ABSTRACT

Title of Document: A COMPARISON OF ARTIFICIAL NEURAL NETWORKS AND STATISTICAL REGRESSION WITH BIOLOGICAL RESOURCES APPLICATIONS

Jonathan Patrick Resop, Master of Science, 2006

Directed By: Dr. Hubert J. Montas, Ph.D.
Biological Resources Engineering

Artificial neural networks (ANNs) have been increasingly used as a model for streamflow forecasting, time series prediction, and other applications. The high interest in ANNs comes from their ability to approximate complex nonlinear functions. However, the "black-box" nature of ANN models makes it difficult for researchers to design network structure or to physically interpret the variables involved. Recent investigations in ANN research have found connections linking ANNs and statistics-based regression modeling. By comparing the two modeling structures, new insight can be gained on the functionality of ANNs.

This study investigates two primary relationships between ANN and statistical models: the potential equivalence between feed-forward neural networks (FNN) and multiple polynomial regression (MPR) models and the potential equivalence between recurrent neural networks (RNN) and auto-regressive moving average (ARMA) models.

Equivalence is determined through both formal and empirical methods. The real-world phenomenon of streamflow forecasting is used to verify the equivalences found.

Results indicate that both FNNs and RNNs can be designed to replicate many regression equations. It was also found that the optimal number of hidden nodes in an ANN is directly dependant on the order of the underlying physical equation being modeled. These simple relationships can be expanded to more complex models in future research.

A COMPARISON OF ARTIFICIAL NEURAL NETWORKS AND STATISTICAL REGRESSION WITH BIOLOGICAL RESOURCES APPLICATIONS

By

Jonathan Patrick Resop

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Master of Science
2006

Advisory Committee:

Dr. Hubert J. Montas, Chair
Dr. Adel Shirmohammadi
Dr. David R. Tilley

© Copyright by
Jonathan Patrick Resop
2006

Table of Contents

List of Tables

List of Figures

1 Introduction

1.1 Problem Importance

Biological and environmental systems have been historically very difficult for scientists and engineers to model effectively. This can be seen as a result of the large number of variables involved and the complex way in which they interact to produce such phenomena as surface runoff, nutrient transport, or population dynamics. While these systems are challenging to represent, they are nevertheless important for scientists and engineers to model for purposes such as prediction and simulation. Typically, researchers look to create models with two main goals in mind. First, the model should accurately map the input variables to the output variables as is observed in real life situations. And second, the model should be a fitting representation of the system's underlying physical characteristics.

Mathematical models have traditionally been developed from either physical principles or by statistical regression (Salas et al., 2000). Physical models consist of systems of ordinary or partial differential equations. These models try to represent the underlying physical relationship between the variables involved. The benefit of physical models is that they are based on a deep and thorough understanding of the system. However, the limitations of these models include the difficulty of setting up and solving complex differential equations analytically, as well as determining equation coefficients and initial and boundary conditions (Coppola et al., 2005). Usually, these equations must be solved using numerical methods, such as the finite element method.

Statistical models on the other hand are designed by finding the equation that best fits a set of historical or experimental data. These models are useful in that they are

generally simple and straightforward to solve. Statistical regression equations limit the user by requiring a large amount of sample data to estimate the parameters of the equation and to find the data trend. Also, there are difficulties that arise when manually determining the optimal structure of the statistical equation (Hill et al., 1994).

1.2 ANN Background and History

The limitations of regression equations and partial differential equations have led researchers to explore alternative models. One that has become popular over the last decade is the artificial neural network (ANN) (Govindaraju and Rao, 2000). Artificial neural networks are a type of model that was first conceptualized in 1943 by McCulloch and Pitts. ANNs were designed based on biological neurons, and how neurons interact with each other in the brain. McCulloch and Pitts (1943) started with the concept of a perceptron, which is a single artificial neuron (Figure 1.1). This artificial neuron, commonly referred to as a node, is analogous to a biological neuron (Figure 1.2). The node itself is similar to the cell body, and the connections made to other nodes represent the axon and synapses (Mehrotra et al., 2000). In the human brain, there are an estimated 10^{11} neurons and 10^{15} synapses all working in parallel (Veelenturf, 1995). The overall result is a complex system able to take incomplete and noisy information, make connections with pieces of existing memory and make intelligent decisions (Warner and Misra, 1996). This concept has given way to the idea massively parallel processing.

The thrust behind artificial neural network research has come from a desire to improve on the limitations of the modern serial computer and tap into this concept of parallel processing. While a computer has the ability to make complex mathematical

computations much faster than the human brain, biological neurons are able to perform tasks, such as image processing and speech recognition, with the speed and precision that serial computers can not match (Widrow, 1990). Fields such as artificial intelligence have attempted to adapt to the concept of parallel processing, but this connection to biological neurons is mostly lost in the field of modeling. Instead, modelers have focused on the ability of ANNs to estimate highly nonlinear functions.

The artificial perceptron receives inputs from other perceptrons in the system, multiplies each by a weight value, adds all of these products together, and then passes the result through a function typically called the activation function. The activation function, sometimes called the transfer function, is typically something simple and easy to differentiate like a linear function or a sigmoid function. A perceptron also has a bias added to inputs, which is commonly represented by a weight being multiplied by a constant of one. Network weights and biases are randomly initialized when the network is created and they are allowed to hold any real number value (Ellacott and Bose, 1996). The output of the activation function becomes the output of the perceptron, and this output is then sent out to other perceptrons in the network (Figure 1.1). A large group of perceptrons acting together in a single system creates an artificial neural network.

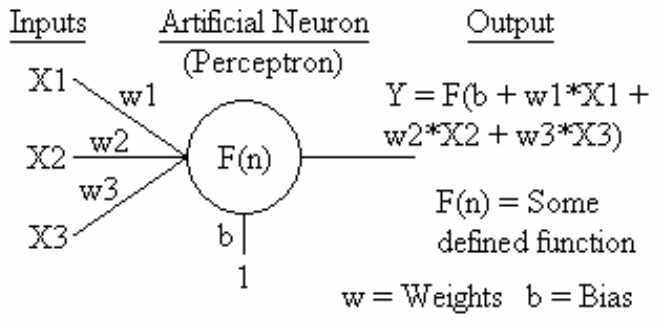


Figure 1.1: *A single artificial neuron, also known as a perceptron.*

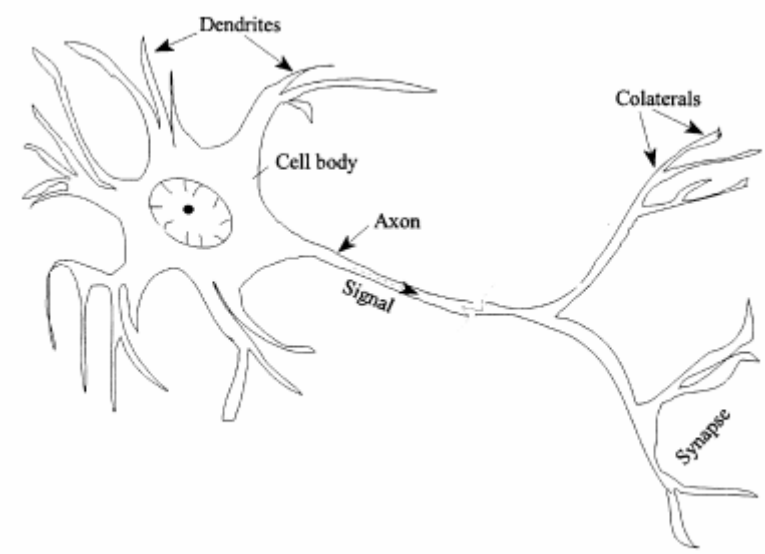


Figure 1.2: *A single biological neuron (Basheer and Hajmeer, 2000).*

A widely used ANN structure among modelers is the feed-forward neural network (FNN), also known as the multi-layer perceptron (MLP) (Cherkassky et al., 1993). Many other structures of ANNs have been developed such as fuzzy neural networks, evolving neural networks and radial basis function neural networks (Hayashi et al., 1992) (Yao, 1999) (Yang, 2006). A feed-forward network typically consists of three layers of neurons. An input layer for the predictor variables, a hidden layer, and an output layer for the criterion variables (Figure 1.3). The network sends information sequentially from left to right, from the input layer to the output layer. Each layer waits for the information from the previous layer before computing and sending its own value. In biological neural networks (Figure 1.4), this process works in parallel, with all neurons firing simultaneously. However, this approach is not yet realized in many ANN applications, such as function approximation and modeling (Couvreur and Couvreur, 1997) where the ANN process typically works serially, simply because contemporary computers are designed to work in serial mode.

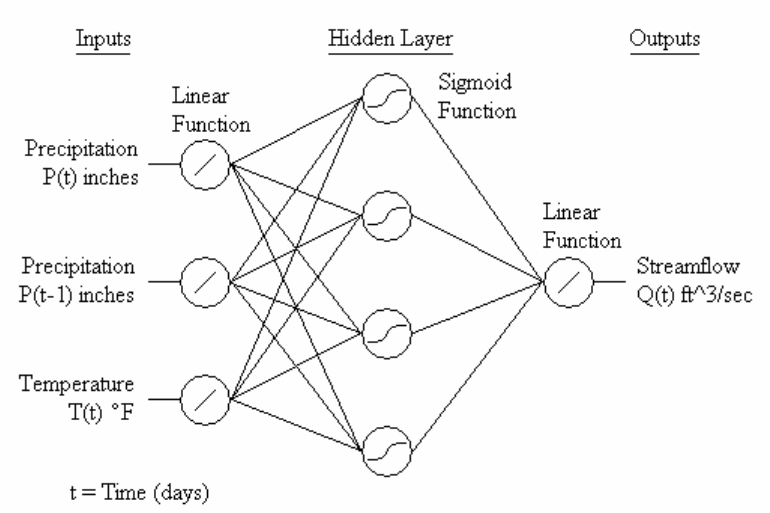


Figure 1.3: *A feed-forward neural network to predict streamflow based on precipitation and temperature.*

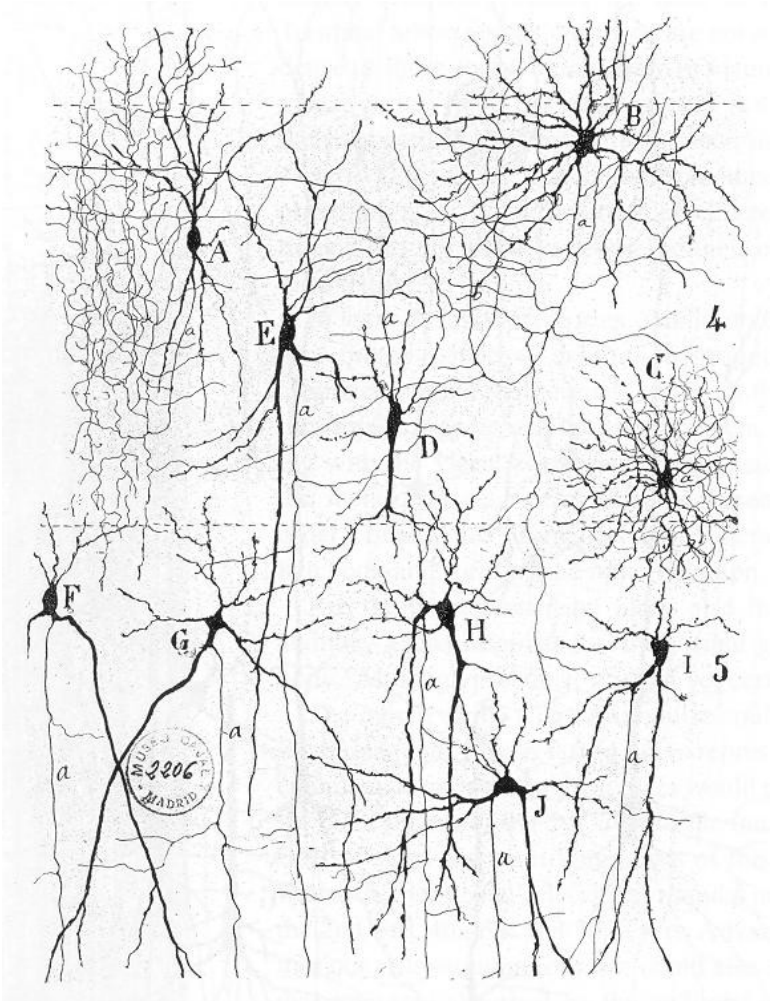


Figure 1.4: *A biological neural network with parallel processing (Cajal, 1999).*

ANNs did not start to become popular among modelers until the 1980's, with the development of the back-propagation algorithm, which is used to train the weights and biases of a network by using experimental data. The parameters of the network are updated with each pass of the training data with respect to the network's prediction error. Before the application of back-propagation to network training, it was difficult for modelers to update the weights and biases of the inner layers of the ANN. Back-propagation solves this problem by "back-propagating" the prediction error to the inner layers, allowing for complete network training. The main limitation of back-propagation is that it is generally very slow compared to alternative methods such as least-squares regression (Kruschke and Movellan, 1991).

Recurrent neural networks (RNN) differ from standard feed-forward networks in that they also allow backwards connections to exist between the nodes (Figure 1.5). This means that the output of the network can also be used as an input. The positives and negatives of using RNNs are similar to those of FNNs or any other artificial neural network. One positive aspect unique to the RNN is that its structure is more analogous to the original biological neural network concept of massive parallel processing. Another benefit is that recurrent networks can use previous output values as inputs to the model, allowing for multiple-day-ahead time series prediction. However, the recursive nature of the RNN makes it more complex to derive and more difficult to easily understand. Also, the recurrent connections in the network can make it more unstable and more sensitive to noise (Mandic and Chambers, 2001).

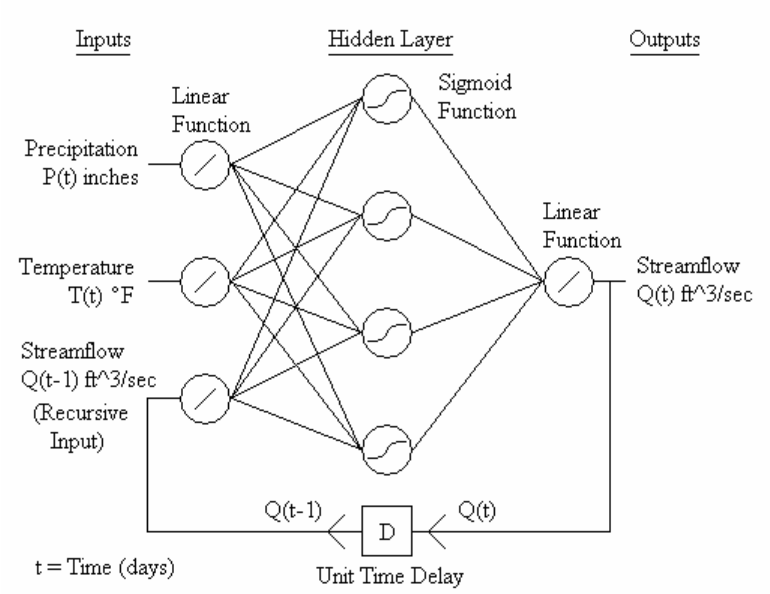


Figure 1.5: *A recurrent neural network used to predict streamflow.*

1.3 ANN Application to Modeling

Artificial neural networks have been researched and used for applications in many different fields. Many of these areas are using ANNs to solve problems previously thought to be impossible or very difficult with traditional methods. These include face recognition, prediction of time series events, function approximation, process optimization, and others (Cheng and Titterington, 1994). There are many reasons for the amount of interest being shown for ANNs. First, the structure of an artificial neural network is generally flexible and robust. Unlike regression, where a specific equation must be predetermined based on the data in the system to relate the input to output variables, the general structure of an ANN can be applied to practically any system (Zealand, 1999). Also, ANNs have been shown to outperform regression models when outliers exist in the data (Denton, 1995). Second, a feed-forward neural network with a sigmoid hidden layer is said to be a universal function approximator (White, 1992). As a

result, artificial neural networks are viewed as a powerful model limited only by the number of hidden nodes in the network. Third, ANNs are able to inherently model highly nonlinear systems such as those that govern the functioning of biological systems (Gevrey et al., 2003). Finally, the black box nature of neural networks is easy to implement for prediction applications in any field, making it appealing to at least some modelers.

Over the past decade, artificial neural network research has found its way into the areas of hydrology, ecology, medical and other biological fields. The American Society of Civil Engineers wrote a report to investigate the usage of ANNs in hydrologic applications, and found it being used for such purposes as rainfall-runoff modeling, streamflow forecasting, groundwater modeling, precipitation prediction, and water quality issues (ASCE Task Committee, 2000b). ANNs have also been used extensively in other areas, particularly for modeling biological and environmental systems. They have been successfully applied to systems such as ozone concentration prediction (Prybutok et al., 2000), classifying ECG signals (Barro et al., 1998), and many others.

For some modelers, the "black-box" nature of artificial neural networks has led way to skepticism (Tingsanchali and Gautam, 2000). While ANNs can predict the output variable of a system based on a set of input variables, the inner workings of the network are not easily understood. This is because the ANN training and application usually involves the composition of nonlinear functions that can be difficult to simplify and reduce to terms that can be understood physically. This also makes it difficult for example to analyze the network after it has been built and trained to determine the

relative importance of the different input variables in predicting the output (Minns and Hall, 1996).

Another problem is that ANN structure is poorly understood, making it difficult to design the optimal structure for a given system a priori. Currently, there are no formal rules for developing networks (Daqi and Genxing, 2003). There is no standard method for finding the ideal number of hidden nodes in a feed-forward network or for determining the best activation function. This is usually a time consuming, trial-and-error process and can lead to inelegant ANN designs (Shigidi and Garcia, 2003). Far too often, people who apply ANNs to problems will develop network structure based on a series of unproven empirical rules and trust the training of the network to result in "intelligent" predictions (Gonzalez, 2000). Another common approach is to simply select an arbitrarily large number of neurons for the model (Xiang et al., 2005). However, having too many nodes in a network can lead to modeling issues such as poor generalization and susceptibility to local minima in the error function instead of converging to the global minimum (Archer and Wang, 1993).

Overcoming the problems of network structure and size would be a large step towards a greater understanding of artificial neural networks, which would allow for more productive and effective use of ANN models in biological and environmental systems. There is a need for research to improve network design and analysis through formal methods that do not rely on trial-and-error (ASCE Task Committee, 2000a).

1.4 Determining the Number of Hidden Nodes

Towards the goal of making ANNs easier to implement and require less trial-and-error, there has been much research to develop rules or methods for determining the optimal number of hidden nodes to use in a network model. The number of hidden layers and number of hidden nodes in each are some of the more difficult parameters to determine for a neural network model, as there is no formal procedure for creating the most efficient network. The hidden nodes are important because they directly relate to model performance. In general, increasing the number of hidden nodes will increase the number of bumps and curves in the network's output function (Russell and Norvig, 2003). However, too many will lead to poor generalization and too few will result in an ineffective model.

According to Sarle (2004), the number of hidden nodes depends on a number of variables such as the size of the training data set, the number of input and output variables, the complexity of the underlying function, the amount of noise in the target variables, and the activation function used. A number of rules of thumb have been suggested (Kaul, 2004) (Prybutok et al., 2000) (Feng and Wang, 2003), but these rules tend to over-simplify the problem and can lead to poor network performance. Subramanian et al. (2003) use Kolmogorov's Theorem to initialize the number of hidden nodes. This theorem states that twice the number of input nodes plus one is a sufficient number of hidden nodes to model any continuous function. However, little information outside of this article was found on this theorem. At best, these rules of thumb are nothing more than a good first guess or estimate. Currently, it is ideal for the number of hidden nodes to be tailored to a specific system through intensive investigation and

thorough trial-and-error. However, there have been some attempts to develop methods to aid in the hidden node selection process.

Fletcher et al. (1998) developed a method for optimizing the number of hidden nodes in a traditional three layer network. First, they recognize that if a network has too few nodes, it will not accurately fit the target function. If a network has too many nodes, it will memorize the training data and will not be able to generalize to other data. Therefore, there must be a minimum when relating the number of hidden nodes to the network error. The method they propose starts with an initial number of hidden nodes and then increments the number of nodes higher or lower depending on a statistical analysis of the error. They conclude that this procedure will find the optimal number of hidden nodes faster than other methods, which includes simple trial-and-error. Although one downside of this method is that it is still an iterative process, and does not formally determine the number of hidden nodes.

Xiang et al. (2005) offered some guidelines and methods for determining the minimum number of hidden nodes to use for function approximation. They suggested that the ideal number of hidden nodes is close to the number of line segments that can represent the target equation. When designing FNNs, to first start with this number of hidden nodes and then increment or decrement the number slowly until the best performance is found. While this still leaves room for trial-and-error, it does give modelers a good place to start. They confirm their results by concluding that a third order polynomial regression equation requires a feed-forward network with three hidden nodes to fully approximate it. The feed-forward network they use has a sigmoid hidden function and a linear output function. They note that interesting innate ability of the

sigmoid activation function is to act as a smoothing function, where normal piecewise linear regression equations lack this ability.

Yitian and Gu (2003) developed an interesting solution to the problem of designing network structure and determining the number of hidden nodes for the application of modeling sediment transport in a river system. Instead of using trial-and-error methods for finding the optimal network structure, they designed a network to replicate the physical flow of water through the river network. A feed-forward network was used, since it flows much like a water system. The network parameters and transfer function were related to the conservation of mass in the system, giving them physical meaning. The results showed that the model could accurately predict sediment discharge from the system.

1.5 Potential Equivalence Between ANNs and Statistics

There has been increasing research over the last decade to address the issues of network design and analysis and shed some light on the "mystery" behind artificial neural network models. One area that has been gaining interest is the comparison between artificial neural networks and statistical regression modeling. Statistical regression is a method that has been used by statisticians and engineers for many years to fit an equation to a set of data. Regression equations have been largely studied by mathematicians and statisticians, as opposed to ANNs, which were developed separately by electrical engineers and computer scientists (Maier and Dandy, 1998).

Statistical regression models have been used by engineers for decades for purposes such as data analysis and prediction. The basic concept of regression is to fit a

specified equation to a series of independent and dependent variables. The parameters or coefficients are estimated by using empirical data and an error minimization procedure such as least-squares regression. Ultimately, the goal is to find an equation whose output has a high correlation with the target data series. The most commonly used regression approach is multiple linear regression (MLR), which uses a first order (linear) model and is relatively easy to implement (Ayyub, 2003).

When a more nonlinear model is necessary, a common approach is to use multiple polynomial regression (MPR), which allows for higher order terms (Kravtsov et al., 2005). The general form of MPR is shown in equation (1.1):

$$Y_k = c_0^k + \sum_{i=1}^{M} (c_i^k * X_i) + \sum_{i=1}^{M} \sum_{j=i}^{M} (c_{i,j}^k * X_i * X_j) + \ldots, \quad k = 1, \ldots, K \tag{1.1}$$

where X is an input variable, Y is an output variable, c is a coefficient, M is the number of inputs and K is the number of outputs. The main benefit for using regression models is that there are well established methods for analysis (such as ANOVA) and ranking the importance of independent variables. However, there are some downsides to using MPR. First, because of the multiple orders, regression terms will innately have a potentially high correlation between them (such as the correlation between X and X^2) (Bradley and Srivastava, 1979). This can lead to undesirable correlations between model parameters. Second, polynomial equations can lead to unintended phenomena such as polynomial swing. This can cause data at the edges of the training domain, and beyond, to be poorly predicted (Stigler, 1971).

On the surface, there are some obvious similarities between regression models and neural network models. Both are empirical in nature and rely extensively on experimental data to determine model parameters. In statistics, the structure of the

equation used to relate the output variable to the input variables is selected by the person developing the model (bivariate, multivariate, linear, polynomial, nonlinear, regressive). This is also the case with neural networks but the process is typically much more iterative, involving a trial-and-error phase, for example, to identify the approximate number of hidden nodes to use. Also, like regression equations, ANN performance is highly dependant on the training sample size and the amount of noise in the data (Markham and Rakes, 1998).

Similarities can also be seen in the methods used to derive the parameters for these models. For ANNs the training method is generally back-propagation and for statistical models it is least-squares regression. However, back-propagation can be described as a generalized form of the least-squares algorithm (Mehrotra et al., 2000). Both of these methods rely on the principle of finding the gradient of the error function. The error for a model is calculated by equation (1.2):

$$e = (\hat{y} - y) \tag{1.2}$$

where the measured output (y) is subtracted from the predicted output ($\hat{y}$). The error function, or objective function, F, calculates the sum of the errors squared, as shown in equation (1.3):

$$F = min \sum_{i=1}^{n} e^2 \tag{1.3}$$

The model parameters are then determined by taking the derivatives of F with respect to each parameter and identifying where these derivatives are null (Ayyub, 2003).

A major difference between ANNs and statistical regression however, is the ease with which model parameters and structure can be understood. The parameters generated by a regression equation can be easily analyzed to identify input-output relationships in

the model and compare the effectiveness of each input variable. However, when using artificial neural networks, researchers are inclined to rely on the network to "learn" the relationships between variables (Kisi, 2005). This information is then stored in convoluted network weights and biases, which are difficult to interpret.

An extension on normal statistical regression and feed-forward network models is the area of time series analysis. This subject deals with data series that have a temporal aspect, usually dealing with discrete time intervals although it can also be represented in continuous time (Brockwell and Davis, 2002). The variables used in these models are assumed to vary in some manner with respect to time. Time series models are mostly used for the purpose of prediction and forecasting. Two models that will be investigated in this study are recurrent neural networks and auto-regressive moving average (ARMA) models, defined by the equation:

$$Y_t = c_0 + \sum_p (c_p * Y_{t-p}) + \sum_q (c_q * E_{t-q}) \quad (1.4)$$

in which Y is the time series variable at time t, E is the error term, and c is a coefficient.

Auto-regressive moving average models can be viewed as a more general form of regression equations, allowing for previous values of the dependent variable to be used as input variables. A feature that is unique to the ARMA equation is the concept of an error term. This term is included as an input to the equation and is calculated as the prediction error from previous time values. These models are typically designed to be linear for simplicity, but this limits their effectiveness in nonlinear systems (Zhang, 2003). One of the appeals of recurrent neural networks is that they do not have this limitation, since sigmoid activation functions can easily be added to a network. Nonlinear ARMA equations, or NARMA, also exist, but they are not commonly used by modelers due to

their complexity. Another issue when modeling with ARMA is determining how far back, in terms of the number of previous outputs, the equation should go (Zhang, 2003). This is not an intuitive decision and no formal methods exist for finding the optimal ARMA equation. The structure of the equation will depend on each application and must usually be determined through trial-and-error methods.

1.6 Comparisons Between FNNs and MPR

Most of the research comparing artificial neural networks and statistical regression has been empirical in nature. The main focus has been to compare both models in terms of their accuracy and performance. Sargent (2001) performed a literature review on approximately thirty articles comparing ANNs to statistical regression for biomedical applications and found that ANNs outperformed statistics in only ten of the cases. The other articles either found that both models had equivalent performance or that regression models were better. In applications with large sample sizes, it was found that ANNs never performed better than regression. He speculated that the reason ANN did not dominate over statistical models is because both are heavily limited by the data being collected, in terms of the amount of data and the amount of error or noise in the data.

Sarle (1994) showed that that multi-layer perceptrons could be viewed as nonlinear regression models. He showed that simple linear multivariate regression can be represented with a single linear perceptron. Sarle also made simple comparisons between nonlinear regression and feed-forward networks (with nonlinear activation functions) and showed how neural networks can be designed to represent polynomial regression (using

different polynomial activation functions). While the sigmoid activation function was compared to nonlinear regression, it was not compared to MPR. Sarle suggested that one may potentially be able to design an artificial neural network to represent the structure of any regression model, and vice versa. However, his research was limited by providing only a basic, theoretical comparison between the models. No consideration was made to the number of hidden nodes in the network. Another limitation is that Sarle did not determine any formal equations to relate the parameters of ANNs to those of statistical regression equations.

Warner and Misra (1996) continued this train of thought by performing empirical tests with synthetic data to show how FNNs and statistical regression models perform similarly. Both models were fit to linear and nonlinear data and results showed that the neural network was able to produce a best-fit line comparable to linear and power regression. They demonstrated that a feed-forward neural network with a sigmoid activation function can act as a function approximator and that this is an advantage over traditional regression when the underlying function of the system is unknown. They further suggested that if the physical relationship is known between the input and output of a given system, then a specific regression equation would be more desirable.

Salas et al. (2000) used feed-forward neural networks to predict the daily average streamflow of the Little Patuxent River in Maryland. A trial-and-error process was used to determine the most accurate model structure. The network input sets consisted of combinations of variables such as precipitation, temperature, evaporation, snow water equivalent, and previous streamflow. The number of nodes in the hidden layer ranged from one to four hundred. The best-fit model was found to have precipitation,

precipitation for the previous day and temperature as inputs and ten hidden nodes. This network model was then compared to a statistical model, a simple conceptual rainfall-runoff (SCRR) model, and results showed that the ANN performed better than the SCRR model for predicting streamflow.

Tokar and Johnson (1999) also investigated the modeling of daily runoff of the Little Patuxent River watershed, measured in the form of streamflow. They compared the prediction abilities of feed-forward networks and regression equations. The models were tested using different combinations of input parameters. The feed-forward networks had sigmoid activation functions and the number of hidden nodes ranged from one to four hundred. Selection of the number of hidden nodes was made in part in relation to the data size of the training set (either one, two or three years worth of data). The structure of the regression equations were a combination of linear and power models. Using error values to measure performance, the best-fit ANN model was better then all of the regression models by a considerable degree (validation error for the best ANN was 0.42 and for the best regression model it was 0.64). Tokar and Johnson noted that the number of parameters in the best-fit neural network was fifty-one, while the number of parameters in the regression equations was never greater than nine. They believed that this difference in parameters allowed the ANN to reach a higher level of flexibility and complexity. Another reason for the advantage of the network models was the fact that they used a nonlinear activation function, while most of the regression equations were linear.

Kaul et al. (2004) compared the effectiveness of feed-forward networks and multiple linear regression for predicting corn and soybean yield. The input data was

scaled to a range of 0 to 1, and the input variables were chosen from a list of twenty parameters which included soil rating for plant growth and various rainfall values. Three layer FNNs were used, with a linear function in the input layer and sigmoid functions in the hidden and output layers. The number of hidden nodes was set to be initially equal to one-half of the total number of inputs and outputs, but no rationale was given to explain this choice. They increased or decreased the number of hidden nodes by one as a method of fine-tuning the model and improving performance. The statistical equation used to compare with the FNN was a basic linear regression equation. Trials compared the performance of both models to predict corn and soybean yield using the same input parameters and same training data sets. Their results showed that FNNs consistently produced higher r^2 values, indicating higher accuracy, compared to MLR. However, it should be noted that it seems unfair to compare ANNs with sigmoid activation functions, which are highly nonlinear, to a linear regression equation.

Schnabel and Maneta (2005) investigated the comparison between FNNs and multiple quadratic regression. Both models were applied to the issue of estimating sediment transport in rivers. Sensitivity analysis was used to determine the most effective input variables for each model. The feed-forward network was defined to have ten hidden nodes, but it was not explained why this number was used. The activation function for the nodes was not specified. For the regression equation, the linear and quadratic forms of each variable were used, but any cross-terms were ignored. The results found that both neural networks and statistical regression could effectively predict sediment transport, and that the performance of the two models was similar. However, the best-fit model determined by the regression equation used a different set of input

parameters than that used by the feed-forward network. The authors did not offer an explanation for this difference and did not elaborate on the structures of the FNNs and regression equations.

Cobourn et al. (2000) compared a nonlinear regression model with a feed-forward neural network for their ability to predict ozone concentrations. The regression model was a combination polynomial and power model. No specific mention was made of the activation function or the number of hidden nodes in the FNN. Both models produced practically equivalent predictions for daily ozone concentration in Louisville, Kentucky. However, it was found that both models used a different set of optimal input variables, with the regression model including more inputs than the FNN. In particular, the FNN did not include one parameter, air-mass trajectory, which was highly significant in the nonlinear regression equation.

Dedecker et al. (2005) investigated the effects of different river characteristics on the population of the aquatic species *Gammarus pulex L*. A feed-forward network was used with twenty four input nodes, ten hidden nodes, and one output node. All of the nodes used a logistic sigmoid activation function. The twenty four input nodes represented the different river characteristics that were being compared for the study. No rationale was given for the choice of ten hidden nodes. They then used four different network variable comparison methods as described by Gevrey et al. (2003) to determine the most significant river parameters. The "Weights" method was based on partitioning the network weights among the inputs, the "Profile" method involved varying one input while keeping the others fixed, the "PaD" method used partial derivatives, and the "Perturb" method studied small changes to the network. Dedecker et al. (2005) found

that all four methods were effective at determining variable importance and produced similar results.

1.7 Comparisons Between Recurrent Models

Connor and Martin (1994) discussed the relationships found between auto-regressive moving average models and recurrent neural networks. Their work essentially extended that of Sarle from simple regression models to auto-regressive models and from feed-forward neural networks to recurrent neural networks. Also, they proposed that by filtering outliers out of the training data, time series models such as ARMA or RNNs can be come more robust than by training with least-squares or back-propagation alone.

Chon (1997) showed that feed-forward neural networks with polynomial activation functions can be used to accurately predict the parameters of a single input and single output ARMA model. His research showed that by using a polynomial activation function in the hidden layer, a neural network can become mathematically equivalent to both linear and nonlinear ARMA models. Also, the network parameter training function of back-propagation was observed to be slightly more accurate at predicting ARMA parameters than the least square method, although least-squares took less computation time to complete. However, Chon (1997) did not use any recurrent neural networks, only feed-forward networks, limiting the conclusions of his results. Also, the commonly used sigmoid activation function and the concept of an error term were not included in the investigation, leaving room for further research.

Zhang (2003) developed a hybrid model for one-day-ahead time series prediction by combining linear ARMA equations and nonlinear FNNs. The neural network was

used to add nonlinearity to the model as well as provide the flexibility that ARMA can not provide. The hybrid model proposed first uses ARMA to model the linear part of the system, and then trains a FNN to the error found by the ARMA equation. The theory was that the ARMA equation would model the linear aspects, leaving the nonlinear elements for the FNN. Both of the models were then added together to produce the complete prediction. Results showed that the hybrid model outperformed both models independently. The author also suggested that by adding the ARMA equation first, the network did not over fit to data as easily. Perhaps both of these models can be combined to form one neural network, with one linear hidden layer and one nonlinear hidden layer.

Kumar et al. (2004) investigated the application of ANNs to streamflow forecasting using auto-regressive inputs. For this research, feed-forward networks and recurrent networks were used to predict monthly streamflow of the Karnataka River in India. Trial-and-error was used to design the network structures by varying the number of inputs from three to five and number of hidden nodes from five to twenty. Both networks were set up to have four layers, which included two hidden layers. The final FNN structure had fourteen nodes in both hidden layers, and the final RNN had ten nodes in both. The five previous monthly streamflow values were used for input. The recurrent network also passed three previous output values back to the input layer. No mention was made in the article of the activation functions used (it is likely that all nodes had a sigmoid function). Training was accomplished using fifty-thousand epochs, which seems rather high. Fifty years of data was used for training the models, while seven years were used for validation. Results found that the RNN performed more accurately than the

FNN. This is understandable, as the RNN is able to take advantage of its ability to factor in previous outputs, which acts as an error term similar to those used in ARMA models.

Amnala et al. (2000) used FNNs and RNNs along with polynomial regression and a form of power time series model to model watershed runoff. The network structure and hidden nodes were determined through trial-and-error. Results showed that feed-forward networks did not significantly outperform regression models, although the recurrent network was better than both. Also, it was noted that ANNs needed more parameters for producing the same prediction accuracy as a regression model.

1.8 Summary of the Literature Review

Many experiments have compared the results of using both artificial neural networks and statistical regression equations for modeling and predicting biological systems. Most papers concluded that the use of artificial neural networks produced data predictions more accurate or at least comparable to regression models. However, many of these papers also found inconclusive or conflicting results, such as Schnabel and Maneta (2005). There is enough evidence to support the connection between neural networks and statistical regression, but it is obvious that more research needs to be done to get a better understanding of how both models are related. Few articles explored the theoretical and mathematical connections between the structure of the two models, such as the articles written by Sarle (1994) and Chon (1997). However, the formal results of these papers were limited to simple comparisons. Also, with the exception of Chon (1997), existing research has not gone into much depth in the area of comparing the parameters of both models term by term to find formal equivalency equations.

While there has been some work done on comparing statistical and neural network models, more research needs to be done to directly link the two models to each other. This knowledge would be beneficial for many reasons. First, it could potentially allow one to take a neural network and convert it into an equivalent regression model for analysis. This is useful when modeling environmental systems whose variables have complex relationships. A neural network could be trained to produce accurate output predictions, and then an equivalent regression equation could be deduced from the network. This regression equation could then be analyzed through statistical methods such as ANOVA (Ayyub, 2003). Perhaps regression equation parameters could be extracted from a corresponding ANN and vise-versa.

Second, one can use the knowledge of a system gained through statistical analysis to develop the best network structure to fit a given system. This can be useful for developing a concrete method for determining the number of hidden nodes needed for a network, since currently the number of hidden nodes is commonly determined through trial-and-error.

Lastly, research in this area is important simply to bridge the gap of knowledge between statistics and neural networks. Both of these models have been developed independently of each other. Statistics have a history of being developed and studied by mathematicians for centuries, while artificial neural networks were originally developed only a few decades ago by computer scientists as a form of artificial intelligence to replicate the biological neurons in the brain. If connections can be made between the two models to show how similar (or different) they really are, the field of modeling will benefit as a whole.

2 Objectives

The overall goals of this research are to identify potential equivalences between artificial neural networks and statistical regression and to verify these equivalences when applied to modeling biological resources systems. Two main connections will be investigated: the relationship between feed-forward neural networks (FNN) and multiple polynomial regression (MPR) equations and the relationship between recurrent neural networks (RNN) and auto-regressive moving average (ARMA) equations.

The specific objectives are to:

1. Identify potential formal and empirical equivalences between FNNs and MPR using synthetic data.
2. Identify potential formal and empirical equivalences between RNNs and ARMA using synthetic data.
3. Apply the equivalences derived in 1 and 2 to the prediction of the bioenvironmental phenomenon of streamflow using real-world data.

Potential equivalences between artificial neural networks and statistical regression equations will be pursued using both formal and empirical methods. In all cases, the models will use the same set of input and output variables. The formal equivalences between FNN and MPR will be identified by Taylor series expansion of the ANN nonlinear sigmoid activation function followed by algebraic manipulation of the network output equation for an arbitrary number of hidden nodes. For two models to be formally equivalent, they should contain the same terms with the same order of nonlinearity. The parameters of one model should be able to be mathematically transformed to find the parameters of the other model, and vice-versa. The equivalence will be confirmed with

empirical data by comparing graphically and through fit and error statistics the output prediction functions and model parameters after being trained to the same series of data. Potential equivalences between RNN and ARMA will be sought in a similar fashion using a combination of formal techniques and empirical strategies. When comparing models, particular attention will be made to finding the minimum, or optimal, number of hidden nodes required for artificial neural networks to replicate a given statistical regression equation.

The scope of this research will be limited to a defined range of model structures. First, the artificial neural networks and statistical models will have one output. Second, polynomial regression orders will be investigated up to fifth order and ANNs will use up to forty-five hidden nodes. The artificial neural networks will use zero, one, or two hidden layers. Finally, only the training methods of least-squares and back-propagation will be used for estimating model parameters. The purpose of these limitations is to simplify the investigation and make it easier to find potential equivalences between the two models.

3 Methods and Materials

Three major tasks will be accomplished to meet the objectives of this study: 1) The potential equivalence of FNNs and MPR will be investigated; 2) The potential equivalence of RNNs and ARMA and; 3) The application of these equivalences to biological phenomena. The procedures and tools used to perform these tasks are described in the following three sections.

3.1 Equivalence of FNN and MPR

The first connection between ANNs and regression that will be investigated is the relationship between feed-forward networks and polynomial regression. Both of these models are non-recursive and use independent variables for predicting the output variable. This section describes the prediction equations, training methods, error metrics, and pre-scaling of input data involved in the application and comparison of FNN and MPR models. This is followed by individual subsections describing the specific formal and empirical perceptron-level and network-level comparisons of ANNs and regression models to be performed.

The basic structure of a feed-forward neural network is shown in Figure 3.1. A FNN will have either three or four layers, with the number of nodes in the input layer equal to the number of input variables and the output layer representing the number of output variables. The input and output layers typically have linear activation functions. The hidden layer, or layers, can have any number of nodes and any type of activation function. Current neural network technology leaves it up to the researcher to determine

empirically the optimal number of hidden nodes and the optimal activation function. The general equation for a feed-forward network can be written as:

$$Y_k = F_2(\sum_j(w_j * F_1(\sum_i (w_i * X_i) + b_j)) + b_k) \quad (3.1)$$

Where X_i is the ith input variable, Y_k is the kth output variable, the w's are the network weights, the b's are the network biases, F_1 is the activation function of the hidden layer and F_2 is the activation function of the output layer (Ripley, 1996).

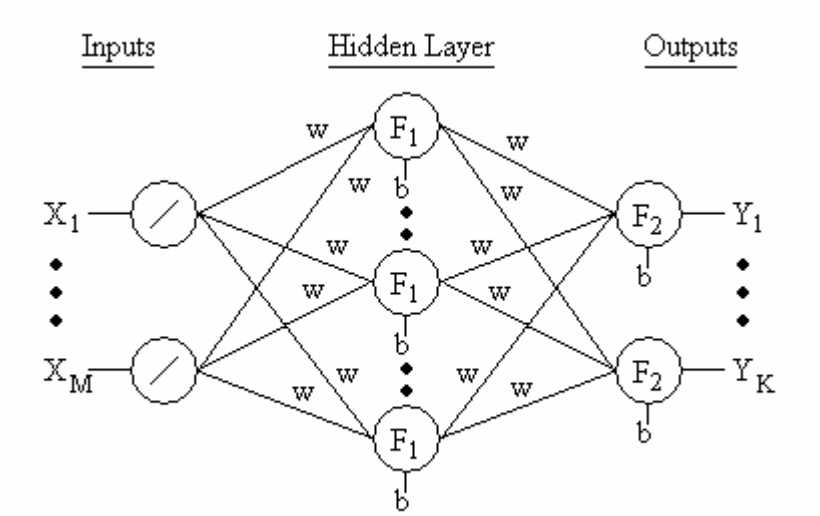


Figure 3.1: *General structure of a three layer feed-forward network.*

The total number of adjustable parameters (weights and biases) in a three-layer FNN is equal to:

$$P_n = 1+(1+M+K)*H \quad (3.2)$$

where M is the number of inputs, K is the number of outputs, and H is the number of hidden nodes (Salas et al., 2000). This is assuming that all of the connections between the input, hidden and output layers have a weight and all of the nodes in the hidden and output layers have a bias.

In this study, a linear activation function will be used for nodes in the output layer (F_2). For the hidden layer, the activation function F_1 will be either one of three functions:

Linear: $f(x) = x$ (3.3)

Polynomial: $f(x) = x^n$ *(n = order of polynomial)* (3.4)

Sigmoid: $f(x) = \tanh(x)$ (3.5)

The multiple polynomial regression equation is a more general form of the commonly used multiple linear regression equation. However, the equation also allows for higher order terms as well as cross-multiplied terms. The general form of a MPR equation can be written as:

$$Y_k = c_0^k + \sum_{i=1}^{M}(c_i^k * X_i) + \sum_{i=1}^{M}\sum_{j=i}^{M}(c_{i,j}^k * X_i * X_j) + \ldots, \quad k = 1, \ldots, K \tag{3.6}$$

Where X_i is the *i*th input variable, Y_k is the *k*th output variable, and the *c*'s are the regression coefficients for *M* input variables and *K* output variables (Ayyub, 2003). The order *N* of a polynomial equation is equal to the highest order out of all of the terms. An interesting difference to note between MPR and FNNs is how they function in multiple output systems. For MPR, a separate regression equation is estimated for each output variable. However, for FNNs, only one network is needed to model multiple outputs.

For a single output variable, the number of parameters in a MPR equation depends on the number of input variables and the maximum order of the equation. This is assuming that all of the terms created by polynomial expansion are included in the regression equation. For example, for a second order polynomial that uses X_1 and X_2 as input parameters, the expansion of $(1 + X_1 + X_2)^2$ produces the six terms X_1, X_2, X_1*X_2, X_1^2, X_2^2, and a constant. Pascal's triangle can be applied to find the number of parameters, seen in Table 3.1.

Table 3.1: *The number of terms in an MPR of Nth order with M input variables.*

Equation Order, *N*	Number of Input Variables, *M*					
	0	1	2	3	4	5
0	1	1	1	1	1	1
1	1	2	3	4	5	6
2	1	3	6	10	15	21
3	1	4	10	20	35	56
4	1	5	15	35	70	126
5	1	6	21	56	126	252

The number of parameters in a MPR equation can also be calculated using equation (3.7) for any number of inputs *M* and any order *N*.

$$P_r = \prod_{i=1}^{M} \frac{(N+i)}{i} \tag{3.7}$$

The software used to construct and test both the neural network and regression models will be MATLAB Version 7.0 (MATLAB, 2004). The neural networks will be constructed using the Neural Network Toolbox Version 4.0 (Demuth and Beale, 2004), which has the ability to model and train many different network structures. The FNN will be trained using the back-propagation algorithm, which will estimate the network weights for a given data set. For regression, the least-squares method will be used to estimate the equation coefficients. Sample MATLAB code for all tests is presented in Appendix A.

The back-propagation algorithm iterates for a number of epochs set by the modeler. For each epoch, the error is determined using equation (1.2). The weights and biases of the output layer are then updated according to equation (3.8) where *k* is the interval, w_i is the *i*th weight of the network, and *F* is the error function defined by equation (1.3) (Hagan et al., 2002).

$$w_i(k+1) = w_i(k) - \alpha \frac{\partial F}{\partial w_i} \tag{3.8}$$

Updating the weights and biases of the other layers is done in a similar manner, but is slightly different because the error for each layer output cannot be directly calculated. Instead, the error from the output layer must be back-propagated to the other layers by starting from the end and working backwards to the input layer. ANN training ends either when the network error is below a set level or when the maximum number of epochs has been reached. In MATLAB, back-propagation will be run with the *train* function. Five hundred epochs will be used for training feed-forward networks.

Unlike back-propagation, the least-squares algorithm does not require multiple iterations. Instead, the error function (Equation 1.3) is derived with respect to each of the regression parameters and set equal to zero, as shown in equation (3.9) (Ayyub, 2003). The system of equations is then solved to find a solution for all of the parameters.

$$\frac{\partial F}{\partial c_i} = 0 \tag{3.9}$$

The least-squares method will be run in MATLAB using *mldivide* or the left-division (backslash) operator as shown in equation (3.10). The vector c represents the estimated regression coefficients, X is the matrix of input variables raised to the powers found in the regression equation, and Y is the vector of target output values.

$$c = X \setminus Y \tag{3.10}$$

The parameters of each model will be compared analytically to find a formal equivalence between the two models. Both ANNs and statistical regression will be analyzed to compare their mathematical structures. Specifically, this means comparing the components of the neural network, such as number of hidden nodes and the activation function, with the components of a polynomial regression model, such as the number of terms and the highest order of the predictor variables. Then, the artificial neural network

equation will be transformed using Taylor series expansion of the sigmoid functions. Similar terms will be collected and the simplified equation will be compared to the regression equation. Ultimately, the goal is to find an analytical relationship between the weights and biases of an artificial neural network and the coefficients of a multiple polynomial regression model. Formal equations will be developed to define the regression coefficients in terms of the network parameters. Also, the minimum number of parameters required by each model to produce the desired equation should be comparable. Once a mathematical relationship is found, one can then reduce a neural network to an equivalent regression model through analytical methods. These formal equations will then be confirmed using empirical data.

The formal results will be confirmed empirically using synthetic data. Data will be divided into two sets, training data and validation data. The training data will be used to train the neural networks and to estimate the parameters for the regression equations. During the training stage, the parameters are changed to minimize the prediction error. Afterwards, the models will be tested using the validation data and the model parameters will be kept static. The purpose of keeping the validation data separate from the training data is to test the model's ability to generalize and to make sure it is not memorizing the training set. Both the training set and the validation set will be taken from the same population.

When comparing these models empirically, different criteria will be used to assess their relationship. First, both the neural network and the regression model will be optimized to predict the output data of a synthetic system. The accuracy of both models will be compared to determine if they are able to achieve the same level of predictability.

Accuracy will be determined by computing the standard error ratio, *s(e)/s(y)*, which is the standard deviation of the error over the standard deviation of the target output values (Equation 3.11).

$$\frac{s(e)}{s(y)} = \sqrt{\frac{\sum(\hat{y} - y)^2}{n - p}} \Big/ \sqrt{\frac{\sum(y - \bar{y})^2}{n - 1}} \tag{3.11}$$

Where $\hat{y}$ is the predicted output, y is the measured output, $\bar{y}$ is the mean output, n is the sample size, and p is the number of parameters in the model. For regression, the number of parameters is equal to the number of regression terms. For ANNs, the number of parameters is equal to the sum of weights plus biases of all nodes. Standard error ratio values close to zero are considered to be good models, while values close to one are considered to be poor models (Salas et al., 2000). Models with a standard error ratio greater than one are particularly unreliable, since this signifies that the standard deviation for the predicted values is worse than simply using the mean value of the data as a predictor of the entire data set.

The FNNs and MPR equations will both be trained to the same series of synthetic data. The output function of both will be plotted and the prediction error from both will be compared to relate their performance. In particular, function characteristics such as shape, bias, and ability to deal with extreme values will be investigated. Then, the parameters of the ANN will be transformed using the formal equations derived previously into the equivalent regression parameters and will be compared to the real regression parameters determined through least-squares.

When developing models, data is sometimes preprocessed before it is inputted into the model. During the preprocessing stage, the data is transformed to modify its

range or probability density function. This is usually done if the histogram of the data reveals that the data is skewed to one side of the expected range. In ANN applications, input and output data can be transformed to fit the range of the activation function being used in the network. For this study, two different forms of preprocessing, linear scaled and log normal scaled, will be used along with normal un-scaled data to train and validate the models. The different methods of preprocessing will be compared to determine if any particular method gives the model an advantage in accuracy.

Linear scaling is used to transform a data series from its original range as described by equation (3.12). The newly scaled data, represented by x', will retain the distribution of the original series, but will have a new range. The maximum and minimum values of the new range are set by the modeler as R_{max} and R_{min} respectively. This is a common preprocessing technique for ANN applications, because activation functions such as the sigmoid have a defined range of effectiveness. In the case of the sigmoid, the range is somewhere between -1 and +1 (Menon et al., 1995). After the model is trained to the linear scaled data, the output must be rescaled using the same parameters.

$$x' = (R_{\max} - R_{\min}) * \frac{x - \min(x)}{\max(x) - \min(x)} - \frac{(R_{\max} - R_{\min})}{2} \tag{3.12}$$

Log normal scaling will change the distribution of the data series to a normal distribution. It is calculated by subtracting the natural log of the data by the mean of the natural log of the data, and then dividing by the standard deviation of the natural log of the data (Equation 3.13). This process will squash the data, reducing the extremity of outlying data values. The log normal preprocessing method will be used in conjunction

with the linear scaling method (Equation 3.12) so that the data is within the range of sigmoid activation function.

$$x' = \frac{\log_e(x) - mean(\log_e(x))}{std(\log_e(x))} \tag{3.13}$$

3.1.1 Perceptron-level Analysis

Two perceptron-level cases will be investigated: that with a linear activation function and that with a sigmoidal activation function.

3.1.1.1 Linear Perceptron and Linear Regression

A perceptron is defined as an ANN with only one layer and one node. A comparison between a linear perceptron and multiple linear regression equations has already been shown by Sarle (1994). A formal equivalence between the two models will be identified in this study by finding the mathematical relationship between the weights and biases of the perceptron and linear regression coefficients. This will not require Taylor series expansion since the activation function is linear. The equivalence will be confirmed empirically by fitting both models to a linear equation that includes noise and comparing the regression coefficients found by both models as well as the goodness-of-fit of their output functions. For simplicity, a single input variable will be tested.

3.1.1.2 Sigmoid Perceptron and Polynomial Regression

The single artificial perceptron with sigmoidal activation function will be compared to polynomial regression both formally and empirically. Third order Taylor series expansion will be performed on the sigmoid activation function and substituted in

the perceptron equation. The resulting equation will be compared, term by term, to third order polynomial regression, with one input variable, to identify equations that transform perceptron weights and biases into polynomial coefficients, and vice-versa. The equations relating the perceptron parameters to the regression coefficients will be confirmed empirically by fitting the models to a cubic data series with noise and comparing the resulting polynomial coefficients produced by the two models. Both raw data as well as data scaled to a smaller range using equation (3.12) will be tested to determine the effect of input data scaling on perceptron performance.

3.1.2 Network-level Analysis - Polynomial Activation Function

Networks of perceptrons with polynomial activation function will be compared to MPR in two steps. First, a specific example where the target output function is a noisy cubic polynomial will be investigated. The results of this investigation will then be generalized to target multinomials of orders one to five with one to five input variables.

3.1.2.1 Specific Example - Third Order with One Variable

After investigating the abilities and limitations of single perceptrons, a larger three-layer neural network will be examined. FNNs with a linear input layer, polynomial (cubic) hidden layer, and linear output layer will be compared analytically to multiple polynomial regression equations. A one input variable, third order (cubic) polynomial equation will be used as the target MPR. This regression model will be compared to three forms of FNN: a one hidden node network, a two hidden node network, and a modified two hidden node network that has fewer parameters. In all cases, formal

equations will be developed from Taylor series expansion, substitution, and term by term comparison to define the regression coefficients in terms of the weights and biases of the feed-forward network. Issues such as input range, input scaling, prediction accuracy, equivalence of estimated polynomial coefficients and the effect of the training algorithm (back-propagation versus least-squares) will be explored for each case. The minimum number of hidden nodes required to replicate the cubic regression equation will be determined based on FNN performance in all of the cases.

3.1.2.2 Generalization to MPR

The comparison of FNNs with polynomial activation function and MPR will proceed for an increasing number of input variables and polynomial regression orders. Empirical methods will be used to relate the number of hidden nodes in a network to the order and number of input variables in multiple polynomial regression. Specifically, this means to determine the minimum number of hidden nodes required to replicate a given MPR equation.

Multiple polynomial regression equations will be synthetically created with a number of input variables ranging from one to five. The order of the polynomial regression will also range from one to five. Randomly generated normalized coefficients will be multiplied to each term of the MPR equation. The input variables will draw from random normalized data sets, each variable having a sample size of five hundred. The target values for the model will be found by running the input data values through the regression equation.

A series of three layer feed-forward networks will then be trained to model the input-target set relationship. The input and output layers of the networks will use a linear activation function, and the hidden layer will use a polynomial activation function. The polynomial activation function order will be equal to the polynomial regression order being modeled. For example, for a third order target multinomial, the activation function in the hidden layer will be $f(n) = n^3$. The number of hidden nodes in the network will be increased step by step from one to forty-five. For each network, the error value $s(e)/s(y)$ will be determined using equation (3.11). When the error reaches zero, or becomes less than $1 * 10^{-10}$, then that network will be concluded to be able to replicate the target multiple polynomial function. For each combination of number of variables and polynomial order, the minimum number of hidden nodes required to reproduce the target polynomial will be found.

3.1.3 Network-level Analysis - Sigmoid Activation Function

Networks of perceptrons with sigmoidal activation function will be compared to MPR in a similar, two-step, method. First, the FNNs will be investigated where the target output function is a noisy cubic polynomial. Second, the results will be generalized to target multinomials of orders one to five with one to five input variables.

3.1.3.1 Specific Example - Third Order with One Variable

The network-level analysis using a polynomial activation function will be complemented by researching feed-forward neural networks that use a sigmoid activation function in the hidden layer. The procedure for this investigation will be similar to that

for the polynomial activation function. Formal comparison between the FNN and a third order polynomial regression equation will be carried out using Taylor expansion on the sigmoid function. Like before, three cases will be tested: a one hidden node network, a two hidden node network, and a modified two hidden node network that has fewer parameters. The same testing considerations will be used, including the model prediction accuracy and ability to estimate the polynomial coefficients. The results from using the sigmoid activation function will be compared to the polynomial activation function network tested in the previous section.

3.1.3.2 Generalization to MPR

The results from the example sigmoid network will be generalized to other FNNs and MPR equations. The methods will be similar to those used for the polynomial activation function, except now a sigmoid activation function will be used in the hidden layer. Again the number of hidden nodes will range from one to forty-five. Five trials will be run for each network structure, as a means to offset the random initialization of the network parameters. The networks will use the same training and validation sets, both randomly generated with five hundred normal values. The training and validation *s(e)/s(y)* errors will be calculated for each network. The relationship between network prediction error and the number of hidden nodes will be observed for each combination of number of input variables and polynomial regression order. The results will be used to confirm the findings from the previous investigations.

3.2 Equivalence of RNN and ARMA

The second phase of testing will involve recurrent neural networks and auto-regressive moving average functions. These models are similar in structure to FNN and MPR, but they are inherently recurrent, allowing for previous values of the output to influence current predictions. This section describes the prediction equations and training methods involved in the application and comparison of RNN and ARMA models. This is followed by individual subsections describing the specific formal and empirical perceptron-level and network-level comparisons of ANNs and regression models to be performed.

A recurrent neural network is structured similarly to a feed-forward neural network except that it also includes a temporal component. All of the input and output variables are represented with respect to a discrete time step, *t*. To accomplish this function, the RNN allows for two additional components: backwards, or feedback, connections and delays (Connor and Martin, 1994). Backwards network connections link nodes in the output layer to the nodes in the input layer (Figure 3.2). This allows for recursive functionality. A delay is used for time series models to hold a particular value and output it at the next time step. The value for the output from the delay is initially zero for the first time step.

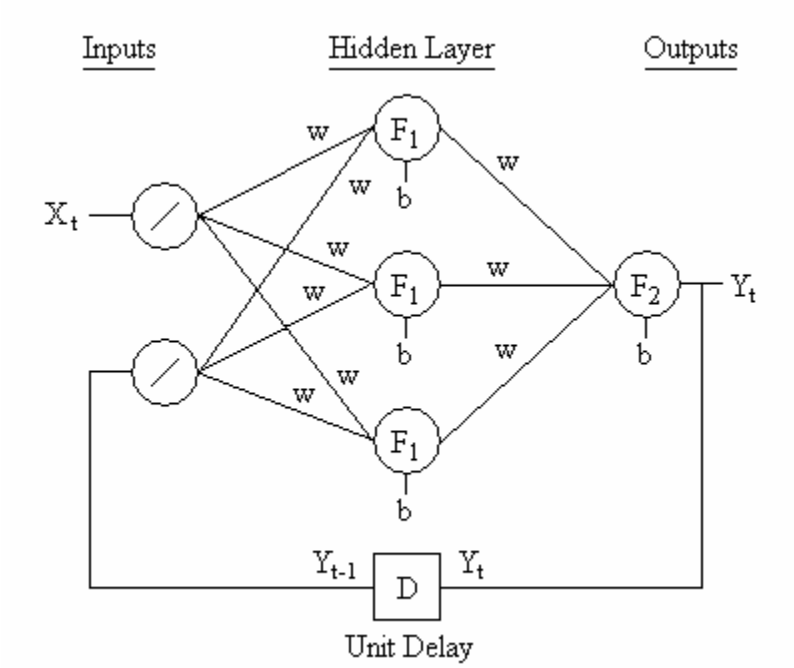


Figure 3.2: *Example of a three layer recurrent neural network.*

The RNN in Figure 3.2 can be defined by the equation:

$$Y_t = F_2(\sum_j(w_j * F_1(\sum_q (w_q * X_{t-q}) + \sum_p (w_p * Y_{t-p}) + b_j)) + b_k) \quad (3.14)$$

Where X_{t-q} is the input variable at time $t-q$, Y_{t-p} is the regressive output variable at time $t-p$, t is the discrete time variable, the w's are the network weights, the b's are the network biases, F_1 is the hidden layer activation function, F_2 is the output layer activation function and D is a one time unit delay.

In this network, the RNN uses one independent variable, X, as well as recursive values of the output, Y. For simplicity, this research will deal with recurrent networks of this form, although it should be pointed out that other forms of RNNs exist in the literature. The network can have any number q of previous and current values of X as input. There can also be any order p of previous network predictions for Y.

The standard linear ARMA model has two components: the auto-regressive (AR) part and the moving average (MA) part. The auto-regressive portion of the equation is

the recursive part that takes into account previous values of the variable being modeled. The moving average part includes the past history of model prediction error. The general form of an auto-regressive moving average model is:

$$Y_t = c_0 + \sum_p (c_p * Y_{t-p}) + \sum_q (c_q * E_{t-q}) \quad (3.15)$$

Where Y_t is the time series variable being modeled at time t, p is the order of the AR part, q is the order of the MA part, and E_t represents the prediction error (Mandic and Chambers, 2000). The equation can be represented shorthand as ARMA(p,q), where p and q are the size of the respective orders. The non-real time values of Y ($t < 1$) are defined to be zero. The error E_t can be calculated with equation (3.16) where $\hat{Y}_t$ is the predicted output at time t.

$$E_t = (\hat{Y}_t - Y_t) \quad (3.16)$$

E_t can also represent an independent input variable, and does not necessarily have to be calculated as the network error. When it is an independent variable, it will be represented by X_t instead of E_t (Chon, 1997).

The linear ARMA is popular in biological applications because it is relatively easy to create and analyze. However, when a more complex model is desired, nonlinear ARMA, or NARMA, models can be used. There are many forms of NARMA equations, but for this study, a polynomial NARMA will be investigated, as defined by equation (3.17) (Chon, 1997).

$$Y_t = (c_0 + \sum_p (c_p * Y_{t-p}) + \sum_q (c_q * E_{t-q}))^n \quad (3.17)$$

It can be seen from equations (3.15) and (3.17) that there will be essentially two different "depths" of ARMA models that can be explored. The first is a temporal

dimension, represented by the orders of p and q and the second is a nonlinear, or polynomial, order represented by n.

Another aspect of the ARMA equation that can be modified is the method of prediction. For one-day-ahead prediction models, the actual value of the previous output is used in the auto-regressive term. In full prediction, or multiple-day-ahead prediction, the estimated value of the output is used (Young and Chan, 1993).

The analysis of RNN and ARMA models will be performed in MATLAB. Recursive neural networks will be developed using the Neural Network Toolbox and trained using back-propagation similar to the way described earlier for FNNs. Comparing the accuracy and structure of the two models will be done in a manner similar to what is done with feed-forward networks and multiple polynomial regression equations.

ARMA model coefficients will be estimated using the method of least-squares. For purely auto-regressive models, or models with independent input variables, the standard least-squares algorithm will be used. However, when the ARMA equation includes an error term for the moving average part, a modified form of least-squares called the long-AR method will be used to estimate the model coefficients (Wolfram Research, 2006). The reason a different estimation method is needed is because the error term E_t is dependent on predicted values of the output (Equation 3.16), which are unknown at the time of coefficient estimation.

The long-AR method first estimates the output of the ARMA equation with an *AR(k)* model, where k is a large value. The next step is to use the predictions from this

auto-regressive model to calculate the error from the target data. Finally, the original ARMA equation is solved for using the estimated error terms.

3.2.1 Perceptron-level Analysis

For the first step in investigating the potential equivalence between recurrent networks and auto-regressive moving average models, a single linear recurrent perceptron will be compared to linear ARMA equations. In other words, the order of the regression polynomial order will be, $n = 1$. Tests will be performed on three variations of ARMA equations. First, an ARMA(1,1) model that uses an independent input. The inputs variables for this equation are the Y_{t-1} and X_t. Second, an ARMA(1,1) model that uses an error term. The input variables for this equation are Y_{t-1} and E_{t-1}. Third, an ARMA(3,0) model with no non-recurrent inputs. The input variables for this equation are Y_{t-1}, Y_{t-2} and Y_{t-3}. In all cases, formal equations will be derived to express ARMA coefficients in terms of the recurrent perceptron weights and biases.

The formal comparisons between RNNs and ARMA equations will be confirmed with empirical data. This will be similar to the methods used in the previous section, on comparing FNNs and MPR equations. The data will be synthetically created and trained to estimate both models. There will be two types of synthetic time series equations tested: stable and unstable equations. The prediction error $s(e)/s(y)$ of both models will be compared as well as the fit of the model output functions. It will be observed if both the RNN and ARMA are able to find the same regression parameters for the synthetic system.

3.2.2 Network-level Analysis

The network-level analysis for the recurrent neural network will combine the research of both the sigmoid hidden layer feed-forward network and the recurrent perceptron. A three-layer recurrent neural network with linear input layer, sigmoid hidden layer, and linear output layer will be compared to a third order NARMA(1,0) model ($n = 3$, $p = 1$ and $q = 0$). A single, recurrent input variable will be used for both models. Taylor expansion will be used to transform the recurrent network equation and the result will be compared to the third order NARMA(1,0) equation, term by term. The formal comparisons will be confirmed by training both models to a stable, synthetic data set. The fit of the output functions as well as the equivalence of the regression coefficients will be taken into consideration.

3.3 Application to Biological Phenomena

The artificial neural network and statistical regression equivalences that are identified through tasks described in the previous sections will be applied to neural network models in biological applications. For this thesis, the research will focus on the field of bioenvironmental engineering and hydrology. Specifically, this research will investigate the application of streamflow forecasting. However, it is important to note that the technology of neural networks and statistical regression as well as the knowledge gained on their potential equivalence can be applied to any other biological field.

The real world biological phenomenon that will be used is the issue of streamflow forecasting. This example will be used to compare the modeling abilities of artificial neural networks and statistical regression equations based on the findings from the

previous section. Both types of models will be used to predict the average daily streamflow of the Little Patuxent River in Maryland. This is the same water system used in a previous experiment that found ANNs to be a successful modeling tool (Salas et al., 2000). Combinations of average daily temperature (°F), daily precipitation (in), and the streamflow (cfs) from previous days will be used as independent variables.

The stream flow data from the Little Patuxent River is obtained from the U.S. Geological Survey (USGS, 2005). Climate data (precipitation and temperature) of the surrounding watershed is measured at the Clarksville gauging station and was obtained from the National Climatic Data Center (NCDC, 2005). Daily values from the years 1979, 1980, and 1984 are combined to form the training set, and the years 1989, 1991, and 1992 are combined to form the validation set. The maximum and minimum values for both sets are shown in Table 3.2. It is notable that the range of the validation data is slightly larger than that of the training data. This means the models will be tested on their ability to project predictions to data outside of their trained range. The graphs and histograms for both the training and validation data are presented in Appendix B.

Table 3.2: *Minimum and maximum values for the streamflow data.*

	Training Set		Validation Set	
	Min	Max	Min	Max
Precipitation (in)	0	3.09	0	3.80
Temperature (°F)	14	101	19	102
Streamflow (cfs)	6.7	2140	3.8	2420

3.3.1 Confirming the Accuracy of Neural Networks

The results from Salas et al. (2000) will be replicated and confirmed. Salas et al. (2000) concluded that a three-layer feed-forward network with sigmoid activation functions in the hidden and output layers outperformed a simple conceptual rainfall-

runoff statistical model. This study found that the best-fit model for predicting streamflow was a feed-forward network that used the current day's temperature and the current and previous days' precipitation as input variables, along with ten nodes in the hidden layer. The accuracy of training and validating the FNN will be verified as well as the optimal number of hidden nodes.

3.3.2 Comparison of ANNs and Regression Models

A series of artificial neural networks and statistical regression equations will be used to model the streamflow of the Little Patuxent River. The variables of interest will be: Q_t, the average daily discharge (cfs) at time t; P_t, the daily precipitation (in); T_t, the average daily temperature (°F); and t, the day. There will be two groups of input parameters used, non-recursive (Functions 1 through 4) and recursive (Functions 5 through 8), as displayed in Table 3.3. Each input data set uses a different combination of precipitation, temperature and streamflow. There will always be only one output for the models, which is streamflow.

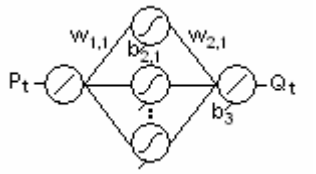
Table 3.3: *The functions and input sets that will be approximated by the models.*

Non-recursive Functions	Recursive Functions
Function 1: $Q_t = f(P_t)$	Function 5: $Qt = f(P_t , Q_{t-1})$
Function 2: $Q_t = f(P_t , P_{t-1})$	Function 6: $Q_t = f(P_t , P_{t-1} , Q_{t-1})$
Function 3: $Q_t = f(P_t , T_t)$	Function 7: $Q_t = f(P_t , T_t , Q_{t-1})$
Function 4: $Q_t = f(P_t , P_{t-1} , T_t)$	Function 8: $Q_t = f(P_t , P_{t-1} , T_t , Q_{t-1})$

3.3.2.1 Non-recursive Input - FNN versus MPR

The non-recursive functions (Functions 1 through 4) will be modeled by a series of feed-forward networks and multiple polynomial regression equations. Function 1 for example, which uses only precipitation as an input to predict streamflow, is represented

by the network and polynomials shown in Figure 3.3. The other functions will be similar, except that they will involve more input variables (combinations of precipitation and temperature).

FNN: 

vs. MPR: $\hat{Q}_t = c_0 + c_1 * P_t + c_2 * P_t^2 + \ldots + c_n * P_t^n$

Figure 3.3: *Models used to estimate Function 1.*

The structure of the feed-forward networks will be either a three or four layer network. For the first phase of tests, a traditional three layer network will be used. The input and output layers will use a linear activation function, while the hidden layer nodes will use a hyperbolic tangent (sigmoid) activation function. Two additional commonly used network structures, shown in Table 3.4, will also be tested to compare the accuracy and functionality of the different networks. For the two hidden layer network, the second sigmoid layer will only have one hidden node. For the polynomial network, the activation function will be the third order Taylor series expansion of the sigmoid function.

Table 3.4: *ANN structures to be tested.*

Network Structure
Linear - Sigmoid - Linear
Linear - Sigmoid - Sigmoid - Linear
Linear - Polynomial - Linear

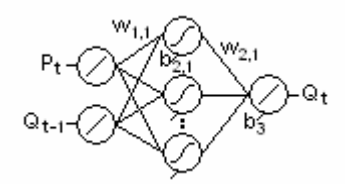
The weights and biases for the ANNs are initialized to random normal values and will be trained using standard back-propagation. In every case investigated, five trial networks will be trained to minimize the effects of the random initialization. A maximum of five-hundred epochs will be used to train the FNNs. ANN predictions and structure will be compared to those of multiple polynomial regression equations.

The streamflow and climate data will be pre-processed to be a better fit with the critical range of the network's activation function. For the hyperbolic tangent activation function used in this study, the best range is from -1 to +1. For one set of trials, the data will be linearly scaled to the range -0.8 to +0.8 by using equation (3.12). For another set of trials, the streamflow data will be transformed by taking the log of the data, standardizing it to a normal distribution, and then scaling it to the range -0.8 to +0.8 by using equation (3.13). The reason for attempting log-scaling is that the streamflow distribution for both the training and validation sets is skewed, with most of the data values being very small. The prediction accuracy for the models will be observed for both types of pre-processing, to see if the log normal transformation has any advantage over simple linear scaling.

The number of hidden nodes in the ANNs will be varied from one to thirty-five. The order of the polynomial regression equations will vary from one to fifteen. The models will be trained and validated using the same data series, with five trials being performed for each. For each trial, the error ratio *s(e)/s(y)* will be calculated, based on equation (3.11). The error ratio for the ANNs for each number of hidden nodes will be compared to the error ratio found for each order of MPR. Two models will be considered to be empirically equivalent if the error ratio between the two is similar. The structure of those models as well as the output functions that they both produce (graph) will then be compared to determine their similarities. If possible, the network parameters will be converted to equivalent regression parameters using the formal equations developed earlier.

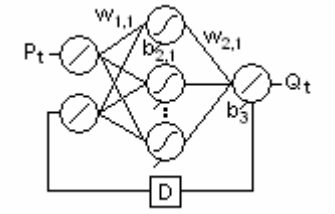
3.3.2.2 Recursive Input - RNN versus ARMA

For the recursive functions (Functions 5 through 8), feed-forward networks and recurrent neural networks will be compared to auto-regressive moving average equations. The models will either be used for one-day-ahead prediction or full prediction. One-day-ahead prediction will use the actual Q_{t-1} as an input value, while full prediction will use the predicted Q_{t-1} value as input. Keeping to the intent of the network structures, FNNs will only be used for one-day-ahead prediction, while RNN will only be used for full prediction (multiple-day-ahead). ARMA equations will be tested for both prediction methods. Figures 3.4 and 3.5 show the models used to estimate Function 5 and the other recursive functions will be tested in a similar manner.

FNN: 

vs. ARMA: $\hat{Q}_t = c_1 + c_2 * P_t + c_3 * Q_{t-1} + c_4 * P_t * Q_{t-1} + \ldots$

Figure 3.4*: One-day-ahead prediction models for Function 5.*

RNN: 

vs. ARMA: $\hat{Q}_t = c_1 + c_2 * P_t + c_3 * \hat{Q}_{t-1} + c_4 * P_t * \hat{Q}_{t-1} + \ldots$

Figure 3.5: *Full prediction models for Function 5.*

The testing in this section will proceed similarly to the non-recursive section. Recurrent networks will have the same structure as the feed-forward networks, but will have a connection from the output layer to the input layer with a single time-unit delay function. Like before, five-hundred epochs will be used when training FNNs, while a maximum of fifty epochs will be used for RNNs. The difference in number of epochs used is due to the much longer training time required by recurrent networks. Instead of

multiple polynomial regression, all ANNs will be compared to nonlinear polynomial auto-regressive moving average (NARMA) models.

A summary of the ANN and regression models to be tested and compared for both the recurrent and non-recurrent sections is presented in Table 3.5.

Table 3.5: *Summarizes the comparisons being tested between ANNs and regression.*

Network	Regression	Prediction Method	Pre-processing	# of Hidden Nodes and # of Orders
FNN RNN	MPR ARMA	One-day-ahead Full prediction	Linear Scaled Log Normal	1-45 Hidden Nodes 1-15 Orders

4 Results and Discussion

4.1 Equivalence of FNN and MPR

4.1.1 Perceptron-level Analysis

4.1.1.1 Linear Perceptron and Linear Regression

The first step in the investigation confirmed the proposal by Sarle (1994) that a neural perceptron with a linear activation function can be reduced to a multiple linear regression equation. A linear perceptron with *k* input variables is represented in Figure 4.1. The linear regression model is defined by:

$$Y = c_0 + c_1 * X_1 + c_2 * X_2 + ... + c_k * X_k \quad (4.1)$$

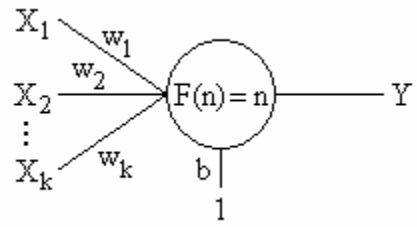


Figure 4.1: *An artificial perceptron with a linear activation function.*

The perceptron in Figure 4.1 can be represented mathematically by equation (4.2).

$$Y = b + w_1 * X_1 + w_2 * X_2 + ... + w_k * X_k \quad (4.2)$$

Comparing (4.1) to (4.2) term by term, the following formal equivalences can be seen:

$$c_0 = b \quad (4.3)$$

$$c_1 = w_1 \quad (4.4)$$

$$c_2 = w_2 \quad (4.5)$$

$$c_k = w_k \quad (4.6)$$

This formal comparison was then tested empirically. Synthetic input and output data was used to determine the accuracy of both the linear perceptron and linear regression. For the training data, the value of *X* was given integer values ranging from

-149 to 150. The target values for Y were then determined by the equation:

$$Y = 100 + 2 * X + 10 * \varepsilon \quad (4.7)$$

A normalized random error, designated by ε, was assigned to the value of Y to simulate noise in the data. A set of validation data was created in the same manner, but with the values of X ranging from 151 to 250. The purpose of this is to test the ability of the models to extrapolate to future values of Y not included in the training set. The linear perceptron and linear regression model were both fit to model the synthetic data. Results show that the models produced identical accuracy and both produced the same best-fit line (Figure 4.2 and Table 4.1).

Once the accuracy was confirmed, the weight and bias of the trained linear perceptron were converted to polynomial coefficients using (4.3) and (4.4) and examined to determine if it was able to accurately predict the parameters of the regression equation. Results are presented in Table 4.2 and indicate that the weights of the linear perceptron match the parameters of the linear regression equation. The structure of the equation representing the perceptron is identical to the traditional form of a multiple linear regression equation. This shows that the two models have the ability to produce the same statistical equation.

It should be noted that for natural events such as watershed response to precipitation a wider range both during training and validation should be considered in order to encompass the stochastic and variability in nature (i.e., the number of dry years, wet years, and normal years in the data set).

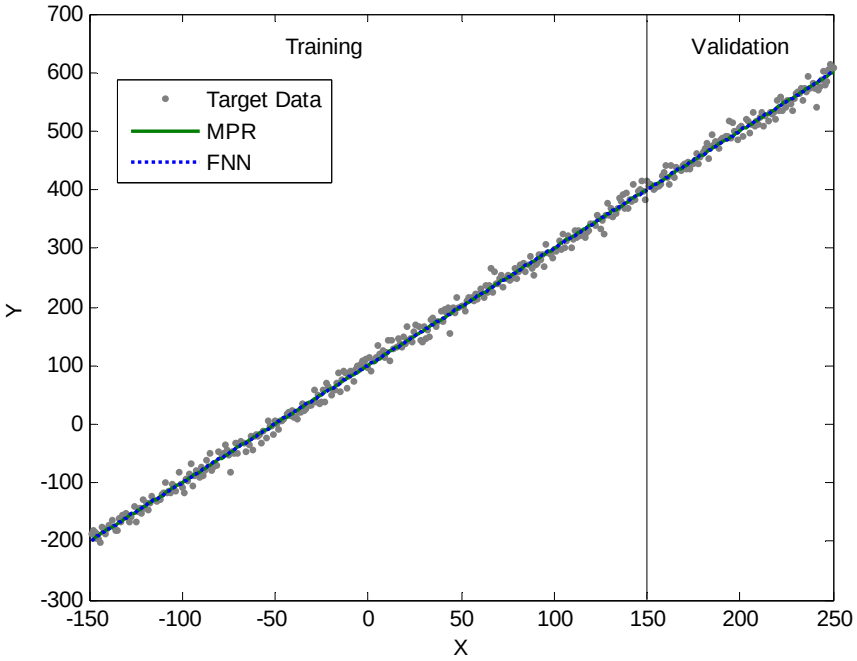


Figure 4.2: *Linear regression and linear perceptron trained to synthetic data.*

Table 4.1: *Linear Perceptron - Prediction error from both models.*

	1st Order MPR	Linear Perceptron
Training Error *s(e)/s(y)*	0.0607	0.0607
Validation Error *s(e)/s(y)*	0.1898	0.1898

Table 4.2: *Linear Perceptron - Regression coefficients found by both models.*

	Target	1st Order MPR	Linear Perceptron
c_0	100	100.2678	100.2678
c_1	2	1.9997	1.9997

4.1.1.2 Sigmoid Perceptron and Polynomial Regression

The artificial perceptron was then tested using a hyperbolic tangent activation function. This is a function commonly used in ANN applications because of its ability to make the network nonlinear. The sigmoid perceptron is shown in Figure 4.3. This perceptron can be represented mathematically by:

$$Y = tanh(b + w_1X_1 + w_2X_2 + ... + w_kX_k) \tag{4.8}$$

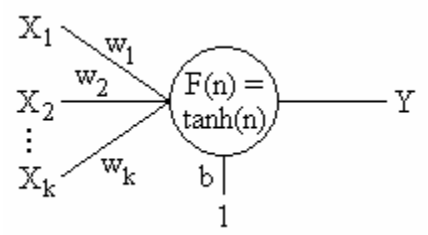


Figure 4.3: *An artificial perceptron with a sigmoid activation function.*

In this form, this equation is equivalent to a nonlinear regression equation that uses the hyperbolic tangent function. The coefficients (b and w_i) of a regression equation in this form can be determined by taking the inverse hyperbolic tangent of the output data, Y, and then performing least squares.

To compare the sigmoid perceptron to multiple polynomial regression, Taylor series expansion on equation (4.8) is performed using the *taylor* function in MATLAB on the hyperbolic tangent function. The following Taylor series expansion is established:

$$\tanh(x) = x - \frac{1}{3}x^3 + \frac{2}{15}x^5 - \frac{17}{315}x^7 + \frac{62}{2835}x^9 - \ldots \tag{4.9}$$

For simplicity, this research will begin by using only the first two terms of (4.9) to estimate the hyperbolic tangent function, up to the third order by:

$$\tanh(x) \approx x - \frac{1}{3}x^3 \tag{4.10}$$

The third order Taylor expansion of the sigmoid activation function is only a rough estimate, and is most accurate at small values of X. A graph of both functions show that equation (4.10) is not accurate outside of the range -0.5 to +0.5 (Figure 4.4).

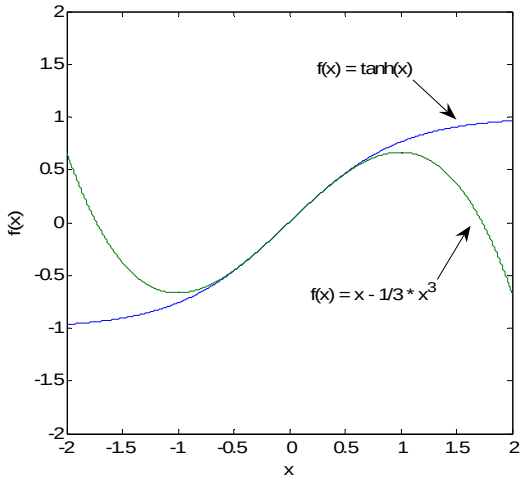


Figure 4.4: *The third order Taylor expansion of tanh(x).*

For one input variable, X, the perceptron equation (4.8) and the third order Taylor expansion equation (4.10) can be combined to form the approximate perceptron equation:

$$Y = (b + wX) - \frac{1}{3}(b + wX)^3 \tag{4.11}$$

Using polynomial expansion and combining like terms produces the equation:

$$Y = (b - \frac{1}{3}b^3) + (w - b^2 w) * X + (-bw^2) * X^2 + (-\frac{1}{3}w^3) * X^3 \tag{4.12}$$

This equation has the structure of a third order polynomial:

$$Y = c_0 + c_1 * X + c_2 * X^2 + c_3 * X^3 \tag{4.13}$$

By comparing similar terms, the following parameter conversion equations can be set up:

$$c_0 = b - \frac{1}{3}b^3 \tag{4.14}$$

$$c_1 = w - b^2 w \tag{4.15}$$

$$c_2 = -bw^2 \tag{4.16}$$

$$c_3 = -\frac{1}{3}w^3 \tag{4.17}$$

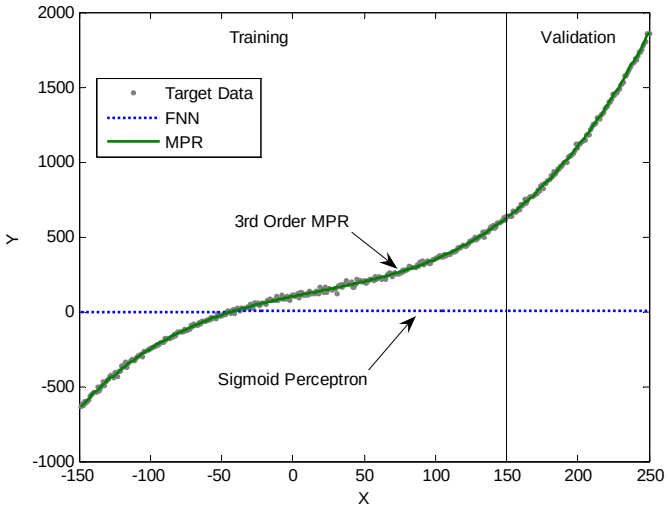
Equations (4.14) to (4.17) show how third order polynomial regression can be estimated by the weight and bias of a sigmoid perceptron. However, the problem with these equations is that there are two variables being used to predict four unknowns. For a given third order equation (4.13), it is not likely that a solution for *b* and *w* can be found.

These formal equations were tested empirically in the same manner as the previous section with the linear perceptron. The following third order polynomial equation was used to create synthetic target data:

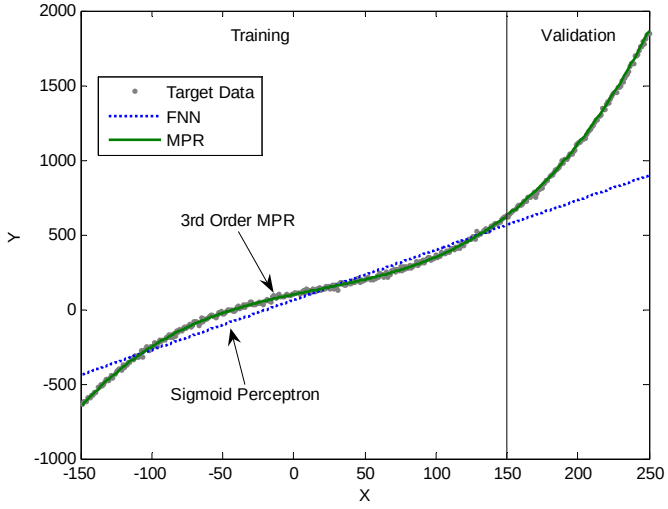
$$Y = 100 + 2 * X - 0.005 * X^2 + 0.0001 * X^3 + 10 * \varepsilon \tag{4.18}$$

Without scaling the data, the perceptron is unable to model the data, since the output is restricted to the range -1 to +1 (Figure 4.5a). However, even when scaling the input and output data to a range of -0.1 to +0.1, the perceptron is still incapable of matching the trend of the data (Figure 4.5b). The perceptron is only able to train itself to have an error as low as *s(e)/s(y)* = 0.2088, which is much larger than the error of the regression equation (0.0338) (Table 4.3). It is interesting to note that the shape of the perceptron output function in Figure 4.5b looks linear. This suggests that with only one perceptron, an ANN is insufficient for modeling regression orders larger than one.

Empirical results confirm that the sigmoid perceptron is unable to estimate the parameters of the polynomial regression equation. Network training found the best-fit weight ($w = 0.7957$) and bias ($b = 0.0109$) for the perceptron. The regression coefficients derived from the network parameters using equations (4.14) to (4.17) are shown in Table 4.4. As expected from the linear output function in Figure 4.5b, the perceptron is not close in predicting the higher order polynomial coefficients. In order to accurately model the regression equation, more parameters are needed in the ANN, which means a larger network is required.



a)



b)

Figure 4.5: *The output function for MPR and sigmoid perceptron with non-scaled (a) and scaled (b) data.*

Table 4.3: *Sigmoid Perceptron - Prediction error from both models for scaled data.*

	3rd Order MPR	Sigmoid Perceptron
Training Error *s(e)/s(y)*	0.0338	0.2088
Validation Error *s(e)/s(y)*	0.0395	1.3935

Table 4.4: *Sigmoid Perceptron - Regression coefficients found by both models (values in scaled domain).*

	3rd Order MPR	Sigmoid Perceptron
c_0	0.0166	0.0109
c_1	0.4719	0.7956
c_2	-1.7106	-0.0069
c_3	53.4606	-0.1679

4.1.2 Network-level Analysis - Polynomial Activation Function

4.1.2.1 Specific Example - Third Order with One Variable

When modeling a one input variable, third order equation, a single artificial perceptron is unable to replicate the results of a regression equation. The investigation will continue by looking at larger artificial neural networks. This time, a three layer feed-forward neural network will be used. Since this is a one input and one output system, the input and output layers will only have one node in each. However, by modifying the number of hidden nodes, the structure and number of parameters in the network will change.

First, a basic FNN with one hidden node will be tested (Figure 4.6). Like the third order regression equation (4.13), this FNN also has four parameters. Combining the equations of the perceptrons in this network produces the overall FNN equation:

$$\begin{aligned} Y &= w_2 * (w_1 X + b_1)^3 + b_2 \\ &= w_2 b_1^3 + b_2 + 3 w_1 w_2 b_1^2 X + 3 w_1^2 w_2 b_1 X^2 + w_1^3 w_2 X^3 \end{aligned} \tag{4.19}$$

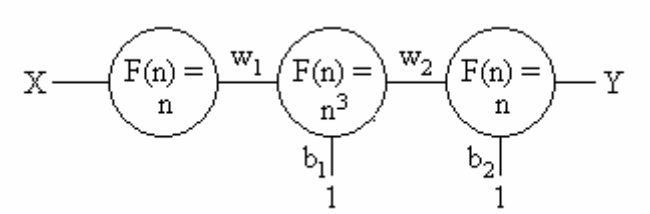


Figure 4.6: *One hidden node FNN with cubic activation function.*

By comparing the terms of (4.12) and (4.19), the following equivalences are found:

$$c_0 = w_2 b_1^3 + b_2 \tag{4.20}$$

$$c_1 = 3 w_1 w_2 b_1^2 \tag{4.21}$$

$$c_2 = 3 w_1^2 w_2 b_1 \tag{4.22}$$

$$c_3 = w_1^3 w_2 \tag{4.23}$$

While there are now four parameters in each model, the network weight w_2 appears in all of the coefficient equations (4.20) to (4.23). As a result, the neural network is unable to train its weights and biases to find the unique solution for the regression parameters. Both models were then trained and validated on the same synthetic data series that was used for the single sigmoid perceptron in the previous section, which was given as equation (4.18).

Figure 4.7 shows that the FNN with one hidden node is unable to model the data with the same degree of accuracy as the third order regression equation. The error for the FNN ($s(e)/s(y)$ = 0.2367) was much higher than for MPR ($s(e)/s(y)$ = 0.0327). This difference in error is even more dramatic in the validation data (2.0150 for FNN as opposed to 0.0368 for MPR) (Table 4.5). Using equations (4.20) to (4.23), the regression coefficients were estimated from the FNN parameters (Tables 4.6 and 4.7). Results indicate that the one hidden node FNN is able to come close to determining the regression coefficients, but is inaccurate enough to produce a different output function as seen in Figure 4.7.

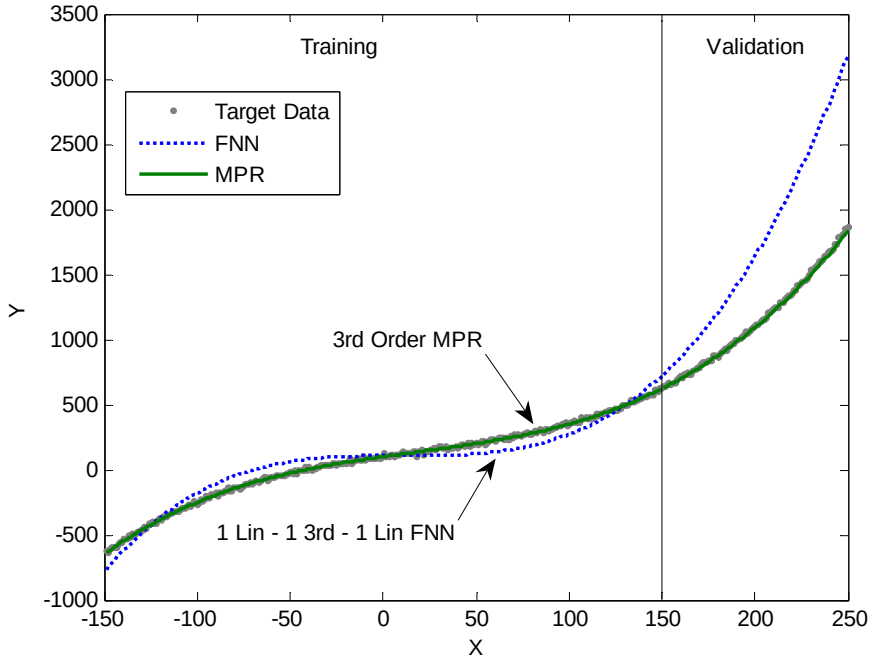


Figure 4.7: *The functions produced by third order regression and one hidden node FNN.*

Table 4.5: *One Cubic Hidden Node - Prediction error from both models.*

	3rd Order MPR	1 Lin - 1 3rd - 1 Lin FNN
Training Error *s(e)/s(y)*	0.0327	0.2367
Validation Error *s(e)/s(y)*	0.0368	2.0150

Table 4.6: *One Cubic Hidden Node - Trained network weights and biases.*

w_1	w_2	b_1	b_2
0.1602	0.0536	-1.4865	107.8844

Table 4.7: *One Cubic Hidden Node - Regression coefficients found by both models.*

	Target	3rd Order MPR	1 Lin - 1 3rd - 1 Lin FNN
c_0	100	99.7267	107.7082
c_1	2	2.0205	0.0569
c_2	-0.005	-0.0050	-0.0061
c_3	0.0001	0.0001	0.00022045

Similar tests were performed using two hidden nodes in the hidden layer (Figure 4.8). Like before, a cubic activation function was used in the hidden nodes. The output from this FNN can be defined by:

$$Y = w_3 * (w_1 X + b_1)^3 + w_4 * (w_2 X + b_2)^3 + b_3 \tag{4.24}$$

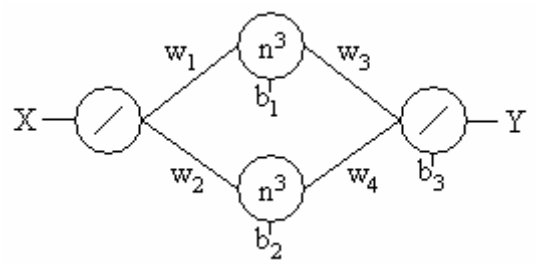


Figure 4.8: *Two hidden node FNN with cubic activation function.*

Using polynomial expansion on equation (4.24) and combining terms, the result is compared to the polynomial regression equation (4.13). The coefficients of the polynomial can then be defined in terms of the network weights and biases as:

$$c_0 = w_3 b_1^3 + w_4 b_2^3 + b_3 \tag{4.25}$$

$$c_1 = 3 w_1 w_3 b_1^2 + 3 w_2 w_4 b_2^2 \tag{4.26}$$

$$c_2 = 3 w_1^2 w_3 b_1 + 3 w_2^2 w_4 b_2 \tag{4.27}$$

$$c_3 = w_1^3 w_3 + w_2^3 w_4 \tag{4.28}$$

The results from the empirical test show that both the FNN and the MPR create the same output function when trained to the synthetic training data (Figure 4.9). Both models also predict the validation data with the same degree of accuracy (Table 4.8). When the network parameters (Table 4.9) are transformed into regression coefficients, the results are identical (Table 4.10). These results confirm that a feed-forward network with two cubic activation functions can be equivalent to a third order regression equation with one input variable.

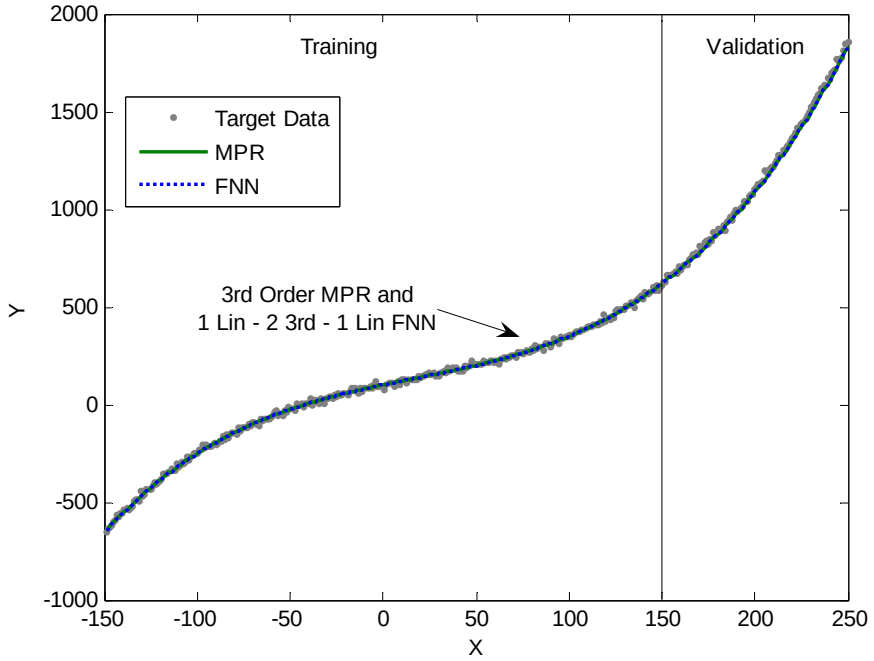


Figure 4.9: *The functions produced by third order regression and two hidden node FNN.*

Table 4.8: *Two Cubic Hidden Nodes - Prediction error from both models.*

	3rd Order MPR	1 Lin - 2 3rd - 1 Lin FNN
Training Error *s(e)/s(y)*	0.0340	0.0341
Validation Error *s(e)/s(y)*	0.0304	0.0309

Table 4.9: *Two Cubic Hidden Nodes - Trained network weights and biases.*

w_1	w_2	w_3	w_4	b_1	b_2	b_3
-0.0170	0.0433	-2.1006	1.0980	-3.7277	-1.9174	0.0583

Table 4.10: *Two Cubic Hidden Nodes - Regression coefficients found by both models.*

	Target	3rd Order MPR	1 Lin - 2 3rd - 1 Lin FNN
c_0	100	101.1230	101.1230
c_1	2	2.0156	2.0156
c_2	-0.005	-0.0050	-0.0050
c_3	0.0001	0.0001	0.000099689

It is interesting to note that the two hidden node FNN did not always match the results of the MPR equation. Some of the trials produced results as seen in Figure 4.10 instead. In this case, the output function for the FNN looks more like the results from the one hidden node network (Figure 4.7) and does not fit the trend of the synthetic data. As should be expected in this situation, the FNN is unable to predict the coefficients to the polynomial equation (Table 4.13). During the training phase for the failed network, all five hundred of the epochs were executed by back-propagation without reaching the target performance. In comparison, correctly trained networks would meet the target performance and stop training after only a few epochs. The reason for the failed trials is likely to be caused by the random initialization of network parameters. In the case of the failed network, the initialized parameters most likely put the network in one of the local minima of the error function, causing it to be unable to reach the true regression equation.

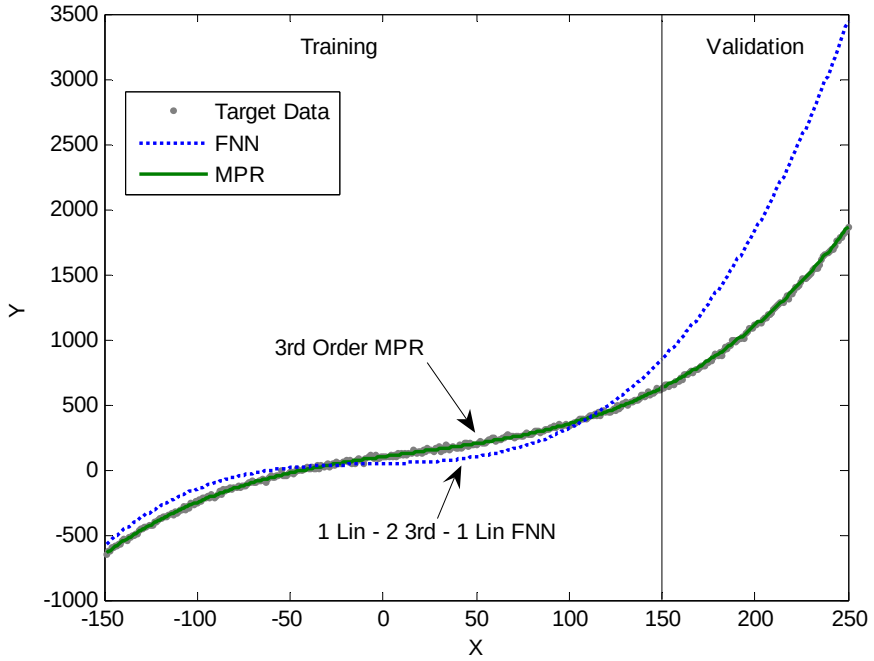


Figure 4.10: *Failed trial with two hidden nodes.*

Table 4.11: *Two Cubic Hidden Nodes (Failed) - Prediction error from both models.*

	3rd Order MPR	1 Lin - 2 3rd - 1 Lin FNN
Training Error *s(e)/s(y)*	0.0328	0.2867
Validation Error *s(e)/s(y)*	0.0255	2.6045

Table 4.12: *Two Cubic Hidden Nodes (Failed) - Trained network weights and biases.*

w_1	w_2	w_3	w_4	b_1	b_2	b_3
1.3882	-0.0090	0.0001	-1.1253	6.8013	-3.4058	2.2951

Table 4.13: *Two Cubic Hidden Nodes (Failed) - Regression coefficients found by failed trial.*

	Target	3rd Order MPR	1 Lin - 2 3rd - 1 Lin FNN
c_0	100	100.7072	46.7756
c_1	2	1.9837	0.3651
c_2	-0.005	-0.0050	0.0038
c_3	0.0001	0.0001	0.00019580

While the two hidden node FNN is able to replicate the third order regression equation, it uses more parameters than the MPR. Seven parameters are used in the FNN in the form of weights and biases, as opposed to the four used to represent the third order polynomial. This can be inconvenient in situations with a large amount of data. Typically modelers would like to find the most efficient equation for representing the system in question, since extra parameters tend to make the equation overly complex.

To make the two hidden node FNN simpler, the weights and biases used for the output node were ignored. In this modified two hidden node network, w_3 and w_4 were set equal to one and b_3 was set to zero (Figure 4.11), reducing the network equation to:

$$Y = (w_1 X + b_1)^3 + (w_2 X + b_2)^3 \tag{4.29}$$

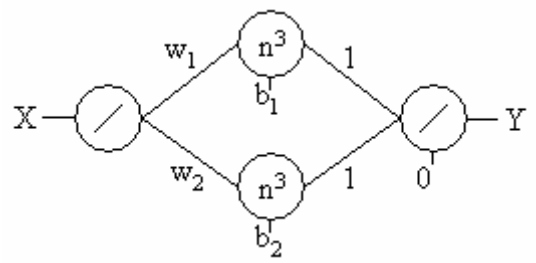


Figure 4.11: *Modified two hidden node FNN with parameters w_3, w_4 and b_3 ignored.*

This network arrangement produces polynomial coefficients determined by:

$$c_0 = b_1^3 + b_2^3 \tag{4.30}$$

$$c_1 = 3w_1 b_1^2 + 3w_2 b_2^2 \tag{4.31}$$

$$c_2 = 3w_1^2 b_1 + 3w_2^2 b_2 \tag{4.32}$$

$$c_3 = w_1^3 + w_2^3 \tag{4.33}$$

When this new network was trained to the empirical data, it was able to produce results just as effectively as the fully parameterized network. The output function matched that of the MPR equation with the same degree of accuracy as the full parameter

network (Figure 4.12). Transforming the network parameters with (4.30) to (4.33) confirm that the network finds the same regression coefficients as MPR (Table 4.16). It should also be noted that this network sometimes failed in the same manner as the full parameter two hidden node network.

It is interesting that this four parameter network succeeded where the other four parameter network (the one hidden node network) failed. This suggests that as long as there are enough parameters before the hidden node, the network will be able to replicate a given MPR of the same order with the same number of parameters. The network weights that were ignored (w_3, w_4 and b_3) are mostly used by the network for scaling and offsetting the output from the hidden layer, where the weights are used for scaling and the bias for offsetting.

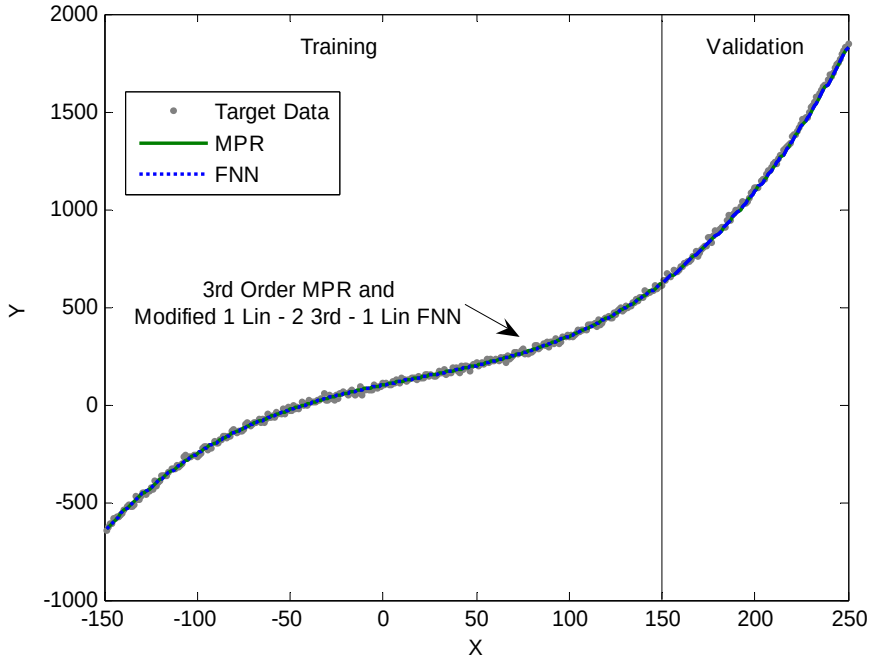


Figure 4.12: *The functions produced by third order regression and modified two hidden node FNN.*

Table 4.14: *Modified Two Cubic Hidden Nodes - Prediction error from both models.*

	3rd Order MPR	Modified 1 Lin - 2 3rd - 1 Lin FNN
Training Error *s(e)/s(y)*	0.0343	0.0343
Validation Error *s(e)/s(y)*	0.0415	0.0415

Table 4.15: *Modified Two Cubic Hidden Nodes - Trained network weights and biases.*

w_1	w_2	w_3	w_4	b_1	b_2	b_3
0.0445	0.0218	1	1	-1.9936	4.7669	0

Table 4.16: *Modified Two Cubic Hidden Nodes - Regression coefficients found by both models.*

	Target	3rd Order MPR	Modified 1 Lin - 2 3rd - 1 Lin FNN
c_0	100	100.3981	100.3981
c_1	2	2.0173	2.0173
c_2	-0.005	-0.0051	-0.0051
c_3	0.0001	0.0001	0.00009859

4.1.2.2 Generalization to MPR

The previous section shows that the number of hidden nodes in an ANN has a large effect on their performance. The number of hidden nodes is an important aspect of ANN modeling. Varying the number of hidden nodes gives neural networks versatility. However, as discussed earlier, there are no formal methods for determining the best number of hidden nodes to use. In this section, FNNs with a variable number of hidden nodes will be used to model polynomial regression equations of various orders. The order and number of variables of the regression equation will be compared to the minimum number of hidden nodes needed to replicate the equation.

The research from the previous section in one input variable, third order regression equations was expanded to investigate higher orders of polynomials with multiple input variables. Synthetic data was used to fit feed-forward neural networks with polynomial activation functions to polynomial multiple regression equations. The minimum number of hidden nodes required to model each MPR equation is presented in Table 4.17. In each case, initial observations indicate that the FNN has at least as many parameters as the equivalent MPR. Most of the time, the total number of parameters is higher for the feed-forward networks. This is due to the large number of connections that are made between each layer of the network. Each of these connections has a weight parameter assigned to it and each node has a bias parameter. Many of these network parameters are most likely not required, but they are included in the interest of making complete networks, which are typically used in real life applications.

The results from this test confirm that for all first order equations, only one hidden node is required. Additional hidden nodes are not required for additional

variables. This is because the linear regression coefficients for each term are controlled by the weights going from the input layer to the hidden node.

For all of the second order MPR equations, the minimum number of hidden nodes appears to be equal to the number of input variables. As the order of the regression equation increases and the number of variables increases, the minimum number of hidden nodes becomes larger. The basic trend that occurs in the results is that as the number of parameters in the underlying regression equation increases, the number of hidden nodes required increases, which is expected.

Table 4.17: *Minimum number of hidden nodes required to replicate a MPR.*

Number of Variables (A)	MPR Order (B)	Number of MPR Parms (C)	Number of Hid Nodes (D)	Number of FNN Parms (E)	Modified FNN (E - D) (F)
1	1	2	1	4	3
	2	3	1	4	3
	3	4	2	7	5
	4	5	2	7	5
	5	6	3	10	7
2	1	3	1	5	4
	2	6	2	9	7
	3	10	3	13	10
	4	15	5	21	16
	5	21	8	33	25
3	1	4	1	6	5
	2	10	3	16	13
	3	20	5	26	21
	4	35	9	46	37
	5	56	16	81	65
4	1	5	1	7	6
	2	15	4	25	21
	3	35	7	43	36
	4	70	15	91	76
	5	126	26	157	131
5	1	6	1	8	7
	2	21	5	36	31
	3	56	10	71	61
	4	126	22	155	133
	5	252	47	330	283

One of the more interesting cases from these results is that for modeling a third order, single input, regression equation. Table 4.17 confirms the results that were found in the previous section. As discussed before, simply looking at the number of parameters in the two types of models one would initially speculate that a feed-forward network with one hidden node (resulting in four parameters) could accurately replicate a third order regression model with one input (which also uses four parameters). However, the results indicate that at least two hidden nodes (with seven parameters) are required by a FNN to reproduce the MPR equation.

As mentioned before, the feed-forward networks always contained more parameters than the equivalent regression equation. This can be undesirable for modelers, because the extra parameters make the model needlessly more complex. This is also a concern for large systems with lots of data, where memory limitations might become an issue. However, in many cases, this can not be avoided. For example, for a two input variable, third order system, it was found that a minimum of three hidden nodes were required. This FNN uses thirteen parameters, compared to the MPR's ten parameters. If only two hidden nodes are used, the network will only have nine parameters, making it unable to duplicate the regression equation.

While this observation is true for many of the multiple polynomial equations, it appears to fail for some of the examples. There are some cases where it is possible to have fewer hidden nodes than the optimal one listed in Table 4.17 and still have more parameters than the regression equation being modeled. Take for example, a third order equation with three variables, which results in twenty regression parameters. The minimal number of hidden nodes found was five, with twenty-six parameters. A four

hidden node network would have twenty-one parameters, which is still larger than twenty. However, this discrepancy can be explained by looking at the number of network parameters minus the number of weights going from the hidden layer to the output layer, shown in Table 4.17 as Column F. As discussed in the previous section, these weights do not contribute much to the uniqueness of the FNN output function. As a result, they can usually be ignored. By subtracting this number from the total number of parameters, the result is a more accurate representation of the number of effective parameters in the neural network. With the three input, third order example, the number of effective parameters for a four hidden node network is twenty-one minus four, or seventeen, which is less than the number of regression parameters. This means that the four hidden node model is not large enough and five hidden nodes are needed.

For all of the cases, the number of total network parameters minus the number of secondary weights (equal to the number of hidden nodes) was still larger than the number of regression parameters. This reaffirms that the weights connecting the hidden layer to the output layer seem to be unnecessary and ineffective at improving the complexity of the overall network function.

One of the problems noticed with higher order MPR equations, mainly with fifth order, is that it took more trials to determine the minimum number of hidden nodes. And in most cases for fifth order polynomials, it seems like minimum number of hidden nodes seen in Table 4.17 could be smaller. For example, for a fifth order, five input variable equation, the lowest number of hidden nodes determined was forty-five. However, based on comparing the number of parameters like before, a forty-two hidden node network should be the optimal number. The complexity of these high ordered equations most

likely makes it difficult for the FNN to find the unique, best-fit solution by using back-propagation. Also, there probably exist a large number of local minima in the error function.

Results showed that neural networks could accurately replicate and predict the models formed by regression based methods. Different combinations of the number of predictor variables used and the highest polynomial order underlying the physical data were used to compare the two models. From this, a relationship was found between the simplest model required for the neural network and the equation used for regression. A trend was found relating the minimal number of hidden nodes required in the feed-forward network and the order of the equation the network represents. It has been shown that a neural network with an activation function of order N can be reduced to a polynomial regression equation also of order N, as long as there are enough parameters in the ANN based on the number of hidden nodes. This is the critical number of hidden nodes required for a particular equation order N and number of input parameters M. Fewer hidden nodes will not allow the ANN to replicate the complexity of the underlying polynomial regression equation. After this critical number of hidden nodes, the model accuracy will not increase any further, because the network has already replicated the regression equation.

4.1.3 Network-level Analysis - Sigmoid Activation Function

4.1.3.1 Specific Example - Third Order with One Variable

The previous section found that two hidden nodes were required for a FNN with a polynomial activation function to replicate a third order regression equation. These results were then tested with a sigmoid activation function in the hidden layer.

First, a sigmoid network with only one hidden node was trained to the synthetic third order polynomial equation (4.18) (Figure 4.13). Interestingly, while this network failed to replicate the third order regression target data, the output function of this network resembles that of a second order regression equation. When a second order MPR is estimated along with the one hidden node network, the results are encouraging. The training and validation error was similar for both models (Table 4.18) as well as their output function (Figure 4.14).

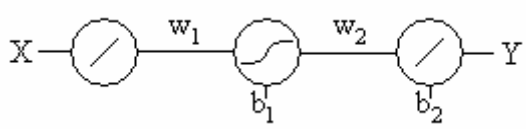


Figure 4.13: *One hidden node FNN with sigmoid activation function.*

However, it would be difficult to convert the network parameters into regression coefficients in this case because the hyperbolic tangent activation function is odd. This means that the Taylor series expansion uses only odd numbered powers. This seems to suggest that with only one hidden node, a FNN with a sigmoid activation function is at most able to model a second order regression equation. This evidence strengthens the argument that there is a relationship between the number of hidden nodes in an ANN and the order of the statistical regression equation it is equivalent to.

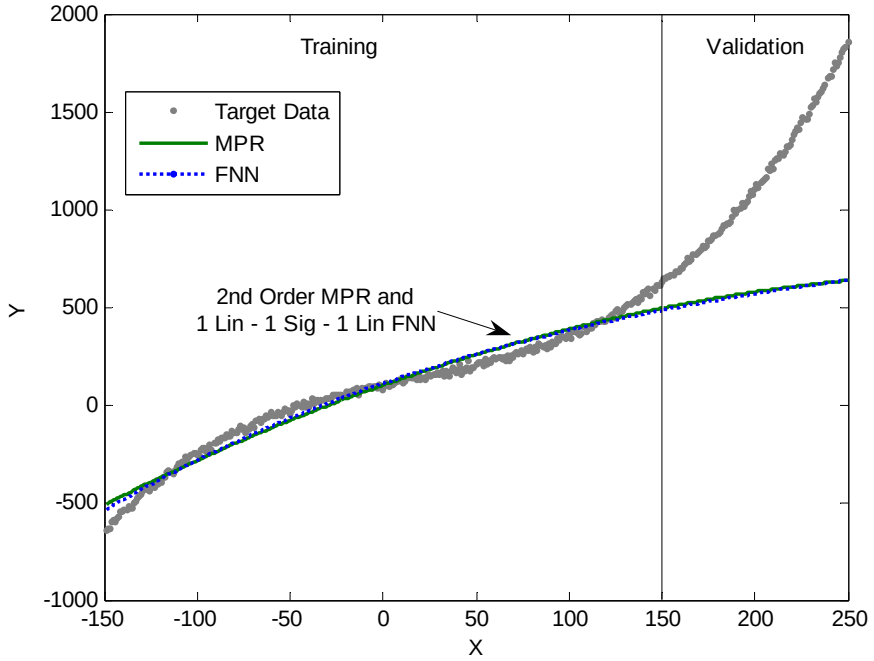


Figure 4.14: *A second order MPR and one hidden node FNN produce similar outputs.*

Table 4.18: *One Sigmoid Hidden Node - Prediction error from both models.*

	2nd Order MPR	1 Lin - 1 Sig - 1 Lin FNN
Training Error *s(e)/s(y)*	0.1748	0.1650
Validation Error *s(e)/s(y)*	1.8705	1.8938

The previous example showed that the one sigmoid hidden node FNN was insufficient to model the third order MPR. This network lacked the number of effective parameters required for the model. To increase the number of parameters, a two hidden node network was then tested (Figure 4.15). The output, *Y*, of this network can be calculated by:

$$Y = w_3 * tanh(b_1 + w_1 X) + w_4 * tanh(b_2 + w_2 X) + b_3 \quad (4.34)$$

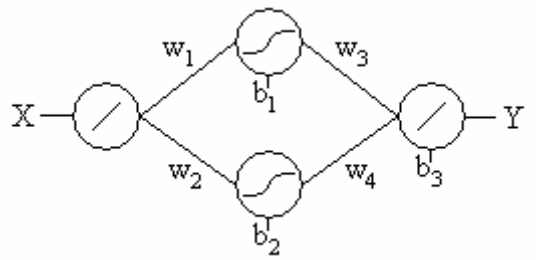


Figure 4.15: *Two hidden node FNN with sigmoid activation function.*

Using the third order Taylor series expansion (Equation 4.10) to estimate the hyperbolic tangent functions, the network equation is transformed into a polynomial equation and like terms are combined. Comparing term by term to equation (4.13), the following formal equations are found for the regression coefficients.

$$c_0 = w_3 b_1 + w_4 b_2 - \frac{1}{3} w_3 b_1^3 - \frac{1}{3} w_4 b_2^3 + b_3 \quad (4.35)$$

$$c_1 = w_3 w_1 + w_4 w_2 - w_3 w_1 b_1^2 - w_4 w_2 b_2^2 \quad (4.36)$$

$$c_2 = -w_3 w_1^2 b_1 - w_4 w_2^2 b_2 \quad (4.37)$$

$$c_3 = -\frac{1}{3} w_3 w_1^3 - \frac{1}{3} w_4 w_2^3 \quad (4.38)$$

The empirical results for the two hidden node FNN show that with the sigmoid activation function, the network is still able to accurately find the correct trend to the training data series (Figure 4.16). Both regression and the FNN produced similar training

error (Table 4.19). However, outside of the training range, the sigmoid network was not able to match the accuracy of the regression model. The network output function curves under the target function.

While the network seems to fit the training data with the same function as MPR, the regression coefficients derived from the formal equations (4.35) to (4.38) are not close to the actual coefficients at all (Table 4.21). The fact that all of the network derived coefficients are negative suggests that if they were put into a third order regression equation, it would not produce the correct function. The reason for the discrepancy is most likely due to the fact that the range of the input and output data was well outside of the effective range of the sigmoid activation function. This means that the data is also outside of the range where the third order Taylor series expansion of the hyperbolic tangent is accurate. In order to get more accurate regression parameter estimations, the data should be within the range of -1 to +1 at least.

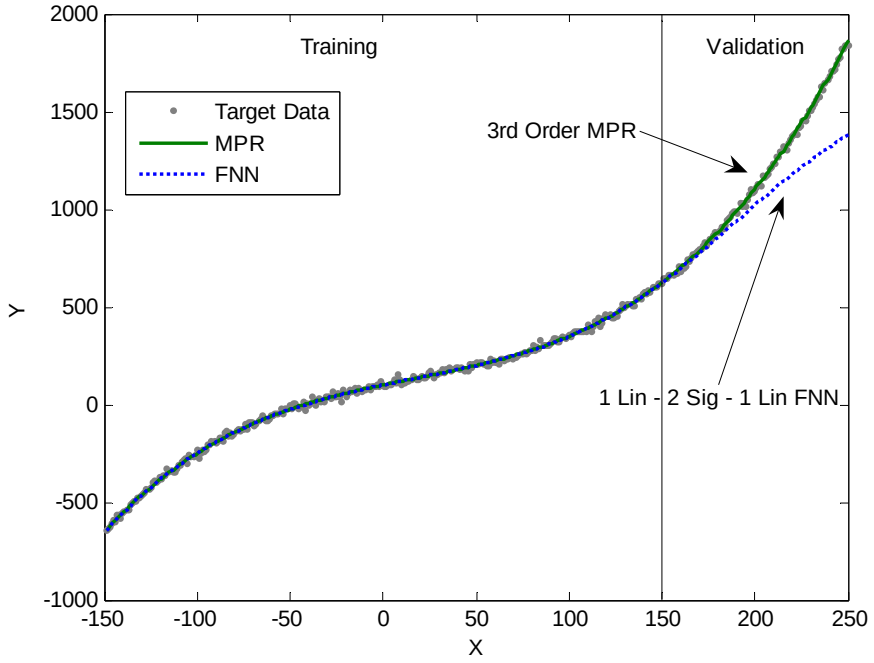


Figure 4.16: *Output function for two hidden node sigmoid FNN.*

Table 4.19: *Two Sigmoid Hidden Nodes - Prediction error from both models.*

	3rd Order MPR	1 Lin - 2 Sig - 1 Lin FNN
Training Error *s(e)/s(y)*	0.0394	0.0396
Validation Error *s(e)/s(y)*	0.0336	0.5522

Table 4.20: *Two Sigmoid Hidden Nodes - Trained network weights and biases.*

w_1	w_2	w_3	w_4	b_1	b_2	b_3
0.0105	-0.0079	779.9	-2033.1	-2.0049	-1.8755	-1087.3

Table 4.21: *Two Sigmoid Hidden Nodes - Regression coefficients found by sigmoid network.*

	Target	3rd Order MPR	1 Lin - 2 Sig - 1 Lin FNN
c_0	100	100.4401	-1213.9
c_1	2	1.9588	-65.1611
c_2	-0.005	-0.0050	-0.0656
c_3	0.0001	0.0001	-0.000635

However, even when the input and output data is scaled to the range of -0.1 to +0.1, it was not found to be possible for the sigmoid hidden layer feed-forward network to replicate the coefficients of the polynomial regression equation (Table 4.22 and 4.23). Network training with the scaled data appears to have the same problem as before with the non-scaled data. The biases of the sigmoid layer (b_1 and b_2) put the range of the input values into the sigmoid function outside of the effective range for the Taylor series expansion. It may also be difficult to compare the regression coefficients using scaled data anyway, because the coefficients are also in the scaled domain. With this particular regression equation and data range, the best the sigmoid activation function is able to do is match trend of the training data. The sigmoid is unable to be analytically converted into the correct polynomial function, which is why it incorrectly graphs the validation data (Figure 4.16).

Table 4.22: *Trained network weights and biases when data is scaled to the range -0.1 to +0.1.*

w_1	w_2	w_3	w_4	b_1	b_2	b_3
-21.6945	-9.5871	-0.0659	-0.9371	2.1625	-2.2231	-0.8356

Table 4.23: *Regression coefficients found when data is scaled (values in scaled domain).*

	3rd Order MPR	1 Lin - 2 Sig - 1 Lin FNN
c_0	0.0151	-2.1047
c_1	0.4746	-40.6751
c_2	-1.6823	-124.3833
c_3	53.7498	-499.6091

Like with the polynomial feed-forward network, the two sigmoid hidden node feed-forward network is modified by removing the weights and biases from the last layer (Figure 4.17). This structure produces the network output function:

$$Y = tanh(b_1 + w_1X) + tanh(b_2 + w_2X) \tag{4.39}$$

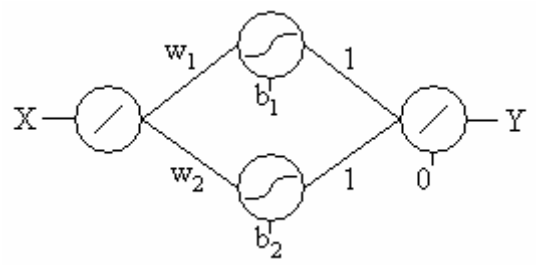


Figure 4.17: *Modified two sigmoid hidden node FNN with parameters w_3, w_4 and b_3 ignored.*

The equivalent regression coefficients based on this network are determined like before by using Taylor series expansion, resulting in equations (4.40) to (4.43).

$$c_0 = b_1 + b_2 - \frac{1}{3}b_1^3 - \frac{1}{3}b_2^3 \tag{4.40}$$

$$c_1 = w_1 + w_2 - w_1b_1^2 - w_2b_2^2 \tag{4.41}$$

$$c_2 = -w_1^2b_1 - w_2^2b_2 \tag{4.42}$$

$$c_3 = -\frac{1}{3}w_1^3 - \frac{1}{3}w_2^3 \tag{4.43}$$

The empirical results show that the modified sigmoid feed-forward network is not able to model the third order polynomial target data (Figure 4.18). The output range for the network is limited to -1 to +1, which is the output range for the hyperbolic tangent function. In order to model this particular target data, the FNN requires the extra weights and bias. As discussed before, it is apparent that these extra parameters serve the purpose of scaling and offsetting the output from the hidden layer. While these weights and biases were unnecessary in the polynomial FNN, they are important in the sigmoid FNN.

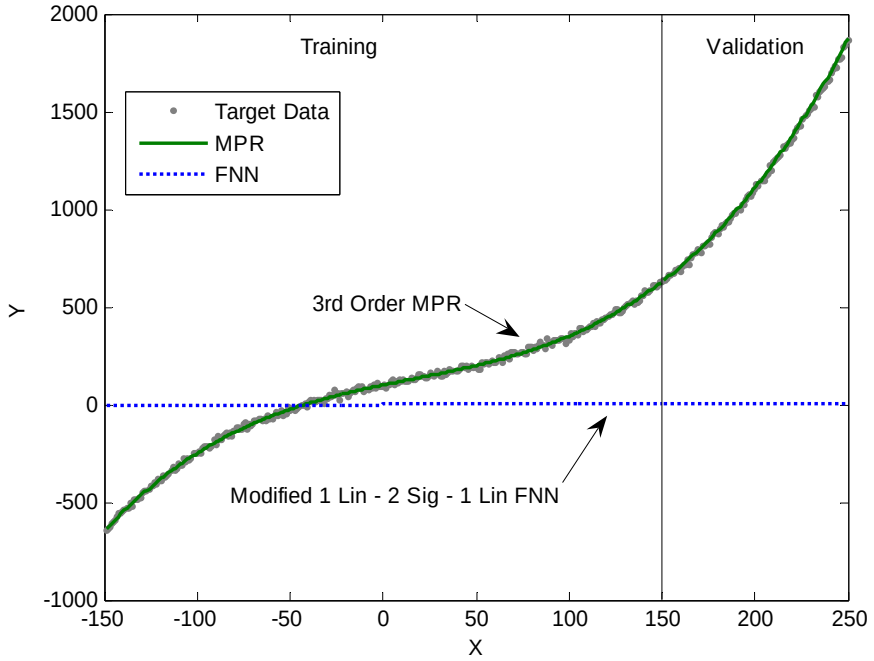


Figure 4.18: *The output functions from third order regression and modified two sigmoid hidden node FNN.*

Table 4.24: *Modified Two Sigmoid Hidden Nodes - Prediction error from both models.*

	3rd Order MPR	Modified 1 Lin - 2 Sig - 1 Lin FNN
Training Error *s(e)/s(y)*	0.0355	1.0237
Validation Error *s(e)/s(y)*	0.0423	3.4436

Table 4.25: *Modified Two Sigmoid Hidden Nodes - Trained network weights and biases.*

w_1	w_2	w_3	w_4	b_1	b_2	b_3
3.8090	0.1142	1	1	-1.7213	4.5599	0

Table 4.26: *Modified Two Sigmoid Hidden Nodes - Regression coefficients found by both models.*

	Target	3rd Order MPR	Modified 1 Lin - 2 Sig - 1 Lin FNN
c_0	100	99.9237	-27.0663
c_1	2	1.9866	-9.7355
c_2	-0.005	-0.0049	24.9138
c_3	0.0001	0.0001	-18.4218

The results have indicated that two hidden nodes are sufficient to model a third order polynomial equation. However, the range of the data tested in the empirical studies was shown to be poor for predicting the polynomial regression coefficients. This also led to poor prediction in the validation stage. In order to get a better understanding of the formal equations developed from Taylor series expansion, a new empirical equation (4.44), with the values of X spanning from -1 to +1 was tested. Values from 1 to 1.5 were used for validation.

$$Y = 0.1 - 0.4 * X + 0.3 * X^2 + 1 * X^3 \quad (4.44)$$

After many trials, the FNN was able to replicate the target function, as seen in Figure 4.19. While the FNN was not able to exactly match the MPR equation, it produced relatively small error (Table 4.27). Not only was the two hidden node FNN able to replicate the MPR output function, it was also able to predict the regression coefficients with some accuracy (Table 4.28 and 4.29). The slight differences are likely due to the data range being from -1 to +1. The coefficients would be more accurate if a smaller range, such as -0.1 to +0.1, was used. As with the previous test with the sigmoid activation function, the validation error for the FNN is slightly worse than for the regression equation. This error is likely due to the imperfect prediction of the regression coefficients by the sigmoid FNN.

These results conflict with the results from Xiang et al. (2005), who determine that a minimum of three hidden nodes is required by a FNN (Linear - Sigmoid - Linear activation functions) to approximate a third order, one input variable system. The target function used in their paper was similar to equation (4.44), with the same values for the independent variable X. As seen in Figure 4.19, this function can be represented by three

linear segments, which is the primary reason they give for using three hidden nodes. However, it is shown here that two hidden nodes are sufficient for modeling this MPR equation, disputing the conclusions of Xiang et al. (2005).

These results show that a 1 Linear - 2 Sigmoid - 1 Linear FNN has the minimum number of hidden nodes required for modeling a third order regression equation. Taylor series expansion on the nonlinear hyperbolic tangent activation function has shown that there is a formal equivalence between the two models. However, the effectiveness of the Taylor expansion is limited by the range of the data. Input data series outside of the range of ±1 can still be effectively modeled by a sigmoid FNN, but the transformation of parameters to an equivalent MPR equation will not be accurate.

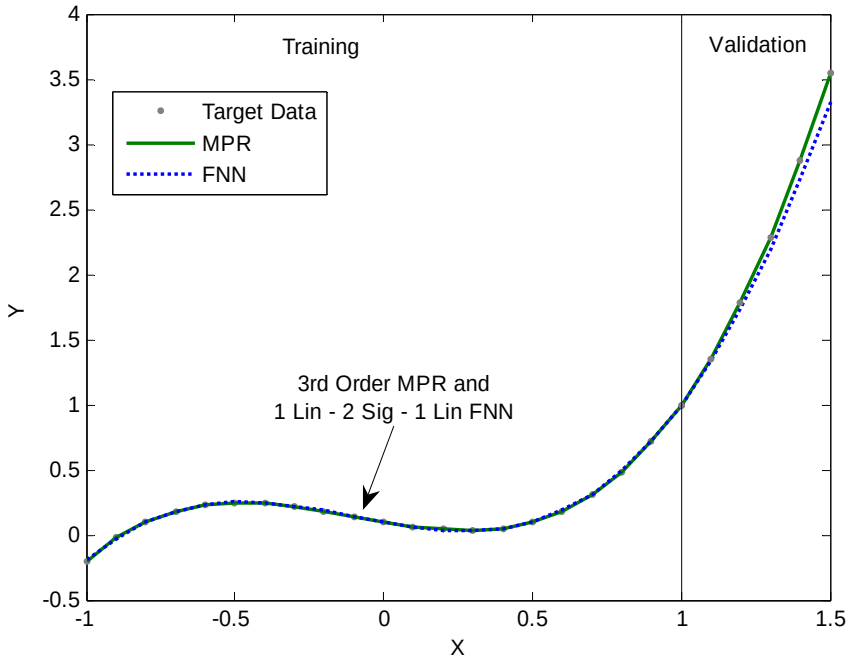


Figure 4.19: *Output function for sigmoid FNN for function with smaller range.*

Table 4.27: *Two Sigmoid Nodes (Smaller Range) - Prediction error from both models.*

	3rd Order MPR	1 Lin - 2 Sig - 1 Lin FNN
Training Error *s(e)/s(y)*	$1.0570 * 10^{-15}$	0.0118
Validation Error *s(e)/s(y)*	$3.4901 * 10^{-16}$	0.1416

Table 4.28: *Two Sigmoid Nodes (Smaller Range) - Trained network weights and biases.*

w_1	w_2	w_3	w_4	b_1	b_2	b_3
-0.1545	0.3753	-176.491	-73.2241	0.0884	-0.0067	15.1664

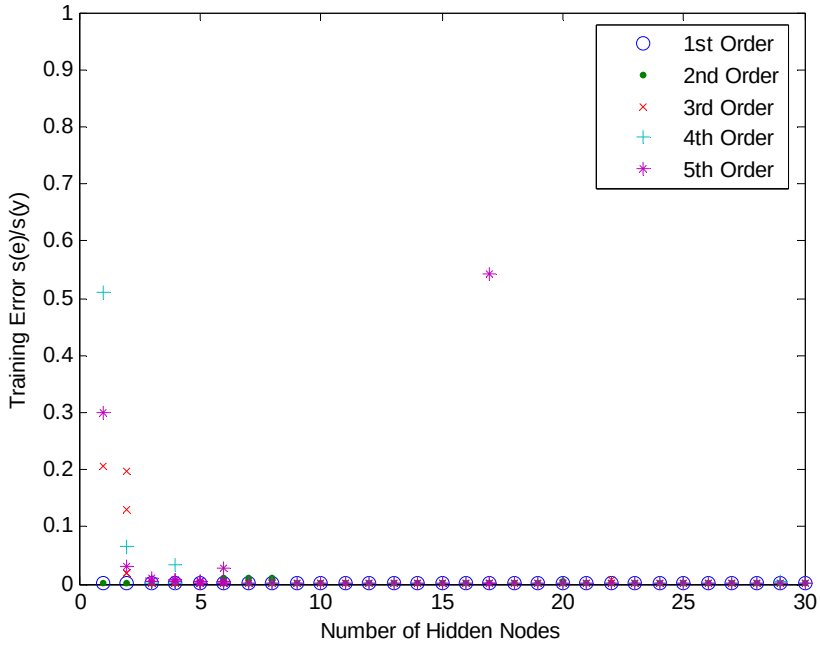
Table 4.29: *Two Sigmoid Nodes (Smaller Range) - Regression coefficients found by sigmoid network.*

	Target	3rd Order MPR	1 Lin - 2 Sig - 1 Lin FNN
c_0	0.1	0.1000	0.1002
c_1	-0.4	-0.4000	-0.4191
c_2	0.3	0.3000	0.3036
c_3	1	1.0000	1.0735

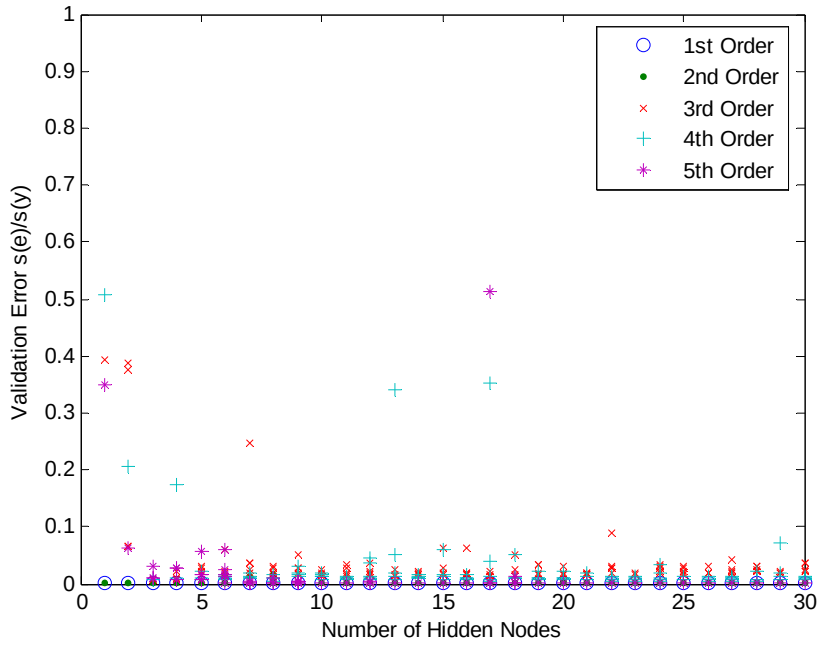
4.1.3.2 Generalization to MPR

The results from using a sigmoid activation function to model a third order, one variable MPR was generalized for different orders and more input variables. The number of hidden nodes was varied for each combination of regression order and number of input variables, similar to what was done with the polynomial activation function. These results were then compared to Table 4.17, which was generated using polynomial hidden nodes. Using the sigmoid activation function better represents the modeling of nonlinear systems than a simple polynomial activation function and is more common in ANN research and application.

In general, the results from these tests confirm the values for the critical, or minimum, number of hidden nodes required to reproduce a MPR of given order and input variables determined in the previous section (Table 4.17). For example, for a one input system, both first order and second order MPRs are modeled at near zero error with only one hidden node (Figure 4.20). The third, fourth and fifth orders, however, are not successfully modeled with only one hidden node, seen by their *s(e)/s(y)* values over 0.2. Once three hidden nodes are used, all five orders are being modeled efficiently by the FNN.



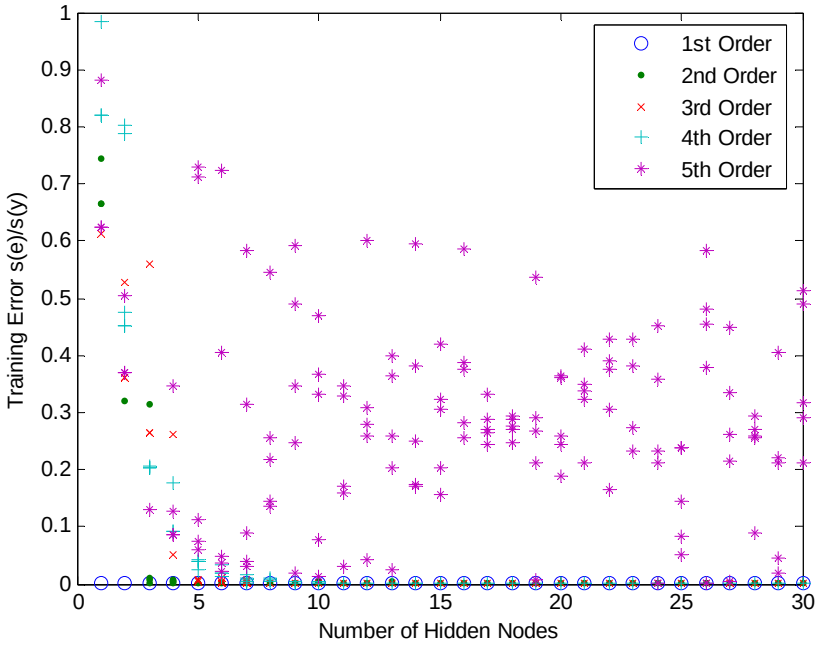
a)



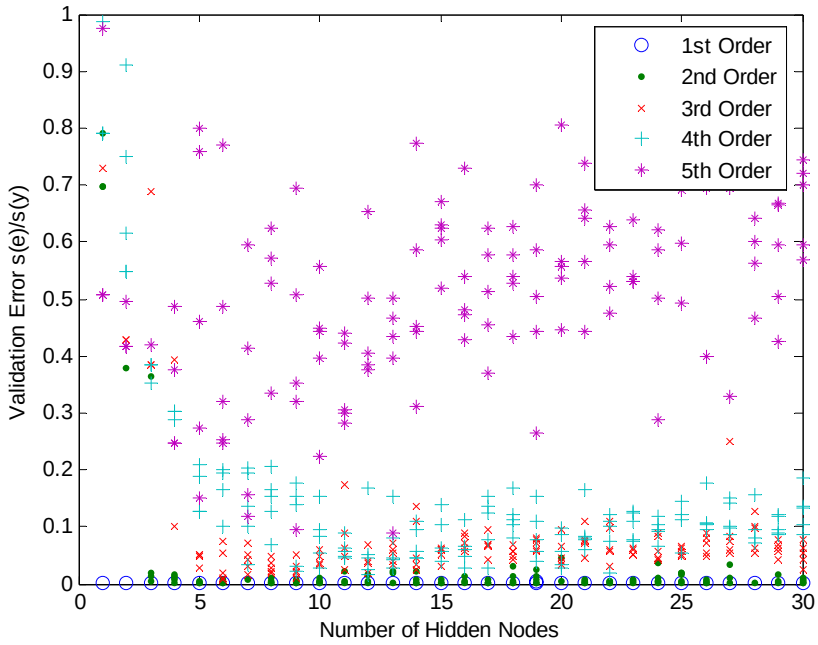
b)

Figure 4.20: *Training (a) and validation (b) error for modeling one variable, orders one through five.*

Similar results can be seen when modeling equations with multiple input variables. With two inputs (Figure 4.21) and three inputs (Figure 4.22), the differences between the orders of the regression equation as well as the difference between the training and validation error are easier to see. However, the minimum number of hidden nodes with the sigmoid activation function required to replicate the MPR appear to be slightly different than the values determined in the previous section with the polynomial activation function (Table 4.17). For example, with two input variables (Figure 4.21), the ideal number of hidden nodes is one for first order, three for second, five for third, eight for fourth, and approximately ten for fifth. With the polynomial function, the ideal number of nodes found was one for first order, two for second, three for third, five for fourth and eight for fifth. The reason for the discrepancy is likely due to the limited number of trials and the inconsistency that was observed before when using artificial neural networks in terms of the random initiation of network parameters. In spite of the discrepancies, it is clear that the order and structure of the underlying polynomial regression equation has a strong effect on the optimal network structure.

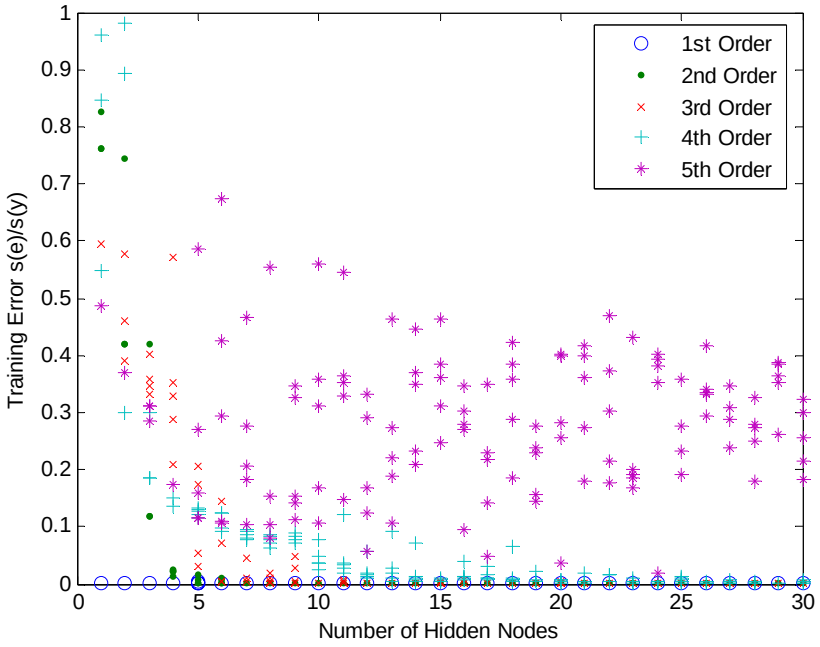


a)

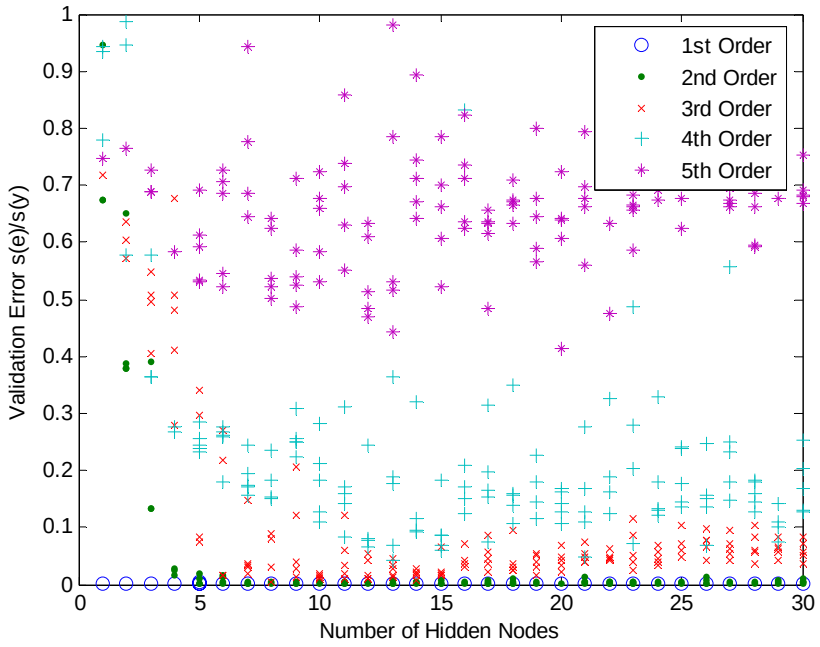


b)

Figure 4.21: *Training (a) and validation (b) error for modeling two variables, orders one through five.*



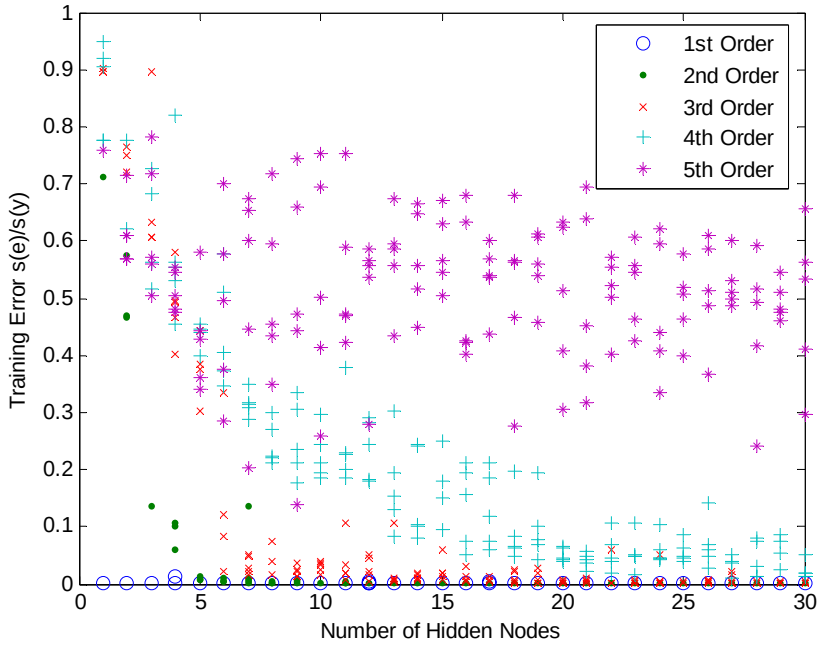
a)



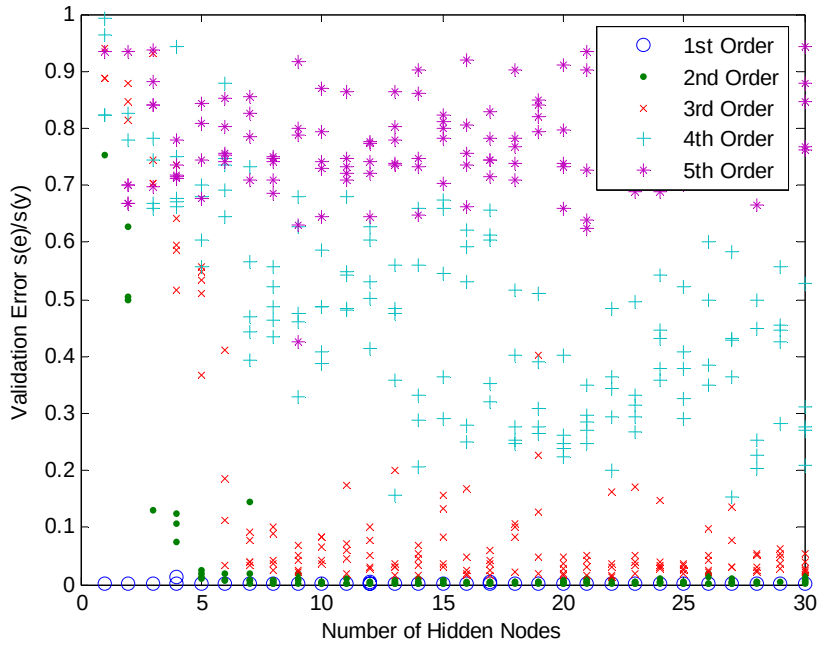
b)

Figure 4.22: *Training (a) and validation (b) error for modeling three variables, orders one through five.*

However, one of the problems noticed was when modeling higher-order equations with multiple input variables, most notably fifth order polynomials. Both training error and validation error was poor when modeling these high order equations no matter how many hidden nodes were added. This phenomenon is observed best with a large number of input variables, such as with four inputs (Figure 4.23) and five inputs (Figure 4.24). Graphing the error versus the number of hidden nodes does not produce the same clear trend that can be observed with lower ordered polynomials. The training error decreases slowly with the addition of new hidden nodes and the validation error is not affected by the number of hidden nodes at all. In fact, the trend shows the validation error generally increasing as the number of hidden nodes increases, with no obvious minimum value to select for the optimal number of nodes. It is also important to note that the performance of the FNN varies greatly from trial to trial, resulting in a large deviation and spread in the trend. This indicates that local minima are impeding the back-propagation training process and it is difficult for the network to find the global minimum error.

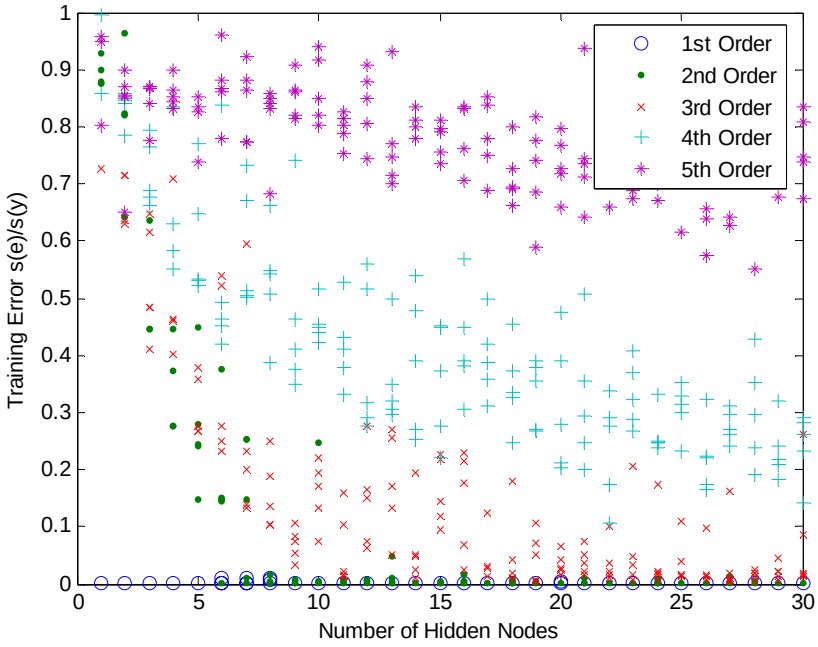


a)

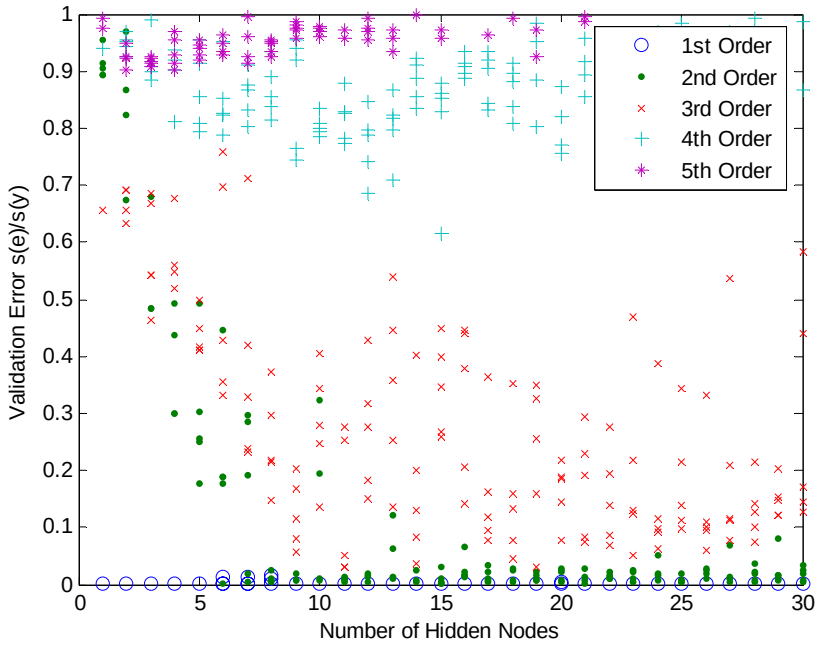


b)

Figure 4.23: *Training (a) and validation (b) error for modeling four variables, orders one through five.*



a)



b)

Figure 4.24: *Training (a) and validation (b) error for modeling five variables, orders one through five.*

An interesting aspect of finding the optimal number of hidden nodes is the relationship between the training error and the validation error. Specifically, it is how these values relate to the number of hidden nodes in the network. To investigate this further, another series of tests were performed on a three input system of various orders. This time, the sample size *n* was decreased to two-hundred and fifty and normally distributed random noise was added to both the training and validation error. The results give a good illustration of how the order of the underlying system's equation affects a FNN's performance (Figure 4.25).

As the order of the MPR describing the system of data increases and more terms and parameters are added to the system, then the FNN requires more hidden nodes to represent the system. Also, the point at which the training error drops to its minimal value corresponds to the critical number of hidden nodes determined in the previous section (Table 4.17). One hidden node for first order, three for second, five for third, and nine for fourth. It is harder to find this point for the fifth order polynomial.

It is important to note that in all of these cases, the number of input variables, output variables, and training sample size remain the same. The only difference is the order of the physical polynomial equation that defines the relationship between the data. This indicates that the underlying physical equation of the data plays a large role in determining the optimal neural network structure. This could be a useful method for implementing ANNs for modeling unknown biological systems. By graphing the error versus the number of hidden nodes, the trend can help the modeler determine the best order of polynomial regression to use to represent the system.

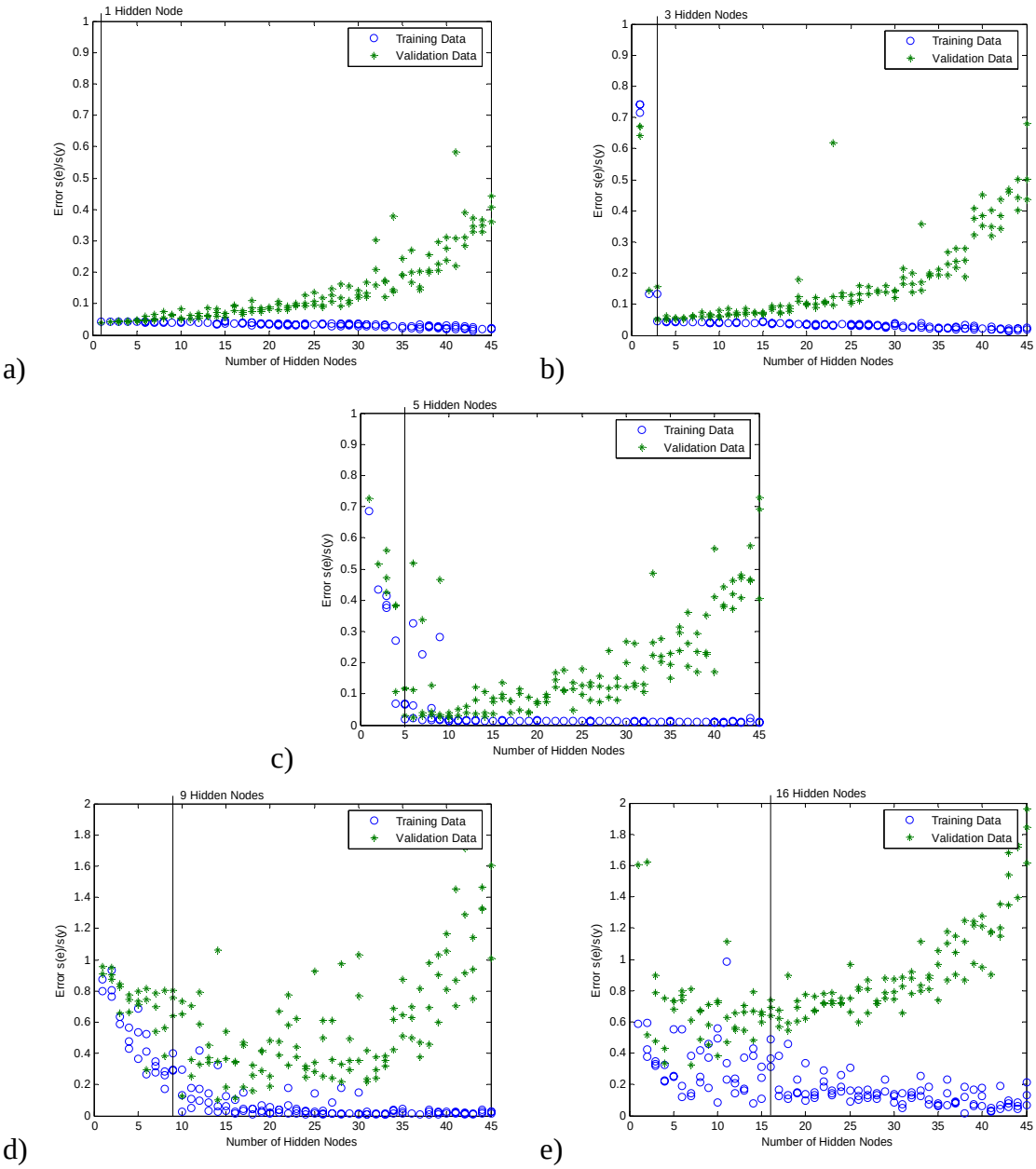


Figure 4.25: *Modeling a three variable MPR with noise and smaller sample size. (a) 1st Order (b) 2nd Order (c) 3rd Order (d) 4th Order (e) 5th Order.*

The general trend found for all of the regression equations being modeled is that the training error of the network begins very high (indicating poor performance). It then decreases quickly until the critical, or minimum, number of hidden nodes is in the network. At this point, the training error decreases slowly but does not change very much.

The validation also follows an interesting trend as the number of hidden nodes varies. Like the training error, the validation error shows poor model accuracy before the required minimum number of hidden nodes is in the network. However, after this point, the validation error starts to increase again. This is due to the FNN memorizing the training set with the extra parameters, resulting in poor generalization. The validation error trend seems to have its minimum value around the point of the critical number of hidden nodes, the point at which the network has at least as many parameters as the underlying regression equation. These results emphasize that great care should be taken when selecting the number of hidden nodes for an ANN. Blindly adding a lot of neurons will only reduce the model's performance. The results indicate that prior knowledge of the system (in terms of the number of input variables and polynomial order) can be used to help determine the number of hidden nodes to include in the network.

As mentioned before, network performance was reduced dramatically when a high order polynomial, such as fifth order, was being modeled. To attempt to improve model accuracy, the tests were repeated for the fifth order polynomial using two sigmoid hidden layers. Feed-forward networks were trained with various numbers of hidden nodes in the second hidden layer with mixed results (Figure 4.26). It was found that adding a second layer could help improve the accuracy of the model for both the training

and validation data. The best network, with the lowest validation error, was found with only using one hidden node in the second layer, around twenty-eight nodes (Figure 4.26b). Adding more nodes to the second layer did not appear to improve the network much more than the original three layer network. However, there was a large amount of variation between the different trials, which means that the issue of local minima is still a problem. Also, because a second sigmoid layer was added to the network, it is more difficult to determine an equivalent MPR equation as was done with only one sigmoid layer. A much higher ordered polynomial equation would be necessary to replicate the network equation produced by these more complex neural networks.

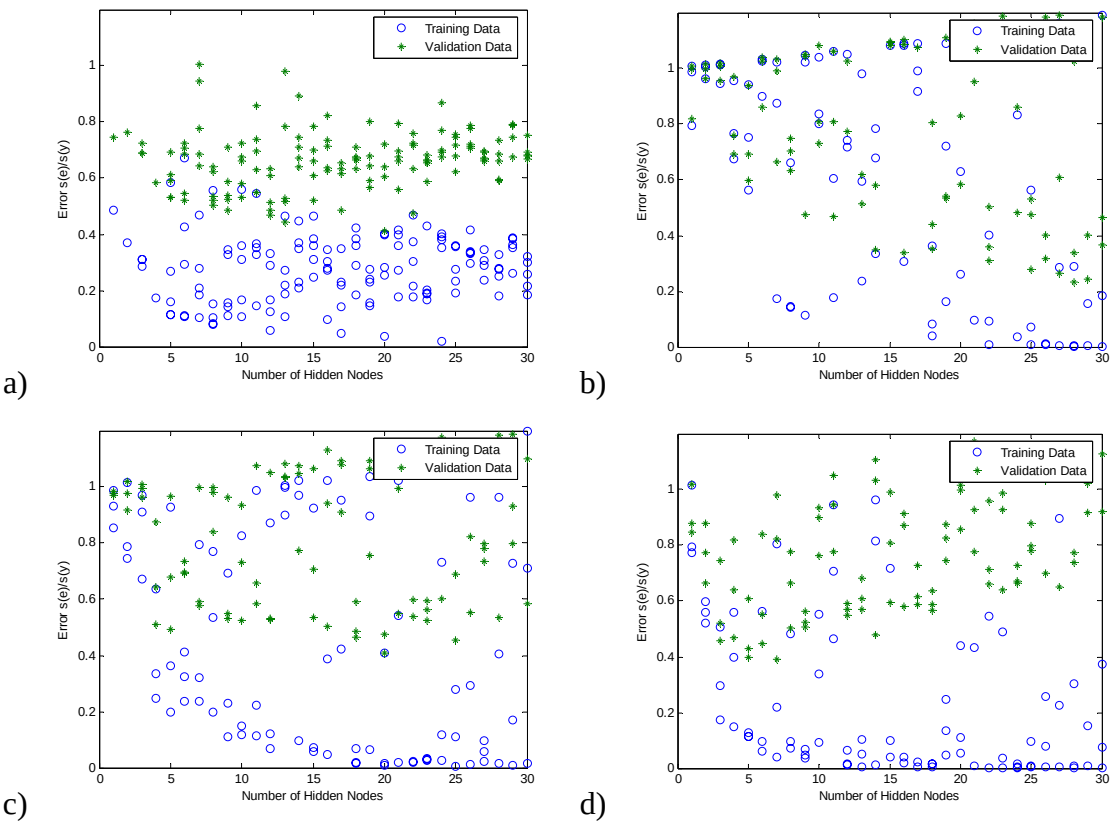


Figure 4.26: *Modeling three variable, fifth order MPR with different model structures: (a) Lin - Sig - Lin (b) Lin - Sig - 1 Sig - Lin (c) Lin - Sig - 2 Sig - Lin (d) Lin - Sig - 5 Sig - Lin.*

4.2 Equivalence of RNN and ARMA

4.2.1 Perceptron-level Analysis

The research will now move into investigating models for time dependant data. For this section, the perceptron will include a connection going back to itself, making it a recurrent perceptron (Figure 4.27). The output for Y using this perceptron can be defined by the equation:

$$Y_t = b + w_1 * X_t + w_2 * Y_{t-1} \tag{4.45}$$

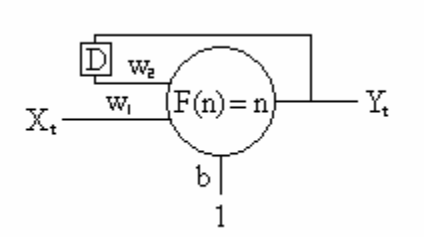


Figure 4.27: *A single recurrent perceptron.*

The output equation for this perceptron is similar to the equation for a ARMA(1,1) model (Equation 4.41). This equation uses one recurrent time step of the previous output for the predicted variable, Y_{t-1}, and an independent input, X_t.

$$Y_t = c_0 + c_1 * X_t + c_2 * Y_{t-1} \tag{4.46}$$

Comparing (4.45) to (4.46) produces the trivial results:

$$c_0 = b \tag{4.47}$$

$$c_1 = w_1 \tag{4.48}$$

$$c_2 = w_2 \tag{4.49}$$

Now that formal equations have been found to relate the recurrent perceptron parameters to ARMA(1,1) parameters, the models can be compared empirically. Both models were trained to estimate a stable synthetic time series defined by equation (4.50). Where X_t is a random normal variable and the initial value for Y_{t-1} is set to zero. The

values for Y_t are then iteratively calculated for the first three-hundred time steps to create the training data.

$$Y_t = -1.5 + 5 * X_t + 0.5 * Y_{t-1} \quad (4.50)$$

Both the recurrent perceptron and an ARMA(1,1) equation were trained to estimate the target equation. Also, both models used full, multiple-day-ahead, prediction. The empirical results from using this equation are very favorable for the recurrent perceptron. As Figure 4.28 shows, the perceptron was easily able to replicate the time series with the same degree of accuracy as the ARMA model. The recurrent network produces an output that is slightly less accurate than the ARMA model. Also, the perceptron successfully predicted the regression coefficients as shown in Table 4.31. This demonstrates that a linear recurrent perceptron is strongly comparable to linear ARMA equations.

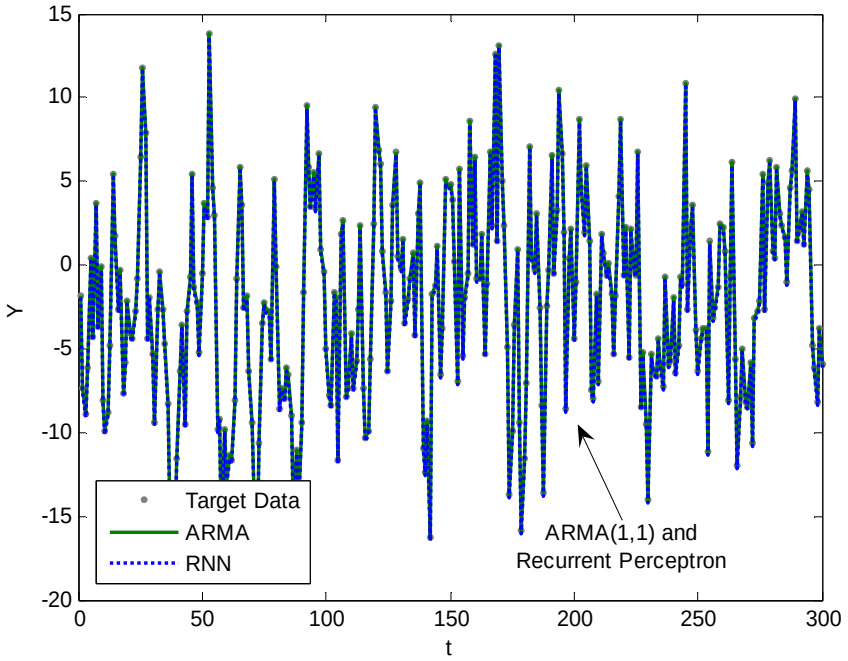


Figure 4.28: *Output from the recurrent perceptron when modeling a stable equation.*

Table 4.30: *Recurrent Perceptron (Stable) - Prediction error from both models.*

	ARMA(1,1)	Recurrent Perceptron
Training Error *s(e)/s(y)*	$3.7151 * 10^{-16}$	0.0295

Table 4.31: *Recurrent Perceptron (Stable) - Time series coefficients found by both models.*

	Target	ARMA(1,1)	Recurrent Perceptron
c_0	-1.5	-1.5000	-1.5959
c_1	5	5.0000	4.9997
c_2	0.5	0.5000	0.5000

The test was performed again with a second synthetic time series equation (4.51). This time, an unstable target equation was chosen to train the models.

$$Y_t = -1.5 + 5 * X_t + 1 * Y_{t-1} \quad (4.51)$$

Results indicate that while ARMA(1,1) was able to correctly estimate the parameters of the time series, the recurrent perceptron failed in all trials. In the first case, Figure 4.29, the recurrent perceptron was able to model the overall trend of the time series, but failed to correctly estimate the coefficient of the independent variable, c_1 (Table 4.33). In the second case, Figure 4.30, the perceptron did better at estimating c_1, but failed to find the correct value for c_2 (Table 4.35), which is why the model did not fit the overall data trend.

It is evident that there is some limitation with the perceptron that prevents it from finding the correct output function, while the auto-regressive moving average function is easily able to model the correct equation simply by using the least-squares method. Perhaps the problem is in the iterative approach of back-propagation. Another potential explanation for the discrepancy is that the single recurrent perceptron model may not have enough network parameters to replicate the ARMA. In the previous section, when comparing FNNs with MPR, it was found that neural networks tend to require more parameters than regression equation. A similar limitation may exist for recurrent networks and perhaps a more complex RNN, with multiple layers, could perform better. However, due to the fact that the time series being estimated is linear, it does not seem likely that a more complex neural network would improve performance by much.

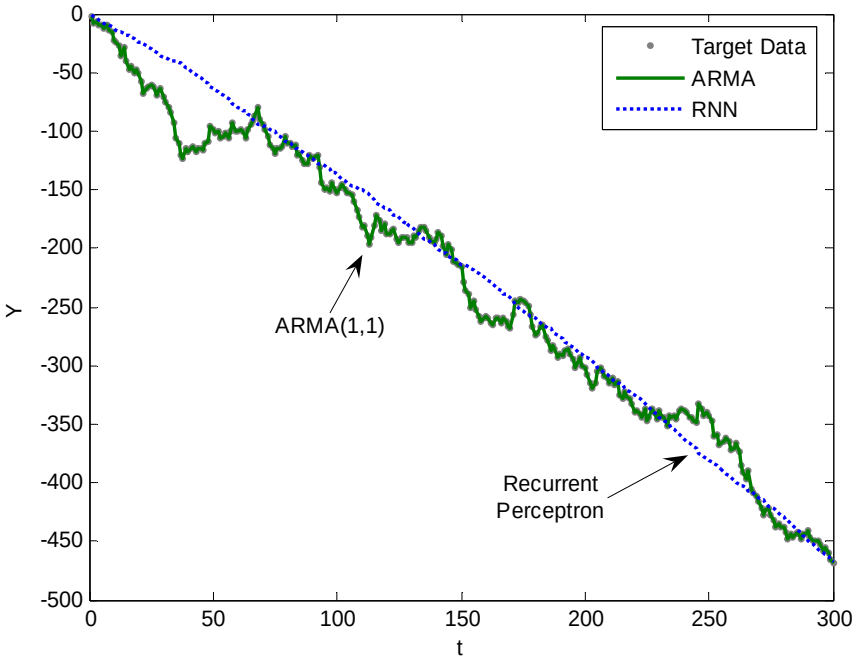


Figure 4.29: *First attempt output from the recurrent perceptron when modeling an unstable equation.*

Table 4.32: *Recurrent Perceptron (Unstable) - Prediction error from both models.*

	ARMA(1,1)	Recurrent Perceptron
Training Error *s(e)/s(y)*	$1.2877 * 10^{-14}$	0.0887

Table 4.33: *Recurrent Perceptron (Unstable) - Time series coefficients found by both models.*

	Target	ARMA(1,1)	Recurrent Perceptron
c_0	-1.5	-1.5000	-1.2921
c_1	5	5.0000	-0.5226
c_2	1	1.0000	1.0012

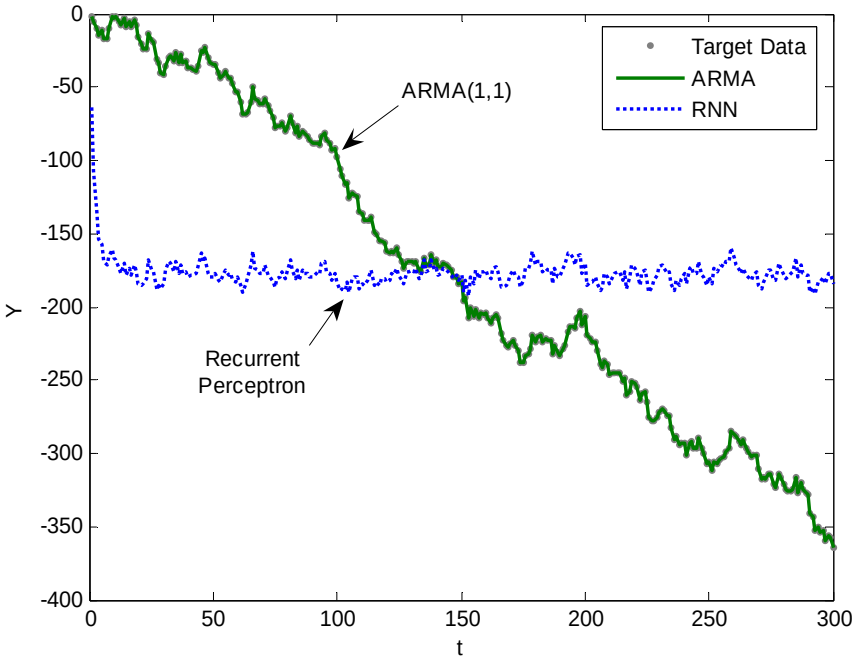


Figure 4.30: *Second attempt output from the recurrent perceptron when modeling an unstable equation.*

Table 4.34: *Recurrent Perceptron (Unstable) - Prediction error from both models.*

	ARMA(1,1)	Recurrent Perceptron
Training Error *s(e)/s(y)*	$3.5497 * 10^{-14}$	0.5132

Table 4.35: *Recurrent Perceptron (Unstable) - Time series coefficients found by both models.*

	Target	ARMA(1,1)	Recurrent Perceptron
c_0	-1.5	-1.5000	-63.4502
c_1	5	5.0000	5.0289
c_2	1	1.0000	0.6450

If the perceptron weights and bias are initialized with the correct values, then it is capable of modeling the synthetic time series (Figure 4.31). This shows that the recurrent perceptron does have the ability to represent the ARMA(1,1) equation if the parameters are right. As mentioned before, the problem probably exists somewhere in the training method of the perceptron. Another possible reason for the poor performance from the perceptron relates to the stability of the function being modeled. There is evidence in the literature to suggest that unstable equations are more difficult to model when using networks due to their similarity to linear filters (Mandic and Chambers, 2001).

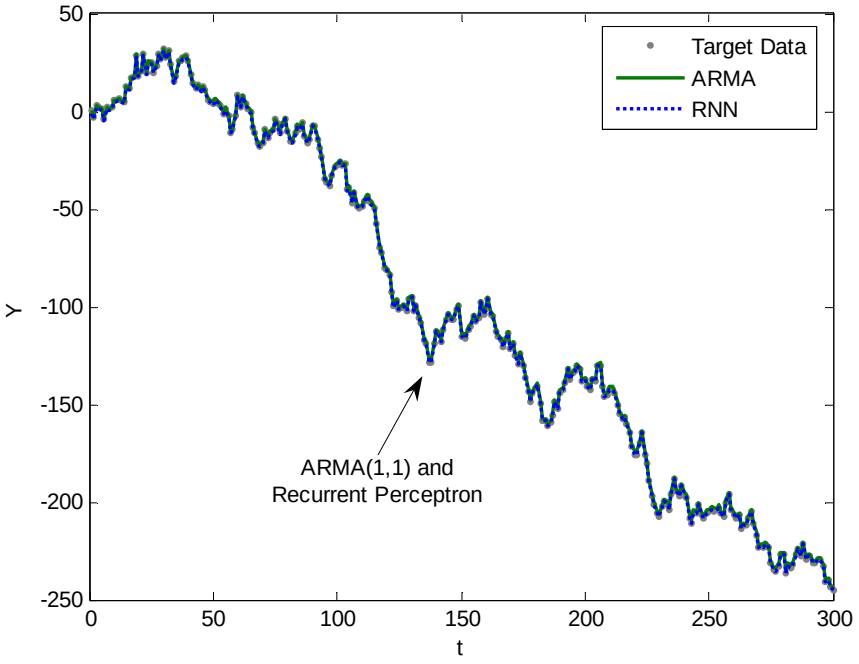


Figure 4.31: *The recurrent perceptron output with correct initial values.*

These results signify the potential equivalence between recurrent neural networks and ARMA models. However, the ability of a RNN to model a time series appears to depend on the stability of the underlying physical equation. Recurrent neural networks seem to have a more difficult time than classic statistical regression equations when modeling unstable equations, but the two models appear to be more on par with each other when the equation is stable.

In the previous tests, an independent variable, X_t, was used as an input to each of the models. Now, this variable will be replaced with the prediction error term E_t. This change modifies the recurrent perceptron from the one used before to the new one seen in Figure 4.32. The output function for this perceptron is defined by:

$$\hat{Y}_t = b + w_1 * \hat{Y}_{t-1} + w_2 * E_{t-1} = b + w_1 * \hat{Y}_{t-1} + w_2 * (\hat{Y}_{t-1} - Y_{t-1}) \qquad (4.52)$$

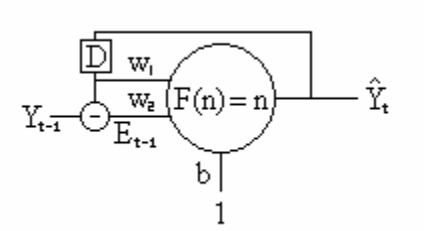


Figure 4.32: *A linear recurrent perceptron that includes an error term.*

As mentioned before, the equation for an ARMA(1,1) model is the same as the recurrent perceptron equation (4.52), so the parameters for both models are the same. One important difference between this setup and the previous one is that value for Y_{t-1} is required as an input to the models so the error term E_{t-1} can be calculated. Also, this equation does not use an independent variable for input. This means that instead of being multiple-day-ahead prediction models, they will only be able to predict one-day-ahead. Because of the error term, the ARMA(1,1) equation will use the long-AR method to estimate its parameters.

First, the models are trained to the stable synthetic equation (4.50). Interestingly, the recurrent perceptron seems to outperform ARMA(1,1) in replicating the time series. This data series shows a clear difference between the performances of the two models for estimating the coefficient to the error term (Figure 4.33). The ARMA model is unable to estimate the magnitude of the error as well as the perceptron (Table 4.37). This is evidence of a possible improvement of RNNs over ARMA equations. Both models have the same structure, but use different training methods to find the regression parameters. The iterative approach of back-propagation has an advantage over the more approximate form of least-squares in the long-AR method used by the auto-regressive moving average model.

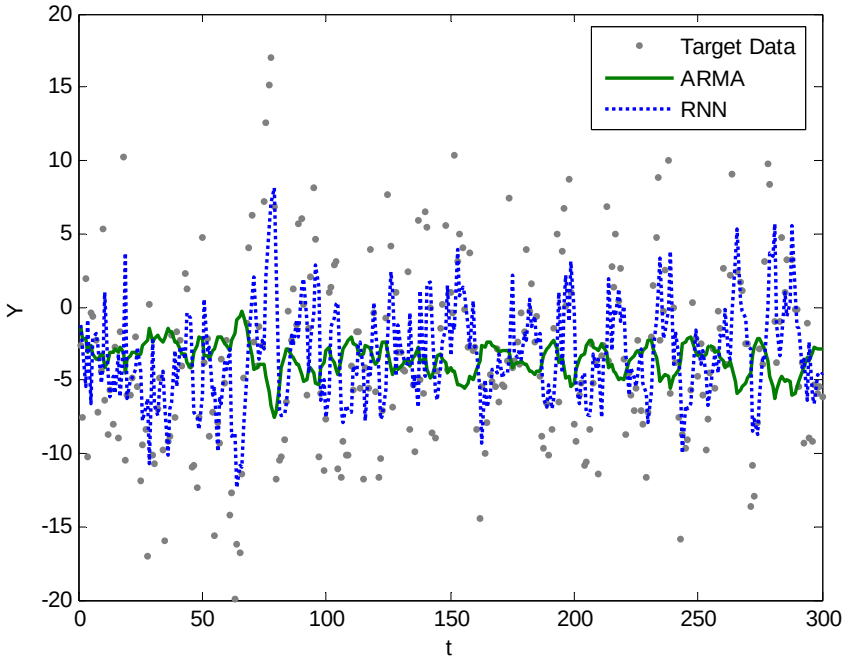


Figure 4.33: *Using the error term to estimate a stable equation.*

Table 4.36: *Recurrent Perceptron (Error Term, Stable) - Prediction error from both models.*

	ARMA(1,1)	Recurrent Perceptron
Training Error *s(e)/s(y)*	0.9632	0.7432

Table 4.37: *Recurrent Perceptron (Error Term, Stable) - Time series coefficients found by both models.*

	Target	ARMA(1,1)	Recurrent Perceptron
c_0	-1.5	-1.3980	-1.2806
c_1	5	0.0879	0.5136
c_2	0.5	0.6002	0.6127

Both time series model are then trained to estimate the unstable synthetic equation (4.51). The results from this test are similar to the last one, in that that recurrent perceptron performs more accurately than the ARMA(1,1) equation. The output of the perceptron closely follows the path of the time series, while the regressive model is only able to graph the overall trend (Figure 4.34). Looking at the parameters found by both equations on Table 4.39, both models do a decent job at estimating the coefficients for the auto-regressive term (c_2) and the bias (c_0). However, the ARMA model poorly estimates the coefficient for the error term, which explains why it is unable to fully predict the time series. The perceptron also does not match the correct error term parameter, but it is still better than the ARMA model. Once again, the training algorithm for the auto-regressive moving average equation does not appear to be adequate enough for estimating the error term, giving the back-propagation method used by recurrent neural networks an advantage.

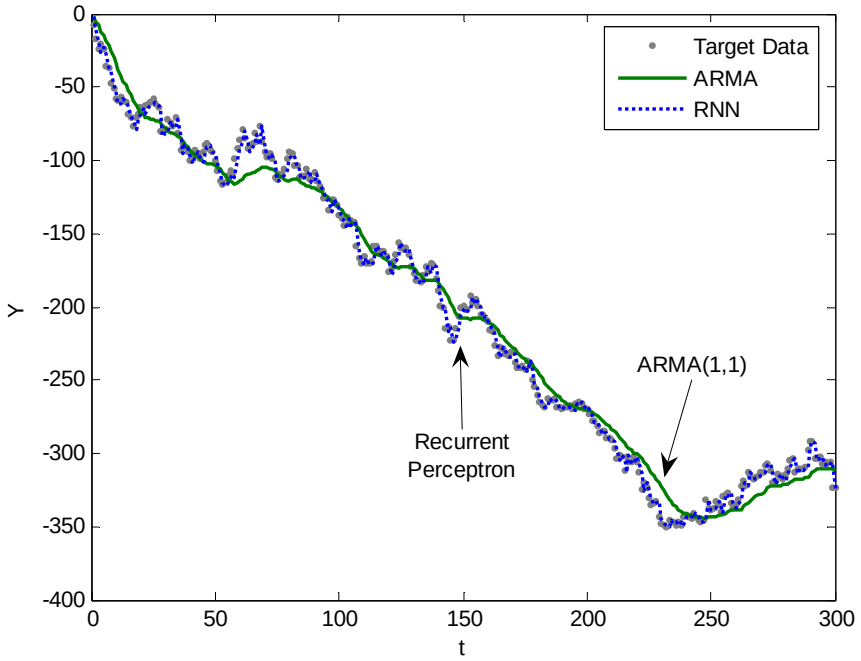


Figure 4.34: *ARMA(1,1) and recurrent perceptron using an error term as input for unstable data.*

Table 4.38: *Recurrent Perceptron (Error Term, Unstable) - Prediction error from both models.*

	ARMA(1,1)	Recurrent Perceptron
Training Error *s(e)/s(y)*	0.0506	0.0226

Table 4.39: *Recurrent Perceptron (Error Term, Unstable) - Time series coefficients found by both models.*

	Target	ARMA(1,1)	Recurrent Perceptron
c_0	-1.5	-2.0981	-2.3294
c_1	5	-0.1062	1.0850
c_2	1	0.9948	0.9939

For the last test with the recurrent perceptron, a higher order time series ARMA model will be replicated. This time, three previous time steps will be used as input, creating an ARMA(3,0) equation. The perceptron used is shown in Figure 4.35 and can be represented mathematically as:

$$\hat{Y}_t = b + w_1 * \hat{Y}_{t-1} + w_2 * \hat{Y}_{t-2} + w_3 * \hat{Y}_{t-3} \quad (4.53)$$

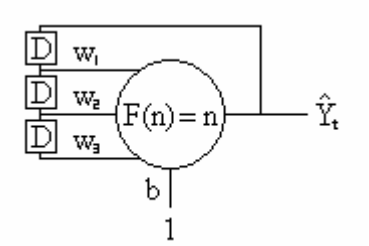


Figure 4.35: *A linear recurrent perceptron that goes back three time steps.*

This equation is equivalent to ARMA(3,0). Since the equation is still linear, a single perceptron should be sufficient for modeling the equation.

The models were first trained to the synthetic equation (4.54). The value for the output Y is defined as a linear regression of the three previous time series values, without any error terms.

$$Y_t = 0.5 + 1 * Y_{t-1} + -0.7 * Y_{t-2} + 0.45 * Y_{t-3} \quad (4.54)$$

The results show that both models have reasonable prediction accuracy for the time series ($s(e)/s(y)$ = 0.0533 for RNN and 0.0144 for ARMA) (Figure 4.36). Both models also stabilize as the data series reaches an asymptote around two. The ARMA model seems to have an advantage over the perceptron in terms of accuracy. Also, the parameters of the ARMA equation are much closer to the target values than the perceptron (Table 4.41).

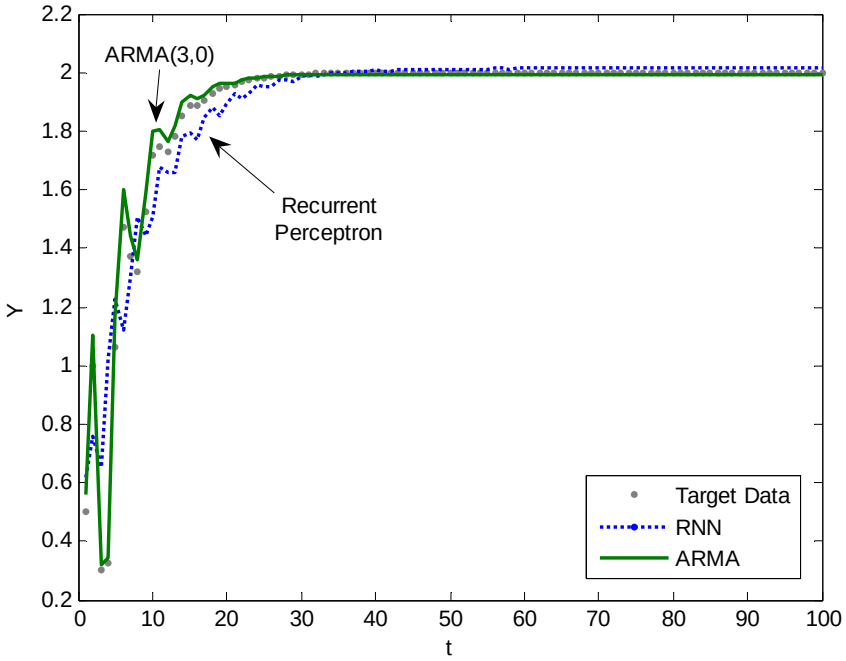


Figure 4.36: *ARMA(3,0) and recurrent perceptron output for first equation.*

Table 4.40: *Recurrent Perceptron (Three Time Steps) - Prediction error from both models.*

	ARMA(3,0)	Recurrent Perceptron
Training Error *s(e)/s(y)*	0.0144	0.0533

Table 4.41: *Recurrent Perceptron (Three Time Steps) - Time series coefficients found by both models.*

	Target	ARMA(3,0)	Recurrent Perceptron
c_0	0.5	0.5601	0.6170
c_1	1	0.9737	0.2311
c_2	-0.7	-0.7107	-0.2222
c_3	0.45	0.4562	0.6847

A second equation (4.55) was also tested with the recurrent perceptron and ARMA(3,0) models. This equation is similar to the previous one, except that a different coefficient is used for the first auto-regressive term.

$$Y_t = 0.5 + 0.05 * Y_{t-1} + -0.7 * Y_{t-2} + 0.45 * Y_{t-3} \quad (4.55)$$

The results with this equation are similar to the previous one in that both models are successful in producing a reasonable estimation of the time series. However, this time, the performance of the two models has been reversed. For this time series, the recurrent perceptron produces a lower error ($s(e)/s(y)$ = 0.0101) than the auto-regressive model (0.0376) (Figure 4.37). Also, the parameters of the recurrent perceptron are much closer to the actual values of the target equation compared to the coefficients found by the regression model (Table 4.43).

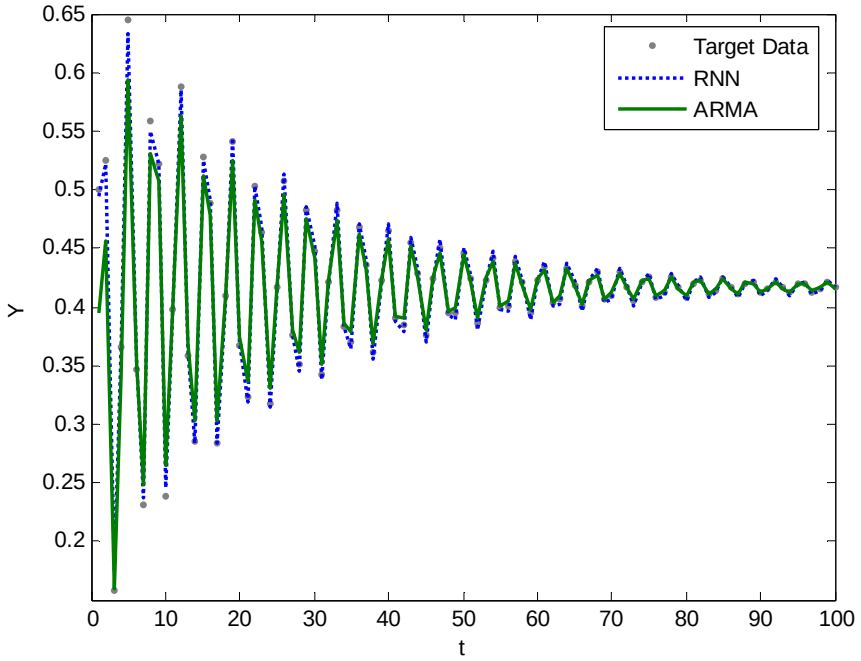


Figure 4.37: *ARMA(3,0) and recurrent perceptron output for second equation.*

Table 4.42: *Recurrent Perceptron (Three Time Steps) - Prediction error from both models.*

	ARMA(3,0)	Recurrent Perceptron
Training Error *s(e)/s(y)*	0.0376	0.0101

Table 4.43: *Recurrent Perceptron (Three Time Steps) - Time series coefficients found by both models.*

	Target	ARMA(3,0)	Recurrent Perceptron
c_0	0.5	0.3950	0.4946
c_1	0.05	0.1534	0.0566
c_2	-0.7	-0.6511	-0.7052
c_3	0.45	0.5504	0.4614

The results from these experiments indicate that just a single recurrent perceptron has the power to potentially model many different complex linear auto-regressive moving average equations. However, the ability of the perceptron to estimate a given time series as well as its ultimate performance appears to be highly dependant on the trend of the data being modeled. For example, the perceptron seems to respond differently depending on whether the data is stable or unstable.

4.2.2 Network-level Analysis

Up until this point, only linear ARMA time series equations have been examined with relation to recurrent neural networks. However, the auto-regressive moving average model can also be structured as nonlinear. This nonlinear ARMA, or NARMA, equation allows the regression model to achieve a higher degree of complexity.

A NARMA(1,0) equation can be represented by a RNN with one recurrent time step and a sigmoid activation function in the hidden layer (Figure 4.38). This network is similar to the one input, two hidden node network discussed in the previous section. Except in this case there are no independent variables. This recurrent network is represented by the equation:

$$Y_t = w_3 * \tanh(b_1 + w_1 Y_{t-1}) + w_4 * \tanh(b_2 + w_2 Y_{t-1}) + b_3 \qquad (4.56)$$

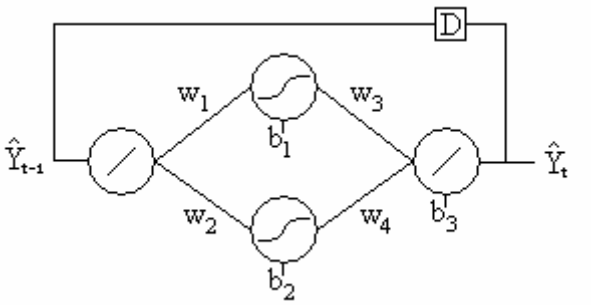


Figure 4.38: *A sigmoid hidden layer RNN to replicate an NARMA(1,0) equation.*

If a third order Taylor series expansion is used on the hyperbolic tangent function in (4.56), then this equation can be reduced to a third order NARMA(1,0) equation (4.57).

$$Y_t = c_0 + c_1 * Y_{t-1} + c_2 * Y_{t-1}^2 + c_3 * Y_{t-1}^3 \quad (4.57)$$

Just like in the non-recursive case in the previous section, the coefficients in the regression model can be defined in terms of the network weights and biases using equations (4.35) to (4.38).

The empirical test was performed by training both models to a series of synthetic data defined by the equation:

$$Y_t = 0.6 - 1.2 * Y_{t-1} + 0.3 * Y_{t-1}^2 + 1 * Y_{t-1}^3 \quad (4.58)$$

Both models were able to replicate the time series output well, with the NARMA model estimating the correct parameters as expected (Figure 4.39). The RNN performed only slightly worse than the regression model (*s(e)/s(y)* = 0.0357). However, like with the sigmoid feed-forward network, the sigmoid recurrent network was not able to correctly predict the regression coefficients (Tables 4.45 and 4.46). Even though the target data is well within the range of -1 to +1, the weights and biases of the network (Table 4.45) make the network equation unsuitable for Taylor series expansion. This is a recurring problem that has been noticed with the use of the sigmoid activation function with ANNs and is a potential hindrance in developing efficient conversion equations for the parameters of both neural networks and statistical regression. More accurate coefficient predictions could be obtained if the weights and biases in the network going into the sigmoid hidden layer were restricted to values within the range of -1 to +1.

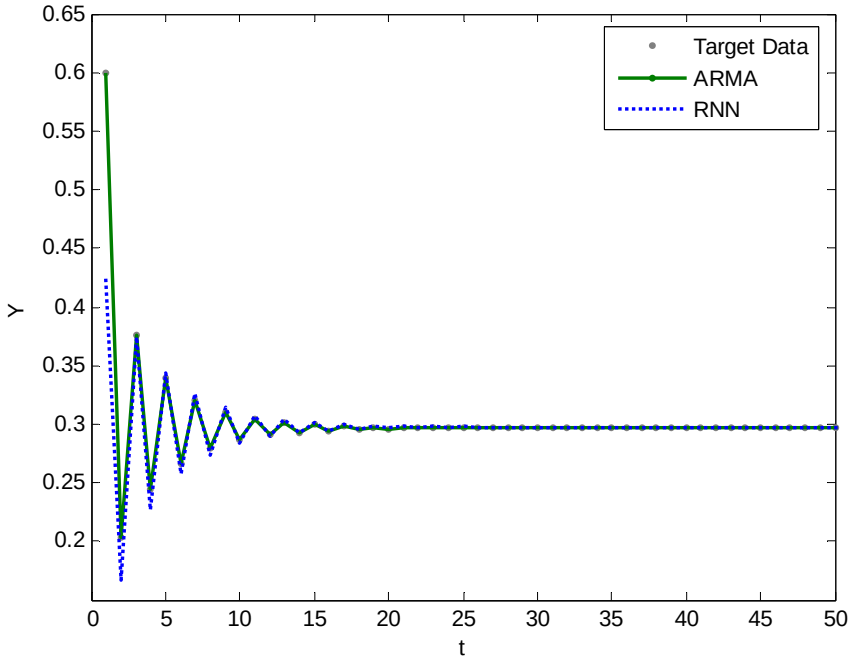


Figure 4.39: *Output function for two hidden node sigmoid RNN.*

Table 4.44: *Two Sigmoid Hidden Nodes (Recurrent) - Prediction error from both models.*

	3rd Order NARMA(1,0)	1 Lin - 2 Sig - 1 Lin RNN
Training Error *s(e)/s(y)*	$2.4345 * 10^{-15}$	0.0357

Table 4.45: *Two Sigmoid Hidden Nodes (Recurrent) - Trained network weights and biases.*

w_1	w_2	w_3	w_4	b_1	b_2	b_3
2.3238	-2.6067	-5.3878	8.0817	-2.7606	-11.2999	3.1603

Table 4.46: *Two Sigmoid Hidden Nodes (Recurrent) - Regression coefficients found by sigmoid network.*

	Target	3rd Order NARMA(1,0)	1 Lin - 2 Sig - 1 Lin RNN
c_0	0.6	0.6000	3775.9
c_1	-1.2	-1.2000	2751.8
c_2	0.3	0.3000	540.21
c_3	1	1.0000	70.252

4.3 Application to Biological Phenomena

The results from the previous two sections support the equivalence between artificial neural networks and statistical regression models. The tests that have been performed so far have been on highly controlled and specifically defined synthetic data. In order to get a better understanding of the practical application of both of these models, the results found before must be applied to real life data.

4.3.1 Confirming the Accuracy of Neural Networks

The results of Salas et al. (2000) were reproduced to get a better understanding of how neural networks and regression models are used for predicting events in real world problems. Salas et al. (2000) uses feed-forward neural networks to forecast streamflow. This paper uses a basic feed-forward network and sigmoid activation function to predict the daily streamflow of the Little Patuxent River, located in Maryland, using combinations of temperature, precipitation, evaporation, snow fall equivalent, and previous values of streamflow as input parameters.

The reproduced results coincided with the results found by Salas et al. (2000). The best-fit model determined by their research used precipitation (P_t and P_{t-1}) and temperature (T_t) to predict daily streamflow (Q_t). The network structure used was a Linear - Sigmoid - Sigmoid FNN with ten hidden nodes. Similar prediction accuracy was found for the training data as well as for the validation data. However, Salas et al. (2000) determined that using ten hidden nodes created the best-fit model, while in this research a best-fit model was found using only five hidden nodes. About a dozen trials were run to

find a best-fit FNN, emphasizing again the problem with using randomized initial network parameters.

Figures 4.40 and 4.41 show the actual and predicted outputs generated by the best-fit feed-forward network using temperature and precipitation as input parameters. Figure 4.40 represents the streamflow data used for training the neural network, while Figure 4.41 represents the data for validating the network. Both sets of data are modeled fairly well by the FNN over the course of three years. Figures 4.42 and 4.43 graph the actual streamflow values versus the predicted streamflow values calculated by the feed-forward neural network. For the training data set seen in Figure 4.42, the predictions are generally unbiased and the accuracy is good even at high streamflow values. Figure 4.43 shows similar results for the validation data. Now that the predictive ability of ANNs has been confirmed, they can now be compared to the structure and performance of statistical regression models.

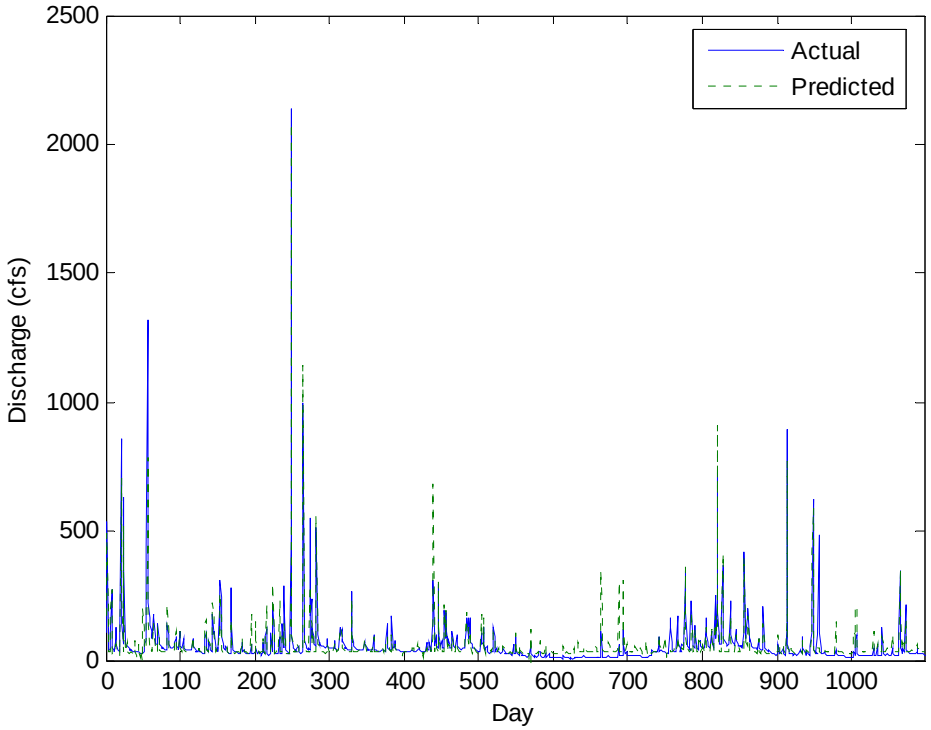


Figure 4.40: *Predicted daily streamflow using 3 Linear - 5 Sigmoid - 1 Sigmoid FNN, training data.*

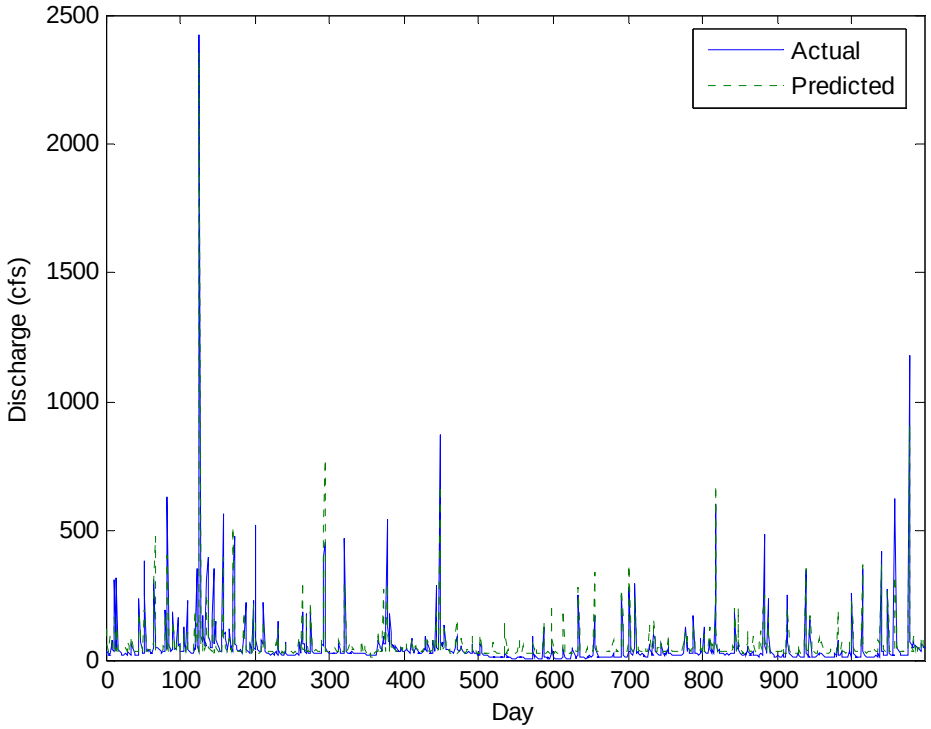


Figure 4.41: *Predicted daily streamflow using 3 Linear - 5 Sigmoid - 1 Sigmoid FNN, validation data.*

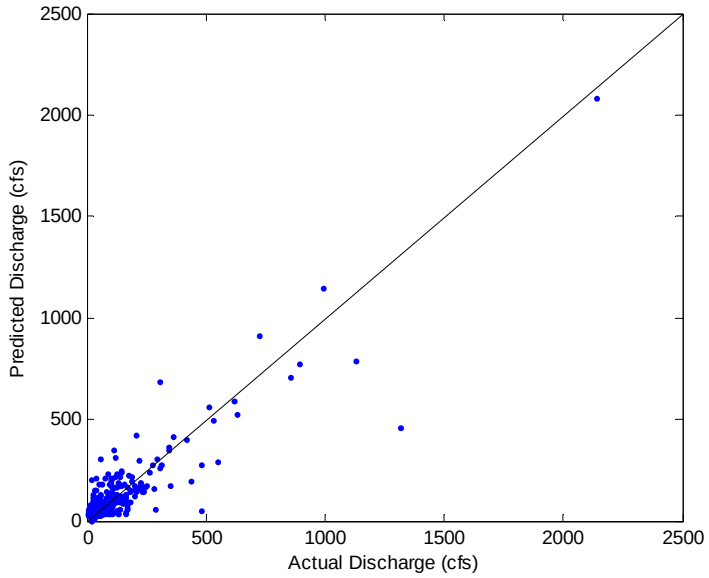


Figure 4.42: *Feed-forward neural network prediction accuracy for training data.*

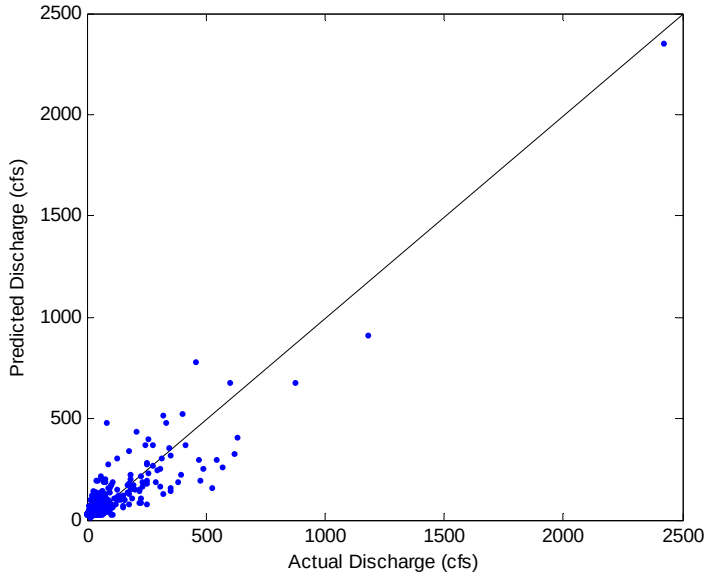


Figure 4.43: *Feed-forward neural network prediction accuracy for validation data.*

4.3.2 Comparison of ANNs and Regression Models

4.3.2.1 Non-recursive Input - FNN versus MPR

The different non-recursive input sets were trained with both feed-forward networks and multiple polynomial regression to model streamflow. Table 4.47 shows the best-fit feed-forward networks that were found for each set of input variables and Table 4.48 lists the best-fit MPR equations. The best-fit models were chosen based on the validation error. The FNNs for these cases are using a three layer Linear - Sigmoid - Linear network structure. The results show that all of the models do a decent job of predicting streamflow, since all of their *s(e)/s(y)* error values are less than one. Neither FNNs nor MPR seem to have an advantage over the other overall. However, in all cases, the network models needed more parameters than the regression equations. The best-fit model overall was a FNN with two hidden nodes estimating Function 2, using P_t and P_{t-1} as input variables.

Table 4.47: *Best-fit FNNs for non-recursive streamflow functions.*

	Function 1 (P_t)	Function 2 (P_t, P_{t-1})	Function 3 (P_t, T_t)	Function 4 (P_t, P_{t-1}, T_t)
# Hidden Nodes	1	2	3	2
# of Parameters	4	9	13	11
Training Error	0.60052	0.53060	0.51755	0.50144
Validation Error	0.79177	0.47087	0.66301	0.50923

Table 4.48: *Best-fit MPR equations for non-recursive streamflow functions.*

	Function 1 (P_t)	Function 2 (P_t, P_{t-1})	Function 3 (P_t, T_t)	Function 4 (P_t, P_{t-1}, T_t)
Equation Order	1	2	2	2
# of Parameters	2	6	6	10
Training Error	0.69371	0.54388	0.60039	0.49883
Validation Error	0.66123	0.55881	0.64322	0.52361

The FNN models and MPR equations used to estimate Function 1 were compared first. This is a simple function that uses a single input variable, precipitation, to predict

streamflow. Figure 4.44 shows the training and validation error found for FNNs based on the number of hidden nodes and Table 4.49 shows the error for the first fifteen orders of MPR that occur when streamflow is predicted using precipitation alone. As expected, the training error decreases for both models as the complexity of the model increases and more parameters are added.

With only one hidden node, the FNN produced the same training error ($s(e)/s(y)$ = 0.60052) and validation error (0.79179) for all five trials, meaning the network converged to a unique solution. Networks with more than one hidden node in general had validation errors larger than one, indicating poor prediction. Also, the FNN results varied from trial to trial between tests with the same number of hidden nodes suggesting non-uniqueness of the trained model. This is probably due to the random initialization of the network parameters. Each network likely found a different local minimum in the error gradient. Only one trial out of these networks managed a validation error less than one, which occurred with twenty-eight hidden nodes ($s(e)/s(y)$ = 0.83878).

When testing multiple polynomial regression, the trials for each order of regression produced the same results. This is expected, since the nature of least-squares will always find the best-fit line for a given regression equation. The training error decreased as the model order increased, but the smallest validation error was produced by the first order equation ($s(e)/s(y)$ = 0.66123). Models larger than third order regression had validation errors greater than one. This is mostly likely due to polynomial swing. As the order of the polynomial increases, the equation becomes less robust to outliers in the data. It is interesting to note that this did not occur with feed-forward networks as the

complexity of the network increased. By the time the MPR equations reached fifteenth order, the validation error was significantly larger than the errors found by the FNNs.

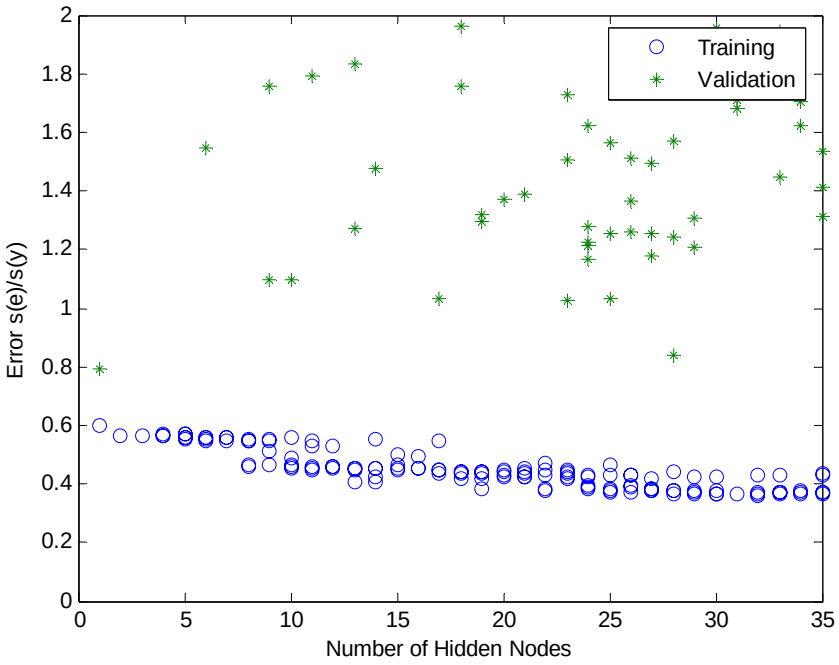


Figure 4.44: *Training and validation error for FNN modeling Function 1.*

Table 4.49: *Results from first fifteen orders of MPR for Function 1.*

Order	Training Error	Validation Error
1st	0.69371	0.66123
2nd	0.62129	0.66140
3rd	0.59879	0.77818
4th	0.58359	1.1603
5th	0.56836	2.7161
6th	0.56672	3.7918
7th	0.56612	5.8912
8th	0.56629	2.8392
9th	0.56346	71.81
10th	0.56335	127.78
11th	0.5632	21.189
12th	0.56334	301.0
13th	0.56296	4088.6
14th	0.5611	31182.0
15th	0.55875	$2.3155 * 10^5$

Comparing the results from the feed-forward network and multiple polynomial regression trials, one can see a number of similarities between the results. Both models were not effective at predicting the validation data after a specific point, FNNs after one hidden node and MPR after the third order. While the training error for both models decreased as the number of parameters increased, this allowed the models to better "memorize" the training data set and reduced their ability to generalize. It is interesting, though, to see how the two types of models correlate.

Comparing the training error shows that both models have similar error based on the number of parameters (Figure 4.45). This suggests again that the maximum order that a FNN can represent is based on the number of hidden nodes it uses. One hidden node (four parameters) will train to the data similarly to a third order regression equation, two hidden nodes (seven parameters) will train similarly to sixth order, and so on.

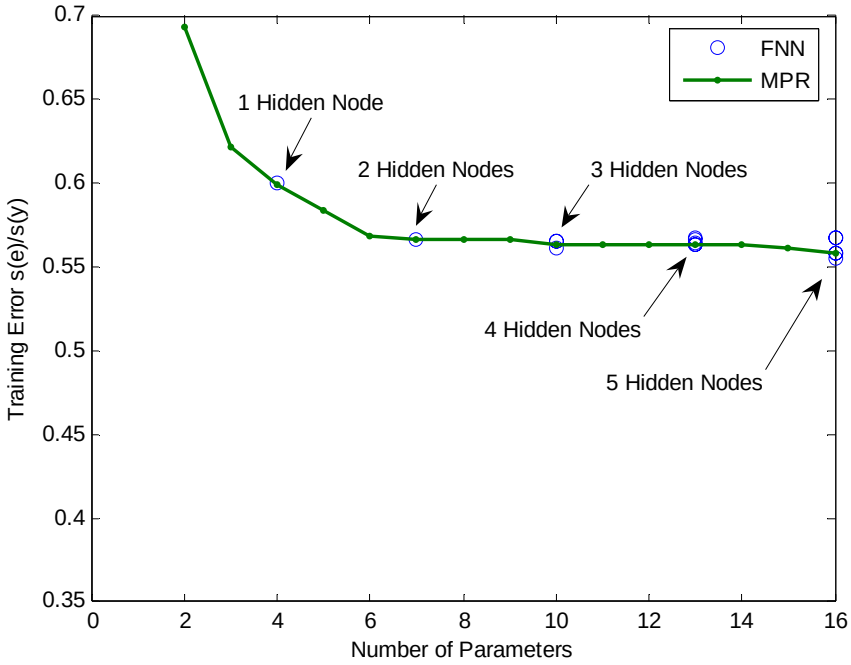
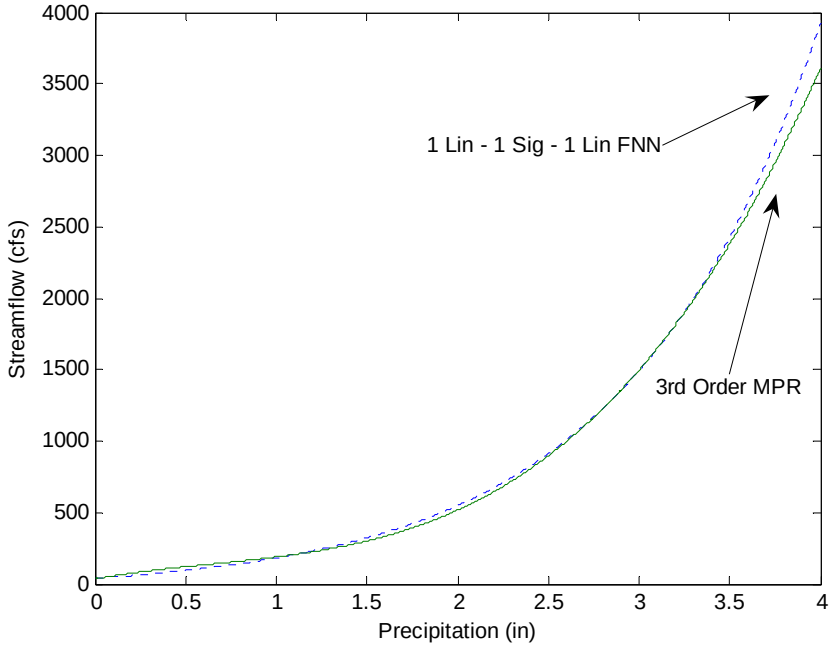


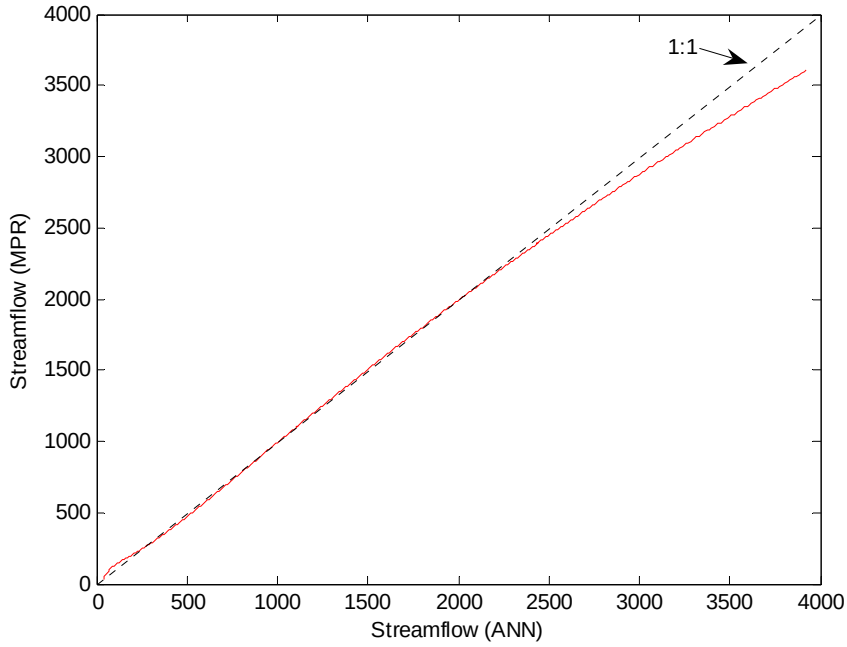
Figure 4.45: *Comparison of training error based on the number of parameters for modeling Function 1.*

The validation results from both models does not compare in the same manner, however (Figure 4.44 and Table 4.49). For MPR, the validation error increases dramatically with order. As the complexity of the model increased with higher orders, the model became less stable. However, the FNN was able to keep the validation error under two for most network structures, no matter how many hidden nodes were used. This shows that the ANN can be more robust than regression equations for predicting data not in the training set.

Based on the validation errors, the best FNN model for the one input, non-recurrent case occurred with one hidden node. Comparing the training and validation error for this feed-forward network to MPR, the closest match is a third order polynomial equation. It is also interesting to note that both of these models have four parameters. Figure 4.46a shows the output function that both models generate for predicting streamflow with an input range of zero to four inches of rainfall. Both functions have a similar shape and curve. The y-intercept is 39.3 cfs for the FNN and 34.8 cfs for MPR. The two models predict similar streamflow for most values of precipitation except for large values of streamflow, where the FNN predicts higher values than the third order MPR equation (Figure 4.46b).



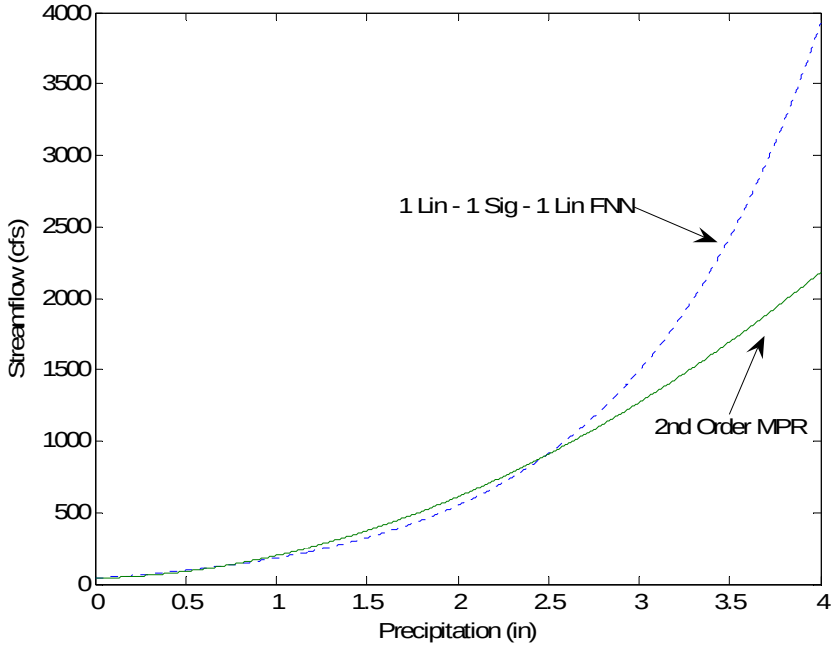
a)



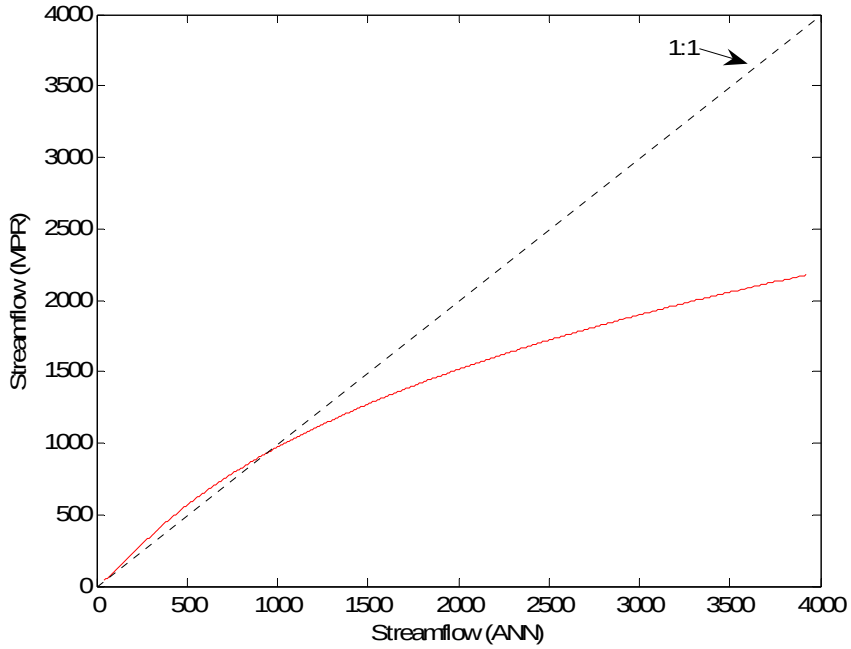
b)

Figure 4.46: *The functions relating P to Q for FNN and MPR (a) and comparing the two models (b).*

While the third order MPR equation fits well with the one hidden node FNN, this seems to contradict earlier empirical and analytical results that showed that two hidden nodes were required for modeling a third order regression equation with one variable. Based on the previous findings, the highest polynomial order that this particular FNN should be able to model is a second order MPR equation. However, when observing the output of the second order equation, the FNN does not compare favorably to it (Figure 4.47). In fact, the validation error for the second order polynomial is much better than the one hidden node feed-forward network (0.66140 for MPR opposed to 0.79179 for the FNN). Looking at the model outputs equations, the FNN has a much steeper slope than the second order MPR.



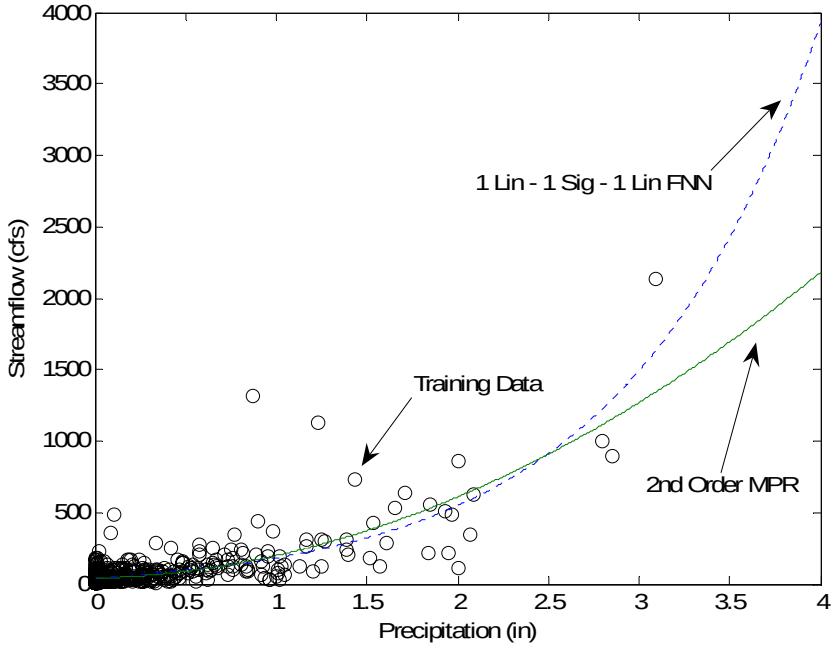
a)



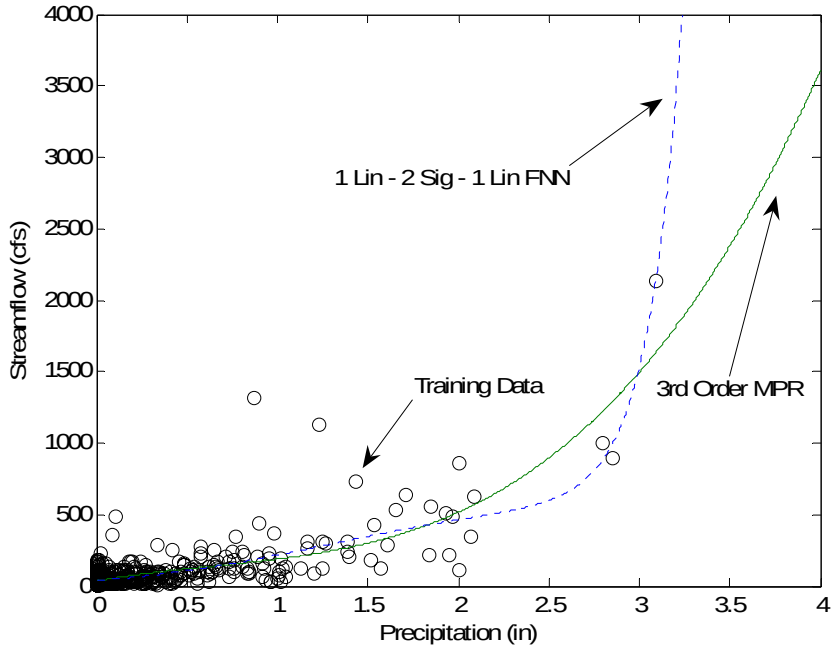
b)

Figure 4.47: *The functions for a FNN and 2nd Order MPR (a) and comparing the two models (b).*

The output of the FNN trained with two hidden nodes is even more dramatic when compared with the third order polynomial. As seen in Figure 4.48, the output function of the two hidden node network shows no similarity to the MPR output function. While the FNN has good accuracy for the training data ($s(e)/s(y)$ = 0.56626), it is terrible for predicting the validation data ($s(e)/s(y)$ = 17.994). It appears from Figure 4.48b that during the training step, the network learned to pass through the three points on the far right of the graph (around a precipitation of three) and the output function continued to skew upwards. This leads to an unreasonably high streamflow prediction for extreme rainfall events.



a)



b)

Figure 4.48: *The output functions for a FNN with 1 (a) and 2 (b) hidden nodes.*

Going back to the one hidden node FNN model, the network weights and biases are used to find the equivalent third order regression coefficients (Equation 4.13) with the formal methods discussed before. The parameters of the network (Figure 4.13) were w_1 = 0.9146, w_2 = 3371, b_1 = -5.0243, and b_2 = 3370. A third order Taylor series expansion is used to transform the sigmoid activation function (Equation 4.10). As Table 4.50 shows, the regression coefficients determined by the network do not match the ones found with least-squares by MPR. It should be noted that the MPR parameters are in the scaled domain. However, the polynomial equation defined by the transformed FNN parameters is not even close to the MPR equation. The problem is likely due to the fact that even though the input to both models was scaled to -0.8 to +0.8, the bias of the hidden node (b_1) moves the range far below this. This puts the range of data going into the sigmoid activation function considerably outside of the effective range of the third order Taylor series expansion estimation.

Table 4.50: *Regression coefficients found by one hidden node sigmoid FNN.*

	3rd Order MPR	1 Lin - 1 Sig - 1 Lin FNN
c_0	-0.5689	128950
c_1	0.4726	-74746
c_2	0.6079	14168
c_3	0.4317	-859.67

The same models were then run through the training and validation process with stream data that was first log normalized and then scaled to the network range. The results from the MPR models showed no improvement over the linear data. However, the log normal transformation of the streamflow data did improve the predictive abilities of the neural network (Figure 4.49). The smallest validation error produced by the log normal trained networks (0.63559 with two hidden nodes) is smaller than the smallest error found by the linear trained networks (0.79179 with one hidden node). This suggests

that using a log normal transformation could have a positive influence over a network's accuracy.

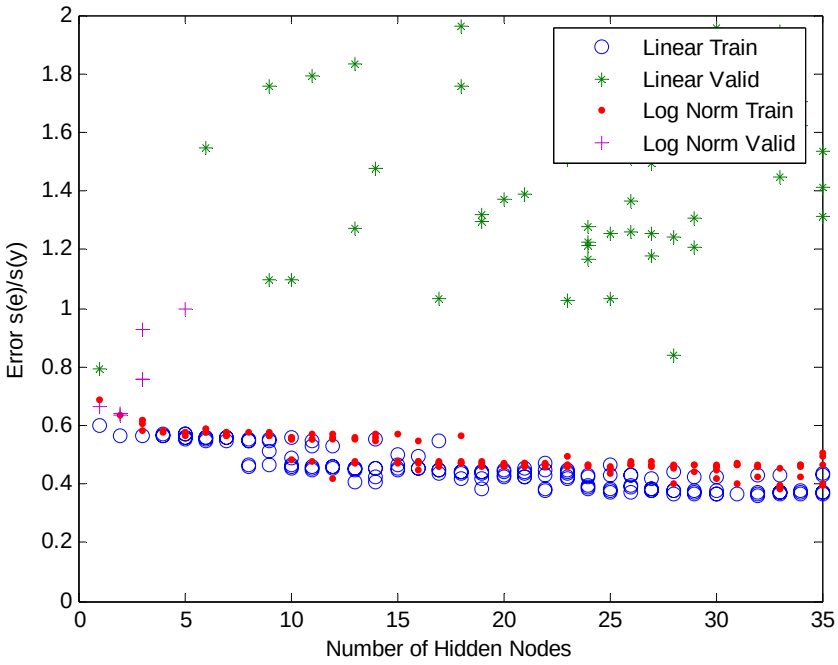
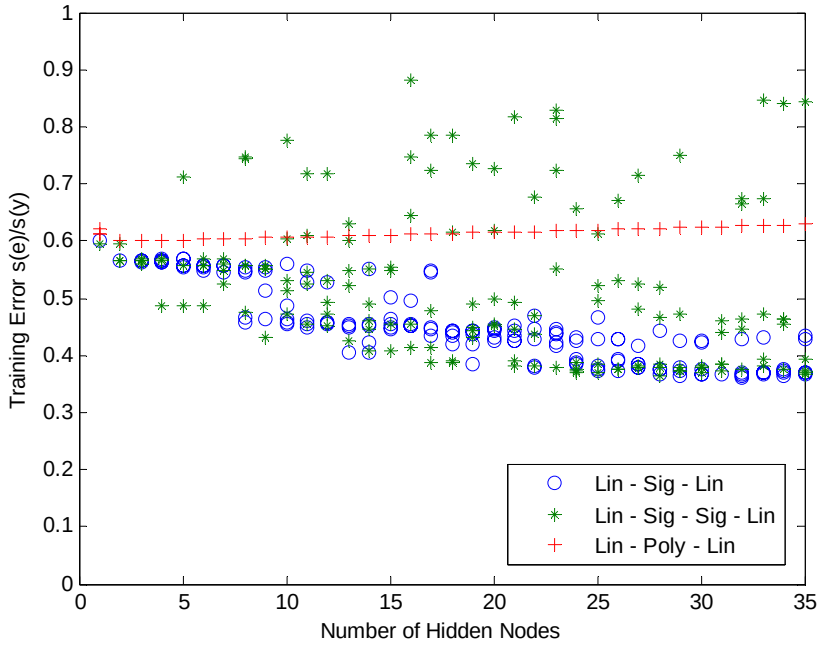


Figure 4.49: *Training and validation error for linear scaled and log normal scaled data.*

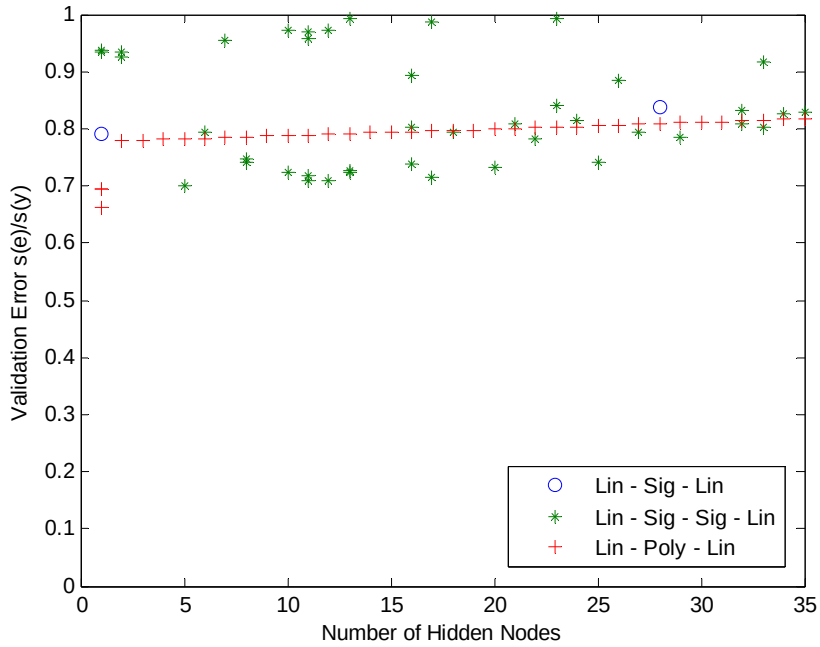
In addition to the Linear - Sigmoid - Linear network structure, two other models were investigated. The first was a FNN with two sigmoid hidden layers, and the second was a FNN with one third order polynomial hidden layer. While one sigmoid hidden layer is commonly used in ANN applications, two sigmoid hidden layers are also used by many researchers to increase the nonlinearity and complexity of the network. The results from training a network with two sigmoid hidden layers indicate that this structure does have the ability to train networks to a higher degree of accuracy than with using only one hidden layer. The two hidden layer network produced validation errors consistently lower than the one hidden layer network (Figure 4.50). However, the downside to using such a structure is that the parallels to regression models are less obvious. There are no formal methods to convert the network weights and biases to equivalent regression

parameters. This limits investigations into the importance of each network parameter. Future research could apply the formal comparisons found for the one sigmoid hidden layer to multiple hidden layer networks, but could be difficult to advance due to the complexity of the models.

The second alternative network structure tested used a polynomial activation function. As expected, this activation function performed differently from the sigmoid activation function as the number of hidden nodes increased. The addition of hidden nodes did not improve the network performance and the best validation error was found with just one node (Figure 4.50). The results coincide with the results from the formal and empirical tests in the previous section on the third order polynomial activation function. The validation error with one hidden node is similar to the error for a second order MPR ($s(e)/s(y)$ = 0.66140) and with two hidden nodes the error is approximate to a third order equation ($s(e)/s(y)$ = 0.77818). The validation error stays around this level even with more hidden nodes because the polynomial activation function limits the network to a third order MPR.



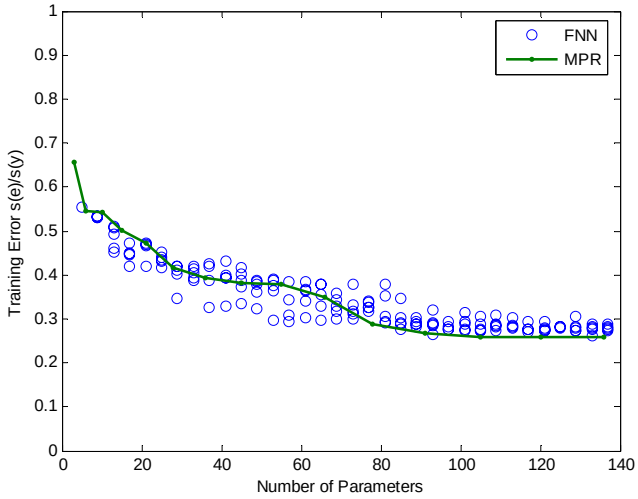
a)



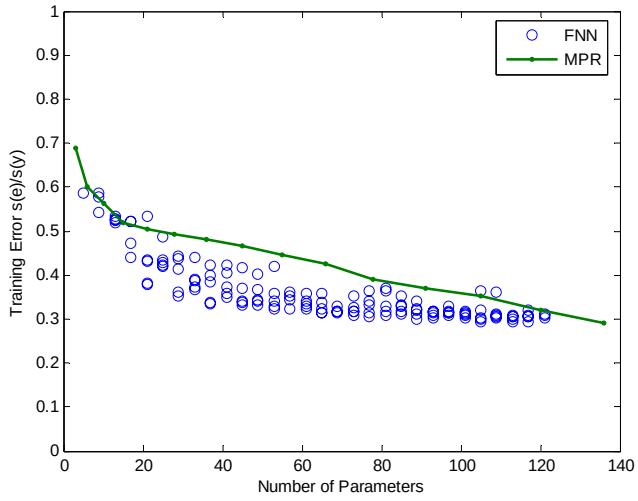
b)

Figure 4.50: *Training (a) and validation (b) error for different network structures.*

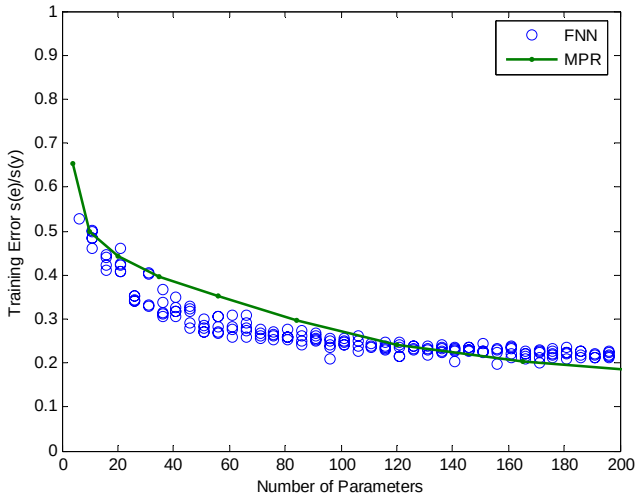
The models for the other non-recurrent functions (Functions 2 through 4) behaved in a manner similar to those for Function 1. The comparison of the number of parameters to the training error for each set of inputs can be seen in Figure 4.51. Function 2, which uses P_t and P_{t-1} as inputs, shows similar results to those found for Function 1 (Figure 4.51a). The error trend for the FNNs and MPR follow each other closely as the number of parameters increases. This means that both models are able to train to the streamflow data with the same degree of accuracy with an equivalent number of parameters. However, for Function 3 and Function 4, the graph of the error terms is not quite the same between the two different models (Figures 4.48b and 4.48c). In both cases, the FNN is able to produce training error values much lower than a MPR equation with the equivalent number of parameters. The reason for this discrepancy is unclear. In fact, based on the results from the previous sections, it would be expected that the FNN would require more parameters than MPR to produce the equivalent model.



a)



b)



c)

Figure 4.51: *Comparison of error based on number of parameters for Functions 2 (a), 3 (b) and 4 (c).*

4.3.2.2 Recursive Input - RNN versus ARMA

For the second set of testing, model functions that are more recurrent were used by adding the term Q_{t-1} as an input variable to all of the input sets. Two different methods of prediction were compared. One-day-ahead prediction used the measured Q_{t-1} value and was modeled by FNNs, while full prediction used the predicted Q_{t-1} value and was modeled by RNNs. Both prediction methods were also modeled by polynomial ARMA equations. Similar to before, the results show that there is comparable accuracy between both the ANNs and the regression models (Table 4.51 and Table 4.52). It was also shown again that in general, ANNs require more parameters. However, in all cases, the recurrent network outperformed the equivalent multiple-day-ahead prediction ARMA model.

Table 4.51: *Best-fit FNNs and RNNs for recursive streamflow functions.*

	Function 5 (P_t, Q_{t-1})		Function 6 (P_t, P_{t-1}, Q_{t-1})		Function 7 (P_t, T_t, Q_{t-1})		Function 8 (P_t,P_{t-1},T_t,Q_{t-1})	
	FNN	RNN	FNN	RNN	FNN	RNN	FNN	RNN
# Hidden Nodes	4	3	4	3	3	3	3	2
# of Parameters	17	13	21	16	16	16	19	13
Training Error	0.4033	0.5200	0.3749	0.4893	0.3775	0.4892	0.3422	0.4914
Validation Error	0.5355	0.5403	0.5316	0.5050	0.5171	0.5662	0.4915	0.4895

Table 4.52: *Best-fit ARMA equations for recursive functions, both one-day-ahead and multiple-day-ahead.*

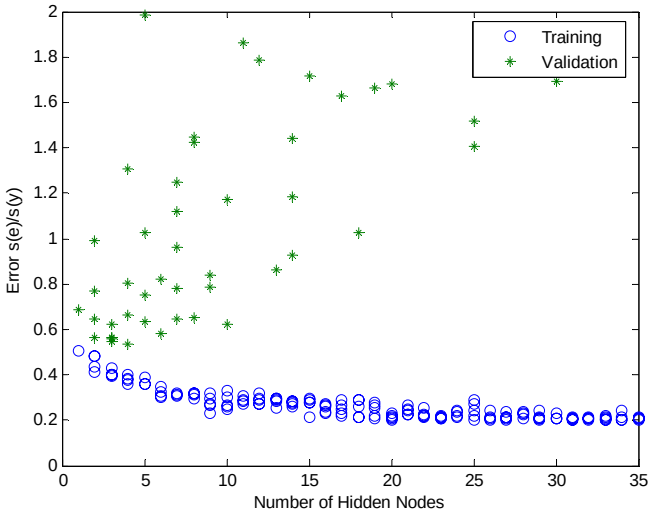
	Function 5 (P_t, Q_{t-1})		Function 6 (P_t, P_{t-1}, Q_{t-1})		Function 7 (P_t, T_t, Q_{t-1})		Function 8 (P_t,P_{t-1},T_t,Q_{t-1})	
	One	Multi	One	Multi	One	Multi	One	Multi
Equation Order	3	3	3	2	2	2	2	2
# of Parameters	10	10	20	10	10	10	15	15
Training Error	0.4133	0.6497	0.3857	0.6735	0.4355	0.6285	0.4157	0.6157
Validation Error	0.5185	0.6162	0.5845	0.6211	0.5188	0.6049	0.5488	0.5888

Another aspect of the results to look at is the comparison between feed-forward networks and recurrent neural networks. In general, the recurrent networks were able to produce best-fit models with less hidden nodes than the FNNs. This is interesting,

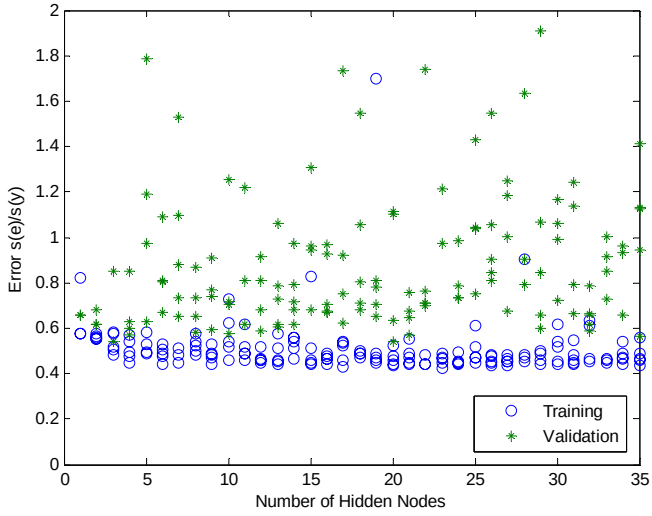
because the RNNs have the extra advantage over FNNs for being fully recursive with respect to streamflow. The recurrent networks do not need to rely on the actual value of the previous day's streamflow, and yet it is still able to perform comparably to the FNN. Also, the best-fit network overall was found to be a recurrent network estimating Function 8 ($s(e)/s(y)$ = 0.4895). This shows that the addition of recurrent connections to a network is able to produce more efficient models than a standard feed-forward network. Looking back at the source of inspiration for ANNs, biological neural networks, it should be reminded that feed-forward neural networks are not an accurate model for how biological networks operate. In particular, they lack the concept of massive parallel feedback. The method of one-directional flow used by FNNs is not able to replicate this concept. This short sight of FNNs may be a reason for why RNNs are shown to be more efficient in these results. The recurrent neural network is a much closer model to the structure of biological networks. This could give RNNs an advantage for modeling biological systems.

Looking at Function 5 for investigating further, this function for streamflow uses P_t and Q_{t-1} as input variables. Like with the one-input case, the training error for the FNN decreased as the number of hidden nodes increased, and validation error increased (Figure 4.52a). The lowest values for the validation error occurred when the network had three hidden nodes. One-day-ahead ARMA trained in a similar manner, with the lowest validation error found in the third order equation (Table 4.53). The recurrent network consistently produced training errors larger than the equivalent feed-forward network. It is expected that the RNN has greater error than the FNN, since the RNN is using full, multiple-day-ahead, prediction. However, it is interesting that the RNN produced more

stable results for validating data with a large number of hidden nodes than was possible with the FNN (Figure 4.52b).



a)



b)

Figure 4.52: *Training and validation error for FNN (a) and RNN (b) for Function 5.*

Table 4.53: *Results for the first five orders, and last order, of ARMA model for Function 5.*

	One-day-ahead		Full Prediction	
Order	Training Error	Validation Error	Training Error	Validation Error
1st	0.63793	0.6492	0.70537	0.67758
2nd	0.45203	0.54305	0.67438	0.64012
3rd	0.41334	0.5185	0.64978	0.61621
4th	0.38680	2.6744	Failed	Failed
5th	0.37328	18.824	Failed	Failed
15th	0.19684	$4.0134 * 10^8$	Failed	Failed

Results from the one-day-ahead prediction models indicate more similarities between the feed-forward neural network and ARMA. The training error for both models is similar in relation to the number of parameters (Figure 4.53). The variance between different FNN trials is likely due to the random initialization of network weights and biases and the existence of local minima. As a result, some networks are luckier than others at predicting the training and validation sets. A similar comparison could not be made between RNNs and full prediction ARMA equations, because ARMA failed to make a stable prediction after the third order (Table 4.53).

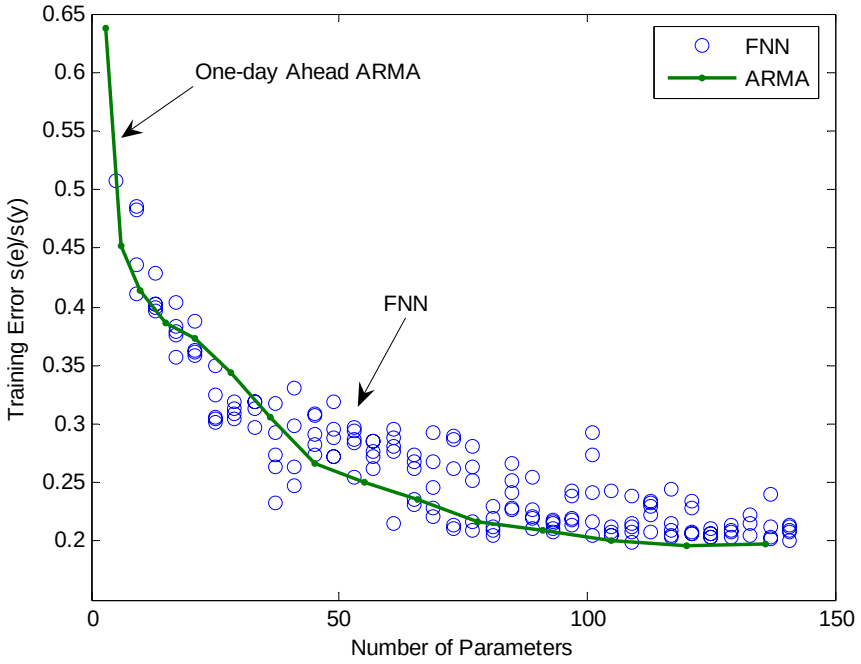


Figure 4.53: *A comparison of the training error for both the FNN and one-day-ahead ARMA based on the number of parameters used in the equation for estimating Function 5.*

The FNN that produced the most accurate and most consistent validation results (three hidden nodes) was then compared to an ARMA equation with similar production (third order polynomial). The third-order equation was also the regression model with the best validation error (0.5185). Both models used a similar number of parameters,

thirteen in the FNN and ten in the ARMA model. Figure 4.54 shows the functions created by both models. Like before, the FNN and regression equation have similar results at low levels of streamflow and are less comparable at the higher levels.

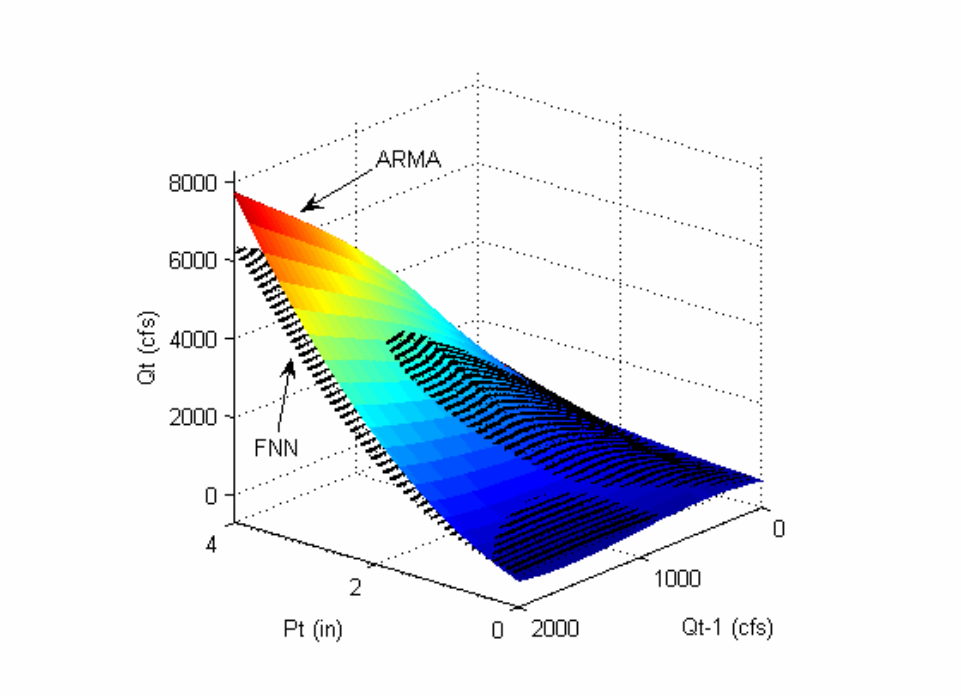


Figure 4.54: *Function estimates for Function 5 produced by a 3 hidden node FNN and a 3rd order one-day-ahead ARMA.*

5 Summary and Conclusions

This study investigated the potential equivalences between artificial neural networks and polynomial regression models in three major steps: 1) a formal and empirical comparison of feed-forward neural networks with multiple polynomial regression equations using synthetic data; 2) a formal and empirical comparison of recurrent neural networks with auto-regressive moving average equations using synthetic data; and 3) an empirical comparison of the ability of recurrent and non-recurrent ANNs and statistical models to simulate the bioenvironmental phenomenon of streamflow using real-world data.

The first two types of comparisons were performed for single perceptrons as well as for networks of perceptrons. Formal comparisons were made by expanding the perceptron activation functions into Taylor series, substituting the expansions into the network equations, comparing the form of the result (term by term) with polynomial statistical models and then developing equations that transform network weights and biases into the coefficients of the potentially equivalent regression models. Empirical comparisons were made by first training the neural networks and regressing the statistical models to pre-determined training data sets, and then: 1) comparing the fit of the ANNs and regression models to the training data sets; 2) comparing the predictions of the ANNs and regression models to a distinct validation data set; and 3) comparing the regressed coefficients of the statistical models to the potentially equivalent coefficients calculated from the weights and biases of the trained neural networks.

The study was limited to neural networks with up to five input nodes, two hidden layers, forty-five hidden nodes, and three delays. The statistical models were limited up

to five input variables, fifth order, three auto-regressive terms and one moving average term. The neural networks were trained by back-propagation while the statistical regression equations were estimated with least-squares.

5.1 Equivalence of FNN and MPR

The results from this research support the theory that feed-forward neural networks and polynomial regression equations are mathematically equivalent models. Formal tests confirmed that FNNs with linear and polynomial activation functions can perfectly replicate target MPR equations, as long as there are enough hidden nodes in the network. It was also demonstrated that FNNs with a sigmoid activation function can model the output function of polynomial equations with accuracy similar to that of FNNs with a polynomial activation function. However, it generates a distinct prediction for data outside the bounds of the training set. Equivalent polynomial regression coefficients were successfully obtained from the weights and biases of trained neural networks when the input data range was small (between -1 and +1) but not when it was large.

A strong relationship was found between the optimal number of hidden nodes in a FNN and the order of the underlying physical (polynomial) equation being modeled by the network. Results demonstrated that for every system there exists an optimal, or critical, number of hidden nodes where validation error is at a minimum (Figure 5.1). This was found to correspond to the point where there are at least as many parameters in the FNN as there are in the target polynomial equation being modeled. Before this point, there are not enough parameters in the network to replicate the underlying polynomial and afterwards the extra parameters lead to memorizing the training data and result in

poor generalization by the network. A table was built (Table 4.17) that identifies this point (the minimum number of hidden nodes) needed to replicate a polynomial model as a function of the number of input variables and the order of the model. Results further suggested that the second layer of network weights (that of the output layer of the network) does not enhance the performance (uniqueness) of the FNN, because the weights are being applied to all of the variables. However, it does appear to play a role in scaling and offsetting to the network output function, which is important when the ANN uses a bounded activation function, such as a sigmoid, in the hidden layer.

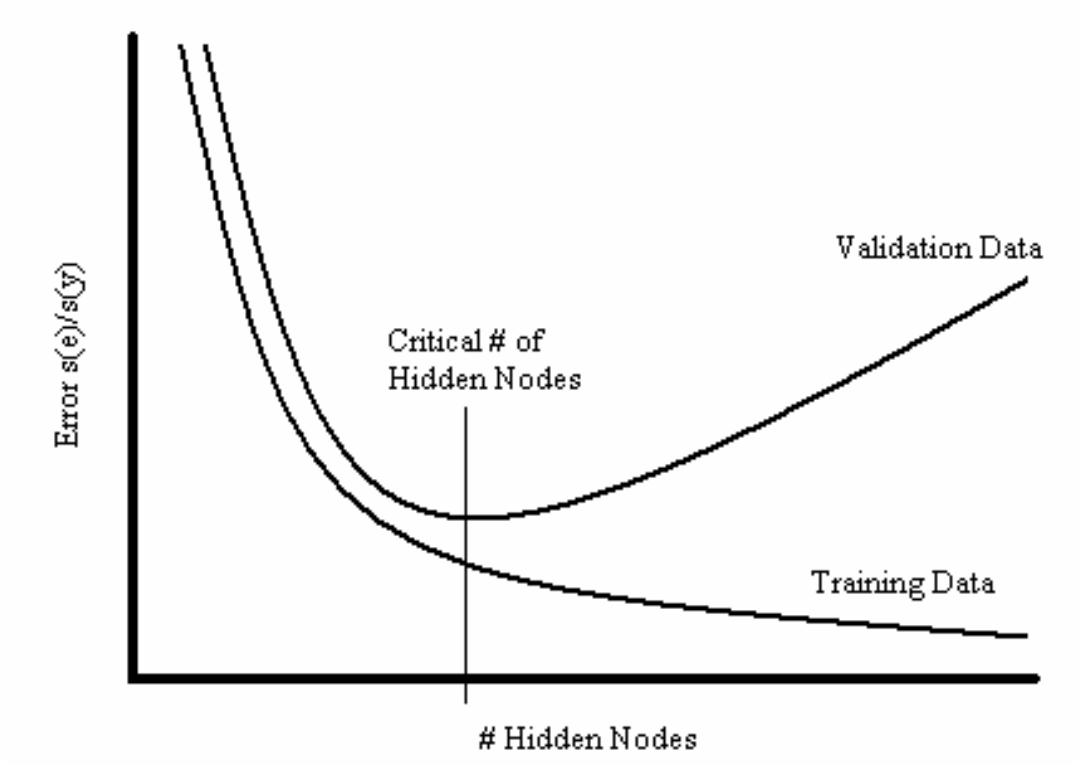


Figure 5.1: *General relationship between the number of hidden nodes and network performance.*

5.2 Equivalence of RNN and ARMA

The results found support the potential equivalence between recurrent neural networks and auto-regressive moving average functions. It was demonstrated formally that a recurrent network has the ability to both replicate the structure of linear ARMA equation and empirically that it can train itself to find the correct regression parameters. However, there was evidence that the physics of the underlying time series equation being modeled has some effect on a RNN's ability to find these parameters. In particular, the neural network converged to the correct parameters when the time series was stable and appeared to have difficulty when modeling unstable equations.

Recurrent networks were also shown to have an advantage over ARMA equations when an error term is used in the model. RNNs are inherently able to include the error term as an input to the network and calculate its weights and biases as it would normally with back-propagation. On the other hand, ARMA models are required to use less accurate methods such as long-AR to estimate the coefficients to the error terms.

5.3 Application to Biological Phenomena

Artificial neural networks and statistical regression equations were both effective models for predicting daily streamflow of the Little Patuxent River. With a small number of hidden nodes, or a low order polynomial equation, both models produced equivalent results. At high polynomial orders, statistical regression equations trained over the calibration data set were not effective at predicting the validation data. However, the ANN model was shown to remain effective even with a large number of hidden nodes. Using a large number of hidden nodes, however, did not do much to increase the network

performance and may hinder the network's ability to generalize and predict outliers in other, arbitrary, validation data sets.

Comparing the two models supported many of the formal and empirical equivalences found in the previous two sections, although there were some discrepancies, which are most likely due to the complex nature of real life systems. The results show that with careful study of the data series being modeled, it is possible to create an efficient ANN that is both an accurate predictor and easier to interpret than traditional methods of trial-and-error.

When comparing the application of feed-forward neural networks to recurrent neural networks, it was observed that RNNs had two major advantages over FNNs: 1) they allow multiple-day-ahead prediction of streamflow and 2) they are more efficient than FNNs. In general, the multiple-day-ahead predictions of RNNs were found to be only slightly less accurate than the one-day-ahead predictions of FNNs. And in some cases, the RNNs were more accurate. Additionally, the best-fit recurrent networks generally used less hidden nodes and fewer parameters than the best-fit feed-forward networks. The best-fit RNNs performed more accurately than the best-fit multiple-day-ahead ARMA equations. The ARMA model failed to make a prediction when the polynomial order was larger than three, while the RNN produced reasonable predictions even with a large number of hidden nodes.

5.4 Overall Conclusions

It has been confirmed that there is a strong relationship between artificial neural networks and statistical regression. Both are empirical models with their own strengths

and weaknesses and while some basic similarities between the two models are obvious, formal connections between them have not been well known to modelers. The following conclusions can be drawn based on the results of this study:

- A single linear perceptron is equivalent to multiple linear regression with respect to the regression coefficients.
- A feed-forward network with polynomial activation functions is equivalent to multiple polynomial regression with respect to the regression coefficients.
- A feed-forward network with sigmoid activation functions is equivalent to multiple polynomial regression with respect to the quality of fit. It is equivalent with respect to the regression coefficients when the range of the input data set is small (between -1 and +1).
- A single linear recurrent perceptron is equivalent to ARMA with respect to the regression coefficients when the target time series is stable. It is not as good as ARMA when the target time series is unstable.
- A single linear recurrent perceptron is better than ARMA with respect to the regression coefficients when an error term is used in the equation instead of an independent input variable.
- A recurrent network with sigmoid activation function is equivalent to NARMA with respect to the quality of fit. It is not as good as NARMA with respect to the regression coefficients.

Hopefully, through research such as this, more connections and more strategies to eliminate the weaknesses of both types of models can be found. Finding equivalences and combining knowledge between these two types of models will allow researchers to

design more effective models that combine the advantages of both artificial neural networks and statistical regression. Modelers would be able to easily design models to describe a complex biological system that are both accurate as well as easy to interpret. The field as a whole would benefit from this knowledge.

Additionally, it would be beneficial for modelers to give more consideration to recurrent networks because of their strong connection to the parallel processing abilities to biological neural networks. Feed-forward networks are currently the more common network structure primarily because of "tradition" and because of their relative ease of use. However, these networks have lost many of the concepts and abilities of the biological networks that inspired them, such as massive parallel processing. It may be conceptually advantageous for modeling, particularly when modeling biological systems, that we retain the properties of biological systems such as the neuron as much as possible.

6 Future Research

There are many areas of research that can be explored further based on the findings from this thesis. The empirical relationship between artificial neural networks and statistical regression models has already been demonstrated in many applications. Future research can be performed to strengthen the formal relationship between the two models. Some specific ideas for future research include:

- Develop formal equations to define network parameters (such as weights and biases) in terms of regression coefficients. This could be useful for initializing network parameters and reducing the chance of a poor network being trained as a result of bad initial values.
- Find formal equations relating more complex neural networks, such as those with two hidden sigmoid layers, to statistical regression equations.
- Examine other network parameters that influence ANN performance, such as the activation function, number of hidden layers, number of epochs, learning rate, and others not encompassed by this thesis.
- Continue to investigate the optimal number of hidden nodes to model a particular system. In particular, determine a stronger formal relationship between the number of hidden nodes and the order of the output function for the network.
- Investigate formal methods for determining the optimal order (p and q) of a RNN or ARMA model for a given time series.
- Expand the investigation of application to biological systems to include other fields and systems where modeling is important. For example, modeling population dynamics or electrical cardiograph signals.

- Explore alternatives to back-propagation for training the parameters of an ANN.
- Evaluate the use of sigmoid functions in statistical regression models, in particular their ability to provide robust predictions.
- Develop a relationship or guideline between the number of parameters in a model and the optimal number of hidden nodes and activation functions in an artificial neural network.
- Investigate potential equivalences between artificial neural networks, statistical regression, and ordinary and partial differential equations. Particularly in the area of time series modeling, where variables change with respect to time. The difference between two time steps can be used to estimate the derivative, similar to the finite difference method. This knowledge could potentially convert discrete-time recurrent neural networks into continuous-time differential equations.

Appendix A - MATLAB Code

```
function [net] = percepnet()
   %This code trains both linear and third order polynomial
   %equations to both perceptrons and regression equations.

   %Training Data
   x=[-149:1:150];
   %1 Var, 1st Order
   %y = 100 + 2*x + 10*randn(1,300);
   %1 Var, 3rd Order
   y = 100 + 2*x - 0.005*x.^2 + 0.0001*x.^3 + 10*randn(1,300);

   %Scale data to range of -0.1 to +0.1
   ox = x;
   oy = y;
   for i = 1:300
      sx(i) = ((0.1 - -0.1) * x(i) + (max(x) * -0.1 - min(x) * 0.1)) / (max(x) - min(x));
      sy(i) = ((0.1 - -0.1) * y(i) + (max(y) * -0.1 - min(y) * 0.1)) / (max(y) - min(y));
   end
   tx = sx;
   ty = sy;

   %Linear Perceptron
   %net = newff([min(tx) max(tx)], [1], {'purelin'});
   %Nonlinear Perceptron
   net = newff([min(tx) max(tx)], [1], {'tansig'});
   net.trainParam.epochs = 500;
   net.trainParam.mu_max = 1.0000e+010;
   input = [tx];
   net = train(net,input,ty);
   ytn = sim(net,input);
   [net.b{1} ; net.IW{1}]

   %Linear Regression
   %input = [ones(300,1) tx'];
   %p = input \ ty'
   %ytr = p(1) + p(2) * tx;
   %Polynomial Regression
   input = [ones(300,1) tx' tx'.^2 tx'.^3];
   p = input \ ty'
   ytr = p(1) + p(2) * tx + p(3) * tx.^2 + p(4) * tx.^3;

   %Rescale data
   for i = 1:300
      sytn(i) = ((max(y) - min(y)) * ytn(i) - (max(y) * -0.1 - min(y) * 0.1)) / (0.1 - -0.1);
      sytr(i) = ((max(y) - min(y)) * ytr(i) - (max(y) * -0.1 - min(y) * 0.1)) / (0.1 - -0.1);
   end
   ytn = sytn;
   ytr = sytr;

   %Output
   se = sqrt(sum((ytn - y).^2)/(300 - 2));
   sy = sqrt(sum((y - mean(y)).^2)/(300 - 1));
   sesyn = se/sy
```

```
    se = sqrt(sum((ytr - y).^2)/(300 - 4));
    sy = sqrt(sum((y - mean(y)).^2)/(300 - 1));
    sesyr = se/sy
    close
    plot(x,y,'o',x,ytn,x,ytr)

    %Validation data
    clear x y
    x=[151:1:250];
    %y = 100 + 2*x + 10*randn(1,100);
    y = 100 + 2*x - 0.005*x.^2 + 0.0001*x.^3 + 10*randn(1,100);

    %Scale
    for i = 1:100
       sx(i) = ((0.1 - -0.1) * x(i) + (max(ox) * -0.1 - min(ox) * 0.1)) / (max(ox) - min(ox));
       sy(i) = ((0.1 - -0.1) * y(i) + (max(oy) * -0.1 - min(oy) * 0.1)) / (max(oy) - min(oy));
    end
    tx = sx;
    ty = sy;

    %Validate
    input = [tx];
    yvn = sim(net,input);
    yvr = p(1) + p(2) * tx + p(3) * tx.^2 + p(4) * tx.^3;

    %Rescale data
    for i = 1:100
       syvn(i) = ((max(oy) - min(oy)) * yvn(i) - (max(oy) * -0.1 - min(oy) * 0.1)) / (0.1 - -0.1);
       syvr(i) = ((max(oy) - min(oy)) * yvr(i) - (max(oy) * -0.1 - min(oy) * 0.1)) / (0.1 - -0.1);
    end
    yvn = syvn;
    yvr = syvr;

    %Output
    se = sqrt(sum((yvn - y).^2)/(100 - 2));
    sy = sqrt(sum((y - mean(y)).^2)/(100 - 1));
    sesyn = se/sy
    se = sqrt(sum((yvr - y).^2)/(100 - 4));
    sy = sqrt(sum((y - mean(y)).^2)/(100 - 1));
    sesyr = se/sy
    hold on
    plot(x,y,'o',x,yvn,x,yvr)
end
```

```
function [net sig] = cubicnet()
    %This code is used to train both polynomial and sigmoid
    %FNNs to cubic ordered polynomial equations.

    %Training Data
    x1 = [-149:1:150];
    t = [-149:1:150];
    %1 Var, 3rd Order
    y = 100 + 2*x1 - 0.005*x1.^2 + 0.0001*x1.^3 + 10*randn(1,201);
    %y = 0 - 0.4*x1 + 0.3*x1.^2 + 1*x1.^3 + 0*randn(1,21);

    %1 or 2 hidden node with third order activation function
    disp('Input - X = -149 to 150, 151 to 250')
    disp('Ouput - Y = 100 + 2*X - 0.005*X.^2 + 0.0001*X.^3 + 0*randn')
    disp('FNN - 1 Lin - 2 Third - 1 Lin + Biases, 500 Epochs')
    net = newff([min(y) max(y)], [1 2 1], {'purelin' 'third' 'purelin'});
    net.trainParam.epochs = 500;
    net.trainParam.mu_max = 1.0000e+020;
    net.biasConnect = [0;1;1];
    net.IW{1} = [1];
    net.inputWeights{1}.learn = 0;
    %net.LW{3,2} = [1 1];
    %net.layerWeights{3,2}.learn = 0;
    %net.b{3} = 0;
    %net.biases{3}.learn = 0;
    input = [x1];
    net = train(net,input,y);
    yt = sim(net,input);

    %Parameters with one hidden node
    %w1 = net.LW{2,1}(1);
    %w2 = net.LW{3,2}(1);
    %b1 = net.b{2}(1);
    %b2 = net.b{3}(1);
    %p1 = w2*b1^3 + b2
    %p2 = 3*w1*w2*b1^2
    %p3 = 3*w1^2*w2*b1
    %p4 = w1^3*w2

    %Parameters with two hidden nodes
    w1 = net.LW{2,1}(1);
    w2 = net.LW{2,1}(2);
    w3 = net.LW{3,2}(1);
    w4 = net.LW{3,2}(2);
    b1 = net.b{2}(1);
    b2 = net.b{2}(2);
    b3 = net.b{3};
    p1 = w3*b1^3 + w4*b2^3 + b3
    p2 = w3*3*w1*b1^2 + w4*3*w2*b2^2
    p3 = w3*3*w1^2*b1 + w4*3*w2^2*b2
    p4 = w3*w1^3 + w4*w2^3

    %Feed-forward network with sigmoid activation function
    disp('FNN - 1 Lin - 2 Sig - 1 Lin + Biases, 500 Epochs')
    sig = newff([min(y) max(y)], [1 2 1], {'purelin' 'tansig' 'purelin'});
    sig.trainParam.epochs = 500;
```

```
    sig.trainParam.mu_max = 1.0000e+050;
    sig.biasConnect = [0;1;1];
    sig.IW{1} = [1];
    sig.inputWeights{1}.learn = 0;
    input = [x1];
    sig = train(sig,input,y);
    ys = sim(sig,input);

    %Regression Equation
    input = [ones(201,1) x1' x1'.^2 x1'.^3];
    p = input \ y'
    yr = p(1) + p(2) * x1 + p(3) * x1.^2 + p(4) * x1.^3;

    %Output
    se = sqrt(sum((yt - y).^2)/(300 - (1+(1+1+1)*2)));
    sy = sqrt(sum((y - mean(y)).^2)/(300 - 1));
    sesyn = se/sy
    se = sqrt(sum((yr - y).^2)/(300 - 4));
    sy = sqrt(sum((y - mean(y)).^2)/(300 - 1));
    sesyr = se/sy
    se = sqrt(sum((ys - y).^2)/(300 - (1+(1+1+1)*2)));
    sy = sqrt(sum((y - mean(y)).^2)/(300 - 1));
    sesys = se/sy
    close
    plot(t,y,t,yt,t,yr,t,ys)

    %Validation Data
    x1 = [151:1:250];
    t = [151:1:250];
    y = 100 + 2*x1 - 0.005*x1.^2 + 0.0001*x1.^3 + 10*randn(1,51);
    %y = 0 - 0.4*x1 + 0.3*x1.^2 + 1*x1.^3 + 0*randn(1,6);

    %Validate
    input = [x1];
    yt = sim(net,input);
    ys = sim(sig,input);
    %yr = p(1) + p(2) * x1 + p(3) * x1.^2;%2nd Order
    yr = p(1) + p(2) * x1 + p(3) * x1.^2 + p(4) * x1.^3;

    %Output
    se = sqrt(sum((yt - y).^2)/(100 - (1+(1+1+1)*2)));
    sy = sqrt(sum((y - mean(y)).^2)/(100 - 1));
    sesyn = se/sy
    se = sqrt(sum((yr - y).^2)/(100 - 4));
    sy = sqrt(sum((y - mean(y)).^2)/(100 - 1));
    sesyr = se/sy
    se = sqrt(sum((ys - y).^2)/(100 - (1+(1+1+1)*2)));
    sy = sqrt(sum((y - mean(y)).^2)/(100 - 1));
    sesys = se/sy
    hold on
    plot(t,y,t,yt,t,yr,t,ys)
end
```

```
function [output] = autopolynet()
   %This code creates synthetic MPR equations with 1 to 5
   %variables from 1st to 5th order.  It then trains a
   %series of FNNs with 1 to 45 hidden nodes.
   tic
   %Set up input variables
   num = 500;
   x1 = randn(1,num);
   x2 = randn(1,num);
   x3 = randn(1,num);
   x4 = randn(1,num);
   x5 = randn(1,num);
   training = [x1;x2;x3;x4;x5];
   xv1 = randn(1,num);
   xv2 = randn(1,num);
   xv3 = randn(1,num);
   xv4 = randn(1,num);
   xv5 = randn(1,num);
   validate = [xv1;xv2;xv3;xv4;xv5];

   %Used to determine the order of the activation function
   actfun = {'purelin' 'second' 'third' 'forth' 'fifth'};
   regvar = {'x1+1' 'x1+x2+1' 'x1+x2+x3+1' 'x1+x2+x3+x4+1' 'x1+x2+x3+x4+x5+1'};

   %Loop through # orders, # variables, and # hidden nodes
   index = 0;
   for ord = 1:5
      ord
      for var = 1:5
         var
         %Set up regression equation
         s = sym(char(regvar(var)));
         s = expand(s^ord);
         s = char(s);
         len = size(s);
         len = len(2);
         clear store;
         temp = '';
         count = 1;
         for i = 1:len
            if (s(i) ~= '+')
               if ((s(i) == '^') || (s(i) == '*'))
                  temp = strcat(temp,'.');
               end
               temp = strcat(temp,s(i));
            else
               store(count) = cellstr(temp);
               count = count + 1;
               temp = '';
            end
         end
         store(count) = cellstr(temp);
         len = size(store);
         len = len(2);
         %Remove coefficients due to polynomial expansion
         for j = 1:len
```

```
            temp = char(store(j));
            if ((temp(1) == '1') || (temp(1) == '2') || (temp(1) == '3') || (temp(1) == '4') || (temp(1) == '5') ||
(temp(1) == '6') || (temp(1) == '7') || (temp(1) == '8') || (temp(1) == '9'))
               l = size(temp);
               l = l(2);
               tem = '';
               star = 0;
               for k = 1:l;
                  if ((temp(k) == '*') && (star == 0))
                     star = 1;
                  elseif (star == 1)
                     tem = strcat(tem,temp(k));
                  else
                     %Do nothing
                  end
               end
               store(j) = cellstr(tem);
            end
         end
         %Give random coefficients to each term
         clear c regeq;
         for j = 1:len
            c(j) = randn;
            if (strcmp(cell2mat(store(j)),''))
               regeq(j) = cellstr('c(j)');
            else
               regeq(j) = cellstr(strcat('c(j)*',char(store(j))));
            end
         end
         %Set up target y values
         y = zeros(1,num);
         for j = 1:len
            y = y + eval(cell2mat(regeq(j)));
         end
         %Error/Noise Term
         %y = y + 0.1 * randn(1,num);
         %Set up validation y values
         xt1 = x1;
         xt2 = x2;
         xt3 = x3;
         xt4 = x4;
         xt5 = x5;
         yt = y;
         x1 = xv1;
         x2 = xv2;
         x3 = xv3;
         x4 = xv4;
         x5 = xv5;
         yv = zeros(1,num);
         for j = 1:len
            yv = yv + eval(cell2mat(regeq(j)));
         end
         %Error/Noise Term
         %yv = yv + 0.1 * randn(1,num);
         %Reset
         x1 = xt1;
```

```
x2 = xt2;
x3 = xt3;
x4 = xt4;
x5 = xt5;
y = yt;
%Test neural networks
for hid = 1:45
   hid
   for trial = 1:5
      %Set up network
      clear inputrange;
      for k = 1:var
         inputrange(k,:) = [-50 50];
      end
      net = newff(inputrange, [var hid 1], {'purelin' char(actfun(ord)) 'purelin'});
      %net = newff(inputrange, [var hid 1], {'purelin' 'tansig' 'purelin'});
      %net = newff(inputrange, [var hid 5 1], {'purelin' 'tansig' 'tansig' 'purelin'});
      net.trainParam.epochs = 500;
      net.trainParam.mu_max = 1.0000e+010;
      net.biasConnect = [0;1;1];
      %net.biasConnect = [0;1;1;1];
      net.IW{1} = eye(var);
      net.inputWeights{1}.learn = 0;
      %Train network
      input = training(1:var,:);
      net = train(net,input,y);
      ynt = sim(net,input);
      %Validate network
      input = validate(1:var,:);
      ynv = sim(net,input);
      %Determine error
      rp = len;
      np = (1+(var+1+1)*hid);
      se = sqrt(sum((ynt - y).^2)/(num-np));
      sy = sqrt(sum((y - mean(y)).^2)/(num-1));
      sesy = se/sy
      %Validation error
      sev = sqrt(sum((ynv - yv).^2)/(num-np));
      syv = sqrt(sum((yv - mean(yv)).^2)/(num-1));
      sesyv = sev/syv
      %Save data to output
      index = index + 1;
      output(index,1) = ord;
      output(index,2) = var;
      output(index,3) = hid;
      output(index,4) = sesy;
      output(index,5) = sesyv;
      output(index,6) = rp;
      output(index,7) = np;
      output(index,8) = (sum((ynt - mean(y)).^2))/(sum((y - mean(y)).^2));
      output(index,9) = (sum((ynv - mean(yv)).^2))/(sum((yv - mean(yv)).^2));
      output(index,10) = sum((ynt - y).^2);
      output(index,11) = sum((ynv - yv).^2);
      %Stop if min error is found
      if (sesy < 1e-10)
         break;
```

```
            end
        end
        if (sesy < 1e-10)
            break;
        end
    end
  end
 end
 toc
end
```

```
function [rec] = armaunit()
   %This code tests AR(1,1) and the recurrent perceptron.

   %Input and Target Time Series
   len = 300;
   x = randn(1,len);
   t = [1:len];
   y(1) = 5 * x(1) + 0.5 * 0 - 1.5 + 0 * randn;
   for i = 2:len
      y(i) = 5 * x(i) + 0.5 * y(i-1) - 1.5 + 0 * randn;
   end
   ym1(2:len) = y(1:(len-1));

   %ARMA(1,1) estimation
   input = [x' ym1' ones(len,1)];
   pq = input \ y'
   yhat(1) = pq(1) * x(1) + pq(2) * 0 + pq(3) * 1;
   for i = 2:len
      yhat(i) = pq(1) * x(i) + pq(2) * yhat(i-1) + pq(3) * 1;
      %y(i-1) = One Day Ahead, yhat(i-1) = Multiple Day Ahead
   end

   %Recurrent Perceptron = Multiple Day Ahead
   rec = newff([-500 500;-500 500], [2 1], {'purelin' 'purelin'});
   rec.trainParam.epochs = 50;
   rec.trainParam.mu_max = 1.0000e+010;
   rec.IW{1} = [1 0;0 0];
   rec.inputWeights{1}.learn = 0;
   rec.layerConnect = [0 1;1 0];
   rec.LW{1,2} = [0;1];
   rec.layerWeights{1,2}.delays = 1;
   rec.layerWeights{1,2}.learn = 0;
   rec.biasConnect = [0;1];
   input = mat2cell([x' zeros(len,1)]',2,ones(len,1));
   y = mat2cell(y,1,ones(1,len));
   [rec,a,e,pf] = train(rec,input,y);
   [ynet pf af] = sim(rec,input);
   ynet = cell2mat(ynet);
   y = cell2mat(y);
   prec = [rec.LW{2,1}(1) ; rec.LW{2,1}(2) ; rec.b{2}]

   %Find Error and Output Results
   err = (yhat - y);
   se = sqrt(sum(err.^2)/(len-3));
   sy = sqrt(sum(y.^2)/(len-1));
   sesyr = se / sy
   nerr = (ynet - y);
   se = sqrt(sum(nerr.^2)/(len-3));
   sy = sqrt(sum(y.^2)/(len-1));
   sesyc = se / sy
   close
   plot(t,y,t,yhat,t,ynet)
end
```

```
function [rec] = armauniterr()
   %This code tests AR(1,1) and the recurrent perceptron.
   %Also uses the error term.

   %Target Time Series
   len = 300;
   x = randn(1,len);
   t = [1:len];
   y(1) = 5 * x(1) + 1 * 0 - 1.5 + 0 * randn;
   for i = 2:len
      y(i) = 5 * x(i) + 1 * y(i-1) - 1.5 + 0 * randn;
   end

   %Long-AR Method for ARMA(1,1)
   j = 7;
   for i = 1:j
      ym((i+1):len,i) = y(1:(len-i));
   end
   input = ym;
   k = input \ y';
   khat = zeros(len,1);
   for i = 1:j
      khat = khat + k(i) * ym(:,i);
   end
   ehat = (khat - y');
   ehatm1(2:len) = ehat(1:(len-1));
   input = [ehatm1' ym(:,1) ones(len,1)];
   pq = input \ y'
   yhat(1) = pq(1) * 0 + pq(2) * 0 + pq(3) * 1;
   for i = 2:len
      yhat(i) = pq(1) * (yhat(i-1) - y(i-1)) + pq(2) * yhat(i-1) + pq(3) * 1;
      %y(i-1) = One Day Ahead, yhat(i-1) = Infinite Day Ahead
   end

   %Recurrent Neural Network = Infinite Day Ahead
   rec = newff([-500 500;-500 500], [2 1], {'purelin' 'purelin'});
   rec.trainParam.epochs = 50;
   rec.trainParam.mu_max = 1.0000e+010;
   rec.IW{1} = [1 0;0 0];
   rec.inputWeights{1}.learn = 0;
   rec.layerConnect = [0 1;1 0];
   rec.LW{1,2} = [-1;1];
   rec.layerWeights{1,2}.delays = 1;
   rec.layerWeights{1,2}.learn = 0;
   rec.biasConnect = [0;1];
   input = mat2cell([ym(:,1) zeros(len,1)]',2,ones(len,1));
   y = mat2cell(y,1,ones(1,len));
   [rec,a,e,pf] = train(rec,input,y);
   [ynet pf af] = sim(rec,input);
   ynet = cell2mat(ynet);
   y = cell2mat(y);
   prec = [rec.LW{2,1}(1) ; rec.LW{2,1}(2) ; rec.b{2}]

   % Find Error and Output Results
   err = (yhat - y);
   se = sqrt(sum(err.^2)/(len-3));
```

```
    sy = sqrt(sum(y.^2)/(len-1));
    sesyr = se / sy
    nerr = (ynet - y);
    se = sqrt(sum(nerr.^2)/(len-3));
    sy = sqrt(sum(y.^2)/(len-1));
    sesyc = se / sy
    close
    plot(t,y,t,yhat,t,ynet)
end
```

```
function [net] = armapnet()
   %This code tests the ARMA(3,0) equation and recurrent perceptron.

   %Target Time Series
   len = 150;
   t = [1:len];
   change = 0.05;
   y(1) = 0.5 + change * 0 - 0.70 * 0 + 0.45 * 0;
   y(2) = 0.5 + change * y(1) - 0.70 * 0 + 0.45 * 0;
   y(3) = 0.5 + change * y(1) - 0.70 * y(2) + 0.45 * 0;
   for i = 4:len
      y(i) = 0.5 + change * y(i-1) - 0.70 * y(i-2) + 0.45 * y(i-3) + 0 * randn;
      ym1(i) = y(i-1);
      ym2(i) = y(i-2);
      ym3(i) = y(i-3);
   end

   %Recurrent Perceptron Training
   net = newff([-0.9 0.9;-0.9 0.9;-0.9 0.9], [3 1], {'purelin' 'purelin'});
   net.trainParam.epochs = 500;
   net.trainParam.mu_max = 1.0000e+010;
   net.biasConnect = [0;1];
   net.IW{1} = [0 0 0;0 0 0;0 0 0];
   net.inputWeights{1}.learn = 0;
   net.layerConnect = [0 1;1 0];
   net.layerWeights{1,2}.delays = [1 2 3];
   net.LW{1,2} = [1 0 0;0 1 0;0 0 1];
   net.layerWeights{1,2}.learn = 0;
   input = zeros(3,len);
   input = mat2cell(input,3,ones(len,1));
   target = mat2cell(y,1,ones(1,len));
   net = train(net,input,target);
   yr = sim(net,input);
   yr = cell2mat(yr);
   wb = [net.b{2};net.LW{2,1}']

   %ARMA(3,0) Estimation
   input = [ones(len,1) ym1' ym2' ym3'];
   pq = input \ y'
   yhat(1) = pq(1) + pq(2) * 0 + pq(3) * 0 + pq(4) * 0;
   yhat(2) = pq(1) + pq(2) * yhat(1) + pq(3) * 0 + pq(4) * 0;
   yhat(3) = pq(1) + pq(2) * yhat(1) + pq(3) * yhat(2) + pq(4) * 0;
   for i = 4:len
      yhat(i) = pq(1) + pq(2) * yhat(i-1) + pq(3) * yhat(i-2) + pq(4) * yhat(i-3);
      %y(i-1) = One Day Ahead, yhat(i-1) = Infinite Day Ahead
   end

   %Find Error and Output Results
   nerr = (yr - y);
   se = sqrt(sum(nerr.^2)/(len-4));
   sy = sqrt(sum(y.^2)/(len-1));
   sesyc = se / sy
   err = (yhat - y);
   se = sqrt(sum(err.^2)/(len-4));
   sy = sqrt(sum(y.^2)/(len-1));
   sesyr = se / sy
```

```
    close
    plot(t,y,t,yr,t,yhat)
end
```

```
function [rec] = armanonlin()
   %This code tests for NARMA(1,0) equation
   len = 300;
   t = [1:len];
   y(1) = -1.2 * 0 + 0.3 * 0 + 1 * 0 + 0.6 + 0 * randn;
   for i = 2:len
      y(i) = -1.2 * y(i-1) + 0.3 * y(i-1)^2 + 1 * y(i-1)^3 + 0.6 + 0 * randn;
   end

   %Set up NARMA(1,0)
   j = 1;
   for i = 1:j
      ym((i+1):len,i) = y(1:(len-i));
   end
   input = [ones(len,1) ym(:,1) ym(:,1).^2 ym(:,1).^3];
   pq = input \ y'
   yhat(1) = pq(1) * 1 + pq(2) * 0 + pq(3) * 0 + pq(4) * 0;
   for i = 2:len
      yhat(i) = pq(1) * 1 + pq(2) * yhat(i-1) + pq(3) * yhat(i-1)^2 + pq(4) * yhat(i-1)^3;
      %y(i-1) = One Day Ahead, yhat(i-1) = Infinite Day Ahead
   end

   %Set up Recurrent Neural Network = Infinite Day Ahead
   rec = newff([-1 1], [1 2 1], {'purelin' 'tansig' 'purelin'});
   rec.trainParam.epochs = 50;
   rec.trainParam.mu_max = 1.0000e+010;
   rec.IW{1} = [0];
   rec.inputWeights{1}.learn = 0;
   rec.layerConnect = [0 0 1;1 0 0;0 1 0];
   rec.LW{1,3} = [1];
   rec.layerWeights{1,3}.delays = 1;
   rec.layerWeights{1,3}.learn = 0;
   rec.biasConnect = [0;1;1];
   input = mat2cell([zeros(len,1)]',1,ones(len,1));
   y = mat2cell(y,1,ones(1,len));
   [rec,a,e,pf] = train(rec,input,y);
   [ynet pf af] = sim(rec,input);
   ynet = cell2mat(ynet);
   y = cell2mat(y);
   w1 = rec.LW{2,1}(1);
   w2 = rec.LW{2,1}(2);
   w3 = rec.LW{3,2}(1);
   w4 = rec.LW{3,2}(2);
   b1 = rec.b{2}(1);
   b2 = rec.b{2}(2);
   b3 = rec.b{3};
   c0 = w3*b1 + w4*b2 - 1/3*w3*b1^3 - 1/3*w4*b2^3 + b3;
   c1 = w3*w1 + w4*w2 - w3*w1*b1^2 - w4*w2*b2^2;
   c2 = -w3*w1^2*b1 - w4*w2^2*b2;
   c3 = -1/3*w3*w1^3 - 1/3*w4*w2^3;
   [c0; c1; c2; c3]

   %Find Error and Output Results
   err = (yhat - y);
   se = sqrt(sum(err.^2)/(len-4));
   sy = sqrt(sum(y.^2)/(len-1));
```

```
    sesyr = se / sy
    nerr = (ynet - y);
    se = sqrt(sum(nerr.^2)/(len-7));
    sy = sqrt(sum(y.^2)/(len-1));
    sesyc = se / sy
    close
    plot(t,y,t,yhat,t,ynet)
end
```

```
function [store] = bigonevarnet(Q,P,T,Qt,Pt,Tt)
   %This code was used to train the FNN to estimate Function 1
   %for streamflow prediction, which uses only Pt as input.

   %Scale training inputs and outputs to a range of [-0.8, +0.8].
   smin = -0.8;
   smax = 0.8;
   for i = 1:size(Q)
      q(i) = ((smax - smin) * Q(i) + (max(Q) * smin - min(Q) * smax)) / (max(Q) - min(Q));
   end
   for i = 1:size(P)
      p(i) = ((smax - smin) * P(i) + (max(P) * smin - min(P) * smax)) / (max(P) - min(P));
      x = size(P);
      if (i < x(1))
         pm1(i+1) = p(i);
      end
   end
   pm1(1) = p(1);
   for i = 1:size(T)
      t(i) = ((smax - smin) * T(i) + (max(T) * smin - min(T) * smax)) / (max(T) - min(T));
   end

   disp('FNN - 1 Lin - X Sig - 1 Lin + Biases')
   disp('Input - P, Output - Q, Data - WDA, 500 Epochs')
   disp('Scaled P, Scaled Q')

   %Test 1 through 45 hidden nodes.
   count = 0;
   for ii = 1:45
      for jj = 1:5
         hid = ii
         count = count + 1;
         store(count,1) = ii;
         %Set up the three-layer feed-forward network.
         net = newff([-0.9 0.9], [1 hid 1], {'purelin' 'tansig' 'purelin'});
         net.trainParam.epochs = 500;
         net.trainParam.mu_max = 1.0000e+010;
         net.biasConnect = [0;1;1];
         net.IW{1} = [1];
         net.inputWeights{1}.learn = 0;

         %Train network.
         input = [p];
         net = train(net,input,q);

         %Simulate network.
         yn = sim(net,input);

         %Rescale output.
         x = size(yn);
         for i = 1:x(2)
            Yn(i) = ((max(Q) - min(Q)) * yn(i) - (max(Q) * smin - min(Q) * smax)) / (smax - smin);
         end

         %Determine error.
         %Change for # of hidden nodes and # of parameters.
```

```
        se = sqrt(sum((Yn - Q').^2)/(x(2)-(1+(1+1+1)*hid)));
        sy = sqrt(sum((Q - mean(Q)).^2)/(x(2)-1));
        sesyn = se/sy
        store(count,2) = sesyn;

        %Test on validation data.  Scale the data using same scale as the training data.
        for i = 1:size(Qt)
           qt(i) = ((smax - smin) * Qt(i) + (max(Q) * smin - min(Q) * smax)) / (max(Q) - min(Q));
        end
        for i = 1:size(Pt)
           pt(i) = ((smax - smin) * Pt(i) + (max(P) * smin - min(P) * smax)) / (max(P) - min(P));
           x = size(Pt);
           if (i < x(1))
              pm1t(i+1) = pt(i);
           end
        end
        pm1t(1) = pt(1);
        for i = 1:size(Tt)
           tt(i) = ((smax - smin) * Tt(i) + (max(T) * smin - min(T) * smax)) / (max(T) - min(T));
        end

        input = [pt];
        ynt = sim(net,input);

        %Rescale output.
        x = size(ynt);
        for i = 1:x(2)
           Ynt(i) = ((max(Q) - min(Q)) * ynt(i) - (max(Q) * smin - min(Q) * smax)) / (smax - smin);
        end

        %Determine error.
        %Change for # of hidden nodes and # of parameters.
        se = sqrt(sum((Ynt - Qt').^2)/(x(2)-(1+(1+1+1)*hid)));
        sy = sqrt(sum((Qt - mean(Qt)).^2)/(x(2)-1));
        sesyn = se/sy
        store(count,3) = sesyn;
        store(count,4) = (1+(1+1+1)*hid);
     end
   end
   store
end
```

```
function [store net rec] = bigrecurnet(Q,P,T,Qt,Pt,Tt)
   %This code was used to train the FNN to estimate Function 8
   %Scale inputs and outputs to a range of [-0.8, +0.8]
   for i = 1:size(Q)
      q(i) = ((0.8 - -0.8) * Q(i) + (max(Q) * -0.8 - min(Q) * 0.8)) / (max(Q) - min(Q));
      x = size(Q);
      if (i < x(1))
         qm1(i+1) = q(i);
      end
   end
   qm1(1) = q(1);
   for i = 1:size(P)
      p(i) = ((0.8 - -0.8) * P(i) + (max(P) * -0.8 - min(P) * 0.8)) / (max(P) - min(P));
      x = size(P);
      if (i < x(1))
         pm1(i+1) = p(i);
      end
   end
   pm1(1) = p(1);
   for i = 1:size(T)
      t(i) = ((0.8 - -0.8) * T(i) + (max(T) * -0.8 - min(T) * 0.8)) / (max(T) - min(T));
   end

   disp('FNN - 4 Lin - X Sig - 1 Lin + Biases (One Day Ahead)')
   disp('RNN - 4 Lin - X Sig - 1 Lin + Biases (Multi Day Ahead)')
   disp('Input - P, P-1, T, Q-1, Output - Q, Data - WDA')

   count = 0;
   for ii = 1:30
      for jj = 1:5
         hid = ii
         count = count + 1;
         store(count,1) = ii;
         %Set up the feed-forward network
         net = newff([-0.9 0.9;-0.9 0.9;-0.9 0.9;-0.9 0.9], [4 hid 1], {'purelin' 'tansig' 'purelin'});
         net.trainParam.epochs = 500;
         net.biasConnect = [0;1;1];
         net.IW{1} = [1 0 0 0; 0 1 0 0; 0 0 1 0; 0 0 0 1];
         net.inputWeights{1}.learn = 0;

         %Train network
         input = [p; pm1; t; qm1];
         net = train(net,input,q);

         %Simulate network
         y2 = sim(net,input);

         %Set up the recurrent network
         rec = newff([-0.9 0.9;-0.9 0.9;-0.9 0.9;-0.9 0.9], [4 hid 1], {'purelin' 'tansig' 'purelin'});
         rec.trainParam.epochs = 50;
         rec.biasConnect = [0;1;1];
         rec.IW{1} = [1 0 0 0; 0 1 0 0; 0 0 1 0; 0 0 0 0];
         rec.inputWeights{1}.learn = 0;
         rec.layerConnect = [0 0 1; 1 0 0; 0 1 0];
         rec.LW{1,3} = [0;0;0;1];
         rec.layerWeights{1,3}.learn = 0;
```

```
rec.layerWeights{1,3}.delays = 1;

%Train + Simulate recurrent network
input = mat2cell([p; pm1; t; zeros(size(qm1))],4,ones(size(Q),1));
q = mat2cell(q,1,ones(1,size(Q)));
rec = train(rec,input,q);
y2c = sim(rec,input);
q = cell2mat(q);
y2c = cell2mat(y2c);

%Rescale output
x = size(y2);
for i = 1:x(2)
   Y2(i) = ((max(Q) - min(Q)) * y2(i) - (max(Q) * -0.8 - min(Q) * 0.8)) / (0.8 - -0.8);
end
x = size(y2c);
for i = 1:x(2)
   Y2c(i) = ((max(Q) - min(Q)) * y2c(i) - (max(Q) * -0.8 - min(Q) * 0.8)) / (0.8 - -0.8);
end

%Determine error
E2 = Y2' - Q;
se = sqrt(sum(E2.^2)/(x(2)-(1+(1+4+1)*hid)));
sy = sqrt(sum((Q - mean(Q)).^2)/(x(2)-1));
disp('FNN - Train')
sesy = se/sy
store(count,2) = se/sy;
E2 = Y2c' - Q;
se = sqrt(sum(E2.^2)/(x(2)-(1+(1+4+1)*hid)));
sy = sqrt(sum((Q - mean(Q)).^2)/(x(2)-1));
disp('RNN - Train')
sesy = se/sy
store(count,3) = se/sy;

%Test on validation data
for i = 1:size(Qt)
   qt(i) = ((0.8 - -0.8) * Qt(i) + (max(Q) * -0.8 - min(Q) * 0.8)) / (max(Q) - min(Q));
   x = size(Qt);
   if (i < x(1))
      qm1t(i+1) = qt(i);
   end
end
qm1t(1) = qt(1);
for i = 1:size(Pt)
   pt(i) = ((0.8 - -0.8) * Pt(i) + (max(P) * -0.8 - min(P) * 0.8)) / (max(P) - min(P));
   x = size(Pt);
   if (i < x(1))
      pm1t(i+1) = pt(i);
   end
end
pm1t(1) = pt(1);
for i = 1:size(Tt)
   tt(i) = ((0.8 - -0.8) * Tt(i) + (max(T) * -0.8 - min(T) * 0.8)) / (max(T) - min(T));
end

input = [pt; pm1t; tt; qm1t];
```

```
        yt = sim(net,input);

        input = mat2cell([pt; pm1t; tt; zeros(size(qm1t))],4,ones(size(Qt),1));
        ytc = sim(rec,input);
        ytc = cell2mat(ytc);

        %Rescale output
        x = size(yt);
        for i = 1:x(2)
           Yt(i) = ((max(Q) - min(Q)) * yt(i) - (max(Q) * -0.8 - min(Q) * 0.8)) / (0.8 - -0.8);
        end
        x = size(ytc);
        for i = 1:x(2)
           Ytc(i) = ((max(Q) - min(Q)) * ytc(i) - (max(Q) * -0.8 - min(Q) * 0.8)) / (0.8 - -0.8);
        end

        %Determine error
        Et = Yt' - Qt;
        se = sqrt(sum(Et.^2)/(x(2)-(1+(1+4+1)*hid)));
        sy = sqrt(sum((Qt - mean(Qt)).^2)/(x(2)-1));
        disp('FNN - Valid')
        sesy = se/sy
        store(count,4) = se/sy;
        Et = Ytc' - Qt;
        se = sqrt(sum(Et.^2)/(x(2)-(1+(1+4+1)*hid)));
        sy = sqrt(sum((Qt - mean(Qt)).^2)/(x(2)-1));
        disp('RNN - Valid')
        sesy = se/sy
        store(count,5) = se/sy;
        store(count,6) = (1+(1+4+1)*hid);
        store(count,7) = (1+(1+4+1)*hid);
      end
   end
   store
end
```

```
function [output] = bignthreg(Q,P,T,Qt,Pt,Tt)
   %This code is used to model any combination of inputs
   %and X number of orders for predicting streamflow.
   counter = 0;
   smin = -0.8;
   smax = 0.8;
   for ii = 1:15
      for jj = 1:1
         clear store input
         clear q qm1 p pm1 t yr Yr
         %Scale training inputs and outputs to a range of [-0.8, +0.8].
         for i = 1:size(Q)
            q(i) = ((smax - smin) * Q(i) + (max(Q) * smin - min(Q) * smax)) / (max(Q) - min(Q));
            x = size(Q);
            if (i < x(1))
               qm1(i+1) = q(i);
            end
         end
         for i = 1:size(P)
            p(i) = ((smax - smin) * P(i) + (max(P) * smin - min(P) * smax)) / (max(P) - min(P));
            x = size(P);
            if (i < x(1))
               pm1(i+1) = p(i);
            end
         end
         pm1(1) = p(1);
         for i = 1:size(T)
            t(i) = ((smax - smin) * T(i) + (max(T) * smin - min(T) * smax)) / (max(T) - min(T));
         end

         num = size(Q);
         if (num(1) > num(2))
            num = num(1);
         else
            num = num(2);
         end

         %Set up regression equation for n orders and x input variables.
         n = ii
         counter = counter + 1;
         output(counter,1) = ii;
         s = sym('p+pm1+t+qm1+1');
         s = expand(s^n);
         s = char(s);
         len = size(s);
         len = len(2);
         temp = '';
         count = 1;
         for i = 1:len
            if (s(i) ~= '+')
               if ((s(i) == '^') || (s(i) == '*'))
                  temp = strcat(temp,'.');
               end
               temp = strcat(temp,s(i));
            else
               store(count) = cellstr(temp);
```

```
            count = count + 1;
            temp = '';
        end
    end
    store(count) = cellstr(temp);
    len = size(store);
    len = len(2);
    for j = 1:len
        temp = char(store(j));
        if ((temp(1) == '1') || (temp(1) == '2') || (temp(1) == '3') || (temp(1) == '4') || (temp(1) == '5') ||
(temp(1) == '6') || (temp(1) == '7') || (temp(1) == '8') || (temp(1) == '9'))
            l = size(temp);
            l = l(2);
            tem = '';
            star = 0;
            for k = 1:l;
                if ((temp(k) == '*') && (star == 0))
                    star = 1;
                elseif (star == 1)
                    tem = strcat(tem,temp(k));
                else
                    %Do nothing
                end
            end
            store(j) = cellstr(tem);
        end
    end

    %Set up input matrix for regression parameter estimation.
    for i = 1:len
        if (strcmp(cell2mat(store(i)),''))
            input(:,i) = ones(num,1);
        else
            input(:,i) = eval(cell2mat(store(i)))';
        end
    end

    %Estimate regression parameters.
    %store
    %Linear
    c = input \ q';
    %Nonlinear
    %c = input \ atanh(q)';
    size(c)

    %One Day Ahead
    yr = zeros(num,1);
    for j = 1:len
        if (strcmp(cell2mat(store(j)),''))
            yr = yr + c(j) * ones(num,1);
        else
            yr = yr + c(j) * eval(cell2mat(store(j)))';
        end
    end
    %Nonlinear
    %yr = tanh(yr);
```

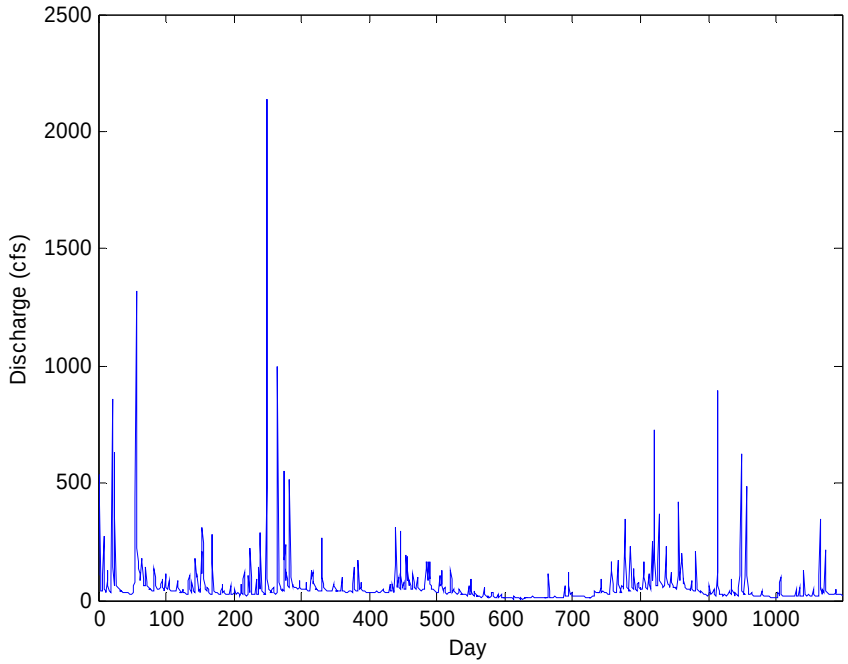
```
%Multi Day Ahead
%multi = store;
%for j = 1:len
%    temp = char(multi(j));
%    l = size(temp);
%    l = l(2);
%    k = 1;
%    while k <= l;
%        if (temp(k) == 'p')
%            temp = strcat(temp(1:k),'(m)',temp(k+1:l));
%        end
%        if (temp(k) == 'q')
%            temp = strcat(temp(1:k-1),'yrm(m-1)',temp(k+3:l));
%        end
%        l = size(temp);
%        l = l(2);
%        k = k + 1;
%    end
%    multi(j) = cellstr(temp);
%end
%yrm = zeros(num,1);
%for m = 2:num
%    for j = 1:len
%        if (strcmp(cell2mat(multi(j)),''))
%            yrm(m) = yrm(m) + c(j) * 1;
%        else
%            yrm(m) = yrm(m) + c(j) * eval(cell2mat(multi(j)));
%        end
%    end
%end

%Rescale output
for i = 1:num
   Yr(i) = ((max(Q) - min(Q)) * yr(i) - (max(Q) * smin - min(Q) * smax)) / (smax - smin);
end

for i = 1:num
   %Yrm(i) = ((max(Q) - min(Q)) * yrm(i) - (max(Q) * smin - min(Q) * smax)) / (smax - smin);
end

se = sqrt(sum((Yr - Q').^2)/(1097-len));
sy = sqrt(sum((Q - mean(Q)).^2)/(1097-1));
sesyr = se/sy
output(counter,2) = se/sy;

%se = sqrt(sum((Yrm - Q').^2)/(1097-len));
%sy = sqrt(sum((Q - mean(Q)).^2)/(1097-1));
%sesyr = se/sy
%output(counter,3) = se/sy;

%figure
%t = 1:num;
%plot(t,Q,t,Yr)
%Yrt = Yr;
```

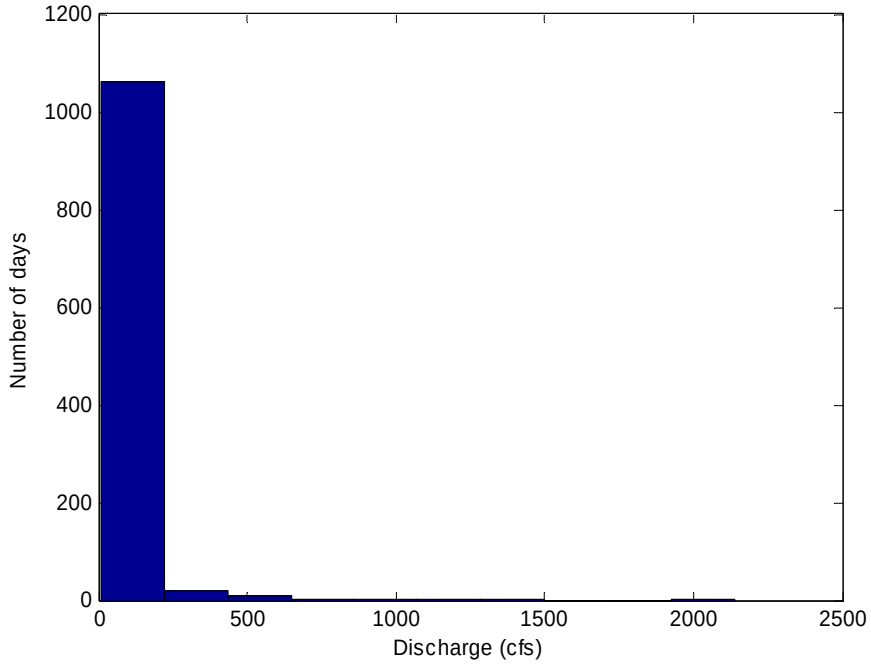
```
%Test on validation data.  Scale the data using same scale as the training data.
clear q qm1 p pm1 t yr yrm Yr Yrm
for i = 1:size(Qt)
   q(i) = ((smax - smin) * Qt(i) + (max(Q) * smin - min(Q) * smax)) / (max(Q) - min(Q));
   x = size(Qt);
   if (i < x(1))
      qm1(i+1) = q(i);
   end
end
for i = 1:size(Pt)
   p(i) = ((smax - smin) * Pt(i) + (max(P) * smin - min(P) * smax)) / (max(P) - min(P));
   x = size(Pt);
   if (i < x(1))
      pm1(i+1) = p(i);
   end
end
pm1(1) = p(1);
for i = 1:size(Tt)
   t(i) = ((smax - smin) * Tt(i) + (max(T) * smin - min(T) * smax)) / (max(T) - min(T));
end

num = size(Qt);
if (num(1) > num(2))
   num = num(1);
else
   num = num(2);
end

%One Day Ahead
yr = zeros(num,1);
for j = 1:len
   if (strcmp(cell2mat(store(j)),''))
      yr = yr + c(j) * ones(num,1);
   else
      yr = yr + c(j) * eval(cell2mat(store(j)))';
   end
end
%Nonlinear
%yr = tanh(yr);

%Multi Day Ahead
%yrm = zeros(num,1);
%for m = 2:num
%    for j = 1:len
%        if (strcmp(cell2mat(multi(j)),''))
%            yrm(m) = yrm(m) + c(j) * 1;
%        else
%            yrm(m) = yrm(m) + c(j) * eval(cell2mat(multi(j)));
%        end
%    end
%end

%Rescale output
for i = 1:num
   Yr(i) = ((max(Q) - min(Q)) * yr(i) - (max(Q) * smin - min(Q) * smax)) / (smax - smin);
end
```

```
        for i = 1:num
           %Yrm(i) = ((max(Q) - min(Q)) * yrm(i) - (max(Q) * smin - min(Q) * smax)) / (smax - smin);
        end

        se = sqrt(sum((Yr - Qt').^2)/(1096-len));
        sy = sqrt(sum((Qt - mean(Qt)).^2)/(1096-1));
        sesyr = se/sy
        output(counter,3) = se/sy;

        %se = sqrt(sum((Yrm - Qt').^2)/(1096-len));
        %sy = sqrt(sum((Q - mean(Qt)).^2)/(1096-1));
        %sesyr = se/sy
        %output(counter,5) = se/sy;
        output(counter,4) = len;

        %figure
        %t = 1:num;
        %plot(t,Qt,t,Yr)
        %Yrv = Yr;
      end
   end
   output
end
```

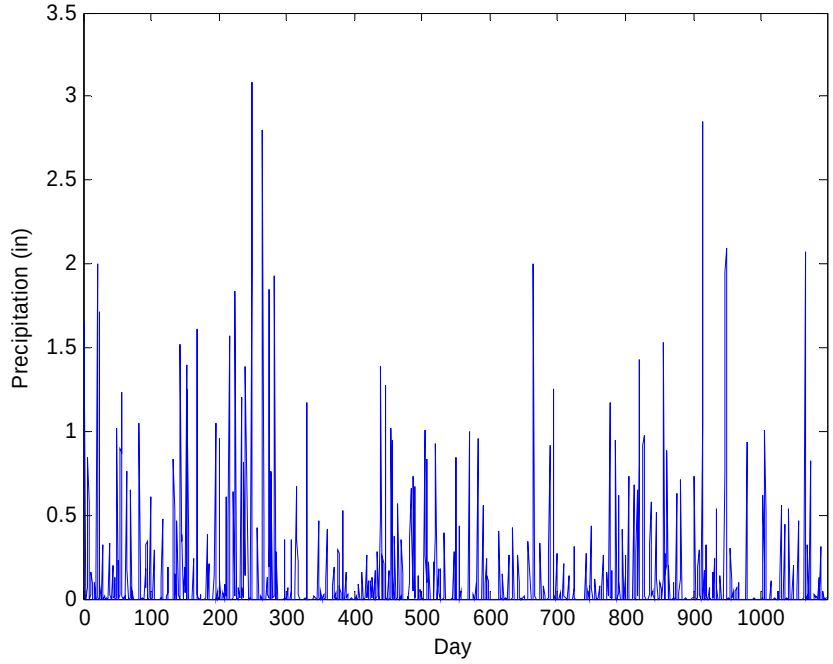
Appendix B - Little Patuxent River Watershed Data

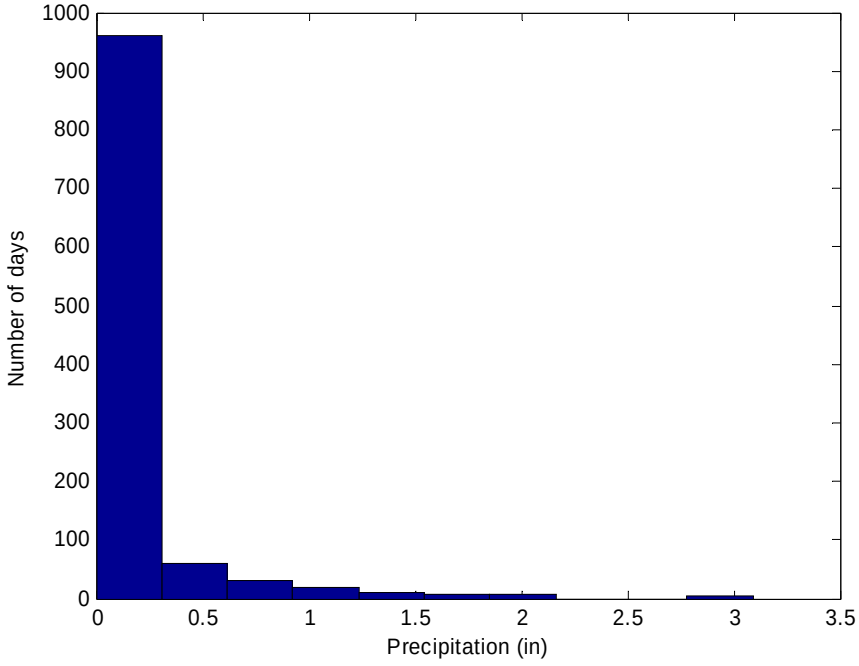
Training Data - Streamflow



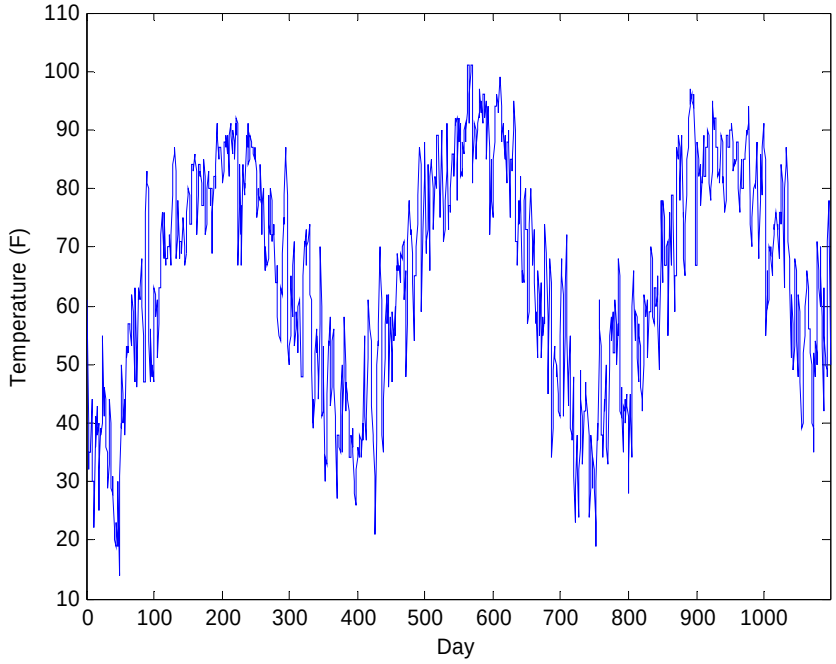


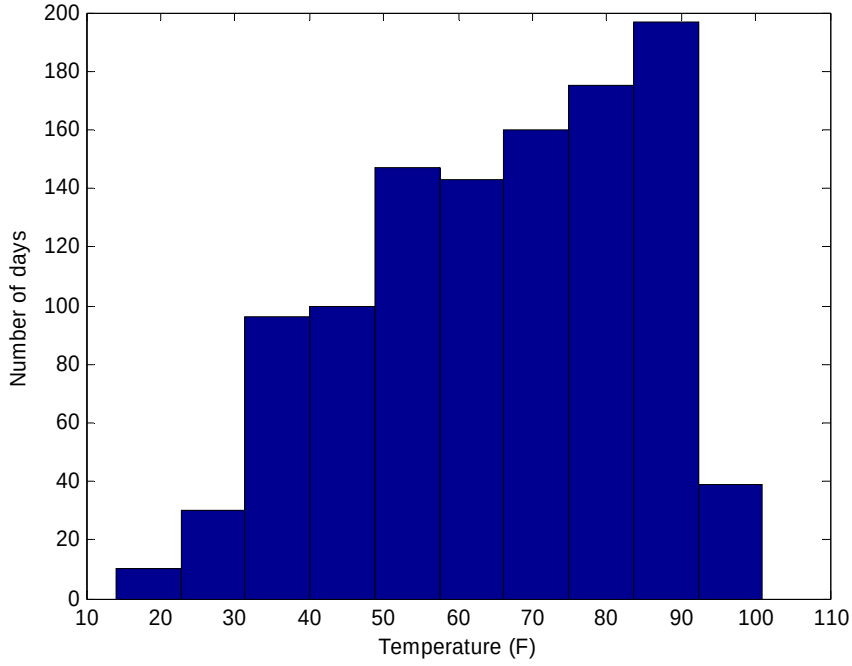
Training Data - Precipitation



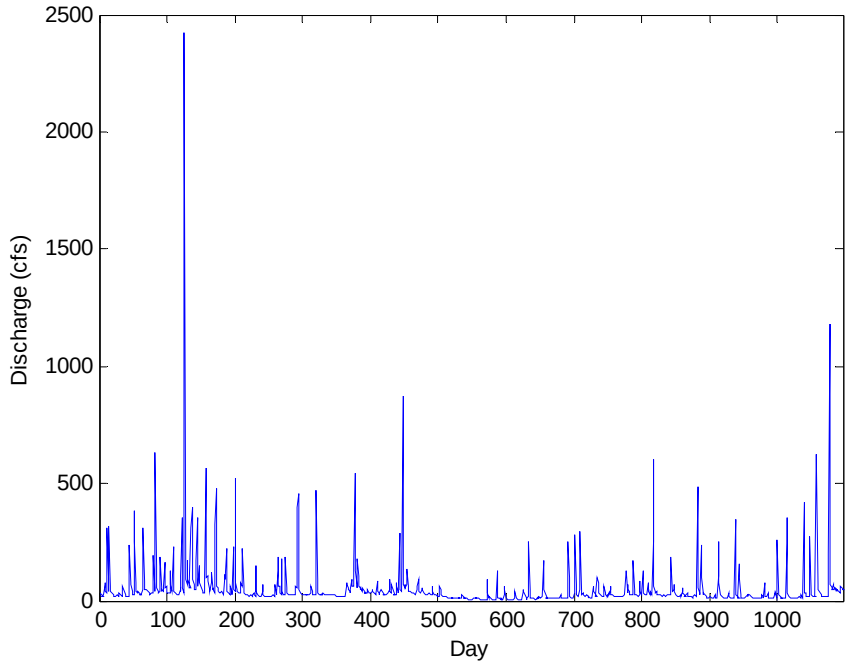


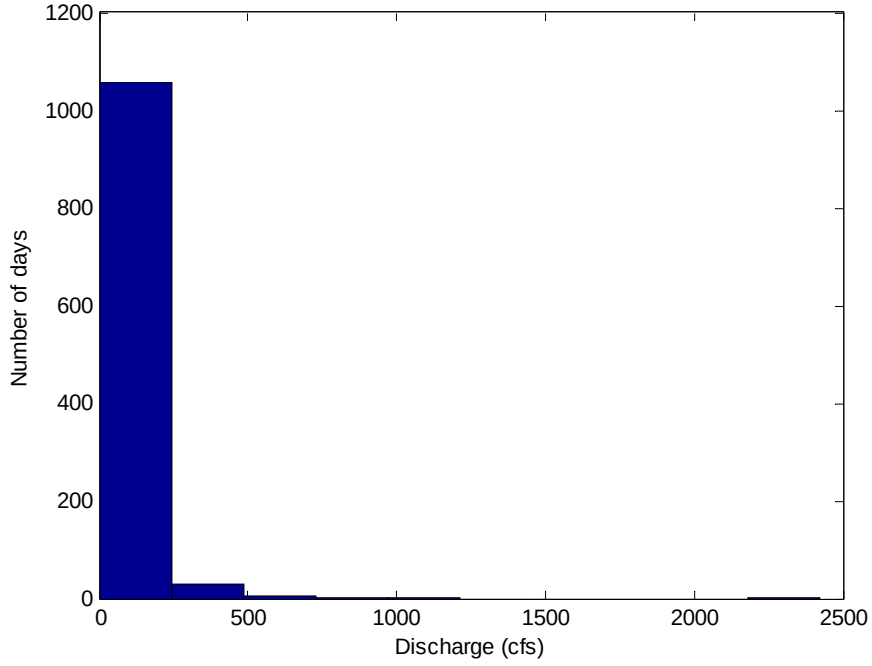
Training Data - Temperature



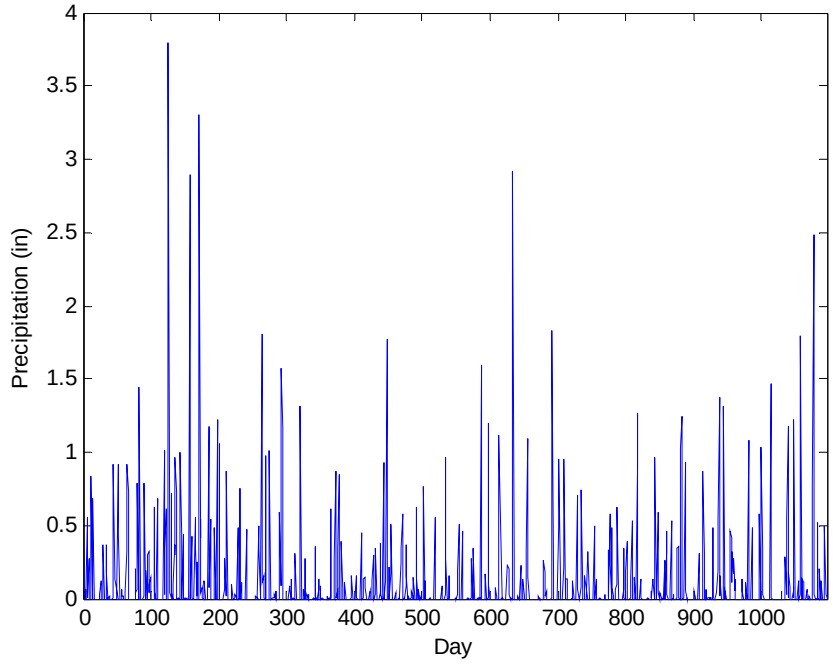


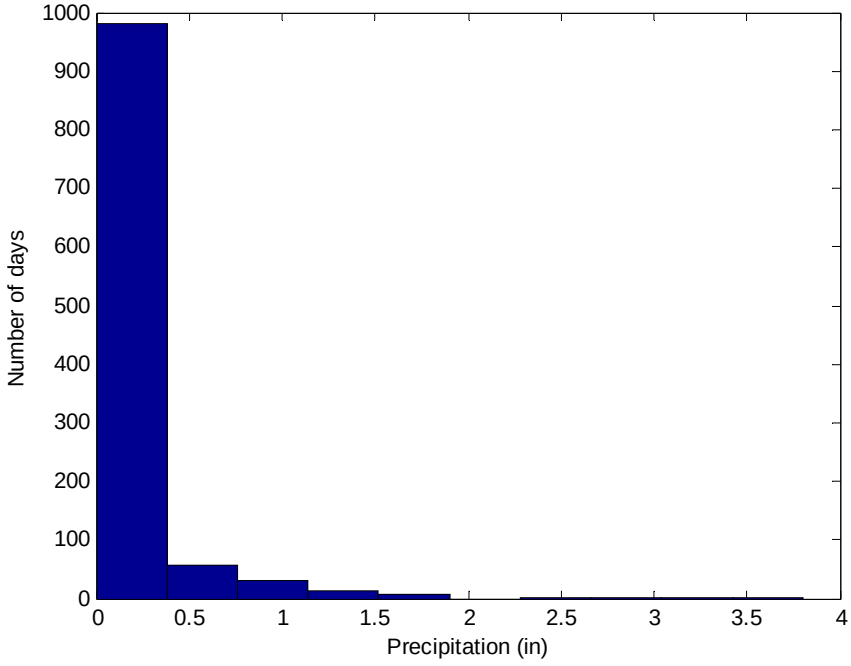
Validation Data - Streamflow



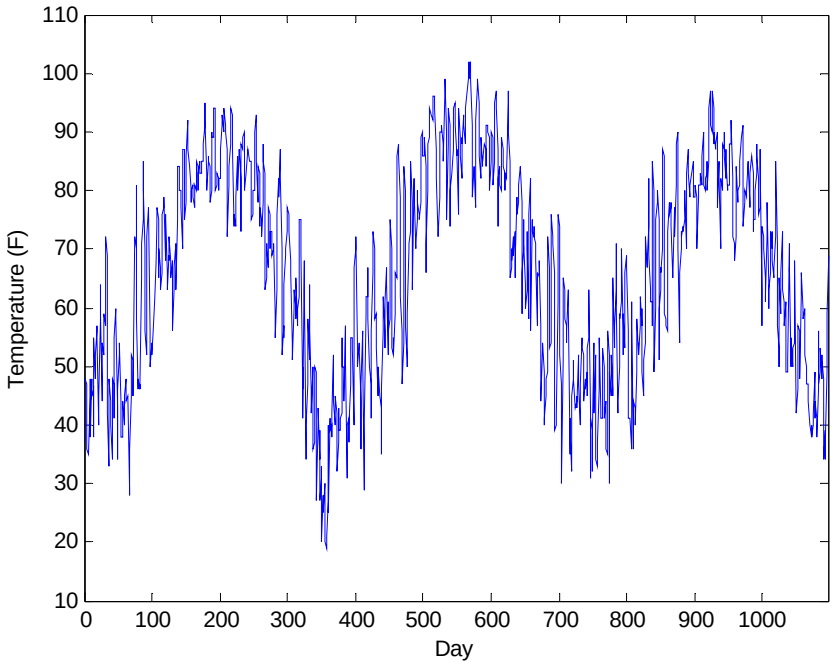


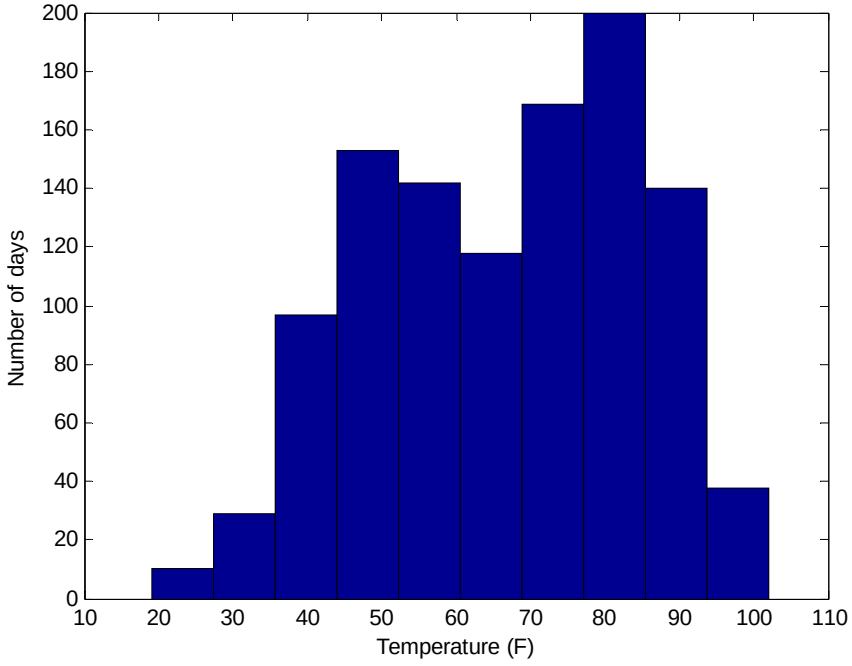
Validation Data - Precipitation





Validation Data - Temperature





References

ASCE Task Committee on Application of Artificial Neural Networks in Hydrology. "Artificial Neural Networks in Hydrology. I: Preliminary Concepts." Journal of Hydrologic Engineering. (April 2000a): 115 - 123.

ASCE Task Committee on Application of Artificial Neural Networks in Hydrology. "Artificial Neural Networks in Hydrology. II: Hydrologic Applications." Journal of Hydrologic Engineering. (April 2000b): 124 - 137.

Anmala, Jagadeesh, B. Zhang, and R. S. Govindaraju. "Comparison of ANNs and Empirical Approaches for Predicting Watershed Runoff." Journal of Water Resources Planning and Management. (June 2000): 156 - 166.

Archer, N. P. and S. Wang. "Application of the Back Propagation Neural Network Algorithm with Monoticity Constraints for Two-Group Classification Problems." Decision Sciences. Vol. 24. (1993): 60 - 75.

Ayyub, Bilal M. and R. H. McCuen. Probability, Statistics, and Reliability for Engineers and Scientists. Boca Raton: Chapman & Hall/CRC Press, 2003.

Barro, S., M. Fernandez-Delgado, J. A. Vila-Sobrino, C. V. Regueiro and E. Sanchez. "Classifying Multichannel ECG Patterns with an Adaptive Neural Network." IEEE Engineering in Medicine and Biology. (January 1998): 45 - 55.

Basheer, I. A. and M. Hajmeer. "Artificial Neural Networks: Fundamentals, Computing, Design, and Application." Journal of Microbiological Methods. Vol. 43. (2000): 3 - 31.

Bradley, Ralph A. and S. S. Srivastava. "Correlation in Polynomial Regression." The American Statistician. Vol. 33. No. 1 (February 1979): 11 - 14.

Brockwell, Peter J. and R. A. Davis. Introduction to Time Series and Forecasting. New York: Springer, 2002.

Cajal, S. R. Texture of the Nervous System of Man and the Vertebrates. New York: Springer, 1999.

Cheng, Bing and D. M. Titterington. "Neural Networks: A Review from a Statistical Perspective." Statistical Science. Vol. 9. No. 1 (February 1994): 2 - 30.

Cherkassky, Vladimir, J. H. Friedman, H. Wechsler. From Statistics to Neural Networks. Berlin: Springer-Verlag, 1993.

Chon, Ki H. and R. J. Cohen. "Linear and Nonlinear ARMA Model Parameter Estimation Using an Artificial Neural Network." IEEE Transactions on Neural Networks. Vol. 44. No. 3 (March 1997): 168 - 174.

Cobourn, W. G., L. Dolcine, M. French, and M. C. Hubbard. "A Comparison of Nonlinear Regression and Neural Network Models for Ground-Level Ozone Forecasting." Journal of the Air and Waste Management Association. Vol. 50. (November 2000): 1999 - 2009.

Conner, James T. and R. Douglas Martin. "Recurrent Neural Networks and Robust Times Series Prediction." IEEE Transactions on Neural Networks. Vol. 5. No. 2 (March 1994): 240 - 254.

Coppola, Emery A., A. J. Rana, M. M. Poulton, F. Szidarovszky and V. W. Uhl. "A Neural Network Model for Predicting Aquifer Water Level Elevations." Ground Water. Vol. 43. No. 2 (2005): 231 - 241.

Couvreur, Christophe and P. Couvreur. "Neural Networks and Statistics: A Naive Comparison." Belgian Journal of Operations Research, Statistics and Computer Science. Vol. 36. (1997): 217 - 225.

Daqi, Gao and Y. Genxing. "Influences of Variable Scales and Activation Functions on the Performance of Multilayer Feedforward Neural Networks." Pattern Recognition. Vol. 36. (2003): 869 - 878.

Dedecker, Andy P., P. L. M. Goethals, T. D'heygere, M. Gevrey, S. Lek, and N. De Pauw. "Application of Artificial Neural Network Models to Analyze the Relationships Between Gammarus Pulex L. (Crustacea, Amphipoda) and River Characteristics." Environmental Modeling and Assessment. Vol. 111. (2005): 223 - 241.

Demuth, H. and M. Beale. Neural Network Toolbox User Guide Version 4.0. Natick: The MathWorks Inc., 2004.

Denton, J. W. "How Good are Neural Networks for Casual Forecasting?" Journal of Business Forecasting. Vol. 14. (1995): 17 - 20.

Ellacott, Steve and D. Bose. Neural Networks: Deterministic Methods of Anaylsis. London: International Thomson Computer Press, 1996.

Feng, Chang-Xue and X. Wang. "Surface Roughness Predictive Modeling: Nerual Networks Versus Regression." IIE Transactions. Vol. 35. (2003): 11 - 27.

Fletcher, L., V. Katkovnik, F. E. Steffens and A. P. Engelbrecht. "Optimizing the Number of Hidden Nodes of a Feedforward Artificial Neural Network." Proceedings, IEEE World Congress on Computational Intelligence, International Joint Conference on Neural Networks, Anchorage, AK. (1998): 1608 - 1612.

Gevrey, Muriel, I. Dimopoulos, and S. Lek. "Review and Comparison of Methods to Study the Contribution of Variables in Artificial Neural Networks." Ecological Modeling. Vol. 160. (2003): 249 - 264.

Gonzalez, Steven. "Neural Networks for Macroeconomic Forecasting: A Complimentary Approach to Linear Regression Models." Working Paper, Department of Finance, Canada, 2000.

Govindaraju, R. S. and A. R. Rao. "Artificial Neural Networks: A Passing Fad in Hydrology?" Journal of Hydrologic Engineering. (July 2000): 225 - 226.

Hagan, Martin T., H. B. Demuth, M. H. Beale. Neural Network Design. Boston: PWS Publishing Co., 2002.

Hill, Tim, L. Marquez, M. O'Connor, and W. Remus. "Artificial Neural Network Models for Forecasting and Decision Making." International Journal of Forecasting. Vol. 10. (1994): 5 - 15.

Hayashi, Yoichi, J. J. Buckley, and E. Czogala. "Fuzzy Neural Network with Fuzzy Signals and Weights." International Joint Conference on Neural Networks. (1992): 696 - 701.

Kaul, M., R. L. Hill, and C. Walthall. "Artificial Neural Network for Corn and Soybean Yield Prediction." Agricultural Systems. Vol. 85. (2005): 1 - 18.

Kisi, Ozgur. "Daily River Forecasting Using Artificial Neural Networks and Auto-Regressive Models." Turkish Journal of Engineering and Environmental Sciences. Vol. 29. (2005): 9 - 20.

Kravtsov, S., D. Kondrashov, and M. Ghil. "Multi-level Regression Modeling of Nonlinear Processes: Derivation and Applications to Climatic Variability." Journal of Climate. (2005).

Kruschke John K. and J. R. Movellan. "Benefits of Gain: Speeding Learning and Minimal Hidden Layers in Back-Propagation Networks." IEEE Transactions on Systems, Man and Cybernetics. Vol. 21. No. 1 (1991): 273 - 280.

Kumar, D. N., K. S. Raju, and T. Sathish. "River Flow Forecasting using Recurrent Neural Networks." Water Resources Management. Vol. 18. (2004): 143 - 161.

National Climatic Data Center (NCDC). <http://www.ncdc.noaa.gov/oa/ncdc.html>. Accessed March 2005.

Maier, Holger R. and G. C. Dandy. "Neural Networks for the Prediction and Forecasting of Water Resources Variables: A Review of Modeling Issues and Applications." Environmental Modeling and Software. Vol. 15. (2000): 101 - 124.

Markham, I. S. and T. R. Rakes. "The Effect of Sample Size and Variability of Data on the Comparative Performance of Artificial Neural Networks and Regression." Computer Operation Resources. Vol. 25. (1998): 251 - 263.

Mandic, Danilo P. and J. A. Chambers. "Advanced RNN Based NARMA Predictors." Journal of VLSI Signal Processing. Vol. 26. (2000): 105 - 117.

Mandic, Danilo P. and J. A. Chambers. Recurrent Neural Networks for Prediction. Chichester: John Wiley & Sons, Inc., 2001.

MATLAB, The Language of Technical Computing, Version 7.0. Natick: The MathWorks Inc., 2004.

McCulloch, W. S. and W. Pitts. "A Logical Calculus of Ideas Immanent in Nervous Activity." Bulletin of Mathematical Biophysics. (1943): 115 - 133.

Mehrotra, Kishan, C. K. Mohan, and S. Ranka. Elements of Artificial Neural Networks. Cambridge: The MIT Press, 2000.

Menon, Anil, K. Mehrotra, C. K. Mohan, and S. Ranka. "Characterization of a Class of Sigmoid Functions with Applications to Neural Networks." Neural Networks. Vol. 9. No. 5 (1996): 819 - 835.

Minns, A. W. and M. J. Hall. "Artificial Neural Networks as Rainfall-Runoff Models." Hydrological Sciences Journal. Vol. 41. No. 3 (1996): 339 - 417.

Prybutok, Victor R., J. Yi and D. Mitchell. "Comparison of Neural Network Models with ARIMA and Regression Models for Prediction of Houston's Daily Maximum Ozone Concentrations." European Journal of Operational Research. Vol. 122. (2000): 31-40.

Ripley, B. D. Pattern Recognition and Neural Networks. Cambridge: Cambridge University Press, 1996.

Russell, Stuart and P. Norvig. Artificial Intelligence: A Modern Approach. Upper Saddle River, New Jersey: Pearson Education, Inc., 2003.

Salas, J. D., M. Markus, and A. S. Tokar, “Streamflow Forecasting Based on Artificial Neural Networks,” in Artificial Neural Networks in Hydrology, R. S. Govindaranu and A. R. Rao Editors, Kluwer, (2000): 23 - 51.

Sarle, Warren S. "Neural Networks and Statistical Methods." Proceedings of the Nineteenth Annual SAS Users Group International Conference. (April 1994).

Schnabel, Susanne and M. Maneta. "Comparison of a Neural Network and a Regression Model to Estimate Suspended Sediment in a Semiarid Basin." Proceedings of the International Conference, Catalonia, Spain. (May 2004): 91 - 100.

Sargent, Daniel J. "Comparison of Artificial Neural Networks with Other Statistical Approaches." Conference on Prognostic Factors and Staging in Cancer Management: Contributions of Artificial Neural Networks and Other Statistical Methods. Arlington, VA. (2001): 1636 - 1642.

Shigidi, Abdalla and L. A. Garcia. "Parameter Estimation in Groundwater Hydrology Using Artificial Neural Networks." Journal of Computing in Civil Engineering. (October 2003): 281 - 289.

Stigler, Stephen M. "Optimal Experimental Design for Polynomial Regression." Journal of the American Statistical Association. Vol. 66. No. 334 (June 1971): 311 - 318.

Subramanian, Narayanaswamy, A. Yajnik, and R.S. R. Murthy. "Artificial Neural Network as an Alternative to Multiple Regression Analysis in Optimizing Formulation Parameters of Cytarabine Liposomes." AAPS PharmSciTech. Vol. 5. No. 1 (November 2003): 1 - 9.

Tingsanchali, Tawatchai and M. R. Gautam. "Application of Tank, NAM, ARMA and Neural Network Models to Flood Forecasting." Hydrological Processes. Vol 14. (2000): 2473 - 2487.

Tokar, A. Sezin and P. A. Johnson. "Rainfall-Runoff Modeling Using Artificial Neural Networks." Journal of Hydrologic Engineering. (July 1999): 232 - 239.

U.S. Geological Survey (USDS). <http://nwis.waterdata.usgs.gov/nwis/>. Accessed March 2005.

Veelenturf, L. P. J. Analysis and Applications of Artificial Neural Networks. London: Prentice Hall, 1995.

Warner, Brad and M. Misra. "Understanding Neural Networks as Statistical Tools." The American Statistician. Vol. 50. No. 4 (November 1996): 284 - 293.

Widrow, Bernard. "30 Years of Adaptive Neural Networks: Perceptron, Madaline, and Backpropagation." Proceedings of the IEEE. Vol. 78. No. 9 (September 1990): 1415 - 1442.

Wolfram Research. "Time Series Documentation: Parameter Estimation." <http://documents.wolfram.com/applications/timeseries/>. Accessed January 2006.

White, Halbert. Artificial neural networks: Approximation and Learning Theory. Cambridge: Blackwell, 1992.

Xiang, Cheng, S. Q. Ding, and T. H. Lee. "Geometrical Interpretation and Architecture Selection of MLP." IEEE Transactions on Neural Networks. Vol. 16. No. 1 (January 2005): 84 - 96.

Yang, Zheng R. "A Novel Radial Basis Function Neural Network for Discriminant Analysis." IEEE Transactions on Neural Networks. Vol. 17. No. 3 (May 2006): 604 - 612.

Yao, Xin. "Evolving Artificial Neural Networks." Proceedings of the IEEE. Vol. 87. No. 9 (September 1999): 1423 - 1447.

Yitian, Li and R. R. Gu. "Modeling Flow and Sediment Transport in a River System Using an Artificial Neural Network." Environmental Management. Vol. 31. No. 1 (2003): 122 - 134.

Young, Evan F. and L. Chan. "Using Recurrent Network in Time Series Prediction." World Congress on Neural Networks. (1993): 1 - 5.

Zealand, Cameron M., D. H. Burn, and S. P. Simonovic. "Short Term Streamflow Forecasting Using Artificial Neural Networks." Journal of Hydrology. Vol. 214. (1999): 32 - 48.

Zhang, G. P. "Time Series Forecasting Using a Hybrid ARIMA and Neural Network Model." Neurocomputing. Vol. 50. (2003): 159 - 175.