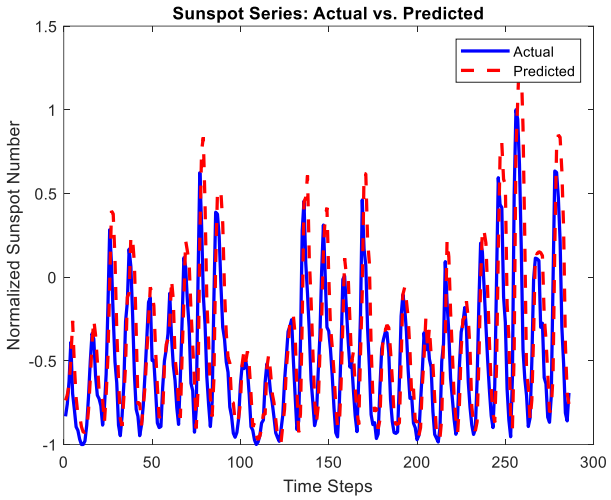
Implementing Second Order Training Algorithms
for MLP Neural Networks

1. **The Skip Connection Network: Test the Newton's algorithm exact Hessian with the simple neural network.**

   NMSE (Normalized Mean Squared Error): 0.57751



Difficulties: Gradient and weight explosion/vanishing. Our output always displayed "NaN" or "Inf Loss" before we adopt any prevent technique.

Techniques we used to tackle the problem:

- Adopt Leaky ReLU as activation function instead of Sigmoid. Because the Leaky ReLU function adjusts for the zero gradient problem for negative values by giving a very small linear component to the negative input. Yet actually it didn't work solely.

```
% Activation functions and derivatives
leaky_relu = @(x) (x > 0) .* x + (x <= 0) .* (0.01 * x);
leaky_relu_derivative = @(x) (x > 0) + (x <= 0) * 0.01;
```

- Gradient Clipping. Limit the gradient value to the range of [-1, 1] to avoid excessive or divergent weight updates caused by excessive gradients.

```
% Gradient clipping
grad_clip_threshold = 1;
grad_weights_IH = max(min(grad_weights_IH, grad_clip_threshold), -grad_clip_threshold);
grad_weights_HO = max(min(grad_weights_HO, grad_clip_threshold), -grad_clip_threshold);
grad_weights_IO = max(min(grad_weights_IO, grad_clip_threshold), -grad_clip_threshold);
```

- Weight Update Clipping. Limit the weight value to the range of [-1, 1] to avoid excessive or divergent weight updates caused by excessive weights.
- L2 Regularization. Prevent the weight value from being too large and help avoid weight explosion.

```
% Update weights
lambda = 1e-1;
update_IH = pinv(hessian_IH + lambda * eye(size(hessian_IH))) * grad_weights_IH(:);
update_HO = pinv(hessian_HO + lambda * eye(size(hessian_HO))) * grad_weights_HO;
update_IO = (hessian_IO + lambda) \ grad_weights_IO;


% loss calculation with regularization
loss_value = mean((y - network_output).^2) + lambda * (sum(weights_IH(:).^2) + sum(weights_HO(:).^2) + weight_IO^2);
```

- Hessian Matrix Regularization. Avoid the occurrence of singular matrices when calculating derivatives, thereby preventing gradient explosion or gradient disappearance.

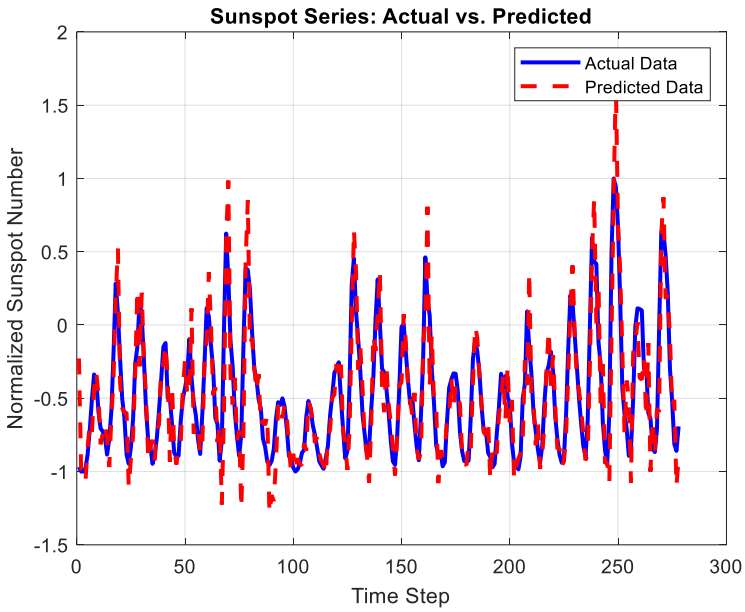
```
% Calculate Hessian with regularization
weights = [weights_IH(:); weights_HO; weights_IO];
hessian = compute_hessian(weights, x(i,:), y(i), leaky_relu, leaky_relu_derivative, linear_activation) + 1e-3 * eye(numel(weights));
```

- Add weight decay in the gradient of the backpropagation in the loss function of Hessian. The weight decay corresponding to L2 regularization can limit the weight and prevent weight explosion.

```
% get accurate first-order gradient information for the loss function with respect to each weight in the network
delta_output = -(y - network_output);
delta_hidden = (delta_output * weights_HO') .* leaky_relu_derivative(hidden_input);
grad_weights_IH = x' * delta_hidden + 2 * lambda * weights_IH;
grad_weights_HO = hidden_output' * delta_output + 2 * lambda * weights_HO;
grad_weights_IO = x(1) * delta_output + 2 * lambda * weight_IO;
```

2. The Newton's algorithm using the approximate Hessian Broyden–Fletcher–Goldfarb–Shanno (BFGS) Algorithm.

NMSE (Normalized Mean Squared Error): 0.28551



Difficulties: Gradient and weight explosion/vanishing. Our output always displayed "NaN" or "Inf Loss" before we adopt any prevent technique.

Techniques we used to tackle the problem:

- Gradient Clipping. Limit the gradient value to the range of [-1, 1] to avoid excessive or divergent weight updates caused by excessive gradients.

```
% Gradient clipping (added)
grad_clip_threshold = 1;
grad_weights_IH = grad(1:NIPT*NHID);
grad_weights_HO = grad(NIPT*NHID+1:end);
grad_weights_IH = max(min(grad_weights_IH, grad_clip_threshold), -grad_clip_threshold);
grad_weights_HO = max(min(grad_weights_HO, grad_clip_threshold), -grad_clip_threshold);
grad = [grad_weights_IH; grad_weights_HO];
```

- BFGS algorithm as requested. Use a quadratic approximation to estimate the curvature of the objective function to speed up convergence and avoid falling in a local minimum value. It updates by the inverse of the approximating Hessian matrix, helping to avoid the vanishing gradient problem.
- Line Search in BFGS optimization. Help to find the appropriate step size to avoid weight diverging or failing to converge.

```
for iter = 1:max_iter
    d = -H * grad;
    step_size = line_search(f, weights, d, grad);
    weights_new = weights + step_size * d;
```

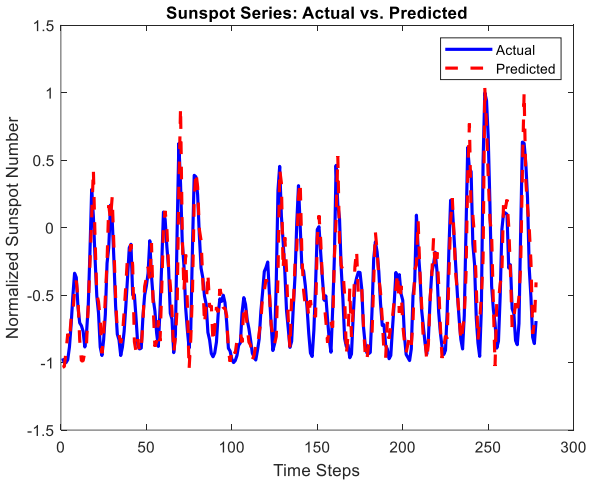
- Convergence check. When the gradient approaches to 0, the optimization process will stop early, helping to avoid overfitting and weight divergence.

```
if all(abs(y) < 1e-8)
    break;
end
```

- Liner Search Function. Ensure to find the effective step size so that the objective function decreases along the gradient descent direction, making sure the descent while avoiding an excessively large step size that could cause skipping the lowest point or generating oscillations.

3. The Classical Backpropagation Algorithm (gradient descent): Fully connected network with 10 inputs and 5 hidden nodes.

NMSE (Normalized Mean Squared Error): 0.19642



Techniques we used:

Gradient Clipping. Since we adopt gradient clipping in the previous algorithms, we decide to use the same technique as well for making the model architecture as close as possible.

```
% Gradient clipping
grad_clip_threshold = 1;
grad_weights_IH = max(min(grad_weights_IH, grad_clip_threshold), -grad_clip_threshold);
grad_weights_HO = max(min(grad_weights_HO, grad_clip_threshold), -grad_clip_threshold);
```

4. The Newton's algorithm using the exact Hessian computed with the R-propagation algorithm.

N/A.

5. Main differences between exact Hessian, approximate Hessian BGFS and Backpropagation (Gradient Descent):

Aspect	Exact Hessian	Approximate Hessian (BFGS)	Backpropagation (Gradient Descent)
Computation of Derivatives	Needs to calculate both first-order and second-order derivatives explicitly from the loss function.	Uses first-order derivatives and approximates the Hessian by using past gradient evaluations.	Uses only first-order derivatives, directly computed from the loss function.

Memory Usage (n is the number of parameters)	High. Needs to store a full Hessian matrix which is of size n × n.	Lower. Constructs an approximation to the Hessian incrementally. Only stores a vectors of size n.	Minimal. Only store gradients and possibly momentum terms depending on the variant.
Computational Complexity (n is the number of parameters)	High. Needs to compute and invert a large matrix that can be computationally expensive for large n.	Generally lower per iteration. Avoids directly compute and invert of the full Hessian.	Low per iteration. Only involves straightforward gradient calculations and parameter updates.
Accuracy	High. Provides precise curvature information that can lead to more accurate updates.	Potentially less accurate. Approximates the curvature information, but usually sufficient for practical use.	Less accurate. Doesn't account for curvature; need careful tuning step size to avoid overshooting.
Convergence	Potentially faster. Consider the number of iterations due to precise updates.	May require more iterations. Yet each iteration is typically cheaper, balancing overall efficiency.	Usually slower and dependent on step size. May get stuck in local minima or require many iterations for complex situations.
Implementation Complexity	Complex. Particularly in manage and update a large Hessian matrix.	Simpler. Leveraging efficient vector operations and updates based on rank-one adjustments.	Simplest. Implemented widely and understood; involves standard loops over gradient calculations and updates.
Use Cases	Suited best for problems with a smaller number of parameters that computational overhead is manageable.	Preferred in large-scale optimization problems because of the scalability and less computational demands.	Most common in machine learning for a wide range of problems, especially when scalability and simplicity is on the first priority.

6. Matlab Code

Exact Hessian in Skip Connection

```
clear all;
clc;

% Exact Hessian in Skip Connection

% load and preprocess(normalize the data) the sunspot dataset
```

```
load sunspot.dat
% the first column is the year, stored in the variable year
% the second column is the associated sunspot number, stored in the variable relNums.
year=sunspot(:,1); relNums=sunspot(:,2);
% calculate the mean (ynrmv) and standard deviation (sigy) of relNums.
% the mean is for subsequent centering (demeaning), and the standard deviation is for
subsequent normalization.
ynrmv=mean(relNums(:)); sigy=std(relNums(:));
% using calculated the mean and standard deviationin in the previous step
% to normalize the raw sunspot data by subtracting the mean and
% dividing by the standard deviation from each data point.
nrmY=(relNums(:)-ynrmv)./sigy;
% calculate the minimum (ymin) and maximum (ymax) values of the normalized data nrmY,
% then applied to linearly transform the data into the range [-1, 1].
ymin=min(nrmY(:)); ymax=max(nrmY(:));
% the normalized data nrmY is further transformed into the range [-1, 1].
% first normalize each value of nrmY to [0, 1] by subtracting ymin and
% dividing by the range (ymax - ymin),
% then adjust the range to [-1, 1] by multiplying by 2.0 and subtracting 0.5.
relNums=2.0*((nrmY-ymin)/(ymax-ymin)-0.5);

% parameters for the neural network
idim = 2;   % input dimension (number of lagged values used as input)
odim = length(relNums) - idim;  % output dimension (number of predicted values)
x = zeros(odim, idim);  % input data matrix
y = zeros(odim, 1); % target output vector

% create inputs and desired outputs
for i = 1:odim
    y(i) = relNums(i + idim);    % target output is the next value after the input sequence
    x(i, :) = relNums(i:i+idim-1)';  % input sequence is a column vector of lagged values
end

% network parameters
NIPT = idim;    % number of input neurons
NHID = 3;   % number of hidden neurons
NOUT = 1;   % number of output neurons
lr = 0.001; % learning rate

% activation functions and derivatives
leaky_relu = @(x) (x > 0) .* x + (x <= 0) .* 0.01 .* x; % leaky ReLU activation function
leaky_relu_derivative = @(x) (x > 0) + (x <= 0) * 0.01; % derivative of leaky ReLU
```

```
linear_activation = @(x) x; % linear activation function for output layer

% initialize Weights
w = [-0.25, 0.33, 0.14, -0.17, 0.16, 0.43, 0.21, 0.25];

% weights from Input layer to Hidden layer
weights_IH = [w(1), 0, w(3);    % connection from x1 to y1, y2 (none), and y3
              0, w(4), w(5)];   % connection from x2 to y1 (none), y2, and y3

% weights from Hidden layer to Output layer
weights_HO = [w(6); w(7); w(8)];    % connections from y1, y2, y3 to output y

% weights from Input layer to Output layer (Skip connection)
weights_IO = w(2); % Direct connection from x1 to output y

% initialize network output
network_output = zeros(odim, 1);    % predicted output values

% train the network
num_epochs = 100;   % number of training epochs
for epoch = 1:num_epochs
    for i = 1:odim
        % feedforward
        hidden_input = x(i, :) * weights_IH;    % input to hidden layer
        hidden_output = leaky_relu(hidden_input);   % output of hidden layer
        output_input = hidden_output * weights_HO + weights_IO * x(i, 1);   % input to output
        layer
        network_output(i) = linear_activation(output_input);    % output of the network

        % backpropagation
        delta_output = -(y(i) - network_output(i)); % output layer error
        delta_hidden = (delta_output * weights_HO') .* leaky_relu_derivative(hidden_input); %
        hidden layer error
        grad_weights_IH = x(i, :)' * delta_hidden;  % gradient of weights from input to hidden
        layer
        grad_weights_HO = hidden_output' * delta_output;    % gradient of weights from hidden
        to output layer
        grad_weights_IO = x(i, 1) * delta_output;   % gradient of weights for skip connection

        % gradient clipping
        grad_clip_threshold = 1;     % threshold for gradient clipping
```

```
grad_weights_IH = max(min(grad_weights_IH, grad_clip_threshold), -
grad_clip_threshold); % clip gradients for weights_IH
grad_weights_HO = max(min(grad_weights_HO, grad_clip_threshold), -
grad_clip_threshold); % clip gradients for weights_HO
grad_weights_IO = max(min(grad_weights_IO, grad_clip_threshold), -
grad_clip_threshold); % clip gradients for weights_IO

% calculate Hessian with regularization
weights = [weights_IH(:); weights_HO; weights_IO];  % combie all weights into a single
vector
% compute Hessian matrix with regularization
hessian = compute_hessian(weights, x(i,:), y(i), leaky_relu, leaky_relu_derivative,
linear_activation) + 1e-3 * eye(numel(weights));
hessian_IH = hessian(1:6, 1:6); % extract Hessian for weights_IH
hessian_HO = hessian(7:9, 7:9); % extract Hessian for weights_HO
hessian_IO = hessian(10, 10);   % extract Hessian for weights_IO

% update weights
lambda = 1e-1;  % regularization parameter
update_IH = pinv(hessian_IH + lambda * eye(size(hessian_IH))) * grad_weights_IH(:); %
update for weights_IH
update_HO = pinv(hessian_HO + lambda * eye(size(hessian_HO))) * grad_weights_HO;    %
update for weights_HO
update_IO = (hessian_IO + lambda) \ grad_weights_IO;    % update for weights_IO

% update clipping
update_clip_threshold = 1;  % threshold for update clipping
update_IH = max(min(update_IH, update_clip_threshold), -update_clip_threshold); % clip
updates for weights_IH
update_HO = max(min(update_HO, update_clip_threshold), -update_clip_threshold); % clip
updates for weights_HO
update_IO = max(min(update_IO, update_clip_threshold), -update_clip_threshold); % clip
updates for weights_IO

% upply updates
weights_IH = weights_IH - lr * reshape(update_IH, size(weights_IH));    % update
weights_IH
weights_HO = weights_HO - lr * update_HO;   % update weights_HO
weights_IO = weights_IO - lr * update_IO;   % update weights_IO

% calculate loss with regularization
```

```
        [loss_value, ~] = compute_loss(weights_IH, weights_HO, weights_IO, x(i, :), y(i),
        lambda, leaky_relu, leaky_relu_derivative, linear_activation);

        % check for NaN or Inf loss
        if isnan(loss_value) || isinf(loss_value)
            disp('Loss value becomes NaN or Inf. Training aborted.');
            break;
        end
    end
end

% calculate NMSE
mean_y = mean(y);
nmse_loss = sum((y - network_output).^2) / sum((y - mean_y).^2);    % normalized mean squared
error
disp(['NMSE Loss: ', num2str(nmse_loss)]);

% plot results
figure;
plot(1:length(y), y, 'b', 'LineWidth', 2);  % plot actual values in blue
hold on;
plot(1:length(network_output), network_output, 'r--', 'LineWidth', 2);  % plot predicted values
in red dashed line
hold off;
title('Sunspot Series: Actual vs. Predicted');
xlabel('Time Steps');
ylabel('Normalized Sunspot Number');
legend('Actual', 'Predicted');

% Hessian calculation with regularization
function H = compute_hessian(weights, x, y, leaky_relu, leaky_relu_derivative,
linear_activation)
    eps = 1e-6; % perturbation size for numerical approximation to slightly perturb the
    weights
    num_weights = numel(weights);   % total number of weights
    H = zeros(num_weights, num_weights);     % initialize Hessian matrix

    % computes the gradient of the loss function without regularization
    % use to calculate the Hessian matrix elements
    [~, grad] = compute_loss(reshape(weights(1:6), 2, 3), weights(7:9), weights(10), x, y, 0,
    leaky_relu, leaky_relu_derivative, linear_activation);
```

```
    % approximate second-order derivatives based on the neural network Hessian matrix
    for i = 1:num_weights
        for j = 1:num_weights
            % perturb the i-th weight
            weights_perturbed_i = weights;
            weights_perturbed_i(i) = weights_perturbed_i(i) + eps;
            % calculate the gradient of the loss function after the i-th perturbation weight
            [~, grad_perturbed_i] = compute_loss(reshape(weights_perturbed_i(1:6), 2, 3),
            weights_perturbed_i(7:9), weights_perturbed_i(10), x, y, 0, leaky_relu,
            leaky_relu_derivative, linear_activation);

            % perturb the j-th weight
            weights_perturbed_j = weights;
            weights_perturbed_j(j) = weights_perturbed_j(j) + eps;
            % calculate the gradient of the loss function after the j-th perturbation weight
            [~, grad_perturbed_j] = compute_loss(reshape(weights_perturbed_j(1:6), 2, 3),
            weights_perturbed_j(7:9), weights_perturbed_j(10), x, y, 0, leaky_relu,
            leaky_relu_derivative, linear_activation);

            % perturb i and j weights at the same time
            weights_perturbed_ij = weights;
            weights_perturbed_ij(i) = weights_perturbed_ij(i) + eps;
            weights_perturbed_ij(j) = weights_perturbed_ij(j) + eps;
            % calculate the gradient after simultaneous perturbation
            [~, grad_perturbed_ij] = compute_loss(reshape(weights_perturbed_ij(1:6), 2, 3),
            weights_perturbed_ij(7:9), weights_perturbed_ij(10), x, y, 0, leaky_relu,
            leaky_relu_derivative, linear_activation);

            % approximate second-order derivatives for weights(i) and (j)
            H(i, j) = (grad_perturbed_ij(i) - grad_perturbed_i(i) - grad_perturbed_j(i) +
            grad(i)) / (eps^2);
        end
    end
end

% Loss calculation with regularization
% calcualte 3 layers and 1 skip connection weights, input X and y, lambda,
% and all acticvation function
function [loss_value, grad] = compute_loss(weights_IH, weights_HO, weight_IO, x, y, lambda,
leaky_relu, leaky_relu_derivative, linear_activation)

    % feedforward
```

```
    hidden_input = x * weights_IH;  % input to hidden layer
    hidden_output = leaky_relu(hidden_input);   % output of hidden layer
    output_input = hidden_output * weights_HO + weight_IO * x(1);   % input to output layer
    network_output = linear_activation(output_input);   % output of the network

    % loss calculation with regularization
    % mean squared error with L2 regularization
    loss_value = mean((y - network_output).^2) + lambda * (sum(weights_IH(:).^2) +
    sum(weights_HO(:).^2) + weight_IO^2);

    % Backpropagation
    % calculate the error of the output layer
    % weight decay
    delta_output = -(y - network_output);   % output layer error
    delta_hidden = (delta_output * weights_HO') .* leaky_relu_derivative(hidden_input); %
    hidden layer error
    grad_weights_IH = x' * delta_hidden + 2 * lambda * weights_IH;  % gradient of weights from
    input to hidden layer with weight decay
    grad_weights_HO = hidden_output' * delta_output + 2 * lambda * weights_HO;  % gradient of
    weights from hidden to output layer with weight decay
    grad_weights_IO = x(1) * delta_output + 2 * lambda * weight_IO; % gradient of weights for
    skip connection with weight decay

    % get accurate first-order gradient information for the loss function with respect to each
    weight in the network
    grad = [grad_weights_IH(:); grad_weights_HO; grad_weights_IO];
end
```

Approximate Hessian (BFGS)

```
clear all;
clc;

% Approximate Hessian (BFGS)

% load and preprocess(normalize the data) the sunspot dataset
load sunspot.dat
% the first column is the year, stored in the variable year
% the second column is the associated sunspot number, stored in the variable relNums.
year=sunspot(:,1); relNums=sunspot(:,2);
% calculate the mean (ynrmv) and standard deviation (sigy) of relNums.
% the mean is for subsequent centering (demeaning), and the standard deviation is for
subsequent normalization.
```

```
ynrmv=mean(relNums(:)); sigy=std(relNums(:));
% using calculated the mean and standard deviationin in the previous step
% to normalize the raw sunspot data by subtracting the mean and
% dividing by the standard deviation from each data point.
nrmY=(relNums(:)-ynrmv)./sigy;
% calculate the minimum (ymin) and maximum (ymax) values of the normalized data nrmY,
% then applied to linearly transform the data into the range [-1, 1].
ymin=min(nrmY(:)); ymax=max(nrmY(:));
% the normalized data nrmY is further transformed into the range [-1, 1].
% first normalize each value of nrmY to [0, 1] by subtracting ymin and
% dividing by the range (ymax - ymin),
% then adjust the range to [-1, 1] by multiplying by 2.0 and subtracting 0.5.
relNums=2.0*((nrmY-ymin)/(ymax-ymin)-0.5);

% parameters for the neural network
idim = 10;  % input dimension (number of lagged values used as input)
odim = length(relNums) - idim;  % output dimension (number of predicted values)
x = zeros(odim, idim);  % input data matrix
y = zeros(odim, 1); % target output vector

% create inputs and desired outputs
for i = 1:odim
    y(i) = relNums(i + idim);   % target output is the next value after the input sequence
    x(i, :) = relNums(i:i+idim-1)'; % input sequence is a column vector of lagged values
end

% network parameters
NIPT = idim;    % number of input neurons
NHID = 5;   % number of hidden neurons
NOUT = 1;   % number of output neurons
lr = 0.01;  % learning rate

% activation functions and derivatives
leaky_relu = @(x) (x > 0) .* x + (x <= 0) .* 0.01 .* x; % leaky ReLU activation function
leaky_relu_derivative = @(x) (x > 0) + (x <= 0) * 0.01; % derivative of leaky ReLU
linear_activation = @(x) x; % linear activation function for output layer

% initialize weights randomly
weights_IH = randn(NIPT, NHID); % weights from input to hidden layer
weights_HO = randn(NHID, NOUT); % weights from hidden to output layer

% neural network forward and backward propagation
```

```
network_output = zeros(odim, 1);    % predicted output values
num_epochs = 100;   % number of training epochs

% main training loop
for epoch = 1:num_epochs
    for i = 1:odim

        % feedforward
        hidden_input = x(i, :) * weights_IH;    % input to hidden layer
        hidden_output = leaky_relu(hidden_input);   % output of hidden layer
        output_input = hidden_output * weights_HO;  % input to output layer
        network_output(i) = linear_activation(output_input);    % output of the network

        % combine weight gradients between different layers to a vector
        [loss_value, grad] = compute_loss(weights_IH, weights_HO, x(i, :), y(i), NIPT, NHID,
NOUT);

        % gradient clipping
        grad_clip_threshold = 1;     % threshold for gradient clipping
        % extracts the gradient components from the total gradient vector that correspond to
the weights between:
        % the input layer and the hidden layer / the hidden layer and the output layer
        % which are used to update these weights
        grad_weights_IH = grad(1:NIPT*NHID);
        grad_weights_HO = grad(NIPT*NHID+1:end);
        grad_weights_IH = max(min(grad_weights_IH, grad_clip_threshold), -grad_clip_threshold);
% clip gradients for weights_IH
        grad_weights_HO = max(min(grad_weights_HO, grad_clip_threshold), -grad_clip_threshold);
% clip gradients for weights_HO

        % combine weight gradients between different layers to a vector
        grad = [grad_weights_IH; grad_weights_HO];

        % update weights by using BFGS
        weights = [weights_IH(:); weights_HO(:)];   % combine weights into a single vector
        % objective function for BFGS
        obj_fun = @(w) compute_loss(reshape(w(1:NIPT*NHID), NIPT, NHID),
reshape(w(NIPT*NHID+1:end), NHID, NOUT), x(i, :), y(i), NIPT, NHID, NOUT);
        weights = bfgs(obj_fun, weights, grad, 10); % update weights using BFGS

        % rebuild the weight matrix for each layer from the updated weight vector
```

```
        weights_IH = reshape(weights(1:NIPT*NHID), NIPT, NHID); % extract weights for input-hidden layer
        weights_HO = reshape(weights(NIPT*NHID+1:end), NHID, NOUT); % extract weights for hidden-output layer
    end
end

% calculate NMSE
mean_y = mean(y);
nmse_loss = sum((y - network_output).^2) / sum((y - mean_y).^2);    % normalized mean squared error
disp(['NMSE Loss: ', num2str(nmse_loss)]);

% plot results
figure;
plot(1:odim, y, 'b', 'LineWidth', 2);   % plot actual values in blue
hold on;
plot(1:odim, network_output, 'r--', 'LineWidth', 2);    % plot predicted values in red dashed line
hold off;
xlabel('Time Step');
ylabel('Normalized Sunspot Number');
legend('Actual Data', 'Predicted Data');
title('Sunspot Series: Actual vs. Predicted');
grid on;

% loss calculation function
function [loss_value, grad] = compute_loss(weights_IH, weights_HO, x, y, NIPT, NHID, NOUT)
    leaky_relu = @(x) (x > 0) .* x + (x <= 0) .* 0.01 .* x; % leaky ReLU activation function
    leaky_relu_derivative = @(x) (x > 0) + (x <= 0) * 0.01; % derivative of leaky ReLU
    linear_activation = @(z) z; % linear activation function for output layer

    % feedforward
    hidden_input = x * weights_IH;  % input to hidden layer
    hidden_output = leaky_relu(hidden_input);   % output of hidden layer
    output_input = hidden_output * weights_HO;  % input to output layer
    network_output = linear_activation(output_input);   % output of the network

    % calculate MSE
    loss_value = mean((y - network_output).^2);

    % Backpropagation
```

```
    % calculate the error of the output layer
    delta_output = -(y - network_output);
    % calculate the error of the hidden layer by the error of backpropagation of the output
layer
    delta_hidden = (delta_output * weights_HO') .* leaky_relu_derivative(hidden_input);
    % calculate the gradient from the input to the hidden layer weight by the hidden layer
error
    grad_weights_IH = x' * delta_hidden;
    % calculate the gradient from the hidden to the output layer weight by the error of the
output layer
    grad_weights_HO = hidden_output' * delta_output;

    % combine weight gradients between different layers to a vector
    grad = [grad_weights_IH(:); grad_weights_HO(:)];
end

% BFGS (Broyden-Fletcher-Goldfarb-Shanno) function
function weights = bfgs(f, weights, grad, max_iter)
    n = numel(weights); % get the number of elements of the weight vector
    H = eye(n); % initialize the approximate Hessian matrix to an identity matrix

    for iter = 1:max_iter   % maximum iterations
        d = -H * grad;  % calculate search direction
        step_size = line_search(f, weights, d, grad); % use linear search to determine the
appropriate step size
        weights_new = weights + step_size * d;  % update weight

        [loss_new, grad_new] = f(weights_new);  % calculate the loss and gradient of the new
weights
        s = weights_new - weights;  % weight change amount
        y = grad_new - grad;    % gradient change amount

        if all(abs(y) < 1e-8) % convergence check for whether the gradient change is too small
toend the iteration early
            break;  % exit the loop if convergence is reached
        end

        rho = 1 / (y' * s); % calculate the scaling factor
        % update the approximate Hessian matrix
        H = (eye(n) - rho * (s * y')) * H * (eye(n) - rho * (y * s')) + rho * (s * s');

        weights = weights_new;  % save the new weights
```

```
        grad = grad_new;    % update gradient
    end
end

% Linear search function
function step_size = line_search(f, x, d, grad)
    alpha = 0.1;    % initial step size
    beta = 0.5; % step size reduction factor
    t = 1;  % set initial trial step size as 1

    % determine the step size is small enough to ensure "f" decreases
    % engough along the direction of the gradient descent "d"
    while f(x + t*d) > f(x) + alpha*t*grad'*d
        t = beta * t;   % reduce step size
    end

    step_size = t;  % determine the step size
end
```

Backpropagation (Gradient Descent)

```
clear all;
clc;

% Backpropagation (Gradient Descent)

% load and preprocess(normalize the data) the sunspot dataset
load sunspot.dat
% the first column is the year, stored in the variable year
% the second column is the associated sunspot number, stored in the variable relNums.
year=sunspot(:,1); relNums=sunspot(:,2);
% calculate the mean (ynrmv) and standard deviation (sigy) of relNums.
% the mean is for subsequent centering (demeaning), and the standard deviation is for
subsequent normalization.
ynrmv=mean(relNums(:)); sigy=std(relNums(:));
% using calculated the mean and standard deviationin in the previous step
% to normalize the raw sunspot data by subtracting the mean and
% dividing by the standard deviation from each data point.
nrmY=(relNums(:)-ynrmv)./sigy;
% calculate the minimum (ymin) and maximum (ymax) values of the normalized data nrmY,
% then applied to linearly transform the data into the range [-1, 1].
ymin=min(nrmY(:)); ymax=max(nrmY(:));
% the normalized data nrmY is further transformed into the range [-1, 1].
```

```
% first normalize each value of nrmY to [0, 1] by subtracting ymin and
% dividing by the range (ymax - ymin),
% then adjust the range to [-1, 1] by multiplying by 2.0 and subtracting 0.5.
relNums=2.0*((nrmY-ymin)/(ymax-ymin)-0.5);

% create the data matrix with lagged values
odim = length(relNums) - idim;  % output dimension (number of predicted values)
x = zeros(odim, idim);  % input data matrix
y = zeros(odim, 1); % target output vector

% create inputs and desired outputs
for i = 1:odim
    y(i) = relNums(i + idim);   % target output is the next value after the input sequence
    x(i, :) = relNums(i:i+idim-1)'; % input sequence is a column vector of lagged values
end

% neural network parameters
idim = 10;  % number of input neurons
NHID = 5;   % number of hidden neurons
NOUT = 1;   % number of output neurons
lr = 0.01;  % learning rate

% define activation functions
leaky_relu = @(x) (x > 0) .* x + (x <= 0) .* 0.01 .* x; leaky ReLU activation function
leaky_relu_derivative = @(x) (x > 0) + (x <= 0) * 0.01; % derivative of leaky ReLU
linear_activation = @(x) x; % linear activation function for output layer

% initialize weights randomly
weights_IH = randn(idim, NHID); % weights from input to hidden layer
weights_HO = randn(NHID, NOUT); % weights from hidden to output layer

% neural network forward and backward propagation
network_output = zeros(odim, 1);    % predicted output values
num_epochs = 100;   % number of training epochs

% main training loop
for epoch = 1:num_epochs
    for i = 1:odim

        % feedforward
        hidden_input = x(i, :) * weights_IH;    % input to hidden layer
        hidden_output = leaky_relu(hidden_input);   % output of hidden layer
```

```
        output_input = hidden_output * weights_HO;  % input to output layer
        network_output(i) = linear_activation(output_input);    % output of the network

        % backpropagation
        delta_output = -(y(i) - network_output(i)); % output layer error
        delta_hidden = (delta_output * weights_HO') .* leaky_relu_derivative(hidden_input); %
hidden layer error
        grad_weights_IH = x(i, :)' * delta_hidden;  % gradient of weights from input to hidden
layer
        grad_weights_HO = hidden_output' * delta_output;    % gradient of weights from hidden to
output layer

        % gradient clipping
        grad_clip_threshold = 1;    % threshold for gradient clipping
        grad_weights_IH = max(min(grad_weights_IH, grad_clip_threshold), -grad_clip_threshold);
% clip gradients for weights_IH
        grad_weights_HO = max(min(grad_weights_HO, grad_clip_threshold), -grad_clip_threshold);
% clip gradients for weights_HO

        % update weights (gradient descent)
        weights_IH = weights_IH - lr * grad_weights_IH; % update weights_IO
        weights_HO = weights_HO - lr * grad_weights_HO; % update weights_HO
    end
end

% calculate and display NMSE loss
mean_y = mean(y);
nmse_loss = sum((y - network_output).^2) / sum((y - mean_y).^2);    % normalized mean squared
error
disp(['NMSE Loss: ', num2str(nmse_loss)]);

% plot the actual vs predicted data
figure;
plot(1:length(y), y, 'b', 'LineWidth', 2);  % plot actual values in blue
hold on;
plot(1:length(network_output), network_output, 'r--', 'LineWidth', 2);  % plot predicted values
in red dashed line
hold off;
title('Sunspot Series: Actual vs. Predicted');
xlabel('Time Steps');
ylabel('Normalized Sunspot Number');
legend('Actual', 'Predicted');
```