Neural Network

Huasong Zhang
Department of Applied Mathematics
University of Washington
Seattle, WA 98195
huasongz@uw.edu

Abstract

In this homework assignment, we are going to train neural network on three different dynamical systems to predict the moving trajectories by giving an initial condition. The dynamical systems we are using here are Kuramoto-Sivashinsky (KS) equation, reaction-diffusion system, and Lorenz equation. From this assignment we can see that it is possible to predict trajectories by using neural networks even though we do not know the true equations or the system is chaotic. However, it was difficult to tune the parameters for neural networks in a short period of time.

1 Introduction and Overview

Machine learning has been widely used in recent years. People are seeking higher accuracy in classification and regression tasks. Therefore, neural networks are invented. In this homework assignment, we assumed we do not have the exact equation for dynamical system and wanted to train neural network to predict the trajectory based on the initial condition.

2 Theoretical Background

Neural network builds multiple layers between the inputs and outputs and put different functions on each layer to map the input layer by layer. In equation, the idea for neural networks look like this:

$$f(A_3, f(A_2, f(A_1, x)))...$$

where A and f map the input x into a new space, and in next iteration, this will be taken as new input and we will have a new A and f. Each A and f is a layer. We keep doing this and our neural network will be built.

2.1 Activation Function

For functions that are used in each layer, we did not use random functions. The functions we used are all easy to differentiate. The most commonly used functions are:

$$f(x) = \begin{cases} x & \text{linear} \\ \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases} & \text{binary step} \\ \frac{1}{1+e^{-x}} & \text{logistic} \\ tanh(x) & \text{tanh} \\ \begin{cases} 0 & x \leq 0 \\ x & x > 0 \end{cases} & \text{ReLU} \end{cases}$$

2.2 Loss Function

In order to see whether we have a good prediction from neural net or not, we need a way to represent what a good prediction looks like. In order to do that, the idea of loss function is introduced. In this way, we quantifies the prediction by a number. When this number is small, which means the prediction is good; and large number means bad predictions. Normally, mean square error is used as loss function, which is computed as:

$$argmin_A ||y - f(A, x)||^2$$

2.3 optimizer

In order to minimize the loss function, an optimizer is needed. Since the input and output are normally high dimensional. Therefore, it is hard to find the global minimum or maximum. At this time, a optimizer is needed to take step by step and reach the optimal. Since the neural networks are actually composition of functions, the back propagation is to apply the chain rule on these equations to find the optimal. The reason why we choose the functions that are easy to differentiate is because usually people use stochastic gradient descent or back propagation as optimizer. And these two ways both need to take gradient which means derivatives. By using easy-to-differentiate activation functions will make our life easier.

2.4 SVD

Sometimes, our input and output are too high dimensional that many of the dimensions are not necessary. In order to speed up computation time, we can apply SVD to reduce the dimension of the data matrix. All matrix can be decomposed into:

$$A = U\Sigma V^*$$

where U are the feature space and V is the matrix of the coefficients on each feature. Σ is a square matrix with singular values on the diagonal and it can tell us how many important components are in there by looking at the values. To get the reduced dimension data matrix, we can just do:

$$A_r = U_r * A$$

which projects the entire data matrix on a low dimension space. SVD can change the high-dimensional data to low-dimensional, while the important features are still remained.

3 Algorithm Implementation and Development

3.1 Part I

The dynamical system we need to predict in this part is Kuramoto-Sivashinsky (KS) equation. First we run to solve the equation and get the data. Since this data has space dimension of 1024 which is too large. I changed it to 64 to reduce the dimension. Since we wanted to predict the trajectory for $t + \Delta t$, I took the first to the second last in time as my input and second to last as output. Therefore, each data point has a corresponding point which is the location after Δt time. After building a neural network with three layers, I put the input and output into the network and trained it. To see whether the prediction is good or not, I change the initial condition from cosine to sine and generated the true trajectory. Then I took the initial condition and iterated using the trained network to generate the predicted trajectory.

3.2 Part II

In this part we need to predict the trajectory of reaction-diffussion reaction equation. We have u and v components, and each u and v is a image. First of all, I reshape each image into a column vector. For each time step, I stack u matrix on top of v matrix. Therefore, I have a tall skinny data matrix. Since this data matrix is high dimensional, I did SVD on that in order to reduce the dimension. I plotted the singular values and found out that only 2 of the components are the most important. I chose 2, 5, 10, and 30 as rank in order to test the results caused by different ranks. Then, I projected the original data matrix on a new space by multiplying with the reduced U matrix. I took the first to the second last as my input and second to the last as my output, and put them in the neural network and train. In order to test the result, I first randomly choose one image from the entire original matrix and run the trained net on that to get prediction for $t + \Delta t$. Then I chose the next one from original matrix as ground truth and compated.

3.3 Part III

In this part we are dealing with Lorenz equation. As we know, Lorenz system is a chaotic system; a small change in the initial condition will change the result a lot. Therefore, we generate the data 100 times to get a big data matrix. Also, in order to generalize the training of neural networks, I took three different values of ρ, which are 10, 28, and 40. Therefore, we have three sets of data in 3 dimensions and each set contains 100 subsets. In order to predict the trajectory, we take the first to the second last data as input and second to the last as output, and them put input and output into the net we built and train. In order to see how the predictions are, we randomly generate an initial condition and choose two different ρ values, and used these to do the test. Firstly, I used the ode45 command in MATLAB to generate the true trajectory; then, I used the net I trained to iterate and generated the predictions. Then, I compared the trajectory of ground truth and prediction.

For the second section of this part, we want to predict when the trajectory will jump to another lobe. At first, I plotted the trajectory and found that the division line between lobes is $y = 2 * x$. I took the same input as before, which is from the first to the second last point. The only thing changed is that I did not need z component. Therefore, I only have two columns in the input. For the output, I calculated where the trajectory was and labeled it. If it is above the $y = 2 * x$ line, I labeled it as 1, otherwise I labeled it as -1. Therefore, I have the coordinates for trajectory and which lobe it located. Since we are asked to predict when it will jump, which means given a point, we want to know how far it is from the jump point. Therefore, I counted backward and created another vector represented the distance. This distance vector is the output I used this time. I built a neural network and putted in input and output to train it. After training, I randomly generated an initial condition and compared the ground truth with the prediction.

4 Computational Results

4.1 Part I

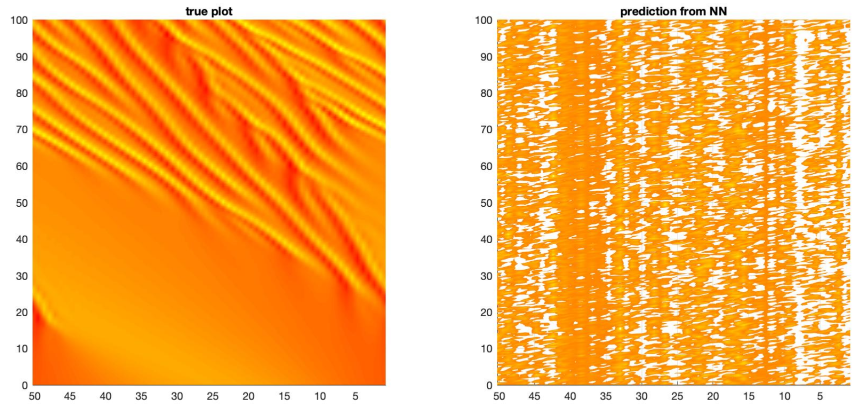


Figure 1: KS true trajectory vs prediction

Figure 1 shows the true KS trajectory and prediction. As we can see from the plot that the prediction is not very good. We can see the stripes from the prediction, but the pattern is not clear enough and it does not match with the true trajectory very much.

4.2 Part II

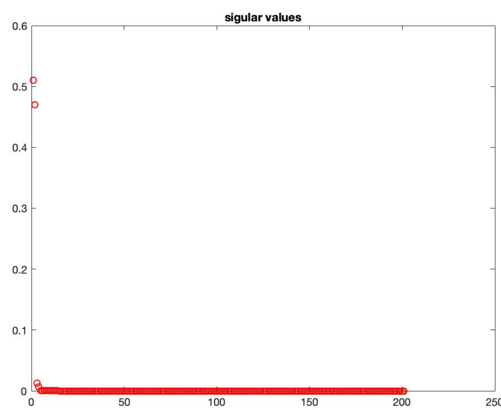


Figure 2: Singular values

Figure 2 shows the singular values from the original data matrix. As shown in this plot, only two components are important in this case.

As shown in Figure 3 Figure 4 and Figure 5 we can see that the predictions are all pretty good. With rank 2, the predictions already did very well. Rank 5 and 10 are not necessary in training neural net and predict.

Figure 6a and Figure 6b shows the comparison of the true trajectory and prediction. And Figure 7 shows the details of the comparison from x, y, and z direction separately. On the left side, it's the result for $\rho = 17.$ and right side is for $\rho = 35$.

Figure 8a shows the line $y = 2 * x$ which separated two lobes. And Figure 8b tells the time in advance that the trajectory will jump to another lobe for $\rho = 28$.

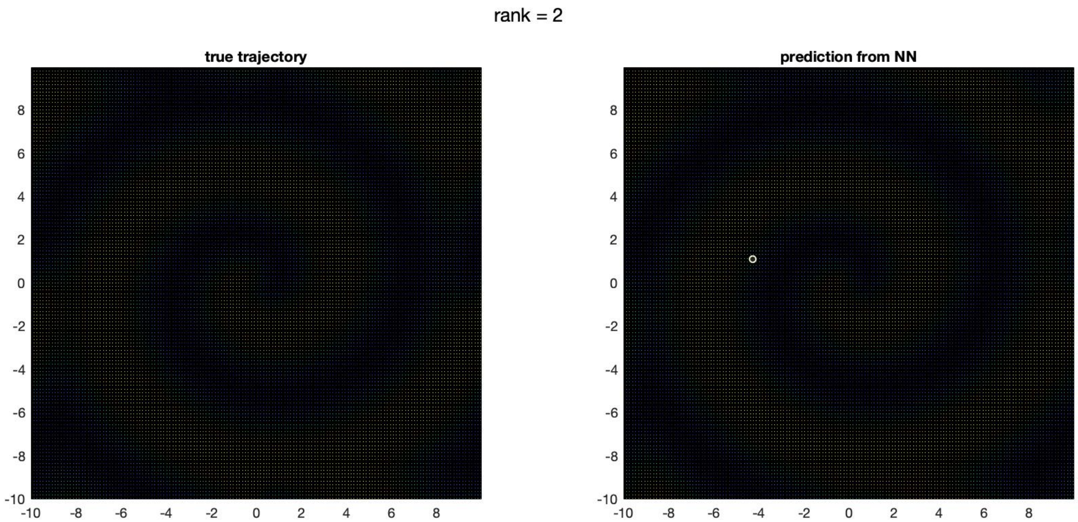


Figure 3: rank = 2

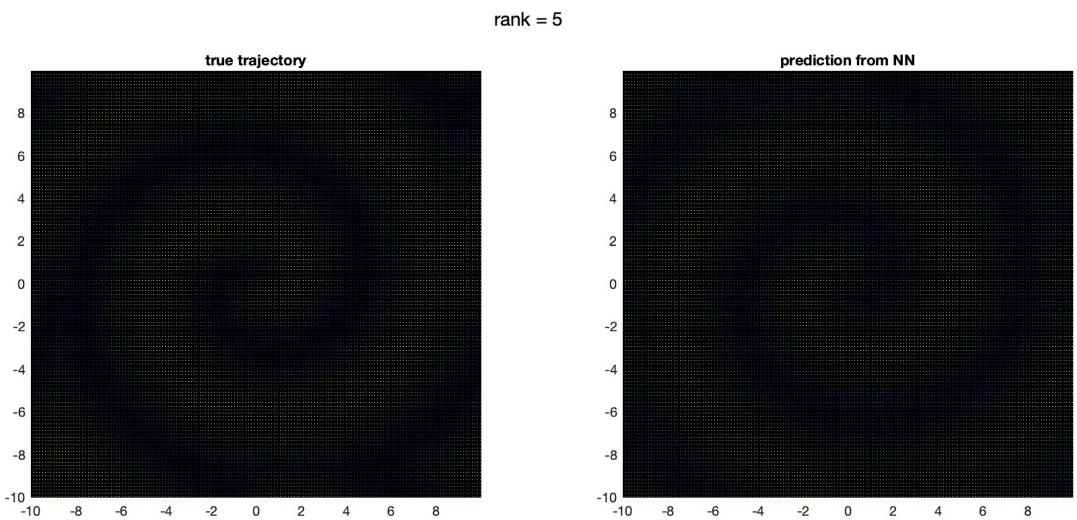


Figure 4: rank = 5

5 Summary and Conclusions

5.1 Part I

As we can see from this bad result, tuning parameters for a neural network is really time consuming and sensitive. A small change will affect the prediction a lot. Due the complexity of the dynamical system, using neural network to predict can be hard.

5.2 Part II

Since I stacked u and v together to train the neural network, the prediction is a little bit dark because it mixed u and v together. However, a pattern can still be seen from it. If I trained u and v separately, I will get a better result.

5.3 Part III

For this chaotic dynamical system, a small change in initial condition will change the trajectory a lot. Therefore, for small ρ we may have trajectory fly off far.

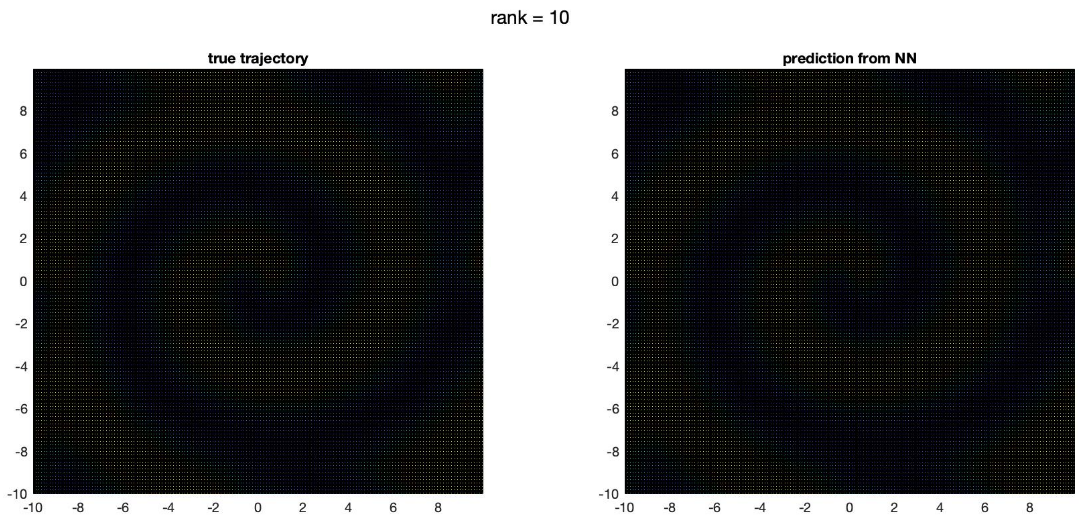
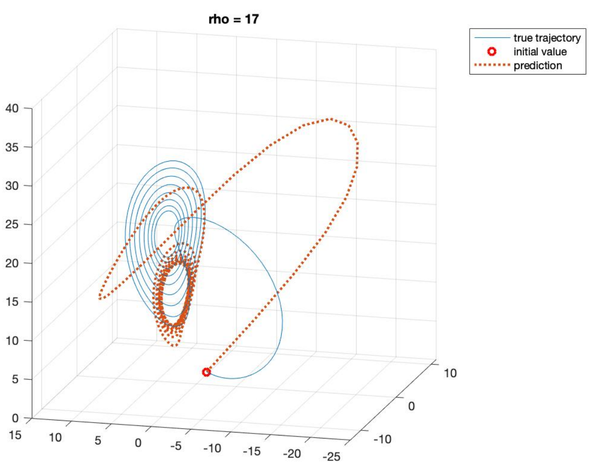
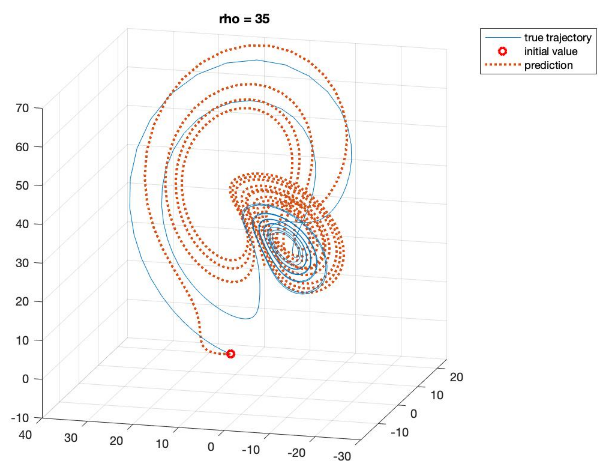


Figure 5: rank = 10



(a) $\rho = 17$



(b) $\rho = 35$

Figure 6: Lorenz Trajectories

From this homework we can see that predicting a real life problem is really hard. There are many factors affecting the result and a small change can affect the result a lot. Due to time constraints, only simple neural networks are used here. If it's possible, I would like to use more complicated networks, such as the convolutional neural network to train the KS equation in the future.

Appendix A Functions used

net = feedforwardnet([10 10 10]);

This command define the number of layers we want and the number of neurons we want on each layer.

net.layers1.transferFcn = 'logsig';

This command define the function we want to use for each layer. This "1" means first layer, and "logsig" shows we want to use the log-sig activation function.

net = train(net,input,output);

After building all the layers we want, we can combine them together and train our neural network by plugging in input and output.

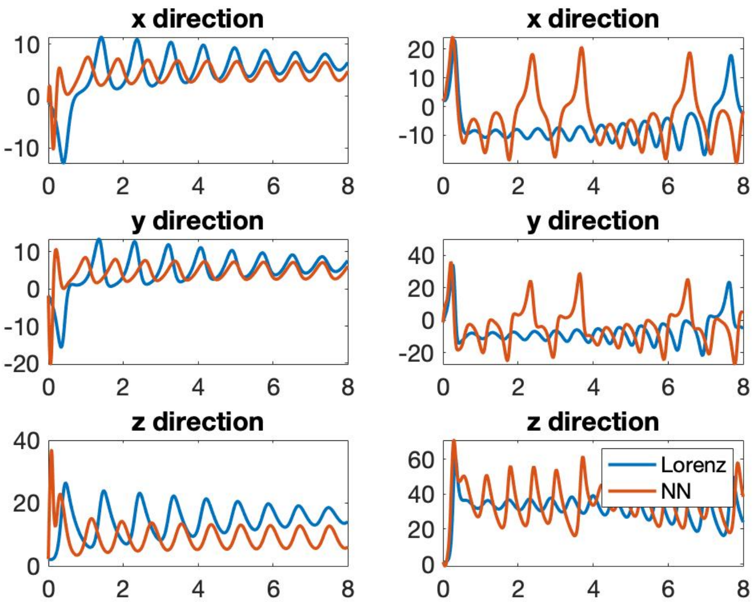
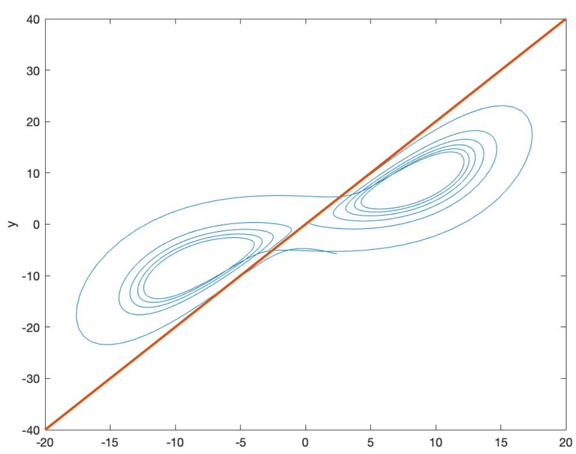
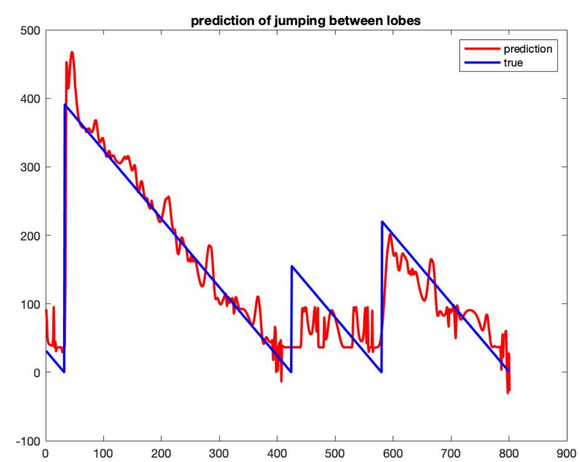


Figure 7: comparison for $\rho = 17$ and $\rho = 35$ in x,y,z direction



(a) division plane between lobes



(b) comparison between truth and prediction about jumping between lobes

Figure 8: Jumping

Appendix B MATLAB codes

Part I

```
clear all; close all; clc

% Kuramoto-Sivashinsky equation (from Trefethen)
% u_t = -u*u_x - u_xx - u_xxxx,  periodic BCs

N = 64;
x = 32*pi*(1:N)'/N;
u = cos(x/16).*(1+sin(x/16));
v = fft(u);

% % % % % %
%Spatial grid and initial condition:
h = 0.025;
k = [0:N/2-1 0 -N/2+1:-1]'/16;
L = k.^2 - k.^4;
E = exp(h*L); E2 = exp(h*L/2);
M = 16;
r = exp(1i*pi*((1:M)-.5)/M);
LR = h*L(:,ones(M,1)) + r(ones(N,1),:);
Q = h*real(mean( (exp(LR/2)-1)./LR ,2));
f1 = h*real(mean( (-4-LR+exp(LR).*(4-3*LR+LR.^2))./LR.^3 ,2));
f2 = h*real(mean( (2+LR+exp(LR).*(-2+LR))./LR.^3 ,2));
f3 = h*real(mean( (-4-3*LR-LR.^2+exp(LR).*(4-LR))./LR.^3 ,2));

% Main time-stepping loop:
uu = u; tt = 0;
tmax = 100; nmax = round(tmax/h); nplt = floor((tmax/250)/h); g = -0.5i*k;
for n = 1:nmax
t = n*h;
Nv = g.*fft(real(ifft(v)).^2);
a = E2.*v + Q.*Nv;
Na = g.*fft(real(ifft(a)).^2);
b = E2.*v + Q.*Na;
Nb = g.*fft(real(ifft(b)).^2);
c = E2.*a + Q.*(2*Nb-Nv);
Nc = g.*fft(real(ifft(c)).^2);
v = E.*v + Nv.*f1 + 2*(Na+Nb).*f2 + Nc.*f3; if mod(n,nplt)==0
        u = real(ifft(v));
uu = [uu,u]; tt = [tt,t]; end
end
% Plot results:
%surf(tt,x,uu), shading interp, colormap(hot), axis tight
% view([-90 90]), colormap(autumn);
%set(gca,'zlim',[-5 50])

%save('kuramoto_sivishinky.mat','x','tt','uu')

%%
%figure(2), pcolor(x,tt,uu.'),shading interp, colormap(hot),axis off

%% train NN
l = length(tt);
input = uu(:,1:l-1);
output = uu(:,2:l);
n = randi([1 250],1,50);
train_input = input(:,n);
train_output = output(:,n);

%%
```

```
net = feedforwardnet([128 64 64]);
net.layers{1}.transferFcn = 'logsig';
net.layers{2}.transferFcn = 'radbas';
net.layers{3}.transferFcn = 'purelin';
net = train(net,train_input,train_output);

%%
xt = 16*pi*(1:N)'/N;
ut = sin(xt/16).*(1+sin(xt/16));
vt = fft(ut);

uu_t = ut; tt_t = 0;
%unn(:,1)=ut;
unn = [ut];
tmax = 100; nmax = round(tmax/h); nplt = floor((tmax/250)/h); g = -0.5i*k;
for n = 1:nmax
t = n*h;
Nv = g.*fft(real(ifft(vt)).^2);
a = E2.*vt + Q.*Nv;
Na = g.*fft(real(ifft(a)).^2);
b = E2.*vt + Q.*Na;
Nb = g.*fft(real(ifft(b)).^2);
c = E2.*a + Q.*(2*Nb-Nv);
Nc = g.*fft(real(ifft(c)).^2);
vt = E.*vt + Nv.*f1 + 2*(Na+Nb).*f2 + Nc.*f3;
if mod(n,nplt)==0
utt = real(ifft(vt));
uu_t = [uu_t,utt];
tt_t = [tt_t,t];
% un=net(ut);
% unn=[unn,un];
% ut=un;
end

end
%Plot results:
subplot(1,2,1)
surf(tt_t,xt,uu_t), shading interp, colormap(hot), axis tight
title('true plot')
view([-90 90]), colormap(autumn);
set(gca,'zlim',[-5 50])

for i = 2:length(uu_t)
    un=net(ut);
    unn=[unn,un];
    ut=un;
end
subplot(1,2,2)
surf(tt_t,xt,unn), shading interp, colormap(hot), axis tight
title('prediction from NN')
view([-90 90]), colormap(autumn);
set(gca,'zlim',[-5 50])
```

Part II

```
clear all; close all; clc

% lambda-omega reaction-diffusion system
%  u_t = lam(A) u - ome(A) v + d1*(u_xx + u_yy) = 0
%  v_t = ome(A) u + lam(A) v + d2*(v_xx + v_yy) = 0
%
%  A^2 = u^2 + v^2 and
%  lam(A) = 1 - A^2
```

```
%   ome(A) = -beta*A^2


t=0:0.05:10;
d1=0.1; d2=0.1; beta=1.0;
L=20; n=512; N=n*n;
x2=linspace(-L/2,L/2,n+1); x=x2(1:n); y=x;
kx=(2*pi/L)*[0:(n/2-1) -n/2:-1]; ky=kx;

% INITIAL CONDITIONS

[X,Y]=meshgrid(x,y);
[KX,KY]=meshgrid(kx,ky);
K2=KX.^2+KY.^2; K22=reshape(K2,N,1);

m=1; % number of spirals

u = zeros(length(x),length(y),length(t));
v = zeros(length(x),length(y),length(t));

u(:,:,1)=tanh(sqrt(X.^2+Y.^2)).*cos(m*angle(X+i*Y)-(sqrt(X.^2+Y.^2)));
v(:,:,1)=tanh(sqrt(X.^2+Y.^2)).*sin(m*angle(X+i*Y)-(sqrt(X.^2+Y.^2)));

% REACTION-DIFFUSION
uvt=[reshape(fft2(u(:,:,1)),1,N) reshape(fft2(v(:,:,1)),1,N)].';
[t,uvsol]=ode45('reaction_diffusion_rhs',t,uvt,[],K22,d1,d2,beta,n,N);

%%
datau(:,1) = reshape(u(:,:,1),N,1);
datav(:,1) = reshape(v(:,:,1),N,1);
for j=1:length(t)-1
ut=reshape((uvsol(j,1:N).'),n,n);
vt=reshape((uvsol(j,(N+1):(2*N)).'),n,n);
u(:,:,j+1)=real(ifft2(ut));
v(:,:,j+1)=real(ifft2(vt));
datau(:,j+1) = reshape(u(:,:,j+1),N,1);
datav(:,j+1) = reshape(v(:,:,j+1),N,1);

figure(1)
pcolor(x,y,v(:,:,j+1)); shading interp; colormap(hot); colorbar; drawnow;
end

%save('reaction_diffusion_big.mat','t','x','y','u','v')

%%
% load reaction_diffusion_big
% pcolor(x,y,u(:,:,end)); shading interp; colormap(hot)

%%
data = [datau;datav];
[uu,ss,vv] = svd(data,'econ');
plot(ss/max(ss),'ro')
title('sigular values')
%%
rank = 30;
data_reduced = uu(:,1:rank).'*data;
%data_reduced = uu*ss(:,1:rank)*vv(:,1:rank).';
%data_reduced = uu(:,1:rank)*ss(1:rank,1:rank)*vv(1:rank,1:rank);
%data_reduced = uu(1:rank,:)*ss*vv(:,1:rank);
train_input = data_reduced(:,1:end-1);
train_output = data_reduced(:,2:end);

```

```
%% train NN
% n = randi([1 200],1,50);
% input = train_input(n,:);
% output = train_output(n,:);

net = feedforwardnet([10 10 10]);
net.layers{1}.transferFcn = 'logsig';
net.layers{2}.transferFcn = 'radbas';
net.layers{3}.transferFcn = 'purelin';
net = train(net,train_input,train_output);

%% test performance
t = randi([1 201],1,1);
test_in = data(:,t);
test_out = data(:,t+1);
l = length(test);

%NN prediction
projection = uu(:,1:rank).'*test_in; % project on reduced domain
next = net(projection);
prediction = uu(:,1:rank)*next; % project back to full domain
subplot(1,2,1)
pcolor(x,y,reshape(test_out(1:(l/2)),512,512));
title('true trajectory');
subplot(1,2,2)
pcolor(x,y,reshape(prediction(1:(l/2)),512,512));
title('prediction from NN');
suptitle('rank = 30')
```

Part III

```
clear all, close all

% Simulate Lorenz system
dt=0.01; T=8; t=0:dt:T;
b=8/3; sig=10;

Lorenz = @(t,x)([ sig * (x(2) - x(1))        ; ...
                  x(4) * x(1)-x(1) * x(3) - x(2) ; ...
                  x(1) * x(2) - b*x(3);
                  0]);
ode_options = odeset('RelTol',1e-10, 'AbsTol',1e-11);

figure(1)
input1=[]; output1=[];
for j=1:100  % training trajectories
    x0=[30*(rand(3,1)-0.5);10];
    [t,y] = ode45(Lorenz,t,x0);
    input1=[input1; y(1:end-1,:)];
    output1=[output1; y(2:end,:)];
    plot3(y(:,1),y(:,2),y(:,3)), hold on
    plot3(x0(1),x0(2),x0(3),'ro')
end
xlabel('x')
ylabel('y')
zlabel('z')
grid on, view(-23,18)

figure(2)
input2=[]; output2=[];
for j=1:100  % training trajectories
    x0=[30*(rand(3,1)-0.5);28];
    [t,y] = ode45(Lorenz,t,x0);
```

```
    input2=[input2; y(1:end-1,:)];
    output2=[output2; y(2:end,:)];
    plot3(y(:,1),y(:,2),y(:,3)), hold on
    plot3(x0(1),x0(2),x0(3),'ro')
end
xlabel('x')
ylabel('y')
zlabel('z')
grid on, view(-23,18)

figure(3)
input3=[]; output3=[];
for j=1:100  % training trajectories
    x0=[30*(rand(3,1)-0.5);40];
    [t,y] = ode45(Lorenz,t,x0);
    input3=[input3; y(1:end-1,:)];
    output3=[output3; y(2:end,:)];
    plot3(y(:,1),y(:,2),y(:,3)), hold on
    plot3(x0(1),x0(2),x0(3),'ro')
end
xlabel('x')
ylabel('y')
zlabel('z')
grid on, view(-23,18)

%%
n = randi([1 8000],1,100);
input = [input1(n,:);input2(n,:);input3(n,:)];
output = [output1(n,:);output2(n,:);output3(n,:)];

%%
net = feedforwardnet([10 10 10]);
net.layers{1}.transferFcn = 'logsig';
net.layers{2}.transferFcn = 'radbas';
net.layers{3}.transferFcn = 'purelin';
net = train(net,input.',output.');

%% p = 17
figure(6)
x0=[20*(rand(3,1)-0.5);17];
[t,y] = ode45(Lorenz,t,x0);
plot3(y(:,1),y(:,2),y(:,3)), hold on % ground truth
plot3(x0(1),x0(2),x0(3),'ro','Linewidth',[2]) % initial value
grid on

ynn(1,:)=x0;
for jj=2:length(t)
    y0=net(x0);
    ynn(jj,:)=y0.'; x0=y0;
end
plot3(ynn(:,1),ynn(:,2),ynn(:,3),':','Linewidth',[2])
legend('true trajectory','initial value','prediction')
title('rho = 17')

figure(7)
subplot(3,2,1), plot(t,y(:,1),t,ynn(:,1),'Linewidth',[2])
title('x direction')
subplot(3,2,3), plot(t,y(:,2),t,ynn(:,2),'Linewidth',[2])
title('y direction')
subplot(3,2,5), plot(t,y(:,3),t,ynn(:,3),'Linewidth',[2])
title('z direction')
```

```
% p = 35
figure(8)
x0=[20*(rand(3,1)-0.5);35];
[t,y] = ode45(Lorenz,t,x0);
plot3(y(:,1),y(:,2),y(:,3)), hold on
plot3(x0(1),x0(2),x0(3),'ro','Linewidth',[2])
grid on

ynn(1,:)=x0;
for jj=2:length(t)
    y0=net(x0);
    ynn(jj,:)=y0.'; x0=y0;
end
plot3(ynn(:,1),ynn(:,2),ynn(:,3),':','Linewidth',[2])
legend('true trajectory','initial value','prediction')
title('rho = 35')

figure(7)
subplot(3,2,2), plot(t,y(:,1),t,ynn(:,1),'Linewidth',[2])
title('x direction')
subplot(3,2,4), plot(t,y(:,2),t,ynn(:,2),'Linewidth',[2])
title('y direction')
subplot(3,2,6), plot(t,y(:,3),t,ynn(:,3),'Linewidth',[2])
title('z direction')
%suptitle('comparison between prediction and true trajectory')

figure(6), view(-75,15)
figure(8), view(-75,15)
figure(7)
subplot(3,2,1), set(gca,'Fontsize',[15],'Xlim',[0 8])
subplot(3,2,2), set(gca,'Fontsize',[15],'Xlim',[0 8])
subplot(3,2,3), set(gca,'Fontsize',[15],'Xlim',[0 8])
subplot(3,2,4), set(gca,'Fontsize',[15],'Xlim',[0 8])
subplot(3,2,5), set(gca,'Fontsize',[15],'Xlim',[0 8])
subplot(3,2,6), set(gca,'Fontsize',[15],'Xlim',[0 8])
legend('Lorenz','NN')

%%
r = 28;
inputx = [];outputx = [];
inputy = [];outputy = [];
%for j=1:100 % training trajectories
    x0=[30*(rand(3,1)-0.5);r];
    [t,y] = ode45(Lorenz,t,x0);
    inputx=[inputx; y(1:end-1,1)];
    inputy = [inputy; y(1:end-1,2)];
    outputx=[outputx; y(2:end,1)];
    outputy = [outputy; y(2:end,2)];
    %plot3(y(:,1),y(:,2),y(:,3)), hold on
    %plot3(x0(1),x0(2),x0(3),'ro')
    %xlabel('x')
    %ylabel('y')
    %zlabel('z')
%end

plot(inputx,inputy)
hold on
plot(-20:20,2.*[-20:20],'LineWidth',[2])
xlabel('x')
ylabel('y')
```

```
lobe = [];
for i = 1:length(inputx)
    if inputy(i) > 2*inputx(i)
        lobe = [lobe;1];
    else
        lobe = [lobe;-1];
    end

%     if output(i)*output(i+1) < 0
%         output(i+1) = -1;
%     else
%         output(i+1) = 1;
%     end
end

label = [1];
a = 0;
for i = length(lobe):-1:2
    if sign(lobe(i)*lobe(i-1)) == -1
        a = 0;
        label = [a;label]; % if jumps to another lobe, denote as 1
    else
        a = a+1;
        label = [a;label]; % if stays, denote as 0
    end

end
figure()
plot(label)
training = [outputx(1:length(label)) outputy(1:length(label))];

%%
net1 = feedforwardnet([10 10 10]);
net1.layers{1}.transferFcn = 'logsig';
net1.layers{2}.transferFcn = 'radbas';
net1.layers{3}.transferFcn = 'purelin';
net1 = train(net1,training.',label.');

%% test NN
x0=[30*(rand(3,1)-0.5);28];
[t,y] = ode45(Lorenz,t,x0);
performance = net1(y(:,1:2).');

lobe1 = [];
for i = 1:length(y(:,1))
    if y(i,2) > 2*y(i,1)
        lobe1 = [lobe1;1];
    else
        lobe1 = [lobe1;-1];
    end
end

label1 = [];
a = 0;
for i = length(lobe1):-1:2
    if sign(lobe1(i)*lobe1(i-1)) == -1
        a = 0;
        label1 = [a;label1]; % if jumps to another lobe, denote as 1
    else
        a = a+1;
        label1 = [a;label1]; % if stays, denote as 0
    end
```

```
end

plot(performance,'r','LineWidth',[2])
hold on
plot(label1,'b','LineWidth',[2])
legend('prediction','true')
title('prediction of jumping between lobes')
```