reading.

First_Input_Column: The first column of input data from the tab delimited text file, from which to begin reading.

Num_Input Columns: The total number of input columns of data in each time interval contained in the tab delimited text file.

First_Output_Column: The first column of output data from the tab delimited text file.

Num_Output_Column: The number of total output columns to the right of the first output column to read in.

Back_Time_Inputs: The number of previous values to gather for each input type.

Beta: The coefficient of the exponential in the symmetrical or non-symmetrical sigmoid function used as the activation function. Increasing the value increases the slope of the activation function, bringing the sigmoid closer to a true step function.

Prediction Time: How many time intervals ahead of the next output the output data should be gathered from.

4.2 Read in Data

The data is read in from tab delimited text files un-normalized into 6 arrays. The upper left, upper right, lower left, and lower right corners of the area to be read are calculated from the constants entered at the beginning of the program.

4.3 Normalize Data

The values in each column are normalized to -1 to 1 by finding the maximum and minimum values in each column of input and output data, and using the following equation: $input_{ij-normalized} = \frac{input_{ij} - min_{column\ i}}{max_{column\ i} - min_{column\ i}}$. These normalized results are stored in the variables described in Figure 2.

The data reading function also supports gathering previous time measurements of the input, and the movement of the prediction time step out from the one minute interval initially planned for.

4.4 Set up Initial Network Matrices

The matrices holding the values to be used for the network are set up next. Cell arrays are first defined for the weight matrix cell array, node activation function output, and node summing junction input. The weight matrix is defined as a row vector of cells of length equal to the number of boundaries between adjacent layers. The node activation function output is a row vector of length equal to the number of layers specified. The node summing junction input is a row vector input of length

equal to the number of layers other than the input layer, which is the same value as the number of boundaries between two adjacent layers, as it is the number of layers minus 1.

Each of these cell arrays are initially populated with data. Each cell of the weight matrix row vector receives a matrix with rows equal to the number of nodes in the layer which the weight will populate during feed forward and columns equal to one more than the number of nodes in the layer which the weight matrix will be multiplied by during feed forward, with contents randomized between -1 and 1. The addition of one to the length of the columns is to take into account the bias value weights. This weight matrix will be transposed when pre-multiplied by a node activation layer output, as during the design of the feed forward and back propagation functions, it was necessary to either transpose the weight matrix during the feed forward stage or the delta stage, and a decision was made that troubleshooting would be easier with a transpose during feed forward, as opposed to transposes during back propagation.

The contents of the first cell of the activation function is populated with the input values concatenated with initial bias values of 1. Each inner layer is initially populated with placeholders of 1, of a row length equal to number of data input sets provided as an input and column length equal to the number of nodes in the layer associated with that activation function matrix.

The contents of the summing junctions of each layer is initially populated with placeholders of 1, of a row length equal to number of data input sets provided as an

input and column length equal to the number of nodes in the layer associated with that summing junction matrix.

Two sets of delta W matrices are also created with an initial population of zero, of the same size as the weight matrix. One of these sets is used to store the amount by which to change the weights before adding the gradient momentum in the current back propagation cycle, and another holds the value of the last change in the weights to be used in adding the momentum.

4.5 Feed Forward

Feed Forward is a fairly simple to perform in Matlab, given the ability to perform matrix functions. Since matrix multiplication populates a cell in the ith row and jth column of the output column with the sum of the ith row of the first matrix, with each value weighted by a value in the jth column of the second matrix; performing weighted summations of the outputs of nodes in one layer into the inputs of nodes in the next layer can be accomplished in a single line of code. For each hidden layer, the activation function is calculated for each value in the summing node, then the layer is concatenated with a bias values of 1. The activation function used is the symmetrical sigmoid, or $y = (\frac{2}{1+e^{-2*Beta*x}} - 1)$. The weighted summation and activation function calculation is performed within a FOR loop which progresses the layer being calculated by one each iteration. For the final layer, the same process is carried out, but no bias values are concatenated to the activation function outputs.

4.6 Back propagation

Back propagation is an attempt to minimize the output error through the use of the delta rule. The delta rule is a means of accomplishing gradient descent, or movement in a step in the direction in which the derivative of the function is most negative. The error present at the output node is calculated as $E = \sum_{r=1}^{N}(T_r - y_r)^2$ [4]. According to the delta rule, the delta value of this function at the node is the derivative of the activation function used, for the symmetrical sigmoid function $f = (1 + y)(1 - y)$, as the output of the node multiplied by the difference between the actual and expected output [4]. At a hidden layer, the delta value is derivate of the output of the node multiplied by the sum of the connecting values and deltas from the next node toward the output [5]. To find the partial derivate of the error function with respect to a weight value, the delta value of the node at which each weight connection terminates is multiplied by the output from which the weight originated.

The purpose of back propagation is to adjust each weight in the direction opposite the partial derivative, in order to decrease the error with respect to that weight [4]. Since the direction of the partial derivative has been calculated, the weight is adjusted by a value equal to the magnitude of the partial derivative multiplied by a constant of much less than 1, known as the learning rate, in the direction of minimizing the error function.

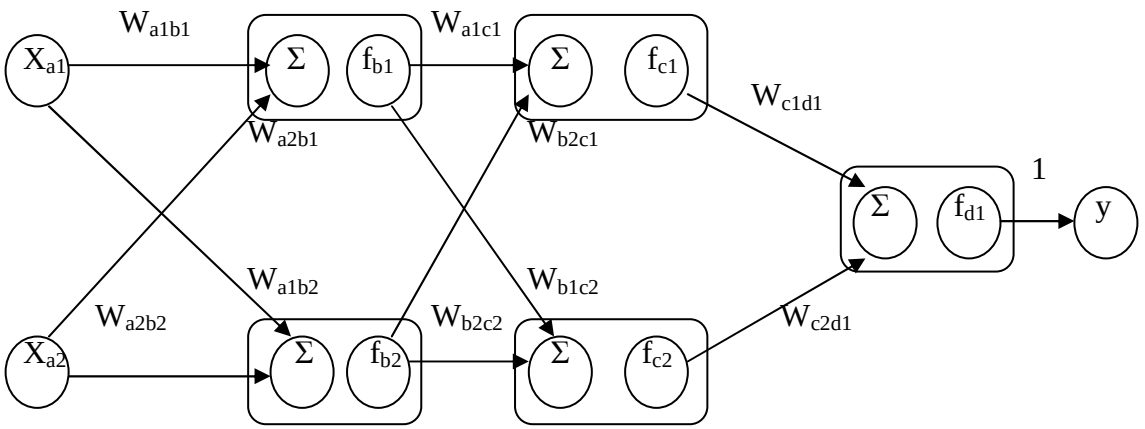


Figure 2 : Example neural network for explanation of back propagation

$$\delta_{d1} = \frac{\mathrm{dy}}{\mathrm{dw}}(T - y) = f'_{d1}(T - y)$$
$$\delta_{c1} = \frac{\mathrm{df_d}}{\mathrm{dw}}(W_{c1d1}\delta_{d1}) = f'_{c1}(W_{c1d1}\delta_{d1})$$
$$\delta_{c2} = \frac{\mathrm{df_d}}{\mathrm{dw}}(W_{c2d1}\delta_{d1}) = f'_{c2}(W_{c2d1}\delta_{d1})$$
$$\delta_{b1} = \frac{\mathrm{df_c}}{\mathrm{dw}}(W_{b1c1}\delta_{c1} + W_{b1c2}\delta_{c2}) = f'_{b1}W_{b1c1}\delta_{c1} + f'_{b1}W_{b1c2}\delta_{c2}$$
$$\delta_{b2} = \frac{\mathrm{df_c}}{\mathrm{dw}}(W_{b2c1}\delta_{c1} + W_{b2c2}\delta_{c2}) = f'_{b1}W_{b2c1}\delta_{c1} + f'_{b1}W_{b2c2}\delta_{c2}$$

Figure 3 : Equations for calculation of delta values in example neural network of Figure 2

$$\Delta w_{a1b1} = \eta\delta_{b1}X_{a1} + \theta\Delta w_{a1b1}$$
$$\Delta w_{a2b1} = \eta\delta_{b1}X_{a2} + \theta\Delta w_{a1b1}$$
$$\Delta w_{a1b2} = \eta\delta_{b2}X_{a1} + \theta\Delta w_{a1b1}$$
$$\Delta w_{a2b2} = \eta\delta_{b2}X_{a2} + \theta\Delta w_{a1b1}$$
$$\Delta w_{b1c1} = \eta\delta_{c1}f_{b1} + \theta\Delta w_{a1b1}$$
$$\Delta w_{b2c1} = \eta\delta_{c1}f_{b2} + \theta\Delta w_{a1b1}$$
$$\Delta w_{b1c2} = \eta\delta_{c1}f_{b1} + \theta\Delta w_{a1b1}$$
$$\Delta w_{b2c2} = \eta\delta_{c2}f_{b2} + \theta\Delta w_{a1b1}$$
$$\Delta w_{c1d1} = \eta\delta_{d1}f_{c1} + \theta\Delta w_{a1b1}$$
$$\Delta w_{c2d1} = \eta\delta_{d1}f_{c2} + \theta\Delta w_{a1b1}$$

Figure 4 : Equations for change in weight calculations for Figure 2

In order to demonstrate how back propagation would be carried out, an example neural network has been included as Figure 2 [6].

η = Learning rate
$\Delta w_{\$\$\#\#}$ = Change in weight linking node $$ and ##
$\delta_{\#\#}$ = Delta Value derived from delta rule for node ##
$f'_{\#\#}$ = Output of activation function for node ##
θ = Gradient momentum
T = Target output value
y = Actual output value

Symbol legend

The associated formulas for the delta of each node have been derived and included as Figure 3, and the weight change equations based on these delta values have been included as Figure 4.

In the neural network, a gradient momentum term has also been included, as seen in Figure 4. Due to the likelihood of local minima in the gradient of the error function, which could catch and hold the weight adjustment, even though the error has not been fully minimized, each time the weights are updated a portion of the last

update of the weights is added to the current update [5]. The portion of the update which carries over is specified with the gradient momentum term, which is initialized at the beginning of the run.

Chapter 5 Testing Plans

Testing of the neural network will first be carried out against the input to output relationship of a XOR gate. Three sets of input and output data will be set up in a tab delimited text file for the main function to read and provide data points for each of the three processes of the neural net program. The ability of the back propagation function to train the weights to match this relationship will be observed. The value of the error function and the root mean square error against the number of iterations run during each of the three processes and the expected versus actual values and the difference between the two for the incremental processes will be plotted, and these results used to confirm the errors are minimized with time and that the neural network is training to a reasonable accuracy.

Once it is confirmed that learning is occurring with the neural network, the neural network will be trained against wind speed data obtained from the National Renewable Energy Laboratory's "National Wind Technology Center" [7]. The data used will be for a specified time period over the year of 2009, and most of the testing will be performed over the first two months of the year. For this purpose, one year of data, spanning the year 2009, has been gathered. This consists of 52,560 one minute interval data points.

Three separate sets of data will be used during the wind speed prediction testing; one for batch training, another for incremental training, and a third for testing. The size of each set of data will be adjusted to find a size during the testing period to find a size for each data set which strikes a good balance between time required for

run and accuracy of the final result. For starting purposes, the size of the batch training portion of the run will be 12 hours (720 one minute intervals), the size of the incremental training set 5 days (7200 one minute intervals), and testing set will span 2 hours (120 one minute intervals).

Based on research, it appears the wind speeds and its standard deviation over the minute of measurement, temperature, wind direction, relative humidity, horizontal irradiance, wind gust and station pressure have the most correlation to the predicted value of the next wind speed [3][9]. These inputs, as measured at 80m, will be used, along with the time of the current wind speed and day of the year, in order to make the prediction of the average wind speed over the next one minute interval at a height of 80m. The number of delayed inputs of the data will begin at 1; if the accuracy increases with an increase in delayed inputs, more delayed inputs will be used.

As a means of judging the function of the results from the neural network, they will also be compared with the results of a traditional method. In this case, non-linear regression will be used as the baseline for comparison. Matlab provides the nlinfit() function which provides a nonlinear regression for a defined function structure. The model function used is a weighted summation of current and past time measurements, with the weights to be determined by regression [10]. The accuracy of this traditional model in wind speed prediction will serve as a baseline for comparison of the accuracy of the neural networks.

Chapter 6 Development and Construction

Development began with the development of feed forward and back propagation functions. The explanation of back propagation for general matrices from Neural Networks: A Systematic Introduction was turned into a set of Matlab steps for performing back propagation in Matlab, and from these steps the back propagation function was developed [4]. The Feed Forward function was easily implemented with simple matrix multiplication in Matlab.

The error calculations were designed next, as these were used to verify the function of the back propagation in reducing the error of the output from back propagation. The absolute error is calculated with the formula below:

$$error_{absolute} = \frac{|T - y|}{T}$$

At this point a function to read in inputs from a tab delimited text file was developed, and the three process design of the neural network was built from the feed forward and back propagation functions described above. This, when tested against a XOR input to output relationship, read in a text file and accomplished a minimization of error. At this point, to easily measure the inputs, outputs and errors, plotting functions were written.

Chapter 7 Integration and Test Results

7.1 XOR Test Results

The neural network was provided with a text file containing A and B columns and an output column. The run used a network with two hidden layers, the following constants and the following XOR truth table:

```
Learning_Rate = .1;
Gradient_Momentum = .05;
Max_RMSE = .025;
Max_Runs = 1000;
Batch_Size = 10;
Incremental_Run_Size = 1;
Incremental_Runs = 1000;
Testing_Size = 4;
Back_Time_Inputs = 0;
Beta = 1.5;
Prediction_Time = 1;

Hidden Layer 1 Nodes: 3
Hidden Layer 2 Nodes: 3
```

Table 1: Defined constants for XOR training

Output = (Input 1) XOR (Input 2)

Input 1	Input 2	Output
0	0	1
0	1	0
1	0	0
1	1	1

Table 2: Truth Table for XOR function

The learning rate and gradient momentum were set to mid-range values to prevent the possibility of instability, while also preventing the possibility of the weights getting stuck in a local minima. Beta, which determines the slope of the activation function, was set to 1.5 rather than 1 to give a more step-like activation function and push the outputs to saturate closer to 0 and 1, since the expected outputs should be only zeros and ones. The size of the training sets were chosen to make the program run to a final high degree of accuracy. Due to the speed with which it was possible to train the weights, this program ran extremely fast during training. The testing points were set to correspond to the inputs and outputs of the truth table.

This resulted in the batch training portion of the code training the network down to the required root mean square error of .025 in 450 epochs of back propagations, as shown in Figure 5. To verify the input to output relationship for a XOR gate, the graphs of the expected and actual input and output for the test have been included below as Figures 7. To demonstrate the error resulting, the absolute difference between the expected and actual output is included as Figure 8. The average of the absolute difference after training was .019. This confirms the function of the artificial neural network in learning a non-linear function.

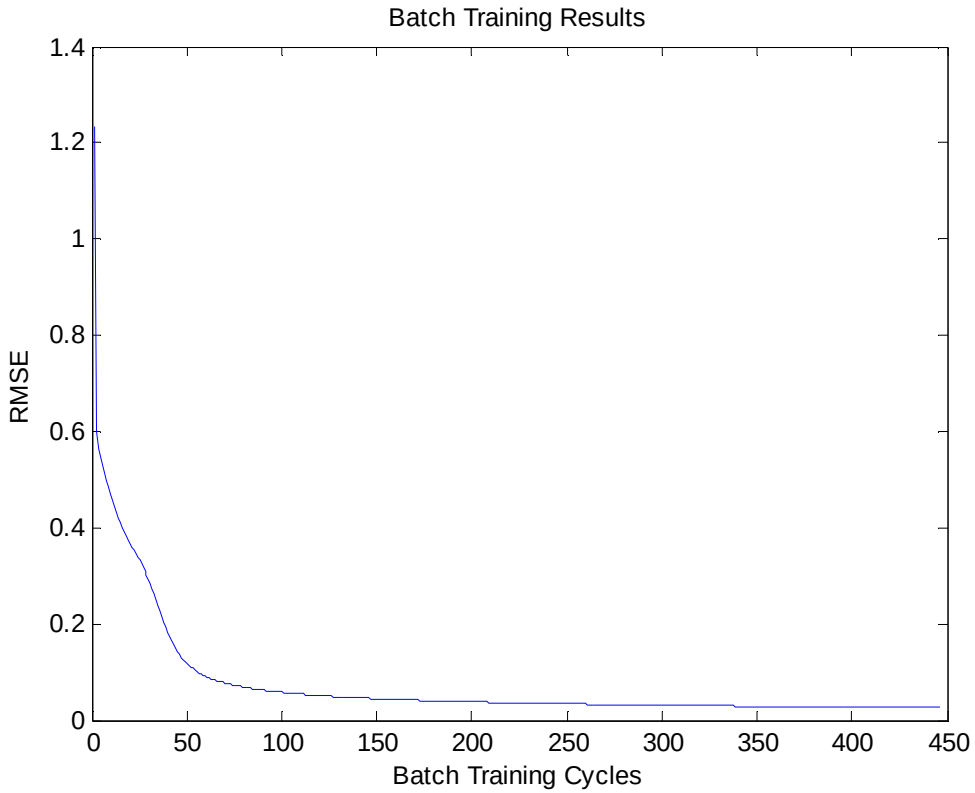


Figure 5: Batch Training Root Mean Error Reduction for XOR Batch Training

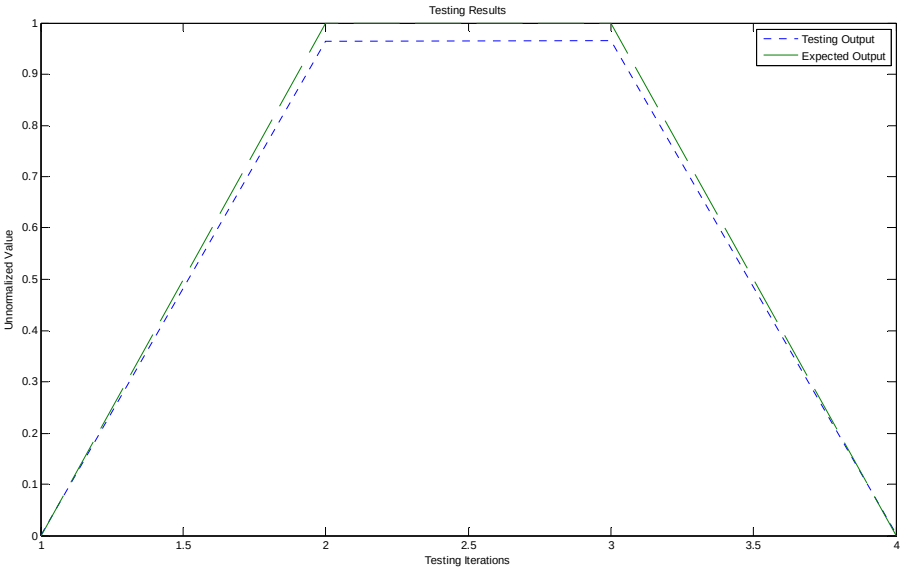


Figure 6: Testing Results: Expected vs. Actual Output for XOR Testing

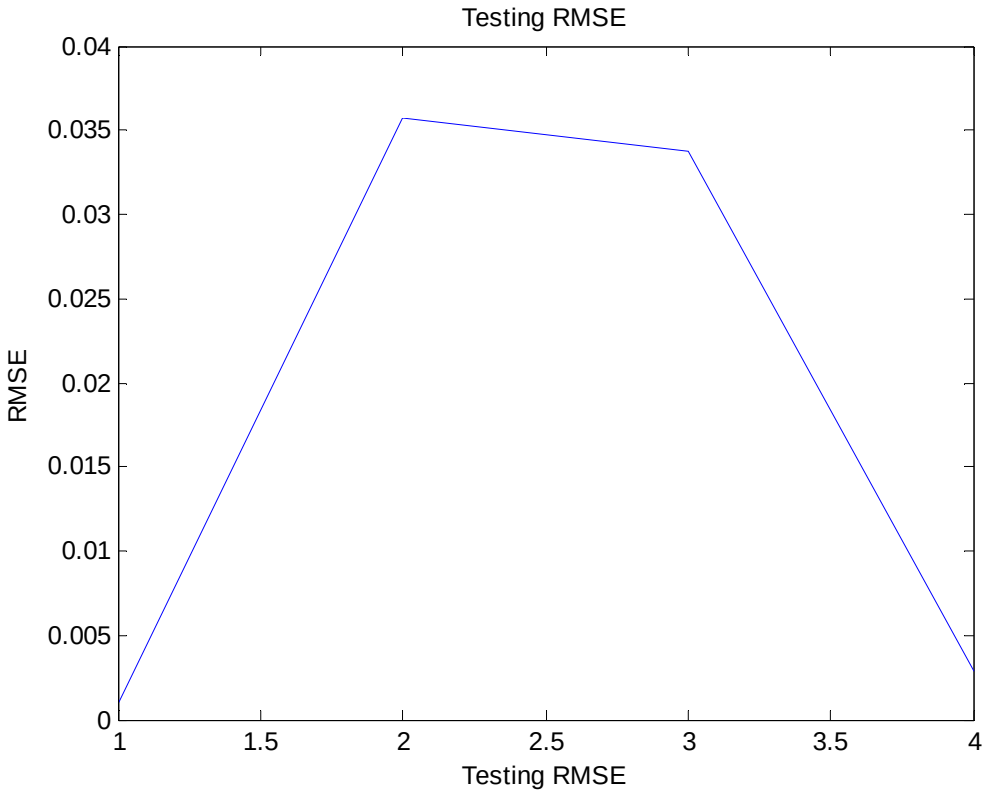


Figure 7: Absolute Difference resulting from XOR Testing

7.2 Wind Speed Results 1

The neural network was provided with the Windspeed_Data.txt file described under Program Listing in the Appendix. The run was made with two hidden layers and used the following constants:

Learning_Rate = η = .1;	First_Input_Row = 100;
Gradient_Momentum = θ = .05;	First_Input_Column = 1;
Max_RMSE = .025;	Num_Input_Columns = 10;
Max_Runs = 1000;	First_Output_Column = 3;
Max_Tries = 1;	Num_Output_Columns = 1;
Batch_Size = 1440;	Delayed_Time_Inputs = 1;
Incremental_Run_Size = 1;	Beta = .95;
Incremental_Runs = 1440*6;	Prediction_Time = 1;
Testing_Size = 120;	

```
Hidden Layer 1 Nodes: 10
Hidden Layer 2 Nodes: 5
```

Table 3: Defined constants for Wind Speed Prediction

The learning rate and gradient momentum were picked to be relatively small numbers, which slows learning but prevents an unstable stable system which oscillates around a correct weight matrix which it is trying to reach, but repeated overshoots during the weight matrix update, because of the large step size. The number of points to read were chosen to provide a big enough sample for learning to occur and for the training to be verified against a large variation in the wind speed, in order to simulate prediction in a time frame large enough to take into account many possible atmospheric conditions, but not so large to require a prohibitive run time. Beta was chosen as .95 in this case to provide a slightly smaller slope to the activation function, and thus a little more differentiation toward middle range outputs.

Batch training was unable to train the network all the way down to the .025 average root mean square error specified for this data set – the batch training reached .014 average root mean square error after 80 epochs and held close to constant, as shown in Figure 9. During the incremental training portion of the test, the resulting average absolute error for the normalized output seen was .07 and the standard deviation of the resulting absolute error was .041. The expected vs. actual output for the testing data set have been included as Figures 10. The average absolute difference between the expected and actual output was .018 during the incremental training portion. Finally, for the final testing portion of the test, the resulting average absolute error of the normalized output was .032, with a standard deviation of .052. The graph of the absolute value of the difference versus iterations has been included in Figure 11.

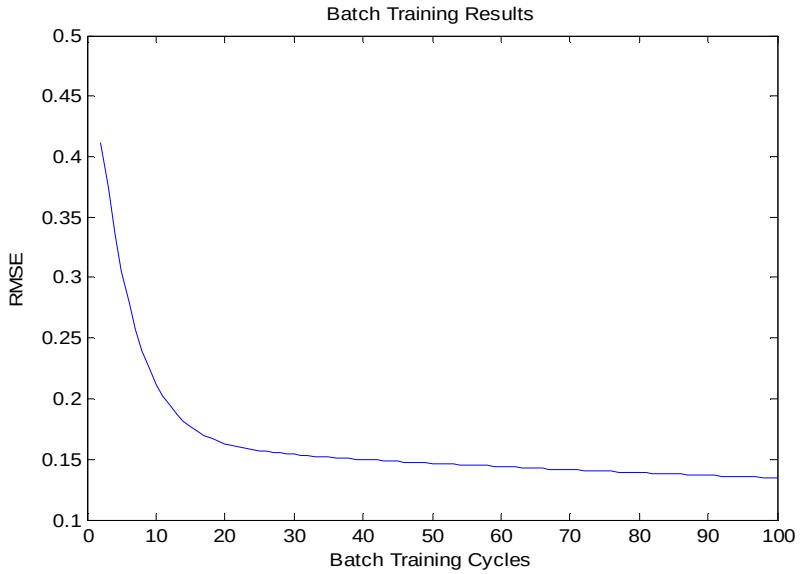


Figure 8: Batch Training – Normalized Root Mean Square Error vs. Iterations

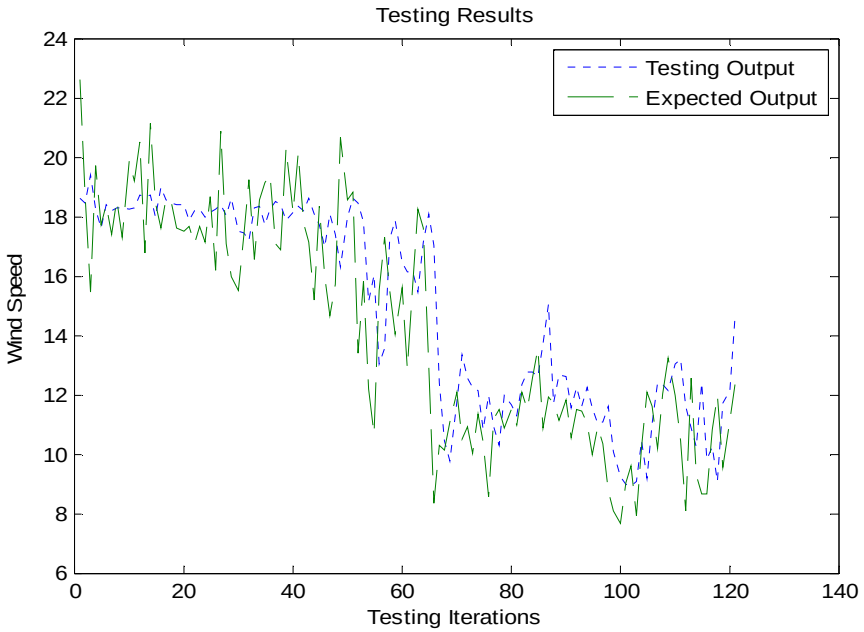


Figure 9: Testing – Actual vs. Expected Wind Speed (mph) for Wind Speed Prediction

Figure 10: Testing – Normalized Difference between Average and Expected Wind Speed

7.3 Idea on Improving the Results

The results obtained might be improved by devising a normalizing process which would normalize the expected output value data non-linearly between the upper and lower bounds of the activation function. During testing, it was observed that low values of wind speed are much more common than high values of wind speed, and it seems as if the neural network has a more difficult time predicting lower wind speeds. If the expected target values at low wind speeds were more easily differentiated between, the network would have an easier time differentiating between the majority of the points, improving its training of the weight matrix to minimize the output error seen during training and testing. This could be accomplished by normalizing with $input_{ij-normalized} = \left(\frac{1}{.35}\right) * \ln\left(\frac{1}{1.18} * \frac{input_{ij} - min_{column\ i}}{max_{column\ i} - min_{column\ i}} + 1\right) + (.24)$ [8]. This produces a distribution from -1 to 1, with Figure # showing the the expanded distribution of inputs from 0 to .4.

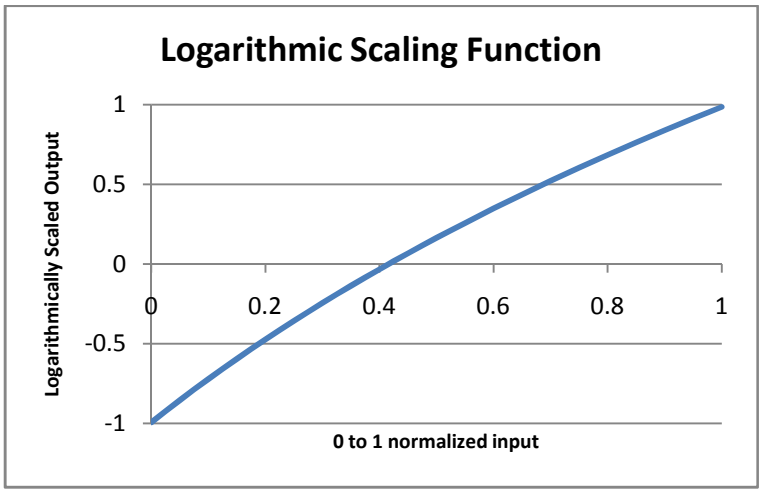


Figure 11: Logarithmic scaling function

7.4 Wind Speed Results 2

The neural network was provided with the Windspeed_Data.txt file described in the Appendix. The normalization function was modified to use the logarithmic function described on the previous page. This also required that the back propagation step be modified to use $f' = (y)(1 - y)$. The run was made with two hidden layers and used the following constants:

Learning_Rate = .1;	First_Input_Row = 100;
Gradient_Momentum = .05;	First_Input_Column = 1;
Max_RMSE = .025;	Num_Input_Columns = 10;
Max_Runs = 1000;	First_Output_Column = 3;
Max_Tries = 1;	Num_Output_Columns = 1;
Batch_Size = 1440;	Delayed_Time_Inputs = 1;
Incremental_Run_Size = 1;	Beta = .95;
Incremental_Runs = 1440*6;	Prediction_Time = 1;
Testing_Size = 120;	

```
Hidden Layer 1 Nodes: 10
Hidden Layer 2 Nodes: 5
```

Table 4: Defined constants for Wind Speed Prediction

Batch training trained the network down to the .0435 average root mean square error specified for the batch training data set after 1000 epochs, as shown in Figure 12. During the incremental training portion of the test, the resulting average absolute error for the normalized output seen was .0926 and the standard deviation of the resulting absolute error was .0773. For the testing data set, the resulting average absolute difference from the normalized output was .0336, with a standard deviation of .0279. The absolute error of the testing portion has been included in Figure 13.

The unnormalized expected vs. actual output graph has been included as Figure 14.

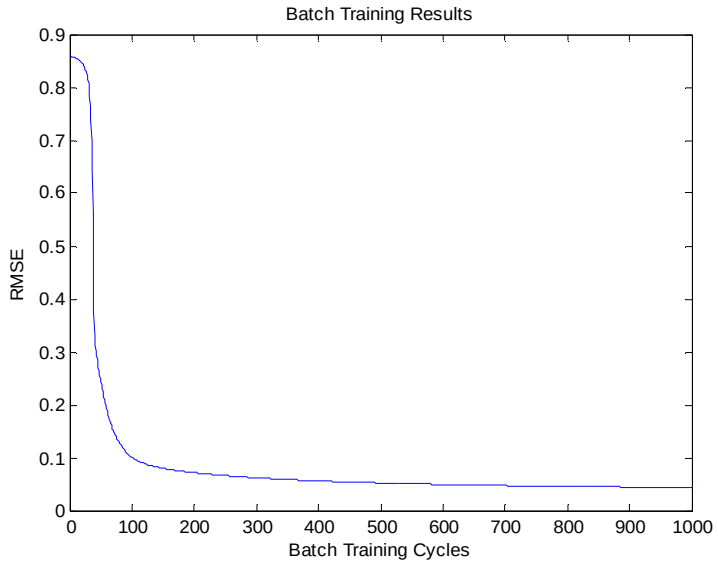


Figure 12: Batch Training – Root Mean Square Error vs. Epochs for Wind Speed Prediction

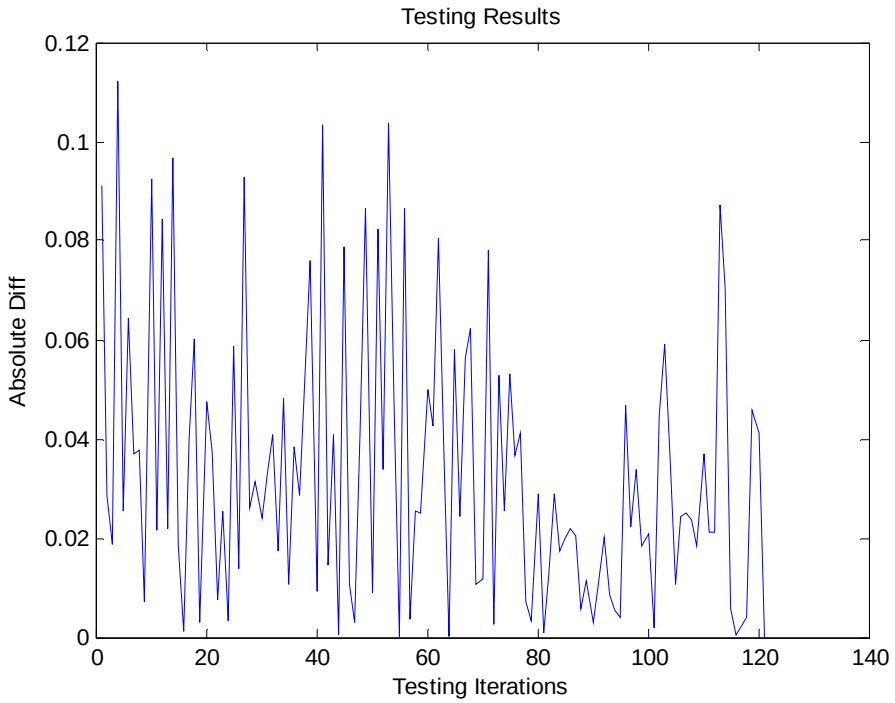


Figure 13: Absolute Difference for Incremental Training and On-line Testing for Wind Speed Prediction

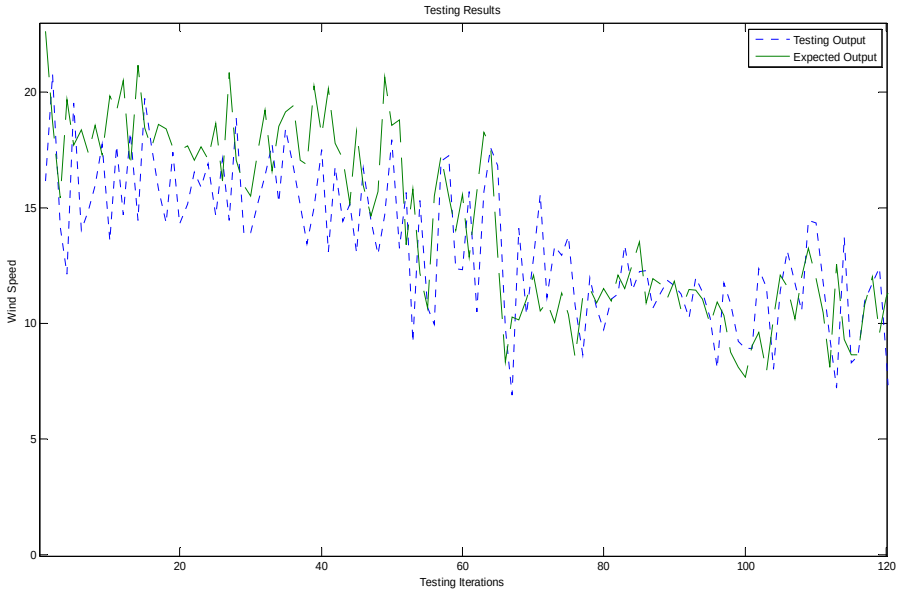


Figure 14: Testing – Actual vs. Expected Output Wind Speed (mph) for Wind Speed Prediction

7.5 Traditional method of Wind Speed Prediction

In order to provide a comparison for the ANN results, a wind speed model was developed through a traditional method – non-linear regression. This is carried out using the nlinfit() function, which estimates non-linear coefficients for a non-linear regression, using a least squares estimation. The inputs against which the curve fitting was carried out were the same as those used in the tests described in Wind Speed Results 1 and 2 – seven days worth of data starting from 1/1/09, with the same inputs as those of the artificial neural network, including the previous time input of each input.

The model function used is the second order system included below [10]:

$$y = \sum_{k=1}^{M} \beta_k {x_k}_t^{\,2} + \beta_{M+k} {x_k}_t + \beta_{2M+k} {x_k}_{t-1}$$

where y is the output, x_{kt} is the kth input, x_{kt-1} is the kth input one time period back in time, and M is the total number of inputs. The nlinfit() function, is provided with the same input data sets used as the input for batch and incremental training of the neural network for the x values. The output data sets of the next minute's wind speed measurement are also provided for the y values, and the nlinfit() function determines the β parameters that will provide a least squares fit of the input and output data sets.

Testing of this model was then carried out on the same testing output which as used for the artificial neural network results. The resulting average absolute difference was .0234, with a standard deviation of .0184. The resulting graph of wind speed prediction vs. actual wind speed has been included in Figure 5 below.

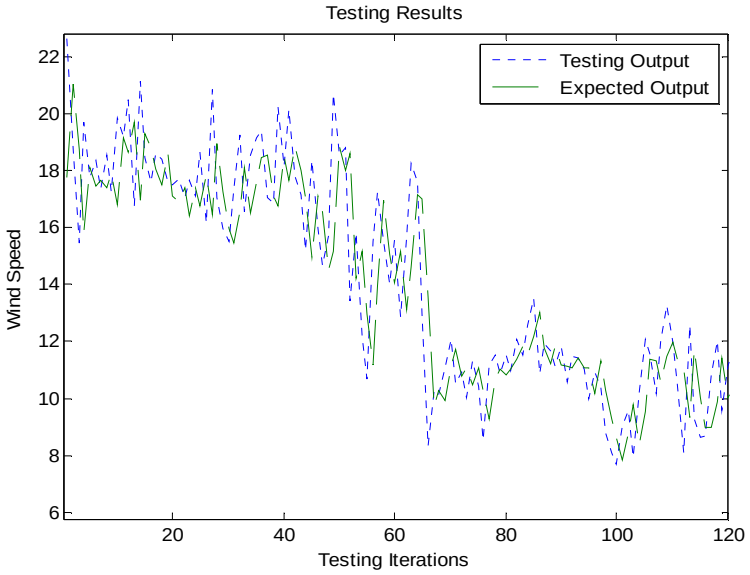


Figure 15: Testing – Actual vs. Expected Wind Speed (mph) for Traditional Wind Speed Prediction

Chapter 8 Conclusion and Recommendations for Future Works

As can be readily seen from the results of the XOR test, the artificial neural network has the ability to train a weight matrix to approximate a non-linear function. The wind speed results also demonstrate the artificial neural network successfully learning the general trend of the wind speed at any given time; however, it also demonstrates the problems the neural network has attempting to learn such a noisy signal. The neural network can be seen to follow the trend of the expected output and anticipate the sudden changes in average level over a range of points; however, the neural network was unable to learn and predict the sudden spikes in the expected output. The results obtained correspond closely to those obtained in "Comparison of Feedforward and Feedback Neural Network Architectures for Short Term Wind Speed Predictions". The authors of that paper obtained results for a similar test of a recurrent artificial neural network showing a mean absolute relative error of .38, a value close to that obtained during testing [3].

For wind speed test results 1, some of the higher error values, especially during testing, can be explained by the fact that, at that time, the expected output was hitting the minimum value which was included during the range being tested. In order for the output to output the expected value, the activation function would have needed to pull much more strongly down on the value than was common during the training. Due to the use of one-half for beta, which provides an activation function that is less steep, it becomes difficult for the activation function to pull up or down to the maximum or minimum values of the output, explaining the better tracking seen

when the output was closer to the middle values that the activation function differentiates more easily with a lower beta value.

It was hoped that normalizing with a logarithmic function to obtain a better spread of values over the more common lower values of wind speed would correct these issues, and produce lower average errors. This appears to be the case. The measures of average absolute difference between the expected and actual runs show a smaller difference using the logarithmic normalization function, and this result is repeatable.

Unfortunately, the use of the artificial neural network in wind prediction does not appear to produce better results than the results obtained through the more traditional method of regression. The traditional regression technique produced a better prediction for the training and prediction scheme used in this test.

However, this test was performed with the testing performed in one time interval shortly after the end of the training. While the neural network does not perform as well just after the end of the training period, the adaptive nature of the neural network would likely provide an advantage over time; once the regression is performed, the values are determined and help in place, while the neural network would continue to grow and learn new conditions, such as changing seasons. The comparison between the result of regression producing a model and a continually training neural network over time would be a good area in which to continue research in this area.

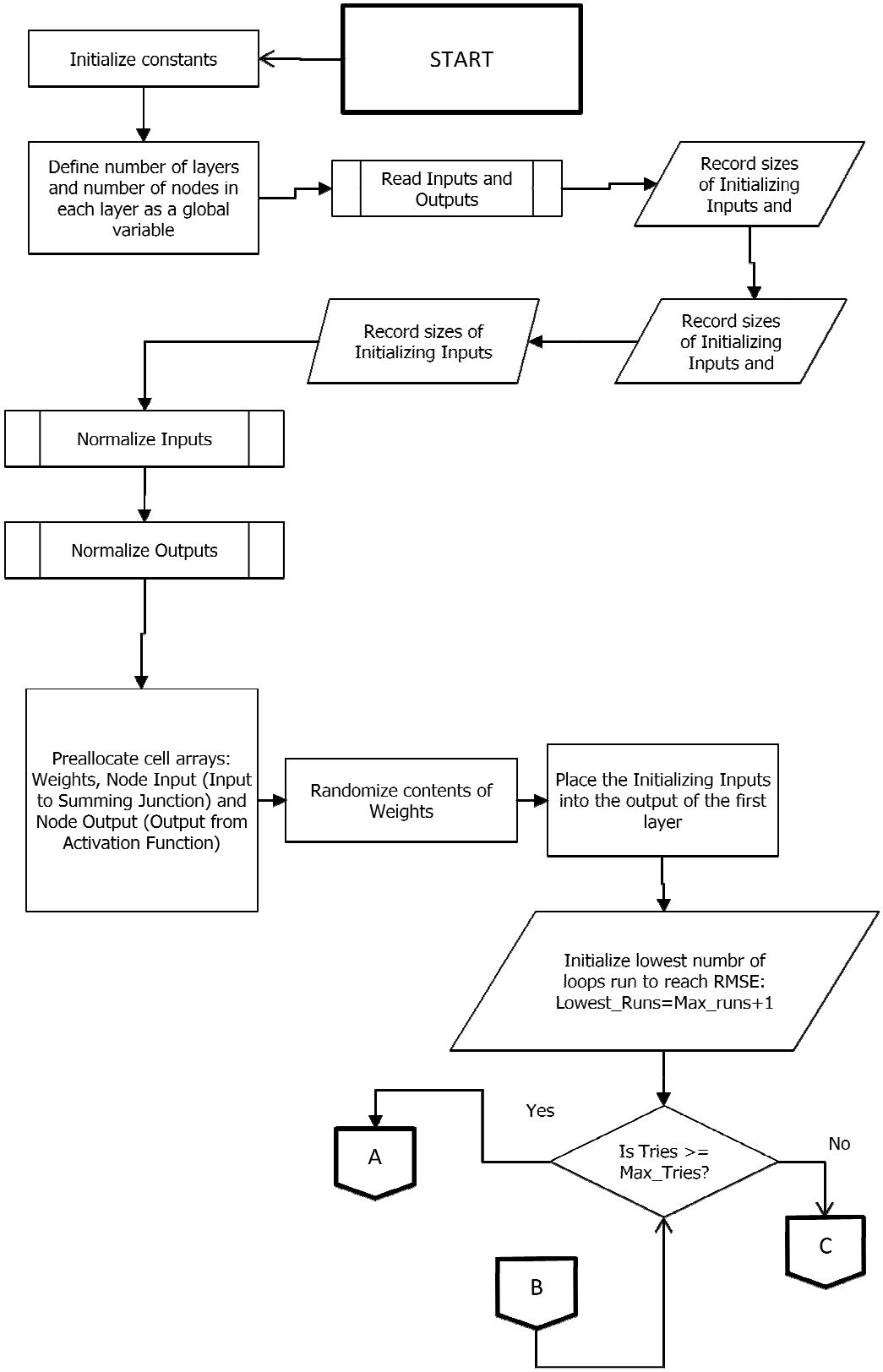
References

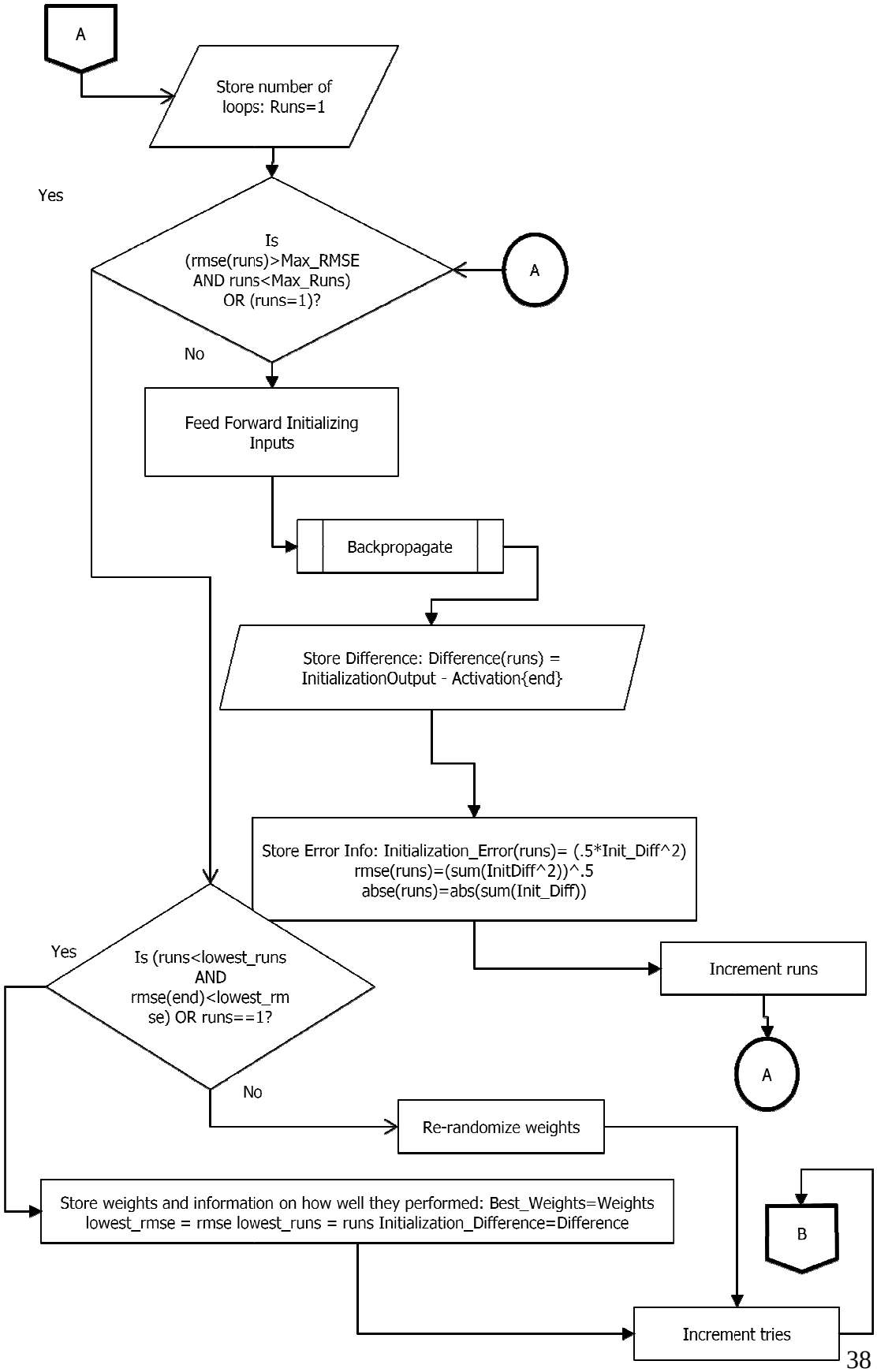
1) Christos Stergiou and Dimitrios Siganos**.** NEURAL NETWORKS. http://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol4/cs11/report.html.
2) Science Daily. http://www.sciencedaily.com/releases/2009/04/090430081233.htm.
3) Richard Welch, Stephen M. Ruffing and Ganesh K. Venayagamoorthy. "Comparison of Feedforward and Feedback Neural Network Architetures for Short Term Wind Speed Prediction". Proceedings of International Joint Conference on Neural Networks, Atlanta, Georgia, USA, June 14-19, 2009
4) Neural Networks - A Systematic Introduction. *Raul Rojas.* Springer-Verlag, Berlin, New-York, 1996.
   Individual chapters available at http://page.mi.fu-berlin.de/rojas/neural/index.html.html and whole book available at http://page.mi.fu-berlin.de/rojas/neural/neuron.pdf.
5) Technology Exponent. Neural Networks - An Introduction. http://www.tek271.com/?about=docs/neuralNet/IntoToNeuralNets.html.
6) Principles of training multi-layer neural network using backpropogation. http://galaxy.agh.edu.pl/~vlsi/AI/backp_t_en/backprop.html.
7) National Wind Technology Center M2 Tower. http://www.nrel.gov/midc/nwtc_m2/.
8) T.M.Lillesand and R.W.Kiefer. Remote Sensing & Image Interpretation. 6th Edition, 2007, John Wiley & Sons, Inc.
9) K. Sreelakshmi, P. Ramakanthkumar. Neural Networks for Short Term Wind Speed Prediction. Proceedings of World Academy of Science, Engineering, and Technology. Volume 32. August 2008.
10) Sagar Gadsing. Wind Speed Prediction. http://web.cecs.pdx.edu/~edam/Presentations/2004/Gadsing.pdf

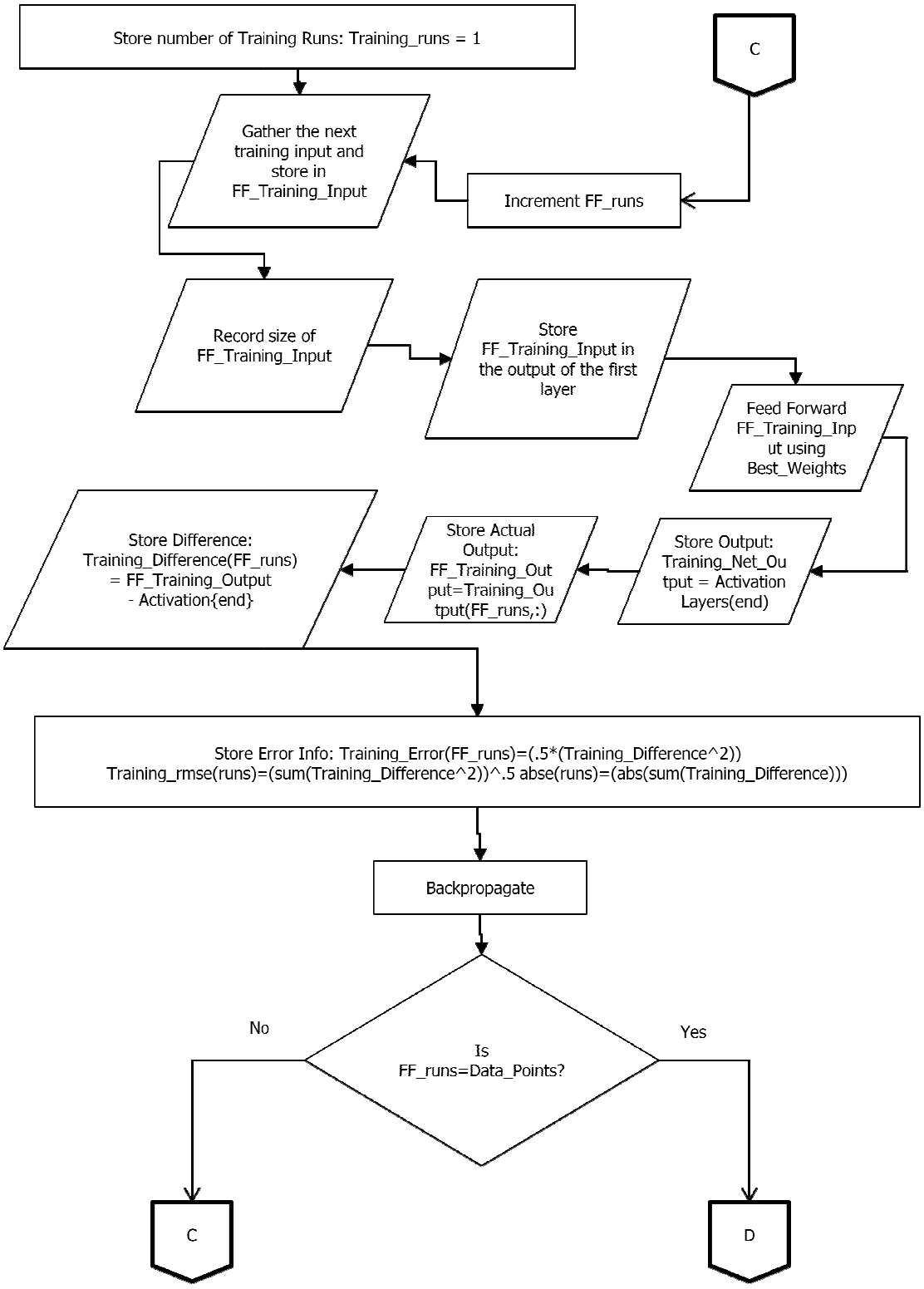
Sources Consulted During Design

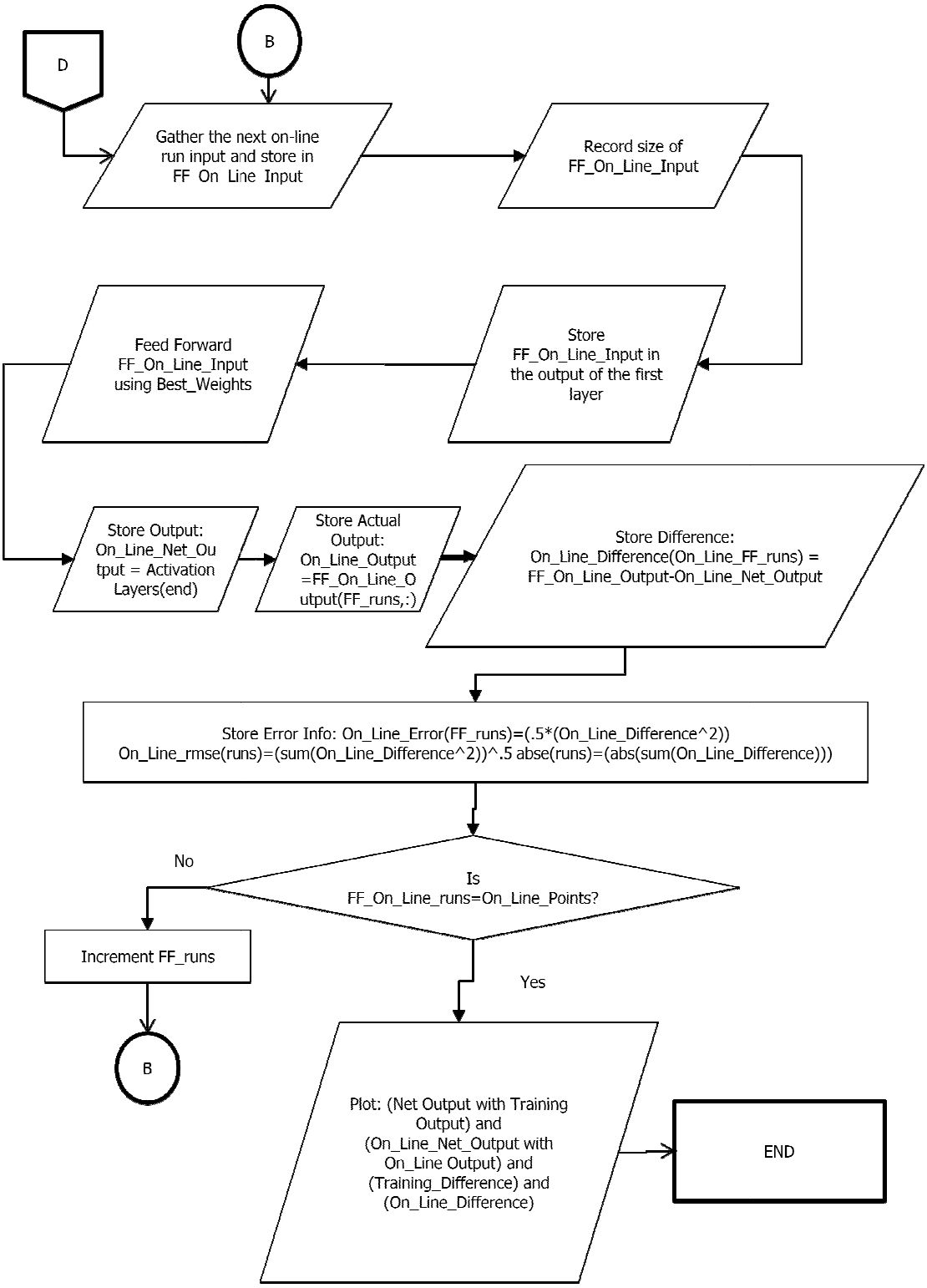
1) Neural Networks - A Systematic Introduction. *Raul Rojas.* Springer-Verlag, Berlin, New-York, 1996.
   Individual chapters available at http://page.mi.fu-berlin.de/rojas/neural/index.html.html and whole book available at http://page.mi.fu-berlin.de/rojas/neural/neuron.pdf.
2) Technology Exponent. Neural Networks - An Introduction. http://www.tek271.com/?about=docs/neuralNet/IntoToNeuralNets.html.
3) Back-propogation Neural Net Example in C++. http://www.codeproject.com/KB/recipes/BP.aspx.
4) 59.771 Research Topics in ML. http://www.speech.sri.com/people/anand/771/html/node30.html. Slides 30 through 38.
5) AI FAQ - Neural Nets. ftp://ftp.sas.com/pub/neural/FAQ.html.
6) Principles of training multi-layer neural network using backpropogation. http://galaxy.agh.edu.pl/~vlsi/AI/backp_t_en/backprop.html.
7) Artificial Neural Networks for Beginners. http://arxiv.org/ftp/cs/papers/0308/0308031.pdf.
8) Learning_and_Neural_Networks. http://en.wikiversity.org/wiki/Learning_and_Neural_Networks.

Flowchart


START
Initialize constants
Define number of layers and number of nodes in each layer as a global variable
Read Inputs and Outputs
Record sizes of Initializing Inputs and
Record sizes of Initializing Inputs and
Record sizes of Initializing Inputs
Normalize Inputs
Normalize Outputs
Preallocate cell arrays: Weights, Node Input (Input to Summing Junction) and Node Output (Output from Activation Function)
Randomize contents of Weights
Place the Initializing Inputs into the output of the first layer
Initialize lowest numbr of loops run to reach RMSE: Lowest_Runs=Max_runs+1
Is Tries >= Max_Tries?
Yes
No
A
B
C


A
Store number of loops: Runs=1
Yes
Is (rmse(runs)>Max_RMSE AND runs<Max_Runs) OR (runs=1)?
A
No
Feed Forward Initializing Inputs
Backpropagate
Store Difference: Difference(runs) = InitializationOutput - Activation{end}
Store Error Info: Initialization_Error(runs)= (.5*Init_Diff^2)
rmse(runs)=(sum(InitDiff^2))^.5
abse(runs)=abs(sum(Init_Diff))
Yes
Is (runs<lowest_runs AND rmse(end)<lowest_rmse) OR runs==1?
Increment runs
A
No
Re-randomize weights
Store weights and information on how well they performed: Best_Weights=Weights lowest_rmse = rmse lowest_runs = runs Initialization_Difference=Difference
B
Increment tries


Store number of Training Runs: Training_runs = 1
C
Gather the next training input and store in FF_Training_Input
Increment FF_runs
Record size of FF_Training_Input
Store FF_Training_Input in the output of the first layer
Feed Forward FF_Training_Input using Best_Weights
Store Difference: Training_Difference(FF_runs) = FF_Training_Output - Activation{end}
Store Actual Output: FF_Training_Output=Training_Output(FF_runs,:)
Store Output: Training_Net_Output = Activation Layers(end)
Store Error Info: Training_Error(FF_runs)=(.5*(Training_Difference^2)) Training_rmse(runs)=(sum(Training_Difference^2))^.5 abse(runs)=(abs(sum(Training_Difference)))
Backpropagate
Is FF_runs=Data_Points?
No
Yes
C
D


D
B
Gather the next on-line run input and store in FF On Line Input
Record size of FF_On_Line_Input
Store FF_On_Line_Input in the output of the first layer
Feed Forward FF_On_Line_Input using Best_Weights
Store Output: On_Line_Net_Output = Activation Layers(end)
Store Actual Output: On_Line_Output =FF_On_Line_Output(FF_runs,:)
Store Difference: On_Line_Difference(On_Line_FF_runs) = FF_On_Line_Output-On_Line_Net_Output
Store Error Info: On_Line_Error(FF_runs)=(.5*(On_Line_Difference^2)) On_Line_rmse(runs)=(sum(On_Line_Difference^2))^.5 abse(runs)=(abs(sum(On_Line_Difference)))
Is FF_On_Line_runs=On_Line_Points?
No
Increment FF_runs
B
Yes
Plot: (Net Output with Training Output) and (On_Line_Net_Output with On_Line Output) and (Training_Difference) and (On_Line_Difference)
END

Program Listing

1) Neural_Network.m: The script file for the artificial neural network.
2) Read_In_Data.m: The function file for reading from the tab delimited text file.
3) Normalize.m: The function file for scalar normalization of the input and output arrays.
4) Log_Normalize.m: The function file for normalizing logarithmically the input and output arrays.
5) Preallocate_Net_Matrices.m: Creates the Matrices used throughout the script file.
6) Feed_Forward.m: Feed forward a set of inputs through the neural network.
7) Delta_Rule.m: Uses the delta rule to perform back-propagation.
8) Traditional_NLin_Curve_Fitting.m: The script file to perform a non-linear curve fitting to produce a model for wind speed prediction by regression.
9) ModelFun.m: The function file which holds the model function on Traditional_NLin_Curve_Fitting.m will curve fit to match the inputs and outputs.

10) Randomized_XOR_Data.txt: This text file is a tab delimited text database. Contain 10,000 data points of XOR inputs and the related outputs for testing. There is a single row of data labels in row 1, and 10,000 columns in which:

    • Column 1 is the first input to the XOR
    • Column 2 is the second input to the XOR
    • Column 3 is the output of a XOR gate for the previous row of inputs to the XOR gate

11) Windspeed_Data.txt: This text file is a tab delimited text database. Contains 525,604 data points of wind speed in 11 columns, corresponding to the data measurements made at the National Renewable Energy Laboratory's "National Wind Technology Center" between 12:00am January 1, 2009 to 12:00am January 1, 2010. There are two rows of data labels in row 1 and 2, and 11 columns in which:

• Column 1 is the year
• Column 2 is the day of the year
• Column 3 is the time of measurement
• Column 4 is the average wind speed over the last 1 min interval at 80 meters
• Column 5 is the average wind direction over the last 1 min interval at 80 meters
• Column 6 is the standard deviation of wind speed over the last 1 min interval at 80 meters
• Column 7 is the peak wind speed over the last 1 min interval at 80 meters
• Column 8 is the average temperature over the last 1 min interval at 80 meters
• Column 9 is the average relative humidity over the last 1 min interval
• Column 10 is the average station pressure over the last 1 min interval
• Columns 11 is the horizontal hemispheric shortwave irradiance, from a pyranometer mounted several feet above ground level.

Function Code

```
% Script File: Neural Network.m
%
% Purpose: Train a neural network to use a set of atmospheric measurment
% conditions to predict atmospheric conditions in the future.  Use this
% network to predict wind speed 30 min into the future for a set of
% consecutive atmospheric condition measurements, then compare the
% prediction sets to the actual values from the consecutive set of
% measurements
%
% Record of Revisions:
%   Date        Programmer  Description of changes
%   ========    ==========  =====================
%   3/15/10     Justin Tracy    Original Code
%
% Define Input/Output Variables:
%
% No input/output variables
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    % Set Back Propogation options

    Learning_Rate = .1;
    Gradient_Momentum = .05;
    Max_RMSE = .025;
    Max_Runs = 1000;
    Max_Tries = 1;
    Batch_Size = 1440*10;
    Incremental_Run_Size = 1;
    Incremental_Runs = 1440*10;
    Testing_Size = 120;
    First_Input_Row = 100;
    First_Input_Column = 1;
    Num_Input_Columns = 10;
    First_Output_Column = 3;
    Num_Output_Columns = 1;
   Delayed_time_Inputs = 1;
    Beta = .95;
    Prediction_Time = 1;

    %File_Name = 'C:/WindSpeed/Randomized_XOR_Data.txt';
    File_Name = 'C:/WindSpeed/Wind_Speed_Data_2009_80m.txt';

    %Node_Structure = [Num_Input_Columns*((Back_Time_Inputs+1)); 3; 3;...
    %    Num_Output_Columns];
    Node_Structure = [Num_Input_Columns*((Back_Time_Inputs+1)); ...
        round(Num_Input_Columns*((Back_Time_Inputs+1)))/2; ...
        round(Num_Input_Columns*((Back_Time_Inputs+1)))/2;  Num_Output_Columns];

    [Unnorm_Batch_Input, Unnorm_Incremental_Input, Unnorm_Testing_Input,...
        Unnorm_Batch_Output, Unnorm_Incremental_Output, ...
        Unnorm_Testing_Output] = Read_In_Data(File_Name, ...
        First_Input_Row, First_Input_Column, Num_Input_Columns, ...
        First_Output_Column, Num_Output_Columns,Delayed_time_Inputs, ...
        Prediction_Time, Batch_Size, Incremental_Run_Size, Testing_Size);
 %    Unnorm_Testing_Input = [0,0;0,1;1,0;1,1];
%    Unnorm_Testing_Output = [0;1;1;0];
```

```
    % N_B_In is the number of rows of the input, representing the number
    % of inputs for training
    % M_B_In is the number of columns of the input, representing the number
    % of input values per input
    [N_B_In,M_B_In] = size(Unnorm_Batch_Input);
    % N_B_Out is the number of rows of the output, representing the number
    % of outputs for training
    % M_B_Out is the number of columns of the input, representing the
    % number of output values per output
    [N_B_Out,M_B_Out] = size(Unnorm_Batch_Output);
    % TN_In is the number of rows of the Training_Input, representing the
    % number of Training_Inputs for training
    % TM_In is the number of columns of the Training_Input, representing
    % the number of Training_Input values per Training_Input
    [N_I_In,M_I_In] = size(Unnorm_Incremental_Input);
    % TN_Out is the number of rows of the Training_Output, representing
    %the number of Training_Outputs for training
    % TM_Out is the number of columns of the Training_Input, representing
    %the number of Training_Output values per Training_Output
    [N_I_Out,M_I_Out] = size(Unnorm_Incremental_Output);
    % TN_In is the number of rows of the Training_Input, representing the
    %number of Training_Inputs for training
    % TM_In is the number of columns of the Training_Input, representing
    %the number of Training_Input values per Training_Input
    [N_T_In,M_T_In] = size(Unnorm_Testing_Input);
    % TN_Out is the number of rows of the Training_Output, representing
    %the number of Training_Outputs for training
    % TM_Out is the number of columns of the Training_Input, representing
    %the number of Training_Output values per Training_Output
    [N_T_Out,M_T_Out] = size(Unnorm_Testing_Output);

   Max_In=max([max(Unnorm_Batch_Input);max(Unnorm_Incremental_Input);...
        max(Unnorm_Testing_Input)]);
   Min_In=min([min(Unnorm_Batch_Input);min(Unnorm_Incremental_Input);...
        min(Unnorm_Testing_Input)]);
   Max_Out=max([max(Unnorm_Batch_Output);max(Unnorm_Incremental_Output);...
        max(Unnorm_Testing_Output)]);
   Min_Out=min([min(Unnorm_Batch_Output);min(Unnorm_Incremental_Output);...
       min(Unnorm_Testing_Output)]);

    [Batch_Input, Batch_Output, Incremental_Input, Incremental_Output, ...
        Testing_Input, Testing_Output] = Normalize(Unnorm_Batch_Input, ...
        Unnorm_Batch_Output, Unnorm_Incremental_Input, ...
        Unnorm_Incremental_Output, Unnorm_Testing_Input, ...
        Unnorm_Testing_Output, Max_In, Min_In, Max_Out, Min_Out, M_B_In,...
        M_B_Out, M_I_In, M_I_Out, M_T_In, M_T_Out, N_B_In, N_B_Out, ...
        N_I_In, N_I_Out, N_T_In, N_T_Out);
%   [Batch_Input, Batch_Output, Incremental_Input, Incremental_Output, ...
%       Testing_Input, Testing_Output] = Log_Normalize( ...
%       Unnorm_Batch_Input, Unnorm_Batch_Output, ...
%       Unnorm_Incremental_Input, Unnorm_Incremental_Output, ...
%       Unnorm_Testing_Input, Unnorm_Testing_Output, Max_In, Min_In, ...
%       Max_Out, Min_Out, M_B_In, M_B_Out, M_I_In, M_I_Out, M_T_In, ...
%       M_T_Out, N_B_In, N_B_Out, N_I_In, N_I_Out, N_T_In, N_T_Out);

    %%Create size variables%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    % Num_Layers holds the number of layers in the provided node structure
    % Size1 will always hold 1 and is a dummy variable
    [Num_Layers,Size1] = size(Node_Structure);
```

```
Num_Boundaries = Num_Layers - 1;
Num_Inner_Layers = Num_Layers - 2;

%Create a cell array to hold the weight matrices%

% One weight matrice is to be held for each space between two adjacent
% layers
Weights = cell(Num_Boundaries,1);
Best_Weights = cell(Num_Boundaries,1);

%Set up weight matrix

% For each layer but the last, a random number is distrubuted into
% each row but the last row, which recieves all zeros
for i=1:Num_Layers-2
Weights{i} = [1 - 2.*rand(Node_Structure(i+1),Node_Structure(i)+1); ...
    zeros(1,Node_Structure(i)+1)];
end

Weights{end} = 1-2.*rand(Node_Structure(end),Node_Structure(end-1)+1);

%Create a cell array to hold the activation matrice and Summing Node
%Junction matrice

[Activation, Neural_Net] = Preallocate_Net_Matrices(Num_Layers, ...
    Num_Boundaries, Batch_Input, N_B_In, Node_Structure)

%%Create a cell array to hold the change in weight matrices%%%%%%%%%%

% The cell arrays will hold the value in each node, which will be
% passed to the activation function
delta_W = cell(size(Weights));

%%Create a cell array to hold the neural network%%%%%%%%%%%%%%%%%%%%

% The cell arrays will hold the value in each node, which will be
% passed to the activation function
Total_delta_W = cell(size(Weights));

%%Set up initial delta W matrices%%%%%%%%%%%%%%%%%%%%%%%%%%%%

for i=1:Num_Layers-1
    delta_W{i} = zeros(size(Weights{i}));
    Total_delta_W{i} = zeros(size(Weights{i}));
end

% Perform Back Propogation

sae = 0;
lowest_rmse = 1000;
lowest_runs = Max_Runs+1;
tries = 0;

while (tries < Max_Tries)

    runs = 1;
    rmse = zeros(Max_Runs,1);

    while ((rmse(runs) > Max_RMSE) && (runs < Max_Runs)) || (runs == 1)
```

```
            % Feed Forward
            [Weights, Activation, Neural_Net] = Feed_Forward(Weights, ...
                Activation,Neural_Net, Num_Layers, Beta, N_B_In);

            % Delta Rule: (deltaW)ji = Learning rate *
            %(Difference from expected) * derivative(activation function) *
            % weighted sum of neuron inputs j * input i

            [Weights] = Delta_Rule(Activation, Batch_Output, Weights, ...
                delta_W, Total_delta_W, Num_Layers, Beta, Learning_Rate,...
                N_B_In, Gradient_Momentum);

        rmse(runs+1)=((sum(sum((Batch_Output-Activation{Num_Layers}).^2 ...
            ./ (M_B_Out * N_B_Out))))^.5);
        mae = max(max(abs(Batch_Output-Activation{Num_Layers})));
        runs = runs + 1
        end

        if ((runs < lowest_runs) || (runs == 1))
            if ((rmse(end) < lowest_rmse(end)) || (runs == 1))

                Best_Weights = Weights;
                lowest_rmse = rmse;
                lowest_runs = runs;
                diff = Batch_Output-Activation{Num_Layers};

            end
        else
        %%Set up weight matrix%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        % For each layer but the last, a random number is distrubuted into
        % each row but the last row, which recieves all zeros
            for i=1:Num_Layers-2
                Weights{i} = [1 - 2.*rand(Node_Structure(i+1), ...
                    Node_Structure(i)+1) ; zeros(1,Node_Structure(i)+1)];
            end
            Weights{end} = 1 - 2.*rand(Node_Structure(end), ...
                Node_Structure(end-1)+1);

        end

        tries = tries + 1;

    end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    Incremental_rmse = zeros(Incremental_Runs,N_I_In);
    Incremental_Average_P_Diff = zeros(Incremental_Runs,N_I_In);
    Incremental_Diff = zeros(Incremental_Runs,N_I_In);
    Incremental_Net_Output = zeros(Incremental_Runs,N_I_In);

    for Incremental_FF_runs = 1:Incremental_Runs



        [Activation, Neural_Net] = Preallocate_Net_Matrices(Num_Layers, ...
            Num_Boundaries, Incremental_Input, N_I_In, Node_Structure);


        %Set up initial delta W matrices
```

```
for i=1:Num_Layers-1
    delta_W{i} = zeros(size(Best_Weights{i}));
    Total_delta_W{i} = zeros(size(Best_Weights{i}));
end

% Feed Forward
[Best_Weights, Activation, Neural_Net] = Feed_Forward( ...
    Best_Weights,Activation,Neural_Net, Num_Layers, Beta, N_I_In);

% Delta Rule: (deltaW)ji = Learning rate*(Difference from expected)
%*derivative(activation function)*weighted sum of neuron
%Training_Inputs j*Training_Input i

[Weights] = Delta_Rule(Activation, Incremental_Output, ...
    Best_Weights, delta_W, Total_delta_W, Num_Layers, Beta, ...
    Learning_Rate, N_I_In, Gradient_Momentum);

Incremental_FF_runs

Incremental_Diff(Incremental_FF_runs,:) = ...
    abs(Activation{Num_Layers} - Incremental_Output);


Incremental_Training_rmse(Incremental_FF_runs,:) = ...
    ((sum(sum((Incremental_Output-Activation{Num_Layers}).^2 )))...
    ./ (M_I_Out * N_I_Out))^.5;
Incremental_Average_P_Diff(Incremental_FF_runs) = ...
    sum(sum(abs(Activation{Num_Layers} - Incremental_Output) ...
    ./ (Activation{Num_Layers} * (M_I_Out * N_I_Out))));
Incremental_Average_P_Diff(Incremental_FF_runs)
Incremental_Diff(Incremental_FF_runs,:);


Incremental_E = .5*(Incremental_Diff.^2);

% Gather another set of data for incremental run
if ((Incremental_Runs > 1) && ...
        (Incremental_FF_runs < Incremental_Runs))

    Randomized_Time_Period = round(Incremental_Runs*rand());

    Unnorm_Incremental_Input = ...
        dlmread(File_Name,'\t',[First_Input_Row+(Batch_Size)+ ...
        (Incremental_Run_Size*Randomized_Time_Period)+ ...
       Delayed_time_Inputs First_Input_Column First_Input_Row+ ...
        Batch_Size+(Incremental_Run_Size* ...
        (Randomized_Time_Period+1))+Back_Time_Inputs  ...
        First_Input_Column+Num_Input_Columns-1]);

    Unnorm_Incremental_Output = dlmread(File_Name,'\t',...
        [First_Input_Row+Batch_Size+(Incremental_Run_Size*...
        Randomized_Time_Period)+Back_Time_Inputs+...
        Prediction_Time First_Output_Column First_Input_Row+...
        Batch_Size+(Incremental_Run_Size*...
        (Randomized_Time_Period+1))+Back_Time_Inputs+...
        Prediction_Time First_Output_Column+Num_Output_Columns-1]);

    ifDelayed_time_Inputs > 0
            for i= 1:Back_Time_Inputs
```

```
                    Unnorm_Incremental_Input=[Unnorm_Incremental_Input ...
                        dlmread(File_Name,'\t',[First_Input_Row+...
                        Batch_Size+(Incremental_Run_Size*...
                        Randomized_Time_Period)+Back_Time_Inputs-i ...
                        First_Input_Column First_Input_Row+...
                        Batch_Size+(Incremental_Run_Size*...
                        (Randomized_Time_Period+1))+...
                       Delayed_time_Inputs-i First_Input_Column+...
                        Num_Input_Columns-1])];
                end
            end

            for i = 1:M_I_In
                for j = 1:N_I_In
                 Incremental_Input(j,i)=...
                     ((Unnorm_Incremental_Input(j,i) -...
                     (Min_In(i)+10^-16)) ./ ((Max_In(i)+10^-16)...
                     - (Min_In(i)-10^-16)));
                end
            end

            for i = 1:M_I_Out
                for j = 1:N_I_Out
                Incremental_Output(j,i)=((Unnorm_Incremental_Output(j,i)...
                 - (Min_Out(i)+10^-16)) ./ ((Max_Out(i)+10^-16) ...
                 - (Min_Out(i)-10^-16)));
                end
            end

        end

    end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    Testing_Diff = zeros(1,N_T_In);
    Testing_Net_Output = zeros(1,N_T_In);
    Testing_rmse = zeros(1,N_T_In);

    for Testing_FF_runs = 1:Testing_Size

        FF_Testing_Input = Testing_Input(Testing_FF_runs,:);
        FF_Testing_Output = Testing_Output(Testing_FF_runs,:);

        % TN_In is the number of rows of the Training_Input, representing
        % the number of Training_Inputs for training
        % TM_In is the number of columns of the Training_Input,
        % representing the number of Training_Input values per
        % Training_Input
        [FF_N_T_In,FF_M_T_In] = size(FF_Testing_Input);
        [FF_N_T_Out,FF_M_T_Out] = size(FF_Testing_Output);

        % Set up Activation and Summing Junction Input matrices
        [Activation, Neural_Net] = Preallocate_Net_Matrices(Num_Layers, ...
            Num_Boundaries, FF_Testing_Input, FF_N_T_In, Node_Structure);

        %Set up initial delta W matrices

        for i=1:Num_Layers-1
            delta_W{i} = zeros(size(Best_Weights{i}));
```

```
            Total_delta_W{i} = zeros(size(Best_Weights{i}));
        end

        %Feed Forward
        [Weights, Activation, Neural_Net] = Feed_Forward(Best_Weights,...
            Activation,Neural_Net, Num_Layers, Beta, FF_N_T_In);

        % Record the error
        Testing_Net_Output(Testing_FF_runs) =  Activation{Num_Layers};
        Testing_Diff(Testing_FF_runs) = abs(Testing_Net_Output(...
            Testing_FF_runs) - FF_Testing_Output);


        Testing_rmse(Testing_FF_runs) = ((sum(sum(((...
            Testing_Net_Output(Testing_FF_runs) - FF_Testing_Output).^2 ...
            ))./ (FF_M_T_Out * FF_N_T_Out)))^.5;

        Testing_Diff(Testing_FF_runs)

    end

    Testing_E = .5*(Testing_Diff.^2);

    % Unnormalize the testing output
    for i = 1:M_T_Out
       for j = 1:N_T_Out
       Unnorm_Testing_Net_Output(i,j)=((((((Testing_Net_Output(i,j)))+1)...
             /2).*(Max_Out(i) - Min_Out(i)))+  Min_Out(i)) ;
       end
    end

 % Find the percent difference for the unnormalized output
 Testing_P_Diff=abs(Unnorm_Testing_Output-Unnorm_Testing_Net_Output')./...
     abs(Unnorm_Testing_Output);


 % Plotting Functions
    figure(1)
        plot_start = 2;
        plot_stop = lowest_runs;
        plot(plot_start:plot_stop,lowest_rmse(plot_start:plot_stop))
        xlabel('Batch Training Cycles')
        ylabel('RMSE')
        title('Batch Training Results')

    figure(2)
        plot_start = 1;
        plot_stop = length(Testing_Net_Output);
    plot(plot_start:plot_stop,Testing_Net_Output(plot_start:plot_stop),...
        ':',plot_start:plot_stop,...
        Testing_Output(plot_start:plot_stop),'--');
        xlabel('Testing Iterations')
        legend('Testing Output', 'Expected Output')
        ylabel('Normalized Value')
        title('Testing Results')

    figure(3)
%       plot_start = 1;
%       plot_stop = length(Incremental_Net_Output);
%       subplot(2,1,1)
```

```
%       plot(plot_start:plot_stop,Incremental_rmse(plot_start:plot_stop));
%           xlabel('Incremental Training RMSE')
%           ylabel('RMSE')
%           title('Incremental Training Results')
       plot_start = 1;
       plot_stop = length(Testing_Net_Output);
%       subplot(2,1,2)
          plot(plot_start:plot_stop,Testing_rmse(plot_start:plot_stop));
          xlabel('Testing RMSE')
          ylabel('RMSE')
          title('Testing RMSE')

   figure(4)
%       plot_start = 1;
%       plot_stop = length(Incremental_Net_Output);
%       subplot(2,1,1)
%       plot(plot_start:plot_stop,Incremental_Diff(plot_start:plot_stop));
%           xlabel('Incremental Training Absolute Difference')
%           ylabel('Absolute Diff')
%           title('Incremental Training Results')
       plot_start = 1;
       plot_stop = length(Testing_Net_Output);
%       subplot(2,1,2)
          plot(plot_start:plot_stop,Testing_Diff(plot_start:plot_stop));
          xlabel('Testing Iterations')
          ylabel('Absolute Diff')
          title('Testing Results')

figure(5)
plot_start = 1;
plot_stop = length(Testing_Net_Output);
plot(plot_start:plot_stop,Testing_P_Diff(plot_start:plot_stop));
xlabel('Testing Iterations')
ylabel('% Diff')
title('Testing Results')

figure(6)
plot_start = 1;
plot_stop = length(Testing_Net_Output);
plot(plot_start:plot_stop, ...
    Unnorm_Testing_Net_Output(plot_start:plot_stop),...
    ':',plot_start:plot_stop,...
    Unnorm_Testing_Output(plot_start:plot_stop),'--');
xlabel('Testing Iterations')
legend('Testing Output', 'Expected Output')
ylabel('Unnormalized Value')
title('Testing Results')

 figure(7)
 plot_start = 1;
 plot_stop = Incremental_Runs;
 plot(plot_start:plot_stop,...
     Incremental_Average_P_Diff(plot_start:plot_stop));
 xlabel('Incremental Runs')
 ylabel('Average %Diff')
 title('Incremental Run Avg %Diff')
```

```
% Function File: Read_In_Data.m
%
% Purpose: Read in data from a tab delimited text file.
%
% Record of Revisions:
%   Date        Programmer  Description of changes
%   ========    ==========  =====================
%           Justin Tracy    Original Code
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [Unnorm_Batch_Input, Unnorm_Incremental_Input, ...
    Unnorm_Testing_Input, Unnorm_Batch_Output,...
    Unnorm_Incremental_Output, Unnorm_Testing_Output] =...
    Read_In_Data(File_Name, First_Input_Row, First_Input_Column,...
    Num_Input_Columns, First_Output_Column, Num_Output_Columns,...
   Delayed_time_Inputs, Prediction_Time, Batch_Size, Incremental_Run_Size,...
    Testing_Size)
    Unnorm_Batch_Input=dlmread(File_Name,'\t',...
        [First_Input_Row+Back_Time_Inputs First_Input_Column First_Input_Row+...
        Batch_Size+Back_Time_Inputs First_Input_Column+Num_Input_Columns-1]);
    Unnorm_Batch_Output=dlmread(File_Name,'\t',[First_Input_Row+...
       Delayed_time_Inputs+Prediction_Time First_Output_Column First_Input_Row+...
        Batch_Size+Back_Time_Inputs+Prediction_Time First_Output_Column+...
        Num_Output_Columns-1]);
    Unnorm_Incremental_Input=dlmread(File_Name,'\t',[First_Input_Row+...
        Batch_Size+Back_Time_Inputs First_Input_Column First_Input_Row+...
        Batch_Size+Incremental_Run_Size+Back_Time_Inputs First_Input_Column+...
        Num_Input_Columns-1]);
    Unnorm_Incremental_Output=dlmread(File_Name,'\t',[First_Input_Row+...
        Batch_Size+Back_Time_Inputs+...
        Prediction_Time First_Output_Column First_Input_Row+Batch_Size+...
        Incremental_Run_Size+Back_Time_Inputs+...
        Prediction_Time First_Output_Column+Num_Output_Columns-1]);
    Unnorm_Testing_Input=dlmread(File_Name,'\t',[First_Input_Row+...
        Batch_Size+Incremental_Run_Size+...
       Delayed_time_Inputs First_Input_Column First_Input_Row+Batch_Size+...
        Incremental_Run_Size+Testing_Size+Back_Time_Inputs First_Input_Column+...
        Num_Input_Columns-1]);
    Unnorm_Testing_Output=dlmread(File_Name,'\t',[First_Input_Row+...
        Batch_Size+Incremental_Run_Size+Back_Time_Inputs+...
        Prediction_Time First_Output_Column First_Input_Row+Batch_Size+...
        Incremental_Run_Size+Testing_Size+Back_Time_Inputs+...
        Prediction_Time First_Output_Column+Num_Output_Columns-1]);
    ifDelayed_time_Inputs > 0
        for i= 1:Back_Time_Inputs
            Unnorm_Batch_Input = [Unnorm_Batch_Input dlmread(File_Name,...
                '\t',[First_Input_Row+Back_Time_Inputs...
                -i First_Input_Column First_Input_Row+Batch_Size+...
               Delayed_time_Inputs-i First_Input_Column+Num_Input_Columns-1])]-.5;
            Unnorm_Incremental_Input = [Unnorm_Incremental_Input dlmread...
                (File_Name,'\t',[First_Input_Row+Batch_Size+Back_Time_Inputs...
                -i First_Input_Column First_Input_Row+Batch_Size+...
                Incremental_Run_Size+Back_Time_Inputs-i First_Input_Column+...
                Num_Input_Columns-1])]-.5;
            Unnorm_Testing_Input = [Unnorm_Testing_Input dlmread(...
                File_Name,'\t',[First_Input_Row+Batch_Size+...
                Incremental_Run_Size+Back_Time_Inputs...
                -i First_Input_Column First_Input_Row+Batch_Size+...
                Incremental_Run_Size+Testing_Size+Back_Time_Inputs...
                -i First_Input_Column+Num_Input_Columns-1])]-.5;
        end
    end
end % Read In Data Function
```

```
% Function File: Normalize.m
% Purpose: Normalize the input and output sets.
%
% Record of Revisions:
%   Date        Programmer  Description of changes
%   ========    ==========  ======================
%   3/15/10     Justin Tracy    Original Code
%
% Define Input/Output Variables:
%
%   Unnorm_Batch_Input,
%   Unnorm_Incremental_Input,
%   Unnorm_Testing_Input,
%   Unnorm_Batch_Input,
%   Unnorm_Incremental_Input,
%   Unnorm_Testing_Input: Unnormalized data sets
%   Max_In: Contains the max of each input.| N_B_In: # batch input sets.
%   Min_In: Contains the min of each input.| N_B_Out: # batch output sets.
%   Max_Out: Contains max of each output.  | N_I_In: # batch input sets.
%   Min_Out: Contains min of each output.  | N_I_Out:# incrementaloutputs.
%   M_B_In: # batch inputs.                | N_T_In:# testing outputsets.
%   M_B_Out: # batch inputs.               | N_T_Out:# testing outputsets.
%   M_I_In: # incremental inputs.
%   M_I_Out:# incremental inputs.
%   M_T_Out: # testing outputs.
%   M_T_In: # testing inputs.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [Batch_Input,Batch_Output,Incremental_Input,Incremental_Output,...
    Testing_Input, Testing_Output] = Normalize(Unnorm_Batch_Input,...
    Unnorm_Batch_Output, Unnorm_Incremental_Input, ...
    Unnorm_Incremental_Output, Unnorm_Testing_Input,...
    Unnorm_Testing_Output, Max_In, Min_In, Max_Out, Min_Out, ...
    M_B_In, M_B_Out, M_I_In, M_I_Out, M_T_In, M_T_Out, N_B_In, ...
    N_B_Out, N_I_In, N_I_Out, N_T_In, N_T_Out)
for i = 1:M_B_In
    for j = 1:N_B_In
        Batch_Input(j,i) = 2*(((Unnorm_Batch_Input(j,i) -...
            (Min_In(i)-(10^-24))) ./ ((Max_In(i)+(10^-24)) -...
            (Min_In(i)-(10^-24)))))-1;
    end
    for j = 1:N_I_In
        Incremental_Input(j,i) = 2*(((Unnorm_Incremental_Input(j,i) -...
            (Min_In(i)-10^-24)) ./ ((Max_In(i)+10^-24) -...
            (Min_In(i)-10^-24))))-1;
    end
    for j = 1:N_T_In
        Testing_Input(j,i) = 2*(((Unnorm_Testing_Input(j,i) -...
            (Min_In(i)-(10^-24))) ./ ((Max_In(i)+(10^-24)) -...
            (Min_In(i)-(10^-24)))))-1;
    end
end
for i = 1:M_B_Out
    for j = 1:N_B_Out
        Batch_Output(j,i) = 2*(((Unnorm_Batch_Output(j,i) - ...
            (Min_Out(i)-(10^-16))) ./ ((Max_Out(i)+(10^-16)) -...
            (Min_Out(i)-(10^-16)))))-1;
    end
    for j = 1:N_I_Out
        Incremental_Output(j,i) = 2*(((Unnorm_Incremental_Output(j,i)...
```

```
            - (Min_Out(i)-(10^-16))) ./ ((Max_Out(i)+(10^-16))...
            - (Min_Out(i)-(10^-16)))))-1;
    end
    for j = 1:N_T_Out
        Testing_Output(j,i) = 2*(((Unnorm_Testing_Output(j,i) -...
            (Min_Out(i)-(10^-16))) ./ ((Max_Out(i)+(10^-16))...
            - (Min_Out(i)-(10^-16)))))-1;
    end
end
end % Normalize Function
```

```
% Function File: Log_Normalize.m
% Purpose: Scale data logarithmically to more widely space lower data.
% Record of Revisions:
%   Date        Programmer  Description of changes
%   ========    ==========  ======================
%   3/15/10     Justin Tracy    Original Code
%
% Define Variables:
% Same as Normalize.m.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [Batch_Input, Batch_Output, Incremental_Input,...
    Incremental_Output, Testing_Input, Testing_Output] =...
    Normalize(Unnorm_Batch_Input, Unnorm_Batch_Output,...
    Unnorm_Incremental_Input, Unnorm_Incremental_Output,...
    Unnorm_Testing_Input, Unnorm_Testing_Output, Max_In, Min_In,...
    Max_Out, Min_Out, M_B_In, M_B_Out, M_I_In, M_I_Out, M_T_In, M_T_Out,...
    N_B_In, N_B_Out, N_I_In, N_I_Out, N_T_In, N_T_Out)
for i = 1:M_B_In
    for j = 1:N_B_In
       Batch_Input(j,i) = (1/.35)*log(1.8^-1*((Unnorm_Batch_Input(j,i) -...
            (Min_In(i)+(10^-24))) ./ ((Max_In(i)+(10^-24)) -...
            (Min_In(i)-(10^-24))))+1)+(.24);
    end
    for j = 1:N_I_In
        Incremental_Input(j,i) =...
            (1/.35)*log(1.8^-1*((Unnorm_Incremental_Input(j,i) -...
            (Min_In(i)+10^-24)) ./ ((Max_In(i)+10^-24) -...
            (Min_In(i)-10^-24)))+1)+(.24);
    end
    for j = 1:N_T_In
      Testing_Input(j,i) =...
          (1/.35)*log(1.8^-1*((Unnorm_Testing_Input(j,i) -...
          (Min_In(i)+(10^-24))) ./ ((Max_In(i)+(10^-24)) -...
          (Min_In(i)-(10^-24))))+1)+(.24);
    end
end
for i = 1:M_B_Out
    for j = 1:N_B_Out
        Batch_Output(j,i) = ...
            (1/.35)*log(1.8^-1*((Unnorm_Batch_Output(j,i) -...
            (Min_Out(i)+(10^-16))) ./ ((Max_Out(i)+(10^-16)) -...
            (Min_Out(i)-(10^-16))))+1)+(.24);
    end
    for j = 1:N_I_Out
        Incremental_Output(j,i) = ...
            (1/.35)*log(1.8^-1*((Unnorm_Incremental_Output(j,i) -...
            (Min_Out(i)+(10^-16))) ./ ((Max_Out(i)+(10^-16)) -...
            (Min_Out(i)-(10^-16))))+1)+(.24);
    end
    for j = 1:N_T_Out
        Testing_Output(j,i) = ...
            (1/.35)*log(1.8^-1*((Unnorm_Testing_Output(j,i) -...
            (Min_Out(i)+(10^-16))) ./ ((Max_Out(i)+(10^-16)) - ...
            (Min_Out(i)-(10^-16))))+1)+(.24);
    end
end
end % Log Normalize Function
```

```
% Function File: Preallocate_Net_Matrices.m
%
% Purpose: Preallocate the needed matrices.
%
% Record of Revisions:
%   Date        Programmer  Description of changes
%   ========    ==========  ======================
%   3/15/10     Justin Tracy    Original Code
%
% Define Input/Output Variables:
%
%   Activation: The activation function output matrix.
%   Num_Layers: The number of layers of the ANN.
%   Num_Boundaries:  The number of boundaries between two layers.
%   N_In: The number of input sets.
%   Neural Net:  The matrix holding the input to each node's summing
%   junction.
%   Node_Structure: The number of nodes in each layer.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [Activation, Neural_Net] = Preallocate_Net_Matrices(Num_Layers,...
    Num_Boundaries, Input, N_In, Node_Structure)

% One activation matrice is to be appended to the end of each layer
Activation = cell(Num_Layers,1);

%Create a cell array to hold the neural network%

% The cell arrays will hold the value in each node, which is the sum of
% the weights multiplied by the activation function
Neural_Net = cell(Num_Boundaries,1);

%Set up activation function matrix%

% Use another command to append
Activation{1} = [Input ones(N_In,1)];

for i=2:Num_Layers-1
    Activation{i} = ones(N_In,Node_Structure(i)+1);
end

Activation{end} = ones(N_In,Node_Structure(end));

%%Set up neural net matrix%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

for i=1:Num_Layers-2;
    Neural_Net{i} = ones(N_In,Node_Structure(i+1)+1);
end

Neural_Net{end} = ones(N_In,Node_Structure(end));

end % Preallocate Net Matrices function

% Function File: Feed Forward.m
%
% Purpose: Feed a set of inputs through a neural network.  At each layer,
% calculate the weighted sum into each summing junction and the activation
% function output at the output of each node.
%
```

```
% Record of Revisions:
%   Date        Programmer  Description of changes
%   ========    ==========  =====================
%   3/15/10     Justin Tracy    Original Code
%
% Define Input/Output Variables:
%
% Weights:  The matrix holding the current weight matrix.
% Activation:  The matrix holding the current activation function outputs.
% Neural_Net:  The matrix holding the input to each node's summing
% junction.
% Num_Layers:  The number of layers in the neural network.
% Beta: Determines the slope characteristic of the activation function.
% N_In:  The number of input/output sets per feed forward.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [Weights, Activation, Neural_Net] = Feed_Forward(Weights,...
    Activation,Neural_Net, Num_Layers, Beta, N_In)

% For each layer, feed forward the Output of the last activation function
% after multiplying it by the weight function, then calculate the
% activation function result for the next
    for i=1:Num_Layers-2
      Neural_Net{i} = Activation{i} * Weights{i}';
% Modified version of the standard sigmoid activation function: multiplying
% by 2 and subtracting by 1 produces an activation function that allows
% outputs from -1 to 1 (allows negative outputs) and gives more versatility
% to the Back Propogation function. The output of each nodes' weighted
% summation is fed into the activation function to produce an output for
%the layer to pass to the next layer
    Activation{i+1}=[(2./(1+exp(-2*Beta*Neural_Net{i}(:,1:end-1))))-1 ...
         ones(N_In,1)];
    end
% For the final layer, feed forward the Output from the second to last
% layer activation function and multiply it by the Weight vector
        Neural_Net{Num_Layers-1} = Activation{Num_Layers-1} * ...
            Weights{Num_Layers-1}';
% Feed the last weighted summation into the Activation function to produce
% a final value
Activation{Num_Layers}=(2./(1 + exp(-2*Beta*Neural_Net{Num_Layers-1})))-1;

end
```

```
 % Feed Forward function

% Function File: Delta Rule.m
%
% Purpose: Use the Delta Rule to perform back propagation for an ANN when
% called.  Returns the adjusted weight matrix.
%
% Record of Revisions:
%   Date        Programmer  Description of changes
%   ========    ==========  =====================
%   3/15/10     Justin Tracy    Original Code
%
% Define Input/Output Variables:
%
%   Weights: The weight matrix being adjusted by back-prop.
%   Activation: The activation function output matrix.
%   Output: The target output for this run
%   delta_W: The matrix for holding the delta value frm the delta rule.
%   Total_delta_W:  The matrix holding the value by which the weights will
%   be changed; includes gradient momentum term.
%   Num_Layers: The number of layers of the ANN.
%   Beta: The slope of the activation function.
%   Learning_Rate: Adjusts the rate at which error in the output is
%   reflected in a change in the weights.
%   N_In: The number of input sets.
%   Gradient_Momentum: The amount of previous weight changes to factor back
%   into the change in weights.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [Weights] = Delta_Rule(Activation, Output, Weights, delta_W, ...
    Total_delta_W, Num_Layers, Beta, Learning_Rate, N_In, Gradient_Momentum)

% Error calculation for the outer-most layer
    e = (Beta)*(1+Activation{Num_Layers}) .* (1-Activation{Num_Layers}) ...
        .* (Output - Activation{Num_Layers});

    for i=Num_Layers-1:-1:2
% Only works out when e is transposed?
        Total_delta_W{i} = Learning_Rate * (e') * Activation{i};

         e = (Beta)*(1 + Activation{i}) .* (1-Activation{i}) .* ...
            (e*Weights{i});

    end

        Total_delta_W{1} = Learning_Rate * e' * Activation{1};

    for i=1:Num_Layers-1
        delta_W{i} = (Total_delta_W{i} ./ N_In) + (Gradient_Momentum * ...
            delta_W{i});
        Weights{i} = Weights{i} + delta_W{i};
    end

end % Delta Rule function
```

```
% Script File: Traditional_NLin_Curve_Fitting.m
%
% Purpose: Predict wind speed through a traditional non-linear least
% squares curve fitting model for a comparison to the ANN results.
%
% Record of Revisions:
%   Date        Programmer  Description of changes
%   ========    ==========  =====================
%   3/15/10     Justin Tracy    Original Code
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


[N_In M_In] = size(Testing_Input)

beta = nlinfit(Batch_Input, Batch_Output, @ModelFun, [rand(),rand(),...
    rand(),rand(),rand(),rand(),rand(),rand(),rand(),rand(),rand(),...
    rand(),rand(),rand(),rand(),rand(),rand(),rand(),rand(),rand()]);

Traditional_Diff = zeros(1,N_In);
Traditional_rmse = zeros(1,N_In);
Traditional_Pdiff = zeros(1,N_In);

for i = 1:N_In
    Traditional_Output(i,:) = ModelFun(beta, Testing_Input(i,:));
end

    Traditional_Diff = (abs(Traditional_Output - Testing_Output));
    Traditional_Pdiff = (abs(Traditional_Output - ...
    Testing_Output)./(Traditional_Output * (M_In)));

    figure(2)
        plot_start = 1;
        plot_stop = length(Testing_Net_Output);
    plot(plot_start:plot_stop,Testing_Net_Output(plot_start:plot_stop),...
        ':',plot_start:plot_stop,...
        Traditional_Output(plot_start:plot_stop),'--');
        xlabel('Testing Iterations')
        legend('Testing Output', 'Expected Output')
        ylabel('Normalized Value')
        title('Testing Results')
```

```
% Function File: ModelFun.m
%
% Purpose: A model function structure for use wth the nlinfit function,
% which is used for curve ftting, to test a cure fitting least squares
% approach agains the ANN results.
%
% Record of Revisions:
%   Date        Programmer  Description of changes
%   ========    ==========  =====================
%           Justin Tracy    Original Code
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [ yhat ] = ModelFun(beta, x )

yhat = beta(1)*x(:,1)+beta(2)*x(:,2)+beta(3)*x(:,3)+beta(4)*x(:,4)+...
    beta(5)*x(:,5)+beta(6)*x(:,6)+beta(7)*x(:,7)+beta(8)*x(:,8)+...
    beta(9)*x(:,9)+beta(10)*x(:,10)+beta(11)*x(:,1)^2+...
    beta(12)*x(:,2)^2+beta(13)*x(:,3)^2+beta(14)*x(:,4)^2+...
    beta(15)*x(:,5)^2+beta(16)*x(:,6)^2+beta(17)*x(:,7)^2+...
    beta(18)*x(:,8)^2+beta(19)*x(:,9)^2+beta(20)*x(:,10)^2+...
    beta(11)*x(:,11)+beta(12)*x(:,12)+beta(13)*x(:,13)+beta(14)*x(:,14)+...
    beta(15)*x(:,15)+beta(16)*x(:,16)+beta(17)*x(:,17)+beta(18)*x(:,18)+...
    beta(19)*x(:,19)+beta(20)*x(:,20);
```

Schedule

Weeks 1-5: Develop Back Propagation and Feed forward functions.

Week 5: Develop error calculation function and test against XOR inputs.

Week 5-6: Correct the Back Propagation functions and retest until proof of learning is obtained.

Week 6: Develop Text File read functions and produce data sets of tab delimited text files.

Week 6-7: Test the Neural Network against XOR further, and begin testing of wind speed prediction.

Week 7: Develop plotting functions an continue testing wind speed predictions.

Week 8: Return to all the functions previously produced and adjust increase the modular nature of the functions used – use more variables rather than hardcoded numbers and provide more ability to adjust the Neural Network prior to a testing run

Week 8-10: Continue testing of wind speed data and try different testing schemes and network configurations to determine the best set of configurations.

Summary of Functional Requirements

This project implements an artificial neural network for the prediction of wind speed based on current and previous measurements. When the Neural Network is provided with a set of input and output data, it is capable of learning the relationship between the two data sets over a number of iterations, and is tested by providing a separate set of inputs to the network, and comparing to the actual output expected for those inputs to the value obtained from the neural network.

Primary Constraints

The primary constraint on this neural network is a trade-off between the accuracy to which the learning will occur and the complexity of the function being learned. While it can learn a fairly simple XOR function to a high degree of accuracy, testing for wind speed prediction showed that, in order to get accurate results, it is necessary to train the network over a large data set with a very large weight matrix, resulting in a relatively slow process.

Economic

As this project is implemented in software, and the Matlab compiler was available through the school, almost no cost was incurred in the design of this project. The true cost in developing a project such as this in an industrial setting would be the cost of labor. During the development of this program, the time required to develop exceeded what was expected, due largely to the presence of bugs that inhibited, rather than prevented, the learning function of the neural network. The time required for training was also greater than expected, as the size of the data set needed to train a

significantly large weight matrix with a difficult to learn function like wind speed prediction makes testing much more time consuming than was originally expected.

The only cost required to use this project would be the purchase of the Matlab computer program on which the program is run. The Matab program typically costs $2,000 for a copy of the basic program, which is all that is needed to run the artificial neural network.

Commerical Manufacture

As this is a computer program, no costs other than the labor costs for development are associated with its manufacture. Once the program is developed, the cost of transferring this code to other computers is negligible.

Environmental

This project, in conjunction with the development of smart grids for electrical distribution systems, has the potential to make the use of wind power a more attractive option for power generation. This would result in a beneficial impact in terms of CO_2 emissions, as an increase in the use of renewable resources will lower the CO_2 emission rate. There are some other environmental issues associated with an increased use of wind power, however, one of the most cited being the danger to bird species posed by wind farms. As environmental groups have documented,

Sustainability

This project stands to provide an improvement in the use of a sustainable resource, wind, in the generation of power, rather than other, non-sustainable sources

of energy such as coal. This would make an already highly sustainable power generation option more attractive, which could potentially increase the use.

Ethical

There are no ethical issues related to the use or misuse of this project.

Health and Safety

This project poses no health or safety risks.

Social and Political

This project stands to provide additional impetus to the push for the use of smart grids, particularly in wind generation areas. This could provide increased jobs in this area and provide an increased focus on the potentials of green energy.

Development

The artificial neural network was a new concept for me. During initial steps of writing the code for this neural network, I learned the basics of neural networks and, during the refinement and testing of the neural network, I refined my knowledge and understanding of neural networks, the concepts needed for their use and the function in which they can be used. I also learned a great deal about the effect of various parameters on a neural network's back-propagation and some of the different variations on the neural network.