Yangyang Xu 43442401

Prac 4

Q2

Process: blurring
Code Implementation: Function
Inputs: x: data. lambda: λ (radius)

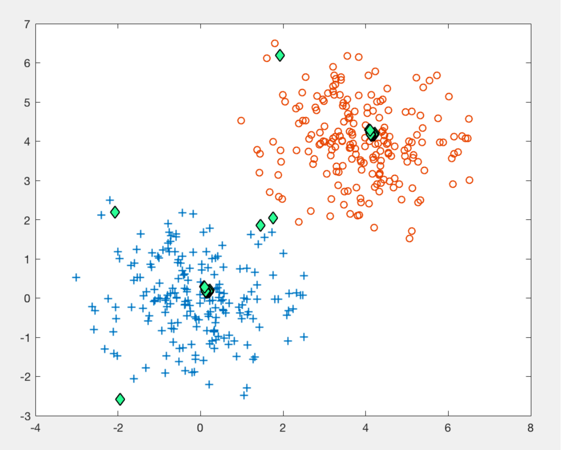
```
function prac4_test(x,lambda)
%----------------Plot X START---------------
    a = randn(200,2);
b=a+4;
c=a;
c(:,1) = 3*c(:,1);
c=c-4;
d=[a;b];
e=[a;b;c];
plot(a(:,1),a(:,2),'+');
hold on
plot(b(:,1),b(:,2),'o');
plot(c(:,1),c(:,2),'*');
 %----------------Plot X END------------------
 times = 0;
 ct = 0;
 old_Center = 0;
 this_Center = 0;
 meanlist = [];

    for i = 1:size(x,1)
        this_Center = x(i,:);
        times = times +1

        while(this_Center ~= old_Center)
            index = 0;
            s = 0;
            for j = 1:size(x,1)
                o = x(j,:);
                k = norm(o-this_Center);
                if k <= lambda
                    index = index+1;
                    s = s + o;
                end
                    s = s + o;
                end
            end
            old_Center = this_Center;
            this_Center = s ./ index;
            if isnan(this_Center)
                break;
            end
        end

        if (~ismember(this_Center,meanlist))
             ct = ct+1;
             meanlist(ct,:) = this_Center;
             plot(this_Center(:,1),this_Center(:,2),'kd','MarkerSize',8,'MarkerFaceColor',[.49 1 .63])
             hold on;
        end
    end
    hold off;
end
```

Q3

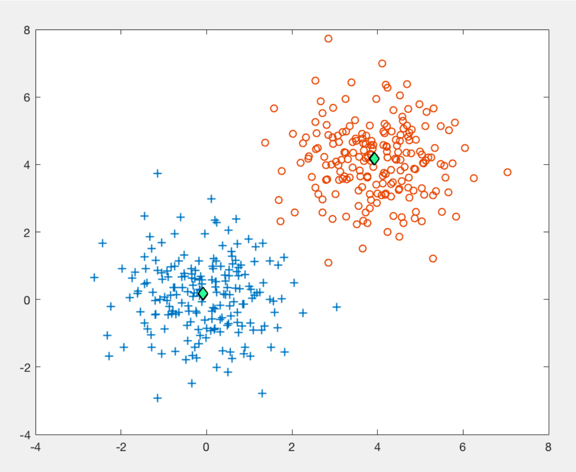
Cluster Centres: green diamond mark
Red Marks (‘o’): b
Blue Marks (‘+’): a
Yellow Marks (‘*’): c

Test Data with 3 different lambda(*Blue region is not included in this test***)**:
d=[a;b];

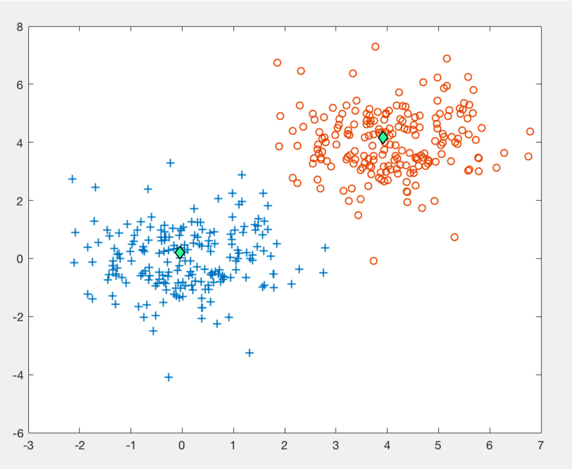
Lambda: 1



Lambda: 2

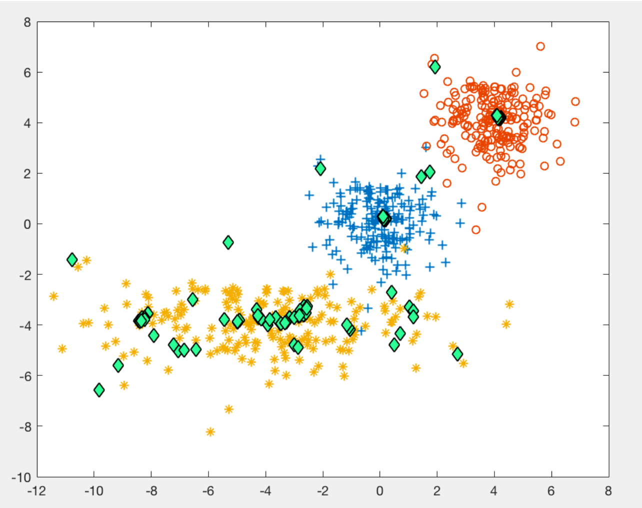


Lambda: 3

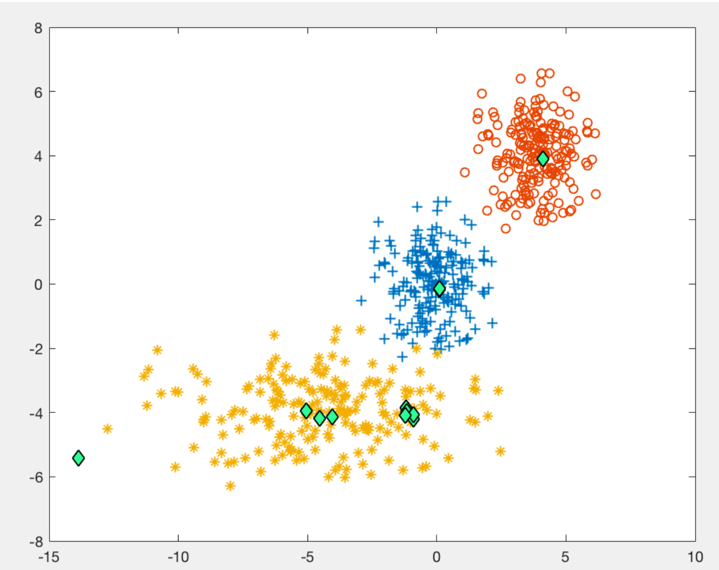


Test Data with 3 different lambda:
e=[a;b;c];

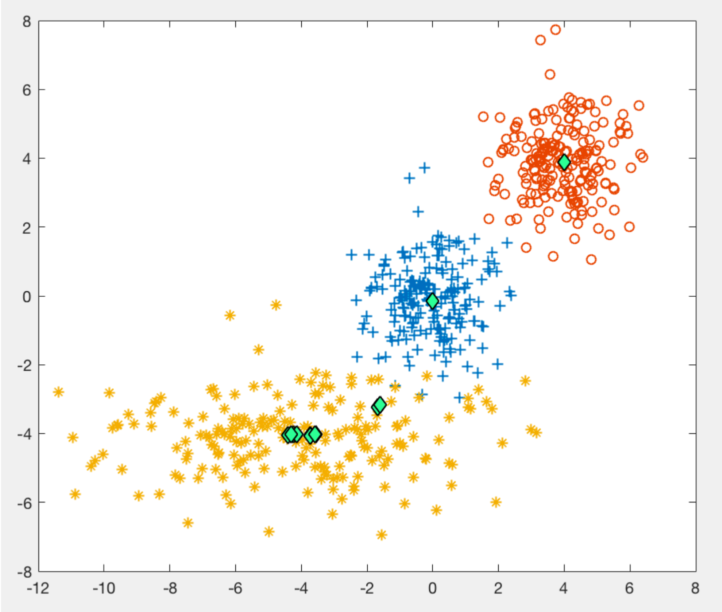
Lambda: 1



Lambda: 2



Lambda: 3



Larger lambda can get much more concentrated and less cluster centres. This effect is apparent on yellow regions.

Because larger lambda can make bigger move step to next mean centre and less iteration, thus larger lambda also affect the speed of running the mean shift algorithm.

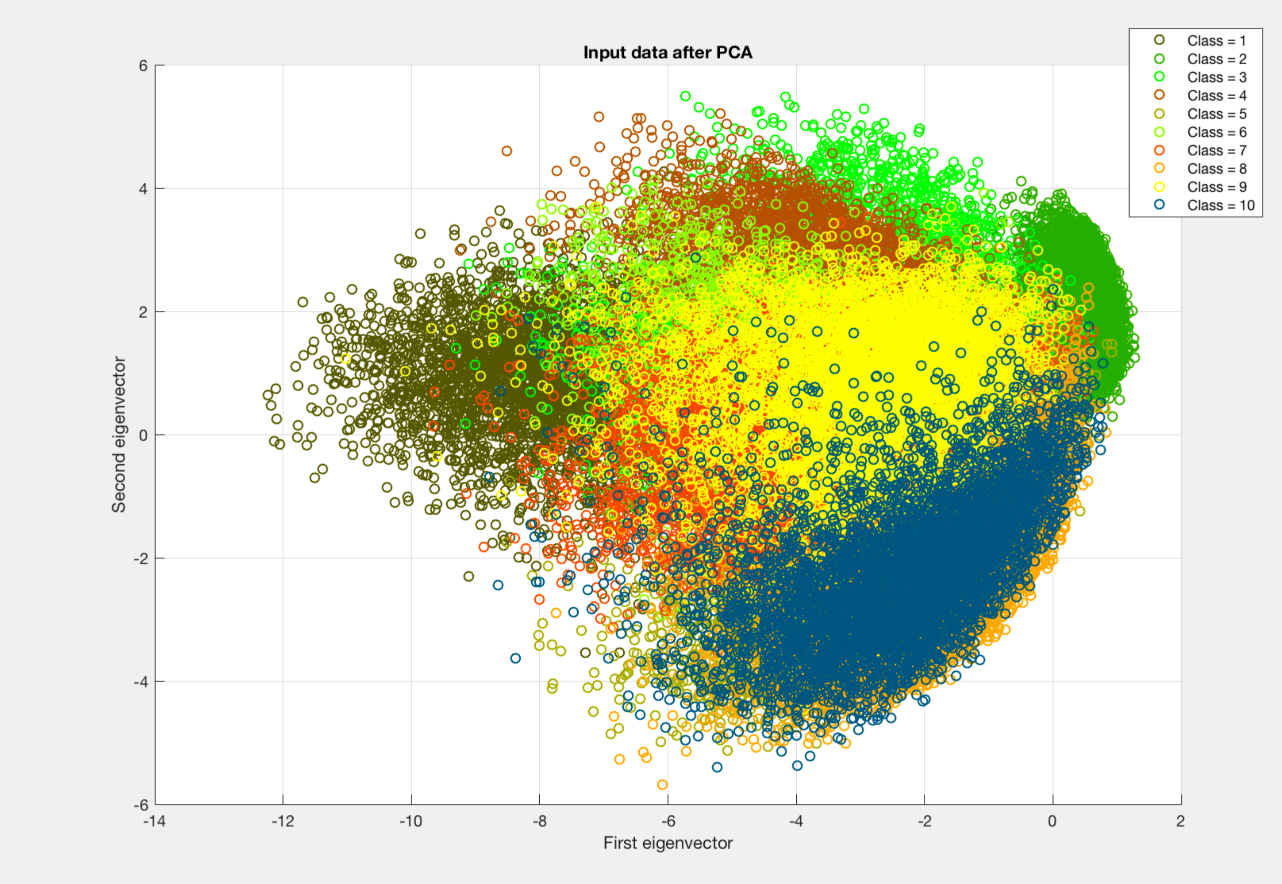
Prac 5

Q1

```
function [povResult,reducedData] = PCA(train_X,train_labels,n)
%prac5_1 Principal Component Analysis
%
%   [P, beta] = PCA(dataset1, dataset2, number)
% Inputs:
%   dataset1: Raw data (For MNIST: train_X)
%   dataset2: Data labels (For MNIST: train_labels)
%   number:   Expected classes (But only Max 64 can be displayed on
%             Figure 1)
% Outputs:
%   ReducedData: Reduced to 2D, included First & Second principal component
%   povResult:   Proportion of variance for Q2(b)
    %% Q1
    x = train_X; %n = 10;
    data = cov(x);
    [vector,value] = eig(data);
    %First principal component: 1st largest -> Dimention Reduction
    X1 = x*vector(:,end);
    %Second principal component: 2nd largest -> Dimention Reduction
    X2 = x*vector(:,end-1);
    %Reduced Data
    reducedData = [X1,X2];
    size(reducedData)
    %Added lables onto reduced data
    X1 = horzcat(X1,train_labels);
    X2 = horzcat(X2,train_labels);
    %% Q2
    c=colorcube;
    eval0 = eig(data); % inputed_raw eigenvalues
    labels = [];
    for ii = 1:n
        a = X1(X1(:,2)==ii);%x_axis
        b = X2(X2(:,2)==ii);%y_axis
        scatter(a,b,[],c(ii,:,:));
        legendInfo{ii} = ['Class = ' num2str(ii)];
        hold on;
    end

    grid on;
    legend(legendInfo);
    xlabel('First eigenvector');
    ylabel('Second eigenvector');
    title('Input data after PCA');
    %% Q3 & Q4
    sort_value = sort(eval0,'descend');
    figure,subplot(2,1,1);
    plot(sort_value,'+-');
    grid on;
    xlabel('Eigenvectors')
    ylabel('Eigenvalues')
    title('(a) Scree graph for Input data');

    hold on;
    %Proportion of variance: used feature(1->k)/inputed_raw features(1->d)
    povResult = (eval0(end)+ eval0(end-1))/sum(eval0);
    subplot(2,1,2);
    cumsum_sort_value = cumsum(sort_value);
    plot(cumsum_sort_value/sum(eval0),'+-');
    hold on;
    txt = ['Q2\_(b) =',num2str(povResult)];
    plot(povResult,'kd','MarkerSize',8,'MarkerFaceColor',[.49 1 .63])
    hoz_line = refline([0 povResult]);
    hoz_line.Color = 'r';
    text(0,povResult,txt,'HorizontalAlignment','left')
    grid on;
    xlabel('Eigenvectors')
    ylabel('Prop. of var.')
    title('(b) Proportion of variance explained');
    hold off;
end
```

Q2

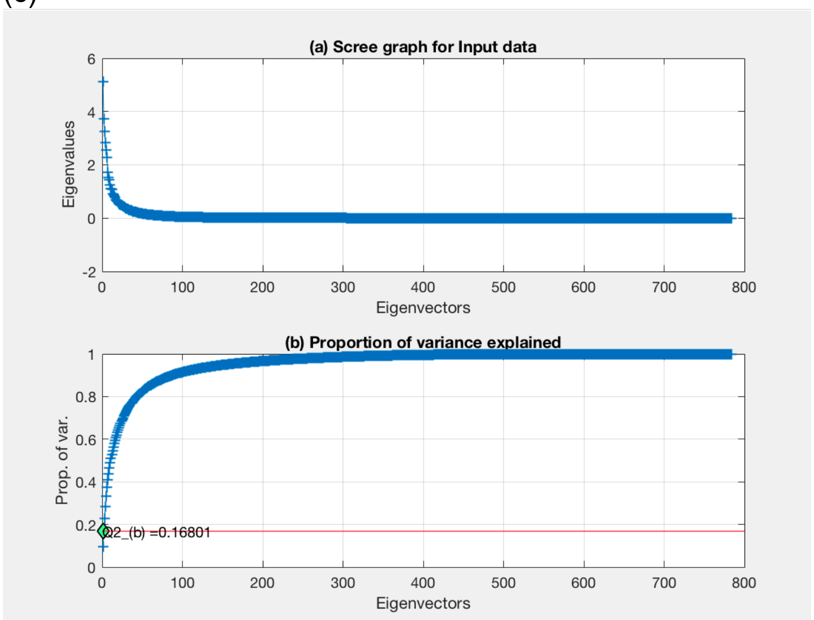
(a)



(b)

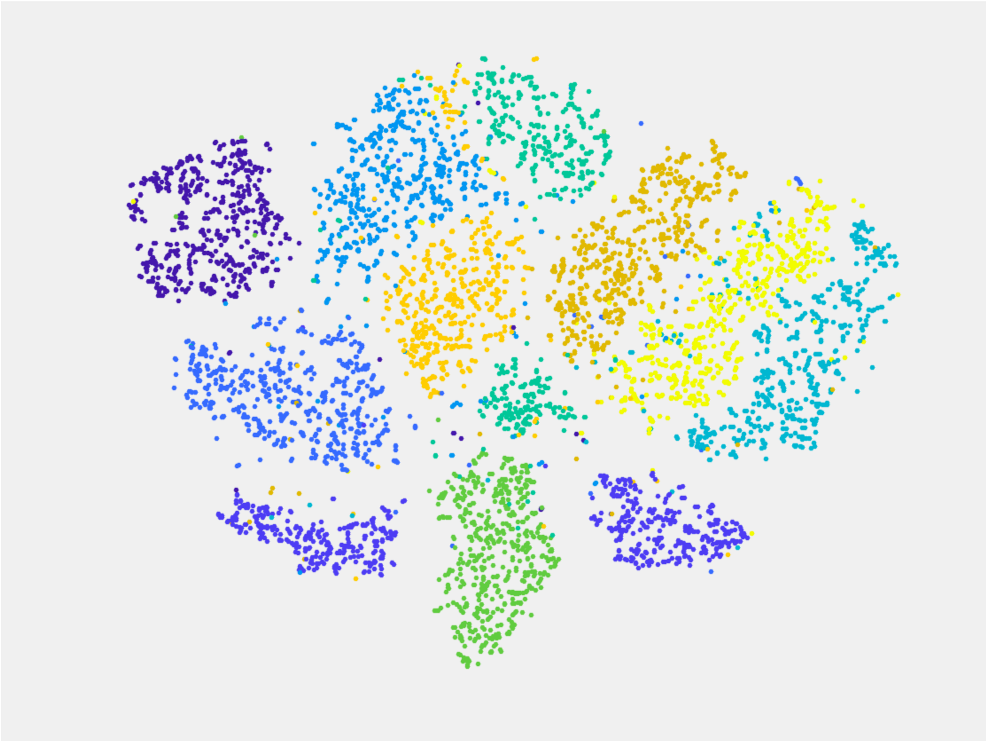
Percentage of the data variance is accounted for by the first two principal components:
0.16801(*Green diamond mark with red line on below graph*)
Code: povResult = (eval0(end)+ eval0(end-1))/sum(eval0);
The **eval0(end)** means the first principal component.
The **eval0(end-1)** means the second principal component.
The **sum(eval0)** means the total Eigenvalues got from the data.

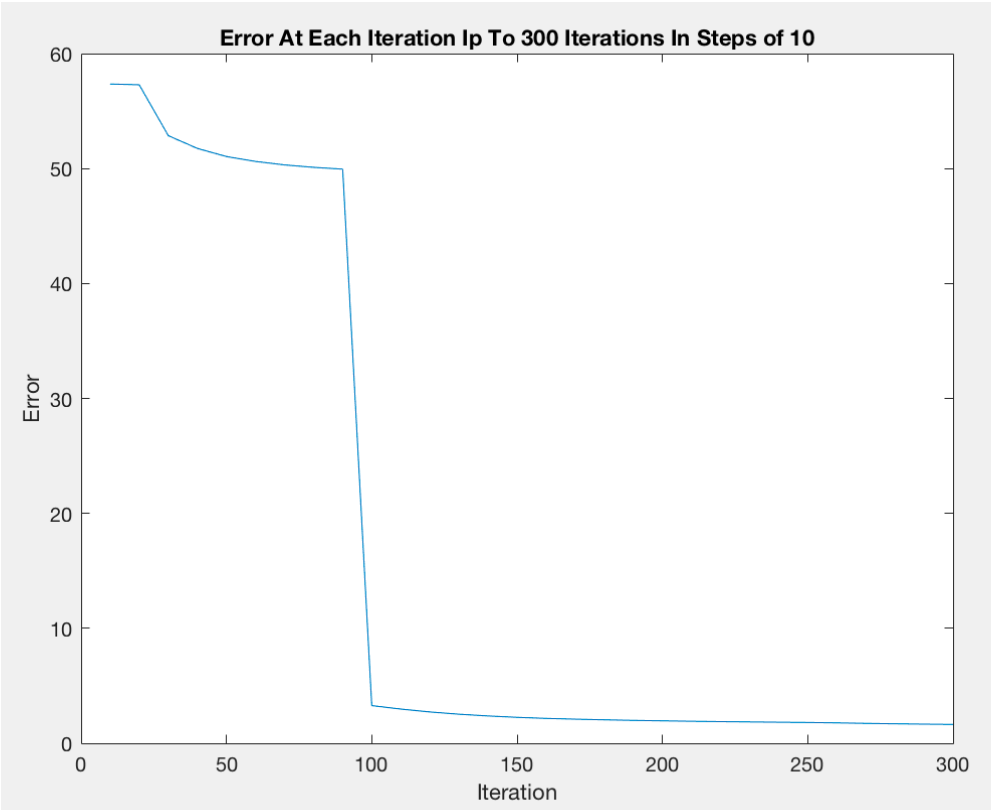
(c)



Q6

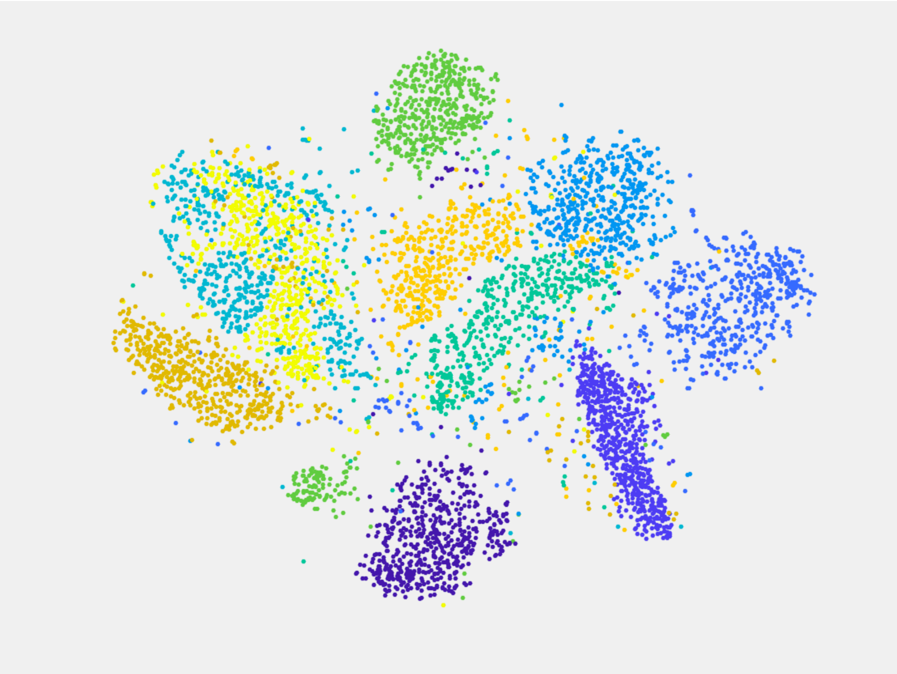
Screenshot after 300 iterations:

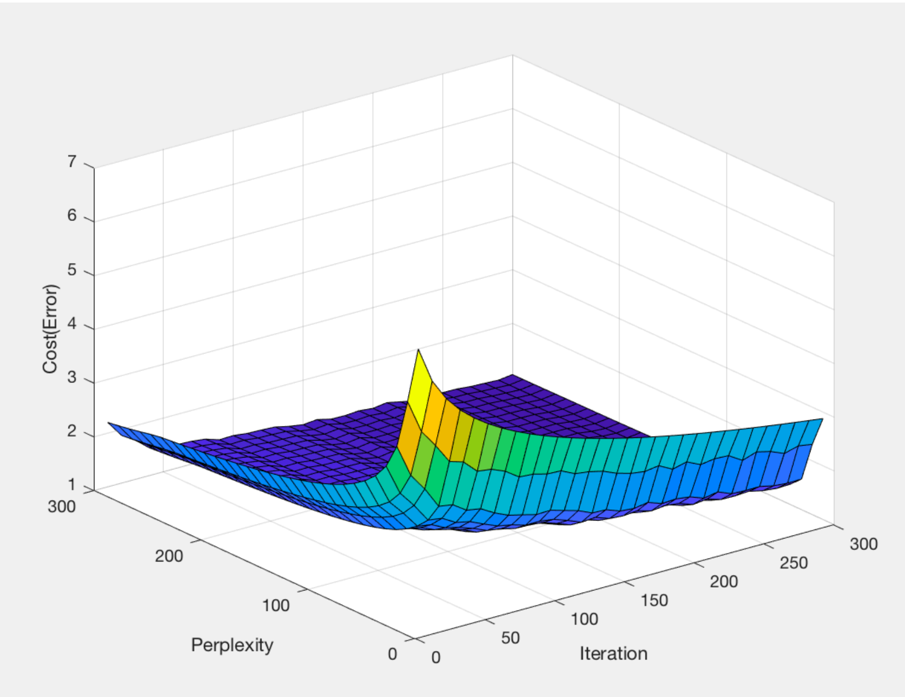




Q8

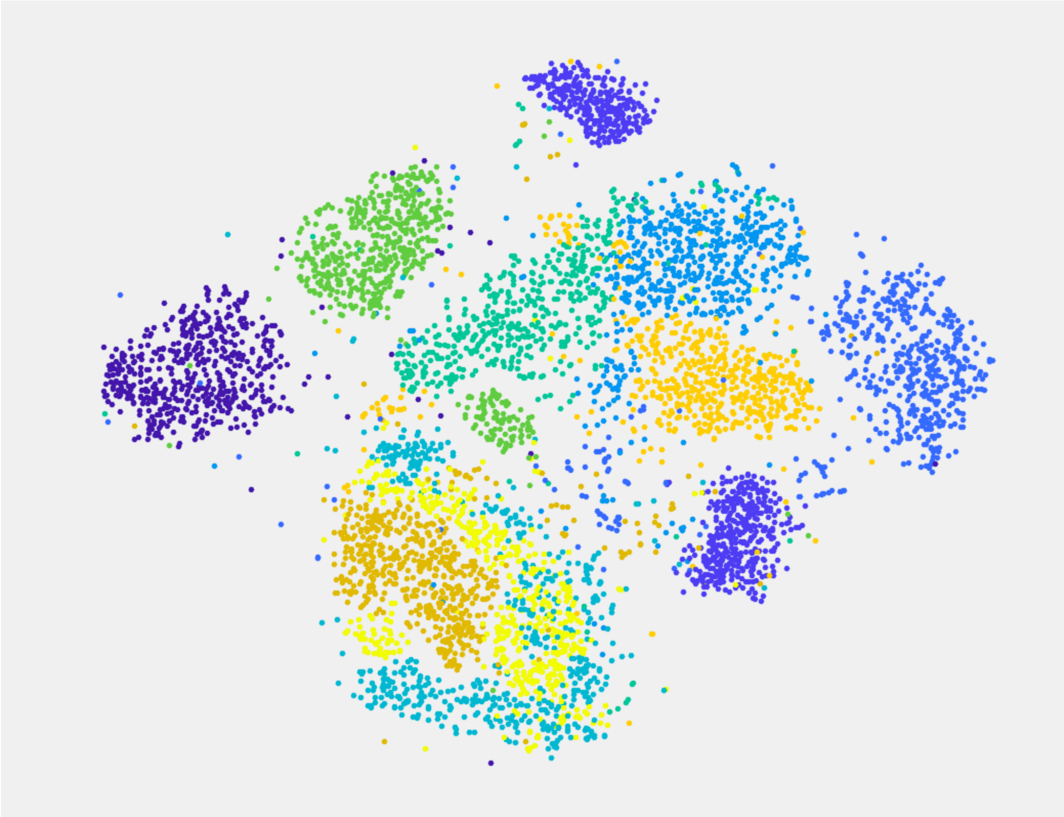
Screenshot after setting the perplexity

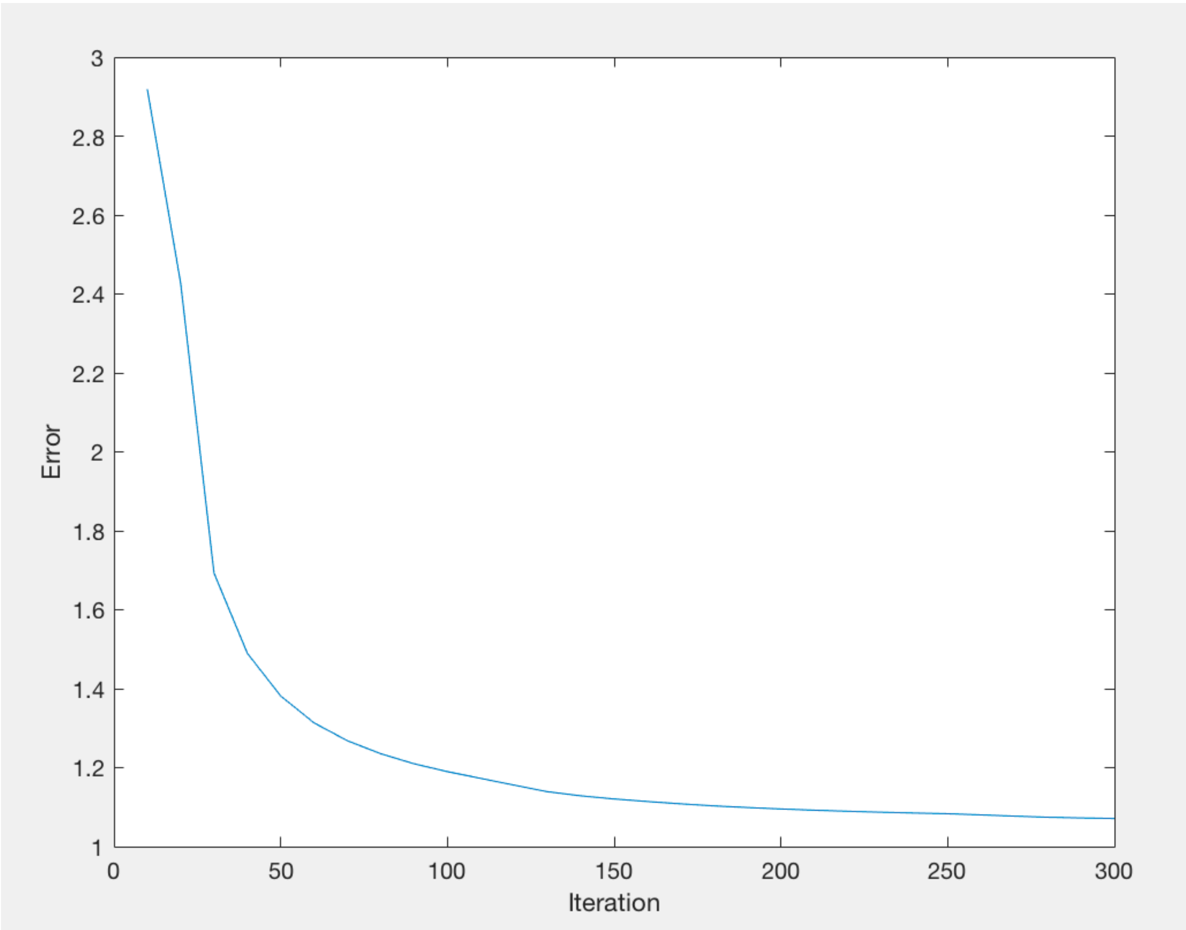




Heuristic for choosing the perplexity:

When watch at the 3D graph, when Cost becomes lower, the perplexity becomes larger, thus, the best choice of perplexity by now will be 290, and it is smaller than 300, and it uses less time to go through all possible perplexity.





The 2D visualisation is not good as Q6, Q6 got clearer classification. Perplexity is also bigger than Q6, thus perplexity really affect the classification results.

Because there are some codes deleted, search area becomes smaller, and the errors are also much smaller than Q6.

Prac 6

Q3

gd: Gradient descent backpropagation
scg: Scaled conjugate gradient backpropagation

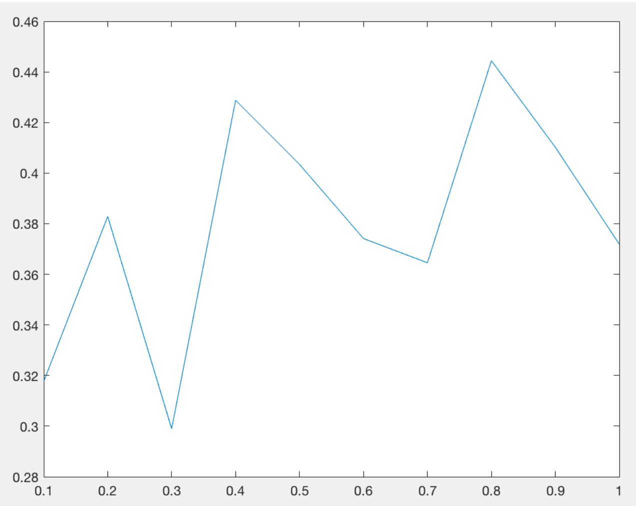
Tested learning rate, hidden layer size, epochs independently.

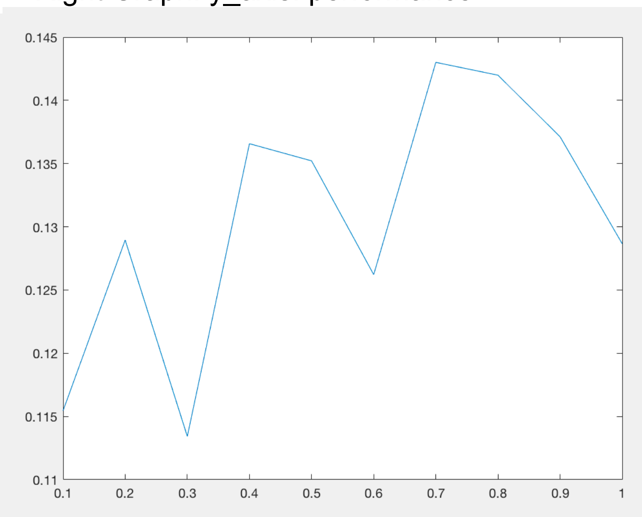
Learning Rate Range(x_axis): 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1

<Learning rate can only be positive, and the maximum is 1, default is 0.01>

Training Function: gd
Left Graph: y_axis: percentage Error

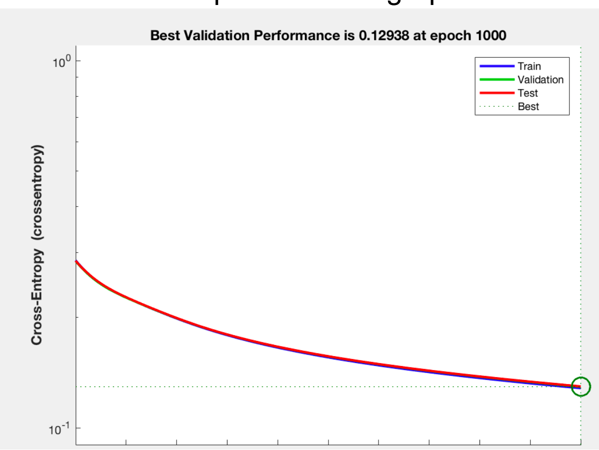
Right Graph: y_axis: performance





After plotting both results of percentage errors and performance values for each learning rate. The best learning rate in my test samples is 0.3, because both graph and training functions indicate the local minimum point at 0.3. The graphs also oscillate apparently at each 0.2 difference.

gd performance during running each learning rate shows: there is no overfitting or under fitting. And most of the performance graphs are similar to below graph.

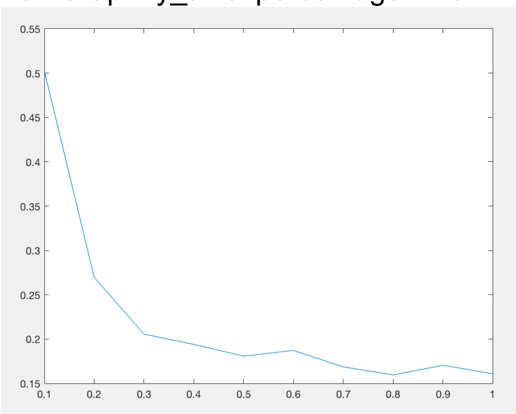


<Because only gradient descent algorithm has learning rate, thus I only tested the gd.>

Hidden Layer Size Range(x_axis): 10 20 30 40 50 60 70 80 90 100

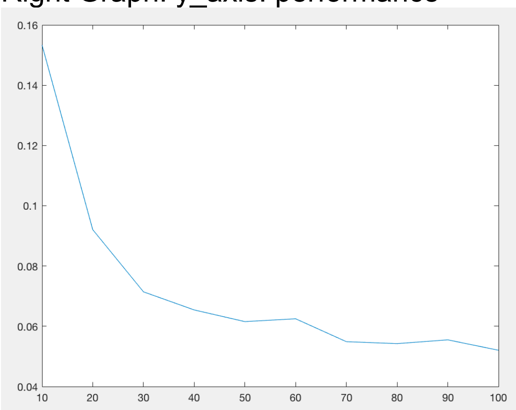
Training Function: gd

Left Graph: y_axis: percentage Error



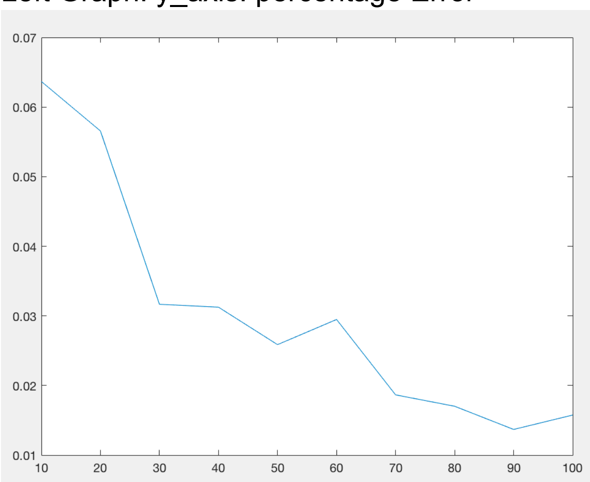
<Correct: The decimal number on above X-axis will be like below graph: 10 20 30 40 50 60 70 80 90 100>

Right Graph: y_axis: performance

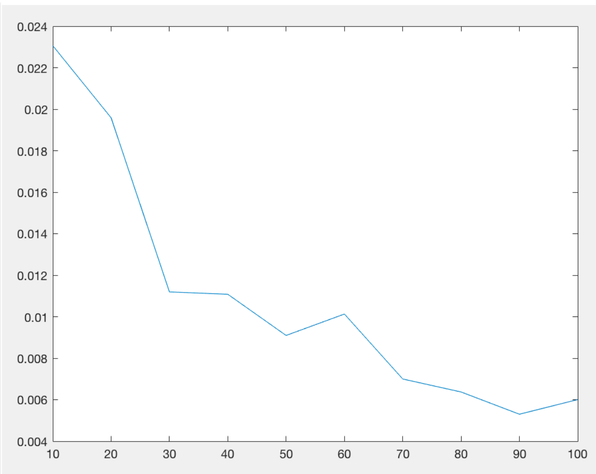


Training Function: scg

Left Graph: y_axis: percentage Error

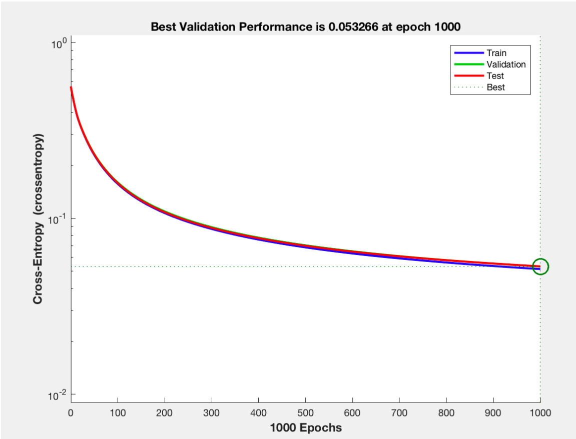


Right Graph: y_axis: performance

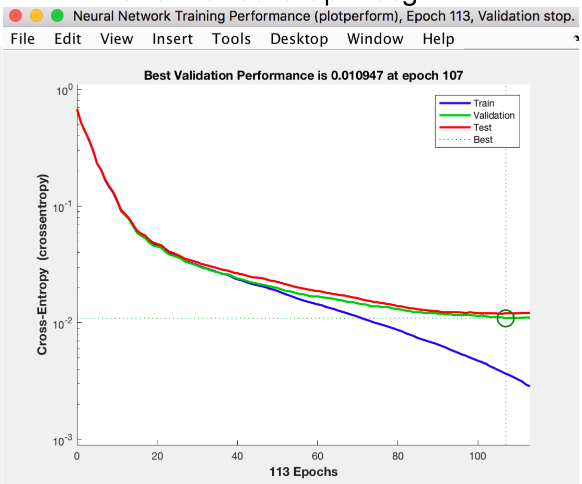


Both of **scg** and **gd** got local minimum error and performance value points around 80 to 90. Thus, in this test samples, the hidden layer set around 80 for **dg**, and around 90 for **scg** will be best.

Performance Graph: gd



Performance Graph: scg



gd performance during running each hidden layer shows: there is no overfitting or under fitting. And most of the performance graphs are similar.
scg performance during running each hidden layer shows: got overfitting in the end, when validation check reaches to 6.

Set hidden layer size to 200(apparent larger):

Training Function: gd
percentage Error = 0.1389 performance = 0.0461

Training Function: scg
percentage Error = 0.0106 performance = 0.0043

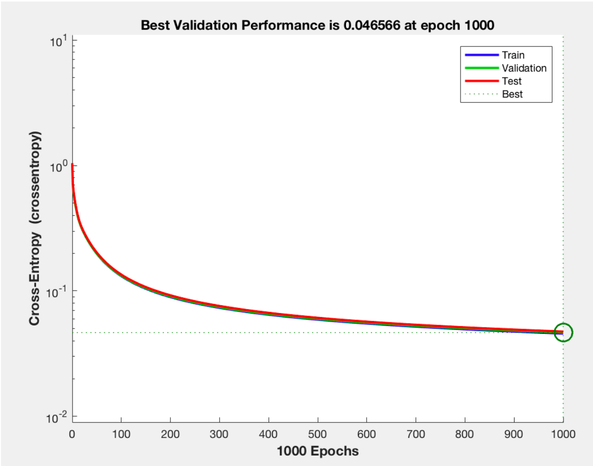
The changing of errors of gd is smoother. The changing of performance becomes not too apparent for both training functions when layer increases.

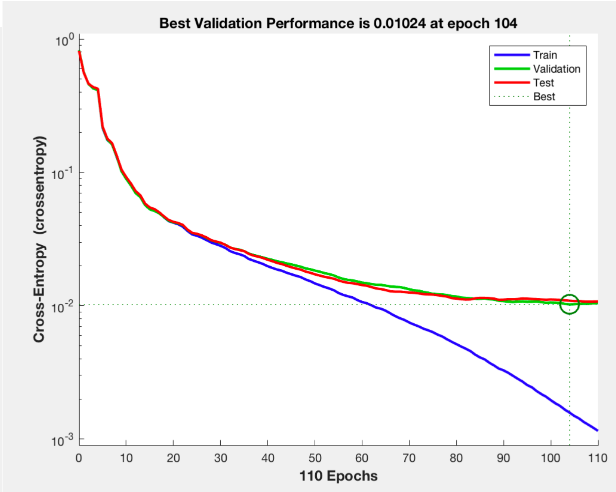
After comparing the errors and performance value of **gd** to the hidden layer which is 80 on above graph, the values are smaller, thus larger hidden layer size still affects the predict result. By now, the test sample shows 200 hidden layers could be the best choice for **gd** training function with such input data.

After comparing the errors and performance value of **scg** to the hidden layer which is 90 on above graph, the value are also about the same. Thus, 90 could be the best choice for this training function with such input data. And this error is the most lowest one for using the neural network by just using more than 90 hidden layers.

Performance Graph: gd

Performance Graph: scg



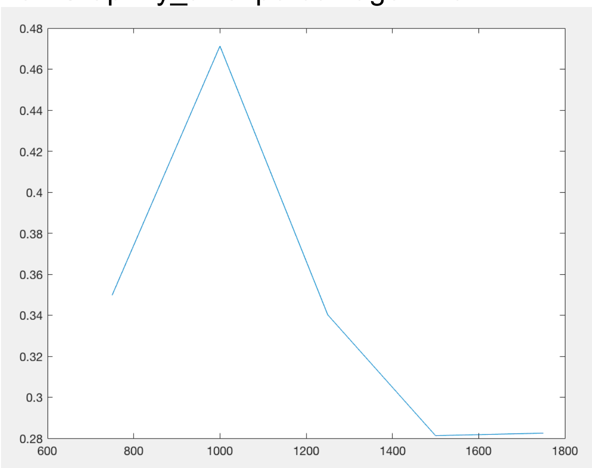
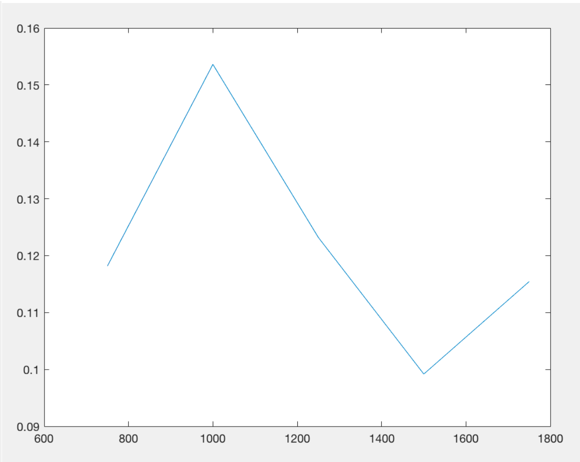


For **gd**, the performance graph is still not overfitting or undercutting.
For **scg**, the performance graph shows the overfitting in the end.

Epochs Range(x_axis): 750 1000 1250 1500 1750

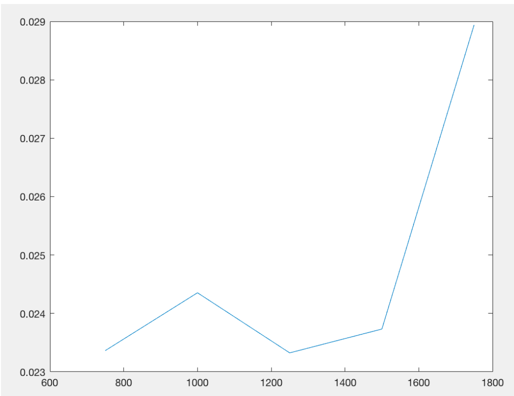
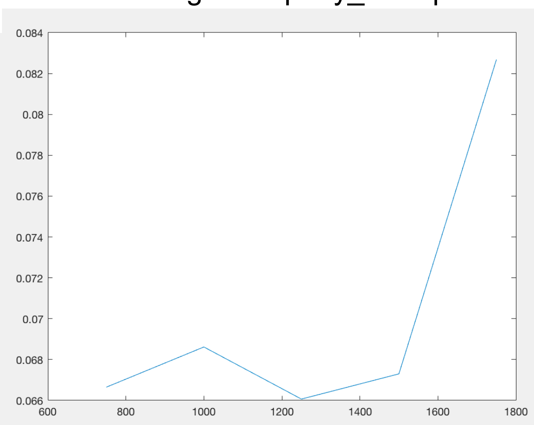
Training Function: **gd**

Left Graph: y_axis: percentage Error Right Graph: y_axis: performance

Training Function: **scg**

Left Graph: y_axis: percentage Error Right Graph: y_axis: performance

For gd, the local minimum of error is at 1500, for scg, the lowest error is at 1250. Both training function shows the error becomes local maximum at epochs is 1750.
However, increasing hidden layer size still improves the neural performance better than changing the other two features.