Modelling Dynamic Systems with Neural Networks

Ashley Batchelor

Abstract

I used neural networks to determine time stepping evolution of dynamic systems given an initial condition. I trained the neural networks with differential equations solved with MATLAB's ode45 solver for multiple random initial conditions.

Sec. I. Introduction and Overview

Neural Networks are a useful tool for determining the evolution of dynamic systems based on a set of training trajectories and then providing the neural network with an initial condition. For this exercise, we used known differential equations which we solved with numerical tools to compare the performance of neural network models. In general, neural networks are useful to model dynamics for systems in which the governing equations are unknown and actual measurements are the only way to compare a model to a dynamic system.

Sec. II. Theoretical Background

A lambda-omega reaction-diffusion (RD) system is defined by a system of two partial differential equations.

$$\frac{\partial u}{\partial t} = \lambda(A)u - \omega(A)v + d_1\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) = 0$$

$$\frac{\partial v}{\partial t} = \omega(A)u - \lambda(A)v + d_2\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) = 0$$

$$A^2 = u^2 + v^2$$

$$\lambda(A) = 1 - A^2$$

$$\omega(A) = -\beta A^2$$

The Kuramoto-Sivashinsky equation is a partial differential equation:

$$\frac{\partial U}{\partial t} + \nabla^4 U + \nabla^2 U + \frac{1}{2}|\nabla U|^2 = 0$$

The initial conditions are assumed to be periodic, over the domain of x.

The Lorenz system is defined by a system of three differential equations:

$$\frac{dx}{dt} = \sigma(y - x) \; \frac{dy}{dt} = x(\rho - z) - y \; \frac{dz}{dt} = xy - \beta z$$

A neural network may be trained to predict a trajectory from a time t to a time $t + \Delta t$, by providing a matrix of inputs x_n which represent the system at a series of times t and a matrix of inputs x_{n+1} which represent the system at a series of times $t + \Delta t$. The training data includes multiple trajectories for different randomly selected initial conditions. After training such a neural network, one may provide the network a set of initial conditions and the network will give a prediction for the next time step. Through recursion, by using that next step as initial conditions given to the network, a trajectory of the evolution of the system may be determined by the network.

Sec. III. Algorithm Implementation and Development

For the Kuramoto–Sivashinsky Equation, I created a set of periodic initial conditions by taking a sum of three harmonics of sin(x) with random weights and random phases. I created a set of trajectories as training data. To build a neural network, I used three hidden layers including a log-sigmoid transfer function, a radial basis transfer function, and a linear transfer function. Then I tested the neural network with a random set of initial conditions.

For the Lambda-Omega Reaction-Diffusion Equation, I created a set of periodic initial conditions for u and v like the KS Equation, with a sum of harmonics of sin(x) and sin(y) with random weights and phases.

This was a very large data set, so I projected the data for training trajectories including u and v to a common basis using Singular Value Decomposition (SVD). To begin, I vectorized u and v and stacked them into the same vector in series. I used these vectors as columns for a data matrix for many trajectories. I performed SVD on this data matrix and then looked at the values of the diagonals of the Σ matrix to determine the rank of the system corresponding to about 90% of the total energy. A rank of 15 gave 90% of the energy, but a rank of 18 gave a low rank initial state for the initial u that more closely resembled u in the original basis than the rank 15 system. I used the first 18 rows and columns of the Σ matrix multiplied by the first 18 columns of the V matrix to generate compressed low rank data that I could then separate into input and output data. To build a neural network, I used three hidden layers including a log-sigmoid transfer function, a radial basis transfer function, and a linear transfer function.

Then, I tested the neural network with a random set of initial conditions for u and v by projecting it into the low rank basis with the U matrix and time stepped the trajectory using the neural network. I projected the trajectory back to the original basis for comparison. I used the same initial conditions with a time stepping differential equation solver using Fast Fourier Transforms to create a trajectory to compare with the neural network solution.

For the Lorenz System, I first trained a neural network for trajectories with three different values $\rho = 10,17,28$ and random initial positions x_0, y_0, z_0. The input and output matrices included values for each x, y, z, and ρ of each trajectory. To build a neural network, I used three hidden layers including a log-sigmoid transfer function, a radial basis transfer function, and a linear transfer function. Then I tested the neural network with a random set of initial x, y, and z with two different values, $\rho = 17$ and $\rho = 35$ to test the network.

Next for the Lorenz System, I trained a neural network for $\rho = 28$ and random initial positions x_0, y_0, z_0. This time, for the output data, I added a label column that represented a time remaining until the next lobe transition. I defined a “transition” as when the trajectory crosses x=0. For each entry of positions x, y, and z in the input matrix, I calculated sign(x). Then I calculated an absolute value of the differential of that column which was zero except for immediately after a “transition,” in which case this value was 2. I then iterated backwards over the trajectory data and calculated the “time remaining” backwards starting from each value of 2. I discarded the data from the end of the input matrix from the time after the last transition since there was no way to determine the time to the next transition for those values. For each training trajectory, I used a longer length of time than the earlier Lorenz system exercise (T=20 vs. T=8) to ensure that I could provide enough data in the case of long lobe transition times.

I used the time to the next lobe transition (in increments of $\Delta t = 0.01$) as output labels corresponding to each x, y, and z coordinate of the training data trajectory. To build a neural network I used a single hidden layer with a very large number of nodes, typically 100 or more. To test the model, I used ode45 to create a trajectory with data for random initial states, applied the same analysis to determine the actual time to lobe transitions, and then used the x, y, and z values of this trajectory as inputs to my neural net to test how much time it predicted there would be until the next lobe transition.

Sec IV. Computational Results

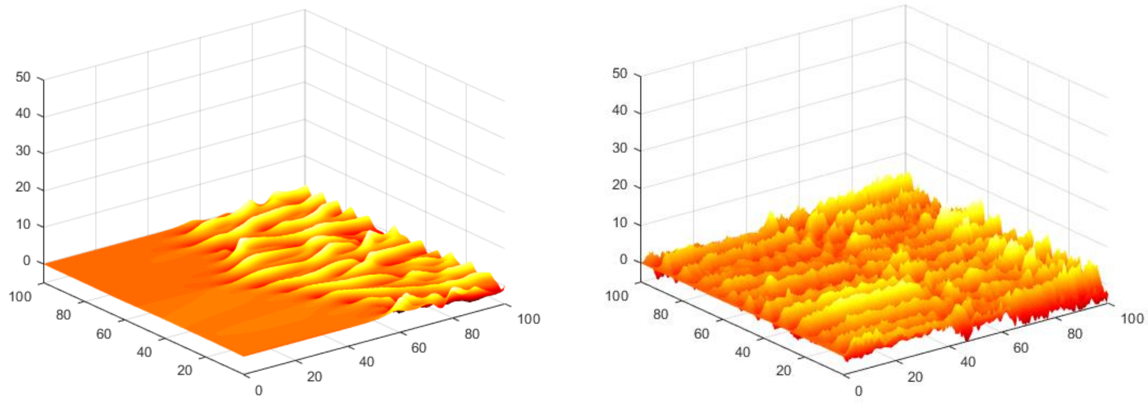

Figure 1 - KS solutions, FFT time stepper left, NN right with 100 training trajectories and three hidden layers with 80 nodes each.

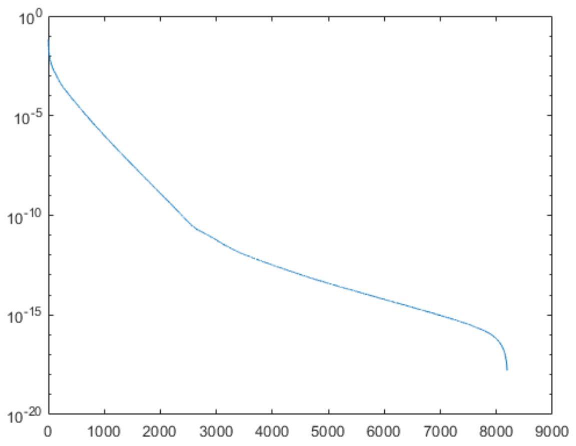

Figure 2 – Diagonals of Σ matrix for RD system.

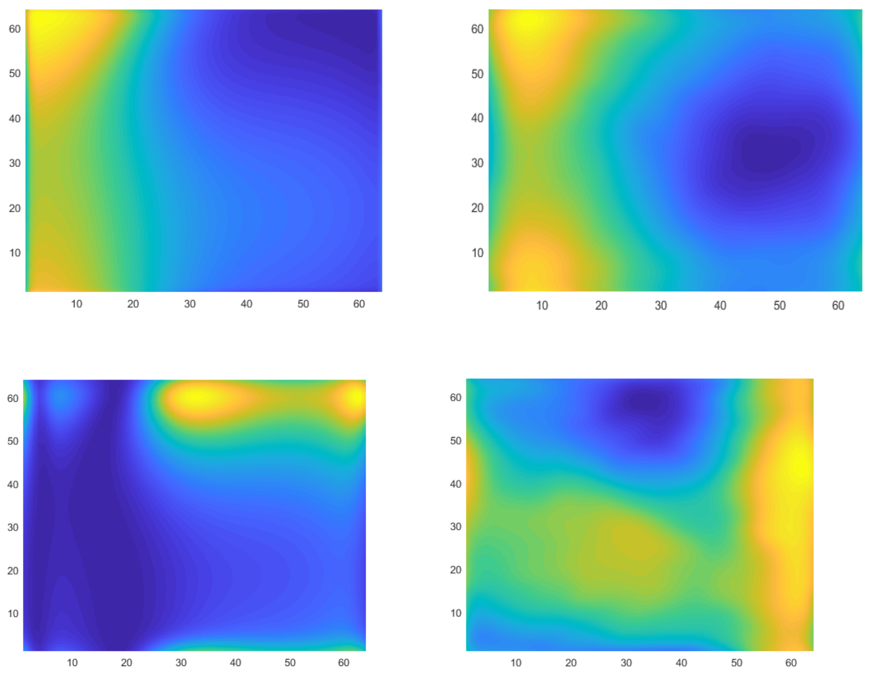

Figure 3 – RD system u state: top - after 10 time increments, ode solver left, NN right; bottom – after 100 time increments, ode solver left, NN right.

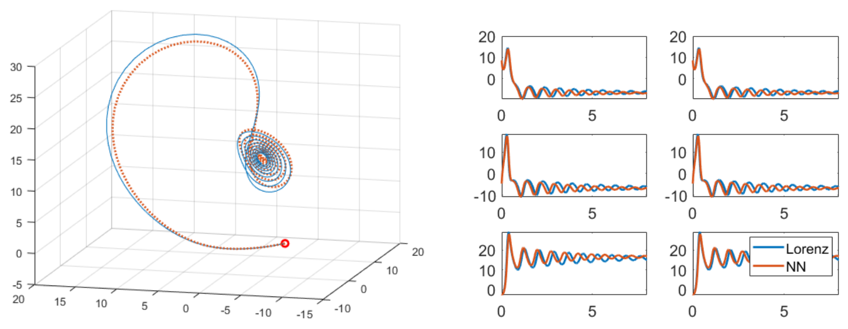

Figure 4 - Lorenz trajectories for ρ=17, blue is ode45 and red is NN solution. Right plot includes x,y,z components from top to bottom.

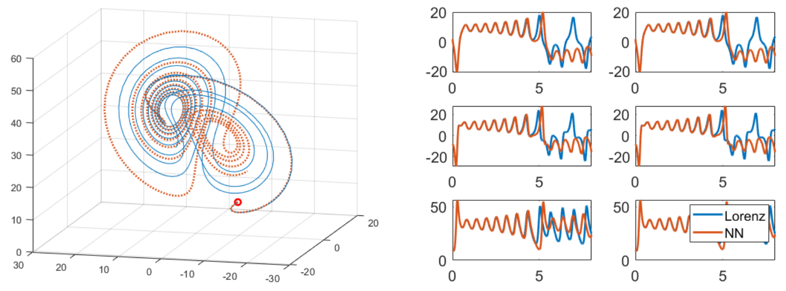

Figure 5 - Lorenz trajectories for ρ=35, blue is ode45 and red is NN solution. Right plot includes x,y,z components from top to bottom.

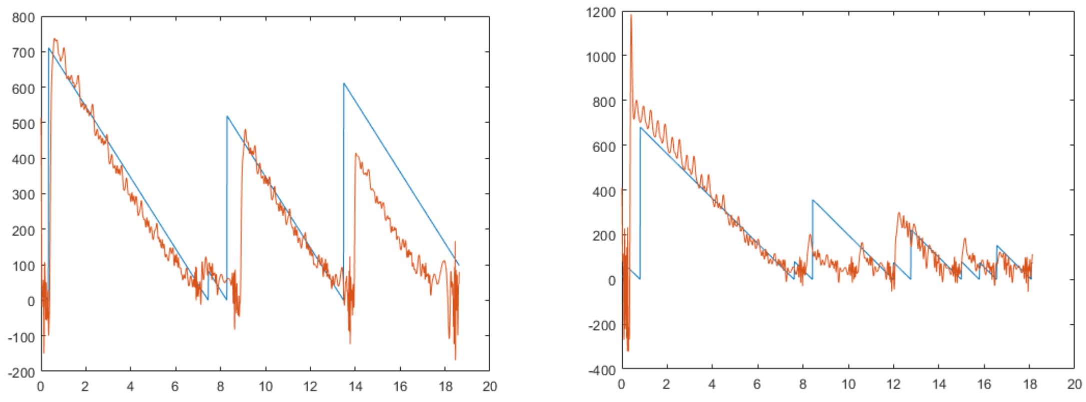

Figure 6 - Predicted time to next lobe transition for ρ=28. The NN is red, ode45 is blue. Vertical axis is time increments in units of Δt=0.01. First is for T=20, 50 nodes, 300 test trajectories. Second is for T=20, 250 nodes, 100 test trajectories.

For the KS system, most of my neural network trajectories reached a steady state very quickly. I had to use a high number of nodes, typically three hidden layers of 50 or more, to get a trajectory that did not rapidly turn into a steady state. The output with many hidden layers appeared to include a lot of high frequency noise.

For the RD system, generating the initial data took a long time. Fortunately, because we were using compressed data, it took less time to train a neural network. I pushed the limits of my GPU memory with the number of nodes I used. Similarly, generating the training data also resulted in running out of memory when I increased the number of trajectories to use for training. Both trajectories initially looked similar, e.g. at 10 time increments (in increments of $\Delta t = 0.05$) over a total range of t=10, but they looked quite different after 100 time increments. This is a system that pushes the limits of computing resources. Using SVD was an excellent way to use a larger amount of training data than we would have been able to use in a full rank system with the original training data.

For the Lorenz system, I was able to generate trajectories for $\rho = 17$ and $\rho = 35$ that closely followed the ode45 solutions. The $\rho = 17$ trajectory required more nodes and training data to produce good results. With more tuning I probably could have improved the performance of both.

The longest my model predicted a lobe transition in advance was about 7 units of T. For some parameters I had false positives, i.e. predicting a transition that did not happen, especially when I used larger numbers of nodes, more than 100. Perhaps this was a result of overfitting. In the end, adding more training data seemed to provide better results than adding more nodes.

Sec. V. Summary and Conclusions

With a sufficient number of nodes, I was able to train data to predict the behavior of the Lorenz system with a reasonable degree of accuracy, based on visual plots. The KS system was difficult to model and required a great amount of tuning to produce results that even vaguely resembled the test trajectories solved numerically from differential equations. The RD system was very slow for generating training data, but the neural networks trained very quickly, even for a high node number pushing the limits of memory available. Using a low rank basis was a vast improvement for training data. I was able to use many more trajectories than if I had used the full data. GPU processing immensely helped the speed of neural network training. In some cases, I was able to process data in 10 minutes that I had processed overnight with CPU processing. My GPU has less memory than my standard RAM (2 GB vs. 8 GB), so I could not use as large of data sets for GPU processing as I could for CPU processing. Nonetheless, my GPU memory was sufficient for the processing I needed.

Appendix A MATLAB functions used and brief implementation explanations

net.layers{1}.transferFcn – This specifies what type of transfer function to use for a hidden layer.

net.train() – Train a neural network for a given set of inputs and outputs with a specified set of parameters. I activated my GPU here with: `'useGPU','yes','showResources','yes'`

net.trainParam.epochs – This specifies the maximum number of epochs for training a neural network.

reshape() – This reshapes an array, e.g. a vector to a matrix or vice versa.

svd() – This calculates the three matrices of an SVD based on the input matrix.

Appendix B MATLAB codes

KS System

```
clear all; close all; clc

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

N = 128;
x = 32*pi*(1:N)'/N;
input=[];
output=[];

for j=1:250
%Randomize initial boundary value function
c1=rand/4; c2=rand/4; c3=rand/4; %Generate random harmonic coefficients
phi1=2*pi*(rand-0.5); phi2=2*pi*(rand-0.5); phi3=2*pi*(rand-0.5); %Generate
random phases
u=c1*cos(2*pi*x/max(x(:))+phi1)+c2*cos(4*pi*x/max(x(:))+phi2)+c3*cos(6*pi*x/m
ax(x(:))+phi3);
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;
%note nplt is divided by 250 in the original code.
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


input=[input;uu(:,1:end-1).'];
output=[output;uu(:,2:end).'];

end



%Train the neural net base on the random trajectories
net = feedforwardnet([60 60 60]);
net.layers{1}.transferFcn = 'logsig';
net.layers{2}.transferFcn = 'radbas';
net.layers{1}.transferFcn = 'purelin';
net.trainParam.epochs=1000;
net = train(net,input.',output.','useGPU','yes','showResources','yes');

%Plot a new trajectory with new initial conditions
save(['KS_data.mat'],'x','tt','uu', 'input','output','-v7','net');

c1=rand/4; c2=rand/4; c3=rand/4; c4=rand/4; %Generate random harmonic
coefficients
phi1=2*pi*(rand-0.5); phi2=2*pi*(rand-0.5); phi3=2*pi*(rand-0.5);
phi4=2*pi*(rand-0.5); %Generate random phases
u=c1*cos(2*pi*x/max(x(:))+phi1)+c2*cos(4*pi*x/max(x(:))+phi2)+c3*cos(6*pi*x/m
ax(x(:))+phi3);
uu=u;





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;
%note nplt is divided by 2500 in the original code.
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

unn=zeros(N,length(tt));
u=uu(:,1);
unn(:,1)=u;
for jj=2:length(tt)
    u0=net(u);
    unn(:,jj)=u0; u=u0;
end

figure(1)
surf(tt,x,uu), shading interp, colormap(hot), axis tight
% view([-90 90]), colormap(autumn);
set(gca,'zlim',[-5 50])

figure(2)
surf(tt,x,unn), shading interp, colormap(hot), axis tight
% view([-90 90]), colormap(autumn);
set(gca,'zlim',[-5 50]
```

RD System

```
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=64; 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;
trajectories=101;

allData=[];

for i=1:trajectories-1

% 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));

c1=rand; c2=rand; c3=rand; %Generate random harmonic coefficients
c4=rand; c5=rand; c6=rand;
phi1=2*pi*(rand-0.5); phi2=2*pi*(rand-0.5); phi3=2*pi*(rand-0.5);
phi4=2*pi*(rand-0.5); phi5=2*pi*(rand-0.5); phi6=2*pi*(rand-0.5);
u(:,:,1)=c1*cos(4*X/L+phi1)+c2*cos(2*X/L+phi2)+c3*cos(3*X/L+phi3)+...
    c4*cos(4*Y/L+phi4)+c5*cos(2*Y/L+phi5)+c6*cos(3*Y/L+phi6);

c1=rand; c2=rand; c3=rand;
c4=rand; c5=rand; c6=rand; %Generate random harmonic coefficients
phi4=2*pi*(rand-0.5); phi5=2*pi*(rand-0.5); phi6=2*pi*(rand-0.5);
v(:,:,1)=c1*cos(4*X/L+phi1)+c2*cos(2*X/L+phi2)+c3*cos(3*X/L+phi3)+...
    c4*cos(4*Y/L+phi4)+c5*cos(2*Y/L+phi5)+c6*cos(3*Y/L+phi6);

% 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);


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));



end

totalFrameInstance=[];
trajData=[];
%reshape u into a vector
for k=1:length(t);
    uFrame=squeeze(u(:,:,k));
    uVector=reshape(uFrame,[1,4096]);
    vFrame=squeeze(v(:,:,k));
    vVector=reshape(vFrame,[1,4096]);
    totalFrameInstance=[uVector,vVector];
    trajData=[trajData; totalFrameInstance];
end
%reshape u and v into vectors
allData=[allData;trajData];
% uInputentry=uData(1:end-numel(uVector));
% uOutputentry=uData(numel(uVector):end);
% vInputentry=vData(1:end-numel(vVector));
% vOutputentry=vData(numel(vVector):end);

% inputentry(:,:,:,2)=v(:,:,1:end-1);
% inputentry(:,:,:,1)=u(:,:,2:end);
% inputentry(:,:,:,2)=v(:,:,2:end);
% input=[input; inputentry];
% output=[output; outputentry];

end

% %Take svd of data matrix
allData=allData.';
[yu, s, vee] = svd(allData,'econ');

rank = 18;
%reducedData = sr*vee; %FIX THIS!
reducedData=s(1:rank,1:rank)*vee(:,1:rank).';
input=[];
output=[];
for i=1:trajectories-1
    inputentry=reducedData(:,(i-1)*length(t)+1:i*length(t)-1);
    input=[input; inputentry.'];
    outputentry=reducedData(:,(i-1)*length(t)+2:i*length(t));
    output=[output; outputentry.'];
end


%Generate NN for trajectory of low rank variables
net = feedforwardnet([50 50 50]);
net.layers{1}.transferFcn = 'logsig';
net.layers{2}.transferFcn = 'radbas';
net.layers{3}.transferFcn = 'purelin';
net.trainParam.epochs=10000;
net = train(net,input.',output.','useGPU','yes','showResources','yes');


c1=rand; c2=rand; c3=rand;
c4=rand; c5=rand; c6=rand; %Generate random harmonic coefficients
phi1=2*pi*(rand-0.5); phi2=2*pi*(rand-0.5); phi3=2*pi*(rand-0.5);
phi4=2*pi*(rand-0.5); phi5=2*pi*(rand-0.5); phi6=2*pi*(rand-0.5);
initialu=c1*cos(4*X/L+phi1)+c2*cos(2*X/L+phi2)+c3*cos(3*X/L+phi3)+...
    c4*cos(4*Y/L+phi4)+c5*cos(2*Y/L+phi5)+c6*cos(3*Y/L+phi6);
c1=rand; c2=rand; c3=rand;
c4=rand; c5=rand; c6=rand; %Generate random harmonic coefficients
phi1=2*pi*(rand-0.5); phi2=2*pi*(rand-0.5); phi3=2*pi*(rand-0.5);
phi4=2*pi*(rand-0.5); phi5=2*pi*(rand-0.5); phi6=2*pi*(rand-0.5);
initialv=c1*cos(4*X/L+phi1)+c2*cos(2*X/L+phi2)+c3*cos(3*X/L+phi3)+...
```

```
    c4*cos(4*Y/L+phi4)+c5*cos(2*Y/L+phi5)+c6*cos(3*Y/L+phi6);
uVector=reshape(initialu,[1,4096]);
vVector=reshape(initialv,[1,4096]);
testState=[uVector,vVector];
testStateMatrix=zeros(trajectories*length(t),length(testState));
testStateMatrix(1,:)=testState;
yu_r=yu(:,1:rank);
initialtestState=yu_r.'*testState.'; %project test state into low rank basis

%Generate trajectory from NN
trajNN=zeros(rank,length(t));
trajNN(:,1)=initialtestState;
trajNext=trajNN(:,1);
for jj=2:length(t)
    traj0=net(trajNext);
    trajNN(:,jj)=traj0.'; trajNext=traj0;
end

%Reshape low rank trajectories back into u and v
originalbasisDataNN=yu_r*trajNN;
uNNFrames=originalbasisDataNN(1:4096,:);
vNNFrames=originalbasisDataNN(4097:8192,:);
uNN=zeros(64,64,length(t));
vNN=zeros(64,64,length(t));
for i=1:length(t)
    uNN(:,:,i)=reshape(uNNFrames(:,i),[64,64]);
    vNN(:,:,i)=reshape(vNNFrames(:,i),[64,64]);
end



%Generate ode solver test trajectory
u = zeros(length(x),length(y),length(t));
v = zeros(length(x),length(y),length(t));
u(:,:,1)=initialu;
v(:,:,1)=initialv;

% REACTION-DIFFUSION SOLVER
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);



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));
end

figure(1)
pcolor(u(:,:,100)), shading interp
figure(2)
pcolor(uNN(:,:,100)),shading interp
```

Lorenz System – Variable ρ

```
lear all, close all

dt=0.01; T=8; t=0:dt:T;
b=8/3; sig=10; rho=[10,28,40];

testrho=17;


input=[]; output=[];
inputentry=zeros(800,4);
for i=1:3
    r=rho(i);
for j=1:100  % training trajectories
    x0=30*(rand(3,1)-0.5);
    Lorenz = @(t,x)([ sig * (x(2) - x(1))       ; ...
                  r * x(1)-x(1) * x(3) - x(2) ; ...
                  x(1) * x(2) - b*x(3)         ]);
    ode_options = odeset('RelTol',1e-10, 'AbsTol',1e-11);
    [t,y] = ode45(Lorenz,t,x0);
    inputentry(:,1:3)=y(1:end-1,:);
    inputentry(:,4)=rho(i);
    input=[input; inputentry];
    output=[output; y(2:end,:)];
end
end

%%
net = feedforwardnet([20 20 20]);
net.layers{1}.transferFcn = 'logsig';
net.layers{2}.transferFcn = 'radbas';
net.layers{3}.transferFcn = 'purelin';
net = train(net,input.',output.');



%%
figure(2)
x0=20*(rand(3,1)-0.5);
Lorenz = @(t,x)([ sig * (x(2) - x(1))       ; ...
                  testrho * x(1)-x(1) * x(3) - x(2) ; ...
                  x(1) * x(2) - b*x(3)         ]);
    ode_options = odeset('RelTol',1e-10, 'AbsTol',1e-11);
    [t,y] = ode45(Lorenz,t,x0);
[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

state=zeros(4,1);
ynn(1,:)=x0;
for jj=2:length(t)
    state(1:3)=x0;
    state(4)=testrho;
    y0=net(state);
    ynn(jj,:)=y0.'; x0=y0;
end
```

```
plot3(ynn(:,1),ynn(:,2),ynn(:,3),':','Linewidth',[2])

figure(3)
subplot(3,2,1), plot(t,y(:,1),t,ynn(:,1),'Linewidth',[2])
subplot(3,2,3), plot(t,y(:,2),t,ynn(:,2),'Linewidth',[2])
subplot(3,2,5), plot(t,y(:,3),t,ynn(:,3),'Linewidth',[2])


figure(3)
subplot(3,2,2), plot(t,y(:,1),t,ynn(:,1),'Linewidth',[2])
subplot(3,2,4), plot(t,y(:,2),t,ynn(:,2),'Linewidth',[2])
subplot(3,2,6), plot(t,y(:,3),t,ynn(:,3),'Linewidth',[2])

%%
figure(2), view(-75,15)
figure(3)
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')
```

Lorenz System – Lobe Transitions

```
clear all, close all

% Simulate Lorenz system
dt=0.01; T=20; t=0:dt:T;
b=8/3; sig=10;

input=[]; output=[];
outputentry=zeros(numel(t(:)),6);


    r=28;
for j=1:300  % training trajectories
    x0=30*(rand(3,1)-0.5);
    Lorenz = @(t,x)([ sig * (x(2) - x(1))       ; ...
                  r * x(1)-x(1) * x(3) - x(2) ; ...
                  x(1) * x(2) - b*x(3)         ]);
    ode_options = odeset('RelTol',1e-10, 'AbsTol',1e-11);
    [t,y] = ode45(Lorenz,t,x0);
    outputentry(:,1:3)=y(1:end,:);
    outputentry(1,4)=sign(x0(1));
    outputentry(1,5)=2;
    for k=2:numel(t(:))-1
      outputentry(k,4)=sign(outputentry(k,1));
      outputentry(k,5)=outputentry(k,4)-outputentry(k-1,4);
    end
    outputentry(numel(t(:))-1,6)=0;
    for k=numel(t(:))-2:-1:1
      if abs(outputentry(k,5))==2
```

```
          outputentry(k,6)=0;
      else
          outputentry(k,6)=outputentry(k+1,6)+1;
      end
    end
    for k=numel(t(:))-2:-1:1
      if abs(outputentry(k,5))==2
        trajectorysize=k-1;
        break
      end
    end
    inputentry=zeros(trajectorysize,4);
    outputentryadjusted=zeros(trajectorysize,1);
    inputentry=y(2:trajectorysize+1,:);
    outputentryadjusted(:,1)=outputentry(2:trajectorysize+1,6);
    input=[input; inputentry];
    output=[output; outputentryadjusted];
end


net = feedforwardnet(60);
% net.layers{1}.transferFcn = 'logsig';
% net.layers{2}.transferFcn = 'radbas';
% net.layers{3}.transferFcn = 'radbas';
net.trainParam.epochs=10000;
net = train(net,input.',output.','useGPU','yes','showResources','yes');


figure(2)
x0=30*(rand(3,1)-0.5);
Lorenz = @(t,x)([ sig * (x(2) - x(1))       ; ...
                  r * x(1)-x(1) * x(3) - x(2) ; ...
                  x(1) * x(2) - b*x(3)         ]);
    ode_options = odeset('RelTol',1e-10, 'AbsTol',1e-11);
[t,y] = ode45(Lorenz,t,x0);

for jj=1:length(t)
    x0=y(jj,:);
    y0=net(x0.');
    ytimer(jj,:)=y0;
end

ytimeActualEntry=zeros(numel(t(:)),6);
ytimeActualEntry(:,1:3)=y(1:end,:);
    ytimeActualEntry(1,4)=sign(x0(1));
    ytimeActualEntry(1,5)=2;
    for k=2:numel(t(:))-1
      ytimeActualEntry(k,4)=sign(ytimeActualEntry(k,1));
      ytimeActualEntry(k,5)=ytimeActualEntry(k,4)-ytimeActualEntry(k-1,4);
    end
    ytimeActualEntry(numel(t(:))-1,6)=0;
    for k=numel(t(:))-2:-1:1
      if abs(ytimeActualEntry(k,5))==2
          ytimeActualEntry(k,6)=0;
      else
          ytimeActualEntry(k,6)=ytimeActualEntry(k+1,6)+1;
      end
    end
    for k=numel(t(:))-2:-1:1
      if abs(ytimeActualEntry(k,5))==2
        trajectorysize=k-1;
        break
      end
    end
yTimeActual=zeros(trajectorysize,1);
yTimeActual(:,1)=outputentry(2:trajectorysize+1,6);
plot(t(1:numel(yTimeActual)),yTimeActual);
hold on
plot(t(1:numel(yTimeActual)),ytimer(1:numel(yTimeActual)));
```