MSc Computational Cognitive Neuroscience

Neural Networks

Goldsmiths ID: 33772192

Coursework 2: Second Order Newton Algorithm Multilayer Perceptron

---

This experiment presents the design and implementation of the second-order Newton's training algorithm for a Multilayer Perceptron (MLP). The first section details the algorithm structure of Newton's algorithm with a) using the approximate Hessian matrix with second-order network derivatives Haykin (2007) and b) exact Hessian with R propagation Nabney (2002).

The last sections discuss the implementation and results generated by i) a partially connected two-input MLP NN and ii) a comparison of fully connected ten inputs MLP NN using a Hessian approximation, backpropagation, and exact Hessian with R propagation.

The last part displays a graph comparing three training algorithms on Sunspot series data backpropagation, Gauss-Newton, and Exact Hessian with R-propagation.

1. Design and implement Newton's algorithm for training MLP neural networks.

A basic code that outlines Newton's algorithm for training an MLP neural network in is file "*NewtonsAlgo.m"*

---

2. Algorithmic structure of Newton's algorithm and the formulae for updating the network weights

2.1 Approximate Hessian

Haykin (2007) explains that Newton's method is an advancement over the steepest descent method, where the weight vector w is adjusted in the opposite direction to the gradient descent. Figure 1 (Haykin, 2007), shows that the trajectory can display zig-zags depending on the learning rate. In contrast, Newton's method provides a more linear and smoother approach towards approximation. The fundamental concept behind Newton's method is to minimise the quadratic approximation of the cost function ε(w) at each iteration for a given point w(n).

The basic algorithmic structure of Newton's algorithm is as follows:

1. Initialise the weights of the network randomly
2. Repeat until convergence or a maximum number of iterations is reached:
    a. Compute the output of the network for the input data.
    b. Compute the error between the network output and the desired output.
    c. Compute the gradient vector $\nabla J(w)$ and Hessian matrix H of the cost function J(w) with respect to the weights w.
    d. Compute the weight update vector Δw using the formula:

$$\mathbf{\Delta w = -H^{(-1)} * \nabla J(w)}$$

    e. Update the weights of the network using the weight update vector:

$$\mathbf{w\_new = w\_old + \Delta w}$$

    f. Check for convergence criteria, such as a minimum error threshold or a maximum number of iterations.

The second-order Taylor series expansion ε(w) around w(n) as (Haykin, 2007, p. 144-46):

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

$$+ \text{(third- and higher-order terms)}$$

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:

$$\mathbf{H} = \nabla^2\mathscr{E}(\mathbf{w})$$

$$= \begin{bmatrix} \frac{\partial^2\mathscr{E}}{\partial w_1^2} & \frac{\partial^2\mathscr{E}}{\partial w_1 \partial w_2} & \cdots & \frac{\partial^2\mathscr{E}}{\partial w_1 \partial w_m} \\ \frac{\partial^2\mathscr{E}}{\partial w_2 \partial w_1} & \frac{\partial^2\mathscr{E}}{\partial w_2^2} & \cdots & \frac{\partial^2\mathscr{E}}{\partial w_2 \partial w_m} \\ \vdots & \vdots & & \vdots \\ \frac{\partial^2\mathscr{E}}{\partial w_m \partial w_1} & \frac{\partial^2\mathscr{E}}{\partial w_m \partial w_2} & \cdots & \frac{\partial^2\mathscr{E}}{\partial w_m^2} \end{bmatrix}$$

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 Δw and setting it to 0, as shown in the following equation (Haykin, 2007, p. 256-57):

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

Solving this equation for $\Delta\mathbf{w}(n)$ yields

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

That is,

$$\begin{aligned}\mathbf{w}(n+1) &= \mathbf{w}(n) + \Delta\mathbf{w}(n) \\ &= \mathbf{w}(n) - \mathbf{H}^{-1}(n)\mathbf{g}(n)\end{aligned}$$

where $\mathbf{H}^{-1}(n)$ is the inverse of the Hessian of $\mathscr{E}(\mathbf{w})$.

Equation 4.121 of Haykin (2007), which is

Δw(n) = -H^(-1) * g(n)

is the essence of Newton's method, according to the author.

Haykin (2007) also mentions that Newton's method doesn't show the zigzag behaviour which characterises the steepest decent method.

To implement Newton's algorithm, we will first need to calculate the Jacobian matrix and then compute the gradient and Hessian matrix of the cost function with respect to the weights.

Gradient Descent with Jacobian for Hessian

Gradient Descent is an iterative optimization algorithm used to find the minimum of a function. It works by iteratively updating the parameters of the function in the direction of the negative gradient of the function.

For a neural network, the gradient of the error function with respect to the weights can be calculated using the backpropagation algorithm. The Jacobian matrix is the matrix of first-order partial derivatives of the error function with respect to the weights. The Hessian matrix is the matrix of second-order partial derivatives of the error function with respect to the weights.

To derive the update rule for Gradient Descent with the Jacobian matrix, we can use the following formula:

From (Nabney, 2002, p. 209):

$$J_{ki} = \left.\frac{\partial y_k}{\partial x_i}\right|_{\boldsymbol{x}}$$

By applying the chain rule to the above equation, we get:

$$J_{ki} = \frac{\partial y_k}{\partial x_i} = \sum_{j=1}^{M} \frac{\partial y_k}{\partial \phi_j} \frac{\partial \phi_j}{\partial x_i}$$

$$= \sum_{j=1}^{M} w_{kj} \frac{\partial \phi_j}{\partial x_i},$$

For a function f:

$$\nabla f = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix} \quad \text{and} \quad \nabla^2 f = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{pmatrix}$$

The above equation is a representation of how Jacobian is a vector of a function's first partial derivatives and Hessian is the matrix of f's second partial derivatives.

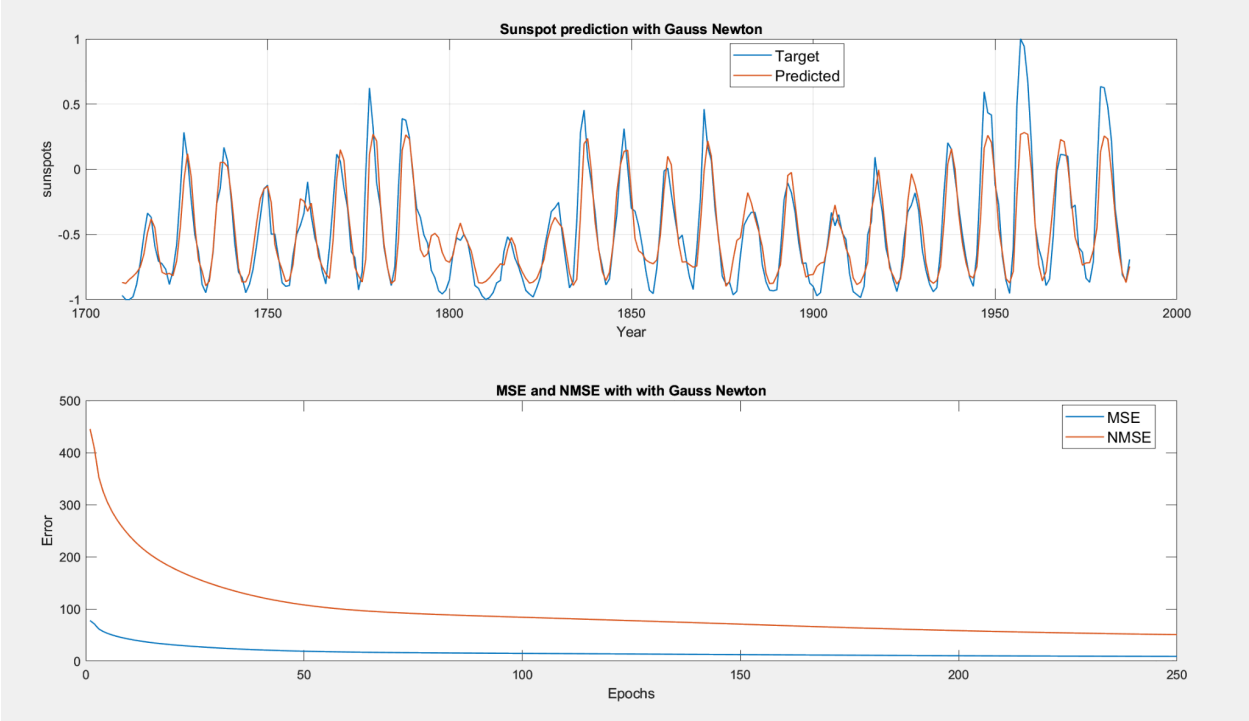


***Figure**: MSE and NMSE with Gauss-Newton*

2.2 Exact Hessian with R Propagation

The approximate Hessian Newton algorithm computes an inverse of the approximation of the Hessian. It is faster than the recursive approach but slower than the one proposed by Nabney (2002). This will be tested in the last section of the assignment.

Furthermore, the version proposed by Nabney (2002) is derived from backpropagation and computes exact Hessian rather than approximate Hessian.

Also called the Fast multiplication by Hessian, Nabney (2002, p. 160-63) shows:

$$\boldsymbol{v}^{\boldsymbol{T}}\boldsymbol{H} \equiv \boldsymbol{v}^{\boldsymbol{T}}\nabla(\nabla E).$$

$$\mathcal{R}\{\boldsymbol{w}\} = \boldsymbol{v},$$

$$\mathcal{R}\{a_j^{(1)}\} = \sum_i v_{ji} x_i$$

$$\mathcal{R}\{z_j\} = g'(a_j^{(1)})\, \mathcal{R}\{a_j^{(1)}\}$$

$$\mathcal{R}\{a_k^{(2)}\} = \sum_j w_{kj}\, \mathcal{R}\{z_j\} + \sum_j v_{kj} z_j,$$

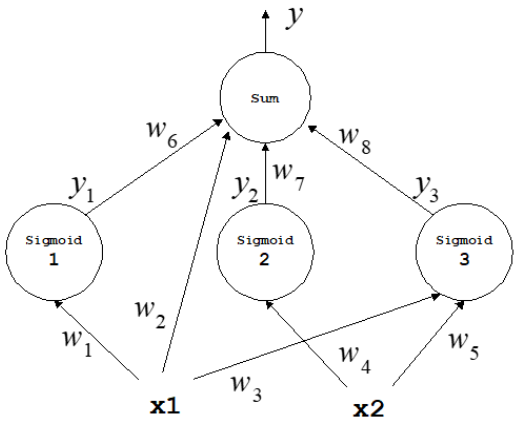
On solving the gradient descent:

$$\mathcal{R}\left\{\frac{\partial E}{\partial w_{kj}}\right\} = \mathcal{R}\{\delta_k\} z_j + \delta_k\, \mathcal{R}\{z_j\}$$

$$\mathcal{R}\left\{\frac{\partial E}{\partial w_{ji}}\right\} = x_i\, \mathcal{R}\{\delta_j^{(1)}\}.$$

---

3. Partially Connected Neural Network



Assume that the initial weights and thresholds are:

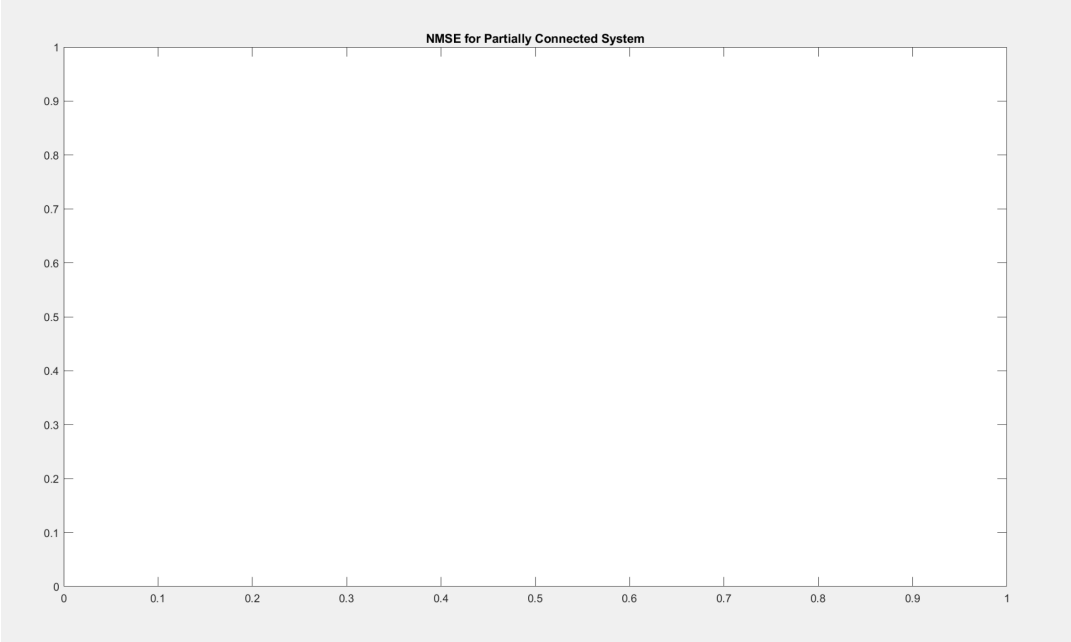
$w_1 = -0.25$ $w_2 = 0.33$ $w_3 = 0.14$ $w_4 = -0.17$ $w_5 = 0.16$
$w_6 = 0.43$ $w_7 = 0.21$ $w_8 = -0.25$

***Figure**: Neural Network to test for the assignment*

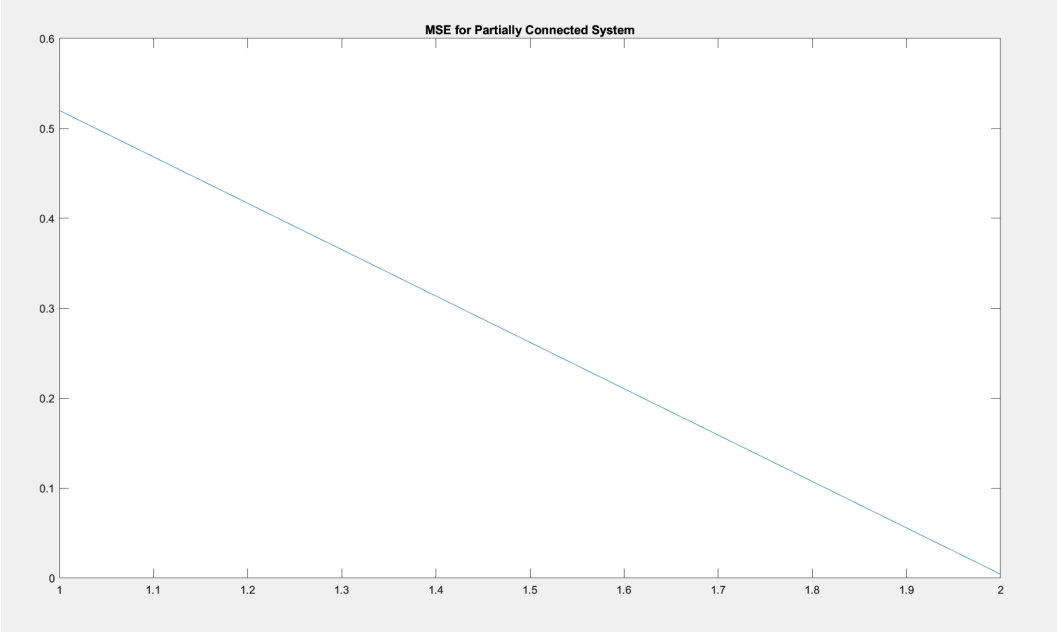
The *"PartiallyConnectedMLP.m"* script for a partially connected neural network consists of two inputs, a single hidden layer with three neurons, and a single output neuron as shown in the figure above. The input data is loaded from two files, 'Input.dat' and 'Output.dat'. The script returns the neural network weights at each epoch and calculates the network's root-mean-square error (RMSE) on the training data.

During each epoch, the script performs forward propagation of the input data through the neural network, followed by backpropagation to adjust the network weights based on the error between the desired output and the actual output. The backpropagation algorithm is implemented using the gradient descent method, where the gradients of the error regarding the weights are computed and used to update the weights. Finally, the script computes the Jacobian matrix and then the Hessian matrix and uses it to perform Newton's method to adjust the network weights. The script calculates and plots each epoch's total sum of squares (TSS) error.

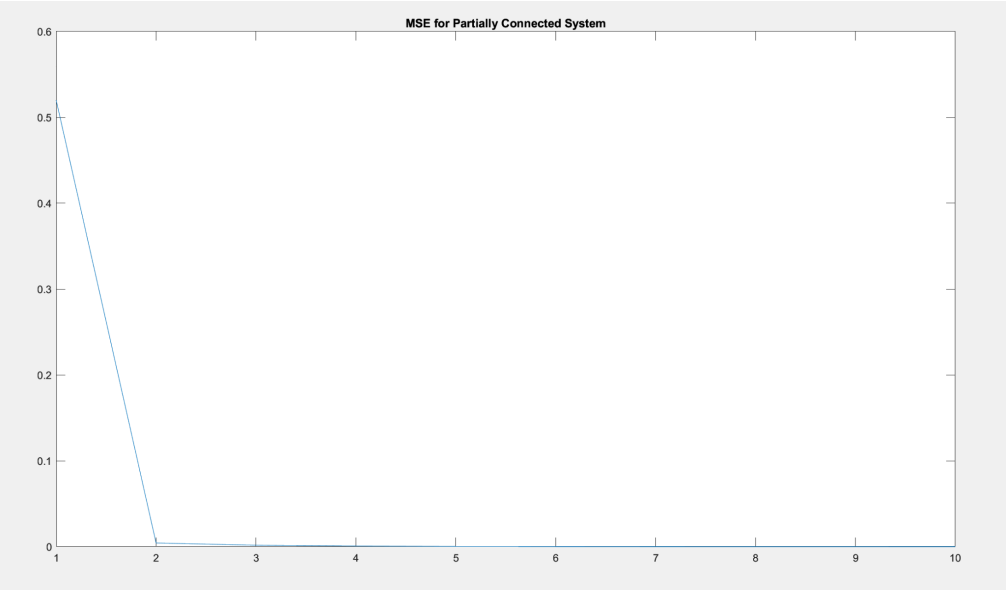
Plot for Normalized mean square error:



Plot for Mean Squared Error:



The curve is a straight line since the epochs stop at 2. If the TSS condition is removed, the curve looks like this:



---

4. Fully Connected Network with 10 inputs and 5 hidden nodes

A realistic fully connected network with 10 inputs and one hidden layer with 5 hidden nodes is in the file *"FullyConnectedNetwork.m"* in the Coursework folder. This code defines and trains a neural network using the backpropagation algorithm and Gauss-Newton method for optimisation. The neural network architecture consists of an input layer with 10 nodes, a hidden layer with 5 nodes, and an output layer with 1 node. The activation function used is the sigmoid function.

The neural network is trained on 100 samples, where X is the input data, and Y is the target output data. The weights of the network are initialised randomly using the rand function. In addition to backpropagation, the Gauss-Newton method is used to optimise the weights. The Jacobian and Hessian matrices are computed for each training pattern, and these matrices are used to update the weights in the direction of the steepest descent. The learning rate is also adjusted using a regularisation term.

The root mean squared error (RMSE) is calculated for each epoch, and the results are plotted in a graph at the end of the training loop.

Plot:

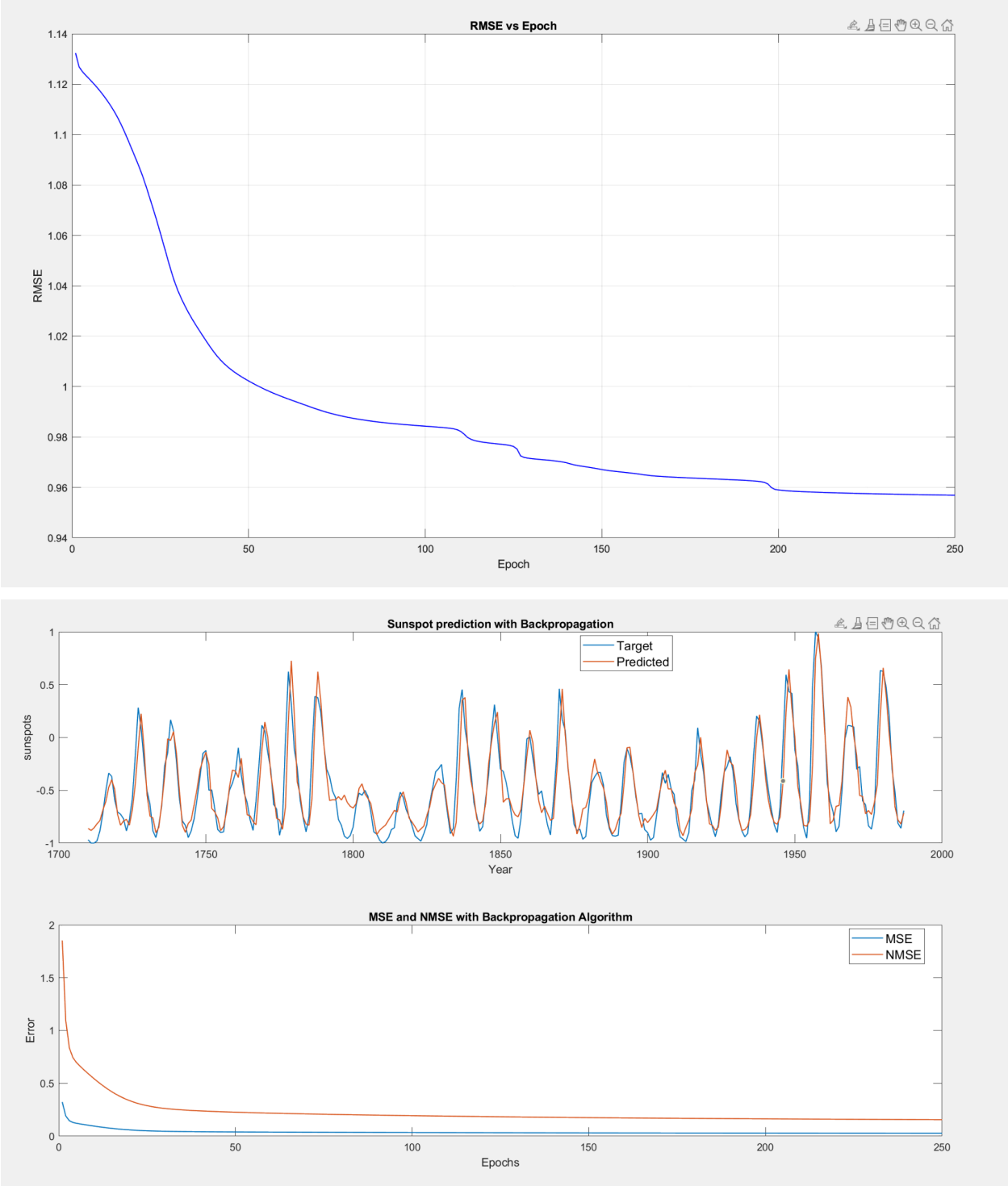


***Figure**: MSE and NMSE with Backpropagation*

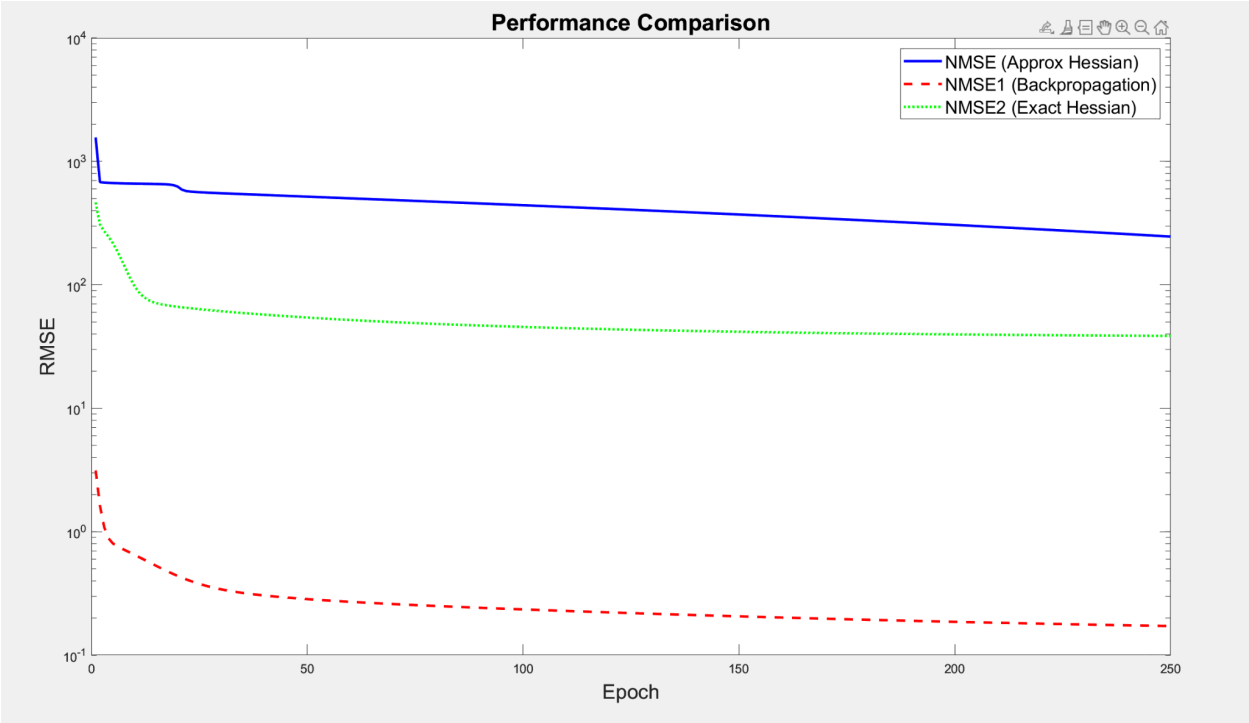
5. Comparison of performance with three algorithms

Separate code files for all three algorithms:

Backpropagation algorithm: "*backpropagation.m"*
Newton with approximate Hessian: "*NewtonApproxHess.m"*
Newton with exact Hessian: "*NewtonExactHess.m*"

To compare the performance of backpropagation, Newton with approximate Hessian, and Newton with exact Hessian, run "***main.m***"

All the different codes have been turned into functions, and the output of the functions are used to create subplots to give the approximation of the Sunspots series with these three algorithms.



Summary

In summary, the backpropagation algorithm is a simple and widely used algorithm, while the Gauss-Newton algorithm is computationally efficient but can converge to a local minimum. The exact Hessian with the R-propagation algorithm is computationally expensive but can converge faster than the regular backpropagation algorithm.

References

1. Haykin, S. S. (2007). Neural networks: A comprehensive foundation. Prentice Hall.
2. Nabney, I. T. (2002). Netlab: Algorithms for pattern recognition. Springer.
3. AndrewRiceMGW (GitHub Repo for R-Propagation Algorithm)