EE5904 Neural Networks: Homework 2

Zhang Fei A0117981x

Q1. Rosenbrock's Valley Problem

a). Steepest (Gradient) descent method:

Source code in Python:

```
import numpy as np
import random
import matplotlib.pyplot as plt

def func_valley(xy):
    x = xy[0][0]
    y = xy[0][1]
    return (1-x)**2 + 100*(y-x**2)**2

def func_dx(x,y):
    return -2 + 2*x - 400*(y-x**2)*x

def func_dy(x,y):
    return 200*(y-x**2)

def grad_func(xy):
    x = xy[0][0]
    y = xy[0][1]
    return np.array([func_dx(x,y),func_dy(x,y)]).reshape(1,2)

xy = np.array([random.uniform(0,0.5),random.uniform(0,0.5)]).reshape(1,2)

print(xy)
iteration = 0
learning_rate = 0.001
x_trajectory = []
y_trajectory = []
v_xy = []
while func_valley(xy) > 0.01 and iteration < 6000:
    x_trajectory.append(xy[0][0])
    y_trajectory.append(xy[0][1])
    v_xy.append(func_valley(xy))
    xy = xy - learning_rate*grad_func(xy)
    iteration += 1
print(iteration)
print('initial x value is '+ str(x_trajectory[0]) +' initial y value is '+ str(y_trajectory
[0]))
print('final x value is '+ str(x_trajectory[-1]) +' final y value is '+ str(y_trajectory[-
1]))

fig = plt.figure()
plt.subplot(1,2,1)
plt.title('Trajectory of (x, y) after %d iterations' % iteration)
plt.xlabel ("x")
plt.ylabel ("y")
plt.plot(x_trajectory,y_trajectory)

plt.subplot(1,2,2)
plt.title('Value of f(x, y) versus number of iterations')
plt.xlabel ("Iteration")
plt.ylabel ("Value")
plt.plot(np.arange(iteration),v_xy)
fig.set_size_inches(8, 5, forward=True)
fig.tight_layout()
plt.savefig('Q1a.png')
plt.show()
```

The gradient can be computed as: g = [-2 + 2x – 400x(y-x^2) 200(y-x^2)]^T. The converge criteria is when value of f(x,y) is < e = 0.0000001. The iteration loop stops when the condition is met. The initial parameters x and y is [0.08381745243420341 0.04145318944720866]^T

When the learning rate η is set to be 0.001, it takes 17128 iterations to meet converge condition. The trajectory of (x, y) is shown in Figure 1.

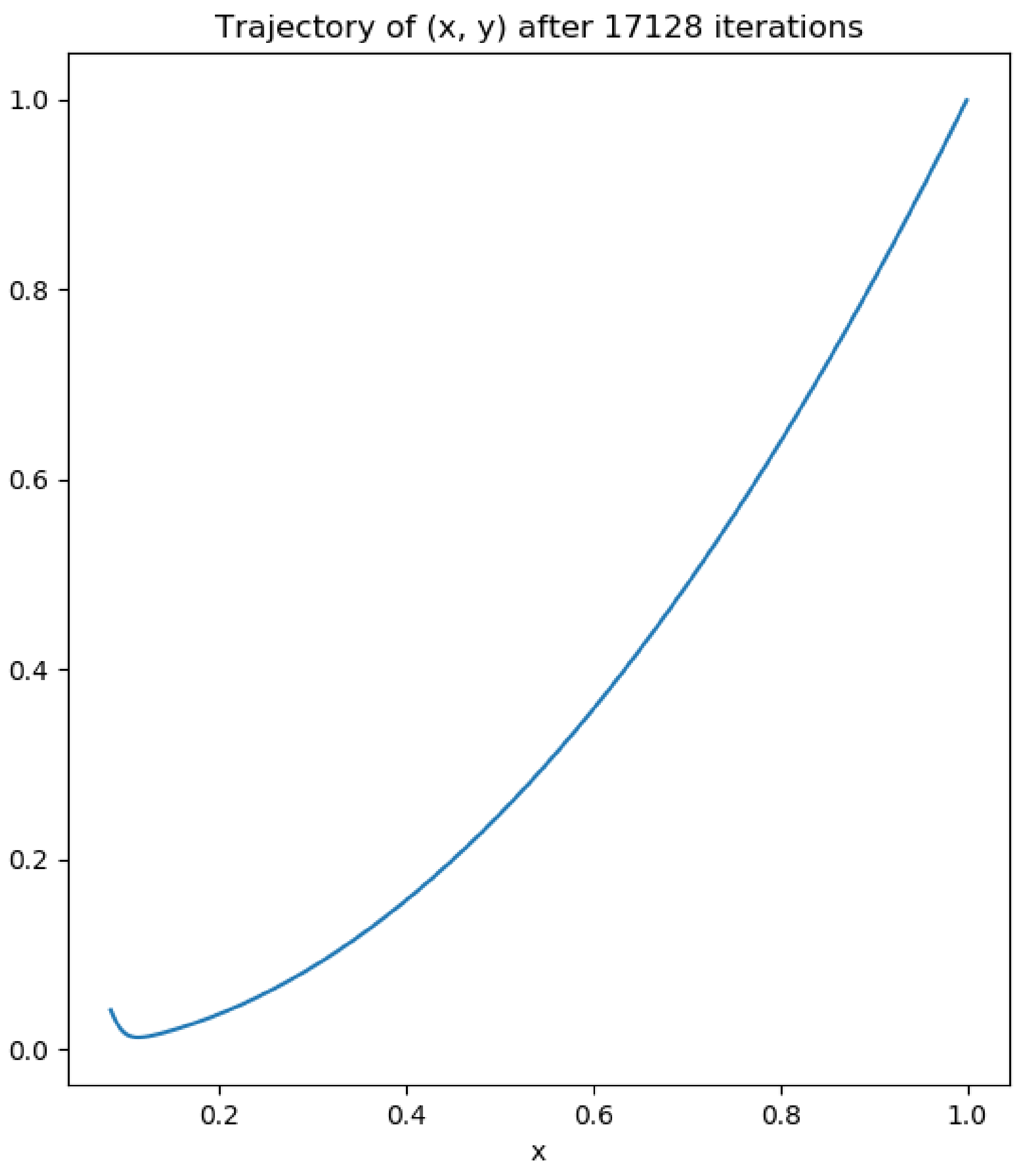


Figure 1 Trajectory of (x, y) after 17128 iterations for Gradient descent

The final value x = 0.9996839085018239, y = 0.9993666518058181 when iteration stops. The trajectory of value f(x,y) versus number of iteration is given in Figure 2. And the value of f(x,y) has a very sharp descent at first 2500 iterations. Then it converges slowly towards optimal value.

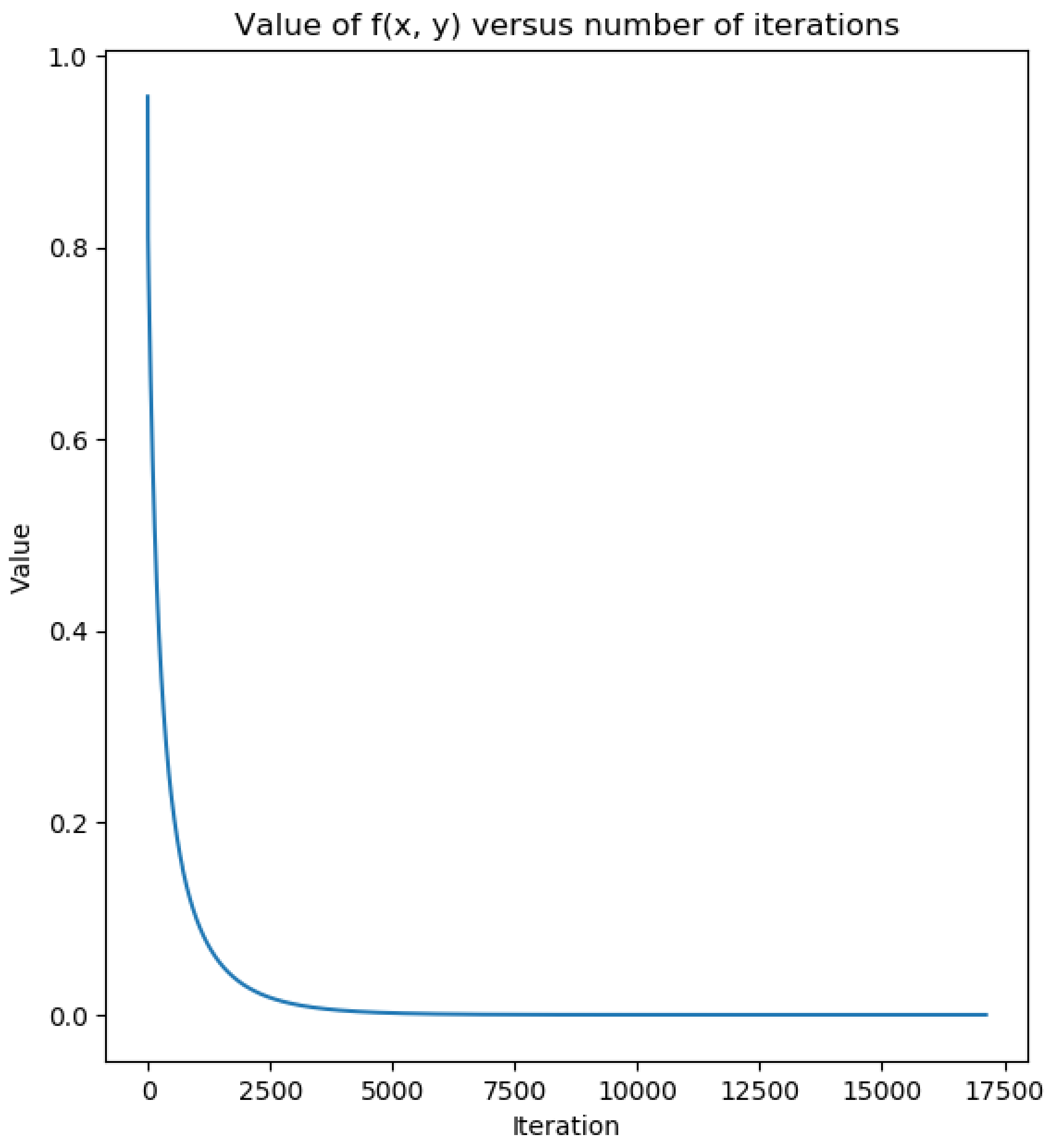


Figure 2 Value of f (x, y) versus number of iterations for Gradient descent

When the learning rate is set to be 0.2, the value of x rises to -inf, y rises to inf, both f and gradient becomes NaN, therefore, if learning rate is set too large, gradient descent can overshoot the minimum. It may fail to converge, or even diverge.

b). Newton's method
Source Code:

```python
import numpy as np
import random
import matplotlib.pyplot as plt

def func_valley(xy):
    x = xy[0][0]
    y = xy[1][0]
    return (1-x)**2 + 100*(y-x**2)**2

def func_dx(x,y):
    return -2 + 2*x - 400*(y-x**2)*x

def func_dy(x,y):
    return 200*(y-x**2)

def grad_func(xy):
    x = xy[0][0]
    y = xy[1][0]
    return np.array([func_dx(x,y),func_dy(x,y)]).reshape(2,1)

def hessian(xy):
    x = xy[0][0]
    y = xy[1][0]
    return np.array([[-400*y+2+1200*x**2,-400*x],[-400*x,200]])

xy = np.array([random.uniform(0,0.5),random.uniform(0,0.5)]).reshape(2,1)
# xy = np.array([0.08381745243420341,0.04145318944720866]).reshape(2,1)
print(xy)
iteration = 0

x_trajectory = []
y_trajectory = []
v_xy = []
while func_valley(xy) > 0.0000001:
    x_trajectory.append(xy[0][0])
    y_trajectory.append(xy[1][0])
    v_xy.append(func_valley(xy))
    xy = xy - np.linalg.inv(hessian(xy)).dot(grad_func(xy))
    print(xy)
    iteration += 1
print(iteration)
print('initial x value is '+ str(x_trajectory[0]) +' initial y value is '+ str(y_tr
ajectory[0]))
print('final x value is '+ str(x_trajectory[-
1]) +' final y value is '+ str(y_trajectory[-1]))

fig = plt.figure()
plt.subplot(1,2,1)
plt.title('Trajectory of (x, y) after %d iterations' % iteration)
plt.xlabel ("x")
plt.ylabel ("y")
plt.plot(x_trajectory,y_trajectory)

plt.subplot(1,2,2)
plt.title('Value of f(x, y) versus number of iterations')
plt.xlabel ("Iteration")
plt.ylabel ("Value")
plt.plot(np.arange(iteration),v_xy)
fig.set_size_inches(8, 5, forward=True)
fig.tight_layout()
plt.savefig('Q1b.png')
plt.show()
```

With the same initial values of x and y from previous gradient descent method, and same converge condition, by using Newton’s method, it only has 10 iterations to meet the converge condition. The trajectory of (x, y) is shown in Figure 3.

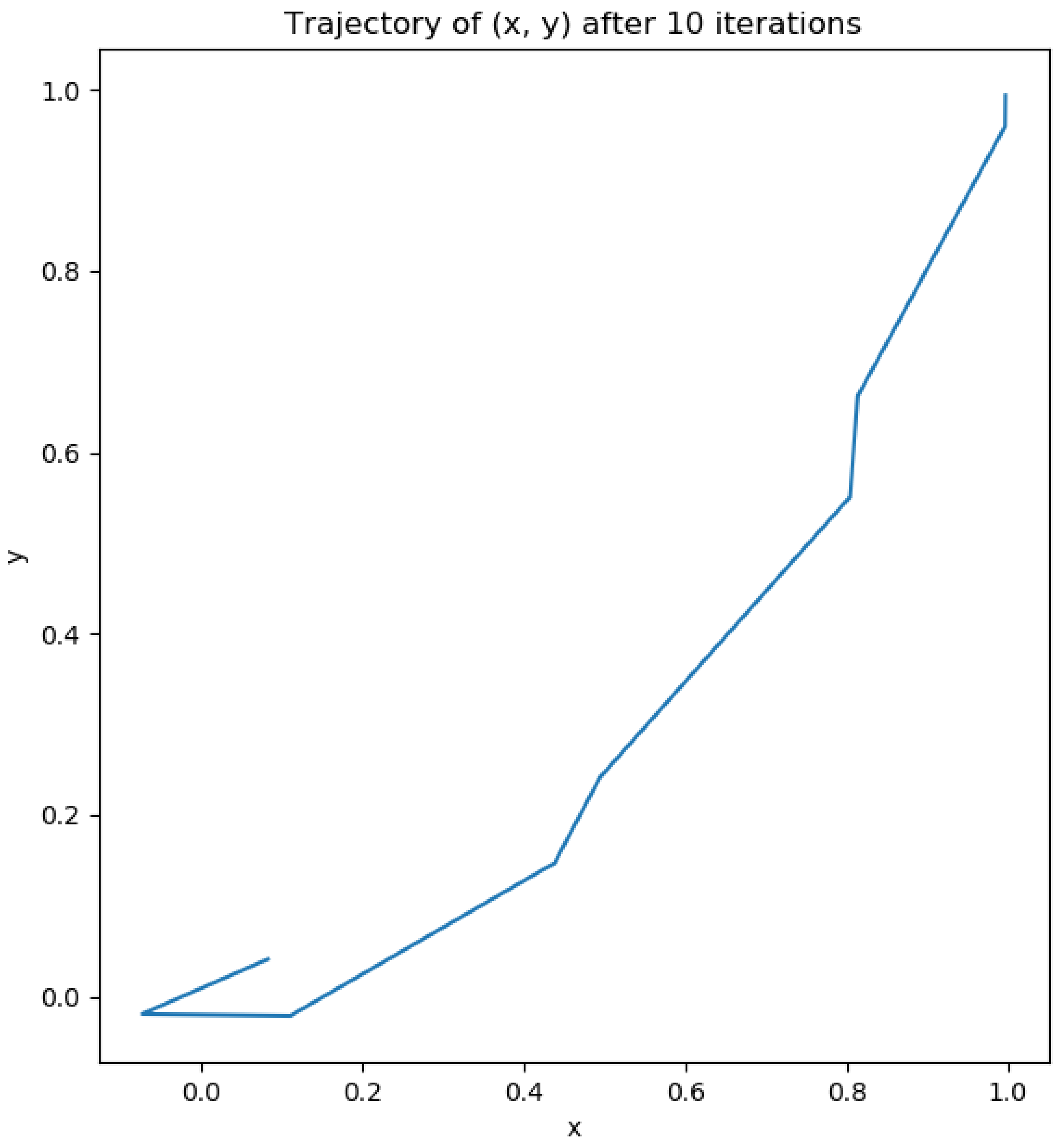


Figure 3 Trajectory of (x, y) using Newton's method

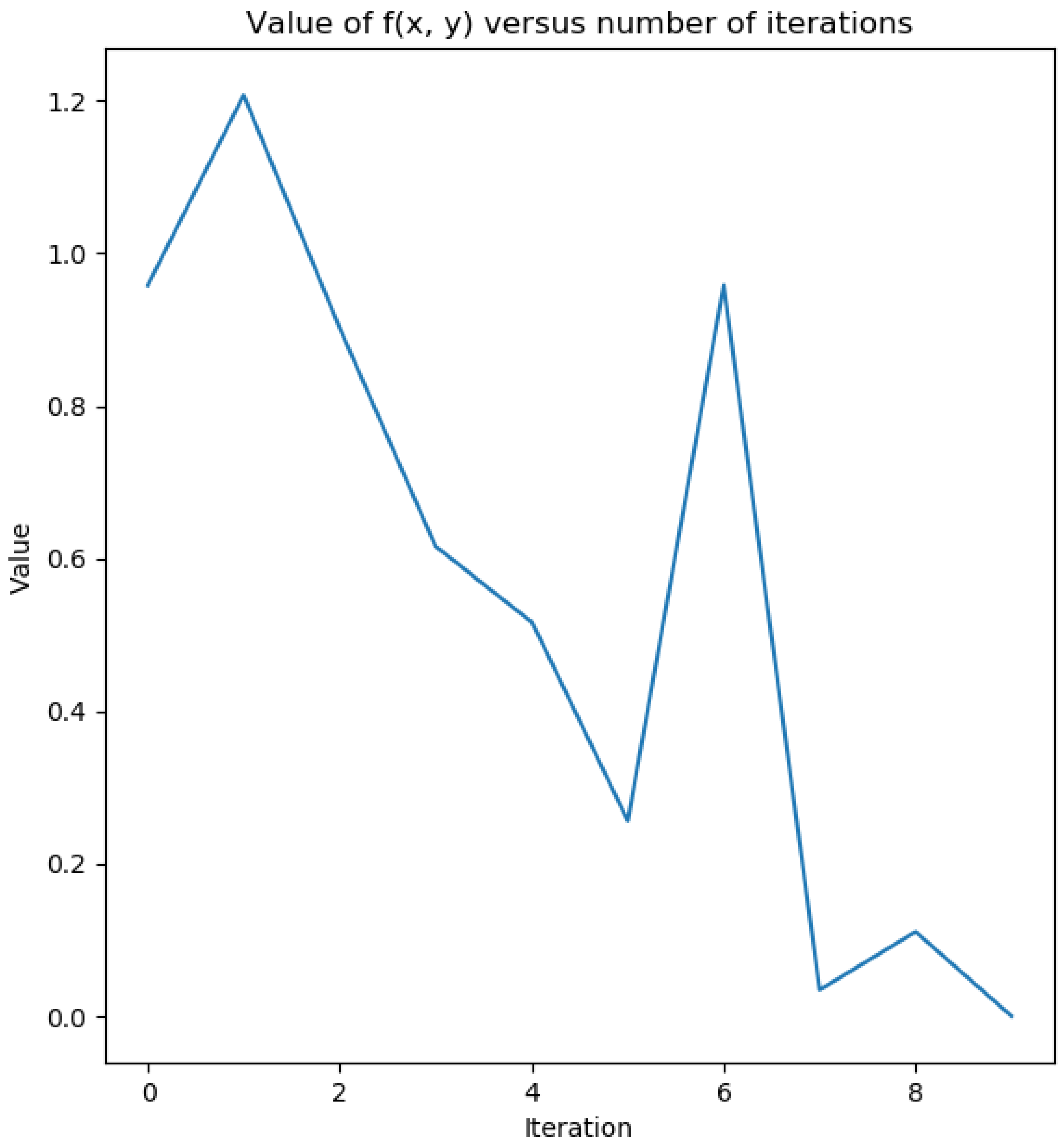


Figure 4 Value of f(x, y) versus number of iterations using Newton's method

And the final x value is 0.9970237075113713, final y value is 0.9940560733158862. Compared with gradient descent method, newton's method is much faster as it takes 10 iterations while gradient descent takes 17128 iterations. However, the computation cost on Hessian matrix maybe great if feature dimension is large.

Q2 Function Approximation

a). Train in sequential mode with BP algorithm

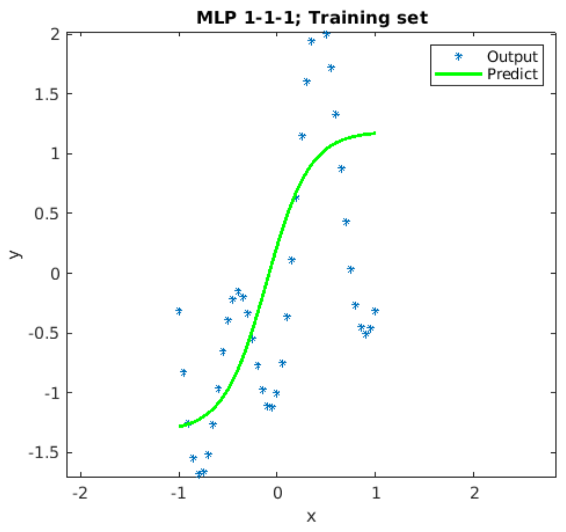
Source code:

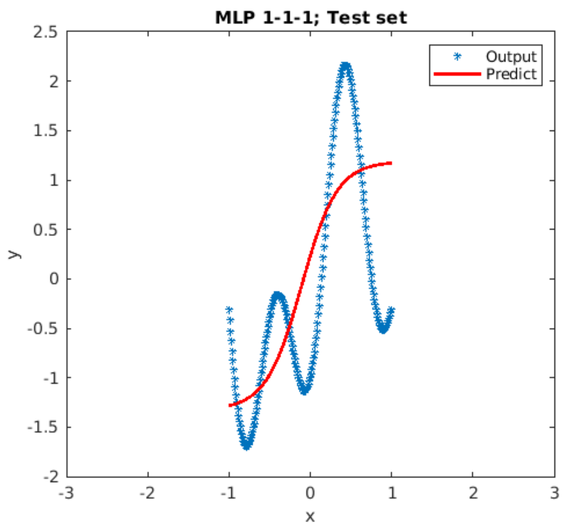
```
clc
clear all;

% training and testing data set
x_train_set = (-1:0.05:1);
y_train = 1.2 * sin(pi*x_train_set) - cos(2.4*pi*x_train_set);


x_test_set = (-1:0.01:1);
y_test = 1.2 * sin(pi*x_test_set) - cos(2.4*pi*x_test_set);


epochs = 100;


for n = [1:10,20,50,100]
[net, accu_train] = train_seq(n, x_train_set, y_train,length(x_train_set), epochs);


% get predict results for both train and test dats set
x_train_pred = sim(net,x_train_set);
x_test_pred = sim(net,x_test_set);


fig = figure(n);
%set(gcf,'PaperPositionMode','auto')
%set(fig, 'Position', [200 200 500 400])
set(gcf,'unit','normalized','position',[0.2,0.2,0.64,0.32]);
% show ploton training set
subplot(1,2,1);
p = plot(x_train_set,y_train,'*',x_train_set,x_train_pred,'g-');
p(1).MarkerSize = 4;
p(2).LineWidth = 2;
title(['MLP 1-',num2str(n),'-1; Training set']);
xlabel('x');
ylabel('y');
legend('Output','Predict')
xlim([-3 3]);


% show ploton test set
subplot(1,2,2);
p = plot(x_test_set,y_test,'*',x_test_set,x_test_pred,'r-');
   p(1).MarkerSize = 4;
   p(2).LineWidth = 2;
   title(['MLP 1-',num2str(n),'-1; Test set']);
   xlabel('x');
```

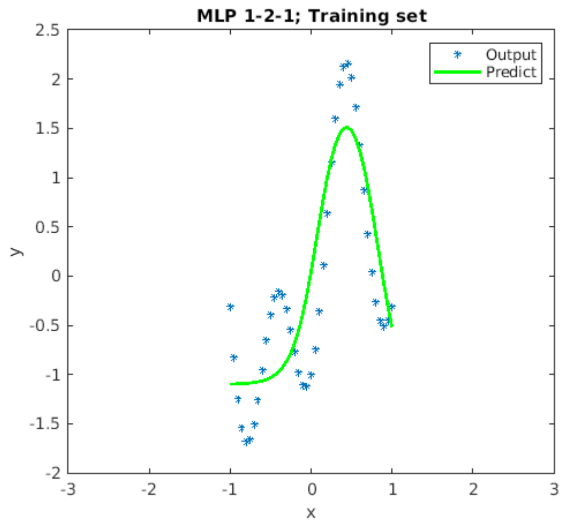
```
    ylabel('y');
    legend('Output','Predict')
    xlim([-3 3]);




    saveas(fig, ['q2a_n',num2str(n),'.png'])
    pred_x3 = net([-3, 3]);
    display(['n = ',num2str(n),' results for x = -3 and x = 3: ', num2str(pred_x3)]);
end


function[ net, accu_train ] = train_seq( n, inputs, labels, train_num, epochs )
%% Construct a 1-n-1 MLP and conduct sequential training.
%
% Args:
% n: int, number of neurons in the hidden layer of MLP.
% inputs: matrix of (input_dim, input_num), containing possibly preprocessed input data as input.
% labels: vector of (1, input_num), containing corresponding label of each input.
% train_num: int, number of training inputs.
% val_num: int, number of validation inputs.
% epochs: int, number of training epochs.


% Returns:
% net: object, containing trained network.
% accu_train: vector of (epochs, 1), containing the accuracy on training set of each eopch during
training
% accu_val: vector of (epochs, 1), containing the accuracy on validation set of each eopch during
training.


%% 1. Change the input to cell array form for sequential training
inputs_c = num2cell(inputs, 1);
labels_c = num2cell(labels, 1);


%% 2. Construct and configure the MLP
net = fitnet(n);
net.divideFcn = 'dividetrain'; % input for training only
net.performParam.regularization = 0.25; % regularization strength
net.trainFcn = 'traingdx'; % 'trainrp' 'traingdx'
net.trainParam.epochs = epochs;
accu_train = zeros(epochs,1); % record accuracy on training set of each epoch


%% 3. Train the network in sequential mode
for i = 1 : epochs
```

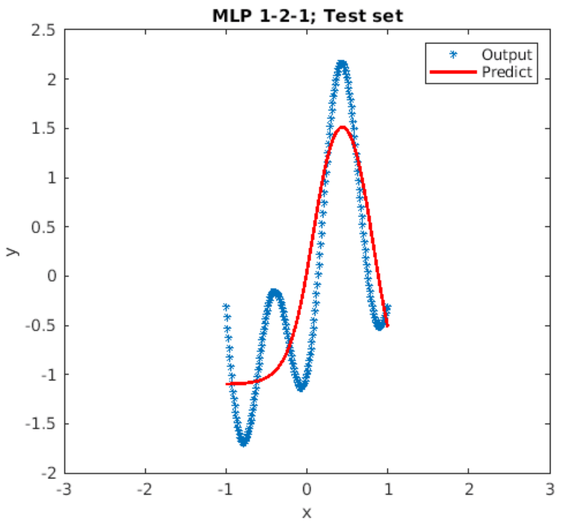
```
    %display(['Epoch: ', num2str(i)])
    idx = randperm(train_num); % shuffle the input
    net = adapt(net, inputs_c(:,idx), labels_c(:,idx));
    pred_train = round(net(inputs(:,1:train_num))); % predictions on training set
    accu_train(i) = 1 - mean(abs(pred_train-labels(1:train_num)));

end


end
```

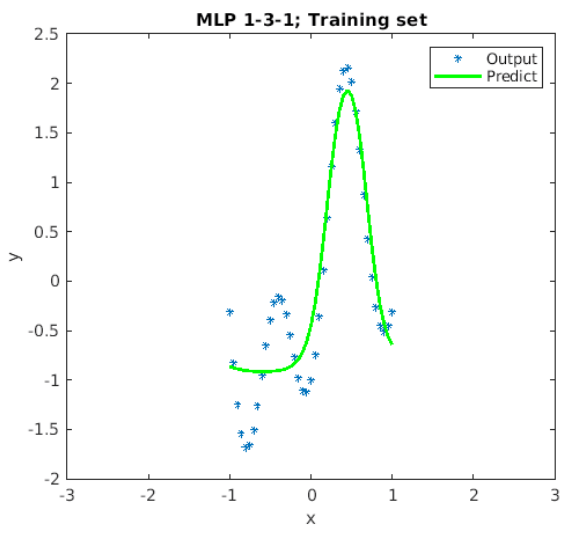
Plots and Results:
The following figures show different implementations of MLP in 1-n-1 models, which are trained using sequential mode of gradient-descent learning, to predict the results of function y = 1.2sin(πx) - cos(2.4πx).

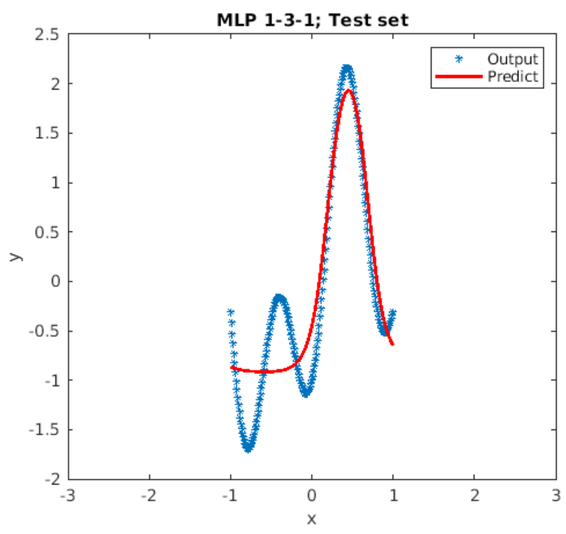


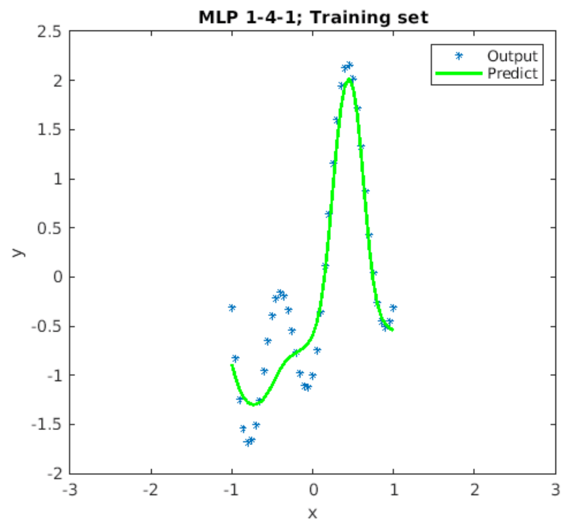


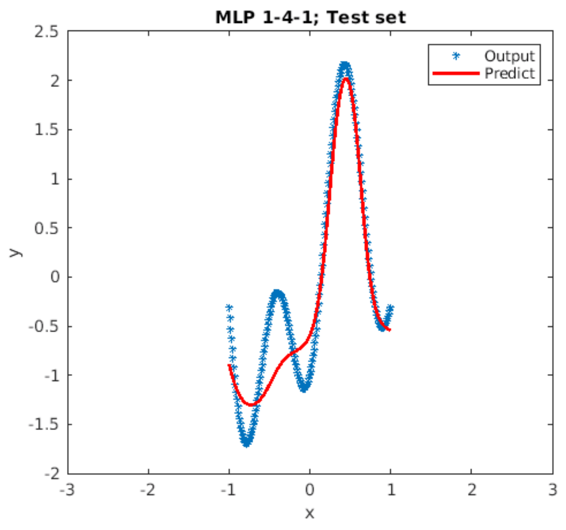


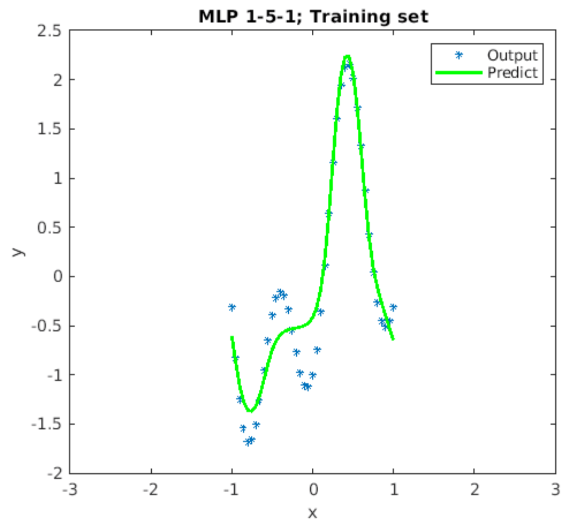


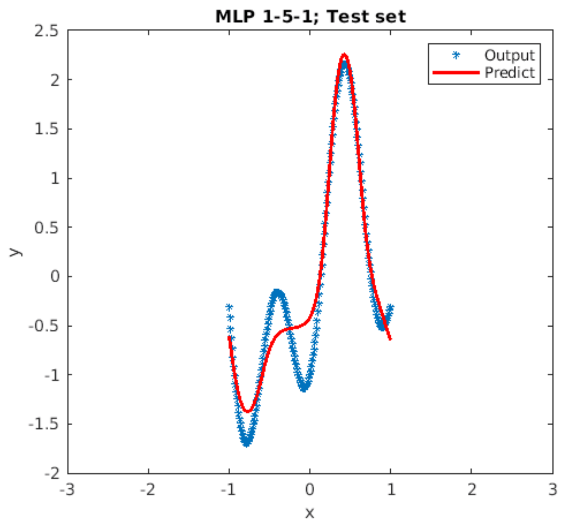

MLP 1-3-1; Training set
Output
Predict
x
y

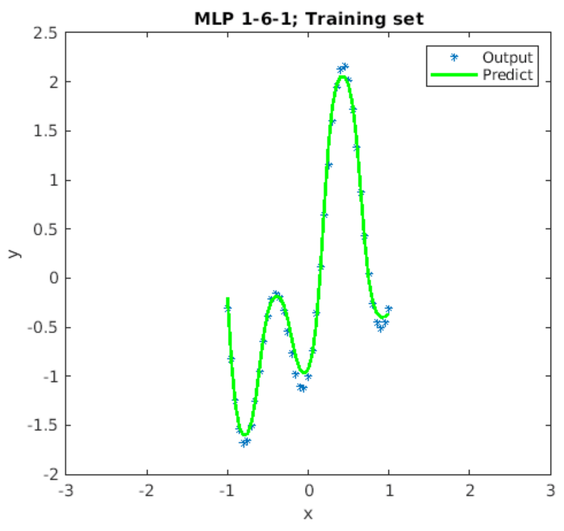

MLP 1-3-1; Test set
Output
Predict
x
y

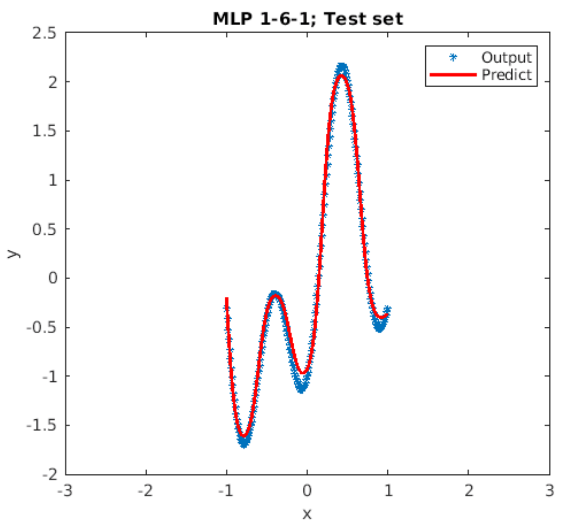

MLP 1-4-1; Training set
Output
Predict
x
y

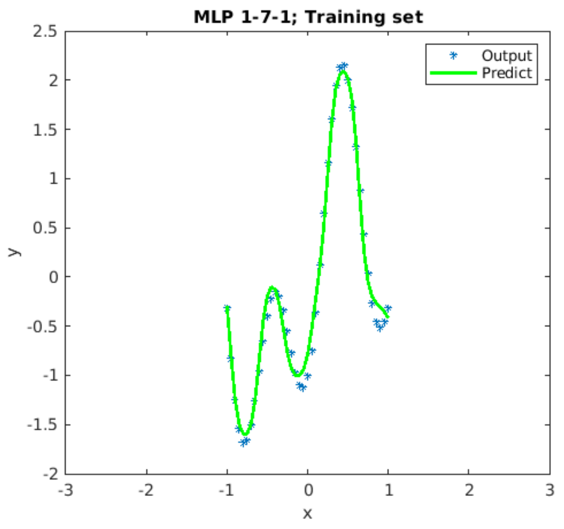

MLP 1-4-1; Test set
Output
Predict
x
y

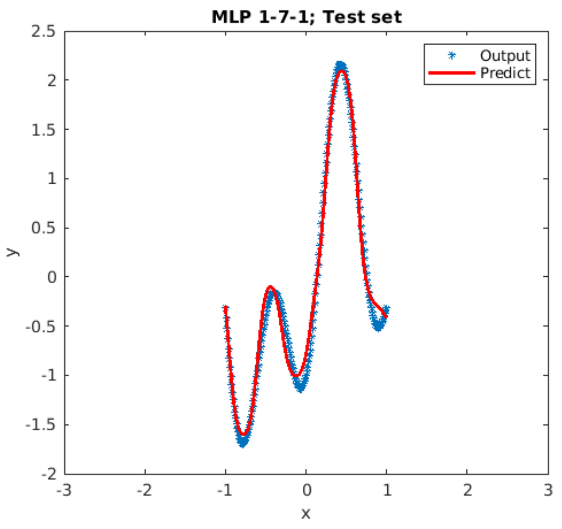

MLP 1-5-1; Training set
Output
Predict
x
y

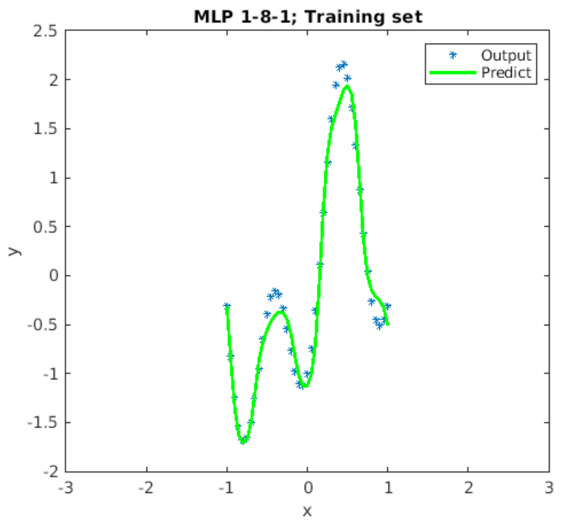

MLP 1-5-1; Test set
Output
Predict
x
y

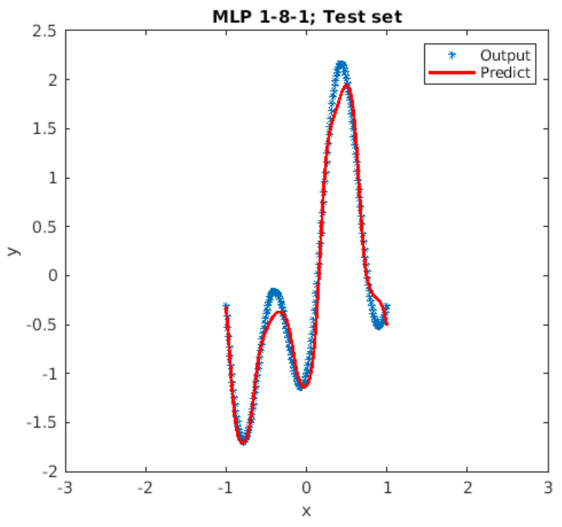

MLP 1-6-1; Training set
Output
Predict
x
y

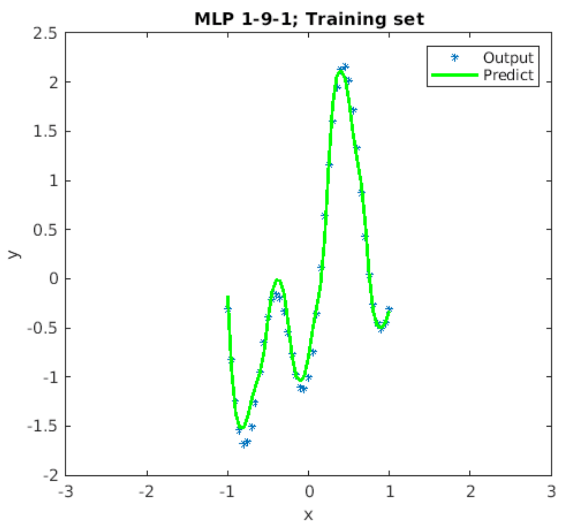

MLP 1-6-1; Test set
Output
Predict
x
y

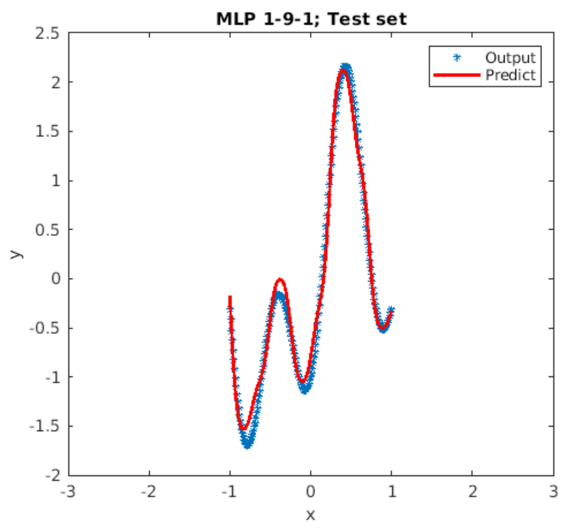

MLP 1-7-1; Training set
Output
Predict
x
y

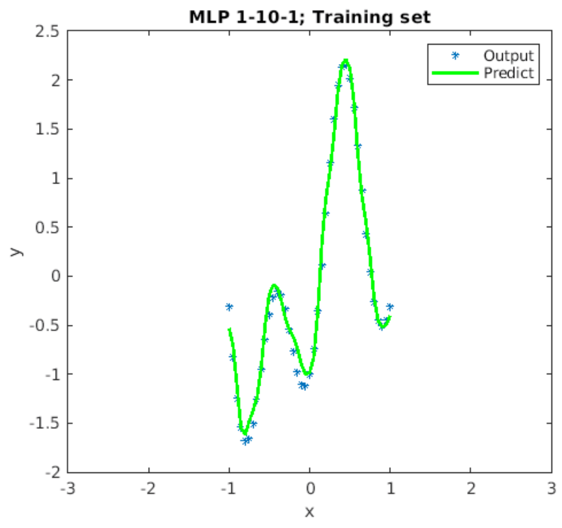

MLP 1-7-1; Test set
Output
Predict
x
y

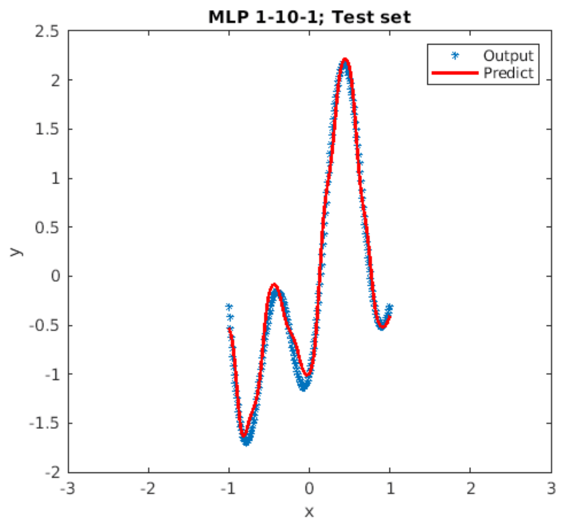

MLP 1-8-1; Training set
Output
Predict
x
y

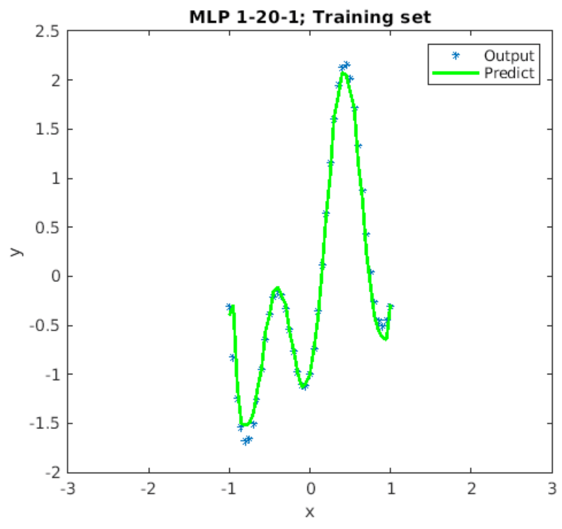

MLP 1-8-1; Test set
Output
Predict
x
y

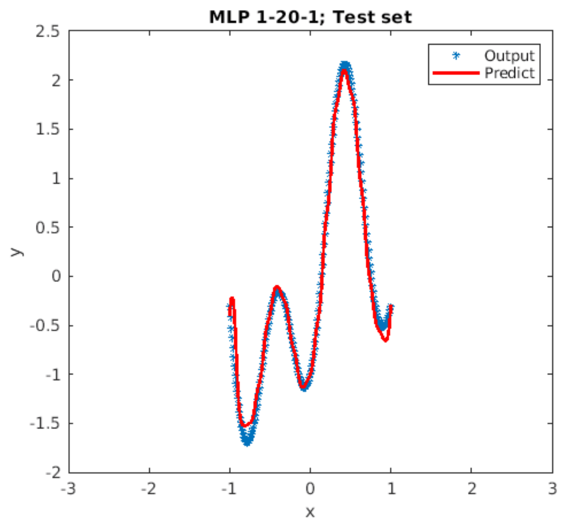

MLP 1-9-1; Training set
Output
Predict
x
y

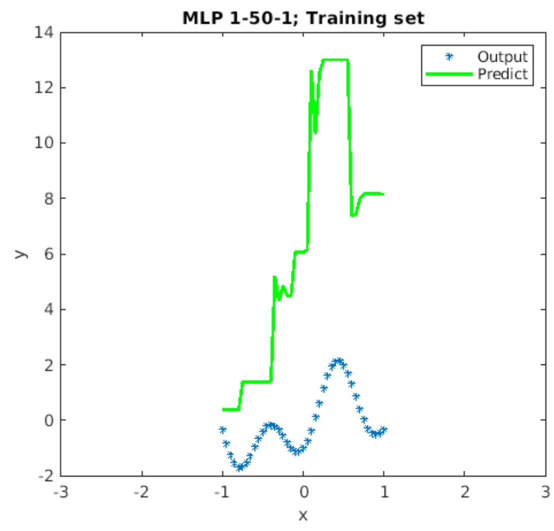

MLP 1-9-1; Test set
Output
Predict
x
y

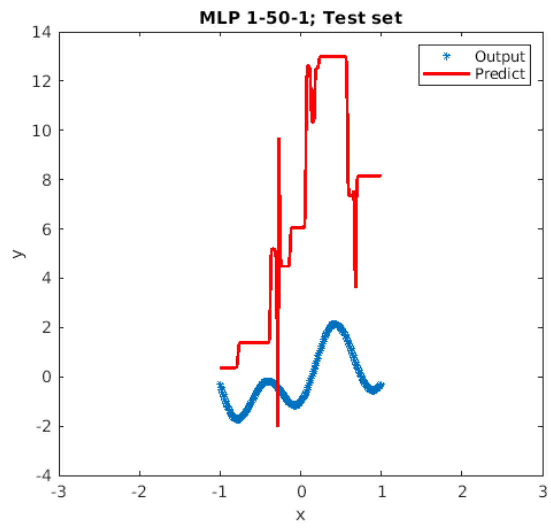

MLP 1-10-1; Training set
Output
Predict
x
y

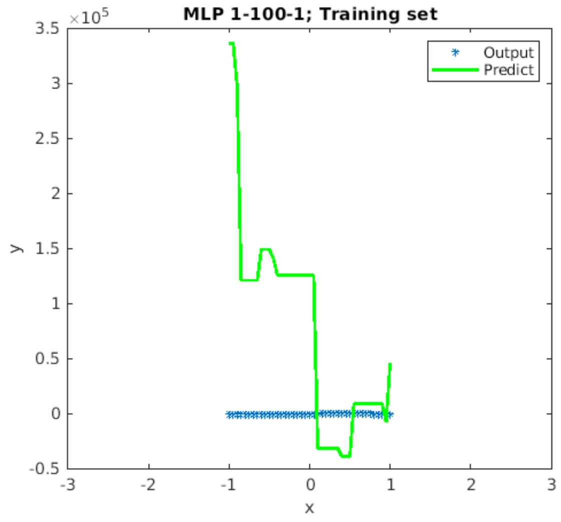

MLP 1-10-1; Test set
Output
Predict
x
y

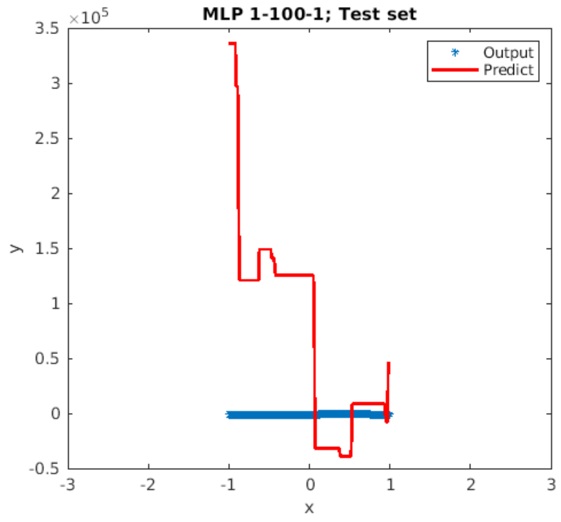

MLP 1-20-1; Training set
Output
Predict
x
y


MLP 1-20-1; Test set
Output
Predict
x
y









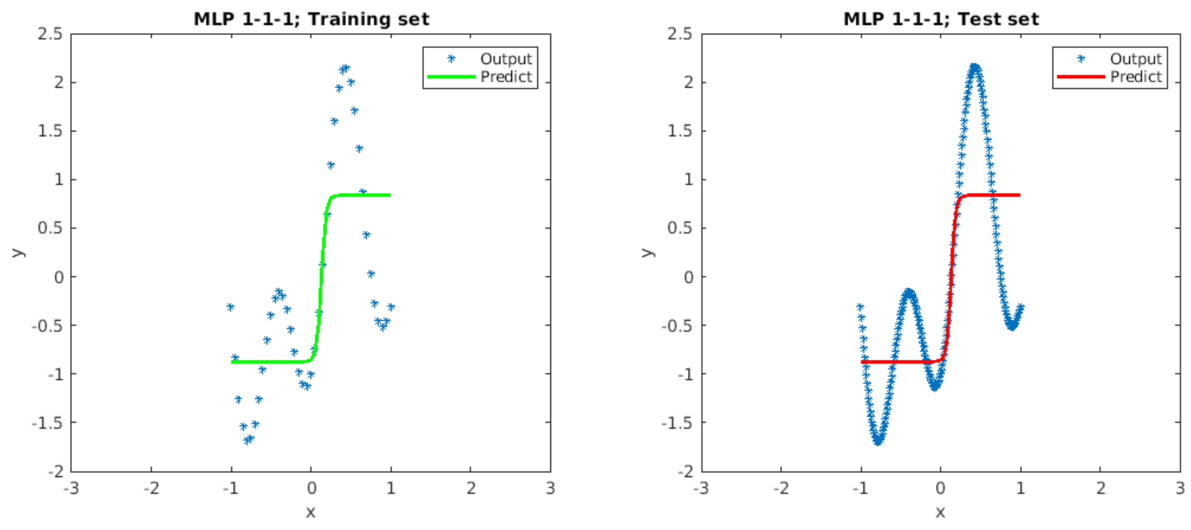
When x = 3 and x = -3, the predicted outputs:
n = 1 results for x = -3 and x = 3: -1.3235 1.186
n = 2 results for x = -3 and x = 3: -1.1011 -1.3466
n = 3 results for x = -3 and x = 3: -0.75982 -0.79311
n = 4 results for x = -3 and x = 3: -0.11309 -0.56454
n = 5 results for x = -3 and x = 3: 0.070917 -0.87691
n = 6 results for x = -3 and x = 3: 2.0719 -0.28276
n = 7 results for x = -3 and x = 3: 0.39344 -0.5636
n = 8 results for x = -3 and x = 3: 0.28598 -2.191
n = 9 results for x = -3 and x = 3: 0.97012 -0.1651
n = 10 results for x = -3 and x = 3: -0.46507 -0.15214
n = 20 results for x = -3 and x = 3: -2.2927 0.54823
n = 50 results for x = -3 and x = 3: 35.6748 13.9149
n = 100 results for x = -3 and x = 3: 197597.2935 -32282.22268

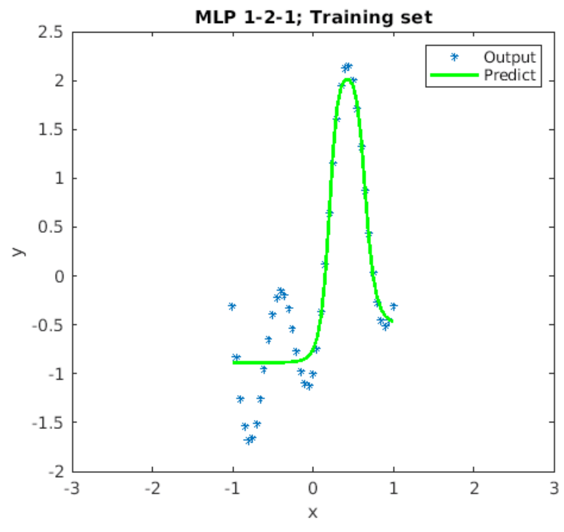
From the fitting plots, we can see that when n < 10, the outputs of test sample are underfit, which means the predicted outputs from such model does not fully match the desired results of function. The outputs of test sample show good fit when n = 10. When n > 10, overfitting appears. And when n = 50 or n = 100, the model leads make error in prediction. Therefore, the minimum number of neurons for this MLP model is 10.

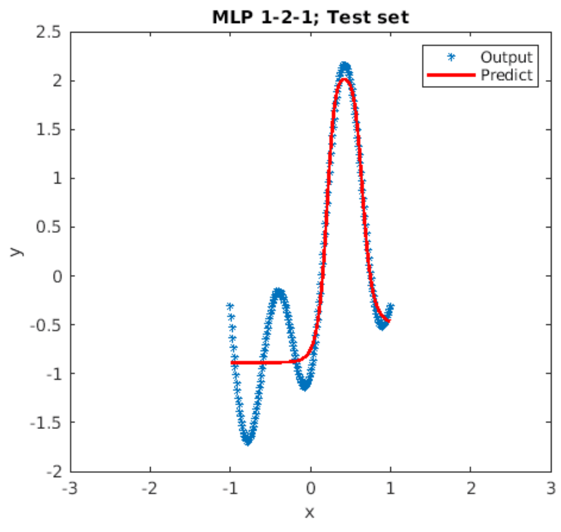
For all the cases of n in such model, the predicted outputs are poor, compared with the desired outputs 0.809 when x = 3 or x = -3. Thus, the MLP cannot make reasonable predictions outside of the domain of the input limited by the training set as there is no training samples for the model to learn.

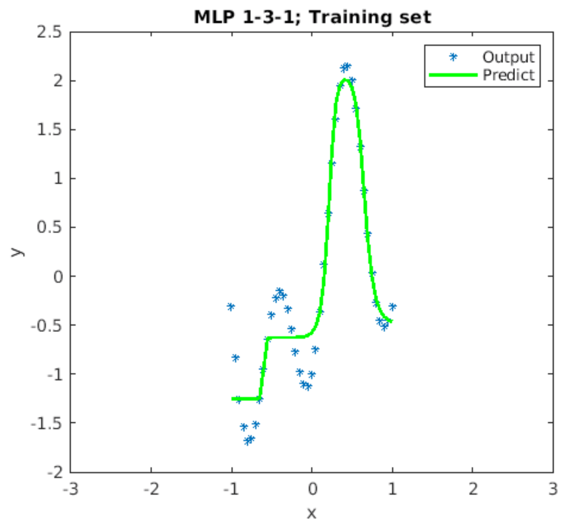
b). Train in batch mode with trainlm algorithm
Source code:

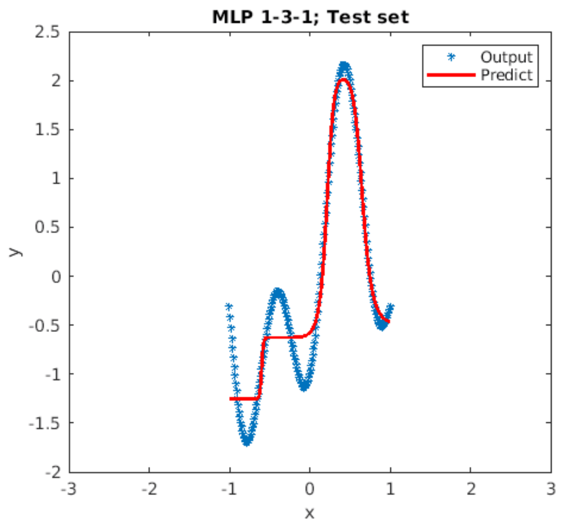
```
clc
clear all;




% training and testing data set
x_train_set = (-1:0.05:1);
y_train = 1.2 * sin(pi*x_train_set) - cos(2.4*pi*x_train_set);


x_test_set = (-1:0.01:1);
y_test = 1.2 * sin(pi*x_test_set) - cos(2.4*pi*x_test_set);


train_func = 'trainlm';
epochs = 50;


for n = [1:10,20,50,100]
    % build model
    net = patternnet(n);
    net.divideFcn = 'dividetrain';
    net.performFcn = 'mse';
    net.trainFcn = train_func;


    % train model
    net = train(net, x_train_set, y_train);
    % get predict results for both train and test dats set
    x_train_pred = net(x_train_set);
    x_test_pred = net(x_test_set);


    fig = figure(n);
    %set(gcf,'PaperPositionMode','auto')
    %set(fig, 'Position', [200 200 500 400])
    set(gcf,'unit','normalized','position',[0.2,0.2,0.64,0.32]);
    % show ploton training set
    subplot(1,2,1);
    p = plot(x_train_set,y_train,'*',x_train_set,x_train_pred,'g-');
    p(1).MarkerSize = 4;
    p(2).LineWidth = 2;
    title(['MLP 1-',num2str(n),'-1; Training set']);
    xlabel('x');
    ylabel('y');
    legend('Output','Predict')
    xlim([-3 3]);


    % show ploton test set
    subplot(1,2,2);
    p = plot(x_test_set,y_test,'*',x_test_set,x_test_pred,'r-');
    p(1).MarkerSize = 4;
    p(2).LineWidth = 2;
    title(['MLP 1-',num2str(n),'-1; Test set']);
    xlabel('x');
    ylabel('y');
    legend('Output','Predict')
    xlim([-3 3]);


    saveas(fig, ['q2a_n',num2str(n),'.png'])
    pred_x3 = net([-3, 3]);
    display(['n = ',num2str(n),' results for x = -3 and x = 3: ', num2str(pred_x3)]);
end
```

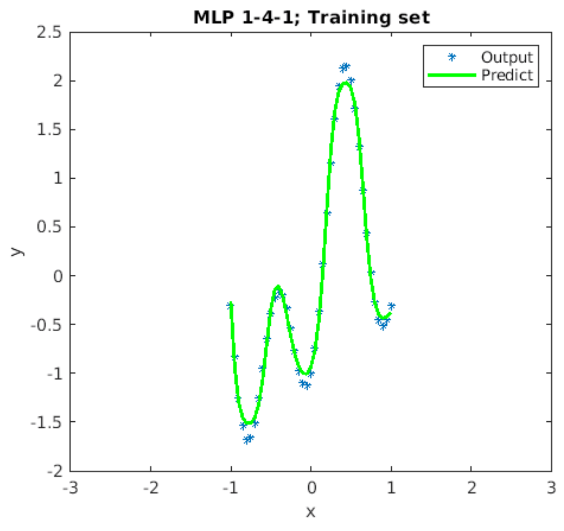


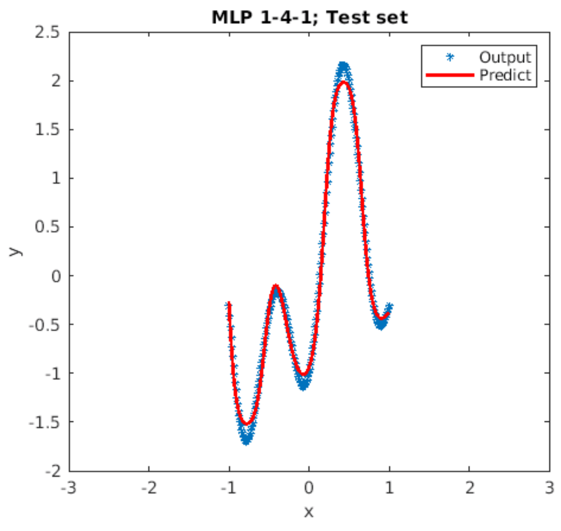

MLP 1-2-1; Training set
Output
Predict
x
y

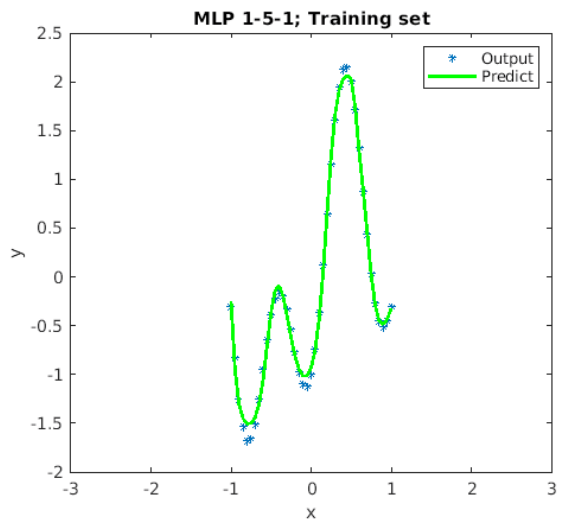

MLP 1-2-1; Test set
Output
Predict
x
y

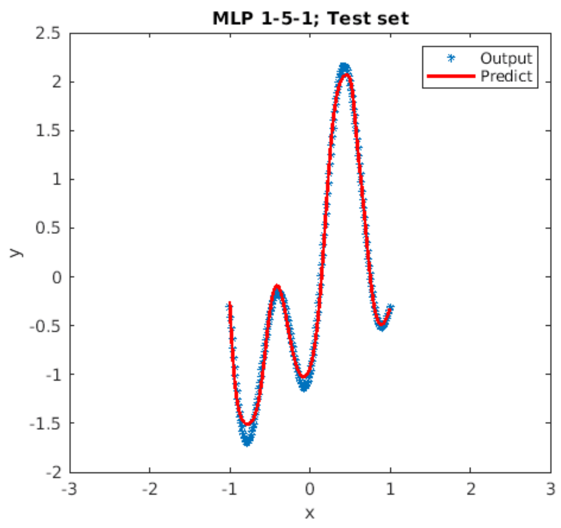

MLP 1-3-1; Training set
Output
Predict
x
y

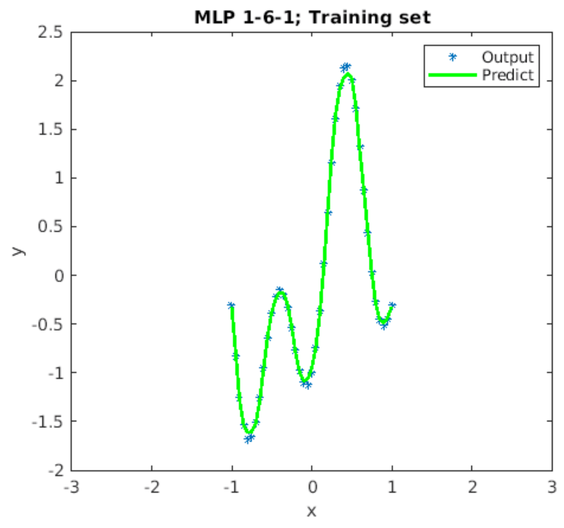

MLP 1-3-1; Test set
Output
Predict
x
y

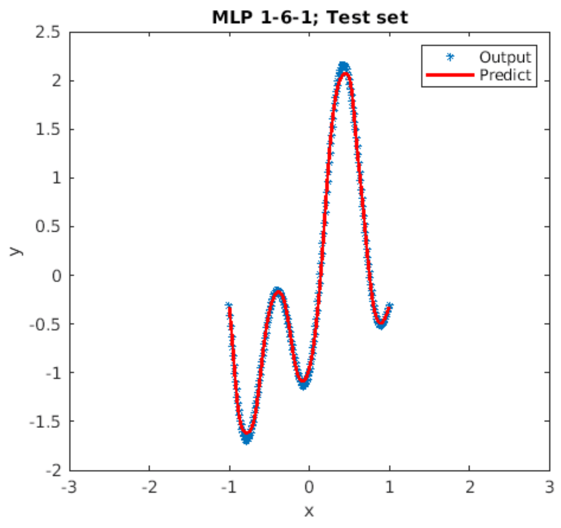

MLP 1-4-1; Training set
Output
Predict
x
y

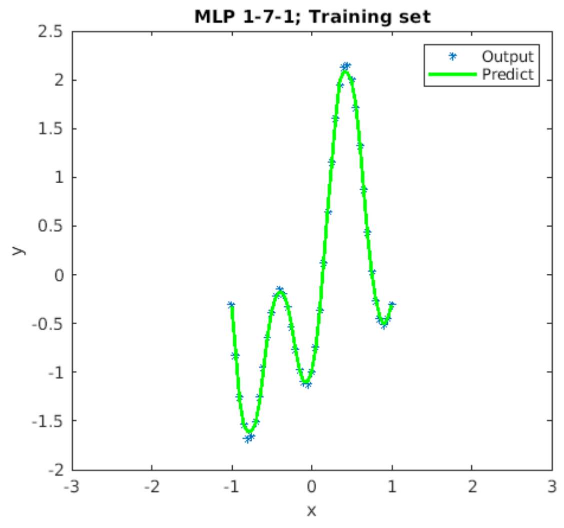

MLP 1-4-1; Test set
Output
Predict
x
y

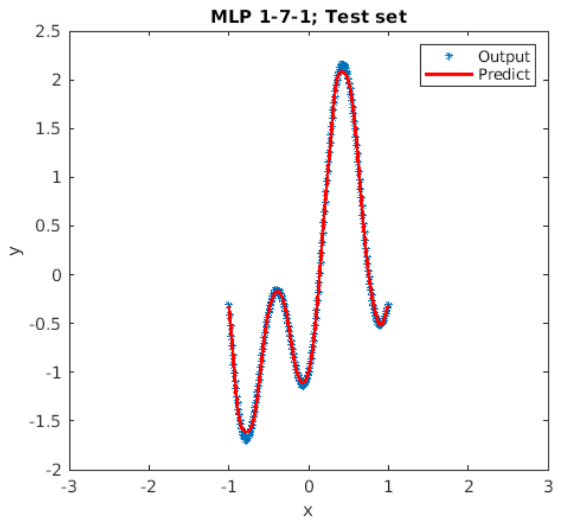

MLP 1-5-1; Training set
Output
Predict
y
x

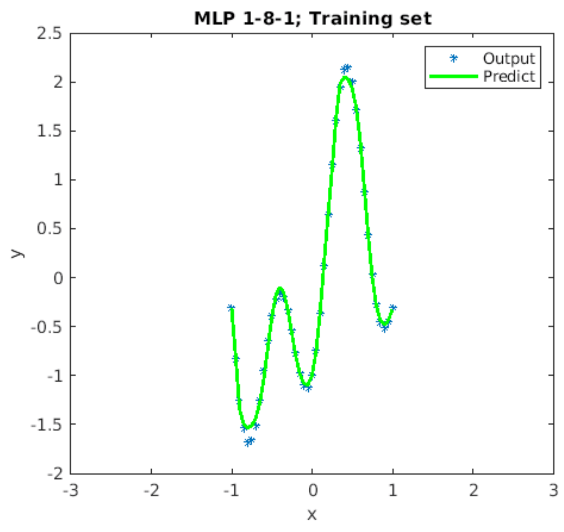

MLP 1-5-1; Test set
Output
Predict
y
x

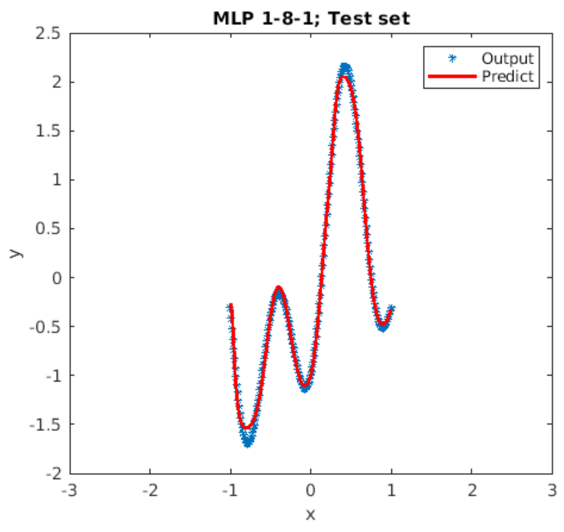

MLP 1-6-1; Training set
Output
Predict
y
x

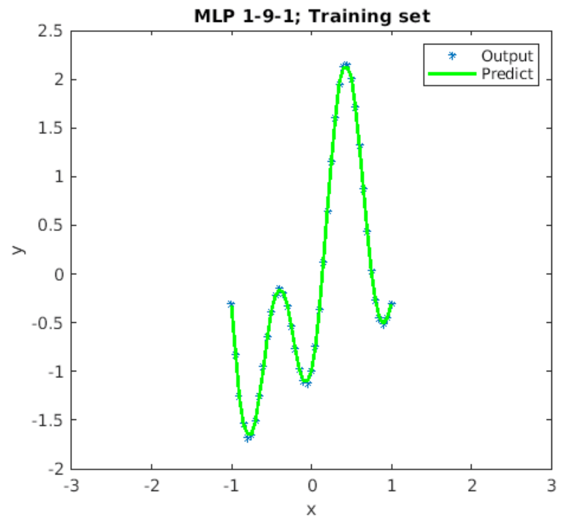

MLP 1-6-1; Test set
Output
Predict
y
x

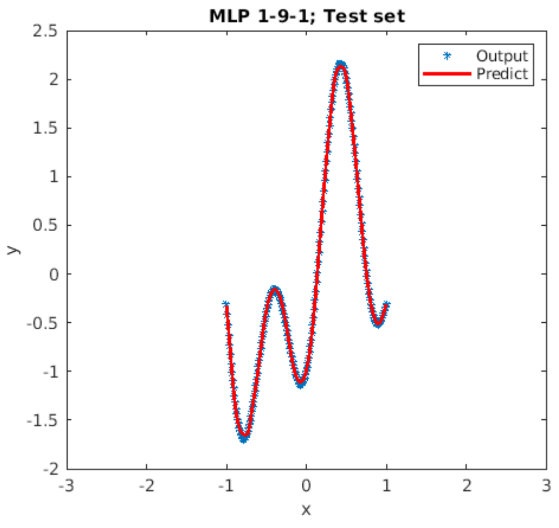

MLP 1-7-1; Training set
Output
Predict
y
x

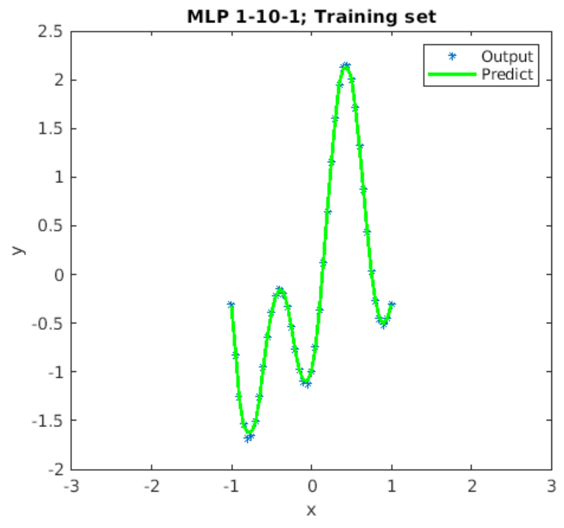

MLP 1-7-1; Test set
Output
Predict
y
x

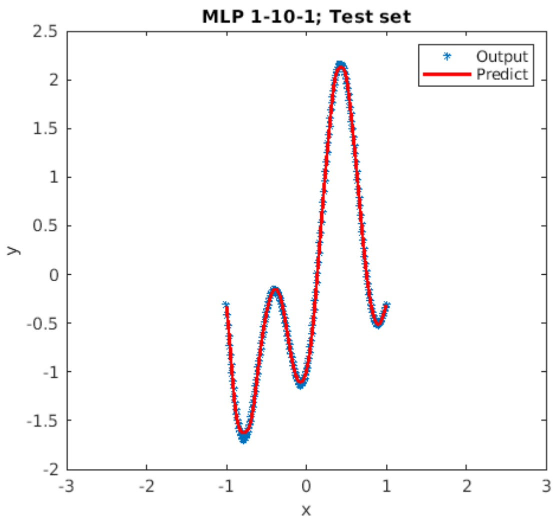

MLP 1-8-1; Training set
Output
Predict
y
x

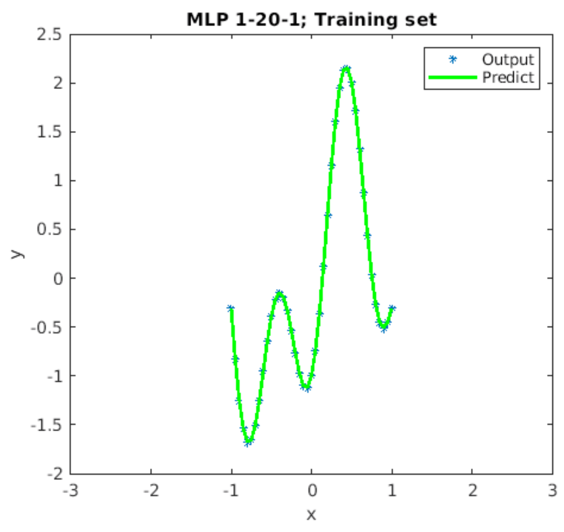

MLP 1-8-1; Test set
Output
Predict
y
x

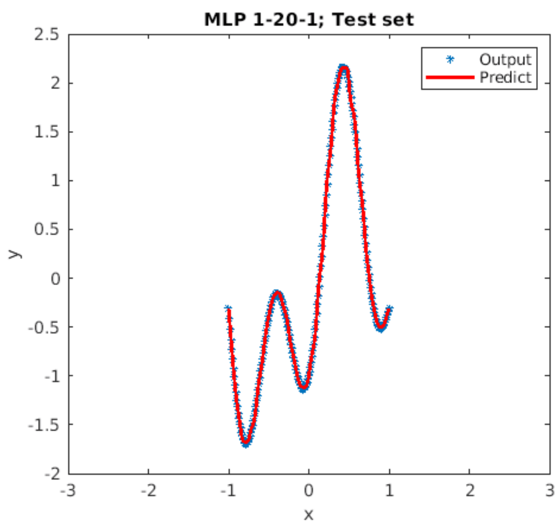

MLP 1-9-1; Training set
Output
Predict
y
x

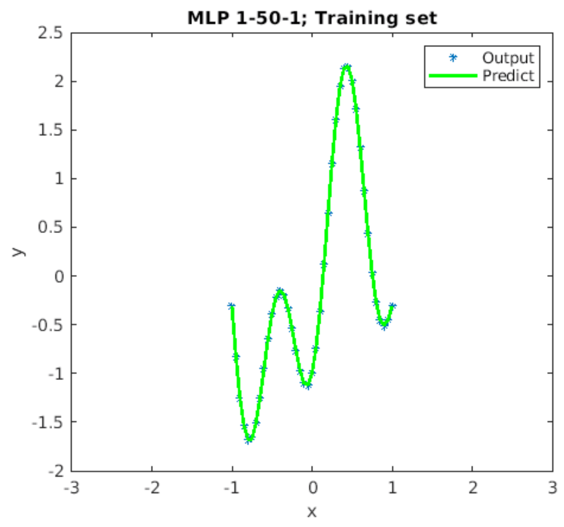

MLP 1-9-1; Test set
Output
Predict
y
x

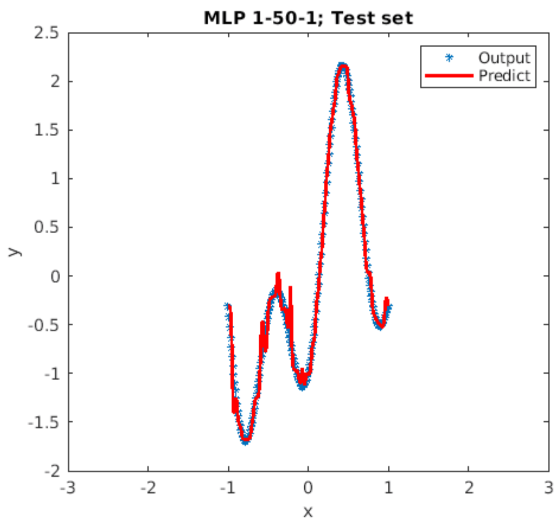

MLP 1-10-1; Training set
Output
Predict
y
x

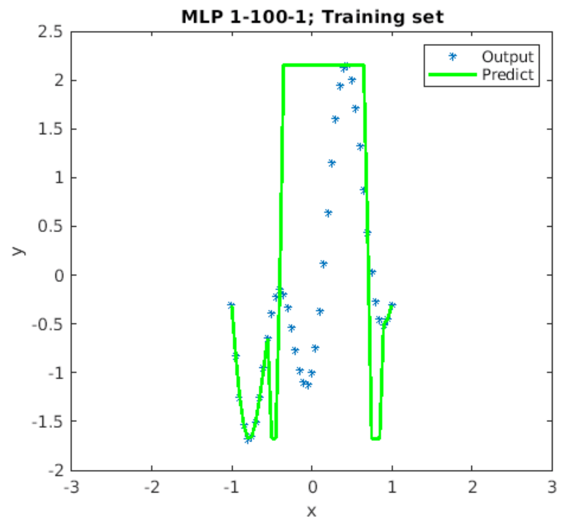

MLP 1-10-1; Test set
Output
Predict
y
x

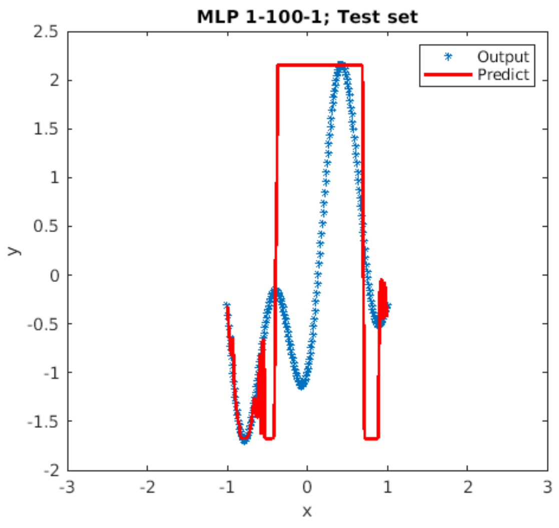

MLP 1-20-1; Training set
Output
Predict
y
x
2.5
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-3
-2
-1
0
1
2
3


MLP 1-20-1; Test set
Output
Predict
y
x
2.5
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-3
-2
-1
0
1
2
3


MLP 1-50-1; Training set
Output
Predict
y
x
2.5
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-3
-2
-1
0
1
2
3


MLP 1-50-1; Test set
Output
Predict
y
x
2.5
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-3
-2
-1
0
1
2
3


MLP 1-100-1; Training set
Output
Predict
y
x
2.5
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-3
-2
-1
0
1
2
3

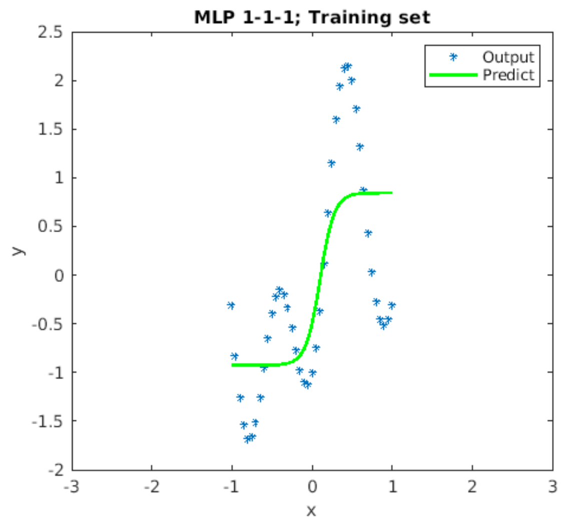

MLP 1-100-1; Test set
Output
Predict
y
x
2.5
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-3
-2
-1
0
1
2
3

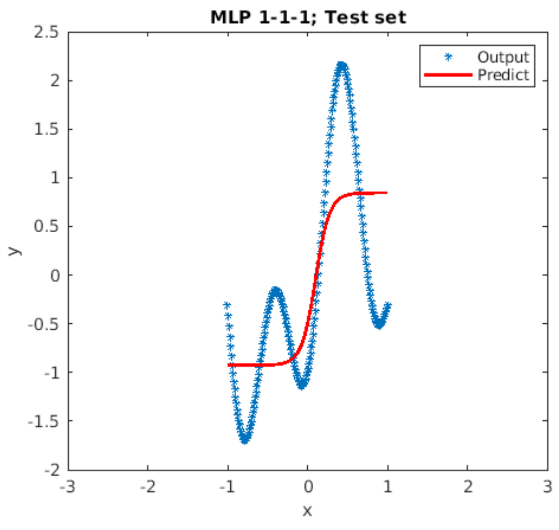
```
n = 1 results for x = -3 and x = 3: -0.87207      0.8359
n = 2 results for x = -3 and x = 3: -0.88457    -0.48873
n = 3 results for x = -3 and x = 3: -1.2522    -0.48608
n = 4 results for x = -3 and x = 3: 2.1457    0.050046
n = 5 results for x = -3 and x = 3: 2.1531      1.0656
n = 6 results for x = -3 and x = 3: 0.23495     0.90692
n = 7 results for x = -3 and x = 3: 0.14997    -0.38968
n = 8 results for x = -3 and x = 3: 2.1538     -1.6739
n = 9 results for x = -3 and x = 3: 0.58264    -0.18048
n = 10 results for x = -3 and x = 3: -1.6254    -0.17865
n = 20 results for x = -3 and x = 3: 0.76242    -0.24448
n = 50 results for x = -3 and x = 3: -0.75879    -0.41197
n = 100 results for x = -3 and x = 3: -0.24459    -0.22177
```

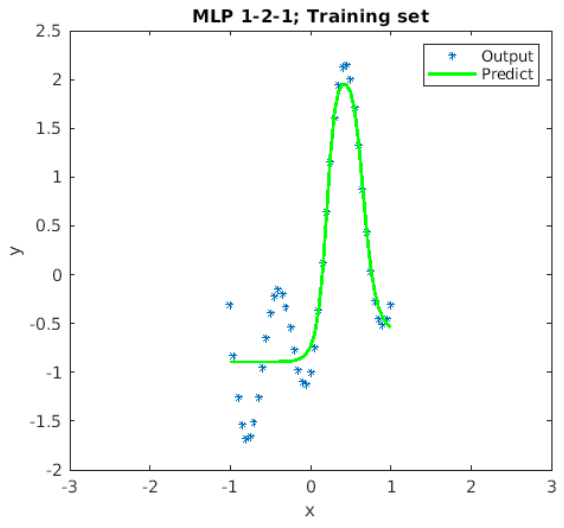
From the fitting plots, we can see that when n < 9, the outputs of test sample are underfit, however, as n increases from 1 to 8, the performance of fit improves as well. When n reaches 9 ,10 and 20, the model has a very good fit. The predicted outputs are matched nicely with desired outputs. When n reaches 50, overfitting occurred. However, as n reaches 100, the plot shows a severe overfitting on the model. Most of predicted outputs are not matched with the desired outputs. Therefore, the minimum number of neurons for this MLP model is 9.
For all the cases of n in such model, the predicted outputs are poor, compared with the desired outputs 0.809 when x = 3 or x = -3. Thus, the MLP cannot make reasonable predictions outside of the domain of the input limited by the training set as there is no training samples for the model to learn.

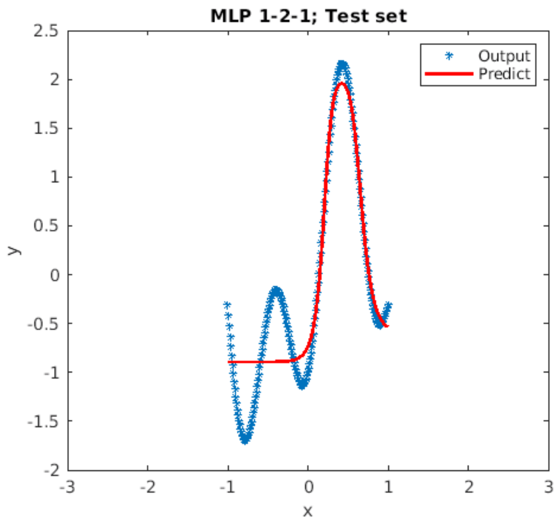
c). Train in batch mode with trainbr algorithm
Source code:
Change parameter value in Q2b source code:

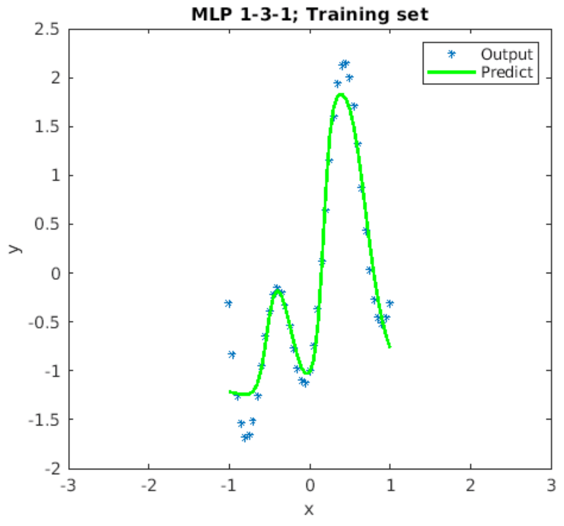
```
% parameters
train_func = 'trainbr';
```

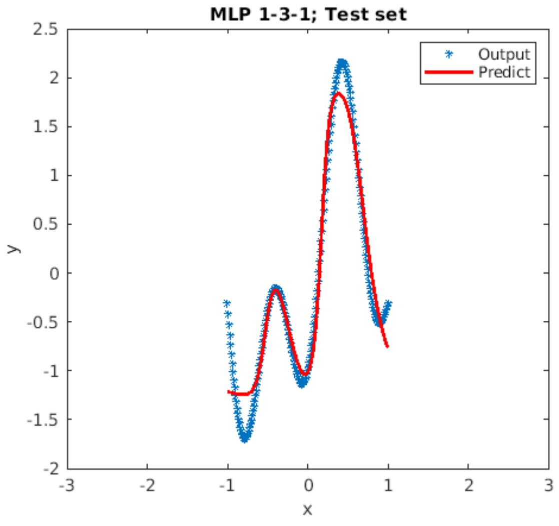


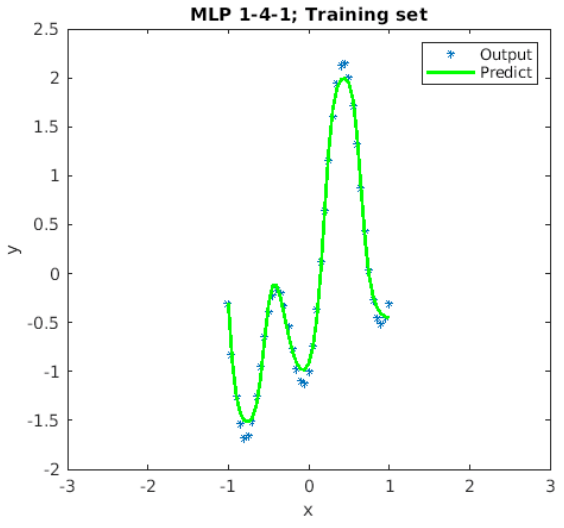


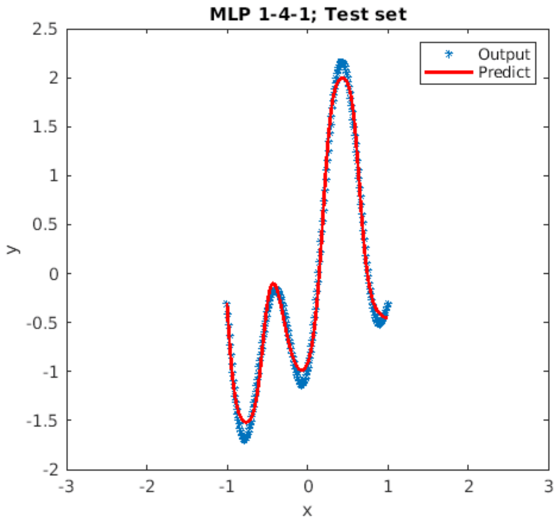

MLP 1-2-1; Training set
Output
Predict
x
y

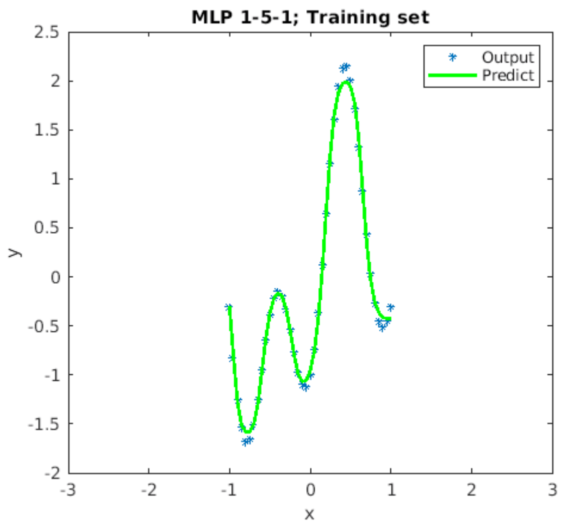

MLP 1-2-1; Test set
Output
Predict
x
y

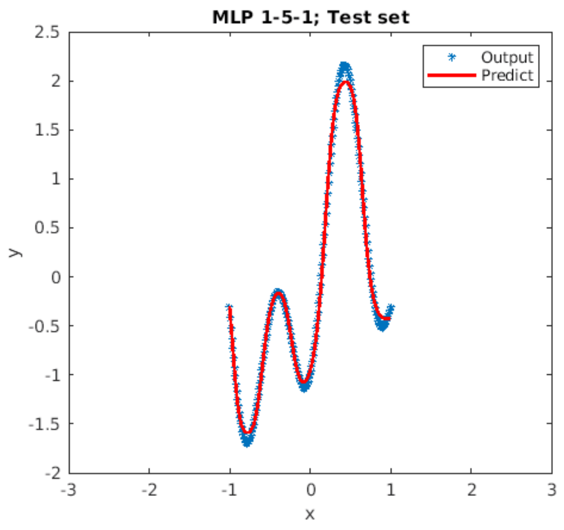

MLP 1-3-1; Training set
Output
Predict
x
y

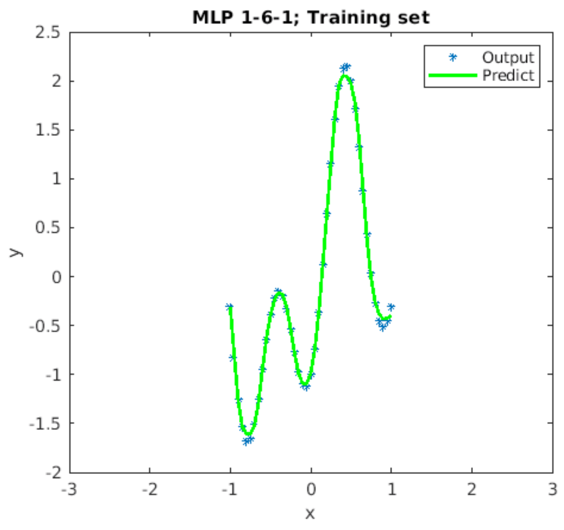

MLP 1-3-1; Test set
Output
Predict
x
y

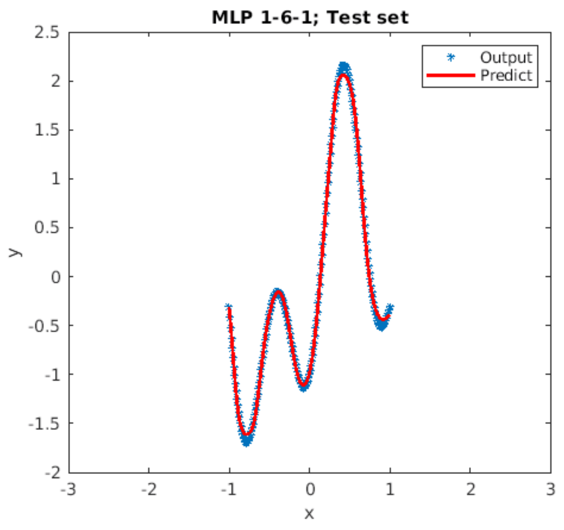

MLP 1-4-1; Training set
Output
Predict
x
y

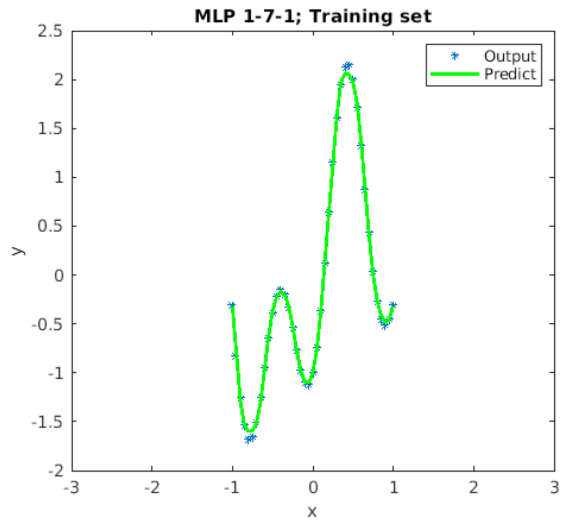

MLP 1-4-1; Test set
Output
Predict
x
y

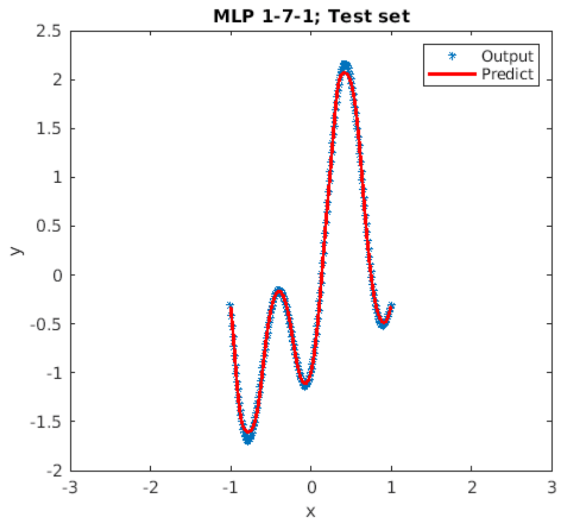

MLP 1-5-1; Training set
Output
Predict
x
y

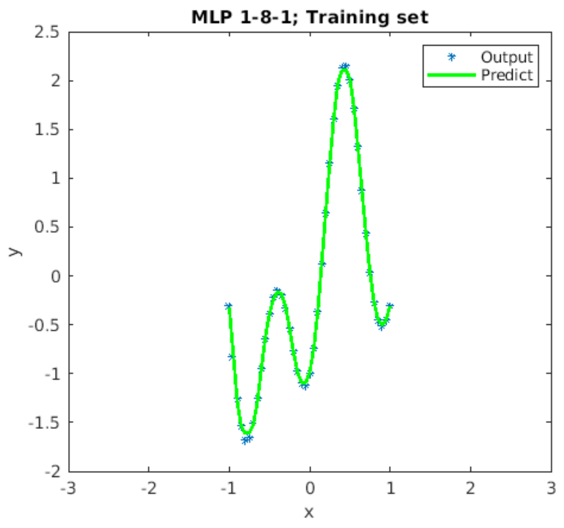

MLP 1-5-1; Test set
Output
Predict
x
y

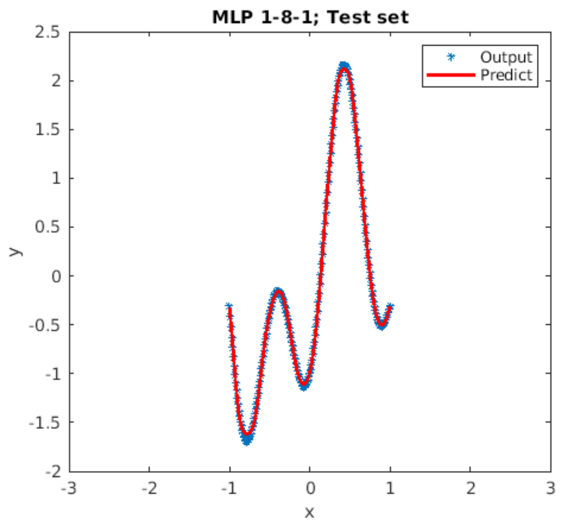

MLP 1-6-1; Training set
Output
Predict
x
y

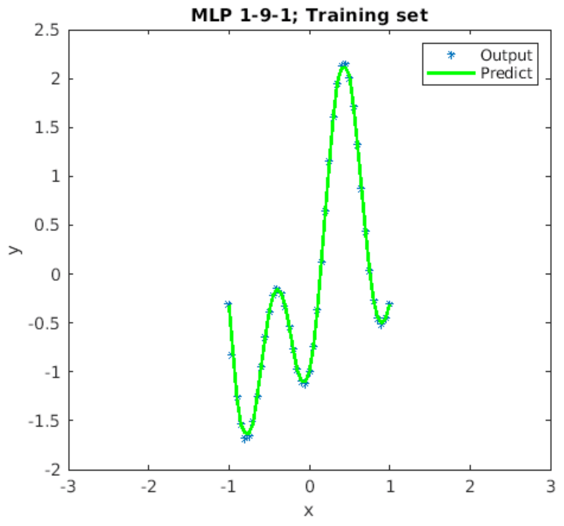

MLP 1-6-1; Test set
Output
Predict
x
y

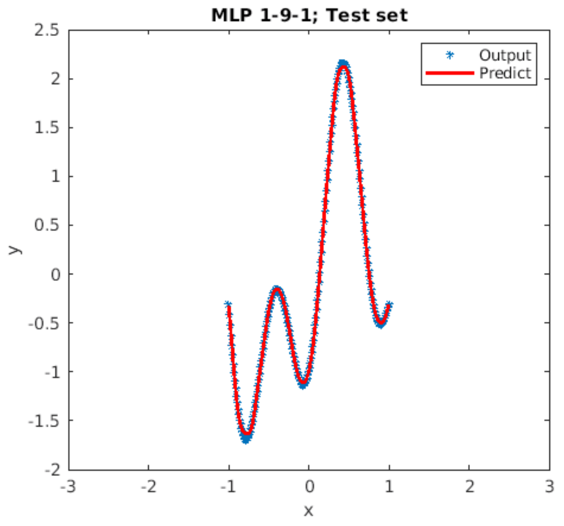

MLP 1-7-1; Training set
Output
Predict
x
y

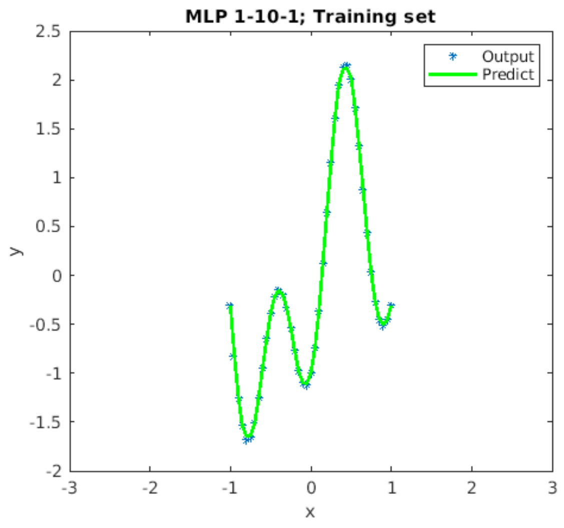

MLP 1-7-1; Test set
Output
Predict
x
y

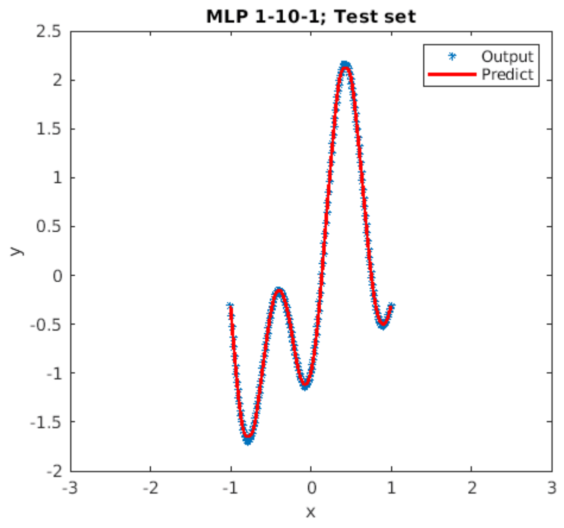

MLP 1-8-1; Training set
Output
Predict
x
y

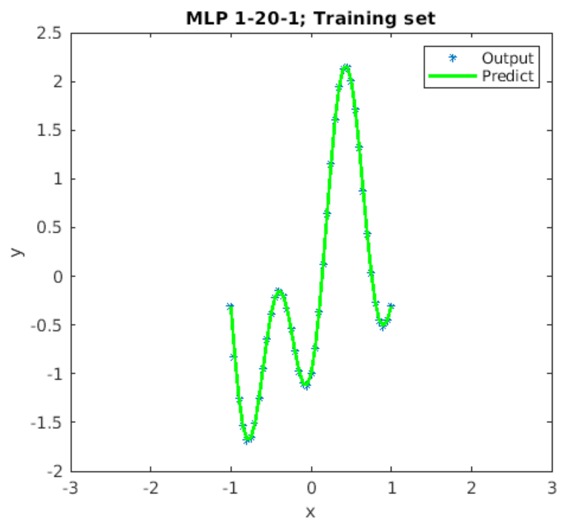

MLP 1-8-1; Test set
Output
Predict
x
y

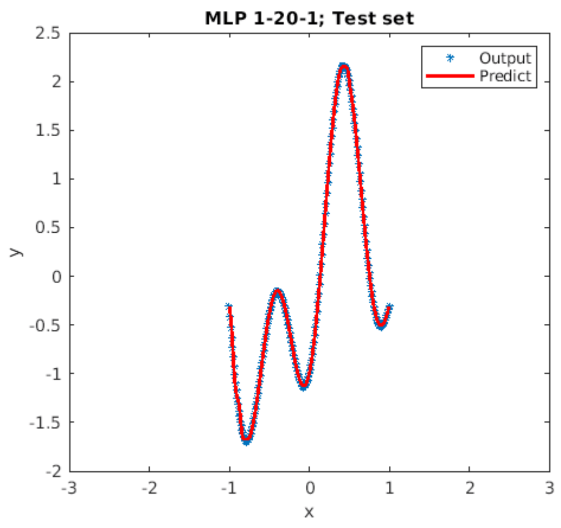

MLP 1-9-1; Training set
Output
Predict
x
y

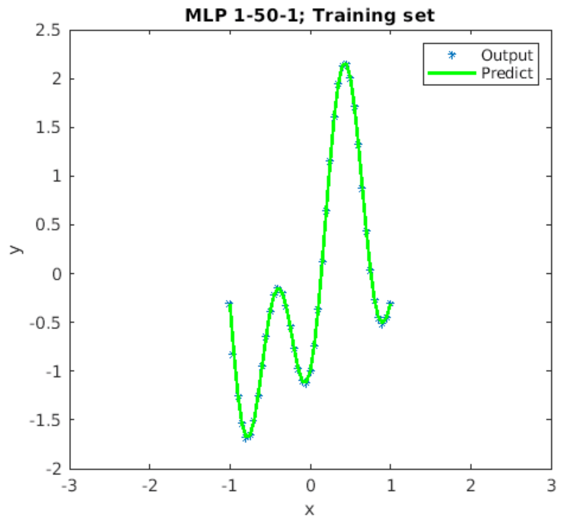

MLP 1-9-1; Test set
Output
Predict
x
y

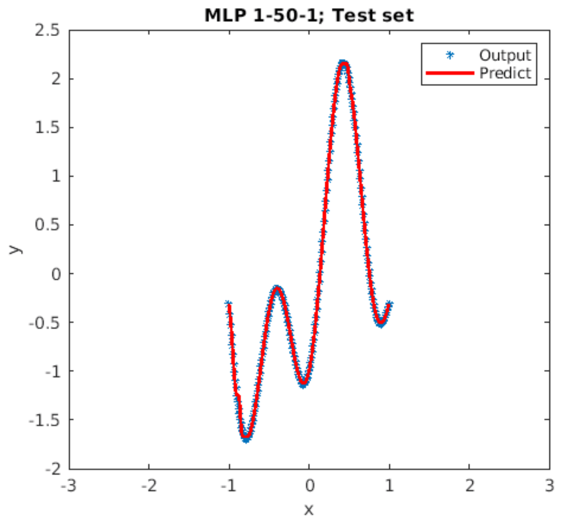

MLP 1-10-1; Training set
Output
Predict
x
y

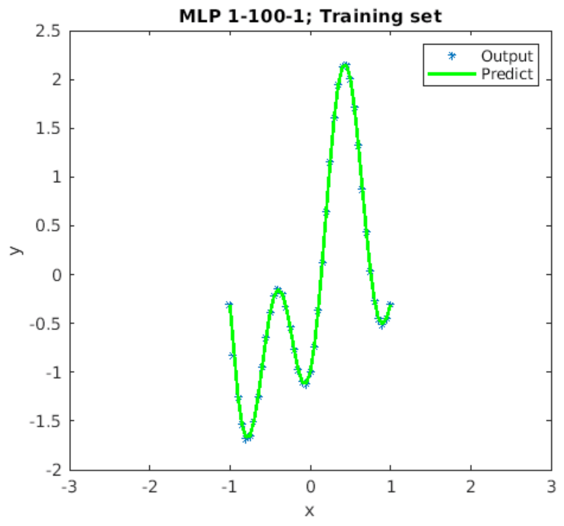

MLP 1-10-1; Test set
Output
Predict
x
y

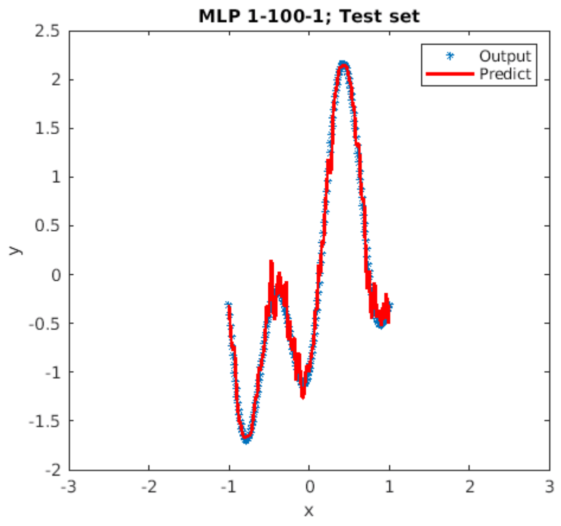

MLP 1-20-1; Training set
Output
Predict
y
x
2.5
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-3
-2
-1
0
1
2
3


MLP 1-20-1; Test set
Output
Predict
y
x
2.5
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-3
-2
-1
0
1
2
3


MLP 1-50-1; Training set
Output
Predict
y
x
2.5
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-3
-2
-1
0
1
2
3


MLP 1-50-1; Test set
Output
Predict
y
x
2.5
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-3
-2
-1
0
1
2
3


MLP 1-100-1; Training set
Output
Predict
y
x
2.5
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-3
-2
-1
0
1
2
3


MLP 1-100-1; Test set
Output
Predict
y
x
2.5
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-3
-2
-1
0
1
2
3

n = 1 results for x = -3 and x = 3: -0.9261   0.8445
n = 2 results for x = -3 and x = 3: -0.89371   -0.60013
n = 3 results for x = -3 and x = 3: -1.1667   -1.2493
n = 4 results for x = -3 and x = 3: 2.1483   -0.46473
n = 5 results for x = -3 and x = 3: 0.56074   -0.35458
n = 6 results for x = -3 and x = 3: 0.20123   0.37594
n = 7 results for x = -3 and x = 3: 0.17282   1.8486
n = 8 results for x = -3 and x = 3: 0.22725   1.7779
n = 9 results for x = -3 and x = 3: 0.54986   1.8311
n = 10 results for x = -3 and x = 3: 2.0614   1.7155
n = 20 results for x = -3 and x = 3: -1.5189   -0.27712
n = 50 results for x = -3 and x = 3: -1.6701   1.9089
n = 100 results for x = -3 and x = 3: -0.18742   -0.014933

From the fitting plots, we can see that when n < 5, the outputs of test sample are underfit, but the underfit is not very worse as the model still can predict in high accuracy. As n increases from 5 to 50, the plots of test samples show good fit on the trained model. And the minimum number of neurons for this MLP model is 10.

For all the cases of n in this model, the predicted outputs are poor, compared with the desired outputs 0.809 when x = 3 or x = -3. Thus, the MLP cannot make reasonable predictions outside of the domain of the input limited by the training set as there is no training samples for the model to learn.

Q3.
My matric number A0117981X, therefore 81%4 + 1 = 2, my assigned folder is group 2. (street and mountain)

a). Rosenblatt's perceptron (single layer perceptron)
Source code in Python:

```python
import numpy as np
import os
from glob import glob
from tqdm import tqdm
import cv2
import matplotlib.pyplot as plt
import re
import random

# image process
def getData(path,image_size):
    img_count = 0
    y =[]
    x =[]
    for img_path in tqdm(glob(path + '/*.jpg')):
        img_count +=1
        #read image
        I = cv2.imread(img_path,cv2.IMREAD_GRAYSCALE)
        # I = cv2.cvtColor(I,cv2.COLOR_BGR2GRAY)
        #resize the image
        I = cv2.resize(I,(img_size,img_size))

        x.append(np.reshape(I,[img_size*img_size,1]))
        #get labels
        if re.findall(r'_.*?_(.*?)_.*?', img_path)[0] == '1':
            y.append(1)
        else :
            y.append(0)
    x_dataset = np.array(x).reshape(img_count,img_size**2).astype(float)
    y_dataset = np.array(y).reshape(img_count,1).astype(float)
    return x_dataset,y_dataset

def sigmoid(z):
    return 1/(1+np.exp(-z))

def predict(x,w):
    return 1 if np.dot(w,x) > 0 else 0

def accu_rate(x,y,w):
    errors = 0
    for i in range(x.shape[0]):
        if predict(x[i],w) != y[i] :
            errors +=1
    return 1- errors/x.shape[0]

# train model to get updated weights
def train(x,y,lr):
    w = np.zeros((1,img_size**2+1))
    _W = w.copy()
    for j in range(epochs):
        for i in range(x.shape[0]):
            update = lr*(y[i] - sigmoid(w.dot(x[i])))[0]
            w += update*x[i]
        _W = np.append(_W,w,axis=0)
    return _W

# image path
train_path = "D:\\group_2\\train"
val_path = "D:\\group_2\\val"
img_size =256
epochs = 100
learn_rate = 0.01

x_train,y_train = getData(train_path,img_size)
x_val,y_val = getData(val_path,img_size)

# add bias
b_train = np.ones([x_train.shape[0],1])
b_valid = np.ones([x_val.shape[0],1])

x_train = np.append(x_train,b_train,axis =1)
x_val = np.append(x_val,b_valid,axis =1)

# Normalize data
x_train = x_train / x_train.max(axis=0)
x_val = x_val / x_val.max(axis=0)

# updated weights
W_train = train(x_train,y_train,learn_rate)

train_accu_set = []
valid_accu_set =[]

for i in range(epochs):
    acc_train = accu_rate(x_train,y_train,W_train[i+1])
    acc_val = accu_rate(x_val,y_val,W_train[i+1])
    train_accu_set.append(acc_train)
    valid_accu_set.append(acc_val)

plt.figure()
plt.plot(range(epochs),train_accu_set,label='train accuracy rate')
plt.plot(range(epochs),valid_accu_set,label='valid accuracy rate')
print("Accuracy for validation set is "+'%.4f' % (valid_accu_set[-1]*100) + '%')
plt.xlabel('epochs')
plt.ylabel('accuracy')
plt.legend()
plt.show()

```

I implement this single layer perceptron in Python. First is to preprocess all the train and valid image data from local path. And the original image is grayscale format with size 256*256. Therefore, by using cv2.imread function from OpenCV in python, the images can be read as 256*256 metric directly. Then reshape each image to one-dimension metric. Then combine all the metric into one metric. The metrics data is normalized before it is fed into the model. This is to ensure each input parameter has a similar data distribution. And this makes convergence faster while training the sample dataset. I make each feature parameter range from [0,1] by dividing by 255 which is the max value per column feature. The implemented model has a learning rate 0.01 and sigmoid activation function.

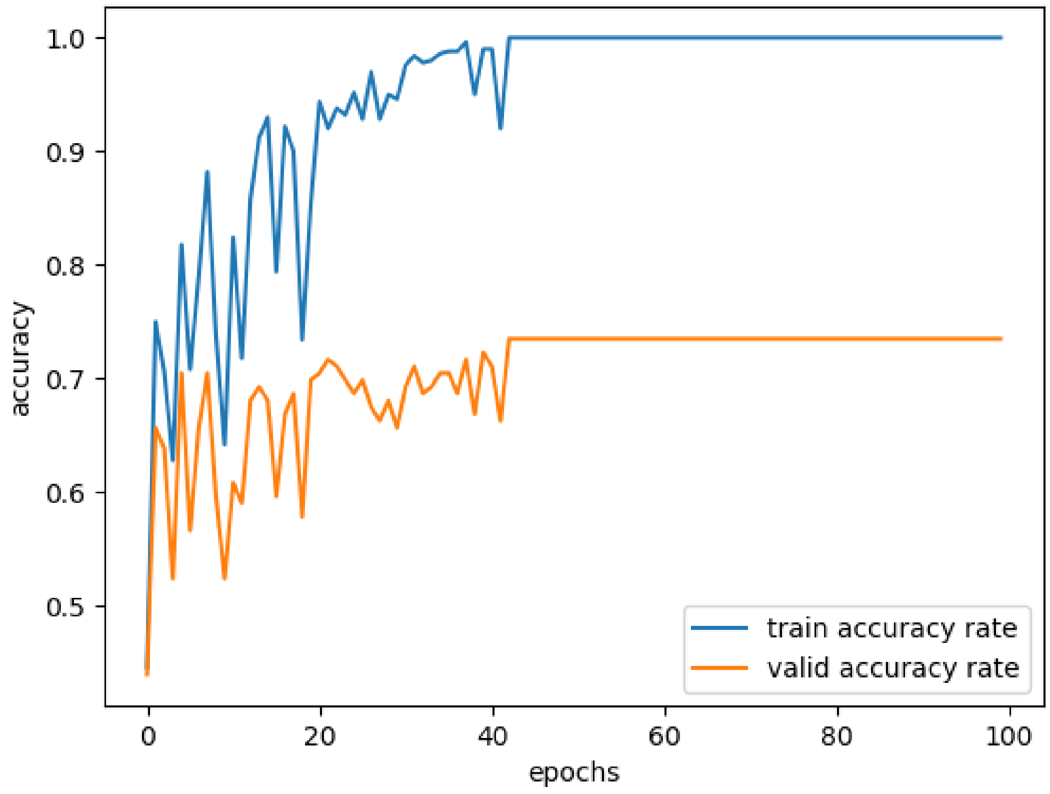
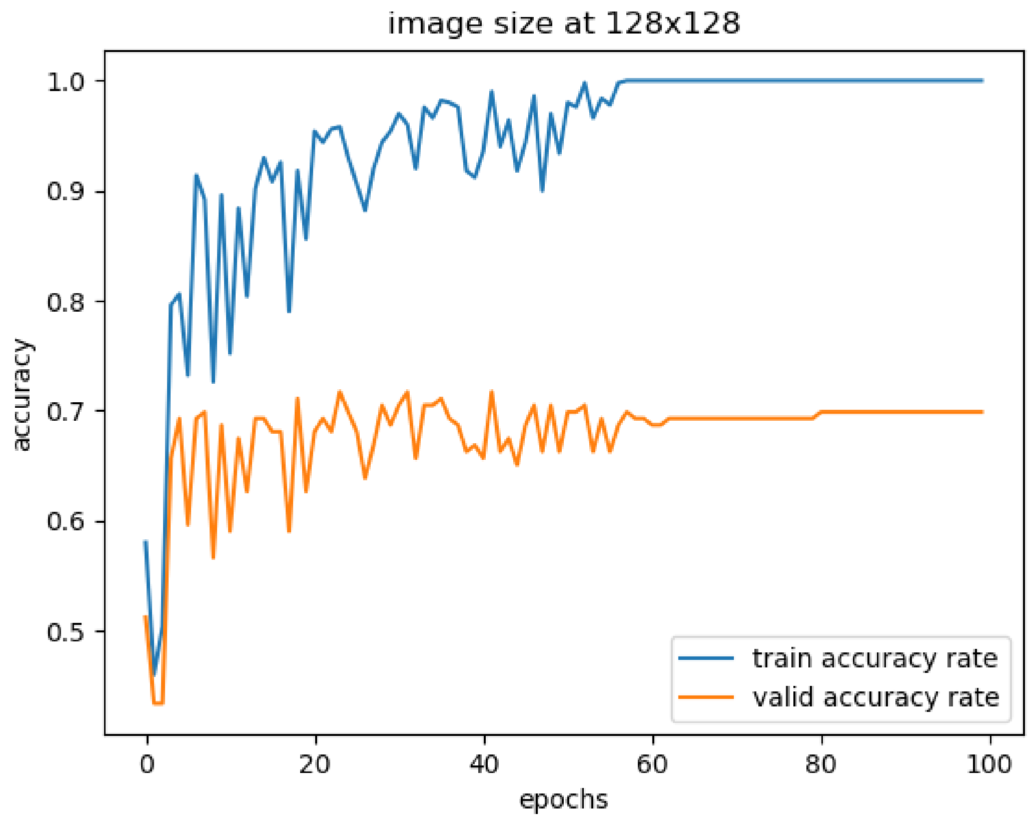


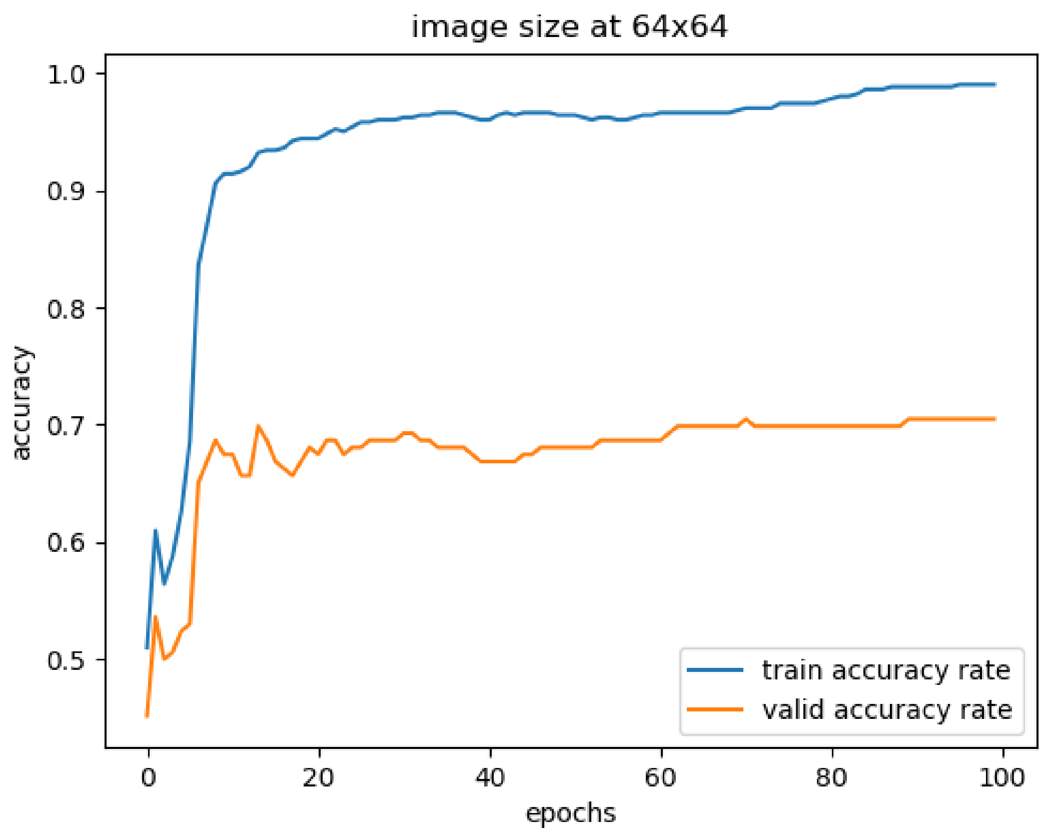
Figure 5 Accuracy versus number of epochs for Rosenblatt's perceptron (Training & Validation set)

As the figure 5 shows, Rosenblatt's perceptron has poor accuracy for validation dataset. Although it only takes about 50 epochs for training set to converge to 100% accuracy and reach a very good performance. However, the validation dataset only has 72.2892% accuracy. And we can observe the fluctuations for both training and valid dataset. The heavy feature dimensions may consist many redundant information which can affect the training model. The limited number of training dataset and heavy features of dimension make model hardly avoids overfitting. Therefore, this network model is not trained well.

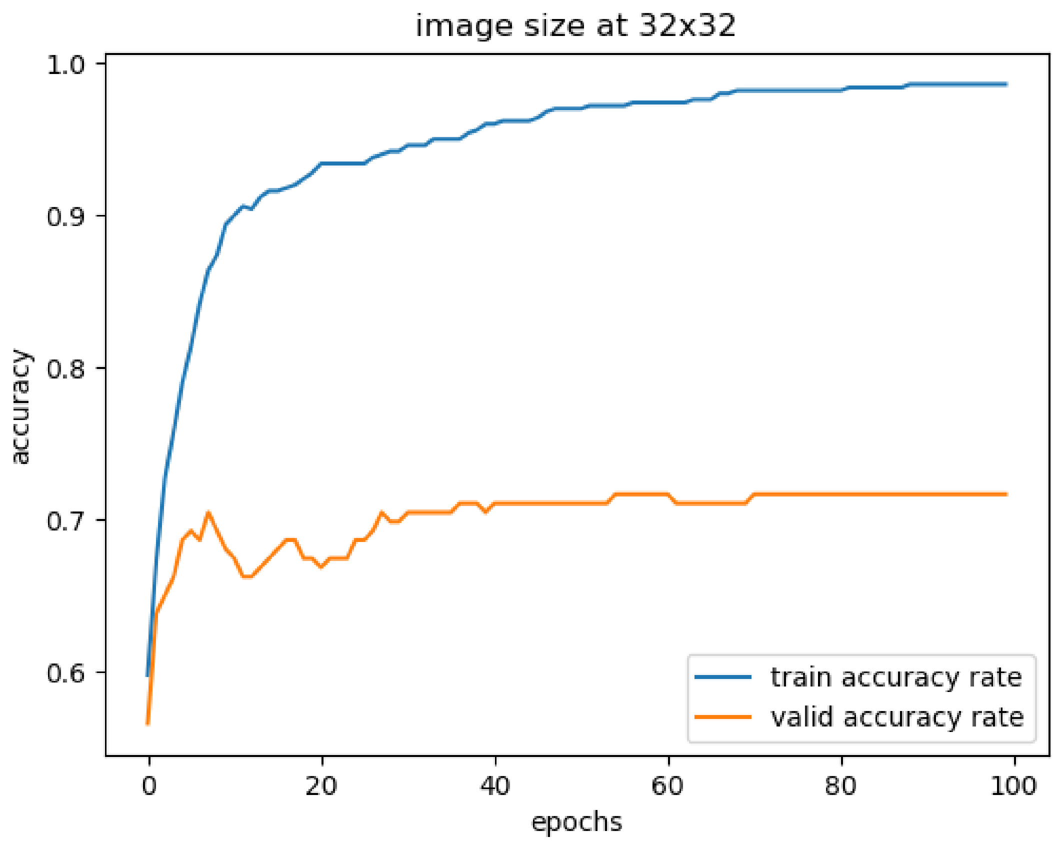
b)
The source code the same as Q3a, just change the variable: img_size to 128, 64, 32 and run it separately.



The accuracy for validation set is 69.8795% for image size 128 x 128



The accuracy for validation set is 70.4819% for image size 64 x 64



The accuracy for validation set is 71.6867% for image size 32 x 32

Image Size	256x256	128x128	64x64	32x32
Training Accuracy	100%	100%	100%	100%
Validation Accuracy	72.2892%	69.8795%	70.4819%	71.6867%

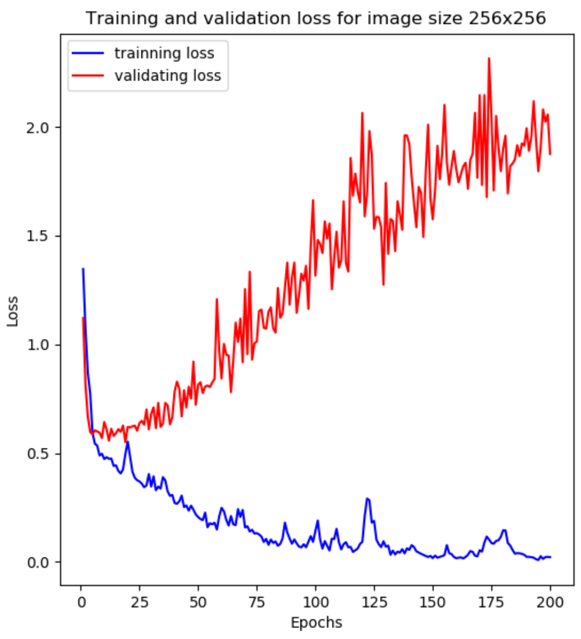
As the table shows, the validation accuracy for different size of image does not change too much. It always shows about 70% accuracy. And the fluctuation in image size 128x128 is not so severe as in image size 256x256. And we can see smooth trajectory of accuracy for both training and validation set in both 64x64 and 32x32 image sizes. Therefore, decrease the size of image can reduce loss as redundant information can be reduced. Moreover, the epochs to reach convergence for accuracy decreases as the image size get smaller. We can see that it takes about 30 epochs for model to reach convergence on validation set.

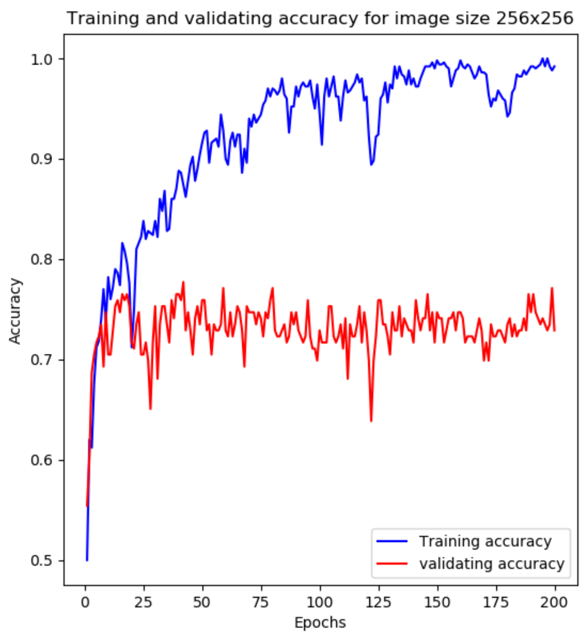
c) Apply MLP using batch mode
Source code:

The MLP is implemented in Python with Keras library. The batch size is 50 and epochs is 200. The model is tested in 4 different sizes of training and validation dataset.

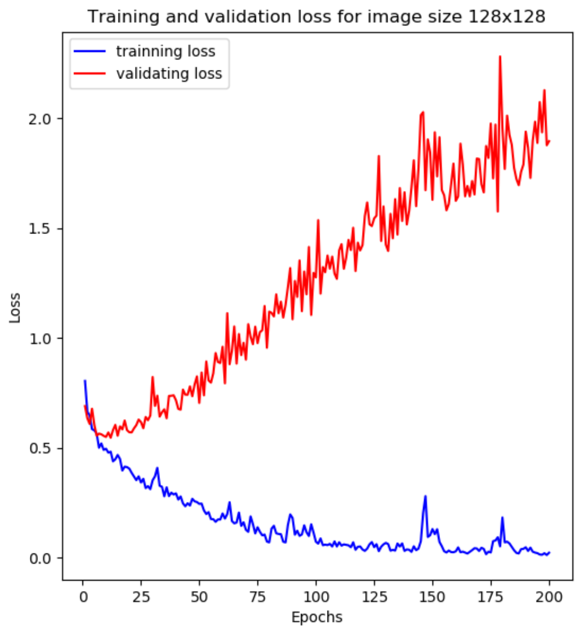
```
import numpy as np
import os
from glob import glob
from tqdm import tqdm
import cv2
import matplotlib.pyplot as plt
import re
import random
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras import regularizers
from keras.optimizers import Adam
from keras import models
from keras import layers
# image process
def getData(path,image_size):
    img_count = 0
    y =[]
    x =[]
    for img_path in tqdm(glob(path + '/*.jpg')):
        img_count +=1
        #read image
        I = cv2.imread(img_path,cv2.IMREAD_GRAYSCALE)

        #resize the image
        I = cv2.resize(I,(img_size,img_size))

        x.append(np.reshape(I,[img_size*img_size,1]))
        #get labels
        if re.findall(r'_.*?_(.*?)_.*?', img_path)[0] == '1':
            y.append(1)
```

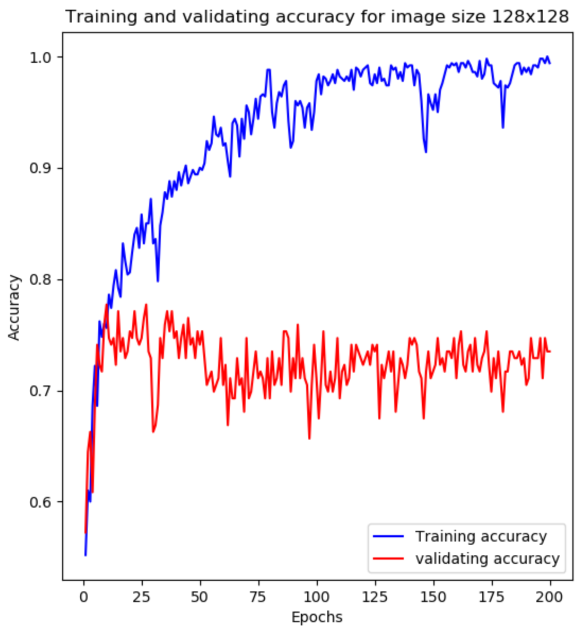
```python
        else :
            y.append(0)
    x_dataset = np.array(x).reshape(img_count,img_size**2).astype(float)
    y_dataset = np.array(y).reshape(img_count,1).astype(float)
    return x_dataset,y_dataset

train_path = "D:\\group_2\\train"
val_path = "D:\\group_2\\val"
img_size = 32

x_train,y_train = getData(train_path,img_size)
x_val,y_val = getData(val_path,img_size)

# Normalize data
x_train = x_train / x_train.max(axis=0)
x_val = x_val / x_val.max(axis=0)

# implement model
model = models.Sequential()
model.add(layers.Dense(16,input_dim=img_size**2,use_bias=True,activation = None))
model.add(Dropout(0.5))
model.add(layers.Dense(16,activation='relu'))
model.add(layers.Dense(1, kernel_initializer = 'normal' ,activation='sigmoid'))
# train model
model.compile(optimizer='adam',loss='binary_crossentropy',metrics=['accuracy'])
# evaluate model
history = model.fit(x_train,y_train,epochs=200,batch_size=50,validation_data=(x_val,y_val))


acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1,len(acc)+1)


# display training and validating loss
plt.figure()
plt.subplot(1,2,1)
plt.plot(epochs,loss,'b',label='trainning loss')
plt.plot(epochs,val_loss,'r',label='validating loss')
plt.title('Training and validation loss for image size '+str(img_size)+'x'+str(img_size))
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
# plt.show()

# display training and validating accuracy
plt.subplot(1,2,2)
# plt.clf()
plt.plot(epochs,acc,'b',label='Training accuracy')
plt.plot(epochs,val_acc,'r',label='validating accuracy')
plt.title('Training and validating accuracy for image size '+str(img_size)+'x'+str(img_size))
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()

```



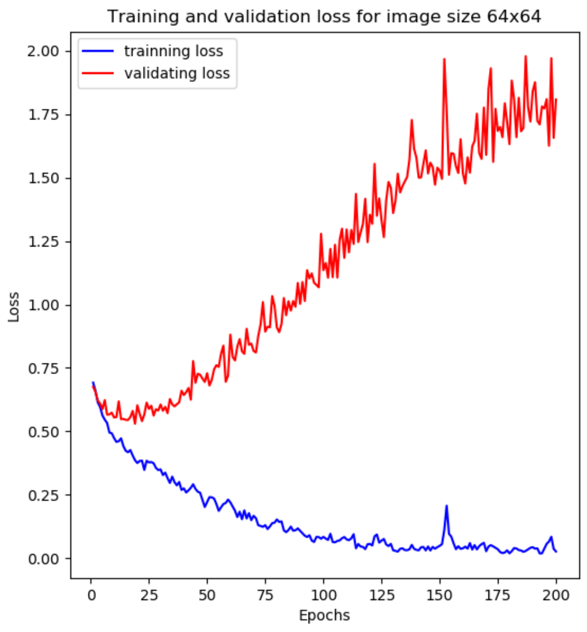


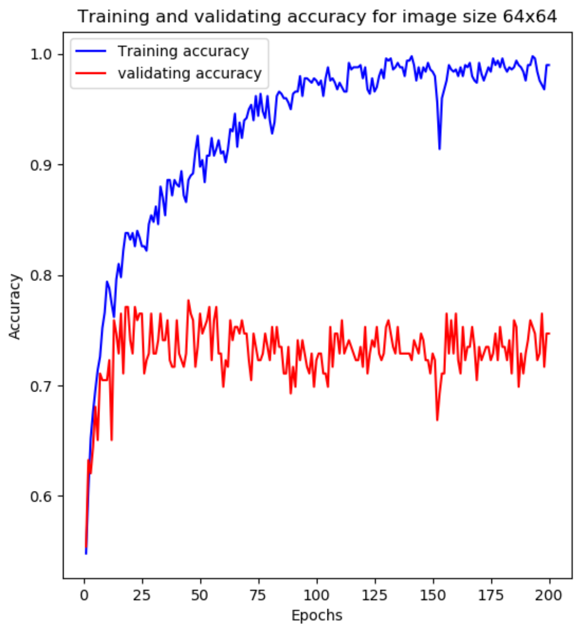
Training loss: 0.0216 - Training accuracy: 0.9920
Validation loss: 1.8752 - Validation accuracy: 0.7289



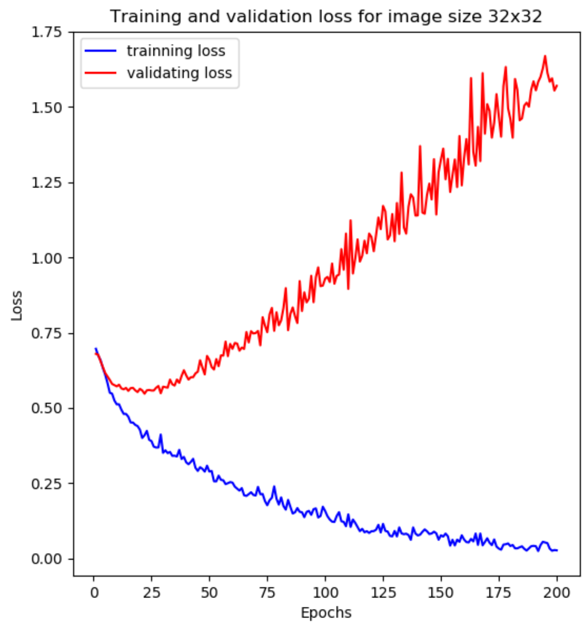


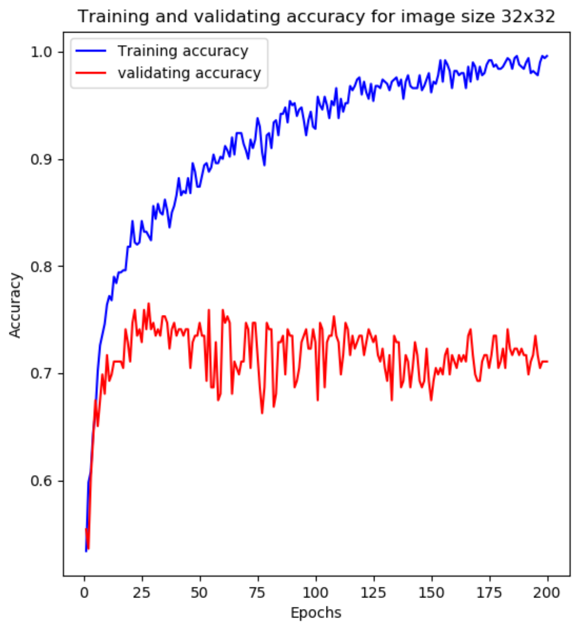
Training loss: 0.0219 -Training accuracy: 0.9940
Validation loss: 1.8967 - Validation accuracy: 0.7349





Training loss: 0.0260 - Training accuracy: 0.9900
Validation loss: 1.80827 - Validation accuracy: 0.7470





Training loss: 0.0276 - Training accuracy: 0.9960
Validation loss: 1.5693 - Validation accuracy: 0.7108

Image Size	256x256	128x128	64x64	32x32
Training Accuracy in Perceptron	100%	100%	100%	100%
Training Accuracy in MLP with batch mode	99.20%	99.40%	99.00%	99.60%

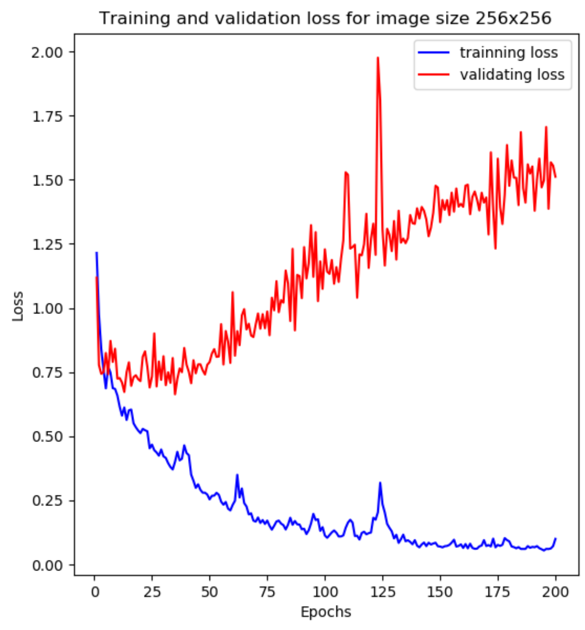
Image Size	256x256	128x128	64x64	32x32
Validation Accuracy in Perceptron	72.2892%	69.8795%	70.4819%	71.6867%
Validation Accuracy in MLP with batch mode	72.89%	73.49%	74.70%	71.08%

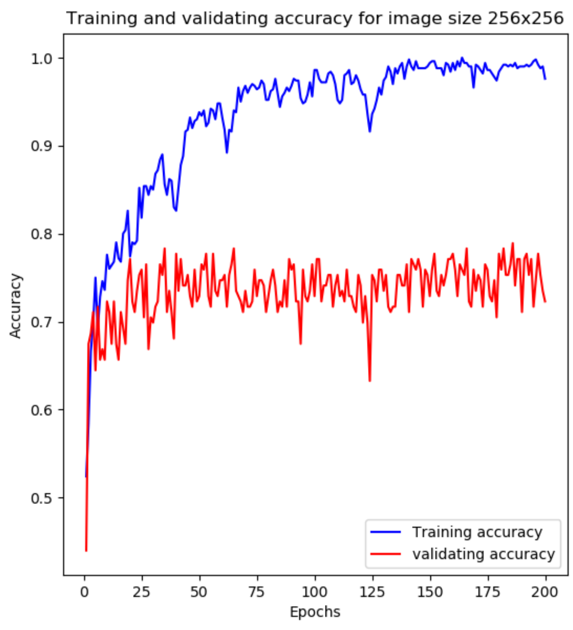
Compare with single Perceptron, the MLP model training accuracy is quite similar as training accuracy in single layer Perceptron; for the validation set, the accuracy is also quite similar in the range of 70% and 74%. And for MLP model, overfitting occurred at early epochs, we can also observe from validation loss when it starts to increase. Overall, the MLP model does not improve its classification performance on the validation set.

3d). Apply MLP using batch mode with regularization
The trained MLP in Question part c is overfitting at early epochs. It shows overfitting at 10 epochs. The turning point of loss start to increase can imply that overfitting occurs.

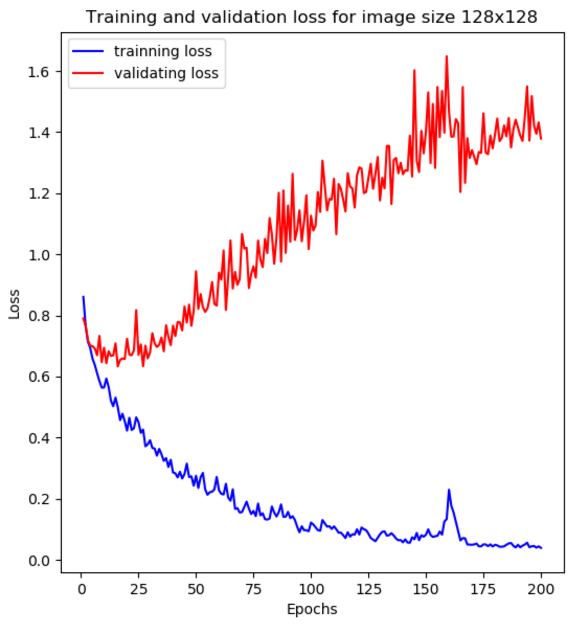
Source code for MLP model with L_2 regularization (0.01):

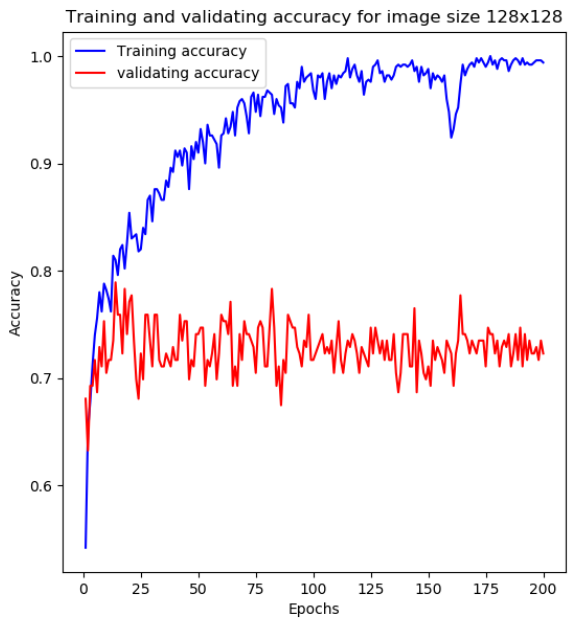
```
# implement model
model = models.Sequential()
model.add(layers.Dense(16,input_dim=img_size**2,use_bias=True,activation = None))
model.add(Dropout(0.5))
model.add(layers.Dense(16,activation='relu', use_bias=True,
kernel_regularizer = regularizers.l2(0.01)))

model.add(layers.Dense(1,kernel_initializer = 'normal',activation='sigmoid'))
# train model
model.compile(optimizer='adam',loss='binary_crossentropy',metrics=['accuracy'])
# evaluate model
history = model.fit(x_train,y_train,epochs=200,batch_size=50,validation_data=(x_val
,y_val))
```



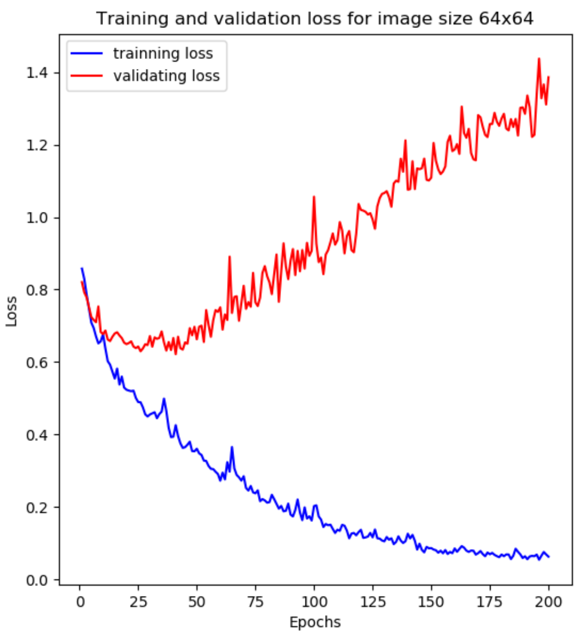


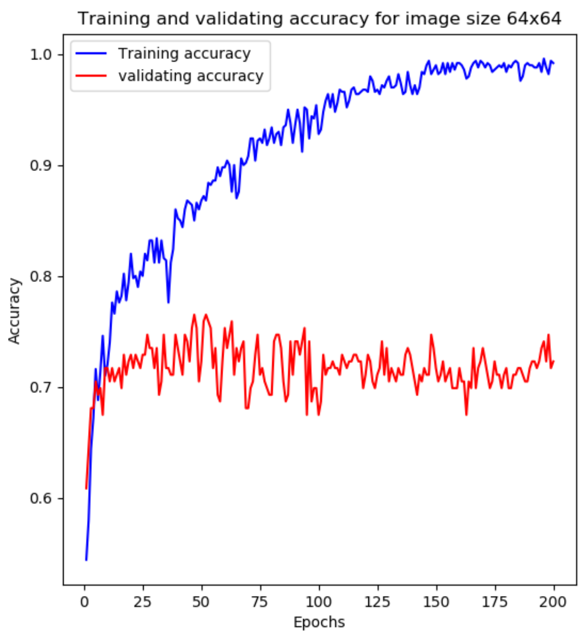
Training loss: 0.1001 – Training accuracy: 0.9760
Validation loss: 1.5117 – Validation accuracy: 0.7229



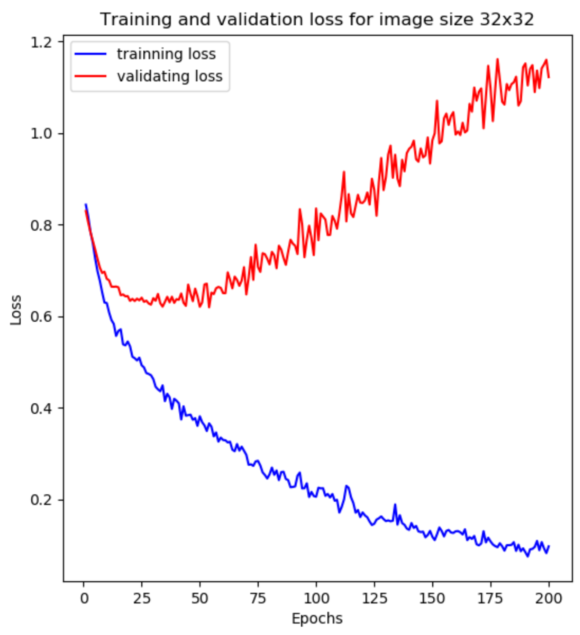


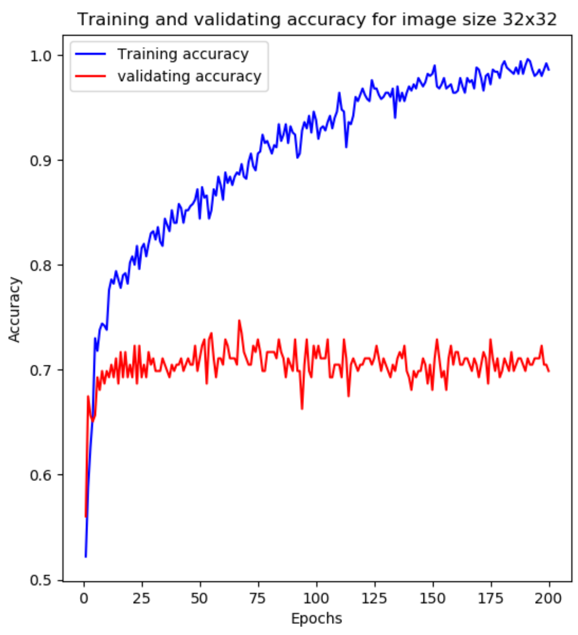
Training loss: 0.0390 - Training accuracy: 0.9940
Validation loss: 1.3788 - Validation accuracy: 0.7229





Training loss: 0.0627 - Training accuracy: 0.9920
Validation loss: 1.3858 - Validation accuracy: 0.7229





Training loss: 0.0738 - Training accuracy: 0.9900
Validation loss: 1.2652 - Validation accuracy: 0.7229

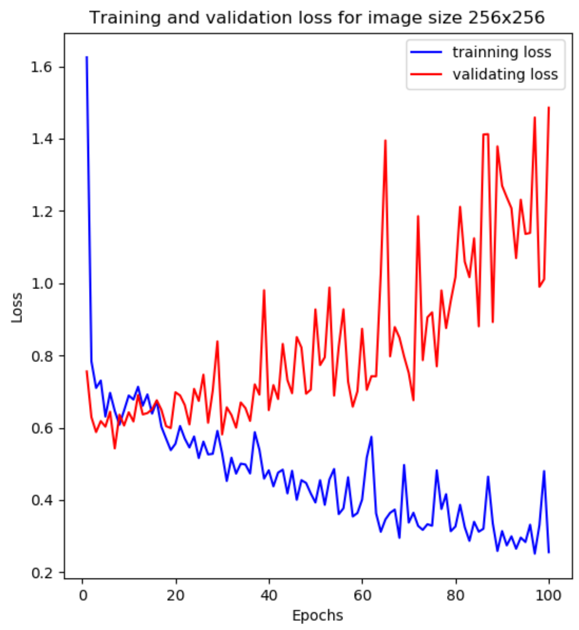
Image Size	256x256	128x128	64x64	32x32
Training Accuracy with L_2 regularization	99.40%	97.60%	99.20%	99.00%
Validation Accuracy L_2 regularization	72.29%	72.29%	72.29%	72.29%

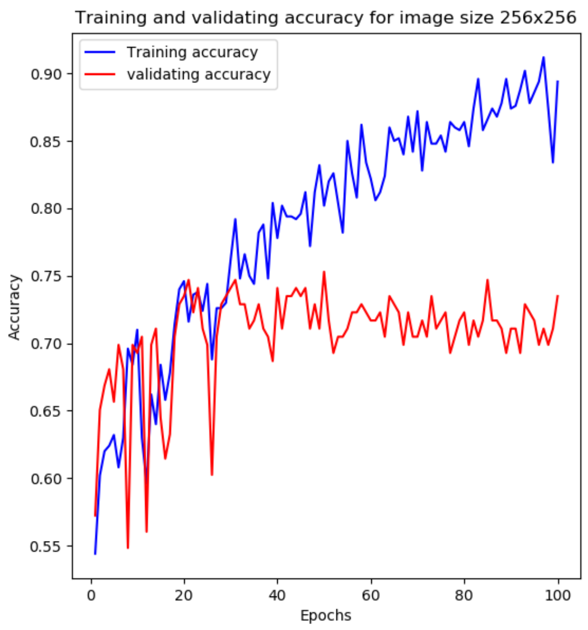
Compare the training accuracy and validation accuracy of the MLP model implemented in Question 3c, the weights regularization does not improve the validation accuracy. And the validation accuracy remains almost the same. But from the loss vs epochs curves, we can observe that the overfitting occurred at 25 epochs.

e). Apply MLP using sequential mode

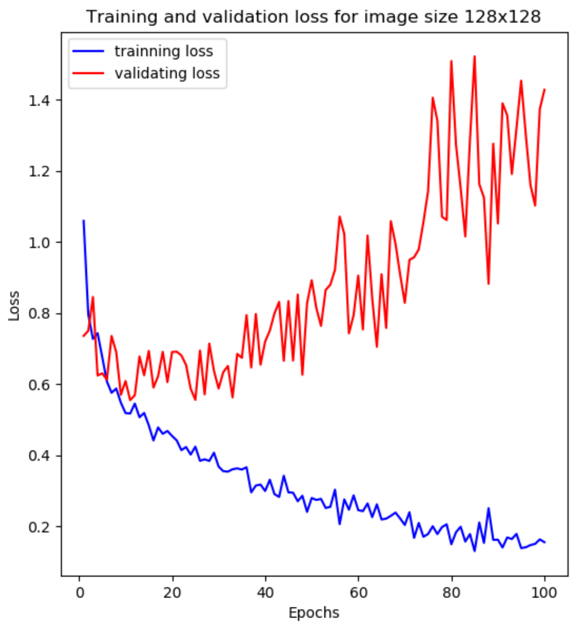
Source code:

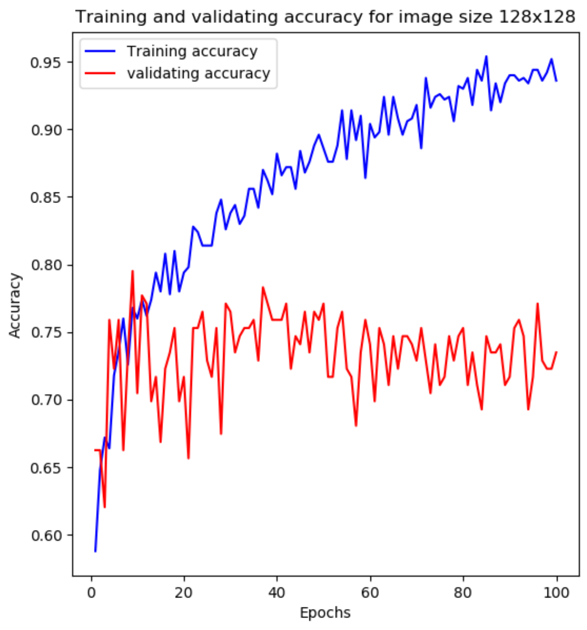
```
# implement model
model = models.Sequential()
model.add(layers.Dense(16,input_dim=img_size**2,use_bias=True,activation = None))
model.add(Dropout(0.5))
model.add(layers.Dense(16,activation='relu',use_bias=True))

model.add(layers.Dense(1,kernel_initializer = 'normal',activation='sigmoid'))
# train model
model.compile(optimizer='adam',loss='binary_crossentropy',metrics=['accuracy'])
# evaluate model
history = model.fit(x_train,y_train,epochs=100,batch_size=1,validation_data=(x_val,
y_val))
```



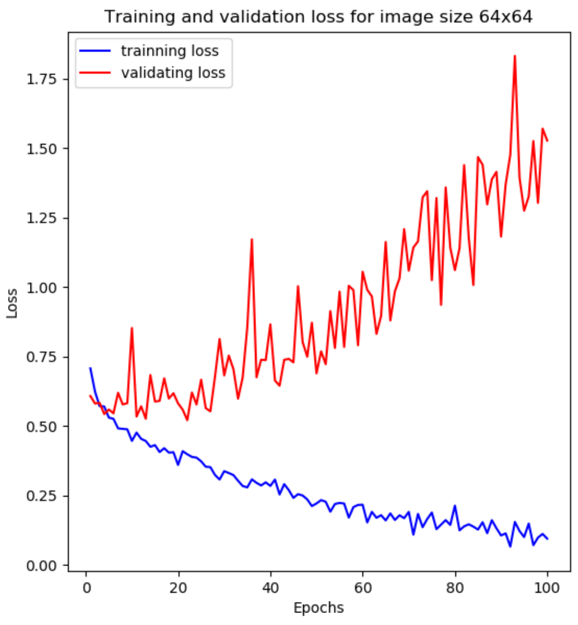


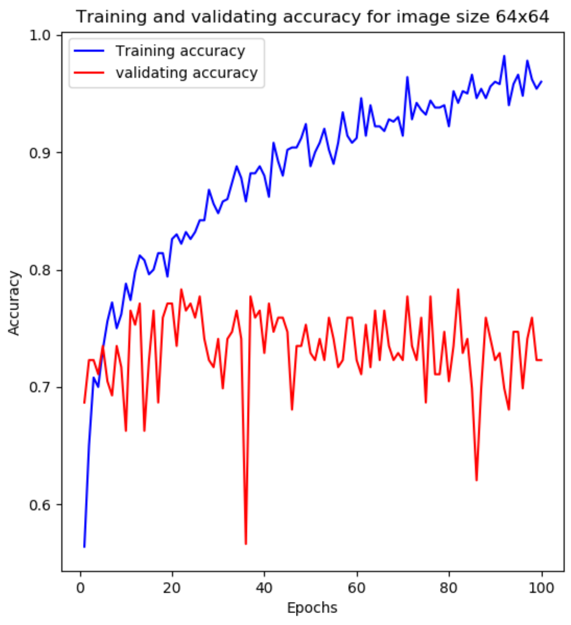
Training loss: 0.2557 - Training accuracy: 0.8940
Validation loss: 1.4856 - Validation accuracy: 0.7349



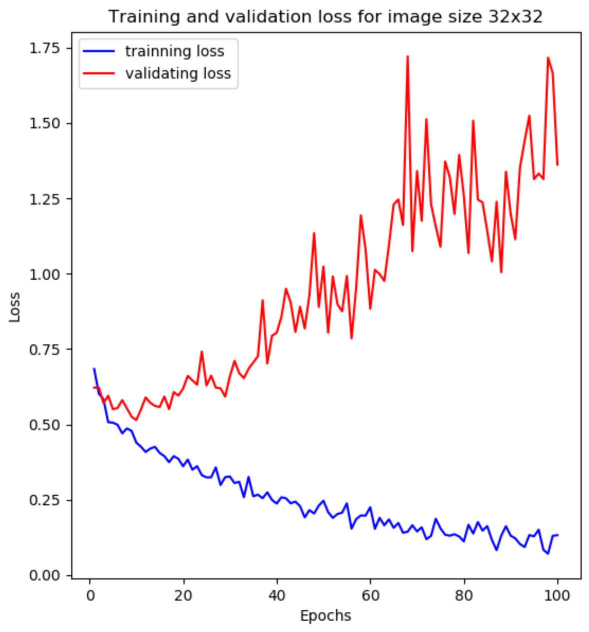


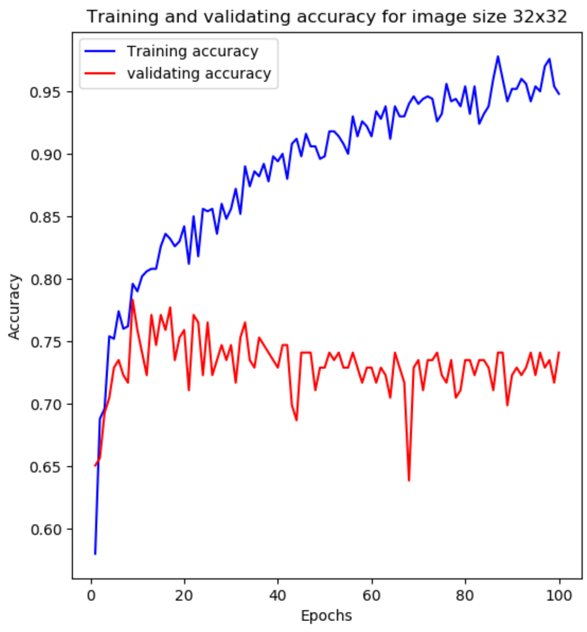
Training loss: 0.1558 - Training accuracy: 0.9360
Validation loss: 1.4274 - Validation accuracy: 0.7349





Training loss: 0.0945 – Training accuracy: 0.9600
Validation loss: 1.5264 - Validation accuracy: 0.7229





Training loss: 0.1322 - Training accuracy: 0.9480
Validation loss: 1.3619 - Validation accuracy: 0.7410

Image Size	256x256	128x128	64x64	32x32
Training Accuracy in Sequential mode	89.40%	93.60%	96.0%	94.80%
Validation Accuracy in Sequential mode	73.49%	73.49%	72.29%	74.10%

Compare the results obtained in Question 3c, we can observe that the validation accuracy in both sequential mode and batch mode is quite similar. However, the sequential mode takes longer time than batch mode. Therefore, I recommend batch mode as its less time consuming than sequential mode.

f). Proposal to improve MLP performance
My proposal is to provide proper initialized weights.
While training neural networks model, the initial weights are assigned randomly. And the random weights generally lead to big error. Although weights get updated when we train the model. But sometimes neural network can converge and stuck at local minima. The performance is not well when we assign random initial weights in MLP model. To have proper initial weights, we can try different random seed to get different random weights, then we select the seed number that works well for the model.