GLOBAL OPTIMIZATION FOR THE FORWARD NEURAL NETWORKS AND THEIR APPLICATIONS

By

K. SUNIL MANOHAR REDDY * G. RAVINDRA BABU ** S. KRISHNA MOHAN RAO ***

** Assistant Professor, Department of Computer Science and Engineering, Matrusi Engineering College, Hyderabad, India.*
*** Professor, Department of Computer Science and Engineering, Trinity College of Engineering & Technology Karimnagar, India.*
**** Professor, Department of Computer Science and Engineering, Siddhartha Institute of Engineering & Technology, Hyderabad, India.*

ABSTRACT

This paper describes and evaluates several global optimization issues of Artificial Neural Networks (ANN) and their applications. In this paper, the authors examine the properties of the feed-forward neural networks and the process of determining the appropriate network inputs and architecture, and built up a short-term gas load forecast system - the Tell Future system. This system performs very well for short-term gas load forecasting, which is built based on various Back-Propagation (BP) algorithms. The standard Back-Propagation (BP) algorithm for training feed-forward neural networks have proven robust even for difficult problems. In order to forecast the future load from the trained networks, the history loads, temperature, wind velocity, and calendar information should be used in addition to the predicted future temperature and wind velocity. Compared to other regression methods, the neural networks allow more flexible relationships between temperature, wind, calendar information and load pattern. Feed-forward neural networks can be used in many kinds of forecasting in different industrial areas. Similar models can be built to make electric load forecasting, daily water consumption forecasting, stock and markets forecasting, traffic flow and product sales forecasting.

Keywords: Neural Networks, Feed Forward Networks, Recurrent Networks, Network Learning, Layered Networks, Back Propagation.

INTRODUCTION

Neural networks, more accurately called Artificial Neural Networks (ANN), are computational models that consist of a number of simple processing units that communicate by sending signals to each other over a large number of weighted connections. They were originally developed from the inspiration of human brains. In human brains, a biological neuron collects signal from the other neurons through a host of fine structures called dendrites. The neuron sends out spikes of electrical activity through a long, thin strand known as an axon, which splits into thousands of branches. At the end of each branch, a structure called a synapse, converts the activity from the axon into electrical effects that inhibit or excite activity in the connected neurons. When a neuron receives excitatory input that is sufficiently large compared with its inhibitory input, it sends a spike of electrical activity down its axon. Learning occurs by changing the effectiveness of the synapses, so that the influence of one neuron on other changes.

Like human brains, neural networks also consist of processing units (artificial neurons) and connections (weights) between them. The processing units, transport incoming information on their outgoing connections to other units. The "electrical" information is simulated with specific values stored in those weights that make these networks have the capacity to learn, memorize, and create relationships amongst the data.

A very important feature of these networks is their adaptive nature, where "learning by example" replaces "programming" in solving problems. This feature renders these computational models very appealing in application domains, where one has little or incomplete understanding of the problems to be solved, but where training data are available.

There are many different types of neural networks, and they are being used in many fields. And new uses for neural networks are devised daily by researchers. Some of the most traditional applications include [1], [2], [17].

- *Classification* – To determine military operations from satellite photographs; to distinguish among different types of radar returns (weather, birds, or aircraft); to identify diseases of the heart from electrocardiograms.
- *Noise reduction* – To recognize a number of patterns (voice, images, etc.) corrupted by noise.
- *Prediction* – To predict the value of a variable given historic value. Examples include forecasting of various types of loads, market and stock forecasting, and weather forecasting. The model built in this thesis falls into this category.

Fundamentals of Neural Networks

Neural networks, sometimes referred to as connectionist models, or parallel-distributed models that have several distinguishing features [3], [18]:

- A set of processing units;
- An activation state of each unit, which is equivalent to the output of the unit;
- Connections between the units. Generally, each connection is defined by a weight $w_{jk,}$ that determines the effect that the signal of unit j has on unit k;
- Propagation rule, which determines the effective input of the unit from its external inputs;
- An activation function, which determines the new level of activation based on the effective input and the current activation;
- An external input (bias, offset) for each unit;
- A method for information gathering (learning rule);
- An environment within which the system can operate, provides input signals and, if necessary, error signals.

Processing Unit

A processing unit (Figure 1), also called a neuron or node, performs a relatively simple job; it receives input from the neighbors or external sources and uses them to compute an output signal that is propagated to other units.

Within the neural systems, there are three types of units:

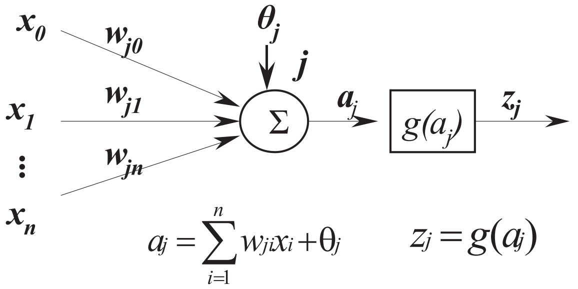


Figure 1. Processing Unit

- Input units, which receive data outside the network;
- Output units, which send data out of the network;
- Hidden units, whose input and output signals remain within the network [18].

Each unit j can have one or more inputs x_0, x_1, x_2, … x_n, but only one output z_j. An input to a unit is either the data outside the network, or the output of another unit, or its own output.

Combination Function

Each non-input unit in a neural network combines values that are fed into it via synaptic connections from other units, producing a single value called net input. The function that combines the values is called the combination function, which is defined by a certain propagation rule. In most neural networks, the authors assume that, each unit provides an additive contribution to the input of the unit with which it is connected. The total input to unit j is simply the weighted sum of the separate outputs from the connected units plus a threshold or bias term θ_j:

$$a_j = \sum_{i=1}^{n} w_{ji}x_i + \theta_j \tag{1}$$

The contribution for positive w_{ji} is considered as an excitation and an inhibition for negative w_{ji}. The units with the above propagation rules are termed as sigma units.

In some cases ,more complex rules for combining inputs are used. One of the propagation rule known as sigma-pi has the following format [3]:

$$a_j = \sum_{i=1}^{n} w_{ji} \prod_{k=1}^{m} x_{ik} + \theta_j \tag{2}$$

Lots of combination functions usually use a "bias" or "threshold" term in computing the net input to the unit. For a linear output unit, a bias term is equivalent to an intercept in a regression model. It is needed in much the same way as the constant polynomial '1' is required for approximation by polynomials.

Activation Function

Most units in neural network transform their net inputs by using a scalar-to-scalar function called an activation function, yielding a value called the unit's activation. Except possibly for output units, the activation value is fed to one or more other units. Activation functions with a bounded range are often called squashing functions. Some of the most commonly used activation functions are [4],[18]:

- Identity function (Figure 2)

$$g(x) = x \qquad (3)$$

It is obvious that, the input units use the identity function. Sometimes a constant is multiplied by the net input to form a linear function.

- Binary step function (Figure 3)

Also known as threshold function or Heaviside function. The output of this function is limited to one of the two values:

$$g(x) = \begin{cases} 1 & \text{if}(x \geq \theta) \\ 0 & \text{if}(x < \theta) \end{cases} \qquad (4)$$

This kind of function is often used in single layer networks.

- Sigmoid function (Figure 4)

$$g(x) = \frac{1}{1+e^{-x}} \qquad (5)$$

This function is especially advantageous for the use in neural networks trained by Back-Propagation; because it

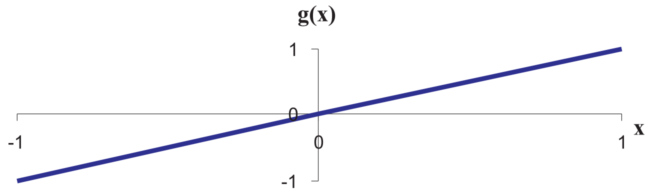


Figure 2. Identity Function

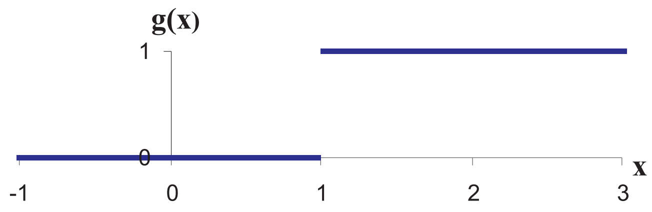


Figure 3. Binary Step Function

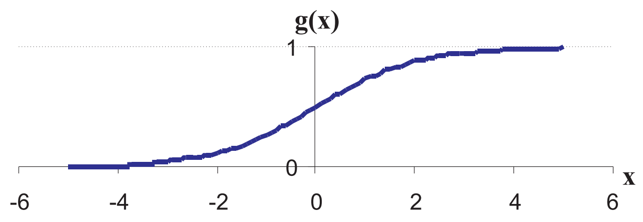


Figure 4. Sigmoid Function

is easy to differentiate, and thus can dramatically reduce the computation burden for training. It applies to applications whose desired output values are between 0 and 1.

- Bipolar sigmoid function (Figure 5)

$$g(x) = \frac{1-e^{-x}}{1+e^{-x}} \qquad (6)$$

This function has similar properties with the sigmoid function. It works well for applications that yield output values in the range of [-1,1].

Activation functions for the hidden units are needed to introduce non-linearity into the networks. The reason is that, a composition of linear functions is again a linear function. However, it is the non-linearity (i.e., the capability to represent nonlinear functions) that makes multi-layer networks so powerful. Almost any nonlinear function does the job, although for Back-Propagation learning, it must be differentiable and it helps if the function is bounded. The sigmoid functions are the most common choices [5].

For the output units, activation functions should be chosen to be suited to the distribution of the target values. The authors have already seen that, for binary [0,1] outputs, the sigmoid function is an excellent choice. For continuous-valued targets with a bounded range, the sigmoid functions are again useful, provided that, either the outputs or the targets to be scaled to the range of the output activation function. But if the target values have no known bounded range, it is better to use an unbounded activation function, most often the identity function (which amounts to no activation function). If the target values are

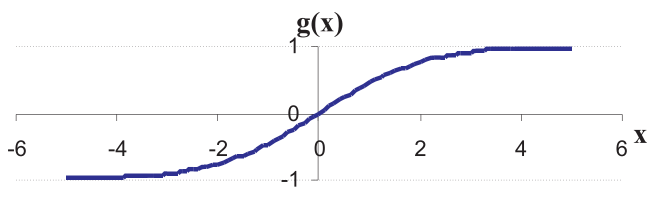


Figure 5. Bipolar Sigmoid Function

positive, but have no known upper bound, an exponential output activation function can be used [5].

Network Topologies

The number of layers, the number of units per layer, and the interconnection patterns between layers defines the topology of a network. They are generally divided into two categories based on the pattern of connections:

1. Feed-forward Networks: The data flow from input units to output units is strictly feed-forward. The data processing can extend over multiple layers of units, but no feedback connections are present. That is, connections extending from the outputs of units to inputs of units in the same layer or previous layers are not permitted. Feed-forward networks are the main focus of this thesis.

2. Recurrent Networks: Itcontains feedback connections. Contrary to feed-forward networks, the dynamical properties of the network are important. In some cases, the activation values of the units undergo a relaxation process such that, the network will evolve to a stable state in which the activation does not change further. In other applications, where the dynamical behavior constitutes the output of the network, the changes of the activation values of the output units are significant (Figure 6).

Network Learning

The functionality of a neural network is determined by the combination of the topology (number of layers, number of units per layer, and the interconnection pattern between the layers) and the weight of the connections within the network. The topology is usually held fixed, and a certain training algorithm determines the weight. The process of adjusting the weights to make the network learn the relationship between the inputs and targets is called learning, or training. Many learning algorithms have been invented to help find an optimum set of weights that result in the solution of the problems. They can roughly be divided into two main groups:

1. Supervised Learning: The network is trained by providing it with inputs and desired outputs (target values). These input-output pairs are provided by an external teacher, or by the system containing the network. The difference between the real outputs and the desired outputs is used by the algorithm to adapt the weights in the network (Figure 7). It is often posed as a function approximation problem - given training data consisting of pairs of input patterns x, and corresponding target t, the goal is to find a function f(x) that matches the desired response for each training input.

2. Unsupervised Learning: With unsupervised learning, there is no feedback from the environment to indicate if the outputs of the network are correct. The network must discover features, regulations, correlations, or categories in the input data automatically. In fact, for most varieties of unsupervised learning, the targets are the same as inputs. In other words, unsupervised learning usually performs the same task as an auto-associative network, compressing the information from the inputs.

1. Purpose

To train a network and measure how well it performs, an objective function (or cost function) must be defined to provide an unambiguous numerical rating of system performance. Selection of an objective function is very important, because the function represents the design

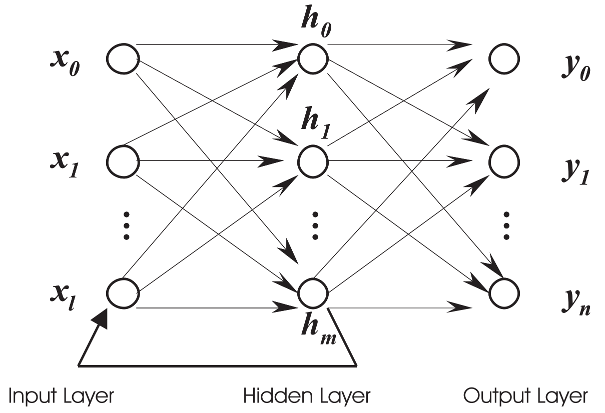


Figure 6. Recurrent Neural Network

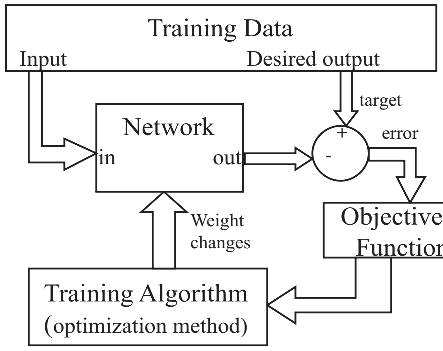


Figure 7. Supervised Learning Model

goals and decides what training algorithm can be taken. To develop an objective function that measures exactly what we want is not an easy task. A few basic functions are very commonly used. One of them is the sum of squares error function,

$$E = \frac{1}{NP}\sum_{p=1}^{P}\sum_{i=1}^{N}(t_{pi} - y_{pi})^2 \qquad (7)$$

where, p indexes the patterns in the training set, i indexes the output nodes, and t_{pi} and y_{pi} are the target and the actual network output for the ith output unit on the pth pattern respectively. In real world applications, it may be necessary to complicate the function with additional terms to control the complexity of the model.

2. Basic Architecture

A layered feed-forward network consists of a certain number of layers, and each layer contains a certain number of units. There is an input layer, an output layer, and one or more hidden layers between the input and the output layer. Each unit receives its inputs directly from the previous layer (except for input units) and sends its output directly to units in the next layer (except for output units). Unlike the Recurrent network, which contains feedback information, there are no connections from any of the units of the input of the previous layers nor to other units in the same layer, nor to the units more than one layer ahead. Every unit acts only as an input to the immediate next layer. Obviously, this class of networks is easier to analyze theoretically than other general topologies because their outputs can be represented by explicit functions of the inputs and the weights.

An example of a layered network with one hidden layer is

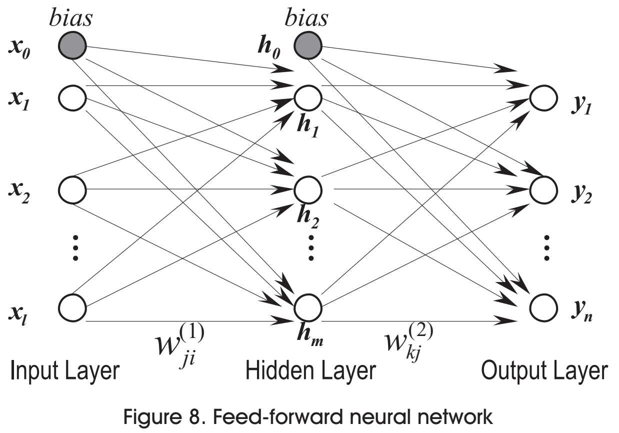


Figure 8. Feed-forward neural network

shown in Figure 8. In this network there are l inputs, m hidden units, and n output units. The output of the jth hidden unit is obtained by first, forming a weighted linear combination of the l input values, then adding a bias,

$$a_j = \sum_{i=1}^{l} w^{(1)}_{ji} x_i + w^{(1)}{}_{j0} \qquad (8)$$

where $w^{(1)}_{ji}$ is the weight from input i to hidden unit j in the first layer and $w^{(1)}_{ji}$ is the bias for hidden unit j. If the bias term are considered as being weights from an extra input $x_0 = 1$, (8) can be rewritten in the form of,

$$a_j = \sum_{i=0}^{l} w^{(1)}_{ji} x_i \qquad (9)$$

The activation of hidden unit j then can be obtained by transforming the linear sum using an activation function g(x):

$$h_j = g(a_j) \qquad (10)$$

The outputs of the network can be obtained by transforming the activation of the hidden units using a second layer of processing units. For each output unit k, first we get the linear combination of the output of the hidden units are obtained as,

$$a_k = \sum_{j=1}^{m} w^{(2)}_{kj} h_j + w^{(2)}_{k0} \qquad (11)$$

Again, the bias is observed and the above equation is rewritten into,

$$a_k = \sum_{j=0}^{m} w^{(2)}_{kj} h_j \qquad (12)$$

Then applying the activation function g2(x) to equations (12) we can get the kth output

$$yk = g2(ak) \qquad (13)$$

Combining equations (9), (10), (12) and (13) the complete representation of network is represented as,

$$y_k = g2\left(\sum_{j=0}^{m} w^{(2)}_{kj}\, g\left(\sum_{i=0}^{l} w^{(1)}_{ji} x_i\right)\right) \qquad (14)$$

The network shown in Figure 8 is a network with one hidden layer. It can extended to have two or more hidden layers easily as long as the above transformation is carried out further.

One thing to be noted is that, the input units are very special units. They are hypothetical units that produce outputs equal to their supposed inputs. These input units do no processing.

3. Back-Propagation

Back-Propagation is the most commonly used method for training multi-layer feed-forward networks. It can be applied to any feed-forward network with differentiable activation functions. This technique was popularized by Rumelhart, Hinton and Williams [6].

For most networks, the learning process is based on a suitable error function, which is then minimized with respect to the weights and bias. If a network has differential activation functions, then the activations of the output units become differentiable functions of input variables, the weights and bias. If the authors also define a differentiable error function of the network outputs such as the sum-of-square error function, then the error function itself is a differentiable function of the weights. Therefore, the derivative of the error with respect to weights can be evaluated, and these derivatives can then be used to find the weights that minimize the error function, by either using the popular gradient descent or other optimization methods. The algorithm for evaluating the derivative of the error function is known as back-propagation, because it propagates the errors backward through the network.

3.1 Error Function Derivative Calculation

The authors consider a general feed-forward network with arbitrary differentiable non-linear activation functions and a differential error function.

We know that, each unit j is obtained by first forming a weighted sum of its inputs of the form,

$$a_j = \sum_i w_{ji} z_i \qquad (15)$$

where z_i is the activation of a unit, or input. The authors then apply the activation function,

$$z_j = g(a_j) \qquad (16)$$

Note that one or more of the variables z_j in equation (15) could be an input, in which case, it will be denoted by x_i. Similarly, the unit j in equation (16) could be an output unit, which we will denote by y_k.

The error function will be written as a sum, the overall patterns in the training set of an error defined for each pattern separately is,

$$E = \sum_p E_p,\ E_p = E(Y;W) \qquad (17)$$

where, p indexes the patterns, Y is the vector of outputs, and W is the vector of all weights. E_p can be expressed as a differentiable function of the output variable y_k.

The goal is to find a way to evaluate the derivatives of the error functions E with respect to the weights and bias. Using equation (17), these derivatives are expressed as sums over the training set patterns of the derivatives for each pattern separately. Now, one pattern at a time is considered.

For each pattern, with all the inputs, the activations of all hidden and output units in the network is obtained by successive application of equations (15) and (16). This process is called forward propagation or forward pass. Once the activations of all the outputs, together with the target values, are available, the full expression of the error function E_p is achieved.

Now, consider the evaluation of the derivative of E_p with respect to some weight w_{ji}. Applying the chain rule can get the partial derivatives,

$$\frac{\partial E_p}{\partial w_{ji}} = \frac{\partial E_p}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}} = \delta_j \frac{\partial a_j}{\partial w_{ji}} = \delta_j z_i \qquad (18)$$

where,

$$\delta_j = \frac{\partial E_p}{\partial a_j} \qquad (19)$$

From equation (18), it is easy to see that, the derivative can be obtained by multiplying the value of δ for the unit at the output end of the weight by the value of z for the unit at the input end. Thus, the task becomes to find the δ_j for each hidden and output unit in the network.

For the output unit, δ_k is very straightforward,

$$\delta_k = \frac{\partial E_p}{\partial a_k} = \frac{\partial E_p}{\partial y_k} g'(a_k) \qquad (20)$$

For a hidden unit, δ_k is obtained indirectly. Hidden units can influence the error only through their effects on the unit k to which they send output connections. So,

$$\delta_j = \frac{\partial E_p}{\partial a_j} = \sum_k \frac{\partial E_p}{\partial a_k}\frac{\partial a_k}{\partial a_j} \qquad (21)$$

The first factor is just the δ_k of unit k. So,

$$\delta_j = \frac{\partial E_p}{\partial a_j} = \sum_k \delta_k \frac{\partial a_k}{\partial a_j} \qquad (22)$$

For the second factor, if unit j connects directly to unit k then $\partial a_k / \partial a_j = g'(a_j) w_{kj}$, otherwise it is zero. So the following Back-propagation formula is given,

$$\delta_j = g'(a_j) \sum_k w_{kj} \delta_k \qquad (23)$$

which means that, the values of δ for a particular hidden unit can be obtained by propagating the δ's backwards from units later in the network, as shown in Figure 9.

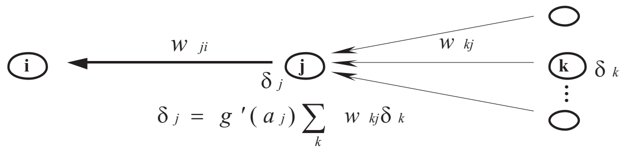


Figure 9. Backward Propagation

Recursively applying the equation gets the δ's for all of the hidden units in a feed-forward network, no matter how many layers it has.

3.2 Weight Adjustment with the Gradient Descent Method

Once, the derivatives of the error function with respect to weights are obtained, it can be used to update the weights so as to decrease the error. There are many varieties of gradient-based optimization algorithms based on these derivatives. One of the simplest of such algorithms is called a gradient descent or steepest descent. With this algorithm, the weights are updated in the direction in which the error E decreases most rapidly, i.e., along the negative gradient. The weight updating process begins with an initial guess for weights (which might be chosen randomly), and then generates a sequence of weights using the following formula,

$$\Delta w_{ji}^{(\tau+1)} = -\eta \frac{\partial E}{\partial w_{ji}} \qquad (24)$$

where, η is a small positive number called the learning rate, which is step size to be taken for the next step.

Gradient descent tells only the direction we will move to, but the step size or learning rate needs to be decided as well. Too low a learning rate makes the network learn very slowly, while too high a learning rate will lead to oscillation. One way to avoid oscillation for large η is to make the weight change dependent on the past weight change by adding a momentum term,

$$\Delta w_{ji}^{(\tau+1)} = -\eta \frac{\partial E}{\partial w_{ji}} + \alpha \Delta w_{ji}^{(\tau)} \qquad (25)$$

That is, the weight change is a combination of a step down the negative gradient, plus a fraction α of the previous weight change, where, $0 \le \alpha < 1$ and typically $0 \le \alpha < 0.9$ [6].

The role of the learning rate and the momentum term are shown in Figure 10 [3]. When no momentum term is used, it typically takes a long time before the minimum is reached with a low learning rate (a), whereas for large learning

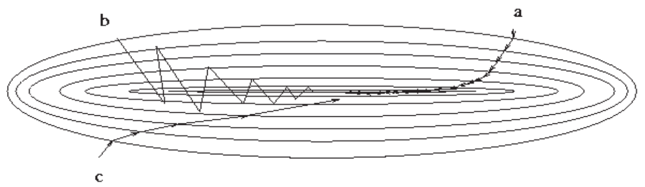


Figure 10. The Descent vs. Learning Rate and Momentum

rates the minimum may be never reached because of oscillation (b). When adding a momentum term, the minimum will be reached faster (c).

There are two basic weight-update variations: batch learning and incremental learning. With batch learning, the weights are updated over all the training data. It repeats the following loop: a) Process all the training data; b) Update the weights. Each such loop through the training set is called an epoch. While for incremental learning, the weights are updated for each sample separately. It repeats the following loop: a) Process one sample from the training data; b) Update the weights.

4. Data Tables

Table 1 provides the input data and Table 2 refers to the data processing table. Given the above input data, the model can be set up to reflect up to the following six effects:

- Temperature: represented by its actual value.
- Wind velocity: represented by its actual value.

Temperature	Wind	Hour	Weekday	Weekend	Month	Load
37	3	00	6	1	1	1168
37	9	01	6	1	1	1213
37	6	02	6	1	1	1316
37	3	03	6	1	1	1417
37	3	04	6	1	1	1534
37	5	05	6	1	1	1680
36	5	06	6	1	1	1819
34	6	07	6	1	1	1967

Table 1. Data Input Table

Date	Hour	Temperature	Wind	Load
02-08-1998	00	37	3	1168
02-08-1998	01	37	9	1213
02-08-1998	02	37	6	1316
02-08-1998	03	37	3	1417
02-08-1998	04	37	3	1534
02-08-1998	05	37	5	1680
02-08-1998	06	36	5	1819
02-08-1998	07	34	6	1967

Table 2. Data Processing Table

- Hour-of-day: represents the 24 hours of a day by 0, 1, 2… 23.
- Weekday: represents Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, and Saturday from 0, 1, 2, 3, 4, 5, and 6 respectively.
- Weekend: represents Monday, Tuesday, Wednesday, Thursday and Friday as 0; Saturday and Sunday as 1.
- Month-of-year: represents the twelve months in a year from 0 to 11 respectively.

The Dataset factory class also acts as a preliminary data filter to eliminate any outliners or bad data that were present. All inputs to the model linearly scale between 0 and 1, using the minimum and maximum values corresponding to the input vector.

Implementation of this project is a front end Java and Back end Oracle server and the authors used Swings to design the project.

4.1 Factors Affecting Load

The most difficult part of building a good model is to choose and collect the training and testing input data. A number of research papers [7][8][9][10][11][12][13] show that the following factors influence the demand of the load:

4.1.1 Weather Conditions

This includes temperature, wind velocity, cloud cover, dew point, rainfall, and snowfall. It has been widely observed that, in most cases, there is a strong correlation between weather (especially temperature and wind velocity) and load demand. In most situations, as temperature goes down, demand for gas goes up and vice-versa. However, this relation is highly non-linear. Other weather effects influence the load to a lesser extent.

4.1.2 Calendar

This includes an hour-of-day, day-of-week, and month-of-year, weekend and holiday effects. Most gas load patterns show a very consistent dependence on the calendar. For example, assuming all other factors remaining constant, the demand for energy at 1:00 AM when most people are sleeping is expected to be different from that at 6:00 AM when most people are getting up. Similar observations exist for the day-of-week. Though, it cannot be generalized, the middle days of the week (Tuesdays, Wednesdays, Thursdays and Fridays) behave differently from the remaining days. The month-of-year captures the seasonal effect. Holidays are again special days; they tend to produce behavior that is more like a weekend day.

4.1.3 Economic Information

This includes market gas price, the price differential between gas and oil, and the price differential between the competitors' price. In many situations, the effect of economic factors on gas demand is non-trivial. A direct influence of economic factors on gas demand can be observed in some instances, such as when the customer has storage fields for injection or withdrawal. If the market gas price is low, even if the temperature is high, there can be a high demand for gas if the customer is injecting gas into the storage field. Similarly, even if the temperature is low, if the gas price is high, customers may use the gas in the storage instead of buying new gas from pipeline companies, thus decrease the demand for load. The price differential between gas and oil plays an important role in the demand, when the customer is a dual fuel use power plant. Here, depending on the price differential between gas and oil, the customer can increase or reduce gas consumption.

The above effects are those, that can be quantified and hence are possible candidates to be used as inputs for neural net training and forecasting. There are other factors, such as contractual obligations that definitely influence gas demand, and these are too difficult to quantify and are therefore impossible to include as influencing variables. In addition, there are a number of other factors such as maintenance or accidents on competitors' lines, that influence the demand of gas load, that at best can only be explained qualitatively.

4.2 The Load Forecast Model

4.2.1 Input Data

The input data used in this model came from one of the author's clients, a pipeline company. They stored all of their historical data in different tables in an Oracle database – weather information in one table, and hourly load history in another table. The authors retrieved these data and stored them as a text.

4.2.2 Network Architecture

The network consists of one input layer, one output layer and one hidden layer. Obviously, there is only one output unit – the load. The number of input units is also fixed, depending on how many factors are included in the model, and how the factors are encoded. The number of hidden units are need to be decided by training with some test sets. Figure 11 is the architecture of the load forecast model including all of the six effects that is mentioned before.

The network requires enough hidden units to learn the general features of the relationship. With too many hidden units, it will cause over fitting while too few will lead to under fitting. The goal is to use as few units in the hidden layer as possible, while still retaining the network's ability to learn the relationships among the data. As mentioned earlier, including more than a single middle layer does not significantly improve the accuracy of the predictions.

The activation functions of the hidden units are sigmoid functions, while the output activation function can be either a sigmoid function or a linear function, which can be selected by the users.

4.3 Implementation of the Back-Propagation Algorithm

The network is trained using the back-propagation algorithm. The standard sum-of-squares error function is used.

$$E = \frac{1}{2}\sum_{k=1}^{n}(t_k - y_k)^2 \qquad (26)$$

Here is the Java code for the error function, which is one of

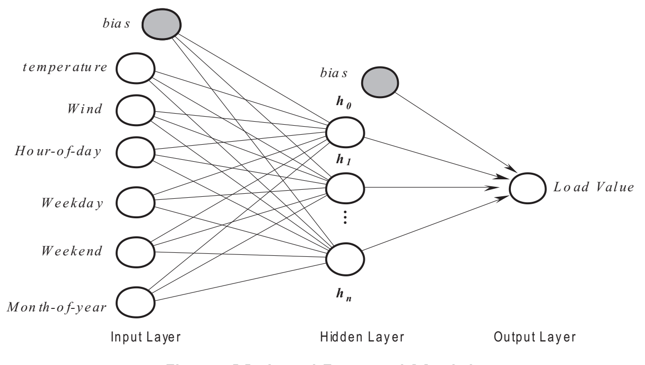


Figure 11. Load Forecast Model

the methods within the Neural Network class:

```
public double errorFunction (double[] x, double[] y) {
double sum = 0.0;
for (int i=0; i<x.length; i++)  sum += (x[i] - y[i])*(x[i] - y[i]);
return 0.5 * sum;
 }
```

As mentioned above, the activation function for the hidden units is the sigmoid function:

$$g(x) = \frac{1}{1+e^{-x}} \qquad (27)$$

This function has a very useful feature – its derivative can be expressed in the following form:

$$g'(x) = g(x)(1 - g(x)) \qquad (28)$$

The above two equations can be easily coded:

```
public double sigmoid(double x) {
                    if (x >  50.) return 1.0;
                    if (x < -50.) return 0.0;
                    return 1.0 /(1.0+Math.exp(-x));
          }
     public double sigmoidDerivative(double x) {
  return x*(1.0-x);
     }
```

The first step for the back-propagation is forward propagation

```
 void feedForward() {
     //For hidden units
     for (int i = 0; i<numberOfHiddenUnit; I++) {
          double sum = 0.0;
          for (int j=0; j<numberOfInputUnit+1; j++) {
             if (j==numberOfInputUnit)
          sum += weightLayer1[j][i]; // Include the Bias term
             else sum += weightLayer1[j][i]*inputs[j];
          }
          hiddens[i] = sigmoid(sum);
      }
     //For output units
      for (int i = 0; I < number of Output Unit; I++) {
          double sum =0.0;
```

```
        for (int j=0; j < number of Hidden Unit; j++) {
            sum += weightLayer2[j][i]*hiddens[j];
        }
        outputs[i] = sigmoid(sum);
    }
}
```

The second step is error Back-propagation. Using the expression derived from equations (20) and (23), the following results are obtained. For the output units, the δ's are given by,

$$\delta_k = y_k - t_k \tag{29}$$

while, for units in the hidden layer, the δ's are found using,

$$\delta_j = z_j(1-z_j)\sum_{k=1}^{c} w_{kj}\delta_k \tag{30}$$

Derivatives with respect to the first layer and second layer weights are then given by,

$$\frac{\partial E}{\partial w_{ji}} = \delta_j x_i \quad \text{and} \quad \frac{\partial E}{\partial w_{kj}} = \delta_k z_j \tag{31}$$

Gradient descent algorithm is used with momentum equation (25) to update the weights:

```
void back propagation (double rate, double alpha) {
        double[] delta1 = new double [number of HiddenUnit];
    double[] delta2 = new double[numberofOutputUnit];
    //Delta  for second layer
    for (int j=0; j<numberOfOutputUnit; j++) {
        delta2[j] = targets[j] - outputs[j];
    }
//Delta  for first layer
    for (int j=0; j<numberOfHiddenUnit; j++) {
        double sum = 0.0;
        for (int k=0; k<numberOfOutputUnit; k++) {
            double term = delta2[k] * weightLayer2[j][k];
                    if (outputActivationType==1) term *=sigmoidDerivative(outputs[k]);
            sum += term;
        }
        delta1[j] = sum;
    }
//Update the second layer weights
    for (int i=0; i<numberOfHiddenUnit; I++) {
        for (int j=0; j<numberOfOutputUnit; j++) {
            double delta = delta2[j]*hiddens[i];
                if (outputActivationType==1)   delta *= sigmoidDerivative(outputs[j]);
                double weightChange = rate * delta +alpha*momentum2[i][j];
            weightLayer2[i][j] += weightChange;
            momentum2[i][j] = weightChange;
        }
    }
//Update the first layer weights
    for (int i=0; i<numberOfInputUnit+1; I++) {
        for (int j=0; j<numberOfHiddenUnit; j++) {
            if (i!=numberOfInputUnit && inputs[i]==0) {
                momentum1[i][j] = 0.;
            }
            else {
  double delta = delta1[j]*sigmoidDerivative(hiddens[j]);
                if (i!=numberOfInputUnit) delta *= inputs[i];
                    double weightChange = rate * delta +alpha*momentum1[i][j];
                    weightLayer1[i][j] += weightChange;
                    momentum1[i][j] = weightChange;
            }
        }
    }
}
```

Batch learning method was adopted to train the networks.

4.4 Network Generalization

The Split-sample (or hold-out validation) method [5] is used to estimate generalization error. With this method, part of the data are reserved as a test set that will not be used in the training. After training, run the network on the test set. The error on the test set provides us an unbiased estimate of the generalization error, with which, the authors can decide whether the model is sufficiently general.

4.5 Features of the System

The Tell Future load forecast system has several useful features:

- It checks the importance of each effect.
- It helps find the optimal number of hidden units.
- It checks the influence of the learning rate and momentum.
- It displays the training and forecasting result in graphics and tabular form.
- It displays training errors.

5. Results

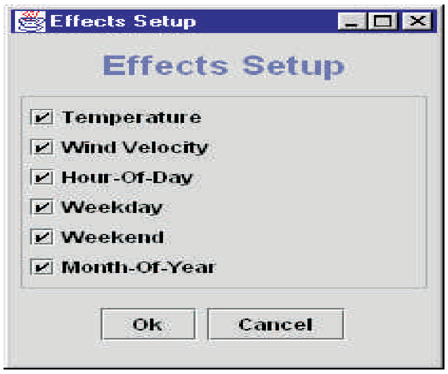
5.1 Home Page

Figure 12 describes the Home Page for the Load forecaster system.

5.2 Effects Setup

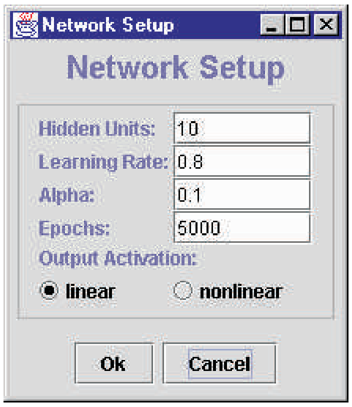
Figure 13 describes the effects setup with Temperature, Wind Velocity, Hour-of-day, Weekday, weekend, and Month of year. Here all the effects are selected.

Figure 12. Tell Future Forecast System Main Screen


Effects Setup

- [✓] Temperature
- [✓] Wind Velocity
- [✓] Hour-Of-Day
- [✓] Weekday
- [✓] Weekend
- [✓] Month-Of-Year

Figure 13. Effects Setup


Network Setup

Hidden Units:	10
Learning Rate:	0.8
Alpha:	0.1
Epochs:	5000

Output Activation: [✓ linear] nonlinear

Figure 14. Network Setup

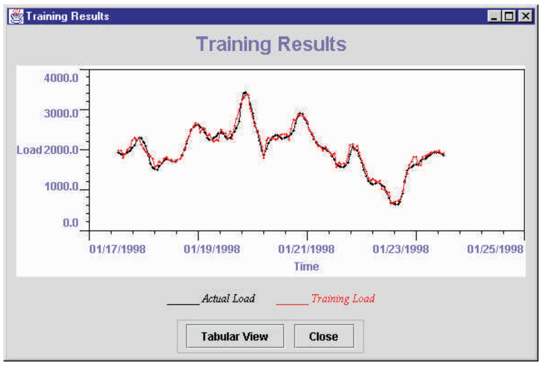


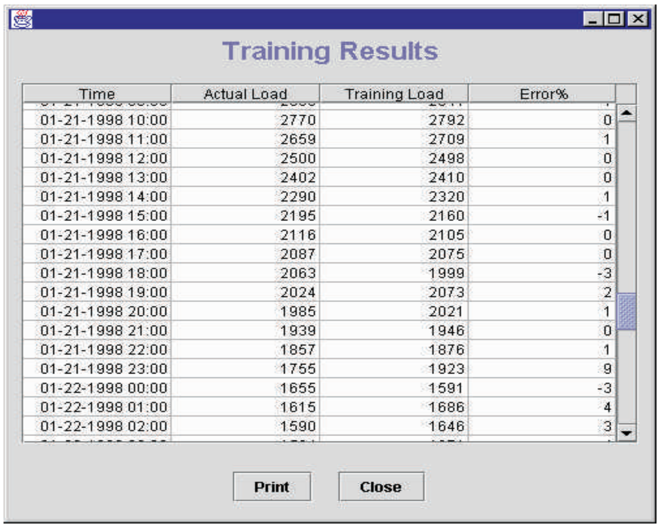
Figure 15. Training Results-Graphic Display

5.3 Network Setup

Figure 14 describes the Network setup, where the authors provide the input as Hidden Units, Learning rate, Alpha, Epochs and then they select linear and click ok.

5.4 Training Results

Figure 15 describes about the Training result on a graphic


Training Results

Time	Actual Load	Training Load	Error%
01-21-1998 10:00	2770	2792	0
01-21-1998 11:00	2659	2709	1
01-21-1998 12:00	2500	2498	0
01-21-1998 13:00	2402	2410	0
01-21-1998 14:00	2290	2320	1
01-21-1998 15:00	2195	2160	-1
01-21-1998 16:00	2116	2105	0
01-21-1998 17:00	2087	2075	0
01-21-1998 18:00	2063	1999	-3
01-21-1998 19:00	2024	2073	2
01-21-1998 20:00	1985	2021	1
01-21-1998 21:00	1939	1946	0
01-21-1998 22:00	1857	1876	1
01-21-1998 23:00	1755	1923	9
01-22-1998 00:00	1655	1591	-3
01-22-1998 01:00	1615	1686	4
01-22-1998 02:00	1590	1646	3

Figure 16. Training Results in Tabular Display

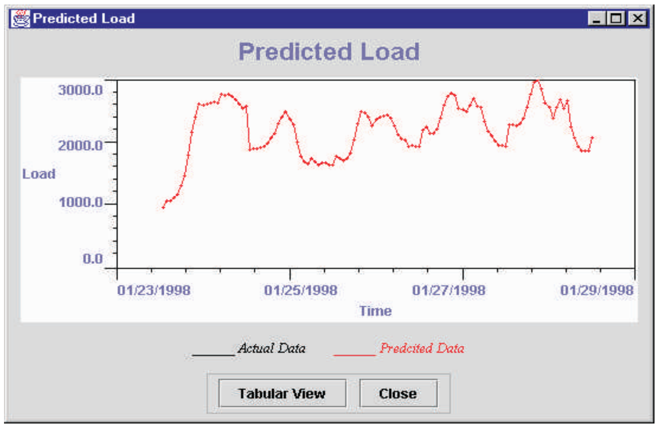


Figure 17. Load Forecast-Graphic Display

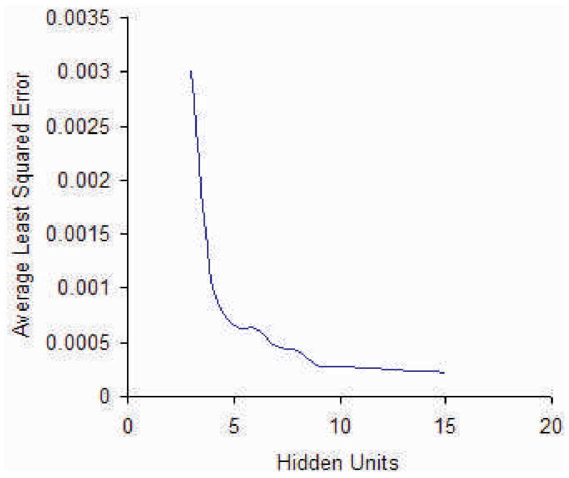


Figure 18. Average Squared Error vs Hidden Units

display for the given input. Here the Tabular View button is clicked.

5.5 Training Results in a Tabular Display

Figure 16 displays the Training results in a tabular form for the given input.

5.6 Load Forecast Graphic Display

Figure 17 displays the Load forecast graphic for the given input by the user. Here the Tabular View button is clicked in order to display the results in the form of table.

5.7 Average Squared Error vs Hidden Units

Figure 18 displays the graph for Average squared error vs. hidden units.

5.8 Training Epochs vs Learning Rate Momentum

Table 3 shows the epochs that the training processes taken to meet the error tolerants (average square error is 0.0005) or reach the epoch limit (9999) with different

Learning Rate	Test	Momentum				
		0	0.1	0.2	0.4	0.6
	1	1493	1867	1933	3030	4806
	2	2014	1233	1207	2683	4728
	3	961	1649	3260	3898	5547
	4	2099	1642	2127	4230	2841
	5	1538	3158	2061	1967	9999
1.2	6	2789	1834	3498	1804	9999
	7	1135	1650	1215	5708	2091
	8	2271	4508	1668	8503	9999
	9	1257	1936	2127	3454	3238
	10	4696	1253	1267	1625	9999
	Average	2025	1728	2036	3690	6325
	1	2914	2813	4211	3421	9999
	2	3469	1106	2496	1701	4419
	3	1108	1770	2472	1898	9999
	4	2179	3019	1448	2568	9999
	5	1957	1468	2045	1567	9999
1.0	6	1354	2086	961	2101	5333
	7	887	1796	2393	3286	9999
	8	2315	1442	1027	1268	1496
	9	1058	2143	1462	3698	3109
	10	1838	1626	1248	1324	3504
	Average	1908	1927	1976	2283	6786
	1	1414	1689	2027	2022	1888
	2	4411	916	2374	6034	3147
	3	1600	1010	1960	1271	3100
	4	1532	1093	3634	1952	2992
	5	1237	1519	1603	1775	1782
0.8	6	1011	1037	1448	3415	3256
	7	2622	1829	7767	1096	1002
	8	1509	1073	1293	1058	7131
	9	1565	2288	879	3006	2614
	10	1425	1141	2590	3372	2193
	Average	1833	1360	2558	2500	2911
	1	1493	1129	1000	630	3945
	2	5471	2582	845	1981	2091
	3	5461	1031	3003	1226	767
	4	1972	2734	2952	640	913
	5	2364	1796	1119	2405	1358
0.6	6	1943	870	7932	1377	1099
	7	2088	5425	999	2209	953
	8	1753	2490	3348	3547	1517
	9	1887	2077	5242	1040	2705
	10	2728	2753	2968	2395	2205
	Average	2716	2289	2941	1745	1755
	1	3861	2272	5690	1467	962
	2	5253	6670	1925	5917	1671
	3	2622	1506	1324	1302	765
	4	1762	5158	1489	2203	3333
	5	1554	3009	1138	1620	1370
0.4	6	1722	1727	3962	1084	1340
	7	2507	6240	1948	1253	1351
	8	2222	4112	2035	814	3877
	9	1966	2170	1761	2560	2094
	10	2367	955	1667	1794	1207
	Average	2584	3382	2294	2001	1797
	1	9999	2684	3611	4148	2639
	2	4065	4114	4564	2219	1441
	3	4187	3325	2874	3783	2594
	4	4821	4627	8726	5053	2388
	5	2713	2401	2090	1718	2297
0.2	6	7866	3322	4891	7496	2481
	7	3908	8884	9999	6844	2681
	8	1619	1767	3095	3529	2623
	9	4740	6457	2221	2218	2282
	10	9999	3613	1483	1907	1563
	Average	5392	4119	4355	3892	2299

Table 3. Training Epochs vs Learning Rate Momentum

values of learning rate and momentum, where each pair had 10 tests. It is easy to see that too large and too small learning rates converge slowly, while high momentum helps small learning rate to converge faster. The best learning rate and momentum term are 0.8 and 0.1 respectively for this model. There are no big differences between using a sigmoid activation function and a linear activation function for the output unit.

Conclusion

Neural networks can learn to approximate any function and behave like associative memories by using just an example data that is representative of the desired task. They are model free estimates and are capable of solving complex problems based on the presentation of a large number of training data. This gives them a key advantage over traditional approaches to function the estimation such as the statistical methods. Neural networks estimate a function without a mathematical description of how the outputs functionally depend on the inputs - they represent a good approach that is potentially robust and fault tolerant.

In this paper, the authors have examined the properties of the feed-forward neural networks and the process of determining the appropriate network inputs and architecture, and built up a short-term gas load forecast system - the Tell Future system. This system performs very well for short-term gas load forecasting. The forecast accuracy has been in excess of 90%.

In order to forecast the future load from the trained networks, the authors need to use the history loads, temperature, wind velocity, and calendar information in addition to the predicted future temperature and wind velocity. Compared to other regression methods, the neural networks allow more flexible relationships between temperature, wind, calendar information and load pattern. It has also been shown by other researchers that multi-layer feed-forward neural network performs best for short-term load forecasting [7],[14].

The authors have utilized only temperature, wind and calendar information, since they are the only information available to us. Use of additional variables such as cloud coverage and economic information should yield better results [7]. Since, the neural networks simply interpolate between the training data, it will give high errors with the test data that is not close enough to any training data.

Feed-forward neural networks can be used in many kinds of forecasting in different industrial areas. Similar models can be built to make electric load forecasting, daily water consumption forecasting, stock and markets forecasting, traffic flow and product sales forecasting [15],[16] as long as correct relationships between the inputs and the outputs can be captured and put in to the models. But there is no universal network paradigm suitable for all kinds of forecasting problems. For each problem, a detailed analysis of the domain data and the acquisition of prior knowledge are necessary to find a suitable model. The introduction of prior knowledge in input selection, input encoding, or architecture determination is often very useful, especially when the available domain data are limited.

The standard Back-propagation algorithm for training feed-forward neural networks have proven robust even for difficult problems. However, its high performance results are attained at the expense of a long training time to adjust the network parameters, which can be discouraging in many real-world applications. Even on relatively simple problems, it often requires a lengthy training process in which, the complete set of training examples is processed hundreds or thousands of time. Thus, some accelerating techniques or advanced training algorithms can be applied to improve the performance of the networks.

References

[1]. **L. Fausett, (1994).** *Fundamentals of Neural Networks: Architectures, Algorithms, and Applications*. Prentice-Hall, Inc.

[2]. **W.S. Sarles, (1997).** "Neural Network FAQ", Retrived from: ftp://ftp.sas.com/pub/neural/FAQ.html

[3]. **R.D. Reed and Robert J. Mark, (1999).** *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*. The MIT Press.

[4]. **C.M. Bishop, (1995).** *Neural Networks for Pattern*

Recognition. Oxford University Press.

[5]. B.D. Ripley, (1996). *Pattern Recognition and Neural Networks*. Cambridge University Press.

[6]. Rumelhart, D.E., Hinton, G.E., and Williams, R.J., (1986). "Learning Internal Representations by Error Propagation". *ACM Digital Library*, Vol. 323, pp.533-536.

[7]. W. P. Wagner, (1995). "Daily Peak Load Electricity Forecasting using Artificial Neural Networks". Retrived from: http://hsb.baylor.edu/ramsower/acis/papers/ wagnerw. htm.

[8]. A. Khotanzad. M. H. Davis, A. Abaye, and D. J. Maratukulam, (1996). "An Artificial Neural Network Hourly Temperature Forecaster with Application in Load Forecasting". *IEEE Transaction on Power Systems*, Vol.11, pp.870-876.

[9]. S. T. Chen, D. C. Yu, and A. R. Moghaddamjo, (1992). "Weather Sensitive Short-Term Load Forecasting using Nonfully Connected Artificial Neural Network". *IEEE Transaction on Power Systems*, Vol.7, pp.1098-1104.

[10]. A.G. BakIrtzIs, V. PetrIdIs, and S. J. KlartzIs, (1995). "A Neural Network Short Term Load Forecasting Model for the Greek Power System". *IEEE Transaction on Power Systems*, Vol. 11, pp.858-862.

[11]. Peng, T.M., Hubele, N.F., and Karady, (1993). "An Adaptive Neural Network Approach to One-Week Ahead Load Forecasting". *IEEE Transactions on Power Systems*, Vol.8, pp.1195-2003.

[12]. J. Angstenberger, (1996). *Neural Networks and their Applications*, John Wiley & Sons.

[13]. X. Ding, and S. Canu, (1996). "Neural Network Based Model for Forecasting". Retrived from: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10. 1.1.49.5576&rep=rep1&type=pdf

[14]. H. C. A. M. Withagen, (1997). *Neural Networks: Analog VLSI Implementation and Learning Algorithms*. Technische Universiteit, Eindhoven.

[15]. T. Masters, (1995). *Advanced Algorithms for Neural Networks: a C++ Sourcebook*. John Wiley & Sons, Inc.

[16]. T. Masters, (1995). *Neural, Novel & Hybrid Algorithms for Time Series Prediction*. John Wiley & Sons, Inc.

[17]. Amrender Kumar, (2014). "Artificial Neural Network": Retrived from: http://www.iasri.res.in/ebook/fet/Chap%2014_Artificia%20Neural%20Networks_Amrender.pdf

[18]. Gurvinder Singh, (2009). *Quantum Neural Netework Application for Weather Forecasting*. Thesis.

ABOUT THE AUTHORS

K. Sunil Manohar Reddy is currently working as an Assistant Professor in the Department of Computer Science and Engineering at Matrusri Engineering College, Hyderabad, India. His areas of interest are Neural Networks, Artificial Intellegence, Software Engineering, Data Warehousing and Data Mining. He has presented and published various papers in National and International Conferences and Journals.

Dr. G. Ravindra Babu is currently working as a Professor in the Department of Computer Science and Engineering at Trinity College of Engineering & Technology, Karimnagar, India. His areas of interest are Neural Networks, Artificial Intellegence, and Computer Networks. He has presented and published around 45 papers in National and International Conferences and Journals.

Dr. S. Krishna Mohan Rao is currently working as a Professor in the Department of Computer Science and Engineering at Siddhartha Institute of Engineering & Technology, Hyderabad, India. His areas of interest are Artificial Intellegence, Neural Networks, and Databases. He has presented and published around 39 papers in National and International Conferences and Journals.