Name: Alperen Unal
Module: Neural Networks

DETAILED ANALYSIS OF NEURAL NETWORK APPLICATION IN TIME-SERIES PREDICTION OF SUNSPOT NUMBERS

Abstract

This report examines the application of a neural network in predicting sunspot activity—a crucial factor affecting solar radiation and, consequently, space and terrestrial weather. The dataset comprises a historical time series of observed sunspot counts, which exhibit cyclical behaviour over time. The predictive model is built using a feedforward neural network, trained via backpropagation and an exploratory Hessian-based method. A comparative study of these methods is presented to evaluate the performance of the neural network in forecasting future sunspot numbers.

Introduction

Sunspot prediction stands at the confluence of astrophysics and atmospheric science, which is key to unlocking the mysteries of solar dynamics and their terrestrial consequences. These dark blemishes on the Sun's surface, known as sunspots, signify intense magnetic activity, and their occurrence patterns are deeply tied to the solar cycle. Accurate predictions can inform us about space weather, affecting satellite operations, communication systems, and power grids on Earth. This study leverages the capabilities of neural networks to decipher the time series of sunspot numbers, extracting patterns from complex historical data to anticipate future activities. Our approach uses this time-honoured data to train a neural network, seeking to refine the prediction of sunspot emergence and behaviour prediction, which are pivotal for scientific inquiry and practical applications.

Dataset and Pre-processing

The sunspot dataset (sunspot.dat) is a historical record of yearly sunspot numbers dating back several centuries. The raw data undergoes pre-processing to improve the neural network's learning efficiency.

Normalization

The data is normalized using min-max scaling to the range [-1, 1], aiding the network's convergence during training. The normalization is described by:

$$\text{normalizedRelNums} = 2\left(\frac{\text{relativeNumbers}-\min(\text{relativeNumbers})}{\max(\text{relativeNumbers})-\min(\text{relativeNumbers})} - 0.5\right)$$

Time-Lagged Input Construction

The neural network's design capitalizes on historical sunspot sequences to predict future occurrences, recognizing that preceding patterns hold predictive power for subsequent activity. This approach enables the model to extrapolate past trends to forecast upcoming solar phenomena accurately.

Neural Network Design: The network architecture consists of an input layer, a single hidden layer, and an output layer. A schematic of the network's architecture is as follows:

- Input Layer: 10 neurons

- Hidden Layer: 5 neurons
- Output Layer: A single neuron

Methodology

Backpropagation

The backpropagation training process implemented for sunspot prediction is delineated into several key mathematical operations. It begins with the forward propagation of inputs through the network to generate an output, followed by a backward pass where the error is used to update the network weights.

- **Forward Pass:** The neural network computes the output for the hidden layer using the bipolar sigmoid activation function:

  $$h_{out} = 1 - \frac{2}{1+e^{-2(wih \cdot inputs)}}$$

  Then, the final output is calculated by the summation of weighted hidden outputs:

  $$output_{backprop} = \sum(who \cdot h_{out})$$

- **Error Computation:** For each input pattern, the error between the predicted output and the actual target value is computed:

  $$error = target - output_{backprop}$$

- **Backward Pass (Weight Update):** During the backward pass, the error is used to calculate the gradient of the loss function with respect to the weights. This is done by taking the derivative of the error with respect to the weights, leading to the following update rules:

  For the hidden-to-output weights (who):

  $$\Delta who = -\eta \cdot error \cdot h_{out}$$

  For the input-to-hidden weights (wih): The update takes into account the derivative of the activation function:

  $$\Delta wih = -\eta \cdot (error \cdot who) \cdot h_{out} \cdot (1 - h_{out}) \cdot inputs$$

  where η is the learning rate, $error$ is the computed error from the output, $hout$ is the output from the hidden layer neurons, and $inputsinputs$ are the input patterns.

- **Total Squared Error (TSE):** At the end of each epoch, the TSE is computed as the cumulative sum of squared errors for all patterns: $\text{TSE} = \sum_{i=1}^{n} (target_i - output_{backprop,i})^2$

Hessian Method

The Hessian method, an advanced technique in neural network training, is introduced to refine the weight update process by considering the curvature of the error landscape.

Unlike traditional backpropagation that relies on the gradient of the error for weight updates, the Hessian method uses second-order information, significantly altering the training dynamics.

- **Error Function and Gradient:** The error function is similar to that used in backpropagation, measured as the squared difference between predicted and actual sunspot numbers:

  $$E = \frac{1}{2} \sum (y_{actual} - y_{predicted})^2$$

  The gradient needed for the Hessian matrix is calculated as the first derivative of the error with respect to the weights.

- **Hessian Matrix Computation:** The Hessian matrix (H) represents the second-order partial derivatives of the error function concerning the weights. For a network with weights *w*, the Hessian is:

  Where *w*1,*w*2,…,*wn* are the weights in the neural network, and *E* is the error function. The elements of the Hessian matrix involve second derivatives of the error function, where each element $\frac{\partial^2 E}{\partial w_i \partial w_j}$ measures how the error changes as both weight *wi* and weight *wj* are varied. However, the computation of the full Hessian matrix can be computationally expensive, especially for large networks, which is why it is often approximated in practice.

  $$H = \begin{bmatrix} \frac{\partial^2 E}{\partial w_1^2} & \cdots & \frac{\partial^2 E}{\partial w_1 \partial w_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 E}{\partial w_n \partial w_1} & \cdots & \frac{\partial^2 E}{\partial w_n^2} \end{bmatrix}$$

- **Weight Update Rule:** The update rule incorporating the Hessian matrix is defined as: $w_{new} = w_{old} - H^{-1} \nabla E$

Where H^{-1} is the inverse of the Hessian matrix, and ∇E is the gradient vector of the error.

- **Regularization for Stability:**

  A small value δ is added to the diagonal elements of the Hessian to prevent issues with non-invertibility:

  $$H_{ii} = H_{ii} + \delta$$

Experimental Results

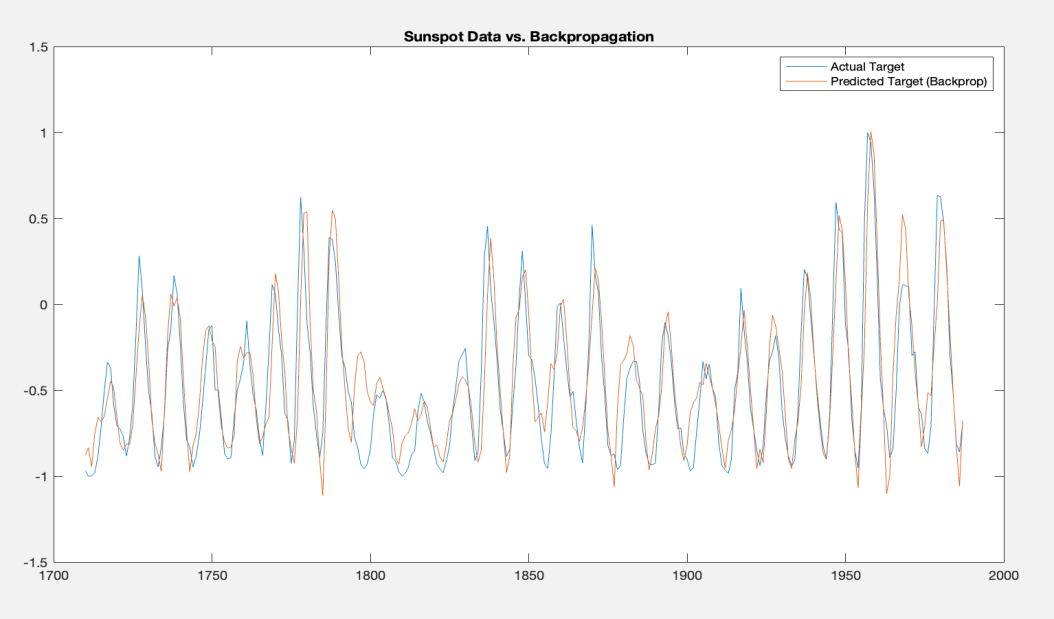
Performance Metrics: The Mean Squared Error (MSE) is computed for both the backpropagation and Hessian methods, serving as the primary performance metric. The MSEs after 50 and 100 epochs of the Hessian method and the unoptimized dataset's MSE are tabulated for comparison.

MSE Comparison Table:

Epochs / Method	MSE (Hessian, 50 epochs)	MSE (Hessian, 100 epochs)	MSE (Dataset)
MSE Values	0.095485	0.104531	0.173516

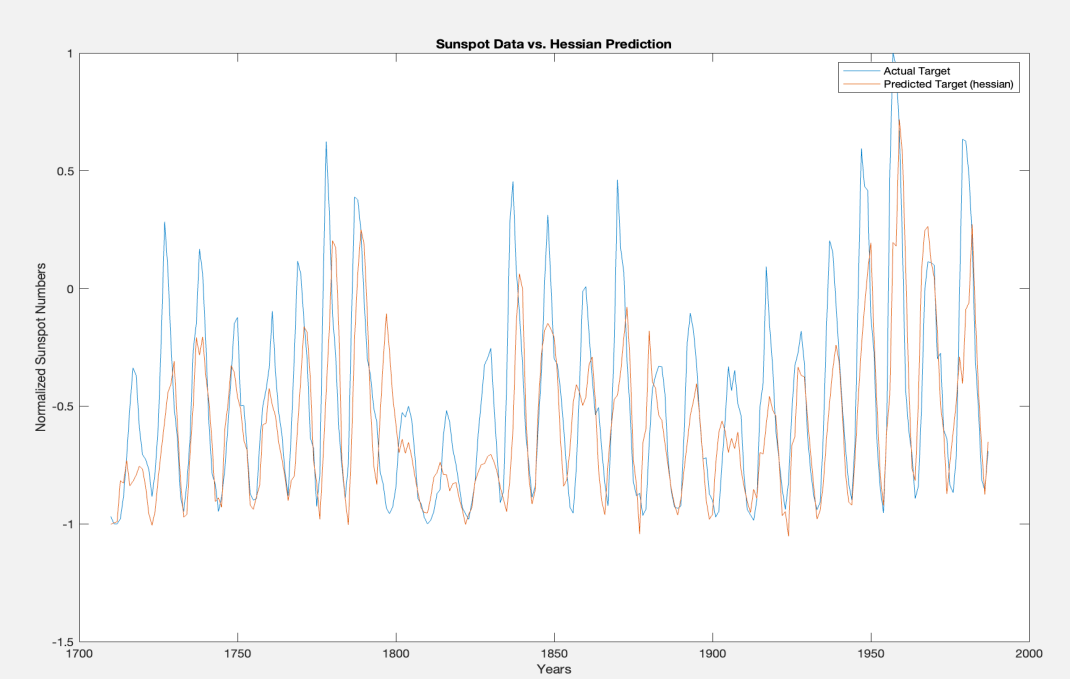
The MSE values indicate that the Hessian method at 50 epochs outperforms the unoptimized dataset, demonstrating the effectiveness of neural network optimization.

Visual Analysis

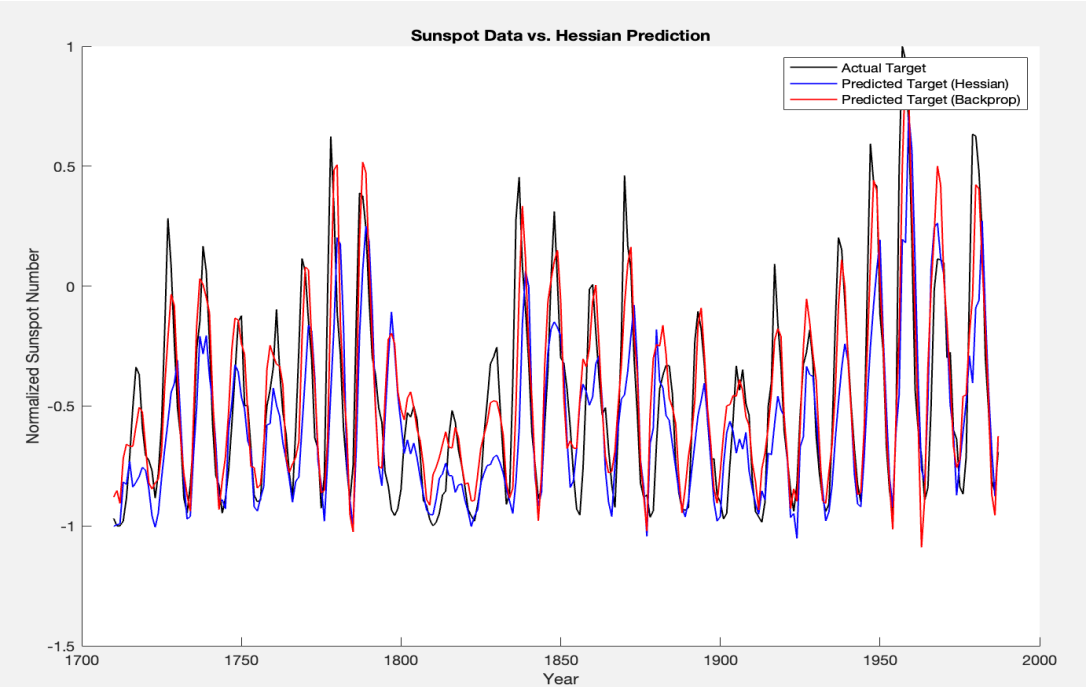


The backpropagation vs. sunspot data targets graph compares actual historical sunspot data with the predicted values obtained from a neural network trained using the backpropagation algorithm. The visual similarity between the two lines indicates that the model has learned to

approximate the cyclical pattern of sunspots to a reasonable degree, although some discrepancies suggest room for further model refinement.



The graph displays a comparison between actual sunspot numbers and those predicted by a Hessian-based neural network model, indicating a moderate alignment with some noticeable prediction errors.



The comparative graph showcases the performance of two neural network training algorithms, Hessian and backpropagation, against the actual normalized sunspot numbers over time. The

backpropagation predictions (in red) closely follow the actual data (in black), indicating a relatively good fit, while the Hessian method predictions (in blue) also track the cycle but with less precision in amplitude. This visual comparison suggests that while both methods capture the cyclical nature of sunspot activity, the backpropagation algorithm may provide a closer match to the historical data.

Conclusion

The Hessian method offers a nuanced approach to weight updates by accounting for the curvature of the error function. In theory, this method should enable more informed updates and possibly faster convergence. However, its implementation can be computationally intensive due to the calculation of the Hessian matrix and its inversion. When applied to sunspot prediction, the Hessian method must be carefully managed to exploit its advantages in capturing the underlying patterns within the historical data. As such, it holds promise for enhancing the predictive accuracy of the neural network model when appropriately utilized.

Matlab Code

```
% Clearing all variables and console
clear all;
clear;
clc;

% Loading sunspot data from a file
load sunspot.dat;

% Extracting years and sunspot numbers into separate variables
years = sunspot(:,1);
relativeNumbers = sunspot(:,2);

% Calculating the mean and standard deviation of sunspot numbers for
potential normalization
meanRelativeNums = mean(relativeNumbers(:));
stdRelativeNums = std(relativeNumbers(:));

% Normalizing sunspot numbers to range [-1, 1] for better neural network
performance
minValue = min(relativeNumbers(:));
maxValue = max(relativeNumbers(:));
normalizedRelNums = 2.0 * ((relativeNumbers - minValue) / (maxValue -
minValue) - 0.5);

% Transposing normalized data to match the expected input orientation for
training
transposedRelNums = normalizedRelNums';

% Set dimensions for neural network input and output
inputDimension = 10;
outputDimension = length(transposedRelNums) - inputDimension;

% Prepare storage for input patterns and target outputs
y = zeros(1, outputDimension);
x = zeros(outputDimension, inputDimension);

% Filling input and output arrays with lagged sequences of the normalized
data
```

```
for i = 1:outputDimension
   y(i) = transposedRelNums(i + inputDimension);
   for j = 1:inputDimension
       x(i,j) = transposedRelNums(i - j + inputDimension);
   end
end

% Rearranging input data into a matrix form suitable for the network
inputPatterns = x';
targetOutputs = y;

% Initializing neural network parameters
NINPS = 10;
NHIDS = 5;
NOUTS = 1;
NPATTS = length(inputPatterns);
rng(56);
lr = 0.001; % Set learning rate
wih = 0.5 * (rand(NINPS, NHIDS) - 0.5);
whout = 0.5 * (rand(NHIDS, NOUTS) - 0.5);
wih_backp = wih;  % Backup initial weights
whout_backp = whout;  % Backup initial weights

% Backpropagation
% Setting the number of iterations for training using backpropagation
backp_iters = 200;
for epoch = 1:backp_iters
    backp_out = zeros(1, NPATTS);  % Storing network outputs for all
patterns
    totalSquaredError_backp = 0;  % Initializing total error for the epoch

    for pIn = 1:NPATTS
        patternInput = inputPatterns(:, pIn);
        netInputHidden = zeros(1, NHIDS);  % Computing inputs to hidden
layer

        % Calculating hidden layer inputs by summing weighted inputs
        for hIn = 1:NHIDS
            for iIn = 1:NINPS
                netInputHidden(hIn) = netInputHidden(hIn) +
patternInput(iIn) * wih_backp(iIn, hIn);
            end
        end
        % Activation function for hidden layer
        h_out = 1 - 2 ./ (exp(2 * netInputHidden) + 1);

        % Calculating output layer inputs by summing weighted hidden
outputs
        net_h_out = zeros(1, NHIDS);
        for hIn = 1:NHIDS
            net_h_out(hIn) = h_out(hIn) * whout_backp(hIn);
        end
        output_backp = sum(net_h_out);
        backp_out(pIn) = output_backp;

        % Error and update total squared error
        error = targetOutputs(pIn) - output_backp;
        deltaOutput = error;  % Derivative for output error
        totalSquaredError_backp = totalSquaredError_backp + error^2;
```

```
        deltaHidden = zeros(1, NHIDS);  % Error derivative for hidden
layers

        % Updating weights from hidden to output layers
        for hIn = 1:NHIDS
            deltaHidden(hIn) = deltaOutput * whout_backp(hIn) * h_out(hIn)
* (1 - h_out(hIn));
            whout_backp(hIn) = whout_backp(hIn) + lr * deltaOutput *
h_out(hIn);
        end
    end

    % Display training progress
    fprintf('Epoch %3d: Approximate Backpropagation Method Error = %f\n',
epoch, totalSquaredError_backp);
end

% Hessian (50 epochs)
% Initializing Hessian-based training iterations
hessian = 0;
for epoch = 1:50
    hessian_out = zeros(1, NPATTS);  % Output storage
    for p = 1:NPATTS
        patternInput = inputPatterns(:, p);
        netInputHidden = zeros(1, NHIDS);  % Input calculation for hidden
layer

        % Sum weighted inputs for each hidden neuron
        for j = 1:NHIDS * NINPS
            hj = mod(j-1, NHIDS) + 1;  % Determining hidden neuron index
            ij = ceil(j / NHIDS);  % Determining input index
            netInputHidden(hj) = netInputHidden(hj) + patternInput(ij) *
wih(ij, hj);
        end
        % Activation for hidden neurons
        h_out = 1 - 2 ./ (exp(2 * netInputHidden) + 1);

        % Outputs for each hidden neuron connected to the output neuron
        net_h_out = zeros(1, NHIDS);
        for k = 1:NHIDS
            net_h_out(k) = h_out(k) * whout(k);
        end
        % Sum hidden neuron outputs to form the final network output
        output_hessian = sum(net_h_out);
        hessian_out(:, p) = output_hessian;

        % Error calculation for the current pattern
        Error = targetOutputs(:, p) - output_hessian;
        TSS = sum(Error.^2);  % Total squared error calculation

        % Error gradients for backpropagation
        BetaOutput = Error;
        BetaHidden = zeros(1, NHIDS);
        for j = 1:NHIDS
            BetaHidden(j) = BetaOutput * whout(j) * h_out(j) * (1 -
h_out(j));
        end

        % Gradients for weight updates
```

```
        gradient_who = BetaOutput * h_out;
        gradient_wih = patternInput * BetaHidden;

        % Combining all gradients for Hessian matrix calculation
        gradient = [gradient_who(:); gradient_wih(:)];

        % Simple assumption for error sensitivity
        Beta_out_Hessian = 1;
        Gradient_hidd_out_hessian = Beta_out_Hessian * h_out;

        % Hessian components for each hidden neuron
        Beta_Hidden_Hessian = zeros(1, NHIDS);
        for j = 1:NHIDS
            Beta_Hidden_Hessian(j) = Beta_out_Hessian * whout(j) *
h_out(j) * (1 - h_out(j));
        end

        % Gradient from input neurons for the Hessian matrix
        Hess_grad_hidd_in = patternInput * Beta_Hidden_Hessian;

        % Updating Hessian matrix with new gradient information
        Hessian_gradient = [Gradient_hidd_out_hessian(:);
Hess_grad_hidd_in(:)];
        hessian = hessian + Hessian_gradient * Hessian_gradient';

    end
    % Normalizing the Hessian matrix by the number of patterns
    hessian = (1 / NPATTS) * hessian;

    % Adding a small value to the diagonal elements of the Hessian for
numerical stability
    smallValue = 0.0001;
    for idx = 1:min(size(hessian))
        hessian(idx, idx) = hessian(idx, idx) + smallValue;
    end

    % Solving for weight updates using the inverted Hessian matrix
    delta_Weight = (hessian \ gradient) * lr;
    delta_wih = delta_Weight(1:length(wih(:)));  % Updates for input-to-
hidden weights
    delta_who = delta_Weight(length(wih(:)) + 1:end);  % Updates for
hidden-to-output weights

    % Applying weight updates
    wih = wih + reshape(delta_wih, size(wih));
    whout = whout + reshape(delta_who, size(whout));

    % Error information for each Hessian epoch
    fprintf('Epoch %3d:  Approximate Hessian Method Error =
%f\n',epoch,TSS);
end

% Preparing to plot the results in a new figure
figure;
hold on;

% Actual target values
plot(years(11:end), targetOutputs, 'k', 'LineWidth', 1);
```

```
% Predictions from the Hessian method
plot(years(11:end), hessian_out, 'b', 'LineWidth', 1);

% Predictions from the Backpropagation method
plot(years(11:end), backp_out, 'r', 'LineWidth', 1);

% Legend and titles for clarity
legend('Actual Target', 'Predicted Target (Hessian)', 'Predicted Target
(Backprop)');
title('Sunspot Data vs. Hessian Prediction');
xlabel('Year');
ylabel('Normalized Sunspot Number');

% Release plot hold to finish plotting
hold off;

% The Mean Squared Error for the Hessian method
% Initializing a variable to accumulate squared errors for Hessian
totalSquaredError_hessian = 0;
for p = 1:NPATTS
    % Squared errors for each pattern
    error = targetOutputs(p) - hessian_out(p);
    totalSquaredError_hessian = totalSquaredError_hessian + error^2;
end
MSE_hessian = totalSquaredError_hessian / NPATTS;
fprintf('Final MSE for Hessian : %f\n', MSE_hessian);

% The mean of the target outputs
meanTargets = mean(targetOutputs);

% The MSE as the average squared difference from the mean
datasetMSE = mean((targetOutputs - meanTargets).^2);
fprintf('Dataset MSE without Hessian: %f\n', datasetMSE);
```

References

Nikolaev, N. (2024) Machine Learning Lab Codes and Lecture Slides, Goldsmiths, University of London.

Bishop, C.M. (1995). Neural Networks for Pattern Recognition, Oxford University Press, Oxford, UK.

Nabney, I. (2002). Netlab: Algorithms for Pattern Recognition, Springer series Advances in Pattern Recognition.

Haykin, Simon (1999). Neural Networks. A Comprehensive Foundation.,
Second Edition, Prentice-Hall, Inc., New Jersey.