EE5904 NEURAL NETWORK

HOMEWORK TWO

Name: Yu Shixin

Matriculation Number.: A0195017E

Date:2019/2/18

Solution 1

(a)

According to the description of Q1, the Rosenbroch's Valley function

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

has a global minimum at $(x,y) = (1,1)$, where $f(x,y) = 0$. The figure plotted by MATLAB shows as below (Fig.1).

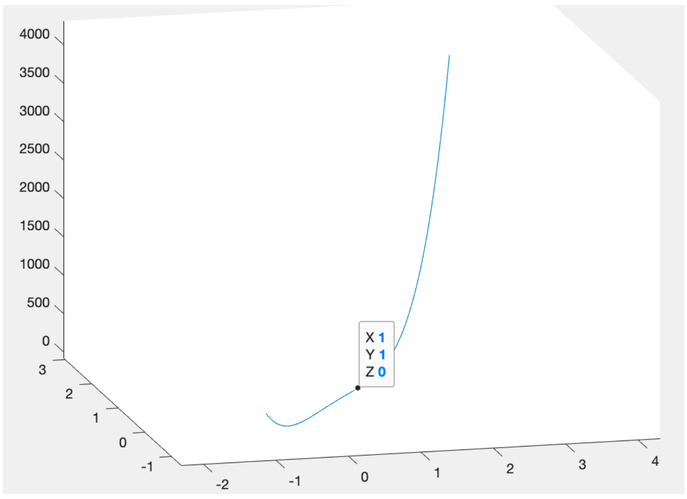


Fig.1. the Rosenbroch's Valley function's figure

(b) Steepest (Gradient) descent method

First, we calculate the gradient vector that

$$\frac{\partial f}{\partial x} = 2(1-x)(-1) + 200(y-x^2)(-2x) = 400x^3 - 400xy + 2x - 2$$

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

So the gradient vector $g(n) = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix} = \begin{bmatrix} 400x^3 - 400xy + 2x - 2 \\ 200y - 200x^2 \end{bmatrix}$

The Steepest descent algorithm's iteration equation is formally described by

$$w(n+1) = w(n) - \eta g(n)$$

The learning rate $\eta = 0.001$, and the starting point is randomly initialized in the open interval $(-1,1)$ for x, y.

Code.1 Steepest (Gradient) descent method

```
%para
eta=0.001;%learning rate
i=1;%iteration number

xr=zeros(1,1000);%init vector
yr=zeros(1,1000);
fr=zeros(1,1000);
ir=zeros(1,1000);
x=-1+2*rand;%random x,y
y=-1+2*rand;
f=(1-x).^2+100.*(y-x.^2).^2;

while f>0.000001
    xr(i)=x;%recording trajectory
    yr(i)=y;
    fr(i)=f;
    ir(i)=i;

    dx=400*x^3-400*x*y+2*x-2;%derivation
    dy=200*y-200*x^2;
    x=x-eta*dx;%update
    y=y-eta*dy;
    f=(1-x).^2+100.*(y-x.^2).^2;
    i=i+1;
end
```

In this case, when the number of iteration is **14266** , the value of the function $f = \mathbf{9.9937} \times \mathbf{10^{-7}}$ approaches the global minimum and the iteration stops. The trajectory of (x, y) and the trajectory of the function $f(x, y)$ value versus iteration are shown in Fig.2 and Fig.3. So we can see clearly in Fig.3 that the velocity, $f(x, y)$ converging to 0, is quite large in the first few iterations. And then it keeps smooth till the iteration stops.

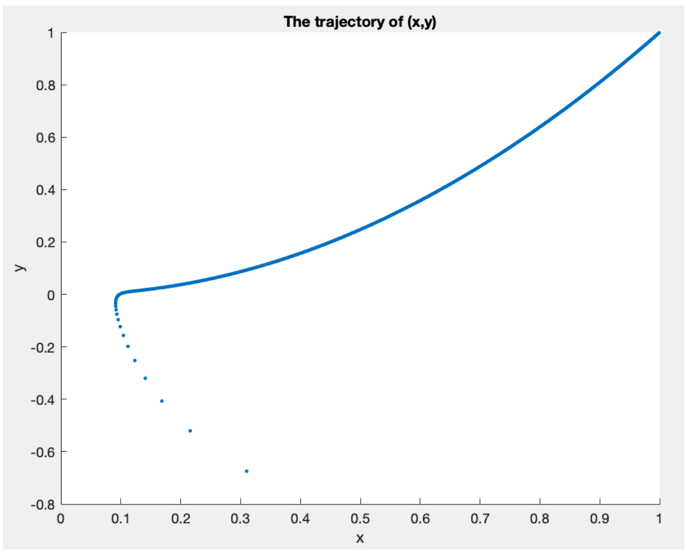


Fig.2. The trajectory of (x,y)

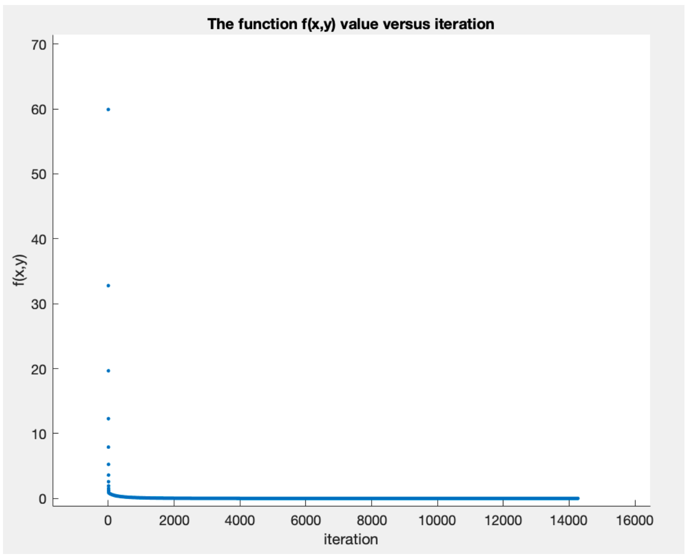


Fig.3. The function f(x,y) value versus iteration

Then changing learning rate η into 1.0, the output is:

$$f(NaN, \mathrm{Inf}) = \mathrm{NaN}, \mathrm{i} = 7$$

We can only see 2 points (A/B) in Fig.4 and 1 point (C) in Fig.5 (in this case), and after 7 iterations (in any case), the value of x, y, f become to large to show in the MATLAB.

Thus, the learning rate should not be too large in this algorithm.

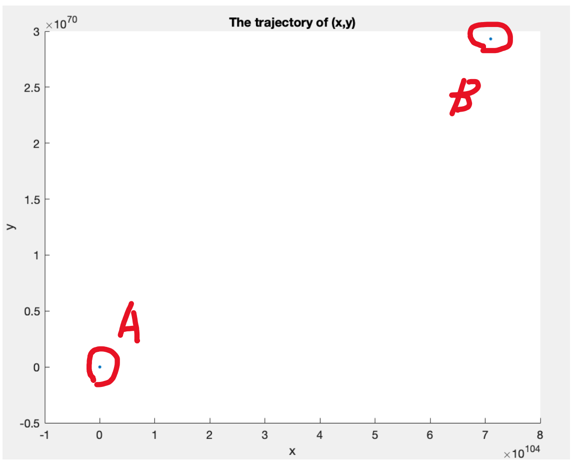


Fig.4. The trajectory of (x,y) η=1

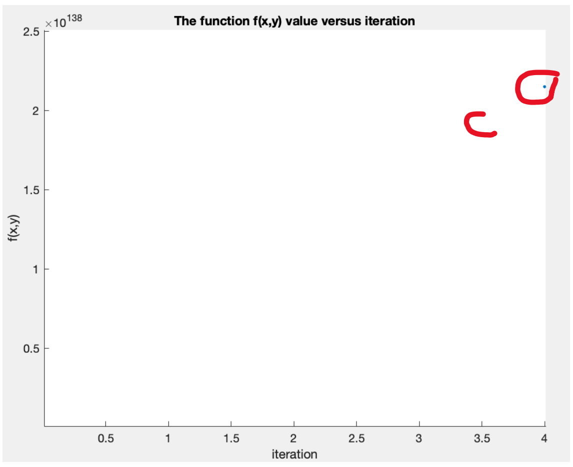


Fig.5. The function f(x,y) value versus iteration

(c) Newton's method

First, we calculate the Hessian matrix that

$$\frac{\partial^2 f}{\partial x^2} = 1200x^2 - 400y + 2; \frac{\partial^2 f}{\partial x \partial y} = -400x; \frac{\partial^2 f}{\partial y^2} = 200$$

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

The Newton's method's iteration equation is formally described by

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

The starting point is randomly initialized in the $(-1,1)$ for x, y.

Code.2 Newton's method

```
%para
i=1;%iteration number

xr=zeros(1,1000);%init recording vector
yr=zeros(1,1000);
fr=zeros(1,1000);
ir=zeros(1,1000);

x=-1+2*rand;%random x,y
y=-1+2*rand;
dx=400*x^3-400*x*y+2*x-2;%derivation
dy=200*y-200*x^2;
dxx=1200.*x.^2-400.*y+2;
dyy=200;
dxy=-400.*x;
f=(1-x).^2+100.*(y-x.^2).^2;
H=[dxx,dxy;dxy,dyy];

while f>0.000001
    xr(i)=x;%recording trajectory
    yr(i)=y;
    fr(i)=f;
    ir(i)=i;

    dx=400*x^3-400*x*y+2*x-2;%derivation
    dy=200*y-200*x^2;
    dxx=1200.*x.^2-400.*y+2;
    dyy=200;
    dxy=-400.*x;
    H=[dxx,dxy;dxy,dyy];
    h=inv(H);

    x=x-h(1,1).*dx-h(1,2).*dy;%update
    y=y-h(2,1).*dx-h(2,2).*dy;
    f=(1-x).^2+100.*(y-x.^2).^2;
    i=i+1;
end
```

In this case, when the number of iteration is **6** , the value of the function $f=$ $\mathbf{6.0767\times 10^{-18}}$ approaches the global minimum and the iteration stops. The trajectory of

(x, y) and the trajectory of the function $f(x, y)$ value versus iteration are shown in Fig.6 and Fig.7.

Comparing with Steepest (Gradient) descent method, iteration from 14266 to 6, which decreases dramatically. Besides, the function value is quite close to 0, when the iteration stops in the same threshold condition (0.000001).

In conclusion, Newton's method is faster than Steepest (Gradient) descent method.

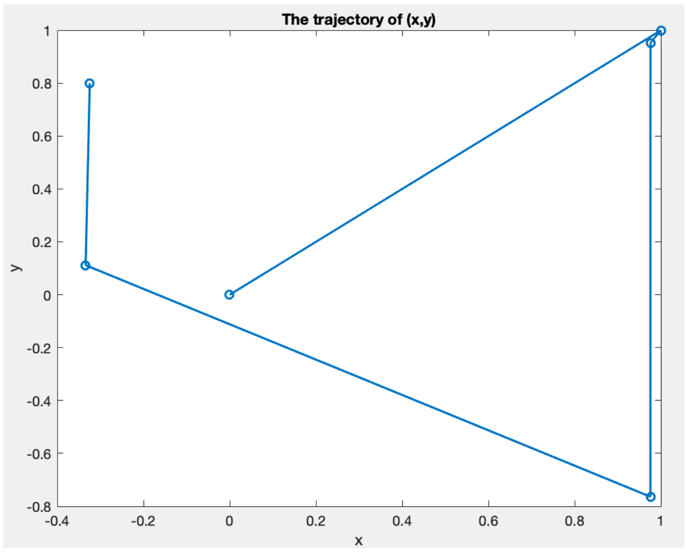


Fig.6. The trajectory of (x,y)

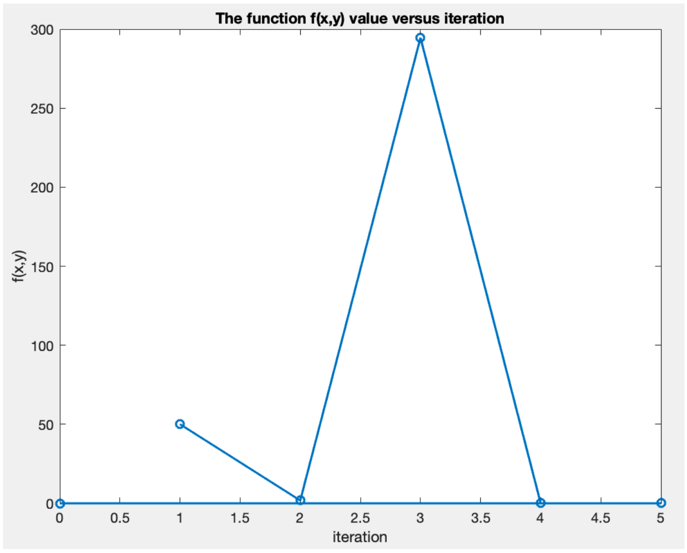
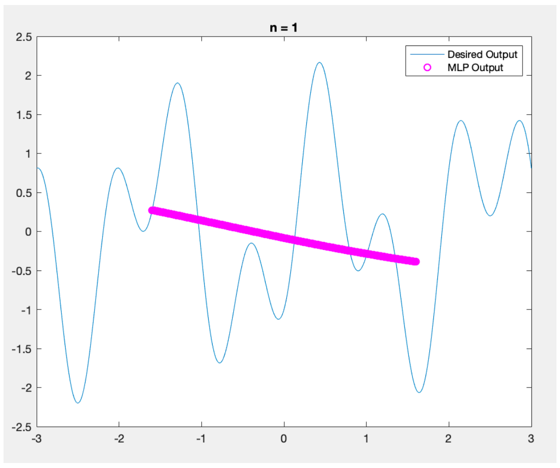


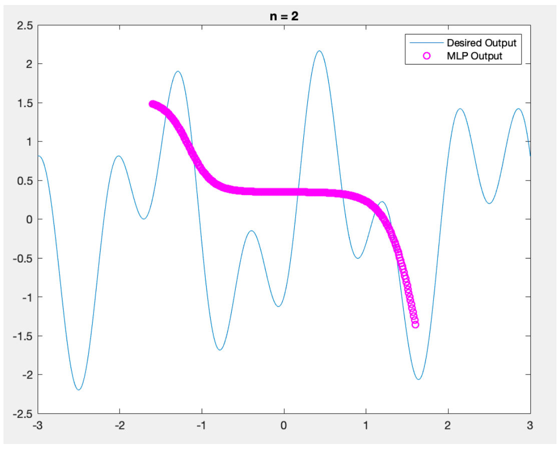
Fig.7. The function f(x,y) value versus iteration

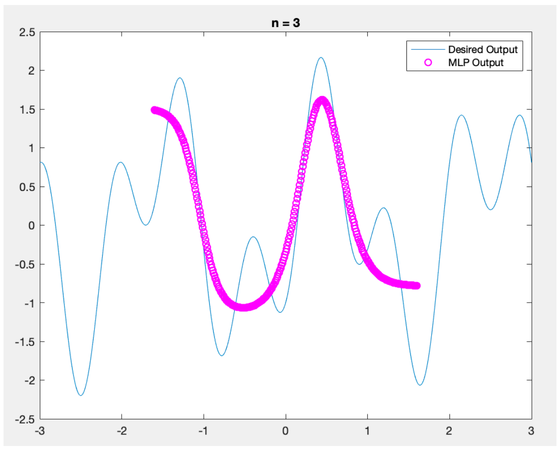
Solution 2

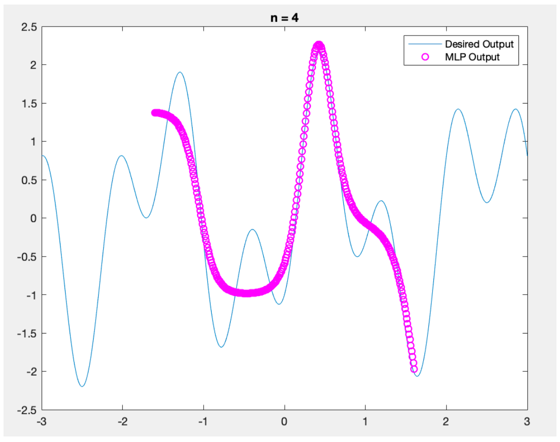
(a) sequential mode with BP algorithm

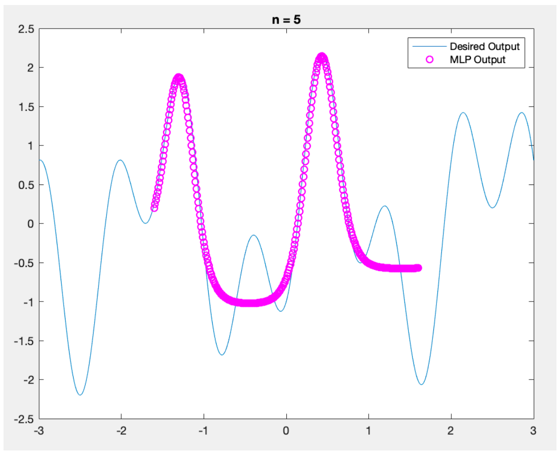
By inputting different n (where n = 1,2,3,4,5,6,7,8,9,10,20,50,100), we can get different structures of the MLP: 1-n-1. All the figures are shown as below.

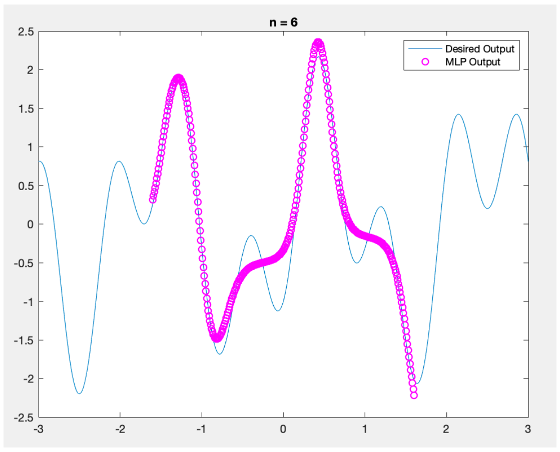


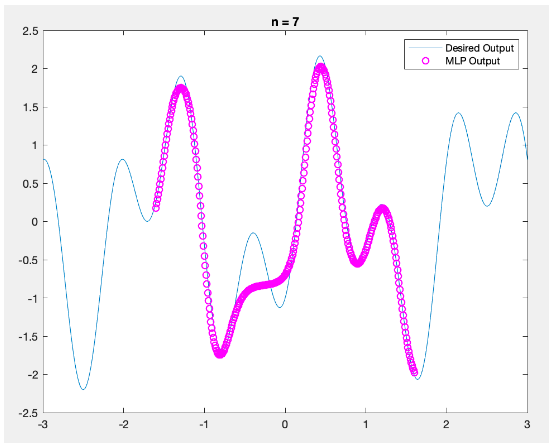


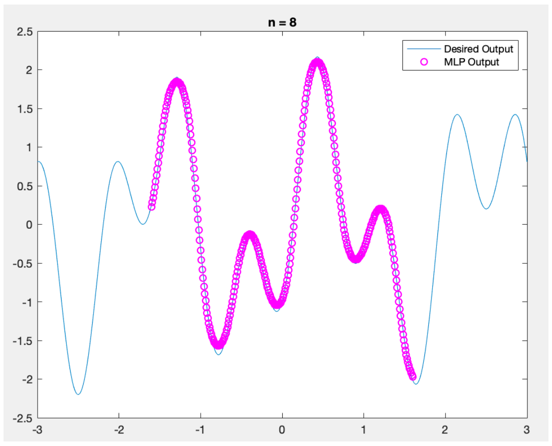


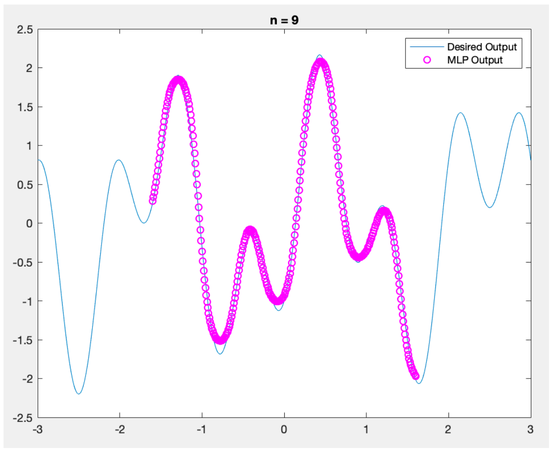


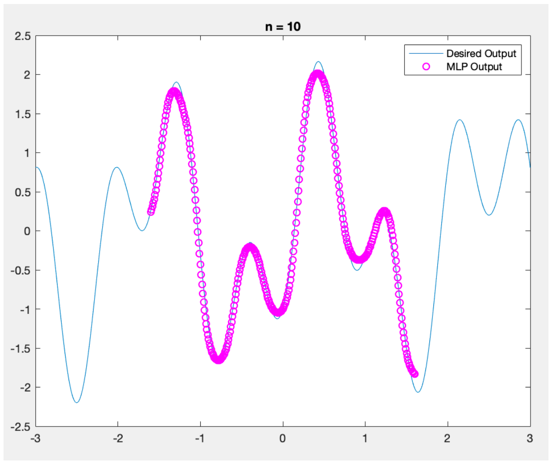


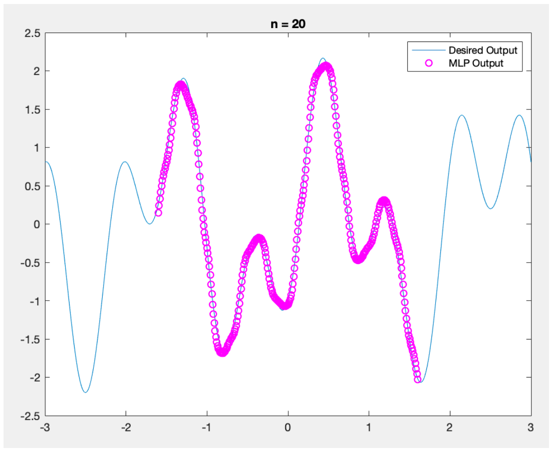


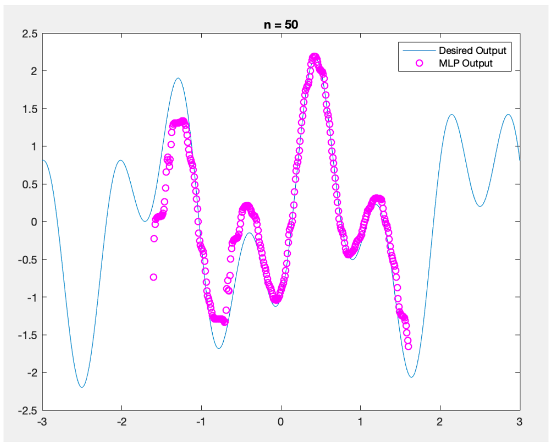

n = 7
Desired Output
MLP Output

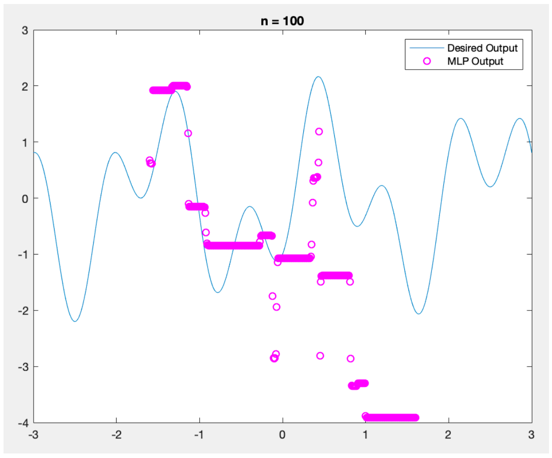

n = 8
Desired Output
MLP Output


n = 9
Desired Output
MLP Output


n = 10
Desired Output
MLP Output


n = 20
Desired Output
MLP Output


n = 50
Desired Output
MLP Output


n = 100
Desired Output
MLP Output

So we can see clearly that

$$\begin{cases} under-fitting, & n = 1,2,3,4,5,6,7 \\ proper-fitting, & n = 8,9,10,20 \\ over-fitting, & n = 50,100 \end{cases}$$

From the figures, we find that the minimal number of hidden neurons from the experiment is n=8. And the result estimated by using guideline given in the lecture slides is 8, too. Thus, in this case, the minimal number of neurons is consistent with the guideline. But outside of the test-input domain [-1.6,1.6], this MLP can not make reasonable predictions (the predictions of outside of the domain [-1.6,1.6] shown in Fig.1).

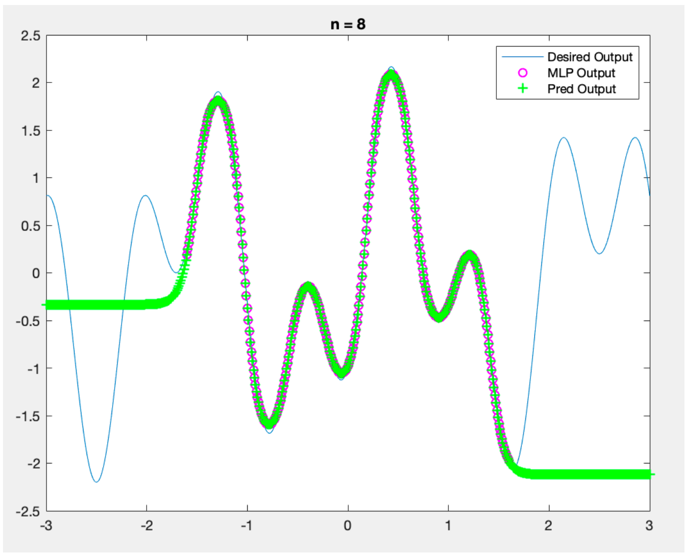
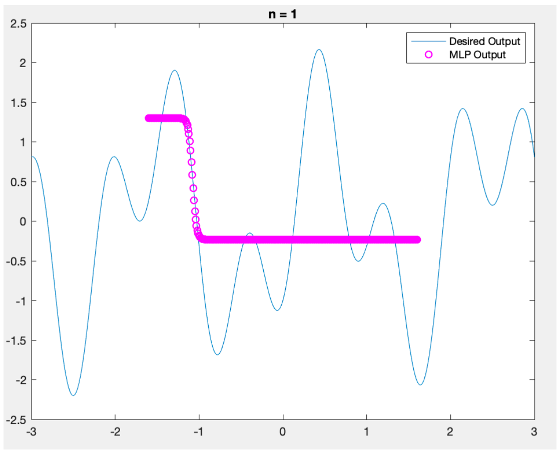


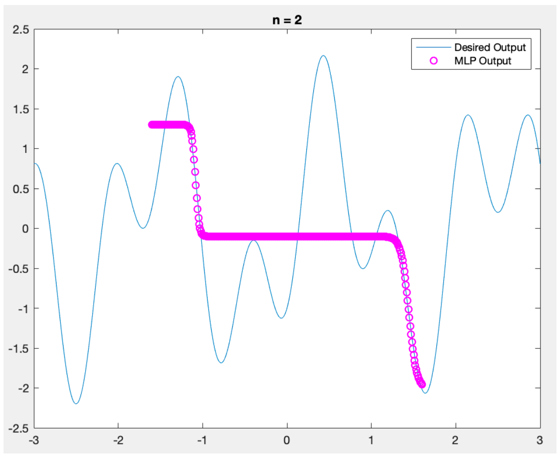
Fig.1. Outside of domain predictions

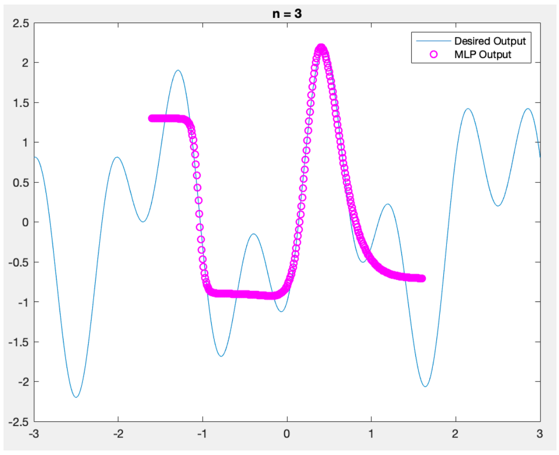
In this case, x=-3,y=-0.3339;x=3,y=-2.12.

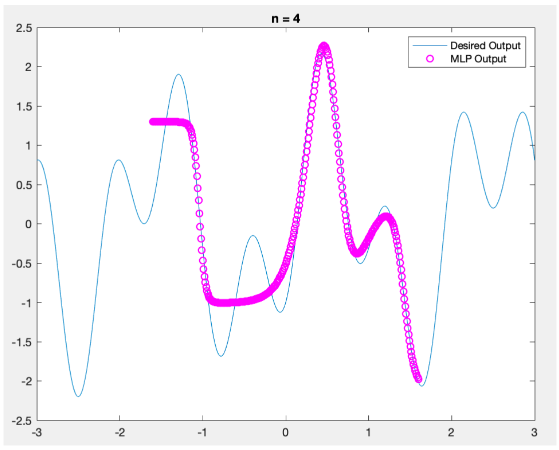
(b) batch mode with trainlm algorithm

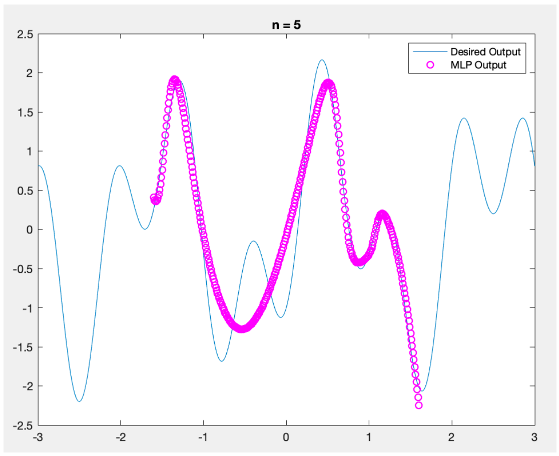
By inputting different n (where n = 1,2,3,4,5,6,7,8,9,10,20,50,100), we can get different structures of the MLP: 1-n-1. All the figures are shown as below

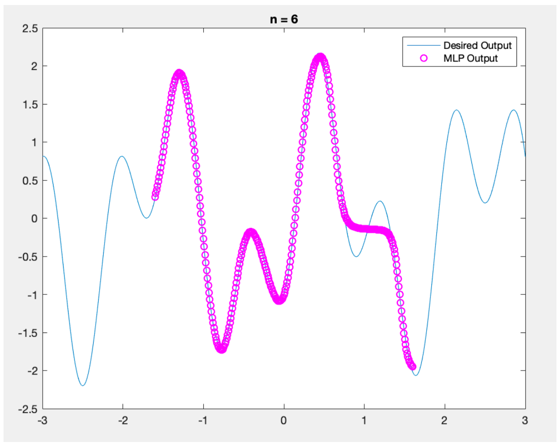

n = 1
Desired Output
MLP Output

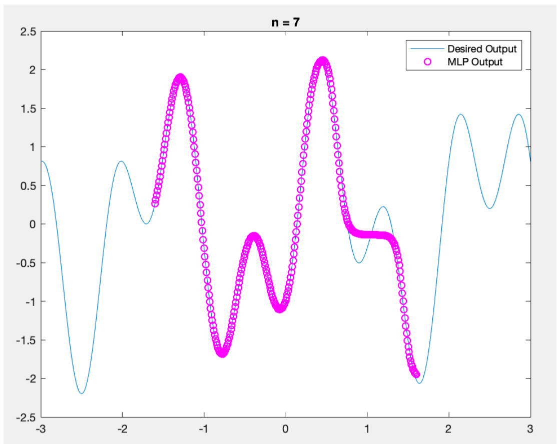

n = 2
Desired Output
MLP Output

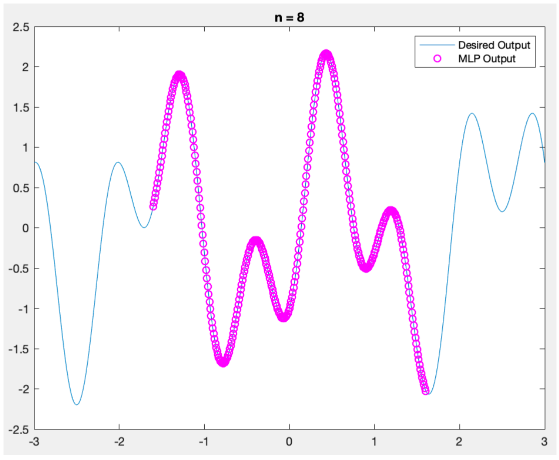

n = 3
Desired Output
MLP Output

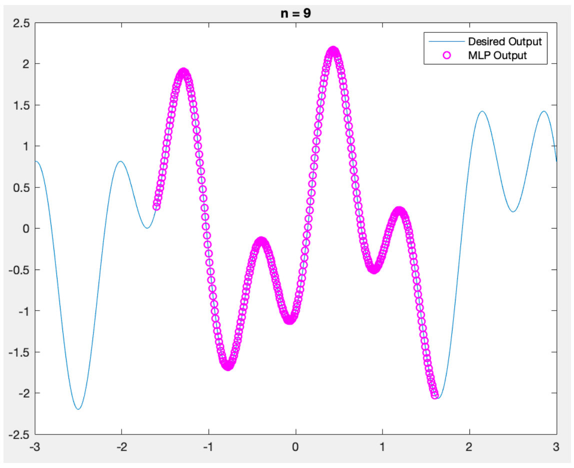

n = 4
Desired Output
MLP Output

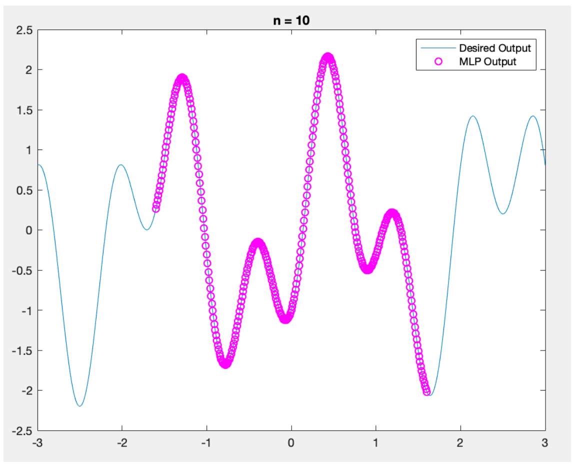

n = 5
Desired Output
MLP Output

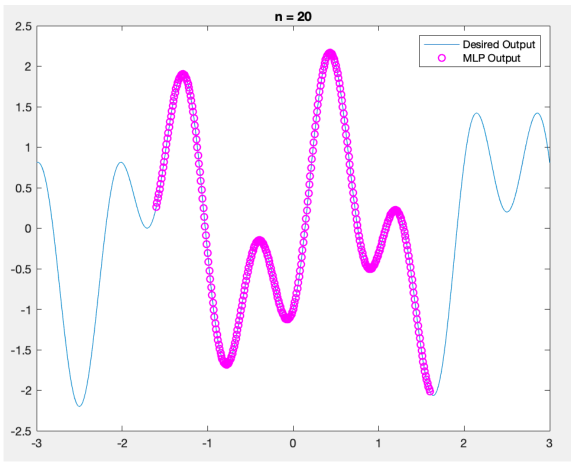

n = 6
Desired Output
MLP Output

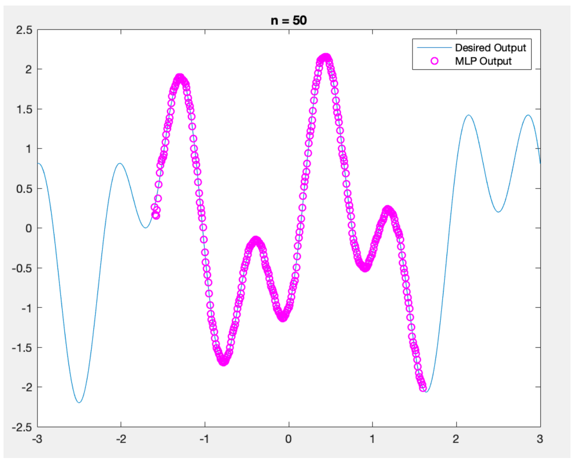

n = 7
Desired Output
MLP Output

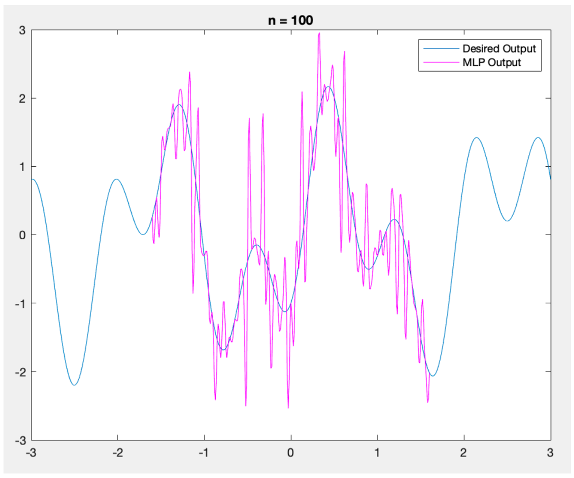

n = 8
Desired Output
MLP Output











So we can see clearly that

$$\begin{cases} under-fitting, & n=1,2,3,4,5,6,7 \\ proper-fitting, & n=8,9,10,20,50 \\ over-fitting, & n=100 \end{cases}$$

From the figures, we find that the minimal number of hidden neurons from the experiment is n=8. And the result estimated by using guideline given in the lecture slides is 8, too. Thus, in this case, the minimal number of neurons is consistent with the guideline. But outside of the test-input domain [-1.6,1.6], this MLP can not make reasonable predictions (the predictions of outside of the domain [-1.6,1.6] shown in Fig.2).

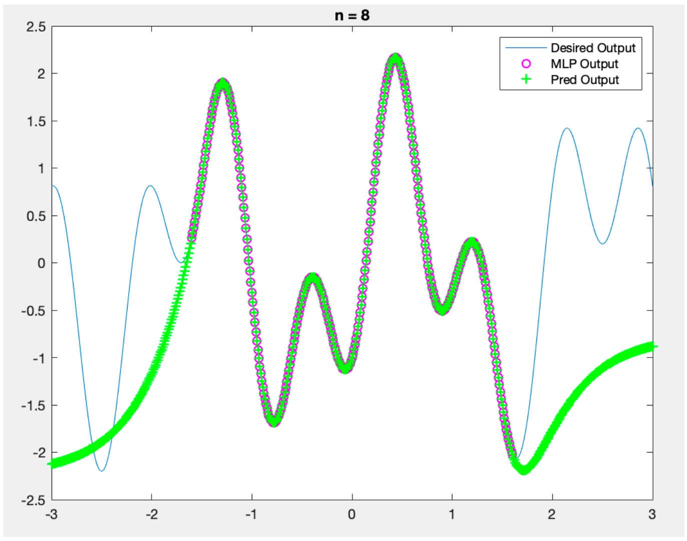
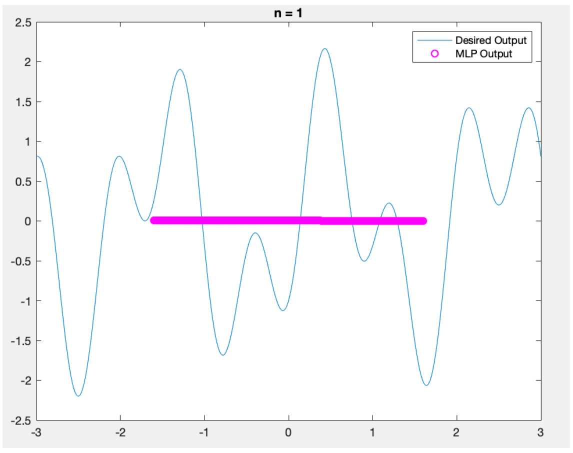


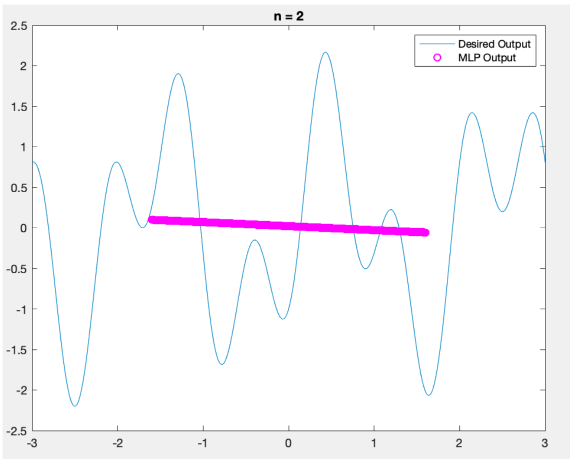
Fig.2. Outside of domain predictions

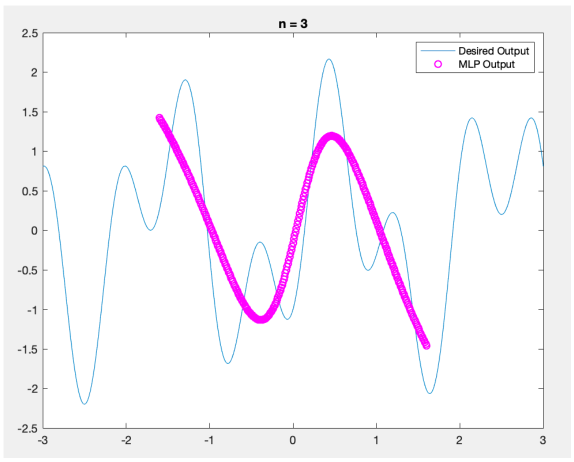
In this case, x=-3,y=-2.125;x=3,y=-0.8802.

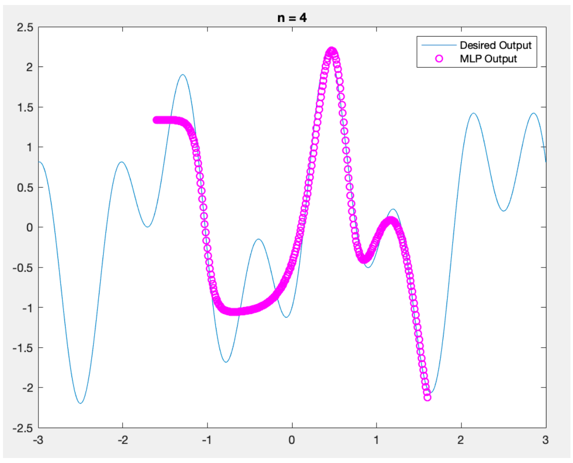
(c) batch mode with trainbr algorithm

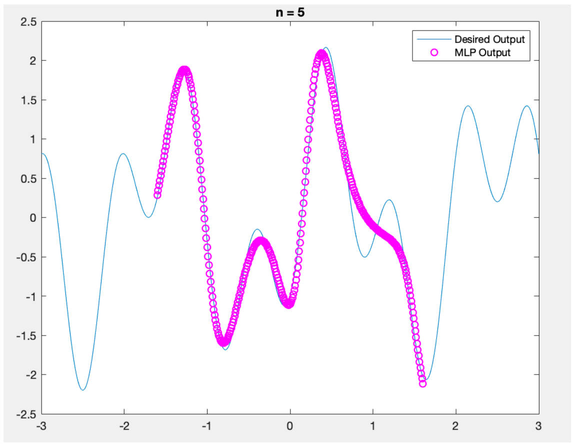
By inputting different n (where n = 1,2,3,4,5,6,7,8,9,10,20,50,100), we can get different structures of the MLP: 1-n-1. All the figures are shown as below.

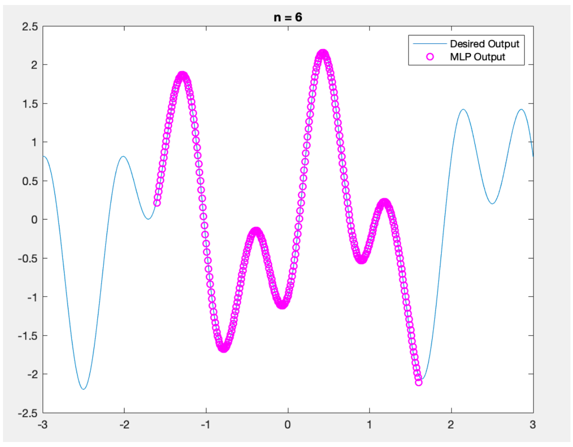


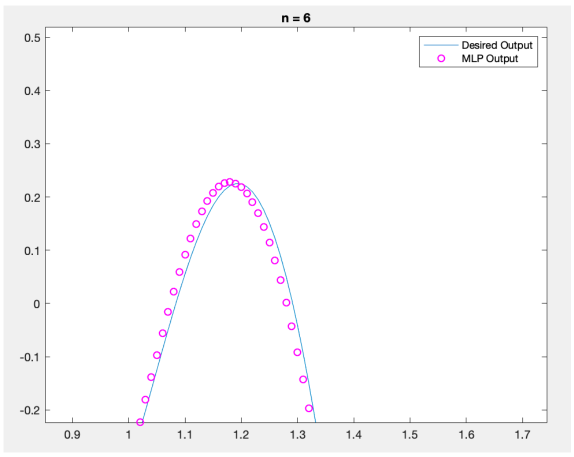


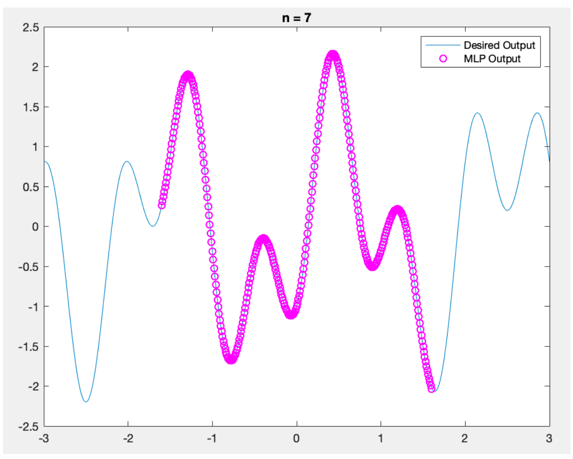

n = 3
Desired Output
MLP Output
2.5 2 1.5 1 0.5 0 -0.5 -1 -1.5 -2 -2.5
-3 -2 -1 0 1 2 3

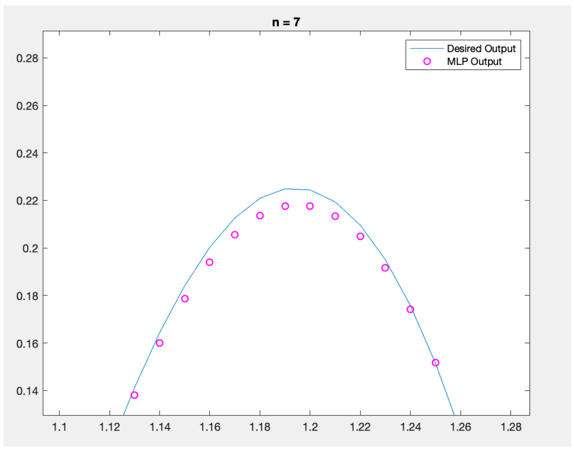

n = 4
Desired Output
MLP Output
2.5 2 1.5 1 0.5 0 -0.5 -1 -1.5 -2 -2.5
-3 -2 -1 0 1 2 3

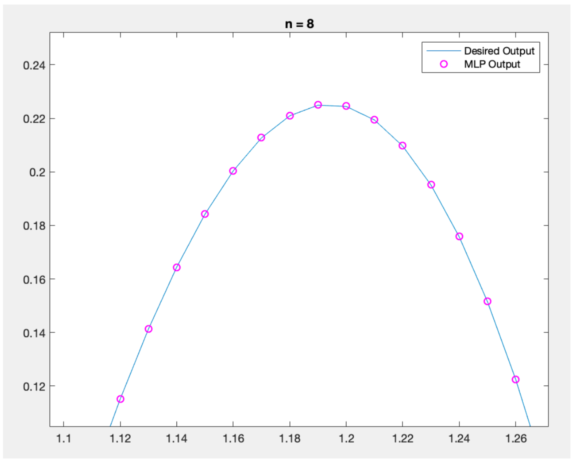

n = 5
Desired Output
MLP Output
2.5 2 1.5 1 0.5 0 -0.5 -1 -1.5 -2 -2.5
-3 -2 -1 0 1 2 3

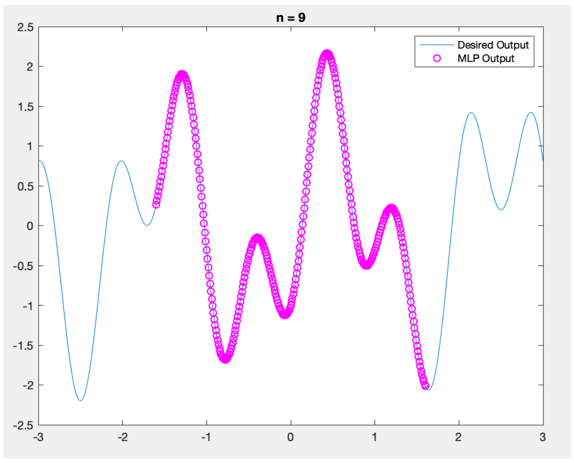

n = 6
Desired Output
MLP Output
2.5 2 1.5 1 0.5 0 -0.5 -1 -1.5 -2 -2.5
-3 -2 -1 0 1 2 3

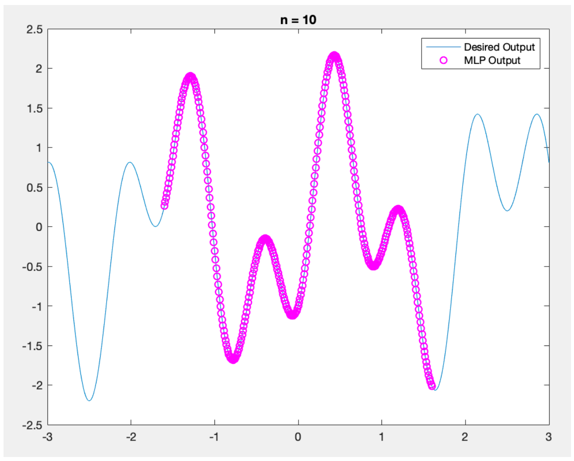

n = 6
Desired Output
MLP Output
0.5 0.4 0.3 0.2 0.1 0 -0.1 -0.2
0.9 1 1.1 1.2 1.3 1.4 1.5 1.6 1.7

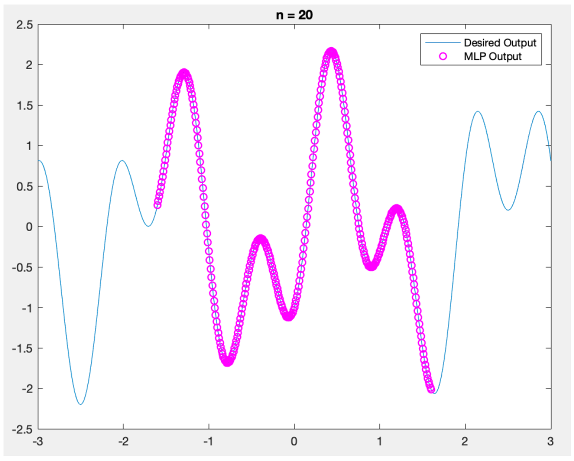

n = 7
Desired Output
MLP Output
2.5 2 1.5 1 0.5 0 -0.5 -1 -1.5 -2 -2.5
-3 -2 -1 0 1 2 3

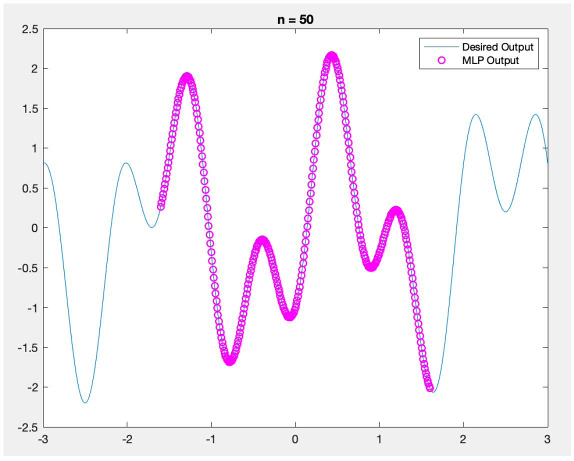

n = 7
Desired Output
MLP Output
0.28 0.26 0.24 0.22 0.2 0.18 0.16 0.14
1.1 1.12 1.14 1.16 1.18 1.2 1.22 1.24 1.26 1.28

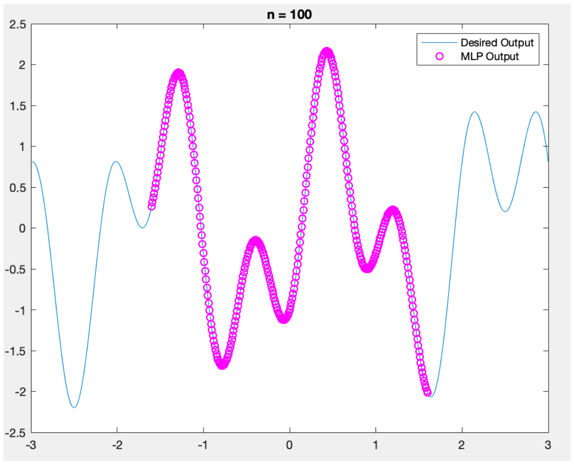

n = 8
Desired Output
MLP Output
0.24 0.22 0.2 0.18 0.16 0.14 0.12
1.1 1.12 1.14 1.16 1.18 1.2 1.22 1.24 1.26











So we can see clearly that

$$\begin{cases} \quad under - fitting, \quad n = 1,2,3,4,5,6,7 \\ proper - fitting, \quad n = 8,9,10,20,50,100 \end{cases}$$

It is not over-fitting when n = 100, so I increase n. In the end, it will be over-fitting, as long as n large enough.

From the figures, we find that the minimal number of hidden neurons from the experiment is n=6. When n=6&7, the graph is very close to the desired output, although it still exists some errors by blowing up the graph. And the result estimated by using guideline given in the lecture slides is 8. Thus, in this case, the minimal number of neurons is **not consistent** with the guideline. But outside of the test-input domain [-1.6,1.6], this MLP can not make reasonable predictions (the predictions of outside of the domain [-1.6,1.6] shown in Fig.3).

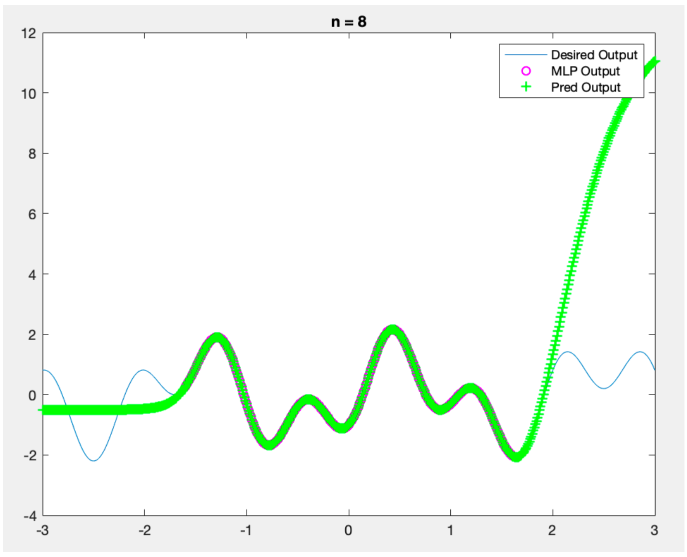


Fig.3. Outside of domain predictions

In this case, x=-3,y=-0.5156;x=3,y=11.07.

Code for 2-a

```
%desired figure
x=-3:0.01:3;
y=1.2*sin(pi*x)-cos(2.4*pi*x);

%set training set domain[-1.6,1.6] uniform step 0.05
x_train=-1.6:0.01:1.6;
y_train=1.2*sin(pi*x_train)-cos(2.4*pi*x_train);

%n=1,2,3,4,5,6,7,8,9,10,20,50,100
n=8;
epochs=100;
total_num=size(x_train,2);
%µ÷ÓÃsequential train º¯Êý
[ net, accu_train, accu_val ]=train_seq( n, x_train, y_train, total_num, 0,
epochs );

%set test set domain[-1.6,1.6] uniform step 0.01
x_test=-1.6:0.01:1.6;

%plot
figure(1);
plot(x,y);%Desired Output
hold on;
MLP_output=net(x_test);
plot(x_test,MLP_output,'o','color','m');%MLP Output
hold on;
```

```
pred_output=sim(net,x);
plot(x,pred_output,'+','color','g');%Pred Output
hold on;
legend('Desired Output','MLP Output','Pred Output');
title(['n = ',num2str(n)]);
```

code for 2-b/c

```
%desired figure
x=-3:0.01:3;
y=1.2*sin(pi*x)-cos(2.4*pi*x);

%set training set domain[-1.6,1.6] uniform step 0.05
x_train=-1.6:0.05:1.6;
y_train=1.2*sin(pi*x_train)-cos(2.4*pi*x_train);

%n=1,2,3,4,5,6,7,8,9,10,20,50,100
n=8;
%% specify the structure and learning algorithm for MLP
net = feedforwardnet(n,'trainbr');
net.layers{1}.transferFcn = 'tansig';%tansig is the preferred in hidden
neurons
net.layers{2}.transferFcn = 'purelin';%purelin for regression
net = configure(net,x_train,y_train);
net.trainparam.lr=0.01;%learning rate
net.trainparam.epochs=10000;
net.trainparam.goal=1e-8;
net.divideParam.trainRatio=1.0;
net.divideParam.valRatio=0.0;
net.divideParam.testRatio=0.0;
%% Train the MLP
[net,tr]=train(net,x_train,y_train);%train the MLP

%set test set domain[-1.6,1.6] uniform step 0.01
x_test=-1.6:0.01:1.6;

%plot
figure(1);
plot(x,y);%desired output
hold on;
net_output=sim(net,x_test);
plot(x_test,net_output,'o','color','m');%MLP Output
hold on;
pred_output=sim(net,x);
plot(x,pred_output,'+','color','g');%Pred Output
hold on;
legend('Desired Output','MLP Output','Pred Output');
title(['n = ',num2str(n)]);
```

Solution 3

Matric No. A0195017E. Group No. mod(17,3) = 2.(deer1/ship0)

The data provided in images in RGB format, in order to be used by MATLAB conveniently, I process the data first.

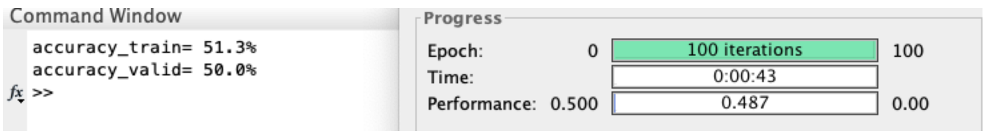
Code.1 data preparation function

```
function [train_image]=dataprep(file_dir)
img_list=dir(strcat(file_dir,'*.jpg'));%format: img_list.name = 000.jpg
N=size(img_list);
image_c=cell(1,500);%init a cell to store data

for i=1:N
    img_name = strcat(file_dir,img_list(i).name);%get each image full path
    img_rgb=imread(num2str(img_name));%read image in rgb format
    img_gray=rgb2gray(img_rgb);%change rgb to gray
    image_c{:,i}=img_gray;
end

image_mat=cell2mat(image_c);%cell to mat
train_image_uint8=reshape(image_mat,1024,500);%reshape from 32*(32*500) to
1024*500
train_image=im2double(train_image_uint8);%change data type from uint8 to
double
end
```

```
%combine ship and deer to generate train and validation set
train_set=horzcat(train_image0(:,1:450),train_image1(:,1:450));
valid_set=horzcat(train_image0(:,451:500),train_image1(:,451:500));
train_set=double(train_set);
valid_set=double(valid_set);

%save train set and validation set
save('train_set.mat','train_set');
save('valid_set.mat','valid_set');
```

(a) Rosenblatt's perceptron

By training Rosenblatt`s perceptron, I find that it need a long time iteration, before getting a good performance. So it is a time-consuming perceptron. The statistics of iteration 100/500/1000 are shown as below. From 100 times iteration to 1000, the performance increases a little. Thus, if we want to get a good result, we need train this perceptron as long as possible.



```
Command Window
accuracy_train= 51.3%
accuracy_valid= 50.0%
>>
```

```
Progress
Epoch:          0     100 iterations    100
Time:                 0:00:43
Performance:  0.500   0.487             0.00
```

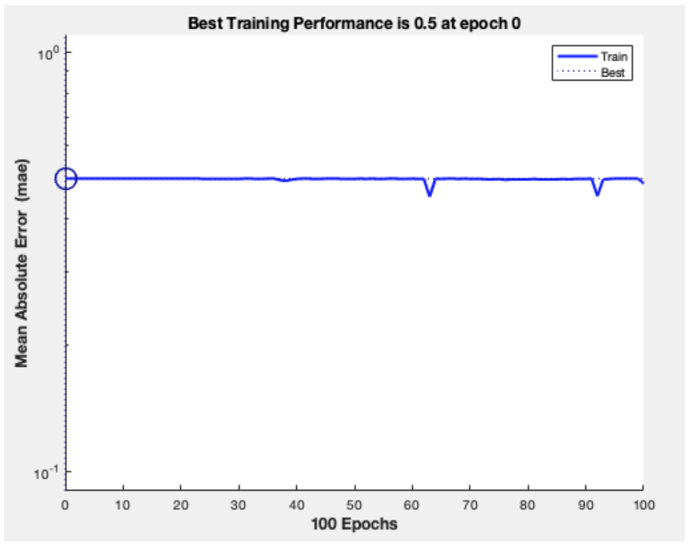
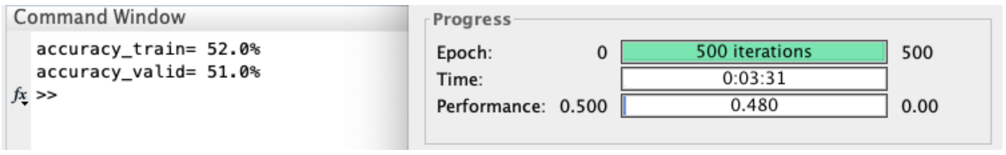


Fig.1 epoch=100



```
Command Window
accuracy_train= 52.0%
accuracy_valid= 51.0%
>>
```

```
Progress
Epoch:            0   500 iterations   500
Time:                 0:03:31
Performance:  0.500   0.480            0.00
```

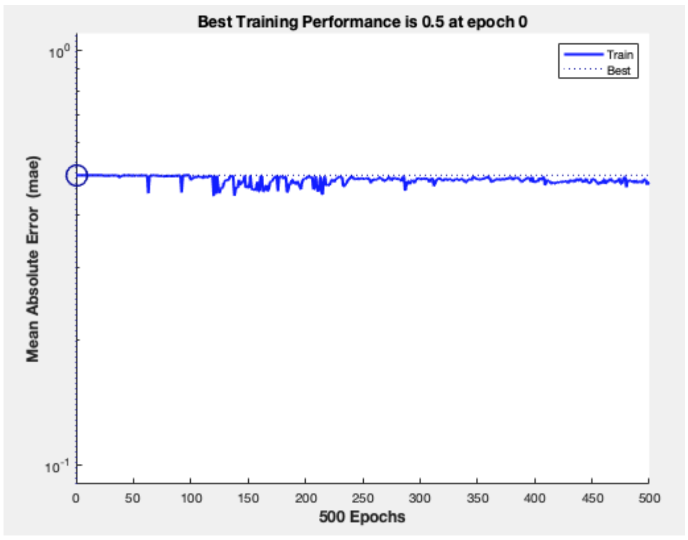
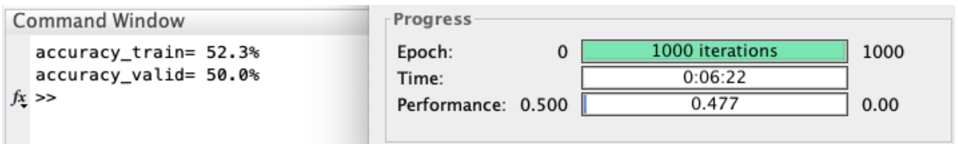


Fig.2 epoch=500



```
Command Window
accuracy_train= 52.3%
accuracy_valid= 50.0%
>>
```

```
Progress
Epoch:            0   1000 iterations   1000
Time:                 0:06:22
Performance:  0.500   0.477             0.00
```

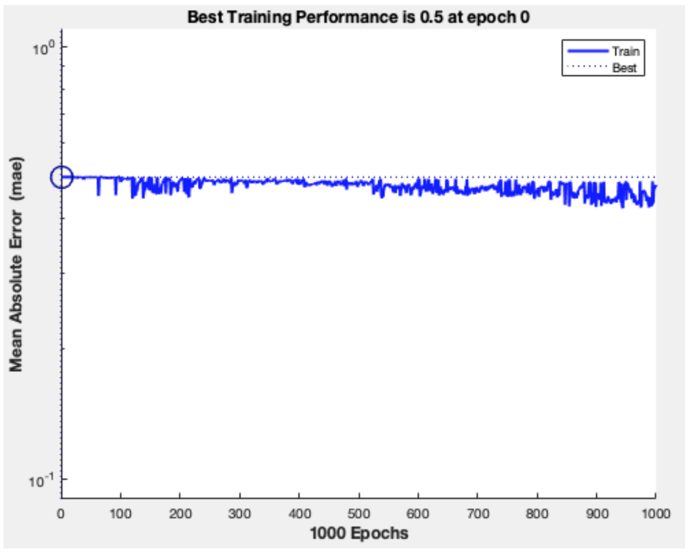


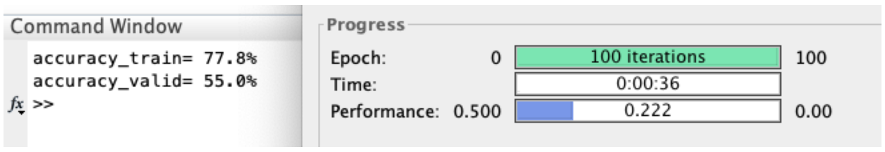
Fig.3 epoch=1000

Code.2 Rosenblatt's perceptron

```
%load data set
load('train_set.mat');
load('valid_set.mat');

%init train label & valid label
train_label=zeros(1,900);
valid_label=zeros(1,100);

for i=1:450
    train_label(1,i+450)=1;
end
for i=1:50
    valid_label(1,i+50)=1;
end

%set network
net=perceptron('hardlim','learnpn');
net.trainParam.epochs=100;
net.trainParam.show=50;
net.divideFcn = 'dividetrain';
net.performParam.regularization = 0.1;

%train
net=train(net,train_set,train_label);

%validate
output_valid=sim(net,valid_set);

%accuracy
output_train=sim(net,train_set);
valid=0;
train=0;
for i=1:100
    value=abs(output_valid(i)-valid_label(i));
    if(value<0.5)
        valid=valid+1;
    end
end
for i=1:900
    value=abs(output_train(i)-train_label(i));
    if(value<0.5)
        train=train+1;
    end
end
```

(b) Rosenblatt`s perceptron in normalization of the data

In part a, the training data without normalization, so it is too big to compute with long time. In this part, I normalize train data with global mean and standard deviation. In order to compare with that in a, I apply Rosenblatt`s perceptron in same situation.



```
Command Window
accuracy_train= 77.8%
accuracy_valid= 55.0%
>>
```

```
Progress
Epoch:        0   100 iterations   100
Time:             0:00:36
Performance: 0.500   0.222        0.00
```

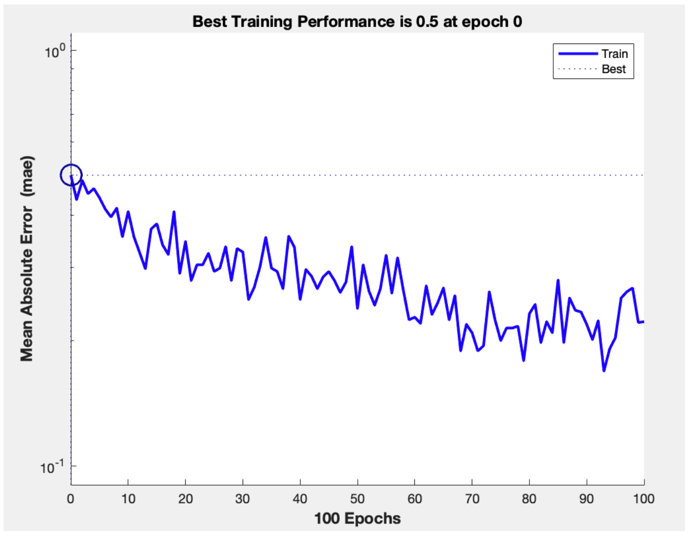
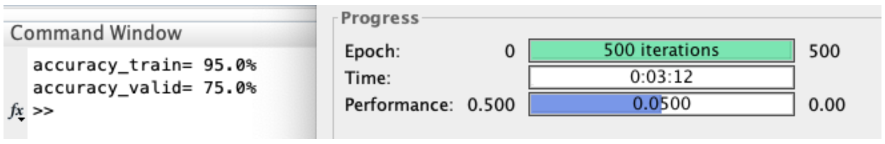


Fig.4 epoch=100



```
Command Window
accuracy_train= 95.0%
accuracy_valid= 75.0%
>>
```

```
Progress
Epoch:        0   500 iterations   500
Time:             0:03:12
Performance: 0.500   0.0500       0.00
```

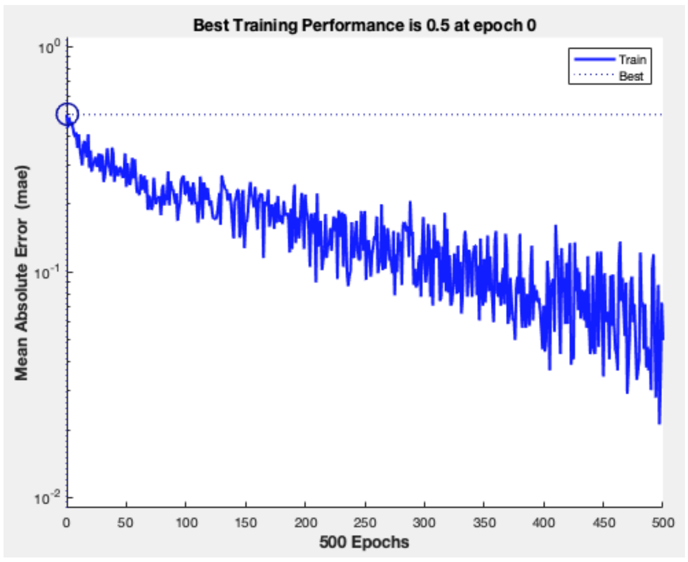
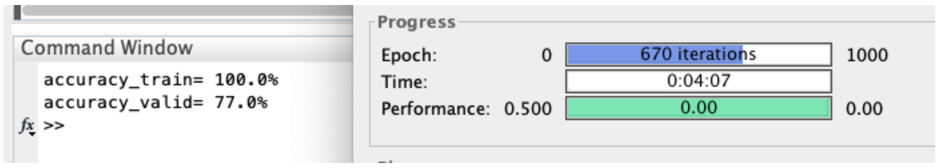


Fig.5 epoch=500



```
Command Window
  accuracy_train= 100.0%
  accuracy_valid= 77.0%
fx >>
```

```
Progress
Epoch:          0   670 iterations   1000
Time:                  0:04:07
Performance: 0.500       0.00        0.00
```

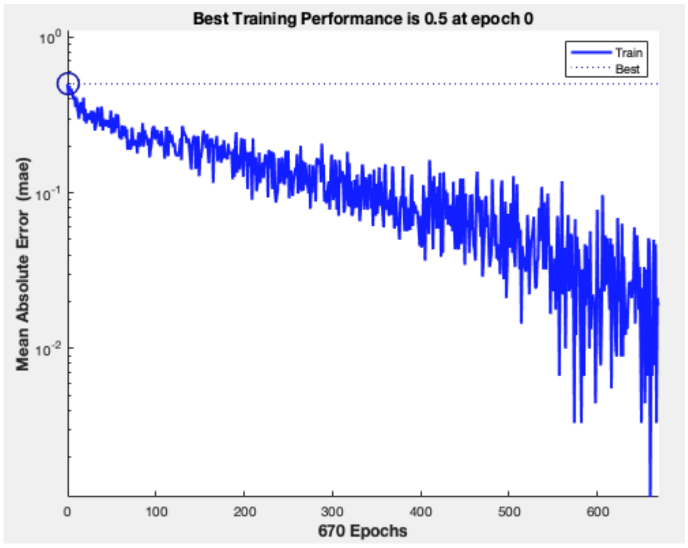
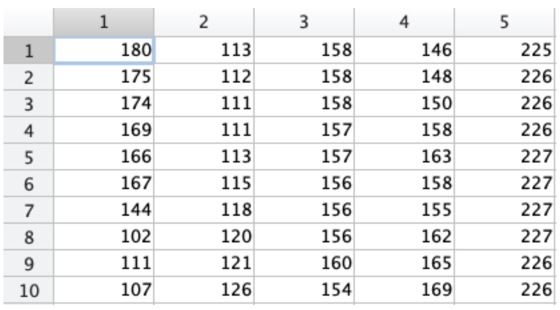


Fig.6 epoch=1000

Compared with the results of a, respectively.

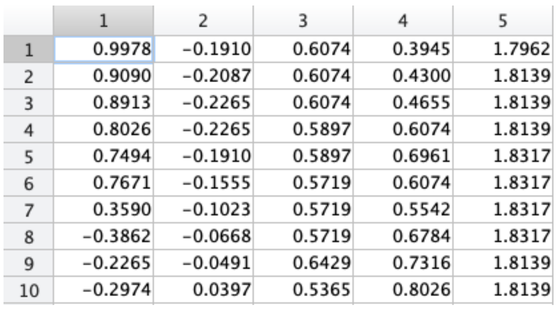
	Epoch=100			**Epoch=500**			**Epoch=1000**		
	tra/val	time	perf	tra/val	time	perf	tra/val	time	perf
a	51.3%/50%	43s	0.487	52%/51%	3m31s	0.48	52.3%/50%	6m22s	0.477
b	77.8%/55%	36s	0.222	95%/75%	3m12s	0.0500	100%/77%	4m7s	0.00

Thereinto, the part b training procedure stops, when the iteration is 670. We can see very clearly that the normalization accelerates the training procedure and improve the accuracy obviously in same training situation.



	1	2	3	4	5
1	180	113	158	146	225
2	175	112	158	148	226
3	174	111	158	150	226
4	169	111	157	158	226
5	166	113	157	163	227
6	167	115	156	158	227
7	144	118	156	155	227
8	102	120	156	162	227
9	111	121	160	165	226
10	107	126	154	169	226

Fig.7. train set before normalization.



	1	2	3	4	5
1	0.9978	-0.1910	0.6074	0.3945	1.7962
2	0.9090	-0.2087	0.6074	0.4300	1.8139
3	0.8913	-0.2265	0.6074	0.4655	1.8139
4	0.8026	-0.2265	0.5897	0.6074	1.8139
5	0.7494	-0.1910	0.5897	0.6961	1.8317
6	0.7671	-0.1555	0.5719	0.6074	1.8317
7	0.3590	-0.1023	0.5719	0.5542	1.8317
8	-0.3862	-0.0668	0.5719	0.6784	1.8317
9	-0.2265	-0.0491	0.6429	0.7316	1.8139
10	-0.2974	0.0397	0.5365	0.8026	1.8139

Fig.8. train set after normalization

The reason why normalization is so effective is that the perceptron adjust each neuron weight and bias in each learning step. If the domain of the input is too big that some input elements are big and some elements are small, it will spend long time on learning procedure, which need more iterations. Thus, normalization of training set is vital.

Code.3 normalization

```
%normalizaition
train_mean=mean(train_set,'all');
train_m=train_set-train_mean*ones(1024,900);
train_std=std(train_m,0,'all');
train_set_N=train_m/train_std;

valid_mean=mean(valid_set,'all');
valid_m=valid_set-valid_mean*ones(1024,100);
valid_std=std(valid_m,0,'all');
valid_set_N=valid_m/valid_std;
```

(c) MLP of batch mode training

In this part, I decide to use function patternnet() to build neural network, because of its better performance in pattern recognition. According to the "Choose a Multilayer Neural Network Training Function" , I choose two training functions ***trainrp*** and ***trainscg*** to compare, due to they almost same good in pattern recognition. Comparing they in n=10/20/50, ***trainscg*** performs better not only in accuracy of validation, but also in the score of performace. Thus I choose ***trainscg*** in the following.

n	1	2	3	4	5	6	7	8	9	10
train	99.0	97.6	99.7	100	100	100	100	100	100	100
validation	71	77	77	76	74	78	78	82	81	81

epoch	584	1379	2314	691	431	385	257	204	172	143
n	15	20	30	40	50	60	100	150	200	300
train	100	100	100	100	100	100	100	100	100	100
validation	82	82	81	82	83	81	82	84	84	81
epoch	108	103	104	83	84	90	87	89	85	92

According to the figures, we can find that the accuracy of training set can reach 100%, as long as n more than 3. And when n larger than 7, the accuracy of validation set keeps stable(ignoring small fluctuation). Compared with part a, it can get better performance by using MLP for this question.

Code.4 MLP of batch mode training network setting

```
%set network
n=10;
net=patternnet(n,'trainscg','crossentropy');
net=configure(net,train_set_N,train_label);
net.trainParam.epochs=5000;
net.trainParam.lr=0.01;
net.trainParam.show=50;
net.divideFcn = 'dividetrain';
```

(d)

Choosing n=8 to experiment this case (1-8-1 MLP) and regulation (0/0.4) to compare. The figures shown as below. In Fig.9, we can observe that before 40 epochs, both accuracy increase sharply. When epoch=70, the accuracy of validation decrease, which suffer from overfitting. By setting the regulation=0.4, we can get Fig.10. While n under 25 epochs, both accuracy increase, it is under-fitting during this period. Then the accuracy of validation keep stable with some fluctuation from n= 25 to 155, which is proper-fitting. Finally, the accuracy of train set close to the 100%, but the accuracy of validation set starts to decrease, which is over-fitting beyond 160 epochs..

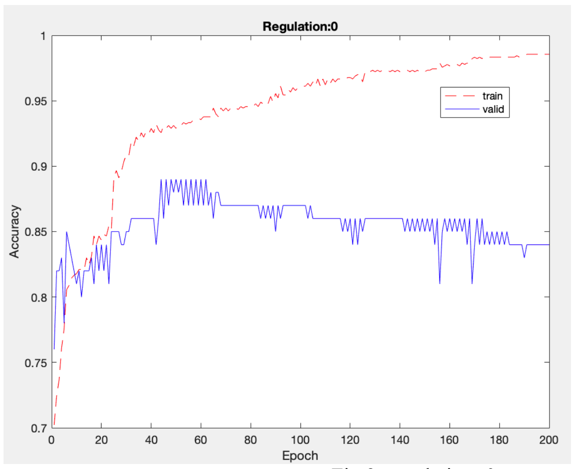


Fig.9. regulation=0

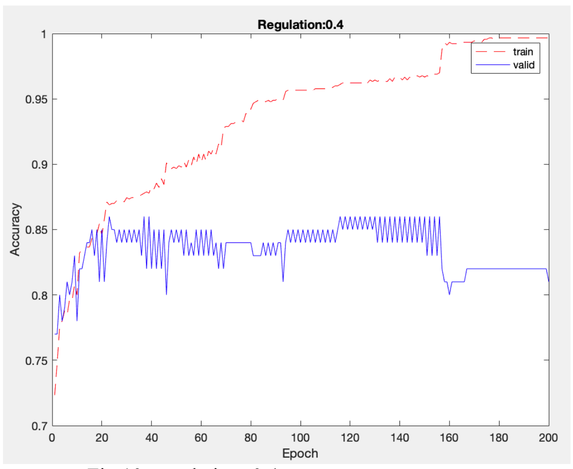


Fig.10. regulation=0.4

```
r_accu_t=zeros(1,200);
r_accu_v=zeros(1,200);
for j=1:200
    %train
    net=train(net,train_set_N,train_label);
    %validate
    output_valid=sim(net,valid_set_N);
    %accuracy
    output_train=sim(net,train_set_N);
    m_valid=0;
    m_train=0;
    for i=1:100
        value=abs(output_valid(i)-valid_label(i));
        if(value<0.5)
            m_valid=m_valid+1;
        end
    end
    for i=1:900
        value=abs(output_train(i)-train_label(i));
        if(value<0.5)
            m_train=m_train+1;
        end
    end
    r_accu_t(j)=m_train/900;
    r_accu_v(j)=m_valid/100;
end

x=1:200;
figure(2);
plot(x,r_accu_t(1:200),'--','color','r'),hold on;
plot(x,r_accu_v(1:200),'color','b'),hold on;
legend('train','valid');
title('Regulation:0.6');
xlabel('Epoch');
ylabel('Accuracy');
```

(e) MLP of sequential mode training

In this part, I apply MLP to dataset by using sequential mode training with epoch=100 and n from 1 to 10. All the accuracy of train set and validation set shown as below.

n	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
train	93.8	94.9	97.9	97.8	98	99.1	99.7	99.9	99.9	99.9
validation	79	84	80	79	81	82	79	84	83	81

Because the limitation of epoch (100), the accuracy of train set does not reach 100%. However, the accuracy of validation set is very close to that in batch mode training. Thus, we know that these two modes perform at classification similarly.

Speaking of recommendation to choose these two modes in a certain question. Sequential mode is a time-consuming method, but has small storage requirement. On the contrary, batch mode is faster than sequential mode, but it need more storage. So for some small and time-sensitive dataset, we can apply MLP by using batch mode training.

(f) Scheme to improve the performance of MLP

a. Comparing part c(batch mode training) and e(sequential mode training), we should choose batch mode training. Because sequential mode training

b. Comparing part a and b, we should normalization first, which not only improving the accuracy of classification, but also some dataset can not experiment without normalization.

c. For different problem, we should choose special train function. For example, trainlm is the fastest convergence train function, with lower mean square error, and trainrp is the fastest pattern recognition train function, with small storage requirement. So in order to improve the performance of MLP, we should choose the train function with the aim of certain problem.