University of Sheffield

Neural ODEs for physical systems

Jagpreet Jakhar

Supervisor: Dr. Matthew Ellis

A report submitted in fulfilment of the requirements
for the degree of MSc in CyberSecurity and AI

in the

Department of Computer Science

September 13, 2023

Declaration

All sentences or passages quoted in this report from other people's work have been specifically acknowledged by clear cross-referencing to author, work and page(s). Any illustrations that are not the work of the author of this report have been used with the explicit permission of the originator and are specifically acknowledged. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this project and the degree examination as a whole.

Name: Jagpreet Jakhar

Signature: Jagpreet Jakhar

Date: 13/09/2023

Acknowledgements

I would like to thank Professor Matthew Ellis for his patience, support, and academic guidance during the process of my dissertation. I would also like to thank Professor Steve Brunton for his YouTube videos on dynamical systems, Professor Takemasa Miyoshi for his advice on tackling the Lorenz Attractor Problem, and finally, Alexander Norcliffe for his advice regarding NODE models and SONODE approaches.

Abstract

This thesis explores the application of Neural Ordinary Differential Equations (Neural ODEs) in modeling complex physical systems. The study focuses on three distinct physical systems: the double pendulum, the Lorenz attractor, and the domain wall problem. We investigate how Recurrent Neural Networks (RNNs), Long Short-Term Memory Networks (LSTMs), and Neural ODEs perform in modeling these systems and compare their predictive capabilities.

First, we analyze the behavior of the double pendulum, a chaotic and highly nonlinear dynamical system. We implement RNNs, LSTMs, and Neural ODEs to capture the intricate dynamics of this system. By comparing the model performances, we gain insights into the strengths and weaknesses of each approach in handling chaotic systems.

Next, we turn our attention to the Lorenz attractor, a well-known chaotic system in atmospheric science. We apply RNNs, LSTMs, and Neural ODEs to model the Lorenz attractor's behavior. Through comparative analysis, we assess the ability of these neural network architectures to capture the complex and chaotic nature of atmospheric phenomena.

In the final part of our study, we evaluate Neural ODEs' performance on the domain wall problem, a problem of significant importance in spintronics. We investigate the potential of Neural ODEs in modeling domain wall motion and compare their performance to conventional neural networks.

Throughout this thesis, we aim to contribute to the understanding of how Neural ODEs can be applied to model physical systems with chaotic and complex dynamics. By comparing these models to traditional neural network architectures, we gain valuable insights into their strengths and limitations. This research provides a foundation for the future development of Neural ODEs in the context of modeling and simulating physical systems.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Since time immemorial, different tools and approaches have been used to understand, control, and modify the world around us. According to Jürgen Schmidhuber[24], The method of Least Squares (Linear Regression) of Adrien-Marie Legendre and Johann Carl Friedrich Gauss is mathematically identical to today's Linear Neural Network with a same basic algorithm, error function, and adaptive parameters. Its application by Gauss on Giuseppe Piazzi led to the re-discovery of the dwarf planet Ceres[26].

Traditional approaches in machine learning to model physical systems have followed the Physics Inspired Neural Networks approach, where the prior theoretical physical knowledge is used to make models that simulate the physical system. However, this places a limitation on the type of systems that can be modelled as the complete state space of the model may not be available or not completely understood. These approaches fail to model dynamical and chaotic systems and make necessary new approaches that can deal with systems.

In this dissertation, an attempt is made to use the combination of Ordinary Differential Equations with Deep learning in the Neural Ordinary Differential Equations (NODE)[8] model by examining its suitability to model dynamic and chaotic physical systems.

1.1 Aims and Objectives

This Dissertation aims to explore the potential of a particular ML approach known as Neural Ordinary Differential Equations (NODEs)[8] for modeling specific physical systems that exhibit complex non-linear dynamics. This can be used for predicting the system's time evolution and computational analysis of the system properties, which may be unfeasible in an experimental setting.

Specific objectives of the Dissertation are :

- Evaluating NODEs on Double Pendulum, and comparing to performance of RNNs and LSTM by comparing Mean Squared Errors, visualization, and with the help of vector fields(see Section 2.2.1)

- Evaluating NODEs on Lorenz63 Model[21], and comparing to performance of RNNs and LSTM by comparing Mean Squared Errors, visualization, and with the help of vector fields simulation.
- Evaluating NODEs for predicting Domain Wall dynamics and drawing vector fields to compare if the model predictions mimic the system's dynamics.
- Analysing results of the above experiments, particularly the dynamics of each model's predictions(vector fields) and future directions.
- Critically analyzing the advantages, disadvantages, and limitations of Neural ODEs.

1.2 Overview of the Report

This Dissertation is comprised of six chapters, including this one. The Second chapter contains the Literature Review, which lays down the theoretical ground to provide a better understanding to the reader. Chapter 3, Design & Implementation, contains how the models implemented are designed, the tools and functions used, the training paradigm of these models, etc. Chapter 4, Analysis & Results, discusses the findings and implications of the work done in Chapter 3 and the analysis of Neural Ordinary Differential Equations as an architecture. Chapter 5 Conclusions summarises the work done and results obtained throughout the dissertation. Chapter 6, Future Work, gives a brief overview of the work that should be pursued further in line with the current objective to become able to model chaotic and dynamical systems in an interpretable way.

Chapter 2

Literature Review

In this chapter, a brief overview is given of the concepts and physical systems used in the dissertation. First, Ordinary Differential Equations are introduced along with Initial Value Problem(IVP) and ODE Solvers used. Then briefly, dynamical systems and chaotic systems are described moving on to the systems that are being considered in this dissertation, like double pendulum, Lorenz attractor, and Domain Wall dynamics. Finally, the models that are implemented to try to learn these systems are discussed, namely, Recurrent Neural Networks, Long Short Term Memory RNNs, and Neural Ordinary Differential Equations.

2.1 Ordinary Differential Equations(ODEs)

In contrast to algebraic equations, whose solution is a set of values, the solution to a differential equation is solved with the help of a function that can describe the system. Differential equations operate on the assumption that time and space are a continuum and, as a result, can be broken into infinitesimally small pieces. They were first described by Isaac Newton in his fluxional equations. ODEs are fundamental mathematical tools used to model dynamic systems and describe the evolution of continuous phenomena in various fields, including physics, engineering, biology, and economics. ODEs are essential for understanding processes that change continuously over time.

Ordinary Differential Equations are equations that depend upon a single independent variable The general form of an ODE:

$$\frac{d}{dt}(y(t)) = f(t, y(t)) \tag{2.1}$$

From initial state x_0 at $t = 0$, the state at $t = c$ is given by:

$$x(c) = x_0 + \int_0^c f(t, x(t))\, dt \tag{2.2}$$

If the equation can be solved, we can get an analytical solution. If not, we have to rely on numerical methods. In this dissertation, the problems that are tackled require numerical

solutions, and therefore, the help of ODE Solvers is taken, described in the section below.

Initial Value Problem (IVP): An IVP is a specific type of ODE problem where the goal is to determine the function $y(t)$ that satisfies both the ODE and the initial condition $y(t_0) = y_0$. In essence, we seek to understand how the dependent variable $y(t)$ evolves over time from the initial state x_0 according to the ODE. Numerical methods are used for solving IVP, as closed-form solutions seldom exist.

2.1.1 ODE Solvers

Solving ODEs analytically is often impractical or impossible for complex real-world problems. Numerical methods can provide effective solutions by approximating the continuous behavior of ODEs. In this dissertation, three ODE solvers are used, namely Euler's method, Runge-Kutta 4^{th} order method(RK4), and Dormand-Prince method(DOPRI5).

- Euler's method is a simple and intuitive numerical technique for solving ODEs given by Leonhard Euler in 1768 in his book 'Introductio in analysin infinitorum' [14]. It is based on the following formula:

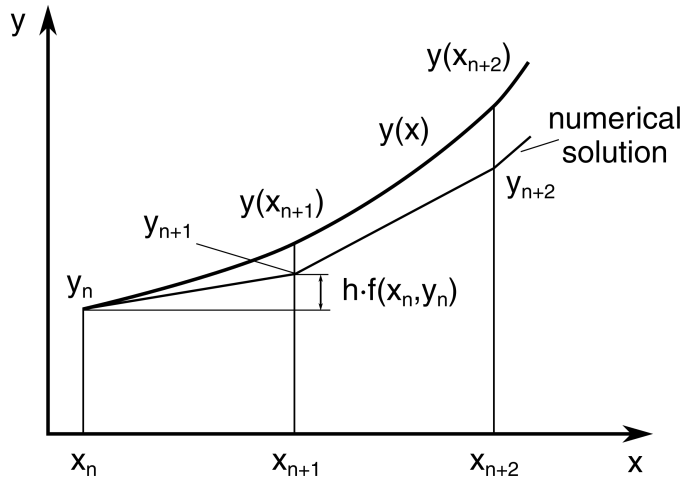


Figure 2.1: Euler's Method for solving and ODE

$$y_{n+1} = y_n + h \cdot f(t_n, y_n) \tag{2.3}$$

Where:

y_{n+1} is the new approximation of the solution,

y_n is the current approximation,

h is the step size,

$f(t_n, y_n)$ represents the derivative of the function at time t_n and state y_n.

Euler's method is easy to implement but has limited accuracy, particularly for ODEs with rapid changes or stiff behavior.

- The Runge-Kutta 4$^{\text{th}}$ order (RK4) method[20] is a higher-order fixed-step numerical technique that provides improved accuracy compared to Euler's method. It calculates four intermediate values to approximate the solution at the next time step. The algorithm is as follows:

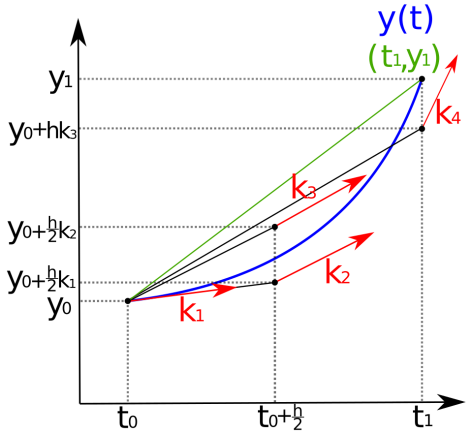


Figure 2.2: Runge-Kutta 4$^{\text{th}}$ Order Method for solving and ODE

$$
\begin{aligned}
k_1 &= h \cdot f(t_n, y_n) \\
k_2 &= h \cdot f(t_n + \frac{h}{2}, y_n + \frac{k_1}{2}) \\
k_3 &= h \cdot f(t_n + \frac{h}{2}, y_n + \frac{k_2}{2}) \\
k_4 &= h \cdot f(t_n + h, y_n + k_3) \\
y_{n+1} &= y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)
\end{aligned}
$$

Where:

k_1 is the first slope estimate using y {Euler's method} .

k_2 is the second slope estimate, which is calculated by using y and k_1

k_3 is the third slope estimate, calculated by using y and k_2.

$k_4$final step calculated by using y and k_3

- The Dormand-Prince method[6] (DOPRI5) is an adaptive step-size fifth-order method that combines fifth and fourth-order approximations.DOPRI5 adjusts the step size

dynamically to maintain a desired level of accuracy while minimizing computational cost.

The DOPRI5 method is as follows:

$$
\begin{aligned}
k_1 &= h \cdot f(t_n, y_n) \\
k_2 &= h \cdot f(t_n + \frac{h}{5}, y_n + \frac{k_1}{5}) \\
k_3 &= h \cdot f(t_n + \frac{3h}{10}, y_n + \frac{3}{40}k_1 + \frac{9}{40}k_2) \\
k_4 &= h \cdot f(t_n + \frac{4h}{5}, y_n + \frac{44}{45}k_1 - \frac{56}{15}k_2 + \frac{32}{9}k_3) \\
k_5 &= h \cdot f(t_n + \frac{8h}{9}, y_n + \frac{19372}{6561}k_1 - \frac{25360}{2187}k_2 + \frac{64448}{6561}k_3 - \frac{212}{729}k_4) \\
k_6 &= h \cdot f(t_n + h, y_n + \frac{9017}{3168}k_1 - \frac{355}{33}k_2 + \frac{46732}{5247}k_3 + \frac{49}{176}k_4 - \frac{5103}{18656}k_5) \\
y_{n+1} &= y_n + \frac{35}{384}k_1 + \frac{0}{1}k_2 + \frac{500}{1113}k_3 + \frac{125}{192}k_4 - \frac{2187}{6784}k_5 + \frac{11}{84}k_6
\end{aligned}
$$

The error at each step depends upon step size; the method adapts the step size using an algorithm, which can be controlled using the Absolute tolerance parameter by the user.

2.2 Dynamical and Chaotic Systems

Dynamics began as a branch of physics with Newton, who combined differential equations and his laws of physics with Kepler's laws of motion to solve the two-body problem, i.e., the problem of calculating the motion of Earth around the Sun[25]. However, this approach could not solve the three-body problem (sun, earth, and moon) until Henri Poincare gave the dynamical systems theory with his geometric approach.

Henri Poincare also propounded chaos theory, as he observed while attempting to solve the three-body problem the existence of aperiodic orbits, which neither increased nor approached a fixed point. It was observed that deterministic systems exhibit aperiodic behavior due to sensitivity to initial conditions, making long-term predictions impossible[25].

The availability of computation power in the late 1950s led to the discovery of the Lorenz 63 model by Edward Lorenz, reigniting interest in chaotic and dynamical systems. Later, Feigenbaum's discovery of certain universal laws under which different systems can go chaotic similarly gave the link between chaos and phase transitions, further piquing interest in chaos theory.

Studying chaotic systems has important implications, from sending probes to the moon, quantum mechanics, weather forecasting, etc.

2.2.1 Vector Fields for interpretation of Dynamics

Vector fields are a powerful mathematical tool used to describe many physical phenomena such as Gravitation, Electromagnetism, etc., and give a visual intuition for interpreting these systems. In the context of neural networks and deep learning, vector fields can help us gain insights into the behavior and training dynamics of these complex models by visualizing weight updates, gradient flows, and activations during the training phase; for example, understanding whether trajectories lead to stable fixed points or oscillate, or on predictions made by the model in the testing phase. In this dissertation, they are utilized for the predictions made by the models to gain insights and identify any discernible patterns.

However, there are also a few limitations of vector fields when used in the context of neural networks as they are approximations due to the high dimensionality of neural networks and can be quite complex to understand

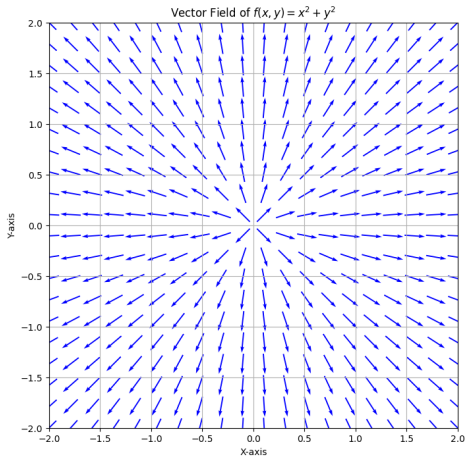


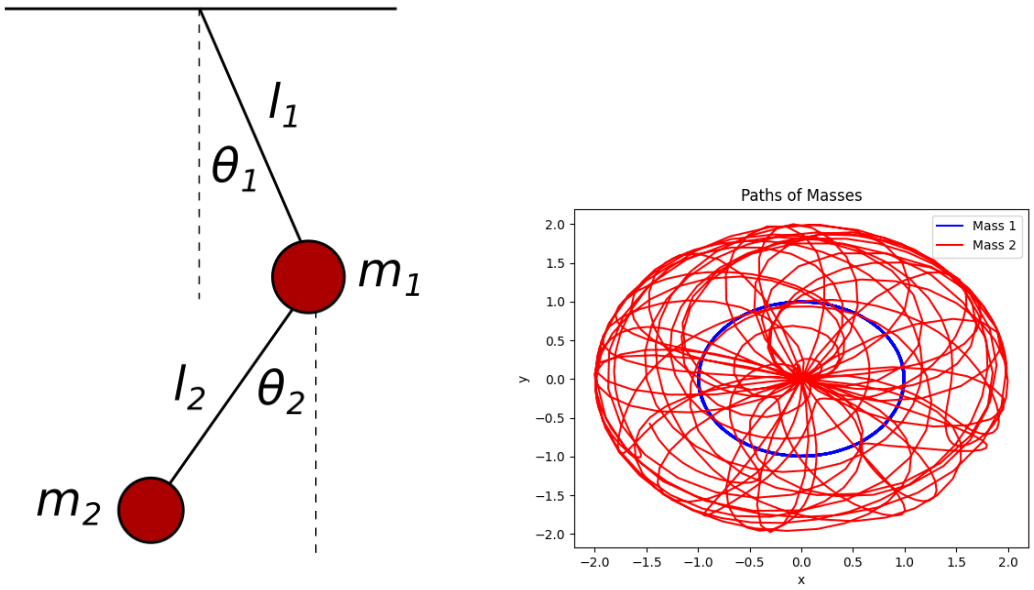
Figure 2.3: Vector Field of $x^2 + y^2$

2.3 Double Pendulum Problem

The double pendulum is a classic example of a chaotic dynamical system. It consists of two pendulum arms attached to each other, creating a system with complex and non-linear dynamics. For small angles, the system behaves stably, but for large angles, it goes chaotic. Understanding the motion of a double pendulum is not only a fascinating mathematical challenge but also has practical implications in fields such as robotics and celestial mechanics.

2.3.1 Equations of Motion

The equations of motion for a double pendulum are derived from the principles of classical mechanics. Let's denote the angles of the two pendulum arms with respect to the vertical as θ_1 and θ_2. The lengths of the arms(assumed to be weightless) are L_1 and L_2, and the masses at the ends of the arms are m_1 and m_2. The gravitational acceleration is g.



(a) Double Pendulum (b) Paths of masses

Figure 2.4: Left- Double Pendulum, Right-Paths sketched by Masses 1 and 2

The equations of motion for the double pendulum are given by:

$$\dot{\theta}_1 = \omega_1 \tag{2.4}$$

$$\dot{\theta}_2 = \omega_2 \tag{2.5}$$

$$\dot{\omega}_1 = \frac{-g(2m_1 + m_2)\sin(\theta_1) - m_2 g \sin(\theta_1 - 2\theta_2) - 2\sin(\theta_1 - \theta_2)m_2(\omega_2^2 L_2 + \omega_1^2 L_1 \cos(\theta_1 - \theta_2))}{L_1(2m_1 + m_2 - m_2\cos(2\theta_1 - 2\theta_2))} \tag{2.6}$$

$$\dot{\omega}_2 = \frac{2\sin(\theta_1 - \theta_2)(\omega_1^2 L_1(m_1 + m_2) + g(m_1 + m_2)\cos(\theta_1) + \omega_2^2 L_2 m_2 \cos(\theta_1 - \theta_2))}{L_2(2m_1 + m_2 - m_2\cos(2\theta_1 - 2\theta_2))} \tag{2.7}$$

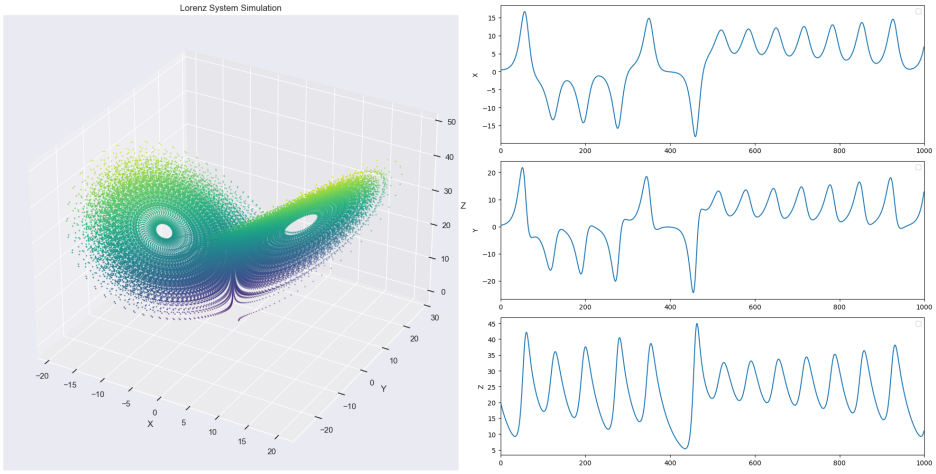
Where:

- θ_1 and θ_2: Angles of the two pendulum arms with respect to the vertical.
- L_1 and L_2: Lengths of the pendulum arms.
- m_1 and m_2: Masses at the ends of the pendulum arms.
- g: Gravitational acceleration.
- $\dot{\theta}_1$ and $\dot{\theta}_2$: Angular velocities of the top and bottom pendulum arms.
- $\dot{\omega}_1$ and $\dot{\omega}_2$: Angular accelerations of the top and bottom pendulum arms.

2.4 Lorenz Attractor

The Lorenz attractor[21] is a well-known chaotic system with complex and unpredictable behavior. It was first studied by Edward Lorenz in 1963 as a simplified model of Earth's atmosphere and has since become a prominent example in the field of chaos theory.

The system is described by a set of three coupled ordinary differential equations known as the Lorenz equations. These equations capture the system's dynamics and govern the evolution of its state variables over time.



(a) Lorenz Attractor (b) X,Y and Z coordinates

Figure 2.5: Left- Lorenz 63 attractor, Right-x,y,z coordinates

The Lorenz equations are given by:

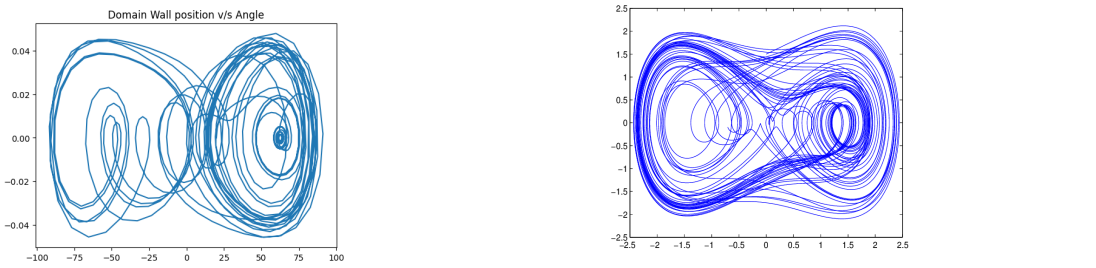
$$\begin{aligned}\frac{dx}{dt} &= \sigma \cdot (y - x) \\ \frac{dy}{dt} &= x \cdot (\rho - z) - y \\ \frac{dz}{dt} &= x \cdot y - \beta \cdot z\end{aligned}$$

where:

- $x, y,$ and z are the state variables representing the position of the system in three-dimensional space.
- $\sigma, \rho,$ and β are constants that determine the behavior of the system. They control the system's sensitivity to initial conditions and its overall dynamics.

2.5 Magnetic Domain Wall Problem

In this dissertation, the dynamics of a domain wall placed between two anti-notches (defects designed to repel the domain walls) of nickel nanowire[1] are examined, which is fed an oscillating magnetic field that causes it to go into non-linear phase, similar to a Duffing Oscillator.



(a) Domain Wall Phase (b) Duffing Oscillator Phase Portrait

Figure 2.6: Left Domain Wall, Right Duffing Oscillator

Domain Walls in nanomaterials occur because, at the nanoscale, the geometry of the ferromagnetic material dominates over properties of the material leading to irregular magnetization distribution (Domains)[4]. Domain Walls are the boundaries that separate these regions or domains of different magnetic orientations. This can be controlled through careful engineering to produce useful configurations which can be manipulated by applying a magnetic field.

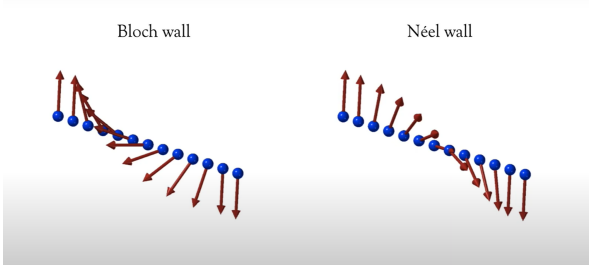


Figure 2.7: One Dimensional Domain Walls - Bloch[3] snf Neel Wall [23]

We are only interested in One-dimensional Domain Walls for the purposes of this dissertation, in particular with Bloch given by Felix Bloch[3] and Neel Wall given by Louis Neel[23]. In the Bloch wall, as shown in the figure above, the magnetization in the center is perpendicular to the direction of the domain wall resulting in spatial separation of magnetic charges. In the Neel Wall, on the other hand, the magnetization at the center is aligned along the direction of the domain wall resulting in strong interactions between magnetic charges.

The aim is to model the dynamics of the domain wall in two cases, one a single field case where a single field is applied for a duration and measures the domain Wall position

and angle (or its magnetization angle); the other case is when multiple fields with the same duration are applied and measurements taken.

The successful simulation of these dynamics can provide many benefits, including saving time from time-intensive simulations such as mumax simulations as done by Chen et.al[9], or advances their applications in Neuromorphic Computing in line with Ellis et al. [1]

2.6 Recurrent Neural Networks(RNNs)

One of the challenges when modelling systems or processes has been to incorporate Time in the model, which can be a factor in how a system evolves and how the interactions of its components evolve. This challenge persists in various fields ranging from Natural Language Processing to modelling physical systems. Jeffrey Elman in his paper "Finding Structure in Time"[13] introduced the Simple Recurrent Neural Network(SRNN) or Elman Network, which could learn long-term dependencies and gave the foundation to today's RNNs we use.

2.6.1 Architecture

The central idea in Elman's paper[13] was to represent time by the effect it has i.e to give the network memory. It was achieved by using a context layer, which was a copy of the hidden layer and was fed back to the hidden layer at each timestep to maintain the memory of previous inputs.

The architecture of a traditional RNN is as follows :

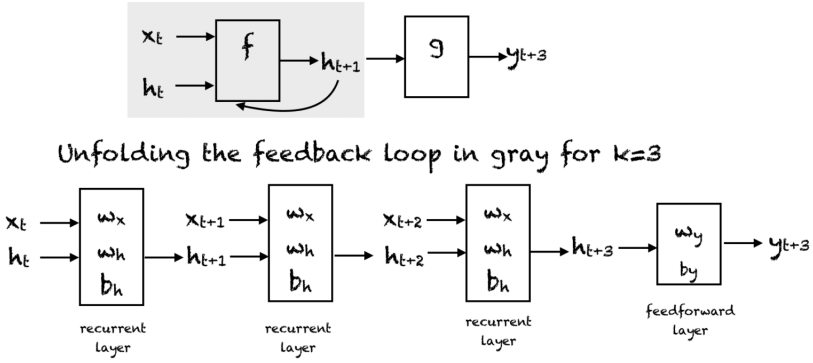


Figure 2.8: Recurrent Neural Network (RNN) Architecture[5]

The hidden state at time step t is updated using the following equation:

$$h(t) = f(W(x \cdot h(t-1)) + b) \tag{2.8}$$

where:

$h(t)$ is the hidden state at time step t
f is an activation function (e.g., tanh or ReLU)
W represents the weight matrices
x is the input at time step t
b is the bias term

The output at time step t can be calculated as follows:

$$y(t) = g(U \cdot h(t) + c) \tag{2.9}$$

where:

$y(t)$ is the output at time step t
g is an activation function (e.g., softmax for classification)
U represents the weight matrix for the output layer
c is the bias term for the output layer

Training

The training of a Recurrent Neural Network is different than of a Multi-Layer Perceptron, where the Backpropagation algorithm is modified as Backpropagation through time(BPTT)[27] where the network is unrolled, errors calculated and accumulated for each timestep. The algorithm is as follows :

- To compute the gradient of the loss with respect to the output at time t, use standard backpropagation for feedforward layers. Let $\delta(t)$ represent the gradient of the loss at time t with respect to the output $y(t)$:

$$\delta(t) = \frac{\partial L(t)}{\partial y(t)} \tag{2.10}$$

- Compute the gradient of the loss with respect to the hidden state at time t ($\frac{\partial L(t)}{\partial h(t)}$) by backpropagating the gradient from the output layer and through the hidden layer using the chain rule:

$$\frac{\partial L(t)}{\partial h(t)} = \delta(t) \cdot U^T + \frac{\partial L(t+1)}{\partial h(t+1)} \cdot W \tag{2.11}$$

- Use the gradients from eq(2.4) to update the model's parameters.

Although RNNs give several advantages over traditional Multilayer Perceptrons, they also suffer from many limitations, one of which is the issue of Vanishing Gradients or

Exploding gradients if the input sequences are very long, and places a limit on the long-term dependencies it can model.

2.7 Long short-term memory RNNs(LSTM)

LSTM was introduced to tackle the problem of vanishing and exploding gradients in RNNs by Hochreiter and Schmidhuber in 1997[18]. The central idea was introducing memory cells (cell state) and gate units in the hidden state of the RNN that allow storing select information in long-term memory in cell states, with the hidden state storing the short-term memory and the gates allowing the flow of gradients through time.

2.7.1 Architecture

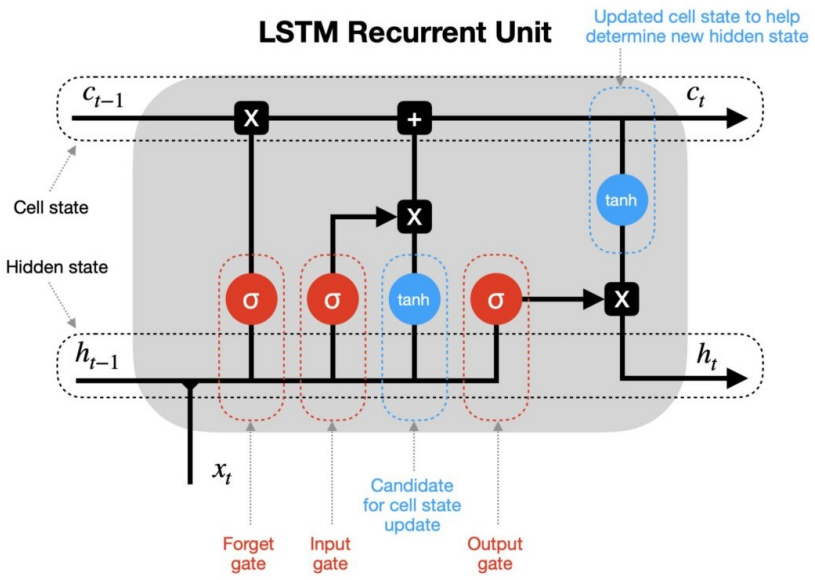


Figure 2.9: Long Short Term Memory(LSTM) Architecture[11]

The key components of an LSTM unit are as follows:

1. Cell State (c_t): The cell state represents the memory of the network and can retain information over long sequences.

$$\tilde{c}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c) \tag{2.12}$$

which is updated as :

$$c_t = f_t \cdot c_{t-1} + i_t \cdot \tilde{c}_t \tag{2.13}$$

2. Hidden State (h_t): The hidden state carries relevant information to the current time

step and is influenced by both the cell state and the current input.

$$h_t = o_t \cdot \tanh(c_t) \tag{2.14}$$

3. Gates: - Forget Gate (f_t): Controls what information from the previous cell state should be forgotten.

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{2.15}$$

- Input Gate (i_t): Regulates the update to the cell state, deciding which new information to store.

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{2.16}$$

- Output Gate (o_t): Determines what part of the cell state should be exposed as the hidden state.

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{2.17}$$

Where:

- x_t is the input at time step t.
- $[h_{t-1}, x_t]$ represents the concatenation of the previous hidden state h_{t-1} and the current input x_t.
- W_f, W_i, W_c, W_o are weight matrices for the forget gate, input gate, candidate cell state, and output gate, respectively.
- b_f, b_i, b_c, b_o are bias terms.
- tanh represents the hyperbolic tangent activation function.

The training of LSTMS is similar to RNNs where Backpropagation through time(BPTT) is combined with some optimization algorithm like Stochastic Gradient Descent(SGD).

Although LSTMs were able to overcome many limitations of the RNNs, they still lack the sufficient capacity to be able to model chaotic and dynamical systems, as shown later in the Analysis section.

2.8 Neural Ordinary Differential Equations(NODEs)

Neural Ordinary Differential Equations(NODEs) as a machine learning approach were popularized by Ricky T.Q. Chen et.al[8] in their paper "Neural Ordinary Differential Equations" where a Residual Network[17] with skip connections is seen as a solution of an Ordinary Differential Equation with Euler's method wherein the residual connections are the discretized time steps of the Euler's method.

Residual Networks use skip connections to learn small iterative changes to the input as shown below :

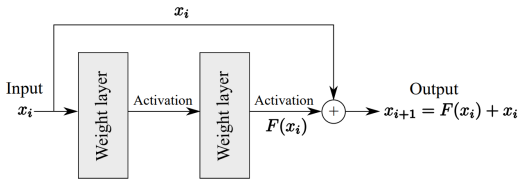


Figure 2.10: ResNet Architechture [2]

$$x_{t+1} = x_t + hf(x_t, \theta_t), \quad h = 1 \tag{2.18}$$

where x_t is the input to the current block, $f(x_t, \theta_t)$ is a function parametrized by θ_t, and x_{t+1} is the layer output, as seen above. This can be seen as a single step of forward Euler discretization (section 2.1.1):

$$y_{i+1} = y_i + hF(t_i, y_i) \tag{2.19}$$

of the ODE $\frac{dy(t)}{dt} = F(t, y(t)$ (Eq. 2.1) The idea is that when the number of layers n in the network is increased and the step size h is decreased, in the limit $n \to \infty$, $h \to 0$, the continuous derivative can be parametrized by a neural network[2][8]:

$$\frac{dy(t)}{dt} = f(y(t), t, \theta) \tag{2.20}$$

Starting with an initial condition $y(0)$, the output $y(T)$ can be specified as the solution to the ODE at time T which can be computed with any ODE solver as:

$$y(T) = y(0) + \int_0^T \frac{dy(t)}{dt} dt = y(0) + \int_0^T f(y(t), t, \theta) dt. \tag{2.21}$$

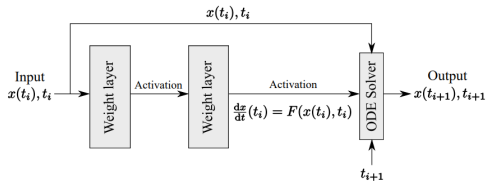


Figure 2.11: NODE Architecture [2]

Therefore, these models are continuous in-depth models where the ODE solver's steps are equivalent to the number of layers required. This is illustrated below where the dynamics(vector fields) of a Residual Network and a NODE model are shown[8]:

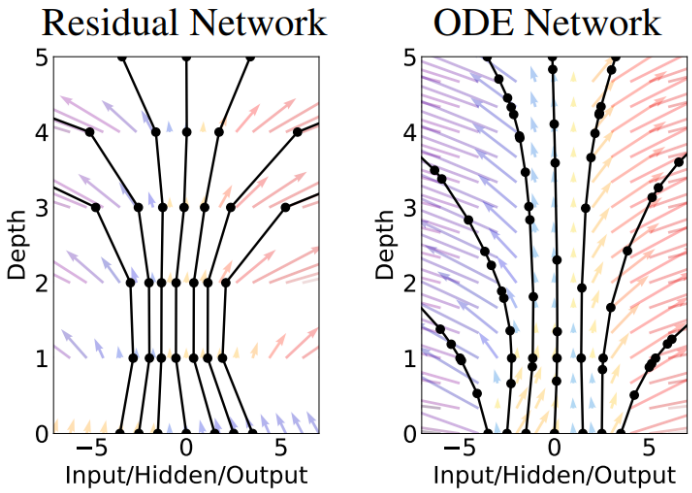


Figure 2.12: ResNet vs NODE dynamics [8]

2.8.1 Adjoint Method

The challenge when a Neural Network is parameterized by an ODE Solver is high memory cost and numerical instability when backpropagating through the ODE Solver. The Differentiate than Discretize approach[7] in NODE with the adjoint sensitivity method where the backward pass is by solving a second Augmented ODE backward in time by using adjoint states. This method saves memory, is numerically stable, and scales linearly with problem size[8].

2.8.2 Training

The central idea is to use the data points available in the form of a time series by integrating the model between any intermediate data points, calculating Loss, and then backpropagating to update the model parameters.

Chapter 3

Design & Implementation

In this chapter, the tools and libraries used, as well as the design of the model setup, dataset creation, and simulation, along with implementation, are discussed.

3.1 Tools used

3.1.1 Torchdiffeq[8]

The Torchdiffeq library was used for ODE SOlvers as it provides full integration with PyTorch, as well as has many types of ODE solvers available from fixed step such as Runge-Kutta 4^{th} order to adaptive size solver such as Dormand-Prince method. It also supports backpropagation using ODE Solvers and the adjoint method, which can save memory when running the models.

3.1.2 Weights and Biases

Weights and Biases was used to find hyperparameters and log various model runs in different configurations to select optimum hyperparameters for the model, such as lookback for LSTM and RNN models and stepsize or choice of solver for the NODE model, etc.

3.2 Choice of Loss Function

Mean Squared Error Loss (MSE) or L2 loss measures the average of the squared differences between predicted and actual values, measuring how well a model's predictions match the true data. It is given as :

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - y_{\text{true},i})^2 \tag{3.1}$$

Where:

N is the number of data samples.

y_i is the predicted value for the i-th sample.

$y_{\text{true},i}$ is the true (actual) target value for the i-th sample.

MSE loss is used as the default loss function for the first two problems, namely Double Pendulum and Lorenz Attractor. It is used as the primary metric for the evaluation of all three systems

Huber loss was used as the loss function for training the NODE on the domain Wall in both cases - Single Field and Multiple Fields, as it is less sensitive to outliers than the Mean Squared Error Loss function. It is given by :

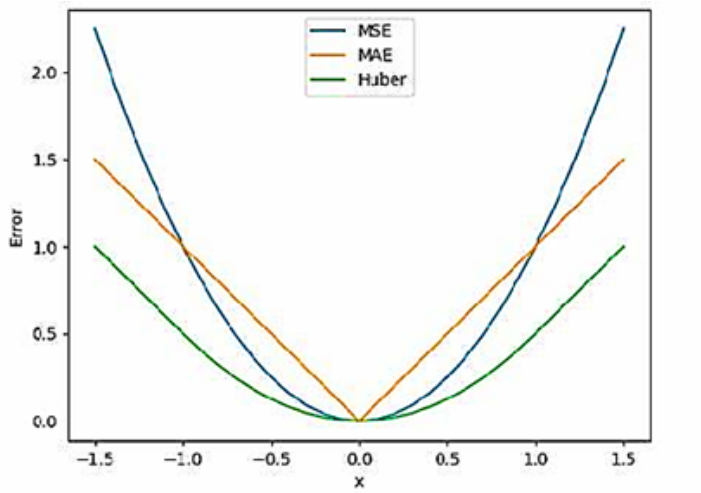


Figure 3.1: Huber Loss vs MSE vs MAE

$$L(y, \hat{y}) = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & \text{if } |y - \hat{y}| \leq \delta \\ \delta|y - \hat{y}| - \frac{1}{2}\delta^2 & \text{otherwise} \end{cases}$$

Where:

y is the true target value.

$\hat{y}$ is the predicted value.

δ is a user-defined threshold determining when to switch MSE and MAE loss components.

When loss is large due to outliers, the MAE is used, which restricts the weight we put on outliers, whereas when the loss is less than one(default δ), MSE loss forces the model to train on high error data points.

However, Huber loss, due to its complexity, can be more difficult to interpret than MSE,

so MSE is used as a metric when analyzing model performance on the test set in the Analysis chapter.

3.3 Choice of Activation Functions

Choosing activation functions is a critical decision that can significantly impact the model's behavior and performance. The following Activation Functions are used in this dissertation for various models and problems as specified below.

Sigmoid Activation (Logistic Function) It is defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Sigmoid squashes input values to the range (0, 1), making it suitable for binary classification problems where the output represents probabilities. In this dissertation, it is used in the vanilla LSTM models, where it is used as the activation function for input, forget, and output gates, as shown in Eq. (2.15 to 2.17).

Tanh Activation (Hyperbolic Tangent) It is is defined as:

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

It squashes input values to the range (-1, 1), offering advantages for training deep networks as it provides zero-centered outputs. It is used as the activation function on the Cell state and for calculating the hidden state using the Cell state in the implementation of LSTM. Tanh is also used as the activation function on the last layer in the Neural Ordinary Differential Equations model in combination with LeakyReLu activation on previous layers.

ReLU (Rectified Linear Unit) Activation It is defined as:

$$\text{ReLU}(x) = \max(0, x)$$

ReLU introduces non-linearity by outputting zero for negative inputs and passing positive inputs unchanged. It has many advantages, such as it is computationally efficient and can accelerate training; it also mitigates the vanishing gradient problem by providing a constant gradient for positive inputs. Many activations are zero, making the network sparse.

Leaky ReLU Activation ReLu activation can cause the "dying ReLU" problem, where neurons get stuck with zero output during training and never recover. The Leaky ReLU activation function is a variant of ReLU that allows a small, non-zero gradient for negative

inputs to prevent the "dying ReLU" problem. It is defined as:

$$\text{LeakyReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{otherwise} \end{cases}$$

where α is a small positive slope. It retains the benefits of ReLU, while the only disadvantage being an additional hyperparameter to tune.

3.4 Double Pendulum Problem

The data for the Double Pendulum problem is created using ten random initial conditions and then using the torchdiffeq[8] odeint solver to generate the data. The data is further divided into a train set (containing data generated from the first six random initial conditions), a validation set (containing the data generated from the seventh initial condition), and finally, the test set, which contains the data generated from the last three initial conditions.

The initial conditions are as follows :

Table 3.1: Data Split for Training, Validation, and Test

Double Pendulum Dataset	$\boldsymbol{\theta_1}$	$\boldsymbol{\theta_2}$	$\boldsymbol{\omega_1}$	$\boldsymbol{\omega_2}$
Training	3.7692	1.1990	-0.1639	-0.2575
Training	1.8363	2.1619	0.3461	-0.9771
Training	5.1896	0.2047	0.1286	-0.9194
Training	1.3936	4.8303	-0.3305	0.4154
Training	3.7759	5.8581	-0.9284	0.8907
Training	2.2445	4.1690	-0.7012	-0.0152
Validation	0.4165	1.1977	0.0542	-0.8938
Test	1.0960	4.7987	-0.0536	0.3550
Test	4.4249	5.7488	0.8258	0.8145
Test	4.3848	2.4989	0.0342	-0.6619

Table 3.2: Double Pendulum Dataset Split

Then three model architectures are trained and tested on this data, namely a vanilla Recurrent Neural Network(RNN), a vanilla Long Short Term Memory RNN (LSTM), which are used as a baseline to compare against the performance of the final NODE model. Mean Squared Error is used as the metric along with vector fields of the predictions made by the models to draw analysis and conclusions. Single-step predictions were not undertaken as they are not suitable for the experiments done in this dissertation and have limited utility anyhow. The abilities of the model being evaluated are Forecasting Ability and Generalizability.

3.4.1 Double Pendulum - RNN

An RNN model with a single hidden layer with a width of 512 Neurons and a ReLu activation function is used to train on the date of Double Pendulum Simulation. It takes as input four values, namely the angles θ_1 and θ_2, and the angular momentum of the two masses, namely, ω_1 and ω_2, and outputs the values forward in time.

A lookback of 10 was selected after experimenting with multiple values such as 5,10,15,20,30, etc. The model is trained for 1000 epochs, similar to the LSTM and NODE models. A batch_size of 8 was chosen to see outputs eight timesteps forward.

When training, the MSE loss function is used, and the model with the lowest error is saved at every checkpoint by comparing the loss at every epoch. This model is later used for evaluation on the test set, wherein the model is given an initial condition and is evaluated by giving its outputs as its next inputs. The results of the evaluation of the test set and further comparisons are made in Chapter 4.

3.4.2 Double Pendulum - LSTM

Similar to the RNN approach, a vanilla LSTM is used to train on the training set and then evaluated on the test set. The same lookback value is used for the LSTM, and the same optimizer, loss function, training approach, and evaluation approach on the test set. Further analysis is done in Chapter 4.

3.4.3 Double Pendulum - NODE

The Neural ODE approach to training on Double Pendulum is slightly different than the RNN and LSTM approach. The model is trained by giving it an initial condition and integrating it into 50 times steps, and then loss is calculated based on error. The NODE is trained for 3000 epochs, with batch sizes of 50.

The NODE is comprised of an ODE Solver, which comprises an MLP with two hidden layers with the ReLu activation function. MSE is used as the loss function to train the model and Adam optimizer. The trained model is then evaluated on the test sets by giving it an input of the initial condition of the test set and then using the model as the function to the ODE Solver to forecast the full timesteps of the test data.

3.5 Lorenz Attractor Problem

The dataset for Lorenz attractor is made by using ten random initial points within a range of 15 to -15 [25] which cause the attractor to behave chaotically, displaying the butterfly-like effect when plotted. The dataset is divided into with first six points as the training set, the seventh point as the validation set, and the last three as the test set, an approach similar to that of the double pendulum considered previously.

DataSet	X	Y	Z
Training	-8.8545	-0.2770	-3.8285
	1.8363	2.1619	0.3461
	5.1896	0.2047	0.1286
	1.3936	4.8303	-0.3305
	3.7759	5.8581	-0.9284
	2.2445	4.1690	-0.7012
Validation	0.4165	1.1977	0.0542
Test	1.0960	4.7987	-0.0536
	4.4249	5.7488	0.8258
	4.3848	2.4989	0.0342

Table 3.3: Lorenz Data Split into Training, Validation, and Test Sets

Now, the three models are trained on the training data, Mean Squared Error, Visualisation, and Vector Fields and evaluated to compare the three models. The abilities of the model being evaluated are Forecasting Ability and Generalizability.

3.5.1 Lorenz Attractor - RNN

The vanilla RNN architecture is again used here for the Lorenz problem; The RNN model consists of a single hidden layer, with a width of 512 neurons and ReLu activation function, with Adam optimizer and MSE as loss function.

The model's state with the lowest errors is stored at every checkpoint during training and then is used to evaluate the test data. The evaluation of test data is done by giving the RNN model the initial state of the test data and then using it to forecast up to the entire length of the test data by using its outputs after each step as inputs for the next steps and then calculating error.

A lookback of size ten is used in conjunction with a batch size of 8 to train the model; the Single-step prediction is again ignored in favor of experimenting with prediction over longer horizons.

3.5.2 Lorenz Attractor - LSTM

The implementation and evaluation of the LSTM model are very similar to the RNN model above.

3.5.3 Lorenz Attractor - NODE

The NODE model for the Lorenz attractor is a single hidden layer MLP parameterized by an ODE, with three inputs and three outputs. A batch size of 50 is used to train the model by giving it the initial condition of the batch, which is then integrated by the ODE solver using the model as the function, Then errors are calculated, and parameters of the MLP are updated using backpropagation. The NODE model is run for 3000 epochs, and the model

state with the lowest loss is stored after checking at every checkpoint, which is then used to evaluate the test set.

The batch size of 50 was chosen after considering other batch sizes such as 15,20,25, 75,100, etc. However the batch size of 50 can capture the nonlinearities in the data and balance the numerical error that can creep up on longer timescales.

3.6 Magnetic Domain Wall

The Magnetic Domain Wall dynamics are simulated using 'DW_oscillator.py' in Appendix A.1. The class is initialized with various physical parameters, including the magnetic saturation moment (M_s), exchange stiffness constant (A), Gilbert damping coefficient (α), dimensions of the magnetic system (L), anisotropy constants (a and b), external magnetic field amplitude (h_{const}), oscillation frequency (f set to 0.5), and time-dependent external magnetic field (h_{time}).

The function run_field_sequence (RFD) takes the input range of fields(low, high), number of fields to be generated, duration of each field, initial domain wall position, and angle to generate and simulate the set of fields randomly within that range. It returns the domain Wall position, angle, and input sequence upon which the NODE is trained.

The external magnetic field, which serves as the oscillating input field to the domain wall, is also used by calculating its gradient using a function sequence in Appendix A.2 to attach its gradient in the forward pass of the Multilayer Perceptron always to have the correct driving force.

3.6.1 General Architecture of NODE used

The ODE-solver parametrizes a 5-layer MultiLayer Perceptron, which takes an input of size three and outputs two values, namely the predicted Domain Wall Position and Angle. The data simulated by the RFD function is first normalized to the range -1 to 1, as the domain wall position initially is at a different scale than the angles, which are in radians. they are then stacked together to form the data along with the time given by the RFD function.

This data is then fed to the ODESolver using the adjoint method and Dormand-prince solver(dopri5) to make a prediction on the domain wall position and angle using the initial position and time steps selected randomly from the data using a get_batch function.

The Huber Loss function is used over the Mean Squared Error Loss function as it gives better results than the MSE when training. Once the prediction error is calculated, Backpropagation of error signals is done using the Adjoint method[8]. Finally, the trained model is tested on a set containing randomly generated fields by the RFD function. AdamW optimizer, a modification of Adam optimizer, is used, which decouples weight decay from gradient update.

3.6.2 Domain Wall Single Field NODE

For the Single Field Case, the following initial conditions were given to generate the training data :

Number of Fields	Field Magnitudes	Duration	Initial Position	Initial Angle
1	100.	60.	0.0	0.0

Table 3.4: Single Field Initial Conditions

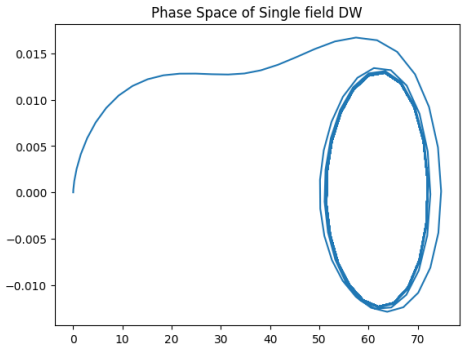
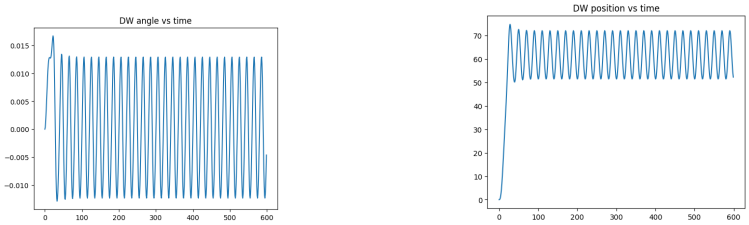


Figure 3.2: Single Field Domain Wall phase space with magnitude: 100.



(a) Single Field Domain Wall Angle (b) Paths of masses

Figure 3.3: Left- Single Field Domain Wall Angle, Right- Single Field Domain Wall Position

As is observable from Figures 3.2 and 3.3, the dynamics of the domain wall settle down in a stable oscillatory phase after some time steps. The NODE model for the single field case only uses the LeakyReLu activation function after running experiments where the tanh activation function was used on the last hidden layer; it does not give any discernible benefits and slows down the computation when training the model.

the model is trained by selecting a batch size of 16 and training for 1000 epochs; these hyperparameters were selected after running experiments with batch sizes of 8,16,24,32. The batch size of 16 and integrating to next 16th timestep to predict the position and angle of the domain wall is able to capture the non-linearities well.

After training the model is tested on three different test conditions specified below :

Number of Fields	Fields	Duration	Initial Position	Initial Angle
1	100.	100.	20.0	0.0
1	100.	100.	60.0	0.0
1	177.58	100.	0.0	D0.0

Table 3.5: Single Field Test Conditions

The Loss function when testing the model on the Test Set used is MSE loss, as Huber Loss is not as easy to interpret. The results and vector fields of predictions are discussed in the Analysis Section.

3.6.3 Domain Wall Multiple Fields

For the multiple field case, the following initial conditions were used to generate the simulation for training data:

Fields	Time Duration (s)	Position	Angle
253.39532837	6	0	0
20.1991129	6	0	0
311.00652871	6	0	0
6.54651878	6	0	0
2.71507284	6	0	0
214.04519583	6	0	0
263.01181776	6	0	0
60.59147089	6	0	0
67.90516097	6	0	0
459.52798364	6	0	0

Table 3.6: Multipl;e fields training simulation

For the multiple fields case, a combination of activation functions was used wherein the last hidden layer was subject to the tanh activation function and the rest to the LeakyReLu activation function. The performance was better than when using only LEaky ReLu or alternating the application of both LeakyReLu and tanh to the hidden layers.

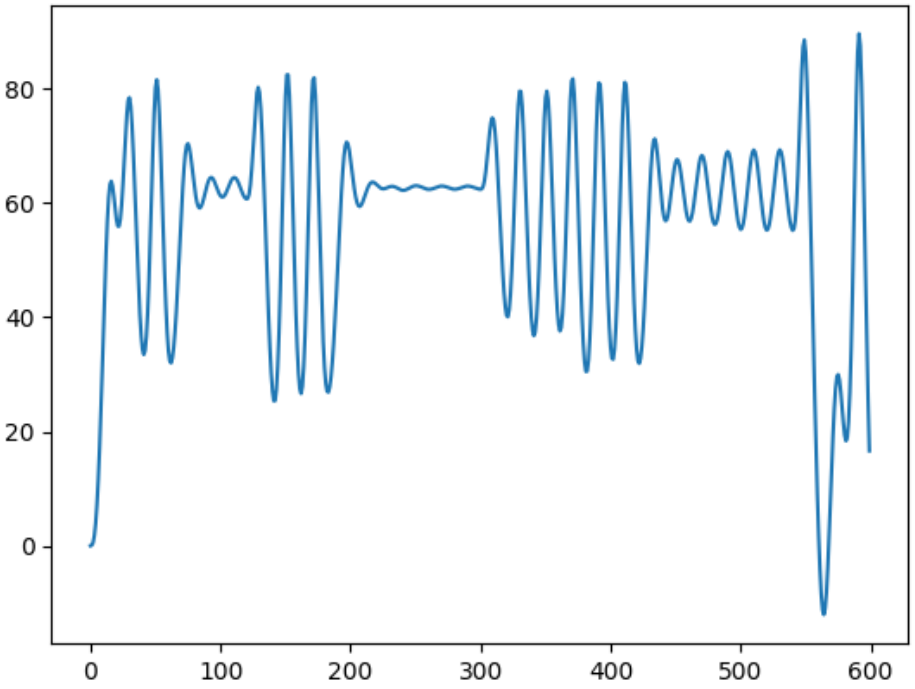
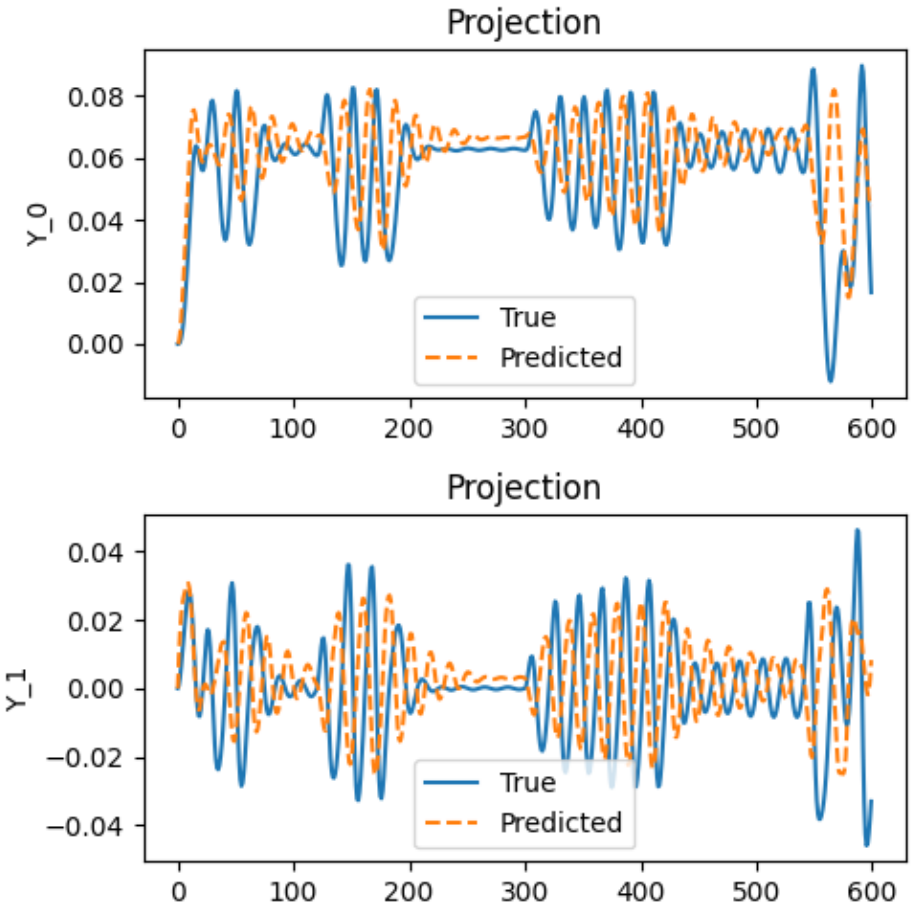


Figure 3.4: mutiple fields - Domain Wall position



(a) Multiple Fields -Domain Wall position

Figure 3.5: Training Multiple Fields

Chapter 4

Analysis & Results

This chapter discusses the findings of the experiments conducted with the approaches given in Chapter 3; the comparison and model evaluation is done using Mean Squared Error as the Metric for comparison, as well as the model's ability to forecast, generalize, and capacity of the model to capture the underlying dynamics of the system. The model's ability to learn the underlying dynamics of the system is evaluated primarily by plotting the vector fields of the predictions made by the model. [1]

4.1 Double Pendulum Analysis

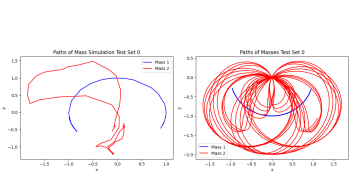
Here the three models - RNN, LSTM, and NODe are analyzed by their performance on the three test sets one by one using Error Metrics, visualizations, and vector fields.

Dataset	**MSE (Mean Squared Error)**		
	RNN	**LSTM**	**NODE**
Test1	12.17	1449.53	5.06
Test2	20.03	763.12	5.57
Test3	7.09	234.32	1.61

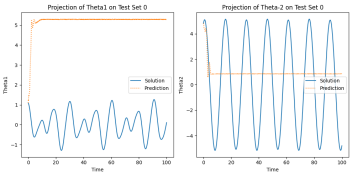
Table 4.1: Model Performance on Different Datasets Double Pendulum Problem

Out of the three models, NODE gives the lowest error on test sets; it also performs the best out of all three models, as will be evident by the figures below of the plots of the angles, paths of masses of the pendulum, and vector fields.

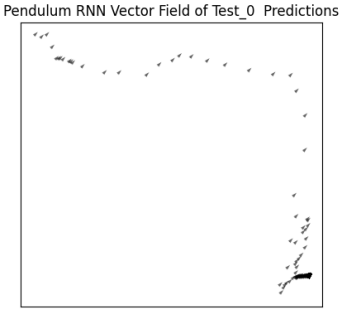
[1] Code and images available at Dissertation Repository



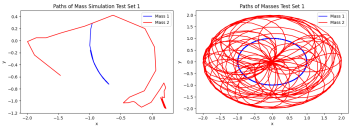
(a) Paths of Masses RNN test 1



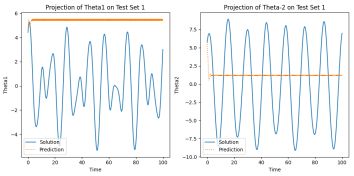
(b) RNN prediction Angles Test 1



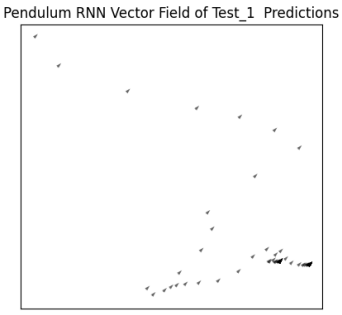
(c) Vector Field Rnn Test 1



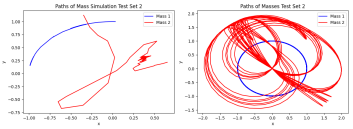
(d) Paths of Masses RNN test 2



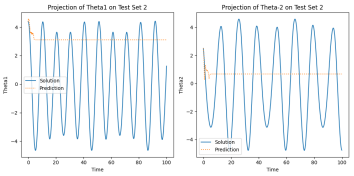
(e) RNN prediction Angles Test 2



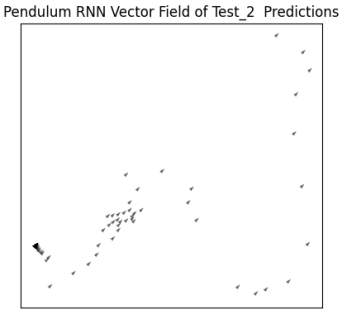
(f) Vector Field Rnn Test 2



(g) Paths of Masses RNN test 3



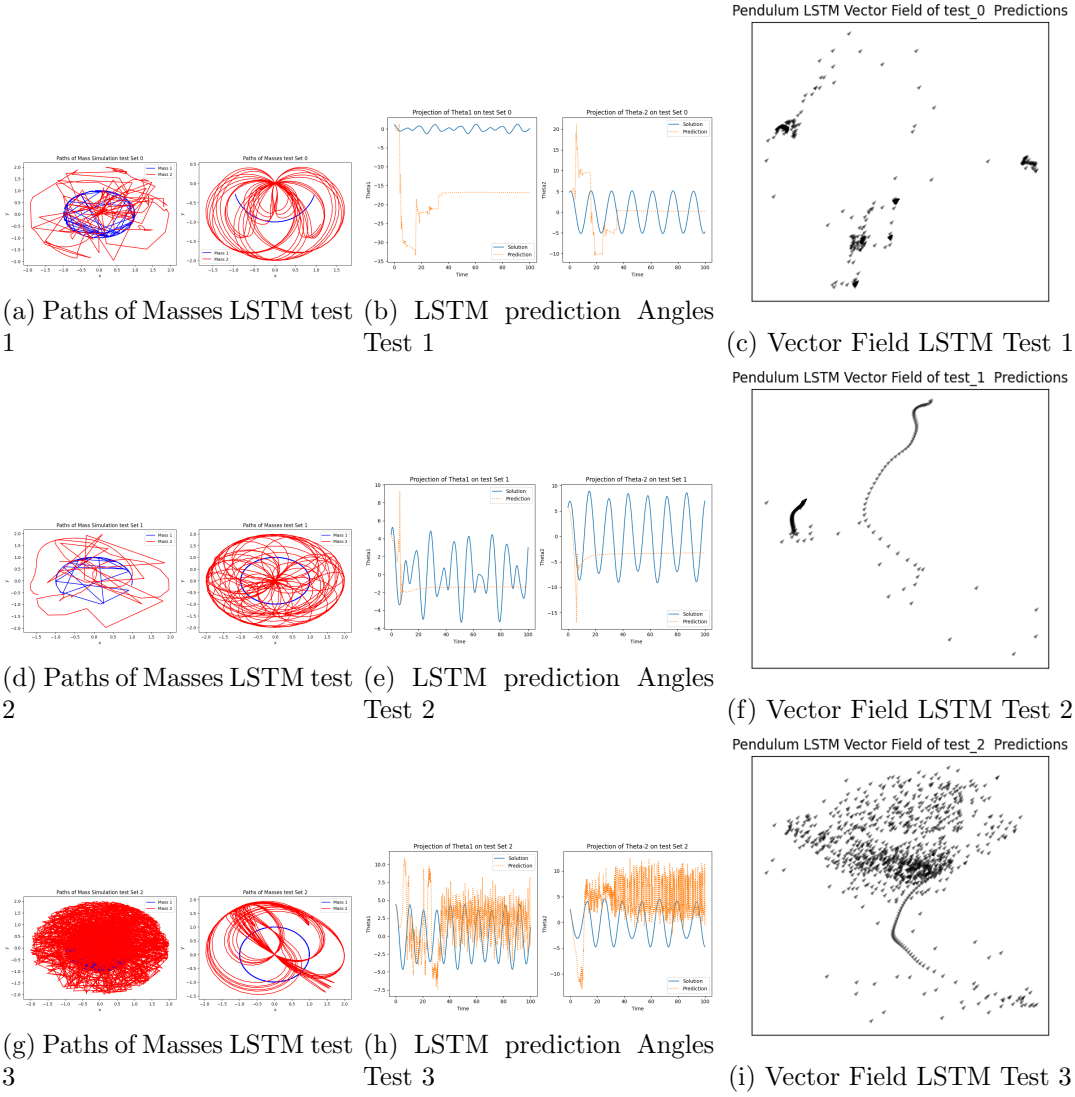
(h) RNN prediction Angles Test 3



(i) Vector Field Rnn Test 3

Figure 4.1: RNN performance on Double Pendulum Problem

RNN fails to perform on the test set, as is evidenced by the dismal performance on forecasting ability and generalizing capability on all three test sets. The vector fields are sparse and do not capture the dynamics of the Double pendulum well.



(a) Paths of Masses LSTM test 1

(b) LSTM prediction Angles Test 1

(c) Vector Field LSTM Test 1

(d) Paths of Masses LSTM test 2

(e) LSTM prediction Angles Test 2

(f) Vector Field LSTM Test 2

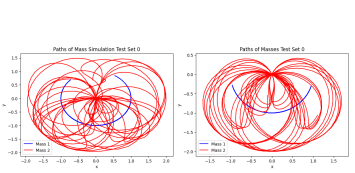
(g) Paths of Masses LSTM test 3

(h) LSTM prediction Angles Test 3

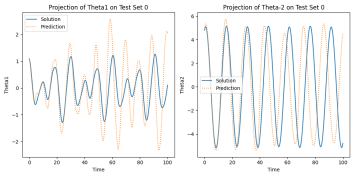
(i) Vector Field LSTM Test 3

Figure 4.2: LSTM performance on Double Pendulum Problem

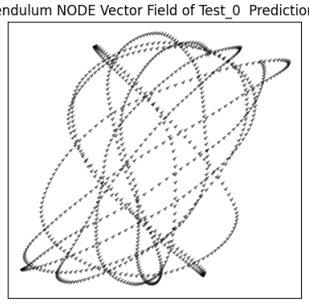
LSTM's predictions' vector fields are not as sparse as the RNN's ones are, but they still do not capture the dynamics of the Double Pendulum simulation. The high error is due to the predictions made by LSTM containing huge oscillations and outliers to test data.



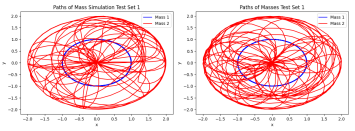
(a) Paths of Masses NODE test 1



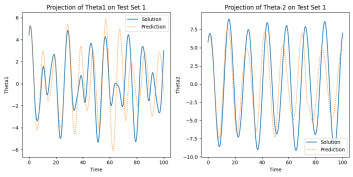
(b) NODE prediction Angles Test 1



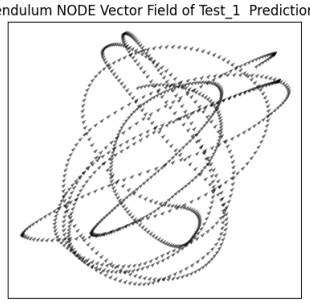
(c) Vector Field NODE Test 1



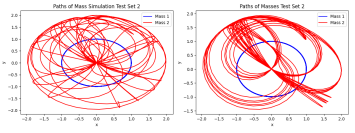
(d) Paths of Masses NODE test 2



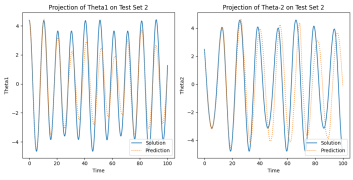
(e) NODE prediction Angles Test 2



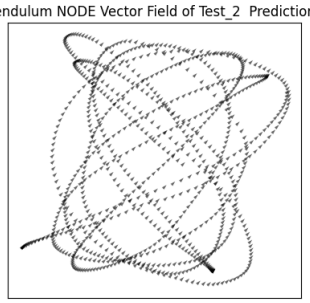
(f) Vector Field NODE Test 2



(g) Paths of Masses NODE test 3



(h) NODE prediction Angles Test 3



(i) Vector Field NODE Test 3

Figure 4.3: NODE performance on Double Pendulum Problem

NODE performs very well on the Double Pendulum problem as can be seen from the prediction's vector fields which mimic the dynamics of the test data so well. The NODE can be in phase when predicting θ_1 & θ_2 for the test set.

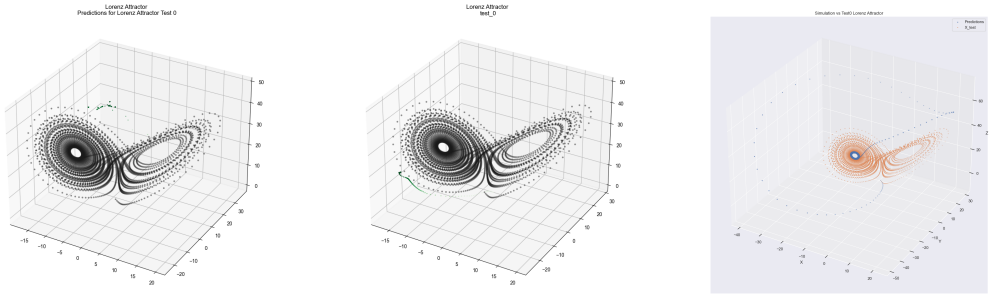
Overall RNN and LSTM fail miserably on the evaluation for forecasting and generalizability, NODE however can forecast and retain the dynamics well, maintinag a global stability , but diverges away from the test data after some initial steps, or is out of phase

4.2 Lorenz Attractor Analysis

For the First Test Set, the NODEs model gives the least MSE error, which is also reflected in the visualization of the predicted v/s test 1 data in the figure below.

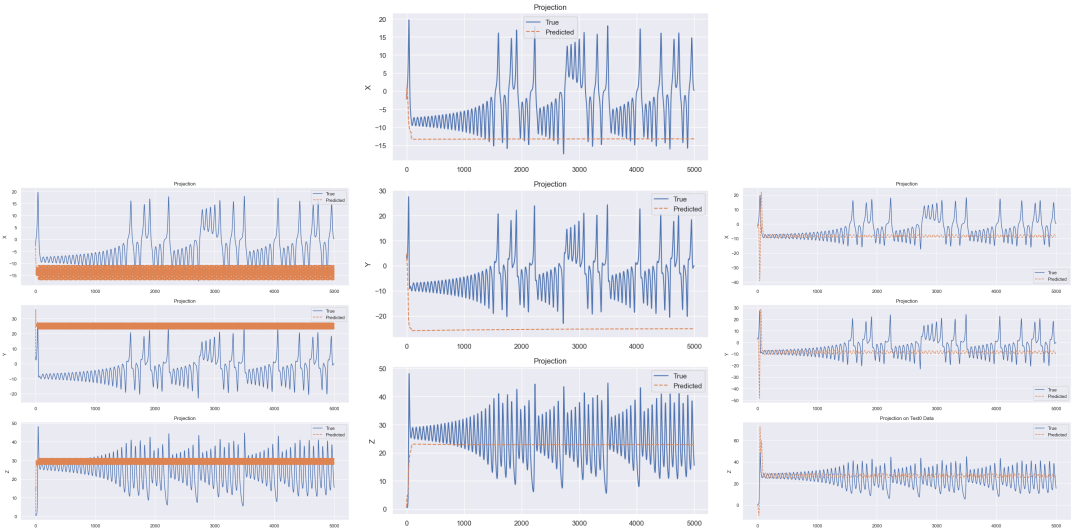
Models	MSE error
RNN	392.83
LSTM	243.47
NODE	86.63

Table 4.2: Comparison of Model Performance Lorenz Test 1



(a) RNN test 1 (b) LSTM test 1 (c) NODE test 1

Figure 4.4: Models performance on Test set 1



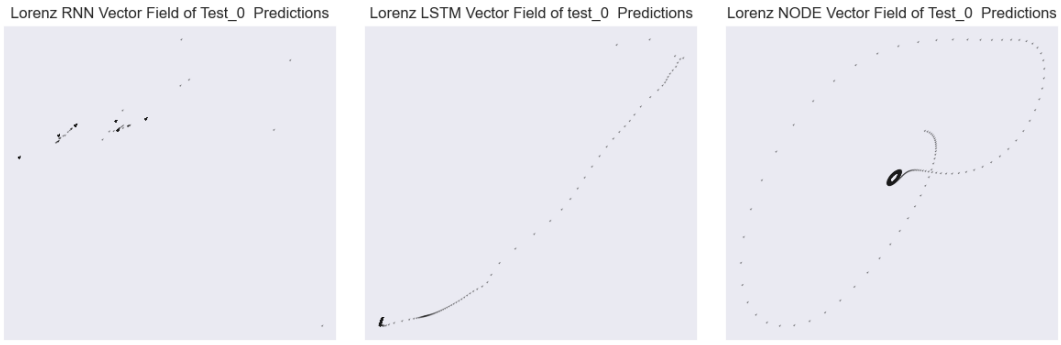
(a) RNN test 1 (b) LSTM test 1 (c) NODE test 1

Figure 4.5: Models performance on Test set 1 projection

The RNN and LSTM fail miserably to approximate any structure of the dynamics of

the Lorenz attractor; NODE also does not give a very accurate prediction but can at least approximate some dynamics of the test data as shown in the figure below, which shows the projection of predictions on the x,y, and z axis over the test 1 data.

Vector Field Analysis of Test Set 1 also shows how the vector fields of RNN's predictions and LSTM's predictions are very sparse, whereas NODE's prediction's vector field, while not very accurate of the dynamics, still approximates some dynamic structure of the test set.



(a) RNN vector field (b) LSTM vector field (c) NODE vector field

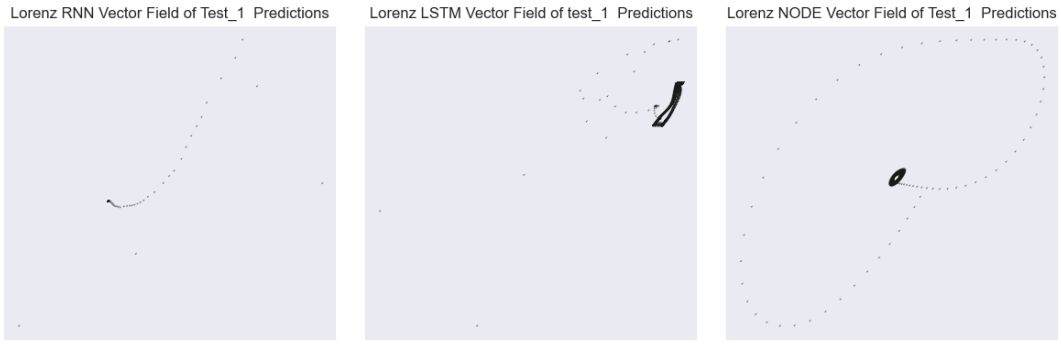
Figure 4.6: vector fields on Test set 1 projection

For the second test set, although the RNN gives the lowest error, it does not translate well into the prediction of the system, as evidenced by the figures below, where the predictions of RNN behave like the curve of exponential decay in projection figures

Models	**MSE error**
RNN	99.64
LSTM	465.43
NODE	134.89

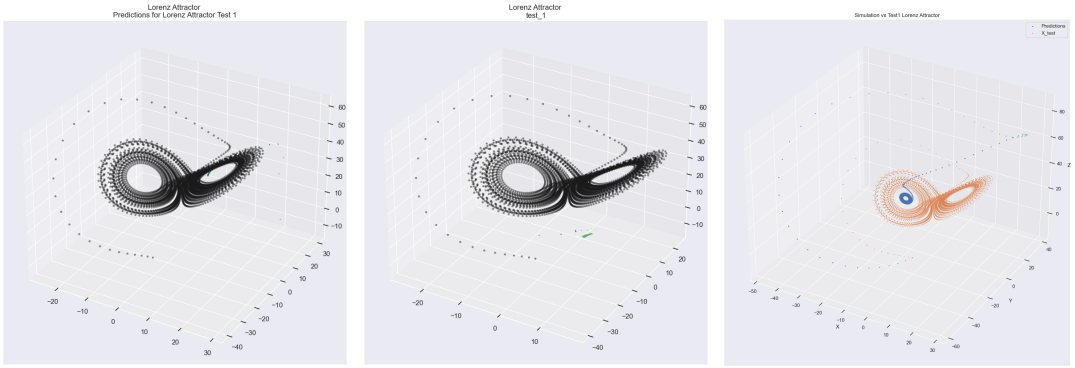
Table 4.3: Comparison of Model Performance Lorenz Test 2

Similarly, LSTM and NODE both make terrible predictions on test set 2 and capture very few dynamics in the vector fields; the NODE can only approximate some of the dynamics of the left half of the attractor.



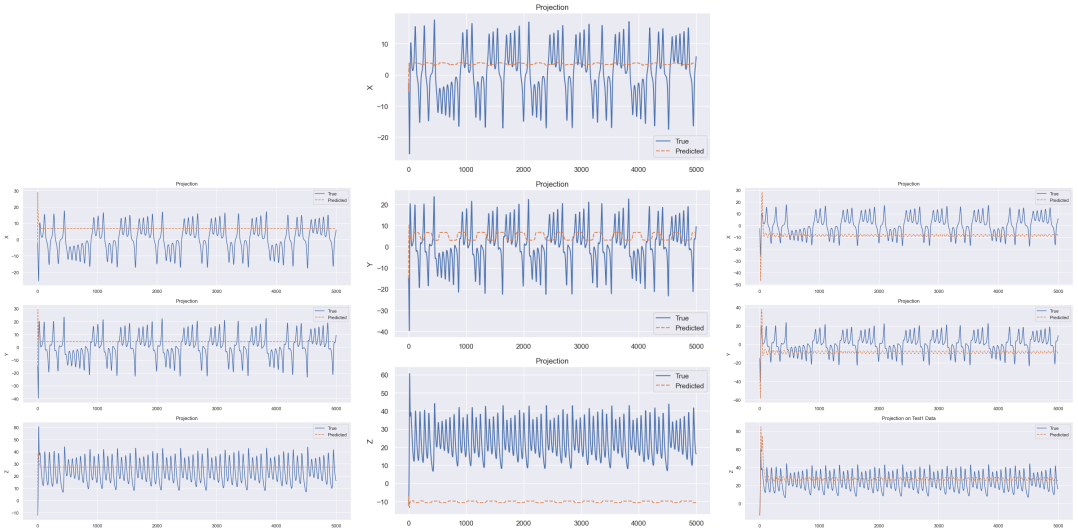
(a) RNN vector field (b) LSTM vector field (c) NODE vector field

Figure 4.7: vector fields on Test set 2 projection



(a) RNN test 2 (b) LSTM test 2 (c) NODE test 2

Figure 4.8: Models performance on Test set 2



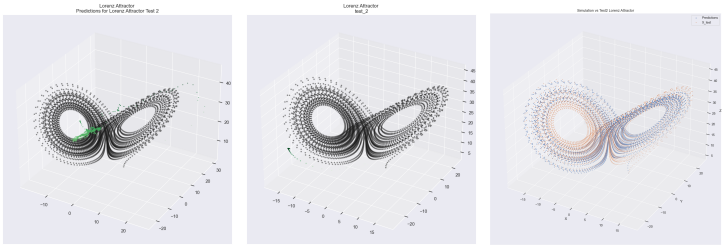
(a) RNN test 2 (b) LSTM test 2 (c) NODE test 2

Figure 4.9: Models performance on Test set 2 projection

For test set 3, We get the lowest MSE error from RNN followed by NODE, and again the lower error does not translate well into figures of the whole simulation, projections on x,y, and z axes or vector fields, where despite marginally higher error NODE can capture the dynamics well and is reflected in the vector field which shows it captures the dynamics of the test set 3 nicely. however, NODE also falls out of phase only after a few time steps.

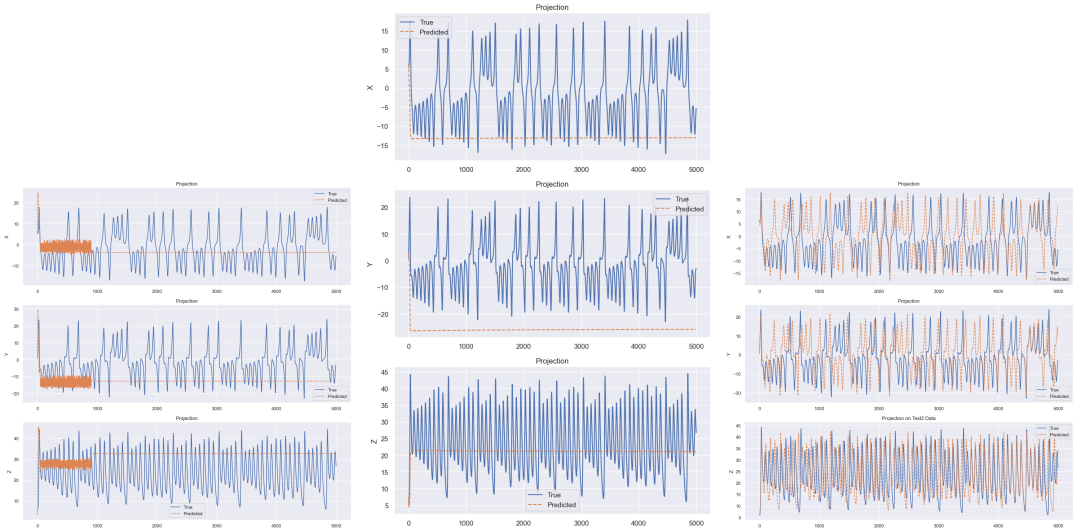
Models	**MSE error**
RNN	137.67
LSTM	308.76
NODE	140.37

Table 4.4: Comparison of Model Performance Lorenz Test 3



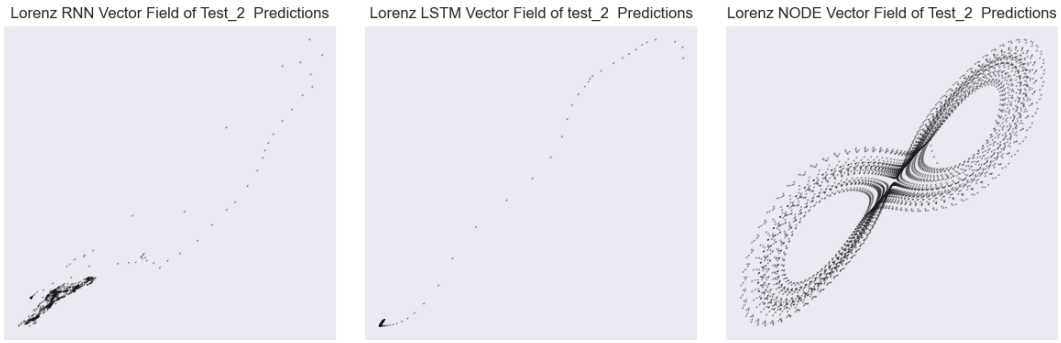
(a) RNN test 3 (b) LSTM test 3 (c) NODE test 3

Figure 4.10: Models performance on Test set 3



(a) RNN test 3 (b) LSTM test 3 (c) NODE test 3

Figure 4.11: Models performance on Test set 3 projection



(a) RNN vector field (b) LSTM vector field (c) NODE vector field

Figure 4.12: vector fields on Test set 3 projection

4.3 Magnetic Domain Wall Analysis

In this section, we analyze the results of the experiments done by implementing the NODE model on the two cases of the Domain Wall Problem, first when only a single magnetic field is given as input to observe the dynamics of the domain wall. Second, when multiple fields are fed as inputs to observe the dynamics of the Domain Wall simulation.

Three test sets are used for testing and evaluating the model for its forecasting and generalizability abilities, along with calculating the Mean Squared Error.

4.3.1 Single Field Case

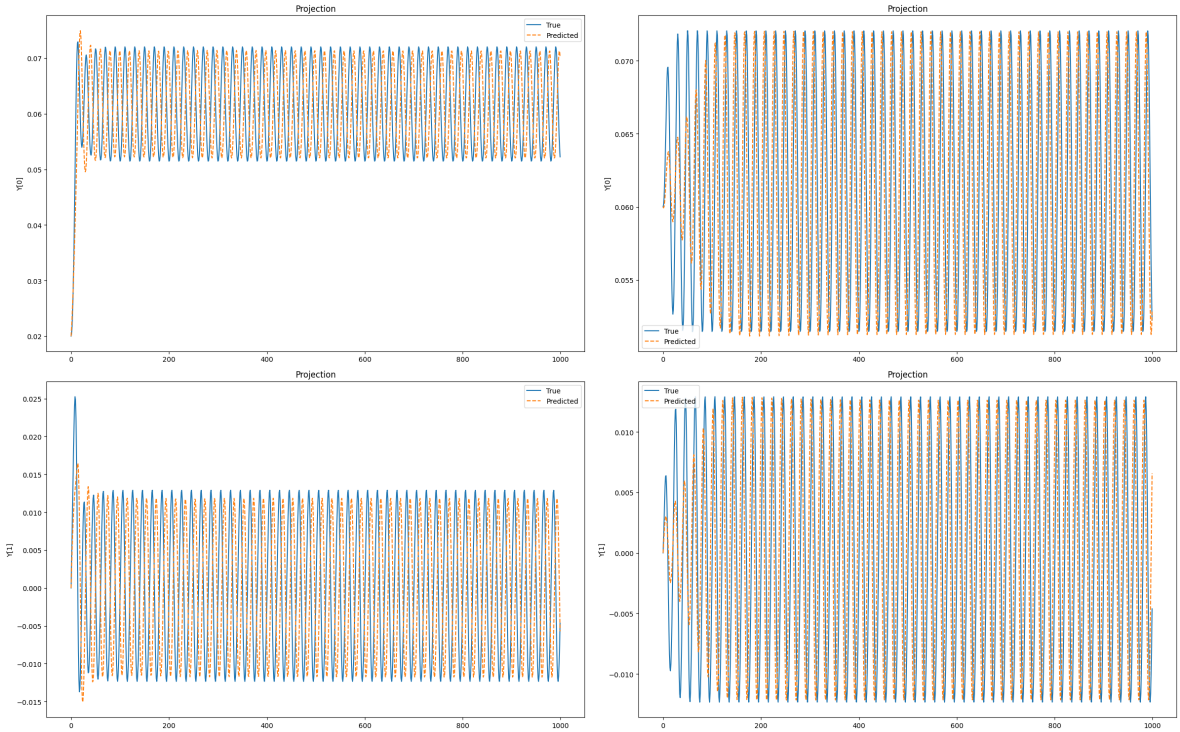
The following are the test conditions on which the model was tested, along with the MSE Loss on each set, followed by true data and vector fields of the predictions made by the model, and then analysis.

Number of Fields	Fields	Duration	Initial Position	Initial Angle	MSE Loss
1	100.	100.	20.0	0.0	0.00020
1	100.	100.	60.0	0.0	0.00021
1	177.58	100.	0.0	D0.0	0.00038

Table 4.5: Single Field Test Analysis

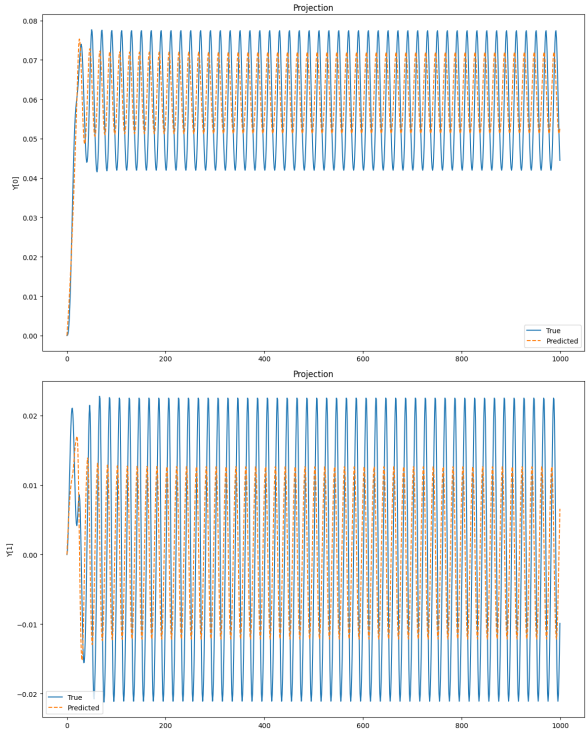
Forecasting Abilities The model can retain the dynamics beyond the time steps(1000 timesteps) it was trained upon(600 timesteps) and on different initial domain wall positions. However, on every test set, the performance is far from ideal, as in the first test set(Test Set 0), the prediction is slightly off phase from the beginning, as shown in the figures.

Generalisation Abilities The Generalisation ability of the model is also not perfect as it's out of phase in Test Set 0, while in the second test set (Test Set 1), it is not able to handle the different initial positions in the start as evident from the poor performance in the initial timesteps. The model also performs worse on the third test set (Test Set 2) where it is confined to only a part of the range of the real test data, as shown in the figures below.



(a) Test Set 1 predictions

(b) Test Set 2 predictions

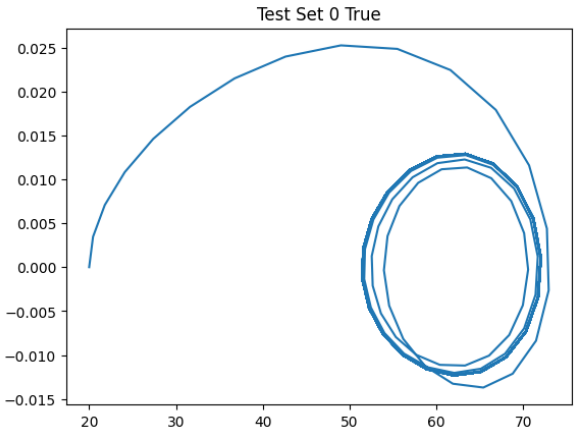


(c) Test Set 3 predictions

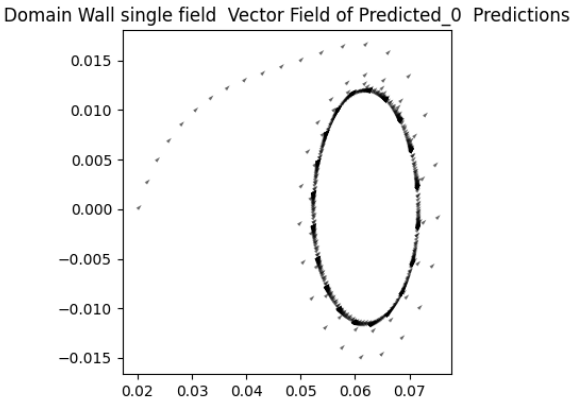
Figure 4.13: Predictions on Different Test Sets

Vector Fields Analysis

- As shown in the figure, the model is able to capture the dynamics well, as the vector field of the predictions made by the model is very similar to the phase portrait of the Test Set in the case of the first Test Set.

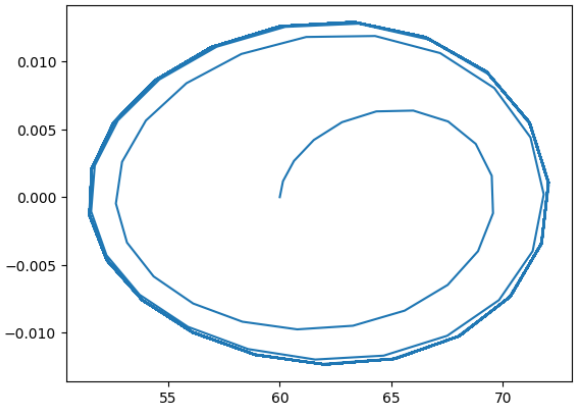


(a) SFDW Test Set 1 Phase Space

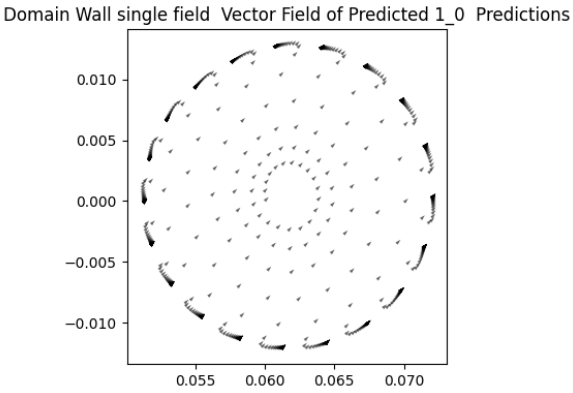


(b) NODE Predictions Vector Field test set 1

- In the case of the Second test set, it can be seen from the vector field of the predictions how it has much-constricted phase transitions in comparison to the phase space of test set 2, which is evident in the high error in the initial steps, but later on it was able almost to overlap the test set 2 data which is also reflected in the vector field with the outer circle settling in similarly to the phase space portrait.



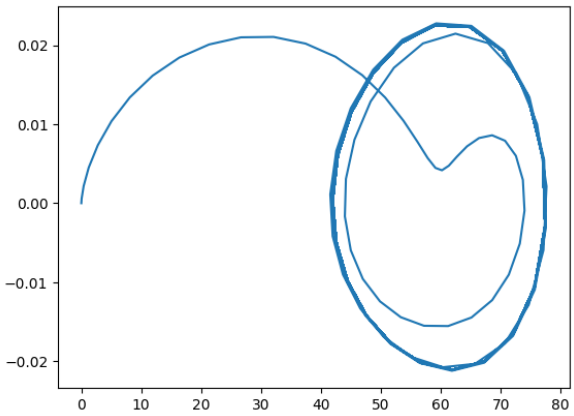
(a) SFDW Test Set 2 Phase Space



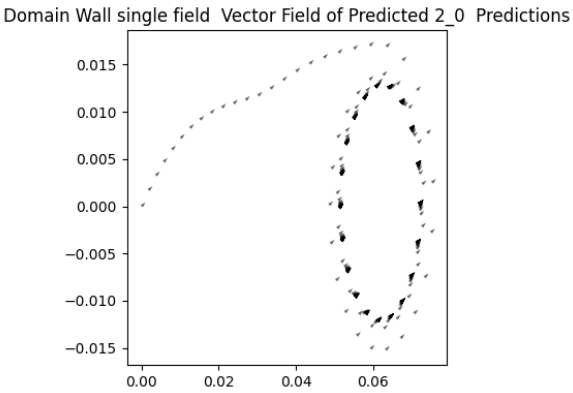
(b) NODE Predictions Vector Field test set 2

- the high MSE error and poor performance can also be reflected when comparing the vector field of predictions with the real test set 3 data, as the values in which the vector field of prediction is limited to a range of the phase space of the test data.

To conclude, For the Single Field Case, the model can learn the dynamics well when the test data contains the initial positions on the transient phase (before it settles into the



(a) SFDW Test Set 3 Phase Space



(b) NODE Predictions Vector Field test set 3

oscillatory phase), but performance is worse on different fields. The Single Field Case was used as the test bed for performing different runs to get optimal hyperparameters, such as batch size, number of epochs, etc.

4.3.2 Multiple Fields Case

For the multiple field case, the following were the test set 1 conditions:

Fields	Time Duration (s)
329.96166693	10.0
342.84893081	10.0
241.58321796	10.0
27.10567685	10.0
132.94038863	10.0
79.2643228	10.0
126.6714536	10.0
182.32972763	10.0
409.97049514	10.0
382.05819553	10.0

Table 4.6: Multiple Fields and Time Durations Test 1

Forecasting Abilities The model cannot retain the dynamics beyond the time steps it was trained upon and shows poor forecasting abilities on the multiple fields case.

Generalisation Abilities The Generalisation ability of the model also worsens in the Multiple Fields case significantly.

The initial position and angle were kept to zero for the first test set. The MSE loss on the test set was 0.0001, which does not translate well when we observe the predictions in the figures below; the model's predictions are limited to a limited range of the test data, but it captures the overall structure of dynamics reasonably well, as is evident when it can forecast

well the dynamics beyond the timesteps it was trained on. The vector field and phase space comparison also shows that the model cannot fully capture the complexity of the phase space into the vector fields of predictions.

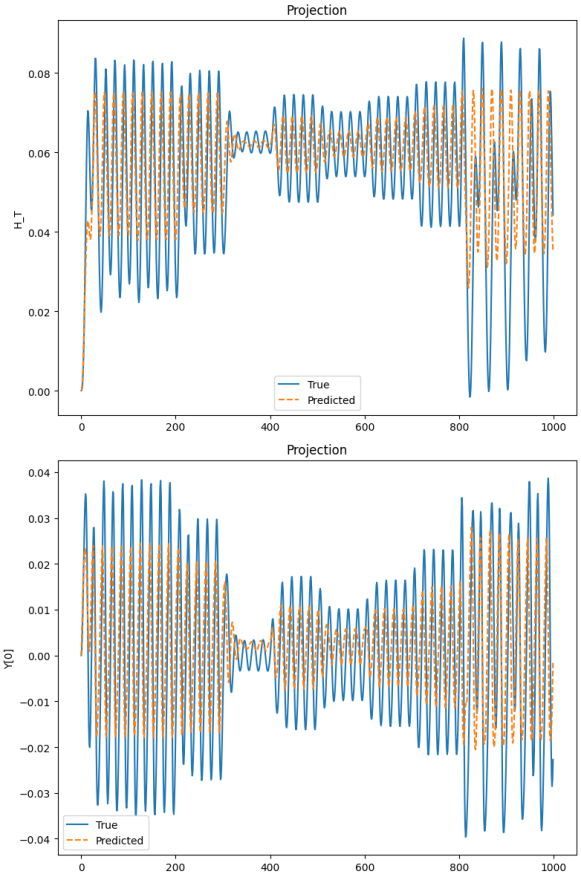
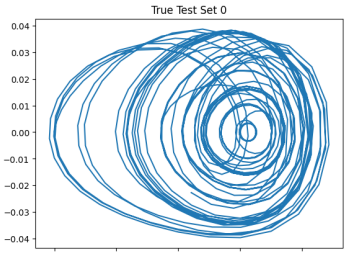
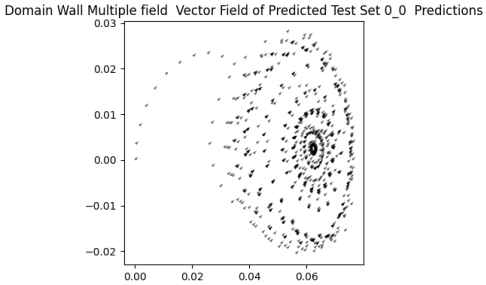


Figure 4.17: Test Set 1 prediction



(a) Test Set 2 phase space



(b) Test Set 1 prediction Vector Field

Figure 4.18: Test Set 1

For test set 2, again, the initial position and angle were kept to zeros, and the fields were as follows:

Fields	Time Duration (s)
144.94786709	10.0
298.91929209	10.0
486.3229457	10.0
57.64407563	10.0
173.45140805	10.0
190.65888406	10.0
492.39231278	10.0
361.11110949	10.0
171.57925701	10.0
48.30100152	10.0

Table 4.7: Multiple Fields and Time Durations Test 2

The MSE for this test set was 0.0008; we can observe the inability of the model to handle abrupt phase changes in the test data where the field magnitudes change abruptly from 298 to 486 to 57. This is also reflected in the vector field of the predictions, which is much simpler than the phase space of test set 2.

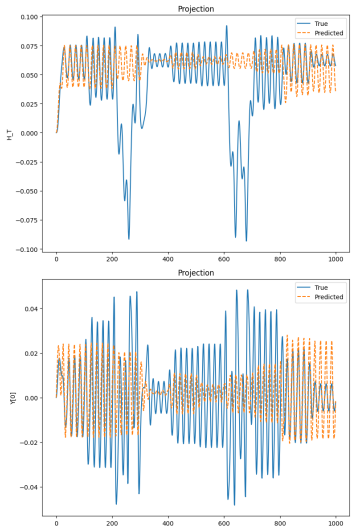
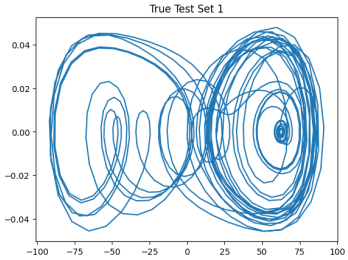
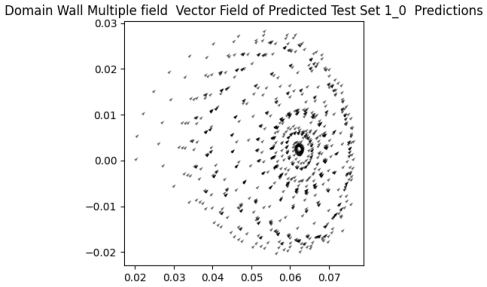


Figure 4.19: Test Set 2 prediction



(a) Test Set 2 phase space



(b) Test Set 2 prediction Vector Field

Figure 4.20: Test Set 2

For Test Set 3, the initial Domain Wall position was set to 20.0, and the angle was kept at 0. Following were the test set 3 conditions:

The MSE error for test set 3 was 0.0016, and again we can observe the model fails to

Fields	Time Duration (s)
368.39828591	10.0
437.63284412	10.0
140.76849796	10.0
10.75675635	10.0
478.17111307	10.0
465.64212293	10.0
413.51737694	10.0
481.89220302	10.0
22.12625991	10.0
40.6631365	10.0

Table 4.8: Multiple Fields and Time Durations Test 3

adapt to abrupt phase changes, and produces predictions whose vector field is a much simpler approximation of the test data.

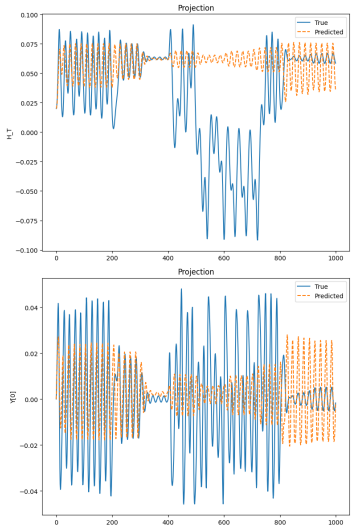
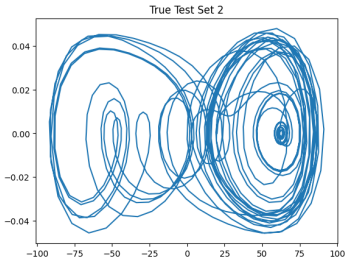
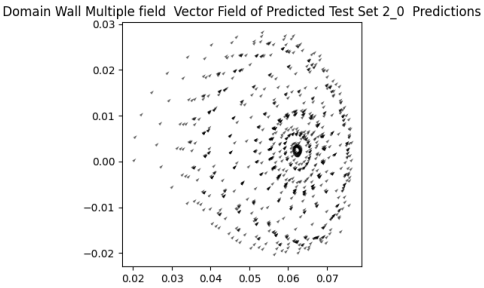


Figure 4.21: Test Set 3prediction



(a) Test Set 3phase space



(b) Test Set 3 prediction Vector Field

Figure 4.22: Test Set 3

To Conclude, the NODE model is an improvement over the vanilla RNNs and LSTM models, but it is still incapable of learning the chaotic systems satisfactorily. one persistent problem evidenced was that the model was always making exact predictions shifted by a small amount; this problem remains open for further investigation and analysis. The NODE architecture is critically analyzed in the next section, and future possible approaches are discussed later in the Future Work chapter.

4.4 Neural Ordinary Differential Equations Model Analysis

Here, the advantages, disadvantages, and limitations of this model are analyzed derived from the work by Patrick Kidger[19], Dupont et al. [12] and Norcliffe et al. [22], combined with the experience of applying the model to systems in this dissertation.

some of the advantages of Neural Ordinary Differential Equations(NODEs) model are:

- The first advantage is that they are solved using ODE Solvers, which are differentiable and allow us to select other hyperparameters such as step size, relative and absolute tolerance for errors, and the choice of the solver to make a decision when trading off between accuracy and computations used. This flexibility can make these viable for low-power devices for inference.
- The adjoint-based backpropagation[8] allows constant memory cost w.r.t network depth and scales linearly with the size of inputs. There is no need to store intermediate activations at each layer in the MLP architecture or for recalculation for backpropagation. In MLP, layers are not generally invertible, but in NODE, we can integrate backward with the adjoint sensitivity method.
- Established theoretical background of Differential Equations to build upon to have newer architectures such as Neural Controlled Differential Equations, Neural Stochastic Differential Equations(see kidger [19].
- Diverse Modelling Capabilities: NODEs can model continuous time series very well or problems that can become simpler when integrating for continuous time rather than discrete time, particularly physics problems.

There are also drawbacks of using the NODEs and limitations inherent in the architecture of NODEs, which limits its applicability :

- NODEs are not able to solve Stiff ODEs, and an attempt has been made by Ghosh et al. [15] to solve this, but it is still an open problem.
- Because the trajectories of ODe cannot cross each other, NODES must preserve the topology of its input space and limit the functions it can represent. NODEs are not universal approximations as shown in [19] and in [12]. A simple illustration is NODEs fail to learn the simple fine function because they cannot linearly separate the regions.
- Another drawback is that NODEs must solve for the whole trajectory[22], which may not always be optimal and expensive at inference times.
- Another drawback is NODEs when solving the ODE work on the assumption that the dynamics are deterministic, so they struggle when the system displays dynamic and chaotic behavior.
- The extra hyperparameters that give NODEs flexibility also make it more expensive to train as more hyperparameters must be optimized at training time.

Chapter 5

Conclusions

In this dissertation, First, the necessary theoretical knowledge about the tools and the systems that are being modeled was studied, then the machine learning models and "Architectures" such as RNN, LSTM, and NODE were studied and summarised in Chapter 2.

Then the design and implementation of the work were discussed in Chapter 3 wherein the choice for loss metrics and functions, the choice of activation functions, hyperparameters such as batch sizes, lookbacks for RNN and LSTM, solvers for NODE, etc. were searched, and analyzed, then using them to make predictions about the systems which were the Double Pendulum, Lorenz Attractor and the Domain Wall Problem.

In Chapter 4, the results of the experiments conducted were analyzed using loss metrics, visualizations, and vector fields to draw conclusions and find any patterns.

Based on the previous work in this dissertation, Following Conclusions may be drawn:

- Neural ODEs are a significant improvement over RNNs and LSTMs for modeling dynamical and chaotic systems over long time horizons. However, they are far away before they can be used to simulate these systems accurately.
- Neural ODEs as architecture have inherent limitations, which limit the functions they can learn or represent i.e., they are not universal approximations [19].
- The NODEs struggle to learn the dynamics of the domain wall, despite the fact that they are a coup[led first order differential equations, which it should have been able to learn.

In the next chapter, future approaches and pathways are discussed, which may provide a way for developing models that can learn the dynamical and chaotic systems in an interpretable way.

Chapter 6

Future Work

As discussed in previous sections, NODEs(Neural Ordinary Differential Equations) suffer from several limitations. This section discusses several approaches that address these limitations and are more suited to model the physical systems attempted. Implementation of these has not been attempted due to paucity of time.

6.1 Augmented Neural Ordinary Differential Equations(ANODEs)

As discussed in the Analysis section, NODEs are not universal approximators[19] as they must preserve the topology of input space, resulting in flows that are complex and computationally expensive to solve, limiting their applicability. Dupont et al. [12] show in their paper on ANODEs that simply augmenting the space on which the ODE is solved allows the model to use the additional dimensions to learn more complex functions using simpler flows. The hypothesis is that this makes the learned function smoother, giving rise to smoother flows that the ODE can solve in fewer steps. ANODE also provides better generalization, more stability, and lower losses.

We concatenate every data point x with more dimensions initialised at zeroes and solve the ODE on the augmented space[12].

$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{bmatrix}\mathbf{h}(t)\\ \mathbf{a}(t)\end{bmatrix} = \mathbf{f}\left(\begin{bmatrix}\mathbf{h}(t)\\ \mathbf{a}(t)\end{bmatrix}, t\right), \quad \begin{bmatrix}\mathbf{h}(0)\\ \mathbf{a}(0)\end{bmatrix} = \begin{bmatrix}\mathbf{x}\\ \mathbf{0}\end{bmatrix} \tag{6.1}$$

Where a(t) is an augmented part of the space.

6.2 Second Order Neural ODEs (SONODEs)

The issue with ANODEs is their interpretability, which can be important, particularly for problems associated with classical physics as pointed out by Kidger [19] although ANODEs are universal approximators, Norcliffe[22] points out issues with interpretability as ANODEs learn the dynamics through an abstract alternative ODE where the state and augmented

dimensions are entangled as shown below:

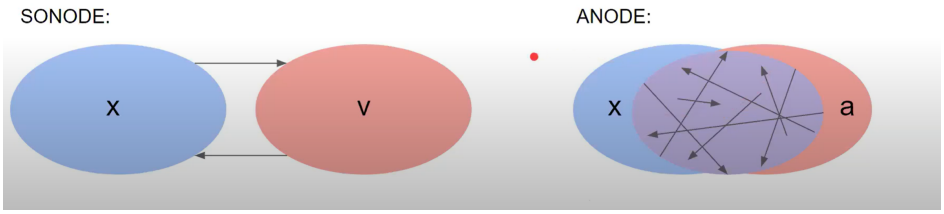


Figure 6.1: Separate dimensions in SONODE vs Entangled representation in ANODE[22]

One solution is to use Second order Neural Differential Equations(SONODE), which place a constraint on the ANODE to force it to learn the second-order dynamics.

$$\mathbf{x}(t_0) = \mathbf{X}_0, \qquad \dot{\mathbf{x}}(t_0) = g(\mathbf{x}(t_0), \theta_g), \qquad \ddot{\mathbf{x}} = f^{(a)}(\mathbf{x}, \dot{\mathbf{x}}, t, \theta_f), \tag{6.2}$$

where $\mathbf{x}(t_0)$ is initial position , $\dot{\mathbf{x}}(t_0)$ is initial velocity, $\ddot{\mathbf{x}}$ and acceleration, where $f^{(a)}$ is a neural network with parameters θ_f.

SONODE then can be thought of as ODEs defined by acceleration, not by velocity, and as a system of coupled first-order neural ODEs with state[22] $\mathbf{z}(t) = [\mathbf{x}(t), \mathbf{a}(t)]$:

$$\mathbf{z} = \begin{bmatrix} \mathbf{x} \\ \mathbf{a} \end{bmatrix}, \quad \dot{\mathbf{z}} = f^{(v)}(\mathbf{z}, t, \theta_f) = \begin{bmatrix} \mathbf{a} \\ f^{(a)}(\mathbf{x}, \mathbf{a}, t, \theta_f) \end{bmatrix}, \quad \mathbf{z}(t_0) = \begin{bmatrix} \mathbf{X}_0 \\ g(\mathbf{X}_0, \theta_g) \end{bmatrix}. \tag{6.3}$$

SONODEs give many advantages over ANODEs as they train faster on second-order systems, are robust to noise, and are interpretable, which makes a good approach as many classical physics problems are second-order ODEs. However, they cannot model higher order systems than second order and need double the dimensions of real space, in contrast to ANODEs, which can learn the dynamics with minimal augmentation where number of augmented dimensions is less than the dimensionality of real space[22].

6.3 Hamiltonian Neural Networks(HNN)

Another approach similar to Physics Inspired Neural Networks is the approach given by Greydanus et al.[16] of using Hamiltonian mechanics to give physics priors to neural networks so that they are constrained by conservation laws in an unsupervised manner. These models are perfectly reversible, giving memory advantages as there is no need to store intermediate activations for backpropagation. Also, the conserved quantity can be manipulated, allowing for rich experiments with simulations.

6.4 Lagrangian Neural Networks

This approach by Cranmer et al.[10] attempts to solve the basic problem with Neural Networks they struggle to learn basic symmetries and conservation laws. Similar to Hamiltonian Neural Networks, they apply Lagrangian mechanics to Neural Networks to enforce the conservation of total energy. The advantage they provide over Hamiltonian Neural Networks is that HNNs struggle to learn chaotic systems as they are restricted to canonical coordinates, whereas Lagrangian Neural Networks can use arbitrary coordinates, allowing the modeling of chaotic systems.

Bibliography

[1] Ababe, R. V., Ellis, M. O. A., Vidamour, I. T., Devadasan, D. S., Allwood, D. A., Vasilaki, E., and Hayward, T. J. Neuromorphic computation with a single magnetic domain wall. *Scientific Reports 11*, 1 (2021), 15587.

[2] Bergström, R., Lindroth, E., and Johansson, K. H. Modelling dynamical systems using neural ordinary differential equations. *arXiv preprint arXiv:2201.07966* (2022).

[3] Bloch, F. Zur theorie des ferromagnetismus. *Zeitschrift für Physik 61*, 3-4 (1930), 206–219.

[4] Boulle, O., Malinowski, G., and Kläui, M. Current-induced domain wall motion in nanoscale ferromagnetic elements. *Materials Science and Engineering: R: Reports 72*, 9 (2011), 159–187.

[5] Brownlee, J. An introduction to recurrent neural networks and the math that powers them, 2023.

[6] Butcher, J. C., and Prince, J. M. A fifth-order runge-kutta method with stepsize control. *Journal of Computational Physics 10*, 1 (1972), 1–17.

[7] Bézenac, E. *Modeling Physical Processes with Deep Learning: A Dynamical Systems Approach*. PhD thesis, 10 2021.

[8] Chen, R. T. Q., Rubanova, Y., Bettencourt, J., and Duvenaud, D. Neural ordinary differential equations, 2019.

[9] Chen, X., Araujo, F. A., Riou, M., Torrejon, J., Ravelosona, D., Kang, W., Zhao, W., Grollier, J., and Querlioz, D. Forecasting the outcome of spintronic experiments with neural ordinary differential equations. *Nature Communications 13*, 1 (2022), 1016.

[10] Cranmer, M., Greydanus, S., Hoyer, S., Battaglia, P., Spergel, D., and Ho, S. Lagrangian neural networks, 2020.

[11] Dobilas, S. Lstm recurrent neural networks — how to teach a network to remember the past, 12 2021.

[12] Dupont, E., Doucet, A., and Teh, Y. W. Augmented neural odes, 2019.

[13] Elman, J. L. Finding structure in time. *Cognitive Science 14*, 2 (1990), 179–211.

[14] Euler, L. *Introductio in analysin infinitorum*. Marc-Michel Bousquet, Lausanne, 1768.

[15] Ghosh, A., Behl, H. S., Dupont, E., Torr, P. H. S., and Namboodiri, V. Steer: Simple temporal regularization for neural odes.

[16] Greydanus, S., Dzamba, M., and Yosinski, J. Hamiltonian neural networks, 2019.

[17] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition, 2015.

[18] Hochreiter, S., and Schmidhuber, J. Long Short-Term Memory. *Neural Computation 9*, 8 (11 1997), 1735–1780.

[19] Kidger, P. On neural differential equations, 2022.

[20] Kutta, M. Beitrag zur näherungsweisen integration totaler differentialgleichungen. *Zeitschrift für Mathematik und Physik 46*, 3 (1901), 415–424.

[21] Lorenz, E. N. Deterministic nonperiodic flow. *Journal of Atmospheric Sciences 20*, 2 (1963), 130 – 141.

[22] Norcliffe, A., Bodnar, C., Day, B., Simidjievski, N., and Liò, P. On second order behaviour in augmented neural odes, 2020.

[23] Néel, L. Théorie du traînage magnétique des ferromagnétiques en grains fins avec applications aux terres cuites. *Annales de géophysique 5*, 2 (1948), 99–136.

[24] Schmidhuber, J. Annotated history of modern ai and deep learning, 2022.

[25] Strogatz, S. H. *Nonlinear Dynamics and Chaos: With Applications to Physics, Biology, Chemistry and Engineering*. Westview Press, 2000.

[26] Teets, D., and Whitehead, K. The discovery of ceres: How gauss became famous. *Mathematics Magazine 72*, 2 (1999), 83–93.

[27] Werbos, P. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE 78*, 10 (1990), 1550–1560.

Appendices

Appendix A

Code for Simulation

A.1 Domain Wall Simulation

```
import numpy as np

class DW:
    def __init__ (self, Ms, A, alpha, L, a, b, hconst, f, htime = None):
        self.mu_0 = np.pi*4e-7
        self.Ms = Ms
        self.A = A
        self.alpha = alpha
        self.L = L
        self.a = a
        self.b = b
        self.hconst = hconst
        self.Bconst = self.hconst * self.mu_0
        self.f = f
        self.omega = 2*np.pi*self.f

        self.htime = htime

        # cross section
        self.S = L[1]*L[2]

        r1 = L[1]/L[2]
        r2 = L[2]/L[1]
        Ny = 0.0884731  #1 - (2*np.arctan(r1) + 0.5*r2*np.log10(1 + r1**2) - 0.5*r1*np.log1
        Nz = 0.911527 #1 - (2*np.arctan(r2) - 0.5*r2*np.log10(1 + r1**2) + 0.5*r1*np.log10(1
        self.Bk = self.mu_0*Ms*(Nz - Ny)
```

```
        self.X = 0.0
        self.phi = 0.0

        self.delta = self.DW_width(self.phi)

        self.ap = - 1e-9*self.a /(self.Ms*self.S)
        self.bp = -2*(1e-27)*self.b /(self.Ms*self.S)
        self.gamma = 1.76e2 # in per nanosecond per Tesla
        self.beta = self.gamma/(1+self.alpha**2)

    def DW_width(self, phi):
        dw = 1e9*np.pi*np.sqrt( 2*self.A / (self.mu_0*self.Ms**2*np.sin(phi)**2 + self.Ms*se
        return dw

    def fields(self, x, phi, t):
        Bx = self.Bapp(t) + self.ap * x + self.bp * x**3
        Bphi = -0.5*self.Bk*np.sin(2*phi)
        return Bx, Bphi

    def Bapp(self, t):
        if self.htime is not None:
            return (self.Bconst + self.mu_0 * self.htime(t))* np.sin(self.omega*t)
        else:
            return self.Bconst * np.sin(self.omega*t)

    def Happ(self, t):
        return self.Bapp(t)/self.mu_0

    def Epin(self, X):
        return self.a * X**2 + self.b*X**4

    def xfield(self, x, t):
        return self.Bapp(t) + self.ap * x + self.bp * x**3

    def set_hconst(self, hconst):
        self.hconst = hconst
        self.Bconst = self.hconst * self.mu_0
        return
```

```
def DW_EoM(t, y, params):
    """
    DW oscillator equation of motion

    Parameters
    ----------
    t  :  time
    y  :  array containing the DW position and angle
    params  :  DW class object containing all the parameter functions

    Returns
    ----------
    gradient : array of equation of motion
    """
    x, phi = y
    Bx, Bphi = params.fields(x, phi, t)
    dx = params.DW_width(phi)*params.beta*(params.alpha * Bx - Bphi)
    dphi = params.beta*(params.alpha*Bphi + Bx)
    return [dx, dphi]




class field_sequence:
    """
    callable class to produce a time dependent field sequence
    """
    def __init__ (self, fields, periods):
        import numpy as np

        self.fields = fields
        self.periods = periods
        self.periods_sum = np.cumsum(periods)

    def __call__ (self, t):
        if t < 0.0:
            val = 0.0
        elif t >= self.periods_sum[-1]:
            val = 0.0
        else:
            t_diff = self.periods_sum - t
```

```
            n = 0
            for i in range(len(t_diff)):
                if t_diff[i] >= 0.0:
                    n = i
                    break
            val = self.fields[n]
        return val


def run_field_sequence(field_low = 0.0, field_high = 1000.0, N_fields = 10, T = 4, dt = 0.1,
    from scipy.integrate import solve_ivp
    rng = np.random.default_rng()
    fields = rng.uniform(field_low, field_high, N_fields)
    periods = np.ones(len(fields))*T
    total_time = np.sum(periods)
    print(fields)
    print(periods)

    htime = field_sequence(fields, periods)

    dw1 = DW(477e3, 1.05e-11, 0.02, (600e-9, 50e-9, 5e-9), -1.28e-6, 1.63e8, 0.0, 0.5, htime

    t_eval = np.arange(0, total_time, dt)
    sol = solve_ivp( DW_EoM, [0, total_time], y0, args=[dw1], t_eval=t_eval)

    h_vals = np.zeros_like(sol.t)
    for i in range(len(h_vals)):
        h_vals[i] = dw1.Happ(t_eval[i])

    return sol.t, sol.y, h_vals, fields, periods
```

A.2 Sequence function for Gradient of Magnetic Field

```
class sequence:
    """
    Callable class to produce a time-dependent field sequence using PyTorch tensors.
    """
    def __init__(self, fields, periods):
        self.fields = torch.tensor(fields, dtype=torch.float64).to(device)
        self.periods = torch.tensor(periods, dtype=torch.float64).to(device)
        self.periods_sum = torch.cumsum(self.periods, dim=0)
```

```
    def __call__(self, t):
        if t < 0.0:
            val = torch.tensor(0.0, dtype=torch.float64)
        elif t >= self.periods_sum[-1]:
            val = torch.tensor(0.0, dtype=torch.float64)
        else:
            t_diff = self.periods_sum - t
            n = 0
            for i in range(len(t_diff)):
                if t_diff[i] >= 0.0:
                    n = i
                    break
            #n = torch.argmax(t_diff >= 0.0)  # Find the first index where t_diff >= 0.0
            val = self.fields[n]
        return val
```