Neural Network

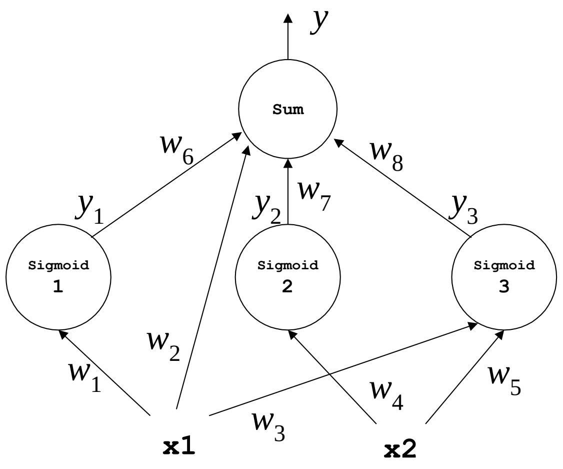
Second Order Newton Algorithm
Multilayer Perceptron

Table of Contents

Requirement

Design and implement the second-order Newton's training algorithm for MLP neural networks. Present the algorithmic structure of the Newton's algorithm and the formulae for updating the network weights. Implement your own version of the algorithm in Matlab using the approximate Hessian matrix with second-order network derivatives with respect to the weights.

Test initially the Newton's algorithm with the simple neural network given below, and after that create another realistic fully connected network model with 10 inputs and 5 hidden nodes suitable for training with the Sunspots series. Compare the performance (in terms of normalized mean squared error) of the Netwon's algorithm using the approximate Hessian on the Sunspots series with: 1) the classical backpropagation algorithm (studied during the lectures); and 2) the Netwon's algorithm using the exact Hessian computed with the R-propagation algorithm (given in the books below). Give the approximation of the Sunspots series with these three algorithms on a common plot to illustrate the differences.



Assume that the initial weights and thresholds are:

w_1 = -0.25	w_2 = 0.33	w_3 = 0.14	w_4 = -0.17	w_5 = 0.16
w_6 = 0.43	w_7 = 0.21	w_8 = -0.25		

Introduction

This experiment presents the design and implementation of the second-order Newton's training algorithm for a Multilayer Perceptron (MLP) neural networks (NN). The first section details the algorithm structure of the Newton's algorithm. The second section details the Hessian approximation proposed by Haykin [2] and discuss its limitations. The third section discusses the exact Hessian computation proposed by Nabney [7]. The last sections respectively discuss the implementation and results generated by i) a partially connected two inputs MLP NN and ii) a fully connected ten inputs MLP NN, using an Hessian approximation to generate the weight update.

Part 1 – Newton's Method

Description

As mentioned by Haykin [2], the Newton's method is an improvement on the steepest descent (i.e. when successive adjustments are applied to the weight vector w in opposite direction to the gradient descent). As show in Figure 1 below, borrowed from Haykin [2], depending on the level of the learning rate, the trajectory could display zig-zags. The Newton's method provides smoother (i.e. more linear) approach to reach approximation. The main idea behind the Newton's method is to minimise the quadratic approximation of the cost function ε(**w**) of a point **w**(n), at each iteration.

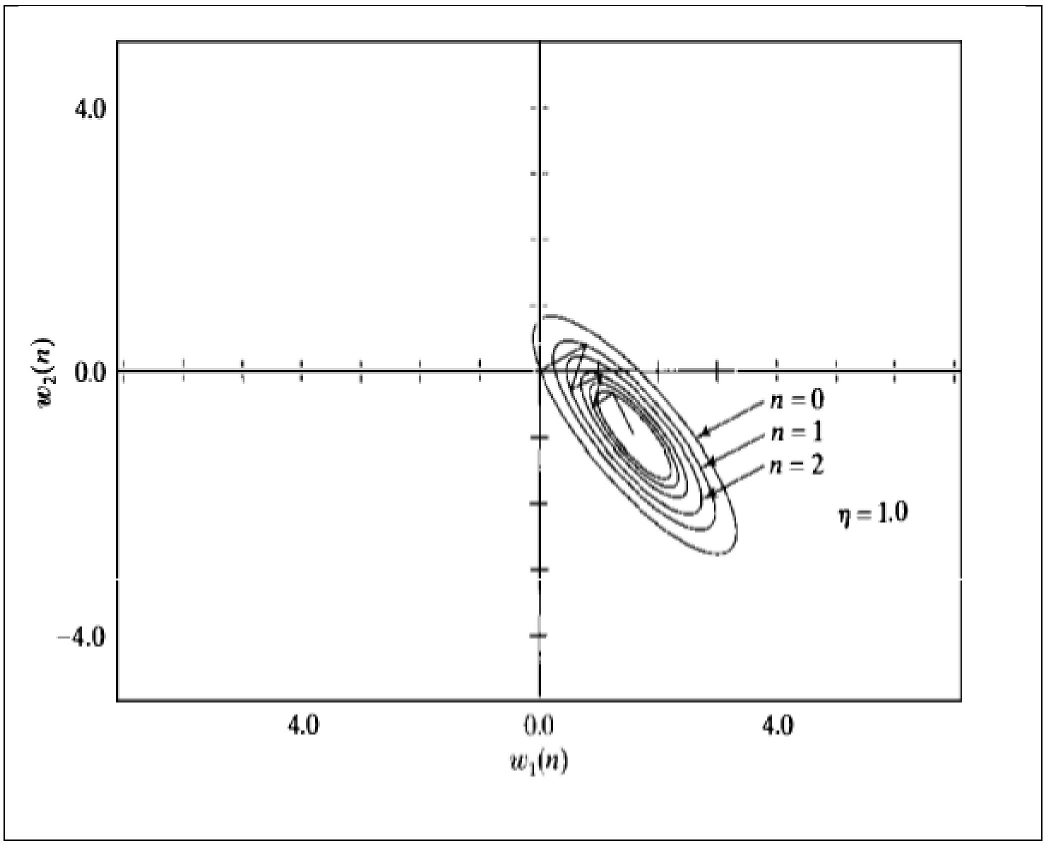


Figure 1 – Trajectory of the steepest descent in a two-dimensional space where the learning rate is set to 1.0. Note: w1 and w2 are coordinates for weight vector **w**.

Haykin [2] defines second-order Taylor series expansion ε(**w**) around **w**(n) as:

$$\Delta\mathscr{E}(\mathbf{w}(n)) = \mathscr{E}(\mathbf{w}(n+1)) - \mathscr{E}(\mathbf{w}(n))$$
$$\simeq \mathbf{g}^T(n)\Delta\mathbf{w}(n) + \frac{1}{2}\Delta\mathbf{w}^T(n)\mathbf{H}(n)\Delta\mathbf{w}(n) \tag{1}$$

where:

- **g**(n) is the m-by-1 gradient vector of ε(**w**), in the vicinity of the point **w**(n).
- **H**(n) is the m-by-m Hessian matrix of ε(**w**), in the vicinity of the point **w**(n).

Haykin [2] also defines **H**(n) as:

$$\mathbf{H} = \nabla^2\mathscr{E}(\mathbf{w})$$
$$= \begin{bmatrix} \frac{\partial^2\mathscr{E}}{\partial w_1^2} & \frac{\partial^2\mathscr{E}}{\partial w_1\partial w_2} & \cdots & \frac{\partial^2\mathscr{E}}{\partial w_1\partial w_m} \\ \frac{\partial^2\mathscr{E}}{\partial w_2\partial w_1} & \frac{\partial^2\mathscr{E}}{\partial w_2^2} & \cdots & \frac{\partial^2\mathscr{E}}{\partial w_2\partial w_m} \\ \vdots & \vdots & & \vdots \\ \frac{\partial^2\mathscr{E}}{\partial w_m\partial w_1} & \frac{\partial^2\mathscr{E}}{\partial w_m\partial w_2} & \cdots & \frac{\partial^2\mathscr{E}}{\partial w_m^2} \end{bmatrix} \tag{2}$$

As the Hessian matrix requires the cost function to be partially derived twice in respect of **w**, it is necessary for ε(**w**) to be twice continuously differentiable, in respect of **w**.

Haykin [2] indicates that minimising equation (1) means taking the first derivative of (1) in respect to Δ**w** and setting it to 0**,** as shown in the following equation:

$$\mathbf{g}(n) + \mathbf{H}(n)\Delta\mathbf{w}(n) = \mathbf{0} \tag{3}$$

This can be presented as (4)

$$\Delta\mathbf{w}(n) = -\mathbf{H}^{-1}(n)\mathbf{g}(n) \tag{4}$$

The weight updates is therefore:

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \Delta\mathbf{w}(n)$$
$$= \mathbf{w}(n) - \mathbf{H}^{-1}(n)\mathbf{g}(n) \tag{5}$$

The Newton's method converges quickly asymptotically and does provide more 'straight line' behaviour for getting to the approximation **w**(n).

Limitations

The Hessian matrix needs to be a positive definite matrix for all n, for the Newton's method to work. However, in practice this is not a guarantee, at each iteration of the algorithm.

Part 2 – Computing the inverse of a Hessian Matrix (the approximation method)

Description

As mentioned by Gill and King [1], the Hessian matrix must be positive and definite and hence invertible in order to compute the variance matrix. Haykin [2] proposes a manageable approach for computing the inverse of a Hessian (**H**$^{-1}$).
In the case of a single output multilayer perceptron (MLP), for a training set, Haykin [2] shows that the cost function ε(**w**), can be expressed as:

$$\mathcal{E}_{\text{av}}(\mathbf{w}) = \frac{1}{2N}\sum_{n=1}^{N}(d(n) - o(n))^2 \qquad (6)$$

Where:
o(n) is the actual output of the network.
d(n) is the desired response (i.e. the expected value)
N the total number of example in the training set

O(n) can be expressed as follows:

$$o(n) = F(\mathbf{w}, \mathbf{x})$$

Where:
F is the input-output mapping function represented by the MLP
x is the input vector
w the synaptic weight vector

Therefore, Haykin [2] demonstrates the second derivatives of the Hessian is given by:

$$\begin{aligned}\mathbf{H}(N) &= \frac{\partial^2\mathcal{E}_{\text{av}}}{\partial\mathbf{w}^2}\\ &= \frac{1}{N}\sum_{n=1}^{N}\left\{\left(\frac{\partial F(\mathbf{w},\mathbf{x}(n))}{\partial\mathbf{w}}\right)\left(\frac{\partial F(\mathbf{w},\mathbf{x}(n))}{\partial\mathbf{w}}\right)^T\right.\\ &\quad \left. - \frac{\partial^2 F(\mathbf{w},\mathbf{x}(n))}{\partial\mathbf{w}^2}(d(n) - o(n))\right\} \end{aligned} \qquad (7)$$

Under the assumption is the training set is large enough, the second term of this function can be ignored. Therefore, the Hessian becomes:

$$\mathbf{H}(N) \simeq \frac{1}{N}\sum_{n=1}^{N}\left(\frac{\partial F(\mathbf{w},\mathbf{x}(n))}{\partial\mathbf{w}}\right)\left(\frac{\partial F(\mathbf{w},\mathbf{x}(n))}{\partial\mathbf{w}}\right)^T \qquad (7\text{bis})$$

There are two ways to resolve this equation, i) using a recursion (the Woodbury's equality) which is not in scope for this implementation and ii) using an approximation defined by Yu and Wilamowski [7] in eq. (8).

$$H \approx J^T J \tag{8}$$

Where:

$$J = \begin{bmatrix} \frac{\partial e_{11}}{\partial w_1} & \frac{\partial e_{11}}{\partial w_2} & \cdots & \frac{\partial e_{11}}{\partial w_{nw}} \\ \frac{\partial e_{12}}{\partial w_1} & \frac{\partial e_{12}}{\partial w_2} & \cdots & \frac{\partial e_{12}}{\partial w_{nw}} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial e_{1no}}{\partial w_1} & \frac{\partial e_{1no}}{\partial w_2} & \cdots & \frac{\partial e_{1no}}{\partial w_{nw}} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial e_{np1}}{\partial w_1} & \frac{\partial e_{np1}}{\partial w_2} & \cdots & \frac{\partial e_{np1}}{\partial w_{nw}} \\ \frac{\partial e_{np2}}{\partial w_1} & \frac{\partial e_{np2}}{\partial w_2} & \cdots & \frac{\partial e_{np2}}{\partial w_{nw}} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial e_{npno}}{\partial w_1} & \frac{\partial e_{npno}}{\partial w_2} & \cdots & \frac{\partial e_{npno}}{\partial w_{nw}} \end{bmatrix}$$

And:
e is the error
w is the weight

Gradient Descent Derivation for the Jacobian and the Hessian

This section demonstrates the derivation of the components present in the J matrix for a fully connected MLP NN, as show in Figure 2 borrowed from Nikolaev [5] and modified to support linear outputs. The derivation of the error by the weights are given in eq. (13) and eq. (20). Please refer to the hand-written pages provided in the complementary document.

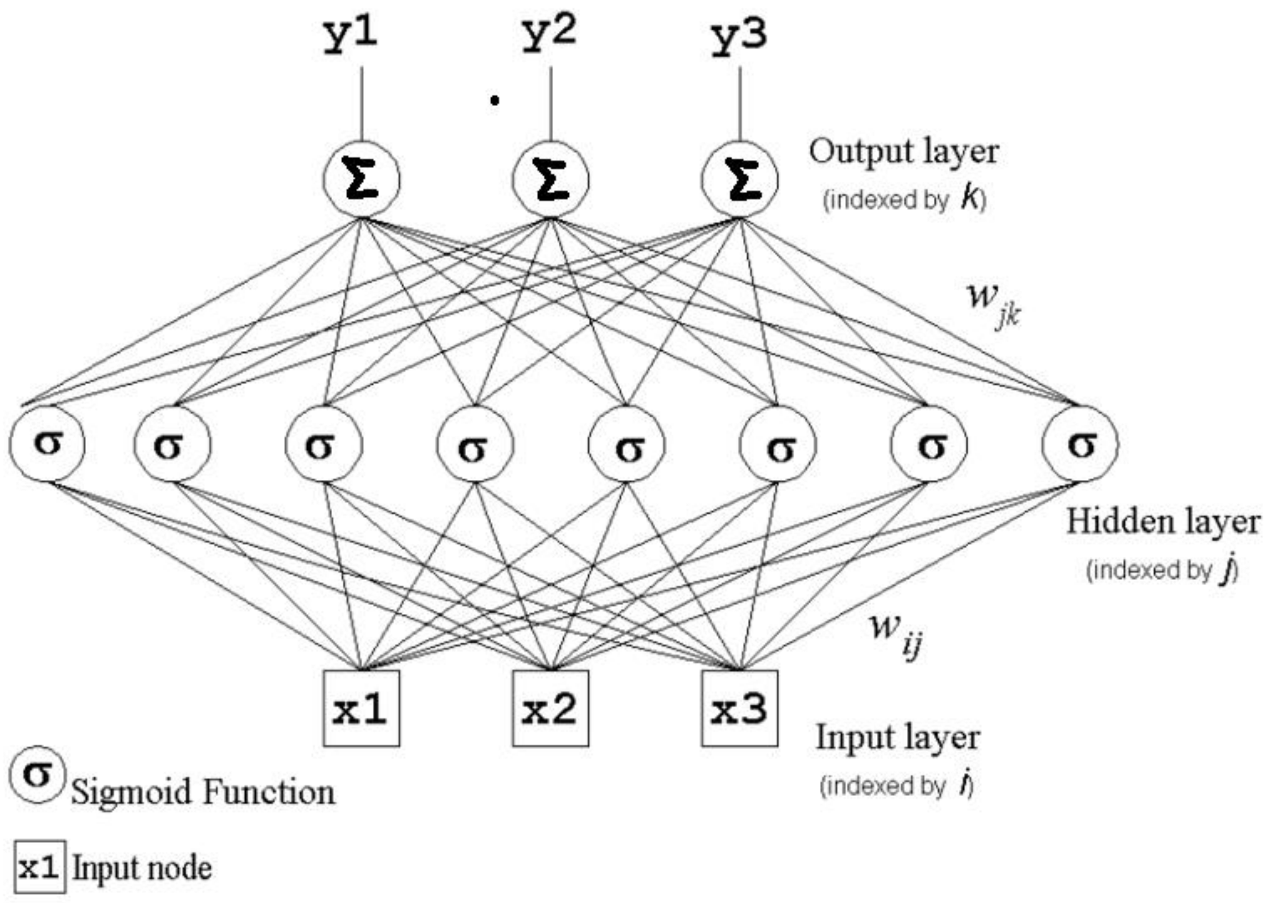


Figure 2. Multilayer Perceptron (MLP)

From (13) and eq. (20), the Jacobian matrix J can be computed as per eq. (8).
The deltaWeight can then be calculated following the formula proposed by Yu and Wilamowski [7]:

$$\text{deltaWeight} = (J^t\ J\ + \mu I)^{-1} J^t e \qquad (21)$$

Where:
$(J^t\ J\ + \mu I)^{-1}$ represents inverse of the Hessian, i.e. H^{-1}
J^t is the transpose of the Jacbian matrix
e is the error

Part 3 – The exact Hessian Matrix

The above algorithm computes an approximation of the Hessian. It is faster than the recursive approach but slower than the one proposed by Nabney [3].
$Furthermore, the\ version\ proposed\ by\ Nabney$ [3], $derived\ from\ the\ back-propagation$ computes an exact Hessian. The latter is not implemented in this experiment. This methodology takes root in the R-propagation algorithm (Pearlmutter [6]), which offers two advantages:

- an efficient algorithm to evaluating vt H, where vt is a vector of weight elements
- the equation structure is close to the forward and backward propagation derivatives $v^t H \equiv v^t \nabla (\nabla E)$, where ∇stands for the gradient descent

To calculate v^t H, Nabney [3] applies the standard forward and backward propagation for *grad E* and the author applies the differential operator $v^t \nabla$. Nabney [3] obtains the following R-forward propagation equation for the linear outputs.

$R\{y_k\} = R\{a_k^{(2)}\}$ (22)
Where
y_k represents the output values
w represents the weights

And
$R\{a_k^{(2)}\} = \sum_j wkj\ R\{zj\} + \sum_j vkj\ zj$ (23)
And
$R\{zj\} = g'(a_j^{(1)})\ R\{a_j^{(1)}\}$ (24)
And
$R\{a_j^{(1)}\} = \sum_j vji\ xi$ (25)

Nabney [3] computes the backward propagation equations as follows:

$R\{\delta_k^{(2)}\} = R\{y_k\}$ (26)
$R\{\delta_k^{(1)}\} = g''(a_j^{(1)})\ R\{a_j\}\ \sum_j wkj\ \delta k + g'(a_j^{(1)})\ \sum_j vkj\ \delta k + g'(a_j^{(1)})\ \sum_j wkj\ R\{\delta k\}$

Nabney [3] generates the gradient formulas in respect of each layer weights by:

$$R\left\{\frac{dE}{dwkj}\right\} = R\{\delta k\}zj + \delta k\ R\{zj\} \qquad (27)$$

$$R\left\{\frac{dE}{dwji}\right\} = xiR\{\delta_j^{(1)}\} \qquad (28)$$

The eq. (27) and (28) corresponds to the v^t H, the Exact Hessian.

Part 3 – Implementation of a partially connected two inputs MLP

The aim is to implement in MATLAB the below MLP NN using the Hessian approximation algorithm as defiled in Part 2. This MLP consists of partially connected layers as well as a perceptron that connects the input x1 to the output y.

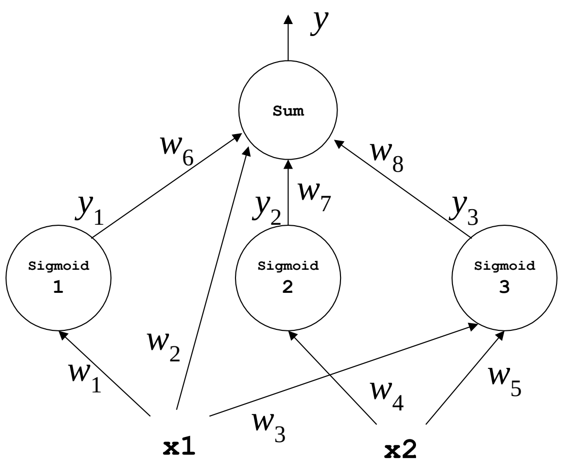


Figure 3: a partially connected two inputs MLP

Number of Epochs	100
Number of Hidden Layers	1
Number of Hidden Nodes	3

The initial weights are assumed to be:
w_1 = -0.25 w_2 = 0.33 w_3 = 0.14 w_4 = -0.17 w_5 = 0.16
w_6 = 0.43 w_7 = 0.21 w_8 = -0.25

The input data consist of two rows of binary data and the corresponding label

Input1	**Input2**	**Label**
1	0	1
1	1	1

As shown in code snippet 1, the gradient (G1 and G2) as well as the hessian are updated incrementally, whereas the weights are updated as a batch.

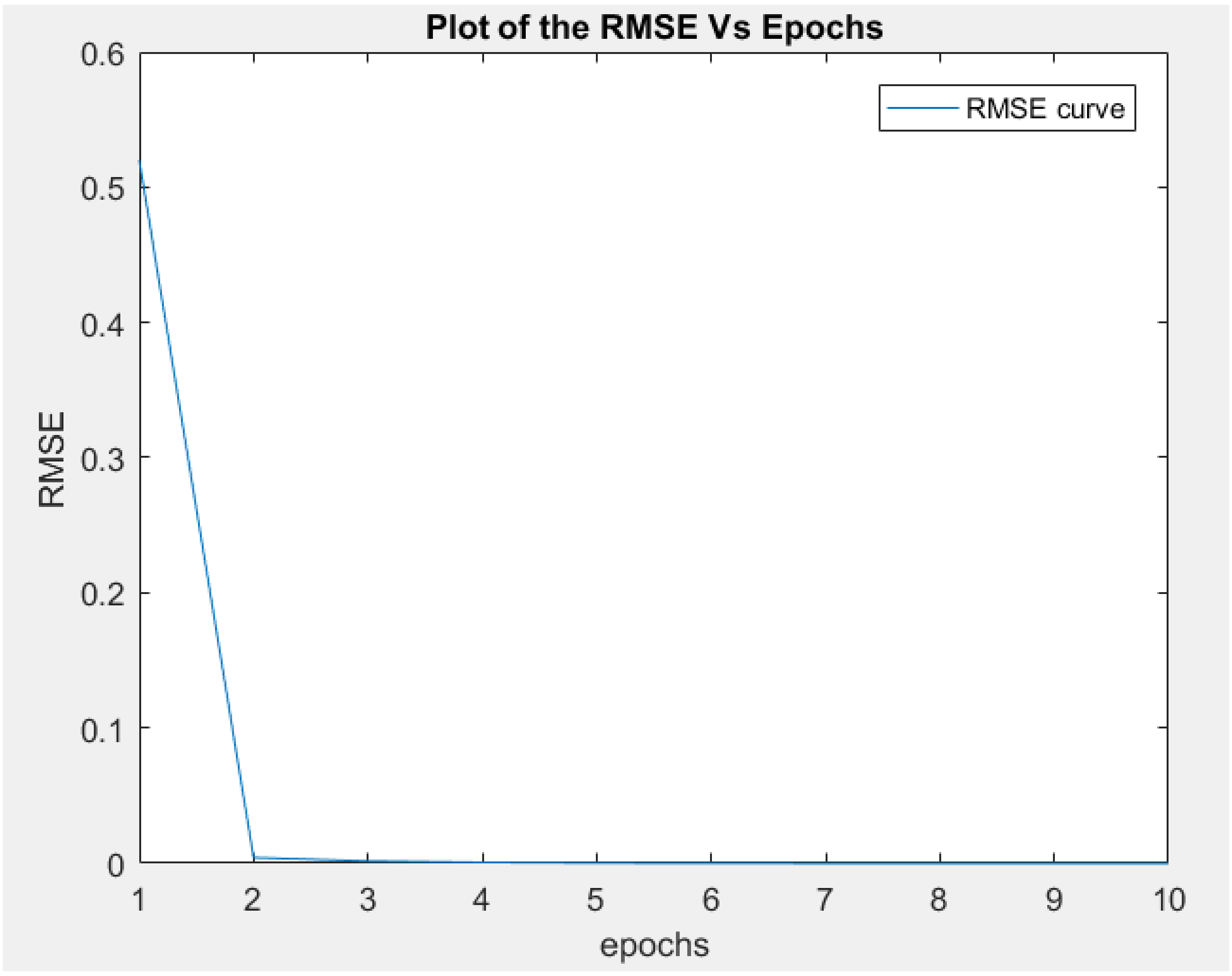
```
clearvars variables
%****************************************************************************
%****************Load the data (1=sun spot time series) *******************
%****************************************************************************
load TrainingData_Assign1.dat;
load TrainingLabels_Assign1.dat;
Patterns = TrainingData_Assign1';
Desired = TrainingLabels_Assign1';

%****************************************************************************
%*************************** SET UP *******************************************
%****************************************************************************
NEPOCH = 300; % The initial number of epoch
NHIDDEN = 3; %Number of hidden nodes
weights = [-0.25 0.33 0.14 -0.17 0.16 0.43 0.21 -0.25];
mseList = []; epochList =[]; TSS_Limit = 0.02;
[NINPUTS,NPATS] = size(Patterns);
[NOUTPUTS,NPATS] = size(Desired);
Inputs1 = [Patterns];
Inputs1Trans = Inputs1';
Out_list = zeros(1,2);

%****************************************************************************
%*************************** TRAIN  *******************************************
%****************************************************************************
for epoch = [1:NEPOCH]
   fprintf('********** Epoch %1d **********\n',epoch);
   epochList(epoch) = epoch;
   TSS = 0;
   deltaW1 = 0; deltaW2 = 0; deltaW1Perceptron = 0;
   G1 =0; G2 =0; G1Perceptron = 0;
   jacobian =0; currentHess =0;

   Weights1 = [weights(1) 0  weights(3) ; 0  weights(4)  weights(5)];
   Weights1_Perceptron = [weights(2)];
   Weights2 = [weights(6) weights(7) weights(8)];

   fprintf('********** Weights **********\n');
   fprintf('Weights1:\n');
   disp(Weights1)
   fprintf('Weights1_Perceptron:\n');
   disp(Weights1_Perceptron)
   fprintf('Weights2:\n');
   disp(Weights2)

   %For each example
   for ex=1:NPATS
     fprintf('***Example %1d\n',ex);

     %Hidden Forward Propagation
     NetIn1 = Weights1' * Inputs1(:,ex);
     Hidden = 1.0 ./( 1.0 + exp( -NetIn1 ));
     %Output Forward Propagation
     Inputs2 = Hidden;
     Out_Hidden = Weights2 * Inputs2
     fprintf('Out_Hidden = %0.4f\n',Out_Hidden);
     %Perceptron Propagation
     Out_Perceptron =  Weights1_Perceptron * Inputs1(1,ex);
     fprintf('Out_Perceptron = %0.4f\n',Out_Perceptron);
     %Output
     Out = Out_Hidden + Out_Perceptron;
     Out_list(ex) = Out;
     fprintf('Out = %0.4f\n',Out);
```

```
  %Errors/Beta Backward Propagation
  Error = Desired(ex) - Out;
  Beta = 1.0;
  fprintf('Beta = %0.4f\n',Beta);
  fprintf('Error = %0.4f\n',Error);
  %HiddenBeta and Deriv Backward Propagation
  bperr = Weights2' * Beta;
  HiddenBetaDeriv = Hidden .* (ones(NHIDDEN,1) - Hidden);
  HiddenBeta = HiddenBetaDeriv .* bperr;
  PrintHiddenBetas(HiddenBeta)
  %Hidden ->  Ouput  dW and gdW Backward Propagation
  dW2h = Beta * Inputs2' ;
  grad2 = Error * dW2h;
  %Input ->  Hidden  dW and gdW Backward Propagation
  dW1h = 0;grad1 =0; idx =1;
  for idx_i= 1:NINPUTS
   for idx_h=1:NHIDDEN
       if (Weights1(idx_i,idx_h) ~= 0)
           factor = 1;
       else
           factor = 0 ;
       end
       dW1h(idx) = HiddenBeta(idx_h) * Inputs1(idx_i,ex) * factor;
       grad1(idx) =  Error * dW1h(idx);
       idx = idx +1;
    end
  end
  %Perceptron ->  Ouput  dW and gdW Backward Propagation
  dW1hPerceptron = Beta * Inputs1Trans(ex,1);
  grad1Perceptron =  Error * dW1hPerceptron;

  %Gradient Descent
  G2 = grad2 + G2;
  G1 = grad1 + G1;
  G1Perceptron = grad1Perceptron + G1Perceptron;

  %TSS errors:
  TSS = TSS + sum( Error^2 );
  fprintf('TSS = %0.4f\n', TSS);

  jacobian = [dW1h(1) dW1hPerceptron dW1h(3) dW1h(5) dW1h(6) dW2h ];
  currentHess = currentHess + ((jacobian') * (jacobian));
end

G = [G1(1) G1Perceptron G1(3) G1(5) G1(6) G2]/ NPATS;
hess = currentHess / NPATS;
hess = hess + (eye ([ size(hess) ]) * 0.001);
newtonDeltaWeights = inv( hess ) * G';

weights = weights + newtonDeltaWeights';

%RMSE
RMSE =0;
if (TSS ~= 0)
   RMSE = sqrt(TSS/NPATS);
end
mseList(epoch) = RMSE;
fprintf('---> RMSE = %0.4f\n',RMSE);

%Stop when convergence is achieved
%(i.e delta weights are close to 0)
%if TSS < TSS_Limit, break, end
```

```
    if (newtonDeltaWeights' < repmat(0.0001,1,size(weights,2)))
        fprintf('---> newtonDeltaWeights = %0.4f\n',newtonDeltaWeights);
        break
    end
end

%Plot Learning Rate
PlotRmse(epochList,mseList);
```

Code Snipppet 1 – Hessian Approximation implementation

The results is provided below:



Part 4 – Implementation of a fully connected ten inputs MLP

Set-up

The aim is to implement in MATLAB the below MLP NN using the Hessian approximation algorithm as defiled in Part 2. This MLP consists of a 10 inputs fully connected layers MLP (c.f. Figure 4). The aim is to predict the sunspot level, one step ahead. In this case, one step represents one year. The number of inputs and the number of nodes are based on relevant research, Haykin [2], because this is a well-known time series.

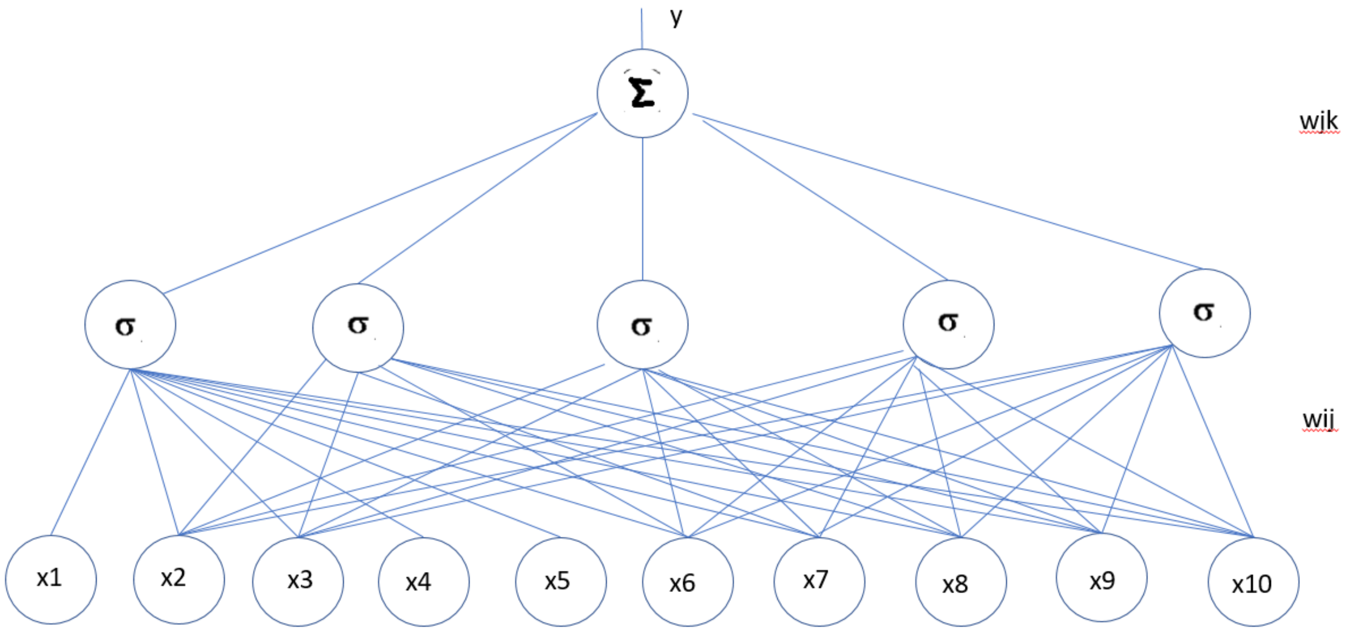


Figure 4: a fully connected ten inputs MLP

Number of time windows	5
Number of Epochs	100
Training data set length	200
Test data set length	78
Number of Hidden Layers	1
Number of Hidden Nodes	5

The weights of the i-> j layer (wij) are defined as below

```
[ 0.0537   -0.0749   -0.1218   -0.1176   -0.0103    0.1105   -0.1952    0.1318   -0.1540    0.0152;
  -0.0249    0.0810    0.0567   -0.0910    0.0697    0.0112   -0.2182    0.0639   -0.1806    0.1806;
  -0.0206   -0.0419    0.0411   -0.1904    0.0224    0.2469   -0.0477    0.1360    0.0981   -0.0076;
  0.0810    0.1710    0.0204    0.2199    0.0737   -0.1407   -0.0258    0.2164   -0.2031   -0.0533;
  0.1351    0.1665    0.1850    0.0728    0.0219   -0.1971   -0.0671    0.2364    0.0127    0.0857]
```

The weights for the j-> k layer (wjk) are defined as below

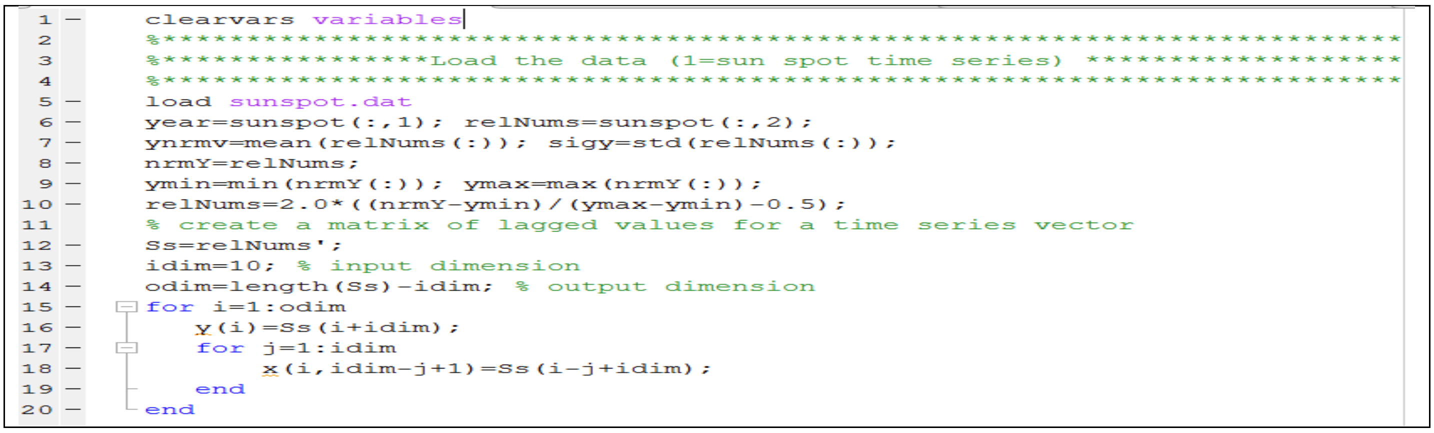
```
 [0.0414    0.1577    0.1895    0.2445   -0.2497]
```

The data relates to sunspots exposure from 1700 to 1987, a sample is provided below:

<u>Year</u>	**<u>sunspot</u>**
1700	5.0
1701	11.0
1702	16.0
1703	23.0
1704	36.0

Training/Testing framework

The Hessian and MLP NNs have been trained and testing against the sunspot data, in the following fashion. First, the sunspot data is loaded is normalised using the min-max scaler (line 10) to produce values between -1.0 and 1.0. The original data, containing 288x1 sunspot entries. The matrix represents 10 years of sunspots, one year per input node. Only the first 278 are used for training. The last 10 rows are used for testing (line 16). From the 278x1 training matrix, a secondary 278x10 matrix is build. Each column contained the data of the previous column shifted left by one year (line 18).



```
1  -    clearvars variables
2       %*********************************************************************
3       %***************Load the data (1=sun spot time series) ****************
4       %*********************************************************************
5  -    load sunspot.dat
6  -    year=sunspot(:,1); relNums=sunspot(:,2);
7  -    ynrmv=mean(relNums(:)); sigy=std(relNums(:));
8  -    nrmY=relNums;
9  -    ymin=min(nrmY(:)); ymax=max(nrmY(:));
10 -    relNums=2.0*((nrmY-ymin)/(ymax-ymin)-0.5);
11      % create a matrix of lagged values for a time series vector
12 -    Ss=relNums';
13 -    idim=10; % input dimension
14 -    odim=length(Ss)-idim; % output dimension
15 -    for i=1:odim
16 -        y(i)=Ss(i+idim);
17 -        for j=1:idim
18 -            x(i,idim-j+1)=Ss(i-j+idim);
19 -        end
20 -    end
```

Code Snippet 1 – The data normalisation algorithm

The time slide window is the preferred approach to perform training (Figure 2). The training data is divided in two sets: a training set (containing 200 rows) and a validation set (containing 78 rows). The algorithm is run for a number of epochs (e.g. 100), and for each sample in the training data. This means the weights get updated for each sample and each epoch. At the end of the epoch, the forward propagation section of the algorithm is called with the updated weighted for this epoch. The next day predicted sunspot is generated from the updated weights obtained at the end of the last epoch and the next day input (from the validation set). The predicted next day sunspot value is then compare with the expected value. These steps are repeat for the number of time slice (e.g. 5). The updated weights at the end of a time slice are passed as input in the next time slice. The Root Mean Square Error (RMSE) are collected for both the training and test phase for each time slice.

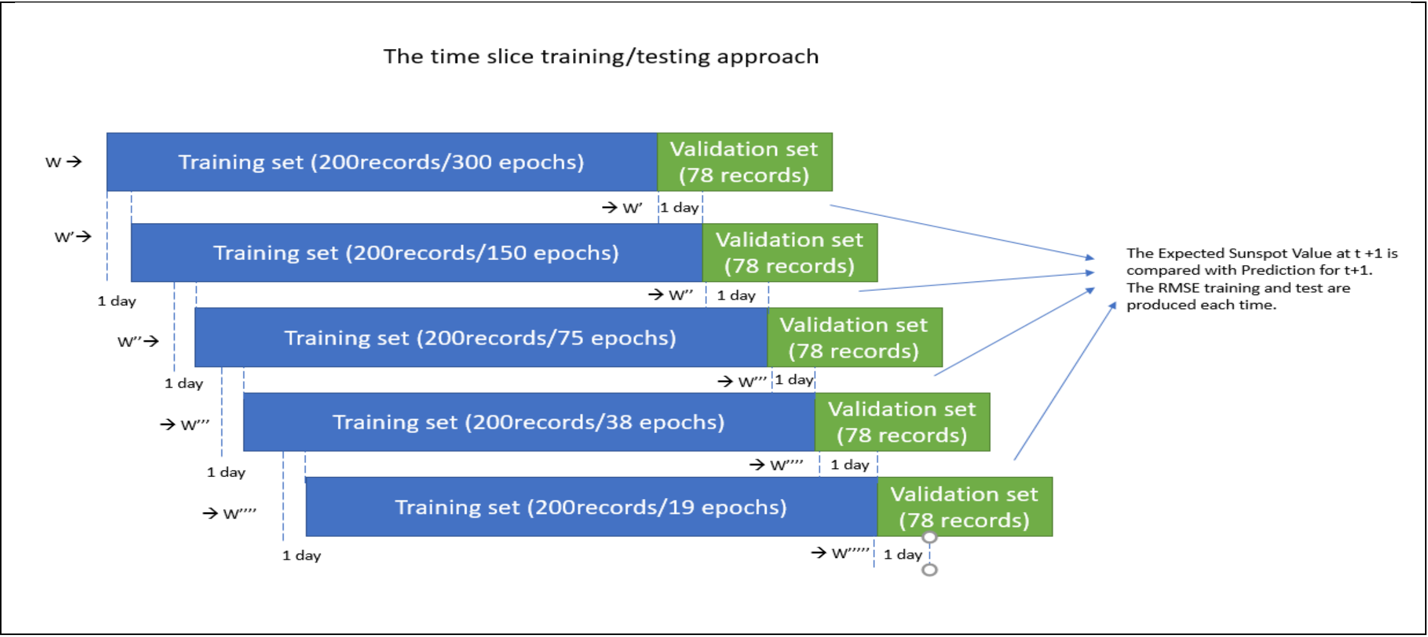
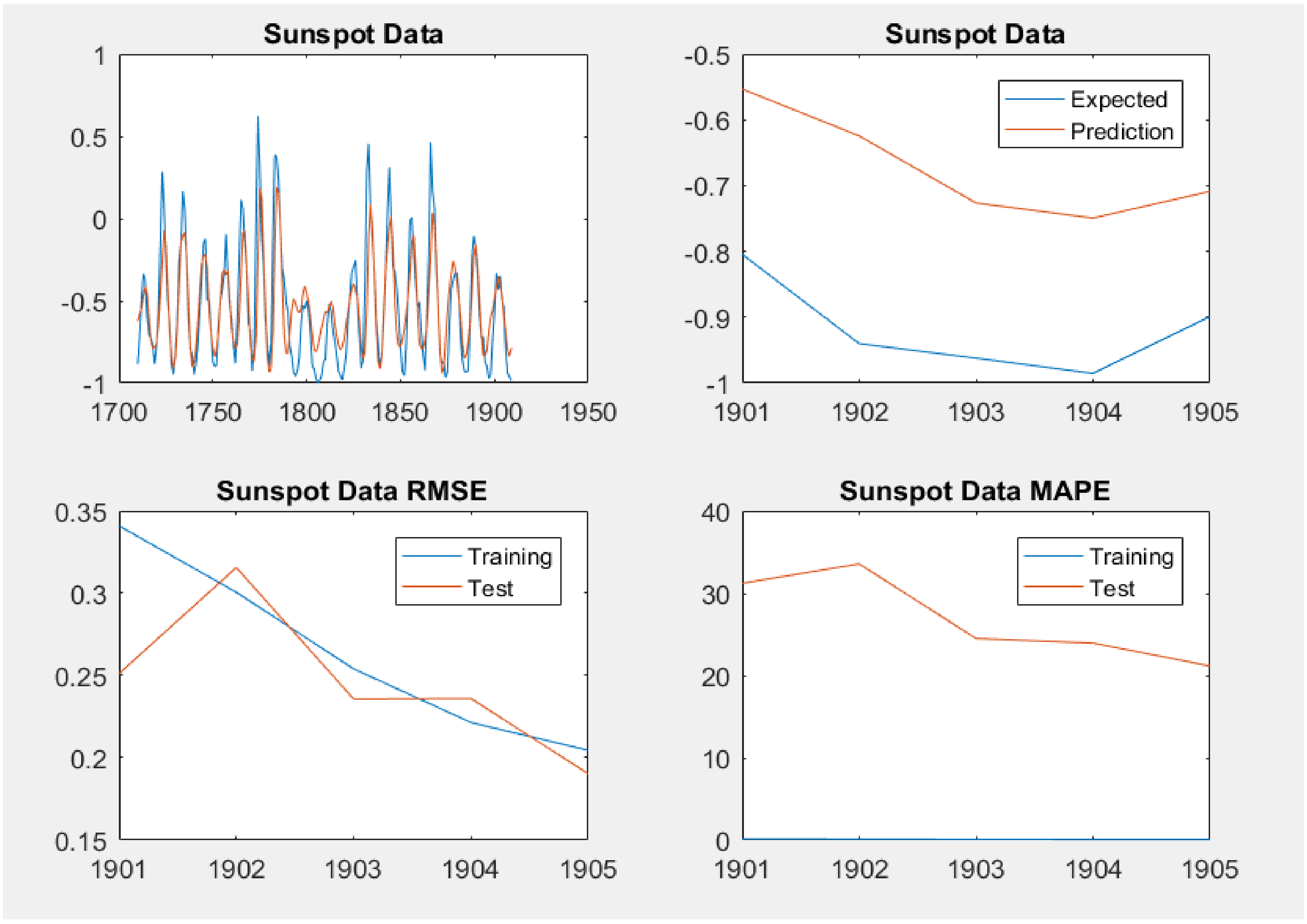

The time slice training/testing approach
w →
Training set (200records/300 epochs)
Validation set (78 records)
→ w'
1 day
w' →
Training set (200records/150 epochs)
Validation set (78 records)
1 day
→ w''
1 day
w'' →
Training set (200records/75 epochs)
Validation set (78 records)
1 day
→ w'''
1 day
→ w''
Training set (200records/38 epochs)
Validation set (78 records)
1 day
→ w''''
1 day
→ w''''
Training set (200records/19 epochs)
Validation set (78 records)
1 day
→ w'''''
1 day
The Expected Sunspot Value at t +1 is compared with Prediction for t+1. The RMSE training and test are produced each time.

Figure 2 – The Training/Testing time slice methodology

Training and Testing MLP with the Classical Backpropagation Algorithm

The below section shows the training and testing results as well as the MATLAB implementation.

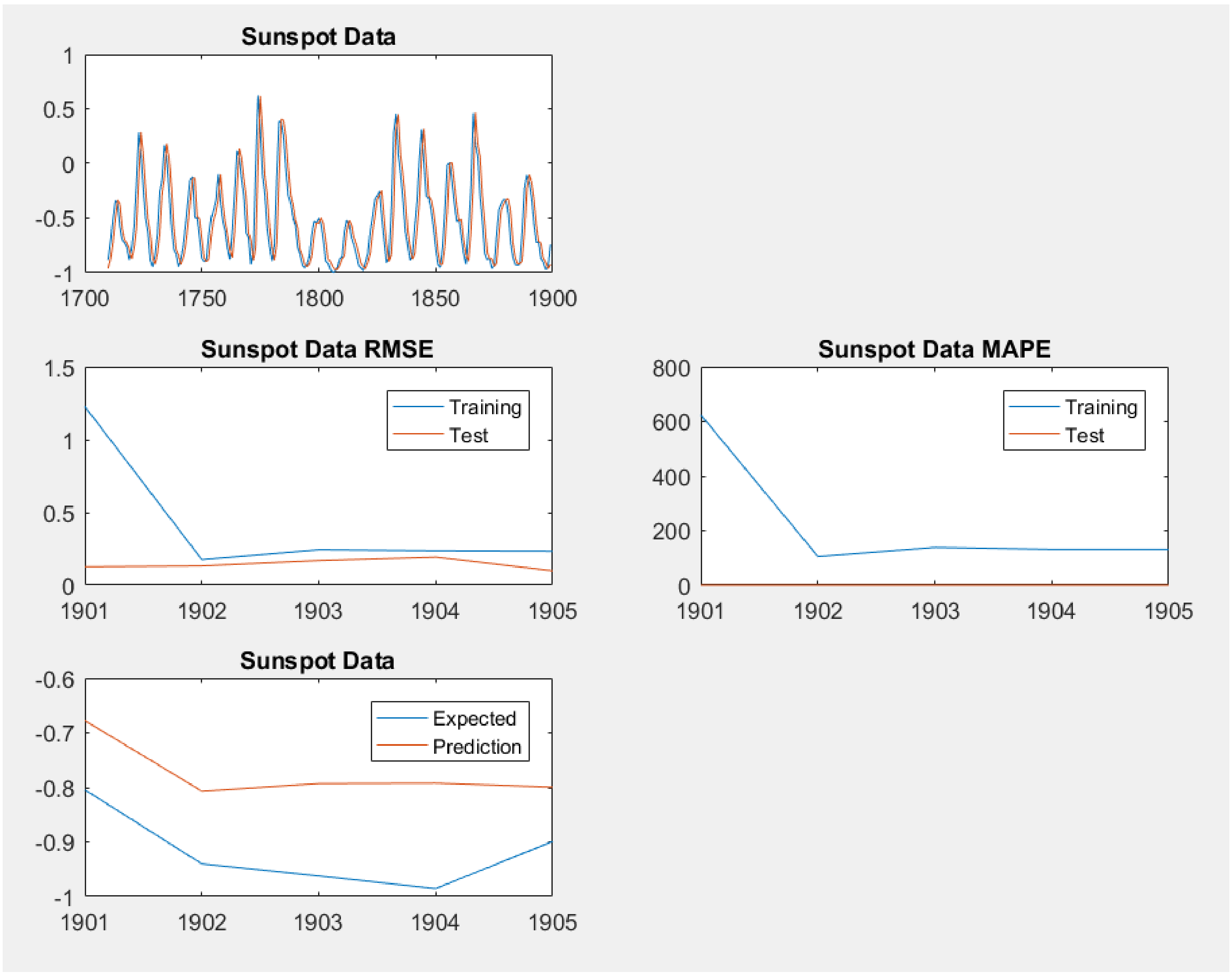


```
% by Dave Touretzky (modified by Nikolay Nikolaev/Frederic Marechal)
% https://www.cs.cmu.edu/afs/cs/academic/class/15782-f06/matlab/

%*************************************************************************
%***************Load the data (1=sun spot time series) *******************
%*************************************************************************
load sunspot.dat
year=sunspot(:,1); relNums=sunspot(:,2); %plot(year,relNums)
ynrmv=mean(relNums(:)); sigy=std(relNums(:));
nrmY=relNums; %nrmY=(relNums(:)-ynrmv)./sigy;
ymin=min(nrmY(:)); ymax=max(nrmY(:));
relNums=2.0*((nrmY-ymin)/(ymax-ymin)-0.5);
% create a matrix of lagged values for a time series vector
Ss=relNums';
idim=10; % input dimension
odim=length(Ss)-idim; % output dimension
for i=1:odim
   y(i)=Ss(i+idim);
   for j=1:idim
       x(i,idim-j+1)=Ss(i-j+idim);
   end
end
```

```
%***************************************************************************
%**************************** SET UP ***************************************
%***************************************************************************
NHIDDENS = 5; %The number of hidden layers
NSTW = 5; % The number of sliding time windows
NEPOCH = 100; % The initial number of epoch
NEPOCH_RATE = 0.5; %The rate of increase/decrease of epoch after each simulation
MAX_TRAIN_INDEX = 200; %The training last index
LearnRate = 0.001; Momentum = 0; DerivIncr = 0; deltaW1 = 0; deltaW2 = 0;
Weights1 =  [ 0.0537   -0.0749   -0.1218   -0.1176   -0.0103    0.1105   -0.1952    0.1318   -
0.1540    0.0152;
             -0.0249    0.0810    0.0567   -0.0910    0.0697    0.0112   -0.2182    0.0639   -
0.1806    0.1806;
             -0.0206   -0.0419    0.0411   -0.1904    0.0224    0.2469   -0.0477    0.1360
0.0981   -0.0076;
              0.0810    0.1710    0.0204    0.2199    0.0737   -0.1407   -0.0258    0.2164   -
0.2031   -0.0533;
              0.1351    0.1665    0.1850    0.0728    0.0219   -0.1971   -0.0671    0.2364
0.0127    0.0857]
Weights2 = [0.0414    0.1577    0.1895    0.2445   -0.2497]
%Weights1 = 0.5*(rand(NHIDDENS,1+NINPUTS)-0.5);
%Weights2 = 0.5*(rand(1,1+NHIDDENS)-0.5);

rmseList = [ ];
rmseList = [{'Sliding Window'} ,{'RMSE_Training'} , {'RMSE_Test'}]
Patterns = x';
Desired = y;
prnout=Desired;
[NINPUTS,NPATS] = size(Patterns); [NOUTPUTS,NP] = size(Desired);
Inputs1= [Patterns];
slidingWindowList = [];
rmseTrainingList = [];rmseTestList = [];
mapeTrainingList = [];mapeTestList = [];
OutPredList = [];
DesiredOneDayPredList = [];

%***************************************************************************
%**************************** TRAIN & PREDICT ****************************************
%***************************************************************************
for slidWinIndex = [1:NSTW]
    fprintf('***************************\n')
    fprintf('slidWinIndex %3d:  Nb Epochs = %f\n',slidWinIndex,NEPOCH);
    fprintf('***************************\n')
    startIndex = slidWinIndex;
    endIndex = MAX_TRAIN_INDEX -1+ slidWinIndex;
    %Create the sliding window
    Inputs1SldWin = Inputs1(1:NINPUTS, startIndex:endIndex)
    DesiredSldWin = Desired(1:NOUTPUTS,startIndex:endIndex)
    [NINPUTS,NPATS] = size(Inputs1SldWin)
    for epoch = 1:NEPOCH
      % Forward propagation
      NetIn1 = Weights1 * Inputs1SldWin;
      %Hidden=1-2./(exp(2*NetIn1)+1);
      Hidden = 1.0 ./( 1.0 + exp( -NetIn1 ));
      Inputs2 = [Hidden];
      NetIn2 = Weights2 * Inputs2;
      Out = NetIn2;
      prnout=Out;
      % Backward propagation of errors
      Error = DesiredSldWin - Out;
      TSS = sum(sum( Error.^2 ));
      MAPE = sum(sum(abs(Error/DesiredSldWin)));
      Beta = Error;
      bperr = ( Weights2' * Beta );
      %HiddenBeta = (1.0 - Hidden .^2 ) .* bperr(1:end,:);
      HiddenBeta = (Hidden .* (ones(NHIDDENS,NPATS) - Hidden)).* bperr(1:end,:);
      % Calculate the weight updates:
      dW2 = Beta * Inputs2';
      dW1 = HiddenBeta * Inputs1SldWin';
      deltaW2 = LearnRate * dW2 + Momentum * deltaW2;
      deltaW1 = LearnRate * dW1 + Momentum * deltaW1;
      % Update the weights:
      Weights2 = Weights2 + deltaW2;
      Weights1 = Weights1 + deltaW1;
      %fprintf('Epoch %3d:  Error = %f\n',epoch,TSS);
      %if TSS < TSS_Limit, break, end
    end
    RMSETrain = sqrt(TSS/NPATS);
    MAPETrain =  (100/ NPATS) * MAPE;

    slidingWindowList (slidWinIndex) =  year(1) + NPATS + slidWinIndex
    rmseTrainingList(slidWinIndex) = RMSETrain;
    mapeTrainingList(slidWinIndex) = MAPETrain;

    %Create the input vs desired set for a one day prediction
    Inputs1OneDayPred = Inputs1(1:NINPUTS, endIndex+1)
    DesiredOneDayPred = Desired(1:NOUTPUTS,endIndex+1)
    [NINPUTS,NPATS1] = size(Inputs1OneDayPred)
    %Generate the prediction using forward propagation
    NetIn1 = Weights1 * Inputs1OneDayPred;
    Hidden = 1.0 ./( 1.0 + exp( -NetIn1 ));
    %Hidden=1-2./(exp(2*NetIn1)+1);
    Inputs2 = Hidden;
    Out = Weights2 * Inputs2;
    %Generate the RMSE for a one day prediction
    Error = DesiredOneDayPred - Out;
    OutPredList(slidWinIndex) = Out;
    DesiredOneDayPredList(slidWinIndex) = DesiredOneDayPred;
    TSSTest = sum(sum( Error.^2 ));
    RMSETest = sqrt(TSSTest/NPATS1);
    rmseTestList (slidWinIndex) = RMSETest;
    MAPETest =  (100/ NPATS1) * (abs(Error/DesiredOneDayPred));
    mapeTestList(slidWinIndex) = MAPETest;

    %Change the number of epochs for each simulation
    %NEPOCH = NEPOCH * NEPOCH_RATE
    %Print weights after simulation...
    fprintf ('Weights1:\n')
    disp(Weights1)
    fprintf ('Weights2:\n')
    disp(Weights2)
end

%Figure
subplot(2,2,1)
plot(year(11:210),DesiredSldWin,year(11:210),prnout)
title('Sunspot Data')

%Figure
subplot(2,2,2)
plot(slidingWindowList,DesiredOneDayPredList,slidingWindowList,OutPredList)
title('Sunspot Data')
legend('Expected', 'Prediction')

%Figure
subplot(2,2,3)
plot(slidingWindowList, rmseTrainingList, slidingWindowList, rmseTestList)
title('Sunspot Data RMSE')
legend('Training', 'Test')

%Figure
subplot(2,2,4)
plot(slidingWindowList, mapeTrainingList, slidingWindowList, mapeTestList)
title('Sunspot Data MAPE')
legend('Training', 'Test')


%Results in a table
disp(rmseTrainingList)
disp(rmseTestList)
disp(mapeTrainingList)
disp(mapeTestList)
```

Training and Testing MLP with the Newton's Algorithm

The below section shows the training and testing results as well as the MATLAB implementation.



```
clearvars variables

%***************************************************************************
%****************Load the data (1=sun spot time series) *******************
%***************************************************************************
load sunspot.dat
year=sunspot(:,1); relNums=sunspot(:,2);
ynrmv=mean(relNums(:)); sigy=std(relNums(:));
nrmY=relNums;
ymin=min(nrmY(:)); ymax=max(nrmY(:));
relNums=2.0*((nrmY-ymin)/(ymax-ymin)-0.5);
% create a matrix of lagged values for a time series vector
Ss=relNums';
idim=10; % input dimension
odim=length(Ss)-idim; % output dimension
for i=1:odim
   y(i)=Ss(i+idim);
   for j=1:idim
       x(i,idim-j+1)=Ss(i-j+idim);
   end
end


%***************************************************************************
%*************************** SET UP ****************************************
%***************************************************************************
NSTW = 5; % The number of sliding time windows
NEPOCH = 100; % The initial number of epoch
NEPOCH_RATE = 0.5; %The rate of increase/decrease of epoch after each simulation
MAX_TRAIN_INDEX = 200; %The training last index
NHIDDENS = 5; %Number of hidden nodes
Weights1 =  [ 0.0537   -0.0749   -0.1218   -0.1176   -0.0103    0.1105   -0.1952    0.1318   -
0.1540    0.0152;
             -0.0249    0.0810    0.0567   -0.0910    0.0697    0.0112   -0.2182    0.0639   -
0.1806    0.1806;
             -0.0206   -0.0419    0.0411   -0.1904    0.0224    0.2469   -0.0477    0.1360
0.0981   -0.0076;
              0.0810    0.1710    0.0204    0.2199    0.0737   -0.1407   -0.0258    0.2164   -
0.2031   -0.0533;
              0.1351    0.1665    0.1850    0.0728    0.0219   -0.1971   -0.0671    0.2364
0.0127    0.0857]
Weights2 = [0.0414    0.1577    0.1895    0.2445   -0.2497]

slidingWindowList = [];
rmseTrainingList = [];rmseTestList = [];
mapeTrainingList = [];mapeTestList = [];
epochList = linspace(1,NEPOCH,NEPOCH)
Patterns = x';
Desired = y;
prnout=Desired;
Inputs1= [Patterns];
[NINPUTS,NPATS] = size(Patterns); [NOUTPUTS,NP] = size(Desired);

OutList = [];
OutPredList = [];
DesiredOneDayPredList = [];
```

```
%***************************************************************************
%**************************** TRAIN & PREDICT *************************************
%***************************************************************************
for slidWinIndex = [1:NSTW]
    fprintf('***************************\n')
    fprintf('slidWinIndex %3d:  Nb Epochs = %f\n',slidWinIndex,NEPOCH);
    fprintf('***************************\n')
    startIndex = slidWinIndex;
    endIndex = MAX_TRAIN_INDEX -1+ slidWinIndex;
    %Create the sliding window
    Inputs1SldWin = Inputs1(1:NINPUTS, startIndex:endIndex)
    Inputs1SldWinTrans = Inputs1SldWin'
    DesiredSldWin = Desired(1:NOUTPUTS,startIndex:endIndex)
    [NINPUTS,NPATS] = size(Inputs1SldWin)
    TSS = 0;
    MAPE =0;

    for epoch = 1:NEPOCH
        deltaW1 = 0; deltaW2 = 0;
        G1 =0; G2 =0;
        jacobian =0; currentHess =0; H1=0;H2=0;
        fprintf('********** Weights **********\n');
        fprintf('Weights1:\n');
        disp(Weights1)
        fprintf('Weights2:\n');
        disp(Weights2)

        %For each example
        for ex=1:NPATS
          fprintf('***Example %1d\n',ex);
          %Hidden Forward Propagation
          NetIn1 = Weights1 * Inputs1SldWin(:,ex);
          Hidden = 1.0 ./( 1.0 + exp( -NetIn1 ));
          %Output Forward Propagation
          Inputs2 = Hidden;
          Out_Hidden = Weights2 * Inputs2
          fprintf('Out_Hidden = %0.4f\n',Out_Hidden);
          %Output
          Out = Out_Hidden


          fprintf('Out = %0.4f\n',Out);

          %Errors/Beta Backward Propagation
          Error = Desired(ex) - Out;
          Beta = 1.0;
          fprintf('Beta = %0.4f\n',Beta);
          fprintf('Error = %0.4f\n',Error);
          %HiddenBeta and Deriv Backward Propagation
          bperr = Weights2' * Beta;
          HiddenBetaDeriv = Hidden .* (ones(NHIDDENS,1) - Hidden);
          %HiddenBetaDeriv = (ones(NHIDDENS,1) - Hidden .^2 );
          HiddenBeta = HiddenBetaDeriv .* bperr;
          PrintHiddenBetas(HiddenBeta)
          %Hidden ->  Ouput  dW and gdW Backward Propagation
          dW2h = Beta * Inputs2' ;
          grad2 = Error * dW2h;
          %Input ->  Hidden  dW and gdW Backward Propagation
          dW1h = 0;grad1 =0; idx =1;
          for idx_i= 1:NINPUTS
            for idx_h=1:NHIDDENS
                 dW1h(idx) = HiddenBeta(idx_h) * Inputs1SldWin(idx_i,ex);
                 grad1(idx) =  Error * dW1h(idx);
                 idx = idx +1;
              end
          end
```

```
           %Accumulate the jacobian
           H1 = H1 + dW1h;
           H2 = H2 + dW2h;
           %Accunulate the gradient
           G2 = G2 + grad2;
           G1 = G1 + grad1;
           %Accunulate currentHess
           jacobian = [dW1h , dW2h];
           currentHess = currentHess + ((jacobian') * (jacobian));

           %TSSE/MAPE:
           TSS = TSS + sum( Error^2 );
           MAPE = MAPE + (abs(Error/Desired(ex)));

       end %end of NPATS

         G = [G1 G2]/ NPATS;
         hess = currentHess / NPATS;

         hess = hess + (eye ([ size(hess) ]) * 0.001);
         newtonDeltaWeights =  inv( hess ) * G';

         Weights1 = Weights1 + reshape(newtonDeltaWeights(1:NHIDDENS*NINPUTS)', [NHIDDENS,
NINPUTS]);
         Weights2 = Weights2 +
reshape(newtonDeltaWeights((NHIDDENS*NINPUTS+1):length(newtonDeltaWeights))', [1, NHIDDENS]);

         RMSETrain = sqrt(TSS/NPATS);
         MAPETrain =  (100/ NPATS) * MAPE;
     end % of epoch

     slidingWindowList (slidWinIndex) =  year(1) + NPATS + slidWinIndex
     rmseTrainingList(slidWinIndex) = RMSETrain;
     mapeTrainingList(slidWinIndex) = MAPETrain;

     %Create the input vs desired set for a one day prediction
     Inputs1OneDayPred = Inputs1(1:NINPUTS, endIndex+1)
     DesiredOneDayPred = Desired(1:NOUTPUTS,endIndex+1)
     [NINPUTS,NPATS1] = size(Inputs1OneDayPred)
     %Generate the prediction using forward propagation
     NetIn = Weights1 * Inputs1OneDayPred;
     Hidden = 1.0 ./( 1.0 + exp( -NetIn ));
     Inputs2 = Hidden;
     Out = Weights2 * Inputs2;
     %Generate the RMSE for a one day prediction
     Error = DesiredOneDayPred - Out;
     OutPredList(slidWinIndex) = Out;
     DesiredOneDayPredList(slidWinIndex) = DesiredOneDayPred;
     TSSTest = sum(sum( Error.^2 ));
     RMSETest = sqrt(TSSTest/NPATS1);
     rmseTestList (slidWinIndex) = RMSETest;
     MAPETest =  (100/ NPATS) * (abs(Error/DesiredOneDayPred));
     mapeTestList(slidWinIndex) = MAPETest;

     %Change the number of epochs for each simulation
     %NEPOCH = NEPOCH * NEPOCH_RATE
     %Print weights after simulation...
  end
```

```
%Figure
subplot(3,2,1)
%plot(year(11:210),DesiredSldWin)
%title('Sunspot Data Initial')
%subplot(3,2,2)
%plot(year(11:210),OutList, 'g')
%title('Sunspot Data Trained')
%plot(year(11:210),DesiredSldWin,year(11:210),OutList)
plot(year(11:210),DesiredSldWin(1:190),year(11:210),Out_list)
title('Sunspot Data')

%Figure
subplot(3,2,3)
plot(slidingWindowList, rmseTrainingList, slidingWindowList, rmseTestList)
title('Sunspot Data RMSE')
legend('Training', 'Test')

subplot(3,2,4)
plot(slidingWindowList, mapeTrainingList, slidingWindowList, mapeTestList)
title('Sunspot Data MAPE')
legend('Training', 'Test')

%Figure
subplot(3,2,5)
plot(slidingWindowList,DesiredOneDayPredList,slidingWindowList,OutPredList)
title('Sunspot Data')
legend('Expected', 'Prediction')

%Results in a table
disp(rmseTrainingList)
disp(rmseTestList)
disp(mapeTrainingList)
disp(mapeTestList)
```

References

[1] Gill J, King G [Online], Available at: *http://gking.harvard.edu/files/gking/files/numhess.pdf*, [Accessed: 04-April-2017]

[2] Haykin, Simon (1999), *Neural Networks: A Comprehensive Foundation*, Prentice Hall Inc

[3] Nabney I. (2002), *Algorithms for Pattern Recognition*, Springer.

[4] Nikolaev N. (NA),Multilayer Perceptrons [Online], Available at: *http://homepages.gold.ac.uk/nikolaev/311multi.htm* [Accessed: 04-April-2017]

[5] Nikolaev N. (NA),Multilayer Perceptrons (Continuation) [Online], Available at: *http://homepages.gold.ac.uk/nikolaev/311bpr.htm* [Accessed: 04-April-2017]

[6] Pearlmutter B.A (1994), Fast exact multiplication by the Hessian. Neural Computation 6 (1),147-160

[7] Yu H., Wilamowski B.M. (NA) Neural NetworkTraining with Second Order Algorithms [Online], Available at: *https://pdfs.semanticscholar.org/47a2/3bdc9a5335dcb481aec0b20803e5afc82557.pdf* [Accessed: 04-April-2017]