NUS
National University
of Singapore

EE4305 Homework 1: Neural Systems

Name: Wah Xinyan Kendrik
Matriculation Number: A0164914B

Table of Contents

Question 1:

Given the Rosenbrock's Valley function:

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

We have a starting point of $(0, 0.5)$, with a global minimum at $(1,1)$ where $f(x,y) = 0$.

By the use of the Steepest Gradient Descent method, we have the following equation,

$w(k+1) = w(k) - \eta g(k)$, with $g(k)$ being a parameter that needs to be calculated.

Thus, inspecting the rate of change of $f(x,y)$ against x and y, we obtain the following relation:

$$g(k) = \begin{pmatrix} \frac{\partial f(x,y)}{\partial x} \\ \frac{\partial f(x,y)}{\partial y} \end{pmatrix} = \begin{pmatrix} 2(x-1) - 400x(y-x^2) \\ 200(y-x^2) \end{pmatrix}$$

The code for Rosenbrock's valley and the gradient is given below:

```
function val = rosen(x, y)
    val = (1-x)^2 + (100 * (y - (x^2))^2);
end

function Df = rosenGrad(x, y)
    k = [x-1; y-(x^2)];
    Df = [2*k(1)-(400*x*k(2)); 200*k(2)];
end
```

1a) With a learning rate of $\eta = 0.001$, it is found that 23530 iterations for the values of x and y to reach 1. The trajectory of x, y and $f(x, y)$ are shown in Fig. 1 and 2 below:

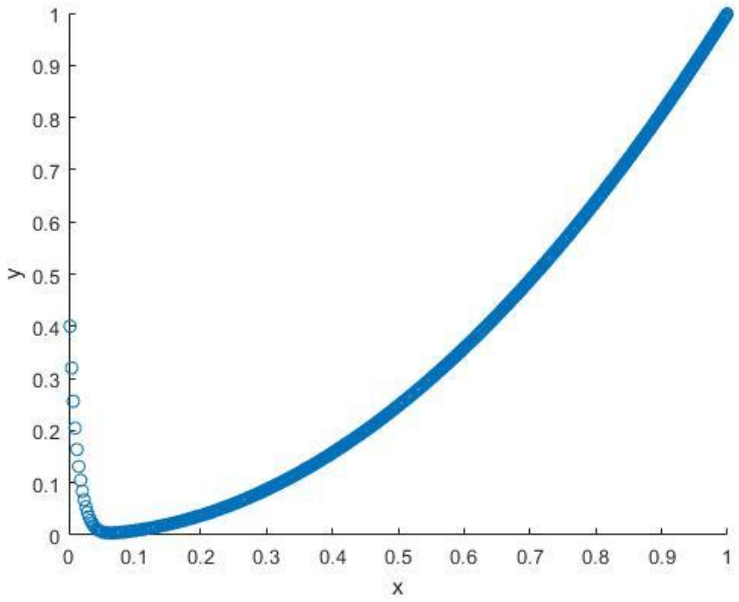

Fig. 1: Trajectories of x and y without contour plot

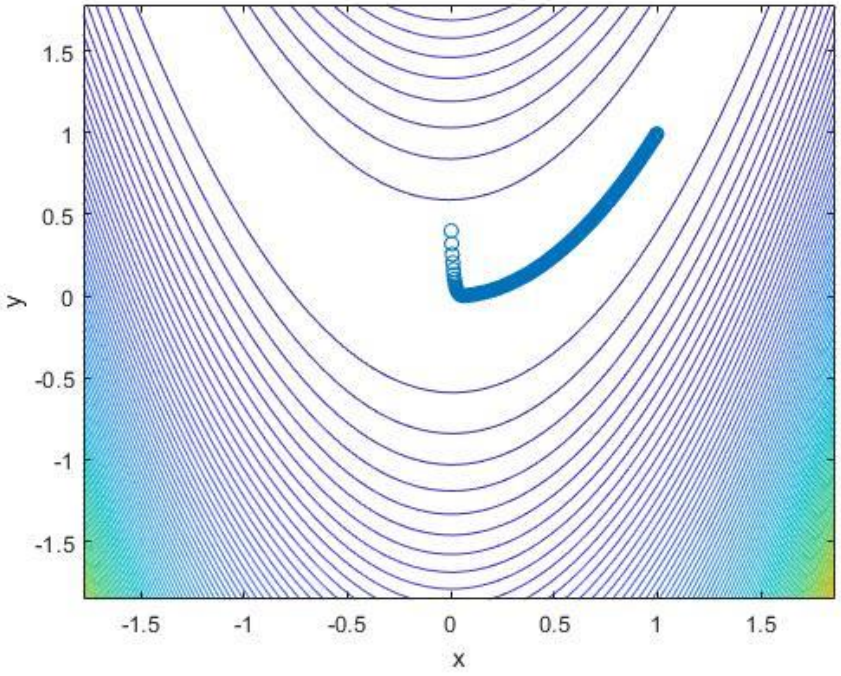

Fig. 2: Trajectories of x and y with contour plot

It is also shown that it takes approximately 23530 iterations for the values of the convergence of x and y to the global minimum at $(1,1)$ in Fig. 3 below.

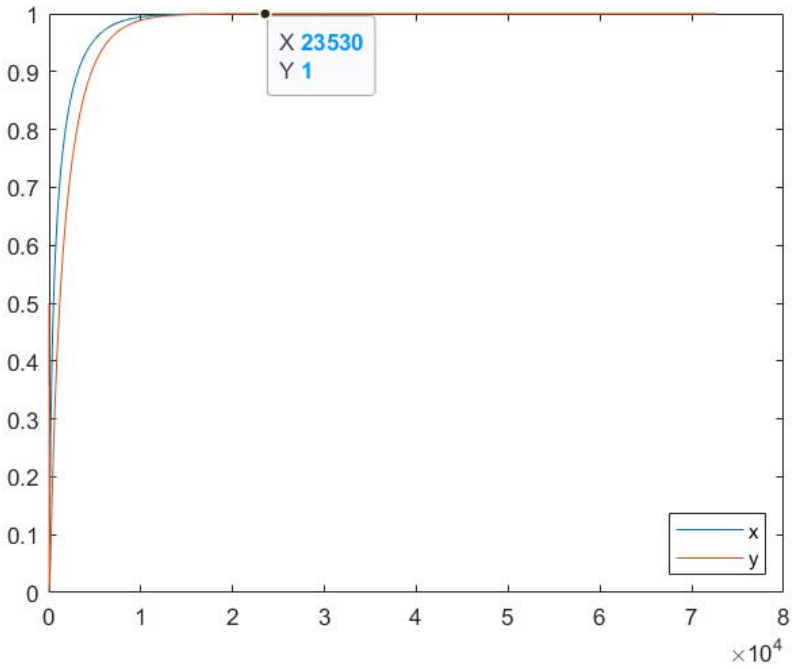

Fig. 3: $x\ and\ y$ plotted against number of iterations

Finally, it is observed that the under steepest descent algorithm, it requires approximately with $f(x,y) = 1.00 \times 10^{-6}$ after 14290 iterations in Fig. 4 below. Note that while it is not visible in Fig. 4 below due to scaling, $f(x,y) = 0$ after 72594 iterations. Yet, it is notable that after 72590 iterations, $f(x,y) \approx 0$. This value is obtained due to the scaling of MATLAB's plot().

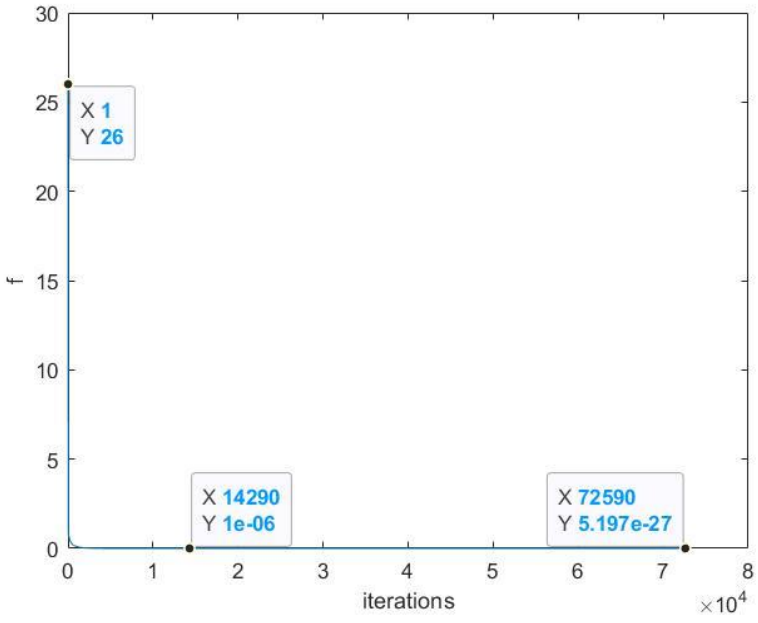

Fig. 4: $f(x,y)$ plotted against number of iterations (there is a blue line, look closely)

However, $\eta = 0.2$, it appears that the gradient descent will run indefinitely. This is because the update in the error is based on Taylor's Series, that the algorithm will only converge for a sufficiently small η.

Gradient Descent Code is inserted in the next page.

```
X = [0; 0.5];
iter_count = 0;
eta_1 = 1e-3;
eta_2 = 2e-1;
stop = 0;
iters = 1e20;
flag = 1;
i = 1;

x = zeros();
y = zeros();
z = zeros();
errors = zeros();
iterations = zeros();

while (flag && i <= iters)

    x(i) = X(1);
    y(i) = X(2);

    X_prev = X;
    g = rosenGrad(X(1), X(2));
    X = X - (eta_1 * g); % can change eta_1 with eta_2 whenever you want.
    error = norm(X - X_prev);

    errors(i) = error;
    iterations(i) = i;

    if (stop >= error)
        flag = 0;
    end

    i = i + 1;
end

figure;
x_values = [-2:0.01:2]; y_values = [-2:0.01:2];
f = @(x,y) (1-x).^2 + 100*(y-x.^2).^2;
[xx, yy] = meshgrid(x_values, y_values);
ff = f(xx,yy);
fn_output = f(x, y);
contour(xx, yy, ff, 100);

scatter(x, y);
xlabel('x');
ylabel('y');

figure;
plot(iterations, x, iterations, y);
legend({"x", "y"}, 'Location', 'southeast');

figure;
plot(iterations, fn_output);
xlabel('iterations');
ylabel('f');
```

1b) It is requested that Newton's method should be used, as given by the following equation:

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

$g(n)$ is the gradient vector as calculated in **1a)** and $H(n)$ is the Hessian matrix.

Thus, the Hessian matrix, $H(n)$ is calculated as follows:

$$H(n) = \begin{pmatrix} \frac{\partial^2 f(x,y)}{\partial x^2} & \frac{\partial^2 f(x,y)}{\partial x \partial y} \\ \frac{\partial^2 f(x,y)}{\partial x \partial y} & \frac{\partial^2 f(x,y)}{\partial y^2} \end{pmatrix} = \begin{pmatrix} 2 - 400y + 1200x^2 & -400x \\ -400x & 200 \end{pmatrix}$$

The computation of the Hessian matrix is given below:

```
function H = rosenHess(x, y)
    df2dx2 = 2 - 400*y + 1200*(x^2);
    df2dy2 = 200
    df2dxdy = -400 * x;
    H = [df2dx2, df2dxdy; df2dxdy, df2dy2];
end
```

It is found that through Newton's Method, while it takes 6 iterations for x *and* y to converge to 1, it will take 7 iterations for of $f(x,y)$ to converge to the global minimum at $f(x,y) = 0$, with the value of $f(x,y)$ being negligibly small at 6 iterations. These observations are reflected in Fig. 5 and 6. as shown below.

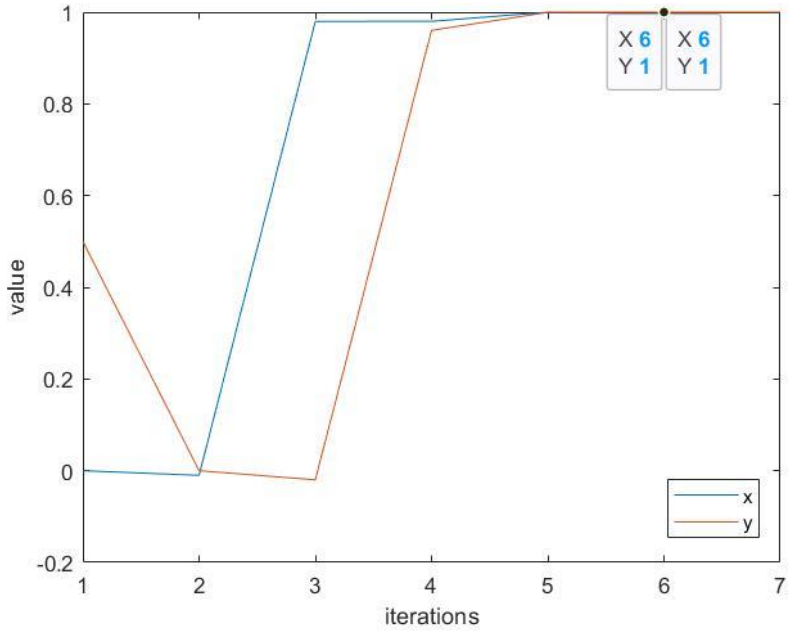

Fig. 5 x and y plotted against number of iterations

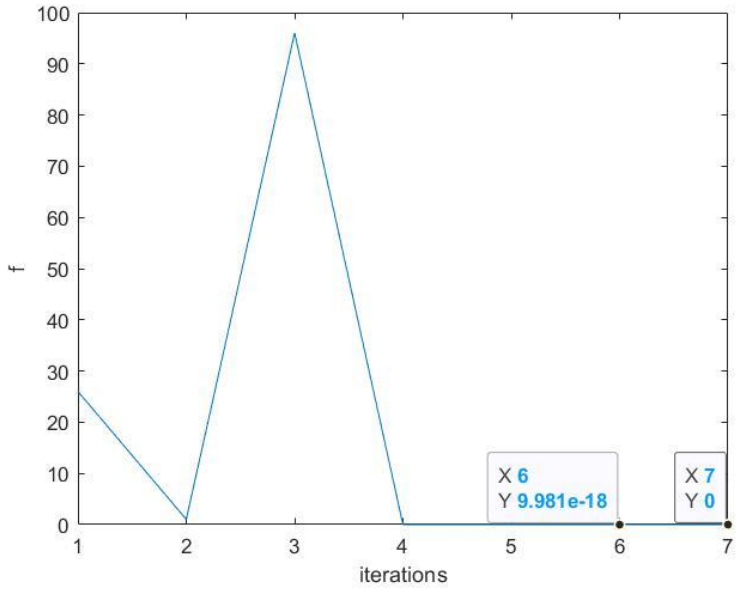

Fig. 6: $f(x, y)$ plotted against number of iterations

Fig. 7 to 9 show the trajectory of $f(x, y)$ as it approaches the global minimum. Yet, it should be noted that the Newton's Method here took significantly less iterations than that of gradient descent. As the dimensions of the matrix was sufficiently small and the computations of the second derivatives were done off-line, the significant cost of computing the Hessian matrix, $H(n)$ was not observable.

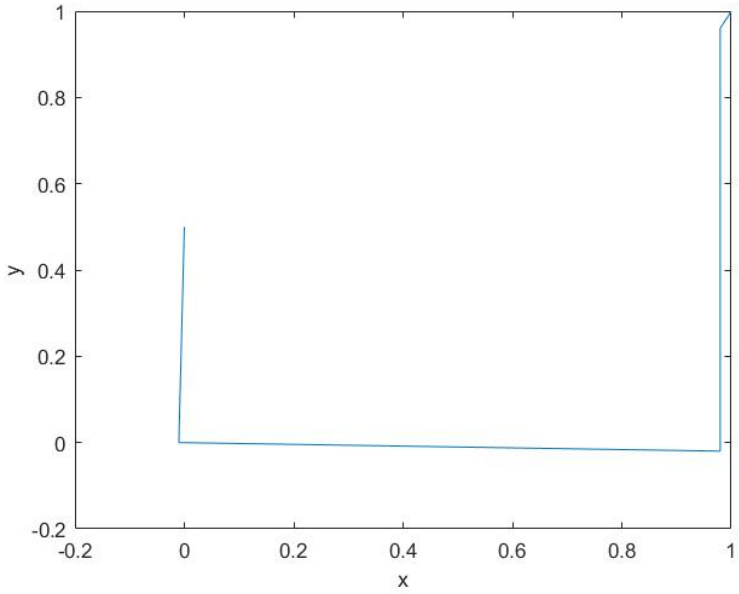

Fig. 7: Trajectories of x and y without contour plot

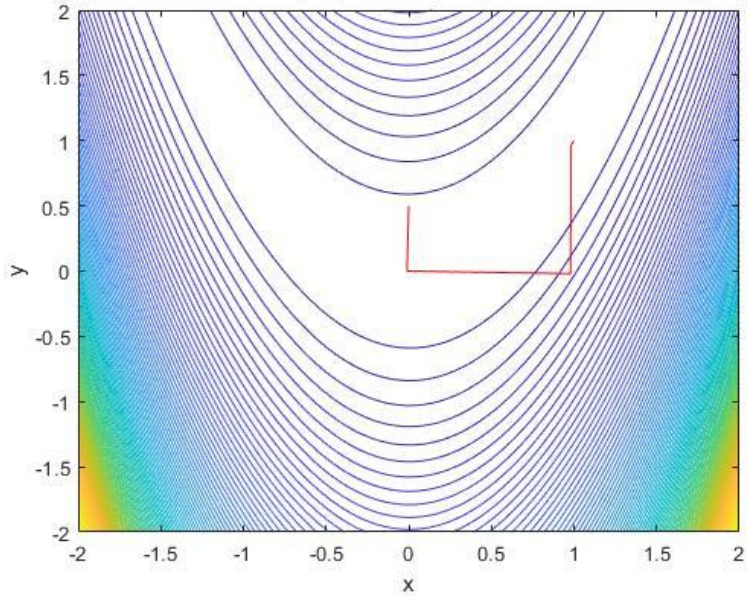

Fig. 8: Trajectories of x and y with 2D contour plot

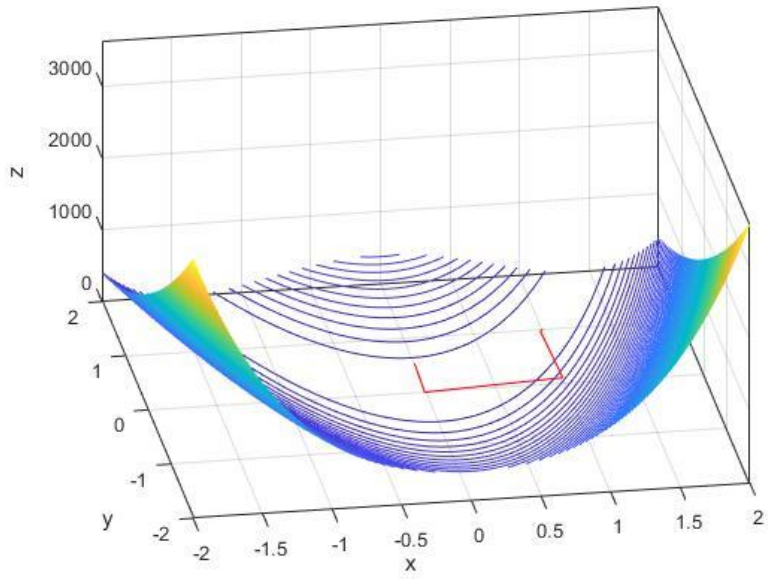

Fig. 8: Trajectories of x and y in 3D contour plot

Newton's Method Code.

```
X = [0; 0.5];
iter_count = 0;
eta = 1e-3;
stop = 0;
iters = 1e20;
flag = 1;
i = 1;

x = zeros();
y = zeros();
z = zeros();
errors = zeros();
iterations = [];

while (flag && i <= iters)

    x(i) = X(1);
    y(i) = X(2);

    X_prev = X;
    del_X = -inv(rosenHess(X(1), X(2))) * rosenGrad(X(1), X(2));
    X = X + del_X;
    error = norm(X - X_prev);

    errors(i) = error;
    iterations(i) = i;

    if (stop >= error)
        flag = 0;
    end

    i = i + 1;
end
```

Plotting code:

```
figure;
x_values = [-2:0.01:2]; y_values = [-2:0.01:2];
f = @(x,y) (1-x).^2 + 100*(y-x.^2).^2;
[xx, yy] = meshgrid(x_values, y_values);
ff = f(xx,yy);
contour(xx, yy, ff, 100);

hold on;
plot(x, y, 'r');
xlabel('x');
ylabel('y');

for i=1:length(x)
   z(i) = rosen(x(i), y(i));
end

figure;
contour3(xx, yy, ff, 100);
hold on;
plot3(x, y, z, 'r');
xlabel('x');
ylabel('y');
zlabel('z');

figure;
plot(iterations, x, iterations, y);
xlabel('iterations');
ylabel('value');
legend({'x', 'y'}, 'Location', 'northeast');

figure;
plot(x, y);
xlabel('x');
ylabel('y');

figure;
plot(iterations, z);
xlabel('iterations');
ylabel('f');
```

Question 2:

It is requested that the following function below is to be approximated:

$$y = 1.2\sin(\pi x) - \cos(2.4\pi x)$$

The code below shows the calculation of this function:

```
function val = fnTrigo(x)
    val = 1.2*sin(pi*x) - cos(2.4*pi*x);
end
```

The training domain is given below:

$$x \in [-1,1], with\ uniform\ step\ length\ of\ 0.05$$

The validation domain is given below:

$$x \in [-1,1], with\ uniform\ step\ length\ of\ 0.01$$

The extrapolated domain is given below:

$$x \in [-3,3], with\ uniform\ step\ length\ of\ 0.01$$

It is postulated in lectures that a guideline that can be followed to determine the number of neurons required is to follow the minimum number of components required to approximate the function. Thus, based on Fig. 9 given below, it would be expected that at least 6 hidden neurons would be required.

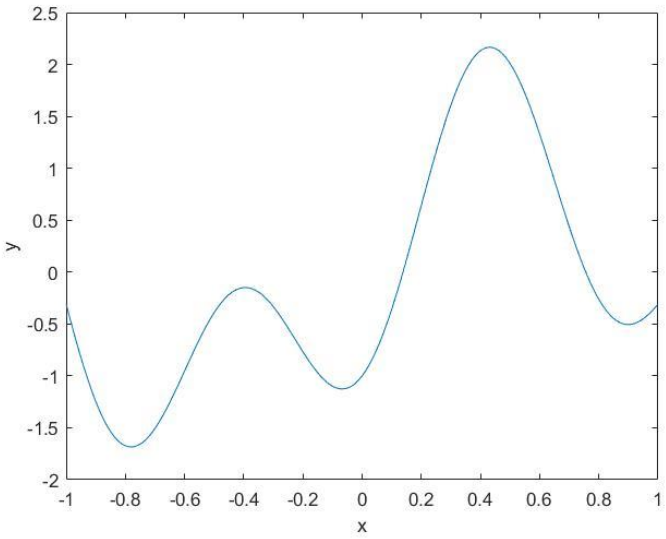

Fig. 9: Plot of $y = 1.2\sin(\pi x) - \cos(2.4\pi x)$

2a) In the following graphs from Fig. 10 to Fig. 21, $\eta = 0.005$ is used, along with the training function being *traingda*, which is known to be gradient descent with adaptive learning rate backpropagation. Notice that sequential learning of 500 epochs is used here.

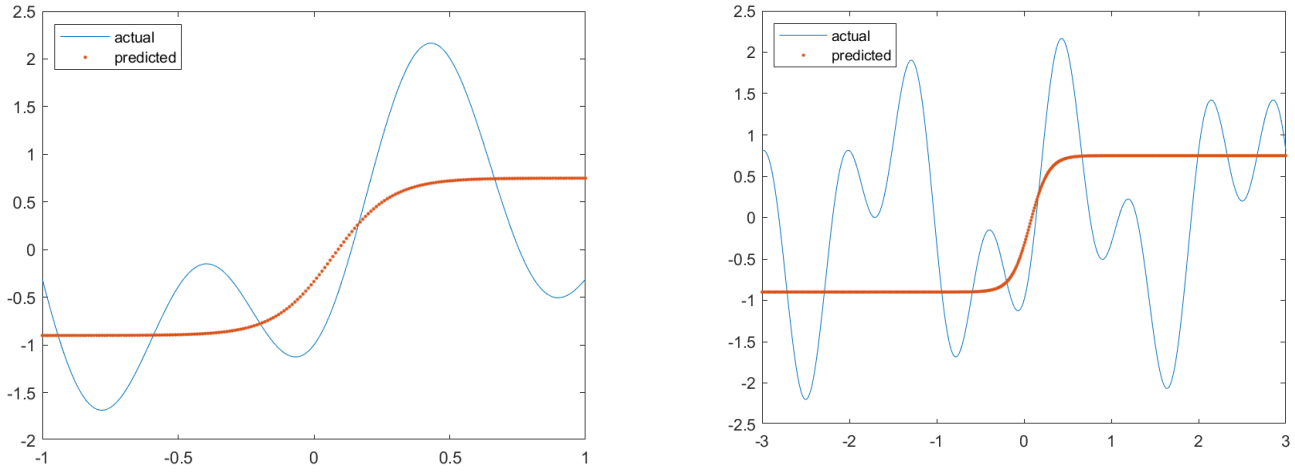

Fig. 10: Function approximation and extrapolation with 1-1-1 (traingda)

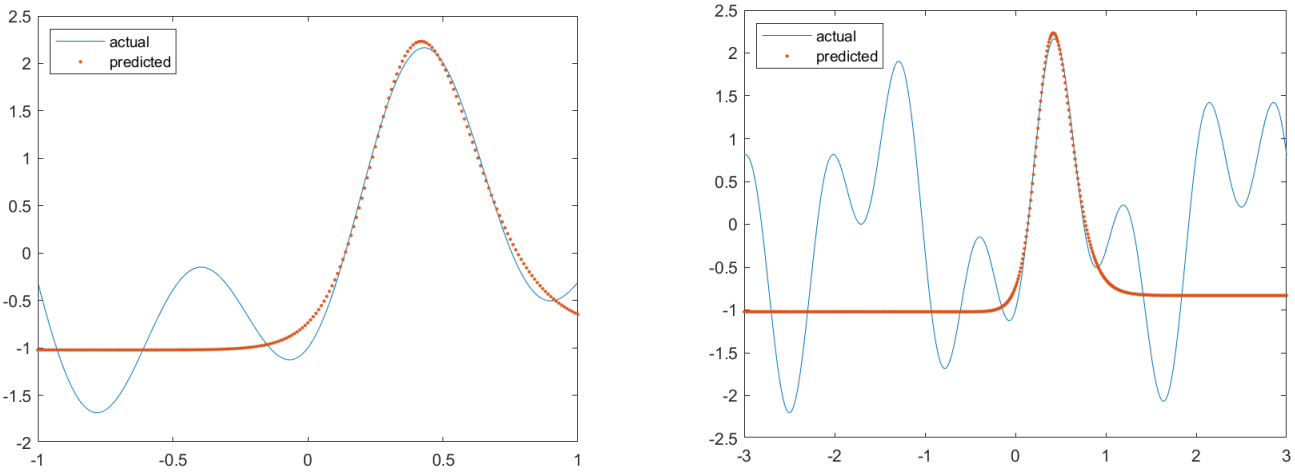

Fig. 11: Function approximation and extrapolation with 1-2-1 (traingda)

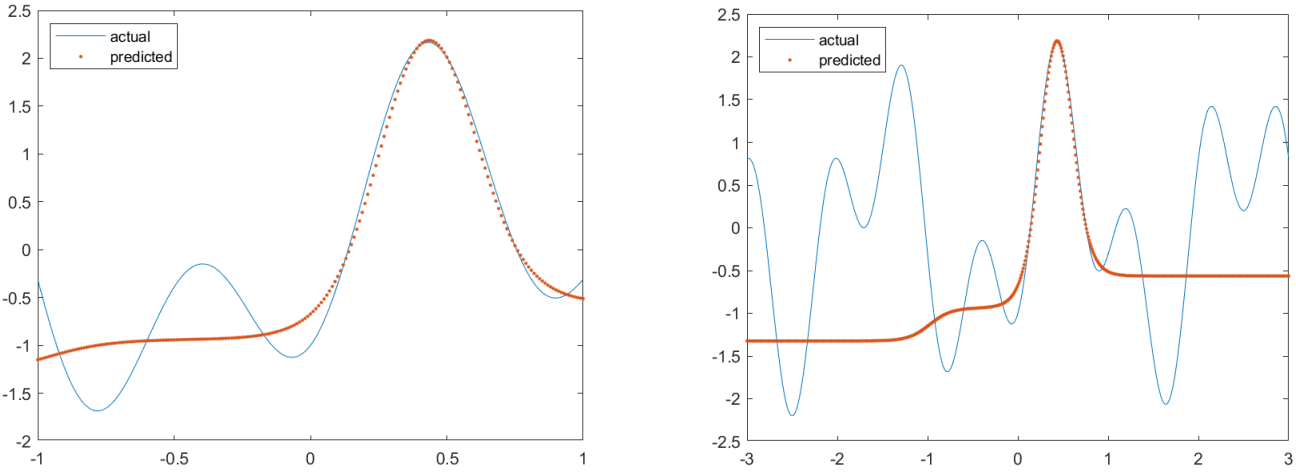

Fig. 12: Function approximation and extrapolation with 1-3-1 (traingda)

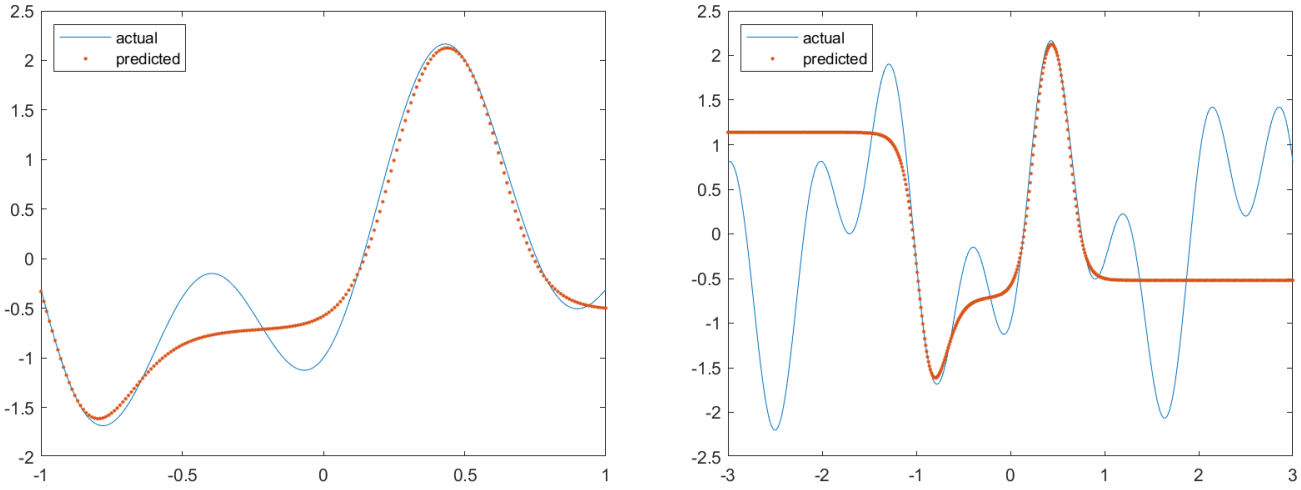

Fig. 13: Function approximation and extrapolation with 1-4-1 (traingda)

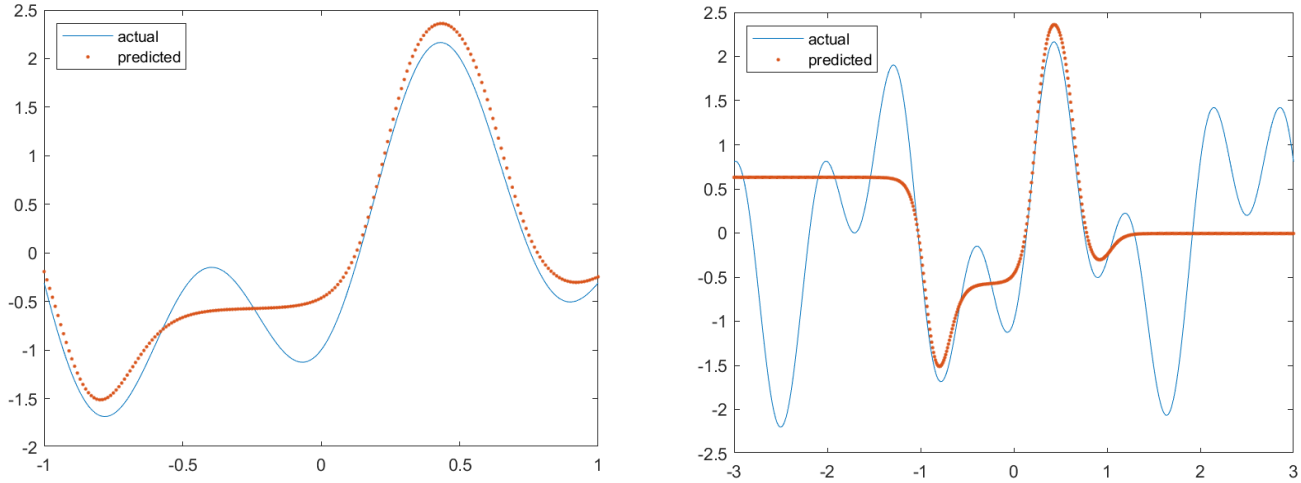

Fig. 14: Function approximation and extrapolation with 1-5-1 (traingda)

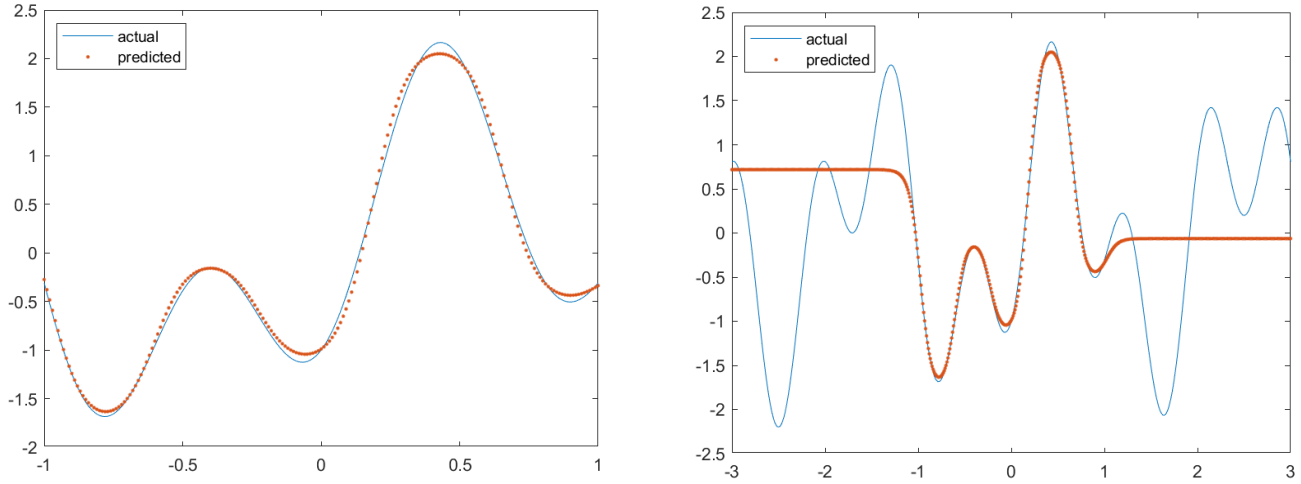

Fig. 15: Function approximation and extrapolation with 1-6-1 (traingda)

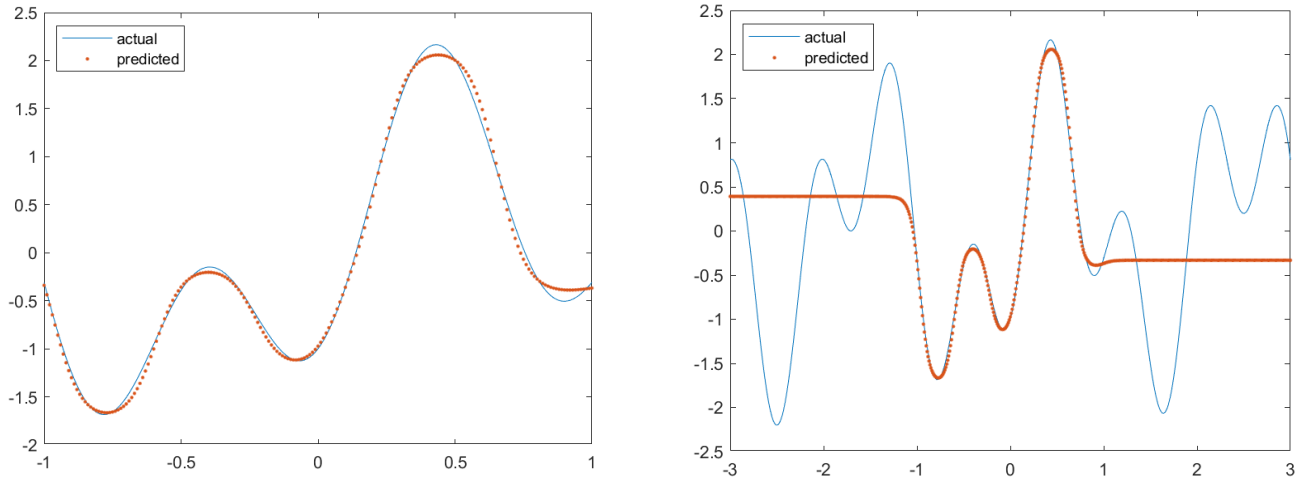

Fig 16: Function approximation and extrapolation with 1-7-1 (traingda)

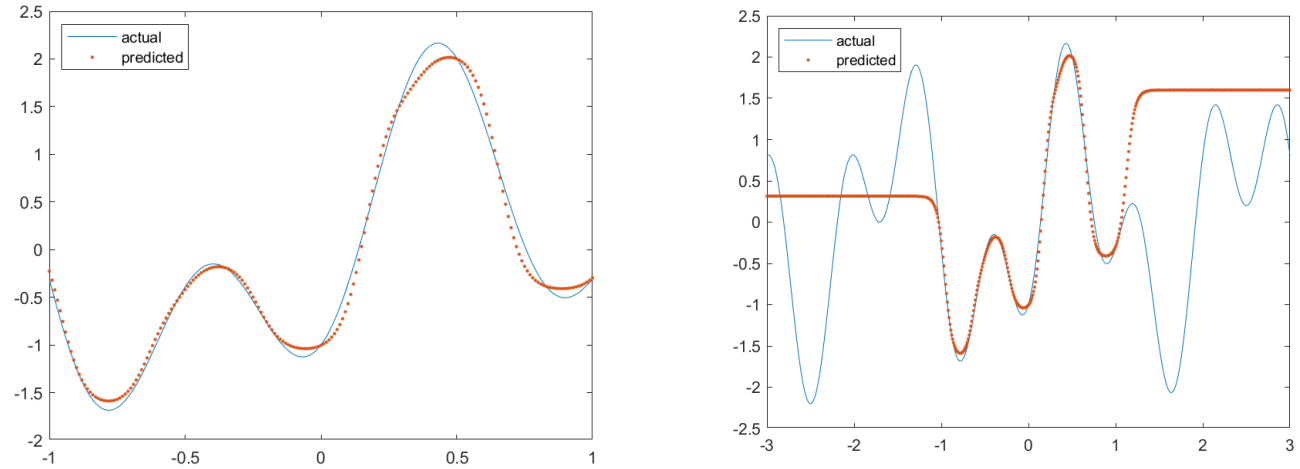

Fig. 17: Function approximation and extrapolation with 1-8-1 (traingda)

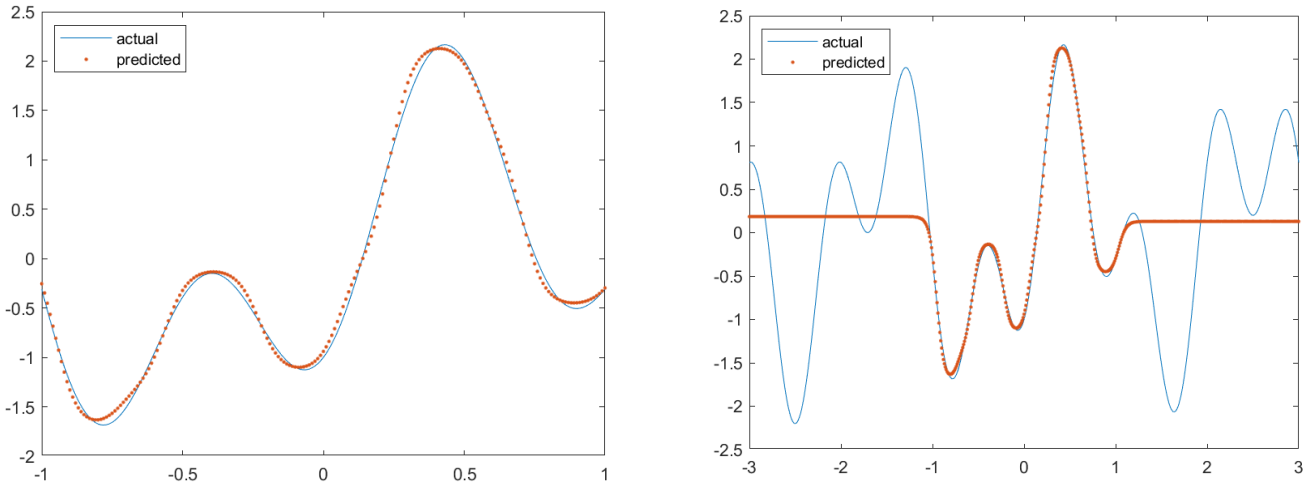

Fig. 18: Function approximation and extrapolation with 1-9-1 (traingda)

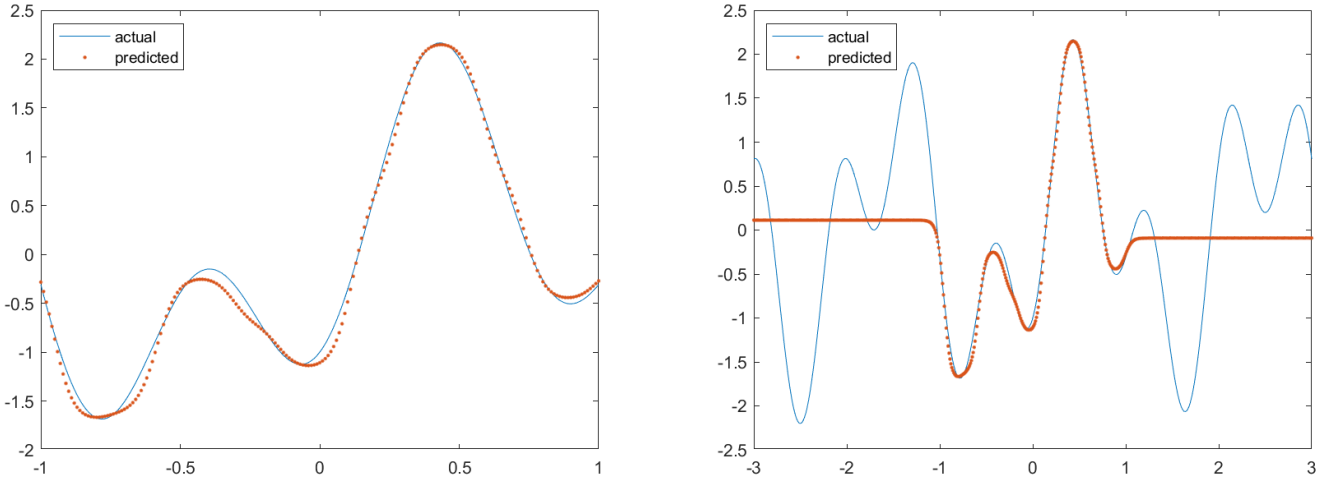

Fig. 19: Function approximation and extrapolation with 1-10-1 (traingda)

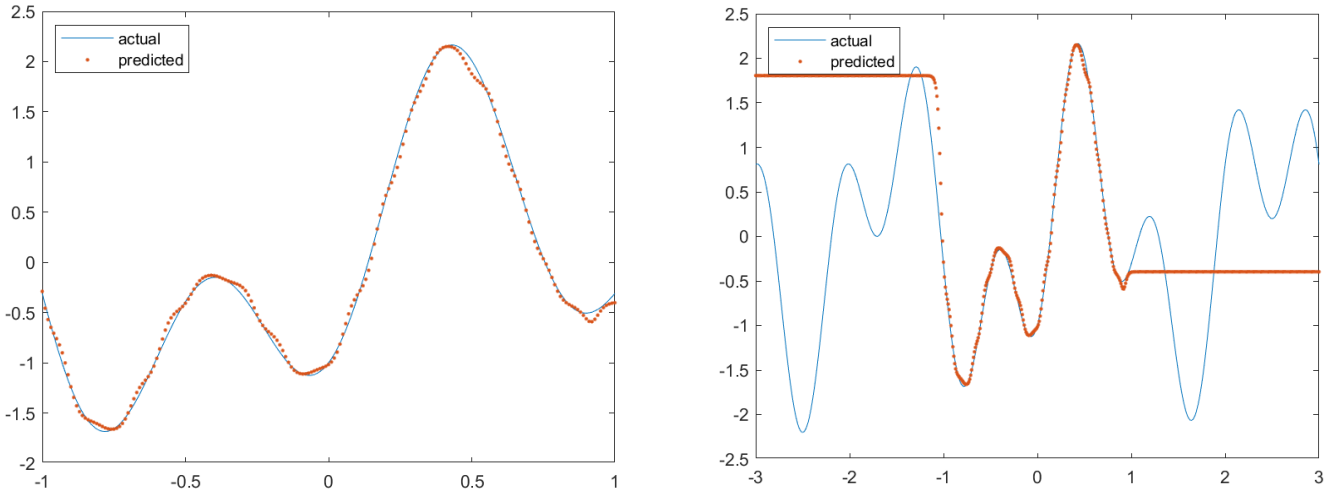

Fig. 20: Function approximation and extrapolation with 1-20-1 (traingda)

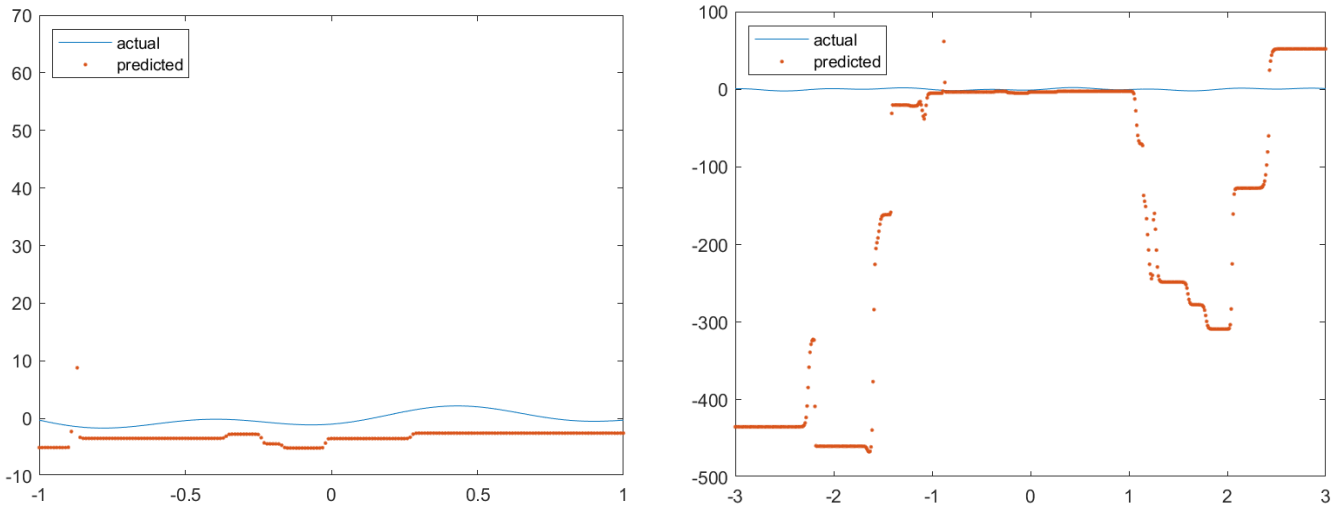

Fig. 21: Function approximation and extrapolation with 1-50-1 (traingda)

Based on the plots from Fig. 10 to 21, if 1 to 5 hidden neurons were implemented, under-fitting would occur and for the case of 6 to 9 hidden neurons were implemented, the proper-fitting would occur. For 10 and 20 hidden neurons, there appears to be more errors despite the predictions seemingly being largely accurate. This indicates that overfitting is beginning to occur. When 50 hidden neurons are implemented, the approximation is clearly inaccurate; overfitting has definitely occurred in this case. The plots also indicate that the multi-layer perceptron can only predict and approximate a function within the range provided in the training data; it is unable to make reasonable predictions outside of the domain of the input limited by the training set.

Keeping $\eta = 0.005$ and number of epochs at 500, for the case of Fig. 22 to 33, the training function *traingdx* is used. The key difference between *traingda* and *traingdx*, despite both being gradient descent algorithms with adaptive learning rate backpropagation, is that *traingdx* accounts for momentum, which adds inertia to the update procedure, in turn accelerating the learning procedure.

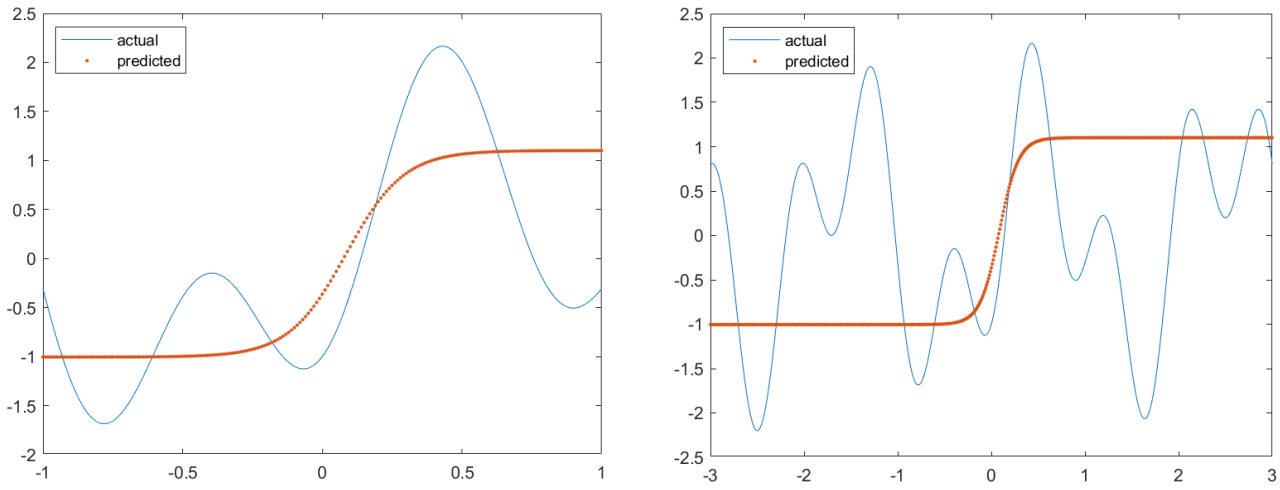

Fig. 22: Function approximation and extrapolation with 1-1-1 (traingdx)

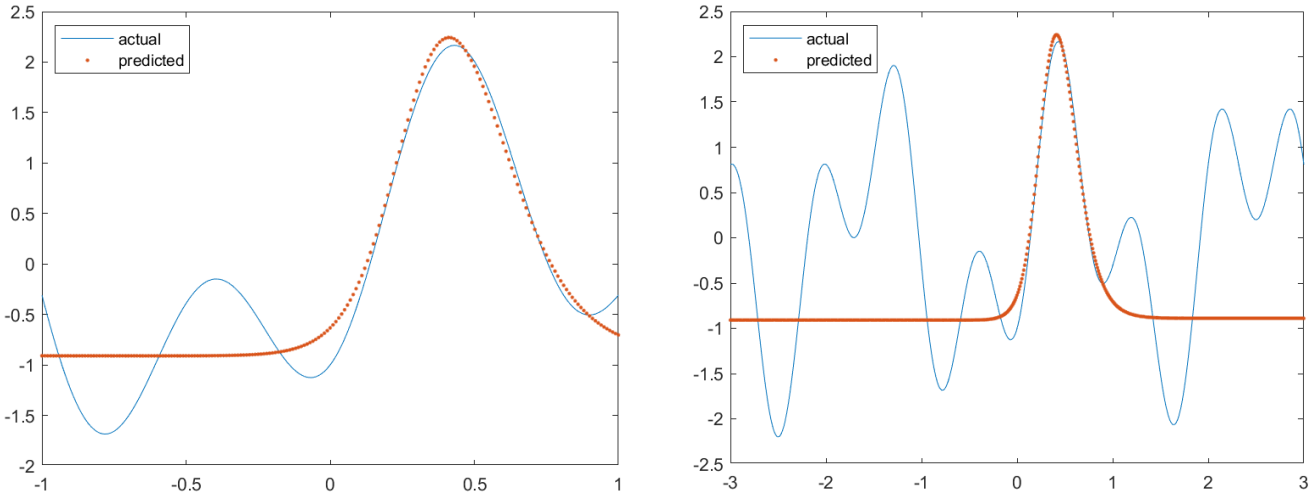

Fig. 23: Function approximation and extrapolation with 1-2-1 (traingdx)

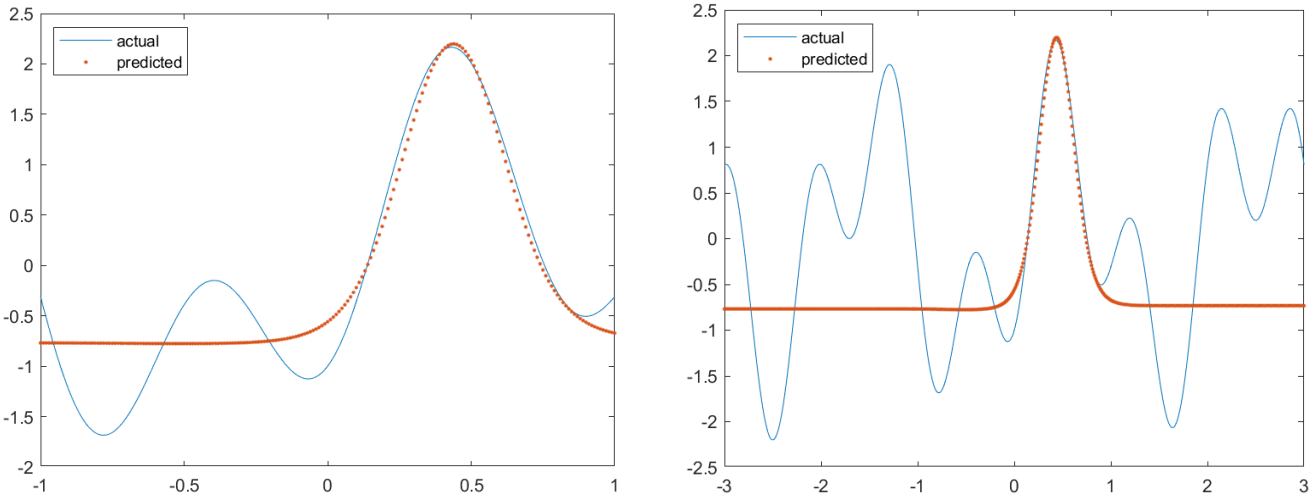

Fig. 24: Function approximation and extrapolation with 1-3-1 (traingdx)

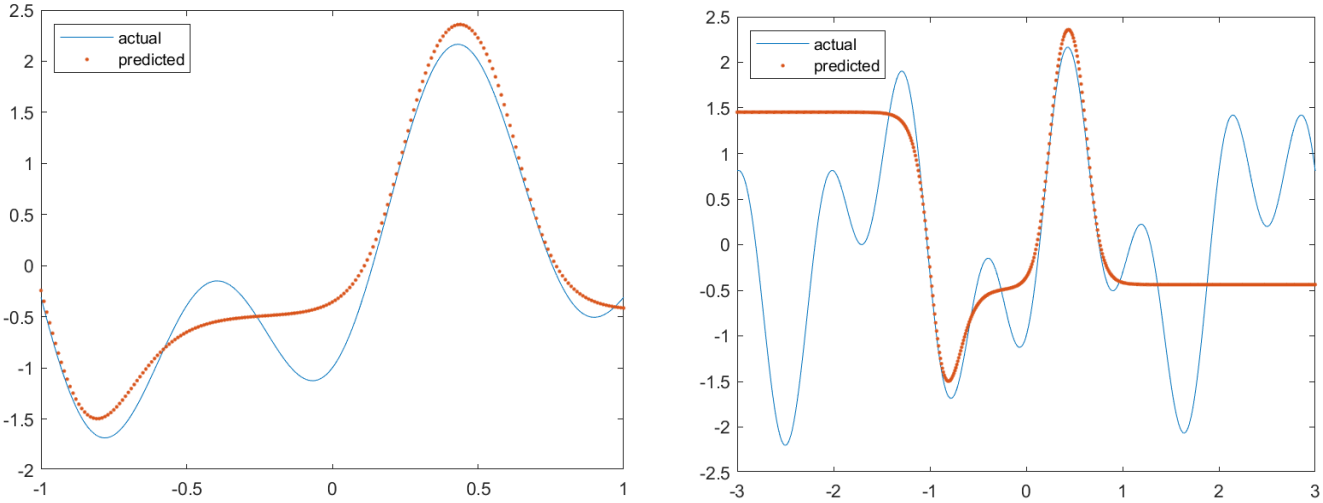

Fig. 25: Function approximation and extrapolation with 1-4-1 (traingdx)

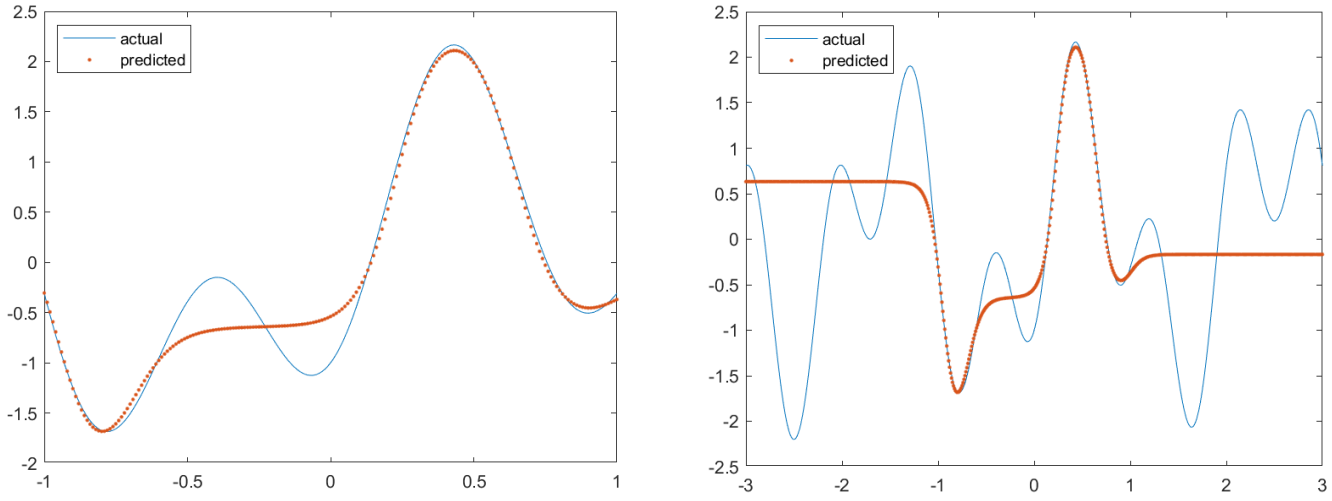

Fig. 26: Function approximation and extrapolation with 1-5-1 (traingdx)

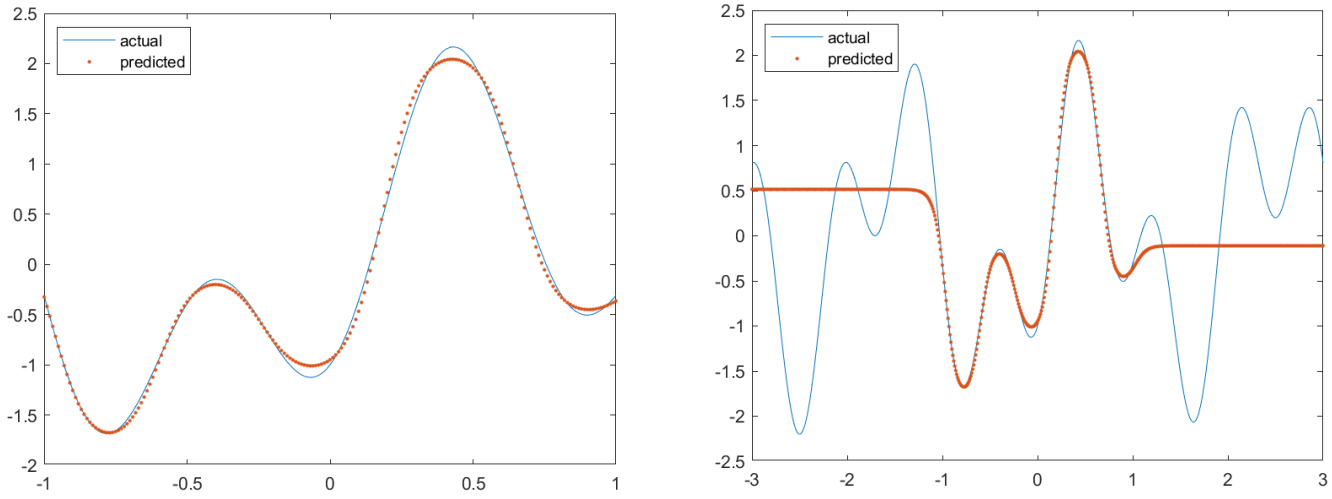

Fig. 27: Function approximation and extrapolation with 1-6-1 (traingdx)

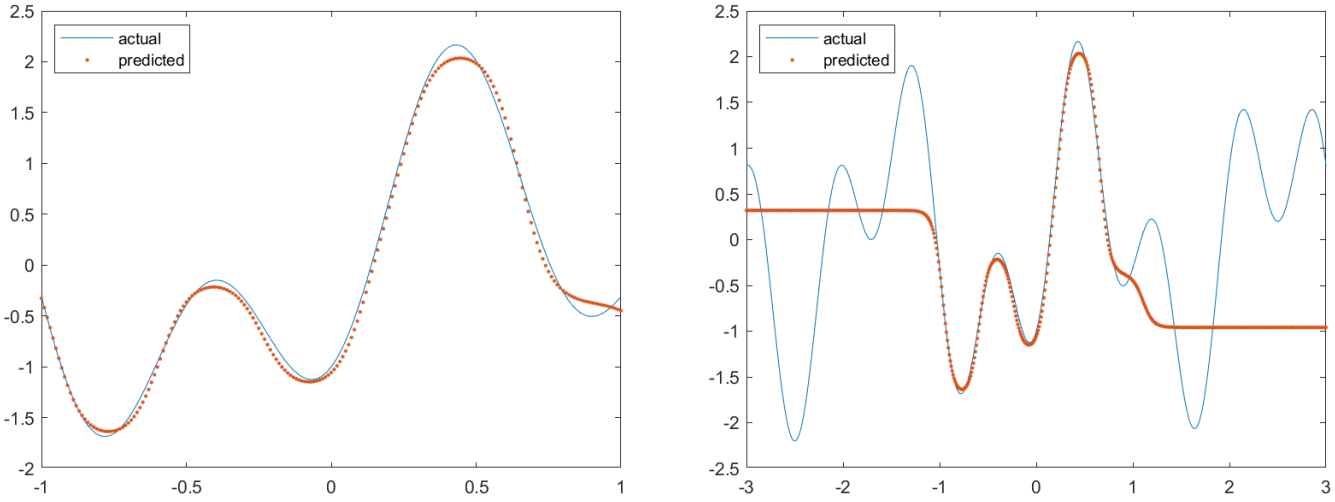

Fig. 28: Function approximation and extrapolation with 1-7-1 (traingdx)

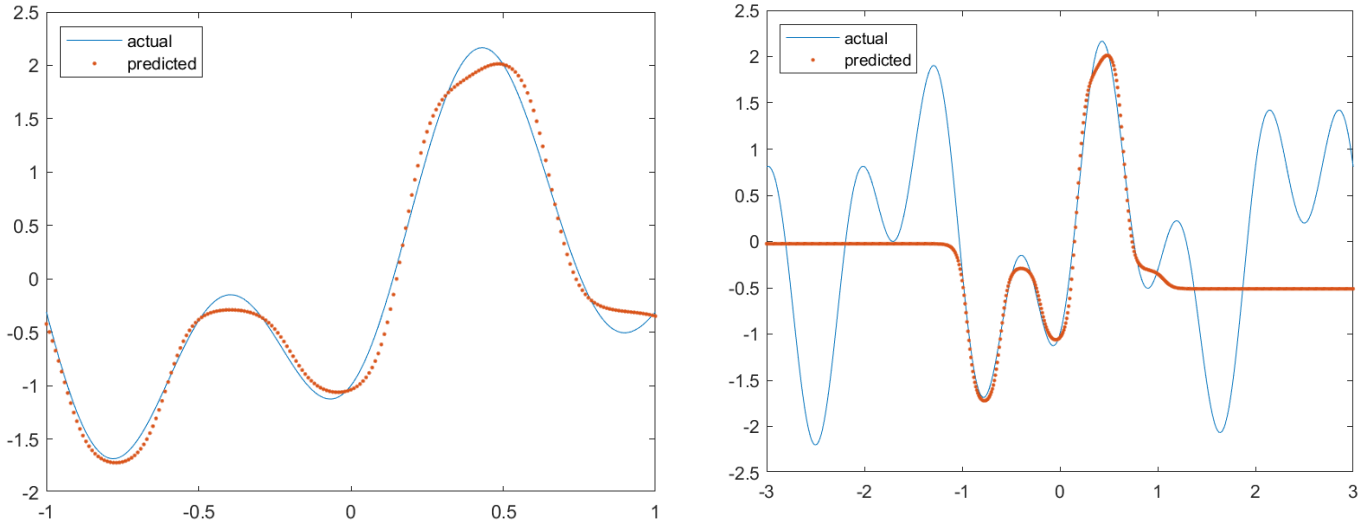

Fig. 29: Function approximation and extrapolation with 1-8-1 (traingdx)

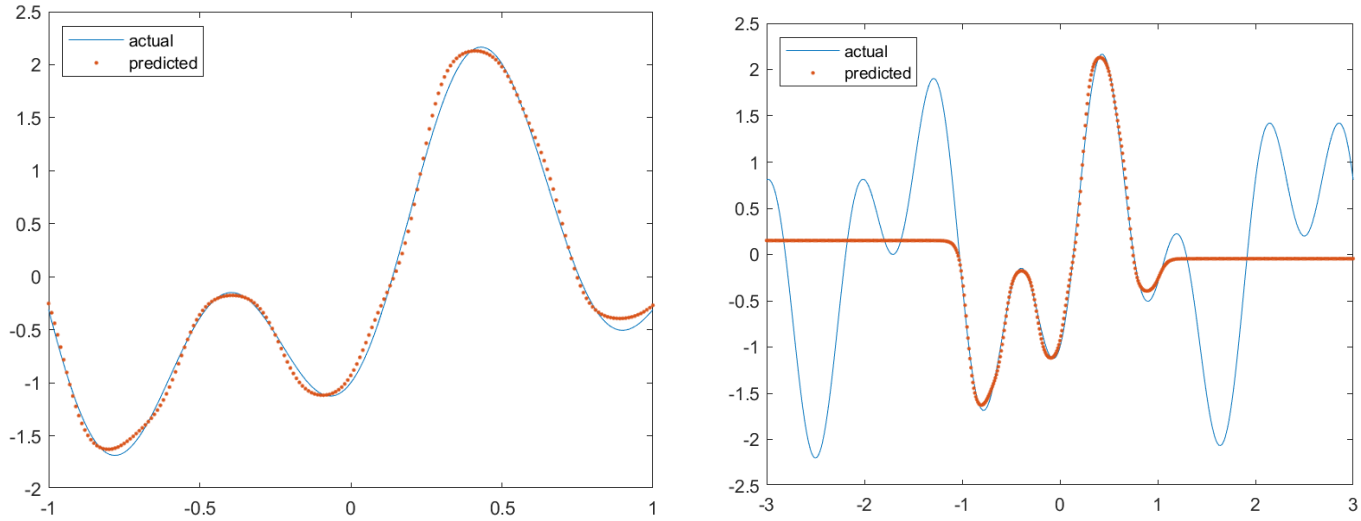

Fig. 30: Function approximation and extrapolation with 1-9-1 (traingdx)

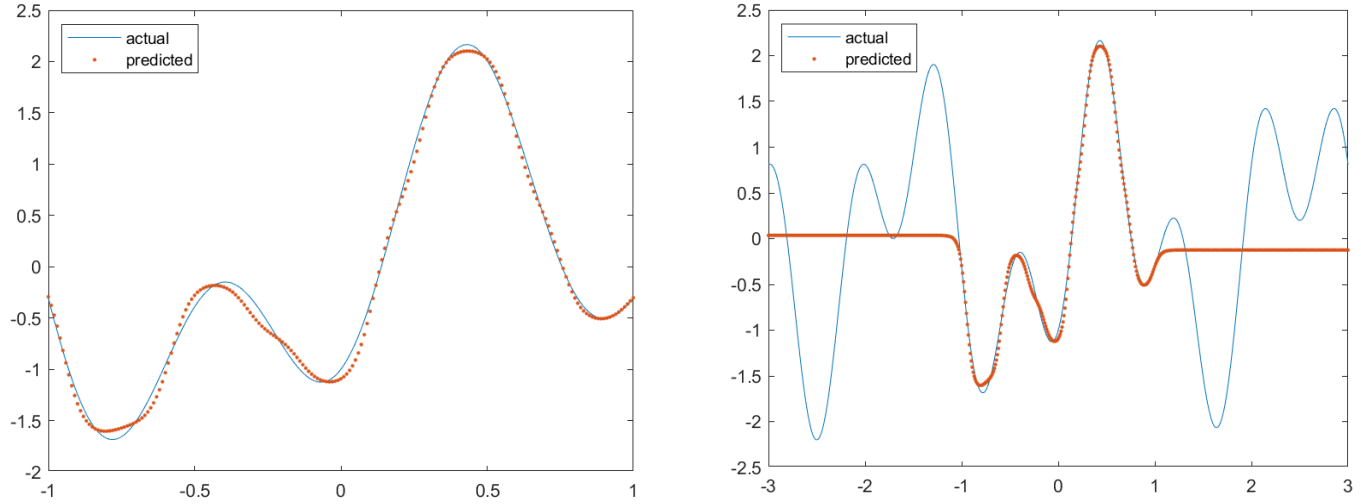

Fig. 31: Function approximation and extrapolation with 1-10-1 (traingdx)

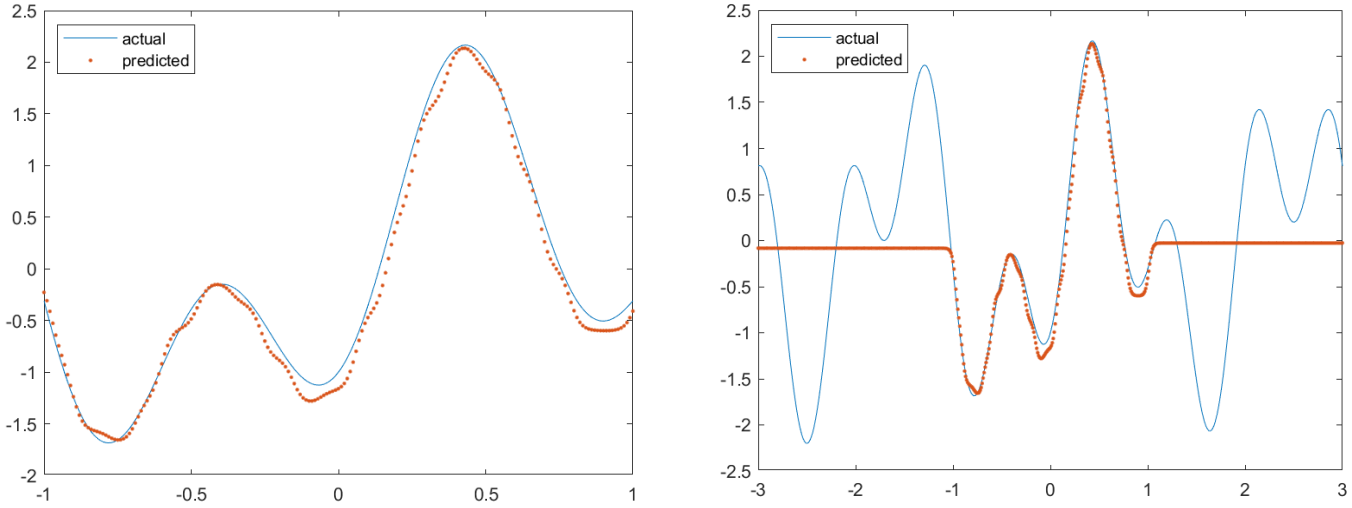

Fig. 32: Function approximation and extrapolation with 1-20-1 (traingdx)

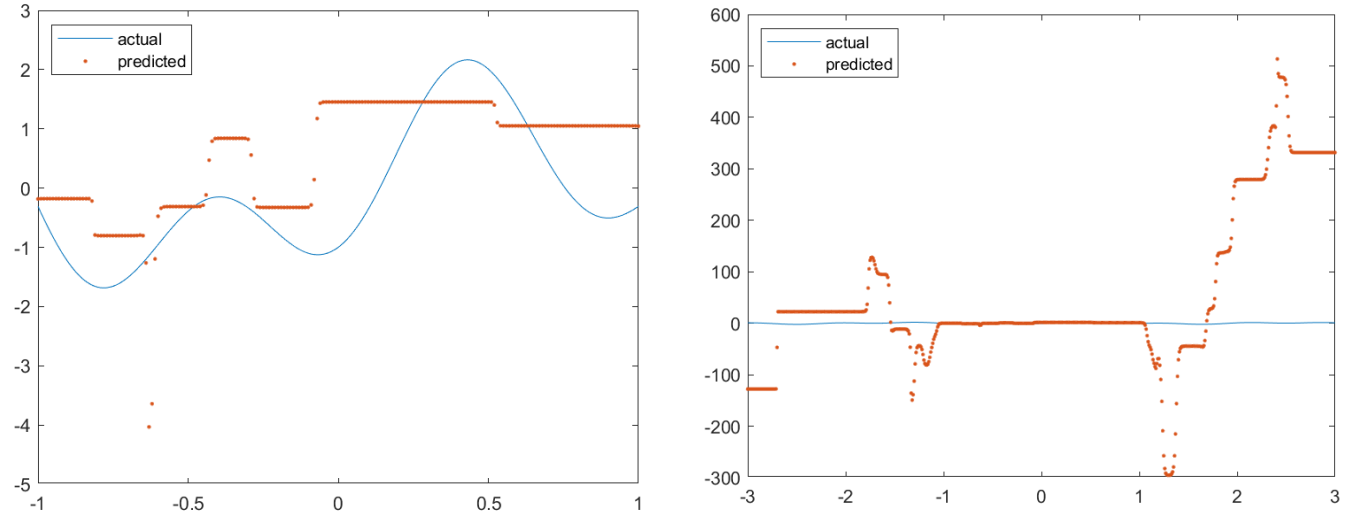

Fig. 33: Function approximation and extrapolation with 1-50-1 (traingdx)

It can be observed that when *traingdx* is used instead, it appears to suffer from more errors as compared to *traingda* under the same learning rate and number of epochs. For the case of 1 to 5 hidden neurons, the multi-layer perceptron is clearly under-fitted. Yet, when 6 to 10 hidden neurons are used in *traingdx*, the function is proper-fitted. When 20 and 50 hidden neurons are applied, it is evident that over-fitting has occurred.

The predictions obtained for both training functions indicate that the proposition that the number of line components approximated can be used as a reference to predict the minimum number of neurons is thus affirmed to a large extent. Just as it was estimated that at least 6 hidden neurons were required, at least 6 but not more than 10 hidden neurons appeared to be required for the proper fitting.

Training loop code. Refer to next page for plotting code.

```
clear;

train_delta = 5e-2;
val_delta = 1e-2;
test_delta = 1e-2;
min = -1;
max = 1;
test_min = -3;
test_max = 3;
epochs = 5e2;
hidden_neurons = [1,2,3,4,5,6,7,8,9,10,20,50];
epoch_arr = [1:1:epochs];
eta = 0.005;

% Training data
train_X = [min: train_delta: max];
train_X_label = fnTrigo(train_X);

% Validation data
val_X = [min: val_delta: max];
val_X_ans = fnTrigo(val_X);

% Test data
test_X = [test_min: test_delta: test_max];
test_X_ans = fnTrigo(test_X);

for i = 1: length(hidden_neurons)

    net = train_seq(hidden_neurons(i), train_X, train_X_label, epochs, eta);
    val_train = net(val_X);
    test_train = net(test_X);

    filename_plot_val = sprintf("q2a_change_hidden\\plot_hidden_%d",
hidden_neurons(i));
    filename_scatter_val = sprintf("q2a_change_hidden\\scatter_hidden_%d",
hidden_neurons(i));
    filename_plot_test = sprintf("q2a_extrapolate\\plot_hidden_%d",
hidden_neurons(i));
    filename_scatter_test = sprintf("q2a_extrapolate\\scatter_hidden_%d",
hidden_neurons(i));

    % Plot code was originally here

end
```

Training sequence code:

```
function [net, val_train, test_train] = train_seq(n, train_X, train_X_output,
epochs, eta)

    x_train = num2cell(train_X);
    x_train_label = num2cell(train_X_output);

    display("number of hidden neurons:", num2str(n)); % for message passing
    display("number of epochs:", num2str(epochs));    % for message passing

    net = fitnet(n, 'traingda'); % can change to traingdx.

    net.divideParam.trainRatio = 1;
    net.divideParam.valRatio = 0;
    net.divideParam.testRatio = 0;
    net.layers{1}.transferFcn = 'tansig';
    net.layers{2}.transferFcn = 'purelin';
    net.trainParam.lr = eta;

    for i=1:epochs
        idx = randperm(length(x_train));
        net = adapt(net, x_train(:, idx), x_train_label(:, idx));
    end
end
```

Code used for plotting:

```
    plot(val_X, val_X_ans);
    xlabel('x');
    ylabel('y');

    plot(val_X, val_X_ans, val_X, val_train);
    xlabel('x');
    ylabel('y');
    legend({'actual', 'predicted'}, 'Location', 'northwest');
    saveas(gcf, filename_plot_val, 'png');

    plot(val_X, val_X_ans);
    hold on;
    scatter(val_X, val_train, '.');
    legend({'actual', 'predicted'}, 'Location', 'northwest');
    saveas(gcf, filename_scatter_val, 'png');
    hold off;

    plot(test_X, test_X_ans, test_X, test_train);
    xlabel('x');
    ylabel('y');
    legend({'actual', 'predicted'}, 'Location', 'northwest');
    saveas(gcf, filename_plot_test, 'png');

    plot(test_X, test_X_ans);
    hold on;
    scatter(test_X, test_train, '.');
    legend({'actual', 'predicted'}, 'Location', 'northwest');
    saveas(gcf, filename_scatter_test, 'png');
    hold off;
```

2b) In the following graphs from Fig. 34 to Fig.45, 500 epochs were taken for the multi-layer perceptron training, with $\eta = 0.005$. Also, the learning method was changed to batch learning. The 'tansig' and 'purelin' transfer functions were used for the hidden and output layers respectively. The training algorithm used was *trainlm*.

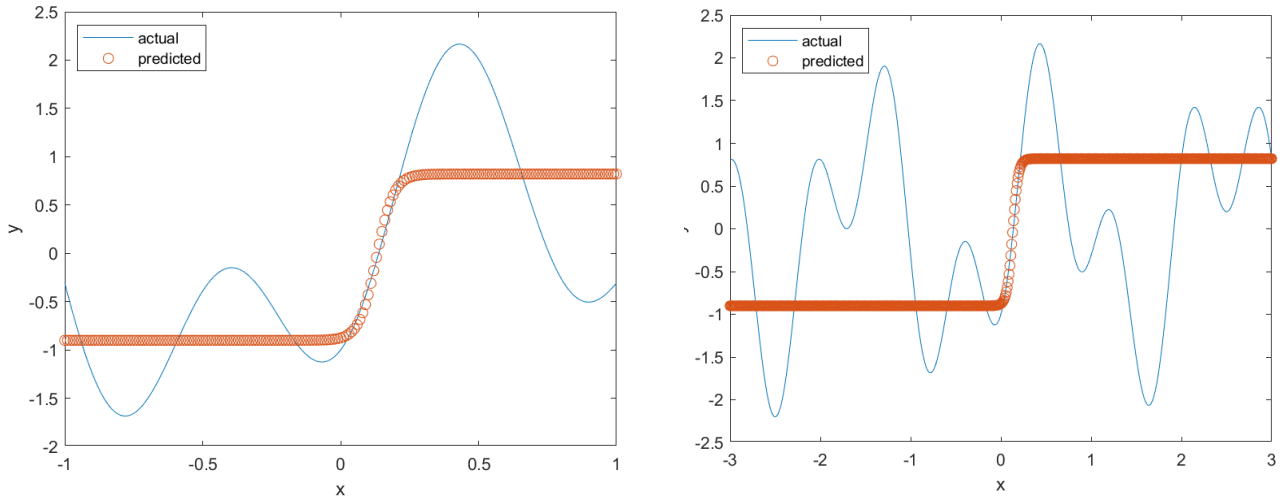

Fig. 34: Function approximation and extrapolation with 1-1-1 (trainlm)

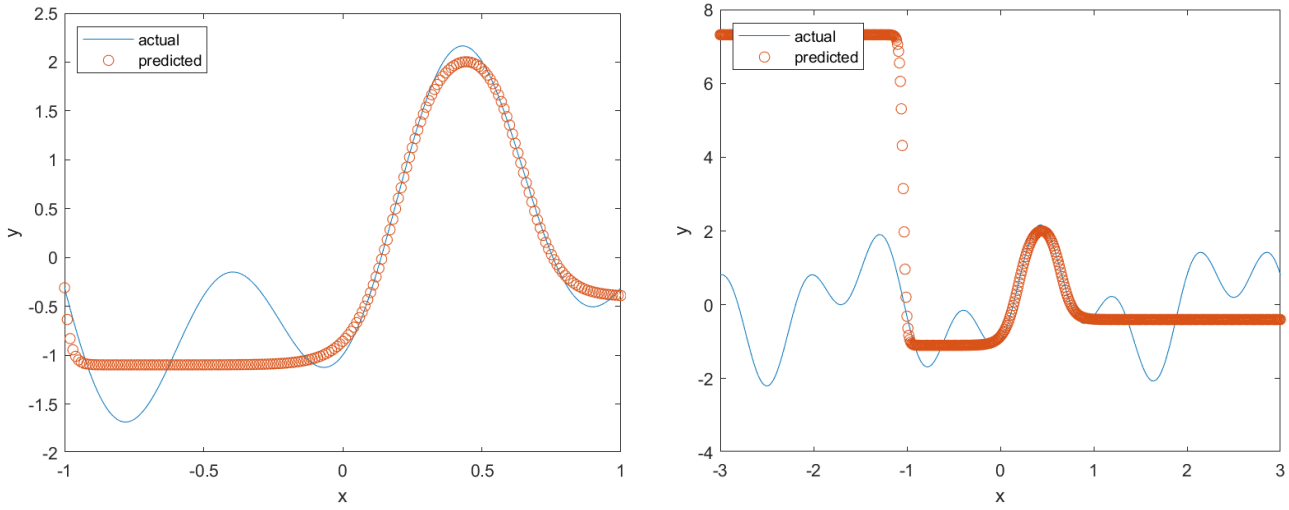

Fig. 35: Function approximation and extrapolation with 1-2-1 (trainlm)

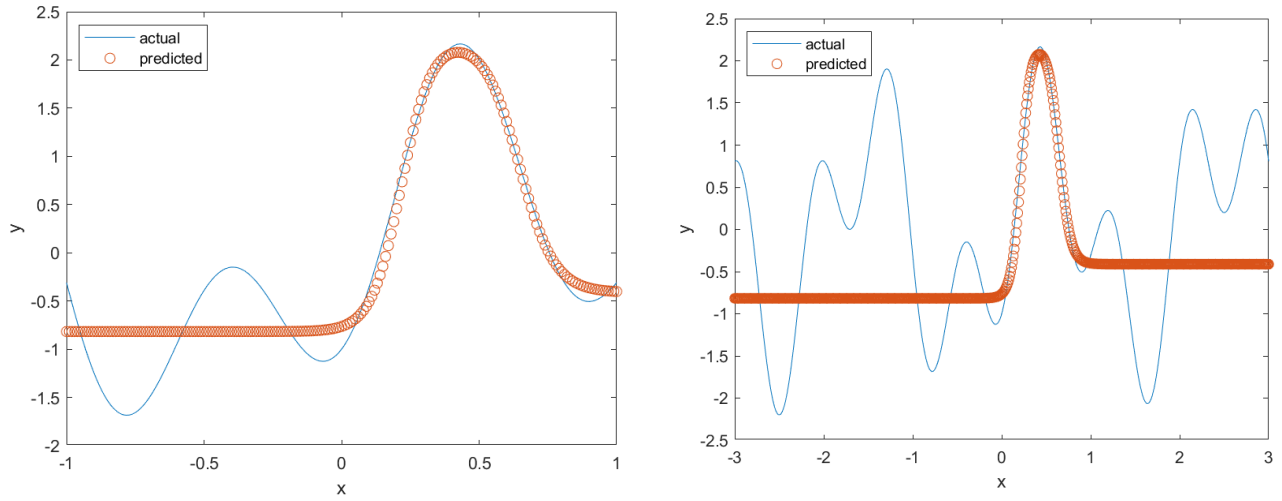

Fig. 36: Function approximation and extrapolation with 1-3-1 (trainlm)

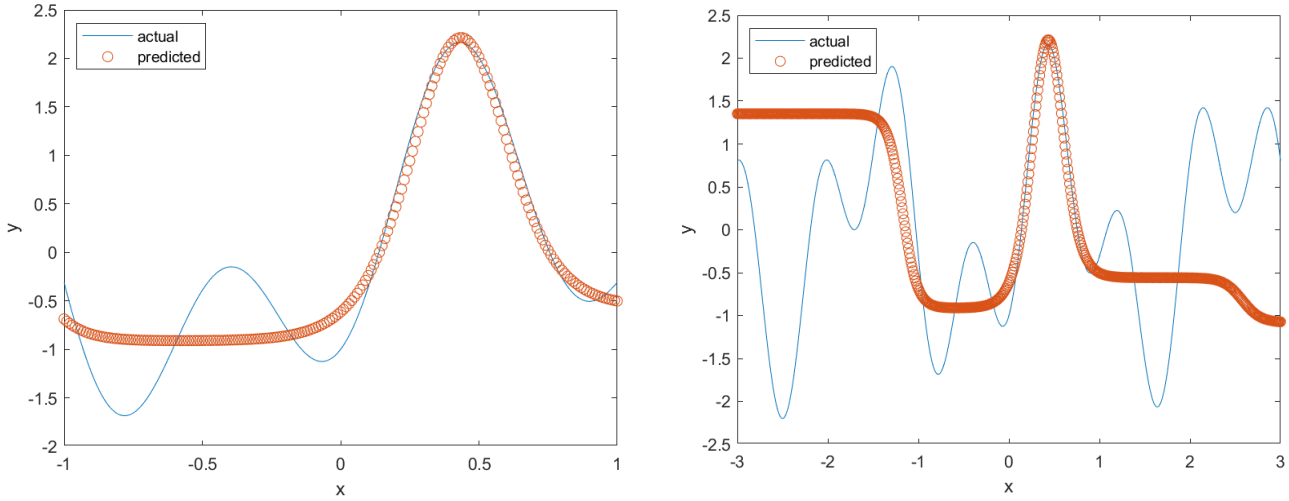

Fig. 37: Function approximation and extrapolation with 1-4-1 (trainlm)

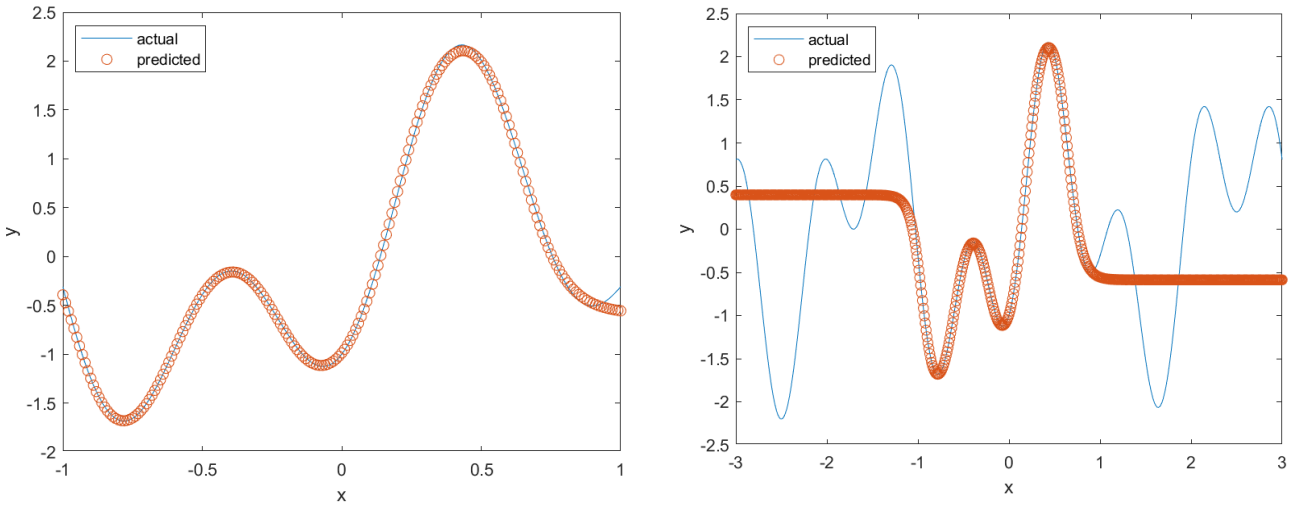

Fig. 38: Function approximation and extrapolation with 1-5-1 (trainlm)

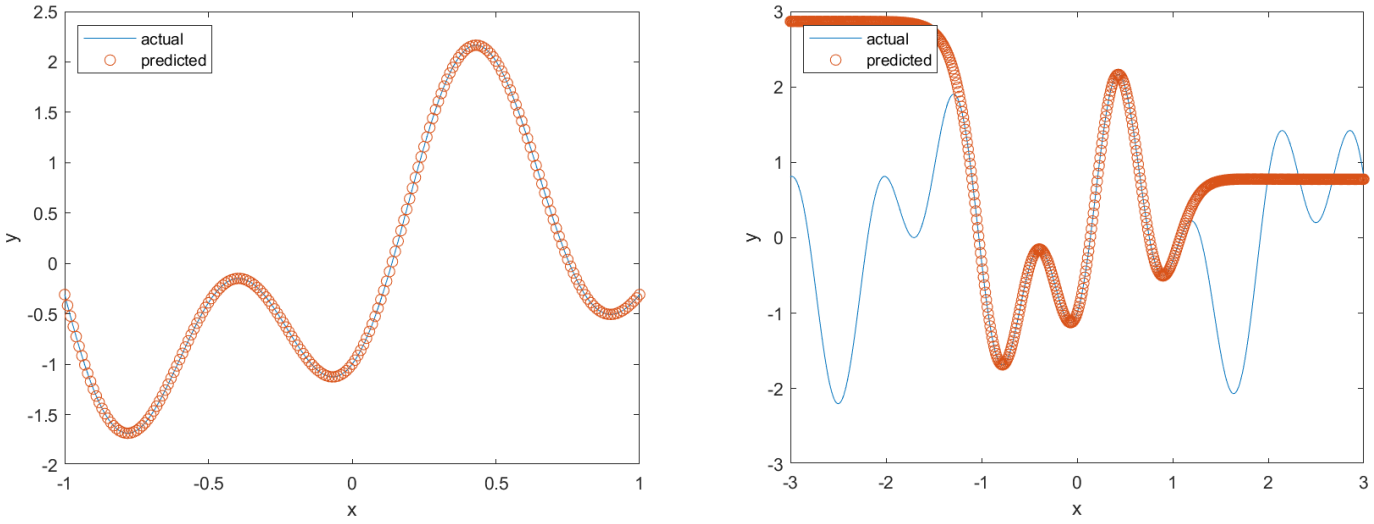

Fig. 39: Function approximation and extrapolation with 1-6-1 (trainlm)

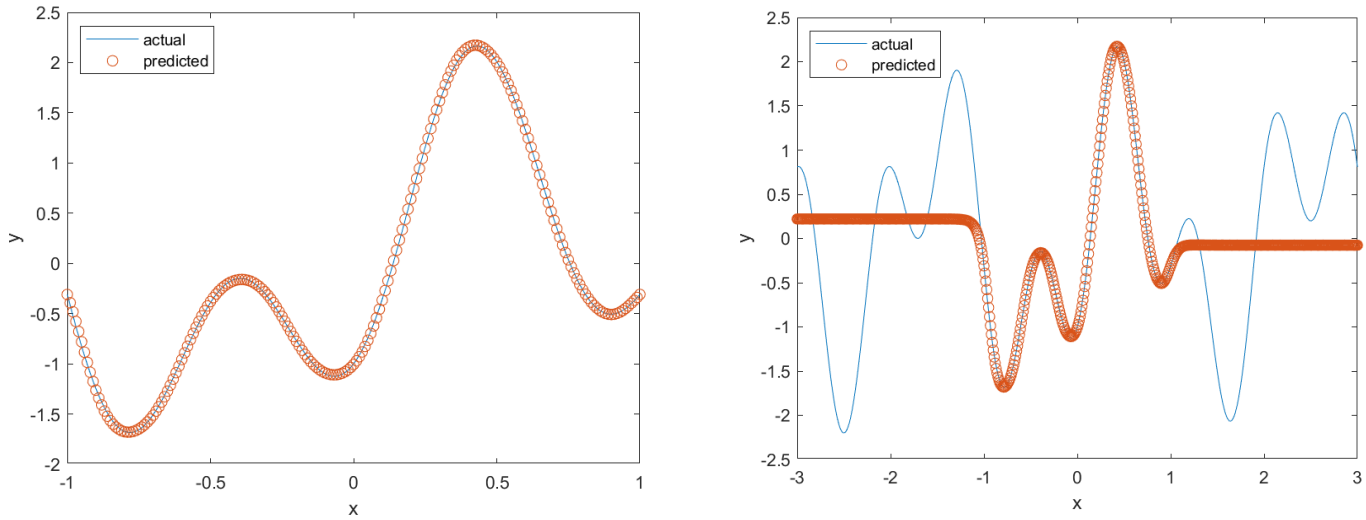

Fig. 40: Function approximation and extrapolation with 1-7-1 (trainlm)

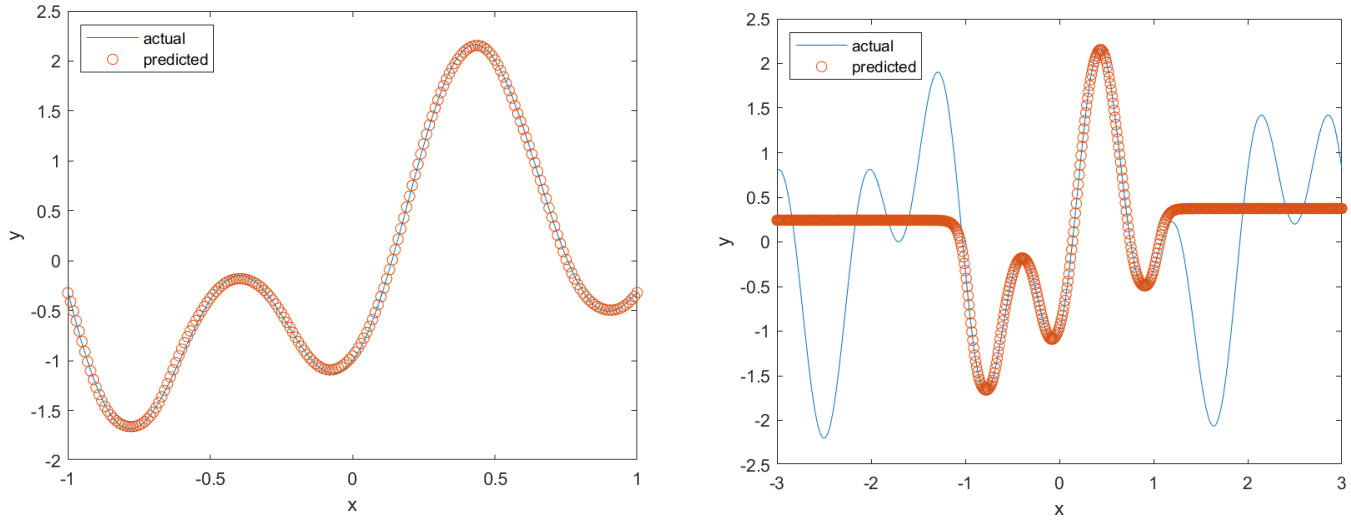

Fig. 41: Function approximation and extrapolation with 1-8-1 (trainlm)

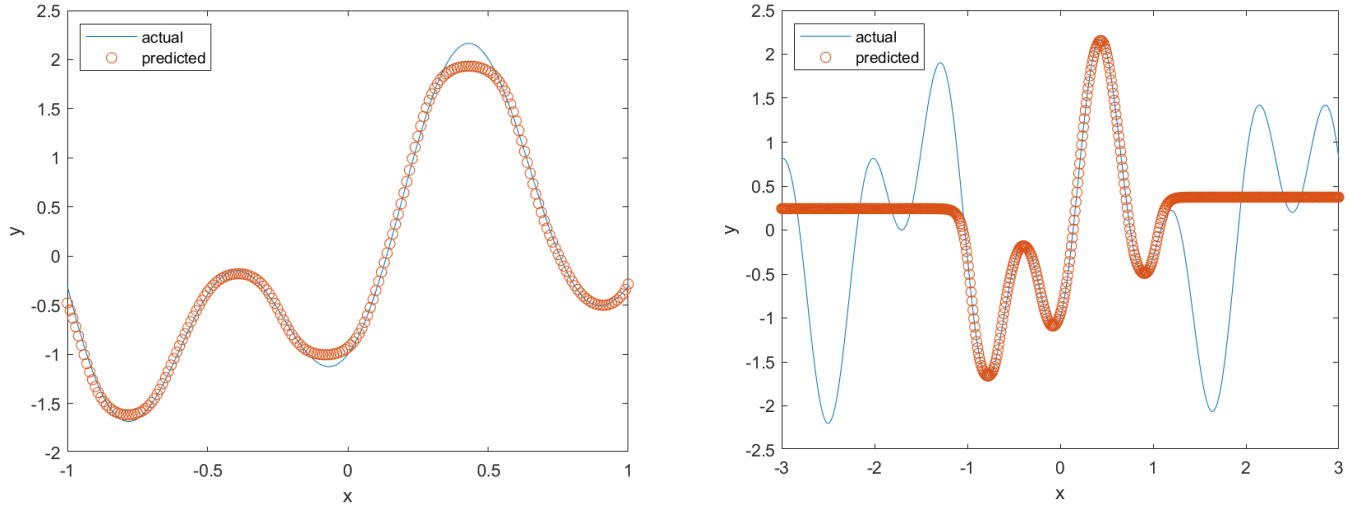

Fig. 42: Function approximation and extrapolation with 1-9-1 (trainlm)

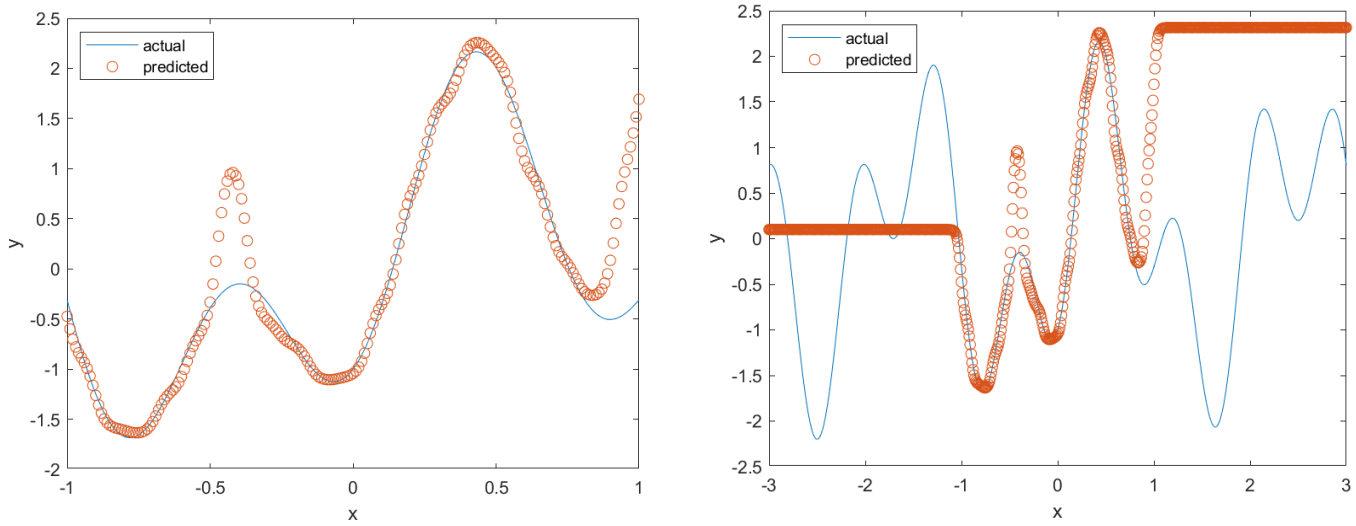

Fig. 43: Function approximation and extrapolation with 1-10-1 (trainlm)

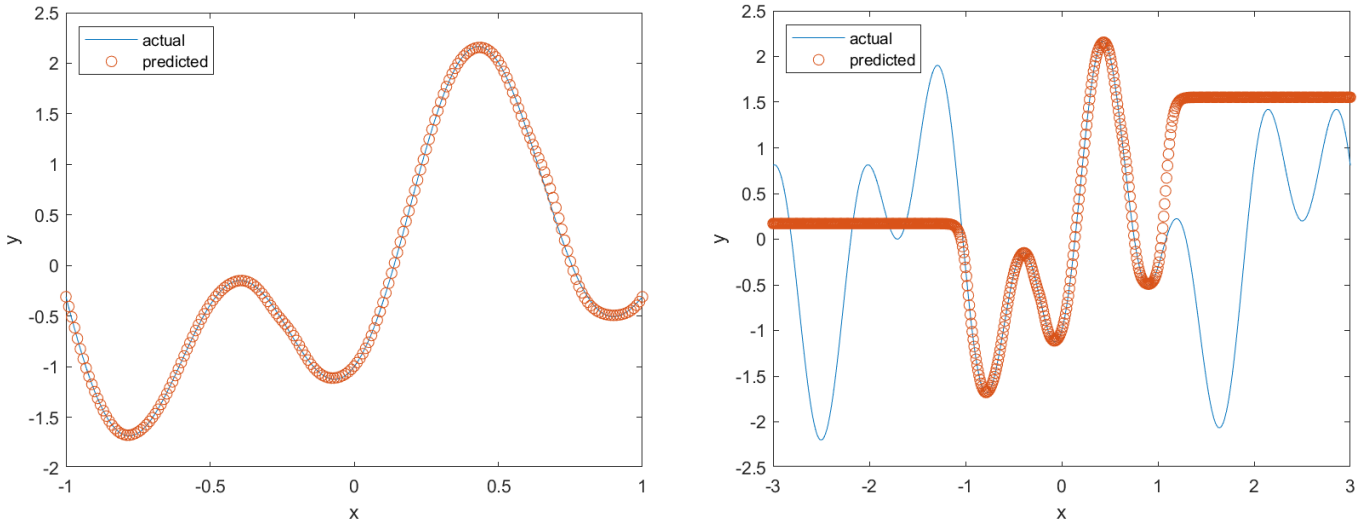

Fig. 44: Function approximation and extrapolation with 1-20-1 (trainlm)

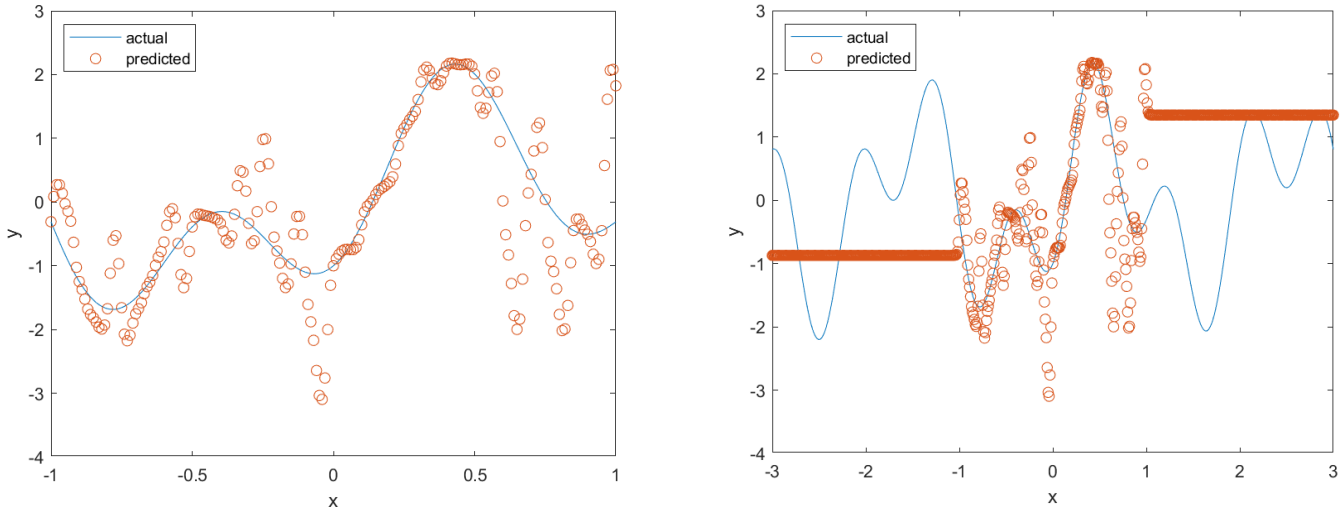

Fig. 45: Function approximation and extrapolation with 1-50-1 (trainlm)

Based on the graph plots, it is clear that when 1 to 5 hidden neurons are used, under-fitting will occur. When 6-9 neurons are used, there is proper-fitting (arguable in the case of 9 neurons as there appears to be a part where the function does not fit). However, for the case of 10, 20 and 50 neurons, over-fitting has occurred, there is tremendous error in the scatter plot for the values of y given values of x. This indicates that at least 6 neurons but not more than 10 neurons are best used to properly fit when *trainlm* is used.

Finally, it is evident that based on Fig. 34 to Fig. 45, the multi-layer perceptron is unable to make predictions outside the domain of the input limited by the training set.

Training loop code:

```
clear;

train_delta = 5e-2;
test_delta = 1e-2;
val_delta = 1e-2;
min = -1;
max = 1;
test_min = -3;
test_max = 3;
epochs = 5e2;

hidden_neurons = [1,2,3,4,5,6,7,8,9,10,20,50];
epoch_arr = [1:1:epochs];
eta = 0.005;

% Training data
train_X = [min: train_delta: max];
train_X_label = fnTrigo(train_X);

% Validation data
val_X = [min: val_delta: max];
val_X_ans = fnTrigo(val_X);

% Test data
test_X = [test_min: test_delta: test_max];
test_X_ans = fnTrigo(test_X);

% Change the number of hidden neurons and run batch training on it.
for i=1:length(hidden_neurons)

    % Train the neural network.
    [net, tr] = train_bat(hidden_neurons(i), train_X, train_X_label, epochs,
eta);

    % Validate the neural network.
    val_X_net_output = sim(net, val_X);

    % Test the neural network.
    test_X_net_output = sim(net, test_X);

    % Plotting code originally here. Refer to next page.
end
```

Batch training code:

```
function [net, tr] = train_bat(n, train_X, train_X_label, epochs, eta)

    net = fitnet(n, 'trainlm');
    net.trainParam.epochs = epochs;
    net.layers{1}.transferFcn = 'tansig';
    net.layers{2}.transferFcn = 'purelin';
    net.trainParam.lr = eta;

    [net, tr] = train(net, train_X, train_X_label);
end
```

Plotting code:

```
    filename_scatter_val = sprintf("q2b_change_hidden\\scatter_hidden_%d",
hidden_neurons(i));
    filename_plot_val = sprintf("q2b_change_hidden\\plot_hidden_%d",
hidden_neurons(i));
    filename_scatter_test = sprintf("q2b_extrapolate\\scatter_hidden_%d",
hidden_neurons(i));
    filename_plot_test = sprintf("q2b_extrapolate\\plot_hidden_%d",
hidden_neurons(i));

    % plot validation data
    plot(val_X, val_X_ans);
    hold on;
    scatter(val_X, val_X_net_output, '.');
    hold off;
    xlabel('x');
    ylabel('y');
    legend({'actual', 'predicted'}, 'Location', 'northwest');
    saveas(gcf, filename_scatter_val, 'png');

    plot(val_X, val_X_ans, val_X, val_X_net_output);
    xlabel('x');
    ylabel('y');
    legend({'actual', 'predicted'}, 'Location', 'northwest');
    saveas(gcf, filename_plot_val, 'png');

    % plot test data
    plot(test_X, test_X_ans);
    hold on;
    scatter(test_X, test_X_net_output, '.');
    hold off;
    xlabel('x');
    ylabel('y');
    legend({'actual', 'predicted'}, 'Location', 'northwest');
    saveas(gcf, filename_scatter_test, 'png');

    plot(test_X, test_X_ans, test_X, test_X_net_output);
    xlabel('x');
    ylabel('y');
    legend({'actual', 'predicted'}, 'Location', 'northwest');
    saveas(gcf, filename_plot_test, 'png');
```

2c) In the following graphs from Fig. 46 to Fig. 57, 500 epochs were taken for the multi-layer perceptron training, with $\eta = 0.005$. Also, the learning method was changed to batch learning. The 'tansig' and 'purelin' transfer functions were used for the hidden and output layers respectively. The training algorithm used was *trainbr*, with 0 regularization.

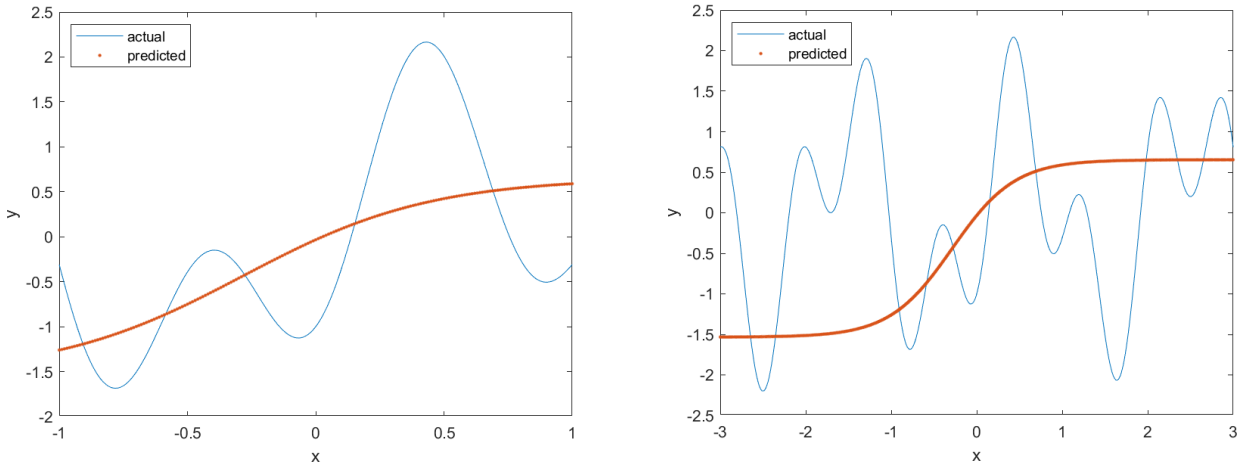

Fig. 46: Function approximation and extrapolation with 1-1-1 (trainbr)

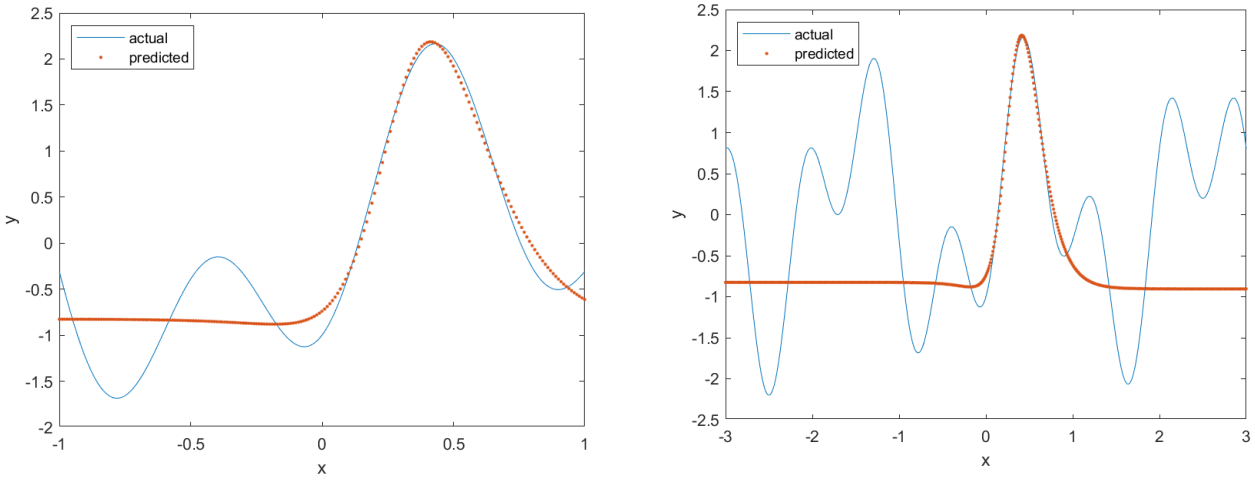

Fig. 47: Function approximation and extrapolation with 1-2-1 (trainbr)

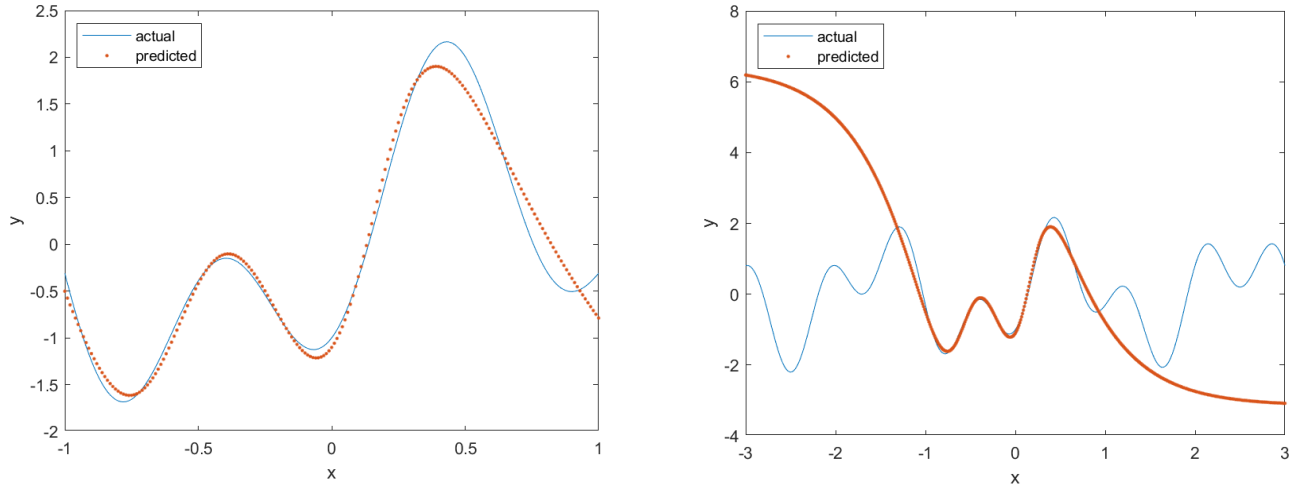

Fig. 48: Function approximation and extrapolation with 1-3-1 (trainbr)

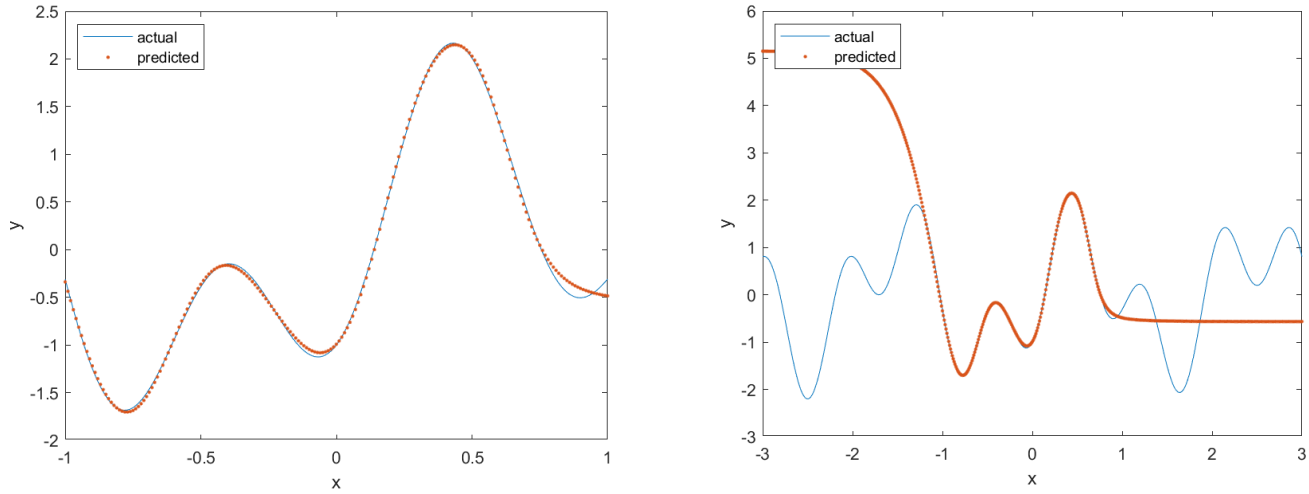

Fig. 49: Function approximation and extrapolation with 1-4-1 (trainbr)

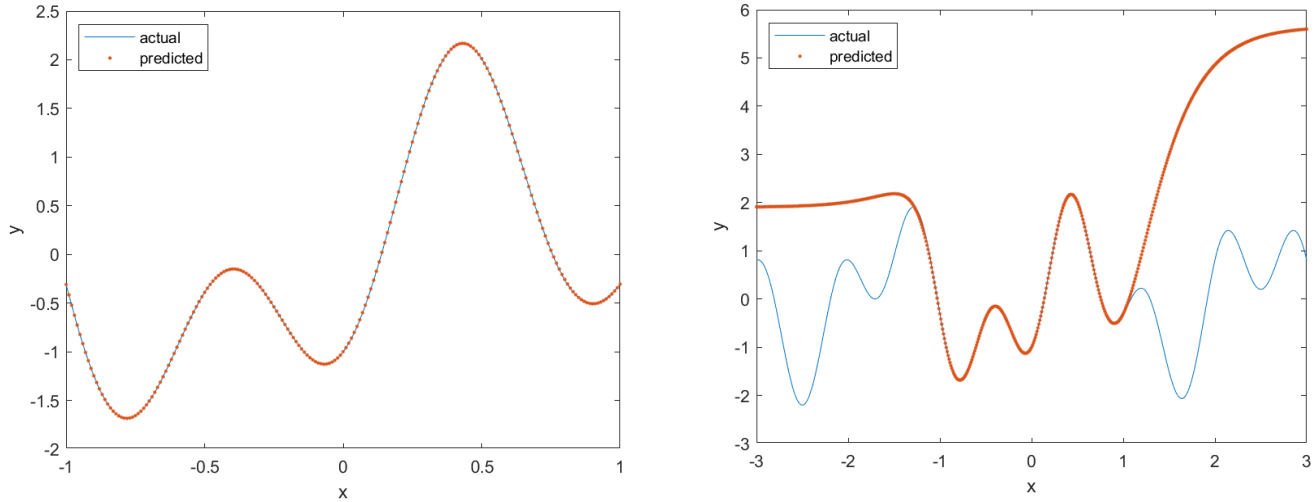

Fig. 50: Function approximation and extrapolation with 1-5-1 (trainbr)

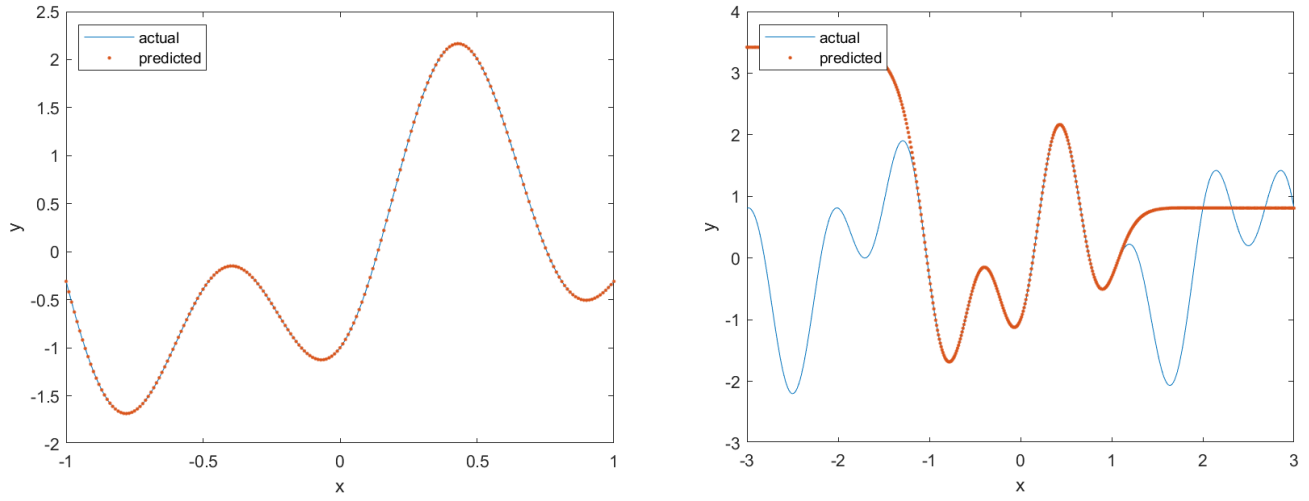

Fig. 51: Function approximation and extrapolation with 1-6-1 (trainbr)

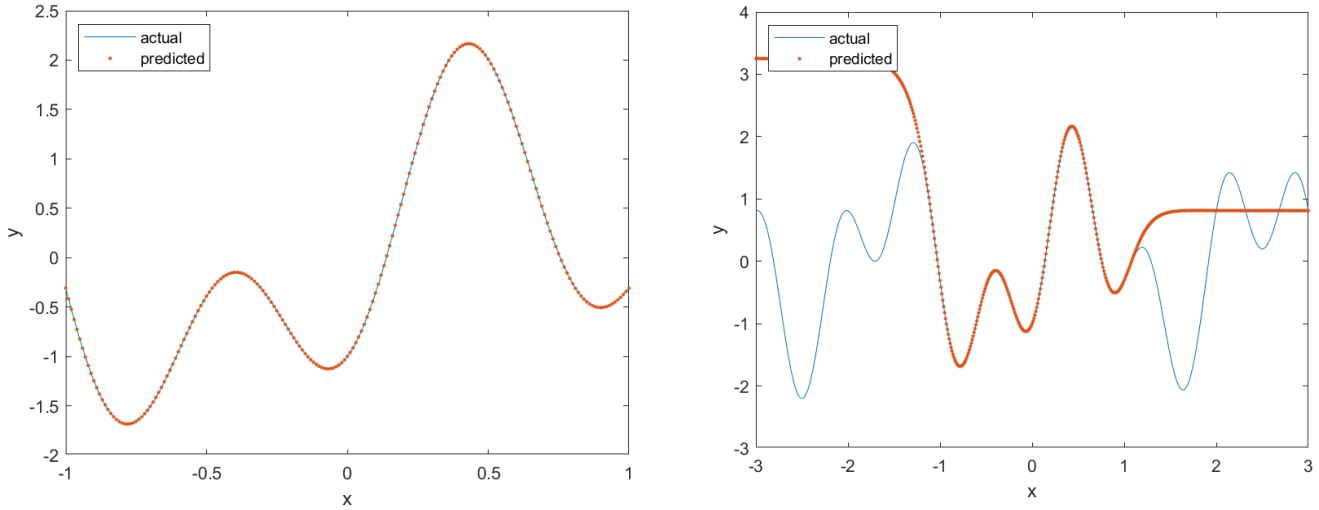

Fig. 52: Function approximation and extrapolation with 1-7-1 (trainbr)

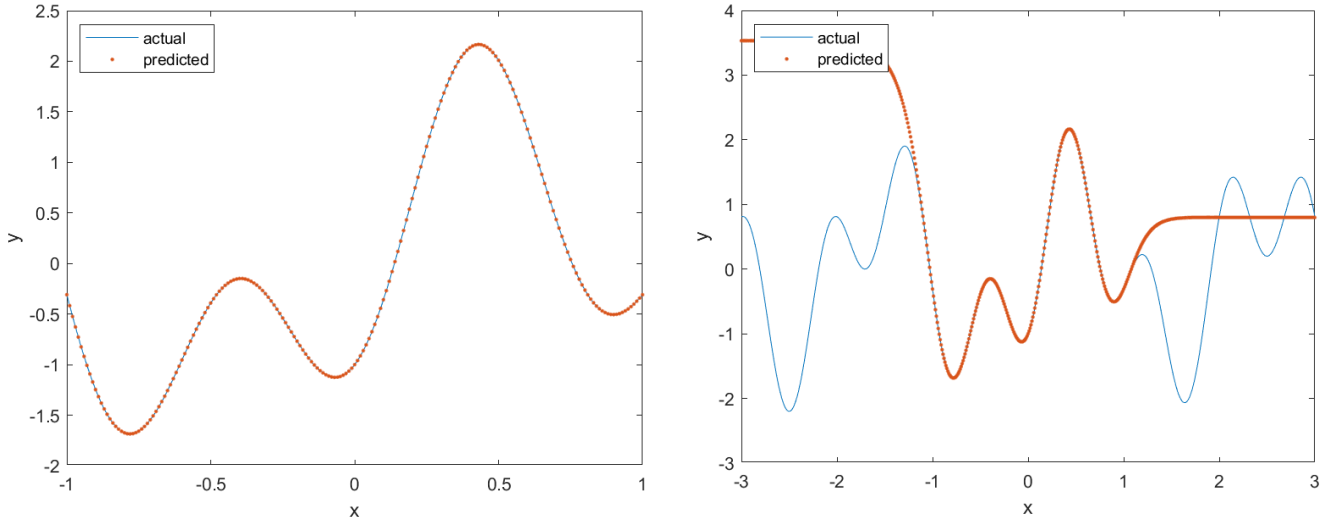

Fig. 53: Function approximation and extrapolation with 1-8-1 (trainbr)

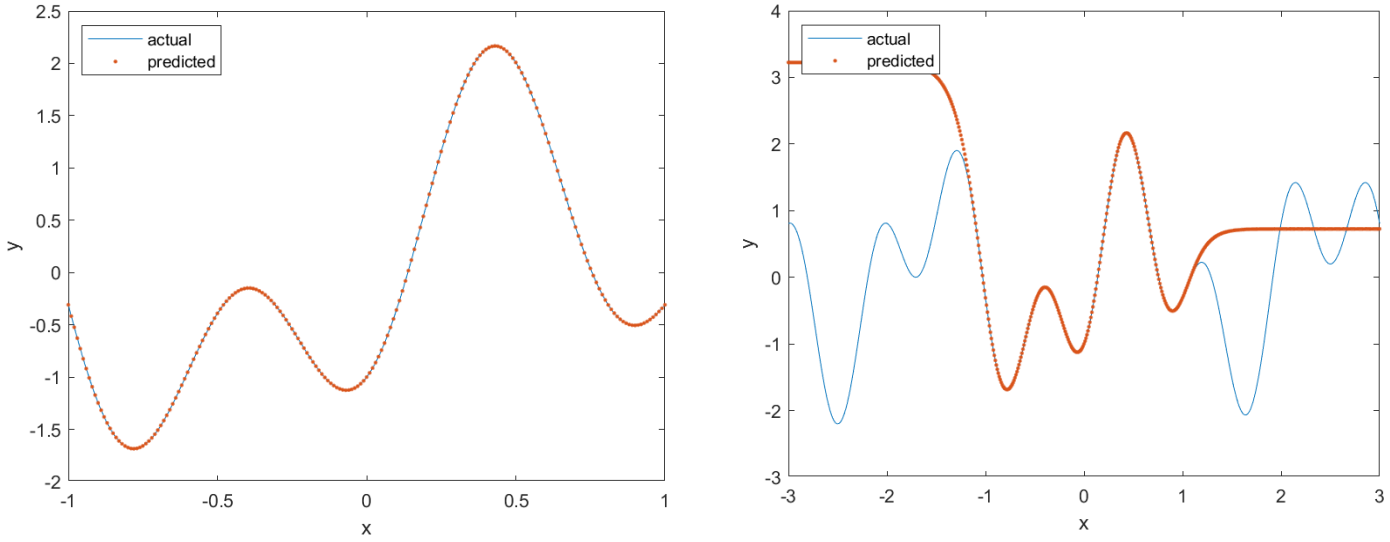

Fig. 54: Function approximation and extrapolation with 1-9-1 (trainbr)

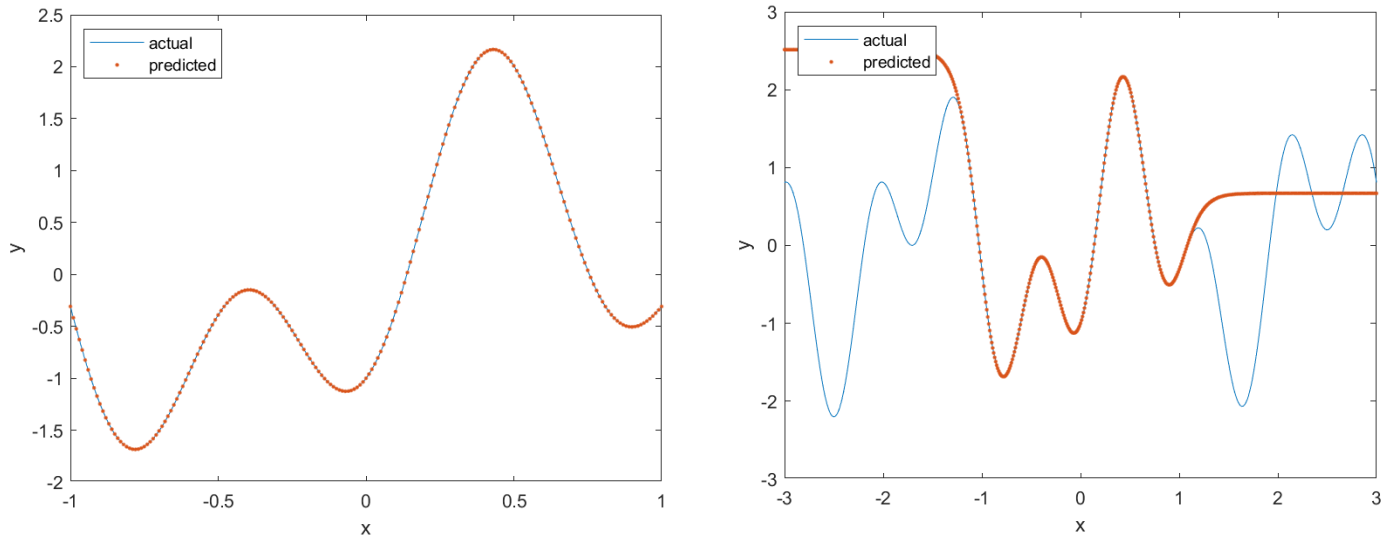

Fig. 55: Function approximation and extrapolation with 1-10-1 (trainbr)

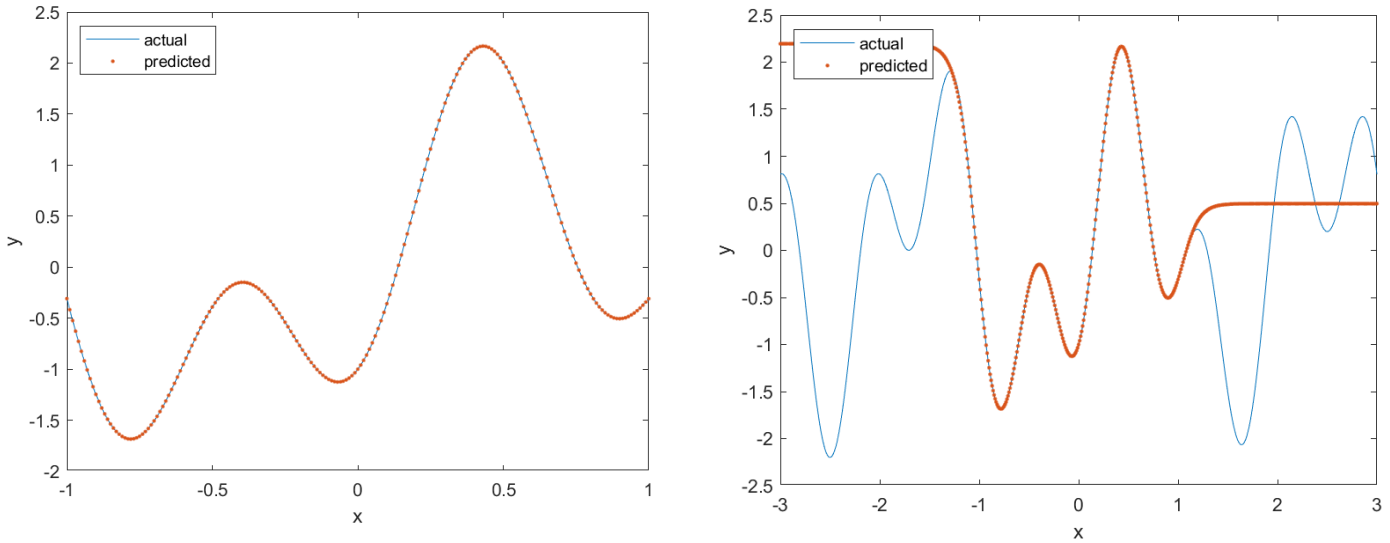

Fig. 56: Function approximation and extrapolation with 1-20-1 (trainbr)

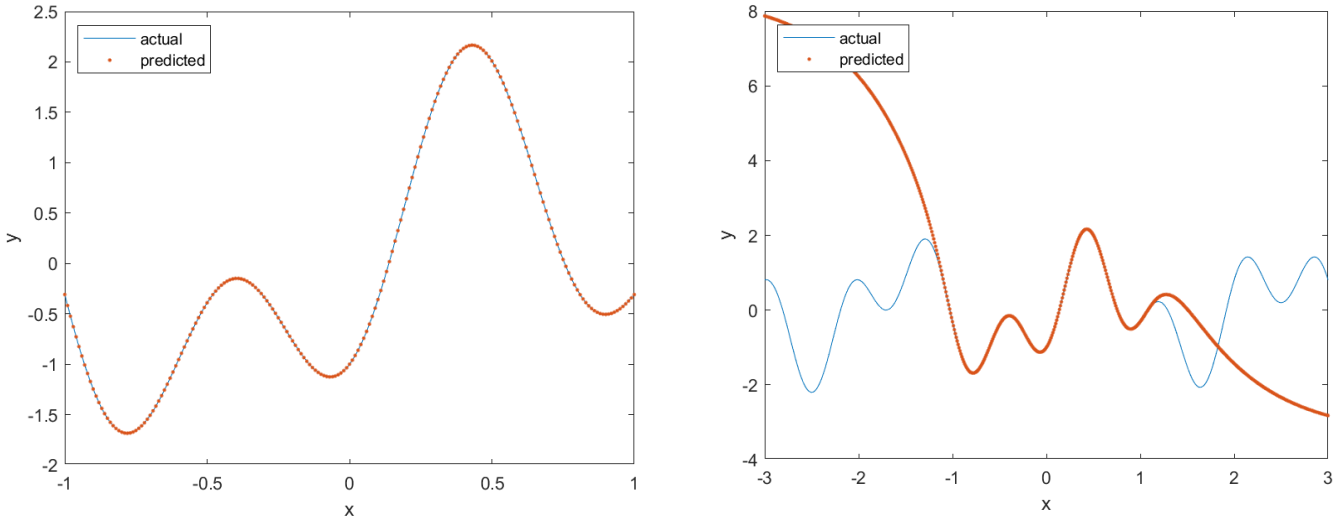

Fig. 57: Function approximation and extrapolation with 1-50-1 (trainbr)

An interesting observation when *trainbr* is used is that when 5 neurons are used, the function approximation begins to show proper-fitting, with 50 neurons even showing proper fitting when 500 epochs were run. This may be because of the number of epochs used in the learning process, as implied later when only 10 epochs are used. However, using 1 to 3 hidden neurons showed under-fitting, with 4 neurons, while being able to approximate most of the function, ultimately fell short as x tended to 1. When considering 10 epochs only, it was observed that 6 epochs showed under-fitting, with 50 epochs showing over-fitting under the same learning rate and no regularization. For the other values of hidden neurons used as prescribed, the results remain unchanged. The changes are reflected in Fig. 58 and 59.

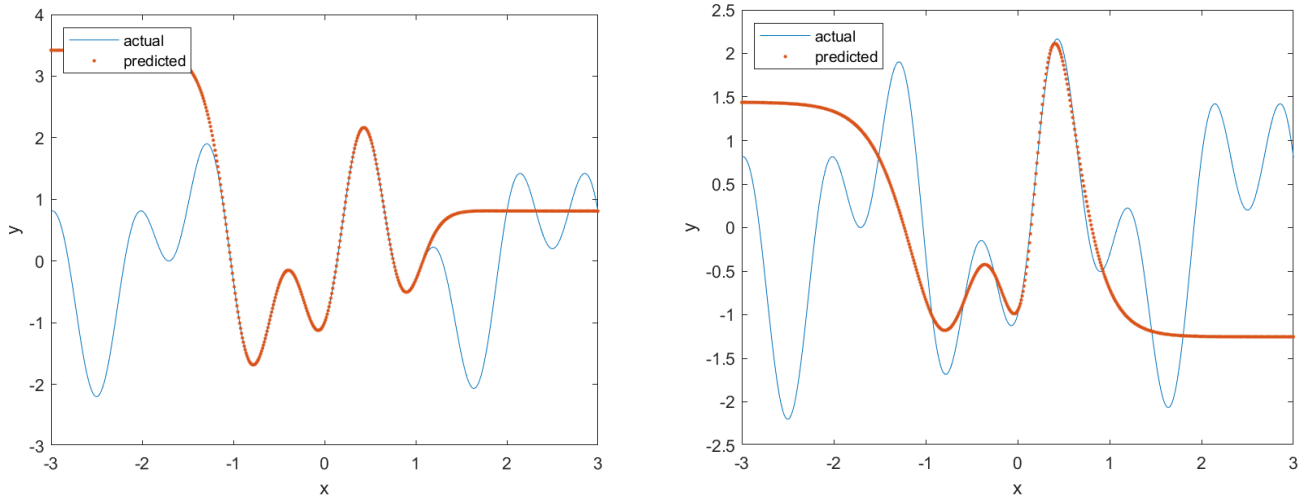

Fig. 58: 1-6-1 extrapolation with *trainbr*, 500 epochs (*left*) and 10 epochs (*right*)

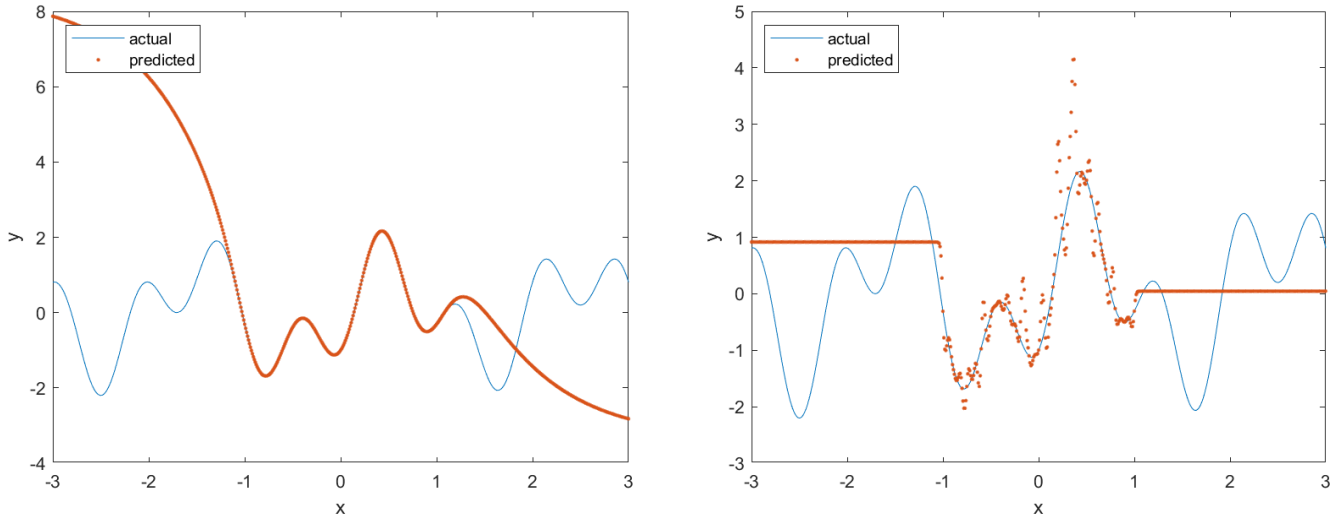

Fig. 58: 1-50-1 extrapolation with *trainbr*, 500 epochs (*left*) and 10 epochs (*right*)

To futher suggest that in this case of *trainbr*, the capability of a function to fit properly is a moderate extent, affected by the number of epochs, 20 epochs will be used for the same learning rate. In Fig. 59 and 60 below and in the next page respectively, while there is under-fitting for 5 neurons, there is proper-fitting for 6 neurons.

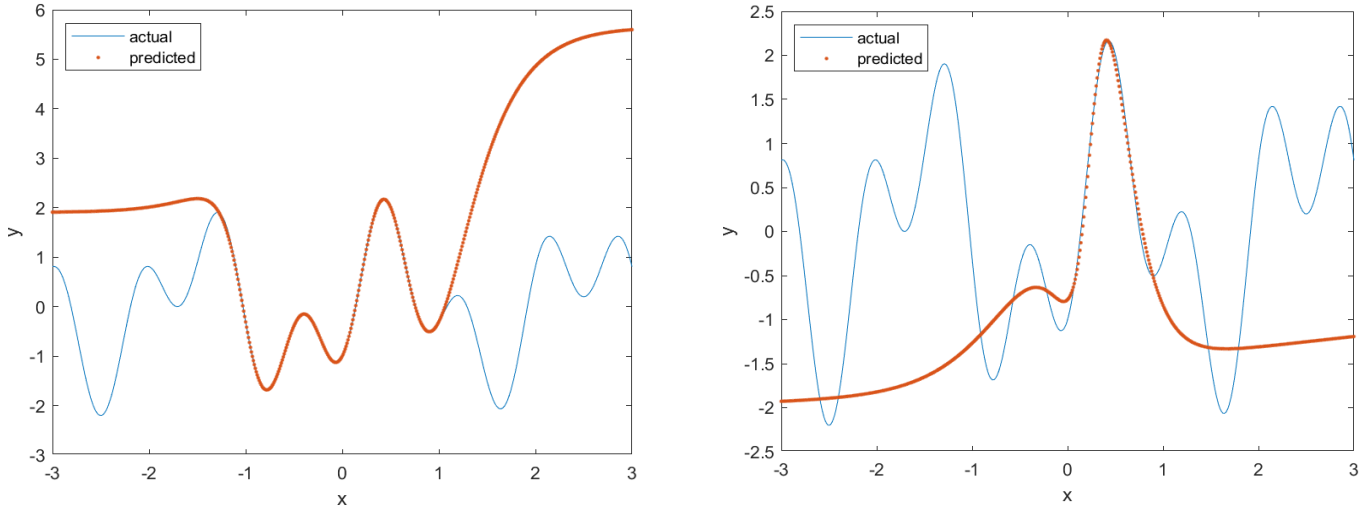

Fig. 59: 1-5-1 extrapolation with *trainbr*, 500 epochs (*left*) and 20 epochs (*right*)

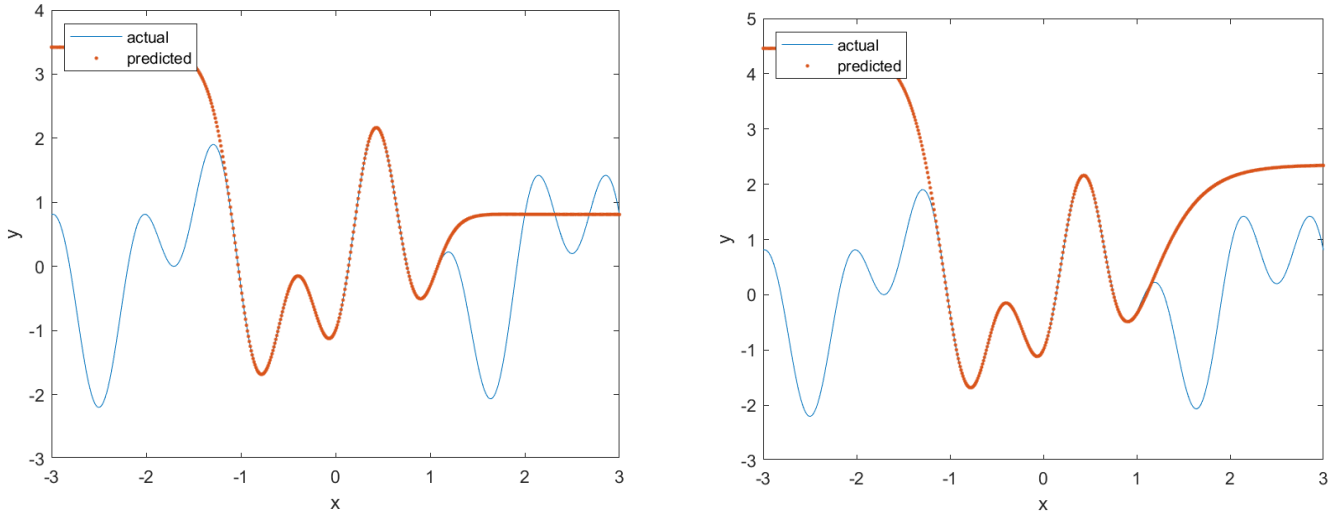

Fig. 60: 1-6-1 extrapolation with *trainbr*, 500 epochs (*left*) and 20 epochs (*right*)

It thus implies that when *trainbr* is used, an increase in the number of epochs can at large be beneficial to the function approximation within the domain of the input range provided by the training set. However, regardless of the learning rate or number of epochs used for the training, it is shown that while *trainbr* can fit functions better than *trainlm*, *traingda* or *traingdx*, it is also unable to approximate the values outside the domain of the input limited by the training set.

Training loop code:

```
clear;

train_delta = 5e-2;
test_delta = 1e-2;
val_delta = 1e-2;
min = -1;
max = 1;
test_min = -3;
test_max = 3;
epochs = 500; % To change to 20 and 10

hidden_neurons = [1,2,3,4,5,6,7,8,9,10,20,50];
epoch_arr = [1:1:epochs];
eta = 0.005;

% Training data
train_X = [min: train_delta: max];
train_X_label = fnTrigo(train_X);

% Validation data
val_X = [min: val_delta: max];
val_X_ans = fnTrigo(val_X);

% Test data
test_X = [test_min: test_delta: test_max];
test_X_ans = fnTrigo(test_X);

% Change the number of hidden neurons and run batch training on it.
for i=1:length(hidden_neurons)

    % Train the neural network.
    [net, tr] = train_bat(hidden_neurons(i), train_X, train_X_label, epochs,
eta);

    % Validate the neural network.
    val_X_net_output = sim(net, val_X);

    % Test the neural network.
    test_X_net_output = sim(net, test_X);

    % Plotting code originally here. Refer to next page.

end
```

Batch training code:

```
function [net, tr] = train_bat(n, train_X, train_X_label, epochs, eta)

    net = fitnet(n, 'trainbr');
    net.trainParam.epochs = epochs;
    net.layers{1}.transferFcn = 'tansig';
    net.layers{2}.transferFcn = 'purelin';
    net.trainParam.lr = eta;

    [net, tr] = train(net, train_X, train_X_label);
end
```

Plotting code:

```
filename_scatter_val = sprintf("q2c_change_hidden\\scatter_hidden_%d",
hidden_neurons(i));
    filename_plot_val = sprintf("q2c_change_hidden\\plot_hidden_%d",
hidden_neurons(i));
    filename_scatter_test = sprintf("q2c_extrapolate\\scatter_hidden_%d",
hidden_neurons(i));
    filename_plot_test = sprintf("q2c_extrapolate\\plot_hidden_%d",
hidden_neurons(i));

    % plot validation data
    plot(val_X, val_X_ans);
    hold on;
    scatter(val_X, val_X_net_output, '.');
    hold off;
    xlabel('x');
    ylabel('y');
    legend({'actual', 'predicted'}, 'Location', 'northwest');
    saveas(gcf, filename_scatter_val, 'png');

    plot(val_X, val_X_ans, val_X, val_X_net_output);
    xlabel('x');
    ylabel('y');
    legend({'actual', 'predicted'}, 'Location', 'northwest');
    saveas(gcf, filename_plot_val, 'png');

    % plot test data
    plot(test_X, test_X_ans);
    hold on;
    scatter(test_X, test_X_net_output, '.');
    hold off;
    xlabel('x');
    ylabel('y');
    legend({'actual', 'predicted'}, 'Location', 'northwest');
    saveas(gcf, filename_scatter_test, 'png');

    plot(test_X, test_X_ans, test_X, test_X_net_output);
    xlabel('x');
    ylabel('y');
    legend({'actual', 'predicted'}, 'Location', 'northwest');
    saveas(gcf, filename_plot_test, 'png');
```

Question 3:

Based on the question requirement, as my matriculation number is A0164914B, the last 2 digits of my matriculation number is 14. Thus, $mod(14,4) + 1 = 3$. With this, I will be classifiying between Coast and Inside City.

For 3a), images were extracted and then processed using the following code:

```
function [img, label] = extract_img(filepath, folder, i)
% Extracts the i-th image and its corresponding label as denoted in the given
filepath. Only one
% image is extracted at a time.
    filename = filepath + '\\' + folder(i).name;
    img = imread(filename);
    img = img(:);
    tmp = strsplit(filename, {'_', '.'});
    label = str2num(tmp{3});
end
```

For 3b) onwards, images were extracted and then processed using the following code; given the necessity to account for multiple components and size, I felt it was justified to automate the processes where image extraction, component finding and plot saving were concerned:

```
function [img, label] = extract_img(filepath, folder, i, scale)
% Extracts the i-th image and its corresponding label as denoted in the given
filepath. Only one
% image is extracted at a time.
    filename = filepath + '\\' + folder(i).name;
    img = imread(filename);
    img = imresize(img, scale);
    img = img(:);
    tmp = strsplit(filename, {'_', '.'});
    label = str2num(tmp{3});
end

function [img_set, label_set] = extract_img_set(filepath, folder, count,
scale)
    sz = (256*scale)^2;
    img_set = zeros([sz count]);
    label_set = zeros([1 count]);
    for i=3:count+2
        [img, label] = extract_img(filepath, folder, i, scale);
        img_set(:,i-2) = img;
        label_set(:,i-2) = label;
    end
end
```

3a) In approaching this problem, I decided to run Rosenblatt's perceptron using the built-in perceptron in MATLAB's Neural Network Toolkit. Moreover, I decided to approach the problem accounting for both batch and sequential training methods, inspecting the implications of an increased number of epochs on the classification accuracy. For the sequential case, $\eta = 0.01$, where the *hardlim* transfer function was used. Fig. 61 shows the performance of the perceptron in when batch training was used. Fig. 62 to 66 shows the performance of the perceptron when sequential learning was used instead. Table 1 below is a summary of the findings obtained in Fig. 61 to 66, where it is evident that given enough epochs to train the perceptron sequentially, the accuracy of image classification will eventually surpass that of a perceptron that learned using batch training. Despite this, with a poor validation accuracy, a multi-layer perceptron might be more suited for this task instead.

Learning Method	Epochs	Training Accuracy (%)	Validation Accuracy (%)
Batch	0	53.89	53.89
	66	100	68.86
	80	100	68.86
Sequential	100	59.88	59.88
	200	62.48	61.48
	500	73.45	64.67
	1000	93.61	70.66
	1250	96.61	71.86

Table 1. Summarised Findings for Rosenblatt's Perceptron for Image Classification for Q3a)

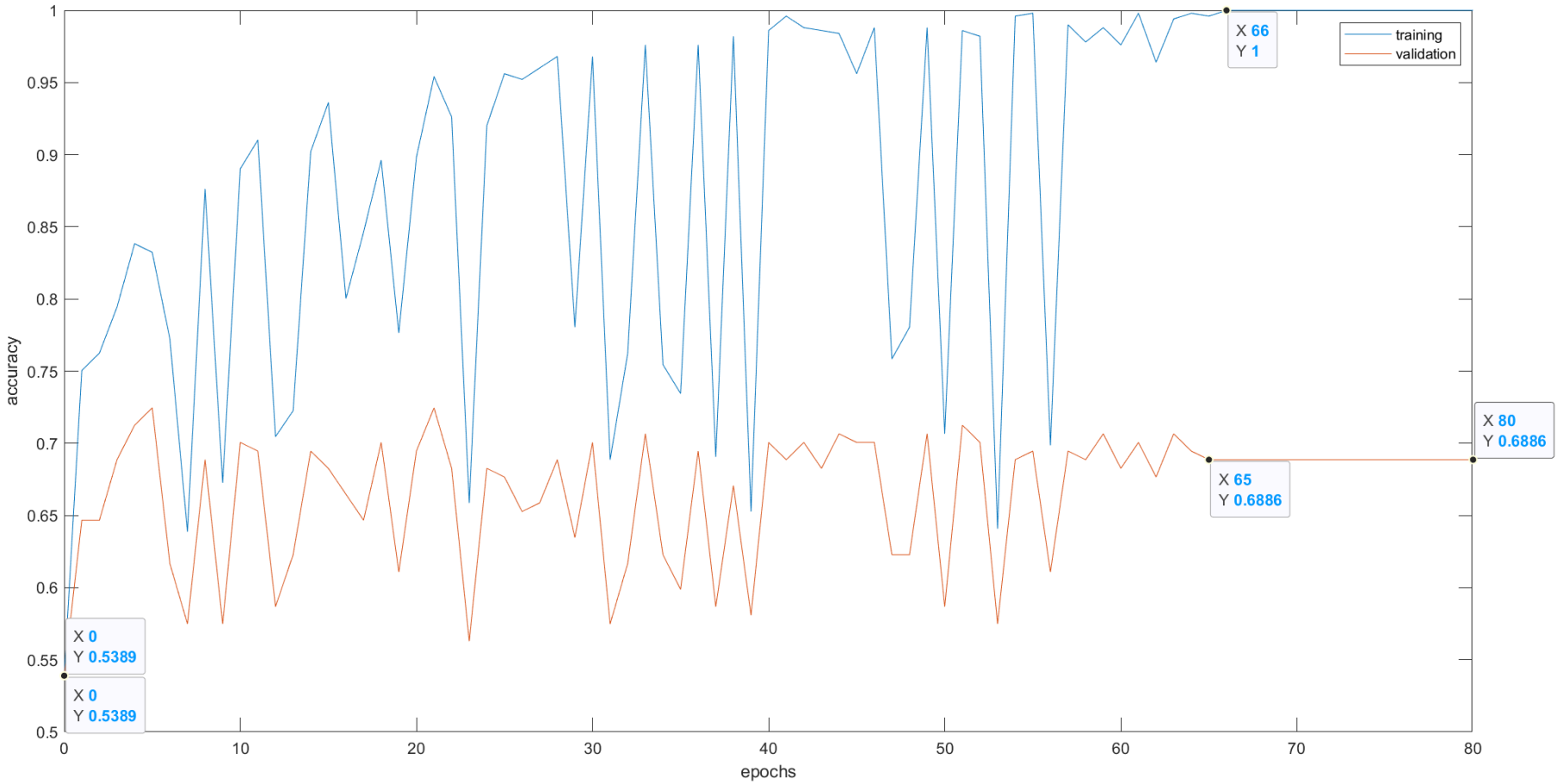

Fig. 61: Image Classification with Rosenblatt's Perceptron (batch training)

In Fig. 61, for an untrained perceptron, both the training and validation set yielded a 53.89% accuracy, with the perceptron obtaining perfect accuracy for the training data after the 66th epoch and the validation data obtaining a constant accuracy of 68.86% accuracy after the 65th epoch. It should be worth noting that this data was obtained using batch learning.

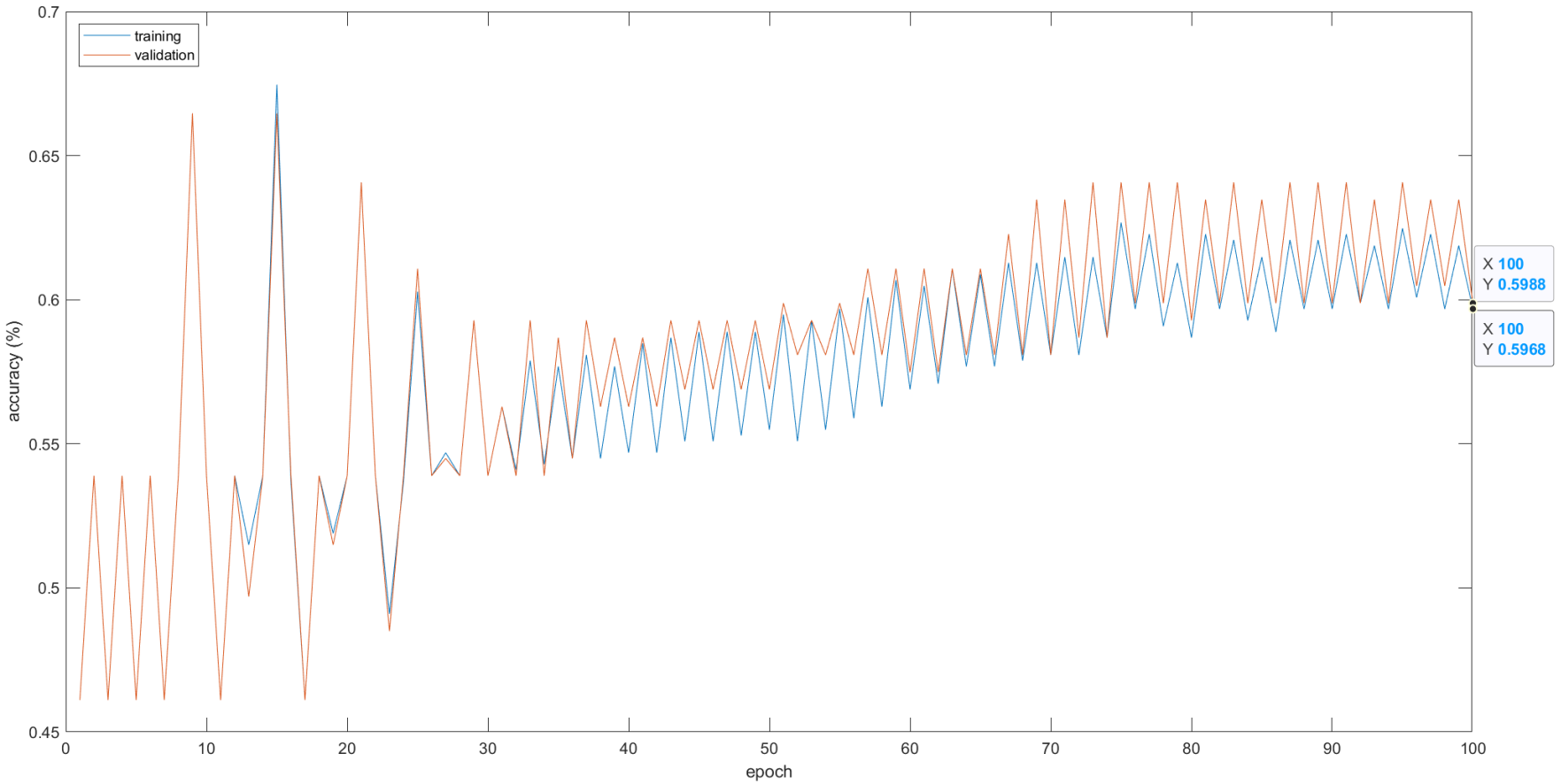

Fig. 62: Image Classification with Rosenblatt's Perceptron (sequential training with 100 epochs)

When 100 epochs were considered, very immediately, the accuracy of the perceptron experienced tremendous fluctuations that will be further enhanced in the subsequent figures, when the number of epochs increase significantly. However, based on this diagram, after 100 epochs, an accuracy of 59.88% was obtained for the training data and 59.68% was obtained for the validation data.

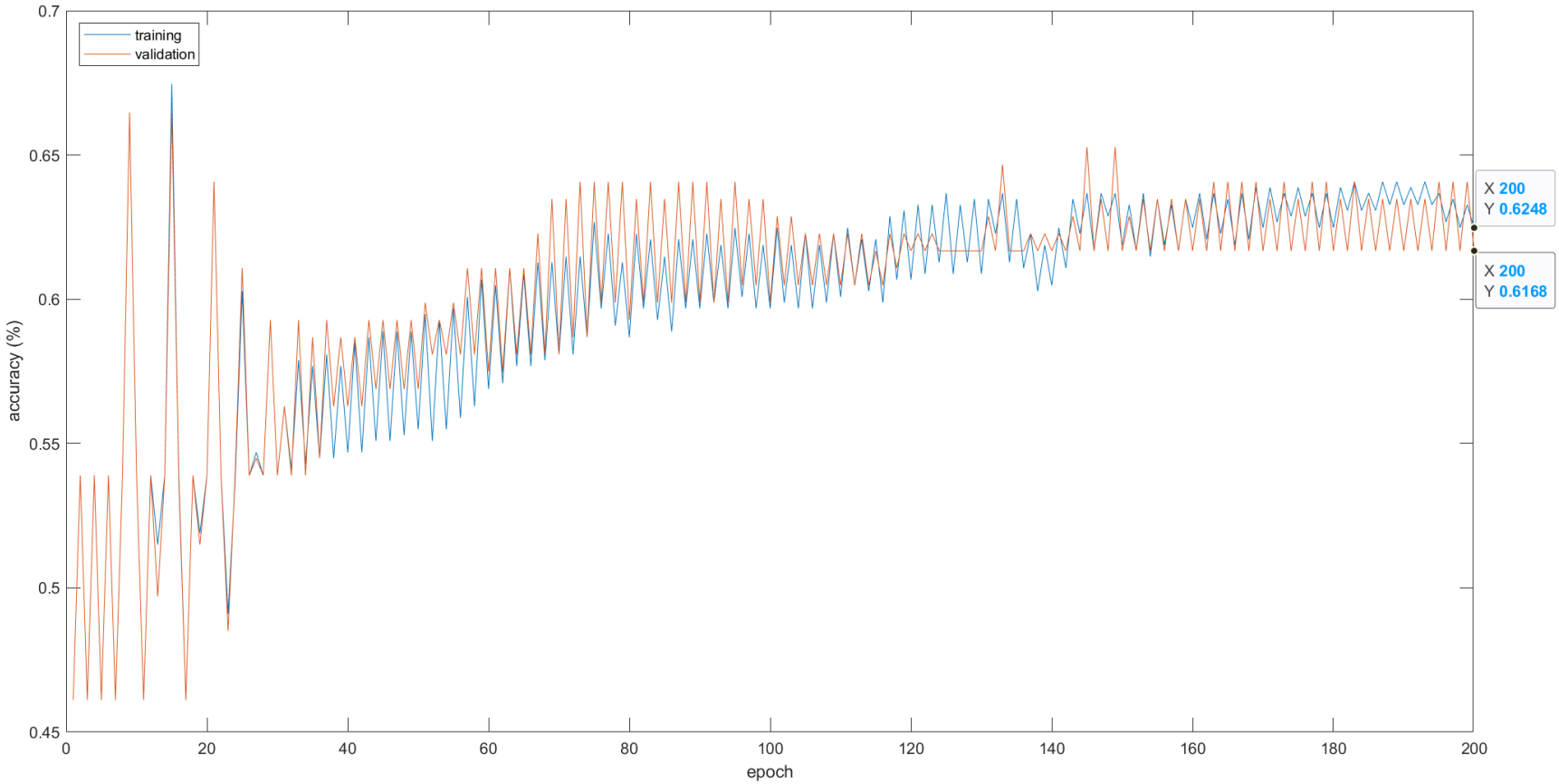

Fig. 63: Image Classification with Rosenblatt's Perceptron (sequential training with 200 epochs)

When 200 epochs were considered, the accuracy of the perceptron on the training set and the validation set still has not begin to vary significantly, implying that the perceptron could still be capable of learning and achieving better accuracies. In the case of 200 epochs, the training data yielded an accuracy of 62.48% and the validation data yielded an accuracy of 61.68%.

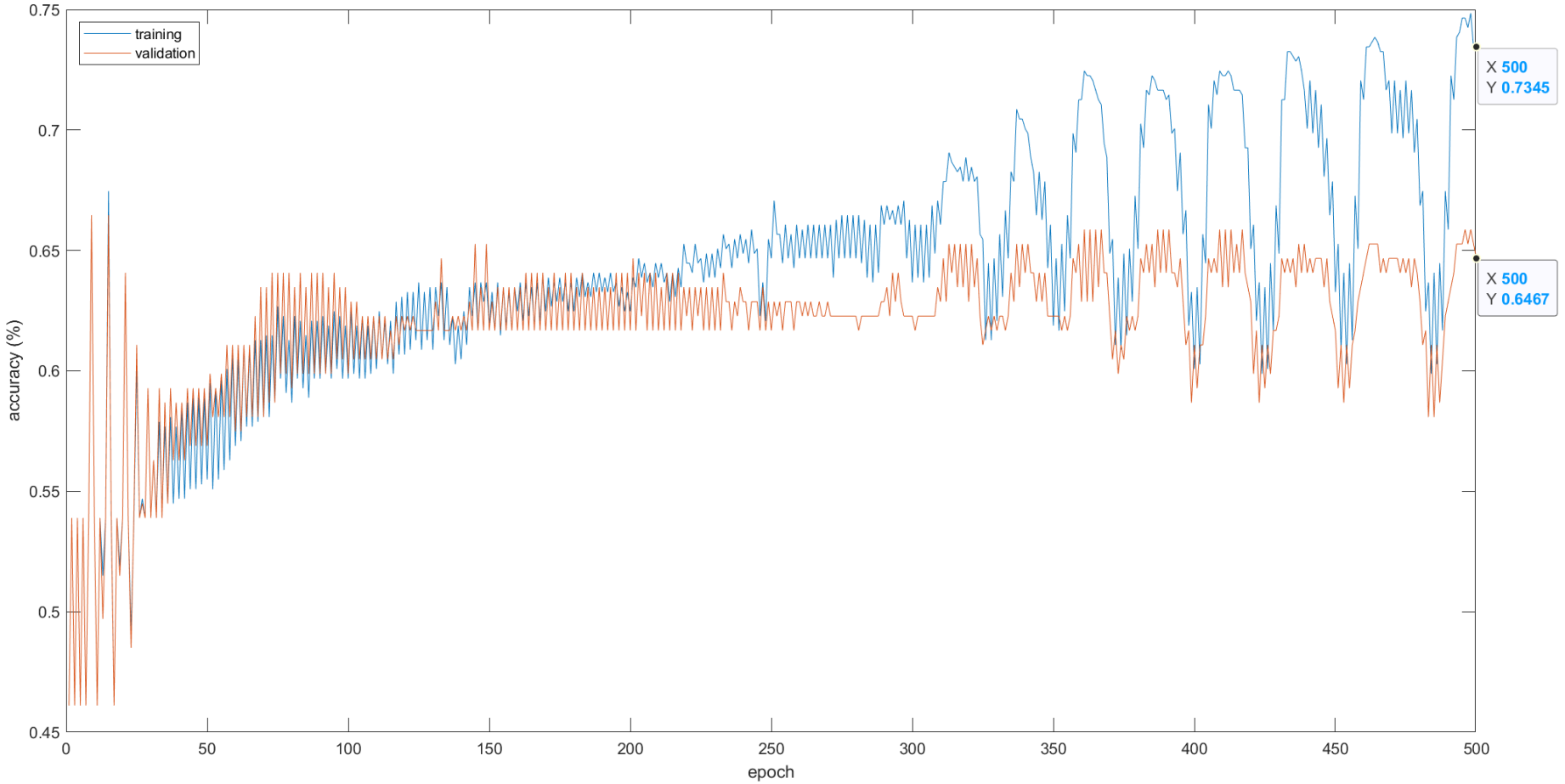

Fig. 64: Image Classification with Rosenblatt's Perceptron (sequential training with 500 epochs)

As shown above, upon reaching some point after the 200th epoch, the accuracy of the training data and validation data begun deviating significantly. When 500 epochs are run, the training accuracy is 73.45% and the validation accuracy is 64.67%.

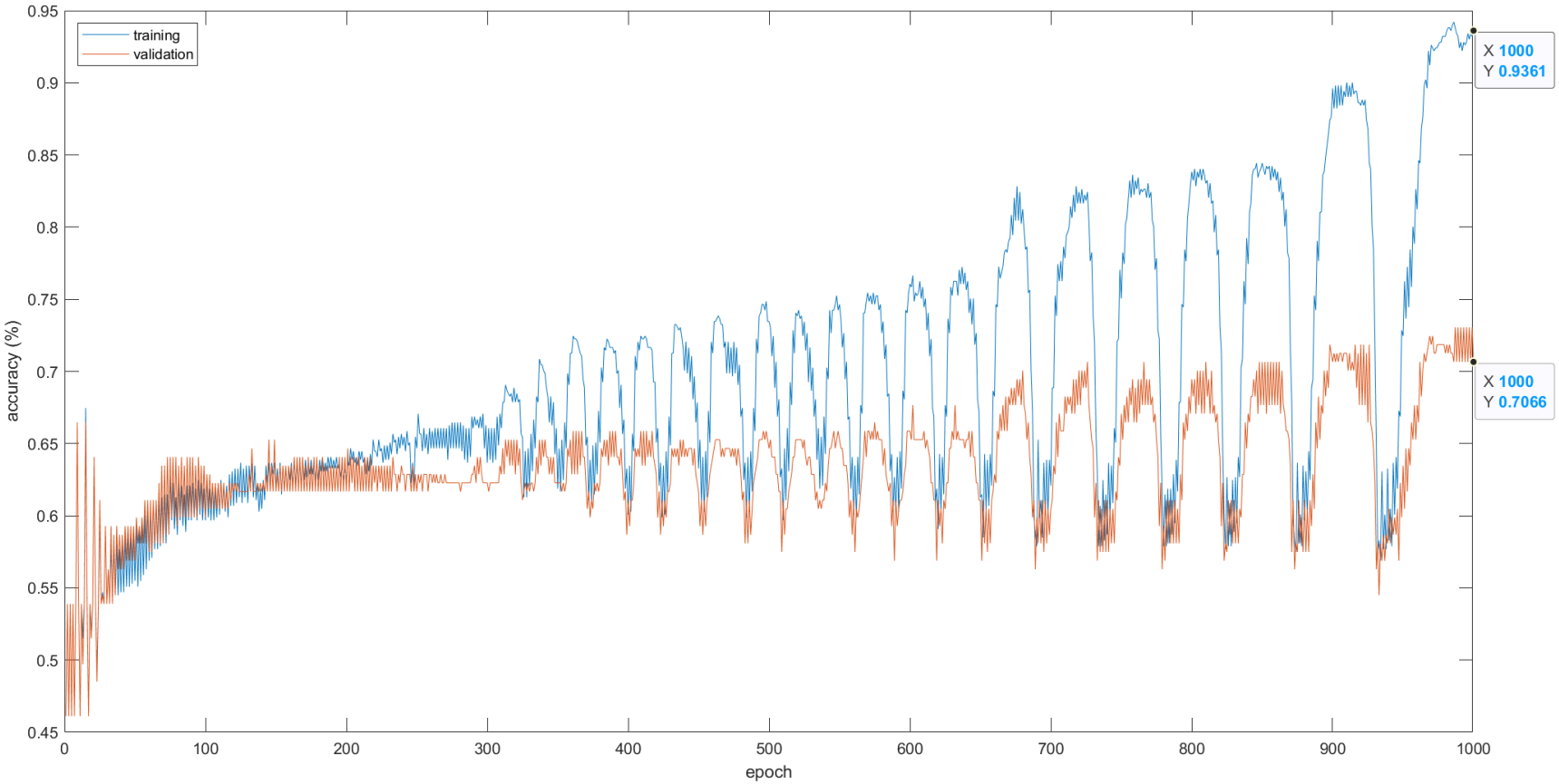

Fig. 65: Image Classification with Rosenblatt's Perceptron (sequential training with 1000 epochs)

As shown above, upon reaching some point after the 200th epoch, the accuracy of the training data and validation data begun deviating significantly. When 1000 epochs are run, the training accuracy is 93.61% and the validation accuracy is 70.66%. This suggests that while sequential learning can help the perceptron yield better accuracy, it would require significantly greater number of epochs, thereby incurring significant trade-offs in time.

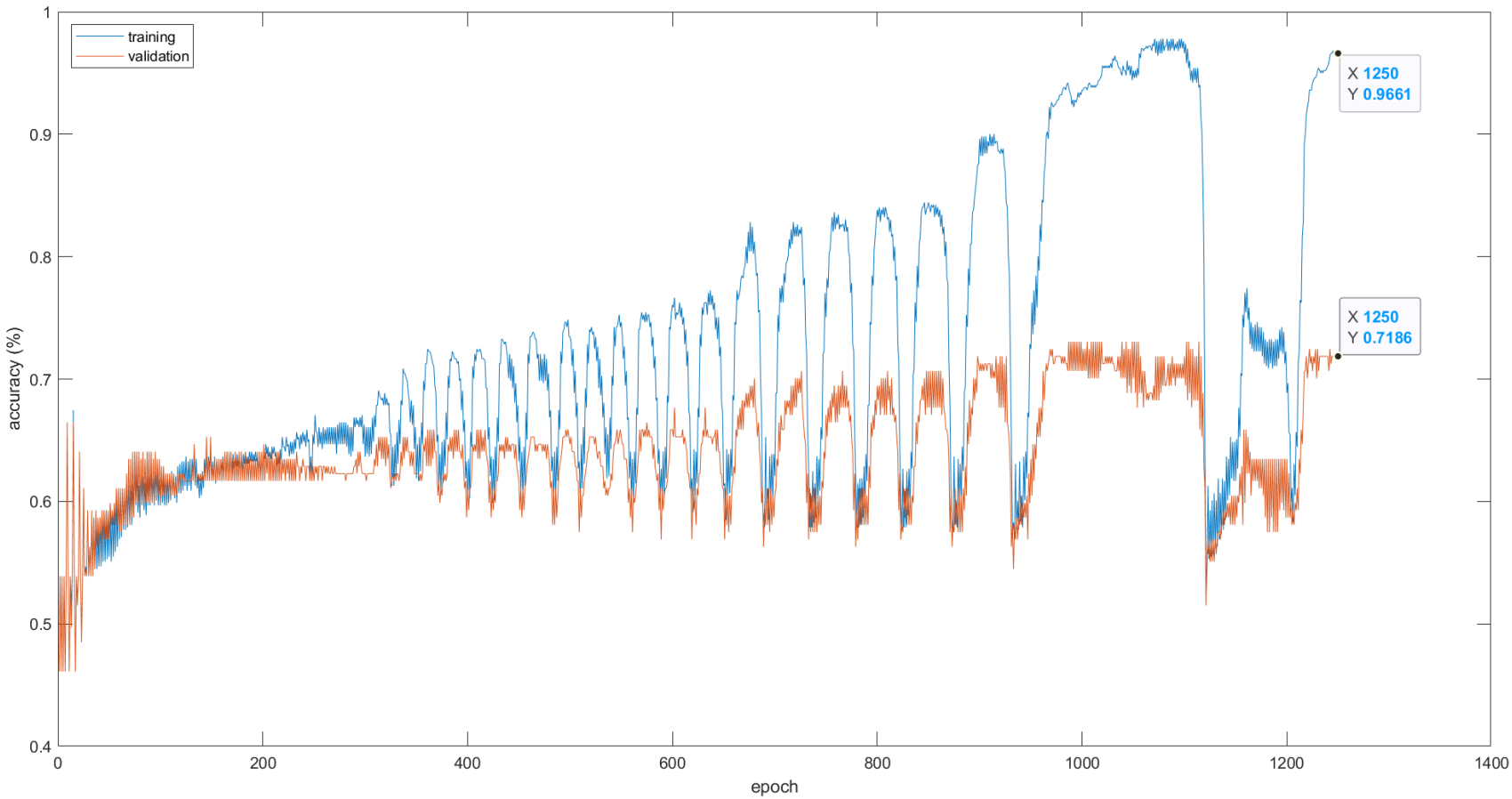

Fig. 66: Image Classification with Rosenblatt's Perceptron (sequential training with 1250 epochs)

As shown above, when 1250 epochs are run, the training accuracy is 96.61% and the validation accuracy is 71.86%. Based on the overall plots, it appears that sequential learning faces tremendous fluctuations, possibly a consequence of the noise introduced during the learning procedure.

Sequential training loop:

```
clear;

% train has 501 items. val has 167 items.
n_train = 501;
n_val = 167;
eta = 0.01;
epoch_test = [1e2, 2e2, 5e2, 1e3, 1.25e3];
training_sets = [1:1:501];

% Defining training data
filepath_train = "group_3\\train";
train_folder = dir(filepath_train);
training_data = zeros([65536, n_train]);
training_label = zeros([1, n_train]);

% Extract training data
for i=3:n_train+2
    [img, label] = extract_img(filepath_train, train_folder, i);
    training_data(:,i-2) = img;
    training_label(:,i-2) = label;
end

% Retrieve validation data
filepath_val = "group_3\\val";
val_folder = dir(filepath_val);
validation_data = zeros([65536, n_val]);
validation_label = zeros([1, n_val]);

% Extract validation data
for i=3:n_val+2
    [img, label] = extract_img(filepath_val, val_folder, i);
    validation_data(:,i-2) = img;
    validation_label(:,i-2) = label;
end

% Training loop
for i=1:length(epoch_test)

   epochs = epoch_test(i);
   epoch = [1:1:epochs];
   [net, accu_train, accu_val] = train_seq(training_data, training_label,
validation_data, validation_label, epochs);

end
```

Sequential training code:

```
function [net, accu_train, accu_val] = train_seq(training_data,
training_label, validation_data, validation_label, epochs)

    net = perceptron;
    net.trainParam.epochs = epochs;

    accu_train = zeros(1, epochs);
    accu_val = zeros(1, epochs);

% Training loop
    for i=1:epochs
        idx = randperm(size(training_data, 2));
        [net,a,e] = adapt(net, training_data(:,idx), training_label(:,idx));

        pred_train = net(training_data(:,idx));
        accu_train(i) = 1 - mean(abs(pred_train - training_label(:,idx)));

        val_train = net(validation_data);
        accu_val(i) = 1 - mean(abs(val_train - validation_label));
    end
end
```

Sequential training plotting code:

```
filename = sprintf("q3a_sequential\\sequential_epoch_%d", epochs);
plot(epoch, accu_train, epoch, accu_val);
xlabel("epoch");
ylabel("accuracy (%)");
legend({'training', 'validation'}, 'Location', 'northwest');
saveas(gcf, filename, 'png');
```

Batch training loop:

```
clear;

n_train = 501;
n_val = 167;
eta = 0.005;
epoch_test = [0:1:80];
train_acc = zeros([1 length(epoch_test)]);
val_acc = zeros([1 length(epoch_test)]);
training_sets = [1:1:501];

% Retrieve training data
filepath_train = "group_3\\train";
train_folder = dir(filepath_train);
training_data = zeros([65536, n_train]);
training_label = zeros([1, n_train]);

for i=3:n_train+2
    [img, label] = extract_img(filepath_train, train_folder, i);
    training_data(:,i-2) = img;
    training_label(:,i-2) = label;
end

% Retrieve validation data
filepath_val = "group_3\\val";
val_folder = dir(filepath_val);
validation_data = zeros([65536, n_val]);
validation_label = zeros([1, n_val]);

for i=3:n_val+2
    [img, label] = extract_img(filepath_val, val_folder, i);
    validation_data(:,i-2) = img;
    validation_label(:,i-2) = label;
end

for i=1:length(epoch_test)

    net = perceptron('hardlim', 'learnp');
    epochs = epoch_test(i);
    net.trainParam.epochs=epochs;
    accu_train = zeros(1, epochs);
    accu_val = zeros(1, epochs);
    epoch = [1:1:epochs];

    net = train(net, training_data, training_label); % Training line here.

    train_count = [1:1:n_train];
    train_out = net(training_data);
    val_count = [1:1:n_val];
    val_out = net(validation_data);

    train_acc(i) = 1 - mean(abs(train_out - training_label));
    val_acc(i) = 1 - mean(abs(val_out - validation_label));

   % Batch training plotting code here. Refer to subsequent pages.
end
```

Batch training plotting code:

```
display("training accuracies: ");
display(train_acc);

display("validation accuracies: ");
display(val_acc);

filename = sprintf("q3a_batch\\batch_accuracy");
plot(epoch_test, train_acc, epoch_test, val_acc);
xlabel("epochs");
ylabel("accuracy");
legend({"training", "validation"}, 'Location', 'northeast');
saveas(gcf, filename, 'png');
```

3b) Through experimentation, it was discovered that rather than using PCA to re-size the image, PCA should be used to improve the clarity of the image, whereas imreisze() was used to resize the image. The effective rank of the image was calculated using SVD and a performance of 99% was taken. After which, the performance methodology once again compared across batch and sequential training, with batch training taking 200 epochs and sequential, 800 epochs. The code below shows the PCA code. **extract_component_numbers will be used for parts c) and d) as well!!!!**

```
function [coeff, img, x_form] = pca_img(filepath, folder, i, nComp, scale)
    filename = filepath + '\\' + folder(i).name;
    old_img = imread(filename);
    img = double(old_img);
    img = imresize(img, scale);
    img_mean = mean(img);
    img_adjusted = img-img_mean;

    [coeff, score] = pca(img_adjusted);
    x_form = score(:,1:nComp)*coeff(:,1:nComp)';
    x_form = x_form + img_mean;
    x_form = uint8(x_form);
end

function nComps = extract_component_numbers(filepath, folder, count, dim)

    eff_ranks = zeros([1 count]);
    filename = filepath + '\\' + folder(i).name;
    for i=3:count+2
        I = imread(filename);
        sing_val = svd(double(I));
        sv_sum = 0;
        k_sv_sum = 0;
        for j = 1:dim
            sv_sum = sv_sum + sing_val(j);
        end

        for k = 1:dim
            k_sv_sum = k_sv_sum + sing_val(k);
            if k_sv_sum/sv_sum >= 0.99
                eff_ranks(i) = k;
                break
            end
        end
    end

    nComps = ceil(mean(eff_ranks));
end
```

It was discovered that once again, sequential training data tended to have significantly more noise which affected the readings taken when 800 epochs were accounted for. For 800 epochs, as the image size decreased, the peak accuracy also tended to decrease in sequential learning. A similar pattern is observed in batch learning as well; it appears that as the size of the image decreases, the peak accuracy for the training set decreased. However, the peak accuracy for the validation set had not decreased as the peak value was attained at the 20th epoch for all sizes except for the image size of 128 pixel, which stabilized at the peak accuracy in the 80th epoch. This is reflected in Table 2 below.

Size	Learning Method	Peak training accuracy (%)	Peak validation accuracy (%)
256 * 256	Batch	100	69.46 (20th epoch)
	Sequential	83.83	69.46
128 * 128	Batch	100	69.46 (80th epoch)
	Sequential	81.04	70.06
64 * 64	Batch	100	69.46 (20th epoch)
	Sequential	74.05	67.66
32 * 32	Batch	95.61	69.46 (20th epoch)
	Sequential	72.06	66.47

Table 2. Peak training and validation accuracies as size decreases

It is also indicated that as the size of the image decreased, it generally took longer for a perceptron to attain a stable accuracy, with the perceptron not attaining a stable accuracy for a 32 * 32 pixels image at all when batch training was used. Table 3 shows the data obtained for batch learning to obtain a stable accuracy.

Size	Stable training accuracy (%)	Epoch	Stable validation accuracy (%)	Epoch
256 * 256	100	80	68.86	80
128 * 128	100	80	69.46	80
64 * 64	100	140	67.07	140

Table 3. Stable training and validation accuracies as size decreases

Fig. 67 to 69 indicate the increase in accuracy and fluctuations thereof as the size of the image increases.

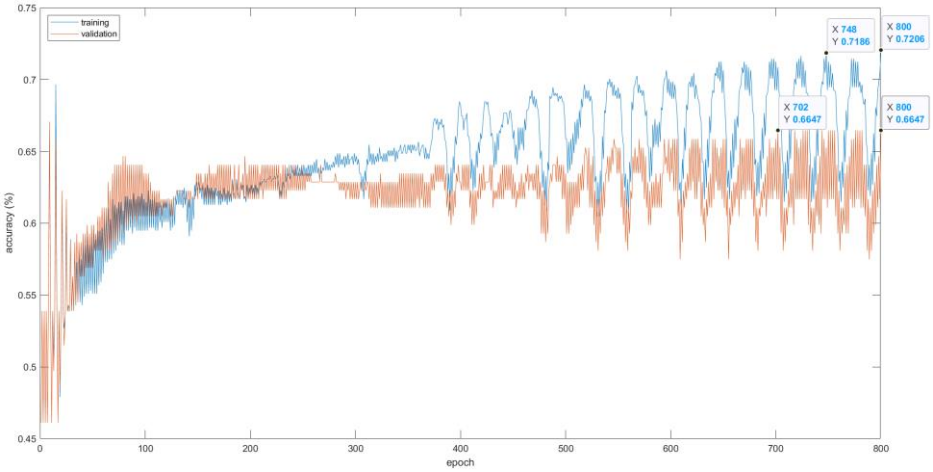

Fig. 67: Training and validation accuracy of sequential learning for 32px images

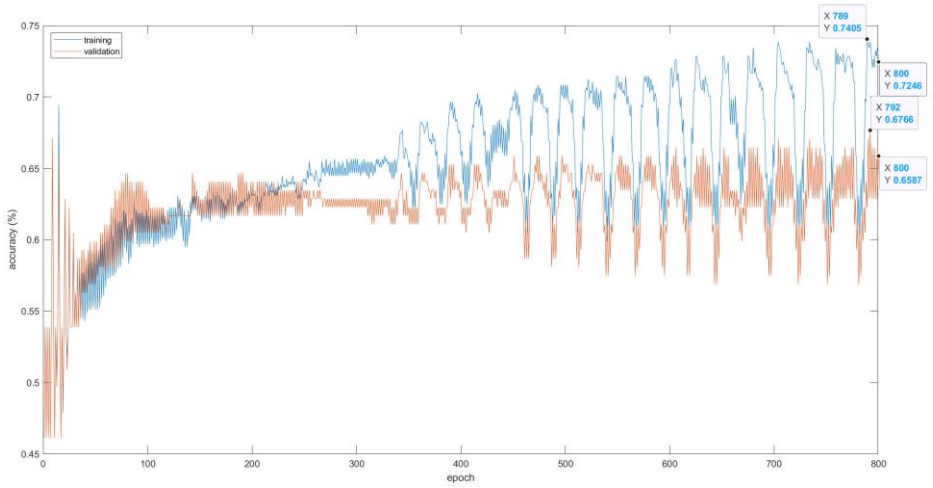

Fig. 68: Training and validation accuracy of sequential learning for 64px images

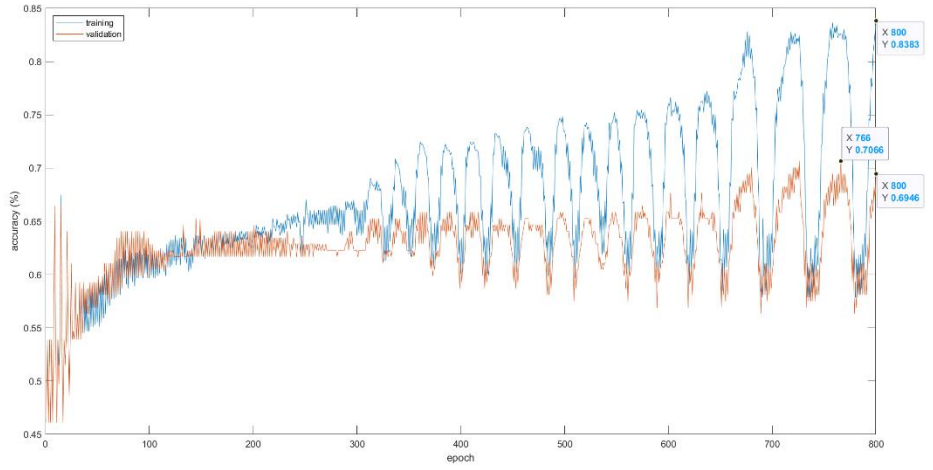

Fig. 69: Training and validation accuracy of sequential learning for 256px images

On the flipside, keeping size constant and altering the number of components, it is noticed that for batch training, there was a consistent pattern of the perceptron attaining a stable accuracy as the number of epochs increased, and as the number of components increased, it appeared that less epochs were required for the perceptron to obtain a stable classification accuracy. One other notable finding is that as the size of the image increased, less epochs were required for the image to obtain a stable classification accuracy, with the drop in number of epochs mostly stopping upon reaching the effective rank. These findings are tabulated in Table 4 below, with the bolded number of components highlighting the effective number of components.s 32 * 32 size was not accounted for as it had not stabilized even at the end of 200 epochs.

Size	Number of components	Epoch taken to stabilize with 200 epochs
256 * 256	1	NA
	16	100
	31	80
	151	80
	171	60
128 * 128	1	NA
	12	120
	23	100
	112	80
	127	80
64 * 64	1	NA
	6	NA
	12	180
	60	140
	63	140

Table 4. Epochs taken to stabilize for a given size and increasing number of components

In the case of sequential learning, as the number of components increased, there tended to be greater fluctuations in accuracies as the number of epochs increased, with the accuracy increasing and fluctuating significantly as the number of components increased for a constant image size. This observation is reflected in Fig. 70 to 72 in the following pages, where 256px images are used for analysis since it is the original dimension of the image.

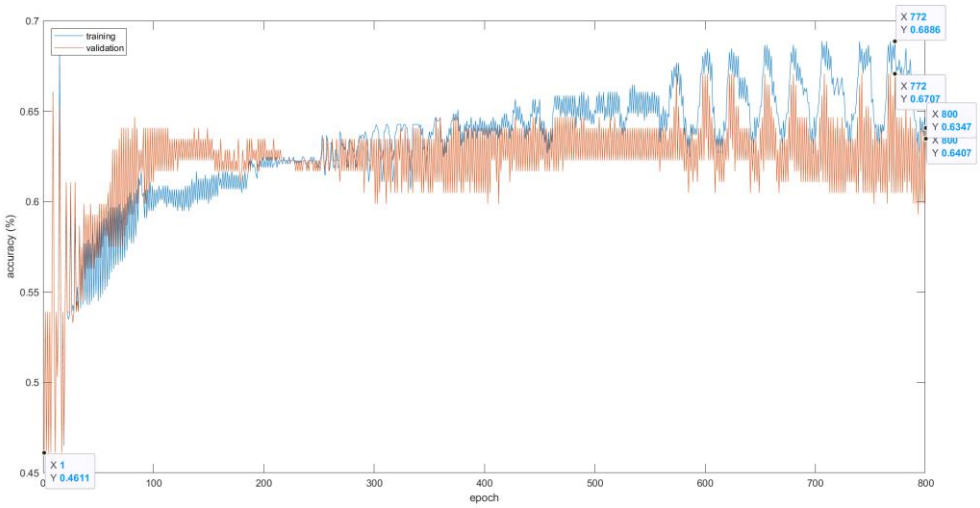

Fig. 70: Training and validation accuracy of 256px image, 1 component

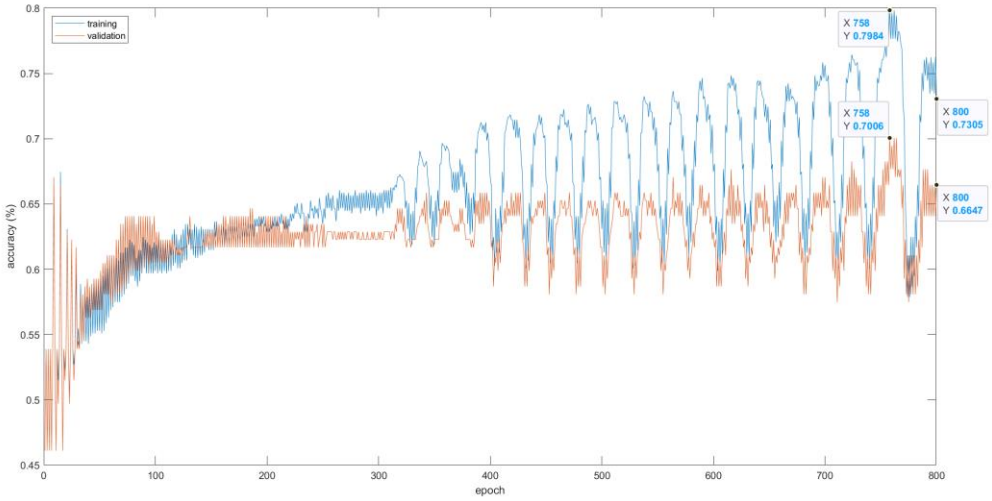

Fig. 71: Training and validation accuracy of 256px image, 16 components

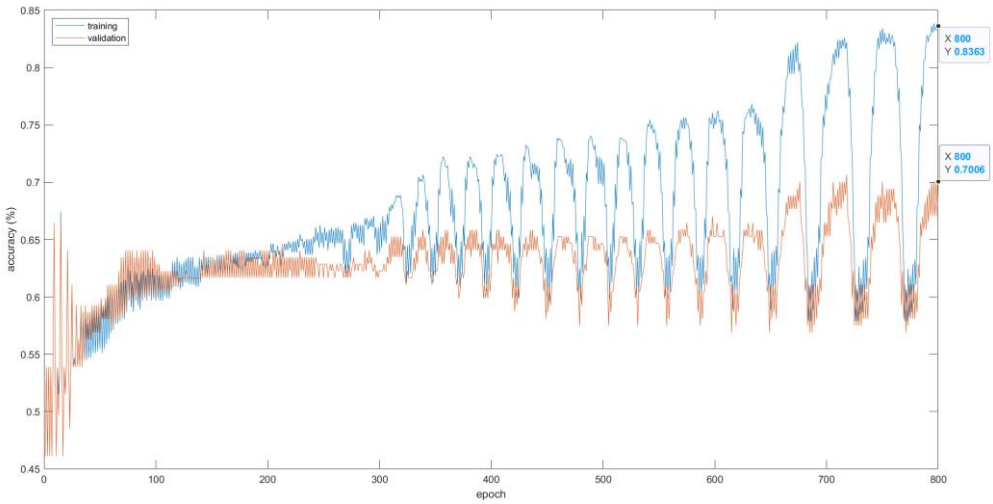

Fig. 72: Training and validation accuracy of 256px image, 151 components

Size	Components	Learning Method	Training Accuracy at end of run (%)	Validation Accuracy at end of run (%)
256 * 256	1	Batch	82.04	63.47
		Sequential	63.47	64.07
	16	Batch	100	71.26
		Sequential	73.05	66.47
	31	Batch	100	70.66
		Sequential	71.46	64.07
	151	**Batch**	**100**	**71.86**
		Sequential	**83.63**	**70.06**
	171	Batch	100	70.66
		Sequential	70.66	62.87
128 * 128	1	Batch	76.65	62.87
		Sequential	65.87	63.47
	12	Batch	99.80	68.86
		Sequential	73.45	65.27
	23	Batch	99.80	68.86
		Sequential	67.66	62.87
	112	**Batch**	**100**	**69.46**
		Sequential	**61.08**	**59.28**
	127	Batch	100	69.46
		Sequential	81.64	70.06
64 * 64	1	Batch	71.06	62.28
		Sequential	62.87	62.28
	6	Batch	79.24	64.07
		Sequential	70.06	66.47
	12	Batch	99.80	66.47
		Sequential	73.65	67.07
	60	**Batch**	**100**	**66.47**
		Sequential	**74.05**	**67.66**
	63	Batch	100	67.47
		Sequential	68.46	63.47
32 * 32	1	Batch	62.28	59.28
		Sequential	62.87	62.28
	2	Batch	83.03	68.26
		Sequential	67.86	64.07
	4	Batch	75.85	67.07
		Sequential	67.47	65.27
	7	Batch	87.23	68.86
		Sequential	62.67	62.28
	31	**Batch**	**68.06**	**64.07**
		Sequential	**72.06**	**66.47**

Table 5. Summarised Findings for Rosenblatt's Perceptron

Table 5 in the previous page is a summarized observation of the accuracies obtained with PCA and image resizing accounted for. It can be observed that at the effective number of components calculated using SVD with an accuracy of 99%, the image classification tended to be more accurate, with any inferior accuracies measured very likely due to noise like in the case of the 128 * 128 image. It should be worth noting that if 800 epochs had not been used, the image classification accuracy would very likely have been higher as recorded, thereby justifying the recording of the peak accuracy value in Table 2. Fig. 73 shows how the noise had affected the reading in the case of the 128 * 128 image.

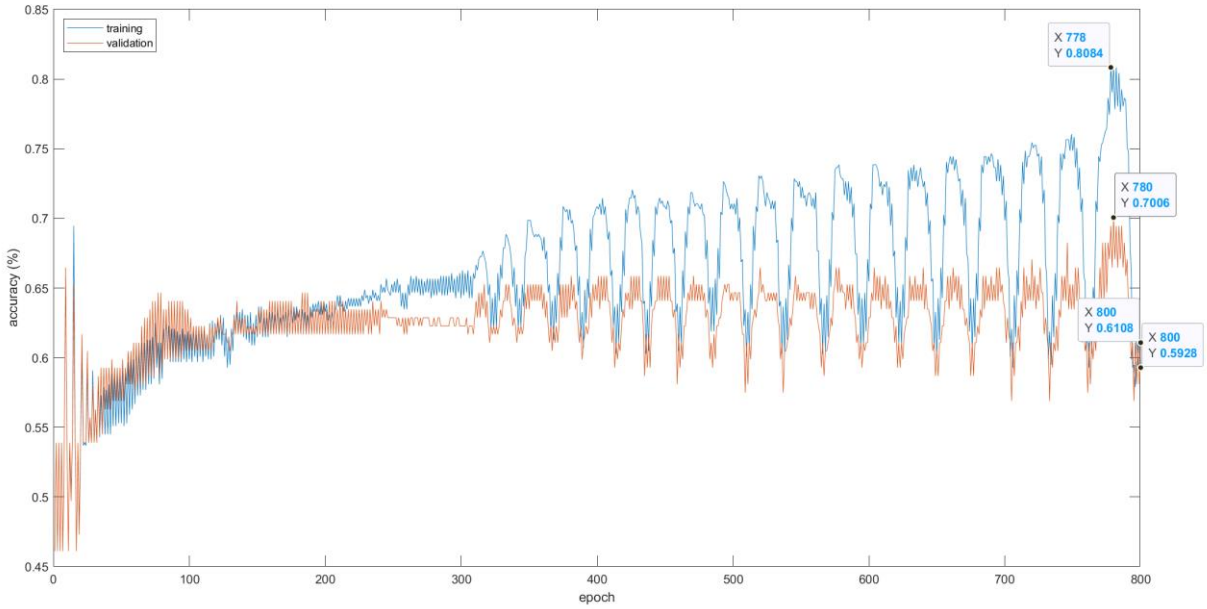

Fig. 73. 128px image with 112 effective components. This is the value derived by SVD.

Notice how the accuracy at the 778th epoch when 112 effective components were used was higher than the accuracy at the 800th epoch.

Main sequential training function:

```
clear;

n_train = 501;
n_val = 167;
eta = 0.01;
epochs = 8e2;
scales = [1, 0.5, 0.25, 0.125];
training_sets = [1:1:501];

filepath_train = "group_3\\train";
train_folder = dir(filepath_train);

filepath_val = "group_3\\val";
val_folder = dir(filepath_val);

for j=1:length(scales)

   dim = 256*scales(j);

   % redefining the data sets according to scales, full clarity
   [training_data, training_label] = extract_img_set(filepath_train,
train_folder, n_train, scales(j));
   [validation_data, validation_label] = extract_img_set(filepath_val,
val_folder, n_val, scales(j));
   epoch = [1:1:epochs];

   n_components = extract_component_numbers(filepath_train, train_folder,
n_train, dim);

   dimensions_to_test = [1, ceil(n_components/20), ceil(n_components/10),
ceil(n_components/5), ceil(n_components/2), n_components, n_components + 20,
n_components, min(dim-1, n_components+20)];
   display(dimensions_to_test);

   for k=1:length(dimensions_to_test)

       train_pca_data = zeros([dim^2, n_train]);

       for l=3:n_train+2
           [coeff, img, x_form] = pca_img(filepath_train, train_folder, l,
dimensions_to_test(k), scales(j));
           train_pca_data(:,l-2) = x_form(:);
       end

       filename =
sprintf("q3b_sequential\\sequential_epoch_%d_components_%dpx",
dimensions_to_test(k), dim);
       [net, accu_train_pca, accu_val_pca] = train_seq(train_pca_data,
training_label, validation_data, validation_label, epochs, filename);
   end

   filename = sprintf("q3b_sequential\\sequential_%dpx", dim);
   [net, accu_train, accu_val] = train_seq(training_data, training_label,
validation_data, validation_label, epochs, filename);
end
```

Sequential training code:

```
function [net, accu_train, accu_val] = train_seq(training_data,
training_label, validation_data, validation_label, epochs, filename)

    net = perceptron;
    net.trainParam.epochs = epochs;
    epoch = [1:1:epochs];

    accu_train = zeros(1, epochs);
    accu_val = zeros(1, epochs);

    % Training loop
    for i=1:epochs
        idx = randperm(size(training_data, 2));
        net = adapt(net, training_data(:,idx), training_label(:,idx));

        pred_train = net(training_data(:,idx));
        accu_train(i) = 1 - mean(abs(pred_train - training_label(:,idx)));

        val_train = net(validation_data);
        accu_val(i) = 1 - mean(abs(val_train - validation_label));
    end

    plot(epoch, accu_train, epoch, accu_val); % Sequential plotting code here
    xlabel("epoch");
    ylabel("accuracy (%)");
    legend({'training', 'validation'}, 'Location', 'northwest');
    saveas(gcf, filename, 'png');
end
```

Main batch training function. Initializations are identical to that of sequential learning.

```
for h=1:length(scales)

    dim = 256*scales(h);
    [training_data, training_label] = extract_img_set(filepath_train,
train_folder, n_train, scales(h));
   [validation_data, validation_label] = extract_img_set(filepath_val,
val_folder, n_val, scales(h));
   epoch = [1:1:epochs];

   n_components = extract_component_numbers(filepath_train, train_folder,
n_train, dim);

   dimensions_to_test = [1, ceil(n_components/20), ceil(n_components/10),
ceil(n_components/5), ceil(n_components/2), n_components, n_components+20,
max(dim-1, n_components+20)];

   for i=1:length(dimensions_to_test)

       display("dimensions:", num2str(dimensions_to_test(i)));

       train_pca_data = zeros([dim^2, n_train]);
       accu_train = zeros(1, length(epoch_test));
       accu_val = zeros(1, length(epoch_test));

       for l=3:n_train+2
           [coeff, img, x_form] = pca_img(filepath_train, train_folder, l,
dimensions_to_test(i), scales(h));
           train_pca_data(:,l-2) = x_form(:);
       end

       for j=1:length(epoch_test)

           net = perceptron;
           epochs = epoch_test(j);
           net.trainParam.epochs=epochs;
           epoch = [0:1:epochs-1];

           net = train(net, train_pca_data, training_label); % Training code.

           train_count = [1:1:n_train];
           train_out = net(training_data);

           val_count = [1:1:n_val];
           val_out = net(validation_data);

           accuracy_train = 1 - mean(abs(train_out - training_label));
           accuracy_val = 1 - mean(abs(val_out - validation_label));

           accu_train(j) = accuracy_train;
           accu_val(j) = accuracy_val;
       end
       % Batch training plotting code here. Refer to next page.
   end
end
```

Batch training plotting code:

```
display("Components: ", num2str(dimensions_to_test(i)));

display("training accuracies: ");
display(accu_train);

display("validation accuracies: ");
display(accu_val);

filename = sprintf("q3b_batch\\batch_accuracy_%dpx_%d_comps",dim,
dimensions_to_test(i));
plot(epoch_test, accu_train, epoch_test, accu_val);
xlabel("epochs");
ylabel("accuracy");
legend({"training", "validation"}, 'Location', 'northeast');
saveas(gcf, filename, 'png');
```

3c) Two training algorithms were considered: *trainscg* and *trainrp*. For both situations, the learning rate of $\eta = 0.01$ was considered and the transfer functions of *tansig* and *logsig* were considered. These transfer functions were considered because of how the values in each cell of the image was not simply '0' or '1' due to the greyscale nature of the image. Although the maximum and minimum values of the images were not exactly known, it is quite likely the values ranged between 0 to 255. Only 256*256 pixel images were considered in this analysis. In approaching the question, the analysis was conducted with considerations to the number of effective components as well, similar to that of 3b). The single perceptron compared with Tables 6 to 9 has also undergone batch training.

It is observed that as the number of components increased, similar to the single perceptron, the number of epochs required for the accuracies to stabilize decreased as well. The comparison is made inTable 6 below which records the training data for the **training set** only, where *trainscg* is used.

Components	Single Perceptron		Multi-layer Perceptron	
	Epochs to stabilize	Accuracy (%)	Epochs to stabilize	Accuracy (%)
1	NA	NA	180	99.20
16	80	100	100	100
31	80	100	60	100
151	60	100	60	100
171	80	100	60	100

Table 6. Training accuracy comparison between single and multi-layer perceptron (*trainscg*)

In Table 7 below, the minimum and maximum accuracy for the **validation data** reflected by the multi-layer perceptron is recorded and then compared to that of a single perceptron, as the value of the perceptron tends to vary very slighly in the case of the multi-layer perceptron but not the single perceptron.

Components	Single Perceptron	Multi-layer Perceptron		
	Stabilized Accuracy (%)	Minimum accuracy (%)	Maximum accuracy (%)	Average accuracy (%)
1	NA	NA	NA	NA
16	71.26% 80 epochs	69.54	70.78	70.16
31	70.66% 80 epochs	69.89	70.55	70.22
151	70.66% 60 epochs	70.35	73.15	71.75
171	68.26% 60 epochs	70.88	72.21	71.545

Table 7. Validation accuracy between single and multilayer perceptron (*trainscg*)

Based on Table 6, a multi-layer perceptron would take longer to train as compared to a single perceptron, as reflected by the number of epochs required when the number of components are small. However, when the effective number of components are used, the multi-layer perceptron appears to perform slightly better than that of the single-layer perceptron, as shown by the classification accuracy of the training set yielding the same accuracy as that of the single perceptron with less epochs required.

In the tabulation of data to form Table 7, the average performance of the multi-layer perceptron is calculated based on the equation below:

$$Accuracy_{avg}(\%) = \frac{Accuracy_{min} + Accuracy_{max}}{2}$$

The two values are obtained starting from the number of epochs where the accuracy of the single perceptron begins to stabilize. Here, once again, it appears that the number of components used affects the accuracy of the multi-layer perceptron; when the number of components used are small, the accuracy of the multi-layer perceptron pales very slightly relative to that of the single perceptron. However, when the number of components are sufficiently large, the accuracy of the multi-layer perceptron when *trainscg* is used begins to perform slightly better than that of the single perceptron. This indicates that for *trainscg*, the number of components is a factor to consider in the accuracy of the neural network in this context.

The data for *trainrp* is reflected in Tables 8 and 9. In the case of *trainrp*, 200 epochs was insufficient for the accuracy to stabilize. As such, the accuracy at the end of run for both cases will be accounted for instead as reflected in Tables 8 and 9.

Components	Single Perceptron	Multi-layer Perceptron
	Accuracy (%)	Accuracy(%)
1	82.04	52.11
16	100	64.30
31	100	70.23
151	**100**	**82.10**
171	100	83.31

Table 8. Training accuracy comparison with single and multi-layer perceptron (*trainrp)*

Components	Single Perceptron	Multi-layer Perceptron
	Accuracy (%)	Accuracy (%)
1	63.47	51.41
16	71.26	61.11
31	70.66	64.97
151	**70.66**	**72.61**
171	68.26	63.96

Table 9. Validation accuracy between single and multi-layer perceptron (*trainrp*)

Tables 8 and 9 reflect that when *trainrp* is used, the accuracy in classifying the training set will always be inferior no matter the number of principal components used in the image analysis relative to the single perceptron. However, when the number of components is equivalent to the one calculated by SVD with a performance ratio of 0.99, the accuracy of the multi-layer perceptron is indeed superior to the single perceptron, which in this case is 151 components (bolded for added clarity). More clearly, as the number of components increase, the use of *trainrp* also leads to a general increase in the classification accuracy of both the training and validation sets as shown in Tables 8 and 9 above.

Fig. 74 and 75 shows the differences between the accuracy relative to the number of epochs when *trainscg* and *trainrp* are used respectively. For both graphs, 151 components were used.

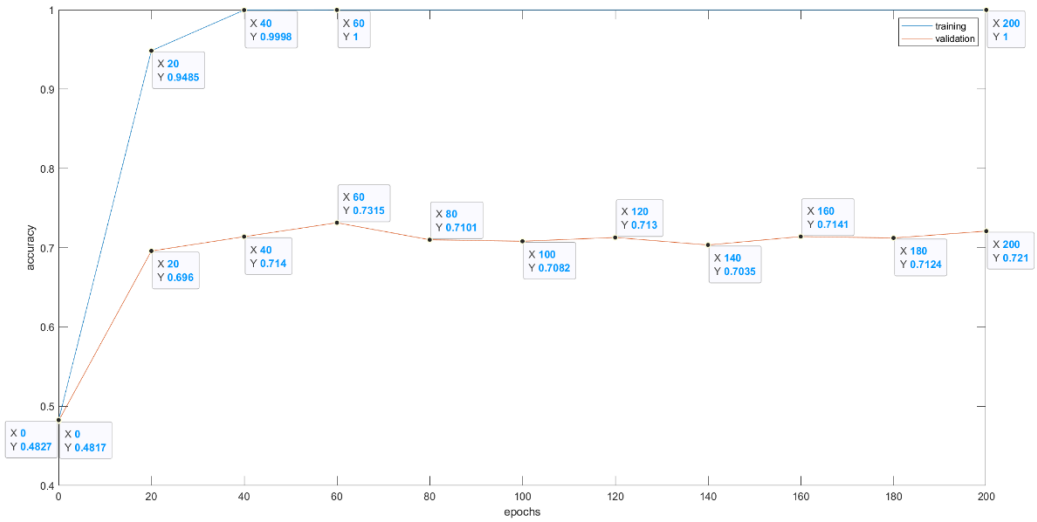

Fig. 74: Accuracy against epoch for multi-layer-perceptron 1-151-1 (*trainscg*)

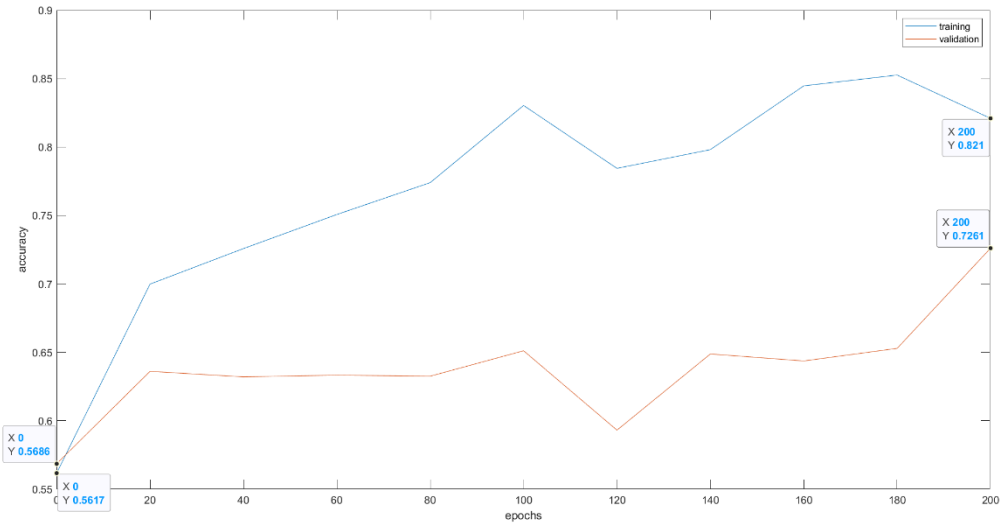

Fig. 75: Accuracy against epoch for multi-layer-perceptron 1-151-1 (*trainrp*)

It is evident that *trainscg* is able to yield a more stable training result for less number of epochs, indicating that with *trainscg*, less time is required to train a neural network for image classification for the allocated set of images.

Table 10 reflects the differences in accuracies for the neural network for the two different training algorithms after 200 epochs.

Components	Training accuracy for *trainscg* (%)	Training accuracy for *trainrp* (%)	Validation accuracy for *trainscg* (%)	Validation accuracy for *trainrp* (%)
1	99.20	52.11	71.10	51.41
16	100	64.30	69.67	61.11
31	100	70.23	70.55	64.97
151	**100**	**82.10**	**72.10**	**72.61**
171	100	83.31	70.76	63.96

Table 10. Training and validation accuracy of *trainscg* and *trainrp*

When *trainscg* is used, the training accuracy is better than that of *trainrp*, regardless of the number of components used. In general, for the validation accuracy, *trainscg* tends to perform better than that of *trainrp*, except for when 151 components were used. However, that difference in accuracy is just 0.51%, which is not appreciably large. All in all, *trainscg* appears to be a better training algorithm in use for a multi-layer perceptron as compared to *trainrp.*

Batch training initialization:

```
clear;

% train has 501 items. val has 167 items.
n_train = 501;
n_val = 167;
eta = 0.01;
epochs = 8e2;
training_sets = [1:1:501];

% Defining training data
filepath_train = "group_3\\train";
train_folder = dir(filepath_train);

% Retrieve validation data
filepath_val = "group_3\\val";
val_folder = dir(filepath_val);

% redefining the data sets according to scales, full clarity
[training_data, training_label] = extract_img_set(filepath_train,
train_folder, n_train, 1);
[validation_data, validation_label] = extract_img_set(filepath_val,
val_folder, n_val, 1);
epoch = [1:1:epochs];

n_components = extract_component_numbers(filepath_train, train_folder,
n_train, 256);

dimensions_to_test = [1, ceil(n_components/20), ceil(n_components/10),
ceil(n_components/5), n_components, n_components+20, 255];
display(dimensions_to_test);
```

Main batch training function:

```
for h=1:length(dimensions_to_test)

    accu_train = zeros(1, length(epoch_test));
    accu_val = zeros(1, length(epoch_test));

    for i=1:length(epoch_test)

        net = patternnet(dimensions_to_test(h));
        net.trainFcn = 'trainscg';
        net.layers{1}.transferFcn = 'tansig';
        net.layers{2}.transferFcn = 'logsig';
        net.trainParam.lr = eta;
        net.divideFcn = 'dividetrain';
        net.trainParam.epochs=epoch_test(i);

        net = train(net, training_data, training_label); % Training line

        train_out = net(training_data);
        acc_train(i) = 1-mean(abs(train_out - training_label));

        val_out = net(validation_data);
        acc_val(i) = 1- mean(abs(val_out - validation_label));
    end

    display("Components: ", num2str(dimensions_to_test(h)));

    display("training accuracies: ");
    display(acc_train);

    display("validation accuracies: ");
    display(acc_val);

    % Plotting code below
    filename = sprintf("q3c\\%d_comp_batch_accuracy", dimensions_to_test(h));
    plot(epoch_test,  acc_train, epoch_test, acc_val);
    xlabel("epochs");
    ylabel("accuracy");
    legend({"training", "validation"}, 'Location', 'northeast');
    saveas(gcf, filename, 'png');
end
```

3d) Similar to that of 3c, two training algorithms were considered: *trainscg* and *trainrp*. For both situations, the learning rate of $\eta = 0.01$ was considered and the transfer functions of *tansig* and *logsig* were considered, for the same considerations as mentioned in 3c. In approaching the question, the analysis was conducted with considerations to the number of effective components as well, similar to that of 3b). Tables 11 shows the comparisons between a single perceptron, a multi-layer perceptron that has undergone sequential training of 800 epochs with *trainscg* and that of *trainrp*. Also, 256 * 256 px images will be used as well.

Components	Single perceptron		Multi-layer perceptron with *trainscg*		Multi-layer perceptron with *trainrp*	
	Training accuracy (%)	Validation accuracy (%)	Training accuracy (%)	Validation accuracy (%)	Training accuracy (%)	Validation accuracy (%)
1	63.47	64.07	92.26	67.72	92.39	69.40
16	73.05	66.47	99.06	71.02	99.00	70.94
31	71.46	64.07	99.31	70.97	99.36	71.01
151	83.63	70.06	99.53	72.83	99.60	71.74
171	70.66	62.87	99.59	71.05	99.56	71.02

Table 11. Training and validation accuracies after sequential training of 800 epochs

Based on Table 11 above, a multi-layer perceptron that has undergone sequential learning performs significantly better than that of the single perceptron in both training and validation image classification, regardless of the training algorithm used. It also shows that as the number of components increase, there is an increase in the classification accuracy of both the training and validation set, with no appreciable difference when either *trainscg* or *trainrp* was used for the sequential learning. As both training algorithms required approximately the same time to train, it can be argued that in this allocated set of images, both *trainscg* or *trainrp* are equally applicable.

The accuracy against epoch plots are available in Fig. 76 below, which clearly shows how the noise generated by sequential learning is at large minimized beyond a specific epoch for both the training and validation sets for a multi-layer perceptron, with the accuracy converging towards a specific value as the number of epochs continue to increase. In contrast, with the noise increasing significantly for a single perceptron, it is clear that the increase in accuracy for both the training and validation set is coupled with sharp decreases, followed by dramatic increases after a specific number of epochs, but ultimately still inferior to that of a multi-layer perceptron. With these observations, it can be argued that not just in terms of accuracy

values, but also in terms of stability, a multi-layer perceptron is more suited for image classification tasks.

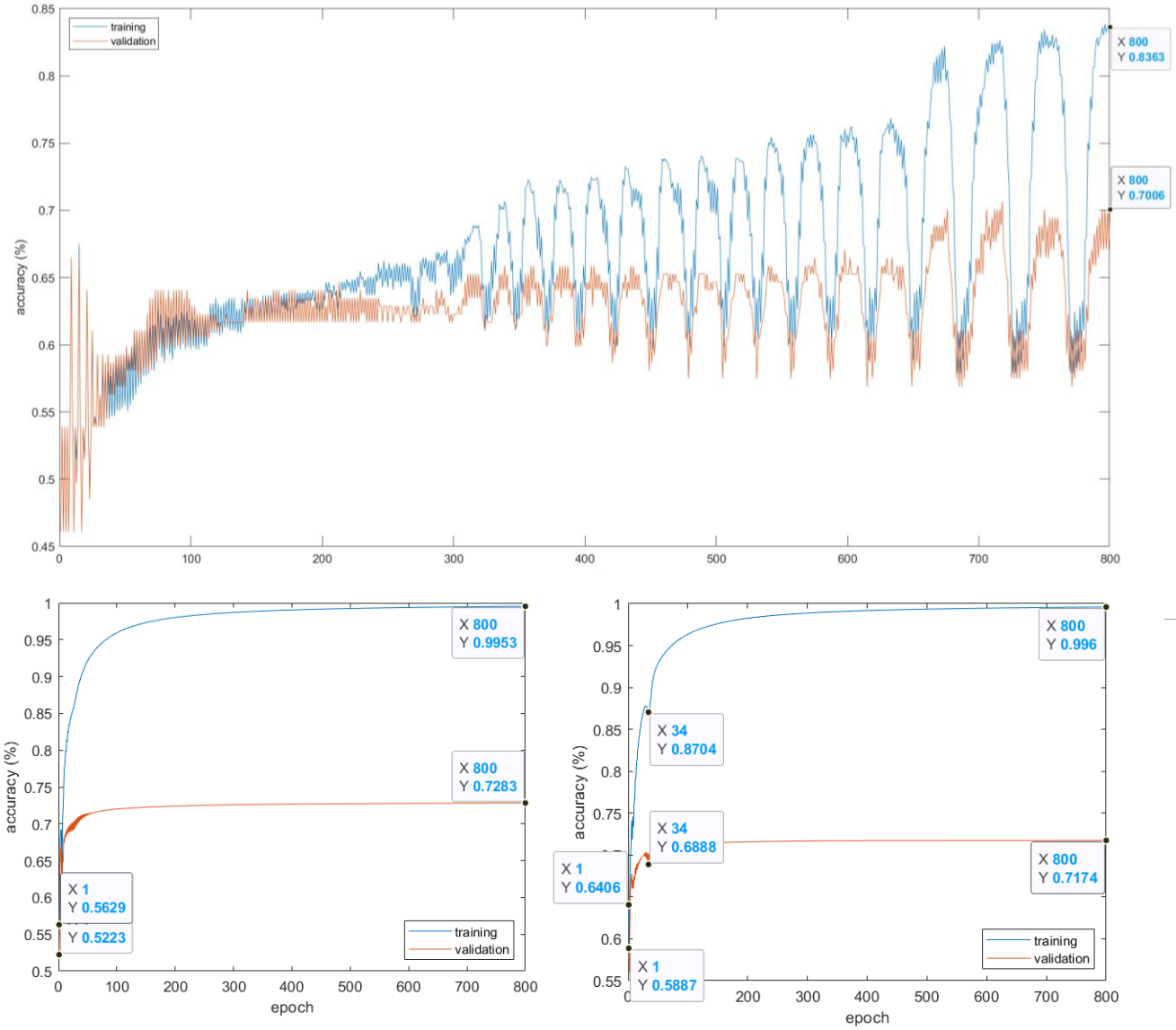

Fig. 76 Accuracy against epoch plots for a single perceptron (*top*), a multi-layer perceptron using *trainscg* (*bottom-left*) and a multi-layer perceptron using *trainrp* (*bottom-right*)

It should be noted however, that in sequential learning, it may be beneficial to set a sufficiently high number of epochs, as it is shown during the training of the multi-layer perceptron that there was appreciable noise when the number of epochs were small, further giving credit to the number of epochs in helping to eliminate the noise during the learning process. This is once again evident in Fig. 76, where fluctuations in accuracy were detected for both *trainscg* and *trainrp* when the number of epochs was less than 50.

Table 12 and 13 below shows how the differences in learning methods adopt can affect the classification accuracy of both the training and validation set respectively when the same training algorithms are used for a multi-layer perceptron. All values in Table 11 are taken at the end of the training procedure. 200 epochs were used for the batch training and 800 epochs were used for the sequential training, as it was observed that their accuracies had insignificant changes beyond the prescribed epochs.

Components	*trainscg*		*trainrp*	
	Batch Accuracy (%)	Sequential Accuracy (%)	Batch Accuracy (%)	Sequential Accuracy (%)
1	99.20	92.26	52.11	92.39
16	100	99.06	64.30	99.00
31	100	99.31	70.23	99.36
151	**100**	**99.53**	**82.10**	**99.60**
171	100	99.59	83.31	99.56

Table 12. Training accuracies for *trainscg* and *trainrp*

Components	*trainscg*		*trainrp*	
	Batch Accuracy (%)	Sequential Accuracy (%)	Batch Accuracy (%)	Sequential Accuracy (%)
1	71.10	67.72	51.41	69.40
16	69.67	71.02	61.11	70.94
31	70.55	70.97	64.97	71.01
151	**72.10**	**72.83**	**72.61**	**71.74**
171	70.76	71.05	63.96	71.02

Table 13. Validation accuracies for *trainscg* and *trainrp*

When *trainscg* was used, classification of the training set was more accurate when batch training was used. However, for the validation set, sequential learning appeared to be slightly more effective when *trainscg* was used. Thus, *trainscg* can to some extent, be used for either batch and sequential learning methods, given the minute difference in accuracy for both training and validation sets. However, the use of *trainscg* required more epochs and thus, time to train the multi-layer perceptron when sequential learning was used. This is reflected in Fig. 77.

When *trainrp* was used, classification of the training set was more accurate when sequential training was used. The same pattern appeared when validation accuracies were compared as well. Moreover, when comparing the accuracies of the two learning methods, it appeared that batch learning was unable to converge to a specific accuracy given 200 epochs, but for sequential learning, the accuracy appeared to continually increase and eventually converge. This observation is seen in Fig. 78. Both Fig. 77 and 78 used 151 components.

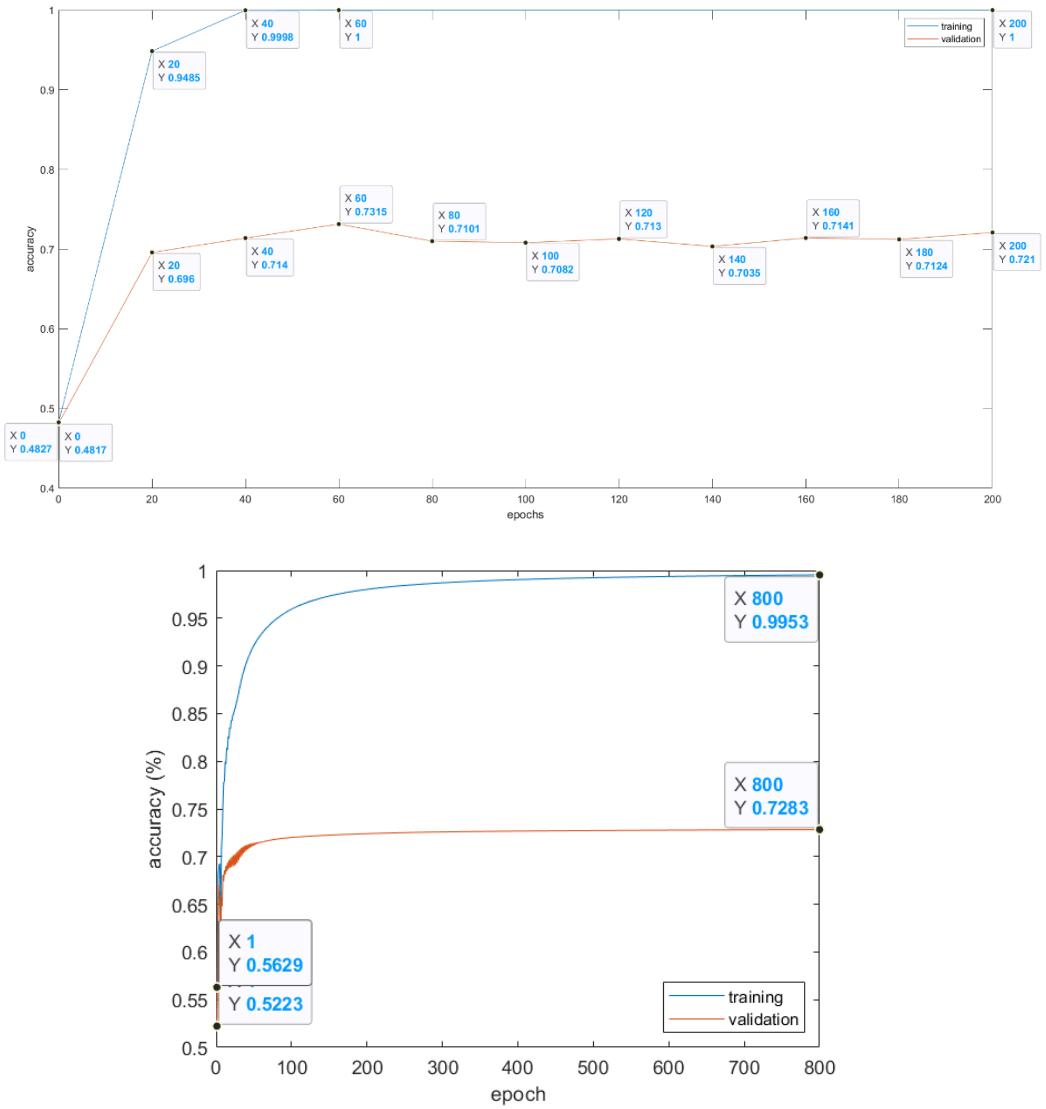

Fig. 77 Comparison of accuracy convergence for batch (*above*) and sequential learning (*below)* for *trainscg*

In Fig. 77, the significant difference in number of epochs required for the multi-layer perceptron to converge to a given accuracy for the validation set is apparent when 151 components were used. For batch learning, it was observed that during the 60th epoch, the accuracy of the validation set had begun to merely fluctuate very slightly at 70 to 72%, indicating a form of convergence. However, this was observable in sequential learning towards the 100th epoch, where the accuracy tended towards 72.83% in the 800th epoch. However, due to the gentle increase as shown above, it could be that more epochs were required, further highlighting that despite the inferior accuracies generated by the batch

learning method, it could be trained significantly faster than that of the sequential learning method when *trainscg* was used.

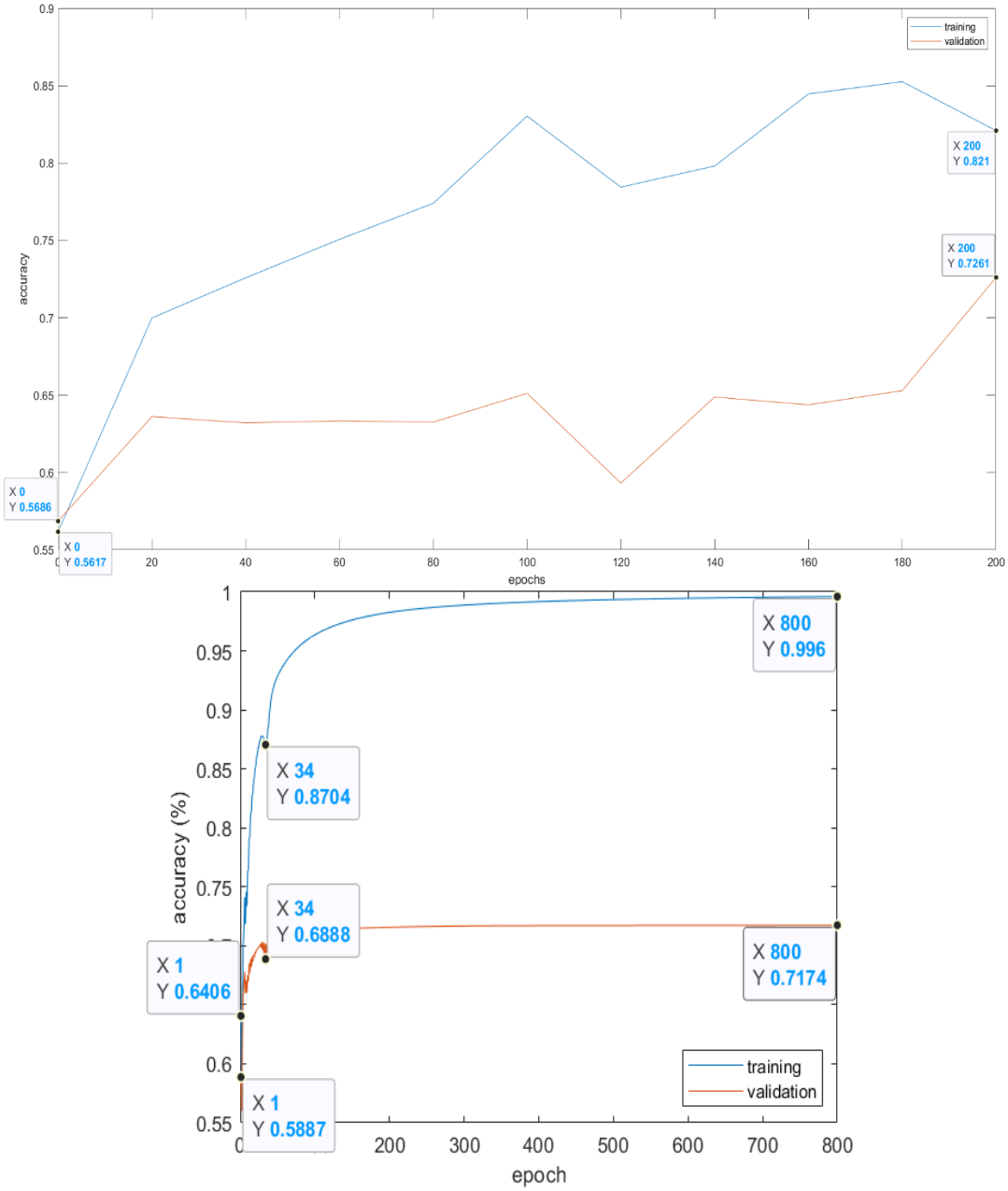

Fig. 78 Comparison of accuracy convergence for batch (*above*) and sequential learning (*below)* for *trainrp*

In Fig. 78, it is evident that for 200 epochs, unlike that of *trainscg*, the multi-layer perceptron never converges, but rather, fluctuates significantly. This is tremendously contrasted by that of the sequential learning method, where fluctuations at large ended by the 34th epoch, only to converge to an appreciable accuracy of 99.60% and 71.74% for the training and validation set respectively. This indicates that despite the significantly less time incurred in training a multi-layer perceptron for *trainrp*, its accuracy tends to be inferior for the allocated set of images,

and as such, it would be better to use sequential learning if *trainrp* is the learning algorithm in consideration.

Sequential training initialization:

```
clear;

% train has 501 items. val has 167 items.
n_train = 501;
n_val = 167;
eta = 0.01;
epochs = 8e2;
training_sets = [1:1:501];

% Defining training data
filepath_train = "group_3\\train";
train_folder = dir(filepath_train);

% Retrieve validation data
filepath_val = "group_3\\val";
val_folder = dir(filepath_val);

% redefining the data sets according to scales, full clarity
[training_data, training_label] = extract_img_set(filepath_train,
train_folder, n_train, 1);
[validation_data, validation_label] = extract_img_set(filepath_val,
val_folder, n_val, 1);
epoch = [1:1:epochs];

n_components = extract_component_numbers(filepath_train, train_folder,
n_train, 256);

dimensions_to_test = [1, ceil(n_components/20), ceil(n_components/10),
ceil(n_components/5), n_components, n_components+20, 255];
display(dimensions_to_test);
```

Main sequential training function:

```
% change number of components
for k=1:length(dimensions_to_test)
   filename = sprintf("q3d\\sequential_epoch_%d_components",
dimensions_to_test(k));
   [net, accu_train_pca, accu_val_pca] = train_seq(training_data,
training_label, validation_data, validation_label, epochs, filename,
dimensions_to_test(k));
end

% sequential training subroutine
function [net, accu_train, accu_val] = train_seq(training_data,
training_label, validation_data, validation_label, epochs, filename, comp)
    net = patternnet(comp);
    net.trainFcn = 'trainrp'; %change with trainrp
    net.trainParam.lr = 0.01;
    net.layers{1}.transferFcn = 'tansig';
    net.layers{2}.transferFcn = 'logsig';
    net.trainParam.epochs=epochs;
    epoch = [1:1:epochs];

    accu_train = zeros(1, epochs);
    accu_val = zeros(1, epochs);

    % Training loop
    for i=1:epochs
        idx = randperm(size(training_data, 2));
        net = adapt(net, training_data(:,idx), training_label(:,idx));

        pred_train = net(training_data(:,idx));
        accu_train(i) = 1 - mean(abs(pred_train - training_label(:,idx)));

        val_train = net(validation_data);
        accu_val(i) = 1 - mean(abs(val_train - validation_label));
    end

    plot(epoch, accu_train, epoch, accu_val); % Sequential plotting code
    xlabel("epoch");
    ylabel("accuracy (%)");
    legend({'training', 'validation'}, 'Location', 'southeast');
    saveas(gcf, filename, 'png');
end
```