EE5904/ME5404 Neural Networks: Homework #2
XU NUO, A0313771H

Q1. Rosenbrock's Valley Problem

Consider the Rosenbrock's Valley function:

$$f(x,y) = (1-x)^2 + 100(y-x^2)^2$$

a). Show that it has a global minimum at $(x, y) = (1,1)$ where $f(x,y) = 0$.

Now suppose the starting point is randomly initialized in the open interval $(-1,1)$ for x and y, find the global minimum using:

b). Steepest (Gradient) descent method

$$w(k+1) = w(k) - \eta g(k)$$

with learning rate $\eta = 0.001$. Here the weight vector refers to the two-dimensional vector $[x, y]$. Record the number of iterations when $f(x,y)$ converges to (or very close to) 0 and plot out the trajectory of (x,y) in the 2-dimensional space. Also plot out the function value as it approaches the global minimum. What would happen if a larger learning rate, say $\eta = 1.0$, is used?

c). Newton's method (as discussed on page 13 in the slides of lecture Four)

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

Record the number of iterations when $f(x,y)$ converges to (or very close to) 0 and plot out the trajectory of (x,y) in the 2-dimensional space. Also plot out the function value as it approaches the global minimum.

A1.

a). First, let the first-order partial derivatives of $f(x,y)$ with respect to x and y be 0 respectively.

$$\frac{\partial f(x,y)}{\partial x} = 400x^3 - 400xy + 2x - 2 = 0 \qquad (1)$$

$$\frac{\partial f(x,y)}{\partial y} = 200y - 200x^2 = 0 \qquad (2)$$

(x,y) that conform to the above two equations are all candidate extreme points of the function. From (2) we can get $y = x^2$. Substituting $y = x^2$ into (1) gives us $(x,y) = (1,1)$. Therefore $(1,1)$ is the only extreme point of the function. The sum of two square terms constitutes the function so obviously $f(x,y) \geq 0$. $f(x,y) \geq f(1,1) = 0$ shows that it has a global minimum at $(x, y) = (1,1)$ where $f(x,y) = 0$.

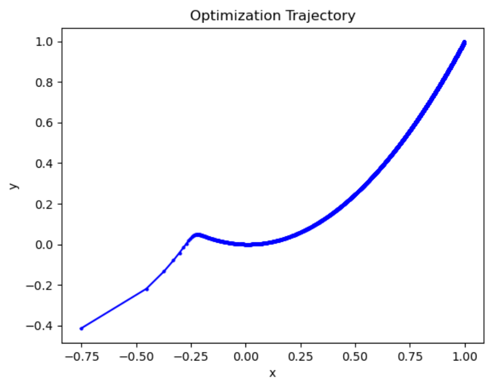
b). I randomly initialize x and y in a strictly open interval of $(-1,1)$ and define the Rosenbrock function and its partial derivative. The learning rate is set to 0.001 and the number of iterations is 100000. The threshold is set to 0.001. When the L2 norm of the gradient is less than the threshold, the function is considered to have basically converged and reached the global minimum. The code is shown below:

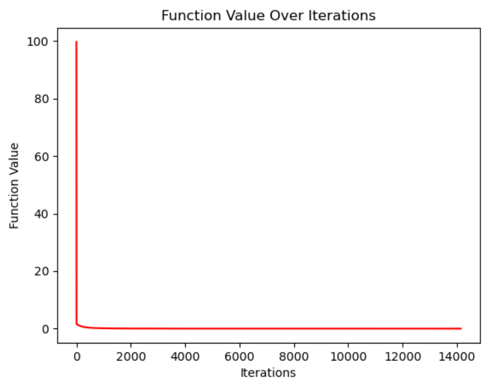
```
# Import necessary libraries
import matplotlib.pyplot as plt
import numpy as np
import random

# Starting point randomly initialized in the open interval (-1,1)
while True:
    x_init = random.uniform(-1, 1)
```

```python
    y_init = random.uniform(-1, 1)
    if (x_init != -1 and x_init != 1) or (y_init != -1 and y_init != 1):
        break

# Define Rosenbrock function
def rosenbrock(x, y):
    return (1-x) ** 2 + 100 * (y - x ** 2) **2

# Define the derivative function of Rosenbrock
def rosenbrock_gradient(x, y):
    dfx = 400 * x ** 3 - 400 * x * y + 2 * x - 2
    dfy = 200 * y - 200 * x ** 2
    return np.array([dfx, dfy])

# Define hyperparameters
xy_trajectory = []
function_values = []
learning_rate = 0.001
epochs = 100000
threshold = 0.001
x_update = x_init
y_update = y_init

# Find the minimum value of the rosenbrock function by steepest gradient descent method
for i in range(epochs):
    xy_trajectory.append((x_update, y_update))
    function_values.append(rosenbrock(x_update,y_update))
    grad = rosenbrock_gradient(x_update, y_update)
    x_update -= learning_rate * grad[0]
    y_update -= learning_rate * grad[1]

    # If the L2 norm of the gradient is less thant the predetermined threshold
    # The function is considered to have basically converged
    if np.linalg.norm(grad) < threshold:
        print(f"Converged at epoch {i+1}")
        break

print(f"Final (x, y): ({x_update:.6f}, {y_update:.6f})")
print(f"Function value at final point: {rosenbrock(x_update, y_update):.6f}")

# Extract x and y values of each interation separately
x_vals, y_vals = zip(*xy_trajectory)

# Plot the trajectory of (x,y)
```

```python
plt.figure()
plt.plot(x_vals, y_vals, marker = "o", linestyle = "-", color = "b", markersize = 2)
plt.xlabel("x")
plt.ylabel("y")
plt.title("Optimization Trajectory")
plt.show(block = False)

# Plot the function value
plt.figure()
plt.plot(range(len(function_values)), function_values, color = "r")
plt.xlabel("Iterations")
plt.ylabel("Function Value")
plt.title("Function Value Over Iterations")
plt.show()
```

```
Converged at epoch 14141
Final (x, y): (0.998884, 0.997764)
Function value at final point: 0.000001
```





The function converged at epoch 14141 at $(0.998884, 0.997764)$ when threshold is set to 0.001. The function value at this point is 0.000001. We can basically determine that the global minimum point of Rosenbrock function is at $(1,1)$ and the function value is 0.

When the learning rate is too high, such as when it is adjusted to 1, the program will prompt error messages as shown below:

```
RuntimeWarning: overflow encountered in scalar power
RuntimeWarning: overflow encountered in scalar multiply
RuntimeWarning: invalid value encountered in scalar subtract
Final (x, y): (nan, nan)
Function value at final point: nan
```

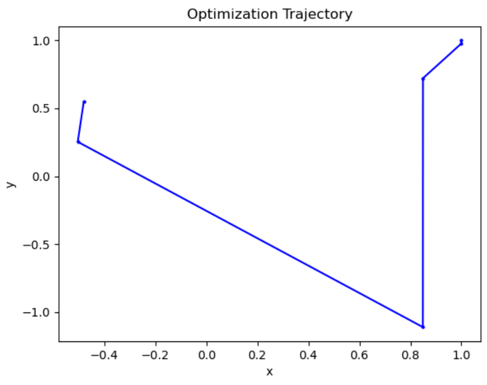
This issue arises because the Rosenbrock function exhibits steep gradient variations. When combined with an excessively large learning rate, the values of x and y extreme updates over multiple iterations, leading to numerical instability. As a result, the values exceed the floating-point range, causing overflow and gradient explosion, rather than gradually converging to the optimal solution.

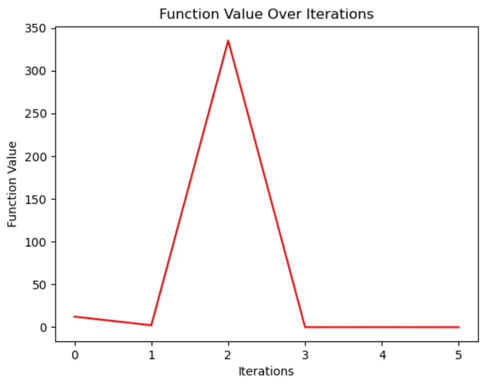
c). The code for finding the global minimum using Newton's method is as follows:

```python
# Import necessary libraries
import matplotlib.pyplot as plt
import numpy as np
import random

# Starting point randomly initialized in the open interval (-1,1)
while True:
    x_init = random.uniform(-1, 1)
    y_init = random.uniform(-1, 1)
    if (x_init != -1 and x_init != 1) or (y_init != -1 and y_init != 1):
        break

# Define Rosenbrock function
def rosenbrock(x, y):
    return (1-x) ** 2 + 100 * (y - x ** 2) **2

# Define the derivative function of Rosenbrock
def rosenbrock_gradient(x, y):
    dfx = 400 * x ** 3 - 400 * x * y + 2 * x - 2
    dfy = 200 * y - 200 * x ** 2
    return np.array([dfx, dfy])

# Define the Hessian Matrix for Rosenbrock
def rosenbrock_hessian(x, y):
    f_xx = 1200 * x ** 2 - 400 * y + 2
    f_xy = -400 * x
    f_yx = -400 * x
    f_yy = 200
    return np.array([[f_xx, f_xy],
                     [f_yx, f_yy]])

# Define hyperparameters
xy_trajectory = []
function_values = []
epochs = 100
threshold = 0.001
x_update = x_init
y_update = y_init

# Find the minimum value of the rosenbrock function by Newton's Method
```

```python
for i in range(epochs):
    xy_trajectory.append((x_update, y_update))
    function_values.append(rosenbrock(x_update,y_update))
    grad = rosenbrock_gradient(x_update, y_update)
    hessian = rosenbrock_hessian(x_update, y_update)
    delta_w = np.linalg.inv(hessian) @ grad
    x_update -= delta_w[0]
    y_update -= delta_w[1]

    # If the L2 norm of the gradient is less thant the predetermined threshold
    # The function is considered to have basically converged
    if np.linalg.norm(grad) < threshold:
        print(f"Converged at epoch {i+1}")
        break

print(f"Final (x, y): ({x_update:.6f}, {y_update:.6f})")
print(f"Function value at final point: {rosenbrock(x_update, y_update):.6f}")

# Extract x and y values of each interation separately
x_vals, y_vals = zip(*xy_trajectory)

# Plot the trajectory of (x,y)
plt.figure()
plt.plot(x_vals, y_vals, marker = "o", linestyle = "-", color = "b", markersize = 2)
plt.xlabel("x")
plt.ylabel("y")
plt.title("Optimization Trajectory")
plt.show(block = False)

# Plot the function value
plt.figure()
plt.plot(range(len(function_values)), function_values, color = "r")
plt.xlabel("Iterations")
plt.ylabel("Function Value")
plt.title("Function Value Over Iterations")
plt.show()
```

```
Converged at epoch 6
Final (x, y): (1.000000, 1.000000)
Function value at final point: 0.000000
```





The function converged at epoch 6 at $(1,1)$ when threshold is set to 0.001. The function value at this point is 0. Using Newton's method to find the global minimum point requires defining the Hessian matrix of the Rosenbrock function and multiplying its inverse matrix with the matrix composed of gradients to obtain the updated values of x and y. The Hessian matrix can automatically adjust the direction and step size, so there will be no oscillation and no need for a learning rate. And because the first-order and second-order derivatives are used at the same time, the convergence speed is quadratic, and the error is reduced by the square of each iteration, so the function converges extremely quickly, but the need to calculate the second-order derivative also increases the amount of calculation for each step.

Q2. Function Approximation

Consider using MLP to approximate the following function:

$$y = 1.2\sin(\pi x) - \cos(2.4\pi x) \qquad for\ x \in [-1.6, 1.6]$$

The training set is generated by dividing the domain $[-1.6,1.6]$ using a uniform step length 0.05, while the test set is constructed by dividing the domain $[-1.6,1.6]$ using a uniform step length 0.01. You may use the MATLAB deep learning toolbox to implement a MLP (see the Appendix for guidance) and do the following experiments:

a): Use the sequential mode with BP algorithm and experiment with the following different structures of the MLP: 1-n-1 (where n = 1, 2, ..., 10, 20, 50, 100). For each architecture plot out the outputs of the MLP for the test samples after training and compare them to the desired outputs. Try to determine whether it is under-fitting, proper fitting or over-fitting. Identify the minimal number of hidden neurons from the experiments, and check if the result is consistent with the guideline given in the lecture slides. Compute the outputs of the MLP when $x = -3$ and $+3$, and see if the MLP can make reasonable predictions outside of the domain of the input limited by the training set.

b). Use the batch mode with *trainlm* algorithm to repeat the above procedure.

c). Use the batch mode with *trainbr* algorithm to repeat the above procedure.

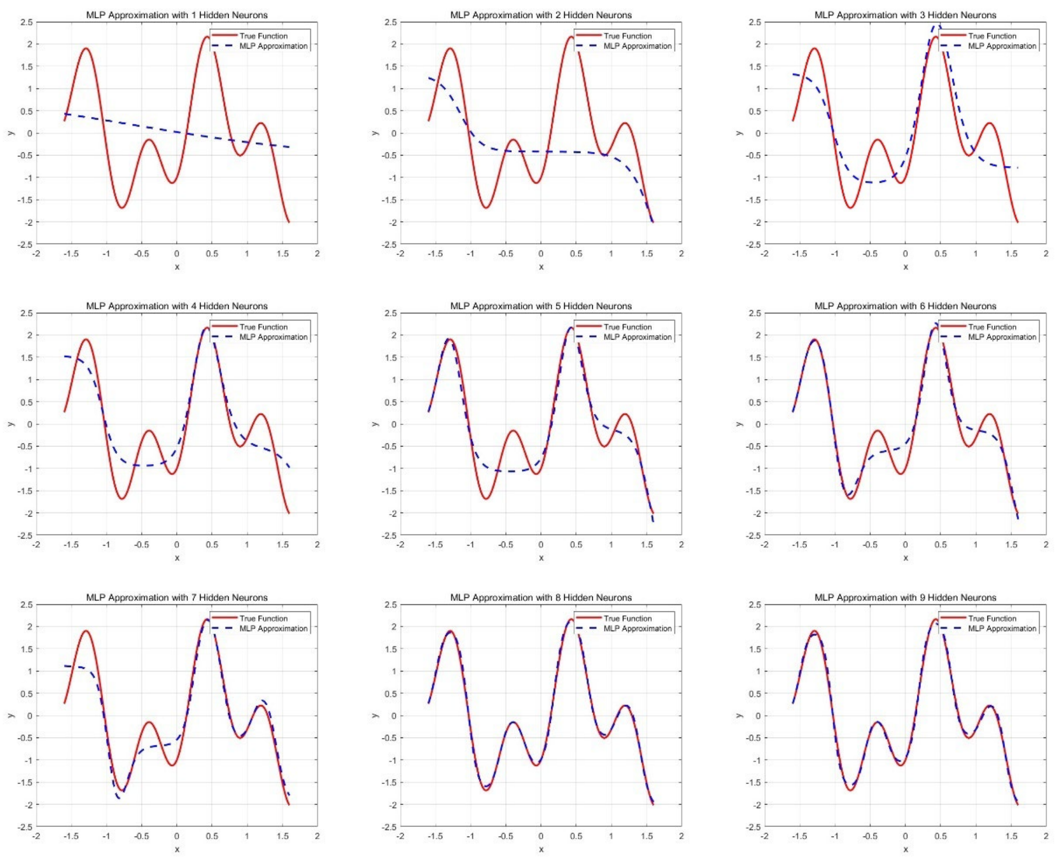
A2.

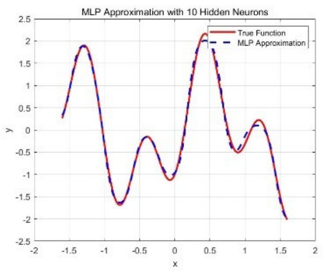
a). I created a 1-n-1 MLP using *feedforward* function in MATLAB. For the parameters, I chose *traingd* for the most common gradient descent backpropagation, *mse* as the loss function, 0.01 as the learning rate, and trained the model for 500 epochs. The training was done in a sample-by-sample sequential mode, training one sample at a time and updating the MLP using *adapt* function. The code is shown below:

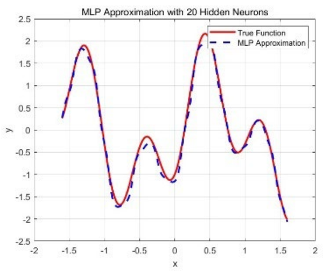
```
clc;
```

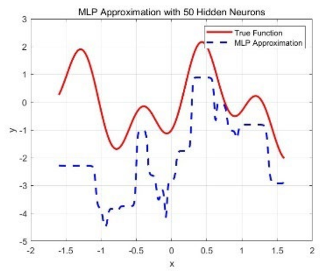
```
clear;
close all;

% Generate data for training and testing
x_train = -1.6 : 0.05 : 1.6;
y_train = 1.2 * sin(pi * x_train) - cos(2.4 * pi * x_train);
x_test = -1.6 : 0.01 : 1.6;
y_test = 1.2 * sin(pi * x_test) - cos(2.4 * pi * x_test);

% The number of hidden neurons
hidden_neurons = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 50, 100];

% Epochs set to 500 for training
epochs = 500;

for i = 1:length(hidden_neurons)

    % Convert from numeric arrays to cell arrays
    % Used by the adapt function for sequential mode training.
    x_cell = num2cell(x_train, 1);
    y_cell = num2cell(y_train, 1);

    n = hidden_neurons(i); % One hidden layer with n neurons
    net = feedforwardnet(n); % Define an MLP with n hidden neurons and SISO
    net.trainFcn = 'traingd'; % Regular gradient descent BP as training function
    net.performFcn = 'mse'; % Mean Square Error as performance function
    net.trainParam.lr = 0.01; % Learning rate set to 0.01
    net.trainParam.epochs = epochs; % Epochs set to 500

    % Sequential mode training
    for j = 1:epochs
        idx = randperm(length(x_train)); % Shuffle the order of training data to improve
generalization ability
        net = adapt(net, x_cell(:,idx), y_cell(:,idx)); % Sample by sample training
    end

    % Using the trained MLP to predict the test set
    y_pred = net(x_test);

    % Plot the original function and the function inferred by the MLP
    figure;
    plot(x_test, y_test, 'r', 'LineWidth', 2);
    hold on;
    plot(x_test, y_pred, 'b--', 'LineWidth', 2);
    legend('True Function', 'MLP Approximation');
    xlabel('x');
    ylabel('y');
    title(['MLP Approximation with ', num2str(n), ' Hidden Neurons']);
    grid on;

    % Preidictions out of the training set
    x_extra = [-3, 3];
    y_extra_pred = net(x_extra);

    disp(['Hidden Neurons: ', num2str(n)]);
    disp('Ground truth for x=-3: 0.8090')
    disp(['MLP Prediction for x=-3: ', num2str(y_extra_pred(1))]);
    disp('Ground truth for x=3: 0.8090')
    disp(['MLP Prediction for x=3: ', num2str(y_extra_pred(2))]);
    disp('--------------------------------');
end
```

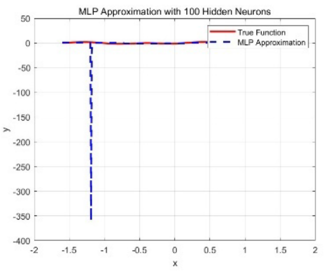
When the number of hidden layer neurons is different, the degree of MLP's fitting of the original function is shown in the following figures:







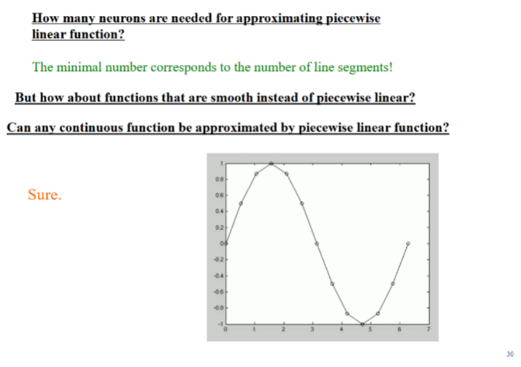




By observing the degree of fit of the original function and the MLP approximated function in the figures, we can decide whether the model is under-fitting, proper-fitting, or over-fitting:

Degree of Fit	Under-fitting									Proper-fitting			
Number of Hidden Neurons	1	2	3	4	5	6	7	50	100	8	9	10	20

From the table, we can see that the minimum number of hidden layer neurons required for proper-fitting is 8. The result is consistent with the guideline given in the lecture slide as shown below:



How many neurons are needed for approximating piecewise linear function?

The minimal number corresponds to the number of line segments!

But how about functions that are smooth instead of piecewise linear?

Can any continuous function be approximated by piecewise linear function?

Sure.

30

Sequential mode updates parameters sample by sample, which makes convergence difficult, especially when n=50 or n=100 which far exceeds the data complexity. The impact of each sample's gradient update on the weight may be amplified. In addition, as can be seen from the figures, the predicted model does not overlap too much with the original function, so I think that n=50 and n=100 make the model underfit rather than overfit.

When the number of neurons in the hidden layer is different, the predicted output of MLP when x=-3 and x=3 are as follows:

```
Hidden Neurons: 1
```

```
Ground truth for x=-3: 0.8090
MLP Prediction for x=-3: 0.51851
Ground truth for x=3: 0.8090
MLP Prediction for x=3: -0.19595
---------------------------------
Hidden Neurons: 2
Ground truth for x=-3: 0.8090
MLP Prediction for x=-3: 1.2323
Ground truth for x=3: 0.8090
MLP Prediction for x=3: -4.1242
---------------------------------
Hidden Neurons: 3
Ground truth for x=-3: 0.8090
MLP Prediction for x=-3: 1.7472
Ground truth for x=3: 0.8090
MLP Prediction for x=3: -0.89462
---------------------------------
Hidden Neurons: 4
Ground truth for x=-3: 0.8090
MLP Prediction for x=-3: 0.80332
Ground truth for x=3: 0.8090
MLP Prediction for x=3: -0.41676
---------------------------------
Hidden Neurons: 5
Ground truth for x=-3: 0.8090
MLP Prediction for x=-3: -0.30949
Ground truth for x=3: 0.8090
MLP Prediction for x=3: -3.8367
---------------------------------
Hidden Neurons: 6
Ground truth for x=-3: 0.8090
MLP Prediction for x=-3: -0.38681
Ground truth for x=3: 0.8090
MLP Prediction for x=3: -3.096
---------------------------------
Hidden Neurons: 7
Ground truth for x=-3: 0.8090
MLP Prediction for x=-3: -0.28911
Ground truth for x=3: 0.8090
MLP Prediction for x=3: -2.0517
---------------------------------
Hidden Neurons: 8
Ground truth for x=-3: 0.8090
MLP Prediction for x=-3: -0.72655
```

```
Ground truth for x=3: 0.8090
MLP Prediction for x=3: -2.1348
---------------------------------
Hidden Neurons: 9
Ground truth for x=-3: 0.8090
MLP Prediction for x=-3: -0.049021
Ground truth for x=3: 0.8090
MLP Prediction for x=3: -2.0468
---------------------------------
Hidden Neurons: 10
Ground truth for x=-3: 0.8090
MLP Prediction for x=-3: 2.5793
Ground truth for x=3: 0.8090
MLP Prediction for x=3: -1.975
---------------------------------
Hidden Neurons: 20
Ground truth for x=-3: 0.8090
MLP Prediction for x=-3: 0.0059098
Ground truth for x=3: 0.8090
MLP Prediction for x=3: -0.3874
---------------------------------
Hidden Neurons: 50
Ground truth for x=-3: 0.8090
MLP Prediction for x=-3: -45.4695
Ground truth for x=3: 0.8090
MLP Prediction for x=3: 129.9262
---------------------------------
Hidden Neurons: 100
Ground truth for x=-3: 0.8090
MLP Prediction for x=-3: 2458.2731
Ground truth for x=3: 0.8090
MLP Prediction for x=3: -1113.6718
---------------------------------
```

From the output of the program, we can see that the model cannot make reasonable predictions outside the domain of the input limited by the training set. This is because the model prefers linear predictions outside the range of the training set and makes poor predictions for nonlinear transformations.

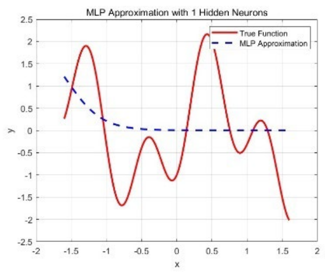
b). The code for training the MLP with *trainlm* using the batch mode is shown below:

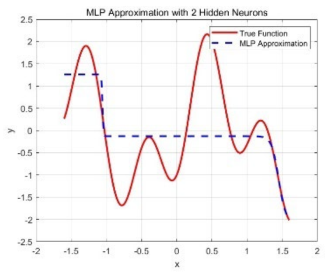
```
clc;
clear;
close all;
```

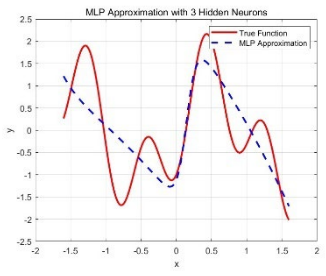
```
% Generate data for training and testing
x_train = -1.6 : 0.05 : 1.6;
y_train = 1.2 * sin(pi * x_train) - cos(2.4 * pi * x_train);
x_test = -1.6 : 0.01 : 1.6;
y_test = 1.2 * sin(pi * x_test) - cos(2.4 * pi * x_test);

% The number of hidden neurons
hidden_neurons = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 50, 100];

for i = 1:length(hidden_neurons)
    n = hidden_neurons(i); % One hidden layer with n neurons

    net = feedforwardnet(n); % Define an MLP with n hidden neurons and SISO
    net.trainFcn = 'trainlm'; % Levenberg-Marquardt BP as training function, no learning
rate involved
    net.performFcn = 'mse'; % Mean Square Error as performance function

    % Train the MLP
    net = train(net, x_train, y_train); % Default epochs set to 1000 and use batch mode

    % Using the trained MLP to predict the test set
    y_pred = net(x_test);

    % Plot the original function and the function inferred by the MLP
    figure;
    plot(x_test, y_test, 'r', 'LineWidth', 2);
    hold on;
    plot(x_test, y_pred, 'b--', 'LineWidth', 2);
    legend('True Function', 'MLP Approximation');
    xlabel('x');
    ylabel('y');
    title(['MLP Approximation with ', num2str(n), ' Hidden Neurons']);
    grid on;

    % Preidictions out of the training set
    x_extra = [-3, 3];
    y_extra_pred = net(x_extra);

    disp(['Hidden Neurons: ', num2str(n)]);
    disp('Ground truth for x=-3: 0.8090')
    disp(['MLP Prediction for x=-3: ', num2str(y_extra_pred(1))]);
    disp('Ground truth for x=3: 0.8090')
    disp(['MLP Prediction for x=3: ', num2str(y_extra_pred(2))]);
    disp('---------------------------------');
```

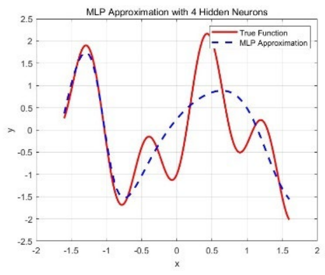
```
end
```

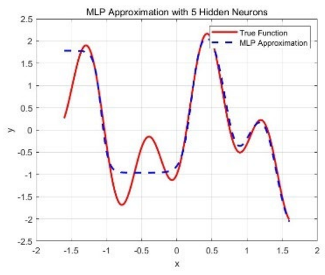
When the number of hidden layer neurons is different, the degree of MLP's fitting of the original function is shown in the following figures:

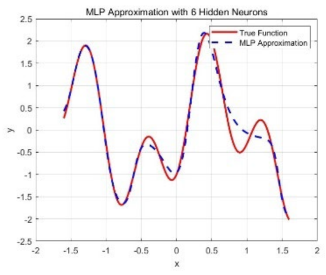


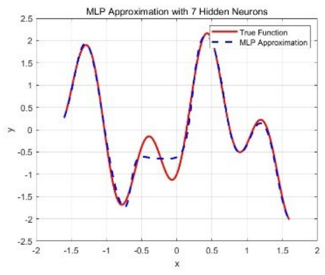


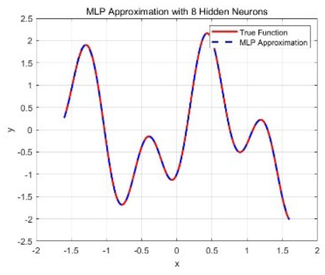


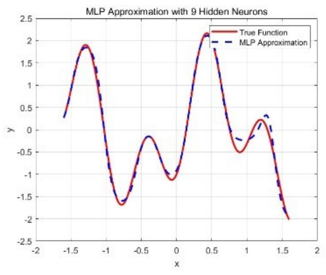


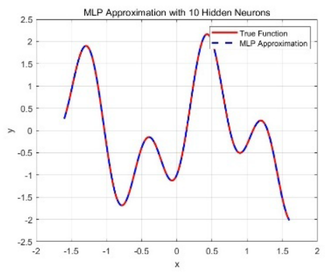


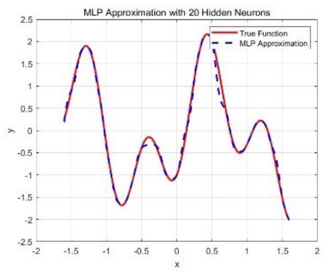


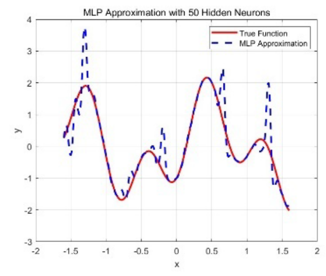


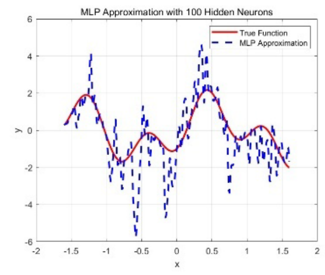












By observing the degree of fit of the original function and the MLP approximated function in the figures, we can decide whether the model is under-fitting, proper-fitting, or over-fitting:

Degree of Fit	Under-fitting							Proper-fitting				Over-fitting	
Number of Hidden Neurons	1	2	3	4	5	6	7	8	9	10	20	50	100

From the table, we can see that the minimum number of hidden layer neurons required for proper-fitting is still 8.

When the number of neurons in the hidden layer is different, the predicted output of MLP when x=-3 and x=3 are as follows:

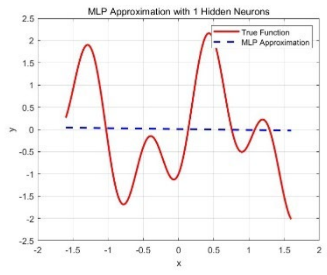
```
Hidden Neurons: 1
Ground truth for x=-3: 0.8090
MLP Prediction for x=-3: 2.2583
Ground truth for x=3: 0.8090
MLP Prediction for x=3: 0.0025707
---------------------------------
Hidden Neurons: 2
Ground truth for x=-3: 0.8090
MLP Prediction for x=-3: 1.26
Ground truth for x=3: 0.8090
MLP Prediction for x=3: -2.0644
---------------------------------
Hidden Neurons: 3
Ground truth for x=-3: 0.8090
MLP Prediction for x=-3: 19.4167
Ground truth for x=3: 0.8090
MLP Prediction for x=3: -7.4566
---------------------------------
Hidden Neurons: 4
Ground truth for x=-3: 0.8090
MLP Prediction for x=-3: -11.0854
Ground truth for x=3: 0.8090
MLP Prediction for x=3: -1.9794
---------------------------------
Hidden Neurons: 5
Ground truth for x=-3: 0.8090
MLP Prediction for x=-3: 1.7841
Ground truth for x=3: 0.8090
MLP Prediction for x=3: -2.4057
---------------------------------
Hidden Neurons: 6
Ground truth for x=-3: 0.8090
MLP Prediction for x=-3: 0.19935
Ground truth for x=3: 0.8090
MLP Prediction for x=3: -2.0201
---------------------------------
```

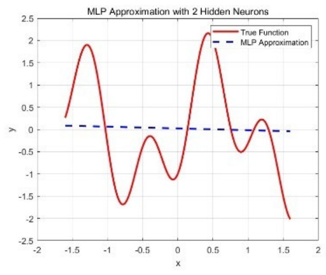
```
Hidden Neurons: 7
Ground truth for x=-3: 0.8090
MLP Prediction for x=-3: -0.12433
Ground truth for x=3: 0.8090
MLP Prediction for x=3: -2.2871
---------------------------------
Hidden Neurons: 8
Ground truth for x=-3: 0.8090
MLP Prediction for x=-3: -0.27241
Ground truth for x=3: 0.8090
MLP Prediction for x=3: -2.3679
---------------------------------
Hidden Neurons: 9
Ground truth for x=-3: 0.8090
MLP Prediction for x=-3: 0.017797
Ground truth for x=3: 0.8090
MLP Prediction for x=3: -2.0174
---------------------------------
Hidden Neurons: 10
Ground truth for x=-3: 0.8090
MLP Prediction for x=-3: -0.37373
Ground truth for x=3: 0.8090
MLP Prediction for x=3: -2.3309
---------------------------------
Hidden Neurons: 20
Ground truth for x=-3: 0.8090
MLP Prediction for x=-3: -0.51566
Ground truth for x=3: 0.8090
MLP Prediction for x=3: -2.113
---------------------------------
Hidden Neurons: 50
Ground truth for x=-3: 0.8090
MLP Prediction for x=-3: -0.083564
Ground truth for x=3: 0.8090
MLP Prediction for x=3: 0.4076
---------------------------------
Hidden Neurons: 100
Ground truth for x=-3: 0.8090
MLP Prediction for x=-3: 0.21512
Ground truth for x=3: 0.8090
MLP Prediction for x=3: -2.2767
---------------------------------
```

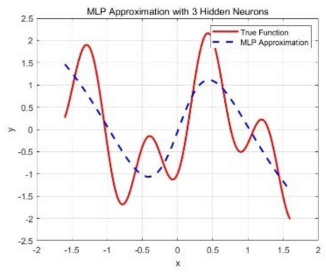
From the output of the program, we can see that the model still cannot make reasonable predictions outside the domain of the input limited by the training set.

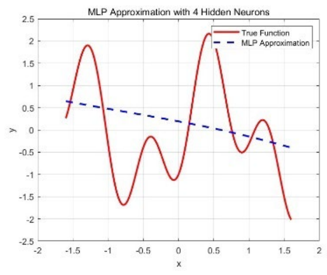
c). Because the code only needs to change the backpropagation algorithm from *trainlm* to *trainbr*, the code is not shown for brevity.

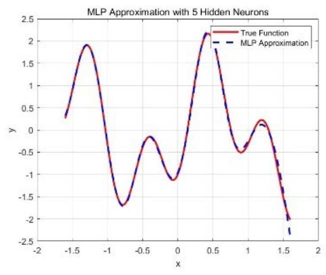
When the number of hidden layer neurons is different, the degree of MLP's fitting of the original function is shown in the following figures:

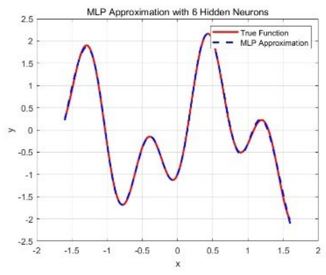


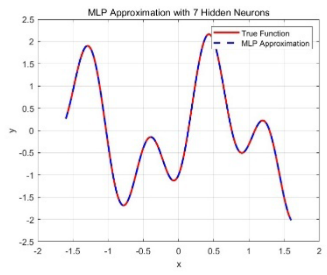


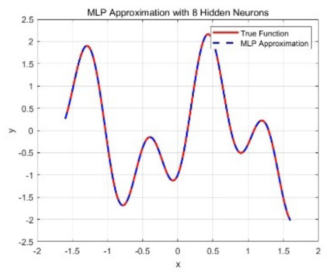


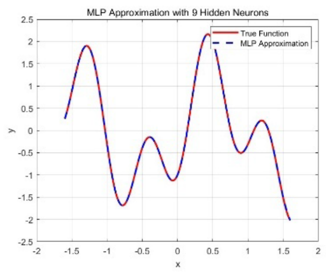


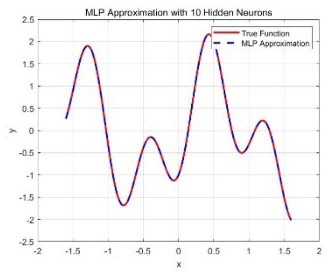


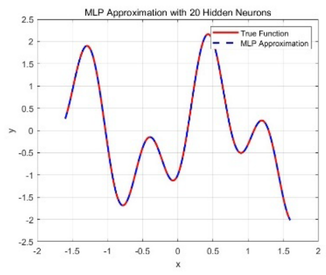


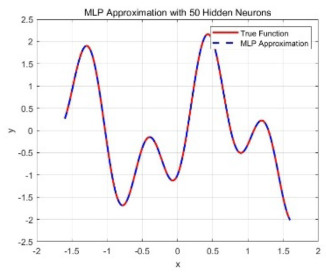


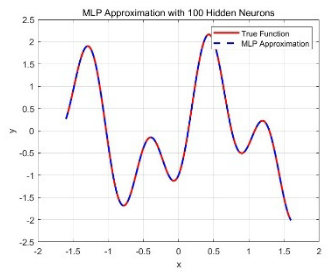












By observing the degree of fit of the original function and the MLP approximated function in the figures, we can decide whether the model is under-fitting, proper-fitting, or over-fitting:

Degree of Fit	Under-fitting				Proper-fitting								
Number of Hidden Neurons	1	2	3	4	5	6	7	8	9	10	20	50	100

From the table, we can see that the minimum number of hidden layer neurons required for proper-fitting is 5. I think it is because the Bayesian regularization property of the *trainbr* algorithm enables it to automatically adjust the complexity of the network, thereby achieving good fitting results even with fewer hidden neurons. It's worth noting that n=50 and n=100 still make the MLP properly fit the function. It's because *trainbr* uses Bayesian Regularization, it automatically adjusts the number of effective parameters of the network, so even if n is large, not all neurons are involved in fitting. Besides, it balances MSE and the weight size. They all contribute to avoiding overfitting.

When the number of neurons in the hidden layer is different, the predicted output of MLP when x=-3 and x=3 are as follows:

```
Hidden Neurons: 1
Ground truth for x=-3: 0.8090
MLP Prediction for x=-3: 0.069324
Ground truth for x=3: 0.8090
MLP Prediction for x=3: -0.049812
---------------------------------
Hidden Neurons: 2
Ground truth for x=-3: 0.8090
MLP Prediction for x=-3: 0.13897
Ground truth for x=3: 0.8090
MLP Prediction for x=3: -0.095833
---------------------------------
Hidden Neurons: 3
Ground truth for x=-3: 0.8090
MLP Prediction for x=-3: 2.8727
Ground truth for x=3: 0.8090
MLP Prediction for x=3: -2.8366
---------------------------------
Hidden Neurons: 4
Ground truth for x=-3: 0.8090
MLP Prediction for x=-3: 1.0464
Ground truth for x=3: 0.8090
MLP Prediction for x=3: -1.0065
---------------------------------
Hidden Neurons: 5
Ground truth for x=-3: 0.8090
MLP Prediction for x=-3: 8.3082
Ground truth for x=3: 0.8090
```

```
MLP Prediction for x=3: -16.2028
---------------------------------
Hidden Neurons: 6
Ground truth for x=-3: 0.8090
MLP Prediction for x=-3: -3.2189
Ground truth for x=3: 0.8090
MLP Prediction for x=3: -4.1539
---------------------------------
Hidden Neurons: 7
Ground truth for x=-3: 0.8090
MLP Prediction for x=-3: -13.006
Ground truth for x=3: 0.8090
MLP Prediction for x=3: 12.7703
---------------------------------
Hidden Neurons: 8
Ground truth for x=-3: 0.8090
MLP Prediction for x=-3: 0.28064
Ground truth for x=3: 0.8090
MLP Prediction for x=3: 3.2699
---------------------------------
Hidden Neurons: 9
Ground truth for x=-3: 0.8090
MLP Prediction for x=-3: 2.1551
Ground truth for x=3: 0.8090
MLP Prediction for x=3: -2.4152
---------------------------------
Hidden Neurons: 10
Ground truth for x=-3: 0.8090
MLP Prediction for x=-3: -0.38695
Ground truth for x=3: 0.8090
MLP Prediction for x=3: 5.0142
---------------------------------
Hidden Neurons: 20
Ground truth for x=-3: 0.8090
MLP Prediction for x=-3: 0.51077
Ground truth for x=3: 0.8090
MLP Prediction for x=3: 2.4324
---------------------------------
Hidden Neurons: 50
Ground truth for x=-3: 0.8090
MLP Prediction for x=-3: -0.0039936
Ground truth for x=3: 0.8090
MLP Prediction for x=3: -1.9679
---------------------------------
```

```
Hidden Neurons: 100
Ground truth for x=-3: 0.8090
MLP Prediction for x=-3: -0.28667
Ground truth for x=3: 0.8090
MLP Prediction for x=3: -1.2572
---------------------------------
```

From the output of the program, we can see that the model still cannot make reasonable predictions outside the domain of the input limited by the training set.

Q3. Image Classification (Mod (71,3) = 2 (deer vs. ship))

Multi-layer perceptron (MLP) can be used to solve real-world pattern recognition problems. In this assignment, MLP will be designed to handle a binary classification task, i.e. animals vs. man-made objects. Specifically, students are divided into 3 groups based on matric numbers and each group is assigned with different dataset as illustrated in the following Table.

Group ID	**Animals** with label **1**	**Man-Made objects** with label **0**
0	cat	airplane
1	dog	automobile
2	deer	ship

You may download the zipped dataset (e.g. group_0.zip) from CANVAS. After unzipping, you will find two folders respectively, one of which is named by an animal and the other of which is named by a man-made object. You should use “0” as the label of man-made objects during training and “1” for animals. In each folder, there are 500 images indexing from 000 to 499. The training set you used should consists of images in each folder indexing from 000 to 449, so there are totally 900 images for training. The remaining images in each folder indexing from 450 to 499 are used as the test set, so there should be totally 100 images in the test set.
All the images are provided in RGB format with size 32*32*3. To simplify the problem, I recommend you transforming colorful images into grayscale, so that each image is in size of 32*32. Besides, you can also reshape the matrix into a one-dimension vector with size 1024. (Hints: Two built-in functions *rgb2gray* and *reshape* would help; Please make full use of the official documentation, you can search all built-in functions from MATLAB official website.)
You are required to complete the following tasks:
a). Apply Rosenblatt’s perceptron (single layer perceptron) to the dataset of your assigned group. After the training procedure, calculate the *classification accuracy* for *both* the training set and validation set, and evaluate the performance of the network.

b). The global mean and variance of a dataset may influence the stability of training and the final performance of the model obtained. You can try to calculate the global mean and variance of the whole dataset, then *subtract* the mean value from each image and *divide* each image by the standard deviation. *Compare* the result with that in a) and *explain* it.

c). Apply MLP to the dataset of your assigned group using *batch mode* training. After the training procedure, calculate the classification accuracy for both the training set and test set, and evaluate the performance of the network.

d). Please determine whether your trained MLP in c) is overfitting. If so, please specify when (i.e. after which training epoch) it becomes overfitting. Try weights regularization and observe if it helps. (you may set the regularization strength by *performParam.regularization*)

e). Apply MLP to the dataset of your assigned group using *sequential mode training*. After the training procedure, calculate the classification accuracy for both training set and test set, and evaluate the performance of the network. Compare the result to part c), and make your recommendation on the two approaches.

f). Try to propose a scheme that you believe could help to improve the performance of your MLP and please explain the reason briefly.

A3.

a). In this part, I set the first 450 images of the two categories as training sets and the last 50 images as test sets, and concatenate them separately. After that, they are converted from 32*32*3 RGB images to 32*32 grayscale images, and then flattened into 1024*1 column vectors. Finally, the training set and test set are completely randomly shuffled to enhance the generalization ability of the model. The model consists of a single neuron and uses 0 and 1 as hard decision criteria. The number of training rounds is 2500, and the learning rate is 0.01. The code is shown below:

```
clc;
clear;
close all;

% Folder path
deer_folder = './deer';
ship_folder = './ship';

% Number of images per class for training
num_train_img = 450;

% Traverse all the pictures in the folder
deer_img = dir(fullfile(deer_folder, '*.jpg'));
ship_img = dir(fullfile(ship_folder, '*.jpg'));

% Create empty cell arrays to store the image paths of the training set and test set
deer_train = {};
ship_train = {};
deer_test = {};
ship_test = {};
```

```
% Separate training set and test set of the deer pictures
for i = 1 : length(deer_img)
    [~, name, ~] = fileparts(deer_img(i).name); % Extract file name
    num = str2double(name); % Convert a file name string to a number
    if num >= 0 && num < num_train_img % Training set
        deer_train{end + 1} = fullfile(deer_folder, deer_img(i).name);
    else % Test set
        deer_test{end + 1} = fullfile(deer_folder, deer_img(i).name);
    end
end

% Separate training set and test set of the ship pictures
for i = 1 : length(ship_img)
    [~, name, ~] = fileparts(ship_img(i).name);
    num = str2double(name);
    if num >= 0 && num < num_train_img
        ship_train{end + 1} = fullfile(ship_folder, ship_img(i).name);
    else
        ship_test{end + 1} = fullfile(ship_folder, ship_img(i).name);
    end
end

% Merge the training and test sets separately and correspond to the correct labels
train_files = [deer_train, ship_train];
test_files = [deer_test, ship_test];
train_labels = [zeros(length(deer_train), 1); ones(length(ship_train), 1)];
test_labels = [zeros(length(deer_test), 1); ones(length(ship_test), 1)];

% Randomly shuffle the training set to improve generalization ability
temp_train = [train_files', num2cell(train_labels)];
temp_train = temp_train(randperm(size(temp_train, 1)), :);
train_files = temp_train(:, 1)';
train_labels = cell2mat(temp_train(:, 2));

% Randomly shuffle the test set
temp_test = [test_files', num2cell(test_labels)];
temp_test = temp_test(randperm(size(temp_test, 1)), :);
test_files = temp_test(:, 1)';
test_labels = cell2mat(temp_test(:, 2));

% Load training and test set images
train_img = imageDatastore(train_files);
test_img = imageDatastore(test_files);
```

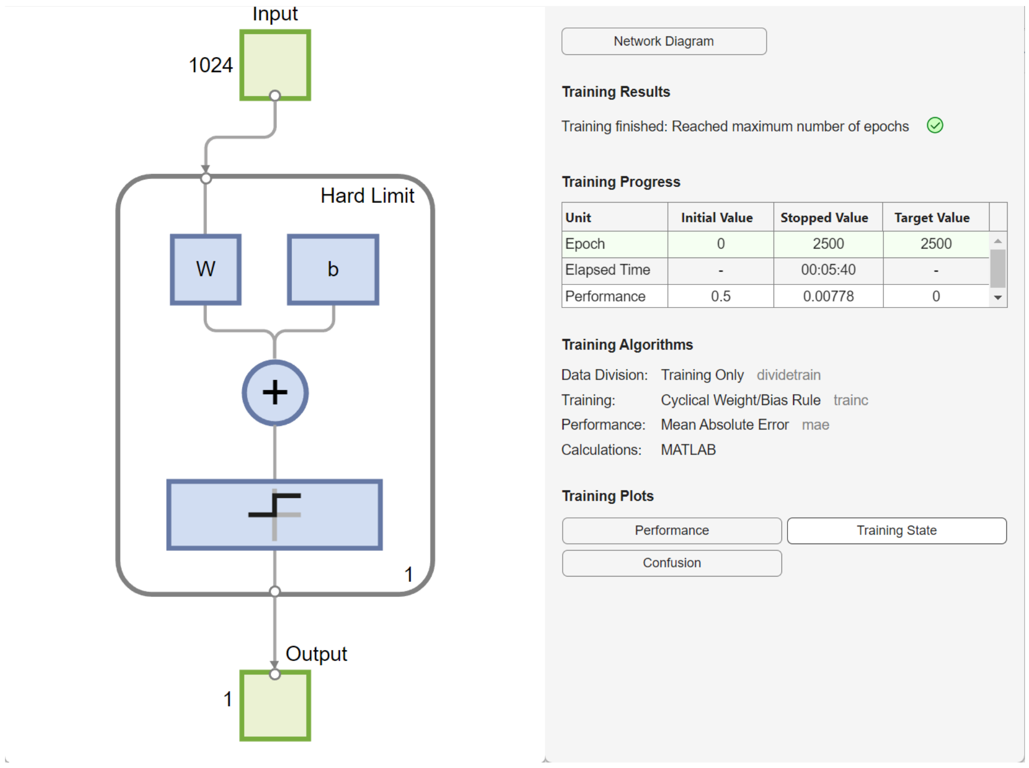
```matlab
numTrain = numel(train_img.Files);
X_train = zeros(1024, numTrain);

for i = 1:numTrain
    img = readimage(train_img, i); % Read training set images
    img_gray = rgb2gray(img); % Convert to grayscale
    % img_gray = double(img_gray) / 255;
    X_train(:, i) = img_gray(:); % Flatten to 1024×1 and store in matrix
end

numTest = numel(test_img.Files);
X_test = zeros(1024, numTest);

for i = 1:numTest
    img = readimage(test_img, i); % Read test set images
    img_gray = rgb2gray(img);
    % img_gray = double(img_gray) / 255;
    X_test(:, i) = img_gray(:);
end

% Create a perceptron
net = perceptron();

% Set hyperparameters
net.trainParam.epochs = 2500;
net.trainParam.lr = 0.01;
net.trainParam.showWindow = true;

% Define labels for training and test sets
Y_train = train_labels;
Y_test = test_labels;

% Train the perceptron
net = train(net, X_train, Y_train');

% Evaluate classification accuracy on the training set
disp('Perceptron Model Training Completed.');
Y_pred_train = net(X_train);
Y_pred_train = round(Y_pred_train);
accuracy_train = sum(Y_pred_train == Y_train') / numel(Y_train) * 100;
disp(['Train Accuracy: ', num2str(accuracy_train), '%']);

% Evaluate classification accuracy on the test set
```

```
Y_pred_test = net(X_test);
Y_pred_test = round(Y_pred_test);
accuracy_test = sum(Y_pred_test == Y_test') / numel(Y_test) * 100;
disp(['Test Accuracy: ', num2str(accuracy_test), '%']);

% Randomly select 9 images from the test set to preview the classification results
figure;
sgtitle('Sample Predictions');
for i = 1:min(9, numTest)
    subplot(3, 3, i);
    img = readimage(test_img, i);
    imshow(img);
    actual = test_labels(i);
    predicted = Y_pred_test(i);
    if actual == 0
        actual_class = 'deer';
    else
        actual_class = 'ship';
    end
    if predicted == 0
        predicted_class = 'deer';
    else
        predicted_class = 'ship';
    end
    title_color = 'green';
    if actual ~= predicted
        title_color = 'red';
    end
    title({['Actual: ' actual_class], ['Pred: ' predicted_class]}, 'Color', title_color);
end
```

As shown in the program output, the classification accuracy for the training set is 99.2222%, the classification accuracy for the test set is 67%.

```
Perceptron Model Training Completed.
Train Accuracy: 99.2222%
Test Accuracy: 67%
```

The structure of the network and an overview of the neural network are shown in the following two figures:



Training Results

Training finished: Reached maximum number of epochs ✓

Training Progress

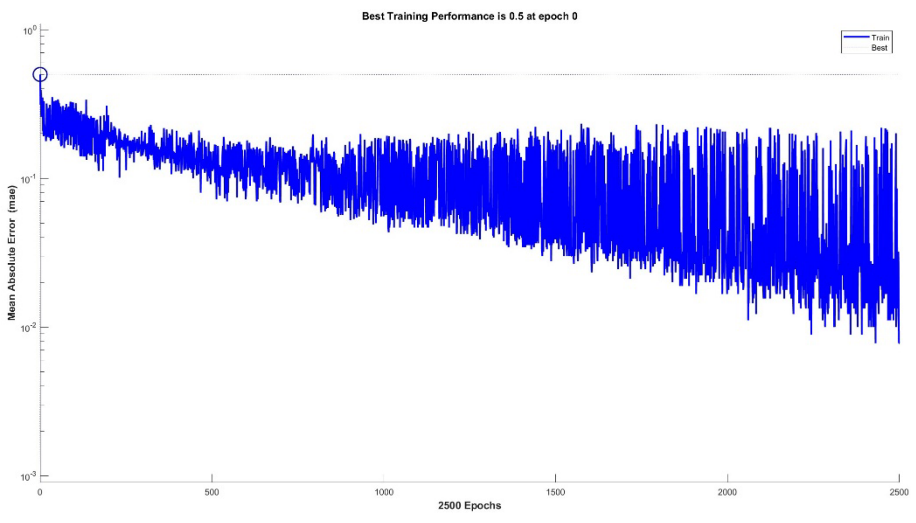
Unit	Initial Value	Stopped Value	Target Value
Epoch	0	2500	2500
Elapsed Time	-	00:05:40	-
Performance	0.5	0.00778	0

Training Algorithms

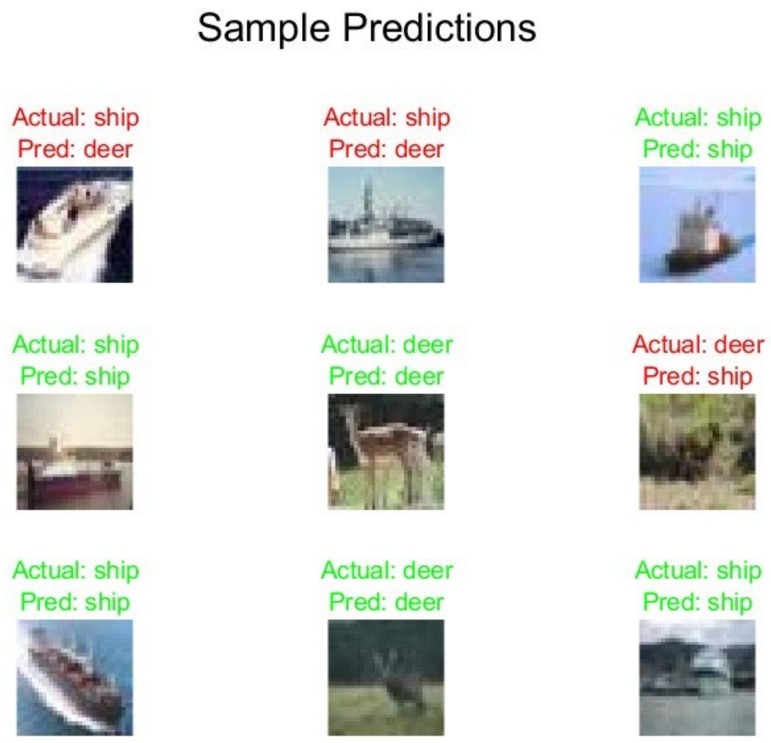
Data Division:	Training Only	dividetrain
Training:	Cyclical Weight/Bias Rule	trainc
Performance:	Mean Absolute Error	mae
Calculations:	MATLAB	

Training Plots

The mean absolute error trajectory over the number of epochs is shown in the figure below:



After the model completes the inference of the test set, I randomly select 9 pictures from the test set to demonstrate the accuracy of the model's classification. The results are shown in the figure below. Among the 9 selected pictures, 6 pictures are correctly classified, which is consistent with the test set classification accuracy of 67% output by the program. Since the feature dimension of the image is expanded to one dimension and the network model is too simple, the classification effect of the model is not accurate.



b). The code for perceptron training by calculating the global mean and standard deviation for normalization is as follows:

```
clc;
clear;
close all;

% Folder path
deer_folder = './deer';
ship_folder = './ship';

% Number of images per class for training
num_train_img = 450;

% Traverse all the pictures in the folder
deer_img = dir(fullfile(deer_folder, '*.jpg'));
ship_img = dir(fullfile(ship_folder, '*.jpg'));

% Create empty cell arrays to store the image paths of the training set and test set
deer_train = {};
ship_train = {};
deer_test = {};
```

```
ship_test = {};

% Separate training set and test set of the deer pictures
for i = 1 : length(deer_img)
    [~, name, ~] = fileparts(deer_img(i).name);
    num = str2double(name);
    if num >= 0 && num < num_train_img
        deer_train{end + 1} = fullfile(deer_folder, deer_img(i).name);
    else
        deer_test{end + 1} = fullfile(deer_folder, deer_img(i).name);
    end
end

% Separate training set and test set of the ship pictures
for i = 1 : length(ship_img)
    [~, name, ~] = fileparts(ship_img(i).name);
    num = str2double(name);
    if num >= 0 && num < num_train_img
        ship_train{end + 1} = fullfile(ship_folder, ship_img(i).name);
    else
        ship_test{end + 1} = fullfile(ship_folder, ship_img(i).name);
    end
end

% Merge the training and test sets separately and correspond to the correct labels
train_files = [deer_train, ship_train];
test_files = [deer_test, ship_test];
train_labels = [zeros(length(deer_train), 1); ones(length(ship_train), 1)];
test_labels = [zeros(length(deer_test), 1); ones(length(ship_test), 1)];

% Randomly shuffle the training set to improve generalization ability
temp_train = [train_files', num2cell(train_labels)];
temp_train = temp_train(randperm(size(temp_train, 1)), :);
train_files = temp_train(:, 1)';
train_labels = cell2mat(temp_train(:, 2));

% Randomly shuffle the test set
temp_test = [test_files', num2cell(test_labels)];
temp_test = temp_test(randperm(size(temp_test, 1)), :);
test_files = temp_test(:, 1)';
test_labels = cell2mat(temp_test(:, 2));

% Load training and test set images
train_img = imageDatastore(train_files);
```

```
test_img = imageDatastore(test_files);

% Get the number of images
numTrain = numel(train_img.Files);
numTest = numel(test_img.Files);

% Store the pixel value of each image
all_pixels = [];

% Traverse the training set and test set to get all pixel values
for i = 1:numTrain
    img = readimage(train_img, i);
    img_gray = rgb2gray(img);
    all_pixels = [all_pixels; double(img_gray(:))];
end

for i = 1:numTest
    img = readimage(test_img, i);
    img_gray = rgb2gray(img);
    all_pixels = [all_pixels; double(img_gray(:))];
end

% Get the global pixel mean and standard deviation
global_mean = mean(all_pixels);
global_std = std(all_pixels);

disp(['Global Mean: ', num2str(global_mean)]);
disp(['Global Standard Deviation: ', num2str(global_std)]);

X_train = zeros(1024, numTrain);
X_test = zeros(1024, numTest);

for i = 1:numTrain
    img = readimage(train_img, i); % Read training set images
    img_gray = rgb2gray(img); % Convert to grayscale
    img_gray = double(img_gray);
    % Normalization by subtracting the mean value from each image and divide each image
by the standard deviation.
    img_gray = (img_gray - global_mean) / global_std;
    X_train(:, i) = img_gray(:); % Flatten to 1024×1 and store in matrix
end

for i = 1:numTest
    img = readimage(test_img, i);
    img_gray = rgb2gray(img);
    img_gray = double(img_gray);
    img_gray = (img_gray - global_mean) / global_std;
    X_test(:, i) = img_gray(:);
end

Y_train = train_labels;
Y_test = test_labels;

% Create a perceptron
net = perceptron();

% Set hyperparameters
net.trainParam.epochs = 2500;
net.trainParam.lr = 0.01;
net.trainParam.showWindow = true;

% Train the perceptron
[net, tr] = train(net, X_train, Y_train');

% Evaluate classification accuracy on the training set
disp('Perceptron Model Training Completed.');
Y_pred_train = net(X_train);
Y_pred_train = round(Y_pred_train);
accuracy_train = sum(Y_pred_train == Y_train') / numel(Y_train) * 100;
disp(['Train Accuracy: ', num2str(accuracy_train), '%']);

% Evaluate classification accuracy on the test set
Y_pred_test = net(X_test);
Y_pred_test = round(Y_pred_test);
accuracy_test = sum(Y_pred_test == Y_test') / numel(Y_test) * 100;
disp(['Test Accuracy: ', num2str(accuracy_test), '%']);

% plot the training curve
figure;
plot(1:length(tr.epoch), tr.perf, 'b-', 'LineWidth', 2);
xlabel('Epochs');
ylabel('Training Performance');
title('Training Performance Over Epochs');
grid on;

% Randomly select 9 images from the test set to preview the classification results
figure;
sgtitle('Sample Predictions');
```

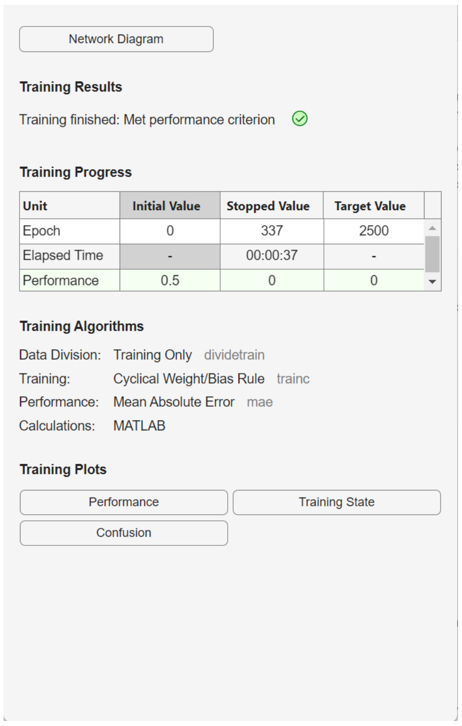
```
for i = 1:min(9, numTest)
    subplot(3, 3, i);
    img = readimage(test_img, i);
    imshow(img);
    actual = test_labels(i);
    predicted = Y_pred_test(i);
    if actual == 0
        actual_class = 'deer';
    else
        actual_class = 'ship';
    end
    if predicted == 0
        predicted_class = 'deer';
    else
        predicted_class = 'ship';
    end
    title_color = 'green';
    if actual ~= predicted
        title_color = 'red';
    end
    title({['Actual: ' actual_class], ['Pred: ' predicted_class]}, 'Color', title_color);
end
```

As shown in the program output, the classification accuracy for the training set is 100%, the classification accuracy for the test set is 78%. The global mean of the dataset is 123.9108 and the global standard deviation is 56.7739.

```
Global Mean: 123.9108
Global Standard Deviation: 56.7739
Perceptron Model Training Completed.
Train Accuracy: 100%
Test Accuracy: 78%
```

From the program output, we can see that the model completed training in the 337th epoch, which is much faster than the 2500 epochs required in a). The classification accuracy of the training set reached 100%, and the classification accuracy of the test set was also improved compared to a), reaching 78%. At this time, because standardization makes all input features at the same scale, the gradient update is more stable and smoother, which speeds up the convergence speed. At the same time, through the mean-variance normalization method, the network can more easily learn the structural characteristics of the data and is not affected by images with a large range of pixel values.

An overview of the training results, the trajectory of model optimization, and the results of the test set sampling verification are shown in the following figures.



Training Results

Training finished: Met performance criterion ✓

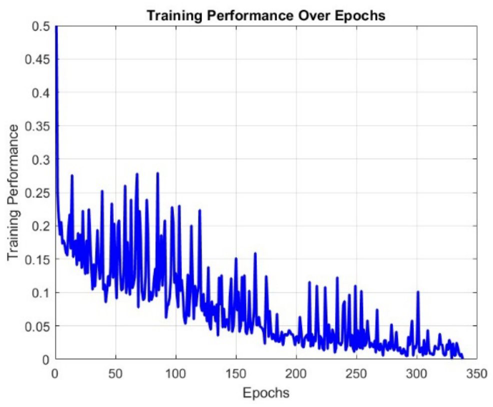
Training Progress

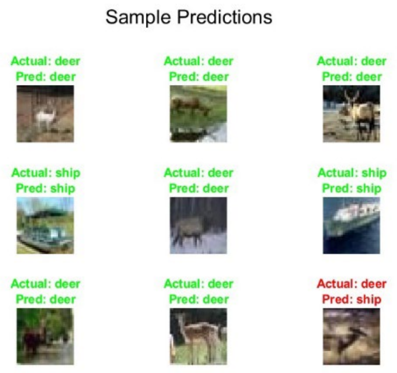
Unit	Initial Value	Stopped Value	Target Value
Epoch	0	337	2500
Elapsed Time	-	00:00:37	-
Performance	0.5	0	0

Training Algorithms

Data Division:	Training Only dividetrain
Training:	Cyclical Weight/Bias Rule trainc
Performance:	Mean Absolute Error mae
Calculations:	MATLAB

Training Plots





c.) Since the mean-standard deviation normalization method has better classification and convergence effects in b), this method is continued in c). I used the *patternnet* function to create an MLP with 100 neurons in its hidden layer. The training method is *traingdx*, which means gradient descent w/momentum & adaptive learning rate backpropagation. This method can greatly speed up the training speed and adjust the learning rate autonomously. The loss function uses the mean squre error function. The code is as follows:

```
clc;
clear;
close all;

% Folder path
deer_folder = './deer';
ship_folder = './ship';

```

```matlab
% Number of images per class for training
num_train_img = 450;

% Traverse all the pictures in the folder
deer_img = dir(fullfile(deer_folder, '*.jpg'));
ship_img = dir(fullfile(ship_folder, '*.jpg'));

% Create empty cell arrays to store the image paths of the training set and test set
deer_train = {};
ship_train = {};
deer_test = {};
ship_test = {};

% Separate training set and test set of the deer pictures
for i = 1:length(deer_img)
    [~, name, ~] = fileparts(deer_img(i).name);
    num = str2double(name);
    if num < num_train_img
        deer_train{end + 1} = fullfile(deer_folder, deer_img(i).name);
    else
        deer_test{end + 1} = fullfile(deer_folder, deer_img(i).name);
    end
end

% Separate training set and test set of the ship pictures
for i = 1:length(ship_img)
    [~, name, ~] = fileparts(ship_img(i).name);
    num = str2double(name);
    if num < num_train_img
        ship_train{end + 1} = fullfile(ship_folder, ship_img(i).name);
    else
        ship_test{end + 1} = fullfile(ship_folder, ship_img(i).name);
    end
end

% Merge the training and test sets separately and correspond to the correct labels
train_files = [deer_train, ship_train];
test_files = [deer_test, ship_test];
train_labels = [zeros(length(deer_train), 1); ones(length(ship_train), 1)];
test_labels = [zeros(length(deer_test), 1); ones(length(ship_test), 1)];

% Randomly shuffle the training set to improve generalization ability
temp_train = [train_files', num2cell(train_labels)];
temp_train = temp_train(randperm(size(temp_train, 1)), :);
```

```matlab
train_files = temp_train(:, 1)';
train_labels = cell2mat(temp_train(:, 2));

% Randomly shuffle the test set
temp_test = [test_files', num2cell(test_labels)];
temp_test = temp_test(randperm(size(temp_test, 1)), :);
test_files = temp_test(:, 1)';
test_labels = cell2mat(temp_test(:, 2));

% Load training and test set images
train_img = imageDatastore(train_files);
test_img = imageDatastore(test_files);

% Get the number of images
numTrain = numel(train_img.Files);
numTest = numel(test_img.Files);

% Store the pixel value of each image
all_pixels = [];

% Traverse the training set and test set to get all pixel values
for i = 1:numTrain
    img = readimage(train_img, i);
    img_gray = rgb2gray(img);
    all_pixels = [all_pixels; double(img_gray(:))];
end

for i = 1:numTest
    img = readimage(test_img, i);
    img_gray = rgb2gray(img);
    all_pixels = [all_pixels; double(img_gray(:))];
end

% Get the global pixel mean and standard deviation
global_mean = mean(all_pixels);
global_std = std(all_pixels);

disp(['Global Mean: ', num2str(global_mean)]);
disp(['Global Standard Deviation: ', num2str(global_std)]);

X_train = zeros(1024, numTrain);
X_test = zeros(1024, numTest);

for i = 1:numTrain
```

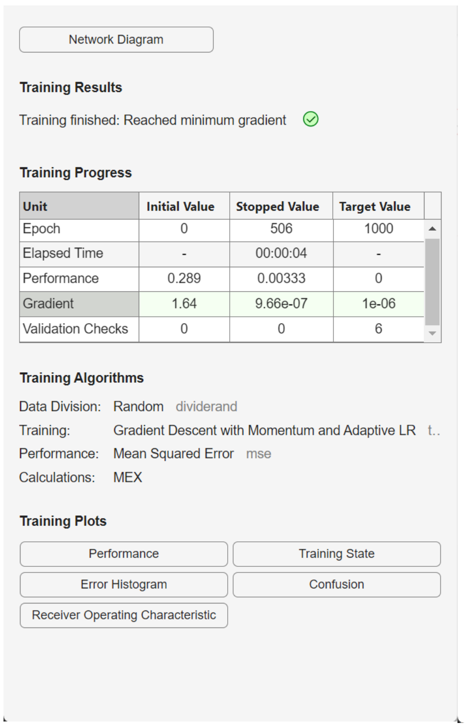
```
    img = readimage(train_img, i); % Read training set images
    img_gray = rgb2gray(img); % Convert to grayscale
    img_gray = double(img_gray);
    % Normalization by subtracting the mean value from each image and divide each image
by the standard deviation.
    img_gray = (img_gray - global_mean) / global_std;
    X_train(:, i) = img_gray(:); % Flatten to 1024×1 and store in matrix
end

for i = 1:numTest
    img = readimage(test_img, i);
    img_gray = rgb2gray(img);
    img_gray = double(img_gray);
    img_gray = (img_gray - global_mean) / global_std;
    X_test(:, i) = img_gray(:);
end

% Create an MLP
hidden_neurons = 100; % Number of hidden neurons
net = patternnet(hidden_neurons);

% Set hyperparameters
net.trainFcn = 'traingdx';
net.performFcn = 'mse';
net.trainParam.epochs = 1000;
net.trainparam.goal = 0;
net.trainParam.min_grad = 1e-6;
net.divideParam.trainRatio = 1;
net.divideParam.valRatio = 0;
net.divideParam.testRatio = 0;
net.trainParam.showWindow = true;

% Change label format
Y_train = full(ind2vec(train_labels' + 1)); % [0,1] → [1 0] & [0 1]
Y_test = full(ind2vec(test_labels' + 1));

% Train the MLP
[net, tr] = train(net, X_train, Y_train);

% Evaluate classification accuracy on the training set
Y_pred_train = net(X_train);
Y_pred_train = vec2ind(Y_pred_train) - 1;
accuracy_train = sum(Y_pred_train == train_labels') / numel(train_labels) * 100;
disp(['Train Accuracy: ', num2str(accuracy_train), '%']);
```

```
% Evaluate classification accuracy on the test set
Y_pred_test = net(X_test);
Y_pred_test = vec2ind(Y_pred_test) - 1;
accuracy_test = sum(Y_pred_test == test_labels') / numel(test_labels) * 100;
disp(['Test Accuracy: ', num2str(accuracy_test), '%']);

% plot the training curve
figure;
plot(tr.epoch, tr.perf, 'b-', 'LineWidth', 2);
xlabel('Epochs');
ylabel('Mean Squared Error (MSE)');
title('MLP Training Performance');
grid on;

% Randomly select 9 images from the test set to preview the classification results
figure;
sgtitle('Sample Predictions');
for i = 1:min(9, numTest)
    subplot(3, 3, i);
    img = readimage(test_img, i);
    imshow(img);
    actual = test_labels(i);
    predicted = Y_pred_test(i);
    if actual == 0
        actual_class = 'deer';
    else
        actual_class = 'ship';
    end
    if predicted == 0
        predicted_class = 'deer';
    else
        predicted_class = 'ship';
    end
    title_color = 'green';
    if actual ~= predicted
        title_color = 'red';
    end
    title({['Actual: ' actual_class], ['Pred: ' predicted_class]}, 'Color', title_color);
end
```

As shown in the program output, the classification accuracy for the training set is 99.6667%, the classification accuracy for the test set is 81%. The global mean of the dataset is 123.9108 and the global standard deviation is 56.7739. Since hidden layer is used and the number of neurons in the hidden layers is large, the nonlinear transformation capability of the model is enhanced, and the classification performance of the test set is slightly better than that of using only a single neuron classification.

```
Global Mean: 123.9108
Global Standard Deviation: 56.7739
Train Accuracy: 99.6667%
Test Accuracy: 81%
```

An overview of the training results, network structure, the trajectory of model optimization, and the results of the test set sampling verification are shown in the following figures.



Network Diagram

Training Results

Training finished: Reached minimum gradient ✓

Training Progress

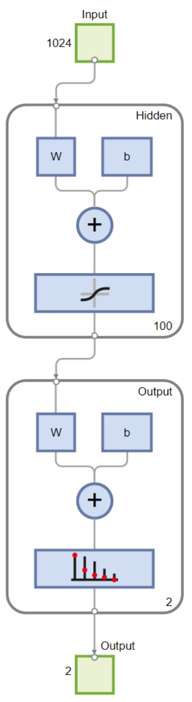
Unit	Initial Value	Stopped Value	Target Value
Epoch	0	506	1000
Elapsed Time	-	00:00:04	-
Performance	0.289	0.00333	0
Gradient	1.64	9.66e-07	1e-06
Validation Checks	0	0	6

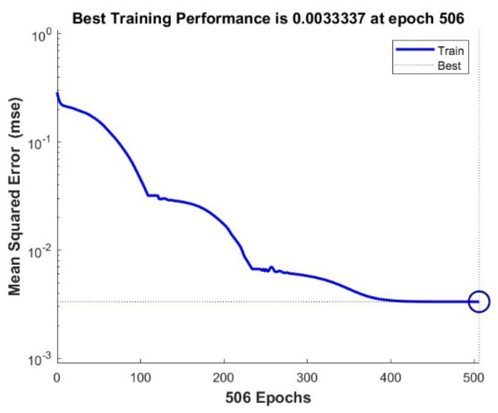
Training Algorithms

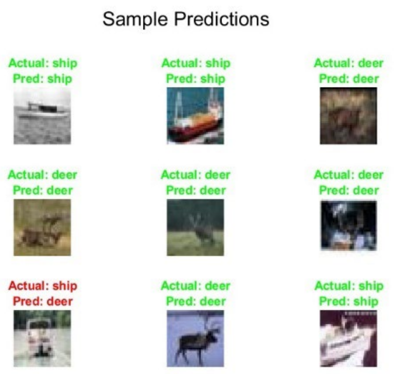
Data Division:	Random dividerand
Training:	Gradient Descent with Momentum and Adaptive LR t..
Performance:	Mean Squared Error mse
Calculations:	MEX

Training Plots

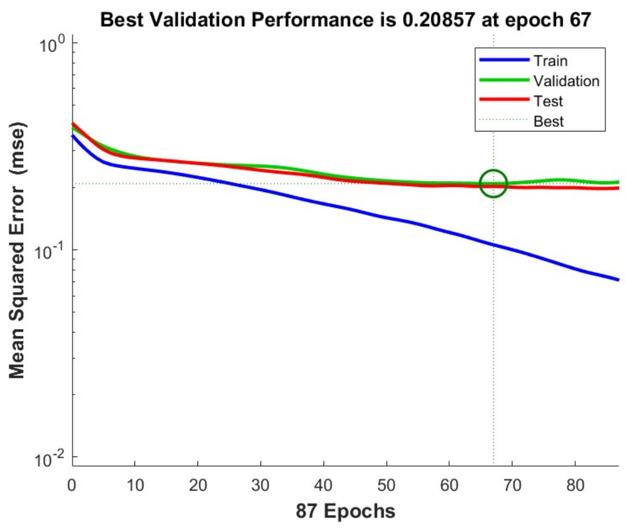
Performance | Training State | Error Histogram | Confusion | Receiver Operating Characteristic







d).



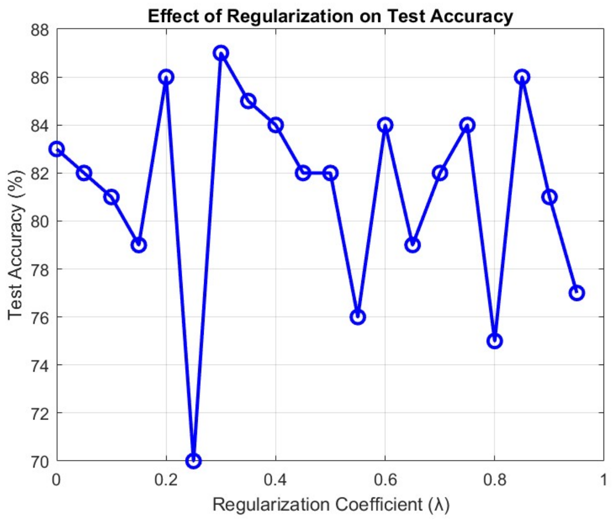
As can be seen from the figure, the MLP I trained overfitted in the 67th epoch. I adjusted the code and set the range of the regularization parameter to 0-0.95 with an interval of 0.05. The regularization parameter was adjusted with the help of *performParam.regularization* function, The curve of the test set classification accuracy as the regularization parameter size changes was plotted later. The code is shown below:

```
clc;
clear;
close all;

% Folder path
deer_folder = './deer';
ship_folder = './ship';

% Number of images per class for training
num_train_img = 450;

% Traverse all the pictures in the folder
deer_img = dir(fullfile(deer_folder, '*.jpg'));
ship_img = dir(fullfile(ship_folder, '*.jpg'));

% Separate training set and test set of the deer pictures
deer_train = {};
ship_train = {};
deer_test = {};
ship_test = {};

% Separate training set and test set of the deer pictures
for i = 1:length(deer_img)
    [~, name, ~] = fileparts(deer_img(i).name);
    num = str2double(name);
```

```matlab
    if num < num_train_img
        deer_train{end + 1} = fullfile(deer_folder, deer_img(i).name);
    else
        deer_test{end + 1} = fullfile(deer_folder, deer_img(i).name);
    end
end

% Separate training set and test set of the ship pictures
for i = 1:length(ship_img)
    [~, name, ~] = fileparts(ship_img(i).name);
    num = str2double(name);
    if num < num_train_img
        ship_train{end + 1} = fullfile(ship_folder, ship_img(i).name);
    else
        ship_test{end + 1} = fullfile(ship_folder, ship_img(i).name);
    end
end

% Merge the training and test sets separately and correspond to the correct labels
train_files = [deer_train, ship_train];
test_files = [deer_test, ship_test];
train_labels = [zeros(length(deer_train), 1); ones(length(ship_train), 1)];
test_labels = [zeros(length(deer_test), 1); ones(length(ship_test), 1)];

% Randomly shuffle the training set to improve generalization ability
temp_train = [train_files', num2cell(train_labels)];
temp_train = temp_train(randperm(size(temp_train, 1)), :);
train_files = temp_train(:, 1)';
train_labels = cell2mat(temp_train(:, 2));

% Randomly shuffle the test set
temp_test = [test_files', num2cell(test_labels)];
temp_test = temp_test(randperm(size(temp_test, 1)), :);
test_files = temp_test(:, 1)';
test_labels = cell2mat(temp_test(:, 2));

% Load training and test set images
train_img = imageDatastore(train_files);
test_img = imageDatastore(test_files);

numTrain = numel(train_img.Files);
numTest = numel(test_img.Files);

% Get the global pixel mean and standard deviation
```

```matlab
all_pixels = [];

for i = 1:numTrain
    img = readimage(train_img, i);
    img_gray = rgb2gray(img);
    all_pixels = [all_pixels; double(img_gray(:))];
end

for i = 1:numTest
    img = readimage(test_img, i);
    img_gray = rgb2gray(img);
    all_pixels = [all_pixels; double(img_gray(:))];
end

global_mean = mean(all_pixels);
global_std = std(all_pixels);

disp(['Global Mean: ', num2str(global_mean)]);
disp(['Global Standard Deviation: ', num2str(global_std)]);

X_train = zeros(1024, numTrain);
X_test = zeros(1024, numTest);

for i = 1:numTrain
    img = readimage(train_img, i);
    img_gray = rgb2gray(img);
    img_gray = double(img_gray);
    img_gray = (img_gray - global_mean) / global_std;
    X_train(:, i) = img_gray(:);
end

for i = 1:numTest
    img = readimage(test_img, i);
    img_gray = rgb2gray(img);
    img_gray = double(img_gray);
    img_gray = (img_gray - global_mean) / global_std;
    X_test(:, i) = img_gray(:);
end

% Record the test set accuracy of different regularization coefficients
lambda_values = 0:0.05:0.95;  % λ ranges from 0 to 0.95, with every 0.05
test_accuracies = zeros(length(lambda_values), 1);

% Number of hidden neurons
```

```
hidden_neurons = 100;

% Traverse different λ training MLP
for idx = 1:length(lambda_values)
    lambda = lambda_values(idx);  % Take different values of λ
    disp(['Training with λ = ', num2str(lambda)]);

    % Create an MLP
    net = patternnet(hidden_neurons);

    % Set hyperparameters
    net.trainFcn = 'traingdx';
    net.performFcn = 'mse';
    net.trainParam.epochs = 500;
    net.performParam.regularization = lambda; % Set regularization parameters
    net.divideParam.trainRatio = 0.7;
    net.divideParam.valRatio = 0.2;
    net.divideParam.testRatio = 0.1;
    net.trainParam.showWindow = true;

    % Change label format
    Y_train = full(ind2vec(train_labels' + 1)); % [0,1] → [1 0] & [0 1]
    Y_test = full(ind2vec(test_labels' + 1));

    % Train the MLP
    [net, ~] = train(net, X_train, Y_train);

    % Evaluate classification accuracy on the test set
    Y_pred_test = net(X_test);
    Y_pred_test = vec2ind(Y_pred_test) - 1;
    accuracy_test = sum(Y_pred_test == test_labels') / numel(test_labels) * 100;
    test_accuracies(idx) = accuracy_test;

    disp(['Lambda = ', num2str(lambda), ' -> Test Accuracy: ', num2str(accuracy_test),
'%']);
end

% Result visualization
figure;
plot(lambda_values, test_accuracies, 'bo-', 'LineWidth', 2, 'MarkerSize', 8);
xlabel('Regularization Coefficient (λ)');
ylabel('Test Accuracy (%)');
title('Effect of Regularization on Test Accuracy');
grid on;
```



As can be seen from the figure, different regularization parameters did not lead to a higher increase in the classification accuracy of the test set. Since the training of the neural network has an early stopping mechanism, the early stopping of training determined by parameters such as gradient and validation checks is sufficient to keep the classification accuracy of the test set at a high level. However, the intervention of regularization can effectively prevent overfitting.

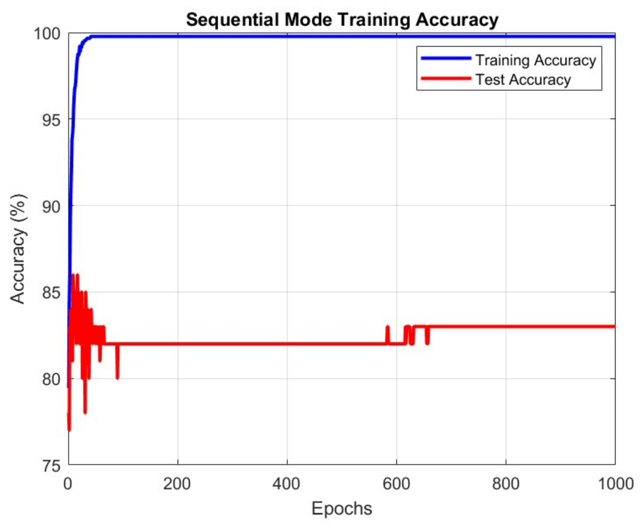
e). The code for training the MLP using sequential mode is as follows:

```
clc;
clear;
close all;

% Folder path
deer_folder = './deer';
ship_folder = './ship';

% Number of images per class for training
num_train_img = 450;

% Traverse all the pictures in the folder
deer_img = dir(fullfile(deer_folder, '*.jpg'));
ship_img = dir(fullfile(ship_folder, '*.jpg'));

% Create empty cell arrays to store the image paths of the training set and test set
deer_train = {};
ship_train = {};
deer_test = {};
ship_test = {};
```

```
% Separate training set and test set of the deer pictures
for i = 1:length(deer_img)
    [~, name, ~] = fileparts(deer_img(i).name);
    num = str2double(name);
    if num < num_train_img
        deer_train{end + 1} = fullfile(deer_folder, deer_img(i).name);
    else
        deer_test{end + 1} = fullfile(deer_folder, deer_img(i).name);
    end
end

% Separate training set and test set of the ship pictures
for i = 1:length(ship_img)
    [~, name, ~] = fileparts(ship_img(i).name);
    num = str2double(name);
    if num < num_train_img
        ship_train{end + 1} = fullfile(ship_folder, ship_img(i).name);
    else
        ship_test{end + 1} = fullfile(ship_folder, ship_img(i).name);
    end
end

% Merge the training and test sets separately and correspond to the correct labels
train_files = [deer_train, ship_train];
test_files = [deer_test, ship_test];
train_labels = [zeros(length(deer_train), 1); ones(length(ship_train), 1)];
test_labels = [zeros(length(deer_test), 1); ones(length(ship_test), 1)];

% Randomly shuffle the training set to improve generalization ability
temp_train = [train_files', num2cell(train_labels)];
temp_train = temp_train(randperm(size(temp_train, 1)), :);
train_files = temp_train(:, 1)';
train_labels = cell2mat(temp_train(:, 2));

% Randomly shuffle the test set
temp_test = [test_files', num2cell(test_labels)];
temp_test = temp_test(randperm(size(temp_test, 1)), :);
test_files = temp_test(:, 1)';
test_labels = cell2mat(temp_test(:, 2));

% Load training and test set images
train_img = imageDatastore(train_files);
test_img = imageDatastore(test_files);
```

```
% Get the number of images
numTrain = numel(train_img.Files);
numTest = numel(test_img.Files);

% Store the pixel value of each image
all_pixels = [];

for i = 1:numTrain
    img = readimage(train_img, i);
    img_gray = rgb2gray(img);
    all_pixels = [all_pixels; double(img_gray(:))];
end

for i = 1:numTest
    img = readimage(test_img, i);
    img_gray = rgb2gray(img);
    all_pixels = [all_pixels; double(img_gray(:))];
end

% Get the global pixel mean and standard deviation
global_mean = mean(all_pixels);
global_std = std(all_pixels);

disp(['Global Mean: ', num2str(global_mean)]);
disp(['Global Standard Deviation: ', num2str(global_std)]);

X_train = zeros(1024, numTrain);
X_test = zeros(1024, numTest);

for i = 1:numTrain
    img = readimage(train_img, i); % Read training set images
    img_gray = rgb2gray(img); % Convert to grayscale
    img_gray = double(img_gray);
    % Normalization by subtracting the mean value from each image and divide each image
by the standard deviation.
    img_gray = (img_gray - global_mean) / global_std;
    X_train(:, i) = img_gray(:); % Flatten to 1024×1 and store in matrix
end

for i = 1:numTest
    img = readimage(test_img, i);
    img_gray = rgb2gray(img);
    img_gray = double(img_gray);
    img_gray = (img_gray - global_mean) / global_std;
    X_test(:, i) = img_gray(:);
end

% Convert data format (suitable for adapt training)
X_train_c = num2cell(X_train, 1);
Y_train_c = num2cell(train_labels', 1);
X_test_c = num2cell(X_test, 1);
Y_test_c = num2cell(test_labels', 1);

% Create an MLP
hidden_neurons = 100;
net = patternnet(hidden_neurons);

% Set hyperparameters
net.trainFcn = 'traingdx';
net.performFcn = 'mse';
net.trainParam.epochs = 1000;
net.trainParam.lr = 0.01;
net.performParam.regularization = 0.25;
net.trainParam.showWindow = true;
num_epochs = 1000;
train_acc = zeros(num_epochs, 1);
test_acc = zeros(num_epochs, 1);

% Disable data set splitting to ensure that all data is used for training
net.divideFcn = 'dividetrain';

for epoch = 1:num_epochs
    disp(['Epoch ', num2str(epoch), ' / ', num2str(num_epochs)]);

    % Randomly shuffle training set
    idx = randperm(numTrain);
    X_train_c = X_train_c(:, idx);
    Y_train_c = Y_train_c(:, idx);

    % Update weights sample by sample
    net = adapt(net, X_train_c, Y_train_c);

    % Evaluate classification accuracy on the training set
    pred_train = round(net(X_train));
    train_acc(epoch) = sum(pred_train == train_labels') / numel(train_labels) * 100;

    % Evaluate classification accuracy on the test set
    pred_test = round(net(X_test));
    test_acc(epoch) = sum(pred_test == test_labels') / numel(test_labels) * 100;
end

% Plot the curves for training and test classification accuracy
figure;
plot(1:num_epochs, train_acc, 'b-', 'LineWidth', 2);
hold on;
plot(1:num_epochs, test_acc, 'r-', 'LineWidth', 2);
xlabel('Epochs');
ylabel('Accuracy (%)');
title('Sequential Mode Training Accuracy');
legend('Training Accuracy', 'Test Accuracy');
grid on;

% Compare the results
disp('Comparing Sequential Mode and Batch Mode:');
disp('Batch Mode Training Accuracy: 96.6667%');
disp('Batch Mode Test Accuracy: 81%');
disp('Sequential Mode Training Accuracy: ' + string(train_acc(end)) + '%');
disp('Sequential Mode Test Accuracy: ' + string(test_acc(end)) + '%');
```

```
Comparing Sequential Mode and Batch Mode:
Batch Mode Training Accuracy: 96.6667%
Batch Mode Test Accuracy: 81%
Sequential Mode Training Accuracy: 99.4444%
Sequential Mode Test Accuracy: 82%
```



As shown in the program output, the classification accuracy for the training set is 99.4444%, the classification accuracy for the test set is 82% by sequential mode. The classification accuracy for the training set is 96.6667%, the classification accuracy for the test set is 81% by batch mode. Combining the accuracy curves of the training set and test set of the MLP trained in sequential mode, batch mode and sequential mode do not have a clear advantage over each other. However, using batch mode to train the model can significantly reduce the training time because the weights can be updated after all samples are input. In summary, I recommend using batch mode to train the model.

f).

1. Randomly rotate, scale, translate, flip, and other operations are performed on the training set images to generate more diverse training samples. That can help the MLP learn more robust features, reduce overfitting and improve generalization ability.
2. A more complex network structure will have stronger feature learning capabilities and can capture more complex patterns. But it’s also necessary to pay attention of the risk of overfitting.
3. Carefully select the hyperparameters for neural network training, adjust and compare them multiple times, and try to find the hyperparameters that are suitable for the current training object to maximize the performance of the MLP.