Neural Nets Applications in Dynamical Systems Replication and Prediction

Maple Tan
AMATH563 Homework 4

May 20, 2019

1 Abstract

Use neural nets to replicate and predict certain dynamical systems.

2 Introduction and Overview

We generate data for dynamical systems such as the Lorenz Attractor, Kuramoto-Sivashinsky equation, and Lambda-Omega Reaction-Diffusion equations to train our neural nets on. The training set consists of an input set which has the first $n-1$ data points and an output set which consists of the 2nd through n data sets. The neural net will find a way to map a data point to the data point one time step ahead. Furthermore, a test data set has been kept separate from the training set and will be used to test our neural net. Since we know the dynamics of the systems, we can visually see how the neural net's predictions compare to the actual dynamics.

3 Theoretical Background

Neural Nets are an interconnected group of nodes organized into layers. There is usually a input layer that contains the inputted data, a series of hidden layers and an output layer containing the proper labels. Each node has a connection to other node in the layer over, with a function that take the data from one layer to the other. The process of training a neural net involves solving an optimization problem that tries to minimize the error between the true solution and the predicted, typically using a gradient descent method. Due to the nature of a neural net, it must be trained on a labeled set and so it is a supervised method of machine learning.

A neural network is a flexible, and has many ways to modify its architecture. For example, one can choose the number of hidden layers, the function that connects one layer to another (activation), the error evalution (loss), etc. As such, a neural net can be modified

to suit many types of data but also requires a lot of tweaking to achieve good results.

As for the data itself, we will look at three different dynamical systems. Starting with the Lorenz Attractor, which is governed by the following equations:

$$\begin{aligned}\frac{dx}{dt} &= \sigma(y-x)\\ \frac{dy}{dt} &= x(r-z)-y\\ \frac{dz}{dt} &= xy-By\end{aligned}$$

Then the Kuramoto-Sivashinsky equation, governed by:

$$\frac{d}{dt}u = -u\frac{d}{dx}u - \frac{d^2}{dx^2}u - \frac{d^4}{dx^4}u$$

Finally the reaction diffusion equations:

$$\begin{aligned}\frac{d}{dt}u &= (1-A)u - \beta A^2 v + d_1 * (\frac{d^2}{dx^2}u + \frac{d^2}{dy^2}u) = 0\\ \frac{d}{dt}v &= -\beta A^2 u + (1-A)v + d_2 * (\frac{d^2}{dx^2}v + \frac{d^2}{dy^2}v) = 0\end{aligned}$$

We will generate data based off these systems and see if the neural net can replicate those dynamics.

For the last dynamical system in particular, we can not work directly with the data set due to its size. We will use an $r - rank$ truncation and projection into the principal component space instead. An r-rank truncation of a matrix X uses the SVD to provide a way to approximate the original matrix. While in the original SVD, a matrix X might be written:

$$\begin{aligned}X = \quad U \quad \Sigma \quad V^*\\ [n\times m] = [n\times n][n\times n][n\times m]\end{aligned}$$

A r-rank truncation can give a good approximate of the original matrix without having utilizing the entirety of the matrices from the SVD:

$$\begin{aligned}X = \quad U_r \quad \Sigma_r \quad V_r^*\\ [n\times m] = [n\times r][r\times r][r\times m]\end{aligned}$$

4 Algorithm Implementation and Development

To create our Neural Net, we will use a library called `keras`. Using the `Sequential()` method to initialize the model and then the `add` method to add layers, where we can

determine the number of nodes and the activation in that layer as well. Unless otherwise stated, the primary architecture used for the neural net is one input layer and one hidden layer, both with 128 nodes and 'relu' activation, and an output layer that has the number of nodes correspond with the number of outputs in the problem and no activation. To train, we use the `fit` method with 50 epochs (iterations), mean squared error as the loss and a batch size of 32.

Starting with the Lorentz Attractor, the data is generated using the `ode45` command in MATLAB, which solves our Lorenz system for the trajectory from a specific initial condition. Varying the r value and the initial conditions results in different trajectories that we will append to the end of our data set after each solve. We solve the system using three different r values: $[10, 28, 40]$ that will start at 150 different initial conditions. This results in a three columned matrix, one for x, y and z with n time steps. We then separate our data matrix into two difference matrices: the first being an input matrix, which consists of the data from $t = 0$ to $n - 1$ time steps and we add fourth column showing the corresponding value of r and the second being an output matrix which consists of the data from $t = 1$ to n time steps. We then train our neural net on this input and output data.

To check the accuracy of our neural net, we generate some test data for $r = 17, 35$ that will be kept separate from the training data. Starting with the initial point in both trajectories, we then use the `predict` method to predict where the initial point goes in the next time step, making sure to wrap the initial point in brackets [ ]. Then using the guess from the neural net, we use that to guess our next point and so on. To compare the accuracy of our neural net, we then graph the trajectories of our predicted versus actual.

Continuing on, we then manipulate the trajectories to record the time it takes before it crosses the $x = 0$ plane. This is equivalent to the trajectory transitioning from one lobe to the other. In this case, rather than having all the trajectories in one big matrix, we only look at the x positions and separate each trajectory into its own columns, taking out one column for the test data. We then train the neural net with 1024 nodes per a layer for 200 epochs with the input being the x position of each separate trajectory and the output being the time it takes before it jump lobes. Finally, we use the test data to predict and compare the performance of the neural net.

Similarly with the Kuramoto-Sivashinsky equation, we randomize the initial conditions (the Fourier coefficients in this case, `f1`, `f2`, and `f3` in the MATLAB script). Record a number of dynamics and append the dynamics to the end of our large data matrix. In this case, column wise is the spatial component of our data and row wise is the time component of our data. Plugging this data into our neural net, we then train the data under the same conditions as the Lorenz data, with 256 nodes per a layer instead. Again in a similar manner, we run a separate random initial condition and store that away for our testing data to evaluate the accuracy of our neural net.

Lastly the final data set, the diffusion reactions. We generate one iteration of the data and will use this to train our model. Unlike the previous examples, the data is a 3D matrix of frames that depict the diffusion. Starting by reshaping each frame into a column vector and compiling into a large matrix, we then perform the SVD on the resulting matrix. Graphing the singular values, we see that the most of the variation exists in first two principal components. We then create a rank-2 truncation of our data matrix and use that to project the data into the principal component space:

$$U_r^* X = \Sigma_r V_r^*$$

The projected data, which is a 2×201 matrix where 2 represents rank and 201 being the number of time steps, is then used to train our model. In this case, the inputs are the 2×1 entries of the projected data and the output is the 2×1 entry one time step ahead. We train for 500 epochs in this case. For the test case, we multiply it by the U_r^* created from the training data to project it into the principal component space. Once projected, it also results in a 2×201 matrix, we use the first 2×1 entry of that matrix to predict the next time step and then our prediction to predict the time step after that and so on.

5 Computational Results

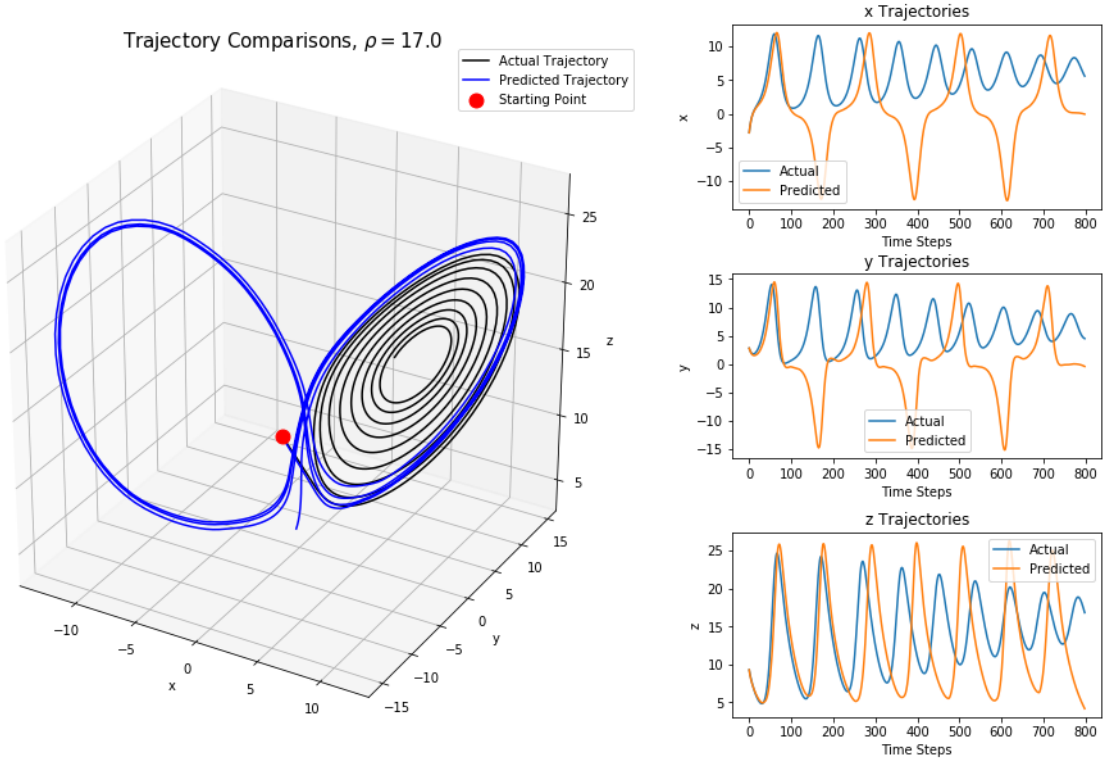

Figure 1: Here We see the predicted trajectory match pretty well up until 250 time steps. Afterwards, even though some dynamics are similar, the x and y trajectories completely diverge.

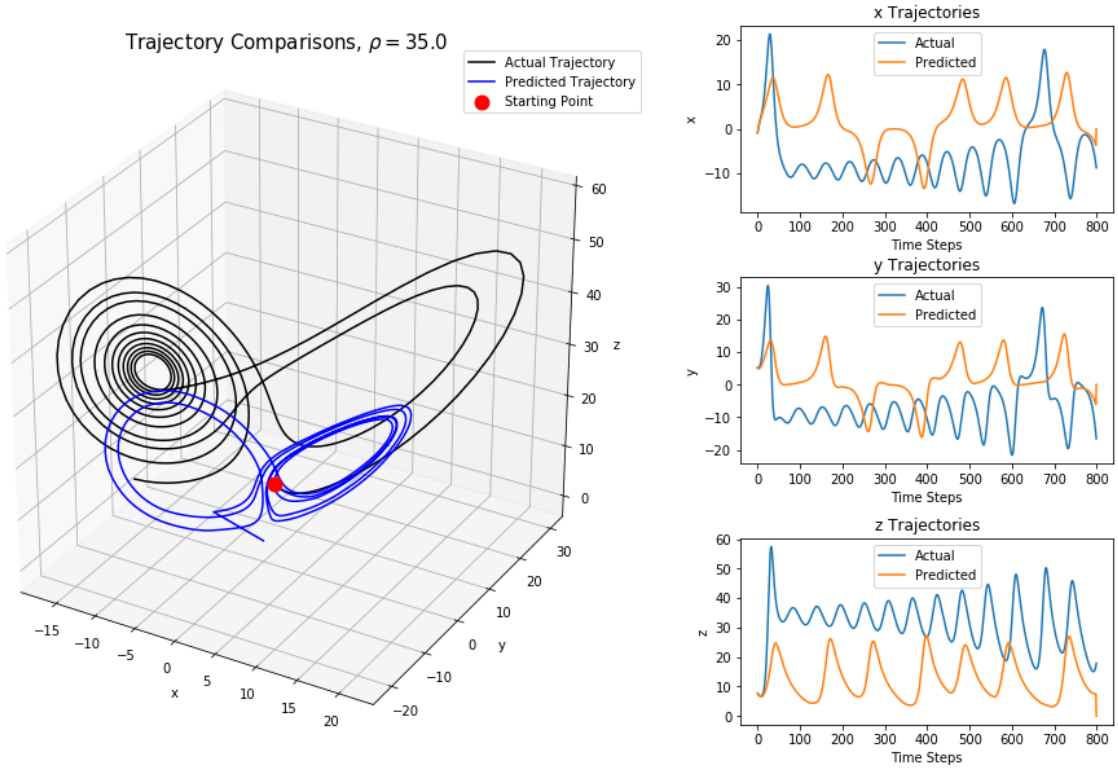

Figure 2: Similarly, the dynamics are similar but there is no matching in this case.

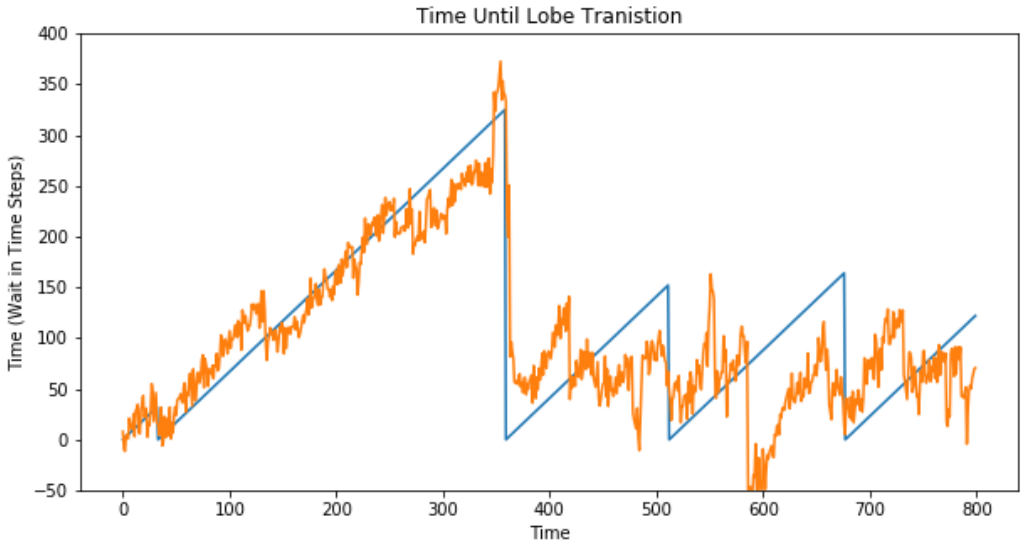

Figure 3: The predicted dynamics follow fairly closely up until about 350 time steps in, where it starts diverging. Similar to the $\rho = 17$ case, it matches fairly well for a set amount of time steps before diverging.

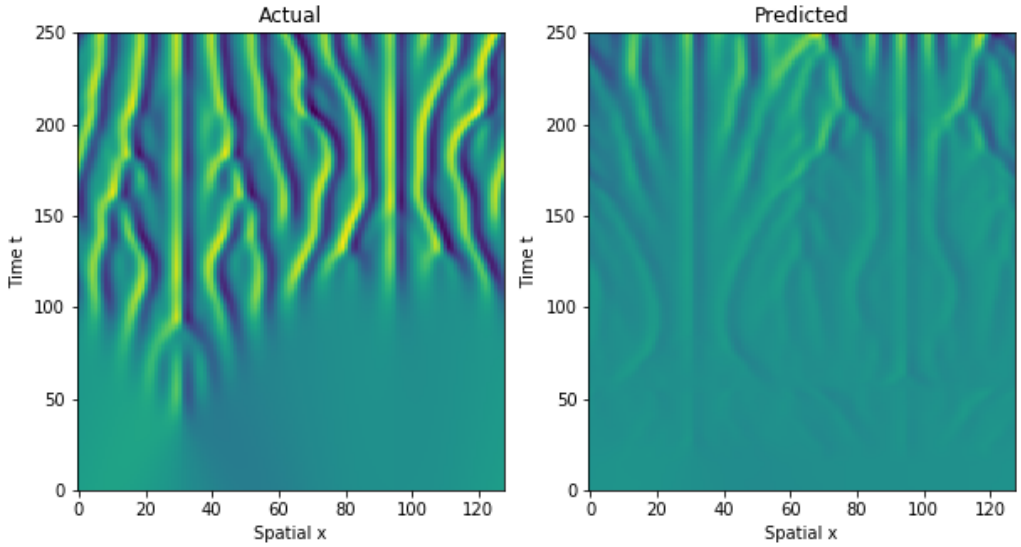

Figure 4: Near the top of the frame, we see that the dynamics are actually fairly similar but it soon diverges after 50 time steps or so. We also see that certain dynamics such as the straight lines around $x = 35$ and 95 are also preserved

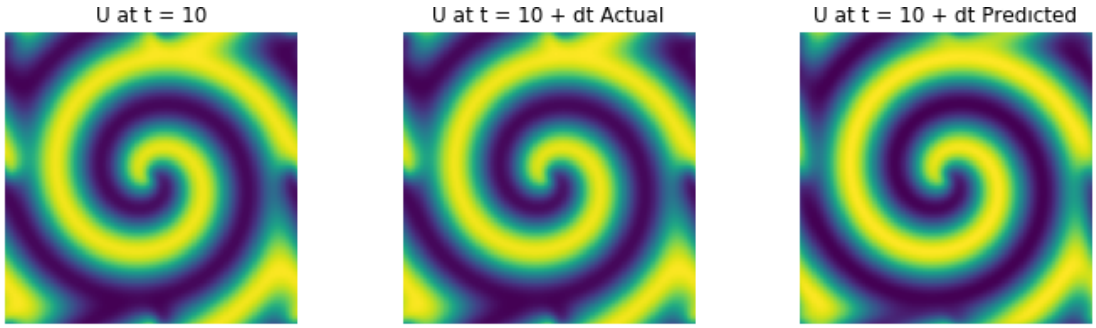

Figure 5: The predicted frame is actually fairly similar to the actual frame. We see a little difference in the values (i.e. slightly different colors) but otherwise, the prediction is fairly close.

6 Summary and Conclusions

Neural nets can predict the dynamics of the system extremely well but can be very finicky when it comes to its architecture, training time, etc. Often times small tweaks are necessary to generate good results and often times, these accurate results only last for a limited time frame.

A MATLAB Code - Generate Data

A.1 Lorenz Attractor

```
clear all; close all; clc
% How to use neural nets in solving/predicting dynamical systems
% Simulate Lorenz system
input = [];
output = [];
rs = [10, 28, 40];
for k = 1:3
    dt=0.01; T=8; t=0:dt:T; % defining one trajectory
    b=8/3; sig=10; r=rs(k);

    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);
    % Get really accurate trajectories from this

    % The goal is to find a mapping that will take our system from time t to
    % time t + dt. This will give us an idea about the future state of the
    % system.

    for j=1:150 % training trajectories
        x0=30*(rand(3,1)-0.5);
        [t,y] = ode45(Lorenz,t,x0);
        input=[input; [y(1:end-1,:),ones(800,1)*r]];
        output=[output; [y(2:end,:)]];
    end
end

save('lorenz.mat', 'input', 'output')
%%
close all;
input2 = [];
output2 = [];
rs2 = [17, 35];
for k=1:2 % training trajectories
        dt=0.01; T=8; t=0:dt:T; % defining one trajectory
    b=8/3; sig=10; r=rs2(k);

    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);
    x0=30*(rand(3,1)-0.5);
    [t,y] = ode45(Lorenz,t,x0);
    input2=[input2; [y(1:end-1,:),ones(800,1)*r]];
    output2=[output2; [y(2:end,:)]];
end
save('lorenz_test.mat', 'input2', 'output2')

plot3(input2(1:800,1), input2(1:800,2), input2(1:800,3))
hold on
plot3(input2(801:end,1), input2(801:end,2), input2(801:end,3), 'r')
legend('17','35')
```

A.2 Kuramoto-Sivashinky

```
clear all; close; clc

% Kuramoto-Sivashinsky equation (from Trefethen)
% u_t = -u*u_x - u_xx - u_xxxx,  periodic BCs
input = [];
output = [];
for its = 1:300
    N = 128;
    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;
    h = rand(1)*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 = rand(1)*h*real(mean( (-4-LR+exp(LR).*(4-3*LR+LR.^2))./LR.^3 ,2));
    f2 = rand(1)*h*real(mean( (2+LR+exp(LR).*(-2+LR))./LR.^3 ,2));
    f3 = rand(1)*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

    input = [input uu(:, 1:250)];
    output = [output uu(:, 2:251)];

end

save('kuramoto_sivishinky.mat','input','output', 'tt', 'x')
%% Create Test Data

N = 128;
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;
h = rand(1)*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 = rand(1)*h*real(mean( (-4-LR+exp(LR).*(4-3*LR+LR.^2))./LR.^3 ,2));
f2 = rand(1)*h*real(mean( (2+LR+exp(LR).*(-2+LR))./LR.^3 ,2));
f3 = rand(1)*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

input_test = uu(:, 1:250);
output_test = uu(:, 2:251);

save('ks_test.mat','input_test','output_test', 'tt', 'x')

% Plot results:
surf(tt,x,uu), shading interp, colormap(hot), axis tight
xlabel('t')
ylabel('x')
zlabel('uu')
% 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)
xlabel('x')
ylabel('t')
```

A.3 Reaction Diffusion

```
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=0:0.05:12;
d1=0.1; d2=0.1; beta=1.0;
L=20; n=256; 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);


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));

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)
```

B Python Code - Neural Net

```
# -*- coding: utf-8 -*-
"""
Created on Tue May 14 11:42:33 2019

@author: Maple
"""
# %%
from scipy.io import loadmat
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras import optimizers
import numpy as np
import matplotlib.pyplot as plt
import numpy.linalg as la
from mpl_toolkits.mplot3d import Axes3D


"""
Receives 2 matrices of x,y,z coordinates and graphs the
    trajectory of both to compare the predicted against actual.
"""
def plot_trajectory(actual, predicted, file_name):
    fig = plt.figure(figsize=(10,10))
    rho = actual[0,3]
    ax = plt.axes(projection='3d')
    ax.plot3D(actual[:,0], actual[:,1], actual[:,2], c='black')
    ax.plot3D(predicted[:,0], predicted[:,1], predicted[:,2], '-', c = 'blue')
    ax.scatter(actual[0,0], actual[0,1], actual[0,2], c='red', s = 120)
    ax.legend(['Actual Trajectory','Predicted Trajectory', 'Starting Point'])
    ax.set_title(r'Trajectory Comparisons, $\rho = {}$'.format(rho), fontsize=15, pad=4
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_zlabel('z')

    n = np.shape(actual)[0]
    ax1 = fig.add_axes([1, 0.7, 0.4, 0.2])
    ax1.plot(np.arange(n), actual[:,0])
```

```
    ax1.plot(np.arange(n), predicted[:,0])
    ax1.legend(['Actual', 'Predicted'])
    ax1.set_xlabel('Time Steps')
    ax1.set_ylabel('x')
    ax1.set_title('x Trajectories')

    ax2 = fig.add_axes([1, 0.43, 0.4, 0.2])
    ax2.plot(np.arange(n), actual[:,1])
    ax2.plot(np.arange(n), predicted[:,1])
    ax2.legend(['Actual', 'Predicted'])
    ax2.set_xlabel('Time Steps')
    ax2.set_ylabel('y')
    ax2.set_title('y Trajectories')

    ax3 = fig.add_axes([1, 0.15, 0.4, 0.2])
    ax3.plot(np.arange(n), actual[:,2])
    ax3.plot(np.arange(n), predicted[:,2])
    ax3.legend(['Actual', 'Predicted'])
    ax3.set_xlabel('Time Steps')
    ax3.set_ylabel('z')
    ax3.set_title('z Trajectories')
    plt.savefig(file_name,bbox_inches="tight")
    plt.show()

#%%
# Load Lorenz Data
dat = loadmat('lorenz.mat')

inp = dat['input']
out = dat['output']

# %% Initialize Neural Net
model = Sequential()
model.add(Dense(128, activation='relu', input_dim=4))
model.add(Dense(128, activation='relu'))
model.add(Dense(3))

model.compile(loss='mse',
              optimizer=optimizers.Adam(lr=0.001),
              metrics=['mse'])

# Train Data
model.fit(inp, out,
          epochs=50,
```

```
          batch_size=32)


#%% Load Test Data for rho = 17,35
test_dat = loadmat('lorenz_test.mat')
input17_test = test_dat['input2'][0:799]
output17_test = test_dat['output2'][0:799]

input35_test = test_dat['input2'][800:]
output35_test = test_dat['output2'][800:]


# Use NN to predict trajectory
output17_pred = np.zeros(np.shape(output17_test))
output35_pred = np.zeros(np.shape(output35_test))
output17_pred[0,:] = input17_test[0,0:3]
output35_pred[0,:] = input35_test[0,0:3]

for i in np.arange(1,np.shape(input17_test)[0]):
    output17_pred[i,:] = model.predict(np.asarray([np.append(output17_pred[i-1,:], 17)]
    output35_pred[i,:] = model.predict(np.asarray([np.append(output35_pred[i-1,:], 17)]

# Plot Predicted vs Actual
plot_trajectory(input17_test, output17_pred, 'rho17.png')
plot_trajectory(input35_test, output35_pred, 'rho35.png')

#%% Predict Transistion for r = 28
dat28 = inp[120000:240000, 0]

dat28_reshape = np.zeros([800, 150])
for k in np.arange(1,151):
    dat28_reshape[:,k-1] = dat28[(k-1)*800:(k)*800]

# %%
out28 = np.zeros(np.shape(dat28_reshape))
for j in np.arange(150):
    for k in np.arange(799):
        aa =  dat28_reshape[k, j]
        bb = dat28_reshape[k+1, j]
        if aa*bb < 0:
            out28[k+1, j] = 0
        else:
            out28[k+1, j] = out28[k, j] + 1
```

```
# %% Create output data
inp28_train = np.transpose(dat28_reshape[:,0:148])
inp28_test = dat28_reshape[:, -1]

out28_train = np.transpose(out28[:, 0:148])
out28_test = out28[:, 149]

# %%
model28 = Sequential()
model28.add(Dense(1024, activation='relu', input_dim=800))
model28.add(Dense(1024, activation='relu'))
model28.add(Dense(800))

model28.compile(loss='mse',
              optimizer=optimizers.Adam(lr=0.001),
              metrics=['mse'])

# %%
model28.fit(inp28_train, out28_train,
              epochs = 200,
              batch_size=32)

# %%
out28_pred = model28.predict([[inp28_test]])

# %%
#plt.plot(inp28_test)
plt.figure(figsize=(10,5))
plt.plot(out28_test)
plt.plot(np.arange(800),np.transpose(out28_pred))
plt.title('Time Until Lobe Tranistion')
plt.ylim(-50,400)
plt.ylabel('Time (Wait in Time Steps)')
plt.xlabel('Time')
plt.savefig('b.png')
plt.show()



#%%
# load KS data
# For the KS data - the uu matrix is matrix where one column is one time shot
```

```
#   and the row is the value at that position x
Ks_dat = loadmat('kuramoto_sivishinky.mat')
outks = Ks_dat['output']
inpks = Ks_dat['input']
tt = Ks_dat['tt']
x = Ks_dat['x']


# %% Initialize NN

modelks = Sequential()
act = 'relu'
modelks.add(Dense(256, activation=act, input_dim=np.shape(x)[0]))
modelks.add(Dense(256, activation=act))
modelks.add(Dense(np.shape(x)[0]))

# compile model
modelks.compile(loss='mse',
              optimizer=optimizers.Adam(lr=0.001),
              metrics=['mse'])

#%%
# Train Data
modelks.fit(np.transpose(inpks), np.transpose(outks),
          epochs=30,
          batch_size=32)

# %%
# Load test data
ks_test = loadmat('ks_test.mat')
inks_test = ks_test['input_test']
outks_test = ks_test['output_test']

# Predict
outks_pred = np.zeros(np.shape(outks_test))
x0 = np.asarray([inks_test[:,0]])

for k in np.arange(np.shape(inks_test)[1]):
    x0 = modelks.predict(x0)
    outks_pred[:,k] = x0

fig, ax = plt.subplots(1,2, figsize=(10,5))
ax[0].imshow(np.transpose(outks_test), aspect='auto', origin = 'lower')
ax[0].set_title('Actual')
```

```
ax[0].set_ylim((0, 250))
ax[0].set_ylabel('Time t')
ax[0].set_xlabel('Spatial x')
ax[1].imshow(np.transpose(outks_pred), aspect='auto', origin = 'lower')
ax[1].set_title('Predicted')
ax[1].set_ylim((0, 250))
ax[1].set_ylabel('Time t')
ax[1].set_xlabel('Spatial x')
plt.savefig('ks.png')
plt.show()


#%% Load reaction diffusion data
datdf = loadmat('reaction_diffusion_big.mat')
U = datdf['u']
V = datdf['v']
t = np.shape(datdf['t'])[0]
x = np.shape(datdf['x'])[1]

# %% Reshape Data
reshape = np.zeros([t, 2*x**2])
for tt in np.arange(t):
    u_frame = U[:,:,tt]
    u_frame_reshape = np.reshape(u_frame,[1, x ** 2])
    v_frame = V[:,:,tt]
    v_frame_reshape = np.reshape(v_frame,[1, x ** 2])
    reshape[tt, 0:(x**2)] = u_frame_reshape
    reshape[tt, (x**2):] = u_frame_reshape


# %%
reshape_train = reshape[0:201, :]
reshape_test = reshape[201:241, :]
u, s, vh = la.svd(np.transpose(reshape_train), full_matrices = False)

# %% Graph Singular values
plt.figure(figsize=(10,5))
plt.plot(s/np.sum(s), 'r.', markersize = 10)
plt.title('Amount of Variation in Singular Values')
plt.xlabel('Singular Value')
plt.ylabel('Variation')
plt.ylim([0,1])
plt.savefig('sv.png')
```

```
# %%
rank = np.arange(0,2)
u_r = u[:,rank]
v_r = vh[rank, :]
s_r = np.diag(s[rank])


vs_mat = s_r@v_r

inpdf = np.transpose(vs_mat[:,0:200])
outdf = np.transpose(vs_mat[:,1:201])


# %%
modelrd = Sequential()
act = 'relu'
modelrd.add(Dense(128, activation=act, input_dim=2))
modelrd.add(Dense(128, activation=act))
modelrd.add(Dense(2))

# compile model
modelrd.compile(loss='mse',
              optimizer=optimizers.Adam(lr=0.001),
              metrics=['mse'])

# %%
modelrd.fit(inpdf, outdf,
          epochs=500,
          batch_size=32)

# %%
frameuv = np.transpose(np.transpose(u_r)@reshape_test[0,:])
frameuvdt_pred = modelrd.predict([[frameuv]])
frameuvdt_pred2 = u_r@np.transpose(frameuvdt_pred)

# %%
frameuvdt_true = reshape_test[1,:]
frame = np.reshape(reshape_test[0,0:(x**2)], (x,x))
framedt = np.reshape(reshape_test[1,0:(x**2)], (x,x))
framedt_pred = np.reshape(frameuvdt_pred2[0:(x ** 2), :], (x,x))

fig, ax = plt.subplots(1,3, figsize=(10,3))
plt.tight_layout()
ax[0].imshow(frame)
```

```
ax[0].set_title('U at t = 10')
ax[0].axis('off')
ax[1].imshow(framedt)
ax[1].set_title('U at t = 10 + dt Actual')
ax[1].axis('off')
ax[2].imshow(framedt_pred)
ax[2].set_title('U at t = 10 + dt Predicted')
ax[2].axis('off')
plt.savefig('df.png')
plt.show()
```