AMATH 563 Homework 4

Daniel Burnham (https://github.com/burnhamdr)

Jun 6, 2020

Abstract

The work presented here is motivated by material covered in AMATH 563 Inferring Structure of Complex Systems regarding the applications of Artificial Neural Network (ANN) models for time series data forecasting. This report will discuss this topic in the context of three dynamical systems scenarios: A reaction-diffusion system of equations, the Kuramoto-Sivashinsky (KS) equation for diffusive instabilities in a laminar flame, and the Lorenz system. Artificial networks will be trained, in the context of the KS equation, to advance the solution from t to $t + \Delta t$. For the reaction-diffusion system, the scale of the data generated through numerical time stepping of the system of equations presented an opportunity to explore ANN forecasting model training on SVD truncated data. Lastly, training an ANN for forecasting of the Lorenz system was pursued. In addition to this, an ANN was trained to classify when transitions from one lobe to the other on the Lorenz attractor occurred. Overall, ANNs were successful in optimizing performance, as measured by mean squared error, across training and validation. The ANN designed for time stepping the reaction-diffusion system appeared to perform the best as evidenced by the qualitatively similar final time step representation of the reaction-diffusion. Notably, ANN training on the Lorenz system demonstrated that though difficult to predict, the Lorenz system may have a predictable feature roughly 500 time steps prior to attractor lobe transition. This finding requires more investigation.

1 Introduction and Overview

The topics explored here are discussed in two sections. The first section involves the analysis of the reaction-diffusion and KS systems. The second section involves training an ANN for forecasting of the Lorenz system and also predicting when a transition from one lobe to another (on the Lorenz attractor) is imminent.

1.1 Kuramoto-Sivashinsky (KS) and Reaction-Diffusion Systems

Two data sets were used for ANN training with data generated by MATLAB code which used an ODE time-stepper to generate system trajectories. In the case of the KS system, data was obtained for 1024 spatial positions, 251 temporal instances. For the reaction-diffusion system of equations, data was generated for each state variable for 262144 spatial positions and 201 temporal positions. The SVD was computed for the reaction-diffusion system data in order to work with rank truncated data sets.

1.2 Lorenz system

Lorenz system data was similarly generated through ODE time-stepper simulation. Data was obtained for 3 spatial positions and 801 temporal instances. An ANN similar to that described in Brunton 2019 Chapter 6, was trained to time step the system from t to $t + \Delta t$ for various ρ parameter values. Additionally, an ANN was trained in the prediction task of when the system switched between lobes of the Lorenz attractor.

2 Theoretical Background

2.1 Singular Value Decomposition (SVD)

SVD is a linear algebra method for factorization of a matrix into a number of constitutive components. It is rooted in the observation that during matrix vector multiplication of a vector $\mathbf{x}$ by a matrix $\mathbf{A}$ the resulting vector $\mathbf{y}$ has a new magnitude and direction. This transformation of the vector $\mathbf{x}$ by a matrix $\mathbf{A}$ implies that perhaps the action of matrix $\mathbf{A}$ can be replicated through component matrices that perform the same magnitude and direction manipulations. Furthermore, it would be beneficial if these components possessed properties that made them easy to work with, such as orthogonality and diagonality. The SVD factorization of the matrix $\mathbf{A}$ achieves these goals by expanding this observation of vector transformation under multiplication by a matrix to $\mathbb{R}^m$. SVD does this by building from the observation that the image of a unit sphere under any m×n matrix is a hyperellipse. Following from this, one can represent this transformation as $\mathbf{A}\mathbf{v}_j = \sigma_j \mathbf{u}_j$ for $1 \leq j \leq n$. Where $\mathbf{v}_j$ are the vectors of the unit sphere transformed by $\mathbf{A}$, and $\sigma_j \mathbf{u}_j$ are the resulting transformations representing the the semiaxes of the hyperellipse. Rearranging this equation allows for the SVD factorization of $\mathbf{A}$ to be written as follows (Kutz 2013):

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^* \tag{1}$$

In this form, $\mathbf{U}$ is unitary and contains the vectors $\mathbf{u}_j$ indicating the directions of the transformed hyperellipse semiaxes, $\mathbf{\Sigma}$ is diagonal and contains the scaling values corresponding to these semiaxes, and $\mathbf{V}^*$ is the Hermitian transpose of $\mathbf{V}$ which contains the orthonormal basis of the vectors that are transformed under $\mathbf{A}$.

It is worth noting that it can be proved that every matrix $\mathbf{A} \in \mathbb{C}^{m \times n}$ has a singular value decomposition and the singular values are uniquely determined (Kutz 2013). Additionally, if the matrix $\mathbf{A}$ is square, the singular vectors $\mathbf{u}_j$ and $\mathbf{v}_j$ are also uniquely determined up to complex signs (Kutz 2013). This is significant because it allows for every $\mathbf{A} \in \mathbb{C}^{m \times n}$ to be factorized by SVD and subsequently represented with lower rank matrix approximations. This will be useful in the context of the face image data set explored here as it represents a way of reducing the dimensionality of the face feature space.

2.2 Artificial Neural Networks (ANNs)

Generically, ANNs are a network of nodes with connections of varying weight. The fundamental idea is that the weights of these connections can be optimized so that the network maps some input to a desired output. ANNs specifically optimize over a compositional function:

$$\arg\min_{A_j} \left(f_M(A_M, \cdots, f_2(A_2, f_1(A_1, x)) \cdots) + \lambda g(A_j)\right) \tag{2}$$

which is often solved using stochastic gradient descent and back propagation algorithms (Brunton 2019). Each matrix A_k denotes the weights connecting the neural network from the k^{th} to $(k+1)^{th}$ layer (Brunton 2019). It is a massively underdetermined system which is regularized by $g(A_j)$ (Brunton 2019). There are lots of decisions to make when designing an ANN to a task of interest, beginning with the type of task at hand, regression or classification? The shape of the data to map from determines the size of the input to the network, and in regression this also determines the shape of the output mapped by the network. In the case of classification, the number of distinct classes determines the size of the network output layer. Hidden layers are sets of network nodes in between the input and output network layers. These are not obligatory to include. Infact, the simplest example of an ANN is simply a connection from input to output and is generally termed a "perceptron" neural network model. In the case where linear mapping occurs between each layer of the ANN:

$$x_j = A_j x_{j-1} \tag{3}$$

Where x_{j-1} is the input into a given layer (i.e. the previous layer output, or in the case of the first layer, the input to the network itself). The matrices A_j contain the coefficients that map each variable from one layer to the next. For the network as a whole these operations can be chained to achieve a compositional

structure mapping between network input and output:

$$y = \left(\prod_{j=M}^{1} A_j \right) x \tag{4}$$

where M is the number of network layers, x is the input to the network, and y is the output. Nonlinear mappings are also possible, and generally used, in constructing ANNs. This simply leads to the compositional structure for mapping between network input and output as shown in Equation 3. Modern stochastic gradient descent and back propagation algorithms enable optimization of these functions. Common nonlinear transformations from input-to-output space represented by

$$y = f(A, x) \tag{5}$$

where $f(\cdot)$ is a specified activation function (transfer function) for the mapping are given by

$$\begin{aligned}
&f(x) = x && \text{linear} \\
&f(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ 1 & \text{if} x > 0 \end{cases} && \text{binary step} \\
&f(x) = \frac{1}{1 + e^{-x}} && \text{logistic (soft step)} \\
&f(x) = tanh(x) && \text{TanH} \\
&f(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if} x > 0 \end{cases} && \text{rectified linear unit (ReLu)}
\end{aligned}$$

Importantly, the chosen function $f(x)$ will be differentiated in order to be used in gradient descent algorithms for optimization.

2.2.1 Back Propagation

At the core of ANN model mapping from input to output is an optimization routine which requires an objective function to determine the weights of node connections in the network. In practice, the objective function chosen for optimization is not the true objective function desired, but rather a proxy for it in order to facilitate the computational tractability of differentiating the function (Brunton 2019). The backpropagation algorithm poses the optimization problem in a form amenable to standard gradient descent optimization (Brunton 2019). In the simple case of a input-to-output for a single node in a hidden layer, the compositional structure can be formulated as:

$$y = g(z, b) = g(f(x, a), b) \tag{6}$$

where $f(\cdot)$ and $g(\cdot)$ are the transfer functions with weighting constants a and b (Brunton 2019). The output error produce by the network can be computed against the ground truth as:

$$E = \frac{1}{2}(y_0 - y)^2 \tag{7}$$

where y0 is the correct output and y is the ANN approximation (Brunton 2019). The goal is to find a and b that minimize the error, E. The minimization thus involves solving (Brunton 2019):

$$\frac{\partial E}{\partial a} = -(y_0 - y)\frac{dy}{dz}\frac{dz}{da} = 0 \tag{8}$$

This process is termed backpropagation because the combination of the compositional nature (Equation 6) of the network and the chain rule (Equation 8) results in the backward pass of error through the ANN (Brunton 2019). Backpropagation results in an iterative, gradient descent update rule (Brunton 2019):

$$\begin{aligned}
a_{k+1} &= a_k + \delta \frac{\partial E}{\partial a_k} \\
b_{k+1} &= b_k + \delta \frac{\partial E}{\partial b_k}
\end{aligned} \tag{9}$$

where δ is the learning rate and $\frac{\partial E}{\partial a_k}$ and $\frac{\partial E}{\partial b_k}$ are computed from Equation 8 (Brunton 2019). The iteration proceeds as follows to convergence: (i) The initial weights of the network are set to random values. (ii) The training data is run through the network to produce an output y, whose ideal ground-truth output is y_0. The derivatives with respect to each network weight are then computed. (iv) For a given learning rate δ, the network weights are updated and the next iteration is started (Brunton 2019).

3 Algorithm Implementation and Development

3.1 Kuramoto-Sivashinsky (KS) System

The main steps of the algorithm are as follows:

1. Run the MATLAB Kuramoto-Sivashinsky script to generate the .mat file containing the ODE time stepper simulation data for training.
2. Input the .mat file, extract the heat data ('uu'), standardize the data, split the data into time t data and time $t + \Delta t$ data for ANN mapping.
3. Train an ANN to advance the solution from t to $t + \Delta t$ for the KS equation.
4. Plot training and validation RMSE, visualize the the ANN and ODE time stepper simulation results.

3.2 Reaction Diffusion System

The main steps of the algorithm are as follows:

1. Run the MATLAB Reaction Diffusion script to generate the .mat file containing the ODE time stepper simulation data for training.
2. Input the .mat file, extract the data for both state variables and perform flattening and horizontal stacking of each state variable spatial data at each time step. Standardize the data and split the data into time t data and time $t + \Delta t$ data for ANN mapping.
3. Train an ANN to advance the solution from t to $t + \Delta t$ for the reaction diffusion equation.
4. Plot training and validation RMSE, visualize the the ANN and ODE time stepper simulation results at the final time step.

3.3 Lorenz System

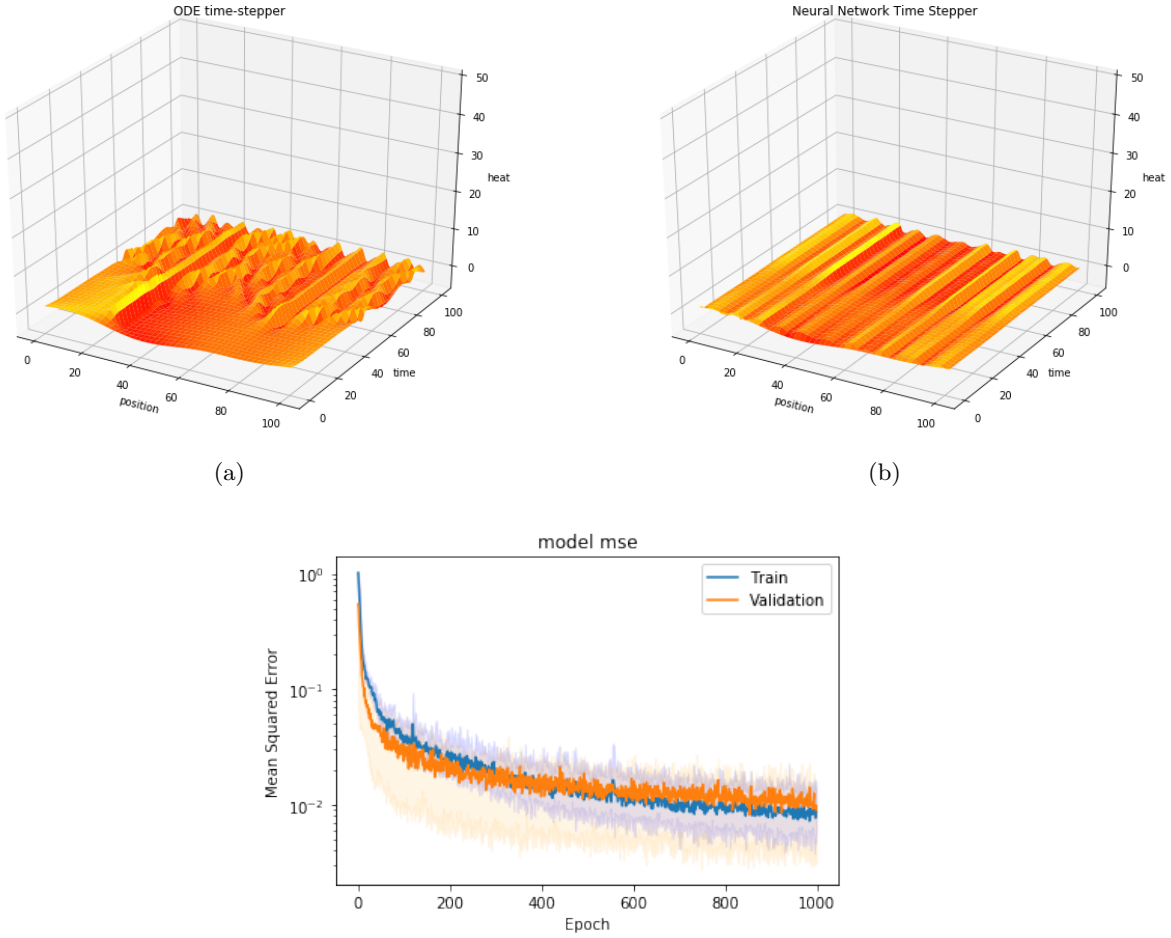
The main steps of the algorithm are as follows:

1. Run the MATLAB Lorenz script to generate the .mat file containing the ODE time stepper simulation data for simulations of the ρ values of interest.
2. Input the .mat file, extract the data for ANN input and output for testing and training.
3. Train an ANN to advance the solution from t to $t + \Delta t$ for the Lorenz system with ρ values of 10, 28 and 40. Test the model on the Lorenz system data generated with ρ values of 17, and 35
4. Plot training and validation RMSE.
5. Label each time step of the Lorenz trajectories for $\rho = 28$ with the number of time steps until the x position coordinate crossed the plane $x = 0$ (indicator of attractor lobe transition).
6. Train an ANN to predict the number of time steps until lobe transition for each position along the trajectories for 100 different initial conditions and $\rho = 28$.
7. Plot training and validation RMSE. Visualize distribution of RMSE values by number of steps until transition within the testing data.

4 Computational Results

4.1 Kuramoto-Sivashinsky (KS) System

Various network architectures were experimented with in designing the model for the KS system forecasting task. Ultimately a network with 3 hidden layers was settled on with the input layer having 250 nodes and the 3 hidden layers having 100 nodes with %20 dropout for regularization. This network after 1000 training epochs achieved mean squared errors on the order of 10^{-2} on both training and validation data (Figure 1c). This model was not able to generate a qualitatively similar time evolution compared to the ODE time stepper (Figure 1a, Figure 1b).



(a) (b)

(c)

Figure 1: (a) The ODE time stepper solution to the KS equation (b) The ANN time stepper solution to the KS equation after training. (c) ANN Mean Squared Error across for training and validation sets.

4.2 Reaction Diffusion System

The Singular Value Decomposition of the reaction diffusion system data suggested that the state space could be well represented with a rank 10 approximation (Figure 2. This was the dimension of the data passed to the ANN for training. Various network architectures were again experimented with in designing the model for the reaction diffusion system forecasting task. Ultimately a network with 2 hidden layers was settled on with the input layer having 100 nodes and the 2 hidden layers having 100 nodes and 100 nodes respectively

with %20 dropout for regularization. This network after 1000 training epochs achieved mean squared errors on the order of 10^3 on both training and validation data (Figure 1c). This model generated a seemingly appropriate time evolution as evidenced by the ANN final time step approximation (Figure 3b) compared to the ODE time stepper generated data (Figure 3a).

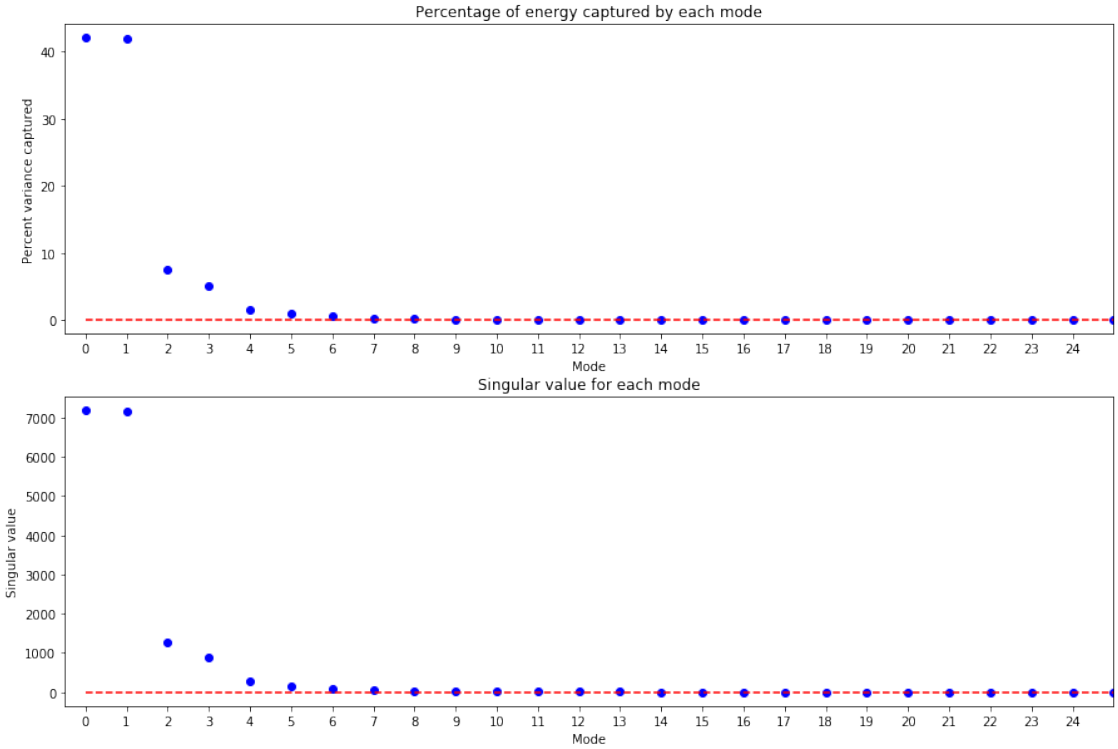


Figure 2: (Top) The percentage of energy captured by each SVD mode. (Bottom) Magnitudes of the singular values

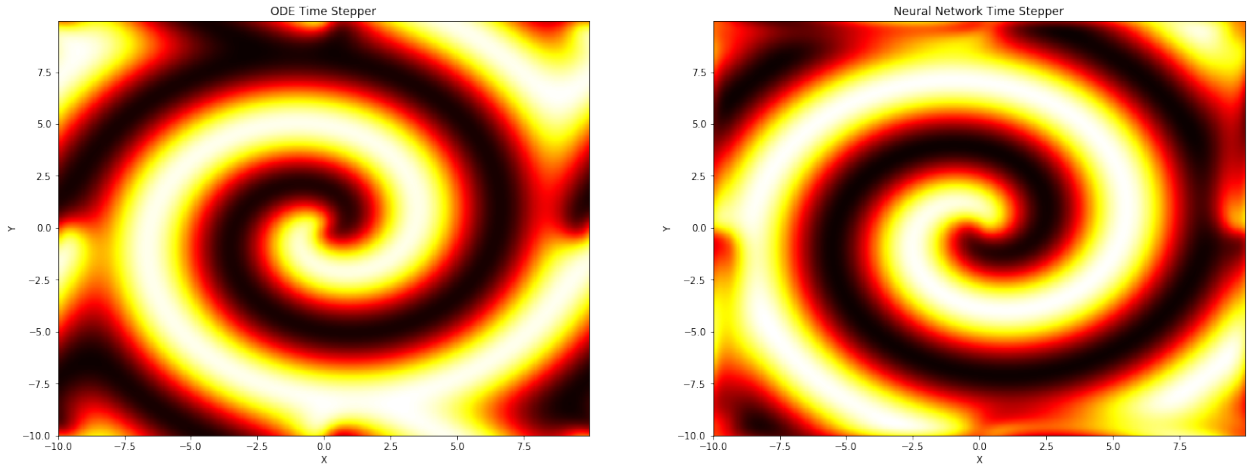
4.3 Lorenz System

4.3.1 ρ Variation Forecasting

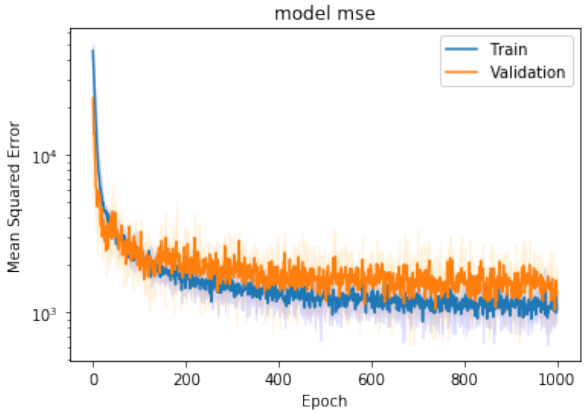
The ANN model trained on Lorenz system data for $\rho = 10$,28 and 40 achieved root mean squared error on the order of 10^0 when predicting the test data for $\rho = 17$ and 35 (Figure 4b).

4.3.2 Lobe Switching Prediction

The lobe switching problem was designed as a regression where the ANN sought to predict the number of time steps that each data instance was from transitioning to the other Lorenz attractor lobe. Various attempted network architectures were largely unsuccessful in achieving a reasonably low mean squared error (Figure 5). The RMSE plot (Figure 7) provides insight into how the ability to predict a transition changes with the proximity of a time step to the next transition. To make this relationship more clear, Figure 8 shows the distribution ANN prediction RMSE by the number steps until lobe transition.



(a) (b)



(c)

Figure 3: (a) The ODE time stepper solution to the reaction diffusion system of equations (b) The ANN time stepper solution to the reaction diffusion system of equations after training. (c) ANN Mean Squared Error across for training and validation sets.

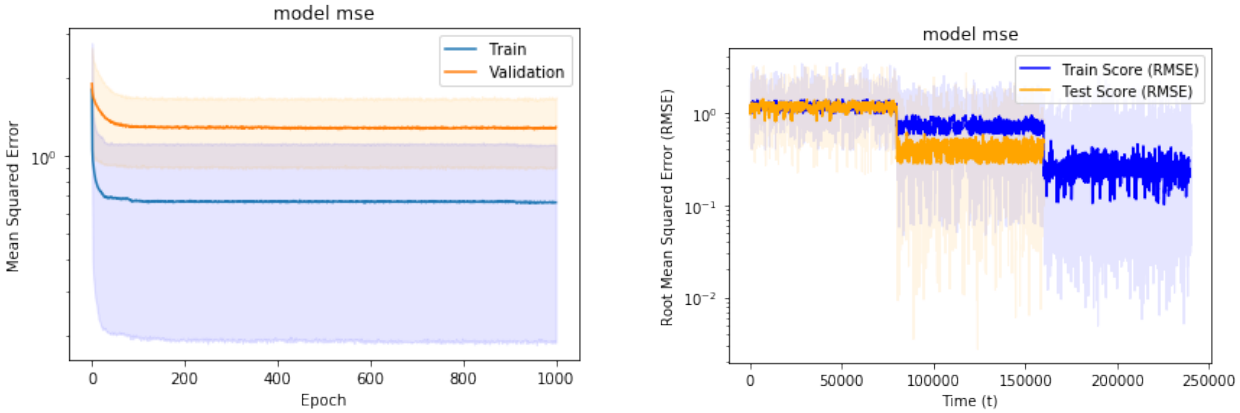
5 Summary and Conclusions

5.1 Kuramoto-Sivashinsky (KS) and Reaction-Diffusion Systems

In the case of the KS system, the ANN model was not able to replicate the time stepping of the ODE method. This could be due to the time evolution of the system appearing to have two distinct phases. Though this dichotomy should presumably be capturable by the ANN through learning, in this particular ANN design, the approximation of the ODE time stepper was very crude. Performance of the ANN time stepper on the reaction-diffusion system was qualitatively better as evidenced by the similarities shown between Figure 3a and Figure 3b. There are identifiable differences in pattern, but the general shape of the reaction-diffusion at the final time step is highly similar.

5.2 Lorenz System

Designing an ANN to predict $\rho = 17$ and 35 from Lorenz system instances with $\rho = 10$, 28 and 40 was largely successful with a simple ANN architecture. The RMSE achieved was sufficiently small across training and testing data sets. The jumps in RMSE in Figure 4b correspond with transitions from sets of data for specific



(a) (b)

Figure 4: (a) ANN Mean Squared Error across for training and validation sets for the Lorenz attractor ANN time stepping task. (b) ANN RMSE running average for each time step of the train and test sets.

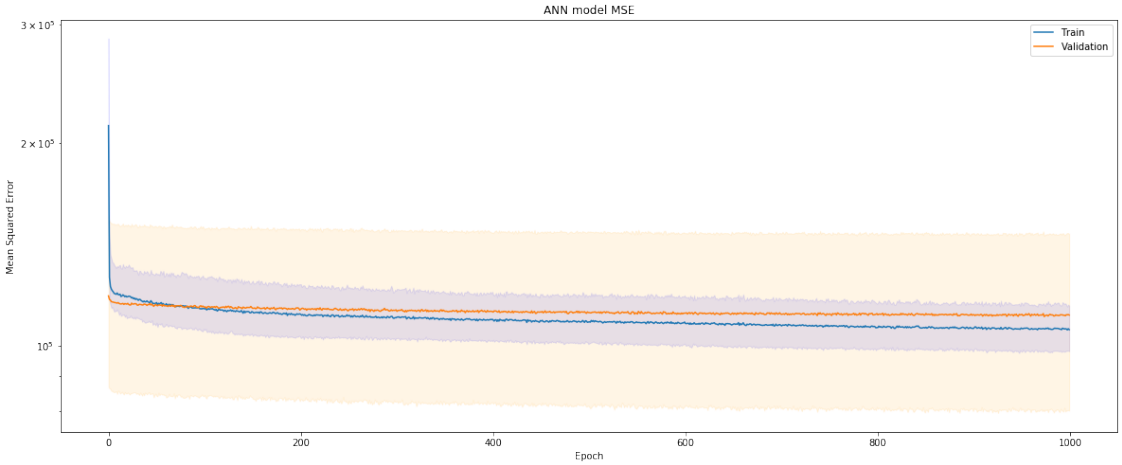


Figure 5: ANN Mean Squared Error for predicting attractor lobe transitions across training and validation sets.

ρ values. This is due to how the data matrices were generated in MATLAB. The data matrices were filled such that 100 Lorenz system trajectories were stacked for ρ values from smallest to largest for both the train ($\rho = 10$, 28 and 40) and test ($\rho = 17$ and 35) sets. Thus it appears that the ANN prediction RMSE decreases for larger ρ values. In the Lorenz system climate model, the ρ parameter is described as representing the difference in temperature between the top and bottom of the system. The results here thus suggest larger ρ values may make the system more predictable. The differences in the trajectories along the attractor for small ($\rho = 10$) and large ($\rho = 40$) can be appreciated in Figure 9. Conversely, training an ANN on Lorenz system trajectories for $\rho = 28$ in an effort to predict attractor lobe transitions was largely unsuccessful as RMSE values for training and validation data were quite large across training epochs. Trends in RMSE by steps until transition (Figure 8) suggest that the attractor has a most predictable feature in a region corresponding to roughly 500 steps until lobe transition as evidenced by a high frequency of low RMSE values in this region. It would be interesting to see what this feature might be on the Lorenz system trajectories. One possibility is that this is simply an artifact of stacking the 100 different initial condition trajectories in the same data set.

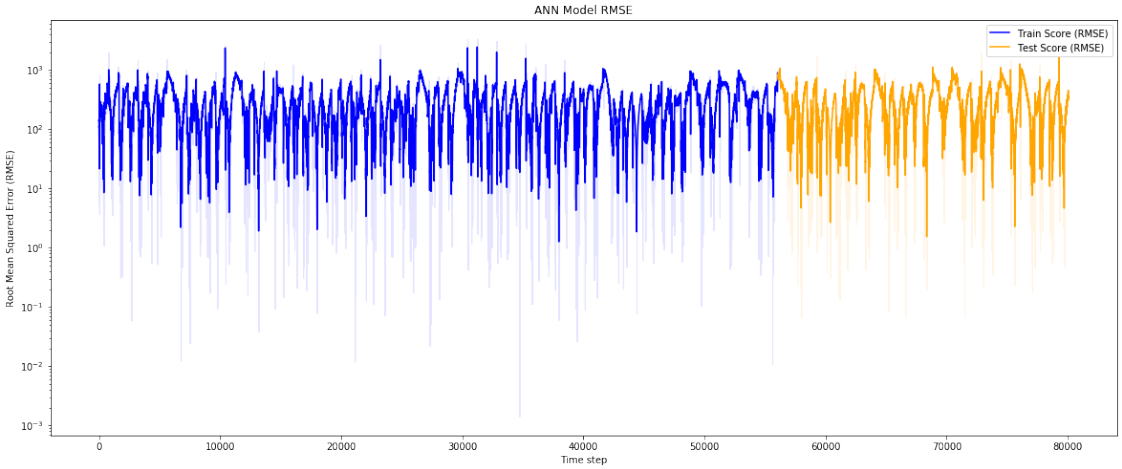


Figure 6: ANN RMSE running average for predicting attractor lobe transitions for each time step of the train and test sets.

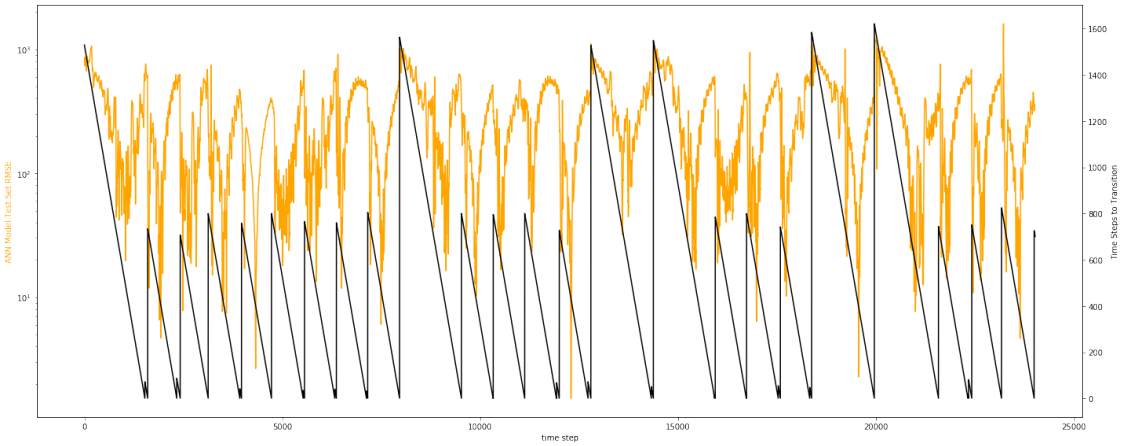


Figure 7: ANN RMSE running average for each time step of the test set with the number of time steps until lobe transition indicated in black.

References

Brunton, Steven L. (Steven Lee) (2019). *Data-driven science and engineering : machine learning, dynamical systems, and control*. Cambridge, United Kingdom ; New York, NY: Cambridge University Press. ISBN: 9781108422093.

Kutz, Jose Nathan (2013). *Data-driven modeling & scientific computation: methods for complex systems & big data*. Oxford University Press.

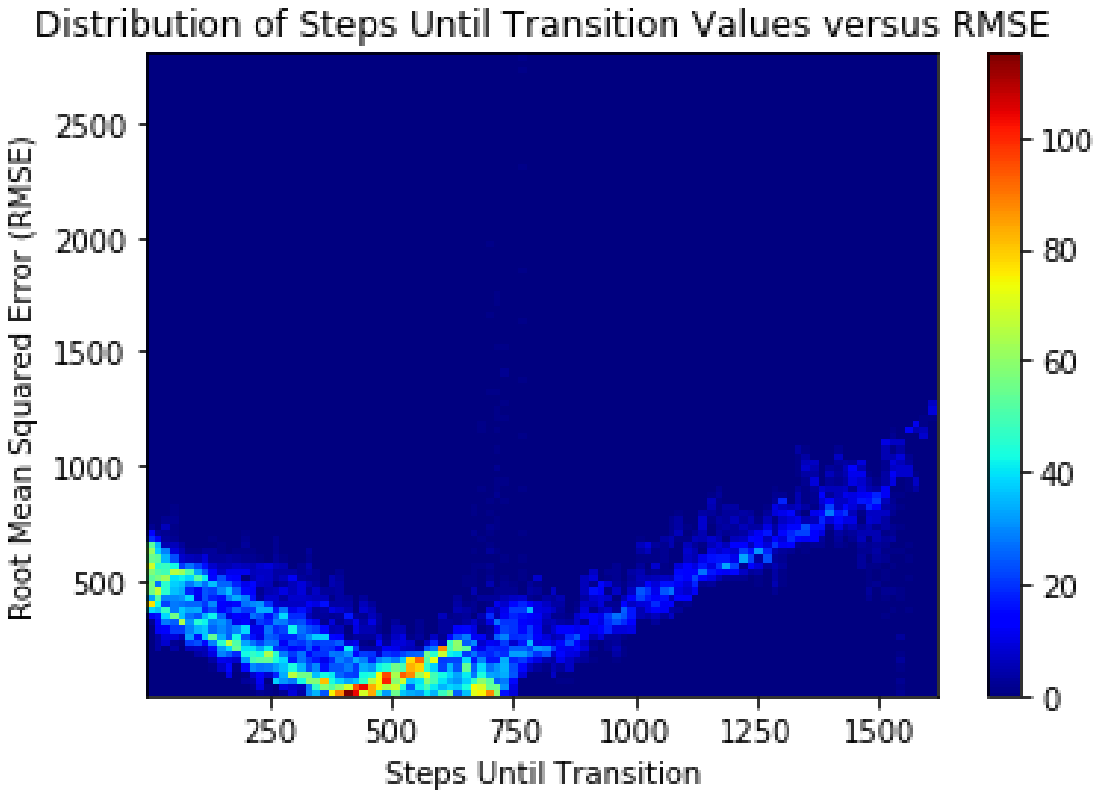
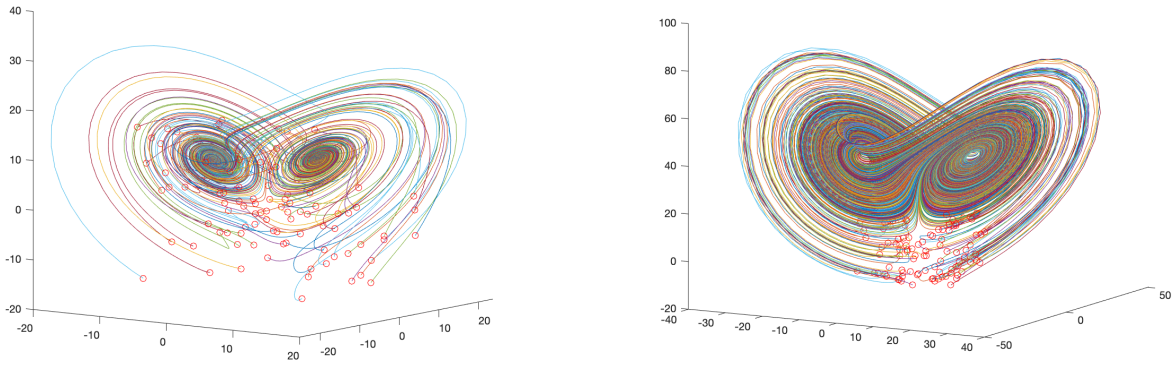


Figure 8: Histogram of the time steps until the next Lorenz attractor lobe transition versus ANN prediction RMSE.



(a) (b)

Figure 9: (a) Lorenz System for 100 different initial conditions and $\rho = 10$ (b) Lorenz System for 100 different initial conditions and $\rho = 40$.

Appendix A Python Functions

- `numpy.linalg.svd(a, full_matrices=True, compute_uv=True, hermitian=False)` Singular Value Decomposition.

- `sklearn.preprocessing.StandardScaler(*, copy=True, with_mean=True, with_std=True)`
  Standardize features by removing the mean and scaling to unit variance
- `sklearn.model_selection.TimeSeriesSplit(n_splits=5, *, max_train_size=None)`
  Time Series cross-validator. Provides train/test indices to split time series data samples that are observed at fixed time intervals, in train/test sets. In each split, test indices must be higher than before, and thus shuffling in cross validator is inappropriate.
- `tf.keras.Sequential(layers=None, name=None)`
  Sequential groups a linear stack of layers into a tf.keras.Model.

Appendix B Python Code

```
#!/usr/bin/env python
# coding: utf-8

# In[157]:


import scipy.io as sio
import keras
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from sklearn import preprocessing
from keras.layers import Layer
from keras import backend as K
from sklearn.model_selection import TimeSeriesSplit
import numpy as np
from keras.layers import LSTM
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
import math
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
from sklearn.decomposition import TruncatedSVD
from scipy.stats import gaussian_kde


# In[101]:


def nn_time_stepper(model, x_prev):
    x_next = model.predict(x_prev)
    return x_next

def runningMean(mylist, N):
    cumsum, moving_aves, max_vals, min_vals = [0], [], [], []
    vals = []
    for i, x in enumerate(mylist, 1):
        cumsum.append(cumsum[i-1] + x)
        vals.append(x)
        if i>=N:
            max_vals.append(max(vals))
            min_vals.append(min(vals))
            moving_ave = (cumsum[i] - cumsum[i-N])/N
            #can do stuff with moving_ave here
            moving_aves.append(moving_ave)
            vals = []
    return moving_aves, max_vals, min_vals


# In[26]:


np.random.seed(7)
#Data matrix contains 2 dimensional handel music sample
ks_ = sio.loadmat('kuramoto_sivishinky.mat')#load MATLAB .mat file into python as a dictionary
uu = ks_['uu']#dimension 1024 x 251. 1024 spatial positions, 251 temporal positions

scaler = preprocessing.StandardScaler().fit(uu)#standardize the data
data = scaler.transform(uu)

data_train = data[:,:int(data.shape[1]*0.7)]#data to train model
```

```
data_test = data[:,int(data.shape[1]*0.7):]#data to to test the model

#To forecast the values of future time steps of a sequence, specify the responses
#to be the training sequences with values shifted by one time step. That is, at each
#time step of the input sequence, the LSTM network learns to predict the value of the next
#time step. The predictors are the training sequences without the final time step.

m, n = data_train.shape
X = np.transpose(data_train[:, 0:n-2])#all rows starting from first column up to the penultimate
y = np.transpose(data_train[:, 1:n-1])#all rows starting from second column up to the last column

X_test = np.transpose(data_test[:, 0:n-2])#all columns starting from first column up to the penultimate
y_test = np.transpose(data_test[:, 1:n-1])#all columns starting from second column up to the last column

tss = TimeSeriesSplit(n_splits=3)
train_mse = []
val_mse = []
for train_index, test_index in tss.split(X):
    X_train, X_val = X[train_index, :], X[test_index,:]
    y_train, y_val = y[train_index], y[test_index]
      # define model
    model = Sequential()
    model.add(Dense(250, activation='relu',  input_shape = (X_train.shape[1],)))
    model.add(Dense(100, activation='relu')) # New hidden layer with 4 params
    model.add(Dropout(0.2))
    model.add(Dense(100, activation='relu')) # New hidden layer with 4 params
    model.add(Dropout(0.2))
    model.add(Dense(100, activation='relu')) # New hidden layer with 4 params
    model.add(Dropout(0.2))
    model.add(Dense(X_train.shape[1]))
    model.compile(optimizer='Adam', loss='mse', metrics=['mean_squared_error'])
    history_train = model.fit(X_train, y_train, epochs=1000, verbose=0)#train the model
    history_validate = model.fit(X_val, y_val, epochs=1000, verbose=0)#validate the model
    train_mse.append(history_train.history['mean_squared_error'])
    val_mse.append(history_validate.history['mean_squared_error'])


train_mse_array = np.transpose(np.asarray(train_mse))
val_mse_array = np.transpose(np.asarray(val_mse))
train_av = np.mean(train_mse_array, axis=1)
val_av = np.mean(val_mse_array, axis=1)
train_top = np.amax(train_mse_array, axis=1)
val_top = np.amax(val_mse_array, axis=1)
train_bottom = np.amin(train_mse_array, axis=1)
val_bottom = np.amin(val_mse_array, axis=1)

fig, ax = plt.subplots()
ax.fill_between(range(0,1000), (train_bottom), (train_top), color='blue', alpha=.1)
ax.fill_between(range(0,1000), (val_bottom), (val_top), color='orange', alpha=.1)
ax.plot( train_av, label='Train')
ax.plot(val_av, label='Validation')
ax.set_title('model mse')
ax.set_ylabel('Mean Squared Error')
ax.set_xlabel('Epoch')
ax.set_yscale('log')
ax.legend(loc='best')

# make predictions
trainPredict = model.predict(X)
testPredict = model.predict(X_test)

# calculate root mean squared error
trainScore = math.sqrt(mean_squared_error(y[0], trainPredict[0,:]))
print('Train Score: %.2f RMSE' % (trainScore))
testScore = math.sqrt(mean_squared_error(y_test[0], testPredict[0,:]))
print('Test Score: %.2f RMSE' % (testScore))


# In[307]:


u_0 = np.transpose(data[:,0].reshape(-1, 1))
u = u_0
U = [u]
for i in range(0, ks_['tt'].shape[1] - 1):
    u_next = nn_time_stepper(model, u)
    U.append(u)
    u = u_next

# Make data.
X = ks_['x']
```

```
T = ks_['tt']
X, T = np.meshgrid(X, T)
U = scaler.inverse_transform(np.transpose(np.concatenate(U, axis=0)))
U_orig = data

fig = plt.figure(figsize=(10, 8))
ax = fig.gca(projection='3d')
# Plot the surface.
surf = ax.plot_surface(np.transpose(X), np.transpose(T), U, cmap=cm.autumn,
                       linewidth=0, antialiased=True)


ax.set_zlim(-5, 50)# Customize the z axis.
ax.set_title("Neural Network Time Stepper")
ax.set_xlabel('position')
ax.set_ylabel('time')
ax.set_zlabel('heat')

fig = plt.figure(figsize=(10, 8))
ax2 = fig.gca(projection='3d')
# Plot the surface.
surf = ax2.plot_surface(np.transpose(X), np.transpose(T), U_orig, cmap=cm.autumn,
                       linewidth=0, antialiased=True)

ax2.set_zlim(-5, 50)# Customize the z axis.
ax2.set_title("ODE time-stepper")
ax2.set_xlabel('position')
ax2.set_ylabel('time')
ax2.set_zlabel('heat')


# In[4]:


rd_ = sio.loadmat('reaction_diffusion_big.mat')#load MATLAB .mat file into python as a dictionary
u = np.transpose(rd_['u'].reshape(-1, rd_['u'].shape[-1]))#dimension 262144 x 201. 262144 spatial positions, 201 temporal positions
v = np.transpose(rd_['v'].reshape(-1, rd_['v'].shape[-1]))#dimension 262144 x 201. 262144 spatial positions, 201 temporal positions

data_stack = np.hstack((u, v))
scaler_u = preprocessing.StandardScaler()
data = scaler_u.fit_transform(data_stack)

rank = 10
U, Sigma, VT = np.linalg.svd(data, full_matrices=False)
# #u_rr = U_u[:, :rank] @ np.diag(Sigma_u[:rank]) @ VT_u[:rank, :]
# data_lr = U[:, :rank]@data


svd = TruncatedSVD(n_components=rank)# define transform
svd.fit(data)# prepare transform on dataset
data_lr = svd.transform(data)# apply transform to dataset

#Plot percentage of variance captured by differnt modes
#Principle component oscillations
f, [ax1, ax2] = plt.subplots(2, 1, figsize=(15, 10))
ax1.plot((Sigma / sum(Sigma)) * 100, 'bo')
ax1.plot(np.arange(len(Sigma)), np.zeros(len(Sigma)), 'r--')
ax1.set_title('Percentage of energy captured by each mode')
ax1.set_ylabel('Percent variance captured')
ax1.set_xlabel('Mode')
ax1.set_xticks(range(0, 25))
ax1.set_xlim(( -0.5, 25))

ax2.plot(Sigma, 'bo')
ax2.plot(np.arange(len(Sigma)), np.zeros(len(Sigma)), 'r--')
ax2.set_title('Singular value for each mode')
ax2.set_ylabel('Singular value')
ax2.set_xlabel('Mode')
ax2.set_xticks(range(0, 25))
ax2.set_xlim(( -0.5, 25))


data_train = data_lr[:int(data_lr.shape[0]*0.7),:]#data to train model
data_test = data_lr[int(data_lr.shape[0]*0.7):,:]#data to to test the model

#To forecast the values of future time steps of a sequence, specify the responses
#to be the training sequences with values shifted by one time step. That is, at each
#time step of the input sequence, the LSTM network learns to predict the value of the next
#time step. The predictors are the training sequences without the final time step.

n, l = data_train.shape
```

```
X = data_train[0:n-2,:]#all cols starting from first column up to the penultimate
y = data_train[1:n-1,:]#all cols starting from second column up to the last column

X_test = data_test[0:n-2,:]#all columns starting from first column up to the penultimate
y_test = data_test[1:n-1,:]#all columns starting from second column up to the last column

tss = TimeSeriesSplit(n_splits=3)
train_mse = []
val_mse = []
for train_index, test_index in tss.split(X):
    X_train, X_val = X[train_index, :], X[test_index, :]
    y_train, y_val = y[train_index, :], y[test_index, :]

    # define model
    model_rd = Sequential()
    model_rd.add(Dense(100, activation='relu',  input_shape = (X_train.shape[1],)))
    model_rd.add(Dense(100, activation='relu')) # New hidden layer with 4 params
    model_rd.add(Dropout(0.2))
    model_rd.add(Dense(75, activation='relu')) # New hidden layer with 4 params
    model_rd.add(Dropout(0.2))
    model_rd.add(Dense(X_train.shape[1]))
    model_rd.compile(optimizer='Adam', loss='mse', metrics=['mean_squared_error'])

    history_train = model_rd.fit(X_train, y_train, epochs=1000, verbose=0)#train the model
    history_validate = model_rd.fit(X_val, y_val, epochs=1000, verbose=0)#validate the model
    train_mse.append(history_train.history['mean_squared_error'])
    val_mse.append(history_validate.history['mean_squared_error'])


train_mse_array = np.transpose(np.asarray(train_mse))
val_mse_array = np.transpose(np.asarray(val_mse))
train_av = np.mean(train_mse_array, axis=1)
val_av = np.mean(val_mse_array, axis=1)
train_top = np.amax(train_mse_array, axis=1)
val_top = np.amax(val_mse_array, axis=1)
train_bottom = np.amin(train_mse_array, axis=1)
val_bottom = np.amin(val_mse_array, axis=1)

fig, ax = plt.subplots()
ax.fill_between(range(0,1000), (train_bottom), (train_top), color='blue', alpha=.1)
ax.fill_between(range(0,1000), (val_bottom), (val_top), color='orange', alpha=.1)
ax.plot( train_av, label='Train')
ax.plot(val_av, label='Validation')
ax.set_title('model mse')
ax.set_ylabel('Mean Squared Error')
ax.set_xlabel('Epoch')
ax.set_yscale('log')
ax.legend(loc='best')

# make predictions
trainPredict = model_rd.predict(X)
testPredict = model_rd.predict(X_test)

# calculate root mean squared error
trainScore = math.sqrt(mean_squared_error(y[0], trainPredict[0,:]))
print('Train Score: %.2f RMSE' % (trainScore))
testScore = math.sqrt(mean_squared_error(y_test[0], testPredict[0,:]))
print('Test Score: %.2f RMSE' % (testScore))


# In[16]:


#visualize results (pcolor(x,y,u(:,:,end)))
uv_0 = data_lr[0,:].reshape(1, -1)#initial time step
uv = uv_0
UV = [uv]
for i in range(0, rd_['t'].shape[0] - 1):
    uv_next = nn_time_stepper(model_rd, uv)
    UV.append(uv)
    uv = uv_next

# Make data.
X = rd_['x']
Y = rd_['y']
X, Y = np.meshgrid(X, Y)

UV = scaler_u.inverse_transform(svd.inverse_transform(np.concatenate(UV, axis=0)))
u_out, v_out = np.split(UV, 2, axis=1)
u_out = u_out.reshape(-1, rd_['u'].shape[0], rd_['u'].shape[0])
u_out_new = np.transpose(u_out, (1, 2, 0))
```

```
U_orig = rd_['u']

fig, ax1 = plt.subplots(figsize=(10, 8))
# Plot the surface.
ax1.pcolor(X, Y, u_out_new[:,:,-1], cmap=cm.hot)


#ax.set_zlim(-5, 50)# Customize the z axis.
ax1.set_title("Neural Network Time Stepper")
ax1.set_xlabel('X')
ax1.set_ylabel('Y')

fig, ax2 = plt.subplots(figsize=(10, 8))
# Plot the surface.
ax2.pcolor(X, Y, U_orig[:,:,-1], cmap=cm.hot)

#ax.set_zlim(-5, 50)# Customize the z axis.
ax2.set_title("ODE Time Stepper")
ax2.set_xlabel('X')
ax2.set_ylabel('Y')


# In[37]:


lor_train = sio.loadmat('lorentz_train.mat')#load MATLAB .mat file into python as a dictionary
lor_test = sio.loadmat('lorentz_test.mat')#load MATLAB .mat file into python as a dictionary
in_train, out_train, x0, t =  lor_train['input'], lor_train['output'], lor_train['x0'], lor_train['t']
in_test, out_test = lor_test['input'], lor_test['output']

X = in_train#all cols starting from first column up to the penultimate
y = out_train#all cols starting from second column up to the last column

X_test = in_test#all columns starting from first column up to the penultimate
y_test = out_test#all columns starting from second column up to the last column

tss = TimeSeriesSplit(n_splits=3)
train_mse = []
val_mse = []
for train_index, test_index in tss.split(X):
    X_train, X_val = X[train_index, :], X[test_index, :]
    y_train, y_val = y[train_index, :], y[test_index, :]

    # define model
    model_lor = Sequential()
    model_lor.add(Dense(100, activation='relu',  input_shape = (X_train.shape[1],)))
    model_lor.add(Dense(100, activation='relu')) # New hidden layer with 4 params
    model_lor.add(Dropout(0.2))
    model_lor.add(Dense(X_train.shape[1]))
    model_lor.compile(optimizer='Adam', loss='mse', metrics=['mean_squared_error'])

    history_train = model_lor.fit(X_train, y_train, epochs=1000, verbose=0)#train the model
    history_validate = model_lor.fit(X_val, y_val, epochs=1000, verbose=0)#validate the model
    train_mse.append(history_train.history['mean_squared_error'])
    val_mse.append(history_validate.history['mean_squared_error'])


train_mse_array = np.transpose(np.asarray(train_mse))
val_mse_array = np.transpose(np.asarray(val_mse))
train_av = np.mean(train_mse_array, axis=1)
val_av = np.mean(val_mse_array, axis=1)
train_top = np.amax(train_mse_array, axis=1)
val_top = np.amax(val_mse_array, axis=1)
train_bottom = np.amin(train_mse_array, axis=1)
val_bottom = np.amin(val_mse_array, axis=1)

fig, ax = plt.subplots()
ax.fill_between(range(0,1000), (train_bottom), (train_top), color='blue', alpha=.1)
ax.fill_between(range(0,1000), (val_bottom), (val_top), color='orange', alpha=.1)
ax.plot(train_av, label='Train')
ax.plot(val_av, label='Validation')
ax.set_title('ANN model MSE')
ax.set_ylabel('Mean Squared Error')
ax.set_xlabel('Epoch')
ax.set_yscale('log')
ax.legend(loc='best')

# make predictions
trainPredict = model_lor.predict(X)
testPredict = model_lor.predict(X_test)
```

```
# calculate root mean squared error
trainScore = []
testScore = []
for i in range(len(y)):
    train_error = math.sqrt(mean_squared_error(y[i], trainPredict[i,:]))
    trainScore.append(train_error)
for i in range(len(y_test)):
    test_error = math.sqrt(mean_squared_error(y_test[i], testPredict[i,:]))
    testScore.append(test_error)

train_RMSE_running_mean, train_maxs, train_mins = runningMean(trainScore, 200)
test_RMSE_running_mean, test_maxs, test_mins = runningMean(testScore, 200)


fig, ax = plt.subplots()
ax.fill_between(range(len(train_RMSE_running_mean)), (train_mins), (train_maxs), color='blue', alpha=.1)
ax.fill_between(range(len(test_RMSE_running_mean)), (test_mins), (test_maxs), color='orange', alpha=.1)
ax.plot(train_RMSE_running_mean, color='blue', label='Train Score (RMSE)')
ax.plot(test_RMSE_running_mean, color='orange', label='Test Score (RMSE)')
ax.set_title('ANN Model RMSE')
ax.set_ylabel('Root Mean Squared Error (RMSE)')
ax.set_xlabel('Time step')
ax.set_yscale('log')
ax.legend(loc='best')


# In[ ]:




# In[160]:


# treat it as a regression problem too, predict the # of time steps
# until the next transition

lor_p28 = sio.loadmat('lorentz_p28.mat')#load MATLAB .mat file into python as a dictionary
data, x0, t =  lor_p28['data'], lor_p28['x0'], lor_p28['t']
x_vals = data[:,0]#get x position values
lobe_indicator = np.sign(x_vals)#get sign of x coordinate (proxy for lobe of lorenz attractor)
t_spots = []#list to hold where transitions occur
for i in range(1, len(lobe_indicator)):
    if lobe_indicator[i - 1] != lobe_indicator[i]:#check if sign has changed on x coordinate (i.e. lobe change)
        t_spots.append(1)#if sign changed, the lobe changed, mark this with 1
    else:
        t_spots.append(0)#otherwise no change, mark with 0

t_spots = np.array(t_spots)#where transistions have occured

transition_indices = np.where(t_spots_in == 1)[0]
steps_to_t = []
j = 0
for i in range(len(t_spots)):
    if j == len(transition_indices) - 1:
        break
    if i == transition_indices[j]:
        j = j + 1
    next_transition = transition_indices[j]
    steps_to_t.append(next_transition - i)

scaler = preprocessing.StandardScaler().fit(data)#standardize the data
data_scaled = scaler.transform(data)

data_train = data_scaled[:int(data.shape[0]*0.7),:]#data to train model
data_test = data_scaled[int(data.shape[0]*0.7):,:]#data to to test the model

X = data_train#all cols starting from first column up to the penultimate
y = np.expand_dims(np.array(steps_to_t[:int(data.shape[0]*0.7)]),  axis=1)#all cols starting from second column up to the last column

X_test = data_test#all columns starting from first column up to the penultimate
y_test = np.expand_dims(np.array(steps_to_t[int(data.shape[0]*0.7):]), axis=1)#all columns starting from second column up to the last column

tss = TimeSeriesSplit(n_splits=3)
train_mse = []
val_mse = []
for train_index, test_index in tss.split(X):
    X_train, X_val = X[train_index, :], X[test_index, :]
    y_train, y_val = y[train_index], y[test_index]
```

```
    # define model
    model_lorp28 = Sequential()
    model_lorp28.add(Dense(10, activation='relu',  input_shape = (X_train.shape[1],)))
    model_lorp28.add(Dense(10, activation='relu')) # New hidden layer with 4 params
    model_lorp28.add(Dropout(0.2))
    model_lorp28.add(Dense(10, activation='relu')) # New hidden layer with 4 params
    model_lorp28.add(Dropout(0.2))
    model_lorp28.add(Dense(1))
    model_lorp28.compile(optimizer='Adam', loss='mse', metrics=['mean_squared_error'])

    history_train = model_lorp28.fit(X_train, y_train, epochs=1000, verbose=0)#train the model
    history_validate = model_lorp28.fit(X_val, y_val, epochs=1000, verbose=0)#validate the model
    train_mse.append(history_train.history['mean_squared_error'])
    val_mse.append(history_validate.history['mean_squared_error'])


train_mse_array = np.transpose(np.asarray(train_mse))
val_mse_array = np.transpose(np.asarray(val_mse))
train_av = np.mean(train_mse_array, axis=1)
val_av = np.mean(val_mse_array, axis=1)
train_top = np.amax(train_mse_array, axis=1)
val_top = np.amax(val_mse_array, axis=1)
train_bottom = np.amin(train_mse_array, axis=1)
val_bottom = np.amin(val_mse_array, axis=1)

fig, ax = plt.subplots(figsize = (20, 8))
ax.fill_between(range(0,1000), (train_bottom), (train_top), color='blue', alpha=.1)
ax.fill_between(range(0,1000), (val_bottom), (val_top), color='orange', alpha=.1)
ax.plot(train_av, label='Train')
ax.plot(val_av, label='Validation')
ax.set_title('ANN model MSE')
ax.set_ylabel('Mean Squared Error')
ax.set_xlabel('Epoch')
ax.set_yscale('log')
ax.legend(loc='best')

# make predictions
trainPredict = model_lor.predict(X)
testPredict = model_lor.predict(X_test)

# calculate root mean squared error
trainScore = []
testScore = []
for i in range(len(y)):
    train_error = math.sqrt(mean_squared_error(y[i], trainPredict[i,:]))
    trainScore.append(train_error)
for i in range(len(y_test)):
    test_error = math.sqrt(mean_squared_error(y_test[i], testPredict[i,:]))
    testScore.append(test_error)


train_RMSE_running_mean, train_maxs, train_mins = runningMean(trainScore, 10)
test_RMSE_running_mean, test_maxs, test_mins = runningMean(testScore, 10)


fig, ax = plt.subplots(figsize = (20, 8))
ax.fill_between(range(len(train_RMSE_running_mean)), (train_mins), (train_maxs), color='blue', alpha=.1)
ax.fill_between(np.arange(len(train_RMSE_running_mean), len(train_RMSE_running_mean) + len(test_RMSE_running_mean)), (test_mins), (test_maxs), col
ax.plot(train_RMSE_running_mean, color='blue', label='Train Score (RMSE)')
ax.plot(np.arange(len(train_RMSE_running_mean), len(train_RMSE_running_mean) + len(test_RMSE_running_mean)),test_RMSE_running_mean, color='orange'
ax.set_title('ANN Model RMSE')
ax.set_ylabel('Root Mean Squared Error (RMSE)')
ax.set_xlabel('Time step')
ax.set_yscale('log')
ax.legend(loc='best')


fig, ax1 = plt.subplots(figsize = (20, 8))
ax1.set_xlabel('time step')
ax1.set_ylabel('ANN Model Test Set RMSE', color='orange')
ax1.plot(range(len(test_RMSE_running_mean)), test_RMSE_running_mean, color='orange', label='Test Score (RMSE)')
ax1.set_yscale('log')

ax2 = ax1.twinx()  # instantiate a second axes that shares the same x-axis

ax2.set_ylabel('Time Steps to Transition', color='black')  # we already handled the x-label with ax1
ax2.plot(range(len(y_test)), y_test, color='black')

fig.tight_layout()  # otherwise the right y-label is slightly clipped
plt.show()
```

```
# In[195]:


fig, ax = plt.subplots()
ax.boxplot(testScore)
ax.set_title('ANN Model RMSE')
ax.set_ylabel('Root Mean Squared Error (RMSE)')
ax.set_yscale('log')

#testScore = np.flatten(np.array(testScore), axis=1)

q1 = np.quantile(testScore, 0.25)#get lower quartile threshold of test RMSE values
q2 = np.quantile(testScore, 0.5)#get lower quartile threshold of test RMSE values
q3 = np.quantile(testScore, 0.75)#get lower quartile threshold of test RMSE values
q4 = max(testScore)#get lower quartile threshold of test RMSE values

q1_inds = np.where(testScore < lq)#get indices where the RMSE is lower than the lower quartile threshold
q2_inds = np.where((testScore < q2) & (testScore > q1))#get indices where the RMSE is in the second quartile threshold
q3_inds = np.where((testScore < q3) & (testScore > q2))#get indices where the RMSE is lower than the lower quartile threshold
q4_inds = np.where((testScore < q4) & (testScore > q3))#get indices where the RMSE is lower than the lower quartile threshold

q1_steps = y_test[q1_inds]
q2_steps = y_test[q2_inds]
q3_steps = y_test[q3_inds]
q4_steps = y_test[q4_inds]

my_dict = {'Q1': q1_steps, 'Q2': q2_steps,'Q3': q3_steps, 'Q4':q4_steps}

fig, ax = plt.subplots()
ax.boxplot(my_dict.values())
ax.set_title('Distribution of Steps Until Transition Values by RMSE Quartile')
ax.set_ylabel('Root Mean Squared Error (RMSE)')
ax.set_xticklabels(my_dict.keys())

fig, ax = plt.subplots()
counts, xedges, yedges, im = ax.hist2d(y_test, testScore, bins = 100, cmap=plt.cm.jet)
ax.set_title('Distribution of Steps Until Transition Values versus RMSE')
ax.set_ylabel('Root Mean Squared Error (RMSE)')
ax.set_xlabel('Steps Until Transition')
fig.colorbar(im, ax=ax)
```