SESUG Paper 283-2018

Building Neural Network Model in Base SAS® (From Scratch)

Soujanya Mandalapu, Yan Wang, Xuelei Sherry Ni, Kennesaw State University.

ABSTRACT

Artificial Neural Networks (ANNs) are extremely popular in deep learning applications such as image recognition and natural language processing. ANNs are also being implemented in finance, marketing, and insurance domains. Most neural network models are implemented in Python, Java, C++, or Scala. Although Base SAS is a preferred language in regulated environments such as finance and clinical trials, it cannot be used to implement ANN models. This brings difficulties for financial modelers who want to use ANNs to improve their models to gain efficiency. This paper aims at those modelers who would like to implement machine learning models using only Base SAS and SAS macros. A standard three-layer (one input, one hidden and one output) feed forward backward propagation algorithm with three separate macros (forward propagation, backward propagation and to control the number of iterations) for each repetitive step was implemented in this paper. This algorithm is scalable to increase as many features and hidden nodes.

INTRODUCTION

Artificial Neural Networks (ANNs) are statistical learning models which are modeled based on the information processing procedure found in the brain (Rashid, 2016). Over the years, Neural Networks have evolved from modeling simple problems to a wide variety of complex problems and this rapid phenomenon has been fueled by the availability of computation ability and novel algorithms. Neural networks, just like the brain can solve complex problems such as image recognition, speech processing, and natural language processing (Gurney, 1997). The artificial equivalents of biological neurons and synapses are the nodes and weights respectively (Gurney, 1997). Several different types of Neural Networks exist based on their application. In this paper, we are concerned with a simple three-layer feed forward backward propagation network as it is a popular multilayer perceptron used to model nonlinear data for prediction and classification tasks (Samarasinghe, 2006) (Larose, 2004).

This paper not only helps the modelers to implement neural networks in Base SAS and SAS macros but is also extremely helpful to new machine learning enthusiasts who are interested to learn the step by step implementation of Neural Network algorithm in Base SAS. The paper is structured as follows. In Section 2, the Feed Forward Backward Propagation Neural Network architecture was explained. In section 3, the algorithm was explained. In section 4, the step by step implementation of the algorithm was discussed. In section 5, the performance of our algorithm is compared with the inbuilt Python function. Section 6 concludes our work.

2. FEED FORWARD AND BACKWARD PROPAGATION NEURAL NETWORK ARCHITECTURE

We review the Feed Forward Backward Propagation Neural Network architecture in this section. A full introduction can be found in (Samarasinghe, 2006) & (Larose, 2004). In a typical feed forward network, the information flows through a single direction and does not allow looping or cycling and the network is completely connected. A simple multi-layer ANN architecture is given in Figure 1. The multi-layer perceptron has three layers namely: an input layer, a hidden layer, and an output layer of neurons. They are denoted by I, H, and O, respectively (Figure 1). These three layers are fully connected, and their strength of the connection is termed as ‘weights’. For a simple three-layer perceptron, two different sets of weights connect

the layers, the *input-hidden weights* and the *hidden-output weights*. These weights are free-flowing parameters and they provide enormous flexibility to model the data.

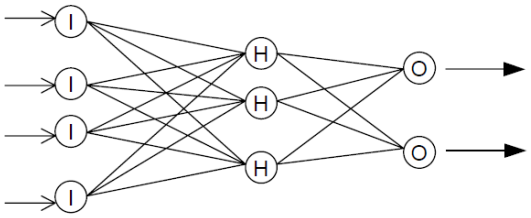

Figure 1. Standard Feedforward Neural Network

Data from the input layer is transmitted to the hidden neuron through the *input-hidden weights*. Each hidden neuron receives the weighted inputs from all the input neurons. The neurons in the hidden layer accumulate and process the weighted inputs using an activation function before sending their outputs to the output neurons via the *hidden-output weights* (Figure 2) (Samarasinghe, 2006). There are several activation functions such as sigmoid, ReLU and TanH. In this paper, we used the sigmoid activation function. The hidden-neuron output is weighted by the corresponding weights and processed to produce the final output. This network is trained to learn by repeated exposure to input-output data until the network produces the desired output. Learning is a process in which the weights are changed incrementally until the network learns to produce the desired output.

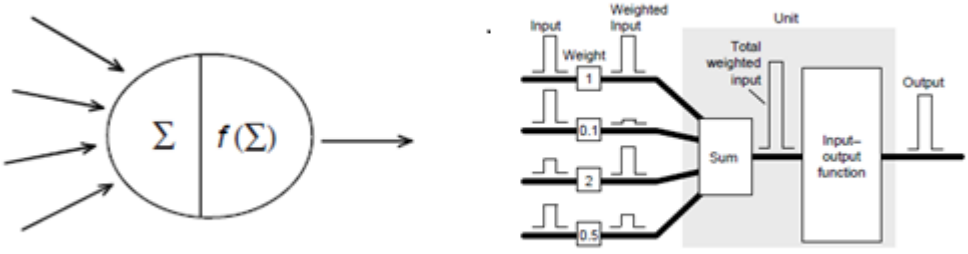

Figure 2. Inputs are weighted, summed and passed through the input-output function

The network is first initialized by setting up all its weights to be small random numbers – say between 0 and 1 (Amardeep & K, 2017). In backward propagation, the margin of error of the output was measured and the weights were adjusted back from the output layer to the input layer accordingly to decrease the error using gradient descent. Neural networks repeat both the forward and backward propagation until the weights are calibrated to accurately predict an output. The forward and backward propagation is performed repetitively on each observation and the adjusted weights at the end of the final observation are applied on whole data set to check the average square error (ASE) and misclassification rate (MCR). Each repetition of the forward and backward propagation to adjust weights on whole data set is called an iteration or an epoch (Gurney, An Introduction to Neural Networks, 1997).

3. OUR APPROACH

In this project, a real dataset provided by a financial company was used. This data contains 5000 observations and more than 500 attributes. For the sake of simplicity, we selected the best five predictors to train the model. In order to reduce the model development time, from the 5000 observations, we randomly selected 1000 observations to train the model and 600 observations to test the model.

We developed macros in Base SAS to implement a standard feed forward backward propagation neural network with one hidden layer (Figure 3). Three separate macros were created for forward propagation, backward propagation and to control the number of iterations (epochs). In this model, we used the sigmoid activation function to transform the input signal into an output signal. The first observation was forward propagated with the random weights, the error was then calculated, and the weights were adjusted through backward propagation. The adjusted weights of the first observation were transferred to the second observation and the process is repeated until the last observation (Figure 4). Once the weights are updated for last observation, those weights are used in the forward propagation of the training and testing data to generate ASE and misclassification rate. This whole process is termed as one iteration. The weights from the first iteration are fed into the second iteration and the process is repeated until the desired number of iterations. Figure 4 depicts the algorithm in a graphical way.

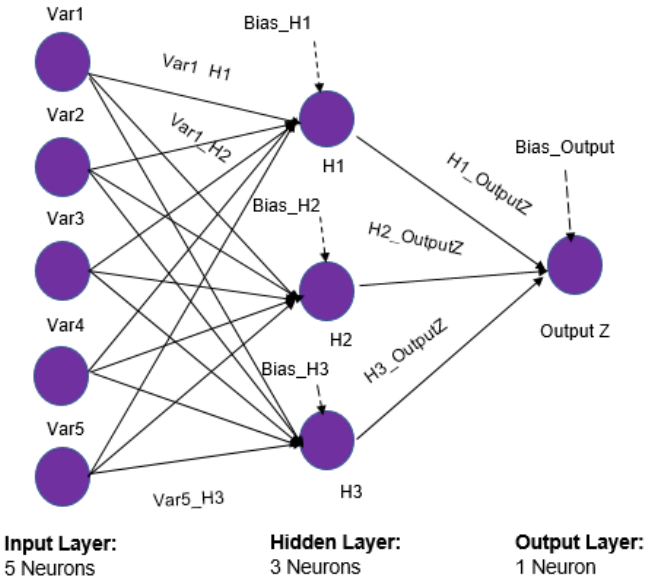

Figure 3. Neural Network Architecture

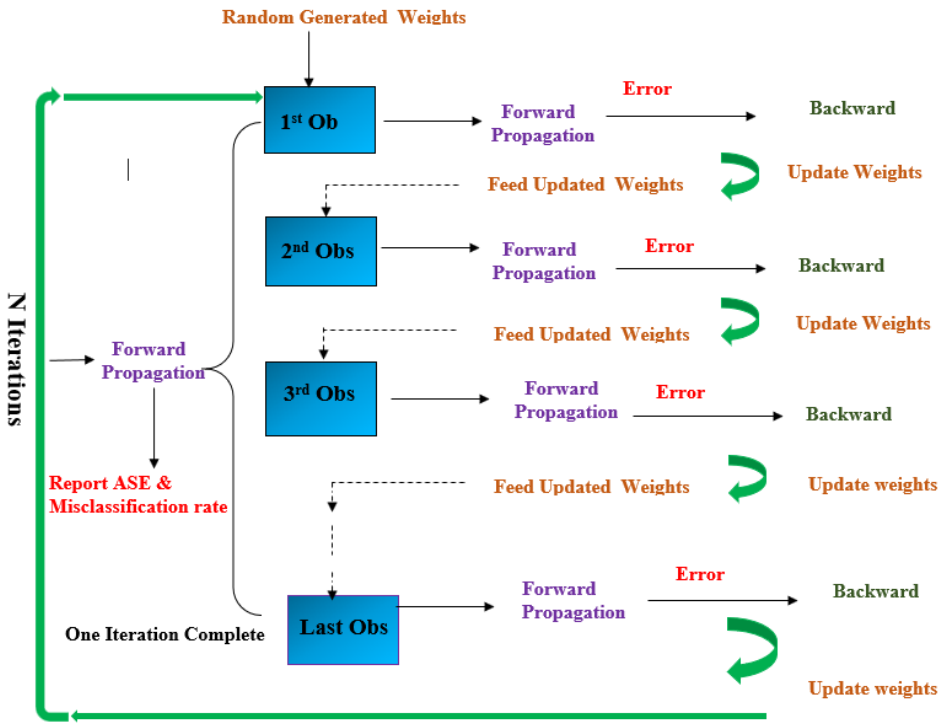

Figure 4. Feed Forward and Backward Propagation Algorithm

Our model has five input neurons (variables), three hidden neurons and one output neuron (Figure 3). Hence, there are fifteen *input-hidden weights*, three *input-hidden bias*, three *hidden-output weights* and one *hidden-output bias*. The components of the neural net architecture are described as below (Table 1).

Variables	Variable Codes in Architecture
Input Neurons	Var1, Var2, Var3, Var4, Var5
Input-Hidden Weights	Var1_H1, Var1_H2, Var1_H3, Var2_H1, Var2_H2, Var2_H3, Var3_H1, Var3_H2, Var3_H3, Var4_H1, Var4_H2, Var4_H3, Var5_H1, Var5_H2, Var5_H3
Input-Hidden Bias	Bias_H1, Bias_H2, Bias_H3
Hidden Nodes	H1, H2, H3
Hidden-Output Weights	H1_Output, H2_Output, H3_Output
Hidden-Output Bias	Bias_Output
Output Neuron	Output Z

Table 1. ANN Architecture Variables

Other variables of the algorithm that determine the learning process are defined in Table 2.

How an observation (instance) is classified	**Actual**
Error after each learning iteration (Actual-OutputZ)	**NNet Error**
If the OutputZ > 0.5 then Predicted = 1 else Predicted =0	**Predicted**

Table 2. Learning Process Variables

4. STEP BY STEP IMPLEMENTATION OF ALGORITHM

We developed several macros to implement the Neural Network Algorithm. They are shown in the following seven steps.

1) We created the main macro NeuralTest with the following parameters**.**

```
/*******************************************************/
* Macro #1: NeuralTest
* Description: Created the main macro with the below parameters so that any dataset can be
fed and intermediate variables like weights and biases are created

* Following are the parameters defined for the Macro:

Dsn                    =    Dataset name
Hiddensuffix           =    Suffix for hidden layer weights & bias
Out suffix             =    Suffix for output layer weights & bias
Outputnodes            =    Number of output nodes in neural network
LR                     =    Learning rate
StartIteration         =    Iteration number to start with
EndIteration           =    Iteration number to end with
*
/*******************************************************/

%macro neuraltest(dsn=, hiddensuffix=, hiddennodes=, outsuffix=,
                  outputnodes=, LR=, StartIteration=, EndIteration=);
/* Open the data set */
%let dsid=%sysfunc(open(&dsn));
/* The variable nvar will contain the number of variables that are in
the dataset*/
%let nvar=%sysfunc(attrn(&dsid, nvars));

/* Using the GENERATEVARLIST macro to gnerate variables names in the
datasets*/
%GENERATEVARLIST(DSN=SOU_FOR.curnnetvar_stdtrain, exclude=RESP_DV MK);
%put &varname;
%let inputvarname = &varname
```

2) As a next step, we need to prepare the data for forward propagation. Below are the steps involved.

a) Generate *input-hidden weights* by adding the suffix '_H' followed by a number for each variable. This number is based on which hidden neuron the weights from the input variable correspond to. For ex., hidden weight Var1_H1 corresponds to the weight from Variable (Var1) to the hidden node (H1). A total number of hidden weights is a product of the number of input variables and a number of hidden nodes. To generate the appropriate number of

hidden input weights nodes with the required naming convention the following code is implemented. All the *input-hidden weights* are assigned random values and stored in the data set 'hidden weights':

```
data hidden_weights (drop=&inputvarname);
set &dsn;
%do p=1 %to &nvar;
 %let var=%sysfunc(varname(&dsid, &p));
   %do q=1 %to &hiddennodes;
     &var=&var&hiddenextension&q;
     &var&hiddenextension&q=rand('uniform');
     %end;
%end;
/* Close the data set */
%let rc=%sysfunc(close(&dsid));
run;

%GENERATEVARLIST(DSN=hidden_weights, exclude=RESP_DV MK);
%put &varname;
%let hidden_weights_name = &varname;
%put &hidden_weights_name;
```

b) Generate *hidden-output weights* by adding the suffix '_Output' to hidden node variables. For ex, *hidden-output weight* (H1_Output) corresponds to the weight from the hidden node (H1) to OutputZ. A total number of hidden output weights is a product of the number of hidden nodes and a number of output nodes. To generate the appropriate number of hidden output weights with the required naming convention the following code is implemented. All the hidden output weights are assigned random values and stored in the data set 'output_weights':

```
data hidden_node;
%do s=1 %to &hiddennodes;
 H&s=0;
 %end;
 run;

 %GENERATEVARLIST(DSN=hidden_node, exclude=RESP_DV MK);
 %put &varname;
 %let hidden_nodename= &varname;
 %put &hidden_nodename;

 /* Randomly generating the weights for output node and bias*/
 %let dsn2 = hidden_node;
 %put &dsn2;
 %let dsid2=%sysfunc(open(&dsn2));
 %put &dsid2;
 %let n=%sysfunc(attrn(&dsid2, nvars));

 data output_weights (drop=&hidden_nodename);
 set hidden_node;
 %do t=1 %to &hiddennodes;
 %let var=%sysfunc(varname(&dsid2, &t));
   %do u=1 %to &outputnodes;
        &var=&var&outextension;
        &var&outextension=rand('uniform');
```

```
%end;
%end;
%let rc=%sysfunc(close(&dsid2));
run;

%GENERATEVARLIST(DSN=output_weights, exclude=RESP_DV MK);
%put &varname;
%let output_weights_name = &varname;
%put &output_weights_name;
```

c) Repeat the steps a) and b) to generate the *input-hidden bias* and the *hidden-output bias* variables and store them in separate datasets 'hidden_intercept' and 'out_intercept' respectively. Also generate the output node variable (OutputZ) in the dataset 'output_node':

```
data hidden_intercept;
%do r=1 %to &hiddennodes;
BIAS_H&r=rand('uniform');
%end; run;

%GENERATEVARLIST(DSN=hidden_intercept, exclude=RESP_DV MK);
%put &varname;
%let hidden_interceptname= &varname;
%put &hidden_interceptname;

data out_intercept;
BIAS&outextension=rand('uniform');
run;

%GENERATEVARLIST(DSN=out_intercept, exclude=RESP_DV MK);
%put &varname;
%let out_interceptname= &varname;
%put &out_interceptname;

data output_node;
OutputZ=0;
run;

%GENERATEVARLIST(DSN=output_node, exclude=RESP_DV MK);
%put &varname;
%let output_nodename= &varname;
%put &output_nodename;

data error;
NNetError=0;
NNetErrorSQ = 0;
NNetPred1 = 0;
NNetPred0=0;NNPrediction=0; NNCorrect=0;
run;

%GENERATEVARLIST(DSN=error, exclude=RESP_DV MK);
%put &varname;
%let error_nodename = &varname;
%put &error_nodename;
```

3) Prepare the data for Backward Propagation. Below are the steps involved.

a) Generate variables to capture the changes in hidden nodes during the backward propagation. We call these variables delta variables by adding the prefix 'delta' and store the variables in dataset 'deltahidden_node' (deltaH1, deltaH2 etc):

```
data deltahidden_node;
%do dhn=1 %to &hiddennodes;
 DELTAH&dhn=0;
   %end;
    run;
```

b) Repeat the step (3a) for *input-hidden weights*, *input-hidden bias*, *hidden-output weights* and *hidden-output bias* during backward propagation and store these variables in respective datasets deltahidden_weights, deltahidden_intercept, deltaoutput_weights:

```
data deltaoutput_weights(drop=&output_weights_name);
set output_weights;
%do dow=1 %to &hiddennodes;
%let var=%sysfunc(varname(&dsid3, &dow));
%put &var;
&var=delta&var;
delta&var=0;
%put &var;
%end;
%let rc=%sysfunc(close(&dsid3));
run;

%let dsn4 = hidden_intercept;
%put &dsn4;
%let dsid4=%sysfunc(open(&dsn4));
%put &dsid4;

data deltahidden_intercept(drop=&hidden_interceptname);
set hidden_intercept;
%do dhi=1 %to &hiddennodes;
%let var=%sysfunc(varname(&dsid4, &dhi ));
%put &var;
&var=delta&var;
 delta&var=0;
 %put &var;
 %end;
 %let rc=%sysfunc(close(&dsid4));

 %let dsn5 = hidden_weights;
 %put &dsn5;
 %let dsid5=%sysfunc(open(&dsn5));
 %put &dsid5;

 %let dhiddenvar=%sysevalf(&nvar*&hiddennodes);
 %put &dhiddenvar;
data deltahidden_weights(drop=&hidden_weights_name);
set hidden_weights;
%do dhw=1 %to &dhiddenvar;
   %let var=%sysfunc(varname(&dsid5, &dhw ));
```

```
      %put &var;
     &var=delta&var;
    delta&var=0;
   %put &var;
%end;
%let rc=%sysfunc(close(&dsid5));
run;

%GENERATEVARLIST(DSN=deltahidden_node, exclude=RESP_DV MK);
%put &varname;
%let deltahidden_nodename= &varname;
%put &deltahidden_nodename;

%GENERATEVARLIST(DSN=deltaoutput_weights, exclude=RESP_DV MK);
%put &varname;
%let deltaoutput_weightname= &varname;
%put &deltaoutput_weightname;

%GENERATEVARLIST(DSN=deltahidden_intercept, exclude=RESP_DV MK);
%put &varname;
%let deltahidden_interceptname= &varname;
%put &deltahidden_interceptname;


%GENERATEVARLIST(DSN=deltahidden_weights, exclude=RESP_DV MK);
%put &varname;
%let deltahidden_weightname= &varname;
%put &deltahidden_weightname;
```

4) Merge all the datasets created in the previous steps:

```
data neuraltestdata;
merge hidden_weights hidden_intercept hidden_node output_weights out_intercept
      output_node error ;
run;

data neuraltestdata_delta ;
     merge neuraltestdata deltahidden_weights deltahidden_intercept deltahidden_node
     deltaoutput_weights ;
run;
```

5) Another macro named FrontPropogation was created to perform forward propagation using the Sigmoid activation function. Forward propagation was discussed elaborately in (Zhang, Dupree, Shah, & Torres, 2005)

```
/*******************************************************/
* Macro #2: FrontPropagation
* Description: Created FrontPropagation macro with the parameters: 'fpdsn' and 'fpdsnout'

 Parameter 'fpdsn' takes the dataset to be forward propagated and performs forward propagaton
using sigmoid activation function and random weights. The forward propagated dataset is being fed
into 'fpdsnout' parameter.
*
/*******************************************************/
```

Following are the steps involved in forward propagation:

a) Create an array for the input variables (Var1- Var 5);
b) Create an array for the *input-hidden weights* variables (Var1_H1- Var 5_H1) and then loop the array as many times as the number of hidden nodes;
c) Create an array for the *input-hidden bias* variables (BIAS_H1- BIAS_H3);
d) Create an array for the *hidden-output weights* variables and *hidden-output bias* variables (H1_RESP_DV -H3_RESP_DV) and BIAS_RESP_DV :

```
%MACRO neuraltestdata(fpdsn=, fpdsnout=);

data &fpdsnout(drop=k l a b);
set &fpdsn;
array inputdata(*) &inputvarname;
array hiddenweights(&nvar, &hiddennodes) &hidden_weights_name;
array hiddenintercepts(*) &hidden_interceptname;
array hiddennodes(*) &hidden_nodename;
array outweights (&hiddennodes, &outputnodes) &output_weights_name;
array outintercepts(*) &out_interceptname;
array outputnodes(*) &output_nodename;
```

e) To calculate the hidden node values, multiply each input variable with their corresponding weights and sum up across all variables. Then, the activation function is applied to this value to generate the output from the node. This process is repeated for all nodes in the hidden layer and the output layer and the corresponding values are stored:

```
put k= hiddennodes(k)= hiddenintercepts(k)=;
hiddennodes(k) = hiddennodes(k)+ hiddenintercepts(k);
do l=1 to &nvar;
put k=l=hiddennodes(k)=;
hiddennodes(k)=hiddennodes(k)+(inputdata(l) * hiddenweights(l,k));
put k=l=inputdata(l)=hiddenweights(l, k)=;
end;
put k=l=hiddennodes(k)=;
hiddennodes(k)= 1/(1+exp(-hiddennodes(k)));
put k=l=hiddennodes(k)=;
end;

do a = 1 to &outputnodes;
put a= outputnodes(a)= outintercepts(a)=;
outputnodes(a) = outintercepts(a);
put a= outputnodes(a)= outintercepts(a)=;
  do b = 1 to &hiddennodes;
  put a=b=outputnodes(a)=;
  outputnodes(a)=outputnodes(a)+(hiddennodes(b) * outweights(b,a));
  put a=b=hiddennodes(b)= outweights(b,a)=;
  end;
put a=b=outputnodes(a)=;
outputnodes(a)= 1/(1+exp(-outputnodes(a)));
  do k=1 to &hiddennodes;
 put a=b=outputnodes(a)=;
 end;

 NNetError=RESP_DV-OutputZ; NNetErrorSQ = NNetError*NNetError;
```

```
NNetPred1 = OutputZ;
NNetPred0=1-NNetPred1;
if NNetPred1 > NNetPred0 then NNPrediction= 1;
else NNPrediction= 0;
NNCorrect = (NNPrediction=RESP_DV);
run;
%MEND ;
```

6) Another macro, Backpropagation was created with two parameters: 'bpdsn' and 'bpdsnout' to perform backward propagation

```
/*****************************************************/
* Macro #2: BackPropagation
* Description: Created FrontPropagation macro with the parameters: 'fpdsn' and 'fpdsnout'

 Parameter 'bpdsn' takes the dataset to be back propagated and performs back propagaton using
sigmoid activation function and random weights. The back propagated dataset is being fed into
'bpdsnout' parameter.
*
/*****************************************************/
 %MACRO BACKPROPAGATION (bpdsn=, bpdsnout= );

 Data &bpdsnout(drop=k l);
 set &bpdsn;
 Array inputdata(*) &inputvarname;
 Array hiddennodes(*) &hidden_nodename;
 Array deltahiddennodes(*) &deltahidden_nodename;
 Array deltahiddenoutweight(*) &deltaoutput_weightname;
 Array deltahiddenintercept(*) &deltahidden_interceptname;
 Array hiddenintercepts(*) &hidden_interceptname;
 Array outweightsbp (*) &output_weights_name;
 Array hiddenweights(&nvar, &hiddennodes) &hidden_weights_name;
 Array deltahiddenweights(&nvar, &hiddennodes)
 &deltahidden_weightname;
```

During this process several delta variables for the hidden and the output layer were created using arrays and the calculations were performed as below:

```
/*Error responsibility for Node Z, an output node*/
/*DELTAZ=outputZ (1-outputZ )(actualZ-outputZ )*/
DELTAZ= OutputZ*(1-OutputZ)*(Resp_DV-OutputZ);
put DELTAZ=;
LR=&LR;
put LR=;
deltaBIAS_RESP_DV = LR*DELTAZ*1;
put deltaBIAS_RESP_DV= ;
put BIAS_RESP_DV=;
BIAS_RESP_DV=(BIAS_RESP_DV) + (deltaBIAS_RESP_DV);
put deltaBIAS_RESP_DV=;
put BIAS_RESP_DV=;
/*Move upstream to Node H1 H2 & H3, hidden layer nodes*/
/*Only node downstream from Node H1, H2, H3 is Node Z*/
do k=1 to &hiddennodes;
/*deltaH1 = H1*(1-H1)*H1_RESP_DV*deltaZ*/
deltahiddennodes(k)= (hiddennodes(k)*(1-hiddennodes(k))
*outweightsbp (k)*DELTAZ);
```

```
/*Adjust weight wAZ using back-propagation rules*/
deltahiddenoutweight(k)=LR*DELTAZ*hiddennodes(k);
put k=outweightsbp(k)=deltahiddenoutweight(k)=;
outweightsbp(k) =outweightsbp(k) + deltahiddenoutweight(k);
put k=outweightsbp(k)=;
put k=deltahiddenintercept(k)=deltahiddennodes(k)=;
deltahiddenintercept(k)=LR*deltahiddennodes(k)*1;
put k=deltahiddenintercept(k)=deltahiddennodes(k)=;
hiddenintercepts(k)=hiddenintercepts(k)+deltahiddenintercept(k);
put k=hiddenintercepts(k)=;
end;
do k=1 to &hiddennodes;

   do l=1 to &nvar;
   deltahiddenweights(l,k)= LR* deltahiddennodes(k)*inputdata(l);
   put k=l=deltahiddenweights(l,k)=;
   end;
end;

do k=1 to &hiddennodes;

   do l=1 to &nvar;
   put k=l=hiddenweights(l,k)= deltahiddenweights(l,k)=;
   hiddenweights(l,k)= hiddenweights(l,k)+deltahiddenweights(l,k);
   put k=l=hiddenweights(l,k)=;
   end;
                     end;
                     run;

%MEND;

/* created a dummy dataset so that SAS does not throw an error
          when started the first iteration*/

data SOU_FOR.neuraltestdataFPTRAIN_ITR0;
set neuraltestdata;
run;

/* If Iteration is starting from 1 we will generate the dataset
with random weights for forward Propagation Else will take the
 weights from the previous batch*/

%IF &StartIteration=1 %THEN
%do;

data neuraltestdataFP;
set neuraltestdata_delta;
run;
%end;

%ELSE;

%do;
%let X= %SYSEVALF(&StartIteration -1);
%put &X;
data neuraltestdataFP;
set SOU_FOR.neuraltestdataFPTRAIN_ITR&X;
```

```
run;
%end;
```

7) Created the final macro NeuralIteration to implement the desired number of iterations:

```
%MACRO NEURALITERATION;

%do ITER= &StartIteration %to &EndIteration;
    %do loop=1 %to &TRAINNUMOBS;

data curnnetvar_s;
set SOU_FOR.curnnetvar_stdtrain (firstobs=&loop obs=&loop);
run;

data neuraltestdataFP;
merge curnnetvar_s neuraltestdataFP;
run;

%FRONTPROPAGATION(fpdsn=neuraltestdataFP,fpdsnout=neuraltestdataFP);

%BACKPROPAGATION(bpdsn=neuraltestdataFP,bpdsnout=neuraltestdataBP);

data neuraltestdataFP;
set neuraltestdataBP(drop= &inputvarname resp_dv);
run;

%end;

/* Saving each iteration weights to use in subsequent batches*/
 data SOU_FOR.neuraltestdataFPTRAIN_ITR&ITER;
 set neuraltestdataBP(drop= &inputvarname RESP_DV);
 run;

/*Dropping input variables and response variables*/
 data neuraltestdataFPTRAIN_ITR1;
 set neuraltestdataBP(drop= &inputvarname RESP_DV);
 run;

/* Saving weights in the separate dataset after each
      iteration */

 data SOU_FOR.neuralWeightsTrain_ITR&ITER;
 set neuraltestdataFPTRAIN_ITR1(keep= &hidden_weights_name
&hidden_interceptname &output_weights_name &out_interceptname);
 run;

/* concatenating  all the datasets with weights and saved
the dataset with name NEURALWeightsAllIterations_TRAIN*/

 data SOU_FOR.NEURALWeightsAllIterations_TRAIN;
 set SOU_FOR.neuralWeightsTRAIN_ITR1-
SOU_FOR.neuralWeightsTRAIN_ITR&ITER;
 run;

/* forward propagating the last observation weight on
whole training dataset*/
data neuraltestdataFPTRAIN_ITR1(drop=i);
```

```
 set neuraltestdataFPTRAIN_ITR1;
    do i = 1 to &TRAINNUMOBS;
       output;g
        end;
     run;

 data neuraltestdataFPTRAIN_ITR1;
 merge SOU_FOR.curnnetvar_stdtrain  neuraltestdataFPTRAIN_ITR1;
 run;

 %FRONTPROPAGATION(fpdsn=neuraltestdataFPTRAIN_ITR1,
                   fpdsnout=neuraltestdataFPTRAIN_FinaLITR&ITER);

 /* calculating the Fit statistics on training dataset*/
 proc sql;
 create table SOU_FOR.NEURALTRAINITER&ITER as
 select mean(nneterrorsq) as NNASE, sum(NNCorrect )/count(*) as
NNAccuracy, (1-(sum(NNCorrect )/count(*))) as NNMCR from
neuraltestdataFPTRAIN_FinaLITR&ITER;
 quit;

 /* Appending all Iterations Fit Statistics into one datset*/
 data SOU_FOR.NEURALTRAINSUMMARY;
 set SOU_FOR.NEURALTRAINITER1-SOU_FOR.NEURALTRAINITER&ITER;
 run;

 /* Merging the Weights and Fit Statistics into dataset for all
 Iteration*/
 data SOU_FOR.NNIterationsWeights_TRAINSummary;
 merge SOU_FOR.NEURALWeightsAllIterations_TRAIN
 SOU_FOR.NEURALTRAINSUMMARY;
 run;

 /*Dropping input variables and response variables*/
 data neuraltestdataFPVALID_ITR1;
 set neuraltestdataBP(drop= &inputvarname RESP_DV);
 run;

 /* Saving the weights in the separate dataset after each iteration with
 name of dataset as neuralWeightsValid_ITR1,neuralWeightsValid_ITR2
 etc*/
 data SOU_FOR.neuralWeightsValid_ITR&ITER;
 set neuraltestdataFPVALID_ITR1(keep= &hidden_weights_name
 &hidden_interceptname &output_weights_name &out_interceptname);
 run;

 /* concatenating all the datasets with weights and saved the dataset
 with name NEURALWeightsAllIterations_Valid*/


 data SOU_FOR.NEURALWeightsAllIterations_Valid;
  set SOU_FOR.neuralWeightsValid_ITR1-
 SOU_FOR.neuralWeightsValid_ITR&ITER;
  run;

  /* Forward propagating the last observation weight on whole validation
 dataset*/
```

```
data neuraltestdataFPVALID_ITR1(drop=i);
set neuraltestdataFPVALID_ITR1;
   do i = 1 to &VALIDNUMOBS;
      output;
      end;run;

data neuraltestdataFPVALID_ITR1;
merge SOU_FOR.curnnetvar_stdvalid neuraltestdataFPVALID_ITR1;
run;

%FRONTPROPAGATION(fpdsn=neuraltestdataFPVALID_ITR1,
                  fpdsnout= neuraltestdataFPVALID_FinaLITR&ITER);

/* calculating the Fit statisics on validation dataset*/
proc sql;
create table SOU_FOR.NEURALVALIDITER&ITER as
select mean(nneterrorsq) as NNASE, sum(NNCorrect )/count(*) as
NNAccuracy,
      (1-(sum(NNCorrect )/count(*))) as NNMCR from
neuraltestdataFPVALID_FinaLITR&ITER;
quit;

/* concatenating all valid iterations accuracy and MCR into one
dataset*/
data SOU_FOR.NEURALVALIDSUMMARY;
set SOU_FOR.NEURALVALIDITER1-SOU_FOR.NEURALVALIDITER&ITER;
run;

data SOU_FOR.NNIterationsWeights_ValidSummary;
merge SOU_FOR.NEURALWeightsAllIterations_Valid
SOU_FOR.NEURALVALIDSUMMARY;
run;
%end;
 /* Ending the macro NEURALITERATION */
%MEND;
%NEURALITERATION;
/*Ending the main macro neuraltest */
%MEND;

/*Running the main macro by passing all parameters*/
%neuraltest(dsn=curnnetvar1, hiddensuffix=_H, hiddennodes=3, LR=0.1,
outsuffix=_Output,outputnodes=1, StartIteration=4, EndIteration=6);
```

To better illustrate how the input data changes within each step of the forward and backward propagation we captured those changes and presented in the following steps.

Note: We request the reader not to confuse with the seven steps mentioned in the Section 4 as the following steps capture only the effects of forward and backward propagation on the data set. Whereas, the steps above illustrate the whole program including data preparation.

Step 1: Input variables are normalized and all other weights, biases, and hidden neurons (H1, H2, H3) are kept zero.

Var1	Var2	Var3	Var4	Var5	Var1_H1	Var2_H1	Var3_H1	Var4_H1	Var5_H1	Bias_H1
1	0.62249	1.0143	0.7190	1.1830	0	0	0	0	0	0

Var1_H2	Var2_H2	Var3_H2	Var4_H2	Var5_H2	Bias _H2	Var1_H3	Var2_H3	Var3_H3
0	0	0	0	0	0	0	0	0

Var4_H3	Var5_H3	Bias _H3	H1_Output	H2_Output	H3_Output	Bias _Output
0	0	0	0	0	0	0

H1	H2	H3	Actual	OutputZ	NNetError	Predicted
0	0	0	0	0	0	0

Step 2: In the second step, the initial weights of the first observation are randomly generated. You may notice that the hidden neurons and the final output – **OutputZ** – are still zero as no computation has taken place so far.

Var1	Var2	Var3	Var4	Var5	Var1_H1	Var2_H1	Var3_H1	Var4_H1	Var5_H1	Bias_H1
1	0.62249	1.0143	0.7190	1.1830	0.92087	0.70797	0.62601	0.90365	0.11236	0.64727

Var1_H2	Var2_H2	Var3_H2	Var4_H2	Var5_H2	Bias _H2	Var1_H3	Var2_H3	Var3_H3
0.17533	0.58943	0.13338	0.52636	0.24904	0.18099	0.91941	0.31108	0.36089

Var4_H3	Var5_H3	Bias _H3	H1_Output	H2_Output	H3_Output	Bias _Output
0.98716	0.44222	0.98709	0.13747	0.33574	0.89631	0.44536

H1	H2	H3	Actual	OutputZ	NNetError	Predicted
0	0	0	0	0	0	0

Step 3: In this step, the hidden neurons (H1, H2, H3), the output value (OutputZ), the Predicted value (1 if OutputZ >0.5 and 0 otherwise), and NNetError (The difference between Actual and OutputZ) are updated after forward propagation. You may notice that NNetError is quite high and the Predicted value of '1' differs from the Actual value of '0'.

Var1	Var2	Var3	Var4	Var5	Var1_H1	Var2_H1	Var3_H1	Var4_H1	Var5_H1	Bias_H1
1	0.62249	1.0143	0.7190	1.1830	0.92087	0.70797	0.62601	0.90365	0.11236	0.64727

Var1_H2	Var2_H2	Var3_H2	Var4_H2	Var5_H2	Bias _H2	Var1_H3	Var2_H3	Var3_H3
0.17533	0.58943	0.13338	0.52636	0.24904	0.18099	0.91941	0.31108	0.36089

Var4_H3	Var5_H3	Bias _H3	H1_Output	H2_Output	H3_Output	Bias _Output
0.98716	0.44222	0.98709	0.13747	0.33574	0.89631	0.44536

H1	H2	H3	Actual	OutputZ	NNetError	Predicted
0.96852	0.82225	0.97586	0	0.84931	0.84931	1

Step 4: Weights are updated using backward propagation and you may notice that the hidden and output node values, NNetError and Predicted values remain the same. Weights are updated minutely in the 1/10000s and this correction depends upon the learning rate. In this algorithm, we used a learning rate of 0.1. These updated weights are transferred to the next observation and Step 3 is repeated. A continuous loop of step 3 and step 4 is performed until the last observation.

Var1	Var2	Var3	Var4	Var5	Var1_H1	Var2_H1	Var3_H1	Var4_H1	Var5_H1	Bias_H1
1	0.62249	1.0143	0.7190	1.1830	0.92082	0.70794	0.62596	0.90361	0.11230	0.64723

Var1_H2	Var2_H2	Var3_H2	Var4_H2	Var5_H2	Bias _H2	Var1_H3	Var2_H3	Var3_H3
0.17480	0.58910	0.13284	0.52597	0.24841	0.18045	0.91919	0.31093	0.36066

Var4_H3	Var5_H3	Bias _H3	H1_Output	H2_Output	H3_Output	Bias _Output
0.98700	0.44195	0.98686	0.12694	0.32680	0.88571	0.43449

H1	H2	H3	Actual	OutputZ	NNetError	Predicted
0.96852	0.82225	0.97586	0	0.84931	0.84931	1

Step 5: Forward propagation was run on all the observations using the updated weights from the last observation. This completes one iteration. After the first iteration, you can see a very slight change in NNetError. In fact, it has increased slightly from 0.8493 (Step 3) to 0.85015 (Step 5) but this increase is temporary. In general, the NNetError decreases as the iterations increase.

Var1	Var2	Var3	Var4	Var5	Var1_H1	Var2_H1	Var3_H1	Var4_H1	Var5_H1	Bias_H1
1	0.62249	1.0143	0.7190	1.1830	0.92087	0.70797	0.62601	0.90365	0.11236	0.64727

Var1_H2	Var2_H2	Var3_H2	Var4_H2	Var5_H2	Bias _H2	Var1_H3	Var2_H3	Var3_H3
0.17533	0.58943	0.13338	0.52636	0.24904	0.18099	0.91941	0.31108	0.36089

Var4_H3	Var5_H3	Bias _H3	H1_Output	H2_Output	H3_Output	Bias _Output
0.98716	0.44222	0.98709	0.13747	0.33574	0.89631	0.44536

H1	H2	H3	Actual	OutputZ	NNetError	Predicted
0.98781	0.91302	0.99075	0	0.85015	0.85015	1

Step 6: After 100 iterations NNetError reduces substantially to 0.18312 and the Predicted value becomes '0' and matches the Actual value.

Var1	Var2	Var3	Var4	Var5	Var1_H1	Var2_H1	Var3_H1	Var4_H1	Var5_H1	Bias_H1
1	0.62249	1.0143	0.7190	1.1830	0.92369	0.70972	0.62887	0.90567	0.11569	0.65009

Var1_H2	Var2_H2	Var3_H2	Var4_H2	Var5_H2	Bias _H2	Var1_H3	Var2_H3	Var3_H3
0.17843	0.59136	0.13653	0.52859	0.25270	0.18409	0.91554	0.30867	0.35696

Var4_H3	Var5_H3	Bias _H3	H1_Output	H2_Output	H3_Output	Bias _Output
0.98438	0.43764	0.98322	-0.71269	-0.45459	0.043792	-0.41526

H1	H2	H3	Actual	OutputZ	NNetError	Predicted
0.98822	0.92202	0.99071	0	0.18312	0.18312	0

5. ALGORITHM PERFORMANCE – A COMPARISON BETWEEN OUR CODES AND PYTHON

As expected, standardizing the input data is critical for this algorithm to perform. This model is scalable to increase the input variables and any number of hidden nodes. A hundred iterations were run on the training

(Figure 5) and the testing data (Figure 6) and the accuracy, misclassification rate and ASE were compared with the same model ran in Python. The number of iterations depends upon the overall classification rate error. We give the comparison in detail regards to the accuracy rate in Figure 5 and Figure 6.

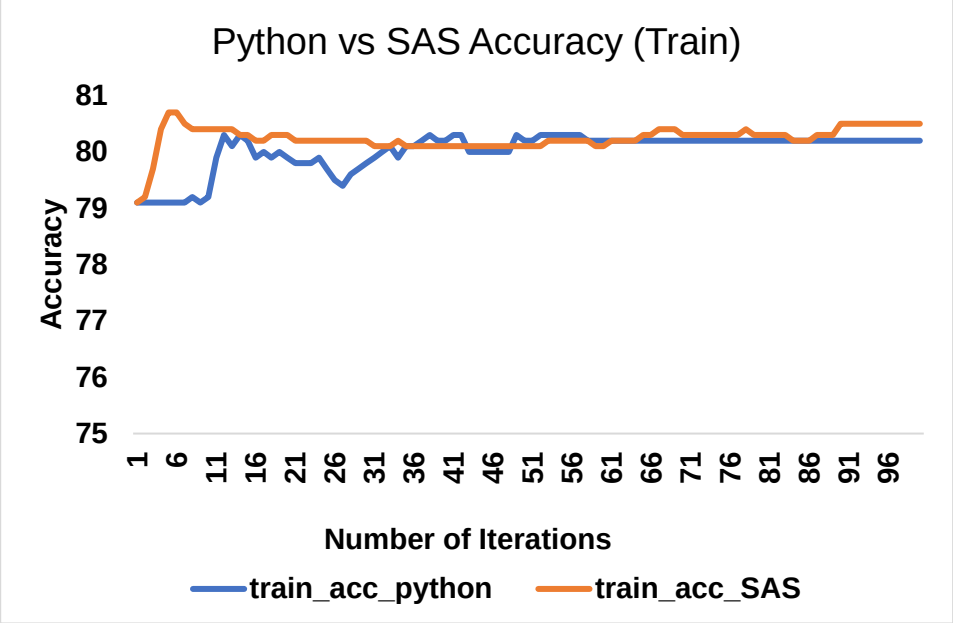

Figure 5. The accuracy of the model on the training data in SAS and Python

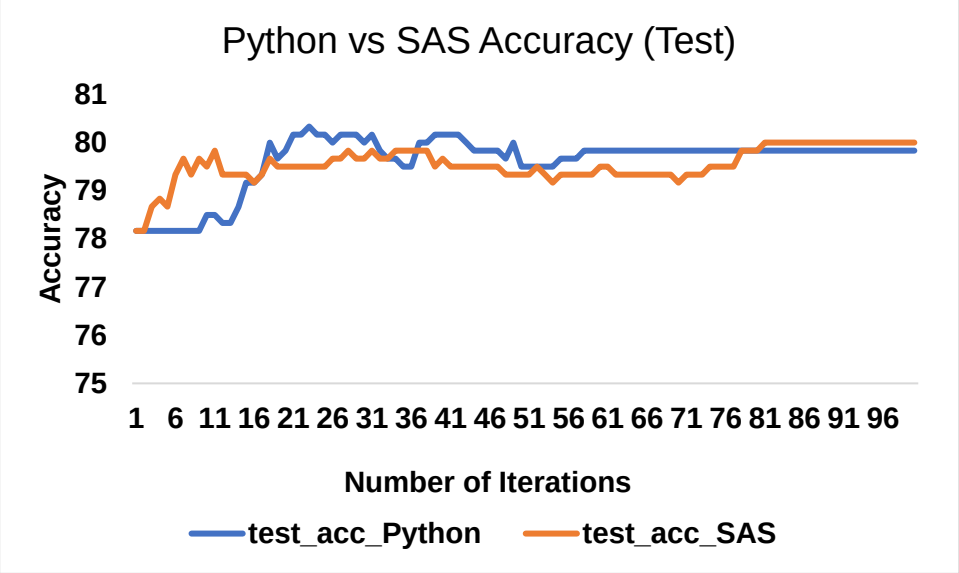

Figure 6. The accuracy of the model on the testing data in SAS and Python

The highest accuracy for SAS Algorithm was attained in the 6th iteration for the training data (80.7%) and the 81st iteration for the testing data (80.0%). The highest accuracy in Python was attained in the 14th iteration for the training data (80.3%) and the 23rd iteration for the testing data (80.3%). The efficiency of the algorithm in Base SAS is similar to Python but it is considerably slower. It took 2 minutes to run the

algorithm in SAS on a dataset that contains a thousand observations and five input variables for one iteration. But the implementation in Python is much quicker. Overall, for the dataset with 7500 observations and five variables, it took around two hours to complete two iterations. Python has vectorized operations using NumPy Library and it is very efficient and faster. This may be one of the drawbacks of implementing this algorithm in Base SAS and macros. However, SAS Enterprise Miner is relatively faster than Base SAS.

CONCLUSION

In this paper, we have shown that Neural Networks can be built in Base SAS with the very close accuracy as in Python. This paper essentially explains the nuts and bolts of the Feedforward Back Propagation algorithm. This algorithm can be useful for financial companies which plan to test their models and compare the efficiency with other machine learning models. Often, Neural Networks are termed as black box machine learning models but it is critical to comprehend the algorithm as it helps to avoid model tuning pitfalls and train the models better.

REFERENCES

Amardeep, R., & K, T. (2017). Training Feed forward Neural Network With Backpropagation Algorithm. *International Journal of Engineering And Computer Science*, 19860-19866.

Gurney, K. (1997). *An Introduction to Neural Networks.* London, UK: UCL Press Ltd.

Larose, D. T. (2004). *Discovering Knowledge in Data: An Introduction to Data Mining.* Hoboken: John Wiley & Sons.

Rashid, T. (2016). *Make Your Own Neural Network: A Gentle Journey Through the Mathematics of Neural Networks, and Making Your Own Using the Python Computer Language.* Scotts Valley: Createspace Independent Publication.

Samarasinghe, S. (2006). *Neural Networks for Applied Sciences and Engineering:From Fundamentals to Complex Pattern Recognition.* Boca Raton, Florida, USA: Auerbach Publications.

Sarle, W. S. (1994). Neural Network Implementation in SA Software. *Proceedings of the Nineteenth Annual SAS User Groups Conference.* Dallas: SAS Institute. Retrieved from http://support.sas.com/resources/papers/proceedings09/TOC.html

Zhang, S., Dupree, J., Shah, U., & Torres, M. (2005). Techniques and Methods to Implement Neural Networks in Using SAS and .net. *South-Central SAS Users Group.* San Antonio: SAS Institute. Retrieved from https://www.lexjansen.com/scsug/2005/Zheng_Techniques%20and%20Methods%20Neural%20Networks%20-%20371.pdf

ACKNOWLEDGMENTS

We are grateful to Brian Stone, Chief Risk Officer at Atlanticus for giving this opportunity. This work would not have been possible without the financial support of the Center for Statistics and Analytics Research (CSAR), Kennesaw State University. We would also like to thank Dr. Herman (Gene) Ray for giving valuable inputs to the project.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Soujanya Mandalapu
Applied Statistics Graduate Student
Kennesaw State University
(443) 905-0009
Soujanya2cu@gmail.com

Yan Wang
Ph.D. Candidate in Analytics and Data Science
Kennesaw State University
yanwang1608@gmail.com

Xuelei Sherry Ni, Ph.D.
Professor of Statistics
Interim Chair, Department of Statistics and Analytical Sciences
Kennesaw State University #1103
(470) 578-2251
sni@kennesaw.edu