

Techniques And Methods To Implement Neural Networks Using SAS and .NET

Shenglin Zheng
Johnny Dupree
Uttam Shah
Marc Torres

ABSTRACT

Neural networks are a powerful method for solving complex, "real world", modeling problems when traditional algorithms cannot be formulated. Most neural networks are implemented in programming languages that can represent data structures well such as C/C++, Java or Visual Basic. SAS is very good at processing longitudinal data. This paper will offer techniques and methods through implementation of neural nets using only BASE SAS and SAS macro's to represent intrinsic iterative learning/training steps of neural network algorithms. This SAS program can be used as building blocks to build one's own neural network models or be used as a starting point to study neural networks for SAS programmers. A numeric example for predicting stock prices is presented in this paper using a .NET application framework.

Introduction

There are different kinds of neural network models. For example, Adaline Madaline, Backpropagation, Bidirectional Associative Memory, Temporal Associative Memory, Brain-State-in-a-box, Counterpropagation, Neocognitron, Adaptive Resonance Theory(ART) etc.. Most research on implementations about neural network models will show that many implementations are done in C++, Java or Visual Basic. These languages have very good abilities to express data structures such as trees, linked lists, queues and stacks.

In this paper will focus on a three-layer **Feedforward Backpropagation** neural net . By being able to design and implement a simple neural net model in Base SAS we can know how to build more complicated neural network models. In addition we will demonstrate this application using a .NET "wrapper" interface that does not use standard SAS external application components such as SAS/Intrnet or Integration Technologies.

Layout

The basic architecture of a **Feedforward Backpropagation** net is shown in **Figure 1**. While there can be many hidden layers, we will illustrate this network with only one hidden layer. Also the number of neurons in the input layer and that in the output layer are determined by the dimensions of the input and output patterns, respectively. It is not easy to determine how many neurons are needed in hidden layer. Here we will show the layout with twenty neurons in the input layer, four neurons in hidden layer and twenty neurons in the output layer, with a few representative connections. Since our demonstration is based on stock price prediction, these twenty neurons are representative of 20 days of historical stock price quotes.

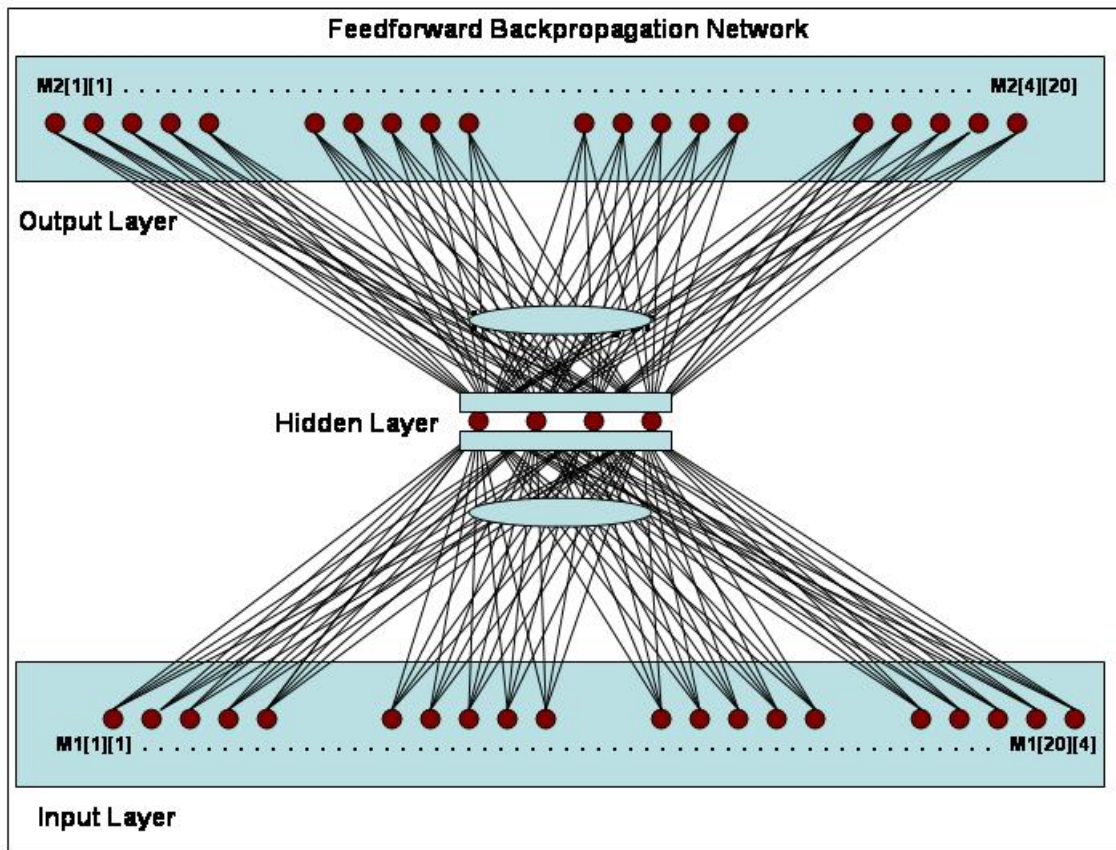


Figure 1

The network has three fields of neurons: one for input neurons, one for hidden processing elements, and one for the output neurons. Connections are for feed forward activity. There are connections from every neuron in field A to everyone in field B, and in turn, from every neuron in field B to every neuron in field C. Thus there are two sets of weights. Those figuring in the activations of hidden layer neurons, and those that help determine the output neuron activations. In training, all of these weights are adjusted by considering what can be called a **cost function** in terms of error in the computed output pattern and the desired output pattern.

Training and Prediction

The first step of building **Feedforward Backpropagation** net is to train the model. This is accomplished by executing a series of SAS Macros. The model needs a finite number of pattern pairs consisting of an input pattern and a desired or target output pattern. The first part of the process does a normal text file read of the downloaded stock data file. From this data an input pattern is assigned from the input file and initialized to the input layer.

The neurons here pass the pattern activations to the next layer of neurons, which are in a hidden layer.

```
%macro hidden ;
data nrlnet.init_file(drop=sigma i j) ;
set nrlnet.init_file ;
retain sigma ;
sigma = 0 ;
retain y1-y4 sigma theta1-theta4
ml_1w1-ml_1w4 ml_2w1-ml_2w4 ml_3w1-ml_3w4 ml_4w1-ml_4w4
ml_5w1-ml_5w4 ml_6w1-ml_6w4 ml_7w1-ml_7w4 ml_8w1-ml_8w4
ml_9w1-ml_9w4 ml_10w1-ml_10w4 ml_11w1-ml_11w4
ml_12w1-ml_12w4 ml_13w1-ml_13w4 ml_14w1-ml_14w4
ml_15w1-ml_15w4 ml_16w1-ml_16w4 ml_17w1-ml_17w4
ml_18w1-ml_18w4 ml_19w1-ml_19w4 ml_20w1-ml_20w4;
array ml_w (20,4)
ml_1w1-ml_1w4 ml_2w1-ml_2w4 ml_3w1-ml_3w4 ml_4w1-ml_4w4
ml_5w1-ml_5w4 ml_6w1-ml_6w4 ml_7w1-ml_7w4 ml_8w1-ml_8w4
ml_9w1-ml_9w4 ml_10w1-ml_10w4 ml_11w1-ml_11w4
ml_12w1-ml_12w4 ml_13w1-ml_13w4 ml_14w1-ml_14w4
ml_15w1-ml_15w4 ml_16w1-ml_16w4 ml_17w1-ml_17w4
ml_18w1-ml_18w4 ml_19w1-ml_19w4 ml_20w1-ml_20w4;
array y (4) y1-y4 ;
array theta (4) theta1-theta4 ;
array indata_x (20) indata_x1-indata_x20 ;
do i = 1 to 4 ;
theta(i) = .2 ;
do j = 1 to 20 ;
sigma = sigma + (indata_x(j) * ml_w(j,i));
* put i=j= indata_x(j)= ml_w(j,i)= sigma= ;
end ;
y(i) = 1/(1+exp(-sigma)) + theta(i) ;
sigma = 0 ;
end ;
%mend hidden ;
```

Figure 2

The outputs of the hidden layer neurons are computed by using a bias, and also a threshold function with the activations determined by the weights and the inputs. These hidden layer outputs become inputs to the output neurons, which process the inputs using an optional bias and a threshold function. The final output of the network (computed pattern) is computed by the activations from the output layer.

```
%macro outputl ;
data nrlnet.init_file(drop=sigma i j) ;
set nrlnet.init_file ;
sigma = 0 ;
retain z1-z4 sigma tau1-tau20 sigma
m2_1w1-m2_1w20 m2_2w1-m2_2w20 m2_3w1-m2_3w20
m2_4w1-m2_4w20;
array m2_w (4,20)
m2_1w1-m2_1w20 m2_2w1-m2_2w20 m2_3w1-m2_3w20
m2_4w1-m2_4w20;
array y (4) y1-y4 ;
array z (20) z1-z20 ;
array tau (20) tau1-tau20 ;
do i = 1 to 20 ;
tau(i) = .3 ;
do j = 1 to 4 ;
put sigma= ;
sigma = sigma + (y(j) * m2_w(j,i));
*put sigma= ;
end ;
z(i) = 1/(1+exp(-sigma)) + tau(i);
sigma = 0 ;
end ;
%mend outputl ;
```

Figure 3

The computed pattern and the input pattern are compared, a function of the error for each component of the pattern is determined, and adjustment to weights of connections between the hidden layer and the output layer is computed. A similar computation, still based on the error in the output, is made for the connection weights between the input and the hidden layers.

The procedure is repeated with each pattern pair assigned for training the network. Each pass through all the training patterns is called a cycle or an epoch. The process is then repeated as many cycles as needed until the error is within a prescribed tolerance.

```
%macro errors ;
data nrlnet.init_file(drop=i) ;
set nrlnet.init_file ;
retain e1-e20 ;
array e (20) e1-e20 ;
array z (20) z1-z20 ;
array desired_p (20) desired_p1-desired_p20 ;
do i = 1 to 20 ;
    e(i) = desired_p(i) - z(i) ;
end ;
run ;
%mend errors ;
```

Figure 4

There can be more than one learning rate parameter used in training in a **Feedforward Backpropagation** net. You can use one with each set of weights between consecutive layers.

After the training process is done we will obtain adjusted weights and threshold/bias and captured the pattern in the training sets. Then we can apply this model to new set of input pattern and use the computed pattern to predict the desired pattern.

Notation and Equations

In order to understand our algorithm clearly now we will give mathematical description for this **Feedforward Backpropagation** net. Here there are two matrices M_1 and M_2 whose elements are the weights on connections.

M_1 refers to the interface between the input and hidden layers, and M_2 refers to that between the hidden layer and output layer. Since connections exist from each neuron in one layer to every neuron in the next layer, there is a weight associated with this connection. For example, $M_1[i][j]$ represent is the weight on the connection from the i th input neuron to the j th neuron in the hidden layer. Similarly, $M_2[i][j]$ denotes the weight on the connection from the i th neuron in the hidden layer and the j th output neuron.

In addition we will use x, y, z for the outputs of neurons in the input layer, hidden layer, and output layer, respectively, with a subscript attached to denote which neuron in a given layer we are referring to. Let P denote the desired output pattern, with P_i as the components. Let m be the number of input neurons, so that according to our notation, $(x_1, x_2, \dots, x_m)$ will denote the input pattern. If P has, say, r components, the output layer needs r neurons.

Let the number of hidden layer neurons be n . Let β_h be the learning rate parameter for the hidden layer, and β_o , that for the output layer. Let θ with the appropriate subscript represent the threshold value or bias for a hidden layer neuron, and τ with an appropriate subscript refer to the threshold value of an output neuron.

Let the errors in output at the output layer be denoted by e_j s and those at the hidden layer by t_i 's. If we use a Δ prefix of any parameter, then we are looking at the change in or adjustment to that parameter. Also, the thresholding function we would use is the sigmoid function, $f(x) = 1 / (1 + \exp(-x))$.

Equations

Output of j th hidden layer neuron:

$$y_j = f(\sum_i x_i M_1[i][j]) + \theta_j)$$

Output of j th hidden layer neuron:

$$z_j = f(\sum_i y_i M_2[i][j]) + \tau_j)$$

i th component of vector of output differences:

$$\text{desired value} - \text{computed value} = P_i - z_i$$

i th component of output error at the output layer:

$$e_i = (P_i - z_i)$$

i th component of output error at the hidden layer:

$$t_i = y_i (1 - y_i) (\sum_j M_2[i][j] e_j)$$

Adjustment for weight between i th neuron in hidden layer and j th output neuron:

$$\Delta M_2[i][j] = \beta_o y_i e_j$$

Adjustment for weight between i th input neuron and j th neuron in hidden layer:

$$\Delta M_1[i][j] = \beta_h x_i t_j$$

Adjustment to the threshold value or bias for the j th output neuron:

$$\Delta \tau_j = \beta_o e_j$$

Adjustment to the threshold value or bias for the j th hidden layer neuron:

$$\Delta \theta_j = \beta_h e_j$$

Model Implementation & Prediction

For the convenience of programming, here we specified the number of nodes for input layer and output layer at 20 and the number of node for hidden layer at 4 since one month contains 20 work days and 4 weeks a month; the normal cycle for stock market tracking. The number in essence has no limitation in theory. Based on any new input we want to predict the next day price. **(Note: Some of the SAS code has been abbreviated due to the length of the macro code).**

```
data nrlnet.predict ;
  set nrlnet.init_file ;
  sigma = 0 ;
  sigma2 = 0 ;
  retain y1-y4 z1-z4 sigma sigma2
  m1_1w1-m1_1w4 ..... m1_20w1-m1_20w4;
  array m2_w (4,20) m2_1w1-m2_1w20 ...
          m2_4w1m2_4w20;
  array y (4) y1-y4 ;
  array theta (4) theta1-theta4 ;
  array indata_x (20) indata_x1-indata_x20 ;
  array z (20) z1-z20 ;
  array tau (20) tau1-tau20 ;
  do i = 1 to 4 ;
    theta(i) = .2 ;
    do j = 1 to 20 ;
      sigma2 = sigma2 + (indata_x(j) * m1_w(j,i));
    end ;
    y(i) = sigma2 + theta(i) ;
    sigma = 0 ;
  end ;
  do i = 1 to 20 ;
    do j = 1 to 4 ;
      sigma = sigma + (y(j) * m2_w(j,i));
    end ;
    z(i) = sigma + tau(i);
    sigma = 0 ;
  end ;
run ;

%mend neuralNet ;
```

Figure 5

The predicted value is calculated or **“PREDICTED”** in: **$Z(i) = \text{sigma} + \text{tau}(i)$** . This data is processed and stored in the SAS dataset for next iterations or next day values. The results are then made available to the user.

.NET Application

A relatively simple user interface is constructed to interact with the neural net using **.NET** (see Figure X). In an ideal environment SAS/Intrnet or SAS/Integration Technologies would be used to establish external connectivity to SAS datasets and software. Our model demonstrates another way.

This interface contains both a **LEARN** and **PREDICT** control. Although it is assumed that SAS software is available for implementation of the neural net, this model was built using the SAS Learning Edition software and the SAS Universal ODBC. The source data was downloaded from the internet and pre-loaded to a SAS dataset with sample stock data.

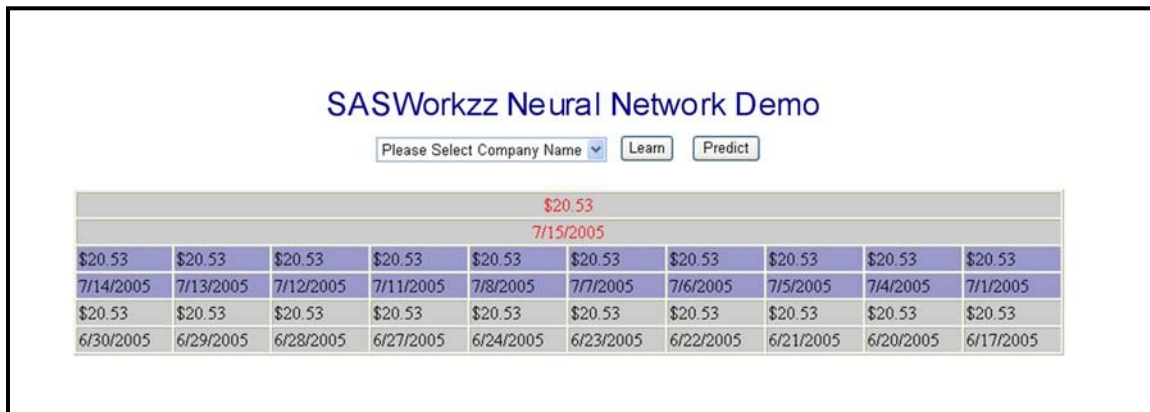


Figure 6

The SAS dataset is read and populates the interface as selected using the following code:

```
OleDbConnection cn = new OleDbConnection();
cn.ConnectionString = "Provider=sas.IOMProvider;
Data Source=SERVERNAME";

OleDbCommand cmd = cn.CreateCommand();
cmd.CommandType = CommandType.TableDirect;
cmd.CommandText = itable;

// Execute the command and get an OleDbDataReader
object.
OleDbDataReader reader = cmd.ExecuteReader();
```

Upon selecting the **LEARN** function from the interface the “learning” aspect of the SAS neural net program is executed using a batch call to a **.BAT** file which contains a Windows operating system call to SAS similar to the following:

```
start " " /min "C:\Program Files\SAS Institute\SAS\V8\sas.exe"
c:\SASNN\learnstock.sas -nosplash -icon
```

It's important to note that by using the .BAT extension the .NET will suspend executing until the program is completed. The necessary data components from the **LEARN** function are stored in the SAS dataset for retrieval. The key .NET code for this call is as follows:

```
private void btnLearn_Click(object sender, System.EventArgs e){
    System.Diagnostics.ProcessStartInfo psi =
        new
    System.Diagnostics.ProcessStartInfo(@"C:\SASNN\learnstock.bat");
    psi.RedirectStandardOutput = true;
    psi.WindowStyle = System.Diagnostics.ProcessWindowStyle.Hidden;
    psi.UseShellExecute = false;
    System.Diagnostics.Process listFiles;
    listFiles = System.Diagnostics.Process.Start(psi);
    System.IO.StreamReader myOutput = listFiles.StandardOutput;
    listFiles.WaitForExit(2000);
    if (listFiles.HasExited)
    {
        string output = myOutput.ReadToEnd();
        this.processResults.Text = output;
    }
}
```

Upon selecting the **PREDICT** control the predicted value is retrieved and displayed back to the interface. The basic calls and method of the **PREDICT** control are similar as the **LEARN** functionality.

Conclusion

The principle goal of this process, which is to develop a neural network using Base SAS and macros is a viable approach. Using the fundamentals of the neural net code demonstrated here, we can use this SAS coding pattern to program more complicated multi-layer neural networks. Although other languages may offer their own advantages, for the SAS professional creating functional neural nets within the standard SAS language allows an approach which can take further advantage of the full and familiar SAS software/environment. Secondly, this process demonstrates how a .NET user interface can be developed and utilized outside the SAS external connectivity software such as SAS/Integration Technologies or SAS/Intrnet.

Contact Information

Full code and application components will be made available on request. Your comments and questions are valued and encouraged.

Shenglin Zheng

E-mail: shenglinzheng@cox.net

Johnny Dupree

E-mail: dupreeaz@cox.net

Uttam Shah

E-mail: ushahx@yahoo.com

Marc Torres

E-mail: marc.torres@usa.net

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.