SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master Thesis

Training Dynamic Neural Networks

Karan Shah

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master Thesis

Training Dynamic Neural Networks

Entschlüsselung der Topologie der dynamische neuronale Netze

Author:	Karan Shah
Supervisor:	Dr. rer. nat. Felix Dietrich
Advisor:	Chinmay Datar, M.Sc.
Submission Date:	15.01.2024

I confirm that this master thesis is my own work and I have documented all sources and material used.

Munich, 15.01.2024 Karan Shah

Acknowledgments

Thank you Mom, Dad and Kavita, for your unconditional love and support.

Thank you Chinmay and Felix, for your invaluable guidance and timely inputs throughout the course of this thesis.

Abstract

Dynamical models estimate and predict the temporal evolution of physical systems. State space systems, a fundamental framework in control theory and system dynamics, provide a powerful mathematical representation for capturing the evolving states of such dynamical systems over time. Data driven techniques e.g. Recurrent Neural Networks have emerged as compelling approaches to model these state space parameters with wide success across a number of challenging tasks, in part due to their impressive capability to extract time-varying features from inputs. They however lack interpretability and training them has historically been very difficult. This master thesis implements a novel network architecture named Dynamic Neural Networks (DNNs), first proposed in [1], whose key idea is to solve a system of ordinary differential equations to compute the states of neurons. Leveraging the dynamic characteristics of these blocks, DNNs are specifically designed for tasks related to sequence modeling and system identification. The backpropagation behavior of the linear dynamical operator is defined w.r.t. both its parameters and input sequence, which facilitates comprehensive end-to-end training of these networks. We will test our model on illustrative examples to demonstrate the efficacy of the proposed methodology. The mapping from the system matrices to the parameters of the DNN also eliminates the need for neural architecture search. We also do a rigorous study of the loss landscape and incorporate our findings into the learning paradigm.

Kurzfassung

Dynamische Modelle schätzen und prognostizieren die zeitliche Entwicklung physikalischer Systeme. Zustandsraumsysteme, ein grundlegender Rahmen in Kontrolltheorie und -system Dynamik bieten eine leistungsstarke mathematische Darstellung zur Erfassung der sich entwickelnden Zustände solcher dynamischer Systeme im Laufe der Zeit. Datengesteuerte Techniken, z.B. Rekurrente neuronale Netze haben sich als überzeugende Ansätze zur Modellierung dieser Zustandsraumparameter mit großem Erfolg bei einer Reihe anspruchsvoller Aufgaben herausgestellt, unter anderem aufgrund ihrer beeindruckenden Fähigkeit, zeitlich variierende Merkmale aus Eingaben zu extrahieren. Allerdings mangelt es ihnen an Interpretierbarkeit, und ihre Ausbildung war in der Vergangenheit sehr schwierig. Diese Masterarbeit implementiert eine neuartige Netzwerkarchitektur namens Dynamic Neural Networks (DNNs), die erstmals in [1] vorgeschlagen wurde und deren Schlüsselidee darin besteht, ein System gewöhnlicher Differentialgleichungen zu lösen, um die Zustände von Neuronen zu berechnen. DNNs nutzen die dynamischen Eigenschaften dieser Blöcke und sind speziell für Aufgaben im Zusammenhang mit der Sequenzmodellierung und Systemidentifikation konzipiert. Das Backpropagation-Verhalten des linearen dynamischen Operators ist bzgl. sowohl der Parameter als auch der Eingabesequenz, was ein umfassendes End-to-End-Training dieser Netzwerke ermöglicht. Wir werden unser Modell anhand anschaulicher Beispiele testen, um die Wirksamkeit der vorgeschlagenen Methodik zu demonstrieren. Durch die Zuordnung von den Systemmatrizen zu den Parametern des DNN entfällt auch die Notwendigkeit einer neuronalen Architektursuche. Wir führen auch eine gründliche Untersuchung der Verlustlandschaft durch und integrieren unsere Erkenntnisse in das Lernparadigma.

Contents

1. Introduction

1.1. Motivation, Literature Survey and Goals

State space modeling has been a cornerstone in understanding and representing the dynamics of complex systems. Accurate parameter estimation is pivotal for effective model performance. Traditionally, methods like Maximum Likelihood Estimation (MLE) and Bayesian approaches have been employed. However, recent years have witnessed a paradigm shift with the integration of theoretically-informed Artificial Neural Networks (ANNs) into the parameter estimation process. In this section, we will discuss a comprehensive overview of this integration, exploring its historical evolution, methodologies, applications, challenges, and future directions.

The history of parameter estimation for state space models dates back to the mid-20th century with the introduction of the Kalman filter by Rudolf E. Kalman [2]. While the Kalman filter addressed the state estimation problem, accurate parameter estimation remained a challenge. Subsequent developments included Maximum Likelihood Estimation (MLE) and Bayesian approaches [3]. These methods laid the foundation for accurate parameter estimation, but their applicability faced limitations in handling non-linearities and high-dimensional data.

The advent of Artifical Neural Networks marked a significant departure from traditional approaches, offering a powerful tool to capture complex and non-linear relationships within dynamical systems. Early works by Chen and Billings (1990) demonstrated the potential of neural networks, specifically feedforward networks, in capturing non-linear system dynamics [4]. The ability of deep learning models to automatically extract hierarchical features from raw data makes them particularly suitable for capturing the complex dynamics of real-world systems. Brunton et al. (2016) demonstrated the utility of neural networks in discovering governing equations from fluid flow data, showcasing the broad applicability of these techniques in understanding complex systems [5].

However, it became evident that the temporal dependencies inherent in many dynamic systems necessitated the use of recurrent neural networks (RNNs) and long short-term memory networks (LSTMs) [6]. Although these networks provided a more sophisticated means of modeling system dynamics, they suffered from the typical problems of vanishing and exploding gradients and just general difficulties in the training paradigm. A Recurrent Neural Network (RNN) can be considered as a network of Multi-layered Perceptrons (MLPs) enhanced by feedback connections. RNNs are considered a cornerstone in the learning theory because of their abilities to reproduce dynamical behaviors by mean of feedback connections and delays

in the propagation of their signals [7]. After learning, a RNN with a sufficient number of neurons is able to estimate any relationships and therefore to reproduce the behavior of any multivariate and nonlinear dynamical systems (Werbos, 1974). Therefore, RNNs received a considerable attention from the modern control community to such an extend that they have been formalized in Model-Referencing Adaptive Control (MRAC) schemes. Their model stability remains one of the most critical aspects.

Peter Meijer developed NEUREKA [1]. - an application that uses DNNs for simulating electrical circuits by their respective reduced-order models. They propose a methodology for constructing efficient models for nonlinear, multi-dimensional, dynamical systems, given input-output data. However, in the proposed methodology, unavailability of the topology of the DNNs beforehand makes the architecture search very computationally expensive. Moreover, local minima are found during the optimization of the parameters.

Prof. Wil Schilders' work [8]. tackles these drawbacks for linear systems. They first apply state-space modelling MOESP algorithm [5], [6] for identifying a reduced-order Linear Time-Invariant (LTI) system from input-output data. Starting from this state-space model, they find a mapping from the state-space matrices to the parameters i.e. the number of neurons and hidden layers in DNNs, thereby eliminating the need for architecture search. Based on this mapping, they also propose a strategy to initialise weights of DNNs to ensure a good initial guess. However, the Bartels-Stewart algorithm used in this mapping requires the state matrix's eigenvalues to have an algebraic multiplicity equal to one.

In our work, we implement the ideas developed in [8]. and propose a way to relax this algebraic multiplicity constraint. Given any LTI system describing the underlying dynamics (either obtained from input-output data using a subspace identification algorithm like MOESP or a known mathematical model), we will determine the number of layers and neurons in each layer of the DNN. We will also try to deduce how these neurons are connected to each other based on the structure of the state space matrices. Lastly, we will come up with a strategy to initialise the weights of the DNN using the domain knowledge in the form of the reduced state-space model. The goal of this thesis is to implement and study how one can train the dynamic neural networks using reverse-mode differentiation and to test the implementation on some numerical examples.

1.2. Thesis Organization

The thesis is structured as follows. Chapter 2 highlights the theoretical background on dynamical systems. We will start with an historical overview of dynamical systems before deep diving into the mathematics of State Space modelling. We will then briefly study Linear Time-Invariant Systems. In Chapter 3, we introduce the concept of a Dynamical Neural Network (DNN). We talk about their formalism, intuition behind the architecture, math required to go from the state space matrices to the parameters of the DNN. We then talk

about the training paradigm: the Forward Pass and the Backpropagation. We also visualize the loss landscape. In Chapter 4, we analyse the results and list our findings. In Chapter 5, we conclude our work with some remarks and talk about the possible next steps.

2. Background

2.1. Overview of Dynamical Systems

The exploration of dynamical systems as a mathematical discipline can be traced back to the late nineteenth century, notably with Henri Poincaré's contributions to celestial mechanics [9]. Poincaré's work focused on formulating equations to describe the movements of planets within a gravitational field, seeking solutions to predict their positions over time. When the complexity or impossibility of finding explicit solutions arises, attention shifts to studying the mathematical structure of the model. This approach, emphasizing understanding the nature and structure of equations for insights into possible solution functions, forms the foundation of what we now recognize as dynamical systems theory.

The concept of dynamical systems, though relatively new, encompasses a broad scope, extending beyond a mere subdiscipline of real analysis. While it primarily investigates the properties of functions with a single real independent variable, dynamical systems draw theoretical and methodological influences from diverse mathematical domains, including analysis, geometry, topology, and algebra. These interconnected influences place dynamical systems within the realm of second-generation mathematics, acting as bridges between various pure areas of mathematical study. In its essence, dynamical systems serve as a comprehensive study of phenomena modeled through functions and equations, evolving over time. Defining this concept precisely requires acknowledging its generality, even if the initial definition may seem broad and less immediately informative.

So, as a means to define this concept more precisely, we begin with arguably a most general and yet least helpful statement:

A dynamical system is a mathematical formalization for any fixed rule that describes the dependence of the position of a point in some ambient space on a parameter. [10]

The parameter here, usually referred to as "time" due to its reference to applications in the sciences, takes values in the real numbers. Usually, these values come in two varieties:
(1) discrete (think of the natural numbers $\mathbb{N}$ or the integers $\mathbb{Z}$), or
(2) continuous (defined by some single interval in $\mathbb{R}$).

The parameter can sometimes take values in much more general spaces, for instance, subsets of $\mathbb{C}$,$\mathbb{R}$, the quaternions, or indeed any set with the structure of an algebraic group.

The ambient space exhibits a "state" characterized by marked positions of all its points that vary with the parameter. Each point possesses a relative position to others, often defined by a complete set of generalized coordinates providing a well-defined notion of position. The term "generalized" originates from classical mechanics, referring to coordinates defining configurations relative to a reference. By fixing the coordinate system and allowing parameter variation, a functional relationship arises between points at different parameter values. This concept involves the notion of topology, providing a mathematical property of space, defining nearness among set elements, and enabling functions on the set to possess properties like continuity and differentiability. Referred to as the state space, it encompasses all potential states a dynamical system can inhabit at any given time.

The governing rule, typically a guideline for transitioning between states in the specified parameter order, is often presented as a function in discrete dynamical systems. This function, mapping the state space to itself, moves each point to its subsequent state through iteration. Past states may be determined by applying the inverse of the function or selecting an element from the set that the function maps to the current state. Continuous systems, posing challenges in defining a successor to a parameter value, employ Ordinary Differential Equations (ODEs) to describe the continuous movement of points in a space [11]. In continuous systems, the ODE plays a role equivalent to the function in discrete systems, implicitly defining the method of transitioning from one state to the next, albeit in an infinitesimal manner. The solution to an ODE (or system of ODEs) is a function whose domain includes points from both the state and parameter space (sometimes termed the trajectory space), returning values to the state space. This function, often called the system's evolution, facilitates the transition from any initial state to another state reachable via a parameter value. While the existence of such a function can be demonstrated, and its properties studied, it is seldom known beforehand or even ascertainable afterward.

Before embarking on a more systematic exploration of dynamical systems, here is another less rigorous definition of a dynamical system: Dynamical Systems as a field of study attempts to understand the structure of a changing mathematical system by identifying and analyzing the things that do not change.

There are many ways to identify and classify this notion of an unchanging quantity amidst a changing system. But the general idea is that if a quantity within a system does not change while the system as a whole is evolving, then that quantity holds a special status as a symmetry. Identifying symmetries can allow one to possibly locate and identify solutions to an ODE. Or one can use a symmetry to create a new system, simpler than the previous, where the symmetry has been factored out, reducing either the number of variables and/or the size of the system. The general goal of an analysis of a dynamical system is typically not to solve the system or to find an explicit expression for its evolution. Many nonlinear systems of ODEs are difficult, if not impossible, to solve. Rather, the goal of an analysis of a dynamical system is a general description of the movement of points under the map or the

ODE. Perhaps the best way to end this section is on a more philosophical note, and allow a possible raison d'etre for why Dynamical Systems even exists as a field of study enmeshed in the world of analysis, topology, and geometry: It is the study of the information contained in and the effects of groups of transformations of a space [10].

2.2. Examples of Dynamical Systems

Celestial Dynamics:
The motion of celestial bodies in space is governed by dynamical systems. Kepler's laws and Newton's law of gravitation describe the orbits of planets around the sun and the motion of moons around planets. These systems involve complex interactions and gravitational forces that contribute to the stability of our solar system.

Climate Systems:
Climate models utilize dynamical systems to simulate and predict atmospheric and oceanic behavior. The Navier-Stokes equations govern fluid dynamics in the Earth's atmosphere and oceans, and these models help researchers understand climate patterns, predict weather events, and study long-term climate change.

Epidemiology:
The spread of infectious diseases within a population can be modeled using dynamical systems. Epidemiological models, such as the SIR (Susceptible-Infectious-Removed) model, help predict the progression of diseases like influenza or COVID-19. These models consider factors such as transmission rates and recovery rates.

Traffic Flow:
The movement of vehicles on roads and highways is a complex dynamical system. Traffic flow models, such as the Lighthill-Whitham-Richards model, help urban planners and engineers optimize traffic signal timings and manage congestion. Understanding traffic dynamics is essential for efficient transportation systems.

Biological Systems:
Biological systems, from cellular processes to ecosystems, involve intricate dynamical interactions. Cellular signaling pathways, gene regulatory networks, and ecological systems can be modeled using dynamical systems to understand how populations of species evolve over time or how cells respond to external stimuli.

Financial Markets:
Financial markets are dynamic and unpredictable, making them ideal candidates for dynamical systems analysis. Stock prices, market indices, and economic indicators can be modeled to understand trends, forecast market movements, and develop trading strategies.

Chaos theory is often applied to study the seemingly random behavior of financial markets.

Aerospace Engineering:
The design and control of aircraft and spacecraft rely on dynamical systems theory. Flight dynamics models describe the motion of aircraft through the air, and spacecraft trajectories are governed by celestial mechanics. Control systems play a vital role in ensuring stable and precise movements.

Robotics: The motion planning and control of robotic systems involve dynamical systems. Robot dynamics models consider factors such as joint angles, velocities, and external forces to achieve accurate and efficient movements. This is crucial in applications ranging from manufacturing to healthcare.

Electrical Circuits:
Electrical circuits are classic examples of dynamical systems. They involve the flow of electric current through components such as resistors, capacitors, and inductors. The behavior of these circuits is governed by differential equations that describe how voltages and currents change over time. For instance, the simple RC circuit involves a resistor (R) and capacitor (C) and exhibits time-dependent behavior in response to input signals.

2.3. History of Dynamical Systems Theory

Many people regard French mathematician Henri Poincaré as the founder of dynamical systems [12]. Poincaré published two now classical monographs, "New Methods of Celestial Mechanics" (1892–1899) and "Lectures on Celestial Mechanics" (1905–1910). In them, he successfully applied the results of their research to the problem of the motion of three bodies and studied in detail the behavior of solutions (frequency, stability, asymptotic, and so on). These papers included the Poincaré recurrence theorem, which states that certain systems will, after a sufficiently long but finite time, return to a state very close to the initial state.

Aleksandr Lyapunov developed many important approximation methods. His methods [13], which he developed in 1899, make it possible to define the stability of sets of ordinary differential equations. He created the modern theory of the stability of a dynamical system.

In 1913, George David Birkhoff proved Poincaré's "Last Geometric Theorem", a special case of the three-body problem, a result [14] that made him world-famous. Birkhoff's most durable result has been his 1931 discovery of what is now called the ergodic theorem. Combining insights from physics on the ergodic hypothesis with measure theory, this theorem solved, at least in principle, a fundamental problem of statistical mechanics. The ergodic theorem has also had repercussions for dynamics.

Time G (semigroup)	Actions
Natural numbers $(N, +)$	Maps
Integers $(Z, +)$	Invertible Maps
Positive real numbers $(R^+, +)$	Semiflows (some PDEs)
Real numbers $(R, +)$	Flows (Differential equations)
Any group $(G, \star)$	Representations
Lattice $(Z^n, +)$	Lattice gases, Spin systems
Euclidean space $(R^n, +)$	Tiling dynamical systems
Free group $(F_n, \circ)$	Iterated function systems

Table 2.1.: Classes of Dynamical Systems

Stephen Smale made significant advances as well. His first contribution was the Smale horseshoe [15] that jumpstarted significant research in dynamical systems. He also outlined a research program carried out by many others.

Oleksandr Mykolaiovych Sharkovsky developed Sharkovsky's theorem [16] on the periods of discrete dynamical systems in 1964. One of the implications of the theorem is that if a discrete dynamical system on the real line has a periodic point of period 3, then it must have periodic points of every other period.

In the late 20th century the dynamical system perspective to partial differential equations started gaining popularity. Palestinian mechanical engineer Ali H. Nayfeh applied nonlinear dynamics in mechanical and engineering systems [17]. His pioneering work in applied nonlinear dynamics has been influential in the construction and maintenance of machines and structures that are common in daily life, such as ships, cranes, bridges, buildings, skyscrapers, jet engines, rocket engines, aircraft and spacecraft.

2.4. State Space Representation

Mathematically, a dynamical system consists of a phase (or state) space P and a family of transformations $\phi_t : P \to P$, where the time t may be either discrete, $t \in Z$, or continuous, $t \in R$ [18]. For arbitrary states $x \in P$ the following must hold:

1. $\phi_0(x) = x$ **(identity)**
2. $\phi_t(\phi_s(x)) = \phi_{t+s}(x) \;\; \forall t, \;\; s \in R$ **(additivity)**

A dynamical system may be understood as a mathematical prescription for evolving the state of a system in time [19]. Additivity ensures that the transformations $\phi(t)$ form an Abelian group.

In other words, any semigroup G acting on a set is a dynamical system. A semigroup $(G, \star)$ is a set G on which we can add two elements together and where the associativity law $(x \star y) \star z = x \star (y \star z)$ holds. The action is defined by a collection of maps T_t on X. It is assumed that $T_{t \star s} = T_t \circ T_s$, where $\star$ is the operation on G (usually addition) and $\circ$ is the composition of maps. In the table 2.1, we can see the different classes of dynamical systems.

In control engineering and system identification, a state-space representation is a mathematical model of a physical system specified as a set of input, output and variables related by first-order (not involving second derivatives) differential equations or difference equations. Such variables, called state variables, evolve over time in a way that depends on the values they have at any given instant and on the externally imposed values of input variables. Output variables' values depend on the values of the state variables and may also depend on the values of the input variables.

The state space or phase space is the geometric space in which the variables on the axes are the state variables. The state of the system can be represented as a vector, the state vector, within state space.

If the dynamical system is linear, time-invariant, and finite-dimensional, then the differential and algebraic equations may be written in matrix form [20]. The state-space method is characterized by significant algebraization of general system theory, which makes it possible to use Kronecker vector-matrix structures. The capacity of these structures can be efficiently applied to research systems with modulation or without it [21]. The state-space representation (also known as the "time-domain approach") provides a convenient and compact way to model and analyze systems with multiple inputs and outputs. With p inputs and q outputs, we would otherwise have to write down $q \times p$ Laplace transforms to encode all the information about a system. Unlike the frequency domain approach, the use of the state-space representation is not limited to systems with linear components and zero initial conditions.

2.5. Linear Time-invariant Systems

Linear time-invariant systems (LTI systems) are a class of systems that are both linear and time-invariant. Linear systems are systems whose outputs for a linear combination of inputs are the same as a linear combination of individual responses to those inputs. Time-invariant systems are systems where the output does not depend on when an input was applied. The constraints of linearity and time-invariance; these terms are briefly defined below:

Linearity means that the relationship between the input $x(t)$ and the output $y(t)$, both being regarded as functions, is a linear mapping: If a is a constant then the system output to $ax(t)$ is $ay(t)$; if $x'(t)$ is a further input with system output $y'(t)$ then the output of the system to $x(t) + x'(t)$ is $y(t) + y'(t)$, this applying for all choices of $a, x(t), x'(t)$. The latter

condition is often referred to as the superposition principle.

Time invariance means that whether we apply an input to the system now or T seconds from now, the output will be identical except for a time delay of T seconds. That is, if the output due to input $x(t)$ is $y(t)$, then the output due to input $x(t-T)$ is $y(t-T)$. Hence, the system is time invariant because the output does not depend on the particular time the input is applied.

The equations relating the current state of a system to its most recent input and past states are called the state equations, and the equations expressing the values of the output variables in terms of the state variables and inputs are called the output equations. he state equations and output equations for a linear time invariant system can be expressed using coefficient matrices: A, B, C, and D, such that $A \in R^{N\times N}, B \in R^{N\times L}, C \in R^{M\times N}, D \in R^{M\times L}$, where N, L and M are the dimensions of the vectors describing the state, input and output, respectively.

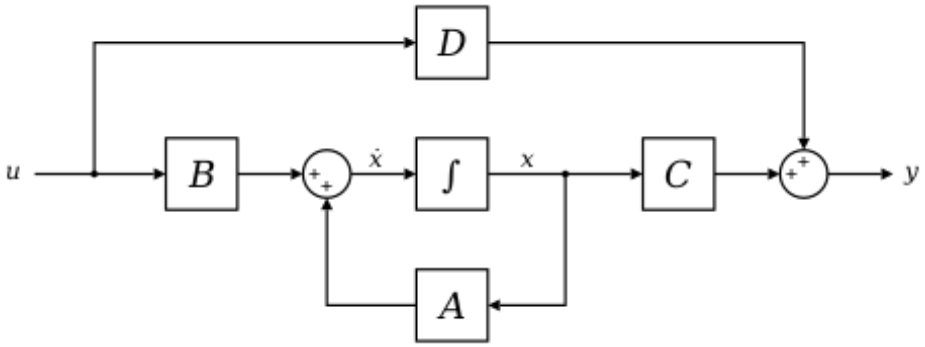

Figure 2.1.: A typical State Space Model of a LTI system

The most general state-space representation of a linear system with p inputs, q outputs and n state variables is written in the following form:
$x'(t) = Ax(t) + Bu(t)$
$y(t) = Cx(t) + Du(t)$
where:

$x(\cdot)$ is called the "state vector", $x(t) \in R^n$;
$y(\cdot)$ is called the "output vector", $y(t) \in R^q$;
$u(\cdot)$ is called the "input vector", $u(t) \in R^p$;
$A(\cdot)$ is called the "state (or system) matrix", $dim[A(\cdot)] = n \times n$;
$B(\cdot)$ is called the "input matrix", $dim[B(\cdot)] = n \times p$;
$C(\cdot)$ is called the "output matrix", $dim[C(\cdot)] = q \times n$;
$D(\cdot)$ is called the "feedthrough (or feedforward) matrix", $dim[D(\cdot)] = q \times p$;
$x'(t) := \frac{d}{dt}\mathbf{x}(t)$

2.6. Control and Observability

Control is the ability to manipulate the behavior of a system in order to achieve desired performance or specifications. In control theory, the goal is often to design a controller that can adjust the system's inputs to produce a desired output or to regulate the system in the presence of disturbances.

The state controllability condition implies that it is possible – by admissible inputs – to steer the states from any initial value to any final value within some finite time window. A continuous time-invariant linear state-space model is controllable if and only if

$$\operatorname{rank}\begin{bmatrix}\mathbf{B} & \mathbf{AB} & \mathbf{A}^2\mathbf{B} & \cdots & \mathbf{A}^{n-1}\mathbf{B}\end{bmatrix} = n,$$

where *rank* is the number of linearly independent rows in a matrix, and where n is the number of state variables.

Observability is the ability to reconstruct the internal state of a system from its outputs. In other words, it is the ability to determine the state of the system by observing its outputs over time. It can be defined as a measure for how well internal states of a system can be inferred by knowledge of its external outputs.

A continuous time-invariant linear state-space model is observable if and only if

$$\operatorname{rank}\begin{bmatrix}\mathbf{C} \\ \mathbf{CA} \\ \vdots \\ \mathbf{CA}^{n-1}\end{bmatrix} = n$$

The observability and controllability of a system are mathematical duals (i.e., as controllability provides that an input is available that brings any initial state to any desired final state, observability provides that knowing an output trajectory provides enough information to predict the initial state of the system).

2.7. Transfer Functions

The transfer function of a Linear Time-Invariant (LTI) system is a mathematical representation that describes the relationship between the system's input and output in the frequency domain. It is a fundamental concept in control theory and signal processing.

For a single-input, single-output (SISO) LTI system, the transfer function $H(s)$ is defined as the ratio of the Laplace transform of the system's output $Y(s)$ to the Laplace transform of the system's input $U(s)$ under zero initial conditions:

$$H(s) = \frac{Y(s)}{U(s)}$$

Here, s is the complex frequency variable, and $Y(s)$ and $U(s)$ are the Laplace transforms of the output and input, respectively.

The transfer function is typically expressed as a rational function in s:

$$H(s) = \frac{N(s)}{D(s)}$$

where $N(s)$ and $D(s)$ are polynomials in s, and the degree of $D(s)$ is greater than or equal to the degree of $N(s)$.

The coefficients of the transfer function are determined by the system's parameters and dynamics. For a system described by linear differential equations, the transfer function provides a concise representation of its behavior in the frequency domain.

The transfer function is a powerful tool for analyzing and designing control systems. It allows engineers to study the system's response to different frequencies and design controllers based on desired performance specifications. The transfer function is also useful for stability analysis, frequency response analysis, and the design of filters in signal processing applications.

Let's consider the LTI system described above with the following state space representation:

$$x' = Ax + Bu$$
$$y = Cx + Du$$

Let $\mathcal{L}$ denote the Laplace transform operator.

Laplace Transform of State Equation:

$$\begin{aligned} \mathcal{L}\{\dot{x}\} &= \mathcal{L}\{Ax + Bu\} \\ sX(s) - x(0) &= AX(s) + BU(s) \\ (sI - A)X(s) &= x(0) + BU(s) \\ X(s) &= (sI - A)^{-1}x(0) + (sI - A)^{-1}BU(s) \end{aligned}$$

Substituting into Output Equation:

$$\begin{aligned} y &= CX \\ Y(s) &= CX(s) + DU(s) \end{aligned}$$

Substituting the expression for $X(s)$ from the state equation:

$$Y(s) = C(sI - A)^{-1}x(0) + C(sI - A)^{-1}BU(s) + DU(s)$$

Transfer Function:

$$H(s) = \frac{Y(s)}{U(s)} = C(sI - A)^{-1}B + D$$

So, the transfer function of the LTI system described by $\dot{x} = Ax + Bu$ and $y = Cx + Du$ is given by:

$$H(s) = C(sI - A)^{-1}B + D$$

2.8. Ordinary Differential Equations (ODEs)

A differential equation is an equation for an unknown function that contains not only the function but also its derivatives. In general, the unknown function may depend on several variables and the equation may include various partial derivatives. The subset of all differential equations for such a function of just a single real variable are called ordinary differential equations — shortly ODE. A most general ODE has the form:

$$F(x, y, y', \ldots, y^{(n)}) = 0$$

where F is a given function of $n + 2$ variables and $y = y(x)$ is an unknown function of a real variable x. The maximal order n of the derivative $y(n)$ in the equation above is called the order of the ODE

Solving ODEs numerically is the common approach to exploring the behavior of dynamical systems and often the simplest one to solve - so simple, in fact, that sometimes you may find a closed form for the solution to the equations, but this is not always the case. Numerical methods, such as Euler's method or Runge-Kutta methods, enable the approximation of solutions, facilitating the study of systems that may not have analytical solutions.

Numerical methods for solving first-order IVPs often fall into one of two large categories: [22] linear multistep methods, or Runge–Kutta methods. A further division can be realized by dividing methods into those that are explicit and those that are implicit. For example, implicit linear multistep methods include Adams-Moulton methods, and backward differentiation methods (BDF), whereas implicit Runge–Kutta methods [23] include diagonally implicit Runge–Kutta (DIRK) [24], singly diagonally implicit Runge–Kutta (SDIRK)[25], and Gauss–Radau [26] (based on Gaussian quadrature [27]) numerical methods. Explicit examples from the linear multistep family include the Adams–Bashforth methods, and any Runge–Kutta method with a lower diagonal Butcher tableau is explicit. A loose rule of thumb dictates that stiff differential equations require the use of implicit schemes, whereas non-stiff problems can be solved more efficiently with explicit schemes.

Forward Euler Method:

The Forward Euler method is a simple and explicit approach. It approximates the derivative at each time step using the slope at the current point. The update step is given by:

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

where h is the step size, t_n is the current time, y_n is the current approximation, and $f(t_n, y_n)$ is the derivative function.

Forward Euler is computationally efficient but can suffer from stability issues. It is known to amplify high-frequency components and may lead to numerical instability in stiff systems.

Backward Euler Method:

The Backward Euler method is implicit, requiring the solution of nonlinear equations at each time step. The update step is given by:

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

Backward Euler is unconditionally stable for linear problems but may lack accuracy for stiff systems. The implicit nature allows for larger time steps in comparison to Forward Euler, but each step involves solving nonlinear equations, which can be computationally demanding.

Trapezoidal Method:

The Trapezoidal method is a compromise between the explicit Forward Euler and implicit Backward Euler. It averages the slopes at the current and next points. The update step is given by:

$$y_{n+1} = y_n + \frac{h}{2} \cdot [f(t_n, y_n) + f(t_{n+1}, y_{n+1})]$$

Trapezoidal method is conditionally stable for linear problems and offers better accuracy than Forward Euler. It strikes a balance between stability and accuracy but still may struggle with stiffness. The implicit nature adds computational cost but is less than fully implicit methods like Backward Euler.

Comparing the different methods:

1. **Accuracy:**
   - Forward Euler: First-order accuracy.
   - Backward Euler: First-order accuracy.
   - Trapezoidal: Second-order accuracy.
2. **Stability:**
   - Forward Euler: Conditionally stable; may be unstable for stiff problems.
   - Backward Euler: Unconditionally stable for linear problems.
   - Trapezoidal: Conditionally stable; better than Forward Euler.
3. **Computational Cost:**
   - Forward Euler: Low.
   - Backward Euler: Moderate due to solving nonlinear equations.
   - Trapezoidal: Moderate; less than fully implicit methods.
4. **Handling Stiffness:**

- Forward Euler: Prone to instability in stiff systems.
- Backward Euler: Handles stiffness better than Forward Euler.
- Trapezoidal: Moderate performance in stiff systems.

3. Dynamic Neural Networks

3.1. Dynamic Neural Networks Formalism

The key idea of Dynamic neural networks (DNNs) is to represent the action of neurons by ordinary differential equations, instead of the generally used static activation functions like ReLU or tanh. For our use case, we use a DNN to represent a system of nonlinear differential equations whose parameters are to be tuned. DNNs are incredibly well-suited for representing such dynamical systems because instead of only using a traditional nonlinear function of the net input, each neuron also involves a linear differential equation with two internal state variables. These state variables are themselves dependent nonlinearly on the net input. On top of this, the net input itself already includes time derivatives of the outputs from the preceding layers. All in all, this enables each neuron to act like a second-order band-pass filter, making them very powerful building blocks for modelling.

Thus [1] motivates the use of second-order ODE as a building block of dynamic neural networks for simulating electronic circuits. We adopt the same framework in our approach. In particular, we solve the following ODE to compute the state of neuron i denoted by ξ_i, when applied to the input s_i :

$$\tau_{2,i}\xi_i'' + \tau_{1,i}\xi_i' + \xi_i = F\left(s_i\right)$$

To be able to predict the topology of the DNN, we choose $F(s_i)$ to be s_i.

$$F(s_i) = s_i$$

The term s_i (net input to the neuron) contains the following information from the previous layers:

$$s_i = \begin{bmatrix} w_{\tilde{\xi}} \end{bmatrix}_i \quad \begin{bmatrix} w_u \end{bmatrix}_i \begin{bmatrix} \tilde{\xi}_i \\ u \end{bmatrix} + \begin{bmatrix} v_{\tilde{\xi}} \end{bmatrix}_i \quad \begin{bmatrix} v_{u'} \end{bmatrix}_i \begin{bmatrix} \tilde{\xi}'_i \\ u' \end{bmatrix} - \theta_i$$

In the above term, W contains the static weights corresponding to state and the input terms. V consists of the dynamic weights which are multiplied with first order time derivatives of the state and the input terms. The θ term denotes the bias. Both the τ terms in ODE facilitate time-integration, thereby time-averaging the input signal and enabling the DNN to produce a non-instantaneous response.

3.2. Transformation of the state-space model

In this thesis, we restrict ourselves to LTI systems that can be reduced to the form:

$$x^{'}(t) = Ax(t) + Bu(t)$$
$$y(t) = Cx(t) + Du(t)$$

Here, $x(t) \in R^n$ is the state vector, $u(t) \in R^m$ is the input vector and $y(t) \in R^l$ is the corresponding output vector. The matrices $A \in R^{n\times n}, B \in R^{n\times m}, C \in R^{l\times n}$ and $D \in R^{l\times n}$ are the system matrices, more specifically, A being the state matrix, B being the input matrix, C being the output matrix and D being the feedforward matrix. In case this reduced order representation is not available, we first use subspace identification algorithms like MOESP to identify the reduced order LTI system from the input-output data.

For simplicity, we assume the Initial Value Problem to be $x(0) = 0$ and $x^{'}(0) = 0$ to simplify the mapping from the state-space matrices to the dynamic neural network parameters. We achieve this by translating the input and output vectors appropriately. The modified input and output vectors are as follows:

$$\hat{u}(t) = u(t) - u(0)$$
$$\hat{y(t)} = y(t) - \{C[\int_0^t e^{A(t-\mu)}\, d\mu]Bu(0) + Du(0)\}$$

Using the translated input and output vectors, our updated state space model is:

$$x^{'}(t) = Ax(t) + B\hat{u(t)}$$
$$\hat{y(t)} = Cx(t) + D\hat{u(t)}$$

As a next step, we pre-process the system described by the equations above into a suitable form. The purpose of the proposed transformation is to decouple the state equation as much as possible and to promote sparsity in the connections of the resulting DNN. Moreover, this also facilitates parallelism to some extent, during the forward and backward pass across the hidden layers, while training the DNNs.

The pre-processing pipeline consists of three main steps:

3.2.1. Block Upper Triangulation using Real Schur Decomposition

We start with the block upper triangulation of the state matrix A using the real Schur decomposition:

$$Q^T AQ = R$$

,

where $Q \in R^{n\times n}$ is an orthogonal matrix and $R \in R^{n\times n}$ is a block upper triangular matrix.

$$R = \begin{bmatrix} R_{11} & R_{12} & \cdots & R_{1q} \\ & R_{22} & \cdots & R_{2q} \\ & & \ddots & \vdots \\ & & & R_{qq} \end{bmatrix}$$

The diagonal block $R_{jj} \in R^1$ if the eigenvalue of R_{jj} is real. However, $R_{jj} \in R^{2\times 2}$ if the corresponding complex eigenvalues constitute a conjugate pair.

3.2.2. Block Diagonalization using Bartels-Stewart algorithm

In numerical linear algebra, the Bartels–Stewart algorithm [28] is used to numerically solve the Sylvester matrix equation $AX - XB = C$. In the next step, we will use Bartels-Stewart algorithm to block diagonalize matrix R. In order to do that, we first need to regroup R into blocks such that the diagonal blocks $R_{jj}^{\sim}$ have disjoint spectra. With the proposed regrouping strategy, if the eigenvalues lying in the resulting diagonal block $R_{jj}^{\sim}$ are real with algebraic multiplicity k, then, $R_{jj}^{\sim}$ is upper triangular.

$$R_{jj}^{\sim} \in R^{k\times k} = \begin{bmatrix} \lambda_j & * & \cdots & \cdots & * \\ 0 & \lambda_j & * & \cdots & * \\ 0 & 0 & \ddots & * & \vdots \\ \vdots & \vdots & 0 & \ddots & * \\ 0 & 0 & \cdots & 0 & \lambda_j \end{bmatrix}$$

However, if the eigenvalues in the diagonal block $R_{jj}^{\sim}$ are complex conjugate pairs with algebraic multiplicity k, then $R_{jj}^{\sim}$ has a sparsity pattern:

$$R_{jj}^{\sim} \in R^{k\times k} = \begin{bmatrix} * & * & * & * & \cdots & \cdots & * & * \\ * & * & * & * & \cdots & \cdots & * & * \\ 0 & 0 & * & * & \cdots & \cdots & * & * \\ 0 & 0 & * & * & \cdots & \cdots & * & * \\ \vdots & \vdots & 0 & 0 & \ddots & & * & * \\ \vdots & \vdots & 0 & 0 & & \ddots & * & * \\ \vdots & \vdots & \vdots & \vdots & 0 & 0 & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \end{bmatrix}$$

Now we introduce another similarity transformation given by:

$$Y^{-1}RY = G$$

This block diagonalizes R using the Bartels-Stewart algorithm such that $R_{ij}^{\sim}$ for $i \neq j$. The diagonal blocks of R $i = j$ are not affected by this transformation. Thus, G looks like:

$$G = \begin{bmatrix} R_{11} & 0 & 0 & 0 \\ 0 & R_{22} & 0 & 0 \\ 0 & 0 & \ddots & \vdots \\ 0 & 0 & 0 & R_{qq} \end{bmatrix}$$

3.2.3. Producing zeros in suitable diagonal entries

Next, we need to transform the first-order ODEs of the LTI systems into second order ODEs represented by the dynamic neural network. For diagonal blocks of G having real eigenvalues, no further transformation is needed. However, we need a transformation for the diagonal blocks having complex eigenvalues.

We first consider $R_{jj}^{\sim} \in R^{(2\times 2)}$ has complex eigen values $(\lambda_j, \bar{\lambda_j})$ with algebraic multiplicity 1.

$$R_{jj}^{\sim} = \begin{bmatrix} \alpha & \beta \\ \gamma & \delta \end{bmatrix}$$

In order to represent two first-order equations (from the transformed state-space model) by one second-order equation represented by the neural network, we define M_{jj} such that:

$$M_{jj} = \begin{bmatrix} 0 & \delta^{-1} \\ 1 & \beta^{-1} \end{bmatrix}$$

This implies:

$$(M_{jj})^{-1} R_{jj}^{\sim} (M_{jj}) = \begin{bmatrix} 0 & (\beta\gamma - \alpha\delta)/\beta\delta \\ \beta\delta & \alpha + \delta \end{bmatrix}$$

We can then generalize this for the case when $R_{jj}^{\sim}$ has complex eigenvalues $(\lambda_j, \bar{\lambda_j})$ with algebraic multiplicity k. Decomposing $R_{jj}^{\sim}$ into 2×2 blocks r_{ij} , we get,

$$R_{jj}^{\sim} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & \cdots & r_{1k} \\ 0 & r_{22} & r_{23} & \cdots & r_{2k} \\ 0 & 0 & r_{33} & \cdots & r_{3k} \\ \vdots & \vdots & \ddots & \vdots & \\ 0 & 0 & \cdots & 0 & r_{kk} \end{bmatrix}$$

We define transformations $M_1, M_2, ...M_k$ corresponding to each diagonal block r_{jj} to produce zeros in the required positions such that

$$M_{jj} \in R_{2k\times 2k} = diag \begin{bmatrix} M_1 & M_2 & \cdots & Mk \end{bmatrix}$$

The final transformation is expressed as:

$$(M_{jj})^{-1}R^{\sim}_{jj}(M_{jj}) = \begin{bmatrix} M_1^{-1}r_{11}M_1 & M_1^{-1}r_{12}M_2 & M_1^{-1}r_{13}M_3 & \cdots & M_1^{-1}r_{1k}M_k \\ 0 & M_2^{-1}r_{22}M_2 & M_2^{-1}r_{23}M_3 & \cdots & M_2^{-1}r_{2k}M_k \\ 0 & 0 & M_3^{-1}r_{33}M_3 & \cdots & M_3^{-1}r_{3k}M_k \\ \vdots & \vdots & \ddots & \vdots & \\ 0 & 0 & \cdots & 0 & M_k^{-1}r_{kk}M_k \end{bmatrix}$$

Note that all the 2×2 diagonal entries $M_j^{-1}r_{jj}M_j$ are now in the required form $\begin{bmatrix} 0 & * \\ * & * \end{bmatrix}$

3.2.4. Piecing together all the transformations

After applying all three similarity transformations sequentially, we get the final transformed state matrix $A^{\sim}$:

$$A^{\sim} = (QYM)^{-1}A(QYM)$$

We now define the coordinate transformation for the state vector as:

$$x(t) = (QYM)\xi(t)$$

Substituting the transformations in the state equation, we get:

$$\xi^{'}(t) = \begin{bmatrix} A^{\sim}_{11} & & & \\ & A^{\sim}_{22} & & \\ & & \ddots & \\ & & & A^{\sim}_{qq} \end{bmatrix} \begin{bmatrix} \Xi_1 \\ \Xi_2 \\ \vdots \\ Xi_q \end{bmatrix} + (QYM)^{-1}B\hat{u}(t)$$

Similarly, the transformed output equation in the new co-ordinate system is:

$$\hat{y}(t) = C(QYM)\xi(t) + D\hat{u}(t)$$

3.3. Mapping from state space matrices to DNN parameters

Based on the transformations derived in the previous section, we then exploit the block-diagonal structure of the state matrix A. This facilitates deriving a mapping from the state equation for $\xi^{'}(t)$ given above to the parameters of the hidden layer of the DNN. By the term *mapping*, we essentially aim to determine:

1. the number of neurons,
2. the number of layers, and
3. how the neurons must be connected to each other.

Block-diagonalization of the state matrix $\tilde{A}$ ensures that the states in different blocks are independent. In other words, one can decouple the state equation into q blocks of equations.

This makes it possible to derive a mapping for each block of state equations and construct the DNN architecture one block at a time.

For example, consider the case when the system matrix $\tilde{A}$ has two diagonal blocks with disjoint spectra. Let the first diagonal block have eigenvalue $\lambda_1 \in \mathbb{R}$ or $(\lambda_1, \overline{\lambda_1}) \in \mathbb{C}$ with algebraic multiplicity k. Moreover, let the second diagonal block have eigenvalue $\lambda_2 \in \mathbb{R}$ or $(\lambda_2, \overline{\lambda_2}) \in \mathbb{C}$ with algebraic multiplicity 1 . The figure 3.1 demonstrates the DNN architecture that emerges out of the mapping process of DNN corresponding to the given system having two blocks with real/complex eigenvalues having algebraic multiplicities k and 1 respectively.

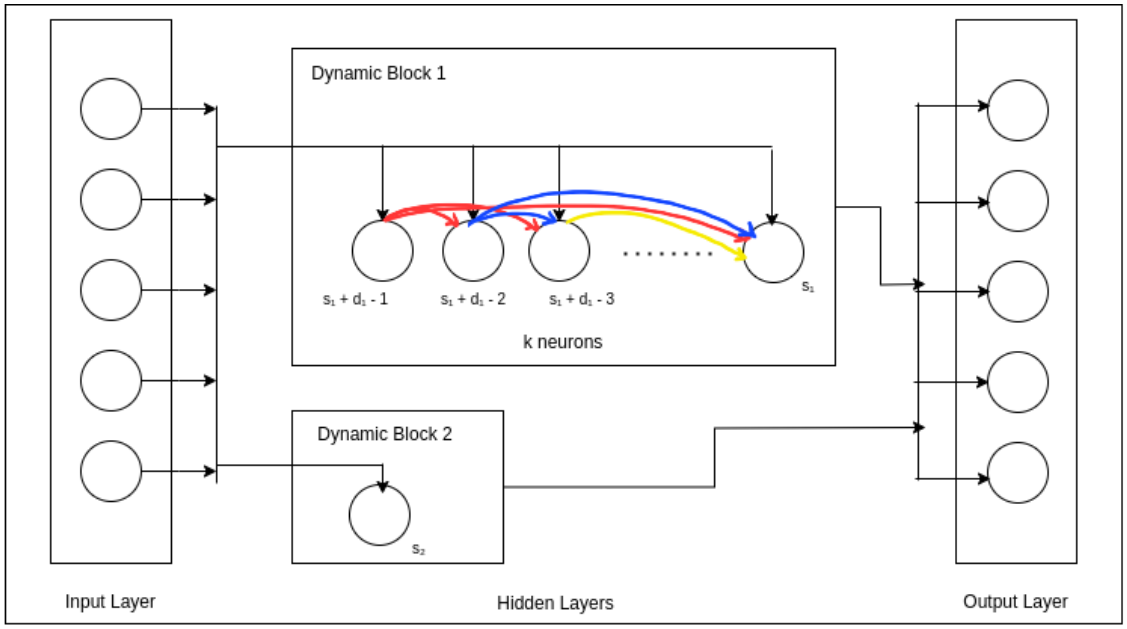

Figure 3.1.: Dynamic Neural Network : Architecture

3.4. The Dataset

For the purpose of demonstrating the training of DNNs, and subsequently benchmarking the performance, we consider three different LTI systems that can be reduced to the form:

$$x^{'}(t) = Ax(t) + Bu(t)$$
$$y(t) = Cx(t) + Du(t)$$

Here, $x(t) \in R^n$ is the state vector, $u(t) \in R^m$ is the input vector and $y(t) \in R^l$ is the corresponding output vector. The matrices $A \in R^{n \times n}, B \in R^{n \times m}, C \in R^{l \times n}$ and $D \in R^{l \times n}$ are the system matrices, more specifically, A being the state matrix, B being the input matrix, C

being the output matrix and D being the feedforward matrix.

Dataset A

This is a 2×2 LTI system with the system matrices being as follows:

$$A = \begin{bmatrix} a & b \\ 0 & a \end{bmatrix},$$

$$B = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix},$$

$$C = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \text{ and}$$

$$D = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

Here, $A \in R^{2\times2}$ is a upper-triangular matrix with real and repeating diagonal element a. On giving a sinusoidal signal as an input $u(t)$ and then simulating the system using *scipy.signal.lsim*, we get an output response as shown in figure 4.8

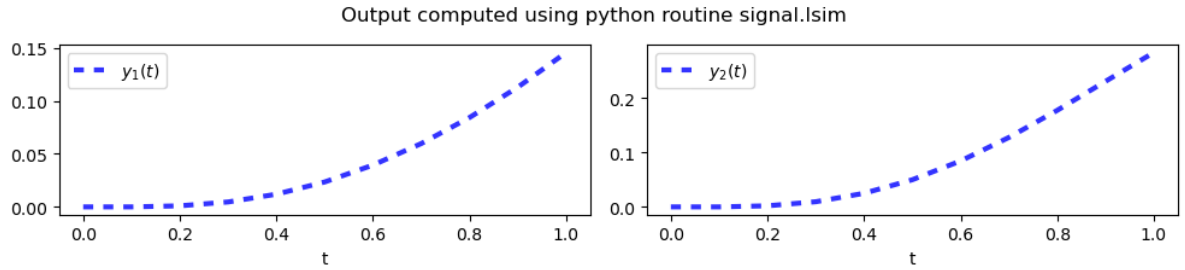

Figure 3.2.: Output Response for the 2x2 LTI system

Dataset B

This is a 4×4 LTI system with the system matrices being as follows:

$$A = \begin{bmatrix} a & b & c & d \\ 0 & a & e & f \\ 0 & 0 & a & g \\ 0 & 0 & 0 & a \end{bmatrix},$$

$$B = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

$$D = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix},$$

Here, $A \in R^{4\times4}$ is a upper-triangular matrix with real and repeating diagonal element a. On giving a sinusoidal signal as an input $u(t)$ and then simulating the system using *scipy.signal.lsim*, we get an output response as shown in figure 3.3

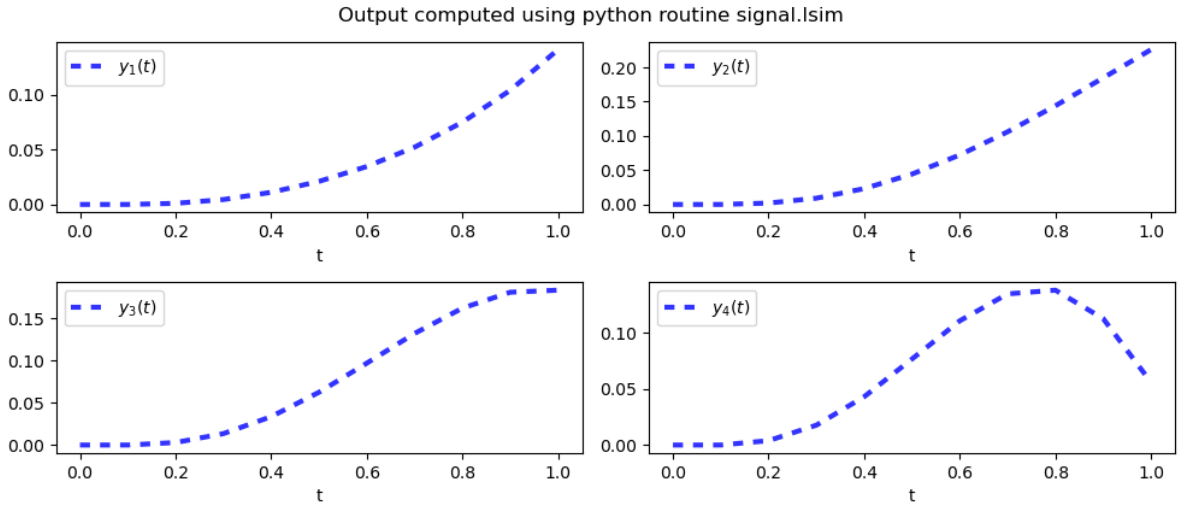

Figure 3.3.: Output Response for the 4x4 LTI system

Dataset C

This is a 16×16 LTI system with the system matrices being as follows:

$$A = \begin{bmatrix} a & b & c & d & \cdots & \cdots & e & f \\ 0 & a & g & h & \cdots & \cdots & i & j \\ 0 & 0 & a & k & \cdots & \cdots & l & m \\ 0 & 0 & 0 & a & \cdots & \cdots & n & o \\ \vdots & \vdots & 0 & 0 & \ddots & & p & q \\ \vdots & \vdots & 0 & 0 & & \ddots & r & s \\ \vdots & \vdots & \vdots & \vdots & 0 & 0 & a & t \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & a \end{bmatrix},$$

$$B = C = \begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & \cdots & 0 & 0 \\ 0 & 1 & 0 & 0 & \cdots & \cdots & 0 & 0 \\ 0 & 0 & 1 & 0 & \cdots & \cdots & 0 & 0 \\ 0 & 0 & 0 & 0 & \cdots & \cdots & 0 & 0 \\ \vdots & \vdots & 0 & 0 & \ddots & & 0 & 0 \\ \vdots & \vdots & 0 & 0 & & \ddots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}, and$$

$$D = \begin{bmatrix} 0 & 0 & 0 & 0 & \cdots & \cdots & 0 & 0 \\ 0 & 0 & 0 & 0 & \cdots & \cdots & 0 & 0 \\ 0 & 0 & 0 & 0 & \cdots & \cdots & 0 & 0 \\ 0 & 0 & 0 & 0 & \cdots & \cdots & 0 & 0 \\ \vdots & \vdots & 0 & 0 & \ddots & & 0 & 0 \\ \vdots & \vdots & 0 & 0 & & \ddots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Here, $A \in R^{16\times16}$ is a upper-triangular matrix with real and repeating diagonal element a. On giving a sinusoidal signal as an input $u(t)$ and then simulating the system using *scipy.signal.lsim*, we get an output response as shown in figure 4.12 and figure 3.5

3.5. The Forward Pass

We implement our forward pass in *pytorch*. We first start by implementing a custom neural network class called *DNN* (Dynamic Neural Network) that inherits from *torch.nn.Module*. This class will encapsulate the architecture of the DNN. In the constructor (*init* method)of the DNN class, we define all the different trainable parameter tensors of our dynamic neural network using PyTorch's *nnmodule*. We also initialize each parameter tensor first *randomly* from a uniform distribution, and then using *xavier* initialization. Next we override

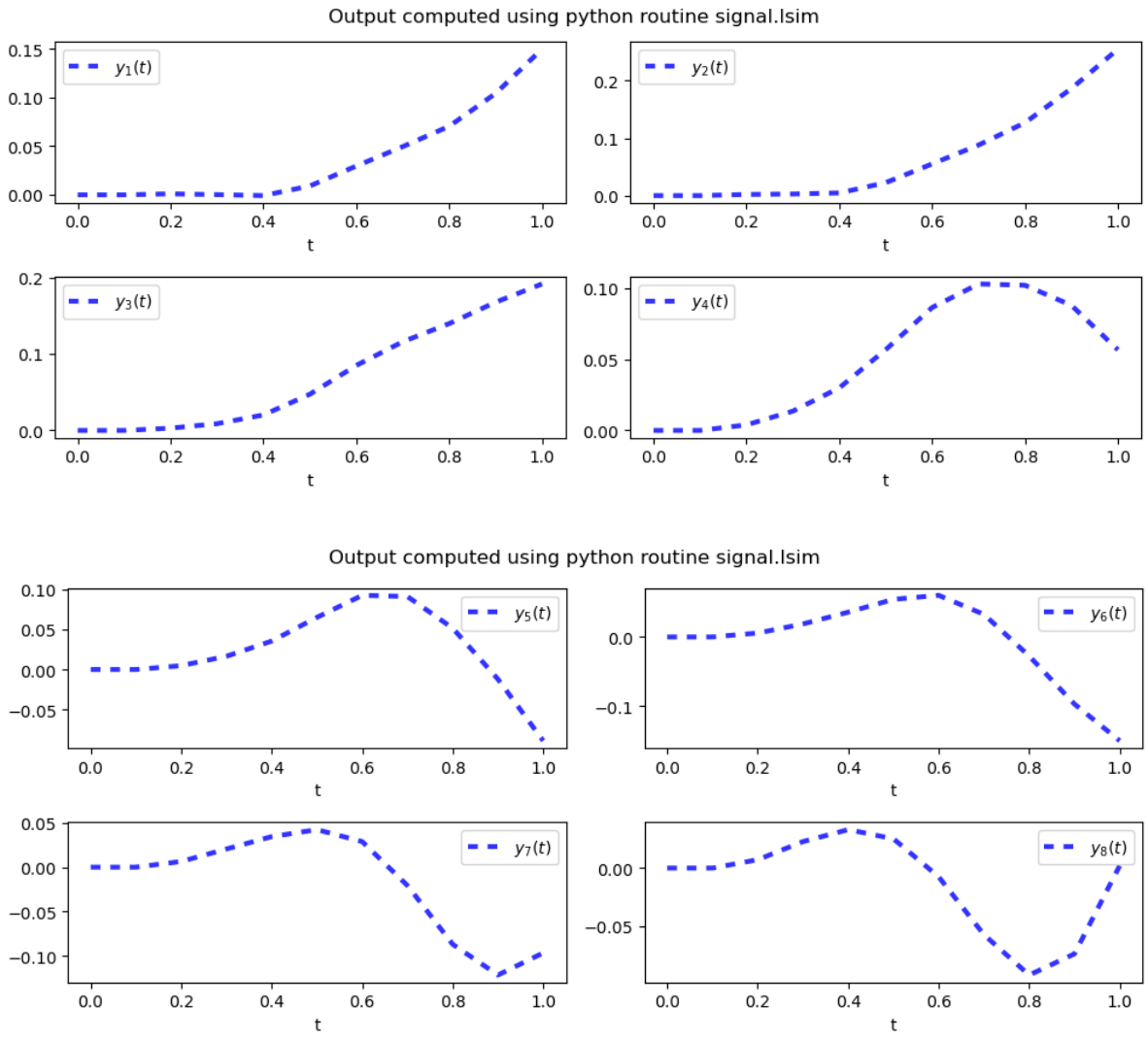

Figure 3.4.: System Response for the 16x16 system - I

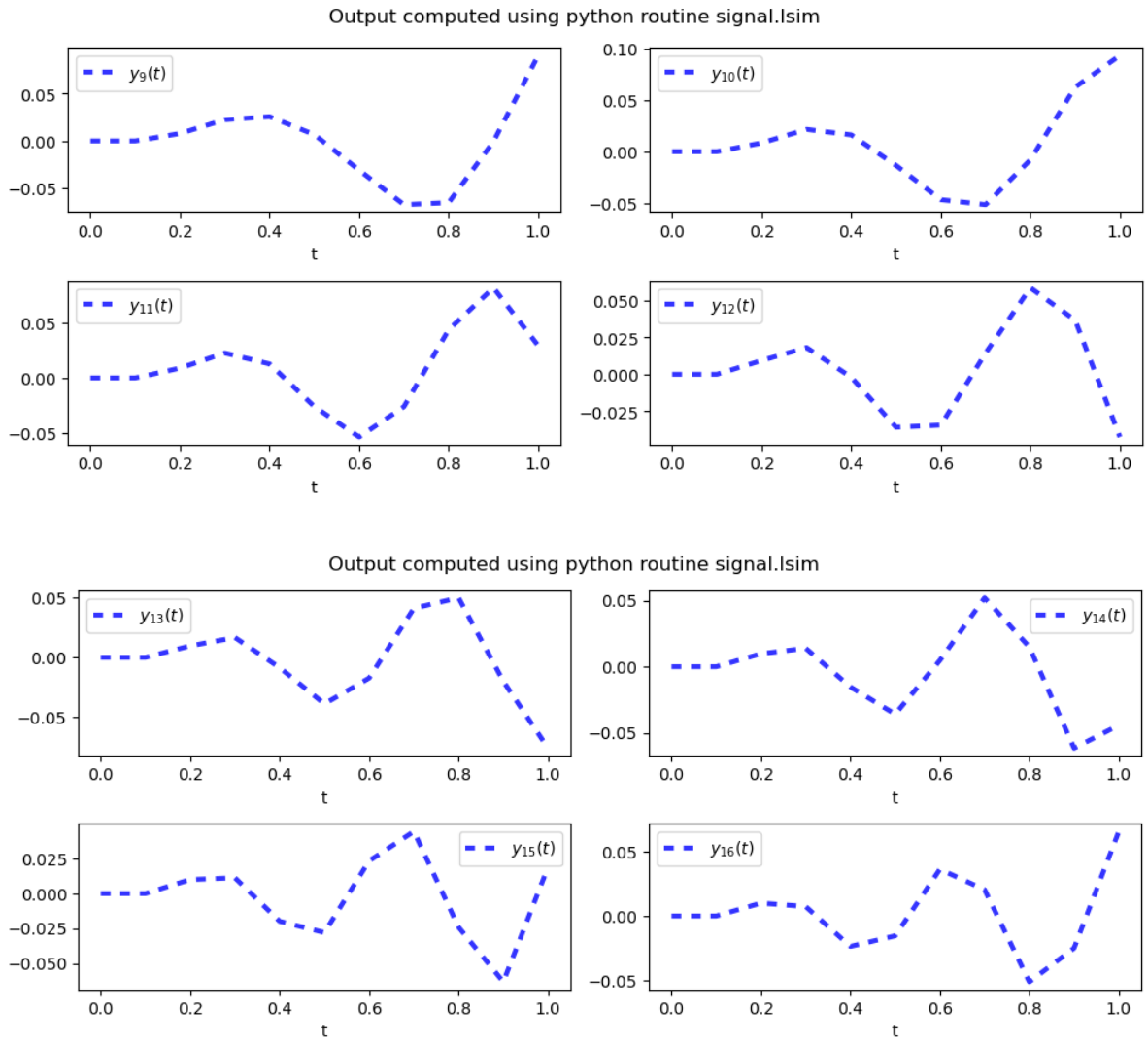

Figure 3.5.: System Response for the 16x16 system - II

the *forward* method of the *nn.Module* class. This method defines how the input data flows through the layers of our network during a forward pass. Inside the *forward* method, we specify the operations and computations that transform the input data into the output. The exact computation performed during the forward pass at each neuron is further explained in detail below.

The differential equation for the output, or excitation, y_{ik} of one particular neuron i in layer $k > 0$ (forward pass) is given by

$$\tau_2 \left(\sigma_{1,ik}, \sigma_{2,ik}\right) \frac{\mathrm{d}^2 y_{ik}}{\mathrm{d}t^2} + \tau_1 \left(\sigma_{1,ik}, \sigma_{2,ik}\right) \frac{\mathrm{d}y_{ik}}{\mathrm{d}t} + y_{ik} = \mathcal{F}\left(s_{ik}, \delta_{ik}\right)$$

This equation can be rewritten into two first order differential equations by introducing an auxiliary variable z_{ik} as in

$$\left[\begin{array}{rl} \mathcal{F}\left(s_{ik}, \delta_{ik}\right) & = y_{ik} + \tau_{1,ik} \frac{\mathrm{d}y_{ik}}{\mathrm{d}t} + \tau_{2,ik} \frac{\mathrm{d}z_{ik}}{\mathrm{d}t} \\ z_{ik} & = \frac{\mathrm{d}y_{ik}}{\mathrm{d}t} \end{array}\right.$$

We then apply the Backward Euler integration method, according to the substitution scheme

$$\boldsymbol{f}(\boldsymbol{x}, \dot{\boldsymbol{x}}, t) = \boldsymbol{0} \quad \rightarrow \quad \boldsymbol{f}\left(\xi_1 \boldsymbol{x} + \xi_2 \boldsymbol{x}', \frac{\boldsymbol{x} - \boldsymbol{x}'}{h}, t\right) = \boldsymbol{0}$$

where values at previous time points in the discretized expressions are denoted by accents ($'$). Consequently, a set of implicit nonlinear differential — or differential-algebraic equations for variables in the vector $\boldsymbol{x}$ is replaced by a set of implicit nonlinear algebraic equations from which the unknown new $\boldsymbol{x}$ at a new time point $t = t' + h$ with $h > 0$ has to be solved for a (known) previous $\boldsymbol{x}'$ at time t'. Different values for the parameters ξ_1 and ξ_2 allow for the selection of a particular integration scheme. The Forward Euler method is obtained for $\xi_1 = 0, \xi_2 = 1$, the Backward Euler method for $\xi_1 = 1, \xi_2 = 0$, the trapezoidal integration method for $\xi_1 = \xi_2 = \frac{1}{2}$ and the second order Adams-Bashforth method for $\xi_1 = \frac{3}{2}, \xi_2 = -\frac{1}{2}$.

Both the Backward Euler integration method and the trapezoidal integration method are numerically very stable—A-stable—methods [29]. The local truncation error of the trapezoidal method is $O(h^3)$, with h the size of the time step, instead of the $O(h^2)$ local truncation error of the Backward Euler integration method. [30]

Integration Method	ξ_1	ξ_2
Forward Euler Method	0	1
Backward Euler Method	1	0
Trapezoidal Method	1/2	1/2
Adams-Bashforth method	3/2	-1/2

Table 3.1.: Choices of different Integration schemes

This gives the algebraic equations:

$$\left[\begin{array}{rl} \tau_{2,ik}\frac{z_{ik}-z'_{ik}}{h} & = \xi_1\left(\mathcal{F}\left(s_{ik},\delta_{ik}\right)-y_{ik}-\tau_{1,ik}z_{ik}\right) \\ & +\xi_2\left(\mathcal{F}\left(s'_{ik},\delta_{ik}\right)-y'_{ik}-\tau_{1,ik}z'_{ik}\right) \\ \frac{y_{ik}-y'_{ik}}{h} & = \xi_1 z_{ik}+\xi_2 z'_{ik} \end{array}\right.$$

Now, due to the particular form of the differential equations, we can solve the equations for y_{ik} and z_{ik} to obtain the behaviour as a function of time, and we find for layer $k > 0$

$$\begin{aligned} [y_{ik} = &\left\{\xi_1^2\mathcal{F}\left(s_{ik},\delta_{ik}\right)+\xi_1\xi_2\mathcal{F}\left(s'_{ik},\delta_{ik}\right)\right. \\ &\left.+\left[-\xi_1\xi_2+\xi_1\frac{\tau_{1,ik}}{h}+\frac{\tau_{2,ik}}{h^2}\right]y'_{ik}+\frac{\xi_1+\xi_2}{h}\tau_{2,ik}z'_{ik}\right\} \\ &/\left\{\xi_1^2+\xi_1\frac{\tau_{1,ik}}{h}+\frac{\tau_{2,ik}}{h^2}\right\} \\ z_{ik} = &\frac{y_{ik}-y'_{ik}}{h\xi_1}-\frac{\xi_2}{\xi_1}z'_{ik} \end{aligned}$$

Particularly, in our implementation, we use the Backward Euler integration method. Hence, we further substitute $\xi_1 = 1, \xi_2 = 0$ which further simplifies the expression for which the s_{ik} are obtained from:

$$s_{ik} = \sum_{j=1}^{N_{k-1}} w_{ijk}y_{j,k-1} - \theta_{ik} + \sum_{j=1}^{N_{k-1}} v_{ijk}z_{j,k-1}$$

The steady state behaviour of one particular neuron i in layer $k > 0$ at time $t = 0$ is given by:

$$\begin{bmatrix} s_{ik}|_{t=0} = \sum_{j=1}^{N_{k-1}} w_{ijk} y_{j,k-1} - \theta_{ik} \\ y_{ik}|_{t=0} = \mathcal{F}(s_{ik}, \delta_{ik}) \\ z_{ik}|_{t=0} = 0 \end{bmatrix}$$

The output of the forward pass is then used to compute the loss during training and eventually to simulate the output of the system during inference.

3.6. Understanding Backpropagation Through Time (BPTT)

In the context of DNNs, which are designed to capture temporal dependencies in data, Backpropagation Through Time (BPTT) extends the traditional backpropagation algorithm to handle sequences of variable lengths. The core idea behind BPTT is to unfold the temporal structure of the DNN over a fixed number of time steps and treat it as a feedforward neural network with shared weights across time. The BPTT algorithm involves iterating through the unfolded network for each time step, computing the forward pass, calculating the loss, and then performing backpropagation to update the model parameters. The gradients are accumulated across all time steps, reflecting the impact of each time step on the final loss. The unfolded structure allows the model to capture dependencies across time, enabling effective learning of sequential patterns.

Despite its effectiveness, BPTT is known to suffer from vanishing or exploding gradient problems, particularly in long sequences [31]. In practice, networks can have a large number of time steps which makes BPTT computationally expensive. Truncated BPTT is a common approach where the backward pass is limited to a fixed number of time steps. This not only saves computational time but also helps mitigate the vanishing/exploding gradient problem.

We will use Truncated Backpropagation Through Time (TBPTT) algorithm for training our DNN. TBPTT addresses the computational challenges associated with BPTT, particularly when dealing with long sequences. Instead of performing backpropagation through the entire sequence, TBPTT truncates the sequence into shorter segments, allowing for more efficient training. The algorithm proceeds as follows:

1. **Initialization:** Initialize the model parameters, including weights and biases. We tested two different initializations: random initialization from a uniform distribution and xavier initialization.

2. **Sequence Truncation:** Choose a fixed number of time steps, known as the truncation length. This length determines the segment of the sequence over which backpropagation will be performed.

3. **Forward Pass:** For each truncated segment, perform a forward pass through the DNN. Calculate the hidden states and predictions for each time step within the truncated segment.

4. **Loss Computation:** Calculate the loss at each time step within the truncated segment by comparing the predicted output with the actual target.

5. **Backpropagation within Truncated Segment:** Perform backpropagation within the truncated segment to compute the gradients of the loss with respect to the model parameters. This involves computing the gradients through the chain rule, considering the contributions from each time step within the truncated segment.

6. **Parameter Update:** Adam (short for Adaptive Moment Estimation) is our choice of optimizer as it combines ideas from two other optimization algorithms, RMSprop and Momentum, and introduces adaptive learning rates.

7. **Repeat for the Entire Sequence:** Repeat the process by moving the truncation window along the sequence. Each truncated segment contributes to the overall learning process, and the model parameters are updated based on the accumulated gradients from all segments.

8. **Repeat for Multiple Epochs:** Continue this process for a fixed number of iterations (epochs) or until convergence is achieved. Truncating the backpropagation sequence provides computational efficiency while still allowing the model to capture dependencies within each segment.

TBPTT strikes a balance between computational efficiency and modeling effectiveness, making it more feasible for training on large datasets with extended temporal dependencies. The choice of truncation length is a hyperparameter that can be tuned based on the characteristics of the data and the available computational resources.

3.7. Implementation

The core of this thesis work is the Pytorch implementation of the Dynamic Neural Network formalism. More specifically, we implement the Forward Pass, Backward Pass, Training Loop and a couple of ODE solvers to generate the ground truth required for training loop.

Here, we initialize the weights of the neural network randomly from a uniform distribution.

```
def random_initialize(self):
      # Weights/Trainable parameters
      self.tau_1 = nn.Parameter(torch.rand_like(-self.evals_inv)).to(self.device)
      self.tau_2 = nn.Parameter(torch.rand(self.d_A)).to(self.device)
      self.W_u_hat = None #(If B =!0)
```

```
        self.W_xi_tilde = nn.Parameter(torch.rand(int((self.d_A - 1) * self.d_A / 2)))
        self.W_u_hat = nn.Parameter(torch.rand_like(self.ssm_B)).to(self.device)
```

Next, we implement the differential equations described earlier to be able to perform the forward pass. Each neuron solves a linear differential equation, driven by a nonlinear function of the net input. The net input itself has time derivatives of outputs from previous layers. Different kinds of dynamic behaviour may arise from an individual neuron, depending on the values of its parameters.

```
def forward_pass(self, timesteps, a_1, a_2, h, inputs):
    u_hat = torch.from_numpy(inputs)

    h_inv = 1./h
    # Initialize the state and state-derivative set to zero
    sol_dnn = torch.zeros((len(timesteps),self.d_A, 2)) # timesteps, neurons, states
        per neuron
    y_dnn = torch.zeros((len(timesteps), self.ssm.dim[2])) # timesteps, neurons,
        states per neuron

    for t in range(1, len(timesteps)):
        f_old = 0
        for i in reversed(range(self.d_A)): # Last row first
            # Solve the ODE and store the state for each time step and update it for
                next time-steps

            # compute the forcing term
            f_old = self.compute_f_new(i, u_hat[t-1], sol_dnn, t-1)

            sol_dnn[t, i, 0] = (h / self.tau_1[i]) * (f_old - sol_dnn[t-1, i, 0]) +
                sol_dnn[t-1, i, 0]

        # Update state and it's derivative after each time-step
        self.xi = sol_dnn[t, :, 0]
        #self.xi_prime = sol_dnn[t, :, 1]
        y_dnn[t, :] = self.compute_output()

    return y_dnn, sol_dnn # tensor with dimensions timesteps * n * 2
```

Next, we implement the Truncated Backpropagation Through Time (TBPTT) algorithm. We backpropagate the gradients backwards not just all the through layers and but also through time. We don't perform backpropagation through the entire sequence of data, but through truncated sequences of shorter segments, allowing for more efficient training.

```
model = DyNN.dynn_blocks[0]
```

```
writer = SummaryWriter()

num_epochs = 2000

trajectory_tau1_1 = np.zeros(num_epochs)
trajectory_wxitilde = np.zeros(num_epochs)
trajectory_wuhat = np.zeros(num_epochs)
sequence_length = 10

with torch.autograd.set_detect_anomaly(False):
# Construct our loss function and an Optimizer. The call to model.parameters()
# in the Adam constructor will contain the learnable parameters (defined
# with torch.nn.Parameter) which are members of the model.
    criterion = torch.nn.MSELoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-2)
    for e in range(num_epochs):

        for sequence in input[0].length():
            # Forward pass: Compute predicted y by passing x to the model

            y_pred = DyNN.predict(inputs[0][sequence:sequence+seq_length],
                seq_length, dt)

            # Compute and print loss
            loss = criterion(y_pred.float(),
                torch.from_numpy(y_gt_trapezoidal).float())

            print(e, loss.item())


            # Zero gradients, perform a backward pass, and update the weights.
            optimizer.zero_grad()

            loss.backward(retain_graph=True)

            trajectory_tau1_1[e] = model.tau_1[0]
            trajectory_wxitilde[e] = model.W_xi_tilde
            trajectory_wuhat[e] = model.W_u hat

            optimizer.step()

            sequence = sequence + seq_length
```

3.8. The Loss landscape

In the realm of deep learning, there exist two fundamental questions: 1.) What attributes make models trainable? And 2.) What factors contribute to models' ability to generalize? The solutions to these queries are intricately tied to the characteristics and fundamental geometry of the loss landscape, a quality inherently dictated by the computational architecture of the model. Neural network training relies on our ability to find "good" minimizers of highly non-convex loss functions. It is well-known that certain network architecture designs produce loss functions that train easier, and well-chosen training parameters (batch size, learning rate, optimizer) produce minimizers that generalize better. However, the reasons for these differences, and their effects on the underlying loss landscape, are not well understood.

Visualizations have the potential to help us answer several important questions about why neural networks work. In particular, why are we able to minimize highly non-convex neural loss functions? To answer this questions, we use high-resolution visualizations to provide an empirical characterization of neural loss functions, and explore how the non-convex structure of neural loss functions relates to their trainability, and how the geometry of neural minimizers (i.e., their sharpness/flatness, and their surrounding landscape), affects their generalization properties.

To do this in a meaningful way, we take Dataset B as a sample scenario and then we use visualizations to explore sharpness/flatness of the loss landscape, as well as the effect of network architecture choices on the training efficiency. Our goal is to understand how loss function geometry affects the learning process in DNNs. Using this dataset, once we map our state space matrices to the DNN parameters using the mapping described previously, we have three trainable parameters - namely τ_1, $w_{\hat{u}}$ and $w_{\tilde{\xi}}$. We take each of these parameters - two at a time - and plot a high resolution loss landscape. These landscapes are shown in figures 4.4, 4.5 and 4.6.

As we can visually infer after seeing the landscapes, the loss landscape of our DNNs gets very flat near the minima, which implies that the loss function has a shallow slope or low gradient across a significant portion of the parameter space around the minima. Interestingly, away from the minima, the landscape is still relatively steep. This characteristic has important implications for the training and optimization of the DNNs.

The landscape suggests that the optimization algorithm may converge slowly during training, which was confirmed later during implementation. With a shallow slope, the updates to the model parameters are small, making it challenging for the optimizer to reach a minimum efficiently. Also, since the region around the minima is extremely shallow, we experienced that the optimizer almost always reached an approximate solution somewhere in the region, and found it very difficult to get it to the theoretical global minima. We will visualize the performance of the optimizer by analyzing the training trajectories in the next section.

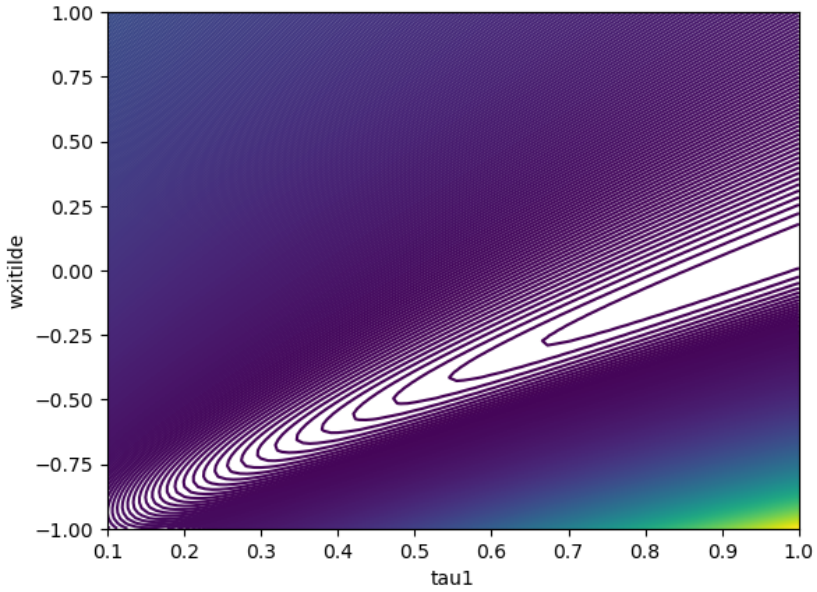

Figure 3.6.: The Loss Landscape - $w_{\tilde{\xi}}$ vs. τ_1

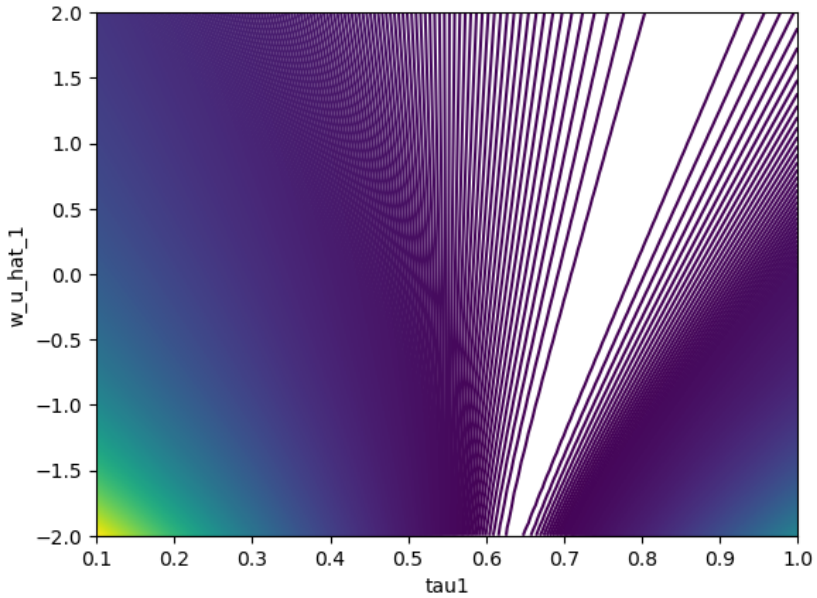

Figure 3.7.: The Loss Landscape - $w_{\hat{u}}$ vs. τ_1

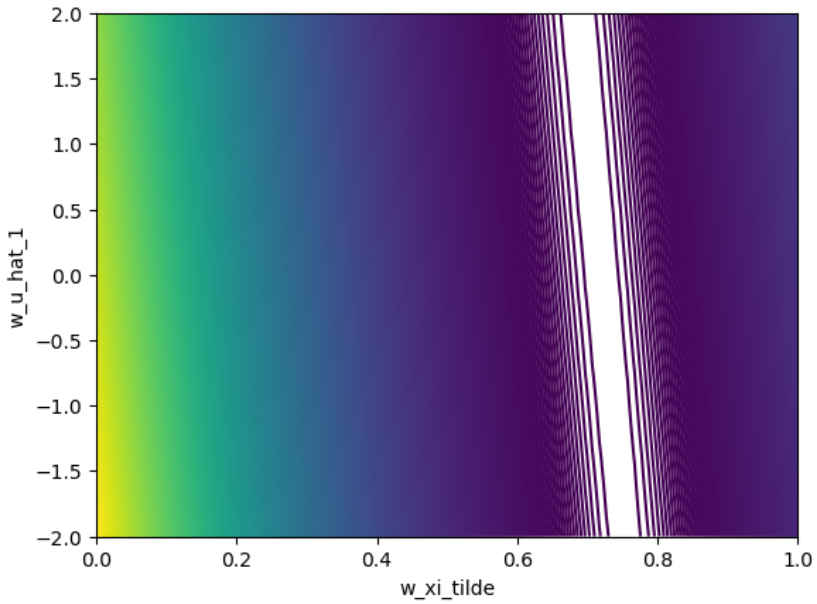

Figure 3.8.: The Loss Landscape - $w_{\hat{u}}$ vs. $w_{\tilde{\xi}}$

4. Results and Discussion

4.1. Performance on different Datasets

For the purpose of demonstrating their efficiency, and to try and benchmark their performance, we trained different DNNs on the three datasets A, B and C as described in the previous section.

Dataset A : 2x2 system with algebraic multiplicity 2

DNN - Mapping

In this dataset, the matrices $A \in R^{2\times2}, B \in R^{2\times2}, C \in R^{2\times2}$ and $D \in R^{2\times2}$ are the system matrices, more specifically, A being the state matrix, B being the input matrix, C being the output matrix and D being the feedforward matrix. Here, the system matrix A has real and repeating eigen values with an algebraic multiplicity of 2. The matrices B and C are Identity matrices and D is a zero or a null matrix. As usual, we give a sinusoidal input pulse $u(t)$ to the system.

For this configuration, we derive the mapping of the DNN using the steps mentioned earlier to get the DNN architecture as described in figure 4.1. In this architecture, we have a single hidden dynamic block with two neurons. The connections between them are as depicted. Each neuron emulates a ODE solver to solve the system equations by means of dynamic connections and trainable parameters τ_1, τ_2, $w_{\hat{u}}$ and $w_{\tilde{\xi}}$.

DNN - Output

As visible from the curve fits in the figure 4.2, the model performs really well and gives us MSE values very close to machine precision when compared with a Backward Euler solver, and reasonably good values when compared to the python routine *signal.lsim*. This is due to our implementation choice of using parameters emulating the Backward Euler solver in our DNN. For the sake of completeness, after playing around with the hyperparameters and performing high level hyperparameter tuning, we got the best results when using a ADAM (Adaptive Moment Estimation) optimizer with a learning rate between $1e-3$ and $1e-2$. In ADAM, instead of adapting learning rates based on the average first moment as in RMSP (Root Mean Squared Propagation), we make use of the average of the second moments of the gradients. This algorithm calculates the exponential moving average of gradients and square gradients and the parameters of $\beta1$ and $\beta2$ are used to control the decay rates of these moving

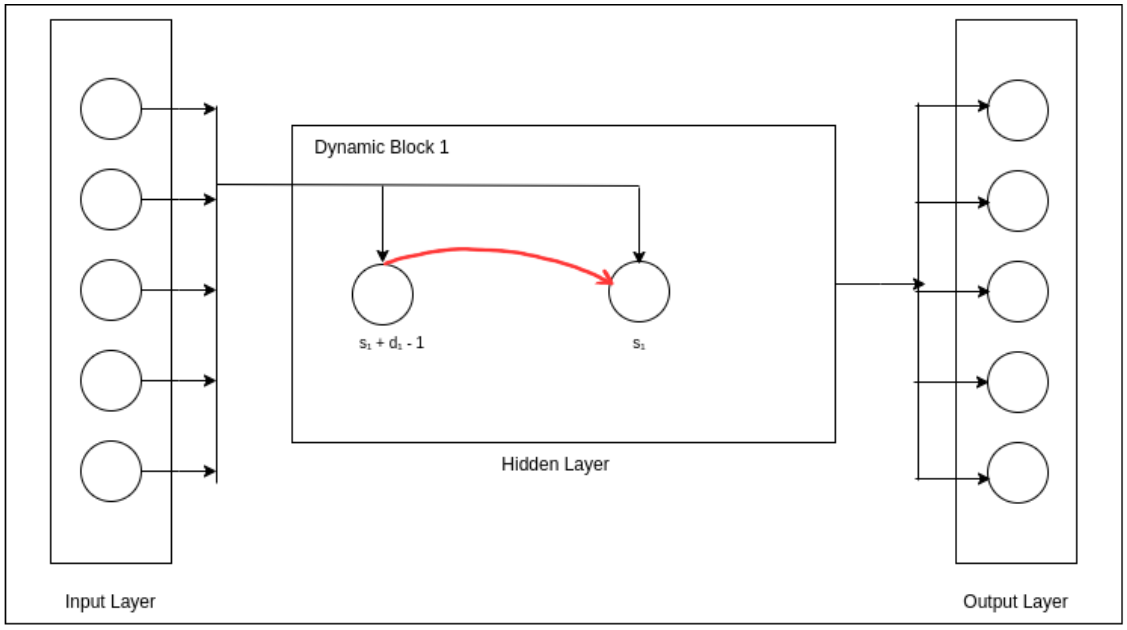

Figure 4.1.: The DNN architecture - 2x2 LTI system

averages. Adam is a combination of two gradient descent methods, Momentum, and RMSP.

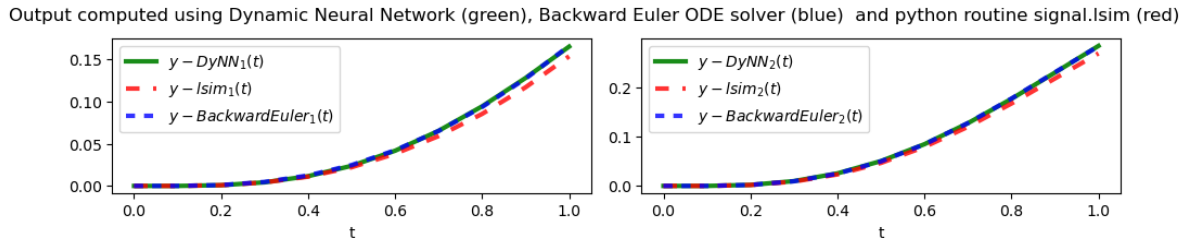

Figure 4.2.: Model Response for the 2x2 LTI system

Loss curve

Learning curves are a widely used diagnostic tool in machine learning for algorithms that learn from a training dataset incrementally. In our study, we plot the learning curves by evaluating our model at every time step, calculating and tracking the value of the loss function (Mean Squared Error in our case). The loss curve helps us in monitoring our training process and also helps us in diagnosing our model performance and gives us a pretty solid intuition on how to further improve it. In figure 4.3, we can see the training curve for the 2×2 system.

As seen from the curve in figure 4.3, we observe that the training was smooth, which

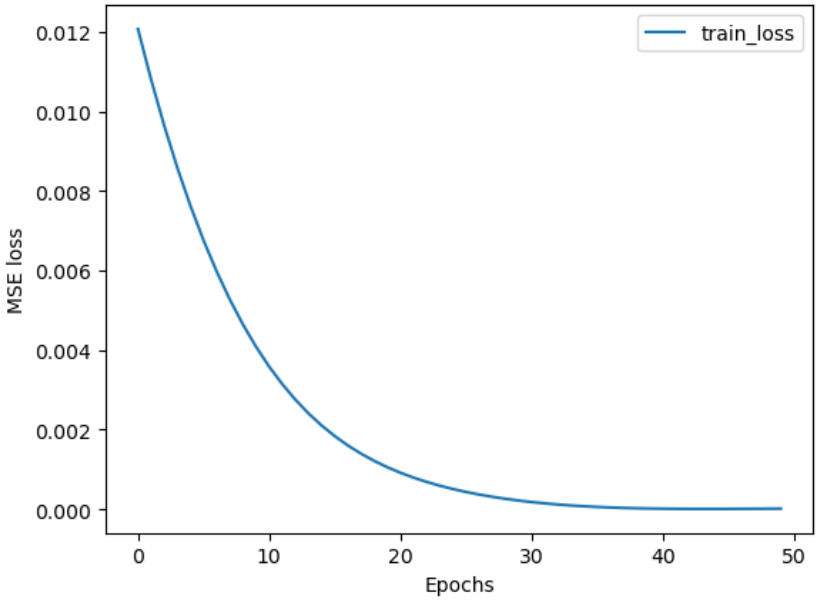

Figure 4.3.: The Loss curve - 2x2 LTI system

indicates that there were no significant problems with the training and the model is behaving as expected as it is steadily improving over time.

Learning trajectories

Deep neural networks exhibit the capability to tackle highly intricate tasks that surpass the capacities of classical algorithms. However, the intricacies of the training process, specifically the trajectory stochastic optimizers traverse through the parameter space—from an initially randomized network to a proficiently trained one—remain inadequately comprehended. While convex optimization benefits from a robust theoretical understanding of the paths taken by various optimization methods and their convergence rates to the global optimum [32], the loss functions associated with deep neural networks are notably non-convex. Although there exist theoretical findings, such as insights into the nature of obstacles within the loss landscape [33], a comprehensive understanding is still lacking.

We have already seen how the loss landscape looks in the previous section, we will now see how our optimizer finds its way from the initialization to the minima in the figures 4.4, 4.5 and 4.6.

In each of these trajectories, we can see that the ADAM optimizer slowly but steadily finds its way to the shallow region surrounding the minima. Admittedly, the converged values of all the trainable parameters are not close to their theoretical equivalents, but this is more about the flatness of the landscape and there existing a relatively large solution space.

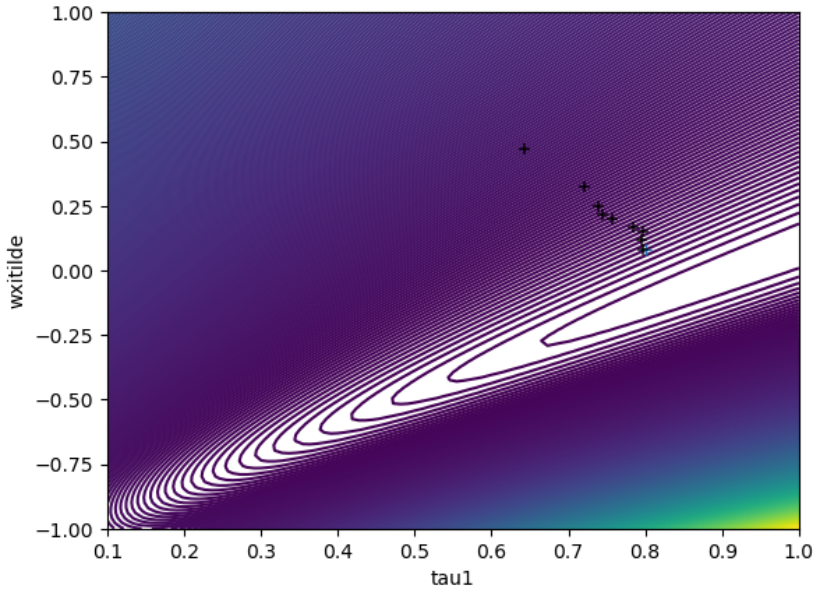

Figure 4.4.: The Loss Trajectory (ADAM) - $w_{\tilde{\xi}}$ vs. τ_1

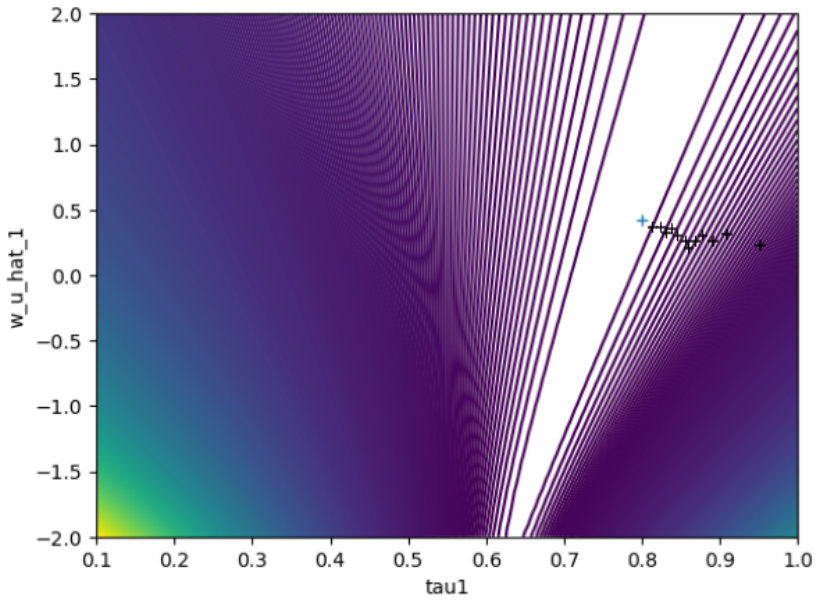

Figure 4.5.: The Loss Trajectory (ADAM) - $w_{\hat{u}}$ vs. τ_1

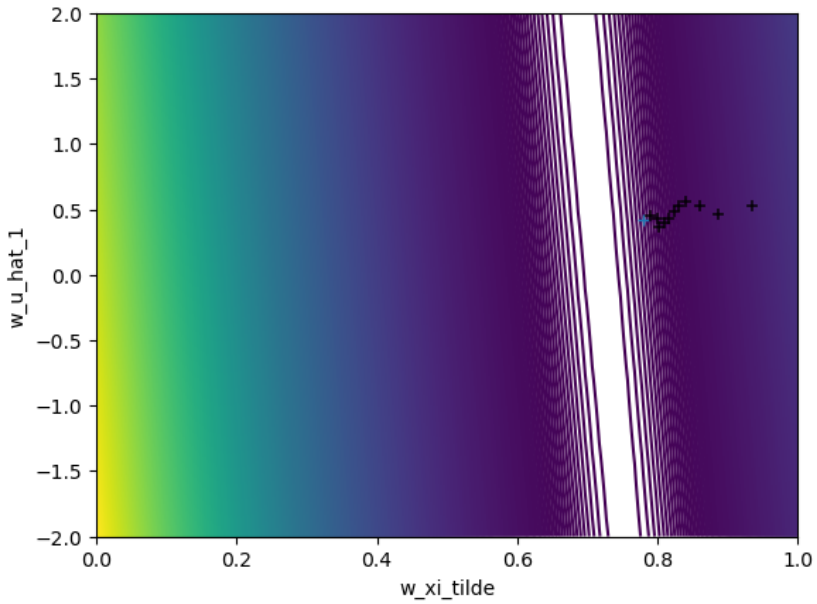

Figure 4.6.: The Loss Trajectory (ADAM) - $w_{\hat{u}}$ vs. $w_{\tilde{\xi}}$

Dataset B : 4x4 system with algebraic multiplicity 4

DNN - Mapping

In this dataset, the matrices $A \in R^{4\times4}, B \in R^{4\times4}, C \in R^{4\times4}$ and $D \in R^{4\times4}$ are the system matrices, more specifically, A being the state matrix, B being the input matrix, C being the output matrix and D being the feedforward matrix. Here, the system matrix A has real and repeating eigen values with an algebraic multiplicity of 4. The matrices B and C are Identity matrices and D is a zero or a null matrix. As usual, we give a sinusoidal input pulse $u(t)$ to the system.

For this configuration, we derive the mapping of the DNN using the steps mentioned earlier to get the DNN architecture as described in figure 4.7. In this architecture, we have a single hidden dynamic block with four neurons. The connections between them are as depicted. Each neuron emulates a ODE solver to solve the system equations by means of dynamic connections and trainable parameters τ_1, τ_2, $w_{\hat{u}}$ and $w_{\tilde{\xi}}$.

DNN - Output

As visible from the curve fits in the figure 4.2, the model again performs really well and gives us MSE values very close to machine precision when compared with a Backward Euler solver, and reasonably good values when compared to the python routine *signal.lsim*. For the sake of completeness, after playing around with the hyperparameters and performing high level

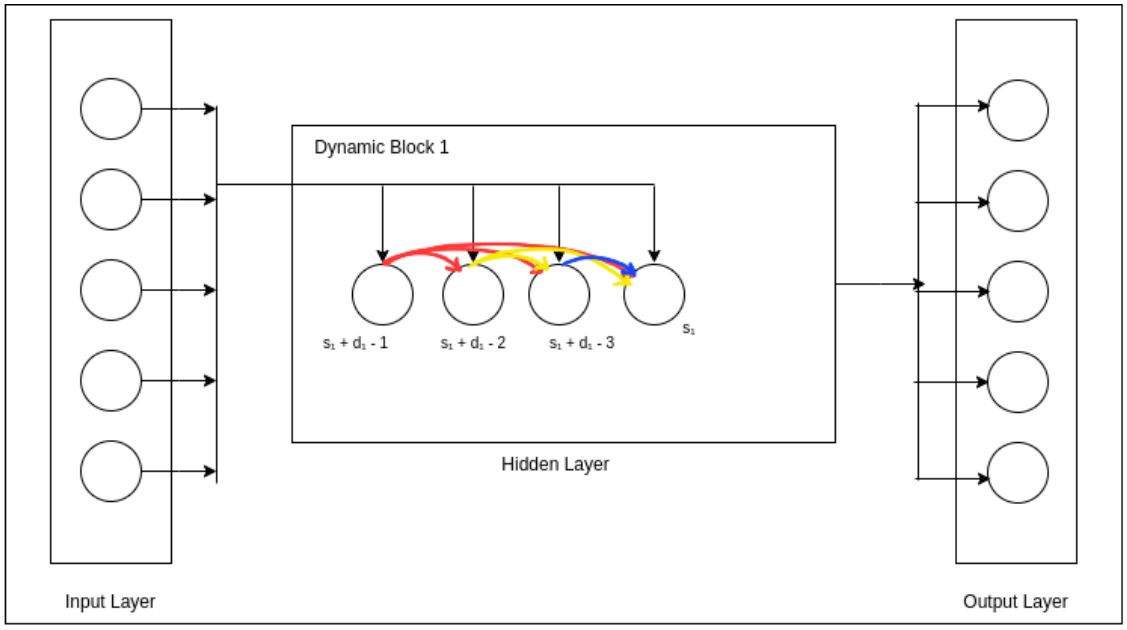

Figure 4.7.: The DNN architecture - 4x4 LTI system

hyperparameter tuning, we got the best results when using a ADAM (Adaptive Moment Estimation) optimizer with a learning rate between $1e-3$ and $1e-2$.

Loss curve

We again plot the loss curve for this example just like we did for the 2×2 system. As seen from the curve in figure 4.9, we observe that the training was smooth, which indicates that there were no significant problems with the training and the model is behaving as expected as it is steadily improving over time. An interesting thing to note here is that because the training is slightly slower than the simpler 2×2 case, it takes more number of epochs for the curve to converge. Also, the curve converges at a higher loss value as compared to the 2×2 example.

Dataset C : 16x16 system with algebraic multiplicity 16

DNN - Mapping

In this dataset, the matrices $A \in R^{16\times16}, B \in R^{16\times16}, C \in R^{16\times16}$ and $D \in R^{16\times16}$ are the system matrices, more specifically, A being the state matrix, B being the input matrix, C being the output matrix and D being the feedforward matrix. Here, the system matrix A has real and repeating eigen values with an algebraic multiplicity of 16. The matrices B and C are Identity matrices and D is a zero or a null matrix. As usual, we give a sinusoidal input pulse

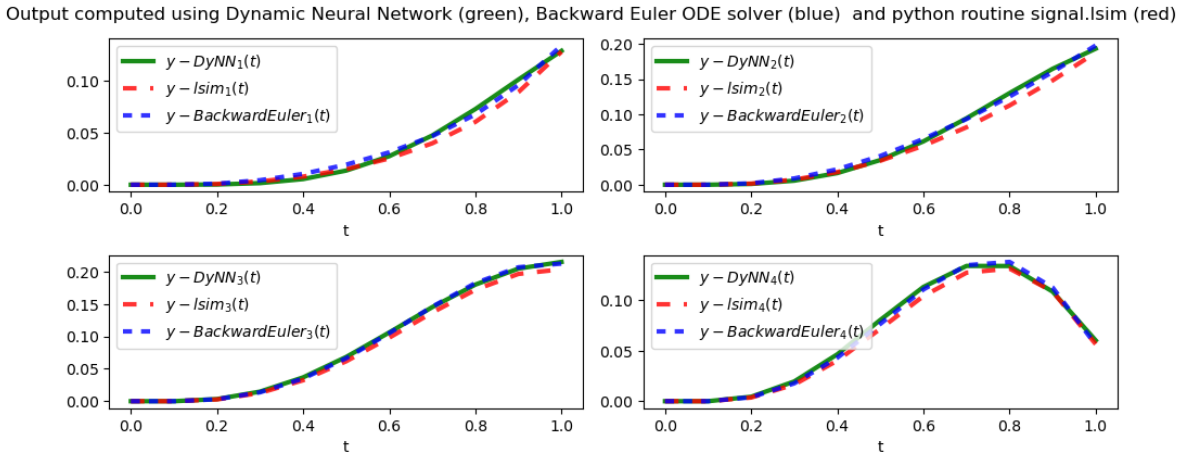

Figure 4.8.: Model Response for the 4x4 LTI system

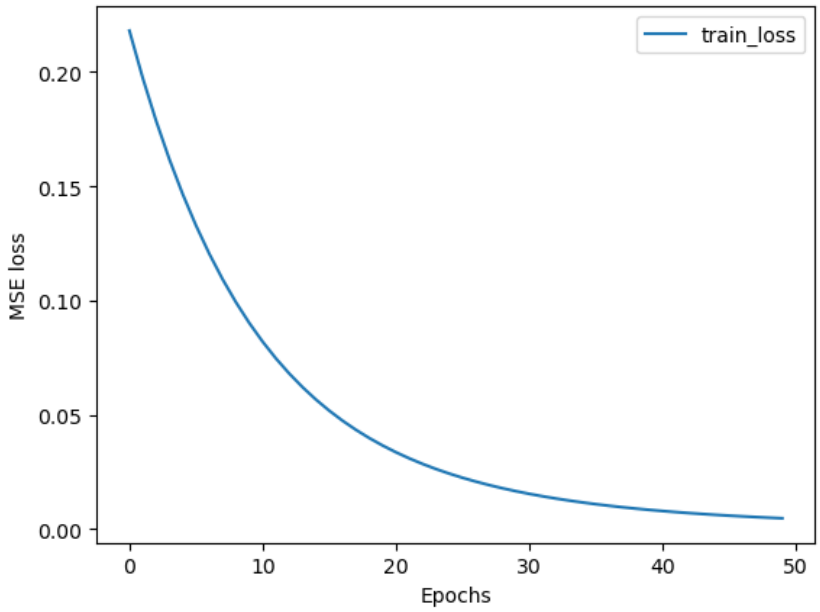

Figure 4.9.: The Loss curve - 4x4 LTI system

$u(t)$ to the system.

In this dataset, the matrices $A \in R^{16\times16}, B \in R^{16\times16}, C \in R^{16\times16}$ and $D \in R^{16\times16}$ are the system matrices. Here, the system matrix A has real and repeating eigen values with an algebraic multiplicity of 16. The matrices B and C are Identity matrices and D is a zero or a null matrix. We give a sinusoidal input pulse $u(t)$ to the system.

For this configuration, we derive the mapping of the DNN using the steps mentioned earlier to get the DNN architecture as described in figure 4.10. In this architecture, we have a single hidden dynamic block with four neurons. The connections between them are as depicted. Each neuron emulates a ODE solver to solve the system equations by means of dynamic connections and trainable parameters τ_1, τ_2, $w_{\hat{u}}$ and $w_{\tilde{\xi}}$.

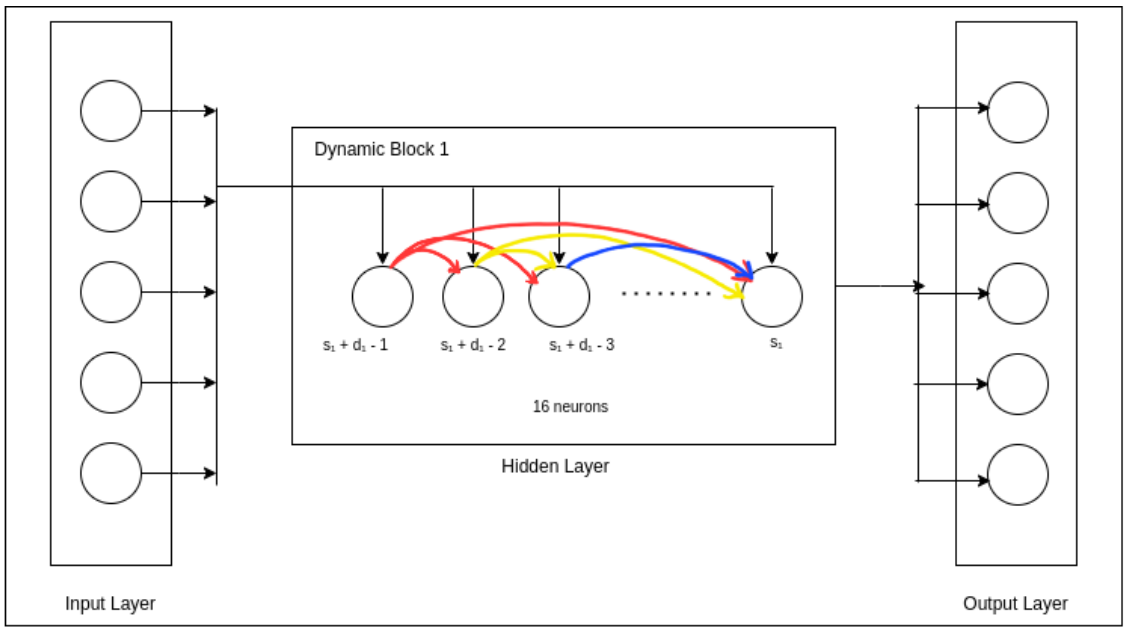

Figure 4.10.: The DNN architecture - 16x16 LTI system

DNN - Output

As visible from the curve fits in the figure 4.2, the model performance is not as good as the previous two cases, giving us MSE values in the range of $1e-4$. For the sake of completeness, after playing around with the hyperparameters and performing high level hyperparameter tuning, we again got the best results when using a ADAM (Adaptive Moment Estimation) optimizer with a learning rate between $1e-3$ and $1e-2$.

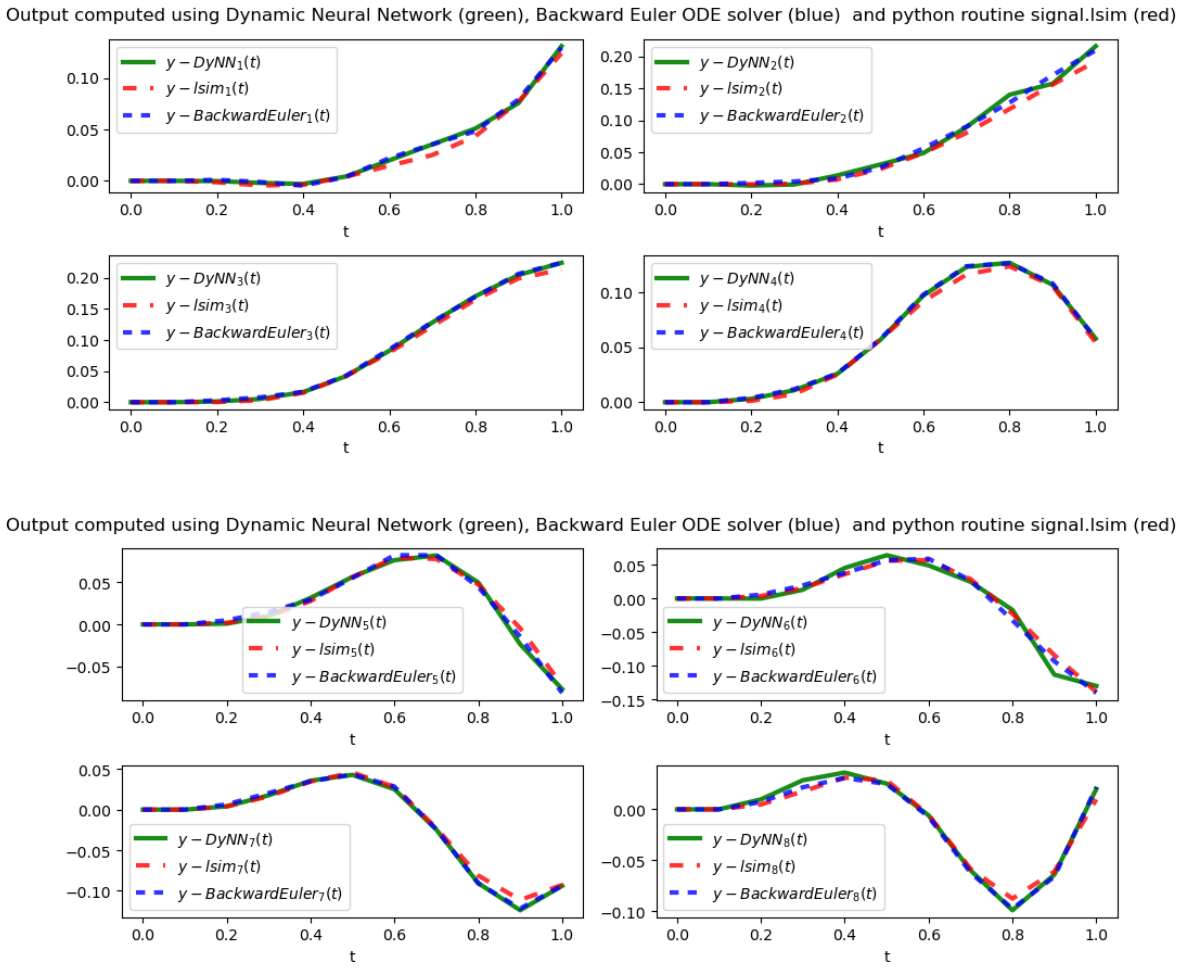

Figure 4.11.: Model Response for the 16x16 system - I

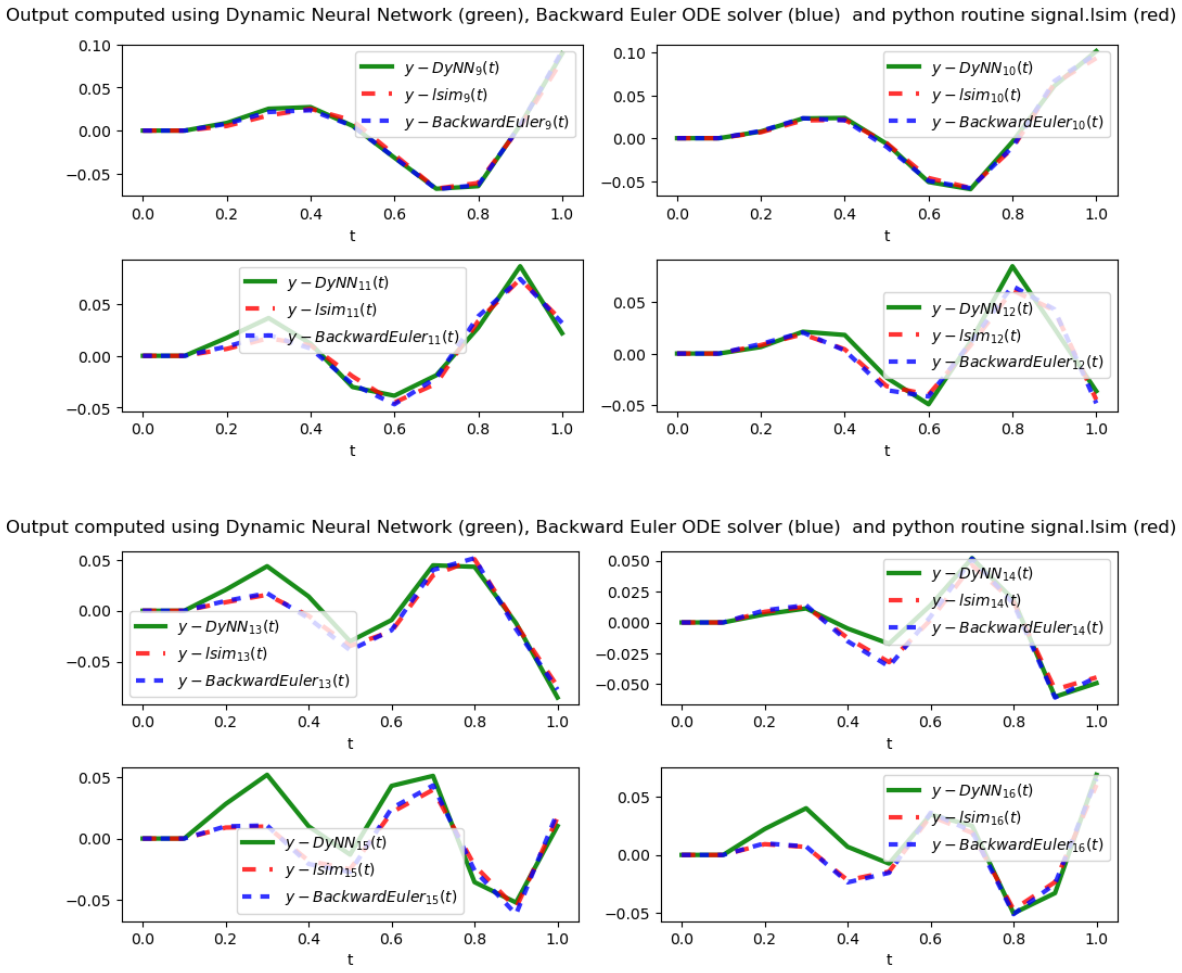

Figure 4.12.: Model Response for the 16x16 system - II

Loss curve

We repeat the step of plotting the loss curve for this example just like we did earlier. As seen from the curve in figure 4.13, we observe that the training was smooth, which again indicates that there were no significant problems with the training and the model is behaving as expected as it is steadily improving over time. The overall loss value at which the curve converges to a much higher value as expected.

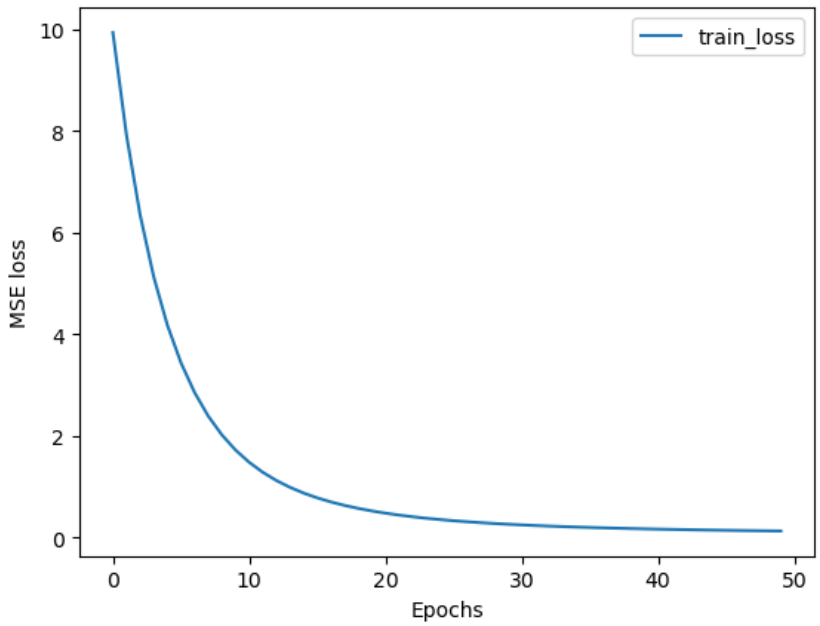

Figure 4.13.: The Loss curve - 16x16 LTI system

Cascaded DNNs

DNN - Mapping

In the next scenario, we assume that we have no prior knowledge of the topology of the DNN, and hence we just naively connect individual dynamic blocks together as we would otherwise conventionally do with Artificial Neural Networks. We use this architecture and train it on our 4x4 dataset. We cascade multiple 1×1 dynamic neuron blocks in multiple vertically stacked hidden layers to achieve similar performance throughput as compared to a regular DNN, if not better. We can cascade arbitrary number of such vertically stacked dynamic blocks, each made up of a single $1x1$ neuron. The choice on the numbers of hidden layers here is arbitrary because it is not possible to isolate a "best" size and depth for a network before continuing to tune other parameters in isolation as the size and depth of the network can be thought of as hyperparameters themselves. We will train our cascaded DNN on dataset B. The system ma-

trices of B will first be decoupled and mapped to a number of individual 1×1 dynamic blocks.

For this configuration, we cascade three vertically stacked dynamic blocks as described in figure 4.14. In each vertically stacked layer, we have a four different dynamic blocks. The connections between them are as depicted. Each neuron emulates a ODE solver to solve the system equations by means of dynamic connections and trainable parameters τ_1, τ_2, $w_{\hat{u}}$ and $w_{\tilde{\zeta}}$.

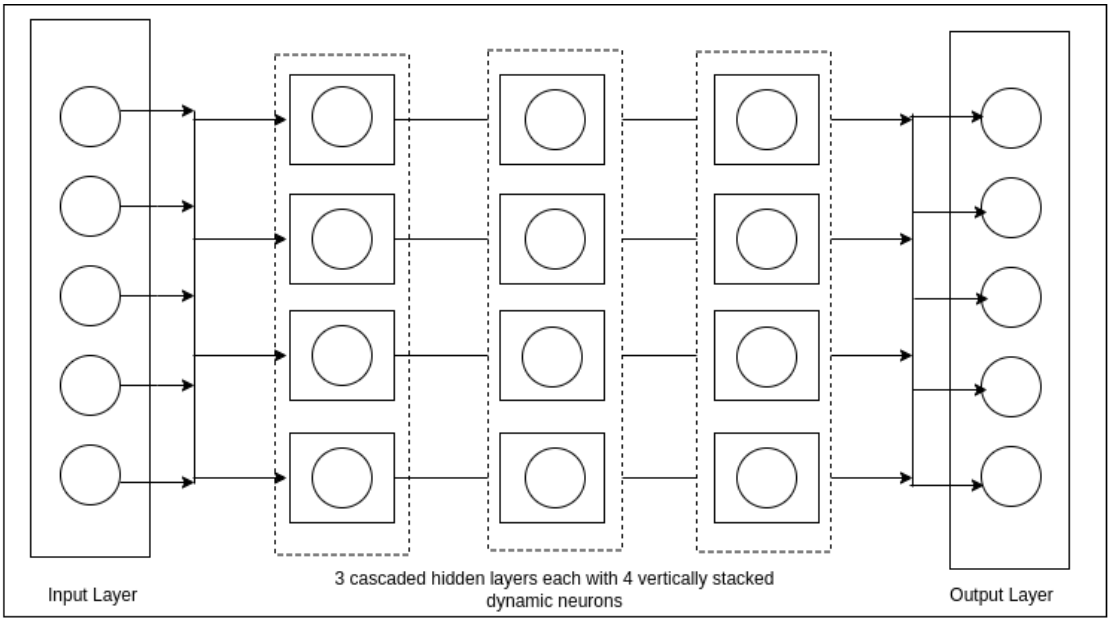

Figure 4.14.: Cascaded DNN - Concept Diagram

DNN - Output

Dataset B is a 4×4 dynamical system. The cascaded DNN, in theory, has more number of trainable units, which may or may not translate to better expressiveness. As visible from the curve fits in the figures 4.15 and 4.16, the model performance is not as good as the uncascaded DNNs, giving us MSE values in the range of $1e-2$. For the sake of completeness, after playing around with the hyperparameters and performing high level hyperparameter tuning, we again got the best results when using a ADAM (Adaptive Moment Estimation) optimizer with a learning rate between $5e-3$ and $1e-2$.

Loss curve

The loss curve for the cascaded DNN is as shown in figure 4.17. We observe that the model performance was not as good as the regular DNNs, but there were no apparent indications

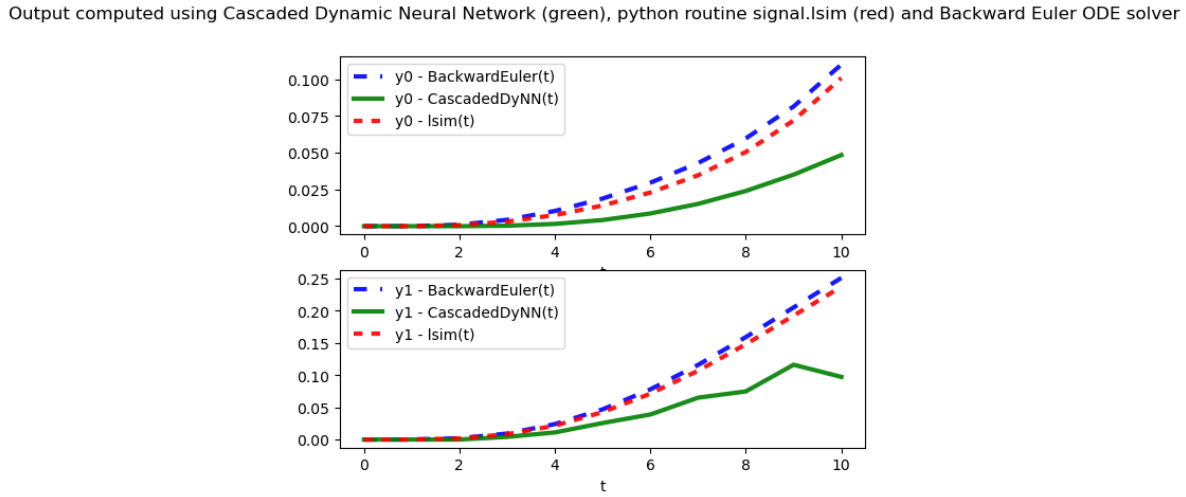

Figure 4.15.: Cascaded DNN - Model Output - I

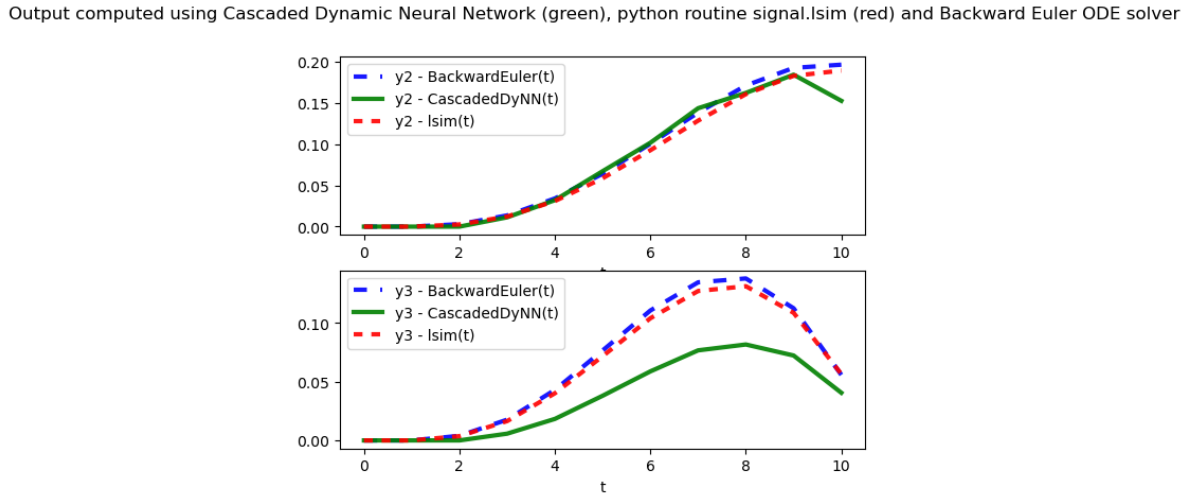

Figure 4.16.: Cascaded DNN - Model Output - II

LTI system	Learning Rate	Loss at convergence
Dataset A - 2x2 system	(1e-3,1e-2)	1e-8
Dataset B - 4x4 system	(1e-3,1e-2)	1e-7
Dataset C - 16x16 system	(1e-3,1e-2)	1e-4
Dataset B - Cascaded DNN	(1e-3,1e-2)	1e-2

Table 4.1.: Summary of Results

on why would this be the case. Our intuition is that the 1×1 individual dynamic blocks in each layer are less expressive as compared to the 4×4 dynamic blocks. Moreover, more number of neurons overall mean more trainable parameters, and hence a more complex loss landscape, which could very well be difficult to navigate. The overall loss value at which the curve converges to a much higher value than expected.

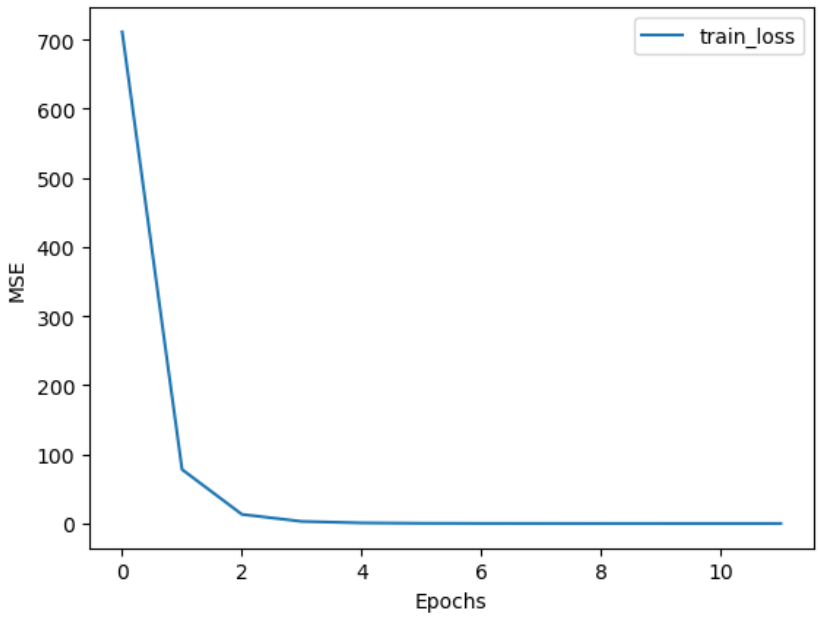

Figure 4.17.: Cascaded DNN - Loss Curve

5. Conclusion and Outlook

The theory of Neural Networks and its extensions have shown great promise when it comes to modelling complex dynamical systems. In this thesis, we extended this research direction by implementing Dynamic Neural Networks - DNNs - in which the behaviour of individual neurons is characterized by a suitably designed differential equation. This differential equation represents a non-linearity, for which appropriate choices had to be made to allow for the accurate and efficient representation of the system in question. Several relevant examples of dynamic behaviour have also been demonstrated to fit the mathematical structure of DNNs, although not all kinds of dynamic behaviour are considered representable at this point in time.

By visualizing and studying the loss landscapes of these systems, we were able to intuitively come up with initialization schemes for various parameters of the DNNs so as to reduce the chance of our DNN optimizer getting stuck in a local minimum. For stability reasons it is assumed that two of the trainable parameters τ_1 and τ_2 are greater than zero. Further, our implementation assumes again that the eigenvalues of A are real and have algebraic multiplicity of greater than 1. When mapping the system matrices onto the parameters of the network, it is desirable that stability is maintained.

A significant portion of this thesis was the Pytorch implementation of the DNN formalism. This implementation will serve as the basis for future research and work in this direction. Several design choices were made while writing the code from a software engineering point of view to ensure modularity and reproducability. The code was well documented for future researchers to continue working on other extensions of DNN. On top of using the inbuilt python routine scipy.signal.lsim, we also implemented two other ODE solvers - Backward Euler and the Trapezoidal solver to better align with the DNN implementation.

Key takeaways from this study:

1. DNNs already show a lot of promise when it comes to representing and emulating LTI systems. The domain knowledge of state space representation of LTI systems can be exploited to initialize the weights of DNNs.

2. Not only that, but the mapping from the state space matrices to the parameters of the DNN is very useful in that it gives us information about the topology of DNNs which more or less eliminates the need for architecture search.

3. The standard backpropagation theory for static multidimensional behaviour in feedforward neural networks can been extended to include the learning of dynamic response

in the time domain as is demonstrated by the implementation of Backpropagation Through Time (BPTT)

4. Analyzing the loss landscape of some of these LTI systems gave us more insights on why training a neural network to get to learn the representation is not particularly easy. However, the shallow slopes around the region of theoretical global minima does suggest that it should be possible to be able to arrive at an approximate solution. Studying the loss surfaces of even higher dimensional systems could be potentially very interesting.

5. Although the results don't match the performance of regular standalone DNNs, cascading lower order DNNs together could potentially prove useful for certain scenarios.

In addition to this, there are a few different propositions on the future direction(s) of research in this domain:

1. Implement the DNN formalism for systems where the state matrix A has complex eigen values. This will involve extending the current implementation, which should be trivial, but it couldn't be done because of time constraints.

2. We currently only looked at equidistantly sampled datasets. It could be an interesting exercise to be able to learn using non-equidistantly sampled datasets.

3. The convergence of the optimization scheme while training DNNs was observed to be not particularly fast, especially considering that a few of the examples were very simplified. A deeper insight into why is it slow and working on accelerating it would be most valuable.

4. In spite of several reasons for caution, the general direction in Automatic Neural Architecture Search as proposed in this thesis seems to have significant potential. However, it must at the same time be emphasized that there may still be a long way to go from encouraging preliminary results to practically useful results.

A. General Addenda

A.1. Multivariable Subspace Identification: MOESP Algorithm

The Multi-Output Error State Parameter (MOESP) algorithm [34] is a powerful subspace identification technique employed in the field of system identification and control. Designed to address the challenges of estimating the state-space matrices of Linear Time-Invariant (LTI) systems from input-output data, MOESP excels particularly in scenarios involving multiple outputs. By leveraging concepts from subspace identification and employing Singular Value Decomposition (SVD) on carefully constructed Hankel matrices [35] derived from collected data, MOESP offers a robust and effective approach for extracting the underlying dynamics of complex systems. This algorithm plays a pivotal role in fields where accurate modeling of dynamical systems is crucial, providing valuable insights into the behavior and characteristics of the underlying processes. Its versatility, especially in handling multivariate systems, has contributed to its widespread use in diverse applications ranging from control systems to signal processing.

Consider a Linear Time-Invariant (LTI) system described by the state-space equations:

$$\dot{x}(t) = Ax(t) + Bu(t)$$
$$y(t) = Cx(t) + Du(t)$$

where:

$x(t)$ is the state vector,
$u(t)$ is the input vector,
$y(t)$ is the output vector,
A is the state matrix,
B is the input matrix,
C is the output matrix, and
D is the direct transmission matrix.

The MOESP algorithm for identifying the system matrices involves the following steps:

1. Data Collection:

Collect input-output data pairs $(u(t), y(t))$ for a given time duration.

2. Data Matrices:

Assemble the Hankel matrices:

$$Y_p = \begin{bmatrix} y(1) & y(2) & \dots & y(p) \\ y(2) & y(3) & \dots & y(p+1) \\ \vdots & \vdots & \ddots & \vdots \\ y(N-p+1) & y(N-p+2) & \dots & y(N) \end{bmatrix}$$

$$U_p = \begin{bmatrix} u(1) & u(2) & \dots & u(p) \\ u(2) & u(3) & \dots & u(p+1) \\ \vdots & \vdots & \ddots & \vdots \\ u(N-p+1) & u(N-p+2) & \dots & u(N) \end{bmatrix}$$

where p is the past window size, and N is the total number of samples.

3. SVD (Singular Value Decomposition):

Perform Singular Value Decomposition on the augmented matrix:

$$\begin{bmatrix} Y_p \\ U_p \end{bmatrix} = \hat{U}\Sigma\hat{V}^T$$

where $\hat{U}$ and $\hat{V}$ are orthogonal matrices, and Σ is a diagonal matrix of singular values.

4. System Matrices:

Extract submatrices from $\hat{U}$, Σ, and $\hat{V}$ to form matrices U_1, U_2, Y_1, Y_2, and Y_3. Compute the system matrices using the following relations:

$$A = U_1 Y_1^+$$

$$B = U_1 Y_2^+$$

$$C = Y_3^+ U_1^T$$

$$D = Y_3^+ U_2^T$$

where $^+$ denotes the pseudoinverse.

The resulting matrices A, B, C, and D are estimates of the state-space matrices for the given LTI system.

List of Figures

List of Tables

Bibliography

[1] P. Meijer. "Neural Networks for Device and Circuit Modelling". In: Aug. 2000.

[2] R. E. Kalman. "A New Approach to Linear Filtering and Prediction Problems". In: *Journal of Basic Engineering* 82.1 (Mar. 1960), pp. 35–45. ISSN: 0021-9223. DOI: 10.1115/1.3662552. eprint: https://asmedigitalcollection.asme.org/fluidsengineering/article-pdf/82/1/35/5518977/35_1.pdf. URL: https://doi.org/10.1115/1.3662552.

[3] A. P. Dempster, N. M. Laird, and D. B. Rubin. "Maximum Likelihood from Incomplete Data via the EM Algorithm". In: *Journal of the Royal Statistical Society. Series B (Methodological)* 39.1 (1977), pp. 1–38. ISSN: 00359246. URL: http://www.jstor.org/stable/2984875 (visited on 01/03/2024).

[4] S. Chen and S. A. Billings. "Representation of non-linear systems: the NARMAX model". In: *International Journal of Control* 49.3 (Mar. 1989). Address: London, pp. 1012–1032. URL: https://eprints.soton.ac.uk/251145/.

[5] S. L. Brunton, J. L. Proctor, and J. N. Kutz. "Discovering governing equations from data by sparse identification of nonlinear dynamical systems". In: *Proceedings of the National Academy of Sciences* 113.15 (2016), pp. 3932–3937. DOI: 10.1073/pnas.1517384113. eprint: https://www.pnas.org/doi/pdf/10.1073/pnas.1517384113. URL: https://www.pnas.org/doi/abs/10.1073/pnas.1517384113.

[6] Z. C. Lipton, J. Berkowitz, and C. Elkan. "A Critical Review of Recurrent Neural Networks for Sequence Learning". In: (2015). DOI: 10.48550/ARXIV.1506.00019. URL: https://arxiv.org/abs/1506.00019.

[7] J. L. Elman. "Finding structure in time". In: *Cognitive Science* 14.2 (1990), pp. 179–211. ISSN: 0364-0213. DOI: https://doi.org/10.1016/0364-0213(90)90002-E. URL: https://www.sciencedirect.com/science/article/pii/036402139090002E.

[8] W. Schilders. "Predicting the topology of dynamic neural networks for the simulation of electronic circuits". English. In: *Neurocomputing* 73.1-3 (2009), pp. 127–132. ISSN: 0925-2312. DOI: 10.1016/j.neucom.2009.06.011.

[9] V. I. Arnold. "Henri Poincaré: Selected Works in Three Volumes. Vol. I New Methods of Celestial Mechanics - Preface. From the editorial board. Comments". In: *Vladimir I. Arnold - Collected Works: Hydrodynamics, Bifurcation Theory, and Algebraic Geometry 1965-1972*. Ed. by A. B. Givental, B. A. Khesin, A. N. Varchenko, V. A. Vassiliev, and O. Y. Viro. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 457–461. ISBN: 978-3-642-31031-7. DOI: 10.1007/978-3-642-31031-7_44. URL: https://doi.org/10.1007/978-3-642-31031-7_44.

[10] R. Brown. *A Modern Introduction to Dynamical Systems*. June 2018. ISBN: 9780198743286.

[11] "Introduction to Ordinary Differential Equations". In: *Ordinary Differential Equations with Applications*. New York, NY: Springer New York, 1999, pp. 1–126. ISBN: 978-0-387-22623-1. DOI: 10.1007/978-0-387-22623-1_1. URL: https://doi.org/10.1007/978-0-387-22623-1_1.

[12] P. Holmes. "Poincaré, celestial mechanics, dynamical-systems theory and "chaos"". In: *Physics Reports 193.3 (1990): 137–163* (1990).

[13] P. C. PARKS. " A. M. Lyapunov's stability theory—100 years on *". In: *IMA Journal of Mathematical Control and Information* 9.4 (Dec. 1992), pp. 275–303. ISSN: 0265-0754. DOI: 10.1093/imamci/9.4.275. eprint: https://academic.oup.com/imamci/article-pdf/9/4/275/6767220/9-4-275.pdf. URL: https://doi.org/10.1093/imamci/9.4.275.

[14] G. D. Birkhoff. *Proof of Poincaré's geometric theorem*. en. 1913. DOI: 10.1090/s0002-9947-1913-1500933-9. URL: http://dx.doi.org/10.1090/S0002-9947-1913-1500933-9.

[15] S. Smale. *Differentiable dynamical systems*. en. 1967. DOI: 10.1090/s0002-9904-1967-11798-1. URL: http://dx.doi.org/10.1090/S0002-9904-1967-11798-1.

[16] K. Burns and B. Hasselblatt. "The Sharkovsky Theorem: A Natural Direct Proof". In: *The American Mathematical Monthly* 118.3 (2011), pp. 229–244. DOI: 10.4169/amer.math.monthly.118.03.229. eprint: https://www.tandfonline.com/doi/pdf/10.4169/amer.math.monthly.118.03.229. URL: https://www.tandfonline.com/doi/abs/10.4169/amer.math.monthly.118.03.229.

[17] G. Rega. "Tribute to Ali H. Nayfeh (1933–2017)". In: *IUTAM Symposium on Exploiting Nonlinear Dynamics for Engineering Systems*. Ed. by I. Kovacic and S. Lenci. Cham: Springer International Publishing, 2020, pp. 1–13. ISBN: 978-3-030-23692-2.

[18] A. Katok and B. Hasselblatt. *Introduction to the Modern Theory of Dynamical Systems*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 1995. DOI: 10.1017/CBO9780511809187.

[19] E. Ott. *Chaos in Dynamical Systems*. 2nd ed. Cambridge University Press, 2002. DOI: 10.1017/CBO9780511803260.

[20] K. Hangos, J. Bokor, and G. Szederkenyi. "Analysis and Control of Nonlinear Process Systems". In: Jan. 2004.

[21] A. Vasilyev and A. Ushakov. *Modeling of dynamic systems with modulation by means of Kronecker vector-matrix representation*. Aug. 2015. DOI: 10.17586/2226-1494-2015-15-5-839-848. URL: http://dx.doi.org/10.17586/2226-1494-2015-15-5-839-848.

[22] D. Griffiths and D. Higham. *Numerical Methods for Ordinary Differential Equations: Initial Value Problems*. Jan. 2010. ISBN: 978-0-85729-147-9. DOI: 10.1007/978-0-85729-148-6.

[23] E. Hairer, S. Norsett, and G. Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems*. Vol. 8. Jan. 1993. ISBN: 978-3-540-56670-0. DOI: 10.1007/978-3-540-78862-1.

[24] L. Skvortsov. "Diagonally Implicit Runge–Kutta Methods for Stiff Problems". In: *Computational Mathematics and Mathematical Physics* 46 (Dec. 2006), pp. 2110–2123. DOI: 10.1134/S0965542506120098.

[25] L. Ferracina and M. Spijker. "Strong stability of singly-diagonally-implicit Runge-Kutta methods". In: *Applied Numerical Mathematics J. Sci. Computing Numer. Anal. J. Sci. Computing Numer. Anal. Math. Comp. Spiteri and Ruuth [R.J. Spiteri, S.J. Ruuth, SIAM J. Numer. Anal. Math. Comput. Simulation* 58 (Nov. 2008), pp. 1675–1686. DOI: 10.1016/j.apnum.2007.10.004.

[26] E. Everhart. "An efficient integrator that uses Gauss-Radau spacings". In: *International Astronomical Union Colloquium* 83 (1985), pp. 185–202. DOI: 10.1017/S0252921100083913.

[27] E. W. Weisstein. ""Gaussian Quadrature."" In: *From MathWorld–A Wolfram Web Resource.* ().

[28] R. H. Bartels and G. W. Stewart. "Algorithm 432 [C2]: Solution of the Matrix Equation AX + XB = C [F4]". In: *Commun. ACM* 15.9 (Sept. 1972), pp. 820–826. ISSN: 0001-0782. DOI: 10.1145/361573.361582. URL: https://doi.org/10.1145/361573.361582.

[29] D. R. Kincaid and E. W. Cheney. *Numerical Analysis: Mathematics of Scientific Computing*. Pacific Grove, Calif.: Brooks/Cole, 1991. ISBN: 0534130143 9780534130145.

[30] E. Leelarasmee. "A Linear Electronic Circuit Analysis Progam". In: *IFAC Proceedings Volumes* 18.9 (1985). IFAC/IFORS Symposium on Control Science and Technology for Development, Beijing, PRC, 20-22 August 1985, pp. 207–210. ISSN: 1474-6670. DOI: https://doi.org/10.1016/S1474-6670(17)60286-6. URL: https://www.sciencedirect.com/science/article/pii/S1474667017602866.

[31] P. Werbos. "Backpropagation through time: what it does and how to do it". In: *Proceedings of the IEEE* 78 (Nov. 1990), pp. 1550–1560. DOI: 10.1109/5.58337.

[32] D. Bertsekas. "Approximate policy iteration: A survey and some new methods". In: *Journal of Control Theory and Applications* 9 (Aug. 2011), pp. 310–335. DOI: 10.1007/s11768-011-1005-3.

[33] Y. Dauphin, R. Pascanu, C. Gulcehre, K. Cho, S. Ganguli, and Y. Bengio. *Identifying and attacking the saddle point problem in high-dimensional non-convex optimization*. 2014. arXiv: 1406.2572 [cs.LG].

[34] M. VERHAEGEN and P. DEWILDE. "Subspace model identification Part 2. Analysis of the elementary output-error state-space model identification algorithm". In: *International Journal of Control* 56.5 (1992), pp. 1211–1241. DOI: 10.1080/00207179208934364. eprint: https://doi.org/10.1080/00207179208934364. URL: https://doi.org/10.1080/00207179208934364.

[35] M. A. Arbib and E. G. Manes. "Foundations of system theory: The hankel matrix". In: *Journal of Computer and System Sciences* 20.3 (1980), pp. 330–378. ISSN: 0022-0000. DOI: https://doi.org/10.1016/0022-0000(80)90012-4. URL: https://www.sciencedirect.com/science/article/pii/0022000080900124.