Q1

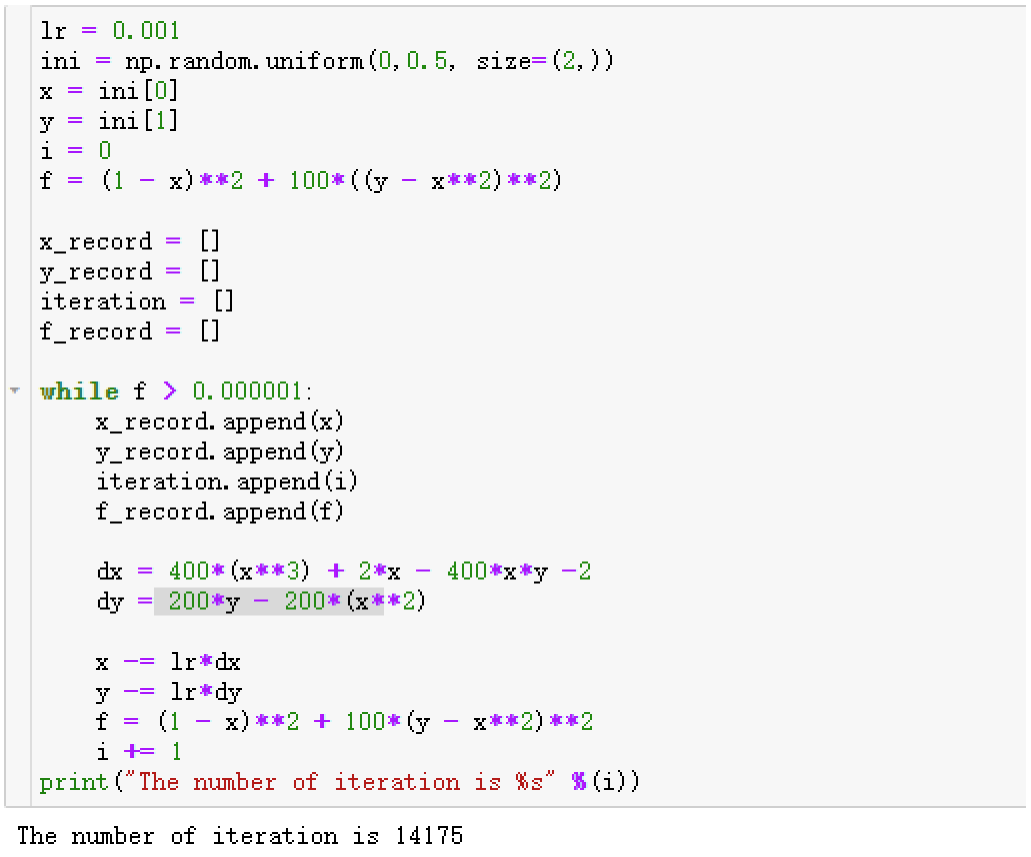
(a) Steepest (Gradient) descent method

The gradient vectors of Rosenbrock's Valley function are:

$$\frac{\partial f}{\partial x} = 400 * (x^3) + 2 * x - 400 * x * y - 2$$

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

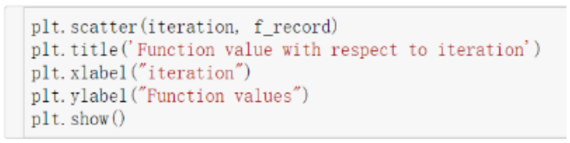
The learning rate is 0.001.



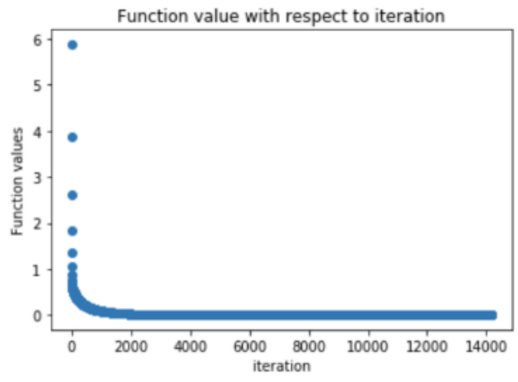
```
lr = 0.001
ini = np.random.uniform(0,0.5, size=(2,))
x = ini[0]
y = ini[1]
i = 0
f = (1 - x)**2 + 100*((y - x**2)**2)

x_record = []
y_record = []
iteration = []
f_record = []

while f > 0.000001:
    x_record.append(x)
    y_record.append(y)
    iteration.append(i)
    f_record.append(f)

    dx = 400*(x**3) + 2*x - 400*x*y -2
    dy = 200*y - 200*(x**2)

    x -= lr*dx
    y -= lr*dy
    f = (1 - x)**2 + 100*(y - x**2)**2
    i += 1
print("The number of iteration is %s" %(i))
```

```
The number of iteration is 14175
```

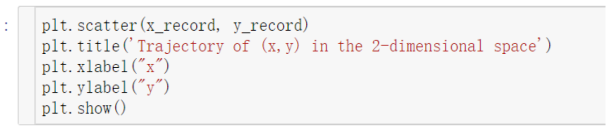
Function value with respect to iteration:



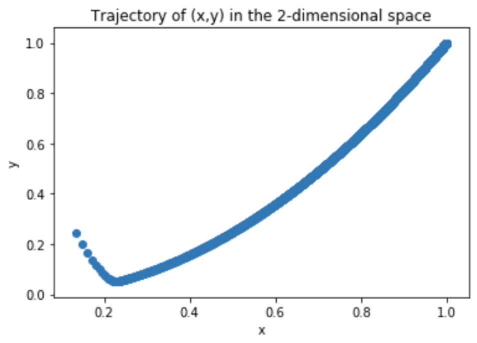
```
plt.scatter(iteration, f_record)
plt.title('Function value with respect to iteration')
plt.xlabel("iteration")
plt.ylabel("Function values")
plt.show()
```



Trajectory of (x, y):

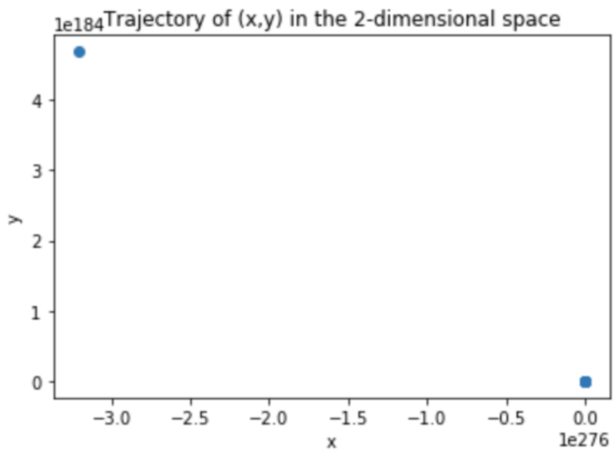


```
plt.scatter(x_record, y_record)
plt.title('Trajectory of (x,y) in the 2-dimensional space')
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```



When learning rate is 0.2, it failed which means it cannot converge. The function value is NAN; the Taylor approximation cannot work.

```
nan
```



(b) Newton's method

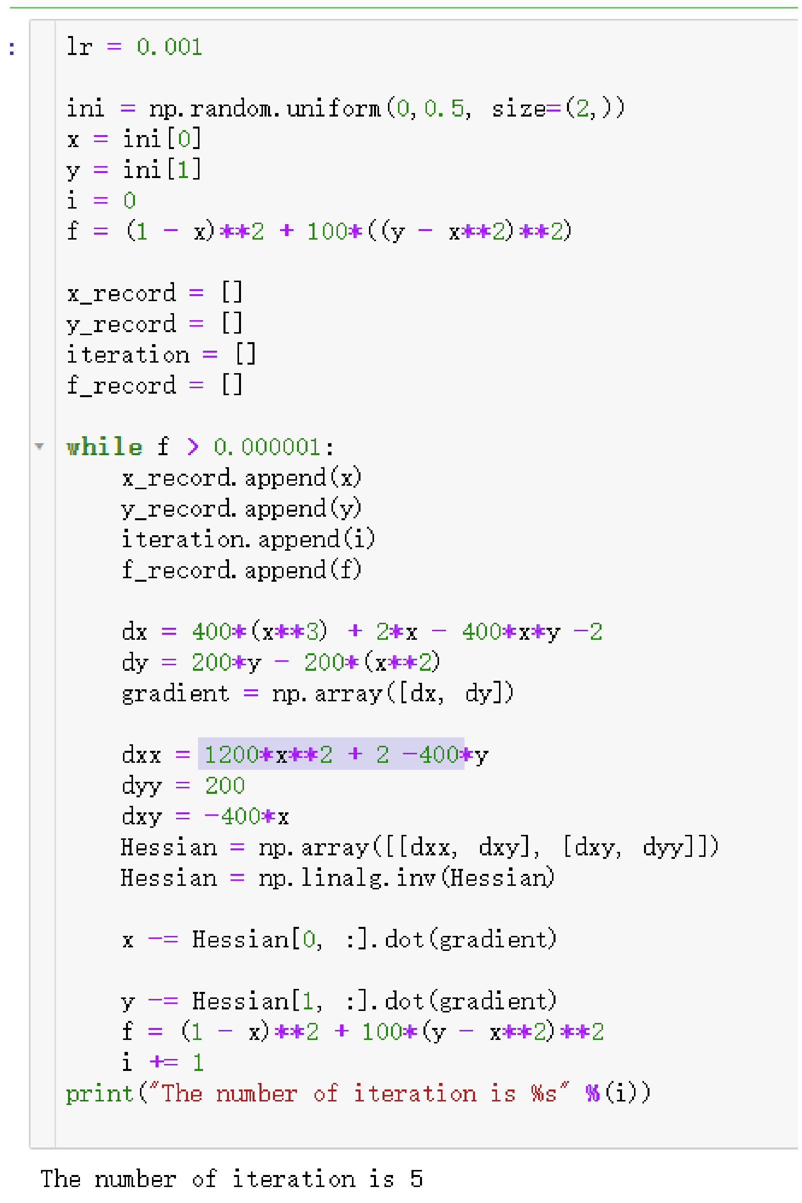
The hessian matrix is:

$$\frac{\partial^2 f}{\partial x^2} = 1200 * x^2 + 2 - 400 * y$$

$$\frac{\partial^2 f}{\partial y^2} = 200$$

$$\frac{\partial^2 f}{\partial xy} = -400 * x$$

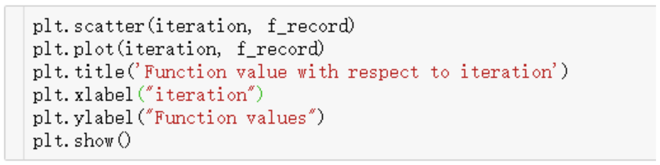
The learning rate is 0.001,



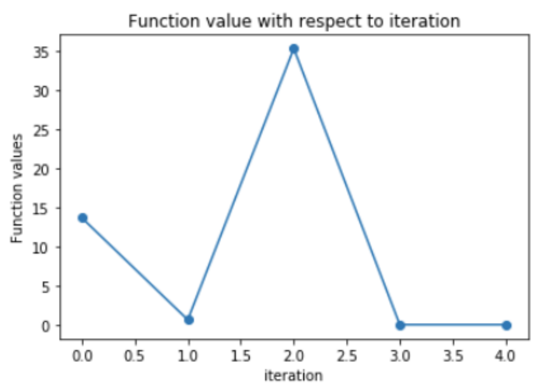
```
lr = 0.001

ini = np.random.uniform(0,0.5, size=(2,))
x = ini[0]
y = ini[1]
i = 0
f = (1 - x)**2 + 100*((y - x**2)**2)

x_record = []
y_record = []
iteration = []
f_record = []

while f > 0.000001:
    x_record.append(x)
    y_record.append(y)
    iteration.append(i)
    f_record.append(f)

    dx = 400*(x**3) + 2*x - 400*x*y -2
    dy = 200*y - 200*(x**2)
    gradient = np.array([dx, dy])

    dxx = 1200*x**2 + 2 -400*y
    dyy = 200
    dxy = -400*x
    Hessian = np.array([[dxx, dxy], [dxy, dyy]])
    Hessian = np.linalg.inv(Hessian)

    x -= Hessian[0, :].dot(gradient)

    y -= Hessian[1, :].dot(gradient)
    f = (1 - x)**2 + 100*(y - x**2)**2
    i += 1
print("The number of iteration is %s" %(i))
```

```
The number of iteration is 5
```

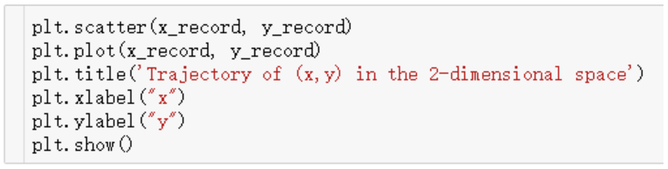
Function value with respect to iteration:



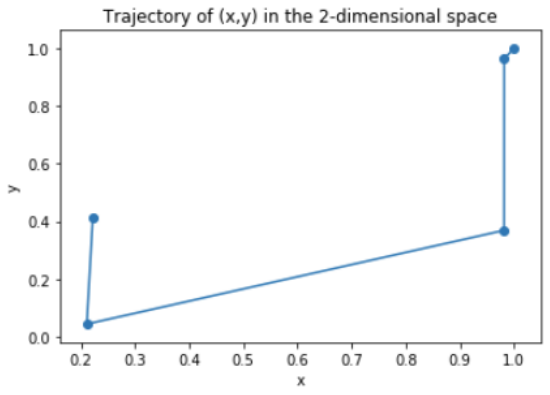
```
plt.scatter(iteration, f_record)
plt.plot(iteration, f_record)
plt.title('Function value with respect to iteration')
plt.xlabel("iteration")
plt.ylabel("Function values")
plt.show()
```



The trajectory:



```
plt.scatter(x_record, y_record)
plt.plot(x_record, y_record)
plt.title('Trajectory of (x,y) in the 2-dimensional space')
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```

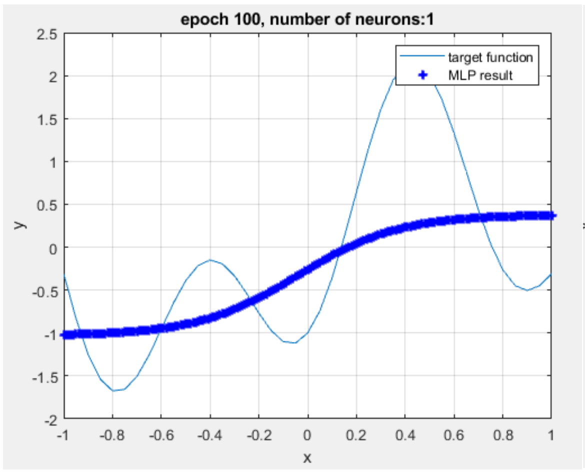


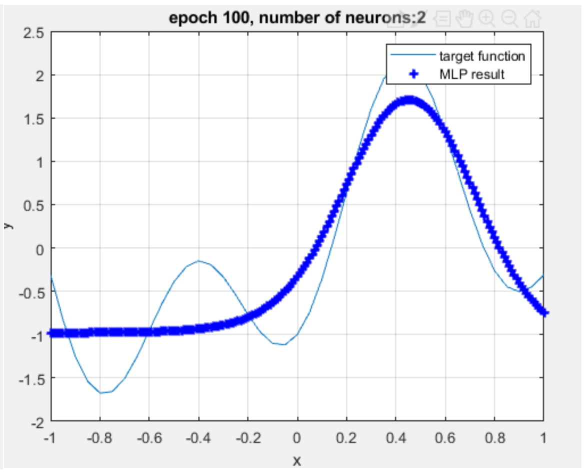
It converges very rapidly that only needs 5 iterations.

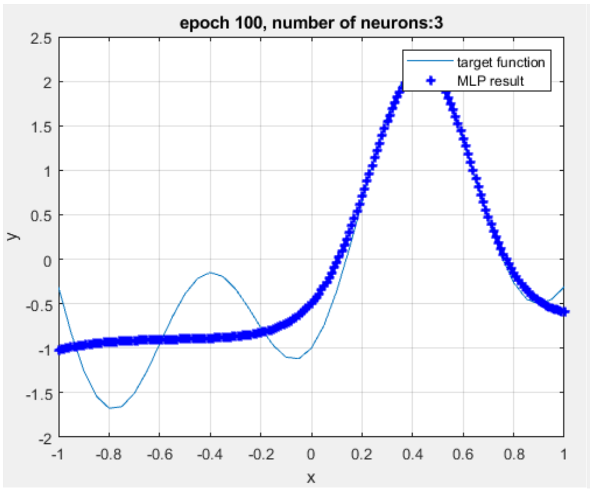
Q2

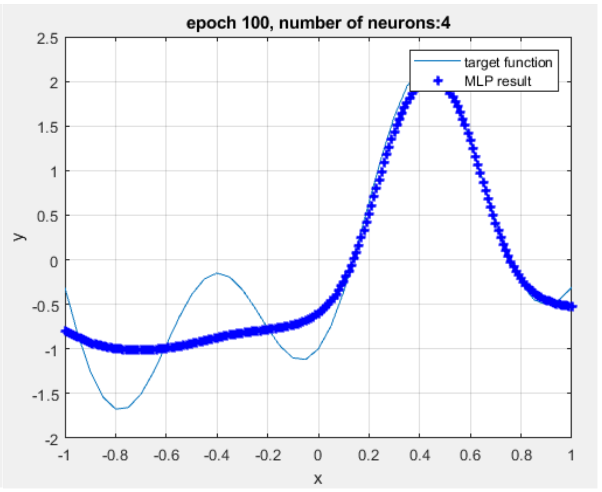
(1) Sequential mode with BP algorithm.

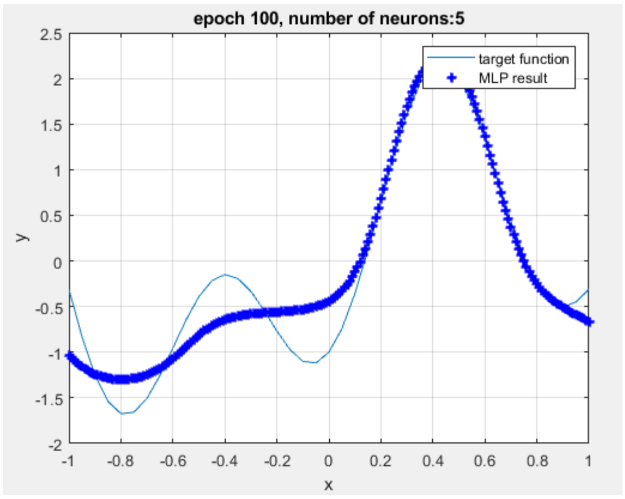
Setting different structures of the MLP: 1-n-1 (where n = 1:10, 20, 50), we can get the following figures:

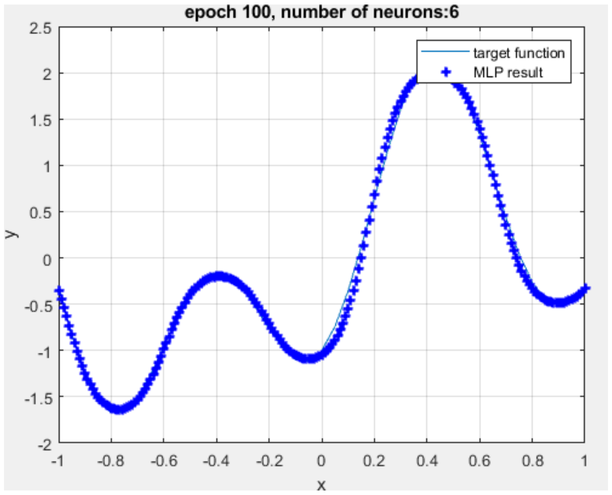


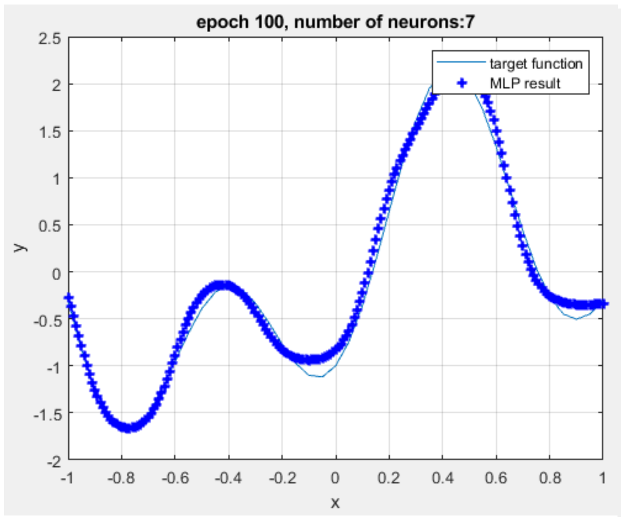


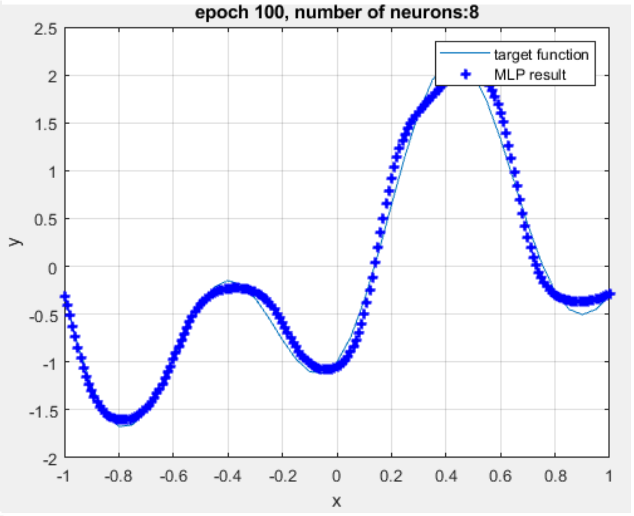


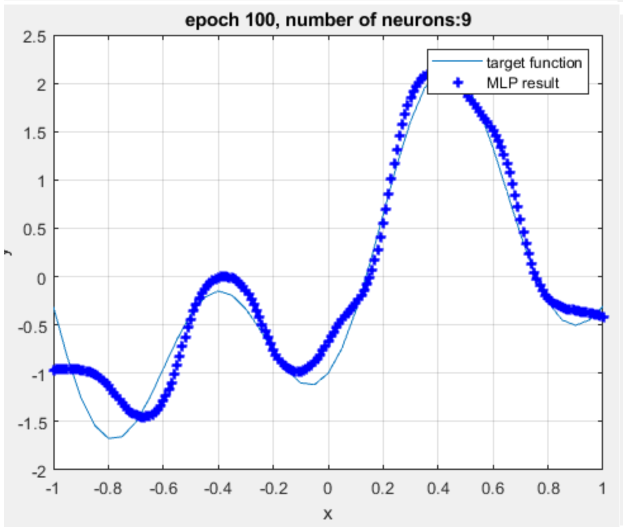


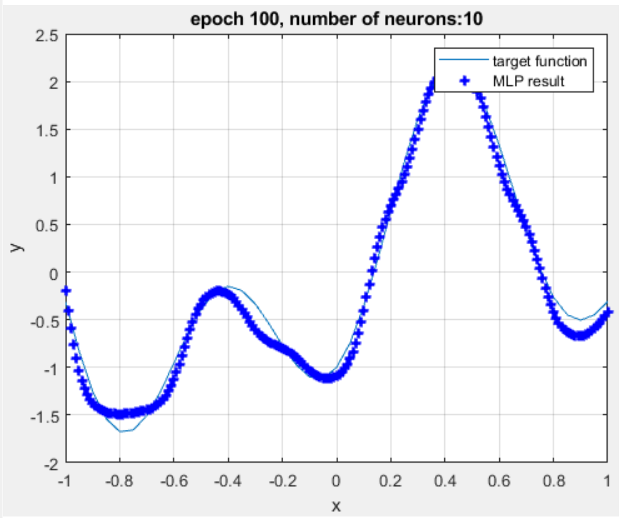


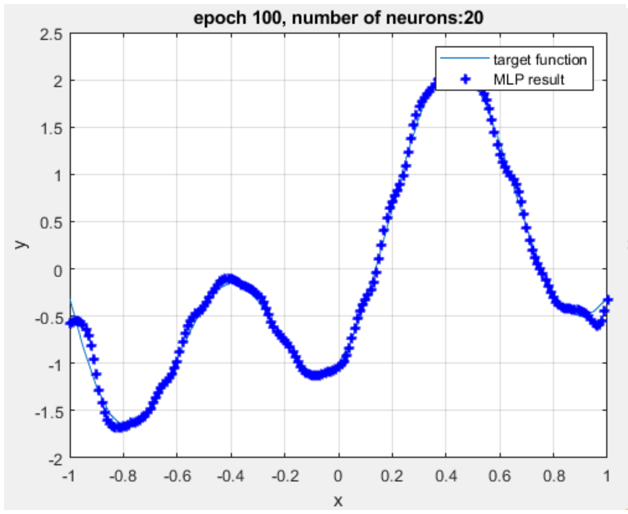


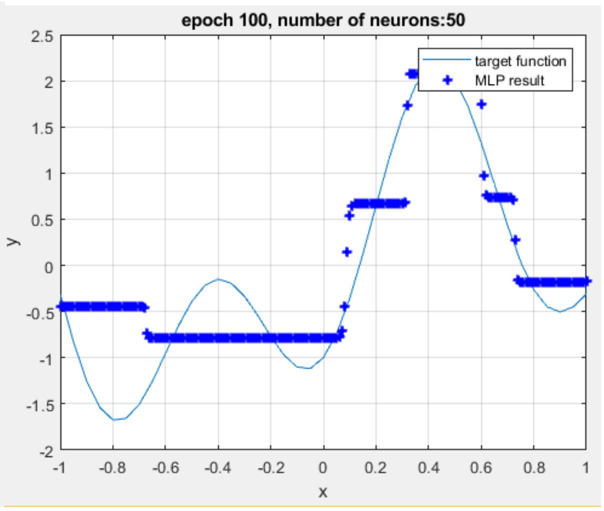

epoch 100, number of neurons:7
target function
MLP result
x
y


epoch 100, number of neurons:8
target function
MLP result
x
y


epoch 100, number of neurons:9
target function
MLP result
x


epoch 100, number of neurons:10
target function
MLP result
x
y


epoch 100, number of neurons:20
target function
MLP result
x
y


epoch 100, number of neurons:50
target function
MLP result
x
y

Summarized the results in table:

	Under-fitting	Proper fitting	Over-fitting
N	1-5	6-10, 20	50

The minimal number of hidden neurons is 6, which is consistent with the guideline given in the lecture slides.

The outputs of MLP when x=-3 and +3:

X=-3 -> y_{gt}=0.8090; y_{pred}= 0.3990

X=+3 -> y_{gt}=0.8090; y_{pred}= -2.7518

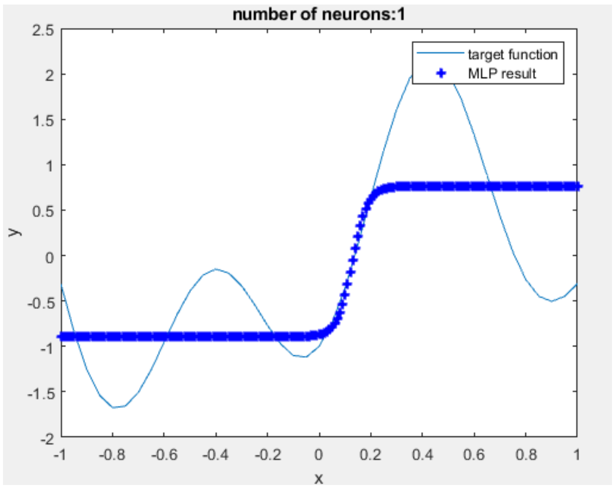
Then we can state the MLP can't make reasonable predictions outside of the domain of the training input.

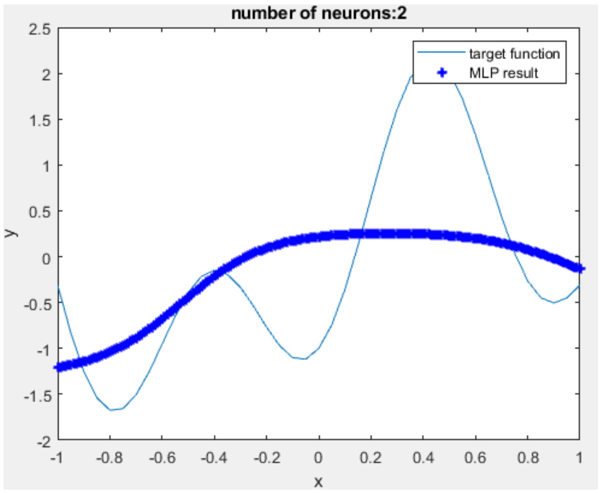
```
syms x y;
y = 1.2*sin(pi*x)-cos(2.4*pi*x);

training_x(:) = -1:0.05:1;
training_gt(1,:) = eval(subs(y,x,training_x(1,:)));

test_x(:) = -1:0.01:1;
test_gt(1,:) = eval(subs(y,x,test_x(1,:)));

%
for n = [6]
    [ net, accu_train, accu_val ] = train_seq(n, training_x, training_gt, size(tr
aining_x, 2), 0, 100);
    disp(n)
    xtest=[-3,3];
    results = sim(net, xtest)
    ground = eval(subs(y,x,xtest(1,:)))
    figure
    plot(training_x,training_gt);
    hold on;
    plot(-1:0.01:1,results(1,:), 'b+', 'LineWidth', 2);
    legend('target function', 'MLP result')
    title(['epoch ', num2str(100), ', number of neurons:', num2str(n)])
    xlabel('x')
    ylabel('y')
    grid
end
```

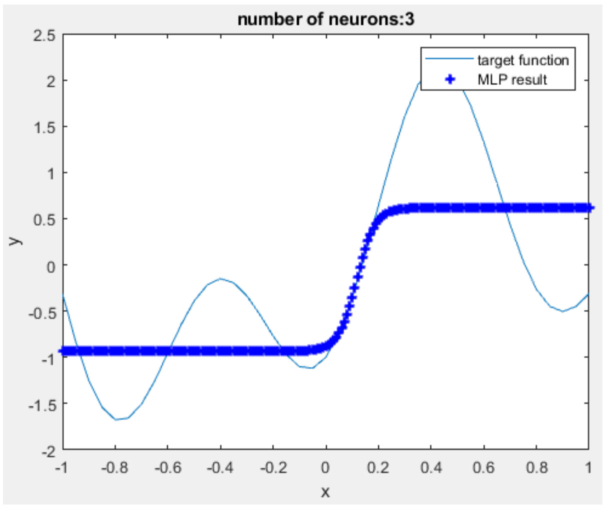
Train_seg:

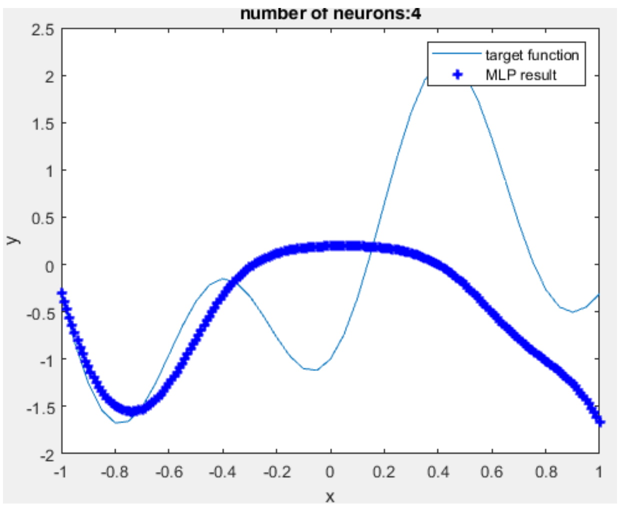
```
net = fitnet(n);
    net.divideFcn = 'dividetrain'; % input for training only
    net.performParam.regularization = 0.2; % regularization strength
    net.trainFcn = 'traingdx'; % 'trainrp' 'traingdx'
    net.trainParam.epochs = epochs;
    net.trainParam.lr = 0.01;
```

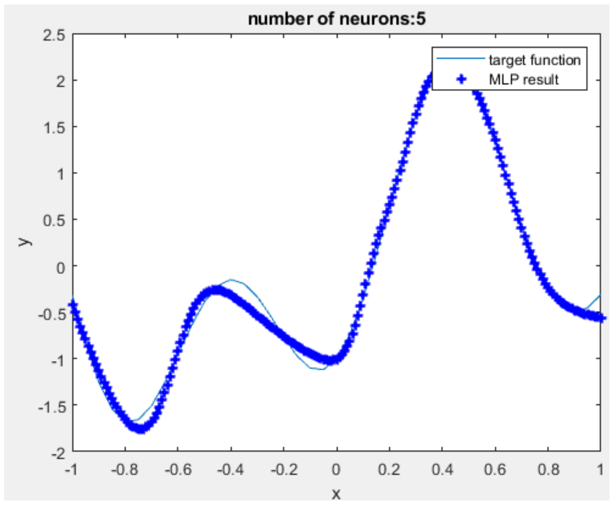
(2) Batch mode with trainlm algorithm

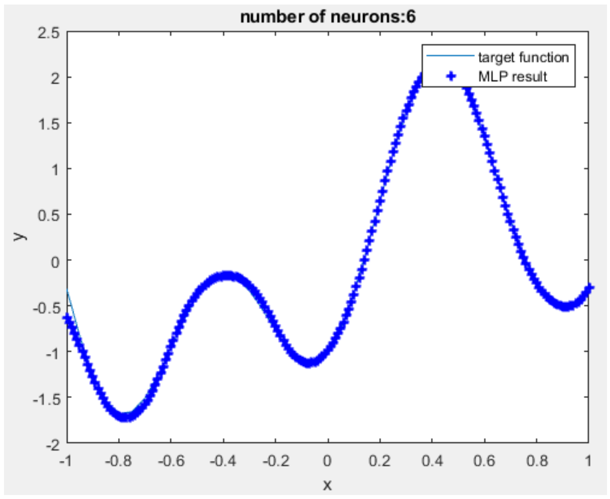


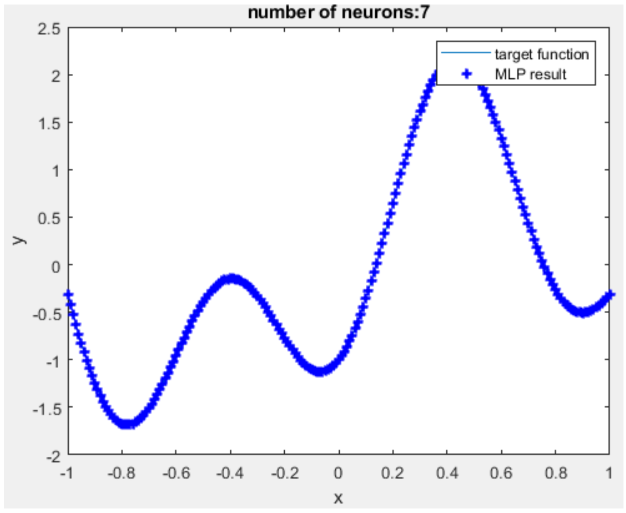


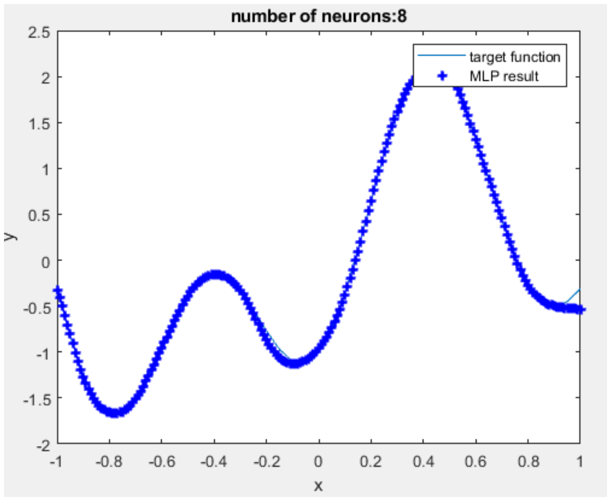


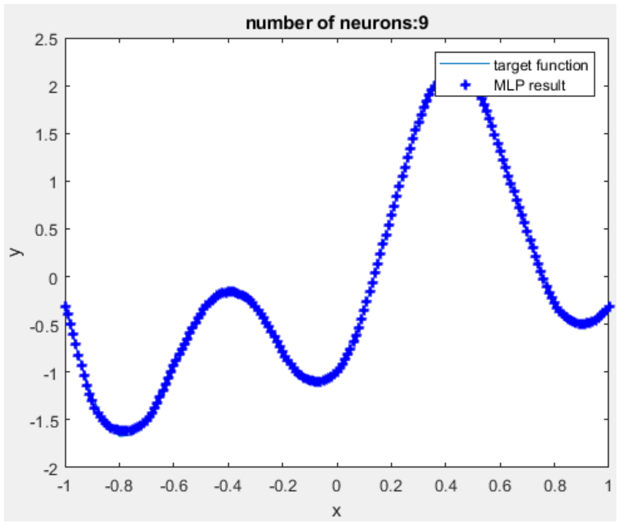


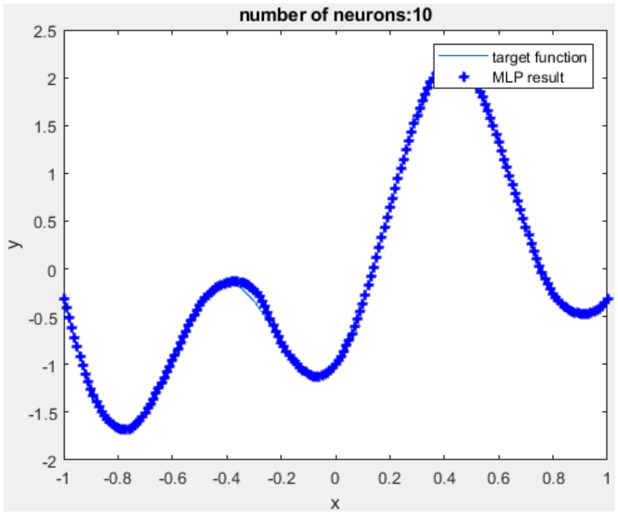

number of neurons:5
target function
MLP result
x
y

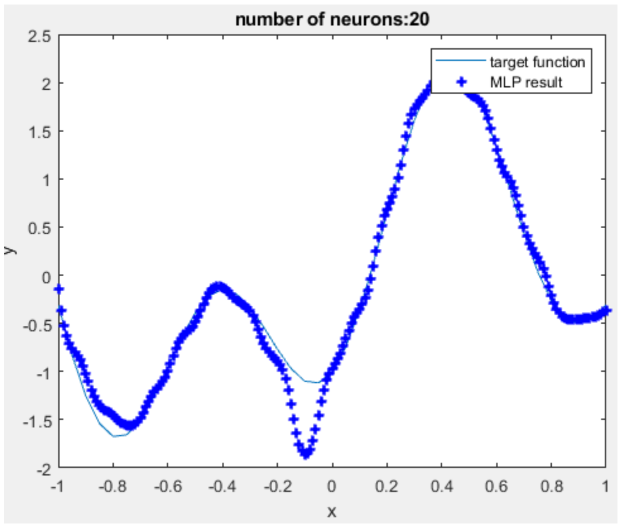

number of neurons:6
target function
MLP result
x
y

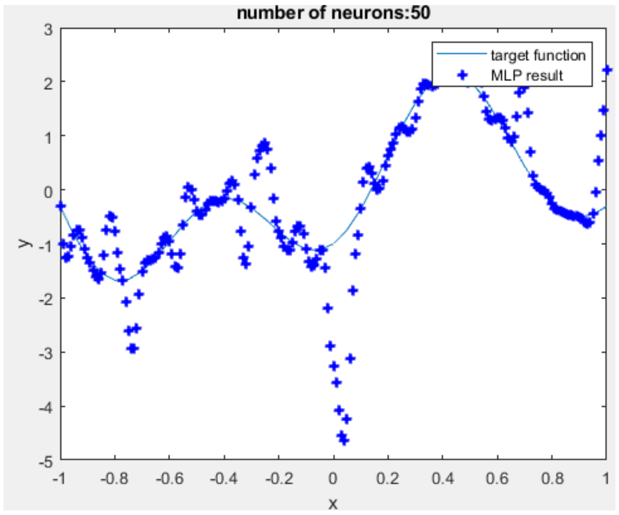

number of neurons:7
target function
MLP result
x
y


number of neurons:8
target function
MLP result
x
y


number of neurons:9
target function
MLP result
x
y


number of neurons:10
target function
MLP result
x
y





Summarized the results in table:

	Under-fitting	Proper fitting	Over-fitting
N	1-5	6-10	20, 50

The minimal number of hidden neurons is 6, which is consistent with the guideline given in the lecture slides.

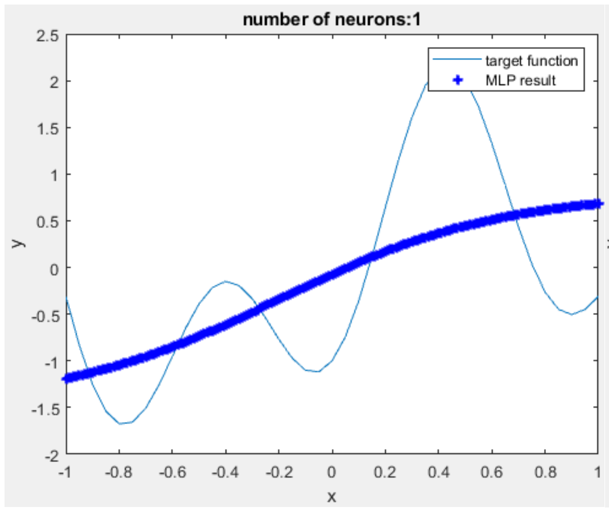
The outputs of MLP when x=-3 and +3:

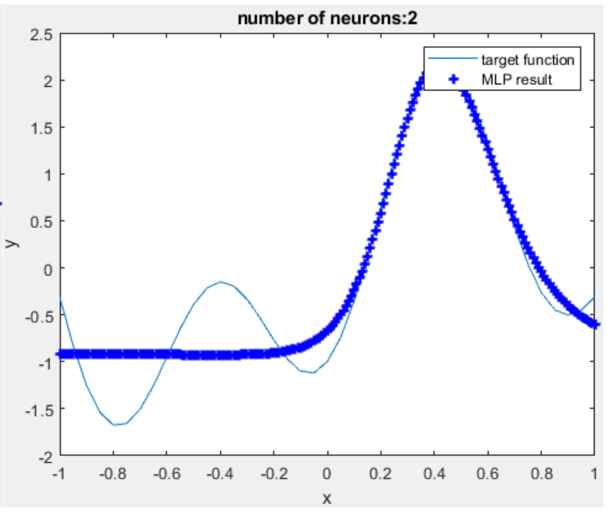
X=-3 -> y_{gt}=0.8090; y_{pred}= 3.2456

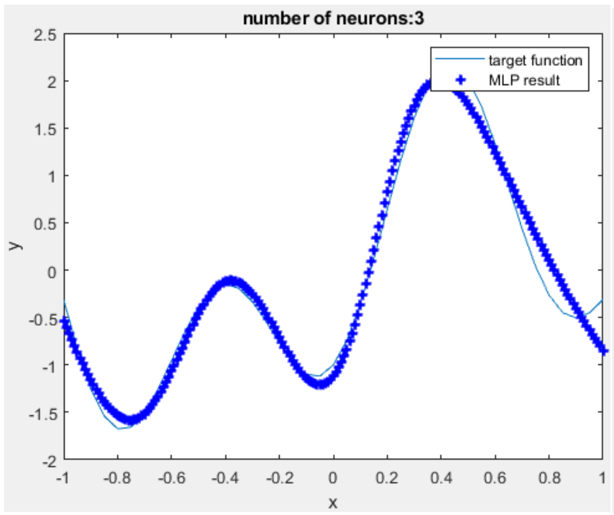
X=+3 -> y_{gt}=0.8090; y_{pred}= 0.5992

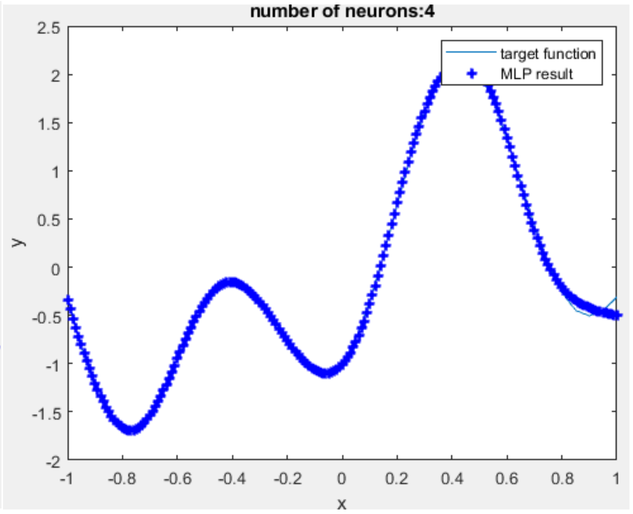
Then we can state the MLP can't make reasonable predictions outside of the domain of the training input.

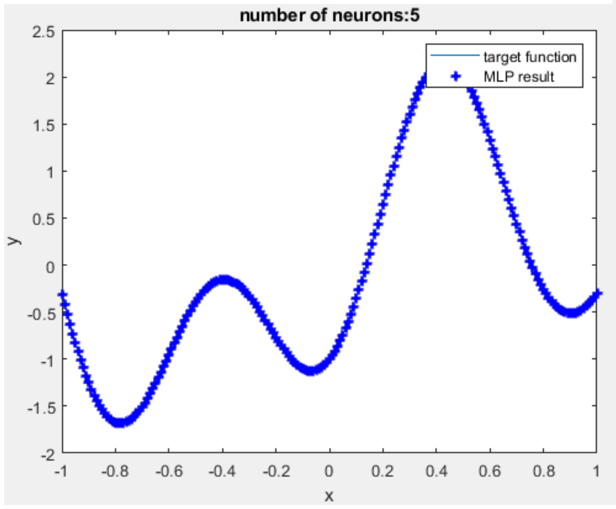
(3) Batch mode with trainbr algorithm

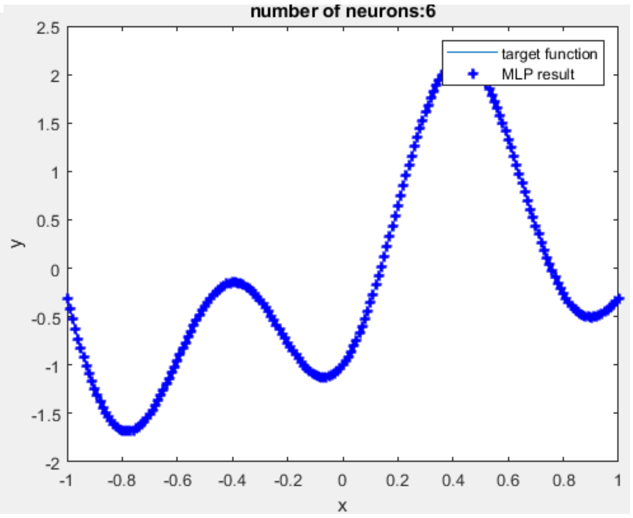


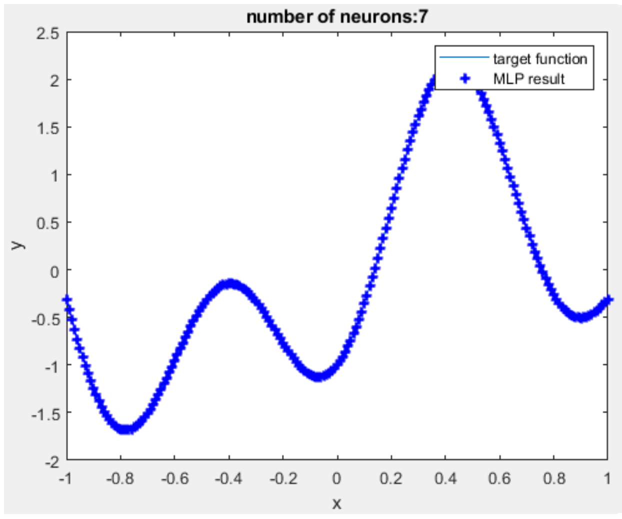


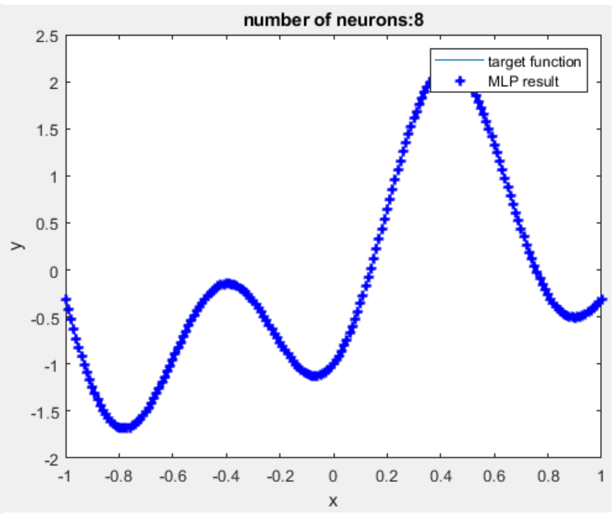

number of neurons:3
target function
MLP result
x
y

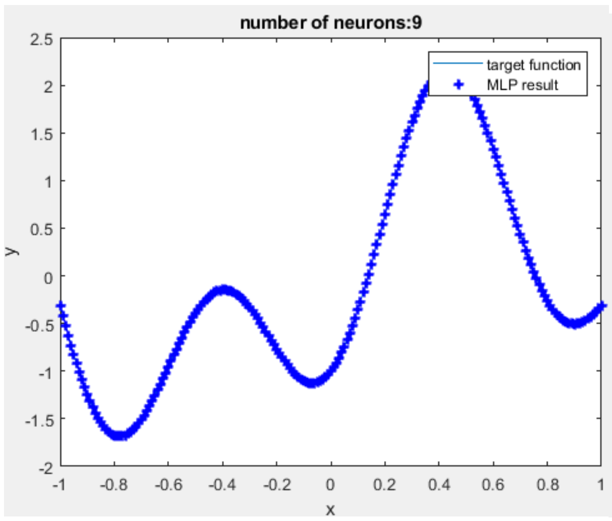

number of neurons:4
target function
MLP result
x
y

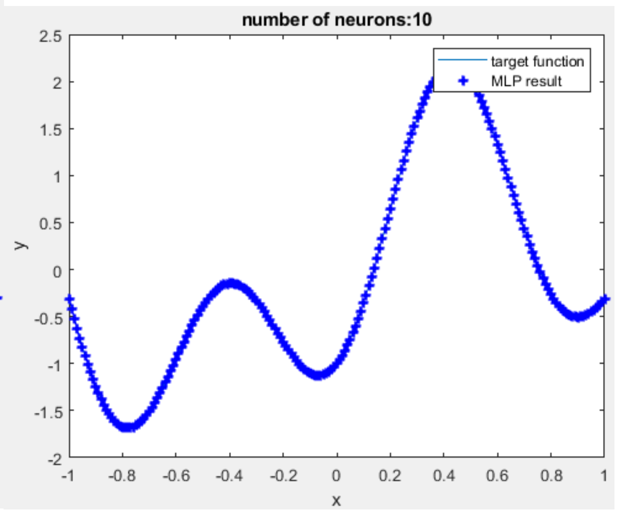

number of neurons:5
target function
MLP result
x
y

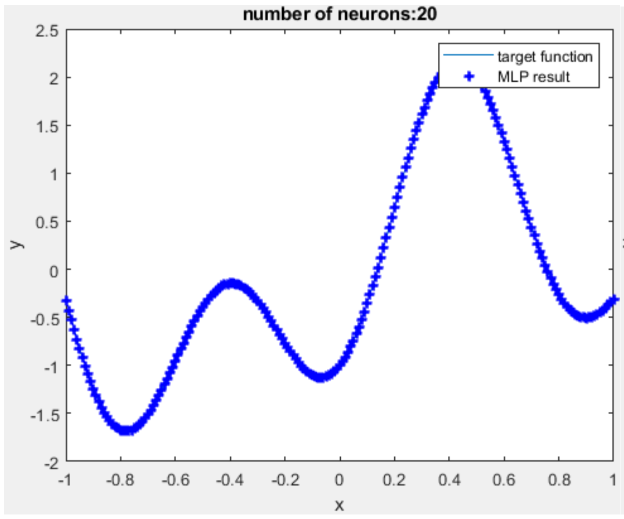

number of neurons:6
target function
MLP result
x
y

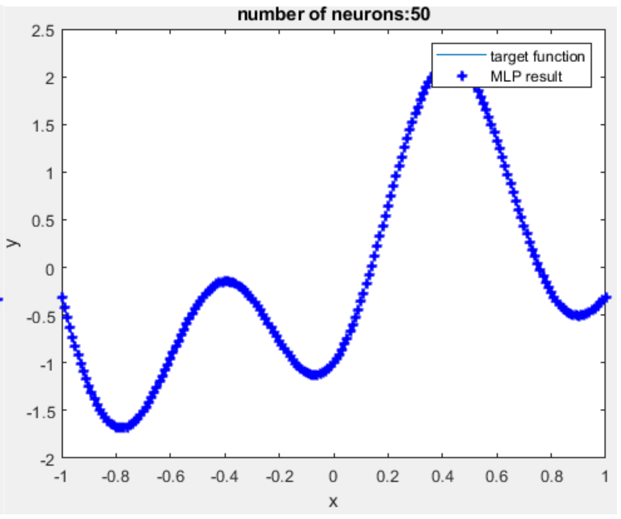

number of neurons:7
target function
MLP result
x
y


number of neurons:8
target function
MLP result
x
y









Summarized the results for trainbr in table:

	Under-fitting	Proper fitting	Over-fitting
N	1-4	5-10, 20, 50	None

The minimal number of hidden neurons is 5, which is not consistent with the guideline given in the lecture slides. The trainlr can have a better performance compared with trainlm and there is no over fitting situation for this algorithm.

The outputs of MLP when x=-3 and +3:

X=-3 -> y_{gt}=0.8090; y_{pred}= 10.4860

X=+3 -> y_{gt}=0.8090; y_{pred}= -0.0876

Then we can state the MLP for trainbr still can't make reasonable predictions outside of the domain of the training input.

```
clc
clear all;
```

```
%% sampling points in the domain of [-1,1]
train_x = -1:0.05:1;
%% generating training data, and the desired outputs
train_y = 1.2 * sin(pi*train_x) - cos(2.4*pi*train_x);
%% specify the structure and learning algorithm for MLP
for n = [5]
    disp(n);
    net = fitnet(n,'trainbr');
    net.layers{1}.transferFcn = 'tansig';
    net.layers{2}.transferFcn = 'purelin';
    net = configure(net,train_x,train_y);
    net.trainparam.lr=0.01;
    net.trainparam.epochs=10000;
    net.trainparam.goal=1e-8;
    net.divideParam.trainRatio=0.8;
    net.divideParam.valRatio=0.05;
    net.divideParam.testRatio=0.15;
    %% Train the MLP
    [net,tr]=train(net,train_x,train_y);
    %% Test the MLP, net_output is the output of the MLP, ytest is the desired ou
tput.
    xtest=-1:0.01:1;
    xtest_1=[-3, 3]
    net_output=sim(net,xtest);
    net_output_1=sim(net,xtest_1)
    % Plot out the test results
    figure
    plot(train_x, train_y)
    hold on;
    plot(xtest,net_output(1, :),'b+', 'LineWidth', 2);
    legend('target function', 'MLP result')
    title(['number of neurons:', num2str(n)])
    xlabel('x')
    ylabel('y')
    hold off
end
```

Q3

Matric number is A0206944B. Group No: mod(44, 4) +1 = 1 (open country Vs Highway)

Data preparation:

```
clc; clear;
cur = pwd
train_path = strcat(pwd, '\group_1\group_1\train\');
val_path = strcat(pwd, '\group_1\group_1\val\');

[train_data, train_label] = dataprepartion(train_path);

[val_data, val_label] = dataprepartion(val_path);
Train_dataset = vertcat(train_data, train_label);
save('Train_dataset_pca.mat', 'Train_dataset');
Val_dataset = vertcat(val_data, val_label);
save('Val_dataset_pca.mat', 'Val_dataset');

function [data, label] = dataprepartion(file_path)
    images = dir(strcat(file_path, '*.jpg'));
    num_images = size(images, 1);

    image_list = cell(1, num_images);
    label = cell(1, num_images);
    data = zeros(65536, 503);

    for i=1:num_images
        image_name = (strcat(file_path, images(i).name));
        img = double(imread(num2str(image_name)));
        image_list{:, i} = img;
        tmp = strsplit(images(i).name, {'_','.'});
        label{i} = str2num(tmp{2});
    end
    image_list_mat = cell2mat(image_list);
    label = cell2mat(label);
    data = reshape(image_list_mat, 65536, num_images);
end
```

(a) Single later perceptron.

In matlab, the perceptron function is used to implement single layer perceptron. The train and validation accuracy for 10, 50, 99 iterations are given in the below table. It would stop at iteration 99 because it reaches the performance goal. At that moment, the training accuracy is 100% and validation accuracy is 68.86%.

Iteration	Train accuracy	Validation accuracy

10	0.7097	0.6407
50	0.8390	0.6587
80	0.9006	0.6766
99	1	0.6886

The single layer perceptron can achieve 68% classification accuracy at most due to its simple structure.

```
clc; clear;

load('Train_dataset.mat');
load('Val_dataset.mat');

train_data = Train_dataset(1:end-1, :);
train_label = Train_dataset(end, :);
val_data = Val_dataset(1:end-1, :);
val_label = Val_dataset(end, :);
train_num = size(Train_dataset,2);
val_num = size(Val_dataset,2);

net = perceptron('hardlim', 'learnp');
net.trainParam.epochs= 100;
net.divideFcn = 'dividetrain';
net.performParam.regularization = 0.1;

net = train(net, train_data, train_label);

%output_train = sim(net, train_data);
output_train = net(train_data);
output_val = sim(net, val_data);
train_acc = 0;
val_acc = 0;
for i=1:train_num
    if output_train(i) == train_label(i)
        train_acc = train_acc+ 1;
    end
end
for i=1:val_num
    if output_val(i) == val_label(i)
        val_acc = val_acc+ 1;
    end
end
train_accuracy = train_acc/train_num
validation_accuracy = val_acc/val_num
```

(b) Data down sampling for (a) single layer perceptron.

We use imresize function to reduce image size and pca to images in data preparation file. Feed the new data file into (a), we can get the results.

Iteration 10:

	Train accuracy	Validation accuracy
128*128	0.6541	0.6347
64*64	0.6163	0.6048
32*32	0.7038	0.6647
PCA	0.7594	0.5988

Iteration 50:

	Train accuracy	Validation accuracy
128*128	0.8648	0.6527
64*64	0.8668	0.6228
32*32	0.7276	0.6467
PCA	0.7813	0.6407

Iteration 80:

	Train accuracy	Validation accuracy
128*128	0.9364	0.6467
64*64	0.9105	0.6766
32*32	0.7753	0.5928
PCA	0.7555	0.6407

Iteration 100:

	Train accuracy	Validation accuracy
128*128	0.9682	0.6407
64*64	0.7734	0.6228
32*32	0.8290	0.6347
PCA	0.7913	0.6527

At most 1000 or reach at performance goal:

	Train accuracy	Validation accuracy
128*128 (215)	1	0.6707
64*64 (449)	1	0.6407
32*32	0.9026	0.6467
PCA	0.7536	0.6048

32*32 and PCA reaching 100% train accuracy:

	Train accuracy	Validation accuracy
32*32	1	0.6467
PCA	1	0.6647

Evaluation:

Smaller dimension image can have similar validation accuracy (litter worse than original dimension), though train accuracy can't reach 100% running identical iterations.

Original image size will stop at epoch 99 but resized image dataset will delay the stopping epoch. Within 1000 iteration, 32*32 and PCA image dataset need more epochs to reach 100% train accuracy.

PCA:

```
train_data = transpose(Train_dataset(1:end-1, :));
val_data = transpose(Val_dataset(1:end-1, :));

% PCA
[coeff,score,latent,~,explained,mu] = pca(train_data);
% count the number of features needed above 95%,
sum_explained = 0;
idx = 0;
while sum_explained < 95
    idx = idx + 1;
    sum_explained = sum_explained + explained(idx);
end
% reduce dimension from 65536 to 247

pca_train_data = transpose(score(:, 1:idx));
train_label = Train_dataset(end, :);

% pass the trained model to have pca val
pca_val_data = transpose((val_data - mu)*coeff(:, 1:idx));
val_label = Val_dataset(end, :);
```

(3) MLP with batch node training.

Patternnet is used for this batch node training, which is a three layer MLP with n hidden neurons(1-n-1). Trainscg is selected as the training function. The learning rate is 0.01. Number of layers are tested to find the best one. The stopping criteria are 1) maximal number of iteration 200, and 2) the gradient less than 1e-6.

N	Train accuracy	Validation accuracy	Stopping epoch
1	1	0.6886	200
2	1	0.7066	200
3	1	0.6886	200
4	1	0.7246	200
5	1	0.7186	200

6	1	0.7066	200
7	1	0.7186	200
8	1	0.7216	200
9	1	0.7066	200
10	1	0.6826	200
20	1	0.7126	151
50	1	0.7246	119

From this table, we can find that the accuracy of training set can reach 100% within 200 iterations. When n is greater than 10, it will stop earlier because the gradient has already less than 1e-16. The validation accuracy remains stable around 0.72 when train accuracy is 100%.

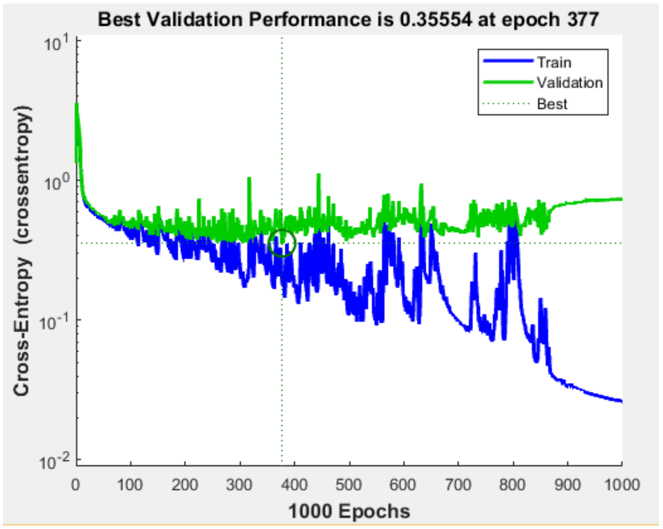
```
clc; clear;

load('Train_dataset.mat');
load('Val_dataset.mat');

train_data = Train_dataset(1:end-1, :);
train_label = Train_dataset(end, :);
val_data = Val_dataset(1:end-1, :);
val_label = Val_dataset(end, :);
train_num = size(Train_dataset,2);
val_num = size(Val_dataset,2);

n =50;
net = patternnet(n,'trainscg','crossentropy');
net.divideFcn = 'dividetrain';
net.trainParam.epochs= 2000;
net.trainParam.lr=0.01;
net.trainParam.max_fail = 2000;
net.trainParam.goal=0.00000001;
% net.performParam.regularization = 0.1;

net = train(net, train_data, train_label);

output_train = sim(net, train_data);
% output_train = net(train_data);
output_val = sim(net, val_data);

train_acc = 0;
val_acc = 0;

for i=1:train_num
    if abs(output_train(i) - train_label(i)) < 0.5
        train_acc = train_acc+ 1;
    end
```

```
end
for i=1:val_num
    if abs(output_val(i) - val_label(i)) < 0.5
        val_acc = val_acc+ 1;
    end
end

train_accuracy = train_acc/train_num
validation_accuracy = val_acc/val_num
```

(4)

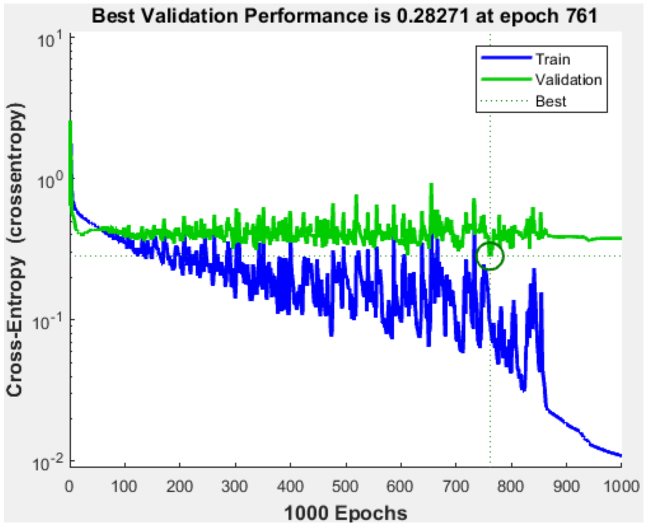
Let's take n=10 as example to explore the starting epoch of overfitting. It is can be easily done by splitting the all dataset into two parts. One is train the other one is validation dataset. I still tried two different regularization parameters 0/0.1 to check its effects. The traingd is used to replace trainsgc. Epoch number is 1000, learning rate 0.05.

Without regularization:



We can find at epoch 377 validation accuracy is the best. After that, the train still goes down however, validation start to increase. So that is overfitting.

Regularization 0.1



The best validation performance appears at epoch 761, which is latter than previous (547). Thus, we can find that the regularization can delay the overfitting.

```
clc; clear;

load('Train_dataset.mat');
load('Val_dataset.mat');

train_data = Train_dataset(1:end-1, :);
train_label = Train_dataset(end, :);
val_data = Val_dataset(1:end-1, :);
val_label = Val_dataset(end, :);
all_data = horzcat(train_data, val_data);
all_label = horzcat(train_label, val_label);

train_num = size(Train_dataset,2);
val_num = size(Val_dataset,2);

n =10;
net = patternnet(n,'traingd','crossentropy');
net.divideFcn = 'divideind';
net.divideParam.trainInd = 1:600;
net.divideParam.valInd = 601:670;

net.trainParam.epochs= 1000;
net.trainParam.lr=0.05;
net.trainParam.max_fail = 1000;
net.performParam.regularization = 0.5;
```

```
[net, tr] = train(net, all_data, all_label);
```

(5)

Sequential mode. The function is mainly based on the appendix. The regularization rate is 0.2, the training function is transcg. The max epoch is 200 which is the same as (3).

N	Train accuracy	Validation accuracy
1	0.6123	0.6108
2	0.8250	0.7066
3	0.8191	0.7485
4	0.6759	0.6048
5	0.8250	0.7485
6	0.7932	0.7545
7	0.8310	0.7605
8	0.8330	0.7066
9	0.8310	0.7126
10	0.7833	0.7246
20	0.8509	0.8084
50	0.9145	0.6886

The train accuracy can’t reach 1 while the validation is much closer to batch mode training even slightly better. In fact, the sequential mode needs more time to run but occupies smaller memory. Considering the size of dataset, parameters tuning step, and processing time, we prefer to use batch mode.

```
clc; clear;

load('Train_dataset.mat');
load('Val_dataset.mat');

train_data = Train_dataset(1:end-1, :);
train_label = Train_dataset(end, :);
val_data = Val_dataset(1:end-1, :);
val_label = Val_dataset(end, :);
train_num = size(Train_dataset,2);
val_num = size(Val_dataset,2);
train_accuracy = zeros(12);
validation_accuracy = zeros(12);
id = 1;

for n = [1:10, 20, 50]
    disp(n)
    [ net, accu_train, accu_val ] = train_seq(n, train_data, train_label, train_n
um, 0, 200);;
```

```
    output_train = sim(net, train_data);
    output_val = sim(net, val_data);

    train_acc = 0;
    val_acc = 0;

    for i=1:train_num
        if abs(output_train(i) - train_label(i)) < 0.5
            train_acc = train_acc+ 1;
        end
    end
    for i=1:val_num
        if abs(output_val(i) - val_label(i)) < 0.5
            val_acc = val_acc+ 1;
        end
    end

    train_accuracy(id) = train_acc/train_num
    validation_accuracy(id) = val_acc/val_num
    id = id + 1;
end
```

Train_seg:

```
net = patternnet(n);
    net.divideFcn = 'dividetrain'; % input for training only
    net.performParam.regularization = 0.2; % regularization strength
    net.trainFcn = 'trainscg'; % 'trainrp' 'traingdx'
    net.trainParam.epochs = epochs;
    net.trainParam.lr = 0.01;
    accu_train = zeros(epochs,1); % record accuracy on training set of each epoch
    accu_val = zeros(epochs,1); % record accuracy on validation set of each epoch
```

(6)

Scheme to improve the performance of the MLP:

a. Considering the size of dataset, parameters tuning and processing time, batch mode is preferred.

b. Training data size is very important, deciding the performance of MLP.

c. Selection of training function. Various training functions have their own impacts on accuracy and computation time.

d. Data augmentation. By resizing and dimension reduction, we can augment data to have more training features.