EE5904 Neural Networks: HW #2

A0206597U Xin Zhengfang

Q1. Rosenbrock's Valley Problem

Q1. (a)

Q1. (a). (i)

We can define the Rosenbrock's Valley function as our cost function, because we want to find minimum value of the function:

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

, where x and y are our weights of the cost function.

According to the Steepest (Gradient) descent method, we can get the update functions as follows:

$$x_{k+1} = x_k - \eta \frac{\partial E}{\partial x_k} = x_k - \eta[2(x - 1) + 400x(x^2 - y)]$$

$$y_{k+1} = y_k - \eta \frac{\partial E}{\partial y_k} = y_k - \eta[200(y - x^2)]$$

Matlab code:

```
clc
clear
% initialization
x = 0;
y = 0.5;
count = 1;
cost = (1-x)^2 + 100*(y-x^2)^2;
eta = 0.001; % learning rate
coords_h = [x,y;]; % history of x and y
cost_h = [cost]; % history of cost

% update process
while cost >= 1e-5
    count = count + 1;
    % update weights
    x = x - eta*(2*(x - 1) + 400*x*(x^2 - y));
    y = y - eta*(200*(y - x^2));
    % new cost
    cost = (1-x)^2 + 100*(y-x^2)^2;
    % record values
    coords_h = [coords_h;x,y;];
    cost_h = [cost_h;cost];
end
```

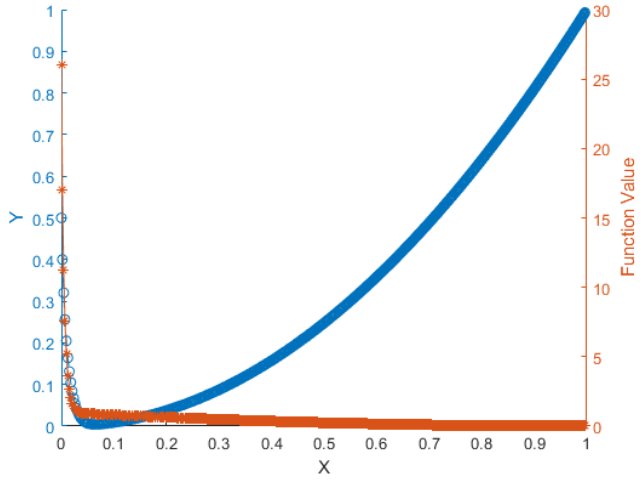

Figure 1. The trajectory of (x,y) and function value ($\eta = 0.001$)

When the cost is lower than tolerance 0.00001, it needs 9747 iterations.

Q1. (a). (ii)

The function value went to infinity. The valley is relatively too small to go in, because the step ($\eta = 0.2$) is too large.

Table I. Function values for $\eta = 0.2$

# Step	Function Value
1	26
2	17583.1200000000
3	4.90167433822695e+15
4	4.82314063251016e+50
5	4.59567939380939e+155
6	Inf
7	NaN

Q1. (b)

By using Newton's method, firstly we need to get gradient matrix and Hassian matrix:

$$g(E) = \begin{bmatrix} \frac{\partial E}{\partial x} \\ \frac{\partial E}{\partial y} \end{bmatrix} = \begin{bmatrix} 2(x-1) + 400(x^3 - xy) \\ 200(y - x^2) \end{bmatrix}$$

$$H(E) = \begin{bmatrix} \frac{\partial^2 E}{\partial x \partial x} & \frac{\partial^2 E}{\partial x \partial y} \\ \frac{\partial^2 E}{\partial y \partial x} & \frac{\partial^2 E}{\partial y \partial y} \end{bmatrix} = \begin{bmatrix} 1200x^2 - 400y + 2 & -400x \\ -400x & 200 \end{bmatrix}$$

Matlab Code:

```
clc
clear
% initialization
x = 0;
y = 0.5;
count = 1;
cost = (1-x)^2 + 100*(y-x^2)^2;
coords_h = [x,y;]; % history of x and y
cost_h = [cost]; % history of cost

% update process
while cost >= 1e-5
    count = count + 1;
    % update weights
    g = [(2*(x - 1) + 400*(x^3 - x*y)); 200*(y-x^2)];
    H = [1200*x^2 - 400*y + 2,-400*x; -400*x,200];
    w = [x;y]-H^(-1)*g;
    x = w(1);
    y = w(2);
    % new cost
    cost = (1-x)^2 + 100*(y-x^2)^2;
    % record values
    coords_h = [coords_h;x,y;];
    cost_h = [cost_h;cost];
end
```

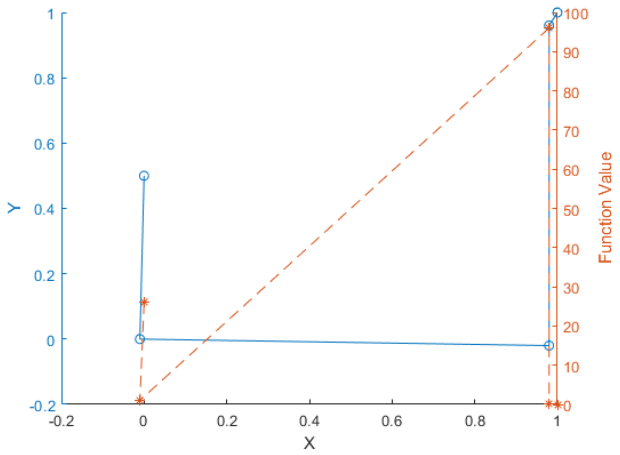

Figure 2. The trajectory of (x,y) and function value for Newton's Method

It takes 6 steps when the cost equals to 0.

Q2. Function Approximation

Q2. (a)

Use the sequential mode with BP algorithm and experiment with the following different structures of the MLP: 1-n-1 (where n = 1, 2, ..., 10, 20, 50). I use the fitnet in Matlab with "trainscg" training function. The number of epochs is 200. More specific settings see my Matlab code as follows:

```
function [net,accu_train] = my_mlp(n,x,y,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.
%   x: vector of (1, train_num), inputs.
%   y: vector of (1, train_num), desired outputs of inputs.
%   train_num: int, number of trianing data.
%   epochs: int, number of training epochs.
%
% Returns:
%
%   net: object, containg trained network.
%   accu_train: vector of (epochs, 1), containg the accuracy on training
%               set of each epoch during training.

    % 1. Change the input to cell array form for sequential training
    x_c = num2cell(x, 1);
    y_c = num2cell(y, 1);

    % 2. Construct and configure the MLP
    net = fitnet(n,'trainscg');% I use Scaled conjugate gradient backpropagation.

    net.divideFcn = 'dividetrain'; % input for training only
    net.divideParam.trainRatio = 1; % all train
    net.divideParam.testRatio = 0; % no test
    net.divideParam.valRatio = 0; % no val
    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, x_c(:,idx), y_c(:,idx));
        pred_train = round(net(x(:,1:train_num))); % predictions on training set
        accu_train(i) = 1 - mean(abs(pred_train-y(1:train_num)));
    end
end
```

The results show in Fig. 3.

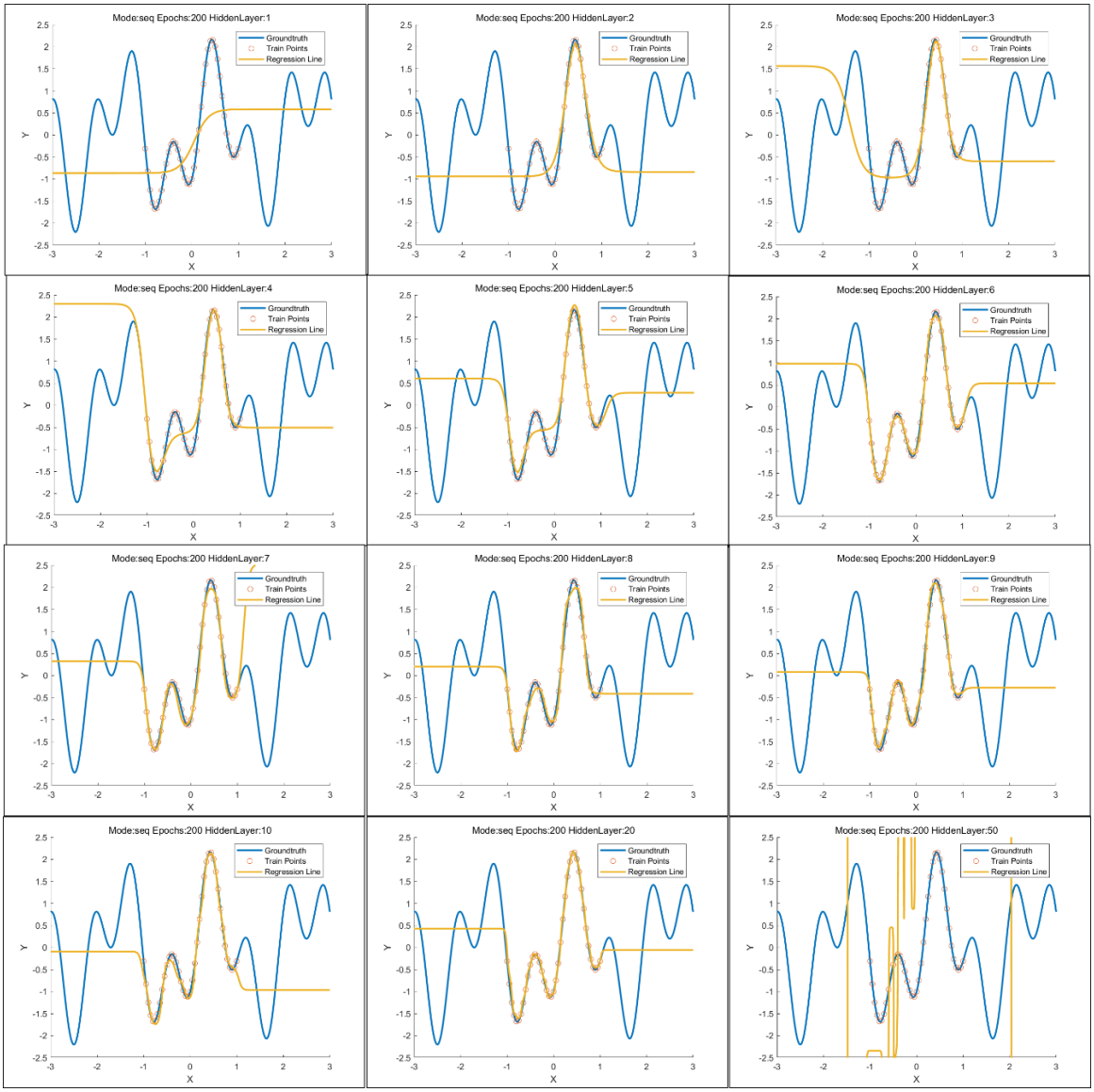

Figure 3. Sequential training of MLP with different hidden layers

The fitting results show in the Table II.

Table II. Sequential training fitting results with different hidden layers

Hidden Layers	1	2	3	4	5	50	6	7	8	9	10	20
Fitting Results	Under-fitting						Proper Fitting					

From the Table II, we can conclude the minimum number of hidden layers is **6**. According the guideline in slides: *Estimate the minimal number of line segments (or hyperplanes in high dimensional cases) that can construct the basic geometrical shape of the target function, and use this number as the first trial for the number of hidden neurons of the three-layered MLP.*

Why hidden layers = 50 is under-fitting? Because the number of weights is too large for sequential learning, update processing only relaying on one point can not find good update direction.

We need **six** basic shapes of sigmoid to construct the function, where $x = [-1,1]$, consistent to our experiment.

Moreover, the MLP cannot fit the range out of $[-3,-1) \cap (1,3]$ from the experiment.

Q2. (b)

Following the net template in slides, I have my own net as follows:

Matlab code:

```
% The code template is copied from the slides
%Matlab code
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);
for i = [1:10,20,50]
    % specify the structure and learning algorithm for MLP
    net = feedforwardnet(i,'trainlm');
    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-9;
    net.divideParam.trainRatio=1.0;
    net.divideParam.valRatio=0.0;
    net.divideParam.testRatio=0.0;
    % 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 output.
    test_x = -3:0.01:3;
    test_y = 1.2*sin(pi*test_x) - cos(2.4*pi*test_x);

    net_output=sim(net,test_x);
    % Plot out the test results
    fig = figure();
    hold on
    plot(test_x,test_y,'-','LineWidth',2)
    scatter(train_x,train_y)
    plot(test_x,net_output,'-','LineWidth',2)
    ylim([-2.5 2.5])
    legend('Groundtruth','Train Points','Regression Line')
    title(sprintf('Mode:Batch Epochs:%d HiddenLayer:%d',tr.num_epochs,i))
    ylabel('Y')
    xlabel('X')
    hold off
    % save image
```

```
    saveas(fig,sprintf('images/HiddenLayer%02d.png',i));
end
```

Train function is ‘trainlm’, and epochs is adaptive. The results show in Figure 4.

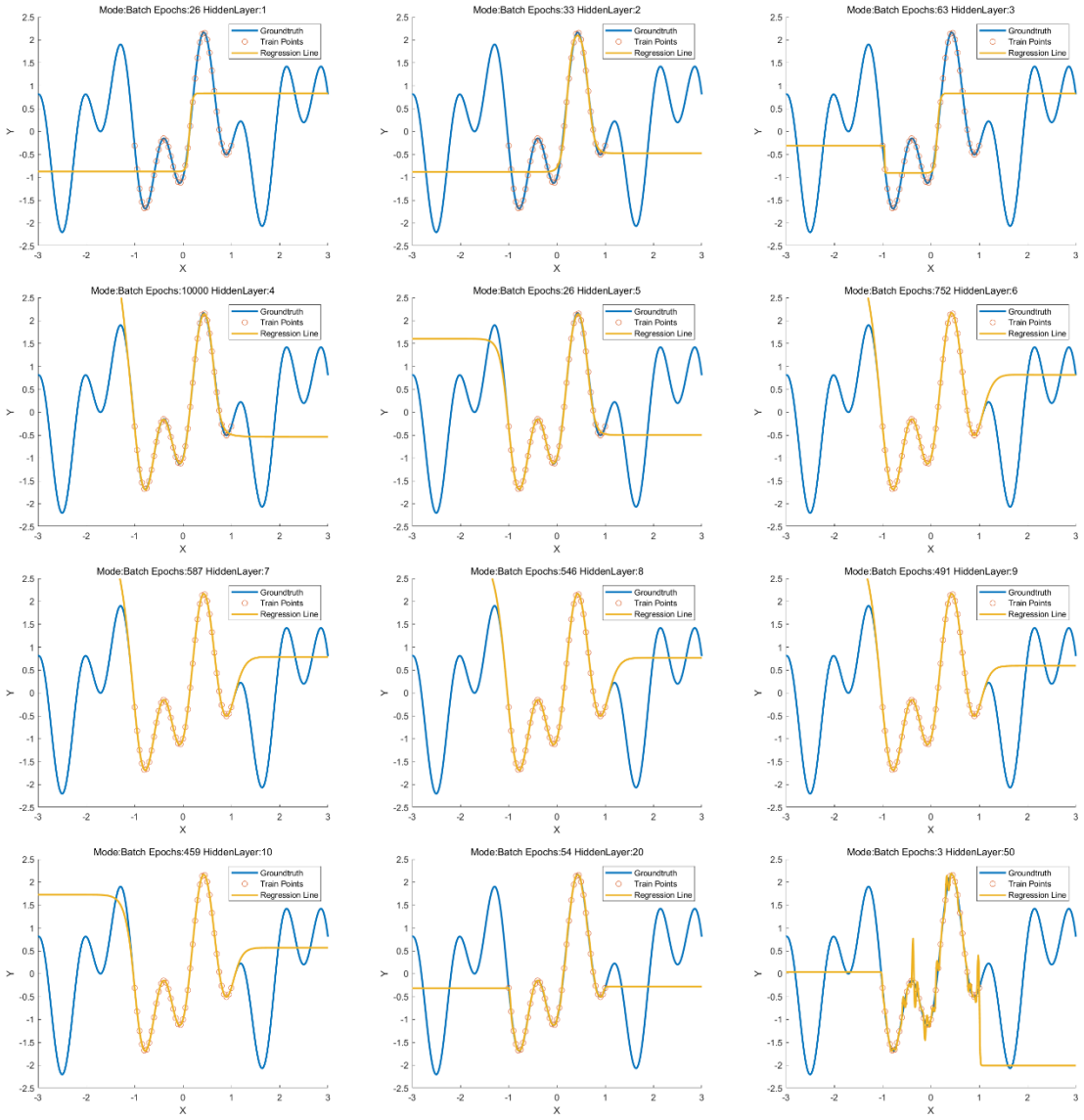

Figure 4. Batch training of MLP with different hidden layers and trainlm

Table III. Sequential training fitting results with different hidden layers

Hidden Layers	1	2	3	4	5	6	7	8	9	10	20	50
Fitting Results	Under-fitting					Proper Fitting						Over-fitting

My results are consistent to the guideline and minimum hidden layer is **6**.

Some of nets can predict a correct little range out of $[-1,1]$. However, generally they don't have the ability to predict correct value out of $[-1,1]$.

Q2. (c)

After changing only one-line code in Q2. (b), $net = feedforwardnet(i,'trainlm')$ to $net = feedforwardnet(i,'train')$, we can get the results in Figure 5.

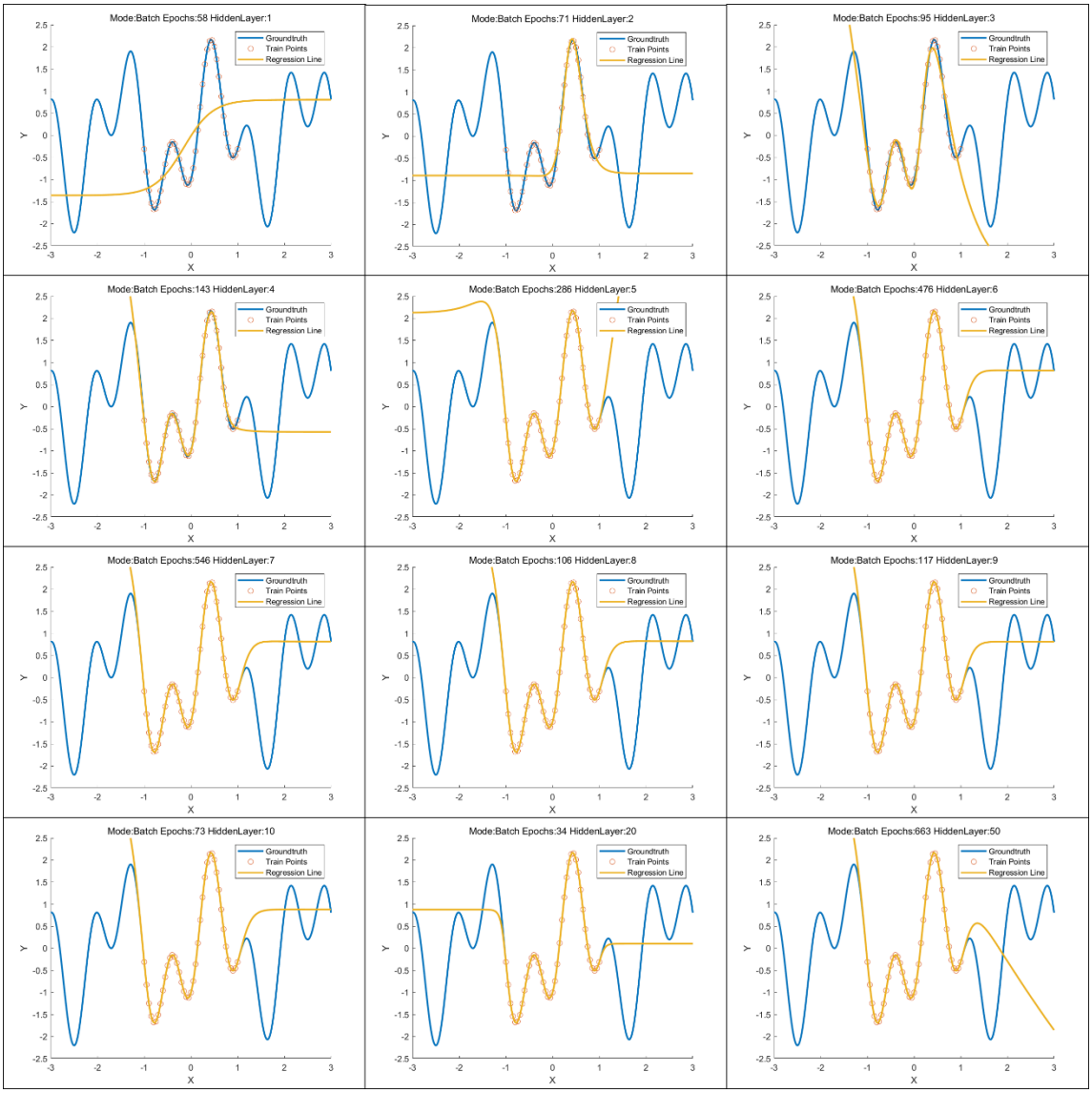

Figure 5. Batch training of MLP with different hidden layers and trainbr

Table IV. Sequential training fitting results with different hidden layers

Hidden Layers	1	2	3	4	5	6	7	8	9	10	20	50
Fitting Results	Under-fitting				Proper Fitting							

The minimum number of hidden layers is **5**, which is not consistent to the guideline. But 5 is also near to 6, choosing initial number of hidden layers by the guideline is still a good choice.

Why there is no over-fitting?

Because “trainbr” automatically add weight regularization during the training process. It minimizes a combination of squared errors and weights.

It is same as the Q2. (b). Some of nets can predict a correct little range out of $[-1,1]$. However, generally they don’t have the ability to predict correct value out of $[-1,1]$.

Q3 Scene Classification (Group ID = mod (97,4) + 1 = 2)

Q3. (a).

Matlab code:

```
clc
clear
% load data
%   train_images (256*256,samples)
%   train_labels (1,samples)
%   val_images (256*256,samples)
%   val_labels (1,samples)
load('SceneData')
% specify the structure and learning algorithm for MLP
net = perceptron();
net = configure(net,train_images,train_labels);
net.trainparam.lr=0.01;
net.trainparam.epochs=1000;
net.trainparam.goal=1e-9;
net.divideParam.trainRatio=1.0;
net.divideParam.valRatio=0.0;
net.divideParam.testRatio=0.0;
% Train the MLP
[net,tr]=train(net,train_images,train_labels);
% accuracy
pred_train = net(train_images);
accu_train = 1 - mean(abs(pred_train-train_labels));
pred_val = net(val_images);
accu_val = 1 - mean(abs(pred_val-val_labels));
fprintf('accu_train: %.02f%%\n',accu_train*100)
fprintf('accu_val: %.02f%%\n',accu_val*100)
```

```
accu_train: 100.00%
accu_val: 69.28%
```

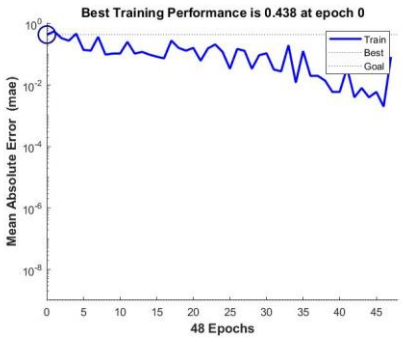

Figure 6. Performance of perceptron

From the result of 100% on training set and 69.28% on validation set, we can see the perceptron is obvious over-fitting.

Q3. (b).

Table V. Accuracies with different dimensionality

Size	256x256	128x128	64x64	32x32	PCA_298
Train Accuracy	100%	100%	100%	100%	100%
Validation Accuracy	69.28%	66.87%	71.69%	70.48%	71.69%

I use pca to capture >= 95% total variance. Then, I get 298 dimensionalities. From Table V., 298 dimensionalities of PCA still can achieve a good performance. Moreover, the large dimensionality not always does well. PCA is also a powerful dimensionality reduction tool comparing with the same performance in 64×64.

Q3. (c).

The hidden size of MLP is 298, because we know that 298 dimensionality has the ability to achieve a good performance from Q3. (b) and input of image size is 64×64. My training function is Gradient descent w/momentum & adaptive lr backpropagation to accelerate the training process. The number of epochs is adaptive.

Matlab code:

```
clc
clear all;
% load data
%   train_images (64*64,samples)
%   train_labels (1,samples)
%   val_images (64*64,samples)
%   val_labels (1,samples)
load('SceneData')

% specify the structure and learning algorithm for MLP
net = patternnet(298,'traingdx');
net = configure(net,train_images,train_labels);
net.trainparam.lr=0.01;
```

```
net.trainparam.epochs=10000;
net.trainparam.goal=1e-6;
net.divideParam.trainRatio=1.0;
net.divideParam.valRatio=0.0;
net.divideParam.testRatio=0.0;
% Train the MLP
[net,tr]=train(net,train_images,train_labels);
% accuracy
pred_train = net(train_images);
accu_train = 1 - mean(abs(pred_train-train_labels));
pred_val = net(val_images);
accu_val = 1 - mean(abs(pred_val-val_labels));
fprintf('accu_train: %.02f%%\n',accu_train*100)
fprintf('accu_val: %.02f%%\n',accu_val*100)
```

```
accu_train: 100.00%
accu_val: 75.89%
```

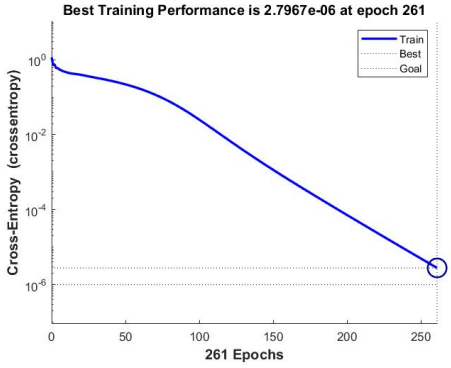

Figure 7. Performance of MLP

From the result of 100% on training set and 75.89% on validation set, we can see the perceptron is obvious over-fitting. But it is much better than perceptron.

Q3. (d).

My trained MLP in c) is overfitting.

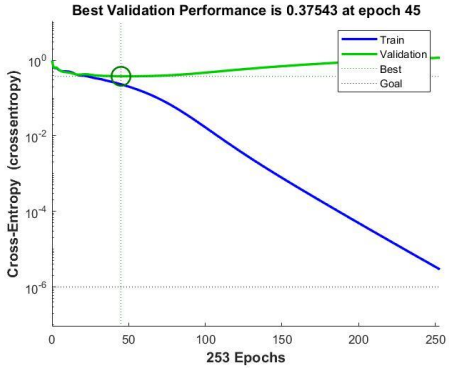

Figure 8. Performance of MLP with val. set.

From the Figure 8., the over-fitting occurs at 47 epochs.

The effect of regularization can be observed by setting:

$$net.performParam.regularization = 0,0.25,0.5,0.9$$

The results are in Table VI.

Table VI. Accuracies with different regularization

Set Category	Training Set Acc.	Val. Set Acc.
Regularization = 0	81.27%	71.03%
Regularization = 0.25	78.98%	70.63%
Regularization = 0.5	79.29%	71.13%
Regularization = 0.9	79.72%	70.82%

From the results in Table VI, the regularization is not helpful in this situation. The early stop strategy is enough. It is hard to try one good result with regularization. Therefore, it is not a proper method in this task. However, the regularization does decrease the overfitting.

Q3. (e)

The performance shows in Figure 9.

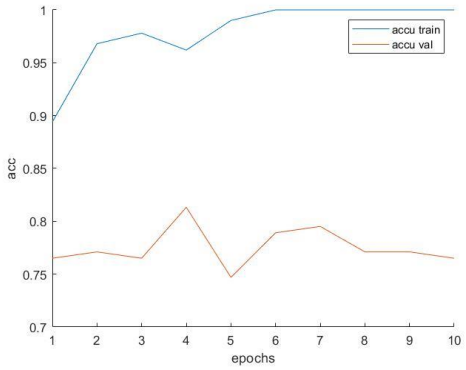

Figure 9. Sequential training performance of MLP on scene data

The final train accuracy is 100% and validation accuracy is 76.01%. It is better than best batch mode performance 75.89%. However, I still recommend batch mode, because it much faster to find correct direction and Matlab support more tools for batch training to choose a good setting. The accuracy 76.01% does not display a strong superior result. Therefore, I still choose batch mode.

Q3. (f)

1) We can use data augment to increase training set based on existing data, such as random crop, random flip, random jitter and so on. The larger dataset can reduce our network overfitting, because larger dataset means it cover more generative patterns.
2) Carefully choose the hyperparameters. It is painful but helpful! Try more possible setting, we can always find a better one than fixed setting
3) "Early stop" is a good method to find a suitable balance between training accuracy and test/validation accuracy. "Early stop" in there means we always record the best model that produce best result on validation set during the training.