Parametric SINDy-SHRED: Combining Discovery and Prediction for Complex PDE Dynamics

Gregorio Coletti

February 6, 2025

Abstract

This work presents a hybrid framework, **parametric SINDy-SHRED**, that combines the **SINDy** (Sparse Identification of Nonlinear Dynamics) method with a **SHRED** (SHallow Recurrent DEcoder) model to address the dual challenge of discovering and predicting dynamics governed by complex partial differential equations (PDEs). The proposed approach enables accurate and efficient data-driven analysis and forecasting. The methodology, applications, and practical implementation are discussed in detail, highlighting the model's robustness and applicability. The results indicate that the model generalizes well for both unseen parameters configurations and unseen time instants, with low approximation errors for both SINDy predictions of the dynamics in the latent space and SHRED reconstruction in the physical space. The identified governing equations in the latent state offer an additional layer of interpretability, making the model more robust and more reliable for scientific applications.

Contents

1 Introduction

The study of partial differential equations (PDEs) is fundamental in understanding a wide range of phenomena across physics, biology, and engineering. These equations describe the evolution of dynamical systems, yet their analytical solutions are rarely accessible. Data-driven approaches have emerged as a powerful tool to bridge this gap by enabling the discovery and prediction of PDE dynamics directly from observed data.

In this work, we introduce **parametric SINDy-SHRED**, a hybrid framework that integrates the **S**parse **I**dentification of **N**onlinear **Dy**namics (**SINDy**) method with a **SH**allow **RE**current **D**ecoder model (**SHRED**). The proposed approach leverages the strengths of both frameworks to provide an efficient and accurate pipeline for analyzing spatio-temporal systems.

This report is structured as follows.

Section 2 : Formulates the problem and outlines the objectives.

Section 3 : Presents the algorithms and methodologies employed.

Section 4 : Describes the implementation workflow, including code structure.

Section 5 : Outlines the test cases and scenarios used for evaluation.

Section 6 : Discusses the numerical results and their implications.

Section 7 : Provides concluding remarks and potential future directions.

Section 8 : Includes a tutorial for replicating and extending the work.

The goal of this work is twofold:

1. **Prediction:** Reconstruct the full spatiotemporal evolution of the system based on limited sensor data and a few known parameters, enabling accurate forecasts across the entire domain.

2. **Discovery:** Identify the underlying governing equations to make the method more robust and reliable and gain a deeper understanding of the system's intrinsic dynamics given some control parameters.

Achieving these goals requires a hybrid approach that combines data-driven techniques with physical insights. By integrating the SINDy framework with the SHRED model, this methodology addresses the dual challenge of discovery and prediction, ensuring both accuracy and interpretability.

2 Problem Formulation

Consider a dynamical system described by the generic form:

$$\begin{cases} \dot{\mathbf{x}}(t, \boldsymbol{\beta}) = \mathbf{f}(\mathbf{x}(t, \boldsymbol{\beta}), t, \boldsymbol{\beta}), & \text{for } \mathbf{x} \in \Omega,\ t \in [0, T], \\ \\ \mathbf{x}(0, \boldsymbol{\beta}) = \mathbf{x}_0, & \text{for } \mathbf{x}_0 \in \Omega. \end{cases} \tag{1}$$

where $\mathbf{x}$ represents the state variables, $\mathbf{f}(\mathbf{x})$ denotes the governing dynamics, $\mathbf{x}_0$ specifies the initial conditions, $\boldsymbol{\beta}$ represents the vector of parameters or forcing terms.

- $\mathbf{x} \in \mathbb{R}^{N_f}$, where N_f is the state dimensionality of the system.
- $\mathbf{f} : \mathbb{R}^{N_f+1+p} \to \mathbb{R}^{N_f}$

- $\boldsymbol{\beta} \in \mathbb{R}^p$, where p represents the number of distinct parameter sets (equivalent to the number of systems being modeled).

These systems often evolve over a spatio temporal domain, making their analysis both critical and challenging. The importance of capturing both spatial and temporal variations lies at the heart of many real-world applications, including fluid dynamics, reaction-diffusion systems, and weather prediction.

When discretizing, the formulation changes to: $\mathbf{x} \in \mathbb{R}^N$, where $N = N_d N_f$ with N_d the number of degrees of freedom obtained when discretizing the domain Ω

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}(t_1;\boldsymbol{\beta_1}) & \cdots & \mathbf{x}(T;\boldsymbol{\beta_1}) & | & \mathbf{x}(t_1;\boldsymbol{\beta_2}) & \cdots & \mathbf{x}(T;\boldsymbol{\beta_2}) & | \cdots & \mathbf{x}(T;\boldsymbol{\beta_p}) \end{bmatrix},$$

$$\dot{\mathbf{X}} = \begin{bmatrix} \dot{\mathbf{x}}(t_1;\boldsymbol{\beta_1}) & \cdots & \dot{\mathbf{x}}(T;\boldsymbol{\beta_1}) & | & \dot{\mathbf{x}}(t_1;\boldsymbol{\beta_2}) & \cdots & \dot{\mathbf{x}}(T;\boldsymbol{\beta_2}) & | \cdots & \dot{\mathbf{x}}(T;\boldsymbol{\beta_p}) \end{bmatrix},$$

Normally N is a very large number, and access to full spatial information is only possible when dealing with synthetic data. However, in practical scenarios, information about the system is frequently limited to a few sensor measurements. These sparse observations introduce significant challenges.

- **Limited spatial coverage:** Measurements are restricted to a small subset of the spatial domain.
- **High-dimensional dynamics:** The full system dynamics must be inferred from minimal data.

Aiming to generalize the model to a more realistic context, we will use a much smaller dimension $n_s << N$ representing the number of sensors that could potentially be set in the real world.

3 Algorithm

The algorithm is based on a combination of **SHRED** (**SH**allow **RE**current **D**ecoder) and **SINDy** (**S**parse **I**dentification of **N**on-Linear **Dy**namics), first suggested from [3]. **SHRED**, first developed in [2], is a neural network that is obtained sequencing a Recurrent Unit and Shallow Decoder with only a couple of hidden layers. So far it has been used to study dynamical systems with limited data measurements. **SINDy** is a dynamic system identification, in the form of 1, based on a sparsed regression and predetermined library of guess functions.

First, we show a brief explanation of both methods separately. Next, we explain the joint algorithm and the novelty brought by this project.

3.1 SHRED

Following the example set by [2], we will use a GRU unit (Generative Recurrent Unit) as recurrent unit. Mathematically, the GRU unit takes a lagged input, from the sensors subset S, $\{\mathbf{s}_{t-L:t}\}_{t=0}^{T}$ and outputs $\{\mathbf{z}_t\}_{t=0}^{T}$, where L represents the added lag.

$$\{\mathbf{z}_t\}_{t=0}^{T} = \mathcal{G}(\{\mathbf{s}_{t-L:t}\}_{t=0}^{T}) \tag{2}$$

Then a Decoder decompresses the hidden representation $\mathbf{z}_t$ to the full dimension of the degrees of freedom N. Now consider one instant t.

$$\tilde{\mathbf{x}}_t = \mathcal{D}(\mathbf{z}_t) \tag{3}$$

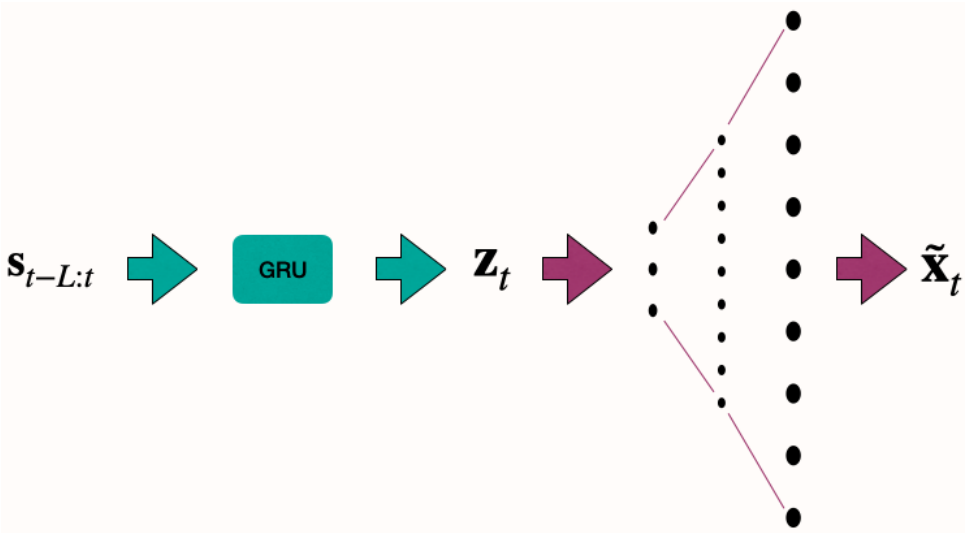


Figure 1: SHRED Architecture for each instant t.

To clarify: $\mathbf{s}_t \in \mathbb{R}^{n_s}$, $\mathbf{z}_t \in \mathbb{R}^h$; with n_s = number of sensors and h hidden size, $\tilde{\mathbf{x}}_t \in \mathbb{R}^N$.

$\tilde{\mathbf{x}}$ is what actually approximates the original data $\mathbf{x}$ for all time steps. So for k = number of iterations, we expect $\tilde{\mathbf{x}} \longrightarrow \mathbf{x}$ for $k \longrightarrow \infty$.

As fully explained by Williams and his collaborators in[2], the assumption justifying the architecture of SHRED is the fact that eventually the information will flow across the domain and be captured from the sensors, so if you accumulate enough measurements time-wise in a few places, you can predict the solution space-wise. The Lag is implemented to ensure that there is available information before starting to predict the solution. Beware that the method can only interfere the solution in statistically dependent regions.

The GRU unit can be seen as both an encoder and decoder of temporal extended data, depending on the comparison between the input size n_s and the hidden size h. Generally for a standard **SHRED** $h > n_s$, however to combine it with **SINDy** we will use a smaller h making the Recurrent Unit behave as an encoder.

The SHRED algorithm has demonstrated exceptional capability in replicating data with minimal initial information. As evidenced by the prior study, it successfully reconstructed a domain comprising approximately 44,000 degrees of freedom utilizing merely three sensors. However, despite the promising results, the model doesn't unravel the complexity of the underlaying dynamic in play.

3.2 SINDy

A **SINDy** model approximates the governing function $\mathbf{f}(\mathbf{x}, \boldsymbol{\beta})$ of the dynamical system:

$$\dot{\mathbf{x}}(t, \boldsymbol{\beta}) = \mathbf{f}(\mathbf{x}(t, \boldsymbol{\beta}), t, \boldsymbol{\beta}),$$

as:

$$\mathbf{f}(\mathbf{x}(t, \boldsymbol{\beta}), t, \boldsymbol{\beta}) \approx \Theta(\mathbf{x}(t, \boldsymbol{\beta}), \boldsymbol{\beta})\Xi,$$

where $\Theta(\mathbf{x}, \boldsymbol{\beta}) \in \mathbb{R}^{m \times l}$ is the library of candidate functions, with m = number of measurements, and l = number of library functions and $\Xi \in \mathbb{R}^{l \times h}$ contains the coefficients that activate each function, with h = number of hidden states. $\mathbf{X}$(t) is stored in a matrix:

$$\mathbf{X}(t, \boldsymbol{\beta}) = \begin{bmatrix} \mathbf{x}^\top(t_1, \boldsymbol{\beta}) \\ \mathbf{x}^\top(t_2, \boldsymbol{\beta}) \\ \vdots \\ \mathbf{x}^\top(t_m, \boldsymbol{\beta}) \end{bmatrix} = \begin{matrix} \text{state} \longrightarrow \\ \begin{bmatrix} x_1(t_1, \boldsymbol{\beta}) & x_2(t_1, \boldsymbol{\beta}) & \cdots & x_h(t_1, \boldsymbol{\beta}) \\ x_1(t_2, \boldsymbol{\beta}) & x_2(t_2, \boldsymbol{\beta}) & \cdots & x_h(t_2, \boldsymbol{\beta}) \\ \vdots & \vdots & \ddots & \vdots \\ x_1(t_m, \boldsymbol{\beta}) & x_2(t_m, \boldsymbol{\beta}) & \cdots & x_h(t_m, \boldsymbol{\beta}) \end{bmatrix} \end{matrix} \begin{matrix} \\ \\ \\ \\ \text{time} \\ \downarrow \end{matrix}$$

When building the library of functions, you can consider any kind of option, from constant values, passing through polynomials, to Fourier functions and spatial operators. The specific choice may vary depending on the input data and the system under analysis. When not considering any knowledge regarding some specific parameters $\boldsymbol{\beta}$. Θ may look like:

$$\Theta(\mathbf{X}) = \begin{bmatrix} 1 & \mathbf{X} & \mathbf{X}^{P_2} & \mathbf{X}^{P_3} & \cdots & \sin(\mathbf{X}) & \cos(\mathbf{X}) & \cdots \end{bmatrix}$$

here higher polynomials are denoted as $\mathbf{X}^{P_2}$, $\mathbf{X}^{P_3}$, etc.., where $\mathbf{X}^{P_2}$ denotes the quadratic combinations of the states of $\mathbf{x}$:

$$\mathbf{X}^{P_2} = \begin{bmatrix} x_1^2(t_1) & x_1(t_1)x_2(t_1) & \cdots & x_2^2(t_1) & \cdots & x_h^2(t_1) \\ x_1^2(t_2) & x_1(t_2)x_2(t_2) & \cdots & x_2^2(t_2) & \cdots & x_h^2(t_2) \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ x_1^2(t_m) & x_1(t_m)x_2(t_m) & \cdots & x_2^2(t_m) & \cdots & x_h^2(t_m) \end{bmatrix}$$

In our cases, we are also considering a parametric dependence $\boldsymbol{\beta}$, inspired by the previous works in [4], with $\boldsymbol{\beta} \in \mathbb{R}^{m \times p}$ where p is the number of parameter we are considering. Therefore, $\Theta(\mathbf{x}, \boldsymbol{\beta})$ may look like the following:

$$\Theta(\mathbf{x}, \boldsymbol{\beta}) = \begin{bmatrix} 1 & \mathbf{x} & \mathbf{x}^{P_2} & \mathbf{x}^{P_3} & \cdots & \boldsymbol{\beta} & \boldsymbol{\beta}\mathbf{x} & \cdots \end{bmatrix}$$

Ξ is the matrix that contains the activating coefficients which multiplicate the library function. This is what has to be estimated to construct a model which fits the data.

What we aim for ultimately, is to find Ξ that approximates $\dot{X}$ the best.

$$\dot{\mathbf{X}} = \Theta(\mathbf{X}, \boldsymbol{\beta})\Xi \tag{4}$$

SINDy proved to be a great method to uncover the hidden dynamics of physical systems. Combining it with discrete temporal methods, such as explicit/implicit Euler, it also allows one to reliably compute future predictions, starting from an initial state $\mathbf{x}_0$. Applying this kind of regression is very reliable and offers a very interpretable result. Despite these advantages, data quality and volume is still very important to ensure a correct regression to the most realistic physical equation.

3.3 SINDy-SHRED

Trying to bring together the best of both world, **SINDy** and **SHRED** have first been combined by [1] with promising results. The SINDy model is placed between the GRU unit and the decoder. So SINDy is not performing a regression over the real variables, but over a latent dimension where the GRU unit sends the input sensor. Is this context SINDy is not intended to offer a direct insight into the physical dynamics, but to act a regulator stabilizing the whole architecture and making it more robust. Nonetheless, having a lens on the latent dimension dynamics can always offer extra knowledge to ponder on, especially when we take a closer look to the physics of the system. The novelty introduced by this work is the presence of a **direct parametric dependency**, making the network a Physics Informed Neural Network (PINN) over which much effort has already been dedicated in the recent literature [9], [10]. [11], [12]. The network takes as input the information from systems, governed by the same dynamic in the same spatial and temporal domains, but which differ in the values of one or more parameters. The data is selected from a fixed number of sensor, randomly placed in the spatial domain. Each sensor provides a lagged temporal sequence, $\{\mathbf{s}_{t-L:t}\}_{t=0}^{T}$, we will use the apex $\boldsymbol{\beta}$ to point the parametric dependence as $\{\mathbf{s}^{\boldsymbol{\beta}}_{t-L:t}\}_{t=0}^{T}$.

$$\boldsymbol{\beta} = \begin{bmatrix} \boldsymbol{\beta}_1 & | & \boldsymbol{\beta}_2 & | & \cdots & | & \boldsymbol{\beta}_p \end{bmatrix}^\top$$

Where $\boldsymbol{\beta}_i \in \mathbb{R}^I$, with I number of parameters for each system. For every instant t the collection of lagged sensors is $\mathbf{s}^{\boldsymbol{\beta}}_{t-L:t}$. The full input tensor $\mathbf{S}$ is:

$$\mathbf{S} = \left[\mathbf{s}^{\boldsymbol{\beta_1}}_{t_1-L:t_1} \quad | \quad \mathbf{s}^{\boldsymbol{\beta_1}}_{t_2-L:t_2} \quad |\cdots| \quad \mathbf{s}^{\boldsymbol{\beta_1}}_{T-L:T} \quad | \quad \mathbf{s}^{\boldsymbol{\beta_2}}_{t_1-L:t_1} \quad |\cdots| \quad \mathbf{s}^{\boldsymbol{\beta_2}}_{T-L:T} \quad |\cdots \mathbf{s}^{\boldsymbol{\beta_p}}_{T-L:T}\right]^\top$$

Its important to mantain the order of the matrix, during training the network will only see batches belonging to one system in order to keep consistency with the SINDy module.

Accordingly to the notation, the GRU unit will output:

$$\mathbf{Z} = \left[\mathbf{z}^{\boldsymbol{\beta_1}}_{t_1} \quad | \quad \mathbf{z}^{\boldsymbol{\beta_1}}_{t_2} \quad |\cdots| \quad \mathbf{z}^{\boldsymbol{\beta_1}}_{T} \quad | \quad \mathbf{z}^{\boldsymbol{\beta_2}}_{t_1} \quad |\cdots| \quad \mathbf{z}^{\boldsymbol{\beta_2}}_{T} \quad |\cdots \mathbf{z}^{\boldsymbol{\beta_p}}_{T}\right]^\top$$

where $\mathbf{z}^{\boldsymbol{\beta}}_t \in \mathbb{R}^h$ is the projection of the sensors to the hidden state space at instant t for the parameter $\boldsymbol{\beta}$. The same $\mathbf{z}^{\boldsymbol{\beta}}_t$ is passed to the decoder that decompresses from the hidden state to $\tilde{\mathbf{X}}_t$ to match the original $\mathbf{X}_t$. SINDy does not directly propagates forward the information, but is used to regularize the GRU output. For each instant t the network takes two inputs $\mathbf{s}^{\boldsymbol{\beta}}_{t-L:t}$ and $\mathbf{s}^{\boldsymbol{\beta}}_{t-1-L:t-1}$, the GRU unit computes $\mathbf{z}^{\boldsymbol{\beta}}_t$. We exploit SINDy to predict $\mathbf{z}^{\boldsymbol{\beta}}_t$ from $\mathbf{z}^{\boldsymbol{\beta}}_{t-1}$.

For a clearer notation, the explicit parametric dependence of $\mathbf{z}_t$ will be set aside.

The latent dynamical system is of the form:

$$\dot{\mathbf{z}} = \mathbf{r}(\mathbf{z}, \boldsymbol{\beta}), \tag{5}$$

where $r : \mathbb{R}^h \longrightarrow \mathbb{R}^h$ is the function representing the dynamic in the hidden states. Now we use a discretization method to discretize in time, any theta method will do the job, here we show the approach using an explicit Euler method. First substitute $\mathbf{r}(\mathbf{z}, \boldsymbol{\beta})$ with $\Theta(\mathbf{z}, \boldsymbol{\beta})\Xi$. $\Xi \in \mathbb{R}^{l\times h}$, where $l = \text{n}^\circ$ of functions in the library

$$\dot{\mathbf{z}} = \Theta(\mathbf{z}, \boldsymbol{\beta})\Xi \tag{6}$$

Discretize in time

$$\tilde{\mathbf{z}}_t = \mathbf{z}_{t-1} + \Delta t \Theta(\mathbf{z}_{t-1}, \boldsymbol{\beta})\Xi \tag{7}$$

where $\tilde{\mathbf{z}}_\mathbf{t} \approx \mathbf{z}_t$.

To stabilize the method we use a $k-ministeps$ approach, we compute $\Delta t_k = \frac{\Delta t}{k}$ and update:

$$\tilde{\mathbf{z}}_t = \sum_{i=0}^{k} \Theta(\mathbf{z}_{t-1+i\Delta t_k}, \boldsymbol{\beta}) \tag{8}$$

where $\mathbf{z}_{t-1+i\Delta t_k} = \mathbf{z}_{t-1} + \Theta(\mathbf{z}_{t-1+(i-1)\Delta t_k}, \boldsymbol{\beta})\Delta t_k$

Therefore, the loss function will take into account the error from the SHRED reconstruction $\tilde{\mathbf{X}} \approx \mathbf{X}$ and from the SINDy reconstruction $\tilde{\mathbf{z}} \approx \mathbf{z}$. Moreover we also set an L-0 regularization on the coefficients Ξ to promote sparsity. The final loss function is:

$$\mathcal{L}oss = \sum_{t=0}^{T} (\|\mathbf{x}_t - \tilde{\mathbf{x}}_t\|_2 + \|\mathbf{z}_t - \tilde{\mathbf{z}}_t\|_2 + \lambda\|\Xi\|_0) \tag{9}$$

Where:

- $\mathbf{x}_t$ the real data at time t.
- $\tilde{\mathbf{x}}_\mathbf{t} = \mathcal{D}(\mathcal{G}(\mathbf{s}_{t-L:t}))$ is the approximation performed by SHRED at time t.
- $\mathbf{z}_t = \mathcal{G}(\mathbf{s}_{t-L:t})$ is the projection in the latent space at time t.

- $\tilde{\mathbf{z}}_t = \sum_{i=0}^{k} \Theta(\mathbf{z}_{t-1+i\Delta t_k}, \boldsymbol{\beta})$ is the approximation performed by SINDy in the hidden space at time t, from time $t-1$.
- λ is the sparsity coefficient, to set arbitrary before training.

At Fig: 2 you can see a graphical scheme. While at algorithm 1 you can find a logical scheme for one epoch of the algorithm.

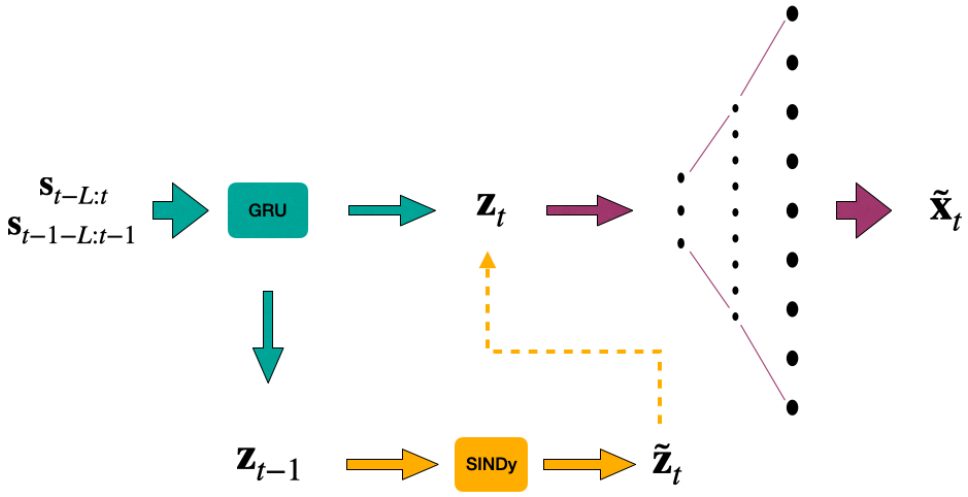


Figure 2: SINDy-SHRED Architecture, a schematic view of what the network does for each timestep t. $\mathbf{s}_{t-L:t}$ and $\mathbf{s}_{t-1-L:t-1}$ are passed to the GRU unit, which output $\mathbf{z}_t$ and $\mathbf{z}_{t-1}$, this last part is passed through SINDy, it computes $\tilde{\mathbf{z}}_t$ which is used to regulate $\mathbf{z}_t$. Lastly, the data is decompressed to get $\tilde{\mathbf{x}}_t$.

Algorithm 1 Latent state space regularization via SINDy, loop for one epoch

Input: $\mathbf{s}^{\boldsymbol{\beta}}_{t-1-L:t-1}$, $\mathbf{s}^{\boldsymbol{\beta}}_{t-L:t}$, $\mathbf{x}^{\boldsymbol{\beta}}_t$, SINDy library $\Theta(\cdot,\cdot)$, timestep Δt.

function FIT_SINDYSHRED_PARAM($\mathbf{s}^{\boldsymbol{\beta}}_{t-L:t}$, $\mathbf{x}^{\boldsymbol{\beta}}_t$, Δt)
  for $t \in \{0, \ldots, T\}$, and $\beta_i \in \{\boldsymbol{\beta_1}, \cdots, \boldsymbol{\beta_p}\}$ **do**
    $\mathbf{z}_t, \mathbf{z}_{t-1} = \mathcal{G}(\mathbf{s}_{t-L:t}), \mathcal{G}(\mathbf{s}_{t-1-L:t-1})$
    for $j \in \{0, 1, \ldots, k\}$ **do**
      $\tilde{\mathbf{z}}_{t+\frac{j}{k}\Delta t} = \tilde{\mathbf{z}}_{t+\frac{j-1}{k}\Delta t} + \Theta(\tilde{z}_{t+\frac{j-1}{k}\Delta t}, \boldsymbol{\beta_i})\Xi\Delta t$ ▷ SINDy forward simulation
    end for
    $\hat{\mathbf{x}}_t = \mathcal{D}(\mathbf{z}_t)$ ▷ SHRED reconstruction
    $\theta_{\mathcal{D}}, \Xi, \theta_{\mathcal{G}} = \arg\min_{\theta_{\mathcal{D}},\Xi,\theta_{\mathcal{G}}} \left\|\mathbf{X}_{t+1} - \hat{\mathbf{X}}_{t+1}\right\|_2 + \left\|\tilde{\mathbf{Z}} - \tilde{\mathbf{Z}}_{t+1}\right\|_2 + \lambda\|\Xi\|_0$
    if $t \mod 10 = 0$ **then**
      if $\|\Xi_i\|_0 <$ threshold **then**
        $\Xi_i = 0$ ▷ promote sparsity
      end if
    end if
  end for
end function

4 Code

4.1 Implementation Details

The following code has been implemented in Python from scratch by the author of this report. The primary references are [1], [2]. The libraries used are standard, including **NumPy**, **Torch**, and **Matplotlib**.

All runs and tests were performed on a 2017 MacBook Air with a 1.8 GHz Dual-Core Intel Core i5 processor and 8 GB 1600 MHz DDR3 RAM. As a result, the code should be accessible on most laptops, provided a reasonable data volume is used.

4.2 GitHub Repository

https://github.com/GregColetti19/SINDySHREDparametric.git

4.3 Overwiev

The code is split into 3 main scripts: **DataPreProcessing.py**, **SindyShredModel.py**, **TestSindyShred.py**. In the repository you can find a tutorial script to better understand the usage **TutorialFlow.ipynb**. An extra folder with example data is offered, the user is free to use whatever data they already have or desire.

- **DataPreProcessing.py** provides the functions to prep the data for training.
- **SindyShredModel.py** defines the actual model and the **fit** function to train it.
- **TestSindyShred.py** offers the functions to test and visualize the model results.
- **TutorialFlow.ipynb** is an example Jupyter Notebook that guides the user step-by-step to a correct use of the code.

4.4 Structure

Hereby is listed a synthesis of the code flow and how the elements should be combined.

- 1°- Load the data as a `np.ndArray`, reshape to match dimensions specified in the code.
- 2°- Preprocess the data: doable in one line or step by step for better care and customization.
  - Reshape the data.
  - Pad the data.
  - Lag the data.
  - decide if training w.r.t parameters, or w.r.t. to time
  - add noise.
  - Split the data into train, validation, test.
- 3°- Define the model. There is a default option already set, the user may choose to tweak the parameter to match the system in analysis, or due to sheer curiosity.
- 4°- Train the model using the train and validation sets declared before.
- 5°- Test the model using the test set, use the various options to visualize the results.

4.5 DataPreProcessing.py

This file provides all the functions needed to correctly prep the data for the model. The model is expecting data **x** from multiple dynamical systems generated from different parameters, where **x** is defined in both space (either 1D, 2D or 3D) and time. **x** shall be stored in a `numpy.ndArray` variable `data` in the shape `(nx, nt, nf, num_param), (nx, ny, nt, nf, num_param) or (nx, ny, nz, nt, nf, num_param)` depending on the spatial dimension. From now on we will assume we are working in $\mathbb{R}^2$ for better clarity.

- nx = size along dim-1
- ny = size along dim-2
- nt = temporal states
- nf = number of features
- num_param = number of parameter sets.

The one function `processData` accomplishes in one line the whole prep job. Hidden inside the function four main steps are happening: `dataOrdering`, `padding`, `lagging`, `dataSplitting`. Fig: 3 and Fig: 4.

`dataOrdering(data` takes in input data of shape`(nx, ny, nt, nf, num_param)`, reshapes `data` and returns a numpy array of shape `(num_param, nt, n_states)`, where $n\_states = nx * ny * nf$.

`padding(data, pad)` pads `data` with `pad` extra zeros at the beginning of the temporal series to ensure a predictions also for the first instants. It returns a numpy array of shape `(num_param, nt+pad, n_states)`.

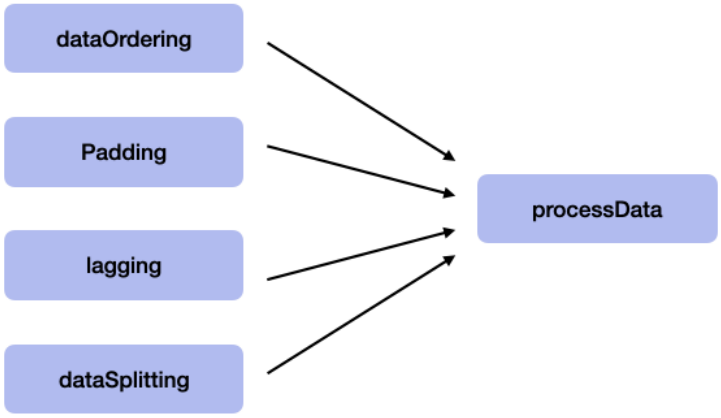


Figure 3: `dataOrdering`, `padding`, `lagging` and `dataSplitting` are called in sequence inside `processData`.

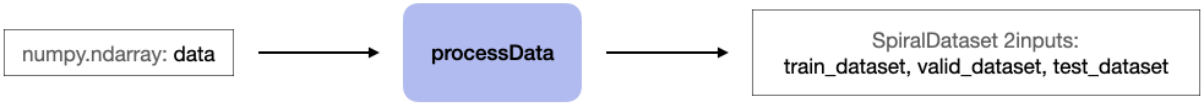


Figure 4: `processData` takes in input the orginal data and return the dataset split into train, validation and test, ready for the network.

`lagging(data, lag)` creates a tensor that include a temporal sequence of length `lag` for each instant t. It return a numpy array of shape `(num_param * nt, lag, n_states)`.

`dataSplitting(data, parameters, nt, num_param, num_sensors, train_ratio, val_ratio, param_split, constant_params, noise)` randomly selects a `num_sensors` of locations that will input their temporal measurements to the model. If `noise!=None` a Gaussian noise will be added to the sensors. Then the function will separate the data into training, validation, and test set. The splitting is performed on the parameter if `split_split = True` (which is default), meaning the training set contains a number $num\_param * train\_ratio$ of systems of time length nt. While the validation set contains a number $num\_param * val\_ratio$ of systems of time length nt. The test set contains the remaining systems. The same splitting is performed on `parameters`.

If `split_split = False`, the splitting will be performed w.r.t. time, meaning that the network will see every systems for a time interval $[0, T_{train}]$, with $T_{train} = nt * train\_ratio$. The input-output of the network are connected through the `TimeSeriesDataset` class. The `SpiralDataset_2inputs` class links the data with the corresponding parameter split. Lastly, `train_dataset`, `val_dataset`, `test_dataset` are returned.

```python
def dataSplitting(data, parameters, nt, num_param, num_sensors,
                  train_ratio=0.8, val_ratio=0.1,
                  constant_params=True, noise=None):
    '''
    This function is used to split the data into training, validation
    and test sets.
    '''
    ntxnum_param, lags, n_states = data.shape
    sensor_locations = np.random.choice(n_states, num_sensors, replace=False)
    lagged_data_sens = data[:, :, sensor_locations]
    if noise is not None:
        lagged_data_sens = addNoise(lagged_data_sens, noise).float32()

    # Generate indices for each parameter
    param_indices = np.arange(num_param)

    # Shuffle the parameter indices
    np.random.shuffle(param_indices)

    # Split the parameter indices
    train_param_indices = param_indices[:int(num_param * train_ratio)]
    valid_param_indices = param_indices[int(num_param * train_ratio):
                                       int(num_param * (train_ratio + val_ratio))]
    test_param_indices = param_indices[int(num_param *
                                           (train_ratio + val_ratio)):]

    # Generate the actual indices for training, validation, and testing
    train_indices = np.concatenate([np.arange(k * nt, (k + 1) * nt)
                                    for k in train_param_indices])
    valid_indices = np.concatenate([np.arange(k * nt, (k + 1) * nt)
                                    for k in valid_param_indices])
    test_indices = np.concatenate([np.arange(k * nt, (k + 1) * nt)
                                    for k in test_param_indices])
```

```
    # Organize data for training
    device = 'cuda' if torch.cuda.is_available() else 'cpu'

    if constant_params:
        params = torch.tensor(parameters, dtype=torch.float32).
                repeat_interleave(nt).unsqueeze(1).to(device)
    else:
        params = torch.tensor(parameters, dtype=torch.float32).to(device)

    # Parameters
    param_basic = params[train_indices]
    param_valid = params[valid_indices]
    param_test = params[valid_indices]
    # Dataset
    train_basic_in = torch.tensor(lagged_data_sens[train_indices,:,:],
                                  dtype=torch.float32).to(device)
    valid_basic_in = torch.tensor(lagged_data_sens[valid_indices,:,:],
                                  dtype=torch.float32).to(device)
    test_basic_in = torch.tensor(lagged_data_sens[test_indices,:,:],
                                  dtype=torch.float32).to(device)

    train_basic_out = torch.tensor(data[train_indices,:,:],
                                  dtype=torch.float32).to(device)
    valid_basic_out = torch.tensor(data[valid_indices,:,:],
                                  dtype=torch.float32).to(device)
    test_basic_out = torch.tensor(data[test_indices,:,:],
                                  dtype=torch.float32).to(device)

    # CONNECT in-and-out
    # BASIC
    train_basic_set = TimeSeriesDataset(train_basic_in, train_basic_out)
    valid_basic_set = TimeSeriesDataset(valid_basic_in, valid_basic_out)
    test_basic_set = TimeSeriesDataset(test_basic_in, test_basic_out)

    train_dataset = SpiralDataset_2inputs(train_basic_set, param_basic)
    valid_dataset = SpiralDataset_2inputs(valid_basic_set, param_valid)
    test_dataset = SpiralDataset_2inputs(test_basic_set, param_test)

    return train_dataset, valid_dataset, test_dataset
```

4.6 SindyShredModel.py

This file contains the main structure of the model. A `SINDySHRED` class is composed from three main modules: `GRUmodule`, `SindyModule`, `DecoderModule`. To help construction and debugging each module is a different class.

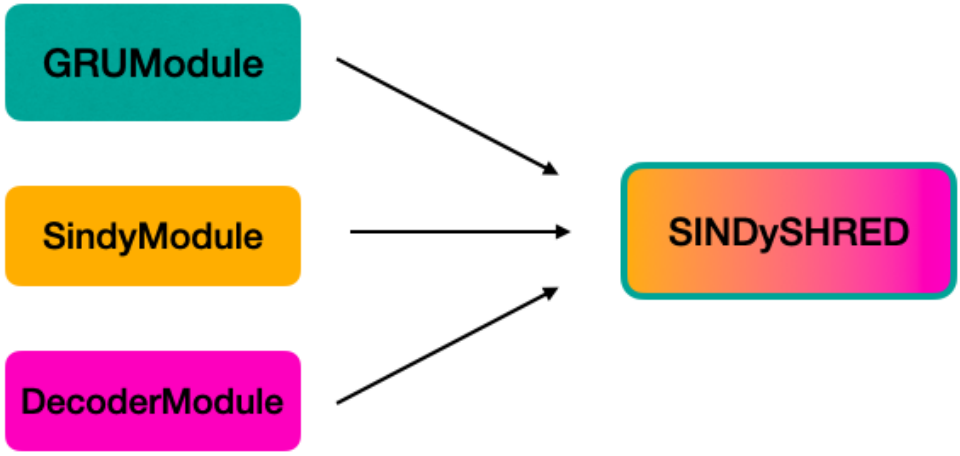


Figure 5: the `SINDySHRED` class combines together its three blocks `GRUmodule`, `SindyModule`, `DecoderModule`

The model class is defined below. The `GRU` module takes 3 inputs variable, the `SINDy` module takes 6 inputs, and the `Decoder` module takes 4. The `forward` function returns $\tilde{\mathbf{x}}_t$ given $\mathbf{s}_{t-L:t}$.

$$\tilde{\mathbf{x}}_t = \mathcal{D}(\mathcal{G}(\mathbf{s}_{t-L:t}))$$

```python
# SINDySHRED model definition
class SINDySHRED(torch.nn.Module):
    def __init__(self, max_degree: int, features_name: list, dt: float,
                 param_names: list, threshold: float,
                 num_sensors: int, num_param: int, n_states: int,
                 hidden_size: int, hidden_layers: int,
                 decoder_sizes=[50, 100], dropout=0.1):
        super(SINDySHRED, self).__init__()

        self.gru = GRUmodule(num_sensors=num_sensors, hidden_size=hidden_size,
                             hidden_layers=hidden_layers)
        self.sindy = SINDyModule(output_features=hidden_size,
                                 max_degree=max_degree,
                                 dt=dt, feature_names=features_name,
                                 param_names=param_names,
                                 threshold=threshold)
        self.decoder = DecoderModule(input_size=hidden_size,
                                     output_size=n_states,
                                     num_param=num_param,
                                     decoder_sizes=decoder_sizes,
                                     dropout=dropout)

    def forward(self, x):
        z = self.gru(x)
        x_approx = self.decoder(z)

        return x_approx
```

SINDy's contribution appears in the fit function.

```python
def fit_SindyShred_param(model, train_dataset, valid_dataset, batch_size,
                    num_epochs, lr, lambda_sindy, verbose=True, patience=5):
train_loader = DataLoader(train_dataset, shuffle=False, batch_size=batch_size)
optimizer = torch.optim.Adam(model.parameters(), lr=lr)
criterion = torch.nn.MSELoss()
train_error_list = []
valid_error_list = []
patience_counter = 0
best_params = model.state_dict()

for epoch in range(1, num_epochs + 1):
    for data, params in train_loader:
        model.train()
        optimizer.zero_grad()
        shred_output = model(data[0][:,1:,:])
        loss_shred = criterion(shred_output, data[1][:,-1,:])
        sindy_input = model.gru(data[0][:,:-1,:])
        sindy_target = model.gru(data[0][:,1:,:])
        sindy_output = model.sindy.simulate_next_explicit(sindy_input,
                                                           params)
        loss_sindy = criterion(sindy_output, sindy_target)
        loss_sparsity = lambda_sindy*torch.norm
                            (model.sindy.coefficients.weight, 0)
        loss = loss_shred + loss_sindy + loss_sparsity
        loss.backward()
        optimizer.step()

    model.eval()
    with torch.no_grad():
        # Train loss
        loss_shred = criterion(model(train_dataset.data.X[:,:-1,:]),
                               train_dataset.data.Y[:,-1,:])
        sindy_input = model.gru(train_dataset.data.X[:,:-1,:])
        sindy_target = model.gru(train_dataset.data.X[:,1:,:])
        sindy_output = model.sindy.simulate_next_explicit(sindy_input,
                                                  train_dataset.params)
        loss_sindy = criterion(sindy_output, sindy_target)
        train_error = loss_shred + loss_sindy
        train_error_list.append(train_error)
        # Validation loss
        loss_shred = criterion(model(valid_dataset.data.X[:,:-1,:]),
                               valid_dataset.data.Y[:,-1,:])
        sindy_input = model.gru(valid_dataset.data.X[:,:-1,:])
        sindy_output = model.sindy.simulate_next_explicit(sindy_input,
                                                  valid_dataset.params)
        sindy_target = model.gru(valid_dataset.data.X[:,1:,:])
        loss_sindy = criterion(sindy_input, sindy_target)
        valid_error = loss_shred + loss_sindy
        valid_error_list.append(valid_error)

```

```
        if verbose:
            print(f"Training epoch {epoch}")
            print(f"Training error: {train_error}")
            print(f"Validation error: {valid_error}")

        if epoch % 10 == 0:
            model.sindy.apply_threshold()

        if verbose and (epoch == 1 or epoch % 10 == 0):
            model.sindy.print_equations()

        # Save parameters
        if valid_error == torch.min(torch.tensor(valid_error_list)):
            patience_counter = 0
            best_params = deepcopy(model.state_dict())
        else:
            patience_counter += 1

        if patience_counter == patience:
            model.load_state_dict(best_params)
            print("Patience reached")
            return torch.tensor(valid_error_list).cpu(),
                   torch.tensor(train_error_list).cpu()

    model.load_state_dict(best_params)
    return torch.tensor(valid_error_list).cpu(),
           torch.tensor(train_error_list).cpu()
```

Remembering how the **Loss function** is computed in eq. 9, in the fit function we have to compute with the `GRU` module both $\mathbf{z}_t^{\boldsymbol{\beta}} = \mathcal{G}(\mathbf{s}_{t-L:t}^{\boldsymbol{\beta}})$ and $\mathbf{z}_{\mathbf{t-1}}^{\boldsymbol{\beta}} = \mathcal{G}(\mathbf{s}_{t-1-L:t-1}^{\boldsymbol{\beta}})$. The function replicates Algorithm 1 to the last epoch or until the `patience` is reached. The function returns a list of training and validation error for every epoch.

4.6.1 GRUmodule class

The `GRUModule` class is fully customizable in its proportion.

Inputs:

`num_sensors` : Represents the number of active sensors; it is important to define the input layer of the GRU unit.

`hidden_size` : The size of the latent dimension. This value should be small (i.e., < 5) to speed up training.

`num_layers` : The number of hidden layers inside the GRU unit. This value should logically be smaller than `lags`. Ideally, having more layers helps the unit learn correlations among lagged data, but too many layers will significantly slow down training.

Only a `forward` function is defines inside the class. The data enter shaped as `(batch_size, lags, num_sensors)` and leave as `(batch_size, hidden_size)`. Look Fig 6.

Figure 6: Effect of the `GRUModule`

4.6.2 SindyModule.py

In order to fit in whole scheme, the SINDy algorithm is formulated as a neural network. Since the approximation of $\mathbf{f}(\mathbf{z})$ is matrix/vector multiplication $\Theta(\mathbf{z}, \boldsymbol{\beta})\Xi$, it is very straight forward to convert the operation to a single dense fully connected layer, with linear activation functions, where the input is the matrix $\Theta(\mathbf{z}_t, \boldsymbol{\beta})$ for each time step, the output is $\tilde{\mathbf{z}}_t$. The coefficients Ξ represent the trainable weights of the network.

Inputs:

`output_features` : The hidden dimension of the projection space.

`dt` : The temporal step, used for time integration of the model.

`max_degree` : The maximum degree for the Θ function library, which uses **polynomial functions**.

`features_names` : The names of hidden variables, needed for visualization.

`param_names` : The names of known parameters, also needed for visualization. (default: None)

`threshold` : The minimum value to consider a coefficient of Ξ, used to promote sparsity.

`include_bias` : Boolean flag to include or exclude the constant bias in the library. (default: False)

`SindyModule` class has 6 functions: `forward`, `apply_threshold`, `print_equations`, `simulate_next_explicit`, `simulate_next_implicit` and `simulate`.

The `forward(self, x, params` function imply performs the matrix/vector multiplication cited above, given the input `x` of size `(batch_size, hidden_size)` and `params` of size `(batch_size, param_size)`. Look Fig 7.

`apply_threshold(self)` sets to zero any coefficients below the class' threshold, used to promote sparsity.

`print_equations(self)` prints the latent dynamical system found by the model.

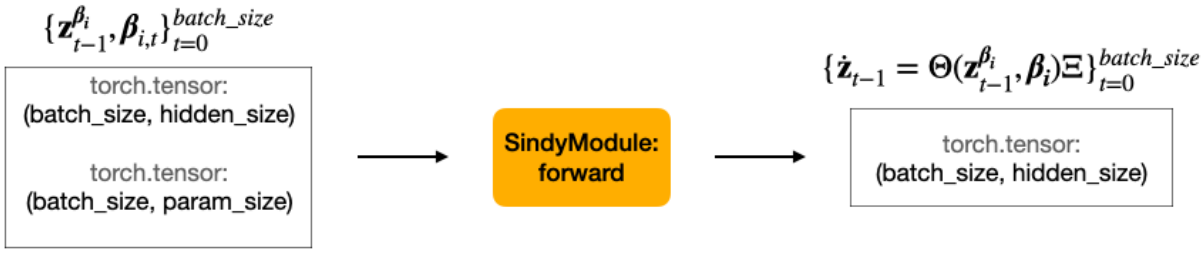


Figure 7: SINDyModule forward function

`simulate_next_explicit(self, x_prev, params, ministeps=10)` computes the approximation through the explict Euler method in eq. 8. It outputs $\tilde{\mathbf{z}}_\mathbf{t}$ given $\mathbf{z}_{t-1}$, in this case `x_prev` is of shape `(batch_size, hidden_size)`, and `params` of size `(batch_size, num_param)` as before. `ministeps` represents the number of intermediate steps, the default is 10. Fig: 8.

`simulate_next_implicit(self, x_prev, params, tol=1e-6, max_iter=100)` outputs $\tilde{\mathbf{z}}_t$ as well, but uses an implicit Newton method.

`simulate(self, initial_state, params, t_span, method='explicit')` compute a simulation of the hidden state given an `initial_state` and temporal interval `t_span`. The user can specify which method of integration wants to use, picking from the explicit and implicit functions depicted above.

A special mention has to go the `PolynomialLibrary` class that builds the library of polynomial functions for the SINDy module.

Inputs:

`features_names` : The list of names for the hidden state.

`param_names` : The list of names for the parameters.

`max_degree` : The highest polynomial degree.

`include_bias` : Boolean flag to set the constant term in the library. (default: False)

The class has two functions: `get_features_names` and `forward`, in Fig: 9.

The first one computes the list of names belonging to the library. The second one actually computes $\Theta(\mathbf{x}, \boldsymbol{\beta})$ given `x` and `params`.

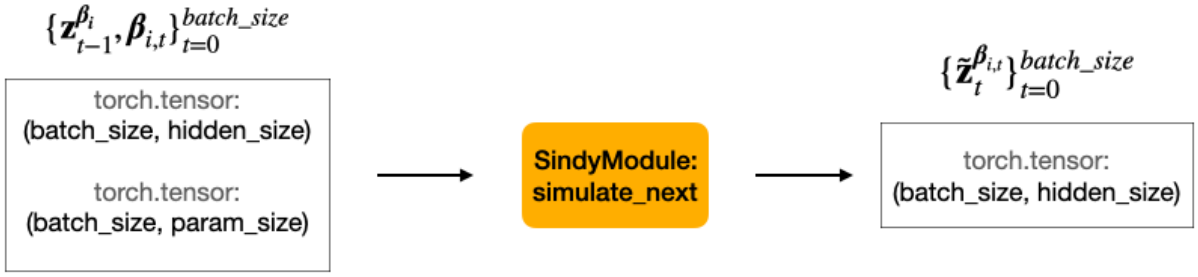


Figure 8: SINDyModule simulate_next function. Scheme valid for both the explicit and implicit case

Figure 9: SINDyModule simulate_next function. Scheme valid for both the explicit and implicit case

4.6.3 DecoderModule.py

This class builds the final decoder of the model. It is designed to accept an arbitrary number of layer of different sizes. Its versatility potentially allows the class to behave has any shaped fully connected network, for example an **auto-encoder**. These options have not been explored to maintain a narrower focus, but the possibility are there.

Inputs:

`input_size` : Dimension of the latent input representation.

`output_size` : Dimension of the reconstructed output.

`num_param` : Number of parameter sets.

`decoder_sizes` : List defining the sizes of hidden layers (default: [40, 100]).

`dropout` : Dropout rate for regularization (default: 0.1).

The `DecoderModule` consists of:

1. Fully connected layers with dimensions specified by `decoder_sizes`.
2. ReLU activation functions applied after each layer.
3. Dropout regularization to prevent overfitting.
4. A final output layer that maps to the target `output_size`.

So far no main advantage occurred when using a decoder deeper than two neural layers, generally speaking a change of size of $\approx 10\times$ is recommended. For example, let the input_size = 4 and output_size = 4000, the two hidden layers shall be sized as [40, 400].

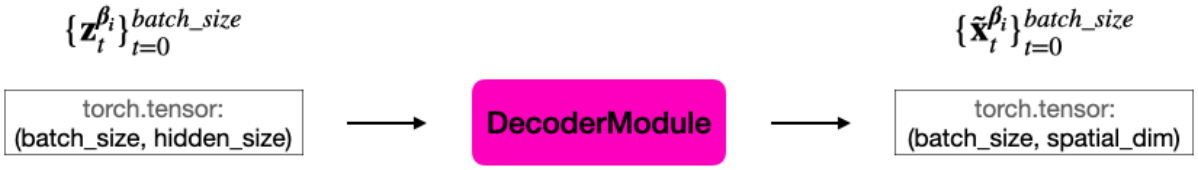


Figure 10: Decoder `forward` function.

4.7 TestSindyShred.py

This file provides all the functions required to evaluate and visualize the performance of the trained model. It includes methods for computing the test error, comparing model predictions with ground truth values, and analyzing results via plots.

Main functionalities:

- Compute model and SINDy errors.
- Compare real data and predictions at selected sensor locations for every instant.
- Compare real data and prediction for every degree of freedom at given instant
- Plot hidden state progession over time and its SINDy approximation.

The functions in `TestSindyShred.py` are necessary for evaluating and interpreting the model's performance. They allow both quantitative error analysis and qualitative visualization of sensor predictions.

4.7.1 Error Computation

The function `test_error` computes the test error for both the SHRED model and the SINDy model.

- **Inputs:**
  - `model`: The trained SHRED-SINDy model.
  - `test_set`: The dataset used for evaluation.
- **Outputs:**
  - `error_shred`: Mean squared error (MSE) between SHRED predictions and ground truth.
  - `error_sindy`: MSE between SINDy-predicted values and sensors values from the GRU module.

The function basically computes the first two terms from the loss function in eq. 9, without considering the sparsity term.

4.7.2 Sensor Data Comparison

The function `plot_sensors_comparison_shred` visualizes the performance of the SHRED model on random sensors.

Inputs

`model` : The trained model.

`test_set` : The test dataset.

`nt` : Number of time steps for evaluation.

`which_param` : The specific parameter index to visualize.

The function selects three random sensor locations and plots: the real sensor measurements and the corresponding SHRED model predictions. For a visual example look Fig: 12.

4.7.3 Full Domain Comparison

The function `plot_shred_comparison_2D` juxtaposes the predicted results of the model and the real data. Example at Fig: 14.

Inputs

`model` : The trained model.

`test_set` : The test dataset.

`instant` : The temporal instant to visualize.

`nx` : Size of the first spatial dimension.

`ny` : Size of the second spatial dimension.

`nf` : Number of features (number of equations in the original system).

`nt` : Number of time steps for evaluation.

`which_param` : The specific parameter index to visualize.

There also is a 1D version `plot_shred_comparison_1D`. It has the same inputs minus `ny`.

4.7.4 SINDy Latent Prediction

The function `plot_sensors_comparison_sindy` plots the true hidden variables behavior in time alongside the prediction made from the SINDy module for every time steps, for a specific paramater. Moreover it prints the latent state system of equations. Example at Fig: 13.

Inputs

`model` : The trained model.

`test_set` : The test dataset.

`instant` : The temporal instant to visualize.

`nt` : Number of time steps for evaluation.

`which_param` : The specific parameter index to visualize.

5 Case Test: spiriling waves

This reaction-diffusion system exhibits spiral waves in a periodic domain, and the PDEs are:

$$\begin{cases} u_t(\mathbf{y},t) = 0.1\nabla^2 u(\mathbf{y},t) + \big(1 - A^2(\mathbf{y},t)\big)u(\mathbf{y},t) + \beta A^2(\mathbf{y},t)v(\mathbf{y},t) & \text{for } \mathbf{y} \in \Omega,\ t \in [0,T], \\ v_t(\mathbf{y},t) = 0.1\nabla^2 v(\mathbf{y},t) - \beta A^2(\mathbf{y},t)u(\mathbf{y},t) + \big(1 - A^2(\mathbf{y},t)\big)v(\mathbf{y},t) & \text{for } \mathbf{y} \in \Omega,\ t \in [0,T] \end{cases} \tag{10}$$

where

$$A^2(\mathbf{y},t) = u^2(\mathbf{y},t) + v^2(\mathbf{y},t).$$

Here, $u(\mathbf{y},t)$ and $v(\mathbf{y},t)$ represent the state variables that depend on space $\mathbf{y}$ and time t.

The spatial domain is $\Omega = [-10,10] \times [-10,10]$, time domain is $[0,10]$. Where β varies from 0.1 to 10.

Numerically, Ω is divided in a 50×50 grid. The temporal interval is split in 100 instants with $dt = 0.05$. $\boldsymbol{\beta}$ is a collection of 100 parameters, where $\beta_1 = 0.1, \beta_2 = 0.2, \cdots, \beta_{100} = 10$.

In Fig: 11 you can get an idea on how much the system differs between the two extreme values of $\boldsymbol{\beta}$.

.

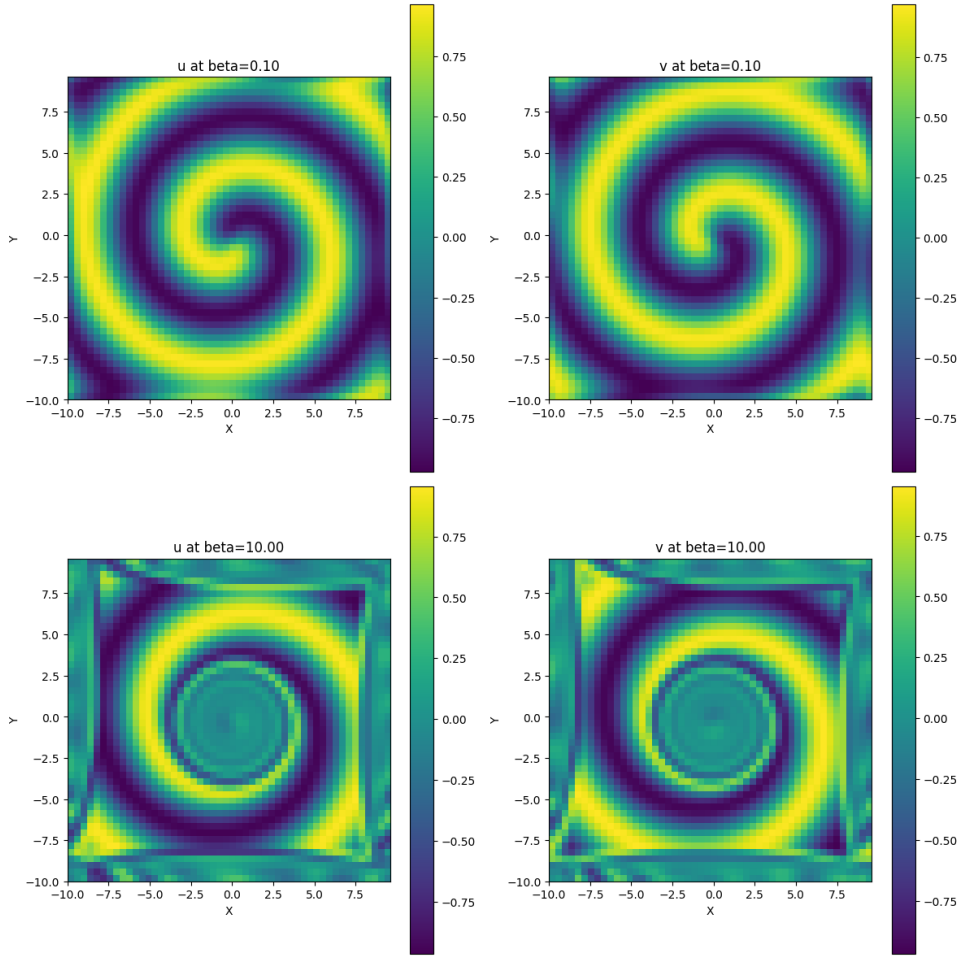


Figure 11: u and v at time $t = 50$, for $\beta = 0.1$ and $\beta = 10$

6 Numerical Results

To keep the results more reliable, from now on will specify the model parameters used.

Regarding the data **preprocessing**:

i) `lags` = 11

ii) `num_sensors` = 40

iii) `train_ratio` = 0.8

iv) `valid_ratio` = 0.1

v) `param_split` = False (default)

vi) `noise` = None (default)

The **model** is defined with:

i) `max_degree` = 1

ii) `hidden_size` = 3

iii) `hidden_layer` = 2

iv) `threshold` = 0.3

v) `num_sensors` $= 40$

vi) `decoder_sizes` $= [40, 400]$

vii) `n_states` $= 5000$

viii) `dropout` $= 0.1$

And finally, the parameters for **training**:

i) `batch_size` $= 20$ for section 6.1, and `batch_size`$= 25$ for section 6.2.

ii) `lr` $= 0.01$, this is the learing rate for the optimizer (ADAM)

iii) `lambda_sindy` $= 0.3$, sparsity term.

iv) `num_epochs` $= 1000$, max number of loops through the dataset.

v) `patience` $= 8$

The model is designed to be trained in two different ways:

1. Training with respect to **time**
2. Training with respect to **parameters**

6.1 Time Training

When training w.r.t. time, we train the model passing every system (in our case 100 systems), but only showing it the network `train_ratio*nt` instants. So in our case **train_set** $= num\_param \times [0, 80)$ instants, **valid_set** $= num\_param \times [80, 90)$ instants and **test_set** $= num\_param \times [90, 100]$ instants. We will show the results relative to the 71-th parameter, $\beta = 7.1$, and consider **test interval** $= [90, 100]$.

6.1.1 Error

The approximation error for SHRED is:

$$\|\{\mathbf{x}_t^{\boldsymbol{\beta}} - \tilde{\mathbf{x}}_t^{\boldsymbol{\beta}}\}_{t=90}^{100}\|_2 = 0.0545 \tag{11}$$

The approximation error for SINDy is:

$$\|\{\mathbf{z}_t - \tilde{\mathbf{z}}_t\}_{t=90}^{100}\|_2 = 0.0010 \tag{12}$$

6.1.2 Sensors Comparison

In Fig: 12 you can see the prediction for three random sensors during the full **test interval** = [90,100]. We can appreciate how the model is able to capture the correct behaviour.

6.1.3 Sindy Prediction

After training, the final system obtained from SINDy is a fairly simple system. In eq. 13 we can clearly see β, meaning that our algorithm is capable of capturing the extra parametric information. Moreover, we can appreciate in Fig: 13 how the system is able to predict reliably the underlying latent dynamic for a time sequence never seen before during training.

$$\begin{cases} z_1' = -1.08\beta z_2, \\ z_2' = -0.33 z_2 + 0.75\beta z_1, \\ z_3' = -0.85 z_3. \end{cases} \tag{13}$$

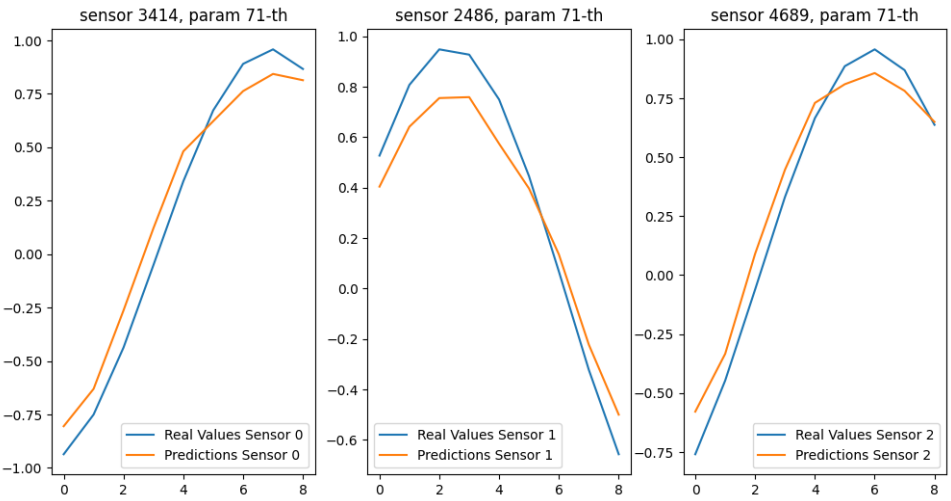
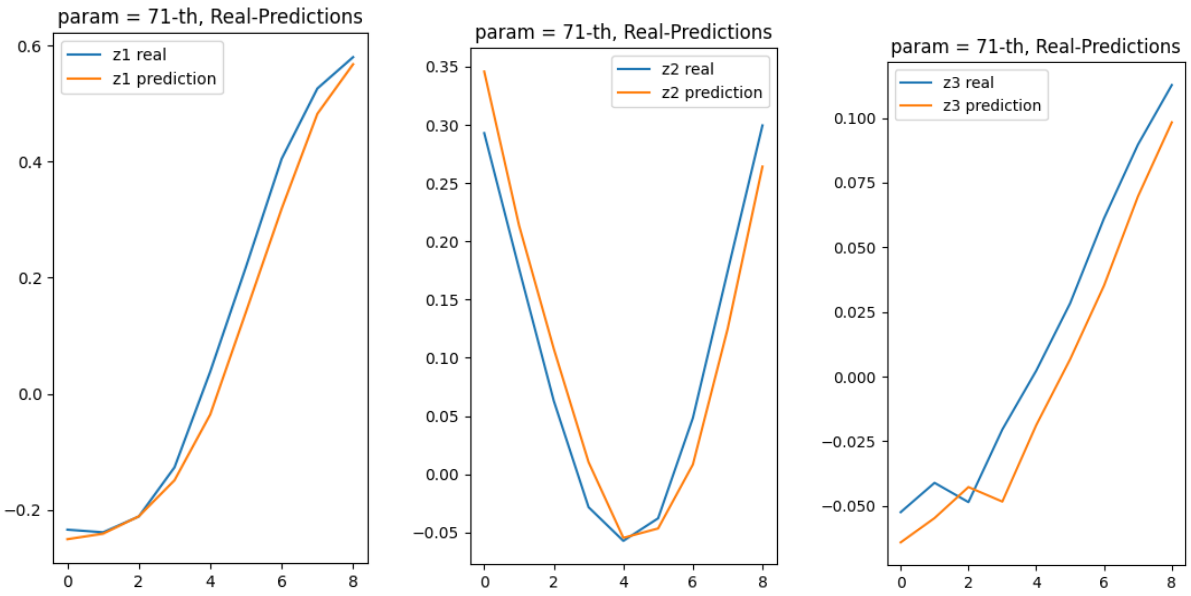
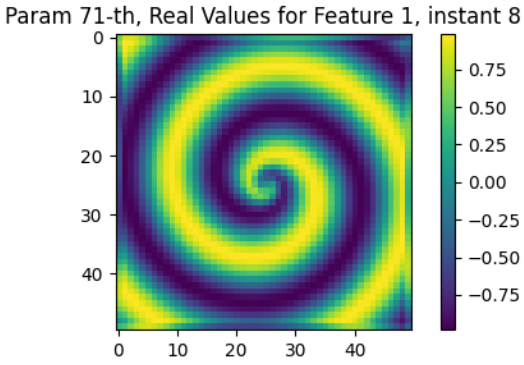


Figure 12: Comparison of the model prediction in three random space location in the **test interval**, for $\beta = 7.1$

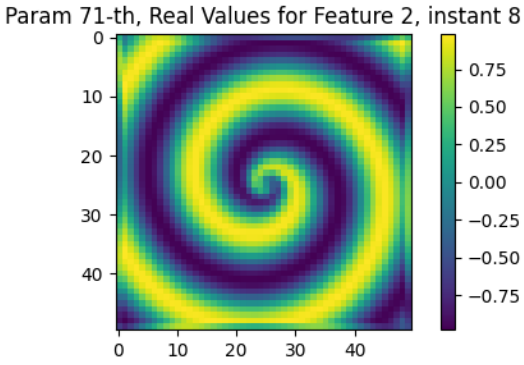


(a) hidden variable z_1 (b) hidden variable z_2 (c) hidden variable z_3

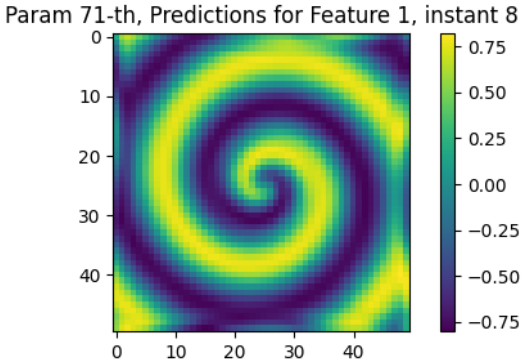
Figure 13: SINDy's prediction for the hidden state in the **test interval**. The blue function represent $\mathbf{z}_t$ while the orange function is $\tilde{\mathbf{z}}_t$.



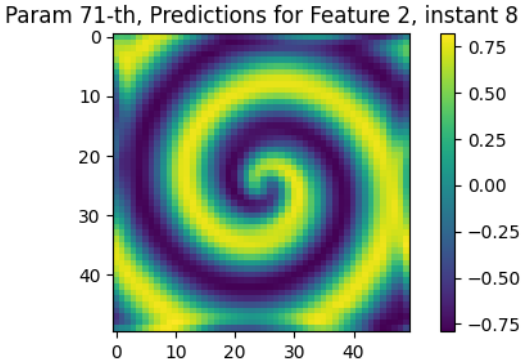
(a) Real u at time 8



(b) Real v at time 8



(c) Prediction u



(d) Prediction v

Figure 14: Comparison of real and predicted images for u and v in the $t = 98$ for $\beta = 7.1$. The first row shows the real values for both u and v. The second row shows the model predictions.

6.1.4 Final approximation

Lastly, let's observe the final reconstruction for the full domain. We can notice in Fig: 14 that the finale prediction is very close to the real system. The main differences are on the edges of the domain, but nothing drastic, how we could already suspect from the low error in eq.11.

6.2 Parameters Training

In this section will show the results when training the model to learn the impact of the parameters. The training will be performed over the whole time interval $[0, 100]$, considering only a `train_ratio * num_param` number of random parameters, in our case 80. So the **train_set** $= 80 \times [0, 100]$, the **valid_set** $= 10 \times [0, 100]$ and the **test_set** $= 10 \times [0, 100]$. Moreover added a Gaussian noise of 10% intensity with respect to the max value of the concentrations (u, v), to show the smoothing effect of the GRU unit. $\boldsymbol{\beta}_{test}$ is the set containing the "new" parameters for the network. The results shown below are relative to the second parameter $\beta_2 \in \boldsymbol{\beta}_{test}$

6.2.1 GRU smoothing

Since we applied a normal noise to resemble real life sensor, let's see the smoothening effect of the **GRU** module. Fig: 15.

We can comfortably say that the **GRU** module does a great job a smoothening the noisy sensors.

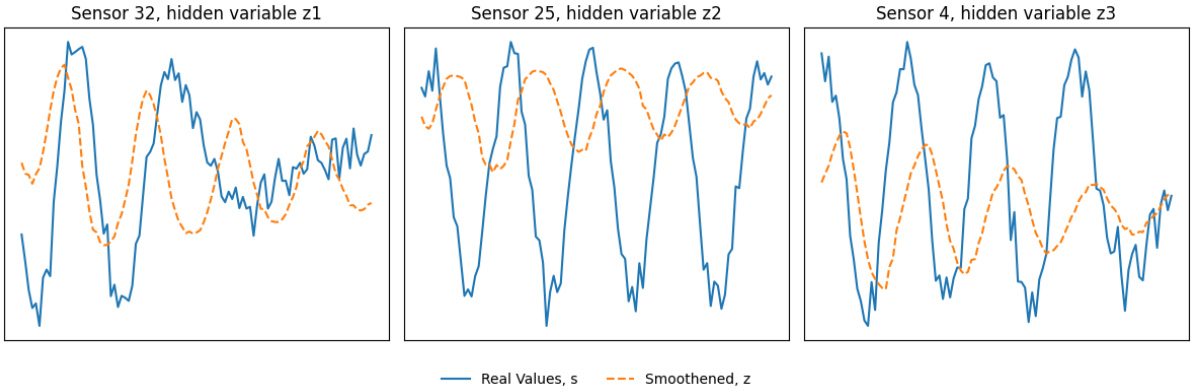


Figure 15: Smoothening effect of the **GRU** module. We show three random sensors, $\mathbf{s}_t^{\beta_2}$, and compare them with the three hidden variable, $\mathbf{z}_t^{\beta_2}$, in the smoothened latent space.

6.2.2 Error

The approximation error for SHRED is:

$$\|\{\mathbf{x}_t^{\beta_i} - \tilde{\mathbf{x}}_t^{\beta_i}\}_{t=0}^{100}\|_2 = 0.0547, \text{ with } \beta_i \in \boldsymbol{\beta}_{test} \tag{14}$$

The approximation error for SINDy is:

$$\|\{\mathbf{z}_t^{\beta_i} - \tilde{\mathbf{z}}_t^{\beta_i}\}_{t=0}^{100}\|_2 = 0.0031, \text{ with } \beta_i \in \boldsymbol{\beta}_{test} \tag{15}$$

6.2.3 Sensor Comparison

In Fig: 16 you can see the prediction for three random sensors during the full **test interval** = [0,100], for β_2. We can appreciate how the model is able to capture the correct behaviour. For the third sensor is notable a loss in accuracy for last time steps.

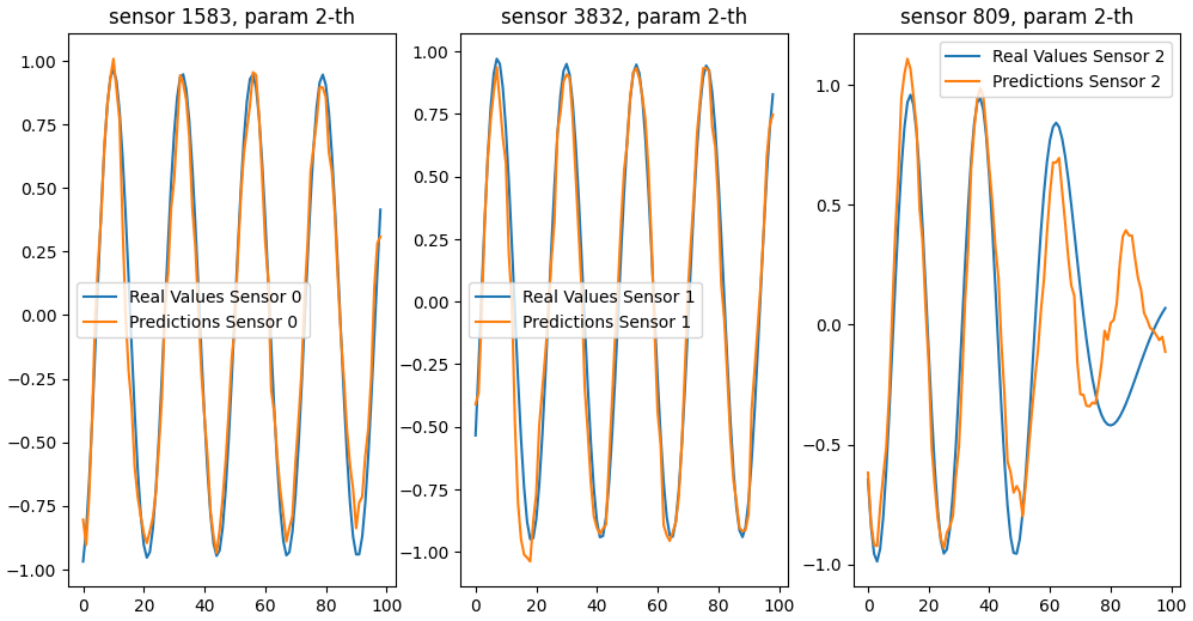
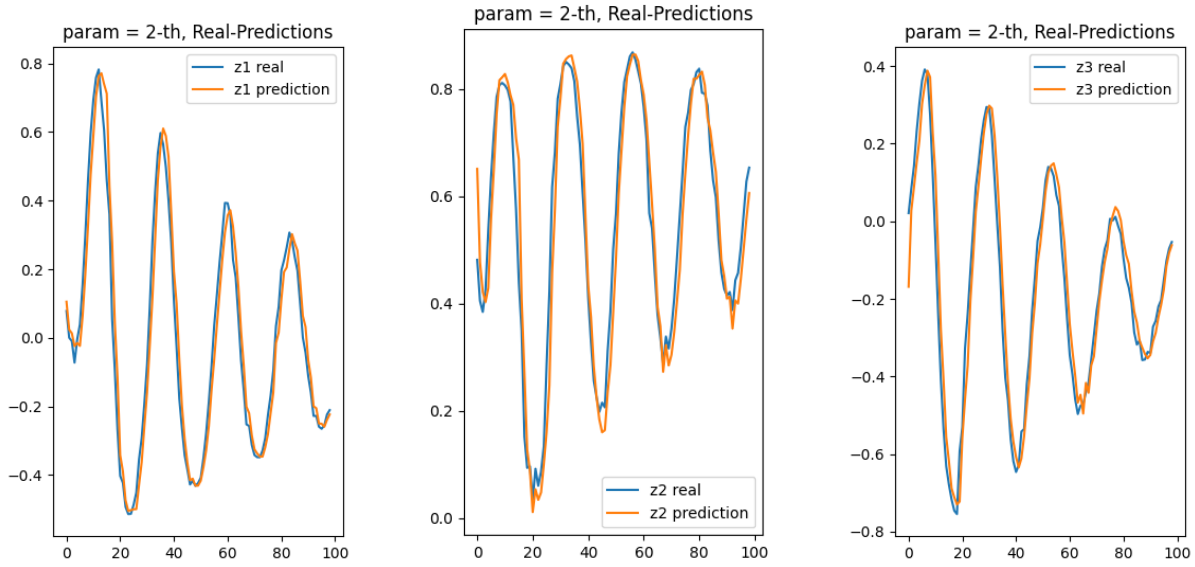


Figure 16: Comparison of the model prediction in three random space location in the Test interval, for $\beta_2 \in \boldsymbol{\beta}_{test}$.

6.2.4 SINDy Prediction

After training, the final system obtain from SINDy similar to the one found before in eq. 13, with slightly more terms. Once again, in eq: 16 we can clearly see β, meaning that our algorithm is capable of capturing the extra parametric information. Moreover, we can appreciate in Fig: 17 how the system is able to predict reliably the underlying latent dynamic for the whole time sequence for a parameter never seen by the network.

$$\begin{cases} z_1' = 0.41z_3 + 0.55\beta z_2 + 0.66\beta z_3, \\ z_2' = 0.62z_1 + 1.05z_3 + -0.47\beta z_1 + 0.35\beta z_3, \\ z_3' = -0.85z_3. \end{cases} \tag{16}$$

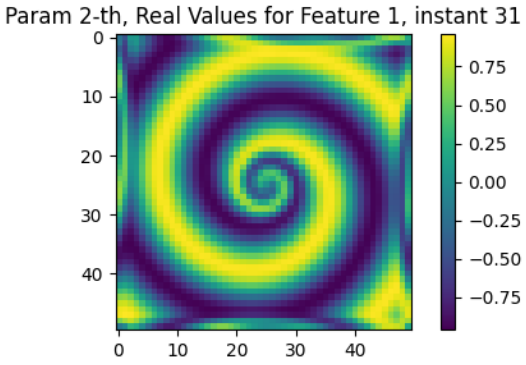


(a) hidden variable z_1 (b) hidden variable z_2 (c) hidden variable z_3

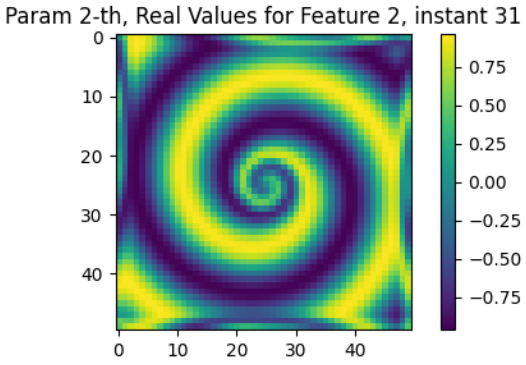
Figure 17: SINDy's prediction for the hidden state, the blue function represent $\mathbf{z}_t$ while the orange function is $\tilde{\mathbf{z}}_t$.

6.2.5 Final Reconstruction

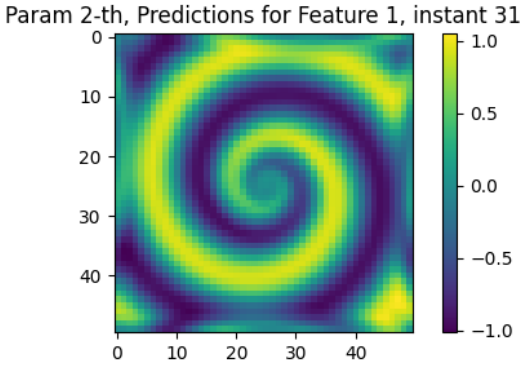
Let's observe the final SHRED reconstruction for the full domain. We can notice in Fig: 18 that the finale prediction is very close to the real system. In this case the prediction loses definition, but it is still very close to the real concentrations.



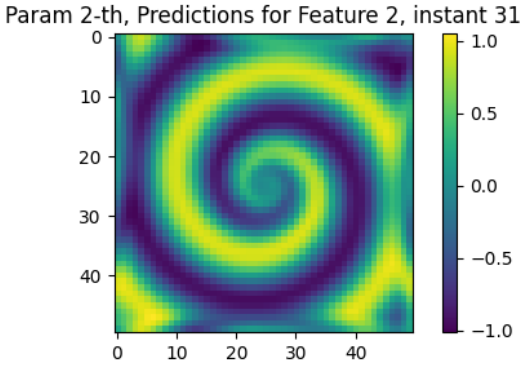
(a) Real u at time 31, param β_2



(b) Real v at time 31, param β_2



(c) Prediction u at time 31, param β_2



(d) Prediction v at time 31, param β_2

Figure 18: Comparison of real and predicted images for u and v in the **test interval** for β_2. The first row shows the real values for both u and v, $\mathbf{x}_{31}^{\beta_2}$. The second row show the model predictions, $\tilde{\mathbf{x}}_{31}^{\beta_2}$.

7 Conclusions

This work has successfully developed an improved version of the SINDy-SHRED framework integrating a direct **parametric** dependence. The proposed hybrid model leverages the strengths of both machine learning and data-driven modelling to tackle the challenges posed by high-dimensional dynamical systems with limited sensor measurements.

7.1 Key Achievements

- **Reconstruction of the Full Domain from Sparse Sensors:** The model demonstrated its ability to reconstruct the full spatial domain of a dynamical system using only a limited number of sensor inputs. By leveraging a recurrent unit and a shallow decoder, the framework effectively interpolates missing spatial information over time.
- **Symbolic Representation of the Latent Space:** The integration of SINDy within the latent space provided a symbolic and interpretable representation of the system's dynamics, allowing for a physics-informed approach that aligns well with the underlying governing equations.
- **Parametric Dependence Integration:** A key novelty of this work is the incorporation of parametric dependencies directly into the SINDy framework, making the model adaptable across a family of systems that share the same governing dynamics but differ

in specific parameter values. This enhancement makes the model capable of generalizing across different parameter settings.

- **Robust Training and Validation:** Through rigorous numerical experimentation, the model was trained and validated using both time-based and parameter-based training approaches. The results indicate that the model generalizes well, even to unseen parameter configurations, with low approximation errors for both the SHRED reconstruction and the SINDy latent state predictions.
- **SINDy-Enhanced Stability and Interpretability:** The inclusion of SINDy not only stabilizes the training process but also provides insight into the learned latent space dynamics. The identified governing equations in the hidden state offer an additional layer of interpretability, making the model more reliable for scientific applications.

7.2 Limitations and Future Work

Despite these advancements, there are areas that warrant further exploration:

- **Scaling to Higher-Dimensional Systems:** The current implementation was tested on a 2D reaction-diffusion system; future work should explore its applicability to more complex, higher-dimensional PDEs. However this should not be a big challenge, since the input `data` is reshaped to deal with the total degree of freedom, not caring about spatial or features dimensions.
- **A more Integrated Library for SINDy:** Integrating a more flexible `Library` class would boost the versatility of the algorithm. In particular building a more customizable class would extend the potentiality of the algorithm to a wide range of dynamics.
- **Adaptive Sensor Placement:** Exploring strategies for optimal sensor placement could further improve reconstruction quality and reduce the number of required sensors.
- **Comparison with Other Data-Driven Methods:** Benchmarking the performance of SINDy-SHRED against other state-of-the-art machine learning and physics-informed methods would provide a clearer understanding of its advantages and trade-offs.
- **Fine Tuning:** A better parameter combination is likely available, so far the project only focused on using a set of favourable parameters, many other options may boost the method.

7.3 Final Remarks

The results of this study highlight the potential of combining machine learning with data-driven modeling for complex dynamical systems. The **parametric SINDy-SHRED** framework offers a promising pathway for discovering interpretable representations of system dynamics while ensuring accurate full-domain reconstructions. Its ability to generalize across parameter variations makes it a powerful tool for scientific modeling, with potential applications in fluid dynamics, climate modeling, and beyond.

8 Tutorial

This section provides a step-by-step guide to using the SINDy-SHRED code. Follow these instructions to set up the environment, preprocess data, train the model, and visualize results.

8.1 Download the Code and Install Dependencies

Clone the repository:

```
git clone https://github.com/GregColetti19/SINDySHREDparametric.git
cd SINDySHREDparametric
```

8.2 Open the Jupyter Notebook

Launch the tutorial notebook:

```
jupyter notebook TutorialFlow.ipynb
```

Follow the notebook instructions step by step.

8.3 Load and Reshape the Data

The dataset should be loaded as a NumPy array and reshaped to match the expected format:

```
import numpy as np
from DataPreProcessing import processData

data, dt = load_your_data() #load the data and the temporal step
params = load_your_params() #load the parameters
data = reshape_to_correct_dimensions(data) #(nx, ny, nt, nf, num_param)
nx, ny, nt, nf, num_param = data.shape
params = reshape_to_correct_dimensions(params) #(num_param, num_paramsForSystem)
```

8.4 Preprocess the Data

Decide if you want to add noise and what kind of training you want to do (w.r.t parameters or w.r.t. time).

```
    noise = None
```

or

```
    noise = 0.1 * np.max(data) #any float works
```

```
    param_split = True
```

or

```
    param_split = False
```

Preprocess the data using the provided function:

```
train_ratio = 0.8
val_ratio = 0.1
data_train, data_val, data_test = processData(data=data, parameters=params,
                                                lags=11, num_sensors=40,
                                                train_ratio=train_ratio,
                                                val_ratio=val_ratio,
                                                param_split = param_split,
                                                constant_params=True,
                                                noise=noise)
```

8.5 Define the Model

Initialize the model with the desired parameters:

```
from SindyShredModel import SINDySHRED

model = SINDySHRED(max_degree=1, hidden_size=3, hidden_layers=2, threshold=0.3,
                   num_sensors=40, decoder_sizes=[40, 500], dropout=0.1)
```

8.6 Train the Model

Train the model using the training and validation datasets:

```
from SindyShredModel import fit_SindyShred_param

train_error, val_error = fit_SindyShred_param(model, train_dataset, val_dataset,
                                              batch_size=25, num_epochs=1000,
                                              lr=0.01, lambda_sindy=0.1, verbose=True,
                                              patience=8)
```

8.7 Test and Plot Results

Evaluate the model and visualize predictions: remember to select a parameter you want to visualize and to specify the test interval.

```
if param_split:
    k = np.random.randint(0, int(num_param*(1-train_ratio-val_ratio)))
    test_range = nt
else:
    k = np.random.randint(0, num_param)
    test_range = int(nt*(1-train_ratio-val_ratio))

from TestSindyShred import test_error, plot_sensors_comparison_shred,
                           plot_sensors_comparison_sindy,
                           plot_shred_comparison_2D

test_err_shred, test_err_sindy = test_error(model, test_dataset)

plot_sensors_comparison_shred(model, data_test, test_range, k)
plot_sensors_comparison_sindy(model, data_test, test_range, k)

instant = np.random.randint(0,test_range)
if param_split:
    plot_shred_comparison_2D(model, data_test, instant, nx, ny, test_range, nf, k)
else:
    plot_shred_comparison_2D(model, data_test, instant, nx, ny, test_range+1, nf, k)
```

This tutorial covers the full workflow, from loading data to training and evaluation. Adjust parameters as needed to tailor the model to your dataset.

References

[1] Liyao Gao, Jan P. Williams, and J. Nathan Kutz, *Sparse identification of nonlinear dynamics with Shallow Recurrent Decoder Networks*. Available at: https://openreview.net/forum?id=iyGkoWP6nA, 2024.

[2] Jan P. Williams, Olivia Zahn, and J. Nathan Kutz, *Sensing with shallow recurrent decoder networks*, Royal Society Open Science, DOI: https://doi.org/10.1098/rspa.2024.0054, 25 September 2024.

[3] S.L. Brunton, J.L. Proctor, and J.N. Kutz, *Discovering governing equations from data by sparse identification of nonlinear dynamical systems*, Proc. Natl. Acad. Sci. U.S.A., 113 (15): 3932–3937, DOI: https://doi.org/10.1073/pnas.1517384113, 2016.

[4] Paolo Conti, Giorgio Gobat, Stefania Fresca, Andrea Manzoni, and Attilio Frangi, *Reduced order modeling of parametrized systems through autoencoders and SINDy approach: continuation of periodic solutions*, Computer Methods in Applied Mechanics and Engineering, 411: 116072, DOI: https://doi.org/10.1016/j.cma.2023.116072, 2023.

[5] K. Champion, B. Lusch, J.N. Kutz, and S.L. Brunton, *Data-driven discovery of coordinates and governing equations*, Proc. Natl. Acad. Sci. U.S.A., 116 (45): 22445-22451, DOI: https://doi.org/10.1073/pnas.1906995116, 2019.

[6] Paolo Conti, Jonas Kneifl, Andrea Manzoni, Attilio Frangi, Jörg Fehr, Steven L. Brunton, J. Nathan Kutz, *VENI, VINDy, VICI: a variational reduced-order modeling framework with uncertainty quantification*, arXiv, DOI: https://doi.org/10.48550/arXiv.2405.20905.

[7] Johannes von Oswald, Christian Henning, Benjamin F. Grewe, João Sacramento, *Continual learning with hypernetworks*, arXiv, DOI: https://doi.org/10.48550/arXiv.1906.00695.

[8] Luca Rosafalco, Paolo Conti, Andrea Manzoni, Stefano Mariani, Attilio Frangi, *EKF–SINDy: Empowering the extended Kalman filter with sparse identification of nonlinear dynamics*, Computer Methods in Applied Mechanics and Engineering, Volume 431: 117264, DOI: https://doi.org/10.1016/j.cma.2024.117264, 2024.

[9] Lawal, Z.K.; Yassin, H.; Lai, D.T.C.; Che Idris, A., *Physics-Informed Neural Network (PINN) Evolution and Beyond: A Systematic Literature Review and Bibliometric Analysis*, Big Data Cogn. Comput., 2022, 6, 140, DOI: https://doi.org/10.3390/bdcc6040140.

[10] Cai, S., Mao, Z., Wang, Z., et al., *Physics-informed neural networks (PINNs) for fluid mechanics: a review*, Acta Mech. Sin., 37, 1727–1738, DOI: https://doi.org/10.1007/s10409-021-01148-1, 2021.

[11] Zeng Meng, Qiaochu Qian, Mengqiang Xu, Bo Yu, Ali Rıza Yıldız, Seyedali Mirjalili, *PINN-FORM: A new physics-informed neural network for reliability analysis with partial differential equation*, Computer Methods in Applied Mechanics and Engineering, Volume 414: 116172, DOI: https://doi.org/10.1016/j.cma.2023.116172, 2023.

[12] Saviz Mowlavi, Saleh Nabi, *Optimal control of PDEs using physics-informed neural networks*, Journal of Computational Physics, Volume 473: 111731, DOI: https://doi.org/10.1016/j.jcp.2022.111731, 2023.