A STUDY OF MACHINE LEARNING TECHNIQUES FOR DYNAMICAL SYSTEM PREDICTION

by

Rishi Pawar

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree of

Master of Science
in Mathematics

at
The University of Wisconsin-Milwaukee
May 2022

ABSTRACT

A STUDY OF MACHINE LEARNING TECHNIQUES FOR DYNAMICAL SYSTEM PREDICTION

by

Rishi Pawar

The University of Wisconsin-Milwaukee, 2022
Under the Supervision of Professor Dexuan Xie

Dynamical Systems are ubiquitous in mathematics and science and have been used to model many important application problems such as population dynamics, fluid flow, and control systems. However, some of them are challenging to construct from the traditional mathematical techniques. To combat such problems, various machine learning techniques exist that attempt to use collected data to form predictions that can approximate the dynamical system of interest. This thesis will study some basic machine learning techniques for predicting system dynamics from the data generated by test systems. In particular, the methods of Dynamic Mode Decomposition (DMD), Sparse Identification of Nonlinear Dynamics (SINDy), Singular Value Decomposition (SVD), and Deep Neural Network (DNN) regression will be studied. Such techniques provide alternatives to determine the dynamics of a system of interest without needing to resort to the computationally expensive elementary methods. From numerically testing a few linear and nonlinear systems of ordinary differential equations, it was observed that the methods of DMD and SVD could approximate linear systems effectively but performed poorly against nonlinear systems. The approach of DNN regression proved effective for both linear and nonlinear dynamical systems.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank Professor Dexuan Xie for advising me through this thesis as well as introducing me to the field of machine learning. I would also like to thank Professors Istvan Lauko and Gabriella Pinter for agreeing to be part of my thesis committee. In addition, I would like to thank Professor Pinter for doing directed reading with me and introducing me to the field of nonlinear dynamics which served as the foundation for this thesis. I would like to thank all my educators who taught me mathematics over the years and who motivated me to pursue and understand this fascinating subject. I would also like to thank my parents for supporting me in my work as well as providing me with guidance over the years. I would like to thank all my friends and colleagues who assisted me in my endeavours. Finally, I would like to thank the University of Wisconsin Milwaukee for admitting me into their graduate program, helping me continue my mathematics education, and providing me with future research and career directions.

Chapter 1

Introduction

A dynamical system is usually defined as a system that changes in time. Dynamical systems have been widely used to model population dynamics, orbit trajectories, fluid flow, control systems, etc. One primary type of dynamical system is a system of differential equations. These differential equations can be ordinary differential equations (ODEs) or partial differential equations (PDEs). This thesis will focus on first-order autonomous systems of ODEs that have the following appearance:

$$\begin{cases} \frac{d\vec{x}(t)}{dt} = f(\vec{x}) \quad \text{for t} > t_0 \\ \vec{x}(t_0) = \vec{x}_0, \end{cases} \tag{1.1}$$

where $\vec{x}(t)$ is an n-dimensional vector function of time t; $\frac{d\vec{x}}{dt}$ is the first derivative of $\vec{x}(t)$, meaning that the first time derivative of every component of $\vec{x}(t)$ is taken; $f(\vec{x})$ is an n-dimensional vector function of $\vec{x}(t)$; and $\vec{x}_0$ denotes an initial value of $\vec{x}$ at the starting time t_0. The above ODE system is autonomous, since the vector function $f(\vec{x})$ depends on $\vec{x}$ only. The ODE system is non-autonomous, if f depends on time t as well, i.e. $f(\vec{x}, t)$ [6]. We can usually set $t_0 = 0$ as this will not alter the underlying dynamics, rather shift our window in time. For simplicity in writing, we will adopt this convention in future exploration. The problem posed in (1.1) is commonly referred to as an initial value problem (IVP)[6].

When each component of f is a linear function of $\vec{x}$, we get a linear dynamical system

of the form:

$$\begin{cases} \frac{d\vec{x}(t)}{dt} = A\vec{x}(t) \quad \text{for t} > 0, \\ \vec{x}(0) = \vec{x}_0, \end{cases} \tag{1.2}$$

where A is a constant matrix of order $n \times n$, i.e. $A \in \mathbb{R}^{n \times n}$ and $f(\vec{x}) = A\vec{x}$.

In real life problems, dynamical systems of interest that need to be solved often have unknown $f(\vec{x})$ or A and must be hypothesized and predicted approximately through collected data and intuition. If this combination of data and intuition suggests that the dynamical system is linear, then the solution to (1.2) will be a linear combination of exponential functions, whose coefficients are determined by the initial condition $\vec{x}_0$. The difficulty in solving (1.2) is dependent on the system's number of variables. In most cases, these dynamical systems are nonlinear and nonlinear techniques are needed to determine their solutions. Since nonlinear techniques are often developed based on linear approximations, techniques for solving the linear system (1.2) can be valuable [2].

Modern techniques of optimization and machine learning have been used to determine the formulations of dynamical systems, which can then be solved through a variety of numerical solutions. The reason for the success of machine learning in the prediction of dynamics for given systems comes from the ability of machine learning to use collected data to learn patterns within the data to generate an approximation function that minimizes some loss criteria [2]. Generally, a trained model that exhibits low loss both over training and validation data is a successfully developed model.

This thesis will describe several procedures to use data generated from test systems to make predictions about these systems. In particular, the methods of Singular Value Decomposition (SVD), Dynamic Mode Decomposition (DMD), Sparse Identification of Nonlinear Dynamics (SINDy), and Machine Learning regression will be studied. For clarity, their studies are presented in Chapters 3, 4, 5, and 6 respectively, along with some results we observed during the numerical tests we conducted using these methods.

Chapter 2

Elementary Theory and Analytical Solution Techniques

2.1 Basic Theory

2.1.1 Existence and Uniqueness of Solutions

Let f_i and x_i denote the i-th component of vectors f and $\vec{x}$, respectively. We assume that a system of the form (1.1) satisfies the hypotheses of an existence and uniqueness theorem given in [10] for $f(\vec{x}, t)$: "If f is continuous and all of its partial derivatives $\partial f_i/\partial x_j$ for $i, j = 1, \ldots, n$ are continuous in $\vec{x}$ for some open connected set $D \subset \mathbb{R}^n$, then for $\vec{x}_0 \in D$, (1.1) has a solution $\vec{x}(t)$ over some time interval, and the solution is unique." Clearly, system (1.2) will possess a unique solution.

2.1.2 Trajectories of Solution Curves

For initial value problems of the form (1.2), the solutions of these problems are continuous functions in time that trace out trajectories in $\mathbb{R}^n$. A given initial condition will generate a unique trajectory through the phase space $\mathbb{R}^n$; in particular, this implies that 2 different initial conditions will generate 2 different solution curves that will never intersect. If these solution curves were to intersect, then an initial condition starting at the point of intersection would generate 2 different trajectories which is not permitted [10]. For

homogeneous linear systems such as (1.2), the origin represents a fixed point, a point whose derivative is the zero vector $\vec{0}$. Fixed points are of interest since they are points that the solution trajectories will either converge to, diverge away from, or orbit around. The behaviors of solutions around a given fixed point are determined by the eigenvalues of the matrix A in (1.2). If all eigenvalues λ have $Re(\lambda) < 0$, $Re(\lambda)$ refers to the real part of λ, then all solution trajectories will converge to the origin along the direction of the eigenvector of the largest magnitude eigenvalue. If at least 1 eigenvalue has $Re(\lambda) > 0$, then solution trajectories whose initial conditions are not asymptotically close to the eigenvectors with $Re(\lambda) < 0$ will diverge along the direction of the eigenvector with $Re(\lambda) > 0$. The previous analysis provides insight to the general behavior of solution curves for a given linear system.

The same analysis can be used to provide insights to the behavior of solution curves near fixed points of nonlinear systems via linear approximation by the Jacobian matrix. Determining the eigenvalues of the Jacobian matrix evaluated at the fixed point will give the local behavior of solution trajectories [10].

Example of a Globally Stable System

Consider the following system:

$$\begin{aligned} \frac{dx}{dt} &= -23/5x - 6y - 24/5z, \\ \frac{dy}{dt} &= 2x + y + 2z, \\ \frac{dz}{dt} &= 6/5x + 2y + 3/5z, \end{aligned} \tag{2.1}$$

which can be written in the form of (1.2) with

$$\vec{x} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \quad A = \begin{bmatrix} -23/5 & -6 & -24/5 \\ 2 & 1 & 2 \\ 6/5 & 2 & 3/5 \end{bmatrix}.$$

The matrix A of this system has eigenvalues $-1, -1-2i$, and $-1+2i$ and their associated eigenvectors $(-1,-1,2)^T$, $(3,-1+i,-1)^T$, and $(3,-1-i,-1)^T$. Here $i = \sqrt{-1}$,

since all the real parts of the eigenvalues are negative, all solution curves will converge to a fixed point at $t = 0$, $(0, 0, 0)$. Since we know the eigenvalues and eigenvectors, we can claim that the solution of (2.1) will have the form

$$\vec{x}(t) = c_1 e^{-t} \begin{bmatrix} -1 \\ -1 \\ 2 \end{bmatrix} + c_2 e^{(-1-2i)t} \begin{bmatrix} 3 \\ -1+i \\ -1 \end{bmatrix} + c_3 e^{(-1+2i)t} \begin{bmatrix} 3 \\ -1-i \\ -1 \end{bmatrix}$$

With Euler's formula $e^{i\theta} = \cos\theta + i\sin\theta$, we can rewrite the second and third exponential terms as

$$e^{(-1-2i)t} = e^{-t}(cos2t - isin2t),$$
$$e^{(-1+2i)t} = e^{-t}(cos2t + isin2t).$$

Combining with their respective vectors and constants, we get real and imaginary portions for each vector

$$c_2 e^{-t} \begin{bmatrix} 3cos2t \\ -cos2t + sin2t \\ -cos2t \end{bmatrix} + c_2 i e^{-t} \begin{bmatrix} -3sin2t \\ sin2t + cos2t \\ sin2t \end{bmatrix},$$
$$c_3 e^{-t} \begin{bmatrix} 3cos2t \\ -cos2t + sin2t \\ -cos2t \end{bmatrix} + c_3 i e^{-t} \begin{bmatrix} 3sin2t \\ -sin2t - cos2t \\ -sin2t \end{bmatrix}.$$

We can combine the real parts and imaginary parts to get

$$(c_2 + c_3) e^{-t} \begin{bmatrix} 3cos2t \\ -cos2t + sin2t \\ -cos2t \end{bmatrix} + (c_2 - c_3) i e^{-t} \begin{bmatrix} 3sin2t \\ -sin2t - cos2t \\ -sin2t \end{bmatrix}.$$

We set $c_2' = c_2 + c_3$ and $c_3' = i(c_2 - c_3)$. Since our solutions must exist in $\mathbb{R}^3$, coefficients c_2' and c_3' will be real numbers and our transformed equation will have the following

appearance

$$\vec{x}(t) = c_1 e^{-t} \begin{bmatrix} -1 \\ -1 \\ 2 \end{bmatrix} + c_2' e^{-t} \begin{bmatrix} 3cos(2t) \\ -cos(2t) + sin(2t) \\ -cos(2t) \end{bmatrix} + c_3' e^{-t} \begin{bmatrix} -3sin(2t) \\ sin(2t) + cos(2t) \\ sin(2t) \end{bmatrix} \tag{2.2}$$

where c_1, c_2', c_3' are determined by the initial condition.

Fundamentally, the imaginary portions of the eigenvalues will cause trajectories to exhibit spiral-like behavior. Four sample trajectories are displayed in Figure (2.1). We generated trajectories by using an ODE solver (scipy.integrate.odeint) from the SCIPY library, where a random initial condition was selected from the closed cube $[-20, 20]^3$, a time interval was set as $[0, 5]$ with a time stepsize of 0.001 [8]. From the figure, it is clear that each trajectory asymptotically converges to the origin.

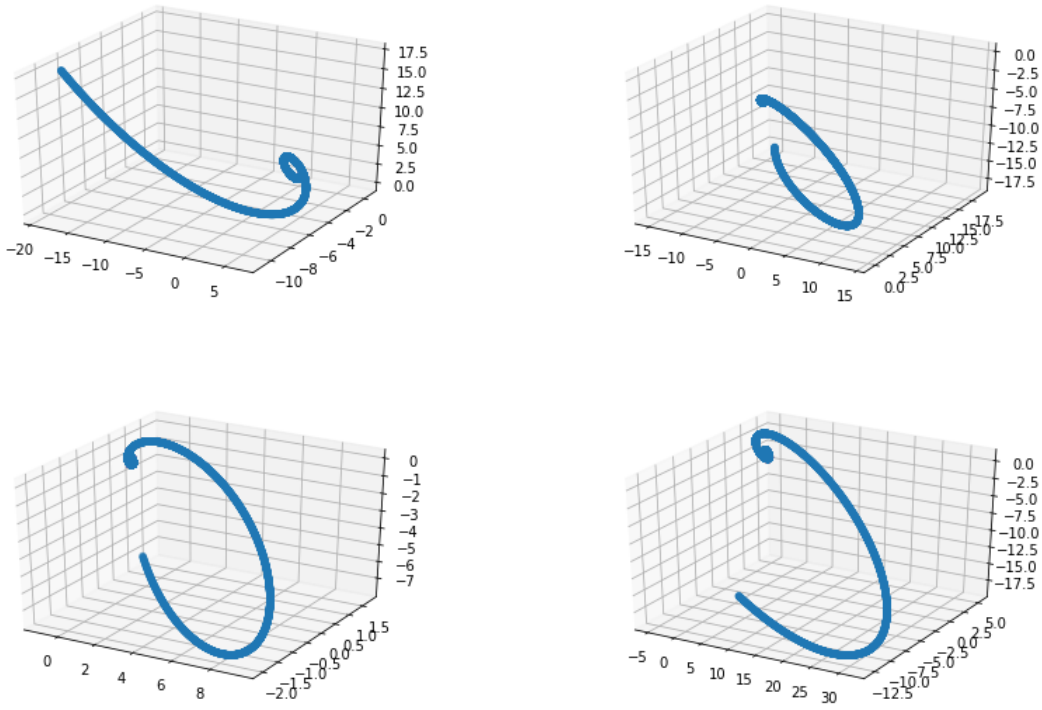

Figure 2.1: Top left image has an initial condition of (-19, -9, 17) and a final position of (-0.046, -0.012, 0.087). Top Right image has an initial condition of (-15, -19, -18) and a final position of (0.208, 0.115, -0.224). Bottom left image has an initial condition of (-1, 1, -7) and final position of (0.003, 0.075, -0.051). Bottom right image has an initial condition of (7, -10, -15) and final position of (-0.141, 0.217, -0.039).

2.2 Spectral Decomposition

When $n = 1$, the system (1.2) becomes a scale equation,

$$\begin{cases} \frac{dx(t)}{dt} = ax(t) \quad \text{for t} > 0, \\ x(0) = x_0, \end{cases} \tag{2.3}$$

whose solution is given by

$$x(t) = x_0 e^{at} \tag{2.4}$$

However, when $n > 1$, it is difficult to get an expression of the analytical solution $x(t)$ of (1.2). Even so, it has been known that system (1.2) can be decoupled to n scalar differential equations, greatly simplifying the study of (1.2) theoretically and numerically. This decoupling process is commonly referred to as a spectral decomposition. It is this spectral decomposition that makes a linear system of (1.2) particularly valuable in the dynamical model development. Hence, we give it a detailed description.

Motivated by the solution (2.4) of (2.3), we take an ansatz about the solution function of (1.2), an ansatz in the form $\vec{x}(t) = e^{\lambda t}\vec{v}$, where λ is a constant scalar and $\vec{v}$ is a constant vector. We then can get that

$$\begin{aligned} \dot{\vec{x}} = \lambda e^{\lambda t}\vec{v} \quad \text{and} \quad A\vec{x} = Ae^{\lambda t}\vec{v} \\ \Rightarrow \quad \lambda\vec{v} = A\vec{v} \end{aligned} \tag{2.5}$$

This is significant because it is indicating that in order for $e^{\lambda t}\vec{v}$ to be a solution, λ and $\vec{v}$ must satisfy (2.5), which is an eigenvalue problem. That is λ and $\vec{v}$ must be an eigenvalue and an eigenvector pair of matrix A.

Suppose that A has n eigenpairs (eigenvalues λ_i and their associated eigenvectors $\vec{v}_i$), going back to the original problem (1.2), our solution vector $\vec{x}(t)$ will be of the form

$$\vec{x}(t) = \sum_{i=1}^{n} c_i e^{\lambda_i t}\vec{v}_i, \tag{2.6}$$

where c_i is the $i-th$ constant to be determined by the initial condition $\vec{x}(0) = \vec{x}_0$.

While the above solution technique works, it is very expensive computationally. This means that for an extremely large matrix the above spectral decomposition is not feasible to implement. Hence, more computationally efficient methods for solving (1.2) been developed over the years without computing the eigenvalues and eigenvectors of coefficient matrix A.

As a brief detour, we next define matrices D and P by

$$D = \begin{bmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_n \end{bmatrix} \quad P = \begin{bmatrix} \vec{v}_1 & \vec{v}_2 & \cdots & \vec{v}_n \end{bmatrix}, \tag{2.7}$$

and the $n \times n$ matrix A as $A = PDP^{-1}$. Then we can reformulate (1.2) as

$$(P^{-1}\vec{x})' = D(P^{-1}\vec{x}) \tag{2.8}$$

Setting a new vector function, $\vec{y}$ by

$$\vec{y} = P^{-1}\vec{x} \tag{2.9}$$

we convert (2.8) into a simpler equivalent system,

$$\dot{\vec{y}} = D\vec{y} \tag{2.10}$$

Or n independent scalar equations

$$\frac{dy_i}{dt} = \lambda_i y_i, \quad i = 1, 2, \ldots, n \tag{2.11}$$

whose solution y_i is given by $y_i(t) = c_i e^{\lambda_i t}$. Hence, by (2.9), the solution $\vec{x}(t)$ of (1.2) is given as a linear combination of the components of $\vec{y}(t)$. The spectral decomposition method is an interesting excursion; however, it is dependent on knowing the eigenvalues

and eigenvectors of the matrix A, whose calculation can be very costly for large n. Hence, it is not guaranteed to work in all instances.

From (2.9), it is evident that $\vec{x} = P\vec{y}$; therefore, $\vec{y}$ must be determined to solve $\vec{x}$. We can get

$$\vec{x}(t) = P\vec{y}(t) = \begin{bmatrix} \vec{v}_1 & \vec{v}_2 & \dots & \vec{v}_n \end{bmatrix} \begin{bmatrix} y_1(t) \\ y_2(t) \\ \vdots \\ y_n(t) \end{bmatrix} = \sum_{i=1}^{n} y_i(t)\vec{v}_i = \begin{bmatrix} \sum_{i=1}^{n} v_{i1}y_i(t) \\ \sum_{i=1}^{n} v_{i2}y_i(t) \\ \vdots \\ \sum_{i=1}^{n} v_{in}y_i(t) \end{bmatrix}$$

$$\text{or} \quad \left[x_j(t) = \sum_{i=1}^{n} v_{ij}y_i(t), \quad j = 1, 2, \dots, n \right]$$

$$\text{where we have denoted } \vec{v}_i \text{ by} \quad \vec{v}_i = \begin{bmatrix} v_{i1} \\ v_{i2} \\ \vdots \\ v_{in} \end{bmatrix}, \quad \text{for } i = 1, 2, \dots, n$$

This shows that each component $x_j(t)$ of our solution vector $\vec{x}(t)$ will be a linear combination of the solutions $y_i(t)$ of (2.11) with coefficients v_{ij} being the j-th component of eigenvectors $\vec{v}_i$ for $i = 1, 2, \dots, n$.

To illustrate the above method, we consider a simple system of (1.2) as follows: with initial condition $\vec{x}_0$ and A.

$$\begin{bmatrix} \frac{dx_1}{dt} \\ \frac{dx_2}{dt} \\ \frac{dx_3}{dt} \end{bmatrix} = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 3 & 0 \\ 1 & 0 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}, \tag{2.12}$$

Where we have set A as a symmetric real constant matrix, i.e,

$$A = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 3 & 0 \\ 1 & 0 & 3 \end{bmatrix}$$

The matrix of (2.12) can be diagonalized as $A = PDP^{-1}$ with

$$D = \begin{bmatrix} 3 & 0 & 0 \\ 0 & 2+\sqrt{6} & 0 \\ 0 & 0 & 2-\sqrt{6} \end{bmatrix} \quad \text{and} \quad P = \begin{bmatrix} 0 & -1+\sqrt{6} & -1-\sqrt{6} \\ -1 & 2 & 2 \\ 2 & 1 & 1 \end{bmatrix}.$$

The solutions of (2.10) for this example can be found analytically in the expression

$$\vec{y} = \begin{bmatrix} y_1(t) \\ y_2(t) \\ y_3(t) \end{bmatrix} = \begin{bmatrix} c_1 e^{3t} \\ c_2 e^{(2+\sqrt{6})t} \\ c_3 e^{(2-\sqrt{6})t} \end{bmatrix}.$$

Using $\vec{x} = P\vec{y}$, we then get the solution $\vec{x}$ of (2.12) as shown below:

$$\vec{x} = \begin{bmatrix} x_1(t) \\ x_2(t) \\ x_3(t) \end{bmatrix} = P\vec{y} = \begin{bmatrix} 0 & -1+\sqrt{6} & -1-\sqrt{6} \\ -1 & 2 & 2 \\ 2 & 1 & 1 \end{bmatrix} \begin{bmatrix} c_1 e^{3t} \\ c_2 e^{(2+\sqrt{6})t} \\ c_3 e^{(2-\sqrt{6})t} \end{bmatrix}$$

$$= c_1 e^{3t} \begin{bmatrix} 0 \\ -1 \\ 2 \end{bmatrix} + c_2 e^{(2+\sqrt{6})t} \begin{bmatrix} -1+\sqrt{6} \\ 2 \\ 1 \end{bmatrix} + c_3 e^{(2-\sqrt{6})t} \begin{bmatrix} -1-\sqrt{6} \\ 2 \\ 1 \end{bmatrix}$$

The coefficients c_1, c_2, c_3 are determined from the initial condition $\vec{x}(0) = \vec{x}_0$. Setting $t = 0$ in the above system, we get

$$\begin{bmatrix} x_1(0) \\ x_2(0) \\ x_3(0) \end{bmatrix} = \begin{bmatrix} 0 & -1+\sqrt{6} & -1-\sqrt{6} \\ -1 & 2 & 2 \\ 2 & 1 & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix}$$

The above linear system can be solved for the coefficients c_1, c_2, and c_3 easily (e.g. by the Gaussian elimination method). Despite this analysis, most systems of interest are nonlinear and numerical approaches must be used to either determine an approximate solution or determine a formulation for the dynamics.

Chapter 3

Singular Value Decomposition

Singular Value Decomposition (SVD) is a factorization of a matrix $A_{m\times n}$, as follows:

$$A = U\Sigma V^*, \tag{3.1}$$

where U and V are unitary matrices of orders $m \times m$ and $n \times n$, respectively, Σ is a diagonal matrix of $m \times n$ with ordered singular values, and * operator refers to the conjugate transpose.

Suppose we want to determine the dynamics of an autonomous dynamical system of the form of (1.1) just from collected state and derivative data, then we may use SVD to determine the parameters of our dynamical system with some degree of accuracy. If the dynamical system is linear, then we will be able to correctly determine the parameters of the dynamical system. For a nonlinear system, we will be able to determine parameters that best linearly approximate the dynamics of the system over the observed time window. The length of the time window depends on how strong the nonlinearity affects the dynamics of the system.

3.1 Linear Case

If our dynamical system is linear and autonomous, then it will have the form of eq (1.2);

$$\frac{d\vec{x}}{dt} = A\vec{x}$$

Suppose we make m observations of $\vec{x}$ and $\frac{d\vec{x}}{dt}$ and collect these observations into the transposes of matrices $\mathbf{X}$ and $\dot{\mathbf{X}}$, in the fashion of equation (5.1), then the i-th column of $\mathbf{X}^T$ and $\dot{\mathbf{X}}^T$ will represent $\vec{x}_i$ and $\frac{d\vec{x}_i}{dt}$ at t_i respectively. Both matrices $\mathbf{X}^T$ and $\dot{\mathbf{X}}^T$ will be order $n \times m$. The above problem can be reformulated as

$$\dot{\mathbf{X}}^T = A\mathbf{X}^T \tag{3.2}$$

The parameter matrix A maps every column vector $\vec{x}$ of $\mathbf{X}^T$ to its respective derivative vector $\frac{d\vec{x}}{dt}$ in matrix $\dot{\mathbf{X}}^T$.

Calculating the SVD of $\mathbf{X}^T$ and multiplying the pseudo-inverse of $\mathbf{X}^T$ to the left hand side of (3.2) will yield an analytic expression for A.

$$A = \dot{\mathbf{X}}^T V \Sigma^{-1} U^* \tag{3.3}$$

Since matrices U and V are unitary, their inverses are their transposes. The matrix Σ^{-1} will be an $m \times n$ matrix whose non-zero entries along the diagonal will contain reciprocal singular values.

The construction of (3.3) is analytically correct, and executable for low dimensional data. For very high dimensional data, (3.3) is prone to numerical inaccuracies due to round off and truncation errors from the many multiplications that occur while performing the matrix product. A more numerically accurate approach is to use the reduced order SVD instead of the full SVD. The reduced order SVD will use the first r singular values of matrix Σ and the first r singular vectors from matrices U and V.

In most cases, the data matrix $\mathbf{X}^T$ will be a matrix of $\mathbb{R}^{n \times m}$ and one of the dimensions will much larger than the other (either $m << n$ or $n << m$). With this in mind, we

can claim that the rank of our data matrix $\mathbf{X}^T$ will be at most $N = min(m, n)$. We can further reduce the order of our data matrix based off the magnitude of the singular values. If the magnitudes of the first r singular values are much greater than that of the remaining $N - r$, then we can create a reduced order SVD utilizing only the first r singular values and associated r singular vectors from matrices U and V. The formulation of equation (3.3) becomes

$$A \approx \dot{\mathbf{X}}^T \tilde{V} \tilde{\Sigma}^{-1} \tilde{U}^* = \hat{A}, \tag{3.4}$$

where $\tilde{U}_{m\times r}$, $\tilde{\Sigma}_{r\times r}$, $\tilde{V}^*_{r\times n}$, and $\hat{A}$ is the low rank SVD approximation to A. This particular approach is beneficial since it reduces the number of multiplications performed and consequently reduces the impact of rounding and truncation errors. For the sample tests performed, the reduced order SVD approach yielded the exact same solution as the full order SVD approach.

The reduced order technique was applied to a test dynamical system and the derived parameter matrix $\hat{A}$ closely approximated the original dynamical system matrix A.

The dynamical system tested had the form

$$\begin{aligned} \tfrac{dx}{dt} &= -12x + 12y - 3z \\ \tfrac{dy}{dt} &= -4x - 2y - z \\ \tfrac{dz}{dt} &= -2x + 4y + 3z \end{aligned} \qquad \Longleftrightarrow \qquad \begin{bmatrix} dx \\ dy \\ dz \end{bmatrix} = \begin{bmatrix} -12 & 12 & -3 \\ -4 & -2 & -1 \\ -2 & 4 & 3 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{3.5}$$

We generated data by creating a linear derivative function that took in the inputs of (x, y, z) and returns the associated derivative vector at that location of (x, y, z). Looking over the time interval $T = [0, 1]$ with a timestep of 0.01 and initial condition $(-8, 8, 27)$, we used scipy's integrate function to generate the phase trajectory for a particle with the given initial data [8]. The integrate function will move us to the next spatial point and we will calculate the derivative at that location. By this method, we created a collection of ordered spatial points and their associated derivatives. As before, matrix $\mathbf{X}^T$ will contain all spatial column vectors $(x, y, z)^T$ and for our scenario will have the dimensions of 3×100. Matrix $\dot{\mathbf{X}}^T$ will have the same dimensions. The phase space trajectory of system (3.5) is given in Figure (3.1).

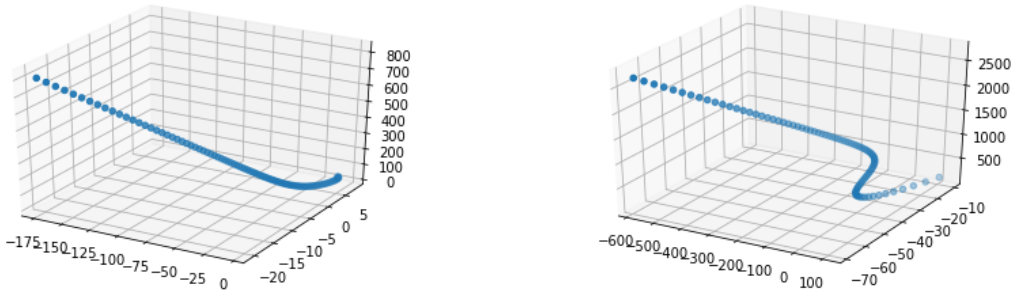

Figure 3.1: The figure on the left shows the trajectory data $\mathbf{X}^T$; clearly the trajectory of (3.5) with the chosen initial condition diverges. The figure on the right shows the path of $\dot{\mathbf{X}}^T$; clearly the derivative vector of each point $(x, y, z)^T$ increases in magnitude.

The objective was to recover the matrix of (3.5) just from the collected data matrices $\mathbf{X}^T$ and $\dot{\mathbf{X}}^T$. Applying the formulation of (3.2) with (3.4), we get

$$A_{3\times 3} \approx \dot{\mathbf{X}}^T_{3\times 100} \tilde{V}^*_{100\times 3} \tilde{\Sigma}^{-1}_{3\times 3} \tilde{U}_{3\times 3} = \hat{A}_{3\times 3}$$

For our given data, the rank of our data matrix $\mathbf{X}^T$ is 3, so we can form a reduced order model utilizing the 3 singular values and associated 3 singular vectors from matrices U and V. From this construction, the predicted parameter matrix A was determined to be

$$\hat{A} = \begin{bmatrix} -12. & 12. & -3. \\ -4. & -2. & -1. \\ -2. & 4. & 3. \end{bmatrix} \approx \begin{bmatrix} -12 & 12 & -3 \\ -4 & -2 & -1 \\ -2 & 4 & 3 \end{bmatrix}$$

Clearly, the derived parameter matrix is nearly identical to the original matrix of (3.5).

To further test the efficacy of this method, we evaluated the average error of the observations from $\dot{\mathbf{X}}^T$ and the approximated observations generated from the product of our derived parameter matrix and the corresponding position from $\mathbf{X}^T$, i.e. $\hat{A}\vec{x}$. Explicitly,

$$error = \left\| \frac{d\vec{x}}{dt} - \hat{A}\vec{x} \right\|_F$$

The average error over our 100 observations was determined to be approximately 1.56×10^{-12} with a maximum error of approximately 5.63×10^{-12}. From this test, we can

definitively say that the reduced order SVD approach can be used to effectively determine the dynamics of a linear dynamical system. However, for a dynamical system which is nonlinear, the efficacy of SVD is dependent heavily on the time interval that is used, the initial value, and the nature of the nonlinear system itself.

3.2 Nonlinear Case

If our dynamical system is nonlinear but autonomous, we will not be able to represent it as a matrix vector product. However, some useful insights can be gained from attempting to apply the linear technique of SVD to nonlinear data.

3.2.1 Weakly Nonlinear System

The first nonlinear system we shall test is system (4.13) for reference,

$$\begin{bmatrix} dx \\ dy \\ dz \end{bmatrix} = \begin{bmatrix} 1x - 1y + 3z \\ -4xy - z \\ -2x + 4y - 3z \end{bmatrix}; \tag{3.6}$$

however, we will additionally impose the initial condition $(-1, 1, 1)$ and use scipy's integrate method to generate positional and derivative data. For the chosen initial value, system (3.6) represents a "weakly" nonlinear system; a system that is nonlinear but bounded for subset time intervals less than 5. The summary data is listed in Table (3.1).

To calculate the "error" of derived $\hat{A}$ matrix, we simply take the 2-norm of the difference of approximated derivative data (product of approximation matrix $\hat{A}$ with position $\vec{x}$) with the actual derivative data. Explicitly,

$$\begin{aligned} error &= \left\| \hat{A}\vec{x} - \frac{d\vec{x}}{dt} \right\|_2 \\ \text{for } \vec{x} &= (x_1, x_2, \ldots, x_n)^T \\ ||\vec{x}||_2 &= \sqrt{x_1^2 + x_2^2 + \ldots + x_n^2} \end{aligned} \tag{3.7}$$

Nonlinear Coefficient -4 with initial condition (-1,1,1)				
Time Interval	Step Size	Min Error	Max Error	Avg Error
[0,5]	0.001	1.1096e-05	3.1070	0.6286
[0,5]	0.01	4.2718e-05	3.1116	0.6314
[0,5]	0.1	0.0004	3.0650	0.7000
[0,1]	0.001	0.0021	2.2024	1.3155
[0,1]	0.01	0.0195	2.1971	1.3184
[0,1]	0.1	0.0148	2.1502	1.3069
[0,0.5]	0.001	0.0015	0.7385	0.2260
[0,0.5]	0.01	0.0015	0.6344	0.2220
[0,0.5]	0.1	0.0731	0.2328	0.1647
[0,0.25]	0.001	0.0005	0.1749	0.0480
[0,0.25]	0.01	0.0004	0.1584	0.0507

Table 3.1: For weakly nonlinear system (3.6), the errors are reasonably restrained for all time intervals.

The average error is the average of all the norms from collected data. We can graphically evaluate the performance of this linear approximation method in Figure (3.2).

From Figure (3.2), we can see that for times less than 0.5 seconds the trajectories generated by the SVD approximated derivative closely match the original trajectories. Beyond 0.5 seconds, the trajectories diverge significantly and one must either re-linearize the system and use a new initial condition or utilize a different method.

Since this system is "weakly" nonlinear for the chosen initial value and coefficients of terms, the error is reasonably restrained. Furthermore, the derived parameter matrix $\hat{A}$ tended to match perfectly the coefficients of the linear terms of the dynamical system. For example, data collected over the time interval $T = [0, 1]$ with a time step of 0.001

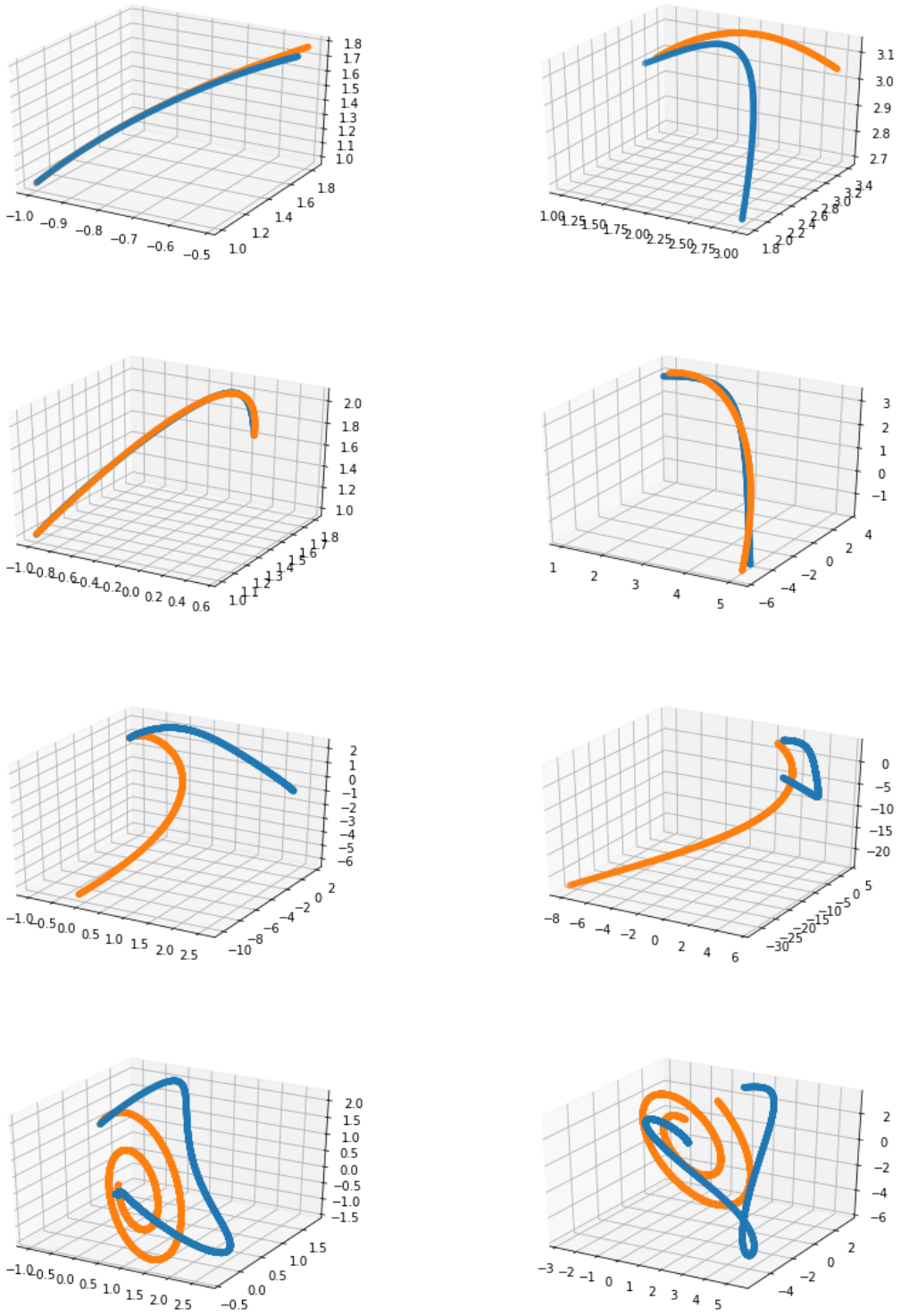

Figure 3.2: The graphs on the left show the trajectories $(x, y, z)^T$ from initial condition $(-1, 1, 1)$, while the graphs on the right show the derivative trajectories for each point. The blue paths represent the original data while the orange paths represent the SVD approximation (0.25, 0.5, 1, 5 from top to bottom).

yielded the following parameter matrix

$$\hat{A}_{0.001} = \begin{bmatrix} 1 & -1 & 3 \\ -0.08945121 & 6.69785451 & -5.92855664 \\ -2 & 4 & -3 \end{bmatrix}$$

Clearly, the coefficients for the linear terms perfectly match those from the original system. This behavior was observed throughout all tested dynamical systems that possessed at least 1 nonlinear equation. As another observation, parameter matrices that were based off data from the same time interval, e.g. $T = [0, 1]$, all had similar parameter values for the nonlinear equation row. For the time interval $T = [0, 1]$ we partitioned the interval with step sizes of 0.001, 0.01, and 0.1. Below are the parameter matrices for the step sizes of 0.01 and 0.1.

$$\hat{A}_{0.01} = \begin{bmatrix} 1 & -1 & 3 \\ -0.07148221 & 6.77724583 & -5.98834744 \\ -2 & 4 & -3 \end{bmatrix}$$

$$\hat{A}_{0.1} = \begin{bmatrix} 1 & -1 & 3 \\ 0.14406132 & 7.63729844 & -6.64345672 \\ -2 & 4 & -3 \end{bmatrix}$$

This behavior is expected since the SVD approximation attempts to minimize the least squares error between the input and output. Decreasing the step size increases the number of points needed minimize the least squares error so it won't change the values of the parameters needed to minimize the system.

Given the relatively restrained error for each tested interval and step size length, it is feasible to use a reduced order SVD approximation to roughly estimate the dynamics for a weakly nonlinear system. This observation is expected but there is some utility in witnessing it first hand.

An item of interest would be to consider the effectiveness of the SVD approximation applied to a stronger nonlinear system.

3.2.2 Lorenz System

For personal interest, we shall explore how the SVD approximation fares in attempting to approximate the dynamics for the famous chaotic Lorenz System.

The Lorenz System is defined formally as

$$\begin{bmatrix} dx \\ dy \\ dz \end{bmatrix} = \begin{bmatrix} \sigma(y-x) \\ x(\rho-z)-y \\ xy-\beta z \end{bmatrix} \rightarrow \begin{bmatrix} 10(y-x) \\ x(28-z)-y \\ xy-8/3z \end{bmatrix} \tag{3.8}$$

For the chosen parameter values of σ, ρ, and β the Lorenz system will exhibit chaotic behavior [10]. Even though the Lorenz system is chaotic, it is still a bounded system. The chaotic nature comes from estimating the long term trajectory of a given initial condition. Chaotic systems are sensitive to initial conditions and 2 initial conditions that are separated by a distance of ϵ will have completely disparate trajectories after a certain period of time [10]. The error has the potential to accumulate and degrade any long term prediction attempts.

From Table (3.2), it is clear that the SVD approximation method to determine the dynamics of the Lorenz System is highly ineffective for most time intervals. Only for a time interval of $T = [0, 0.05]$ can we obtain errors which are reliably under 1. Given how minimal this time interval is, the SVD approach is definitely not ideal for the task. We can more definitively see how ineffective the SVD approximation is for the Lorenz system in Figure (3.3). Clearly, these graphs show that linear approximations to the Lorenz system are mostly ineffective in approximating the dynamics. Only under an extremely small time window is a linear approximation valid.

We can clearly see that for the same time steps, the validation error is not significantly different from the error gained from the difference between our approximated $\mathbf{X}$ and $\dot{\mathbf{X}}^T$.

Lorenz System with initial condition (-8,8,27)				
Time Interval	Step Size	Min Error	Max Error	Avg Error
[0,5]	0.001	26.1846	288.0060	79.2740
[0,5]	0.01	26.2378	288.1263	79.4930
[0,5]	0.1	27.3909	277.2806	80.8556
[0,1]	0.001	8.4580	90.6518	41.3511
[0,1]	0.01	8.8085	91.0210	41.5148
[0,1]	0.1	10.6996	86.9428	40.5739
[0,0.5]	0.001	1.5038	66.8803	32.3721
[0,0.5]	0.01	2.4573	59.6190	31.9170
[0,0.5]	0.1	17.0482	43.0236	26.8837
[0,0.1]	0.001	0.0481	16.8108	3.7319
[0,0.1]	0.01	0.2063	11.5841	4.9308
[0,0.05]	0.001	0.0198	1.4916	0.4655
[0,0.05]	0.01	0.1370	0.4737	0.3017

Table 3.2: The SVD approach works best for very small time intervals.

3.3 Striving for more accuracy

It is evident from the chosen examples that the linearity of the SVD approximation hinders its ability to accurately determine the dynamics of a typical nonlinear system. The condition that the dynamical system is "weakly" nonlinear is circumstantial since the system can have stable and unstable manifolds that alter the trajectories of initial conditions living on or between these manifolds [10]. If one were to successfully utilize the SVD approach s/he would need to "relinearize" the system, or calculate a new SVD approximation over each spatial interval, thus yielding a piecewise linear function. However, having the SVD method operate successfully on an arbitrary nonlinear dynamical system, without heavy alteration, is circumstantial and more sophisticated techniques are required to determine the dynamics of a system simply from collected data.

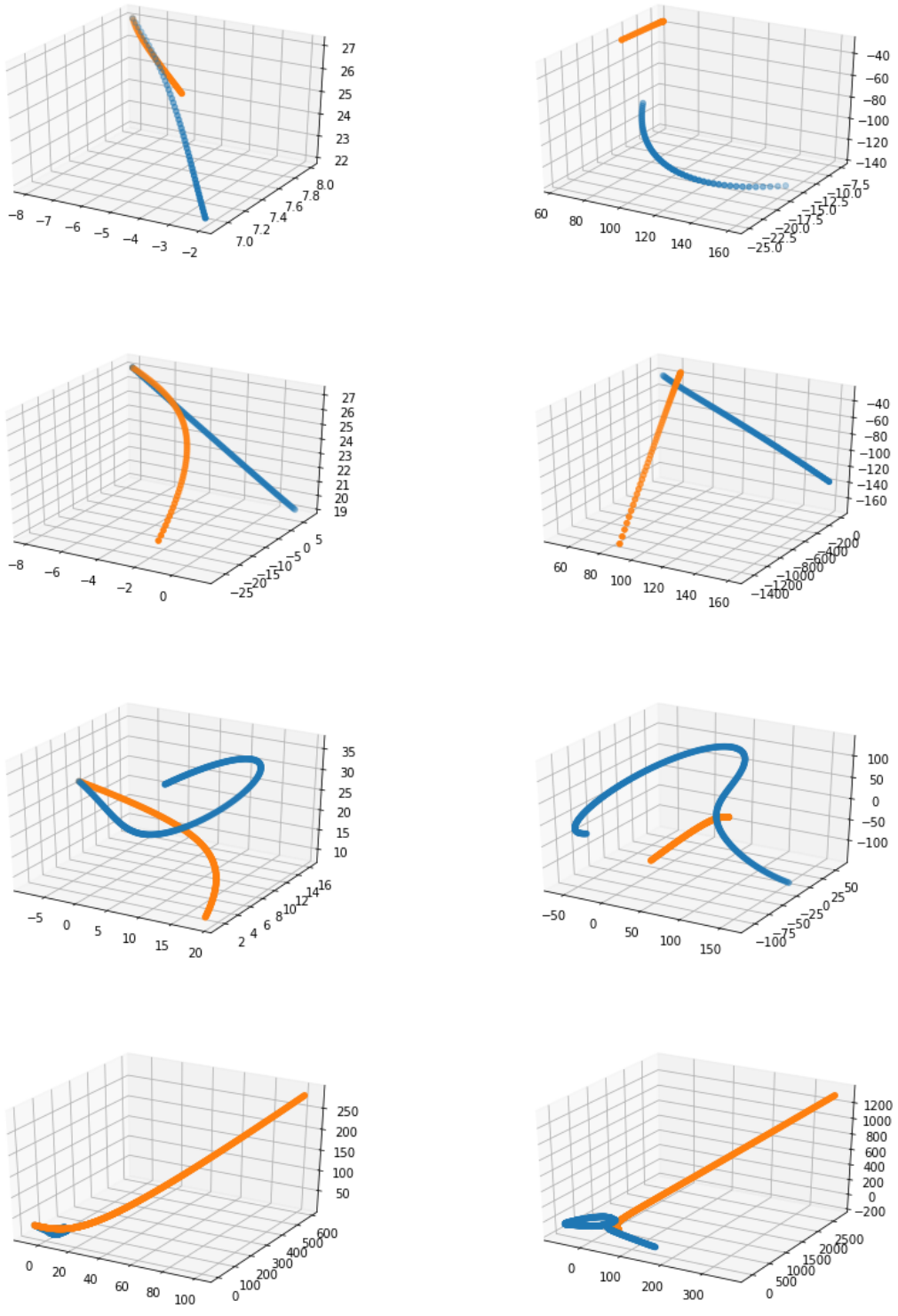

Figure 3.3: The graphs on the left show the trajectories $(x, y, z)^T$ from initial condition $(-8, 8, 27)$, while the graphs on the right show the derivative trajectories for each point. The blue paths represent the original data while the orange paths represent the SVD approximation (0.05, 0.1, 0.5, 1 from top to bottom).

Chapter 4

Dynamic Mode Decomposition

4.1 Background

Dynamic Mode Decomposition (DMD) is now introduced as a method to try to understand the dynamics of a system. This algorithm was first proposed by mathematician Peter Schmid in 2008 to analyze fluid dynamics. The algorithm descibed below is based off of Schimd's design along with Kutz and Brunton's formulation [9] [2].

To use this method, we do not require knowledge of the underlying dynamics. Suppose we observe a given system and collect n-dimensional data over a time interval $[0, T]$. We discretize the time interval into $m+1$ subintervals of equal length by $t_0 < t_1 < t_2 < \ldots < t_m < t_{m+1} = T$, where $t_i = i\tau$ with time step $\tau = \frac{T}{m+1}$. Thus we have $m+2$ vectors $\vec{x}(t_i)$ for $i = 0, 1, 2, \ldots, m, m+1$.

$$\vec{x}(t_i) = \begin{bmatrix} x_1(t_i) \\ x_2(t_i) \\ \vdots \\ x_n(t_i) \end{bmatrix}$$

For simplicity, we denote the jth component of vector $\vec{x}(t_i)$ as x_j^i for $j = 1, 2, \ldots, n$ and $i = 0, 1, 2, \ldots, m, m+1$. We organize these $m+2$ vectors into ordered pairs $\{(\vec{x}(t_k), \vec{x}(t_{k+1})\}_{k=0}^{m}$ for $k = 0, 1, 2, \ldots, m$. It is clear that each ordered pair indicates

1 time step forward by $\vec{x}(t)$. For convenience, we set $t'_k = t_k + \tau$,

$$\vec{x}(t_k) = \begin{bmatrix} x_1(t_k) \\ x_2(t_k) \\ \vdots \\ x_n(t_k) \end{bmatrix}, \quad \vec{x}(t'_k) = \begin{bmatrix} x_1(t_k + \tau) \\ x_2(t_k + \tau) \\ \vdots \\ x_n(t_k + \tau) \end{bmatrix}. \tag{4.1}$$

We then organize these vectors as two matrices:

$$\mathbf{X}_{n\times m} = \begin{bmatrix} \vec{x}(t_0) & \vec{x}(t_1) & \dots & \vec{x}(t_m) \end{bmatrix}, \quad \mathbf{X}'_{n\times m} = \begin{bmatrix} \vec{x}(t_1) & \vec{x}(t_2) & \dots & \vec{x}(t_{m+1}) \end{bmatrix} \tag{4.2}$$

We further make the assumption that there exists a linear mapping A that maps input vector $\vec{x}(t_i)$ to output vector $\vec{x}(t_{i+1})$ such that

$$\vec{x}(t_{i+1}) = A\vec{x}(t_i)$$

If there does not exist an A that exactly relates all vectors $\vec{x}(t_i)$ to $\vec{x}(t_{i+1})$ for all time values t_i, we instead strive for an A whose multiplication with $\vec{x}(t_i)$ approximates $\vec{x}(t_{i+1})$ for all time values t_i,

$$\vec{x}(t_{i+1}) \approx A\vec{x}(t_i)$$

that is valid for all t_i values. Over a sufficiently small time step τ, we can assume that the linear operator A can closely relate the input and output vectors, according to what Brunton and Kutz claim in [2] that, "if the system is nonlinear but slowly varying, a multiple scale argument can permit the assumption of a linear tangent approximation". From the data matrices $\mathbf{X}$ and $\mathbf{X}'$ of (4.2), the best fit operator A can be found as

$$A = \mathbf{X}'\mathbf{X}^{\dagger} \tag{4.3}$$

in the sense of minimizing the loss function

$$\mathcal{L}(A) = \frac{1}{2}||\mathbf{X}' - A\mathbf{X}||_F^2, \tag{4.4}$$

where $||\cdot||_F$ is the Frobenius norm (squared norm) and $\mathbf{X}^\dagger$ is the Moore-Penrose pseudo-inverse of $\mathbf{X}$, which can be defined as

$$\mathbf{X}^\dagger = \mathbf{X}^*(\mathbf{X}\mathbf{X}^*)^{-1}. \tag{4.5}$$

where * operation is the complex conjugate transpose operator.

We give a proof of the best fit formula (4.3) as follows:

Let a_{ij} denote the (i,j) entry of A for $i,j = 1,2,\ldots,n$. We can express A as

$$A = \begin{bmatrix} \vec{a_1}^T \\ \vec{a_2}^T \\ \vdots \\ \vec{a_n}^T \end{bmatrix}$$

where $\vec{a_i}^T$ represents the i-th row of matrix A, i.e.

$$\vec{a_i}^T = \begin{bmatrix} a_{i1} & a_{i2} & \ldots & a_{1n} \end{bmatrix}.$$

This way, we can construct a loss function in terms of the i-th row of matrix A as

$$\mathcal{L}(\vec{a_i}^T) = \frac{1}{2}\|\vec{a_i}^T\mathbf{X} - \mathbf{X}'_{R_i}\|_F^2,$$

where $\mathbf{X}'_{R_i}$ refers to the i-th row of matrix $\mathbf{X}'$. Taking the partial derivative of $\mathcal{L}(\vec{a_i}^T)$ with respect to parameter a_{ik} yields the following expression

$$\frac{\partial \mathcal{L}(\vec{a_i}^T)}{\partial a_{ik}} = \sum_{j=1}^{m}(\vec{a_i}^T\vec{x}_j - x'_{ij})x_{kj},$$

where $\vec{x}_j$ is the j-th column vector of matrix $\mathbf{X}$, x'_{ij} is the entry of matrix $\mathbf{X}'$ located at the (i,j) position, and x_{kj} is the entry of matrix $\mathbf{X}$ located at the (k,j) position. The term $\vec{a_i}^T\vec{x}_j$ represents the inner product taken between the vectors $\vec{a_i}^T$ and $\vec{x}_j$. Using this

derivative formula, the gradient vector of $\mathcal{L}(\vec{a_i}^T)$ can be written as

$$\nabla\mathcal{L}(\vec{a_i}^T) = \left[\sum_{j=1}^m (\vec{a_i}^T\vec{x}_j - x'_{ij})x_{1j} \quad \sum_{j=1}^m (\vec{a_i}^T\vec{x}_j - x'_{ij})x_{2j} \quad \dots \quad \sum_{j=1}^m (\vec{a_i}^T\vec{x}_j - x'_{ij})x_{nj}\right]_{1\times n}$$

Combining all the above gradient row vectors for $i = 1, 2, \dots, n$ together into a matrix, we obtain

$$\nabla\mathcal{L}(A) = \begin{bmatrix} \sum_{j=1}^m (\vec{a_1}^T\vec{x}_j - x'_{1j})x_{1j} & \sum_{j=1}^m (\vec{a_1}^T\vec{x}_j - x'_{1j})x_{2j} & \dots & \sum_{j=1}^m (\vec{a_1}^T\vec{x}_j - x'_{1j})x_{nj} \\ \vdots & \ddots & & \vdots \\ \sum_{j=1}^m (\vec{a_n}^T\vec{x}_j - x'_{nj})x_{1j} & \sum_{j=1}^m (\vec{a_n}^T\vec{x}_j - x'_{nj})x_{2j} & \dots & \sum_{j=1}^m (\vec{a_n}^T\vec{x}_j - x'_{nj})x_{nj} \end{bmatrix}_{n\times n}$$

This formulation can be rewritten into

$$\nabla\mathcal{L}(A) = (A\mathbf{X} - \mathbf{X}')\mathbf{X}^T$$

Setting $\nabla\mathcal{L}(A)$ to be the zero matrix $0_{n\times n}$, we can explicitly solve this for A to get the matrix equation.

$$A = \mathbf{X}'\mathbf{X}^T(\mathbf{X}\mathbf{X}^T)^{-1} = \mathbf{X}'\mathbf{X}^\dagger$$

This completes the proof of (4.3). Therefore, the matrix A of (4.3) that minimizes the loss function of (4.4).

We can further show that this formulation of the pseudo-inverse is equivalent to that involving the SVD. If we let $\mathbf{X} = U\Sigma V^T$, then

$$\begin{aligned} \mathbf{X}^T(\mathbf{X}\mathbf{X}^T)^{-1} &= (U\Sigma V^T)^T(U\Sigma V^T(U\Sigma V^T)^T)^{-1} \\ &= V\Sigma^T U^T(U\Sigma V^T V\Sigma^T U^T)^{-1} \\ &= V\Sigma^T U^T(U\Sigma\Sigma^T U^T)^{-1} \\ &= V\Sigma^T U^T U(\Sigma^T)^{-1}\Sigma^{-1}U^T \\ &= V\Sigma^{-1}U^T. \end{aligned}$$

For most practical purposes, using the SVD to obtain the pseudo-inverse is preferable

since it avoids the expensive operation of taking the inverse of a matrix.

4.2 DMD Algorithm

Let r be the rank of the data matrix $\mathbf{X}$. The algorithm has the following steps:

1. Compute the reduced order Singular Value Decomposition (SVD) of $\mathbf{X}$

$$\mathbf{X} = \tilde{U}\tilde{\Sigma}\tilde{V}^*, \tag{4.6}$$

where the * operation refers to the complex conjugate transpose; $\tilde{U}$ and $\tilde{V}$ are orthogonal matrices of orders $n \times r$ and $m \times r$ respectively satisfying the relations $\tilde{U}_{n\times r}\tilde{U}^*_{r\times n} = I_{n\times n}, \tilde{U}^*_{r\times n}\tilde{U}_{n\times r} = I_{r\times r}, \tilde{V}_{m\times r}\tilde{V}^*_{r\times m} = I_{m\times m}, \tilde{V}^*_{r\times m}\tilde{V}_{m\times r} = I_{r\times r}$; and $\tilde{\Sigma}$ is a diagonal matrix of order $r \times r$ that collects the r nonzero singular values of $\mathbf{X}$.

2. Compute reduced matrix $\tilde{A}$

We start with the original problem

$$\mathbf{X}' = A\mathbf{X}$$

Combined with (4.6),

$$\begin{aligned} \mathbf{X}' &= A\tilde{U}\tilde{\Sigma}\tilde{V}^* \\ \tilde{U}^*\mathbf{X}' &= \tilde{U}^*A\tilde{U}\tilde{\Sigma}\tilde{V}^* \\ \tilde{U}^*\mathbf{X}'\tilde{V}\Sigma^{-1} &= \tilde{U}^*A\tilde{U}. \end{aligned}$$

Set $\tilde{A} = \tilde{U}^*A\tilde{U}$ implies that

$$\tilde{A} = \tilde{U}^*\mathbf{X}'\tilde{V}\Sigma^{-1} \tag{4.7}$$

Reduced matrix $\tilde{A}$ is of order $r \times r$ and has the same nonzero eigenvalues as matrix A. The construction $\tilde{U}^*A\tilde{U}$ is a similarity transformation into the lower rank space $\tilde{U}$ [2].

3. Extract eigenvalues and eigenvectors of $\tilde{A}$

   Since matrix $\tilde{A}$ is of order $r \times r$, its dimensions are much smaller than that of A and diagonalization is feasible.

$$\tilde{A} = \Phi \Lambda \Phi^{-1} \tag{4.8}$$

   Λ possesses the eigenvalues of $\tilde{A}$ that are equivalent to those of A. The columns of Φ correspond to the eigenvectors of $\tilde{A}$. Both Λ and Φ are of order $r \times r$.

4. Reconstruct the eigenvectors of A

   The eigenvalues of $\tilde{A}$ and A are equivalent;

$$\begin{aligned} \tilde{A} &= \Phi \Lambda \Phi^{-1} \\ \tilde{U}^* A \tilde{U} &= \Phi \Lambda \Phi^{-1} \\ A &= \tilde{U} \Phi \Lambda \Phi^{-1} \tilde{U}^* \end{aligned}$$

   From the above, A has the same eigenvalues as $\tilde{A}$ with eigenvectors $\tilde{U}\Phi$. We define

$$\Psi = \tilde{U} \Phi \tag{4.9}$$

   where the columns of $\Psi_{n \times r}$ are the eigenvectors of A. The eigenvectors Ψ are called the DMD modes.

4.3 DMD Expansion

Since the eigenvalues and eigenvectors of our system have been determined, we can perform data-driven spectral decomposition to determine the system's state at time t_k.

$$\vec{x}_k = \sum_{j=1}^{r} \vec{\psi}_j \lambda_j^k c_j = \Psi \Lambda^k \vec{c} \tag{4.10}$$

where $\vec{\psi}_j$ are the DMD modes, λ_j is the corresponding DMD eigenvalues, and c_j is the mode amplitude. From (4.10), $\vec{c}$ can be calculated as

$$\vec{c} = \Psi^{\dagger}\vec{x}_0 \tag{4.11}$$

where $\Psi^{\dagger}$ is the pseudo-inverse of Ψ. The spectral decomposition introduced in (4.10) can be written in continuous time with the introduction of continuous eigenvalues $\omega = log(\lambda)/\tau$:

$$\vec{x}(t) = \sum_{j=1}^{r} \vec{\psi}_j e^{\omega_j t} c_j = \Psi e^{\Omega t}\vec{c} \tag{4.12}$$

where Ω is a diagonal matrix with eigenvalues ω_j and $e^{\Omega t}$ is the matrix exponential of a diagonal matrix Ωt; taking the exponential of a diagonal matrix is equivalent to applying the exponential function to each diagonal entry thus yielding another diagonal matrix [2] [6].

4.4 Simple Tests Involving DMD

To see the efficacy of the DMD algorithm, we shall perform some numerical tests using two simple systems.

4.4.1 Linear System

The first system we shall observe is system (2.1) with a randomly chosen initial condition of $(15, 0, 19)$. For reference, system (2.1) is

$$\begin{bmatrix} \frac{dx}{dt} \\ \frac{dy}{dt} \\ \frac{dz}{dt} \end{bmatrix} = \begin{bmatrix} -23/5 & -6 & -24/5 \\ 2 & 1 & 2 \\ 6/5 & 2 & 3/5 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}.$$

Equation (2.2) gives the form of the analytical solution and using the provided initial condition, we determine the coefficients as

$$\begin{bmatrix} 15 \\ 0 \\ 9 \end{bmatrix} = \begin{bmatrix} -1 & 3 & 0 \\ -1 & -1 & 1 \\ 2 & -1 & 0 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2' \\ c_3' \end{bmatrix} \Rightarrow \begin{bmatrix} c_1 \\ c_2' \\ c_3' \end{bmatrix} = \begin{bmatrix} \frac{72}{5} \\ \frac{49}{5} \\ \frac{121}{5} \end{bmatrix}.$$

Therefore the analytical solution to the above IVP is

$$\vec{x}(t) = \frac{72}{5}e^{-t}\begin{bmatrix} -1 \\ -1 \\ 2 \end{bmatrix} + \frac{49}{5}e^{-t}\begin{bmatrix} 3cos(2t) \\ -cos(2t) + sin(2t) \\ -cos(2t) \end{bmatrix} + \frac{121}{5}e^{-t}\begin{bmatrix} -3sin(2t) \\ sin(2t) + cos(2t) \\ sin(2t) \end{bmatrix}.$$

We shall observe system (2.1) over the time interval $T = [0, 5]$ with a stepsize of 0.001 and the chosen initial condition. Using scipy's odeint function, we generate trajectory data and utilize some initial proportion p of the data to generate the DMD modes and the latter proportion $1 - p$ to evaluate the forecasting capability [8]. For example, if we use 50% of the collected data for the DMD algorithm, that means the other 50% of the remaining data was unused for the DMD algorithm. DMD forecasting uses the last point of the initial 50% of data as an initial condition to generate a prediction trajectory that will then be compared to the unused data to check the accuracy of the method. Results are displayed in Table (4.1). From the collected results, it is clear that for a linear system, DMD yields highly accurate forecasts. Even a small proportion of data can be used to forecast a very accurate trajectory. Figure (4.1) visually displays the accuracy of DMD forecasting applied to a linear system.

DMD Interpolation and Extrapolation with (2.1)				
% of Data	Inter/Extra	Min Error	Max Error	Avg Error
0.8	Inter	0.0000e+00	1.5862e-10	6.8819e-11
0.2	Extra	0.0000e+00	5.0594e-12	3.1475e-12
0.5	Inter	0.0000e+00	6.7626e-11	3.2271e-11
0.5	Extra	0.0000e+00	8.3779e-12	4.7948e-12
0.1	Inter	0.0000e+00	1.2024e-11	6.8062e-12
0.9	Extra	0.0000e+00	6.9282e-10	2.5006e-10
0.01	Inter	0.0000e+00	4.3226e-12	1.5700e-12
0.99	Extra	0.0000e+00	2.0162e-07	7.3534e-08
0.001	Inter	0.0000e+00	9.3133e-13	4.6419e-13
0.999	Extra	0.0000e+00	1.0135e-04	4.7785e-05

Table 4.1: Inter and Extra mean interpolation and extrapolation of the data. Inter means row percentage of data was used for the DMD algorithm and forecasting from the initial value is equivalent to interpolation of that data. Extra means the remaining proportion of data was checked with the DMD forecasting.

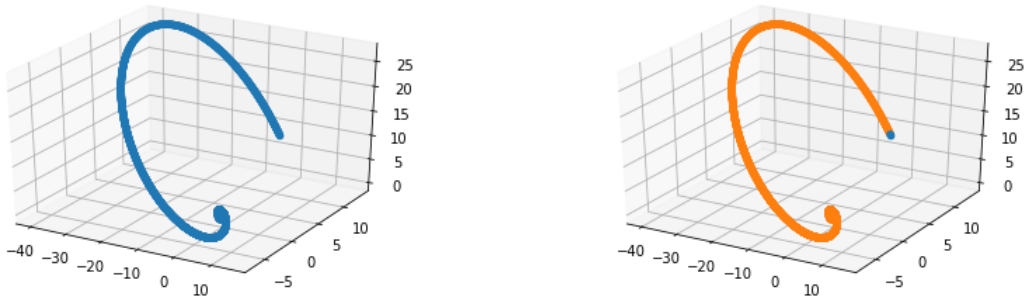

Figure 4.1: The left figure shows the original trajectory of system (2.1) with initial condition $(15, 0, 19)$ and the orange trajectory of the right figure shows the forecasting based off 0.1% collected data (blue portion).

4.4.2 Nonlinear System

The second system that we shall test is

$$\begin{bmatrix} \frac{dx}{dt} \\ \frac{dy}{dt} \\ \frac{dz}{dt} \end{bmatrix} = \begin{bmatrix} 1x - 1y + 3z \\ -4xy - z \\ -2x + 4y - 3z \end{bmatrix} \tag{4.13}$$

This system is nonlinear and possesses 2 fixed points: $(0, 0, 0)$ and $(1/6, 1/18, -1/27)$. We may use the previous eigenvalue analysis to predict local behavior around these fixed points; however, global behavior cannot be completely determined from linear techniques alone. To display the effect of the nonlinearity, the initial condition $(85, 26, 31)$ will be used. Using a time interval of $[0, 5]$ with a step size of 0.001, IVP (4.13) has the graph shown in Figure (4.2):

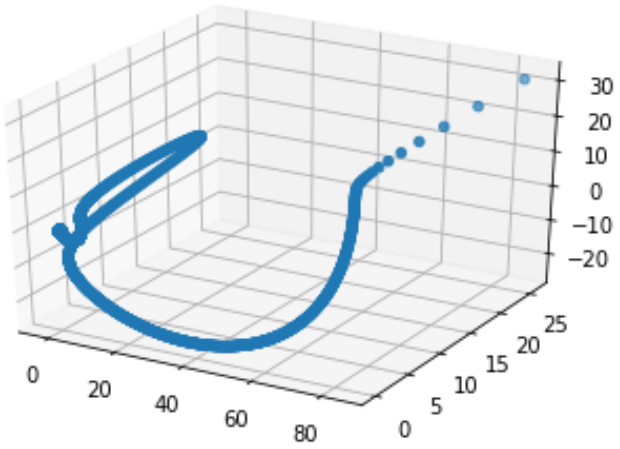

Figure 4.2: Starting at (85, 26, 31), the trajectory of (4.13) attempts to converge to $(0, 0, 0)$.

The same approach to determine the accuracy of DMD interpolation and extrapolation is used. Table (4.2) shows the accuracy of DMD applied to a nonlinear system. The graphs of the actual and predicted trajectories provide additional insights to Table (4.2). Clearly, the DMD algorithm is incapable of correctly forecasting the sudden deflection near the origin as either the interpolation or extrapolation max error becomes large, indicating that it is trying to anticipate the sudden nonlinearity near the origin.

From Figure (4.3), it is clear that the DMD algorithm can accurately forecast nonlinear data over some small interval of time, or sections of data that are sufficiently linear in nature. It is possible to use DMD to forecast the system's state over this time period before "relinearizing" the system, or applying the DMD algorithm over the remaining data [2]. It should also be noted that DMD is a linear method, meaning that the presence

DMD Interpolation and Extrapolation with (4.13)				
% of Data	Inter/Extra	Min Error	Max Error	Avg Error
0.8	Inter	0.0000e+00	2.1862e+01	1.2439e+00
0.2	Extra	0.0000e+00	6.3759e-03	5.2481e-03
0.5	Inter	0.0000e+00	2.1862e+01	1.9807e+00
0.5	Extra	0.0000e+00	2.0307e-02	1.0952e-02
0.1	Inter	0.0000e+00	2.1859e+01	8.3088e+00
0.9	Extra	0.0000e+00	3.9059e-01	6.9740e-02
0.05	Inter	0.0000e+00	7.7144e-01	1.7747e-01
0.95	Extra	0.0000e+00	2.2665e+01	7.8364e-01
0.01	Inter	0.0000e+00	1.7524e-03	6.3641e-04
0.99	Extra	0.0000e+00	2.2760e+01	7.6988e-01
0.001	Inter	0.0000e+00	9.5924e-04	5.0409e-04
0.999	Extra	0.0000e+00	2.2754e+01	7.6140e-01

Table 4.2: In Figure (4.2), when the trajectory approaches the origin, it suddenly deflects away from the origin before returning to it. This uptick is what causes the large errors in the rows of this table. In essence, DMD cannot correctly model the sudden deflection.

of nonlinearities in the data will severely cripple the efficacy of the algorithm as seen in Figure (4.4) where the DMD scheme must use the data from the nonlinear portion of the trajectory data. To treat nonlinear systems, we now turn to a newer algorithm known as the Sparse Identification of Nonlinear Dynamics (SINDy), which boasts the capability to determine the formulation of a dynamical system with nonlinearities, simply from gathered data.

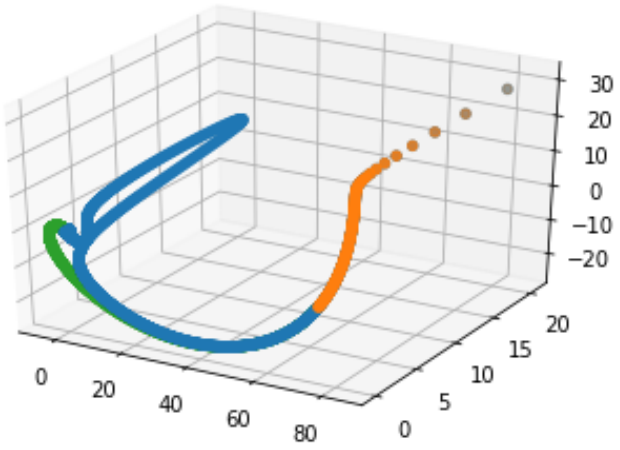

Figure 4.3: The above graph comes from using 0.1% of the data. The blue trajectory is the original data. The orange trajectory is an interpolation of the data used for the DMD algorithm; it overlaps the blue data with high accuracy. The green trajectory is the DMD extrapolation predicting unused data. The transition from the orange curve to the blue curve indicates the beginning of the DMD extrapolation. For a certain time, the green and blue trajectories are nearly indistinguishable.

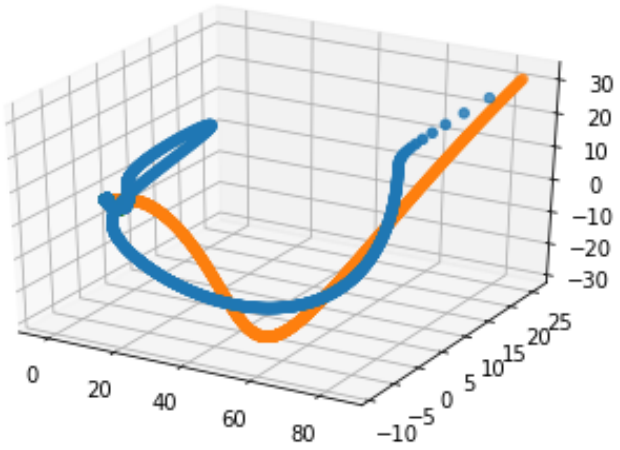

Figure 4.4: The above graph comes from using 10% of the data. Using this amount of data includes the deflection near the origin, thus significantly altering the DMD modes and degrading the effectiveness of the DMD algorithm's to forecast.

Chapter 5

Sparse Identification of Nonlinear Dynamics

5.1 Background

Sparse Identification of Nonlinear Dynamics (SINDy) is introduced as a method to determine the dynamics of nonlinear systems.

We collect data in a fashion similar to that described in Section (4.1). In addition to collected data $\vec{x}(t_i)$, we also collect derivative data $\dot{\vec{x}}(t_i)$; if derivative data is not available, it can be approximated via finite differences. The data is collected into the two matrices

$$\begin{aligned} \mathbf{X} &= \begin{bmatrix} \vec{x}(t_0) & \vec{x}(t_1) & \dots & \vec{x}(t_m) \end{bmatrix}^T \\ \dot{\mathbf{X}} &= \begin{bmatrix} \dot{\vec{x}}(t_0) & \dot{\vec{x}}(t_1) & \dots & \dot{\vec{x}}(t_m) \end{bmatrix}^T \end{aligned} \tag{5.1}$$

Explicitly matrix $\mathbf{X}$ will have the following appearance:

$$\mathbf{X} = \begin{bmatrix} x_1(t_0) & x_2(t_0) & \dots & x_n(t_0) \\ x_1(t_1) & x_2(t_1) & \dots & x_n(t_1) \\ \vdots & \vdots & \ddots & \vdots \\ x_1(t_m) & x_2(t_m) & \dots & x_n(t_m) \end{bmatrix}$$

with $\dot{\mathbf{X}}$ defined analogously. It should be noted that each column of $\mathbf{X}$ describes the evolution of one dynamical variable in time, while the rows describe the state of the system at time t_i.

For the given system (1.1), we will make the assumption that each of the governing equations of $f(\vec{x})$ involve only a few active terms, i.e. the right hand side functions are sparse for each equation. The goal is to create a linear model from a family of functions of collected data $\mathbf{X}$ and to determine the coefficients of active terms in the model. The benefit of such a sparse construction is interpretability as well as the idea that the sparsest (aka the simplest) representation of a system is usually the correct one, i.e. the principle of Occam's Razor. The construction can be summarized as

$$\dot{\mathbf{X}} = \Theta(\mathbf{X})\Xi \tag{5.2}$$

where the columns of matrix $\Theta(\mathbf{X})_{m\times l}$ are functions of data $\mathbf{X}$ and matrix $\Xi_{l\times n}$ has columns of sparse vectors $\vec{\xi} \in \mathbb{R}^l$ whose entries are coefficients to the terms appearing in each component function of $f(\vec{x})$. The dimension l refers to the number of different functions we decide to use to populate matrix $\Theta(\mathbf{X})$.

Our matrix of candidate nonlinear functions $\Theta(\mathbf{X})$ is constructed based off data $\mathbf{X}$ and may have appearance

$$\Theta(\mathbf{X}) = \begin{bmatrix} \mathbf{1} & \mathbf{X} & \mathbf{X}^2 & \dots & \mathbf{X}^d & \dots & sin(\mathbf{X}) \dots \end{bmatrix} \tag{5.3}$$

The submatrix $\mathbf{X}^d$ refers to the construction of d-th degree polynomials based off our observations $\mathbf{X}$. For example, if we record 3 variables x, y, z from our system, and we want to append submatrix $\mathbf{X}^2$ into our matrix $\Theta(\mathbf{X})$, $\mathbf{X}^2$ will have the appearance

$$\mathbf{X}^2 = \begin{bmatrix} x(t_0)^2 & x(t_0)y(t_0) & x(t_0)z(t_0) & y(t_0)^2 & y(t_0)z(t_0) & z(t_0)^2 \\ x(t_1)^2 & x(t_1)y(t_1) & x(t_1)z(t_1) & y(t_1)^2 & y(t_1)z(t_1) & z(t_1)^2 \\ \vdots & & \vdots & & & \vdots \\ x(t_m)^2 & x(t_m)y(t_m) & x(t_m)z(t_m) & y(t_m)^2 & y(t_m)z(t_m) & z(t_m)^2 \end{bmatrix}$$

In accordance with a parsimonious model, we want matrix Ξ to be as sparse as possible. Therefore, we attempt to minimize the output from the following objective function

$$\mathcal{L}(\vec{\xi_k}) = ||\dot{\mathbf{X}}_k - \Theta(\mathbf{X})\vec{\xi_k}||_2 + \lambda||\vec{\xi_k}||_1 \tag{5.4}$$

where $\vec{\xi_k}$ is the kth column of matrix Ξ, $\dot{\mathbf{X}}_k$ is the k-th column of $\dot{\mathbf{X}}$, and λ is a sparsity promoting parameter. The inclusion of the one norm $||\cdot||_1$ defined for $\vec{x} \in \mathbb{R}^n$ as

$$||\vec{x}||_1 = |x_1| + |x_2| + \ldots + |x_n|$$

ensures sparsity of vector $\vec{\xi_k}$. Sparse regression algorithms such as LASSO or sequential thresholded least-squares may be used to determine this vector $\vec{\xi_k}$ [2].

We can place vectors $\vec{\xi_k}$ into a dynamical system of the form

$$f(\vec{x})_i^k = \dot{x}_i^k = \Theta(\vec{x}_i)\vec{\xi_k}$$

In the above, $\dot{x}_i^k$ is the k-th component of $\dot{\vec{x}}$ evaluated at time t_i and $\Theta(\vec{x}_i)$ is a row vector of symbolic functions of $\vec{x}_i$, representing the t_i row of matrix $\Theta(\mathbf{X})$.

5.2 Schematic Explanation

To display how SINDy might be used, we consider the Lorenz system

$$\begin{cases} \dot{x} = \sigma(y - x) = -\sigma x + \sigma y \\ \dot{y} = x(\rho - z) - y = \rho x - y - xz \\ \dot{z} = xy - \beta z = xy - \beta z \end{cases} \tag{5.5}$$

If we take m+1 time points of observation, our data matrix $\mathbf{X}$ and derivative matrix $\dot{\mathbf{X}}$ will have the forms

$$\mathbf{X} = \begin{bmatrix} x(t_0) & y(t_0) & z(t_0) \\ x(t_1) & y(t_1) & z(t_1) \\ & \vdots & \\ x(t_m) & y(t_m) & z(t_m) \end{bmatrix}_{m\times 3} \qquad \dot{\mathbf{X}} = \begin{bmatrix} \dot{x}(t_0) & \dot{y}(t_0) & \dot{z}(t_0) \\ \dot{x}(t_1) & \dot{y}(t_1) & \dot{z}(t_1) \\ & \vdots & \\ \dot{x}(t_m) & \dot{y}(t_m) & \dot{z}(t_m) \end{bmatrix}_{m\times 3}$$

Adopting the form of eq (5.2) we can explicitly write our system as

$$\begin{bmatrix} \dot{x}(t_0) & \dot{y}(t_0) & \dot{z}(t_0) \\ \dot{x}(t_1) & \dot{y}(t_1) & \dot{z}(t_1) \\ & \vdots & \\ \dot{x}(t_m) & \dot{y}(t_m) & \dot{z}(t_m) \end{bmatrix}_{m\times 3} =$$

$$\begin{bmatrix} 1 & x(t_0) & y(t_0) & z(t_0) & x(t_0)^2 & x(t_0)y(t_0) & \dots & z(t_0)^5 \\ 1 & x(t_1) & y(t_1) & z(t_1) & x(t_1)^2 & x(t_1)y(t_1) & \dots & z(t_1)^5 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x(t_m) & y(t_m) & z(t_m) & x(t_m)^2 & x(t_m)y(t_m) & \dots & z(t_m)^5 \end{bmatrix}_{m\times l} \begin{bmatrix} 0 & 0 & 0 \\ -\sigma & \rho & 0 \\ \sigma & -1 & 0 \\ 0 & 0 & -\beta \\ & \vdots & \\ 0 & 0 & 0 \end{bmatrix}_{l\times 3}$$

If the kth column of the Ξ matrix is multiplied with $\Theta(\mathbf{X})$, we will end up with a vector of dimension $m \times 1$ that is algebraically equivalent to the kth column of $\dot{\mathbf{X}}$.

5.2.1 Final Remarks regarding SINDy

Setting the parameter λ is a balancing act of sorts and the ideal value for parameter λ is often determined through experimentation. Increasing the magnitude of λ sparsifies the solution for $\vec{\xi}_k$ while decreasing the magnitude of λ increases the number of active variables in $\vec{\xi}_k$. The SINDy algorithm can be reduced to DMD if we only allow $\Theta(\mathbf{X})$ to contain linear terms and we set parameter λ to 0 in (5.4). SINDy as an algorithm

has had significant success in identifying the coefficients of nonlinear functions in various dynamical systems. SINDy has even been extended to determine the terms of partial differential equations in the modified algorithm PDE-FIND. Despite the massive success SINDy has experienced, there are still challenges that must be overcome before SINDy is fully perfected. Some challenges that SINDy must contend with are the choice of variables for data collection, the choice of functions to populate $\Theta(\mathbf{X})$, and the amount and quality of data collected [2].

Chapter 6

Deep Neural Network Regression

6.1 Elementary Machine Learning Theory

Machine Learning with Deep Neural Networks (DNNs) is at its core a regression problem. From a correctly labeled dataset called training data, the objective of machine learning regression is to determine a function that approximates the function that generates the training data as well as forecasts unseen data with relative accuracy. The criteria for an appropriate function is the minimization of a loss function based off the parameters in the machine learning model. The neural network approach to machine learning has been used in various regression problems ranging from simple linear regression of a few variables to image classification for images with hundreds to thousands of variables. In regression problems, the data used to find appropriate values for the parameters of the model is called training data.

For a given observation $\vec{x}_j$, we assign a label to this observation in form $\vec{y}_j$. This label $\vec{y}_j$ can have various forms, ranging from a scalar value to a vector value. We hope to approximate a function f that relates $\vec{x}_j$ to $\vec{y}_j$, i.e. $\vec{y}_j = f(\vec{x}_j)$. In classification problems, $\vec{y}_j$ is usually a scalar referring to the index of some labeling scheme. In regression problems, $\vec{y}_j$ can be a vector representing the vector output for vector input $\vec{x}_j$. We combine the input and output data into ordered pairs of the form $(\vec{x}_j, \vec{y}_j)$. We next collect these ordered pairs into a set called the training data, labeled as $\mathcal{D}$. This data will be used to

train our DNN. In addition to our training data $\mathcal{D}$, we also create a validation dataset $\mathcal{V}$ containing ordered pairs that are not in our training set $\mathcal{D}$. We use these points to determine the accuracy of our trained model. Our DNN is essentially a function with a set of parameters $\mathbf{\Theta}$ with each $\theta \in \mathbf{\Theta}$ initialized randomly. Through training, these parameters θ are adjusted so that they can relate the inputs and outputs of $\mathcal{D}$ as closely as possible. The DNN model can essentially be denoted as a function of the form $g(\vec{x}; \mathbf{\Theta})$, with the eventual goal that $\vec{y}_j \approx g(\vec{x}_j; \mathbf{\Theta})$ optimally. We get this optimal approximation by attempting to minimize the error of some loss function $\mathcal{L}(\mathbf{\Theta})$ based off adjusting the parameters values θ. This optimal collection of parameters θ can be labeled as $\mathbf{\Theta}^*$ and has the following property that

$$\mathbf{\Theta}^* = argmin_{\mathbf{\Theta}}\, \mathcal{L}(\mathbf{\Theta}) \tag{6.1}$$

An explicit example of a loss function can be

$$\mathcal{L}(\mathbf{\Theta}) = \frac{1}{m} \sum_{j=1}^{m} \|g(\vec{x}_j; \mathbf{\Theta}) - \vec{y}_j\|_2^2$$

Common techniques to optimize the parameters θ involve gradient descent and modifications of gradient descent [7]. Once we have trained our model we calculate the loss over the validation dataset, as usual lower validation loss means a more successfully trained model.

To leverage the power of DNNs to predict the dynamics of test systems, we shall use Pytorch implementations [7]. We will consider some dynamical systems to test the mettle of simple perceptron (feed forward) schemes.

6.2 Linear System

The first system we shall discuss is a linear system. Our system will have the following form

$$\begin{array}{rcl} \frac{dx}{dt} &=& -4x+4y-3z \\ \frac{dy}{dt} &=& -4x-2y-z \\ \frac{dz}{dt} &=& -2x+4y+3z \end{array} \iff \begin{bmatrix} dx \\ dy \\ dz \end{bmatrix} = \begin{bmatrix} -4 & 4 & -3 \\ -4 & -2 & -1 \\ -2 & 4 & 3 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{6.2}$$

The above system is linear and our initial objective in utilizing machine learning will be to determine if the optimized parameters of our network converge to the values in the parameter matrix of (6.2). For this purpose, we shall create a simple network that possesses no hidden layers.

The input will be a 1×3 vector representing our positional vector (x, y, z), followed by a 3×3 matrix with an added bias vector of dimension 1×3, leading to the output vector of size 1×3 representing the derivative vector (dx, dy, dz). Structurally the network has the following appearance

$$[x,y,z] \to [x,y,z] \times \begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix}^T + [b_1,b_2,b_3] \to [dx,dy,dz] \tag{6.3}$$

In (6.3) our parameter matrix is transposed when multiplying the row vector $[x, y, z]$. Such is the standard operation when working with Pytorch; Pytorch prefers to work with row vectors rather than column vectors, in contrast some other machine learning packages.

When evaluating the parameter matrix $\hat{A}$ for (6.2), the model initialized had randomly generated starting parameters from the interval $[0, 1]$. Below is an example of the starting parameter matrix for $\hat{A}$.

$$\hat{A}_{initial} = \begin{bmatrix} -0.1954 & -0.2129 & -0.2127 \\ -0.3678 & 0.4839 & 0.2818 \\ -0.3923 & 0.3733 & 0.0726 \end{bmatrix} \quad \vec{b}_{initial} = \begin{bmatrix} -0.0593 & 0.1987 & 0.1889 \end{bmatrix}$$

For a correctly trained model, $\hat{A}$ should be sufficiently close to the parameter matrix of (6.2), with $\vec{b} = [0, 0, 0]$.

The loss function used was mean-squared error defined formally as

$$MSE = \frac{\sum_{i=1}^{n} ||\vec{v}_{ei} - \vec{v}_{ai}||_2^2}{n} \tag{6.4}$$

In the above formulation, $\vec{v}_{ei}$ represents the estimated i-th output vector, $\vec{v}_{ai}$ represents the actual i-th output vector and $|| \cdot ||_2^2$ represents the square of the Frobenius norm.

The ADAM optimizer was used with a learning rate of 10^{-3} [5][7]. The training scheme will run the training model over a subset of the data (called a mini-batch) and calculate the loss between the actual solution and the approximate solution. To minimize the loss, the values of the training parameters will be modified according to the ADAM optimizer algorithm. These updated weights will be used to approximate the solution and the process is repeated. Over each epoch, the values of the parameters will converge to those found in the matrix of (6.2). The following gives a sample for $\hat{A}$ after successfully training has occurred.

$$\hat{A}_{final} = \begin{bmatrix} -4.0000 & 4.0000 & -3.0000 \\ -4.0000 & -2.0000 & -1.0000 \\ -2.0000 & 4.0000 & 3.0000 \end{bmatrix} \quad \vec{b}_{final} = \begin{bmatrix} -9.5381e-06 & 5.0423e-07 & 1.0266e-05 \end{bmatrix}$$

To determine the parameters for (6.2), 2 approaches for data generation were taken. The differing data generation methods were then trained on similar networks.

6.2.1 Data Generation

1. **Randomly Generated Points**

   Since our dynamical system is autonomous, there is no explicit time dependence in the system, meaning only the input (x, y, z) is needed to determine the derivative at that position. Therefore, our problem of determining the dynamics of system (6.2) turns into a linear regression problem. Because order is irrelevant, we can randomly generate some number of points, in this case 10000, find their mappings, choose an appropriate batch size, and choose an appropriate number of epochs to train our model over. Table (6.1) shows some results from changing the domain of data, the mini-batch size and the approximate number of epochs before the training loss is less then 10^{-6}. When the training loss is at that magnitude, the estimated parameter matrix $\hat{A}$ has effectively converged to a close enough approximation to the desired matrix in equation (6.2).

Randomly Generated Point Data		
Data Domain	Batch Size	No of Epochs for training loss to be $< 10^{-6}$
$[0,1]^3$	16	20
$[0,1]^3$	32	40
$[0,1]^3$	64	60
$[0,10]^3$	16	30
$[0,10]^3$	32	50
$[0,10]^3$	64	80
$[0,100]^3$	16	30
$[0,100]^3$	32	80
$[0,100]^3$	64	120

Table 6.1: Clearly as the domain of our points increases, the number of epochs required for satisfactory training increases.

From Table (6.1), it is clear that for randomly generated linear data, using bounded data translates into fewer required epochs for convergence of the parameters of $\hat{A}$. Similarly using a smaller batch size tended to require fewer epochs.

2. **Randomly Generated Trajectories**

In contrast, we also used sequential trajectories generated from random initial conditions. Since we can control the time interval of observation and the number of time intervals used, we adjusted the number of trajectories calculated for data to ensure that we always uses 10000 datapoints for training; for example, for the time interval $T = [0, 1]$ with a stepsize of 0.001, we generated 10 trajectories to obtain 10000 datapoints. In another scenario if we use a time interval of $[0, 0.5]$ with the same time step of 0.001, then we generated 200 trajectories for data.

Randomly Generated Trajectory Data			
Time Interval	Data Domain	Batch Size	Number of Epochs for training loss to be $< 10^{-6}$
[0,0.05]	$[0, 1]^3$	16	30
[0,0.05]	$[0, 1]^3$	32	60
[0,0.05]	$[0, 1]^3$	64	120
[0,0.1]	$[0, 10]^3$	16	50
[0,0.1]	$[0, 10]^3$	32	70
[0,0.1]	$[0, 10]^3$	64	120
[0,1]	$[0, 100]^3$	16	160
[0,1]	$[0, 100]^3$	32	270
[0,1]	$[0, 100]^3$	64	360

Table 6.2: From the above, trajectory data requires more epochs to minimize the loss

From Table (6.2), we can see that data that is bounded tends to require fewer epochs for satisfactory training. Furthermore for our linear data, using a smaller batch size tended to yield faster convergence.

Fundamentally both data generation schemes are identical. Both schemes generate the same family of points, but faster training is observed with randomly generated data. The true distinction between the 2 schemes is the order of the points present in mini-batches. For the randomly generated point data, there is no intrinsic ordering to the data. 10000 random data points were linearly mapped to their corresponding 10000 derivative points. For the sequential trajectories, the initial conditions were randomly generated but the trajectories based off those initial conditions were not. When training the data, mini-batches of size 16, 32, and 64 were taken. What this means is that when performing a linear regression task via a machine learning scheme, using unordered data is more efficacious for faster training. There are a few possible explanations to why the unordered data yields faster training:

1. Correlated structures, i.e. ordered trajectory data, may force the ML model to try to enforce those correlated structures to all data, despite those structures being specific only to the points on the trajectory. Uncorrelated structures, i.e. the randomly generated points, have no initial structure so the ML model is free to determine a correlated structure without any initial bias and may do so efficiently on its own.

2. The way the ADAM optimizer works may work better for unordered data over ordered data. The derivatives of data from an ordered trajectory will all have similar numerical values since these points are all close to each other and the derivative mapping is a continuous function. More significant optimization will occur if there is some distinction in the values of the data in a given mini batch.

3. The representation of trajectory data points might require more decimal points to properly represent, and a lack of proper representation leads to an accumulation of truncation and rounding errors. Unordered data doesn't suffer from this issue since an accumulation of truncation or rounding error can't occur with only 1 point.

The above issue is interesting in its own right and definitely warrants further investigation as a potential future project.

Although the problem just discussed is nothing more than linear regression, it is useful to see first hand that a machine learning model will converge to the correct parameter matrix for a judiciously created network and that the ordering of the data for batches impacts the required number of epochs for satisfactory training. Now that we have shown that a linear system can be correctly estimated via machine learning, we go to the task of utilizing machine learning for a nonlinear system.

6.3 Nonlinear System Tests

We now attempt to create an approximation of the dynamics via machine learning with the incorporation of activation functions and hidden layers. We shall test the previously discussed systems (3.6) and (3.8), with their corresponding initial conditions, using different machine learning architectures.

The main distinction in architectures will be the number of hidden layers used in each machine learning model. As usual, we require validation data to ensure that our models are not overtraining and memorizing data. To this end, we shall generate validation data over the same time interval but at a different time step. The validation data may have some data points that are repeated from the training data, but overall the validation data will have significantly more new unseen points.

6.3.1 Ordered vs Unordered Data

In the previous section, when training data for the linear dynamical system was collected it could be collected as either randomly generated points or randomly generated trajectories with their outputs. With regards to nonlinear data, a discernible advantage has been observed in using unordered data points vs sequentially ordered trajectories. A small sample test has been run using the same number of points operating on roughly the same domain [0,1] and range [0,5]. The machine learning model had the structure of (6.6) with matrices P_1 and P_2 of dimensions 108×3 and 3×108 respectively, corresponding bias vectors $\vec{b}_1$ and $\vec{b}_2$ of dimensions 1×108 and 1×3 respectively, and activation function of

Randomly Generated Point vs Trajectory Data				
Type	Batch Size	Min Error	Max Error	Avg Error
unordered	32	0.0002	0.0274	0.0041
ordered	32	0.0108	0.4191	0.1134
unordered	64	0.0015	0.0483	0.0076
ordered	64	0.0182	0.2876	0.0607
unordered	500	0.0007	0.2834	0.0358
ordered	500	0.0006	0.4498	0.0508

Table 6.3: Generally unordered data yields lower errors than ordered data

ReLU (Rectified Linear Unit), defined as

$$\sigma(x) = \begin{cases} 0 & x \leq 0 \\ x & x > 0 \end{cases} \tag{6.5}$$

The results for a small sample test are given in Table (6.3).

Clearly for smaller batch sizes, using unordered data yields lower errors. However, for larger batch sizes the errors are comparable. Despite the apparent advantages to using unordered data, we shall proceed with using ordered trajectory data for the following reasons:

- In practical applications, trajectory data is usually acquired from observations and the associated derivative data is calculated from the trajectory data, e.g finite differences [2].
- The advantage of unordered data may be circumstantial as the dynamical systems tested are autonomous and do not involve strongly nonlinear functions in time or the phase space variables.

The observations regarding ordered and unordered data may warrant further exploration for a future project.

6.3.2 Model Structures

We shall observe fundamentally 3 types of model structures: models that have one, two, or multiple hidden layers.

One Hidden Layer Structure

The model formed here will be a perceptron with 1 hidden layer. Schematically the model will have the following appearance:

$$[x, y, z] \to \sigma_1([x, y, z] \times P_1^T + \vec{b}_1) \to \sigma_1([x, y, z] \times P_1^T + \vec{b}_1) \times P_2^T + \vec{b}_2 \to [dx, dy, dz] \quad (6.6)$$

where P_1 is a matrix of dimensions $n \times 3$, $\vec{b}_1$ a vector of $\mathbb{R}^n$, $\sigma_1 : \mathbb{R}^n \to \mathbb{R}^n$ is a nonlinear activation function, P_2 is a matrix of dimensions $3 \times n$, and $\vec{b}_2$ is a vector of $\mathbb{R}^3$.

Due to the sheer number of possible models that can be generated, we restricted our attention to models that only involve the nonlinear activation function ReLU (Rectified Linear Unit) and we evaluated the validation error after training on 40 epochs.

After some preliminary tests over the time intervals $T = [0, 0.05]$ and $T = [0, 0.1]$ using system (3.6), it was observed that letting $n = 108$ for the dimensions of matrices P_1 and P_2 yields the lowest average validation error.

In the Table (6.4), the entry $3/n - n/3$ in the column "Model Structure" refer to matrix P_1 of dimension $3 \times n$ and matrix P_2 of dimensions $n \times 3$. Clearly from the observed data, using a model structure of $3/108 - 108/3$ yields the lowest error so to summarize the derived results, the model structures of $3/108 - 108/3$ shall be shown for the other time intervals tested.

Error Comparison for Different Model Structures				
Time Interval	Model Structure	Min Error	Max Error	Avg Error
[0,0.05]	3/4-4/3	0.0053	1.1725	0.2441
[0,0.05]	3/12-12/3	0.0067	0.3141	0.0848
[0,0.05]	3/36-36/3	0.0022	0.1840	0.0441
[0,0.05]	3/108-108/3	0.0023	0.0851	0.0251
[0,0.05]	3/324-324/3	0.0071	0.1958	0.0781
[0,0.1]	3/4-4/3	0.0023	1.0238	0.2627
[0,0.1]	3/12-12/3	0.0017	0.8314	0.1248
[0,0.1]	3/36-36/3	0.0031	0.2616	0.0406
[0,0.1]	3/108-108/3	0.0051	0.1734	0.0366
[0,0.1]	3/324-324/3	0.0032	0.1178	0.0371

Table 6.4: From the above, it is clear that certain orientations yield a lower average than others.

Two Hidden Layer Structure

The model formed here will be a perceptron with 2 hidden layers. Schematically the model will be similar to that of (6.6) with the following appearance:

$$\begin{aligned}&[x,y,z] \to \sigma_1([x,y,z] \times P_1^T + \vec{b}_1) \to \sigma_2\big(\sigma_1([x,y,z] \times P_1^T + \vec{b}_1) \times P_2^T + \vec{b}_2\big) \\ &\to \sigma_2\bigg(\sigma_1\big([x,y,z] \times P_1^T + \vec{b}_1\big) \times P_2^T + \vec{b}_2\bigg) \times P_3^T + \vec{b}_3 \to [dx,dy,dz]\end{aligned} \tag{6.7}$$

where P_1 is a matrix of dimensions $n_1 \times 3$, $\vec{b}_1$ a vector of $\mathbb{R}^{n_1}$, $\sigma_1 : \mathbb{R}^{n_1} \to \mathbb{R}^{n_1}$ is a nonlinear activation function, P_2 is a matrix of dimensions $n_2 \times n_1$, and $\vec{b}_2$ is a vector of $\mathbb{R}^{n_2}$, $\sigma_2 : \mathbb{R}^{n_2} \to \mathbb{R}^{n_2}$, P_3 is a matrix of dimensions $3 \times n_2$, and $\vec{b}_3$ is a vector of $\mathbb{R}^3$.

Since the number of possible models that can be generated is even larger than the one hidden layer case, we restricted our attention to a few specially chosen orientations (from extensive testing). Some preliminary testing over the interval $[0, 0.05]$ for system (3.6) was conducted and the data of Table (6.5) was collected.

Error Comparison for Different Time Intervals			
Model Structure	Min Error	Max Error	Avg Error
3/4-4/12-12/3	0.1432	1.4570	0.5644
3/12-12/4-4/3	0.1487	1.4832	0.6552
3/12-12/36-36/3	0.0278	0.4257	0.1482
3/36-36/12-12/3	0.0041	0.2272	0.0780
3/36-36/108-108/3	0.0067	0.3327	0.0828
3/108-108/36-36/3	0.0181	0.2763	0.0753
3/108-108/324-324/3	0.0942	0.5059	0.2855
3/324-324/108-108/3	0.0870	0.3508	0.2051
3/108-108/108-108/3	0.0167	0.2234	0.0625

Table 6.5: From the data, it is clear that using more parameters tends to yield lower errors.

From the testing shown in Table (6.5), it was observed that starting with many parameters and reducing the number of parameters through the model tends to yield lower average errors. After extensive testing, it was found that using 108 parameters was effective in yielding lower errors. From Table (6.5), it appeared that the models with structure 3/108-108/108-108/3 performed the best, so this orientation was used in our testing.

Multiple Hidden Layer Structure

The model formed here will be a perceptron with multiple hidden layers. Schematically the model will be similar to the previous 2 models. Since the model works off of repeated composition and matrix multiplication, the most general model structure of an arbitrary number of m hidden layers can be written as

$$\begin{aligned} [x,y,z] &\to \sigma_1([x,y,z] \times P_1^T + \vec{b}_1) = \vec{v}_{n_1} \\ \vec{v}_{n_{i-1}} &\to \sigma_i(\vec{v}_{n_{i-1}} \times P_i^T + \vec{b}_i) = \vec{v}_{n_i} \quad 2 \leq i \leq m-1 \\ \vec{v}_{n_m} &\to \vec{v}_{n_m} \times P_{m+1}^T + \vec{b}_{m+1} = [dx, dy, dz] \end{aligned} \tag{6.8}$$

where P_1 is a matrix of dimensions $n_1 \times 3$, $\vec{b}_1$ a vector of $\mathbb{R}^{n_1}$, $\sigma_1 : \mathbb{R}^{n_1} \to \mathbb{R}^{n_1}$ is a nonlinear activation function, P_i is a matrix of dimensions $n_i \times n_{i-1}$, and $\vec{b}_i$ is a vector of $\mathbb{R}^{n_i}$, $\sigma_i : \mathbb{R}^{n_i} \to \mathbb{R}^{n_i}$, P_{m+1} is a matrix of dimensions $3 \times n_m$, and $\vec{b}_{m+1}$ is a vector of $\mathbb{R}^3$.

As before, the number of possible models that can be generated is extensive and is limited only by one's imagination. To keep the data brief, several models of differing lengths will be discussed with comparable dimensions for the matrices used in each hidden layer. Similar to before, only ReLU will be used as the activation function. After some preliminary testing on possible model structures, using system (3.6), the results of Table (6.6) were generated.

Error Comparison for Different Model Structures over time interval $[0, 5]$			
Model Structure	Min Error	Max Error	Avg Error
3/80-80/70-70/20-20/3	0.0037	0.3802	0.0146
3/108-108/108-108/108-108/3	0.0001	0.3122	0.0129
3/36-36/81-81/108-108/3	0.0011	0.4022	0.0130
3/108-108/81-81/36-36/3	0.0011	0.4807	0.0186
3/81-81/81-81/81-81/3	0.0014	0.4489	0.0208
3/81-81/108-108/36-36/3	0.0007	0.3866	0.0165

Table 6.6: Errors for all model structures are comparable.

From Table (6.6), it is evident that there is no major difference in the average error nor max error for each model structure. However, it is noted that for a model structure of $3/108-108/108-108/108-108/3$ the max and min errors were both minimal. Therefore this model structure shall be used as a template moving forward as it appears to offer some advantage. Also for faster training only 20 epochs were used, since the use of multiple hidden layers yields faster initial loss decrease. Next we needed to consider the number of hidden layers to use. From the data in Table (6.6) and data from time interval $[0, 5]$, the number of hidden layers was altered and compared. Table (6.7) shows the error comparisons.

Clearly from Table (6.7), using more hidden layers doesn't confer any noticeable ad-

Error Comparison for Different Model Structures over time interval $[0, 5]$			
Model Structure	Min Error	Max Error	Avg Error
3/108-108/108-108/108-108/108-108/3	0.0049	0.4447	0.0310
3/108-108/108-108/108-108/108-108/108-108/3	0.0013	0.4722	0.0229
3/108-108/108-108/108-108/108-108/108-108/108-108/3	0.0008	0.7351	0.0229

Table 6.7: The errors for all models are comparable. This indicates that using more layers doesn't automatically imply improved performance.

vantage in error reduction. Furthermore, training models with additional layers requires longer training times. Therefore we limited the number of hidden layers in our models to three.

6.3.3 System (3.6)

In Table (6.8), the time intervals may be different but the time step used for each test was the same 0.001. To ensure fairness in testing, the number of initial conditions (and in conjunction trajectories) was adjusted for each time interval so the total number of data points, 100000, used for each case was the same. For the interval $[0, 0.5]$ 2000 initial conditions were used, for $[0, 1]$ 1000 initial conditions, and for $[0, 5]$ 20 initial conditions. From the table, we can see that the average error was roughly the same magnitude for each time interval tested with comparable min and max errors. The max errors may be different but were roughly on the same order of magnitude. To further elucidate the results, some sample derivative trajectories from each system are plotted as well in Figures (6.1) and (6.2). From these figures a few items of interest can be noted. First, the machine learning approximated functions are not guaranteed to be smooth. Noted in both figures, several curves from machine learning models exhibit corners. This particular behavior may be a result of using the ReLU activation function in each of the models. Second, these functions have the capacity to accurately approximate the original function as

seen with machine learning trajectories that nearly overlap the original trajectory, or are sufficiently close in coverage. Conversely, these functions may offer inaccurate predictions as seen with the last graph in Figure (6.2), where the one hidden layer model predicts a completely different trajectory than that expected. Similarly, the two and multiple hidden layer models also suffer from the same inability to accurately predict the true derivative trajectory, but to a less severe degree.

Error Comparison for Different Time Intervals and Model Structures - System (3.6)				
Time Interval	Model Structure	Min Error	Max Error	Avg Error
[0,0.05]	3/108-108/3	1.6618e-03	1.4064e-01	1.8990e-02
[0,0.05]	3/108-108/108-108/3	3.3701e-04	1.8499e-01	2.9528e-02
[0,0.05]	3/108-108/108-108/108-108/3	1.0012e-03	1.4359e-01	2.7069e-02
[0,0.1]	3/108-108/3	3.5771e-04	1.4197e-01	1.9836e-02
[0,0.1]	3/108-108/108-108/3	1.3430e-03	3.1058e-01	3.6738e-02
[0,0.1]	3/108-108/108-108/108-108/3	6.9223e-04	2.0114e-01	3.5606e-02
[0,0.5]	3/108-108/3	2.7378e-03	3.7421e-01	3.0936e-02
[0,0.5]	3/108-108/108-108/3	8.0196e-04	3.4968e-01	2.9785e-02
[0,0.5]	3/108-108/108-108/108-108/3	2.8385e-02	5.1729e-01	1.7679e-01
[0,1]	3/108-108/3	4.5775e-03	4.3106e-01	5.7347e-02
[0,1]	3/108-108/108-108/3	2.0558e-03	1.4166e-01	5.1877e-02
[0,1]	3/108-108/108-108/108-108/3	1.5813e-03	1.4832e-01	3.5908e-02
[0,5]	3/108-108/3	1.0278e-02	1.8477e+00	7.4068e-02
[0,5]	3/108-108/108-108/3	1.3500e-02	1.4278e+00	5.3442e-02
[0,5]	3/108-108/108-108/108-108/3	6.9037e-03	1.6664e+00	5.4890e-02

Table 6.8: For the most part, the errors for all model structures for data over the same time interval are roughly the identical.

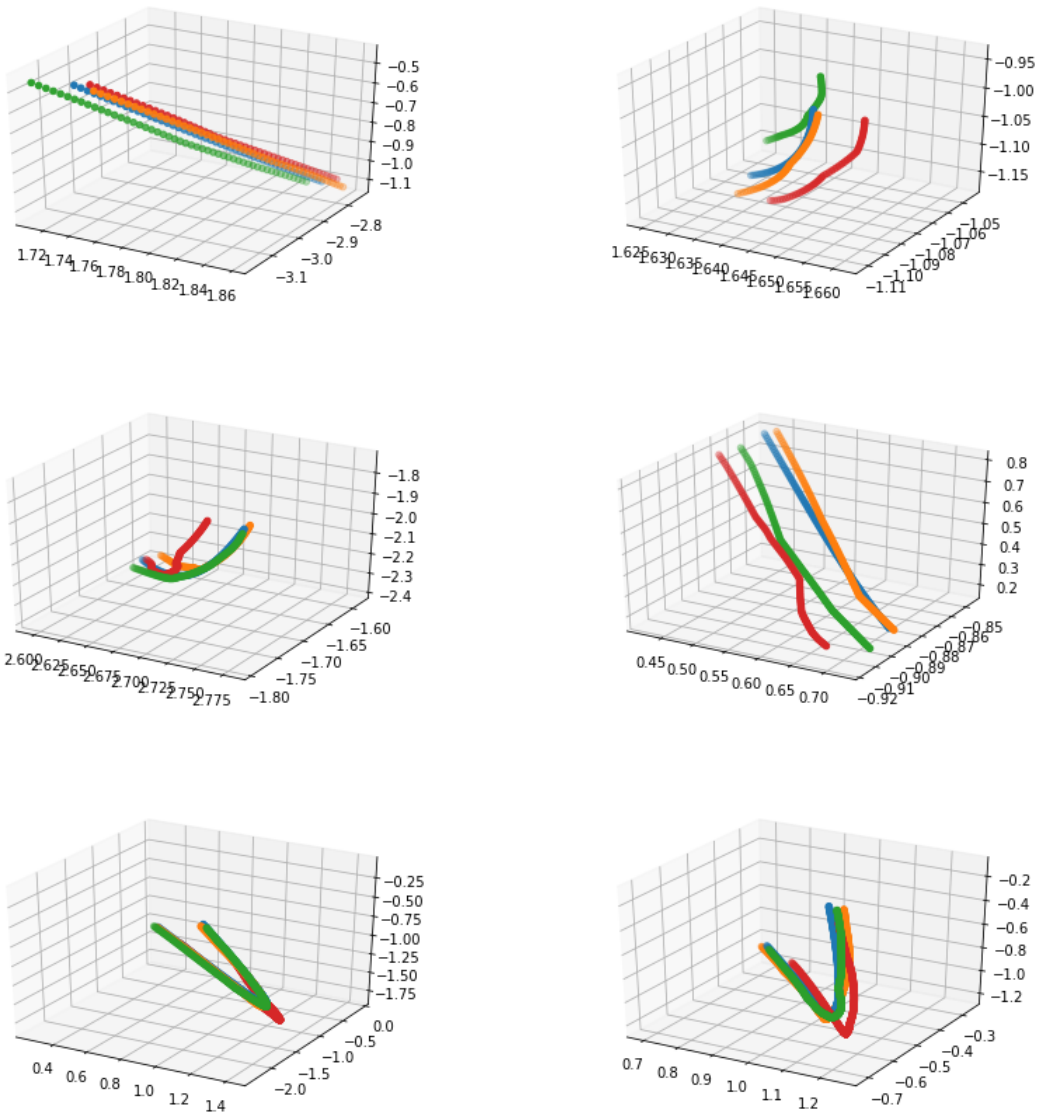

Figure 6.1: The blue curve represents the original derivative trajectory, the orange curve the one hidden layer model trajectory, the green curve the two hidden layer model trajectory, and the red curve the multiple hidden layer model trajectory. The first row represents sample trajectories from $T = [0, 0.05]$, the second $T = [0, 0.1]$, the third $T = [0, 0.5]$.

Clearly, the machine learning model was fairly successful in approximating the dynamical system. For the model trained over the time interval $T = [0, 5]$, training over more epochs and using more data will ensure that the derivative approximation more closely resembles the actual derivative values. As mentioned earlier, one noticeable observation is that the machine learning model yields prediction curves that are less smooth in comparison to the actual derivatives, possibly indicating that the machine learning model is a

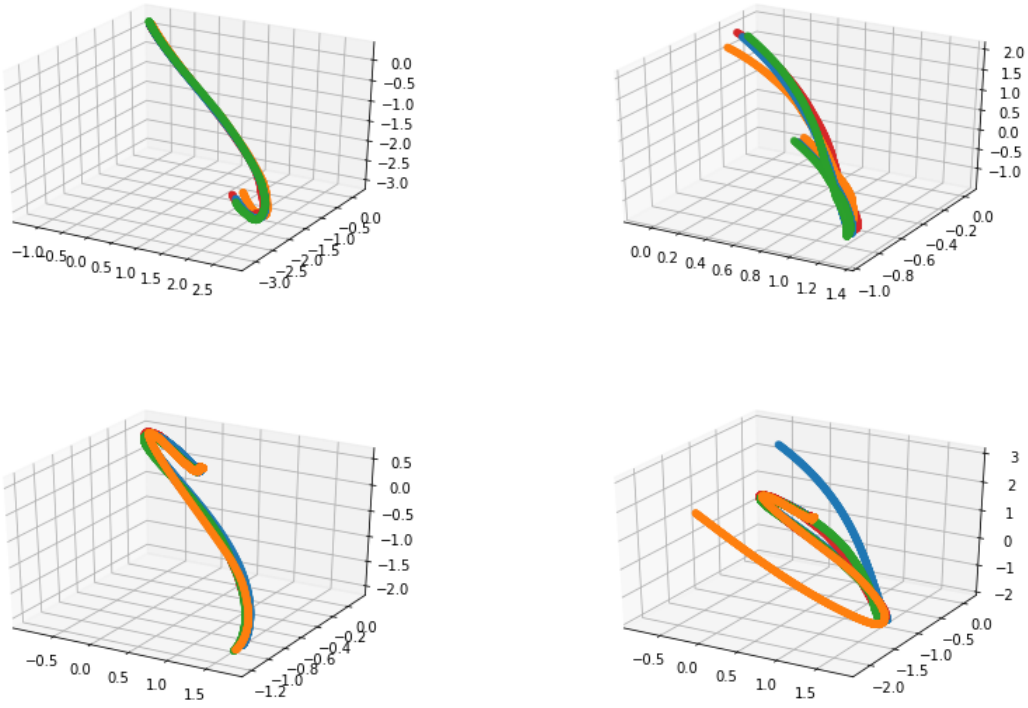

Figure 6.2: The blue curve represents the original derivative trajectory, the orange curve the one hidden layer model trajectory, the green curve the two hidden layer model trajectory, and the red curve the multiple hidden layer model trajectory. The first row represents sample trajectories from $T = [0, 1]$ and the second $T = [0, 5]$.

sort of piecewise continuous function. In comparison to the SVD approach, the machine learning approach provided more accurate predictions for system (3.6).

6.3.4 System (3.8)

Table (6.9) contains the error results from using all model structures over all time intervals for the Lorenz system. Figures (6.3) and (6.4) show more explicitly the graphs of the derivative trajectories applied to validation data. From Table (6.9), it is clear that for larger time windows, e.g. $T = [0, 1]$ or $T = [0, 5]$ using more hidden layers tends to produce a more accurate approximation. Although the graphs of Figure (6.4) show little to no distinction between the the actual and approximated derivative trajectories, the maximum error of the one hidden layer model is nearly 30, 3 times that of the two hidden layer and multiple hidden layer models. From both figures, one can hypothesize that using additional hidden layers works well for highly nonlinear data. On the other hand,

using fewer hidden layers works better for nonlinear data that can be accurately linearized over some small time interval, while using many hidden layers over the same small time interval is less effective to acquiring smaller error.

Error Comparison for Different Time Intervals System (3.8)				
Time Interval	Model Structure	Min Error	Max Error	Avg Error
[0,0.05]	3/108-108/3	6.8786e-04	6.1174e-01	2.7316e-02
[0,0.05]	3/108-108/108-108/3	6.0730e-04	7.0845e-01	2.4936e-02
[0,0.05]	3/108-108/108-108/108-108/3	1.0024e-03	6.0059e-01	4.3093e-02
[0,0.1]	3/108-108/3	5.9290e-03	2.8324e-01	6.1054e-02
[0,0.1]	3/108-108/108-108/3	4.6000e-03	5.9648e-01	1.7455e-01
[0,0.1]	3/108-108/108-108/108-108/3	2.7247e-02	2.7493e+00	9.4708e-01
[0,0.5]	3/108-108/3	9.9410e-02	4.8581e+01	3.7499e+00
[0,0.5]	3/108-108/108-108/3	8.4056e-03	7.4880e+00	1.5430e+00
[0,0.5]	3/108-108/108-108/108-108/3	9.8506e-03	8.5082e+00	1.2265e+00
[0,1]	3/108-108/3	2.0307e-01	3.5834e+01	6.5673e+00
[0,1]	3/108-108/108-108/3	1.5797e-02	9.9434e+00	2.4526e+00
[0,1]	3/108-108/108-108/108-108/3	6.9705e-03	1.0636e+01	2.5399e+00
[0,5]	3/108-108/3	1.1902e-01	2.7290e+01	1.0883e+01
[0,5]	3/108-108/108-108/3	2.6615e-02	8.3476e+00	1.4967e+00
[0,5]	3/108-108/108-108/108-108/3	1.6468e-02	8.9901e+00	1.0596e+00

Table 6.9: From the table above, it is clear that for nearly linear or weakly nonlinear data, using a model structure with fewer hidden layers is a method to achieve an accurate prediction for

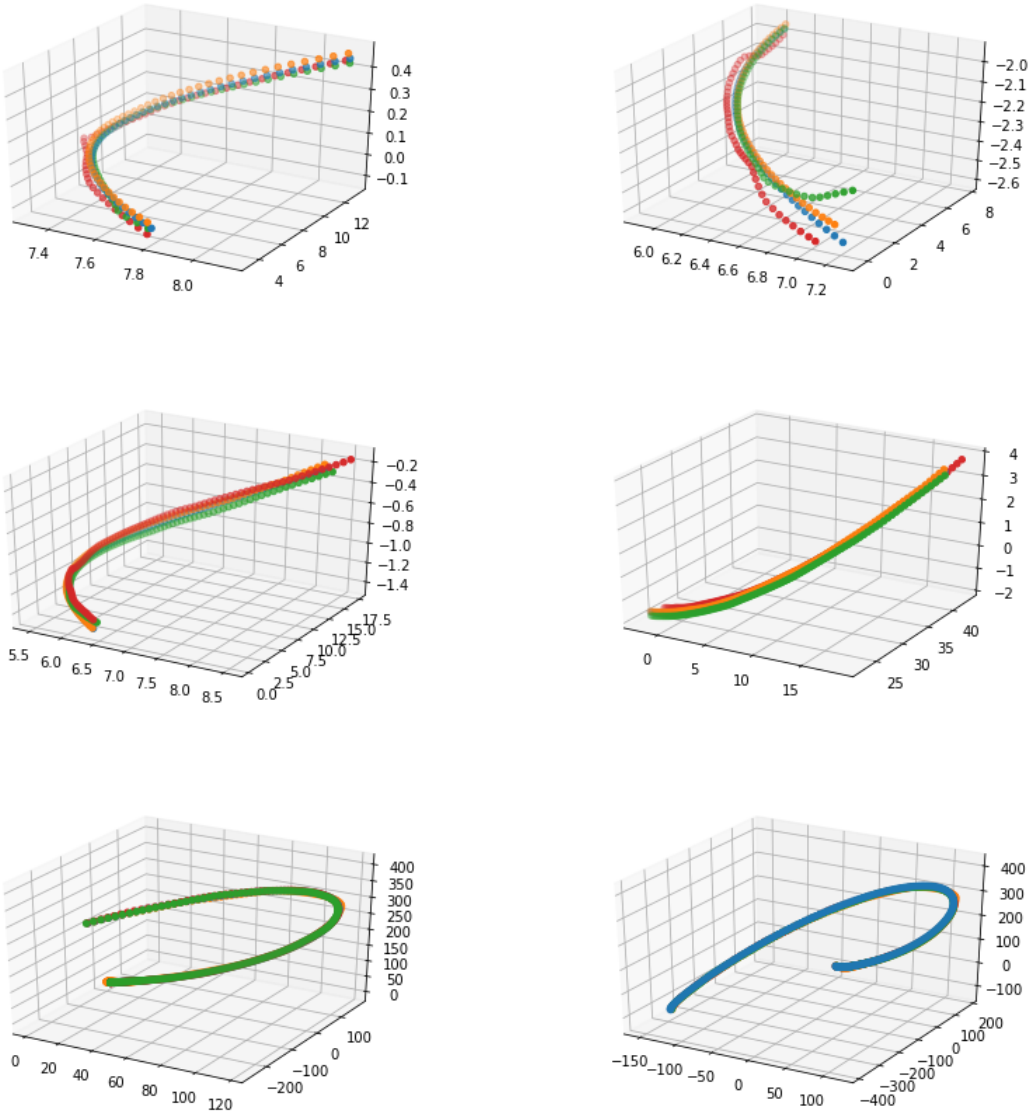

Figure 6.3: The blue curve represents the original derivative trajectory, the orange curve the one hidden layer model trajectory, the green curve the two hidden layer model trajectory, and the red curve the multiple hidden layer model trajectory. The first row represents sample trajectories from $T = [0, 0.05]$, the second $T = [0, 0.1]$, the third $T = [0, 0.5]$.

Despite the larger errors that are observed for larger time intervals, the error observed through DNN regression is significantly smaller in magnitude than that observed from the SVD approach.

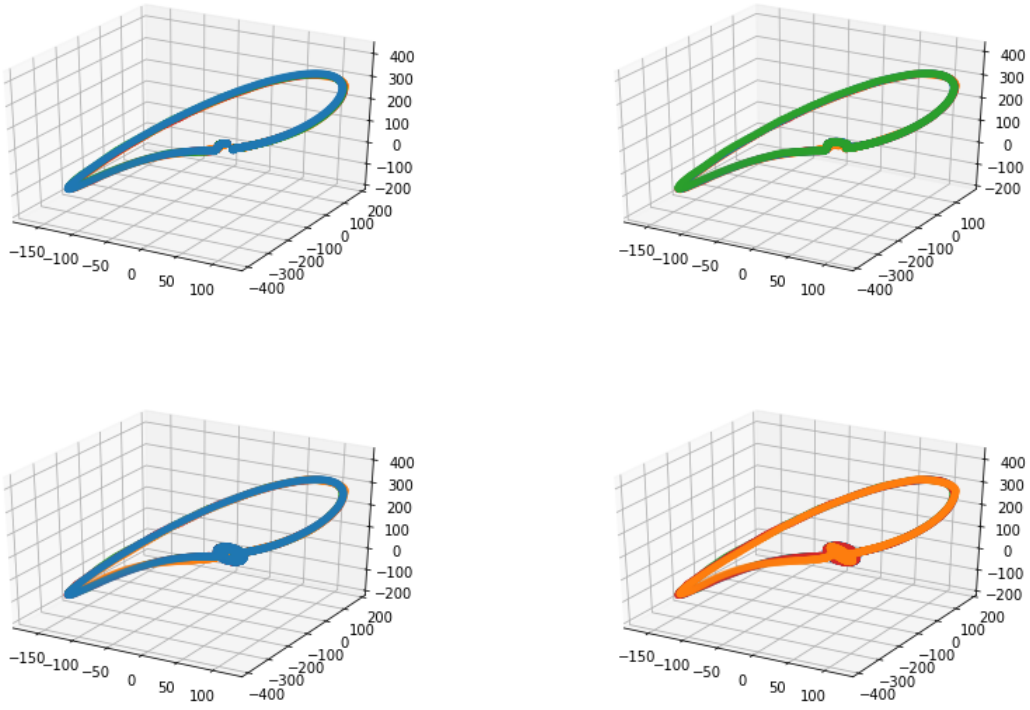

Figure 6.4: The blue curve represents the original derivative trajectory, the orange curve the one hidden layer model trajectory, the green curve the two hidden layer model trajectory, and the red curve the multiple hidden layer model trajectory. The first row represents sample trajectories from $T = [0, 1]$ and the second $T = [0, 5]$.

6.4 Observations over DNN Regression

Utilizing DNN to determine the dynamics of various autonomous dynamical systems devolves to a simple regression problem. It has been noted that for the problem of regression, using unordered randomly generated data points is more effective, in terms of faster loss reduction within fewer epochs, than using trajectory data. There are a few possible hypotheses for this behavio: the machine learning program might be able to more efficiently determine correlations by itself rather than use imposed ones, the ADAM optimizer might have more efficient training (significant and faster loss reduction) when the mini-batch data has some diversity, accumulation of rounding and truncation errors might be more prevalent in the ordered data vs the unordered data. Another observation to note is that for linear systems, the machine learning model will converge to the analytic solution. This gives some meaningful assurance to the accuracy of machine learning optimization

without the need for validation data. The last observation is that for bounded nonlinear systems, machine learning regression will converge to an approximation of the nonlinear system. It should be noted that the trait of being bounded is more important than being chaotic, as the machine learning models were able to approximate the dynamics of the the Lorenz system with a reasonable amount of accuracy. The only downside to the machine learning approach is the fact that the internal workings are mostly unknowable. Attempting to write out the analytical expression of a machine learning model is a foolhardy task as the composition of matrices and activation functions will yield an expression that lacks much intuition. For small models with only a few parameters, it might be possible to obtain a meaningful analytic expression, but for models that have thousands to millions of parameters such an expression is nearly meaningless. This suggests that a way forward with regards to DNN research is parameter reduction for interpretability as well as more efficient training.

Chapter 7

Conclusion

We have observed many different techniques to determine information regarding a dynamical system, ranging from the form of the dynamics to the forecasting of trajectories. DMD is effectively a linear method that is valid only over sufficiently small time intervals where linearization is possible; furthermore the assumption that the dynamics of a system change linearly from one observation to the next is a strong condition that isn't usually upheld. SINDy is powerful but is limited in effectiveness due to the requirement of choosing a family of basis functions, the number of which is limited only by one's imagination, combined with the need to pick the correct family of basis functions. In addition, to use SINDy effectively one must use extremely high quality data and take measurements of relevant quantities. SVD is a valid approach provided the system in question is linear or the time window of observation is sufficiently small for a nonlinear system. DNN regression provides a promising method to determining the dynamics of collected data from a system, but the black box nature of a machine learning model make it slightly unappealing to determine the underlying dynamics, although the point dynamics can be interpolated with a high degree of accuracy. To fully take advantage of the machine learning approach, one needs to understand the dynamics of machine learning itself. Until this is done, the black box nature of machine learning will provide excellent results without much consideration to why those results work.

Bibliography

[1] bepresent. (2018, March 15). Non-linear regression, which network architecture? Deep Learning Course Forums. https://forums.fast.ai/t/non-linear-regression-which-network-architecture/13304. Accessed 16 March 2022

[2] Brunton, Steven L., and J.Nathan Kutz. Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control. Cambridge, MA: Cambridge University Press, 2019.

[3] Chen, X. (2021, October 10). Dynamic mode decomposition for multivariate time series forecasting. Medium. Towards Data Science. https://towardsdatascience.com/dynamic-mode-decomposition-for-multivariate-time-series-forecasting-415d30086b4b. Accessed 2 April 2022

[4] Harris, C.R., Millman, K.J., van der Walt, S.J. et al. Array programming with NumPy. Nature 585, 357–362 (2020). DOI: 10.1038/s41586-020-2649-2

[5] Kingma, Diederik P., and Jimmy Ba. "Adam: A method for stochastic optimization." arXiv preprint arXiv:1412.6980 (2014).

[6] Olver, Peter J, and Chehrzad Shakiban. Applied Linear Algebra. Upper Saddle River, NJ: Pearson/Prentice Hall, 2013.

[7] Paszke, Adam, et al. "Pytorch: An imperative style, high-performance deep learning library." Advances in neural information processing systems 32 (2019).

[8] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, et al. (2020) SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. Nature Methods, 17(3), 261-272.

[9] P. J. Schmid and J. Sesterhenn. Dynamic mode decomposition of numerical and experimental data. In 61st Annual Meeting of the APS Division of Fluid Dynamics. American Physical Society, November 2008

[10] Strogatz, Steven H. Nonlinear Dynamics and Chaos: With Applications to Physics, Biology, Chemistry and Engineering. Boulder, CO: Westview Press, 2015.

[11] Van Rossum, Guido, Fred L. Drake Jr. Python 3 Reference Manual. Createspace, 2009. Print.

Appendix

Python Programs for the Numerical Tests

All of this code is written in Python 3 [11] and utilizes the program packages of Numpy [4], Scipy, [8], PyTorch[7], and Adam Optimizer [5]. Some code is inspired by posts from the following sources: [1], [2], [3].

Dependencies

```
import numpy as np
from scipy import integrate
import torch
from torch import nn
import torch.utils.data as utils_data
from torch.autograd import Variable
from torch.utils.data import DataLoader
from torch import nn, optim
from mpl_toolkits import mplot3d
%matplotlib inline
import matplotlib.pyplot as plt
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print(f'Using {device} device')
```

Listing 7.1: Numpy, Scipy, Pytorch, and MatplotLib dependencies are needed. A GPU is preferred when training, but is not necessary to run the code in this thesis.

Ch. 2.1.2 Trajectory and Data Generation for Linear Case

```
# Initialization of Data
dt = 0.001
T = 5
t = np.arange(dt,T+dt,dt) # vector of times [0.01, 0.02, ... 5]
numint = int(T/dt) # number of steps to integrate over
t0 = 0
def linear_deriv(x_y_z , t0):# returns derivative for x, y, z input
```

```
    x, y, z = x_y_z
    return [-23/5*x - 6*y - 24/5*z, 2*x + y + 2*z, 6/5*x + 2*y + 3/5*z]
# actual test data
size = int(numint)
data = np.zeros([size,3])
x00 = (-19, -9, 17) # randomly chosen initial condition
data = integrate.odeint(linear_deriv, x00, t,rtol=10**(-12),atol
   =10**(-12)*np.ones_like(x00))
# plot the data
fig = plt.figure()
cx = plt.axes(projection='3d')
cx.scatter3D(data[:,0], data[:,1], data[:,2], cmap='Greens');
```

Listing 7.2: This code block can be modified to generate data for any 3 dimensional system by changing the derivative function.

3.1.1 SVD Applied to Linear Test Case

```
dt = 0.01 # time step
T = 1 # max time
t = np.arange(dt,T+dt,dt) # vector of times [0.01, 0.02, ... 5]
x0 = (-8,8,27)
t0 = 0
numint = int(T/dt)
def linear_deriv(x_y_z, t0): # returns derivative for  x, y, z input
    x, y, z = x_y_z
    return [-12*x + 12*y -3* z, -4*x -2*y -z, -2*x +4*y+3*z]
x = integrate.odeint(linear_deriv, x0, t,rtol=10**(-12),atol=10**(-12)*
   np.ones_like(x0))
dx = np.zeros_like(x)
for j in range(len(t)):
    dx[j,:] = linear_deriv(x[j,:],0)
U, S, Vt = scipy.linalg.svd(x)
Sc = np.diag(S) # create diagonal S matrix to take inverse of
r = 3 # rank of data matrix x
Uc = U[:,0:r] # rank 3 approximation
```

```
S_inv = np.diag(1/S)
parameters4 = np.transpose(np.linalg.multi_dot([np.transpose(Vt),S_inv,
    np.transpose(Uc),dx]))# parameters4 represents the SVD approximation
fig = plt.figure()
cx = plt.axes(projection='3d')
cx.scatter3D(x[:,0], x[:,1], x[:,2], cmap='Greens');
fig = plt.figure()
cx = plt.axes(projection='3d')
cx.scatter3D(dx[:,0], dx[:,1], dx[:,2], cmap='Greens');
```

Listing 7.3: The code of Listing (7.2) is applied here with some extensions.

3.2.1 SVD Applied to Weakly Nonlinear Test Case

```
# Parameters before collecting data
dt = 0.001 # 0.001, 0.1
T = 0.25 # 1, 0.5 0.25
t = np.arange(dt,T+dt,dt) # vector of times [0.01, 0.02, ... 5]
alpha = -1
beta =  -3
gamma = -4
delta = -2
x00 =  (-1,1,1)
t0 = 0
def Nlinear_deriv1(x_y_z, t0, alpha = alpha, beta = beta, gamma = gamma
    , delta = delta): # returns derivative for x, y, z input
     x, y, z = x_y_z
     return [alpha * (y - x)-beta*z, x * y* gamma  - z, delta*x - gamma*
    y + beta * z]
# Trajectory generation
xnl1 = integrate.odeint(Nlinear_deriv1, x00, t,rtol=10**(-12),atol
    =10**(-12)*np.ones_like(x00))
# Compute Derivative
dxnl1 = np.zeros_like(xnl1)
for j in range(len(t)):
     dxnl1[j,:] = Nlinear_deriv1(xnl1[j,:],0,alpha,beta,gamma, delta)
```

```
# SVD
Unl1, Snl1, Vnl1t = scipy.linalg.svd(xnl1)
Scnl1 = np.diag(Snl1) # create diagonal S matrix to take inverse of
r = 3
Ucnl1 = Unl1[:,0:r] # U cut
S_invnl1 = np.diag(1/Snl1)
solnl1 = np.transpose(np.linalg.multi_dot([np.transpose(Vnl1t),S_invnl1
    ,np.transpose(Ucnl1),dxnl1])) # solution nonlinear system 1
Numpoints = int(T*(1/dt))
errlst = []
errsum = 0
for j in range(len(xnl1)):
    diff = np.matmul(solnl1,xnl1[j]) - dxnl1[j]
    diffsq = scipy.linalg.norm(diff)
    errlst.append(diffsq)
    errsum = errsum + diffsq
#print('|'+ str(min(errlst))+ ' | ' + str(max(errlst)) + ' | ' + str(
    errsum/Numpoints) + '|')
# Linear Approximation to the nonlinear system
a = solnl1[1][0]
b = solnl1[1][1]
c = solnl1[1][2]
def Nlinear_deriv1app(x_y_z, t0): # returns derivative values for given
    x, y, z input
    x, y, z = x_y_z
    return [x - y + 3*z, a*x +b*y + c*z, -2*x+4*y-3*z ]
# Trajectory generation
xnl1app = integrate.odeint(Nlinear_deriv1app, x00, t,rtol=10**(-12),
    atol=10**(-12)*np.ones_like(x00))
# Compute Derivative
dxnl1app = np.zeros_like(xnl1app)
for j in range(len(t)):
    dxnl1app[j,:] = Nlinear_deriv1app(xnl1app[j,:],0)
# Plot the trajectories
fig = plt.figure()
```

```
cx = plt.axes(projection='3d')
cx.scatter3D(xnl1[:,0], xnl1[:,1], xnl1[:,2], cmap='Greens');
cx.scatter3D(xnl1app[:,0], xnl1app[:,1], xnl1app[:,2], cmap='Greens');
fig = plt.figure()
cx = plt.axes(projection='3d')
cx.scatter3D(dxnl1[:,0], dxnl1[:,1], dxnl1[:,2], cmap='Greens');
cx.scatter3D(dxnl1app[:,0],dxnl1app[:,1],dxnl1app[:,2],cmap='Greens');
```

Listing 7.4: The code of Listing (7.3) is applied here with several modifications; most importantly, the linear approximation is turned into a function to determine the forecasting capacity of the SVD approximation. Then this approximation is compared to the actual data.

3.2.2 SVD Applied to Lorenz System

```
dt = 0.001 # 0.1, 0.01
T = 1 # 0.05, 0.1, 0.5, 1, 5
beta = 8/3 # originally 8/3
sigma = 10 # originally 10
rho = 28 # originally 28
x0 = (-8,8,27)
# Functions for data creation
def lorenz_deriv(x_y_z, t0, sigma=sigma, beta=beta, rho=rho): # returns
    derivative values for given x, y, z input
    x, y, z = x_y_z
    return [sigma * (y - x), x * (rho - z) - y, x * y - beta * z]
x = integrate.odeint(lorenz_deriv, x0, t,rtol=10**(-12),atol=10**(-12)*
   np.ones_like(x0))
```

Listing 7.5: Code is mostly the same as that in Listing (7.4), with differences in the time intervals, parameter values, initial condition, derivative function, and function argument for the integrate function.

4.4.1 DMD Interpolation and Extrapolation Linear Example

```
dataT = np.transpose(data)
p = 0.001 # 0.8, 0.5, 0.1, 0.01 # proportion for use in DMD algorithm
train = int(p * numint) # number of data used for DMD
test = int((1-p)* numint) # number of data used to test forecasting
    ability
dataTinit = dataT[:,:train]
dataTfin = dataT[:,train:]
def DMD(data, r): # data is input in n x m form, r = rank
    ind = data[:,:-1] # in data
    oud = data[:,1:] # out data
    U, S, Vt = scipy.linalg.svd(ind) # svd decomposition
    Smat = np.diag(S) # turn the tuple S into a diagonal matrix
    S_inv = np.diag(1/S)
    V = np.transpose(Vt)
    Vc = V[:,:r]
    Uc = U[:r,:]
    Ut = np.transpose(Uc)
    Atild = np.linalg.multi_dot([Ut.conj(),oud, Vc.conj(), S_inv]) #
    determine reduced order A
    eigval, eigvec = scipy.linalg.eig(Atild) # extract eigenstuff
    eigvec and eigval are for Atild
    phi = np.linalg.multi_dot([oud, Vc.conj(), S_inv, eigvec]) #
    calculate eigenvectors of A
    A = phi @ np.diag(eigval) @ np.linalg.pinv(phi) # matrix A
    determined although it is possible to proceed with just phi and
    eigval
    return eigval, phi, A
eigval, phi, A = DMD(dataTinit,3)
dmdforecast = np.zeros((3,test-1)) # dmd forecasting
dmdinter = np.zeros((3,train-1)) # dmd interpolation
b = np.linalg.pinv(phi)@dataTinit[:,0]
for i in range(1, train-1):
    dmdinter[:,i] = np.real(np.linalg.multi_dot([phi,(np.diag(eigval))
    **i, b])) # imaginary parts are insignificant
b2 = np.linalg.pinv(phi) @ dataTinit[:,-1]
```

```
for i in range(1, test-1):
    dmdforecast[:,i] = np.real(np.linalg.multi_dot([phi,(np.diag(eigval))**i, b2])) # imaginary parts are insignificant
complen1 = train-1
errvec1 = np.zeros(complen1)
for i in range(1,complen1): # for entire interval len(errvec) = numint-1
    errvec1[i] = np.linalg.norm(dmdinter[:,i] - dataT[:,i])
# Calculation of Interpolation Error
min1 = '{:0.4e}'.format(min(errvec1))
max1 = '{:0.4e}'.format(max(errvec1))
avg1 = '{:0.4e}'.format(sum(errvec1)/len(errvec1))
complen2 = test-1
errvec2 = np.zeros(complen2)
for j in range(1, complen2):
    errvec2[j] = np.linalg.norm(dmdforecast[:,j] - dataT[:,j+complen1])
# Calculation of Forecasting Error
min2 = '{:0.4e}'.format(min(errvec2))
max2 = '{:0.4e}'.format(max(errvec2))
avg2 = '{:0.4e}'.format(sum(errvec2)/len(errvec2))
# print('|'+str(p)+'|Inter|'+min1+'|'+max1+'|'+avg1+'|')
# print('|'+str(round(1-p,2))+'|Extra|'+min2+'|'+max2+'|'+avg2+'|')
# Plot the Original Data
fig = plt.figure()
cx = plt.axes(projection='3d')
cx.scatter3D(data[:,0], data[:,1], data[:,2], cmap='Greens');
# Plot the Interpolation and Forecasting Predictions Together
fig = plt.figure()
bx = plt.axes(projection='3d')
bx.scatter3D(dataT[0,:train], dataT[1,:train],dataT[2,:train],cmap = 'Blues')
bx.scatter3D(dmdforecast[0,1:test],dmdforecast[1,1:test],dmdforecast[2,1:test], cmap = 'Blues')
```

Listing 7.6: This block defines the DMD algorithm, applies it, and plots the interpolation and forecasting capacity to the linear data generated in the previous code block.

4.4.2 DMD Interpolation and Extrapolation Nonlinear Example

```
def Nlinear_deriv1(x_y_z, t0): # returns derivative for given x, y, z
    x, y, z = x_y_z
    return [-x+y+3*z, -4*x*y - z, -2*x + 4*y - 3*z] # weakly nonlinear
   system
x00 = (85, 26, 31)
data = integrate.odeint(Nlinear_deriv1, x00, t,rtol=10**(-12),atol
   =10**(-12)*np.ones_like(x00))
```

Listing 7.7: The codes of Listings (7.2) and (7.6) are applied here with the only differences of changing the initial condition, the definition of the derivative function, and changing the function input in the integrate function to this new function.

6.1 + 6.1.1 Linear System and Data Generation

```
# Parameters before collecting data- Simple Linear System Test
dt = 0.001 # 0.001 works
T = 1 # 0.5 works
numint = T/dt
t = np.arange(dt,T+dt,dt) # vector of times [0.01, 0.02, ... 5]
alpha = 4
beta = 3
gamma = -4
delta = -2
numint = int(T/dt)
x0 = (-8,8,10)
t0 = 0
def linear_deriv(x_y_z, t0, alpha = alpha, beta = beta, gamma = gamma,
   delta = delta): # returns derivative values for given x, y, z input
    x, y, z = x_y_z
    return [alpha * (y - x)-beta*z, x *gamma  - z +delta*y, delta*x -
   gamma*y + beta * z]
#Training Data generated by randomly selected initial conditions in
   cube [0,1] x [0,1] x [0,1]
numpoints = 10000
```

```
x_train = np.random.rand(numpoints,3)
x_traintens = torch.tensor(x_train, device = device, dtype = torch.
    float32)
y_train = np.zeros_like(x_train)
for j in range(numpoints):
    y_train[j,:] = linear_deriv(x_train[j,:],0,alpha,beta,gamma, delta)
y_traintens = torch.tensor(y_train, device = device, dtype = torch.
    float32)
batchsize = 16
training_samples = utils_data.TensorDataset(x_traintens, y_traintens)
data_loader = utils_data.DataLoader(training_samples, batch_size=
    batchsize, shuffle=False)
# Trajectory generation training data
inicond = 100
initialconditions = np.random.rand(inicond,3)
size = int(inicond*numint)
data = np.zeros([size,3])
for i in range(inicond):
    data[numint*i:numint*(i+1)] = integrate.odeint(linear_deriv,
    initialconditions[i], t,rtol=10**(-12),atol=10**(-12)*np.ones_like(
    initialconditions[i]))
trainsize = int(1*size)
testsize = int(0*size)
datatens = torch.tensor(data, device = device, dtype = torch.float32)
x_traintens, x_testtens = torch.split(datatens,[trainsize, testsize])
outdata = np.zeros_like(data)
for i in range(size):
    outdata[i,:] =  linear_deriv(data[i,:],0,alpha,beta,gamma, delta)
outdatatens = torch.tensor(outdata, device = device, dtype = torch.
    float32)
y_traintens, y_testtens = torch.split(outdatatens,[trainsize, testsize
    ])
batchsize = numint
training_samples = utils_data.TensorDataset(x_traintens, y_traintens)
data_loader = utils_data.DataLoader(training_samples, batch_size=
```

```
    batchsize, shuffle=False)
model0 = torch.nn.Sequential(torch.nn.Linear(3, 3)).to(device)
print(list(model0.parameters()))
criterion = nn.MSELoss()
optimizer = optim.Adam(model0.parameters())
model0.train()
num_epochs = 40
loss_list = []
for epoch in range(num_epochs):
    for batch_idx, (data, target) in enumerate(data_loader):
        data, target = Variable(data), Variable(target)
        optimizer.zero_grad()
        output = model0(data.float())
        loss = criterion(output, target.float())
        loss.backward()
        optimizer.step()
        if epoch >2:
            if batch_idx % 200 == 0:
                loss_list.append(loss.item())
        if batch_idx % 400 == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.
    format(
            epoch, batch_idx * len(data), len(data_loader.dataset),
            100. * batch_idx / len(data_loader), loss.item()))
print(list(model0.parameters()))
```

Listing 7.8: This code block contains code for simple machine learning scheme with no hidden layers as well as data generation schemes.

6.2.3 One/Two/Multiple Hidden Layer Schemes for Weakly Nonlinear System

```
# Parameters before collecting data
dt = 0.001
T = 5 # 0.05, 0.1, 0.5, 1, 5
t = np.arange(dt,T+dt,dt) # vector of times [0.01, 0.02, ... 5]
alpha = -1
```

```
beta = -3
gamma = -4
delta = -2
numint = int(T/dt)
t0 = 0
# Functions for data creation (system is autonomous system of odes)
def Nlinear_deriv1(x_y_z, t0, alpha = alpha, beta = beta, gamma = gamma
    , delta = delta): # returns derivative values for given x, y, z
    input
     x, y, z = x_y_z
     return [alpha * (y - x)-beta*z, x * y* gamma  - z, delta*x - gamma*
    y + beta * z]
# Trajectory generation training data
inicond = 20 # 2000, 1000, 200, 100, 20
initialconditions = np.random.rand(inicond,3)
size = int(inicond*numint)
data = np.zeros([size,3])
for i in range(inicond):
     data[numint*i:numint*(i+1)] = integrate.odeint(Nlinear_deriv1,
    initialconditions[i], t,rtol=10**(-12),atol=10**(-12)*np.ones_like(
    initialconditions[i]))
trainsize = int(0.8*size)
testsize = int(0.2*size)
datatens = torch.tensor(data, device = device, dtype = torch.float32)
x_traintens, x_testtens = torch.split(datatens,[trainsize, testsize])
outdata = np.zeros_like(data)
for i in range(size):
     outdata[i,:] =  Nlinear_deriv1(data[i,:],0,alpha,beta,gamma, delta)
outdatatens = torch.tensor(outdata, device = device, dtype = torch.
    float32)
y_traintens, y_testtens = torch.split(outdatatens,[trainsize, testsize
    ])
batchsize = numint
training_samples = utils_data.TensorDataset(x_traintens, y_traintens)
data_loader = utils_data.DataLoader(training_samples, batch_size=
```

```
    batchsize, shuffle=False)
# Machine Learning Models
# 1 layer- 4, 12, 36, 108, 324
h1 = 108
plst1 = [h1]
model11 = torch.nn.Sequential(
    torch.nn.Linear(3, h1),
    torch.nn.ReLU(),
    torch.nn.Linear(h1, 3),
    ).to(device)
# 2 layers- 36,12  108,36  108,108
l1, l2 = 108,108
plst2 = [l1,l2]
model12 = torch.nn.Sequential(
    torch.nn.Linear(3, l1),
    torch.nn.ReLU(),
    torch.nn.Linear(l1, l2),
    torch.nn.ReLU(),
    torch.nn.Linear(l2, 3),
    ).to(device)
# multiple Layers
#3:  36,81,108  108, 81, 36  81, 81, 81  81, 108, 36  108,108,108
#4: 108,108,108,108
#5: 108,108,108,108,108
#6: 108,108,108,108,108,108
l1, l2, l3 = 108,108,108
plst3 = [l1,l2,l3]
model1m = torch.nn.Sequential(
    torch.nn.Linear(3, l1),
    torch.nn.ReLU(),
    torch.nn.Linear(l1, l2),
    torch.nn.ReLU(),
    torch.nn.Linear(l2, l3),
    torch.nn.ReLU(),
    torch.nn.Linear(l3, 3)
```

```
    ).to(device)
criterion = nn.MSELoss()
optimizer1 = optim.Adam(model11.parameters())#, lr=1e-3, momentum=0.8)
model11.train()
optimizer2 = optim.Adam(model12.parameters())#, lr=1e-3, momentum=0.8)
model12.train()
optimizerm = optim.Adam(model1m.parameters())#, lr=1e-3, momentum=0.8)
model1m.train()
num_epochs = 40
loss_list = []
for epoch in range(num_epochs):
    for batch_idx, (data, target) in enumerate(data_loader):
        data, target = Variable(data), Variable(target)
        optimizer1.zero_grad()
        optimizer2.zero_grad()
        optimizerm.zero_grad()
        output1 = model11(data.float())
        output2 = model12(data.float())
        outputm = model1m(data.float())
        loss1 = criterion(output1, target.float())
        loss1.backward()
        optimizer1.step()
        loss2 = criterion(output2, target.float())
        loss2.backward()
        optimizer2.step()
        lossm = criterion(outputm, target.float())
        lossm.backward()
        optimizerm.step()
        if epoch >2:
            if batch_idx % 200 == 0:
                loss_list.append(loss1.item())
                loss_list.append(loss2.item())
                loss_list.append(lossm.item())
        if batch_idx % 400 == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss1: {:.8f}\
```

```
    tLoss2: {:.8f}\tLossm: {:.8f}'.format(
103             epoch, batch_idx * len(data), len(data_loader.dataset),
104             100. * batch_idx / len(data_loader), loss1.item(),loss2.
    item(),lossm.item()))
105 # Error Calculation for each model tested over the same validation data
106 errmat1 = model11(x_testtens)- y_testtens
107 errnorm1 = torch.norm(errmat1, dim = 1)
108 minerr1 = '{:0.4e}'.format(torch.min(errnorm1).item())
109 maxerr1 = '{:0.4e}'.format(torch.max(errnorm1).item())
110 avgerr1 = '{:0.4e}'.format((torch.sum(errnorm1)/len(errnorm1)).item())
111 strucstr1 = '3/'
112 for i in range(len(plst1)):
113     strucstr1 = strucstr1 + str(plst1[i])+ '-' +str(plst1[i]) + '/'
114 strucstr1 = strucstr1 + '3'
115 print('|[0,'+str(T)+']'+'|'+str(strucstr1)+'|'+str(minerr1)+'|'+str(
    maxerr1)+'|'+str(avgerr1)+'|')
116 errmat2 = model12(x_testtens)- y_testtens
117 errnorm2 = torch.norm(errmat2, dim = 1)
118 minerr2 = '{:0.4e}'.format(torch.min(errnorm2).item())
119 maxerr2 = '{:0.4e}'.format(torch.max(errnorm2).item())
120 avgerr2 = '{:0.4e}'.format((torch.sum(errnorm2)/len(errnorm2)).item())
121 strucstr2 = '3/'
122 for i in range(len(plst2)):
123     strucstr2 = strucstr2 + str(plst2[i])+ '-' +str(plst2[i]) + '/'
124 strucstr2 = strucstr2 + '3'
125 print('|[0,'+str(T)+']'+'|'+str(strucstr2)+'|'+str(minerr2)+'|'+str(
    maxerr2)+'|'+str(avgerr2)+'|')
126 errmatm = model1m(x_testtens)- y_testtens
127 errnormm = torch.norm(errmatm, dim = 1)
128 minerrm = '{:0.4e}'.format(torch.min(errnormm).item())
129 maxerrm = '{:0.4e}'.format(torch.max(errnormm).item())
130 avgerrm = '{:0.4e}'.format((torch.sum(errnormm)/len(errnormm)).item())
131 strucstrm = '3/'
132 for i in range(len(plst3)):
133     strucstrm = strucstrm + str(plst3[i])+ '-' +str(plst3[i]) + '/'
```

```
strucstrm = strucstrm + '3'
print('|[0,'+str(T)+']'+'|'+str(strucstrm)+'|'+str(minerrm)+'|'+str(
    maxerrm)+'|'+str(avgerrm)+'|')
y_pred1 = model11(x_testtens).cpu().detach().numpy()
y_pred2 = model12(x_testtens).cpu().detach().numpy()
y_predm = model1m(x_testtens).cpu().detach().numpy()
y_test = y_testtens.cpu().detach().numpy()
k = 4 # parameter allows us to choose the trajectory we observe
fig = plt.figure()
ax = plt.axes(projection='3d')
# order: Blue, Orange, Green Red
ax.scatter3D(y_test[numint*k:numint*(k+1),0], y_test[numint*k:numint*(k
    +1),1], y_test[numint*k:numint*(k+1),2], cmap='Greens');
ax.scatter3D(y_pred1[numint*k:numint*(k+1),0], y_pred1[numint*k:numint
    *(k+1),1], y_pred1[numint*k:numint*(k+1),2], cmap='Greens');
ax.scatter3D(y_pred2[numint*k:numint*(k+1),0], y_pred2[numint*k:numint
    *(k+1),1], y_pred2[numint*k:numint*(k+1),2], cmap='Greens');
ax.scatter3D(y_predm[numint*k:numint*(k+1),0], y_predm[numint*k:numint
    *(k+1),1], y_predm[numint*k:numint*(k+1),2], cmap='Greens');
```

Listing 7.9: This code block contains the data generation and machine learning models and training process for a weakly nonlinear system

6.2.4 One/Two/Multiple Hidden Layers Lorenz System

```
# Parameters before collecting data
beta = 8/3
sigma = 10
rho = 28
x0 = (-8,8,27)
def lorenz_deriv(x_y_z, t0, sigma=sigma, beta=beta, rho=rho):
    x, y, z = x_y_z
    return [sigma * (y - x), x * (rho - z) - y, x * y - beta * z]
```

Listing 7.10: This code uses the same code as Listing (7.9) with the exception that initial parameters, initial condition, and function are different.