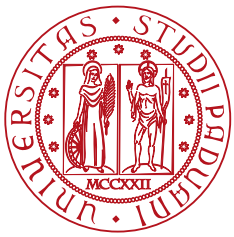


University of Padua

Department of Physics and Astronomy "Galileo Galilei"

Master Thesis in Physics of Data

Physics-Informed Machine Learning for High-Fidelity Synthetic Data Generation

Master Candidate

Daniele Ninni

Student ID 2044721

Internal Supervisor

Prof. Marco Zanetti

University of Padua

External Supervisor

Luca Gilli, Ph.D.

Clearbox AI

Academic Year

2022/2023

Ai miei genitori, a mia sorella e a tutta la mia famiglia
non posso che dirvi, con tutto il mio cuore,
grazie per sempre.

Abstract

Interest in synthetic data has grown rapidly in recent years. Synthetic data is artificially generated data with the same statistical properties as real-world data. This growth of interest can be attributed, on the one hand, to the increasing demand for large amounts of data to train AI/ML models and, on the other hand, to the recent development of effective methods for generating high-quality synthetic data. For example, generative AI models have demonstrated excellent capabilities in synthesizing complex datasets.

Unfortunately, many of the processes of interest are rare events or edge cases. Therefore, the amount of real data that can be used to train generative models is often insufficient, hence limiting their applicability. Furthermore, in the case of processes involving dynamical systems, generative models often fail to capture the underlying laws governing the dynamics, thus resulting in low-fidelity synthetic data. A possible strategy to overcome these limitations is to generate synthetic data using a physics-informed approach, that is, incorporating the knowledge of the governing physical laws into the generative model.

This thesis explores a possible approach for generating high-fidelity synthetic data using physics-informed ML. Specifically, the approach investigated in this work uses the SINDy Autoencoder network introduced by Champion et al. as a synthetic data generator. This approach is benchmarked with a commercial tool developed by Clearbox AI, a synthetic data provider. The generative models under study are tested on two datasets generated by nonlinear dynamical systems: a simulation dataset with dynamics defined by the Lorenz system and a real dataset acquired on a full-scale F-16 aircraft.

The results of the study show that the explored approach is a rather promising solution for generating high-fidelity synthetic data. However, the training procedure is significantly complicated by the presence of multiple competing loss terms. Moreover, the effectiveness of the approach appears to be strongly dependent on the dataset in use and on the complexity of the corresponding dynamical system.

Contents

List of Figures

List of Tables

List of Code Snippets

List of Acronyms

AE Autoencoder

AI Artificial Intelligence

CNN Convolutional Neural Network

DL Deep Learning

ELU Exponential Linear Unit

GAN Generative Adversarial Network

GDPR General Data Protection Regulation

LASSO Least Absolute Shrinkage and Selection Operator

LSQ Least Squares

ML Machine Learning

MMD Maximum Mean Discrepancy

NN Neural Network

PCA Principal Component Analysis

PIML Physics-Informed Machine Learning

ReLU Rectified Linear Unit

SINDy Sparse Identification of Nonlinear Dynamics

SINDyAE SINDy Autoencoder

STLSQ Sequentially Thresholded Least Squares

VAE Variational Autoencoder

1

Synthetic Data

Interest in synthetic data has grown rapidly in recent years. This growth can be attributed to two simultaneous trends. The first is the demand for large amounts of data to train Artificial Intelligence (AI) and Machine Learning (ML) models. The second is the recent development of effective methods for generating high-quality synthetic data. Both trends have led to the realization that synthetic data can solve various challenging problems quite effectively, especially in the AI/ML context. Specifically, some of these problems would be too costly or dangerous to address using more traditional methods (e.g., training autonomous driving models), or they would simply be intractable. As a result, more and more companies are adopting various data synthesis solutions to accelerate their business.

1.1 Definition

As stated by El Emam et al. in [25], synthetic data is not real data, but fake data that has the same statistical properties as real data. This means that if you are working with a synthetic dataset, you should get results similar to what you would get with real data. The degree to which a synthetic dataset is an accurate proxy for real data is a measure of *utility*. The process of generating synthetic data is referred to as *synthesis*. There are several types of data that can be synthesized. For example, they can be:

- *structured* data (e.g., a relational database);
- *unstructured* data (e.g., text, audio, video, images).

From the point of view of the synthesis process, there are three types of synthetic data. In particular, they can be:

1. generated from real data;
2. **not** generated from real data;
3. a hybrid of the previous two.

They are presented in the following subsections.

1.1.1 Synthesis from real data

The first type of synthetic data is generated from real data. Specifically, some real datasets are used to build a model that captures their distributions and structure, i.e. the multivariate relationships and interactions in the data. Once the model is built, synthetic data is sampled or generated from that model. If the model is a good representation of the real data, then the synthetic data will have statistical properties similar to those of the real data.

1.1.2 Synthesis without real data

The second type of synthetic data is not generated from real data. It is created using existing models or background knowledge.

Existing models can be either statistical models of a process (developed through a data collection mechanism) or simulations of a process. Agent-based models are a popular example of techniques capable of synthesizing data by simulating the actions and interactions of multiple autonomous agents in an attempt to reproduce and predict the process of interest.

Background knowledge can be, for example, knowledge of the behavior of a system or knowledge of the statistical distributions underlying a process. In such a case, it is relatively straightforward to build a model and sample from background knowledge to generate synthetic data. If the knowledge of the process is accurate, then the synthetic data will behave consistently with the real data. Of course, the use of background knowledge works only when the phenomenon of interest is thoroughly understood.

1.2 Benefits

In this section, the following benefits of data synthesis are discussed:

- providing more efficient access to data;
- enabling better analytics;
- improving imbalanced data.

1.2.1 More efficient access to data

Data access is critical to AI/ML projects. Specifically, data is needed to train and validate models. More broadly, data is also needed for testing and evaluating AI/ML technologies potentially developed by others.

Typically, data is collected for a particular purpose with the consent of the individual (e.g., for participating in a clinical research study). If you want to use that same data for a different purpose, such as to build a model to predict what kind of person is likely to participate in a clinical trial, then that is considered a secondary purpose.

Access to data for secondary purposes is becoming problematic. At the same time, the public is increasingly concerned about how its data is used and shared, and privacy laws are becoming stricter.

Contemporary privacy regulations, such as the General Data Protection Regulation (GDPR) in Europe, require a legal basis to use personal data for a secondary purpose. An example of such a legal basis would be additional consent or authorization from individuals before their data can be used. However, in many cases, this is impractical and can introduce bias into the data because consenters and non-consenters differ on important characteristics [14].

Given the difficulty of accessing data, open source or public datasets are sometimes used. These can certainly be a good starting point, but they lack diversity and are often not well matched to the problems that the models are intended to solve. Furthermore, open data may lack sufficient heterogeneity for robust training of models. For example, open data may not capture rare cases well enough.

Data synthesis can generate, rather efficiently and at scale, realistic data to work with. Synthetic data would not be considered identifiable personal data. Therefore, privacy regulations would not apply, and no additional consent would be required to use such synthetic data for secondary purposes.

1.2.2 Better analytics

A use case where synthesis can be applied is when real data does not exist – for example, if the process to be modeled is completely new, and creating or collecting a real dataset from scratch would be cost-prohibitive or impractical. Synthesized data can also cover edge or rare cases that are difficult, impractical, or unethical to collect in the real world.

Sometimes real data exists but is not labeled. Labeling a large amount of samples for supervised learning tasks can be time-consuming, and manual labeling is error-prone. Again, synthetic labeled data can be generated to accelerate model development. The synthesis process can ensure high accuracy in the labeling.

Synthetic data can be used to validate assumptions and demonstrate the kind of results that can be obtained with the models of interest. In other words, synthetic data can be used in an exploratory way. If this exploratory step leads to interesting and useful results, then it makes sense to go through the more complex process of getting the real data to build the final versions of the models.

Another scenario in which synthetic data can be valuable is when it is used to train an initial model before the real data is accessible. Once the real data is obtained, then the trained model can be used as a starting point for training with the real data. This can significantly speed up the convergence of the real data model (hence, reducing compute time) and can potentially result in a more accurate model. This is an example of using synthetic data for transfer learning.

The benefits of synthetic data can be dramatic. It can make impossible projects doable, significantly accelerate AI/ML initiatives, or result in a material improvement in the outcomes of AI/ML projects.

1.2.3 Imbalanced data

In many real-world ML scenarios, datasets are often imbalanced, which means that the distribution of classes or labels is significantly skewed, with one or more classes being underrepresented. Synthetic data generation techniques can be leveraged to create artificial samples for the minority classes. By generating synthetic instances that closely resemble the underrepresented classes, the resulting augmented dataset achieves a more balanced distribution, thereby mitigating the impact of class imbalance on model performance.

1.3 Utility evaluation

In this section, the following three possible approaches to assess the utility of synthetic data are presented:

- workload-aware evaluations;
- generic data utility metrics;
- subjective assessments of data utility.

Workload-aware metrics look at specific feasible analyses that would be performed on the data and compare the results from real and synthetic data. Such analyses can vary from simple descriptive statistics to more complex multivariate models. Typically, an analysis planned on real data is replicated on synthetic data.

Generic assessments would consider, for example, the distance between original and synthetic data. These often do not reflect the very specific analysis that will be performed on the data, but rather provide broadly useful utility indicators when future analysis plans are unknown. To interpret generic metrics, they need to be bounded (e.g., from 0 to 1), and there should be some accepted yardsticks for deciding whether a value is high enough or too low.

A subjective evaluation would require a large enough number of domain experts who would look at a random mix of real and synthetic records and then attempt to classify each as real or synthetic. If a record looks realistic enough, then it would be classified as real; if it has unexpected patterns or relationships, then it may be classified as synthetic. For example, for a health dataset, clinicians may be asked to perform the subjective classification. The accuracy of that classification would then be evaluated.

1.3.1 Distinguishability

Distinguishability is an approach to compare real and synthetic data in a multivariate way. It consists in finding out whether it is possible to build a model that can distinguish between real and synthetic records. Thus, a binary label is assigned to each record, i.e. a 1 if it is real and a 0 if it is synthetic (or vice versa). Then, a classification model is built to discriminate between real and synthetic data. This model is used to predict whether a record is real or synthetic.

This classifier can output a probability for each prediction. If the probability is closer to 1, then it is predicting that the record is real. If the probability is closer to 0, then it is predicting that the record is synthetic.

If the two datasets are exactly the same, then there will be no distinguishability between them. This happens when the synthetic data generator is overfitted, and thus effectively re-creates the original data. In such a case, the probability of each prediction will be 0.5, i.e., the classifier will not be able to distinguish between real and synthetic data. Similarly, if the label of *real* versus *synthetic* is assigned to the records completely at random, then the classifier will not be able to distinguish between them. Again, the probability of each prediction will be 0.5.

If the two datasets are completely different, then the classifier will be able to distinguish between them. High distinguishability corresponds to low data utility. In such a case, the probability of each prediction will be either 0 or 1.

In reality, synthetic datasets will fall somewhere in between. They should be at neither of these two extremes. Synthetic data that is difficult to distinguish from real data is considered to have relatively high utility.

It is importannt to consider multiple utility metrics in order to get a broader appreciation of the utility of the synthetic dataset. Each method of assessing utility covers a different dimension of utility that is complementary to the others.

1.4 Generative AI for synthesizing data

Generative AI models have demonstrated excellent capabilities in synthesizing complex datasets. There are two main types of Neural Network (NN) architectures that are used to generate synthetic data. Both can work well, and in some cases they have been combined. Both approaches have demonstrated quite high synthesis utility on complex datasets and are a very active area of research.

1.4.1 VAE

The first architecture is the Variational Autoencoder (VAE). It is an unsupervised method for learning a meaningful representation of a multi-dimensional dataset. Its NN components are the same as a traditional Autoencoder (AE), i.e. the encoder and the decoder. First, the encoder compresses the dataset into a more compact representation with fewer dimensions, which is often a multivariate Gaussian distribution. Then, the decoder takes that compressed representation and reconstructs the original input data. The VAE is trained by optimizing the similarity between the decoded data and the input data. In this context, a VAE works similarly to Principal Component Analysis (PCA), except that it can cap-

ture nonlinear relationships in the data. The main difference compared to traditional AEs is that the training process includes a regularization term that pushes the latent representations to be distributed according to a desired distribution, thus leading to a more meaningful latent space.

1.4.2 GAN

The second architecture is the Generative Adversarial Network (GAN). A GAN is made up of two components: a generator and a discriminator. The generator network takes as input random data, often sampled from a normal or uniform distribution, and generates synthetic data. The discriminator network takes as input both real and synthetic data and discriminates them, outputting a probability for each prediction. The output of that discrimination is then fed back to train the generator. A good synthetic model is created when the discriminator cannot distinguish between real and synthetic data. This adversarial training procedure is quite effective in improving the quality of the data created by the generator. However, it has been shown that this approach could lead to training instability and mode collapse, i.e., the generator could end up generating only a limited set of output types instead of exploring the entire distribution of training data.

1.4.3 Limitations

Synthetic data produced by generative AI models is characterized by some limitations, such as:

- rare events and edge cases;
- processes involving dynamical systems.

First, many of the processes of interest are rare events or edge cases. As a result, the amount of real data that can be provided to the generative model is often insufficient for it to learn to accurately reproduce the process itself.

Second, in the case of processes involving dynamical systems, the generative model often fails to capture the underlying laws governing the dynamics, thus resulting in low-fidelity synthetic data.

A possible strategy to overcome these limitations is to generate synthetic data using a *physics-informed* approach. This corresponds to incorporating the knowledge of the physical laws governing the dynamics of the system of interest into the generative model. Such physics-informed approaches are covered in Chapter 2.

2

Physics-Informed ML

In several real-world and scientific problems, systems that generate data are governed by physical laws. As stated by Hao et al. in [32], the seamless integration of (noisy) data and mathematical physics models can guide the ML model towards physically plausible solutions, improving accuracy and efficiency even in partially understood and high-dimensional contexts. Physics-Informed Machine Learning (PIML) is a learning paradigm aimed at building a model that leverages empirical data and physical knowledge to improve performance on a set of tasks that involve a physical mechanism. Kernel-based or NN-based regression methods offer effective, simple and meshless implementations. Moreover, it is possible to design specialized NN architectures that automatically satisfy some of the physical constraints for better accuracy, faster training and improved generalization.

2.1 How to embed physics in ML

As stated by Karniadakis et al. in [28], no predictive models can be built without assumptions and, consequently, no generalization performance can be expected from ML models without appropriate biases. Specific to PIML, there are currently three pathways that can be followed separately or in tandem to accelerate training and improve generalization of ML models by embedding physics into them. In detail, making a learning algorithm physics-informed amounts to introducing appropriate observational, inductive or learning biases that can steer the learning process towards physically consistent solutions. These biases are not mutually exclusive and can be combined to yield a very broad class of hybrid approaches.

2.1.1 Observational bias

Observational data is arguably the foundation of the recent success of ML. It is also the conceptually simplest way to introduce bias into ML. In the case of physical systems, thanks to the rapid development of sensor networks, it is possible to exploit a wealth of observations and monitor the evolution of complex phenomena across several spatial and temporal scales. Such observational data should reflect the underlying physical laws governing their generation and, in principle, can be used as a weak mechanism for embedding these laws into an ML model during its training phase. However, especially for over-parameterized Deep Learning (DL) models, a large amount of data is typically needed to reinforce these biases and generate predictions that respect certain symmetries and conservation laws. In this case, an immediate difficulty relates to the cost of data acquisition, which for many applications in the physical and engineering sciences could be prohibitively high, as observational data may be generated via expensive experiments or large-scale computational models.

2.1.2 Inductive bias

Another school of thought pertains to efforts focused on designing specialized NN architectures that implicitly embed any prior knowledge and inductive biases associated with a given predictive task. In other words, prior assumptions can be incorporated by tailored interventions to an ML model architecture, such that the predictions sought are guaranteed to implicitly satisfy a set of given physical laws, typically expressed in the form of certain mathematical constraints. It could be argued that this is the most principled way of making a learning algorithm physics-informed, since it allows the underlying physical constraints to be strictly satisfied. Despite their remarkable effectiveness, such approaches are currently limited to tasks characterized by relatively simple and well-defined physics or symmetry groups (e.g. translations, permutations, reflections, rotations) that are known a priori, and often require craftsmanship and elaborate implementations. Furthermore, their extension to more complex tasks is challenging, as the underlying invariances or conservation laws that characterize many physical systems are often poorly understood or hard to implicitly encode into an NN architecture.

2.1.3 Learning bias

Yet another school of thought approaches the problem of endowing an NN with prior knowledge from a different perspective. Instead of designing a specialized architecture that implicitly enforces this knowledge, it is possible to impose such constraints in a soft manner by appropriately choosing loss functions, constraints and inference algorithms that can modulate the training phase of an ML model to explicitly favor convergence towards solutions that adhere to the underlying physics. This approach can be viewed as a specific use case of multi-task learning, in which a learning algorithm is simultaneously constrained to fit the observed data, and to yield predictions that approximately satisfy a given set of physical constraints. Such soft penalty constraints constitute a very flexible way of introducing a broad class of physics-based biases into ML models.

2.2 Limitations

Despite the recent success of PIML across a range of applications, it is worth pointing out some of its current limitations.

2.2.1 Training algorithms and architectures

PIML models often involve training large-scale NNs with complicated loss functions, which generally consist of multiple terms and thus are highly non-convex optimization problems [5]. The terms in the loss function may compete with each other during training. Consequently, the training process may not be robust and sufficiently stable, and therefore convergence to the global minimum cannot be guaranteed [19]. To solve this issue, more robust NN architectures and training algorithms should be developed for diverse applications. Moreover, the design of effective NN architectures is currently done empirically by users, which could be very time-consuming. However, emerging meta-learning techniques can be used to automate this search. Furthermore, training and optimization of DL models is expensive, so it is crucial to speed up learning, for example through transfer learning. In addition, scalable and parallel training algorithms should be developed to take advantage of hardware such as GPUs and TPUs, using both data-parallel and model-parallel paradigms.

2.2.2 Data generation and benchmarks

In the ML community dealing with imaging, speech and natural language processing problems, the use of standard benchmarks is very common in order to assess algorithm improvement, reproducibility of results, and expected computational cost. For example, the UCI Machine Learning Repository is a collection of databases and data generators that are often used to compare the relative performance of new algorithms. Currently, the repository also includes experimental datasets related to the physical sciences. These datasets are useful and intended for data-driven modeling in ML, but in principle they can also be used for benchmarking PIML methods, assuming that proper parameterized physical models can be explicitly included in the databases. However, in many different physics-related applications, full-field data is required. Such data often cannot be obtained experimentally and tax computational resources heavily in terms of both time and memory. Therefore, careful consideration should be given to how to make these data publicly available, how to curate such valuable data, and how to include the physical models and all parameters required for the generation of these databases. In addition, it will take a concerted effort to design meaningful benchmarks that test accuracy and speedup of the new proposed PIML algorithms, which is a non-trivial task. Indeed, even for other established ML applications, there are still new developments on refining existing benchmarks and metrics, especially if software and hardware considerations are also relevant in such evaluations. In physical systems, these difficulties are exacerbated by the fact that the aim is to predict dynamics, and it will be complicated, for example, to determine how to capture or identify bifurcations in dynamical systems and chaotic states.

3

SINDy: Data-driven discovery of governing equations

Extracting governing equations from (noisy) measurement data is a central challenge in many diverse areas of science. The Sparse Identification of Nonlinear Dynamics (SINDy) algorithm is a regression technique, proposed by Brunton et al. in [16], which addresses the dynamical system discovery problem. Specifically, it aims to extract parsimonious dynamics from time-series data. It relies on the assumption that the governing equations are sparse in a high-dimensional nonlinear function space, i.e. there are only a few relevant terms that define the dynamics. This assumption holds for several physical systems in an appropriate basis. The algorithm performs sparse regression to determine the fewest terms required to accurately represent the data. The resulting sparse model identification inherently balances complexity with accuracy, thus avoiding overfitting the model to the data.

3.1 Mathematical formulation

Consider a dynamical system of the form:

$$\frac{d}{dt}\mathbf{x}(t) = \mathbf{f}(\mathbf{x}(t)), \tag{3.1}$$

where:

- $\mathbf{x}(t) \in \mathbb{R}^n$: state of the system at time t;
- $\mathbf{f}(\mathbf{x}(t))$: dynamic constraints defining the equations of motion of the system.

The key assumption is that the function $\mathbf{f}$ consists of only a few terms and is therefore sparse in the space of possible functions. In practice, the algorithm aims to determine this function $\mathbf{f}$ from data. To this end, a time history of the state $\mathbf{x}(t)$ and its time derivative $\dot{\mathbf{x}}(t)$ must be collected. In particular, $\dot{\mathbf{x}}(t)$ can be either measured directly or approximated numerically from $\mathbf{x}(t)$. The data are sampled at several times $t_1, t_2, \ldots, t_m$ and arranged into two matrices:

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}^T(t_1) \\ \mathbf{x}^T(t_2) \\ \vdots \\ \mathbf{x}^T(t_m) \end{bmatrix} = \begin{bmatrix} x_1(t_1) & x_2(t_1) & \cdots & x_n(t_1) \\ x_1(t_2) & x_2(t_2) & \cdots & x_n(t_2) \\ \vdots & \vdots & \ddots & \vdots \\ x_1(t_m) & x_2(t_m) & \cdots & x_n(t_m) \end{bmatrix}, \tag{3.2}$$

$$\dot{\mathbf{X}} = \begin{bmatrix} \dot{\mathbf{x}}^T(t_1) \\ \dot{\mathbf{x}}^T(t_2) \\ \vdots \\ \dot{\mathbf{x}}^T(t_m) \end{bmatrix} = \begin{bmatrix} \dot{x}_1(t_1) & \dot{x}_2(t_1) & \cdots & \dot{x}_n(t_1) \\ \dot{x}_1(t_2) & \dot{x}_2(t_2) & \cdots & \dot{x}_n(t_2) \\ \vdots & \vdots & \ddots & \vdots \\ \dot{x}_1(t_m) & \dot{x}_2(t_m) & \cdots & \dot{x}_n(t_m) \end{bmatrix}. \tag{3.3}$$

Next, an extensive library $\boldsymbol{\Theta}(\mathbf{X})$ is constructed. It consists of p candidate nonlinear functions of the columns of $\mathbf{X}$:

$$\boldsymbol{\Theta}(\mathbf{X}) = \begin{bmatrix} \boldsymbol{\Theta}_1(\mathbf{X}) & \boldsymbol{\Theta}_2(\mathbf{X}) & \cdots & \boldsymbol{\Theta}_p(\mathbf{X}) \end{bmatrix} \in \mathbb{R}^{m \times p}, \tag{3.4}$$

where $m \gg p$, that is, the number of data samples is larger than the number of candidate functions. Each column of $\boldsymbol{\Theta}(\mathbf{X})$ represents a candidate function for the right-hand side of Eq. 3.1. The choice of nonlinear functions typically reflects some knowledge about the system of interest. In practice, it may be helpful to test many different function bases and use the sparsity and accuracy of the resulting model as a diagnostic tool to determine the correct basis in which to represent the dynamics of the system.

By leveraging the assumption that only a few of these nonlinearities are active in each row of $\mathbf{f}$, it makes sense to set up a sparse regression problem to determine the sparse vectors of coefficients $\boldsymbol{\Xi}$ that determine which nonlinearities are active:

$$\boldsymbol{\Xi} = \begin{bmatrix} \boldsymbol{\Xi}_1 & \boldsymbol{\Xi}_2 & \cdots & \boldsymbol{\Xi}_n \end{bmatrix} \in \mathbb{R}^{p \times n}, \tag{3.5}$$

$$\dot{\mathbf{X}} = \boldsymbol{\Theta}(\mathbf{X})\boldsymbol{\Xi}. \tag{3.6}$$

Each column $\boldsymbol{\Xi}_k$ of $\boldsymbol{\Xi}$ is a sparse vector of coefficients determining which terms are active in the right-hand side for the k-th row equation $\dot{\mathbf{x}}_k = \mathbf{f}_k(\mathbf{x})$ in Eq. 3.1. Once $\boldsymbol{\Xi}$ is determined, a model of each row of Eq. 3.1 can be constructed as follows:

$$\dot{\mathbf{x}}_k = \mathbf{f}_k(\mathbf{x}) = \boldsymbol{\Theta}(\mathbf{x}^T)\boldsymbol{\Xi}_k. \tag{3.7}$$

Note that $\boldsymbol{\Theta}(\mathbf{x}^T)$ is a vector of symbolic functions of elements of $\mathbf{x}$ whereas $\boldsymbol{\Theta}(\mathbf{X})$ is a data matrix. Therefore:

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}) = \boldsymbol{\Xi}^T(\boldsymbol{\Theta}(\mathbf{x}^T))^T. \tag{3.8}$$

Each column of Eq. 3.6 requires a distinct optimization to find the sparse vector of coefficients $\boldsymbol{\Xi}_k$ for the k-th row equation.

Realistically, $\mathbf{X}$ is contaminated with noise and is often the only data available, so $\dot{\mathbf{X}}$ must be approximated numerically by differentiation. Thus, Eq. 3.6 does not hold exactly. Sparsity-promoting regression is used to solve for $\boldsymbol{\Xi}$ that result in parsimonious models, ensuring that each $\boldsymbol{\Xi}_k$ is sparse and therefore only a few columns of $\boldsymbol{\Theta}(\mathbf{X})$ are selected. An appealing aspect of this model discovery formulation is that it results in an overdetermined linear system for which many regularized solution techniques exist. For example, the Least Absolute Shrinkage and Selection Operator (LASSO) [6] is an L_1-regularized regression that promotes sparsity and works well with this type of data. However, it may be computationally expensive for very large datasets. The standard SINDy approach uses the Sequentially Thresholded Least Squares (STLSQ) algorithm to find the coefficients $\boldsymbol{\Xi}$, which is a proxy for L_0 optimization and has convergence guarantees [22].

3.2 Sequentially thresholded least squares

The STLSQ algorithm starts with a Least Squares (LSQ) solution for $\boldsymbol{\Xi}$ and then thresholds all coefficients that are smaller than a given cutoff value λ. Once the indices of the remaining nonzero coefficients are identified, another least squares solution for $\boldsymbol{\Xi}$ onto the remaining indices is obtained. These new coefficients are again thresholded using λ. The procedure is repeated until the nonzero coefficients converge. This algorithm is computationally efficient and rapidly converges to a sparse solution in a small number of iterations. The algorithm also benefits from simplicity, with a single parameter λ required to determine the degree of sparsity in $\boldsymbol{\Xi}$. The STLSQ algorithm is reported in Algorithm 1.

Algorithm 1 Sequentially Thresholded Least Squares (STLSQ)

Require: $\lambda \geq 0$
  $\boldsymbol{\Xi} \leftarrow \text{LSQ}(\boldsymbol{\Theta}, \dot{\mathbf{X}})$ {initial guess: least squares}
  repeat
    $\boldsymbol{\Xi}_{\text{old}} \leftarrow \boldsymbol{\Xi}$
    small_idxs $\leftarrow (|\boldsymbol{\Xi}| < \lambda)$ {find small coefficients}
    $\boldsymbol{\Xi}$[small_idxs] $\leftarrow 0$ {threshold small coefficients}
    for $k \in \{1, 2, \ldots, n-1, n\}$ **do**
      big_idxs $\leftarrow$ **not**(small_idxs$[:, k]$)
      $\boldsymbol{\Xi}$[big_idxs, k] $\leftarrow \text{LSQ}(\boldsymbol{\Theta}[:, \text{big\_idxs}], \dot{\mathbf{X}}[:, k])$
    end for
  until $\boldsymbol{\Xi} = \boldsymbol{\Xi}_{\text{old}}$
  return $\boldsymbol{\Xi}$

3.3 Limitations

The performance of the SINDy algorithm depends not only on the quality of the data but also on the choice of the measurement coordinates and the sparsifying function basis. In other words, the right coordinates and function basis are needed to yield sparse dynamics. However, it may be difficult to know them a priori, and the best choice is not always clear. Basic knowledge of the underlying physics may provide a reasonable choice. In fact, the sparsity and accuracy of the proposed sparse identified model may provide valuable diagnostic information about the correct measurement coordinates and basis in which to represent the dynamics. Actually, there is still a need for experts to find and exploit the symmetries of the system. Furthermore, the original SINDy method could be complemented by ML algorithms to extract useful features. A possible development in this direction is discussed in Chapter 4.

4

SINDy Autoencoder

The SINDy algorithm is able to identify both the structure and the parameters of a nonlinear dynamical system from data. The resulting models have the fewest terms necessary to describe the dynamics, balancing model complexity with descriptive ability, and thus promoting interpretability and generalizability. However, this algorithm fundamentally relies on an effective coordinate system in which the dynamics has a simple representation. In this regard, the SINDy Autoencoder (SINDyAE) is a data-driven method, proposed by Champion et al. in [23], for the simultaneous discovery of interpretable, sparse dynamical models and coordinates that enable these simple representations. Specifically, this approach combines a SINDy model and a deep AE network to perform a joint optimization aimed at discovering intrinsic coordinates that have an associated parsimonious nonlinear dynamical model. The resulting modeling framework leverages, on the one hand, the parsimony and interpretability of SINDy and, on the other hand, the universal approximation capabilities of deep NNs [4] to produce interpretable and generalizable models capable of extrapolation and forecasting. The SINDyAE addresses a major limitation of other model discovery approaches, which is that the proper choice of measurement coordinates is often unknown.

4.1 Architecture

The architecture of the SINDyAE is shown in Fig. 4.1.

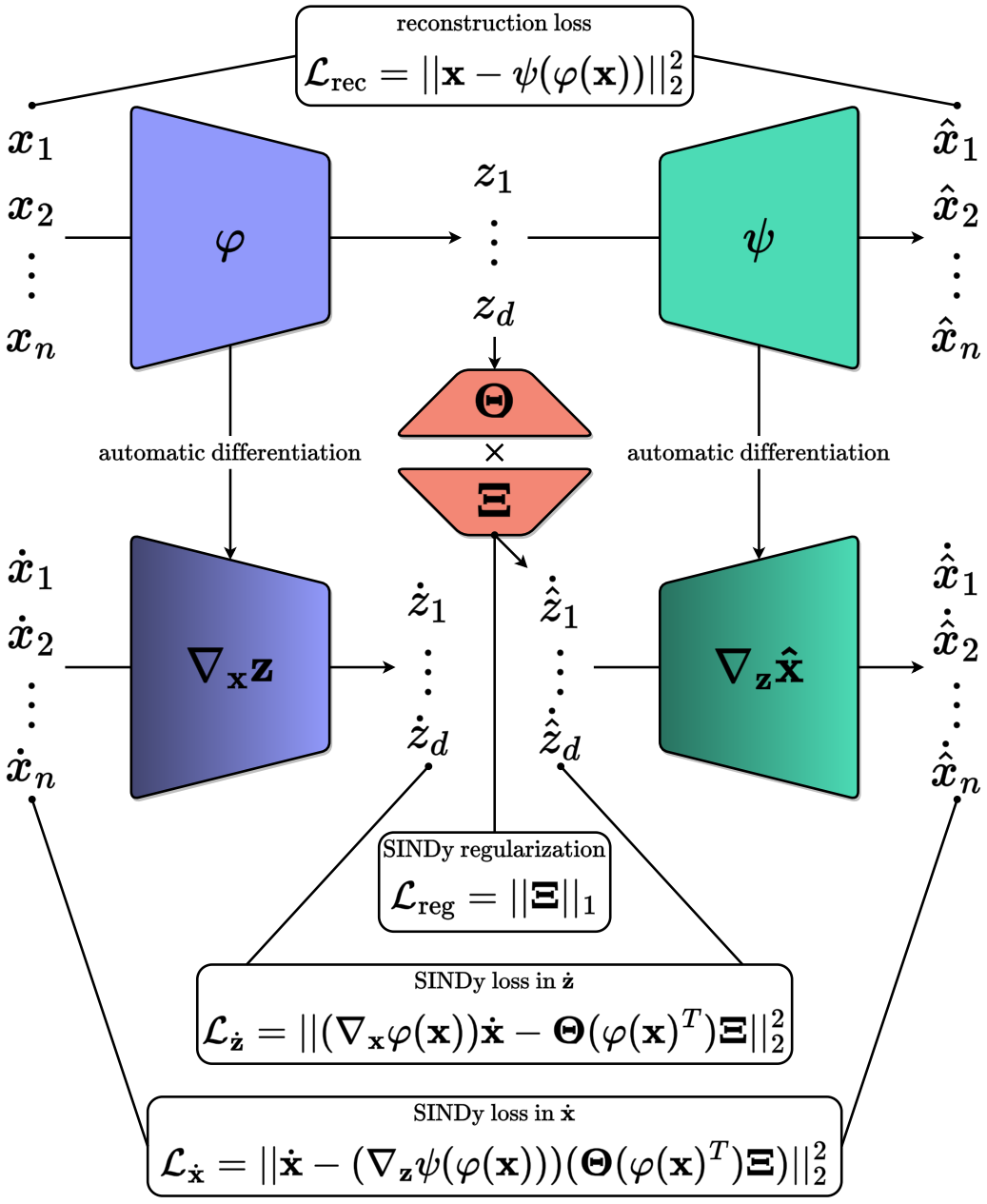


Figure 4.1: Architecture of the SINDyAE.

Consider again a dynamical system of the form 3.1. While this dynamical model may be dense in terms of functions of the original measurement coordinates $\mathbf{x}$, the SINDyAE seeks a set of reduced coordinates:

$$\mathbf{z}(t) = \varphi(\mathbf{x}(t)) \in \mathbb{R}^d, \tag{4.1}$$

where $d \ll n$ and with an associated dynamical model:

$$\frac{d}{dt}\mathbf{z}(t) = \mathbf{g}(\mathbf{z}(t)), \tag{4.2}$$

that provides a parsimonious description of the dynamics, that is, $\mathbf{g}$ contains only a few active terms. Along with the dynamical model, the method provides coordinate transforms that map the measurements to intrinsic coordinates via an encoder φ:

$$\mathbf{z} = \varphi(\mathbf{x}), \tag{4.3}$$

and back via a decoder ψ:

$$\hat{\mathbf{x}} = \psi(\mathbf{z}) \approx \mathbf{x}. \tag{4.4}$$

The coordinate transformation is achieved using an AE network architecture. Specifically, both the encoder and the decoder are fully connected feedforward NNs. Rather than performing a task such as prediction or classification, the network is trained to output an approximate reconstruction of its input, and the restrictions placed on the network architecture (e.g., the type, number, and size of the hidden layers) determine the properties of the intrinsic coordinates [18]. AEs are known to produce nonlinear generalizations of PCA [3]. A common choice is that the dimensionality of the intrinsic coordinates $\mathbf{z}$, determined by the number of units in the corresponding hidden layer, is much lower than that of the input data $\mathbf{x}$. In this case, the AE learns a nonlinear embedding into a reduced latent space. In particular, the SINDyAE takes measurement data $\mathbf{x}(t) \in \mathbb{R}^n$ from a dynamical system as input and learns intrinsic coordinates $\mathbf{z}(t) \in \mathbb{R}^d$, where $d \ll n$ is chosen as a hyperparameter prior to training the network.

While AEs can be trained in isolation to discover useful coordinate transformations and dimensionality reductions, there is no guarantee that the intrinsic coordinates learned will have associated sparse dynamical models. On the contrary, the SINDyAE is required to learn coordinates associated with parsimonious dynamics by simultaneously learning a SINDy model for the dynamics of the intrinsic coordinates $\mathbf{z}$. This regularization is achieved by constructing a library $\boldsymbol{\Theta}(\mathbf{z})$ of candidate basis functions (e.g., polynomials):

$$\boldsymbol{\Theta}(\mathbf{z}) = \begin{bmatrix} \boldsymbol{\Theta}_1(\mathbf{z}) & \boldsymbol{\Theta}_2(\mathbf{z}) & \cdots & \boldsymbol{\Theta}_p(\mathbf{z}) \end{bmatrix}, \tag{4.5}$$

and learning a sparse set of coefficients $\boldsymbol{\Xi}$:

$$\boldsymbol{\Xi} = \begin{bmatrix} \boldsymbol{\Xi}_1 & \boldsymbol{\Xi}_2 & \cdots & \boldsymbol{\Xi}_d \end{bmatrix}, \tag{4.6}$$

that defines the following dynamical system:

$$\frac{d}{dt}\mathbf{z}(t) = \mathbf{g}(\mathbf{z}(t)) = \boldsymbol{\Theta}(\mathbf{z}(t))\boldsymbol{\Xi}. \tag{4.7}$$

While the library must be specified prior to training, the coefficients $\boldsymbol{\Xi}$ are learned with the NN parameters as part of the training procedure. Assuming derivatives $\dot{\mathbf{x}}(t)$ of the original states are available or can be computed, the derivative of the encoder variables can be calculated as:

$$\dot{\mathbf{z}}(t) = \nabla_{\mathbf{x}}\varphi(\mathbf{x}(t))\dot{\mathbf{x}}(t). \tag{4.8}$$

4.2 Loss function

The loss function used to train the SINDyAE is a weighted sum of the following four terms:

- AE reconstruction $\mathcal{L}_{\text{rec}}$;
- SINDy prediction on the input variables $\mathcal{L}_{d\mathbf{x}/dt}$;
- SINDy prediction on the latent variables $\mathcal{L}_{d\mathbf{z}/dt}$;
- SINDy coefficient regularization $\mathcal{L}_{\text{reg}}$.

First, $\mathcal{L}_{\text{rec}}$ ensures that the AE can accurately reconstruct the input data from the intrinsic coordinates.

Then, $\mathcal{L}_{d\mathbf{x}/dt}$ and $\mathcal{L}_{d\mathbf{z}/dt}$ ensure that the discovered SINDy model captures the dynamics of the system. On the one hand, $\mathcal{L}_{d\mathbf{x}/dt}$ ensures that the model can predict the time derivatives of the original variables. On the other hand, $\mathcal{L}_{d\mathbf{z}/dt}$ ensures that the model can predict the time derivatives of the latent variables.

Finally, $\mathcal{L}_{\text{reg}}$ is an L_1 regularization term that promotes sparsity of the SINDy coefficients $\boldsymbol{\Xi}$ and therefore encourages a parsimonious model for the dynamics.

For a dataset with m input samples, each loss is explicitly defined as follows:

$$\begin{aligned}
\mathcal{L}_{\text{rec}} &= \frac{1}{m}\sum_{i=1}^{m} \|\mathbf{x}_i - \psi(\varphi(\mathbf{x}_i))\|_2^2, \\
\mathcal{L}_{d\mathbf{x}/dt} &= \frac{1}{m}\sum_{i=1}^{m} \left\|\dot{\mathbf{x}}_i - (\nabla_{\mathbf{z}}\psi(\varphi(\mathbf{x}_i)))(\boldsymbol{\Theta}(\varphi(\mathbf{x}_i)^T)\boldsymbol{\Xi})\right\|_2^2, \\
\mathcal{L}_{d\mathbf{z}/dt} &= \frac{1}{m}\sum_{i=1}^{m} \left\|\nabla_{\mathbf{x}}\varphi(\mathbf{x}_i)\dot{\mathbf{x}}_i - \boldsymbol{\Theta}(\varphi(\mathbf{x}_i)^T)\boldsymbol{\Xi}\right\|_2^2, \\
\mathcal{L}_{\text{reg}} &= \|\boldsymbol{\Xi}\|_1.
\end{aligned} \tag{4.9}$$

The total loss function L_{tot} is the following:

$$\mathcal{L}_{\text{tot}} = \mathcal{L}_{\text{rec}} + \lambda_1\mathcal{L}_{d\mathbf{x}/dt} + \lambda_2\mathcal{L}_{d\mathbf{z}/dt} + \lambda_3\mathcal{L}_{\text{reg}}, \tag{4.10}$$

where the hyperparameters $\lambda_1, \lambda_2, \lambda_3$ determine the relative weighting of the various loss terms.

4.3 Activation functions

Nonlinear activation functions are applied at all layers of the SINDyAE, except for the last layer of the encoder and the last layer of the decoder. The authors of [23] use the Sigmoid activation function:

$$f(x) = \frac{1}{1+e^{-x}}. \tag{4.11}$$

They state that other activation functions, such as Rectified Linear Unit (ReLU):

$$f(x) = max(0, x), \tag{4.12}$$

and Exponential Linear Unit (ELU):

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ a(e^x - 1), \ a \geq 0 & \text{otherwise} \end{cases}, \tag{4.13}$$

may also be used and appear to achieve similar results.

4.4 Training

4.4.1 Initialization

Each instance of training has a different random initialization of the network weights. The weight matrices are initialized using the Xavier initialization: the entries are sampled from a random uniform distribution over $[-\sqrt{6/a}, \sqrt{6/a}]$, where a is the dimension of the input plus the dimension of the output [10]. The bias vectors are initialized to 0 and the SINDy coefficients $\boldsymbol{\Xi}$ are initialized to 1.

4.4.2 Sequential thresholding

In addition to the L_1 regularization, to obtain parsimonious dynamical models (i.e., models with only a few active terms), a *sequential thresholding* procedure is used during training. It promotes L_0 sparsity on the coefficients in $\boldsymbol{\Xi}$, which represent the dynamics on the latent variables $\mathbf{z}$. This technique is inspired by Algorithm 1 used for SINDy, which combines LSQ fitting with sequential thresholding to obtain a sparse model. In practice, a threshold that determines the minimum magnitude of the coefficients in $\boldsymbol{\Xi}$ is specified. Then, at fixed intervals throughout the training, all coefficients with a magnitude below the threshold are set to 0, effectively removing these terms from the SINDy model. This is achieved using a mask $\boldsymbol{\Upsilon}$, consisting of 1s and 0s, that determines which terms remain in the SINDy model. Thus, the true SINDy terms in the loss function are as follows:

$$\begin{aligned} \mathcal{L}_{d\mathbf{x}/dt} &= \frac{1}{m}\sum_{i=1}^{m} \left\| \dot{\mathbf{x}}_i - (\nabla_{\mathbf{z}}\psi(\varphi(\mathbf{x}_i)))(\boldsymbol{\Theta}(\varphi(\mathbf{x}_i)^T)(\boldsymbol{\Upsilon} \circ \boldsymbol{\Xi})) \right\|_2^2, \\ \mathcal{L}_{d\mathbf{z}/dt} &= \frac{1}{m}\sum_{i=1}^{m} \left\| \nabla_{\mathbf{x}}\varphi(\mathbf{x}_i)\dot{\mathbf{x}}_i - \boldsymbol{\Theta}(\varphi(\mathbf{x}_i)^T)(\boldsymbol{\Upsilon} \circ \boldsymbol{\Xi}) \right\|_2^2, \end{aligned} \tag{4.14}$$

where $\boldsymbol{\Upsilon}$ is passed in separately and not updated by the optimization algorithm. Once a term has been thresholded out during training, it is permanently removed from the SINDy model. Then, training resumes using only the remaining terms. Therefore, the number of active terms in the SINDy model can only be decreased as training continues. The L_1 regularization on $\boldsymbol{\Xi}$ encourages the model coefficients to decrease in magnitude, which combined with the sequential thresholding produces a parsimonious dynamical model.

4.4.3 Fine-tuning

While the L_1 regularization penalty on $\boldsymbol{\Xi}$ promotes sparsity in the resulting SINDy model, it also encourages nonzero terms to have smaller magnitudes. This results in a trade-off between accurately reconstructing the dynamics of the system and reducing the magnitude of the SINDy coefficients, where the trade-off is determined by the relative magnitudes of the loss weight penalties λ_1, λ_2 and the regularization penalty λ_3. The specified training procedure therefore typically results in models with coefficients that are slightly smaller in magnitude than those which would best reproduce the dynamics. To account for this, an additional coefficient refinement period is added to the training procedure. To perform this refinement, the sparsity pattern in the dynamics is locked in by fixing the coefficient mask $\boldsymbol{\Upsilon}$. Then, the SINDyAE is fine-tuned for a certain number of epochs without the L_1 regularization on $\boldsymbol{\Xi}$. This ensures that the best coefficients are found for the resulting SINDy model and also allows the training procedure to refine the encoder and decoder parameters. This procedure is analogous to running a debiased regression following the use of LASSO to select model terms [15].

4.5 Choice of hyperparameters

The training procedure described in the previous section requires the choice of the following hyperparameters:

- number of intrinsic coordinates d;
- loss weight penalties λ_1, λ_2, λ_3;
- function library $\boldsymbol{\Theta}$;
- SINDy coefficient threshold.

These choices greatly impact the success of the training procedure. In this regard, the authors of [23] outline the following guidelines.

The most important choice is the number of intrinsic coordinates d, as this affects the interpretation of the reduced space and the associated dynamical model. To choose d, the authors suggest first training a standard AE without the associated SINDy model to determine the minimum number of coordinates needed to reproduce the input data. As simpler models (i.e. lower d) are typically easier to interpret, the smallest d possible should be used to start. Once this minimum is found, the full SINDyAE can be trained. It is possible that more coordinates

may be needed to capture the dynamics, in which case the method may accurately reproduce the input but fail to find a good model for the dynamics. In this case, d could be increased from the minimum until a suitable dynamical model is found. Choosing d that is greater than necessary may still result in a valid model for the dynamics. However, obtaining a sparse model may be more difficult.

The choice of loss weight penalties λ_1, λ_2, λ_3 also has a significant impact on the success of the training procedure. The first parameter λ_1 determines the importance of the SINDy prediction in the original input space. The authors suggest to choose λ_1 to be slightly less than the ratio $\sum_{i=1}^{m} \|\mathbf{x}_i\|_2^2 / \sum_{i=1}^{m} \|\dot{\mathbf{x}}_i\|_2^2$. This slightly prioritizes the reconstruction of $\mathbf{x}$ over the prediction of $\dot{\mathbf{x}}$ in the training. This is important to ensure that the AE weights are being trained to reproduce $\mathbf{x}$ and that it is the SINDy model that gives an accurate prediction of $\dot{\mathbf{x}}$. Regarding the second parameter λ_2, the authors choose it to be one or two orders of magnitude less than λ_1. If λ_2 is too large, it encourages shrinking of the magnitude of $\dot{\mathbf{z}}$ to minimize $\mathcal{L}_{d\mathbf{z}/dt}$; however, having λ_2 nonzero encourages a good prediction by the SINDy model in the $\dot{\mathbf{z}}$ space. The third parameter λ_3 determines the strength of the regularization of the SINDy model coefficients $\boldsymbol{\Xi}$ and thus affects the sparsity of the resulting models. If λ_3 is too large, the model will be too simple and achieve poor prediction. If it is too small, the models will be non-sparse and prone to overfitting. This loss weight requires the most tuning and should typically be chosen last by trying a range of values and assessing the level of sparsity in the resulting model.

In addition to the hyperparameters used in the NN training, the SINDyAE requires the choice of a library of functions $\boldsymbol{\Theta}$ for the SINDy model. In general, the best library functions to use may be unknown, and choosing the wrong library functions can obscure the simplest model. A recommended practice is to start with polynomial models, as many common physical models contain polynomials representing dominant balance terms or are a coordinate transformation away from a normal form characterized by a few polynomial nonlinearities. Polynomials can also represent Taylor series approximations for a broad class of smooth functions. For some systems, the choice of library functions could be informed by application-specific knowledge. Alternatively, one might attempt to learn the library functions, for example, using another NN layer. However, this approach would generally hamper the interpretability and generalizability of the resulting dynamical model. Other approaches, such as kernel-based methods, also have the potential to enable more flexible library representations [13].

4.6 Limitations

A limitation of the SINDyAE is the requirement for clean measurement data that is approximately noise-free. Fitting a continuous-time dynamical system with SINDy requires reasonable estimates of the derivatives, which may be difficult to obtain from noisy data. Approaches for estimating derivatives from noisy data, such as the total variation regularized derivative, can prove useful in providing derivative estimates [11]. Moreover, there are NN architectures explicitly constructed to separate signals from noise [24], which can be used as a preprocessing step. Alternatively, the SINDyAE can be used to fit a discrete-time dynamical system, in which case derivative estimates are not required. It is also possible to use the integral formulation of SINDy to abate noise sensitivity [21].

4.7 SINDyAE as a synthetic data generator

A major problem with DL approaches is that models are typically neither interpretable nor generalizable. Specifically, NNs trained solely for prediction may fail to generalize to classes of behaviors not seen in the training set. Instead, the SINDyAE is an approach for using NNs to obtain classically interpretable models through the discovery of low-dimensional dynamical systems, which can often have physical interpretations. Although the AE network still has the same limited interpretability and generalizability as other NNs, the dynamical model has the potential to generalize to other parameter regimes of the dynamics.

Such generalization potential of the SINDyAE leads to the main idea behind this thesis work, which is to exploit the SINDyAE as a synthetic data generator. Specifically, the idea is to leverage the sparse dynamical model learned by the SINDyAE in the latent coordinates to generate high-fidelity synthetic data in the original measurement coordinates. The synthesis process should take place according to the following steps:

1. select a sample x_0 in the original measurement space as the initial condition;
2. map x_0 to the latent space via the encoder φ of the SINDyAE to get z_0;
3. starting from z_0, simulate the latent dynamical model learned by the SINDyAE forward in time to generate Z_s, i.e. synthetic data in the latent space;
4. map Z_s back to the original measurement space via the decoder ψ of the SINDyAE to get X_s, i.e. synthetic data in the original measurement space.

The SINDyAE implementation developed for this thesis is discussed in Chapter 5.

5

Implementation

The original implementation of the SINDyAE in TensorFlow is available at the GitHub repository SindyAutoencoders. In this thesis work, the SINDyAE is implemented from scratch in PyTorch 2.0. In detail, the PyTorch Lightning framework is adopted. PyTorch Lightning is an open source Python library that provides a high-level interface for PyTorch. Specifically, it is a lightweight, high-performance framework that organizes PyTorch code to make DL experiments easier to read and reproduce. Moreover, it is designed to create scalable models that can easily run on distributed hardware while keeping them hardware-agnostic. The Python libraries used to implement the SINDyAE are reported in Code 5.1.

```
from tqdm import tqdm, trange
import numpy as np
import pandas as pd
import pysindy as ps
import torch
import lightning.pytorch as pl
```

Code 5.1: Python libraries used to implement the SINDyAE.

The SINDyAE is implemented as a class named `SINDyAutoencoder` which inherits from the `pl.LightningModule` class. Thanks to the LightningModule, it is possible to organize the PyTorch code into the methods reported in Table 5.1. Each of these methods is discussed exhaustively in the following sections.

5.1 Class initialization

The initialization arguments of `SINDyAutoencoder` are reported in Table 5.2.

name	description
`__init__`	class initialization
`init_params`	network parameter initialization
`forward`	forward pass
`training_step`	training loop
`validation_step`	validation loop
`on_train_epoch_end`, `on_train_end`	sequential thresholding
`configure_optimizers`	optimizer and learning rate scheduler
`configure_callbacks`	early stopping
`fit`	fitting
`show_history`	training history
`sindy_coeffs`	SINDy coefficients
`simulate`	synthetic data generation

Table 5.1: Methods in which the PyTorch code for the SINDyAE is organized.

argument	type	description
`experiment_name`	string or None	name of the experiment
`scaler_name`	string or None	name of the data scaler
`seed`	int or None	PyTorch Lightning seed
`dim_input`	int	dimension of measurement space
`dim_latent`	int	dimension of latent space
`layer_widths_encoder`	list[int]	widths of the encoder layers
`layer_widths_decoder`	list[int]	widths of the decoder layers
`activation`	`torch.nn.␣()`	activation function
`loss_weight`	dict{string : float}	loss weight penalties
`max_epochs_main`	int	# epochs for main training
`max_epochs_refinement`	int	# epochs for refinement training
`optimizer`	`torch.optim.␣`	training optimizer
`lr`	float	learning rate
`lr_patience`	int	learning rate patience
`es_patience`	int	early stopping patience
`threshold`	float	sequential thresholding – threshold
`threshold_freq`	int	sequential thresholding – frequency
`poly_order`	int	maximal degree of the polynomial features
`include_bias`	bool	whether to include the bias feature
`include_sin`	bool	whether to include sine features
`include_cos`	bool	whether to include cosine features
`n_frequencies`	int	# trigonometric frequencies to include
`init_weight`	`torch.nn.init.␣`	weight initializer
`init_bias`	`torch.nn.init.␣`	bias initializer
`init_sindy`	`torch.nn.init.␣`	SINDy coefficient initializer

Table 5.2: Initialization arguments of the `SINDyAutoencoder` class.

5.1.1 Encoder and decoder

Both encoder and decoder are implemented as `torch.nn.Sequential` containers. First, the input `torch.nn.Linear` layer is added to the container. Then, the subsequent layers are added to it by calling the `append` method of the container. The activation function is applied to all layers except the last one. Finally, the network parameters are initialized by calling the `init_params` method of the `SINDyAutoencoder` class. This implementation supports the use of an arbitrary number of layers. It is reported in Code 5.2.

```
# encoder
self.encoder = torch.nn.Sequential(torch.nn.Linear(in_features=dim_input,
↪out_features=layer_widths_encoder[0], bias=True), activation)
for i in range(len(layer_widths_encoder)-1):
    self.encoder.append(torch.nn.Linear(in_features=layer_widths_encoder[i],
  ↪out_features=layer_widths_encoder[i+1], bias=True))
    self.encoder.append(activation)
self.encoder.append(torch.nn.Linear(in_features=layer_widths_encoder[-1], out_features=dim_latent,
↪bias=True))
self.encoder.apply(self.init_params)

# decoder
self.decoder = torch.nn.Sequential(torch.nn.Linear(in_features=dim_latent,
↪out_features=layer_widths_decoder[0], bias=True), activation)
for i in range(len(layer_widths_decoder)-1):
    self.decoder.append(torch.nn.Linear(in_features=layer_widths_decoder[i],
  ↪out_features=layer_widths_decoder[i+1], bias=True))
    self.decoder.append(activation)
self.decoder.append(torch.nn.Linear(in_features=layer_widths_decoder[-1], out_features=dim_input,
↪bias=True))
self.decoder.apply(self.init_params)
```

Code 5.2: Encoder and decoder networks.

5.1.2 Feature library

Thanks to the PySINDy library, it is potentially straightforward to instantiate the feature library. In fact, it includes the following submodules:

- `ps.feature_library.PolynomialLibrary` (polynomial features);
- `ps.feature_library.FourierLibrary` (trigonometric features);
- `ps.feature_library.GeneralizedLibrary` (to unify multiple libraries).

However, PySINDy does not support PyTorch tensors: it automatically converts them to NumPy arrays before computing the features. Such conversions significantly slow down the computations involved in training the SINDyAE. This

slowdown is even more evident in the case of GPU training, since each PyTorch tensor must be moved to CPU before it can be converted into a NumPy array.

To overcome this limitation, the following workaround is adopted. Specifically, the PySINDy submodules above are used for the sole purpose of obtaining the expression of all the possible features that can be calculated according to both the dimension of the latent space and the related arguments of Table 5.2. In practice, the `get_feature_names` method of the `GeneralizedLibrary` is called to obtain a list of strings, each representing the expression of a specific feature to be calculated. Then, for each string, each function is mapped into the corresponding PyTorch function through an ad hoc dictionary. Finally, these functions are evaluated in the forward pass using the Python built-in function `eval`. The implementation of the feature library is reported in Code 5.3.

```
feature_library = [ps.feature_library.PolynomialLibrary(degree=poly_order,
↪include_bias=include_bias)]
if n_frequencies > 0:
    feature_library.append(ps.feature_library.FourierLibrary(n_frequencies=n_frequencies,
  ↪include_sin=include_sin, include_cos=include_cos))
self.pysindy_library = ps.feature_library.GeneralizedLibrary(feature_library)
feature_library =
↪ps.feature_library.GeneralizedLibrary(feature_library).fit(np.random.random(size=(1,dim_latent)))
self.input_features  = [f'z{i+1}' for i in range(dim_latent)]
self.output_features = feature_library.get_feature_names(input_features=self.input_features)
feature_library = feature_library.get_feature_names(input_features=self.input_features)
features_dict   = {'^':'**', ' ':'*', 'sin':'torch.sin', 'cos':'torch.cos'}
for i in range(dim_latent):
    features_dict[f'z{i+1}'] = f'z[:,{i}]'
for i in range(len(feature_library)):
    for key, value in features_dict.items():
        if key in feature_library[i]:
            feature_library[i] = feature_library[i].replace(key, value)
self.feature_library = feature_library
```

Code 5.3: Feature library.

5.1.3 SINDy model

The SINDy model is implemented as a `torch.nn.Linear` layer whose weights are the SINDy coefficients $\boldsymbol{\Xi}$. They are initialized through the argument `init_sindy` of Table 5.2. The implementation of the SINDy model is reported in Code 5.4.

```
self.sindy = torch.nn.Linear(in_features=len(self.output_features), out_features=dim_latent,
↪bias=False)
self.init_sindy(self.sindy.weight)
```

Code 5.4: SINDy model.

5.2 Network parameter initialization

Network parameters are initialized by calling the `init_params` method. It uses the arguments `init_weight` and `init_bias` of Table 5.2 to initialize the weights and biases, respectively. In detail, the `apply` method of the specific PyTorch module is called to apply `init_params` recursively to each submodule. The implementation of the network parameter initialization is reported in Code 5.5.

```
def init_params(self, module):

    if isinstance(module, torch.nn.Linear):
        self.init_weight(module.weight)

        if (module.bias is not None) and (self.init_bias is not None):
            self.init_bias(module.bias)
```

Code 5.5: Network parameter initialization.

5.3 Forward pass

5.3.1 Input reconstruction

Each batch `data` of size m consists of a `torch.utils.data.TensorDataset`, that is, a PyTorch Dataset wrapping the following tensors:

- `trajectory` (size: $m \times 1$);
- `t` (size: $m \times 1$);
- `x` (size: $m \times n$);
- `x_dot` (size: $m \times n$);

where n is the dimension of the measurement space. First, `x` is passed through the encoder to get the tensor `z` of size $m \times d$, where d is the dimension of the latent space. Then, `z` is passed through the decoder to get the tensor `x_hat` of size $m \times n$. The implementation of the input reconstruction is reported in Code 5.6.

```
trajectory, t, x, x_dot = data
x.requires_grad = True

z     = self.encoder(x)
x_hat = self.decoder(z)
```

Code 5.6: Input reconstruction.

5.3.2 Feature calculation

The feature matrix `Theta_z` is initialized as a `torch.zeros` tensor of size $m \times p$ where p is the size of the feature library. Then, each feature in the feature library is calculated using the Python function `eval` and assigned to the i-th column of `Theta_z`. The implementation of the feature calculation is reported in Code 5.7.

```
Theta_z = torch.zeros(size=(len(x),len(self.feature_library)), dtype=torch.float,
↪device=self.device)
for i, feature in enumerate(self.feature_library):
    Theta_z[:,i] = eval(feature)
```

Code 5.7: Feature calculation.

5.3.3 Latent derivative estimation

The time derivative of the tensor `z` is estimated from the SINDy model by passing the feature matrix `Theta_z` to the SINDy layer. Thus, the SINDy coefficients `Xi` coincide with the weights of the SINDy layer. The *complexity* of the SINDy model, i.e. the number of nonzero coefficients, is logged throughout the training. The implementation of the latent derivative estimation is reported in Code 5.8.

```
z_dot = self.sindy(Theta_z)
Xi    = self.sindy.weight

# log complexity of SINDy model
self.log('complexity', Xi.count_nonzero(), on_step=False, on_epoch=True, prog_bar=True)
```

Code 5.8: Latent derivative estimation.

5.3.4 Loss calculation

The SINDy losses `loss_x_dot` and `loss_z_dot` and the reconstruction loss `loss_rec` are calculated using the `torch.nn.MSELoss` function for each record in the batch and then averaged over the whole batch.

The SINDy losses, for optimization reasons, are calculated only if their loss weight penalties are positive. Moreover, they require the computation of the Jacobians $\nabla_{\mathbf{x}}\mathbf{z}$ and $\nabla_{\mathbf{z}}\hat{\mathbf{x}}$. However, it is difficult to compute these quantities efficiently using `torch.autograd`, i.e. PyTorch's automatic differentiation engine. This difficulty arises because `autograd` computes vector-Jacobian products. Therefore, to compute the full Jacobian, it would be computed row-by-row using a different

unit vector each time. Instead, it is possible to leverage `torch.vmap` to get rid of the for-loop and vectorize the computation. Indeed, `vmap` pushes the outer loop down into the primitive operations of the functions in order to obtain better performance. PyTorch 2.0 provides two APIs to compute `vmap`-powered Jacobians: `torch.func.jacrev` and `torch.func.jacfwd`. They can be substituted for each other, but have different performance characteristics. In this regard, the PyTorch documentation provides a general rule of thumb. In detail, assuming that the Jacobian is to be computed for a $\mathbb{R}^d \to \mathbb{R}^n$ function, then: it is recommended to use `jacfwd` if $n > d$, otherwise it is recommended to use `jacrev`.

Actually, the SINDy losses require the computation of the Jacobian of a batch of outputs with respect to a batch of inputs. That is, given a batch of inputs of shape $m \times n$ and a $\mathbb{R}^n \to \mathbb{R}^d$ function, a Jacobian of shape $m \times d \times n$ should be computed. Once again, the easiest way to do this is to use `vmap`.

The regularization loss `loss_reg` is calculated only if its loss weight penalty is positive and the SINDyAE is not in the refinement training phase. This loss is calculated using the `torch.abs` function (to take the absolute value of the SINDy coefficients) followed by the `torch.mean` function (to take their mean value).

Each loss is multiplied by its loss weight penalty before the forward pass is completed. The implementation of the loss calculation is reported in Code 5.9.

```
# reconstruction loss
loss_rec = torch.nn.MSELoss(reduction='sum')(x, x_hat) / len(x)
loss_rec = self.loss_weight['reconstruction'] * loss_rec

# SINDy loss in x_dot
loss_x_dot = 0
if self.loss_weight['x_dot'] > 0:
    decoder_jacobian = torch.vmap(torch.func.jacfwd(self.decoder))(z)
    loss_x_dot       = torch.nn.MSELoss(reduction='sum')(x_dot,
  ↪torch.vmap(torch.matmul)(decoder_jacobian, z_dot)) / len(x)
    loss_x_dot       = self.loss_weight['x_dot'] * loss_x_dot

# SINDy loss in z_dot
loss_z_dot = 0
if self.loss_weight['z_dot'] > 0:
    encoder_jacobian = torch.vmap(torch.func.jacrev(self.encoder))(x)
    loss_z_dot       =
  ↪torch.nn.MSELoss(reduction='sum')(torch.vmap(torch.matmul)(encoder_jacobian, x_dot), z_dot) /
  ↪len(x)
    loss_z_dot       = self.loss_weight['z_dot'] * loss_z_dot

# regularization loss
loss_reg = 0
if (self.loss_weight['regularization'] > 0) and (self.training_phase == 'main'):
    loss_reg = torch.mean(torch.abs(Xi))
    loss_reg = self.loss_weight['regularization'] * loss_reg
```

Code 5.9: Loss calculation.

5.4 Training loop

The `training_step` method is called automatically for each training batch. Specifically, the forward pass is performed and the total training loss is calculated. Each of the training losses is logged and then averaged at the epoch level. The implementation of the training loop is reported in Code 5.10.

```
def training_step(self, batch, batch_idx):

    loss_rec, loss_x_dot, loss_z_dot, loss_reg = self.forward(batch)
    loss_total = loss_rec + loss_x_dot + loss_z_dot + loss_reg

    self.log('train_loss_rec',   loss_rec,   on_step=False, on_epoch=True, prog_bar=False)
    self.log('train_loss_x_dot', loss_x_dot, on_step=False, on_epoch=True, prog_bar=False)
    self.log('train_loss_z_dot', loss_z_dot, on_step=False, on_epoch=True, prog_bar=False)
    self.log('train_loss_reg',   loss_reg,   on_step=False, on_epoch=True, prog_bar=False)
    self.log('train_loss_total', loss_total, on_step=False, on_epoch=True, prog_bar=True)

    return loss_total
```

Code 5.10: Training loop.

5.5 Validation loop

The `validation_step` method is called automatically for each validation batch. Specifically, the forward pass is performed and the total validation loss is calculated. Each of the validation losses is logged and then averaged at the epoch level. The implementation of the validation loop is reported in Code 5.11.

```
def validation_step(self, batch, batch_idx):

    loss_rec, loss_x_dot, loss_z_dot, loss_reg = self.forward(batch)
    loss_total = loss_rec + loss_x_dot + loss_z_dot + loss_reg

    self.log('valid_loss_rec',   loss_rec,   on_step=False, on_epoch=True, prog_bar=False)
    self.log('valid_loss_x_dot', loss_x_dot, on_step=False, on_epoch=True, prog_bar=False)
    self.log('valid_loss_z_dot', loss_z_dot, on_step=False, on_epoch=True, prog_bar=False)
    self.log('valid_loss_reg',   loss_reg,   on_step=False, on_epoch=True, prog_bar=False)
    self.log('valid_loss_total', loss_total, on_step=False, on_epoch=True, prog_bar=True)

    return loss_total
```

Code 5.11: Validation loop.

5.6 Sequential thresholding

The SINDyAE sequential thresholding procedure is implemented as a pruning technique by applying the `torch.nn.utils.prune.custom_from_mask` method to the SINDy layer. This method prunes the tensor corresponding to a user-defined parameter of a specific PyTorch module by applying a pre-computed mask. In this case, the connections to be pruned are those with a weight below the initialization argument `threshold`, in the parameter named `weight` of the `self.sindy` layer. In practice, the implementation of the sequential thresholding consists of two parts.

The first part is embedded in the `on_train_epoch_end` method, called automatically at the end of each training epoch. In particular, if the SINDyAE is in the main training phase and the current epoch is a multiple of the thresholding frequency, pruning is performed. First, a mask named `mask1` is created. It keeps all and only the SINDy coefficients above `threshold`. However, if only `mask1` is used, it could happen that all the coefficients of an equation of the SINDy model are set to zero. This would lead to the impossibility of estimating the time derivative of the corresponding latent variable. Therefore, another mask named `mask2` is created. It keeps all the coefficients which would be zeroed by `mask1` and which would be the last ones left for a specific latent variable. Then, these two masks are put in logic OR and the resulting one, named `mask`, is used for pruning.

The second part is embedded in the `on_train_end` method, called automatically at the end of the whole training procedure. In particular, to make the pruning permanent, the `torch.nn.utils.prune.remove` method is used. Note that this does not undo the pruning, as if it never happened. It simply makes it permanent. The implementation of the sequential thresholding is reported in Code 5.12.

```
def on_train_epoch_end(self):
    if (self.training_phase == 'main') and (self.current_epoch % self.threshold_freq == 0) and
  ↪(self.current_epoch > 0):
        coeffs = self.sindy.weight.T
        mask1  = (torch.abs(coeffs) > self.threshold)
        mask2  = torch.logical_and((torch.abs(coeffs) > 0), (torch.count_nonzero(mask1, dim=0) ==
      ↪0))
        mask   = torch.logical_or(mask1, mask2).T
        torch.nn.utils.prune.custom_from_mask(module=self.sindy, name='weight', mask=mask)

def on_train_end(self):
    if self.training_phase == 'refinement':
        torch.nn.utils.prune.remove(module=self.sindy, name='weight')
```

Code 5.12: Sequential thresholding.

5.7 Optimizer and learning rate scheduler

The `configure_optimizers` method is called automatically to set the initialization argument `optimizer` as the optimization algorithm, with learning rate equal to the initialization argument `lr`. The `ReduceLROnPlateau` scheduler (monitoring the total validation loss and with patience equal to the initialization argument `lr_patience`) is also enabled, but only for refinement training. The implementation of the optimizer and learning rate scheduler is reported in Code 5.13.

```
def configure_optimizers(self):
    optimizer = self.optimizer(self.parameters(), lr=self.lr)
    patience  = self.lr_patience if self.training_phase == 'refinement' else self.max_epochs_main
    scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer=optimizer, patience=patience,
  ↪mode='min', verbose=True)
    return {'optimizer': optimizer, 'lr_scheduler': scheduler, 'monitor': 'valid_loss_total'}
```

Code 5.13: Optimizer and learning rate scheduler.

5.8 Early stopping

The `configure_callbacks` method is called automatically to activate the `EarlyStopping` callback with patience equal to the initialization argument named `es_patience`. This callback monitors the SINDy model complexity during the main training phase, while the total validation loss during the refinement training phase. The implementation of the early stopping is reported in Code 5.14.

```
def configure_callbacks(self):
    monitor        = 'complexity' if self.training_phase == 'main' else 'valid_loss_total'
    early_stopping = pl.callbacks.early_stopping.EarlyStopping(monitor=monitor,
  ↪patience=self.es_patience, mode='min')
    return [early_stopping]
```

Code 5.14: Early stopping.

5.9 Fitting

The `fit` method is called by the user to start training the SINDyAE. This method creates the following two instances of `pl.Trainer`:

- `trainer_main` for the main training phase;
- `trainer_refinement` for the refinement training phase.

The `pl.Trainer` handles all loop details automatically. Some examples include:

- automatically enabling/disabling gradients;
- running the training and validation dataloaders;
- calling the callbacks at the appropriate times;
- putting batches and computations on the correct devices.

First, the main training phase is performed by calling the `fit` method of `trainer_main` and its training history is saved to the attribute `self.history_main` as a NumPy array. Then, the refinement training phase is performed by calling the `fit` method of `trainer_refinement` and its training history is saved to the attribute `self.history_refinement` as a NumPy array. Eventually, the function `torch.save` is called to save the trained `SINDyAutoencoder` class to a `.pt` file. The implementation of the fitting is reported in Code 5.15.

```
def fit(self, train_dataloader, valid_dataloader=None, pt_name=None):

    # main trainer
    trainer_main = pl.Trainer(
                            accelerator='auto', devices='auto', strategy='auto',
                            deterministic=True, enable_checkpointing=False,
                            max_epochs=self.max_epochs_main,
                            logger=pl.loggers.CSVLogger(save_dir='logs',
                           ↪name=self.experiment_name),
                            callbacks=[ProgressBar(refresh_rate=len(train_dataloader)//3)])

    # refinement trainer
    trainer_refinement = pl.Trainer(
                            accelerator='auto', devices='auto', strategy='auto',
                            deterministic=True, enable_checkpointing=False,
                            max_epochs=self.max_epochs_refinement,
                            logger=pl.loggers.CSVLogger(save_dir='logs',
                           ↪name=self.experiment_name),
                            callbacks=[ProgressBar(refresh_rate=len(train_dataloader)//3)])

    # main training
    trainer_main.fit(model=self, train_dataloaders=train_dataloader,
  ↪val_dataloaders=valid_dataloader)
    self.history_main = pd.read_csv(self.logger.log_dir +
  ↪'/metrics.csv').drop(columns='step').groupby(['epoch'], as_index=False).max().to_numpy()

    # refinement training
    self.training_phase = 'refinement'
    trainer_refinement.fit(model=self, train_dataloaders=train_dataloader,
  ↪val_dataloaders=valid_dataloader)
    self.history_refinement = pd.read_csv(self.logger.log_dir +
  ↪'/metrics.csv').drop(columns='step').groupby(['epoch'], as_index=False).max().to_numpy()

    self.logged_metrics = list(trainer_main.logged_metrics.keys())

    if pt_name is not None:
        torch.save(self, pt_name)
```

Code 5.15: Fitting.

5.10 Training history

The `show_history` method is called by the user to get the full training history as a Pandas DataFrame. This DataFrame is created by vertically stacking the attributes `self.history_main` and `self.history_refinement`. The implementation of the training history is reported in Code 5.16.

```
def show_history(self):
    df = pd.DataFrame(np.vstack((self.history_main, self.history_refinement)),
  ↪columns=['epoch']+self.logged_metrics)
    df['epoch'] = df.index
    return df
```

Code 5.16: Training history.

5.11 SINDy coefficients

The `sindy_coeffs` method is called by the user to get the SINDy coefficients as a NumPy array. The corresponding implementation is reported in Code 5.17.

```
def sindy_coeffs(self):
    return self.sindy.weight.detach().cpu().numpy().T
```

Code 5.17: SINDy coefficients.

5.12 Synthetic data generation

The `simulate` method is called by the user to generate synthetic data by simulating the SINDy model learned by the SINDyAE forward in time. The user must pass the following arguments to the `simulate` method:

- `x0`: initial conditions from which to start the simulations;
- `n_samples`: number of samples to generate for each initial condition;
- `t_domain`: time domain in which to run the simulations.

First, a `ps.SINDy` module named `simulator` is instantiated for the sole purpose of exploiting its method `simulate` to solve the system of differential equations defined by the SINDy coefficients. Specifically, after setting it with the appropriate feature library and dimension of the latent space, the SINDy coefficients learned by the SINDyAE are loaded into it.

Second, the initial conditions `x0` are mapped to the latent space via the encoder network of the SINDyAE to get `z0`.

Then, the i-th row of `z0` is used as an initial condition to be passed to the `simulate` method of `simulator`, which generates `z`, i.e. the i-th trajectory of synthetic points in the latent space. Afterwards, `z` is mapped back to the measurement space via the decoder network of the SINDyAE to get `x`, i.e. the i-th trajectory of synthetic points in the measurement space. Finally, `x`, its time domain and its trajectory identifier are loaded into the NumPy array `data`.

After completion of all simulations, `data` is used to create the final synthetic dataset in the form of the pandas DataFrame `df`. Eventually, `df` is saved to a `.pkl` file. The implementation of the synthetic data generation is reported in Code 5.18.

```
def simulate(self, x0, n_samples, t_domain, pkl_name=None):

    n_init = len(x0)

    df_columns = ['trajectory', 't'] + [f'x{i+1}' for i in range(self.dim_input)]
    t          = np.linspace(start=t_domain[0], stop=t_domain[1], num=n_samples)
    data       = np.zeros((n_init*n_samples, len(df_columns)))

    simulator = ps.SINDy(optimizer=ps.optimizers.STLSQ(threshold=0),
  ↪feature_library=self.pysindy_library)
    simulator.fit(x=np.random.random((3,len(self.input_features))), t=np.linspace(0,1,3))
    simulator.optimizer.coef_ = self.sindy.weight.detach().cpu().numpy()
    simulator.optimizer.ind_  = (self.sindy.weight.detach().cpu().numpy() != 0)

    z0 = self.encoder(torch.as_tensor(x0, dtype=torch.float,
  ↪device=self.device)).detach().cpu().numpy()

    for i in trange(n_init):

        trajectory = np.full(n_samples, i+1)

        z = simulator.simulate(x0=z0[i], t=t)
        x = self.decoder(torch.as_tensor(z, dtype=torch.float,
      ↪device=self.device)).detach().cpu().numpy()

        data[i*n_samples:(i+1)*n_samples] = np.column_stack((trajectory, t, x))

    df = pd.DataFrame(data, columns=df_columns)
    df['trajectory'] = df['trajectory'].astype('int')

    if pkl_name is not None:
        print(f'\nSaving to "{pkl_name}"...', end=' ')
        df.to_pickle(pkl_name)
        print('DONE!')

    return df
```

Code 5.18: Synthetic data generation.

5.13 Model compilation

PyTorch 2.0 includes the `torch.compile` method which makes PyTorch code run faster by JIT-compiling it into optimized kernels, all while requiring minimal code changes. This speedup mainly comes from reducing Python overhead and GPU read/writes, and thus the observed speedup may vary depending on factors such as model architecture and batch size. For example, if the model architecture is simple and the amount of data is large, then the bottleneck would be GPU compute and the observed speedup may be less significant. The speedup results may also depend on the chosen `mode` argument that specifies what the compiler should be optimizing while compiling. It can be specified as one of the following:

- `default`: tries to compile the model efficiently without taking too long or using extra memory;
- `reduce-overhead`: reduces the framework overhead by a lot more, but costs a small amount of extra memory;
- `max-autotune`: compiles the model for a long time, trying to return the fastest code it can generate.

In general, different modes may need to be experimented with to maximize speedup.

The primary advantage of `torch.compile` over other existing PyTorch compiler solutions (e.g. TorchScript or FX Tracing) lies in its ability to handle arbitrary Python code with minimal changes to existing code. For example, arbitrary Python functions can be optimized simply by passing the callable to the `torch.compile` method.

PyTorch Lightning 2.0, which is compatible with PyTorch 2.0, supports model compilation out of the box. Again, a simple call to the `torch.compile` method is required to compile the `LightningModule`. This automatically compiles the model along with its training and validation steps.

The SINDyAE is compiled using the `max-autotune` mode to try to maximize performance. The code for the model compilation is reported in Code 5.19.

```
model = SINDyAutoencoder(...)
model = torch.compile(model, mode='max-autotune')
```

Code 5.19: Model compilation.

6

Models

In this chapter, the models explored in this thesis work are presented. Note that the model referred to as **M2** was implemented by the development team of Clearbox AI, which is the synthetic data company where the curricular internship was carried out.

6.1 Model **M1**: Generative SINDyAE

The model referred to as **M1** consists of the SINDyAE proposed by Champion et al. in [23], enriched with the synthetic data generation feature. It is implemented as shown in Chapter 5.

6.2 Model **M2**: VAE by Clearbox AI

The model referred to as **M2**, i.e. the Clearbox AI Enterprise Solution, is a commercial software developed to synthesize relational databases. The development team of Clearbox AI started working in 2022 on extending the applicability of its solution to time-series problems as well. At the time of the internship, the solution supports the generation of multivariate time-series with few known limitations, particularly regarding the number of variables.

The generation method behind the solution is the following. Given a dataset containing the multivariate time-series, the following steps are performed in order to generate synthetic data:

1. **Data preparation**: the time-series is decomposed into a set of wavelet coefficients using a Discrete Wavelet Transform. This first step includes a search for optimal wavelet family and truncation order.

2. **Generative model training**: the transformed dataset is used to train a VAE, specifically a Maximum Mean Discrepancy (MMD) VAE. This step includes a grid search to determine the optimal set of hyperparameters such as learning rate, number of epochs and batch size.

3. **Data generation and analysis**: the VAE is used to generate a synthetic clone of the original dataset. This cloned dataset is analyzed to determine the quality of the generated output against a set of metrics.

4. **Data reconstruction**: the synthetic clone, in the form of wavelet coefficients, is reconstructed to the original multivariate form using a wavelet reconstruction process.

The solution can be used, in its current form, for multivariate time-series with a limited number of variables. The choice of datasets to be used in this thesis work is driven by the need to test the solution on more complex data, possibly highlighting potential areas for improvement to better guide future development efforts. The selected datasets are presented in Chapter 7.

7

Datasets

The models explored in this thesis work are tested on two datasets generated by nonlinear dynamical systems. They are presented in this chapter.

7.1 Lorenz system

7.1.1 Description

The Lorenz system is a system of three ordinary differential equations developed by Edward Lorenz as a simplified mathematical model for atmospheric convection. It is notable for having chaotic solutions for certain parameter values and initial conditions. In particular, the *Lorenz attractor* is a set of chaotic solutions of the Lorenz system. The Lorenz equations are the following:

$$\begin{aligned} \dot{z_1} &= \sigma(z_2 - z_1) \\ \dot{z_2} &= z_1(\rho - z_3) - z_2 \\ \dot{z_3} &= z_1 z_2 - \beta z_3 \end{aligned} \tag{7.1}$$

They relate the properties of a two-dimensional fluid layer uniformly warmed from below and cooled from above. In particular, they describe the rate of change of three quantities with respect to time: z_1 is proportional to the rate of convection, z_2 to the horizontal temperature variation, and z_3 to the vertical temperature variation. The constants σ, ρ, β are system parameters proportional to the Prandtl number, Rayleigh number, and certain physical dimensions of the layer itself.

As proposed by the authors of [23], one way to create a high-dimensional dataset with dynamics defined by the Lorenz system is to choose six spatial modes $\mathbf{u}_1, \ldots, \mathbf{u}_6 \in \mathbb{R}^{128}$ and take:

$$\mathbf{x}(t) = \mathbf{u}_1 z_1(t) + \mathbf{u}_2 z_2(t) + \mathbf{u}_3 z_3(t) + \mathbf{u}_4 z_1(t)^3 + \mathbf{u}_5 z_2(t)^3 + \mathbf{u}_6 z_3(t)^3, \tag{7.2}$$

where the dynamics of $\mathbf{z}$ is specified by the Lorenz equations with standard parameter values, i.e. those used by Lorenz himself:

$$\sigma = 10, \quad \rho = 28, \quad \beta = 8/3. \tag{7.3}$$

The spatial modes $u_1, \ldots, u_6$ are chosen to be the first six Legendre polynomials defined at 128 grid points on the 1D spatial domain $[-1, 1]$. To generate the dataset, the system is simulated from multiple initial conditions. For each initial condition, the system is integrated forward in time from $t = 0$ to $t = 5$ with a spacing of $\Delta t = 0.02$. Thus, 250 samples are obtained for each trajectory. Initial conditions are randomly sampled from a uniform distribution over $z_1 \in [-36, 36]$, $z_2 \in [-48, 48]$, $z_3 \in [-16, 66]$. This results in the dataset reported in Table 7.1.

set	# initial conditions	# total samples
training	2048	512000
validation	20	5000
test	100	25000

Table 7.1: Lorenz dataset.

7.1.2 Simulation

The dataset concerning the Lorenz system is generated using the following user-defined Python functions: `lorenz`, `lorenz_df`, and `lorenz_df_high`. The libraries used to implement these functions are reported in Code 7.1.

```
from tqdm import trange
import numpy as np
import pandas as pd
from scipy.integrate import solve_ivp
from scipy.special import legendre
from sklearn.preprocessing import MaxAbsScaler, MinMaxScaler, StandardScaler
from pysindy.differentiation import FiniteDifference
```

Code 7.1: Libraries used by the functions that generate the Lorenz dataset.

First, the `lorenz` function determines the differential equations of the Lorenz system. Its implementation is reported in Code 7.2.

```
def lorenz(t, Z, beta=8/3, rho=28, sigma=10):

   z1, z2, z3 = Z
   z1_dot = sigma*(z2 - z1)
   z2_dot = z1*(rho - z3) - z2
   z3_dot = z1*z2 - beta*z3

   return z1_dot, z2_dot, z3_dot
```

Code 7.2: `lorenz` function.

Second, the `lorenz_df` function creates the three-dimensional Lorenz dataset according to the following steps:

1. the `np.random.uniform` function generates `n_init` initial conditions randomly sampled from a uniform distribution over `Z0_domain`;
2. for each initial condition:
   (a) the `solve_ivp` function numerically integrates the Lorenz system and returns the values of the solution at time points `t`;
   (b) the time derivatives of the solution values are either computed exactly using the `lorenz` function or numerically estimated using the `FiniteDifference` function;
   (c) the solution values and their derivatives, together with the time points and the trajectory identifier, are stored in the NumPy array `data`;
3. `data` is used to create the pandas DataFrame `df`;
4. `df` is scaled either using one of scikit-learn's scalers or dividing all its entries by 40 as done by the authors of [23];
5. `df` is saved to a `.pkl` file.

For this thesis work, the time derivatives are computed exactly and `df` is scaled using scikit-learn's `MaxAbsScaler` function. The implementation of the `lorenz_df` function is reported in Code 7.3.

Third, the `lorenz_df_high` function takes the three-dimensional Lorenz dataset as input to create the corresponding high-dimensional dataset `df_high` according to Eq. 7.2. Then, standard normal noise of strength equal to `noise_strength` is added to `df_high`. Lastly, `df_high` is saved to a `.pkl` file. For this thesis work, a value of `noise_strength` equal to 10^{-6} is used. The implementation of the `lorenz_df_high` function is reported in Code 7.4.

```
def lorenz_df(n_init=2048, n_samples=250, t_domain=(0,5), Z0=None, Z0_domain={'z1':(-36,36),
↪'z2':(-48,48), 'z3':(-16,66)}, beta=8/3, rho=28, sigma=10, exact_derivatives=True,
↪scaler_name='maxabs', pkl_name=None):

    df_columns = ['trajectory', 't', 'z1', 'z2', 'z3', 'dz1', 'dz2', 'dz3']
    t          = np.linspace(start=t_domain[0], stop=t_domain[1], num=n_samples)
    data       = np.zeros((n_init*n_samples, len(df_columns)))

    if Z0 is None:
        Z0 = np.random.uniform(low=[Z0_domain[f'z{i}'][0] for i in range(1,4)],
      ↪high=[Z0_domain[f'z{i}'][1] for i in range(1,4)], size=(n_init,3))

    for i in trange(n_init):

        trajectory = np.full(n_samples, i+1)
        z1, z2, z3 = solve_ivp(fun=lorenz, t_span=t_domain, y0=Z0[i], method='LSODA', t_eval=t,
      ↪args=(beta, rho, sigma), rtol=1e-12, atol=1e-12).y

        if exact_derivatives:
            dz1, dz2, dz3 = lorenz(t, (z1, z2, z3))
        else:
            dz1, dz2, dz3 = [FiniteDifference()._differentiate(zi, t) for zi in (z1, z2, z3)]

        data[i*n_samples:(i+1)*n_samples] = np.column_stack((trajectory, t, z1, z2, z3, dz1, dz2,
      ↪dz3))

    df = pd.DataFrame(data, columns=df_columns)
    df['trajectory'] = df['trajectory'].astype('int')

    if scaler_name is not None:

        if scaler_name == 'maxabs':
            scaler           = MaxAbsScaler()
            df.iloc[:,2:5]   = scaler.fit_transform(df.iloc[:,2:5].to_numpy())
            df.iloc[:,5:]    = scaler.transform(df.iloc[:,5:].to_numpy())

        elif scaler_name == 'minmax':
            scaler           = MinMaxScaler()
            df.iloc[:,2:5]   = scaler.fit_transform(df.iloc[:,2:5].to_numpy())
            scaler.min_      = np.zeros_like(scaler.min_)
            df.iloc[:,5:]    = scaler.transform(df.iloc[:,5:].to_numpy())

        elif scaler_name == 'standard':
            scaler           = StandardScaler()
            df.iloc[:,2:5]   = scaler.fit_transform(df.iloc[:,2:5].to_numpy())
            scaler.with_mean = False
            df.iloc[:,5:]    = scaler.transform(df.iloc[:,5:].to_numpy())

        elif scaler_name == 'paper':
            df.iloc[:,2:]   /= 40

        else:
            print(f'\nScaler "{scaler_name}" not supported!')

    if pkl_name is not None:
        print(f'\nSaving to "{pkl_name}"...', end=' ')
        df.to_pickle(pkl_name)
        print('DONE!')

    return df
```

Code 7.3: `lorenz_df` function.

```python
def lorenz_df_high(df, modes_n=6, modes_dim=128, modes_domain=(-1,1), nonlinear=True,
↪noise_strength=1e-6, pkl_name=None):

    df_high = df.copy(deep=True)

    modes = [legendre(i)(np.linspace(modes_domain[0], modes_domain[1], modes_dim)) for i in
   ↪range(modes_n)]
    for i in range(modes_n):
        df_high[f'u{i+1}'] = len(df_high)*[modes[i]]

    # X
    df_high['X'] = 0
    print('X | linear features')
    for i in trange(1, 4):
        df_high['X'] += df_high[f'u{i}']*df_high[f'z{i}']
    if nonlinear:
        print('\nX | nonlinear features')
        for i in trange(1, 4):
            df_high['X'] += df_high[f'u{i+3}']*(df_high[f'z{i}']**3)

    # dX
    df_high['dX'] = 0
    print('\ndX | linear features')
    for i in trange(1, 4):
        df_high['dX'] += df_high[f'u{i}']*df_high[f'dz{i}']
    if nonlinear:
        print('\ndX | nonlinear features')
        for i in trange(1, 4):
            df_high['dX'] += 3*df_high[f'u{i+3}']*(df_high[f'z{i}']**2)*df_high[f'dz{i}']

    trajectory = df_high['trajectory'].tolist()
    t = df_high['t'].tolist()

    df_high = pd.DataFrame(
                        np.concatenate((df_high['X'].tolist(), df_high['dX'].tolist()), axis=1),
                        columns=[f'x{i+1}' for i in range(modes_dim)]+[f'dx{i+1}' for i in
                       ↪range(modes_dim)]
                        )

    if noise_strength > 0:
        df_high += noise_strength*np.random.standard_normal(df_high.shape)

    df_high.insert(0, 't', t)
    df_high.insert(0, 'trajectory', trajectory)

    if pkl_name is not None:
        print(f'\nSaving to "{pkl_name}"...', end=' ')
        df_high.to_pickle(pkl_name)
        print('DONE!')

    return df_high
```

Code 7.4: `lorenz_df_high` function.

7.2 F-16 aircraft

7.2.1 Description

The experimental data [27] were acquired on a full-scale F-16 aircraft (see Fig. 7.1) on the occasion of the Siemens LMS Ground Vibration Testing Master Class, held in September 2014 at the Saffraanberg military basis, Sint-Truiden, Belgium.

Figure 7.1: Complete structure of the F-16 aircraft.

During the test campaign, two dummy payloads were mounted at the wing tips to simulate the mass and inertia properties of real devices typically equipping an F-16 in flight (see Fig. 7.2a). The aircraft structure was equipped with 145 acceleration sensors. A shaker was attached underneath the right wing to apply input signals (see Fig. 7.2b). The dominant source of nonlinearity in the structural dynamics was expected to originate from the mounting interfaces of the two payloads. These interfaces consist of T-shaped connecting elements on the payload side, slid through a rail attached to the wing side (see Fig. 7.2c). A preliminary investigation showed that the back connection of the right-wing-to-payload interface was the predominant source of nonlinear distortions in the aircraft dynamics.

(a) (b) (c)

Figure 7.2: F-16 instrumentation. (a) Dummy payload mounted at the right wing tip; (b) shaker attached underneath the right wing; (c) back connection of the right-wing-to-payload mounting interface.

Measurements were acquired at a sampling frequency of 400 Hz. Two distinct input signals are made available:

1. voltage measured at the output of the signal generator amplifier, acting as a reference input;
2. actual force provided by the shaker and measured by a impedance head at the excitation location.

Three acceleration signals are provided as output quantities. They were measured:

1. at the excitation location;
2. on the right wing next to the nonlinear interface of interest;
3. on the payload next to the nonlinear interface of interest.

The outputs are listed in this order in the data matrices.

For this thesis work, the datasets concerning sine-sweep excitations with a linear, negative rate of 0.05 Hz/s (sweep down) are used, i.e. those corresponding to the data files named `F16Data_SineSw_Level#.csv`. The covered input frequency range is 15 to 2 Hz. Seven different levels of excitation are provided as benchmark data. The lowest level at 4.8 N input amplitude can be considered as a linear dataset. Three higher excitation levels are given to serve as estimation data in nonlinear regimes of vibration, i.e. datasets # 3, 5 and 7 corresponding to 28.8 N, 67.0 N and 95.6 N, respectively. Datasets # 2, 4 and 6 corresponding to 19.2 N, 57.6 N and 86.0 N, respectively, are originally intended to be used for testing the models estimated using datasets # 3, 5 and 7, respectively.

The F-16 benchmark is associated with three major nonlinear system identification challenges. First, the order of the system is reasonably high. In the 2 to 15 Hz band, the F-16 has about 10 resonance modes:

- few modes < 5 Hz corresponding to rigid-body motions of the structure;
- flexible mode ≈ 5.2 Hz corresponding to wing bending deformations;
- wing torsion mode ≈ 7.3 Hz involving the strongest nonlinear distortions.

Second, the mounting interface of interest is expected to feature nonlinearities in stiffness and damping, due to clearance and friction, respectively. Third, clearance and friction may lead to hard nonlinearities, and hence may not be appropriately modeled using smooth basis functions.

7.2.2 Preprocessing

All datasets corresponding to data files named `F16Data_SineSw_Level#.csv` are preprocessed and then concatenated to obtain a single dataset. The libraries used to perform these operations are reported in Code 7.5.

```
import os
import numpy as np
import pandas as pd
from sklearn.preprocessing import MaxAbsScaler
from pysindy.differentiation import SmoothedFiniteDifference
```

Code 7.5: Libraries used to preprocess and concatenate the F-16 datasets.

First, the bottom rows of these datasets are all zeros and correspond to the phase in which the F-16 aircraft is no longer excited by the shaker. These rows are clearly superfluous and should therefore be discarded. To this end, the number of top rows `n_samples` corresponding to the 15 to 2 Hz input frequency range is computed according to the sampling frequency `sampling_freq` and the sweep rate `sweep_rate`. Then, `n_samples` and `sampling_freq` are used to compute the time points `t`. The code for computing `n_samples` and `t` is reported in Code 7.6.

```
sampling_freq    = 400    # Hz
sweep_rate       = -0.05  # Hz/s
input_freq_range = (15,2) # Hz

dt        = 1/sampling_freq
n_samples = int(sampling_freq*(input_freq_range[1]-input_freq_range[0])/sweep_rate)
t_domain  = (0, (n_samples-1)*dt)
t         = np.linspace(t_domain[0], t_domain[1], num=n_samples)
```

Code 7.6: Computation of `n_samples` and `t`.

Second, each of these datasets is read and then preprocessed according to the following steps:

1. the i-th dataset is read as a pandas DataFrame `df_i`;
2. only the first `n_samples` rows of `df_i` are retained;
3. the time derivatives of the columns of `df_i` are numerically estimated using the `SmoothedFiniteDifference` function, i.e. `FiniteDifference` preceded by a smoother (default: Savitzky-Golay filter [1]) to mitigate noise effects;
4. `df_i` is split by rows into multiple DataFrames `df_ij` of length `len_trajectory` equal to 250;
5. `df_ij` is appended to the list `df`.

After this loop, the list `df` is shuffled by calling the `np.random.shuffle` function and then it is transformed into a single DataFrame.

Third, the DataFrame `df` is scaled using scikit-learn's `MaxAbsScaler` function. Eventually, `df` is saved to a `.pkl` file. The code for preprocessing and concatenating the F-16 datasets is reported in Code 7.7.

```
csv_names = [csv_name for csv_name in os.listdir('data_f16') if 'F16Data_SineSw' in csv_name]
len_trajectory = 250
df = []

for csv_name in csv_names:

    df_i = pd.read_csv(f'data_f16/{csv_name}')
    df_i = df_i[['Force', 'Voltage', 'Acceleration1', 'Acceleration2', 'Acceleration3']]
    df_i = df_i.iloc[:n_samples,:]

    df_i.columns = [f'x{i+1}' for i in range(len(df_i.columns))]

    df_i['d'+df_i.columns] = SmoothedFiniteDifference()._differentiate(x=df_i.to_numpy(), t=t)

    df_i.insert(0, 't', t)

    for j in range(0, len(df_i), len_trajectory):
        df_ij = df_i.iloc[j:j+len_trajectory,:]
        df.append(df_ij)

np.random.shuffle(df)
for i in range(len(df)):
    df[i].insert(0, 'trajectory', i+1)

df = pd.concat(df, ignore_index=True)

scaler = MaxAbsScaler()

columns_X  = df.columns[df.columns.str.startswith('x')]
columns_dX = df.columns[df.columns.str.startswith('dx')]

scaler.fit(df[columns_X].to_numpy())

df[columns_X]  = scaler.transform(df[columns_X].to_numpy())
df[columns_dX] = scaler.transform(df[columns_dX].to_numpy())

df.to_pickle('data/f16_original.pkl')
```

Code 7.7: Preprocessing and concatenation of the F-16 datasets.

This results in the dataset reported in Table 7.2.

set	# initial conditions	# total samples
training	2792	698000
validation	20	5000
test	100	25000

Table 7.2: F-16 dataset.

8

Results

8.1 Introduction

This section describes the procedures implemented to assess the performance and analyze the results of the models presented in Chapter 6.

8.1.1 Model training history and SINDy coefficients

In the case of the **M1** model, the training history is reported. Specifically, the trend over the epochs of the following quantities is shown:

- training losses;
- validation losses;
- total losses;
- complexity of the SINDy model.

Moreover, a heatmap of the coefficients learned by the SINDy model is shown.

8.1.2 Synthetic data generation

Each trained model is used to generate a number of synthetic trajectories equal to the number of test trajectories, that is, 100. This choice avoids class imbalance for the time-series classification problem presented in the next subsection. Furthermore, each synthetic trajectory has a length equal to the length of the original trajectories, that is, 250.

In the case of the **M1** model, the synthetic initial conditions are randomly sampled from the multivariate normal distribution inferred from the initial test conditions. Then, starting from each synthetic initial condition, the latent dynamical model learned by the SINDyAE is simulated forward in time from $t = t^*$ to $t = t^* + \Delta t$, where Δt is the time duration of the original trajectories and t^* depends on the dataset in use. Specifically:

- in the case of the Lorenz dataset, t^* is the initial time point of the original trajectories, i.e. $t^* = 0$;
- in the case of the F-16 dataset, t^* is a time point uniformly sampled from the time domain of the test set.

8.1.3 Classification of real and synthetic time-series

A time-series DL classifier is trained to discriminate between original and synthetic trajectories. As stated in [31], *InceptionTime* [26] is currently one of the best DL models for time-series classification. It is an ensemble of deep Convolutional Neural Network (CNN) models, inspired by the Inception-v4 [20] architecture. Therefore, it is chosen for the purposes of this thesis work. Specifically, the `InceptionTimeClassifier` model included in the sktime library is used.

First, the datasets for training and testing the `InceptionTimeClassifier` model are obtained according to the following steps:

1. the original test dataset is transformed into a 3D NumPy array `original_X` of shape [# trajectories, # dimensions, trajectory length];
2. the synthetic dataset is transformed into a 3D NumPy array `synthetic_X` of shape [# trajectories, # dimensions, trajectory length];
3. `original_X` is paired with a 1D NumPy array `original_y` of binary labels all equal to 0;
4. `synthetic_X` is paired with a 1D NumPy array `synthetic_y` of binary labels all equal to 1;
5. `original_X` and `synthetic_X` are vertically stacked to get `X`;
6. `original_y` and `synthetic_y` are concatenated to get `y`;
7. `X` and `y` are split into random train and test subsets `X_train`, `X_test`, `y_train`, and `y_test` (test size: 30% of the length of `X`).

Then, the `InceptionTimeClassifier` model is trained on `X_train` and `y_train` using the following settings:

- batch size: length of `X`, i.e. 200;
- number of epochs: 10^3;
- learning rate: 10^{-3};
- loss: categorical cross-entropy;
- optimizer: Adam;
- scheduler: `ReduceLROnPlateau` (patience: 50, monitor: loss, threshold: 10^{-4});
- callbacks: early stopping (patience: 100, monitor: loss, threshold: 10^{-4}).

Consequently, the trend over the epochs of the training loss is shown.

Finally, the accuracy of the trained `InceptionTimeClassifier` model is tested on `X_test` and `y_test`. The accuracy, i.e. the fraction of correct predictions, is computed using the `accuracy_score` function included in the scikit-learn library.

8.1.4 Distance between real and synthetic time-series

The `pairwise_distance` function included in the sktime library is used to compute the 2D pairwise distance matrix between the original and synthetic trajectories, which is then represented as a heatmap. The `pairwise_distance` function natively supports the following distance metrics for 3D arrays like `original_X` and `synthetic_X`:

- euclidean;
- squared;
- dynamic time warping (DTW) [2];
- derivative dynamic time warping (DDTW) [7];
- weighted dynamic time warping (WDTW) [12];
- weighted derivative dynamic time warping (WDDTW) [12];
- longest common subsequence (LCSS) [34];
- Edit distance for real sequences (EDR) [9];
- Edit distance with real penalty (ERP) [8].

Such heatmaps are qualitatively interpreted as a measurement of the novelty introduced by the synthetic trajectories compared to the original trajectories.

8.2 Lorenz system

8.2.1 Model **M1**

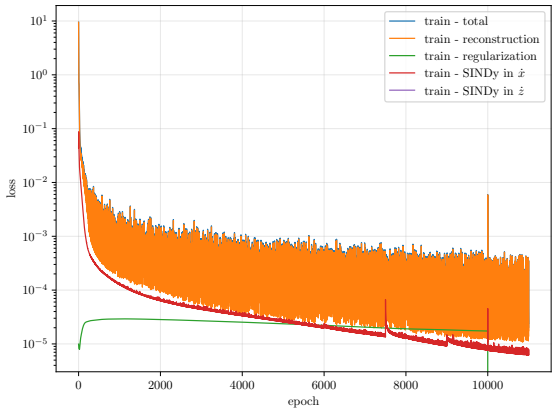
The training details of the experiments conducted with the **M1** model on the Lorenz dataset are reported in Table 8.1. The model training histories and SINDy coefficients are reported in Figs. 8.1 to 8.6. Regarding the training of the `InceptionTimeClassifier` models, the trends over the epochs of the training loss are reported in Fig. 8.7. The accuracy achieved by the `InceptionTimeClassifier` models in discriminating between original and synthetic trajectories is reported in Table 8.2. The heatmaps of the pairwise distance matrices between the original and synthetic trajectories are reported in Figs. 8.8 to 8.13.

experiment version	**v1**	**v2**	**v3**	**v4**	**v5**	**v6**
seed	5	3	9	3	7	10
input dimension	128	128	128	128	128	128
latent dimension	3	3	3	3	3	3
encoder layer widths	64,32	64,32	64,32	64,32	64,32	64,32,16,8
decoder layer widths	32,64	32,64	32,64	32,64	32,64	8,16,32,64
batch size	8000	8000	8000	8000	8000	8000
activation function	Sigmoid	Sigmoid	ReLU	ReLU	ReLU	ReLU
SINDy loss in $\dot{x}$ – weight	10^{-4}	10^{-4}	10^{-4}	10^{-4}	10^{-4}	10^{-4}
SINDy loss in $\dot{z}$ – weight	0	10^{-6}	0	10^{-6}	10^{-5}	10^{-6}
regularization loss – weight	10^{-5}	10^{-4}	10^{-4}	10^{-4}	10^{-3}	10^{-3}
main training – epochs	10^{4}	10^{4}	10^{4}	10^{4}	10^{4}	10^{4}
refinement training – epochs	10^{3}	10^{3}	10^{3}	10^{3}	10^{3}	10^{3}
optimizer	Adam	Adam	Adam	Adam	Adam	Adam
learning rate	10^{-3}	10^{-4}	10^{-3}	10^{-4}	10^{-4}	10^{-4}
learning rate patience	10^{4}	1000	500	1000	1000	1000
early stopping patience	10^{4}	3000	1500	3000	3000	3000
thresholding – threshold	0.1	0.1	0.1	0.1	0.1	0.1
thresholding – frequency	500	500	500	500	500	500
polynomial features – degree	3	3	3	3	3	3
include bias feature	yes	yes	yes	yes	yes	yes
include sine features	no	no	no	no	no	no
include cosine features	no	no	no	no	no	no
# trigonometric frequencies	0	0	0	0	0	0

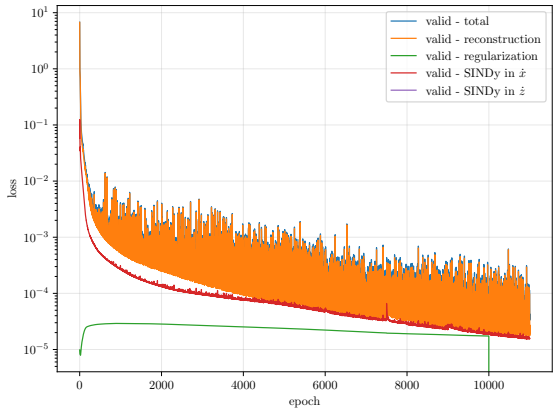
Table 8.1: [Lorenz **M1**] Training details.

experiment version	**v1**	**v2**	**v3**	**v4**	**v5**	**v6**
training accuracy (%)	100	100	96.43	97.86	76.43	71.43
test accuracy (%)	38.33	98.33	78.33	91.67	73.33	73.33

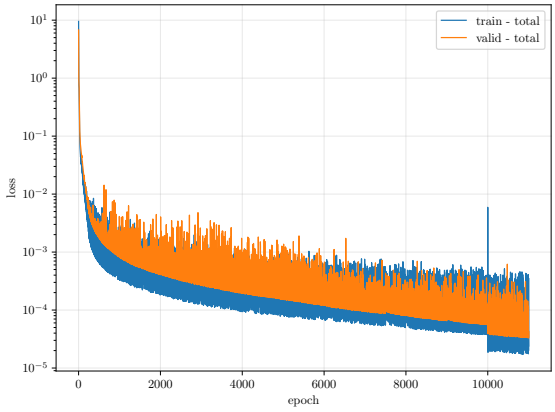
Table 8.2: [Lorenz **M1**] Accuracy of the `InceptionTimeClassifier` models.



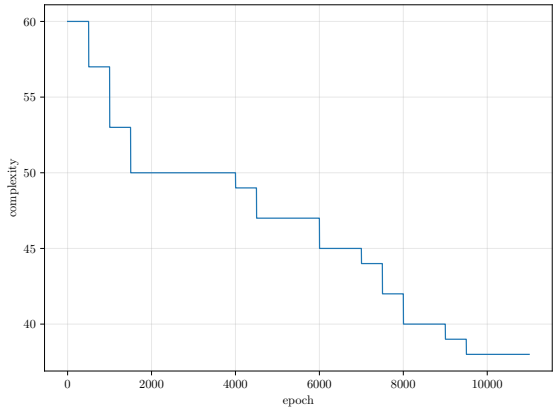
(a) Training losses.



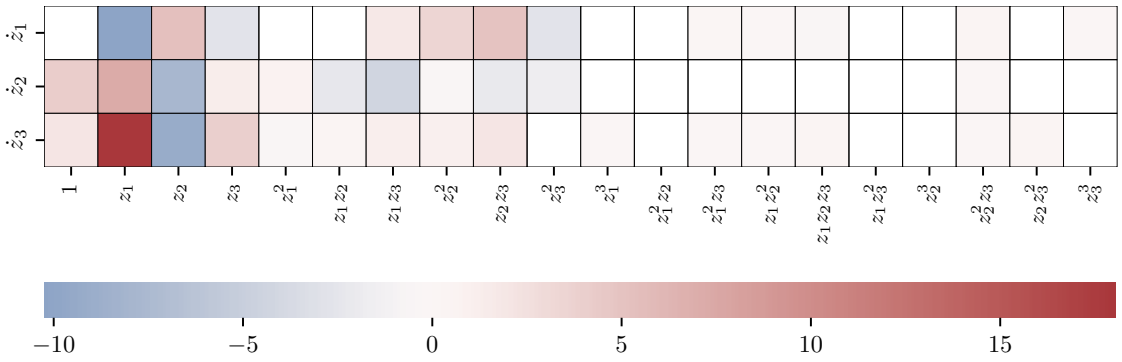
(b) Validation losses.



(c) Total losses.

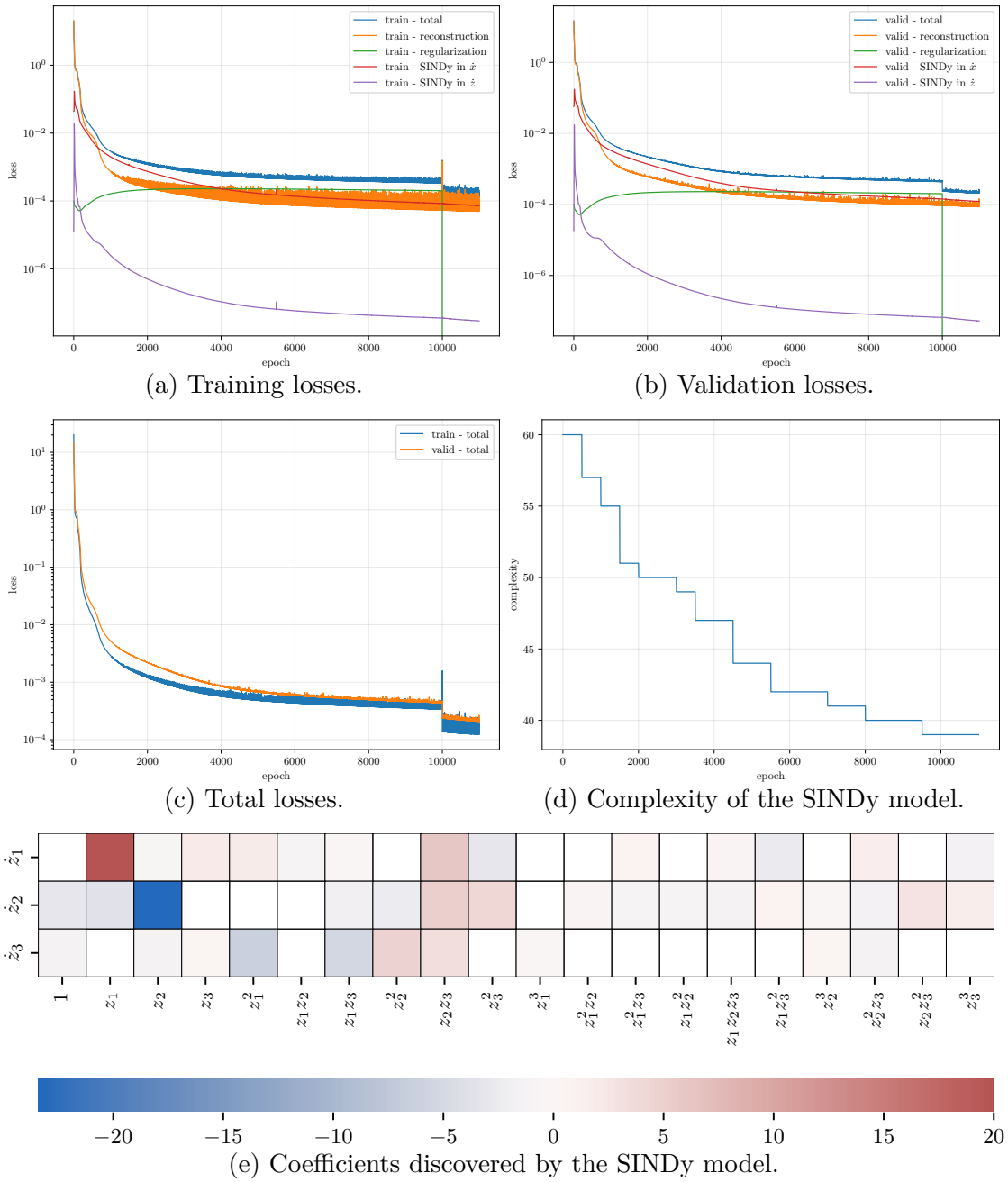


(d) Complexity of the SINDy model.



(e) Coefficients discovered by the SINDy model.

Figure 8.1: [Lorenz **M1v1**] Model training history and SINDy coefficients.



(a) Training losses.

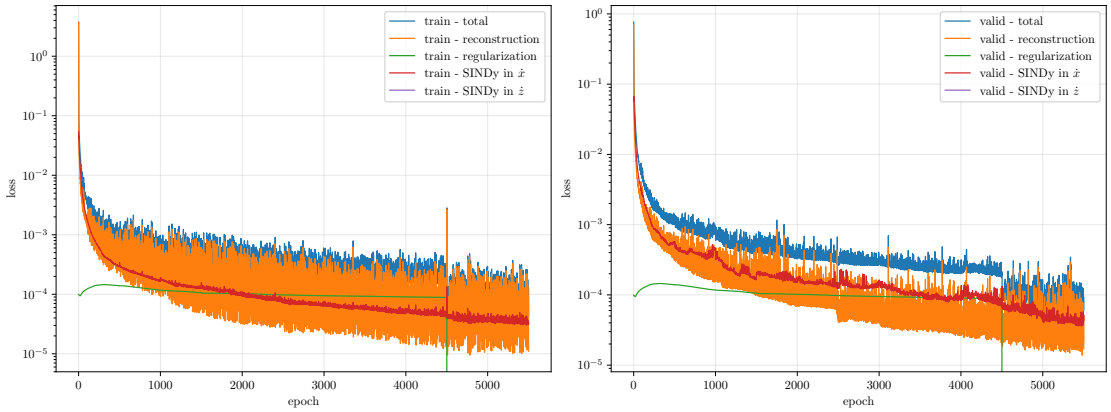
(b) Validation losses.

(c) Total losses.

(d) Complexity of the SINDy model.

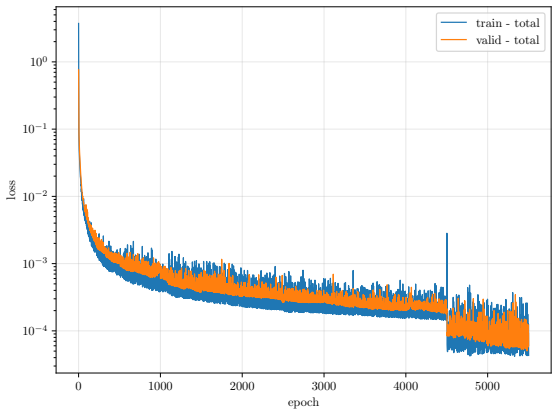
(e) Coefficients discovered by the SINDy model.

Figure 8.2: [Lorenz **M1v2**] Model training history and SINDy coefficients.

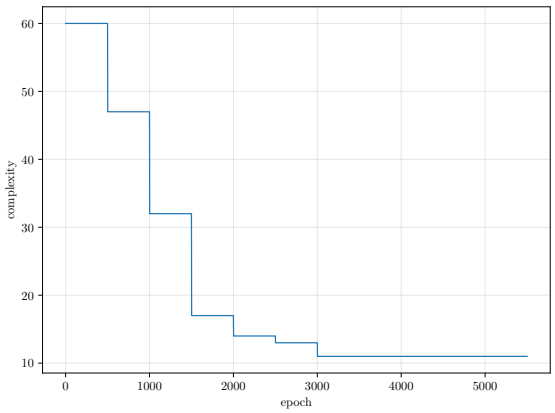


(a) Training losses.

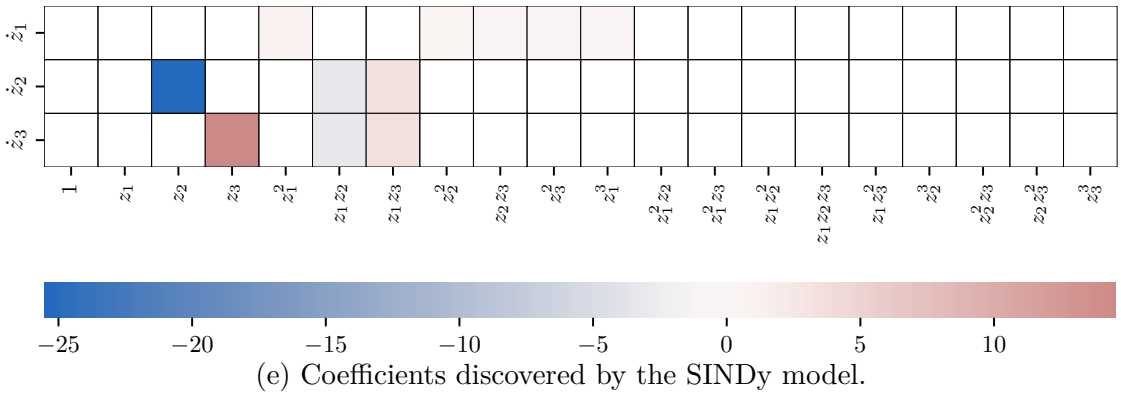
(b) Validation losses.



(c) Total losses.

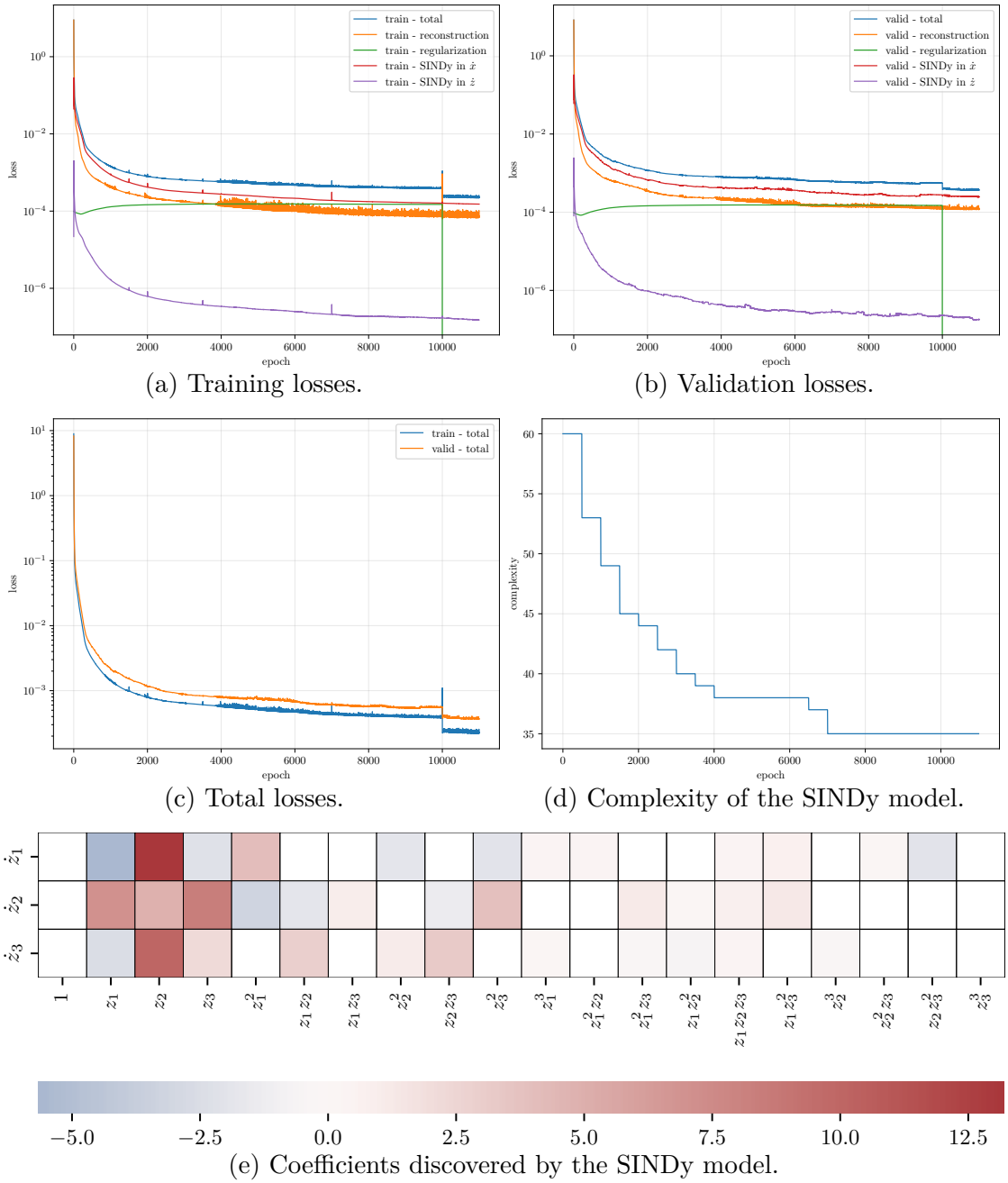


(d) Complexity of the SINDy model.



(e) Coefficients discovered by the SINDy model.

Figure 8.3: [Lorenz **M1v3**] Model training history and SINDy coefficients.



(a) Training losses.

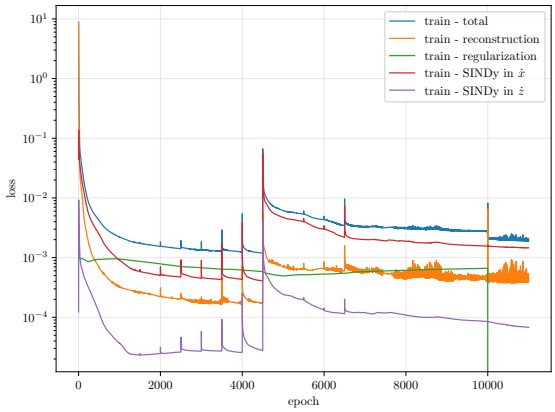
(b) Validation losses.

(c) Total losses.

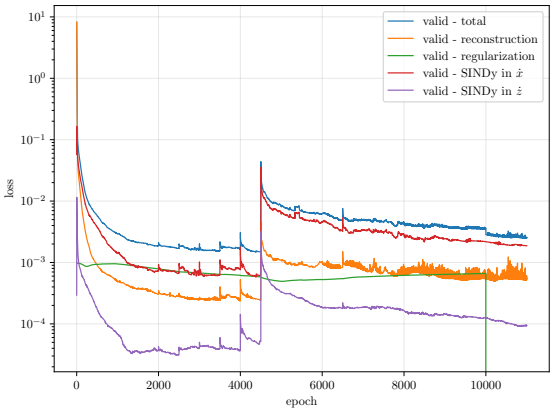
(d) Complexity of the SINDy model.

(e) Coefficients discovered by the SINDy model.

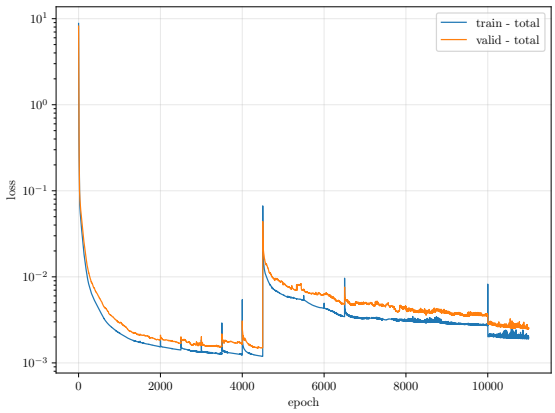
Figure 8.4: [Lorenz **M1v4**] Model training history and SINDy coefficients.



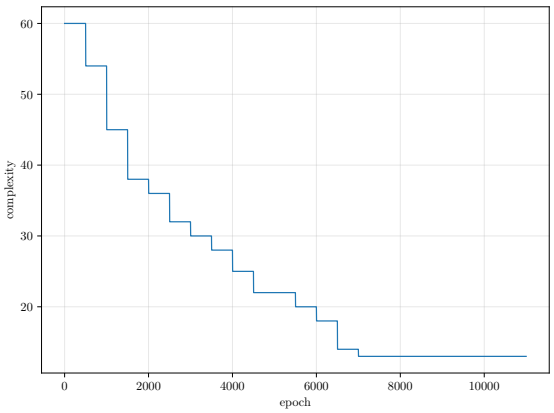
(a) Training losses.



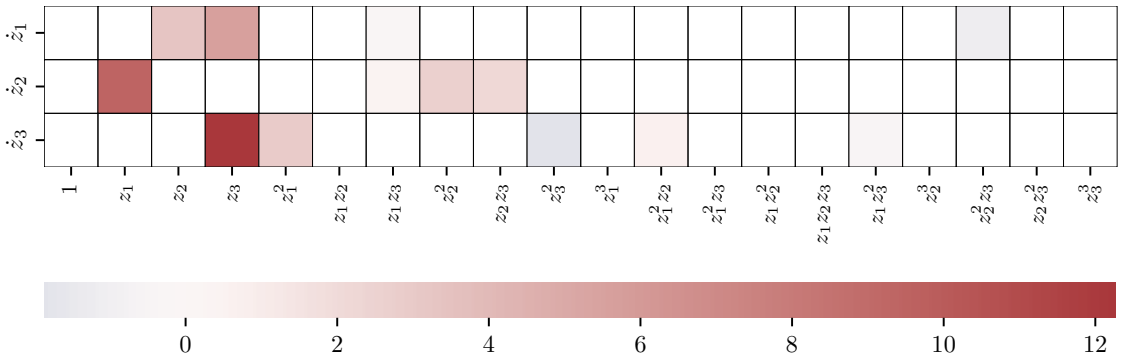
(b) Validation losses.



(c) Total losses.

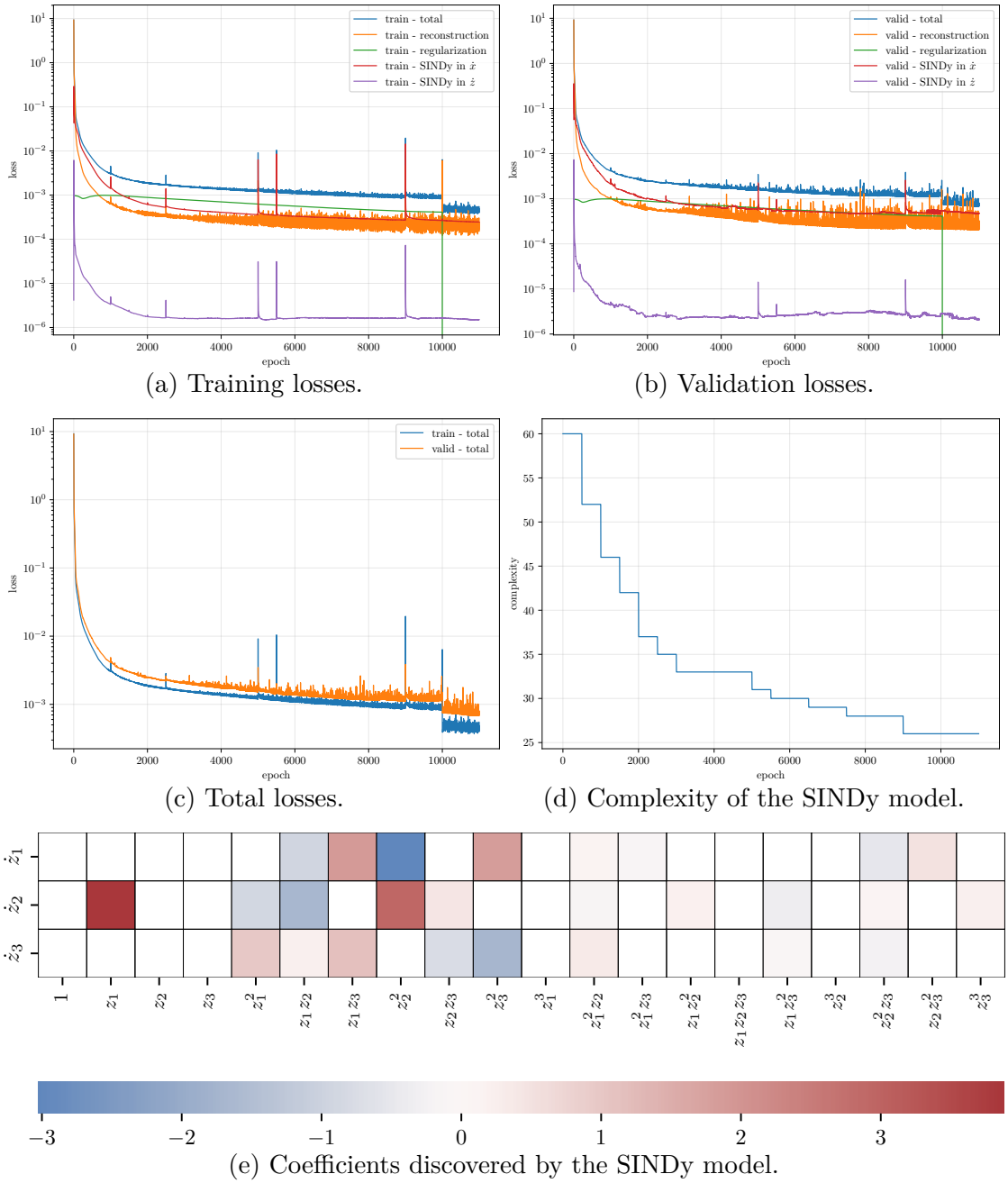


(d) Complexity of the SINDy model.



(e) Coefficients discovered by the SINDy model.

Figure 8.5: [Lorenz **M1v5**] Model training history and SINDy coefficients.



(a) Training losses.

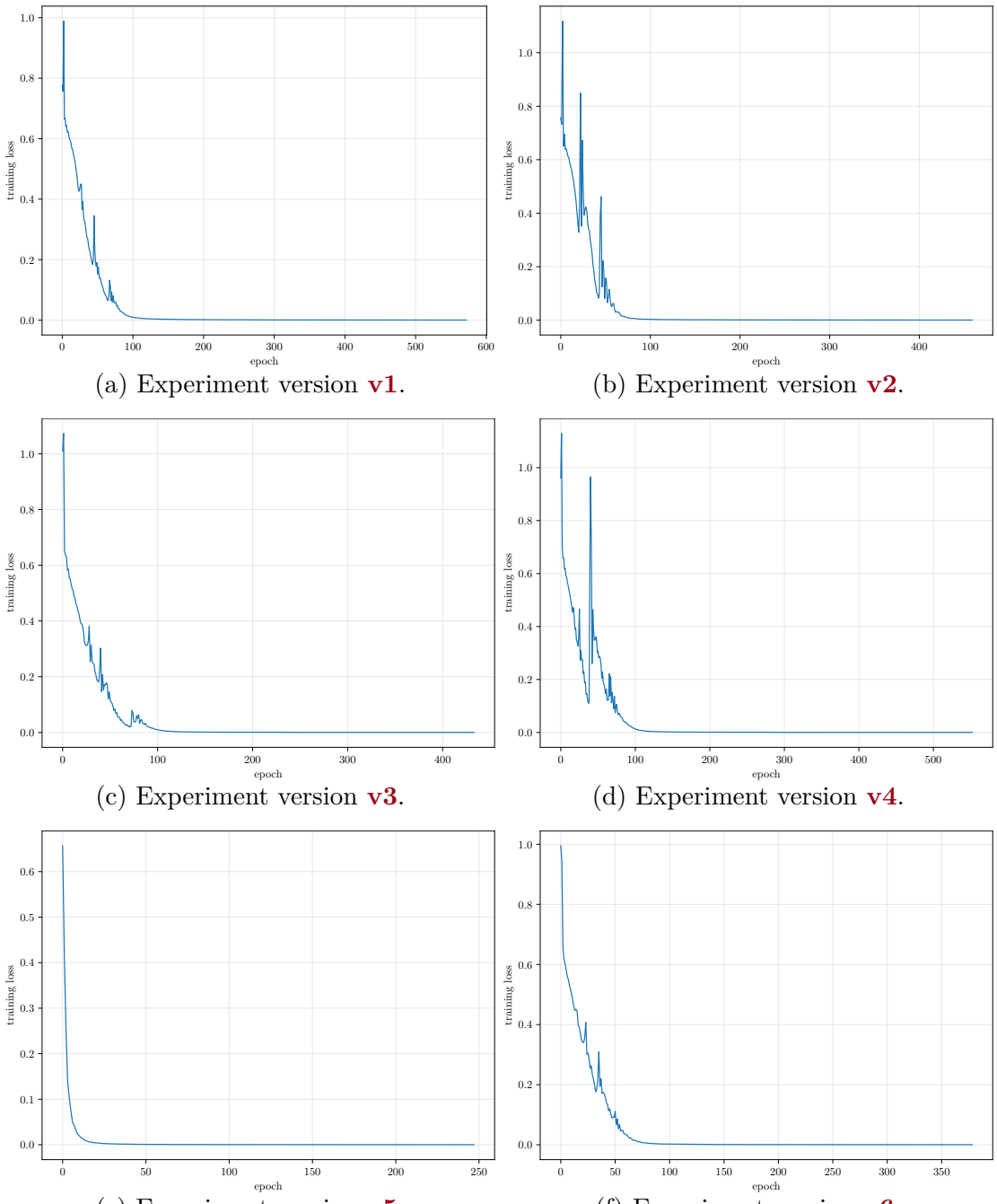
(b) Validation losses.

(c) Total losses.

(d) Complexity of the SINDy model.

(e) Coefficients discovered by the SINDy model.

Figure 8.6: [Lorenz **M1v6**] Model training history and SINDy coefficients.



(a) Experiment version **v1**.

(b) Experiment version **v2**.

(c) Experiment version **v3**.

(d) Experiment version **v4**.

(e) Experiment version **v5**.

(f) Experiment version **v6**.

Figure 8.7: [Lorenz **M1**] Training loss of the `InceptionTimeClassifier` models.

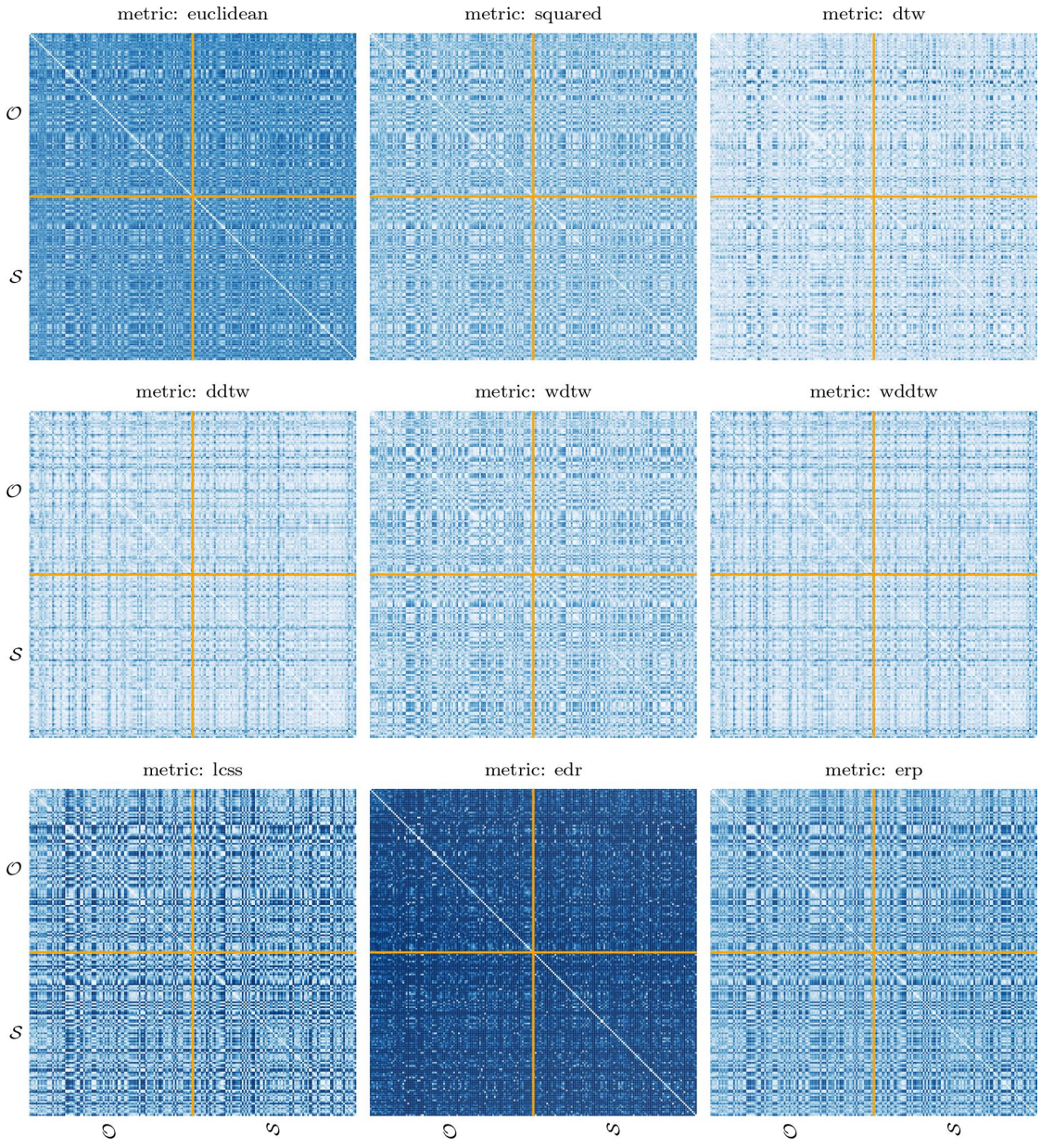


Figure 8.8: [Lorenz **M1v1**] Pairwise distance matrices between the original $\mathcal{O}$ and synthetic $\mathcal{S}$ trajectories.

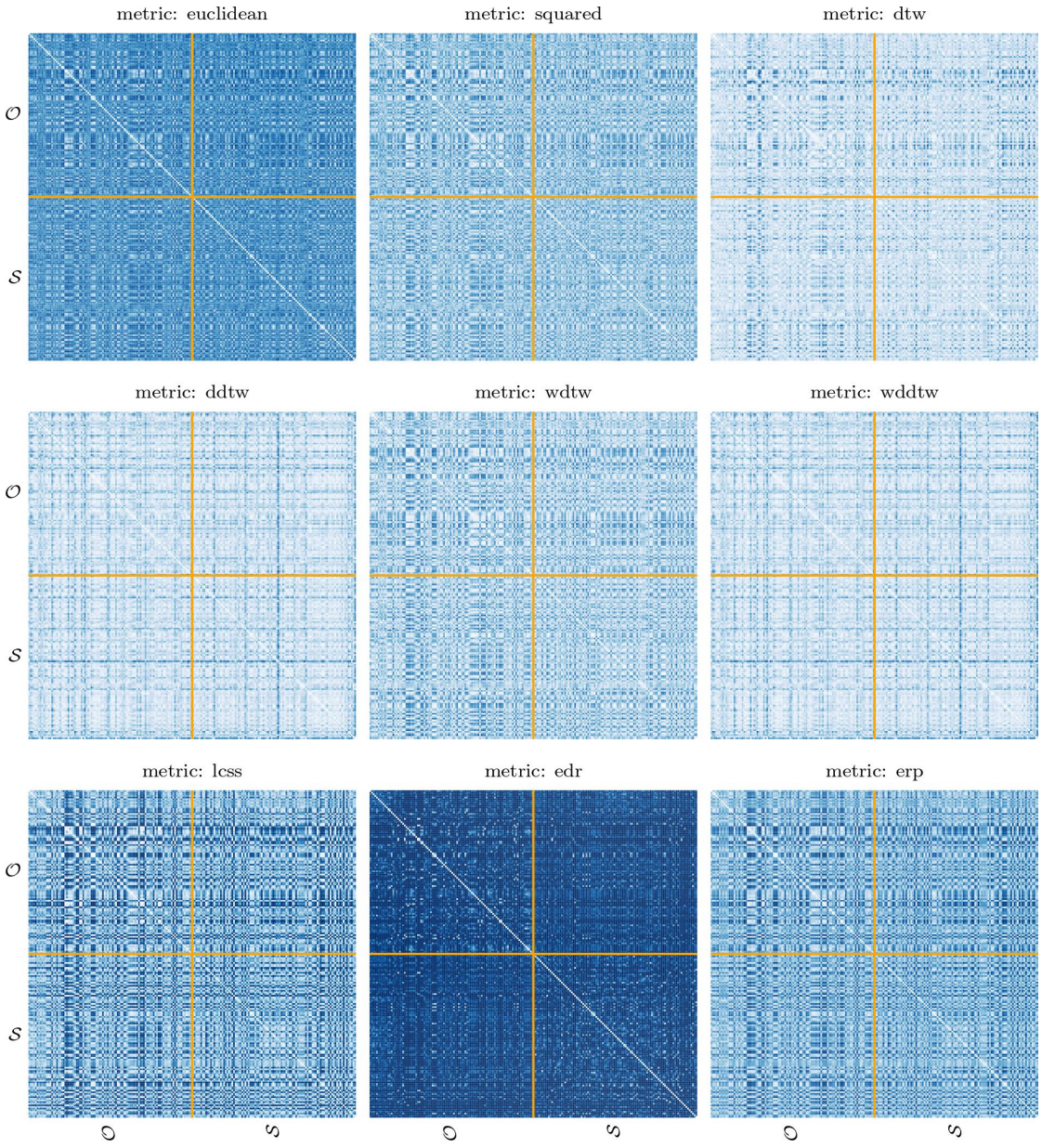


Figure 8.9: [Lorenz **M1v2**] Pairwise distance matrices between the original $\mathcal{O}$ and synthetic $\mathcal{S}$ trajectories.

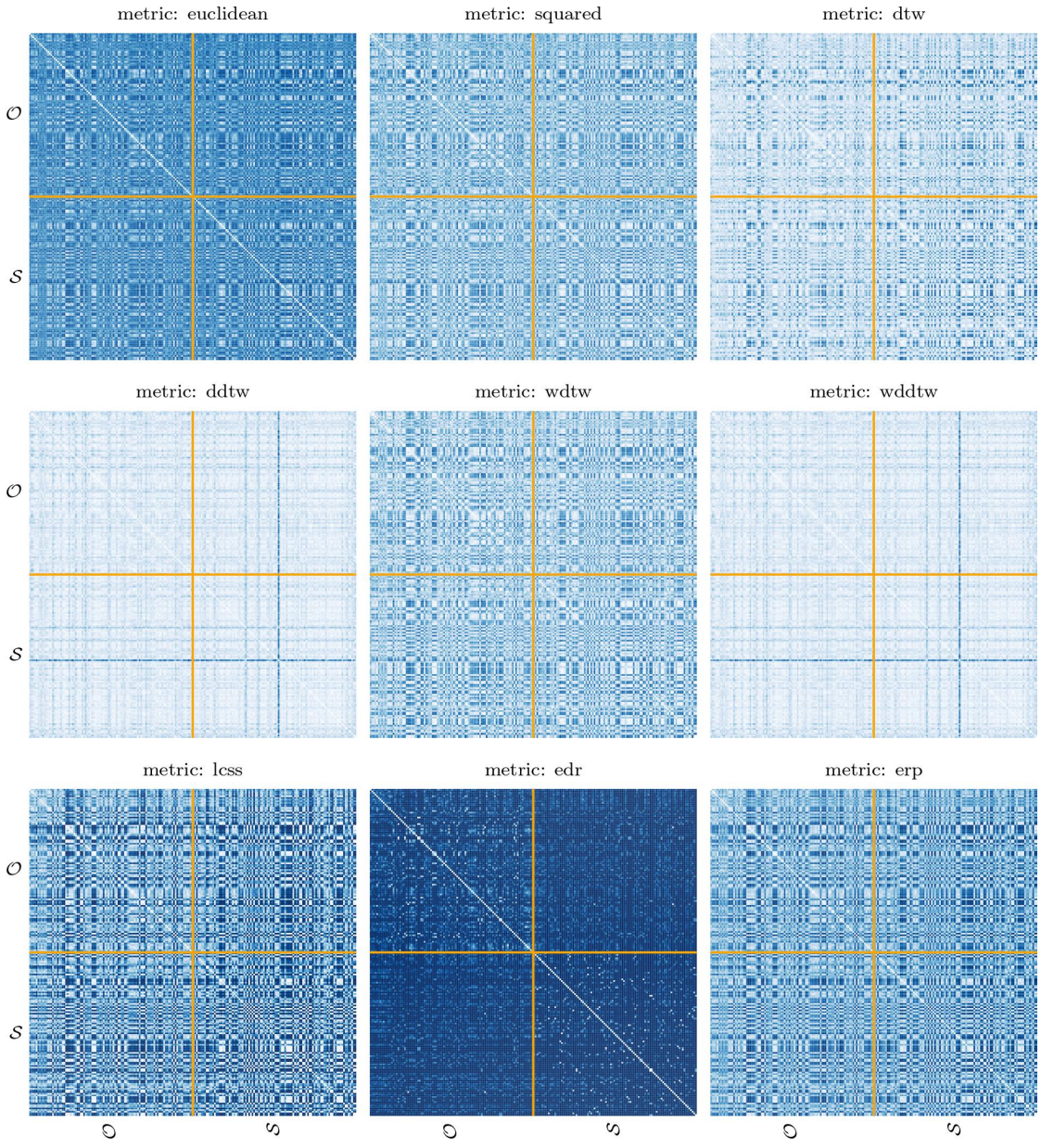


Figure 8.10: [Lorenz **M1v3**] Pairwise distance matrices between the original $\mathcal{O}$ and synthetic $\mathcal{S}$ trajectories.

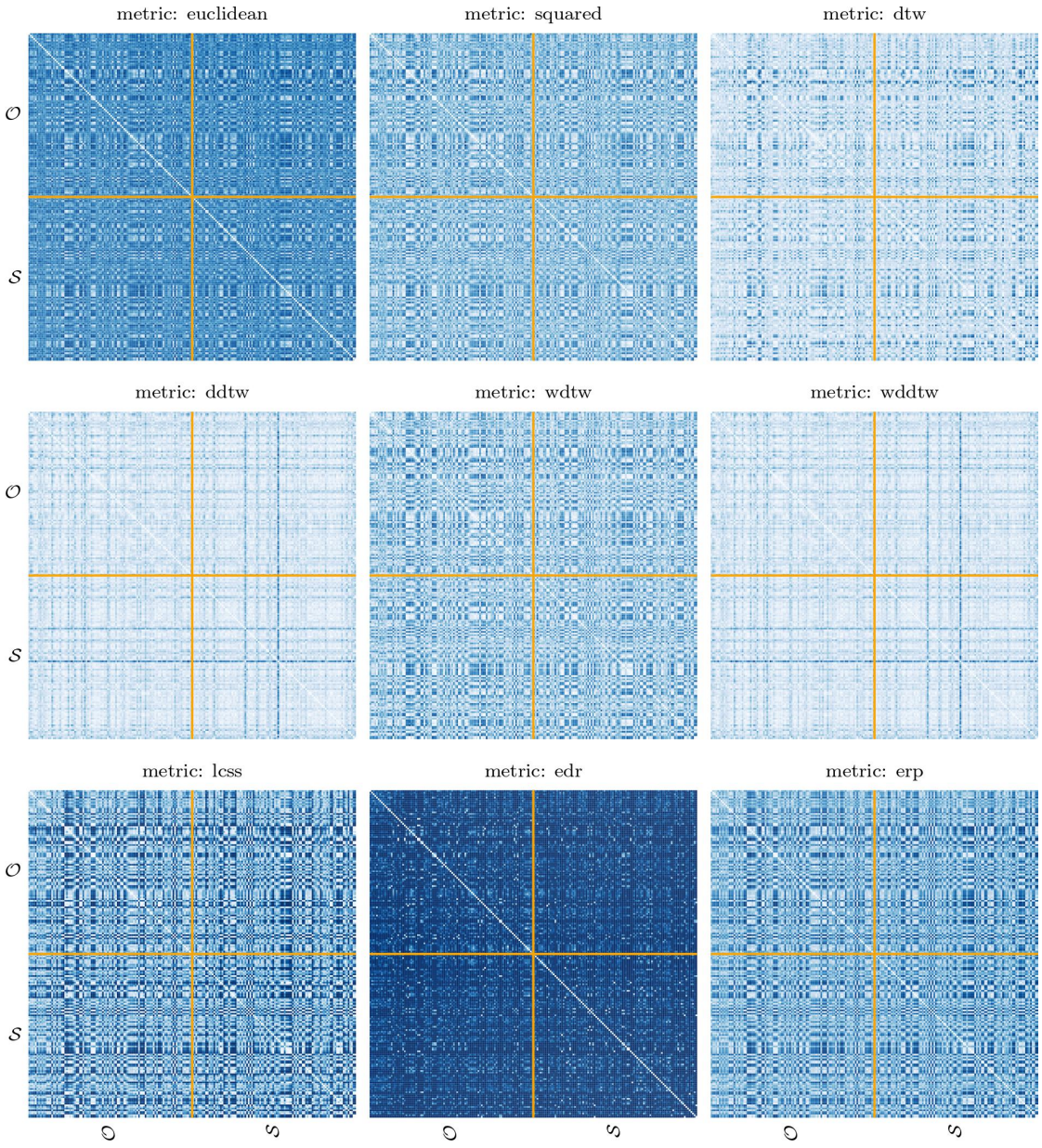


Figure 8.11: [Lorenz **M1v4**] Pairwise distance matrices between the original $\mathcal{O}$ and synthetic $\mathcal{S}$ trajectories.

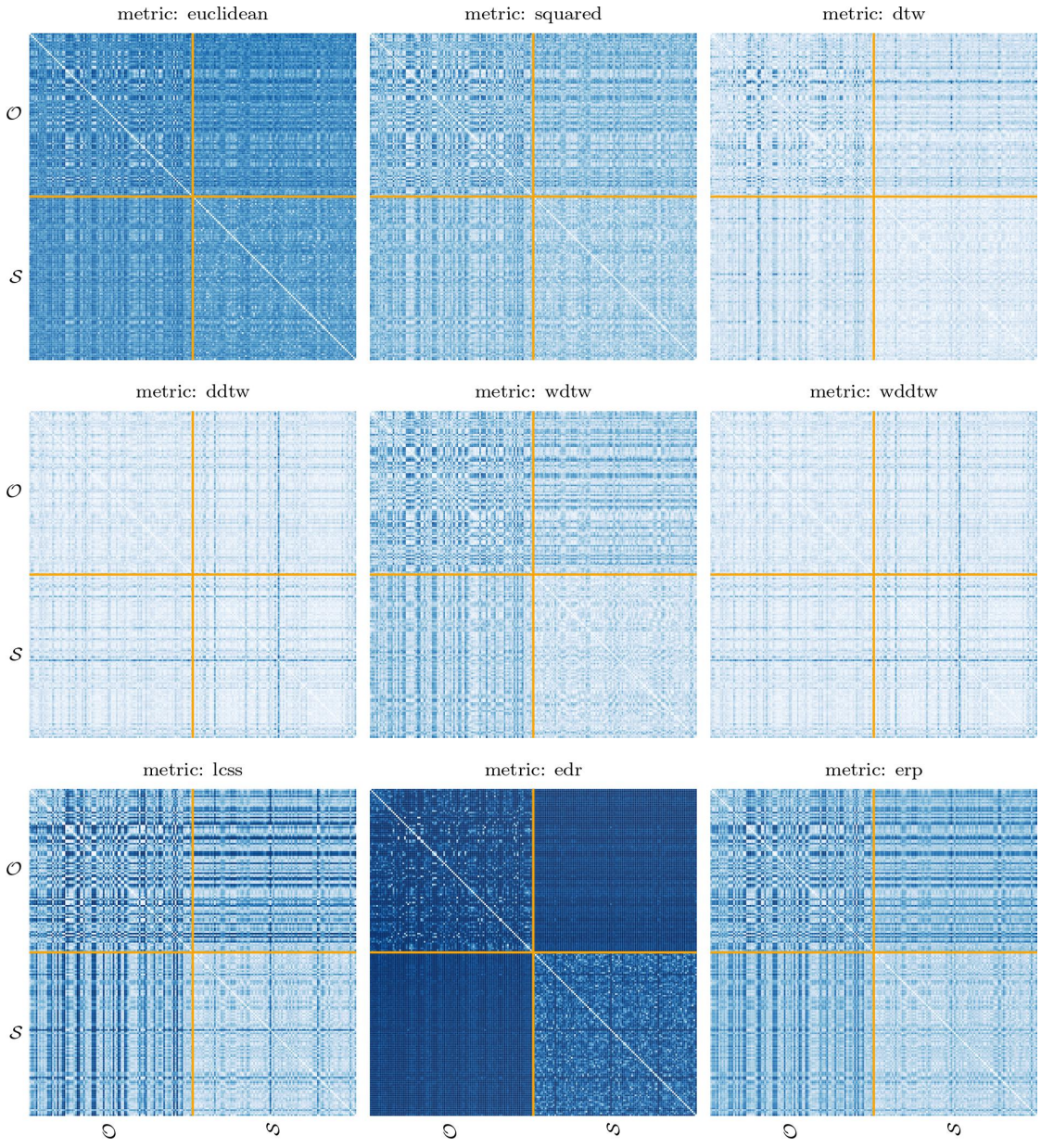


Figure 8.12: [Lorenz **M1v5**] Pairwise distance matrices between the original $\mathcal{O}$ and synthetic $\mathcal{S}$ trajectories.

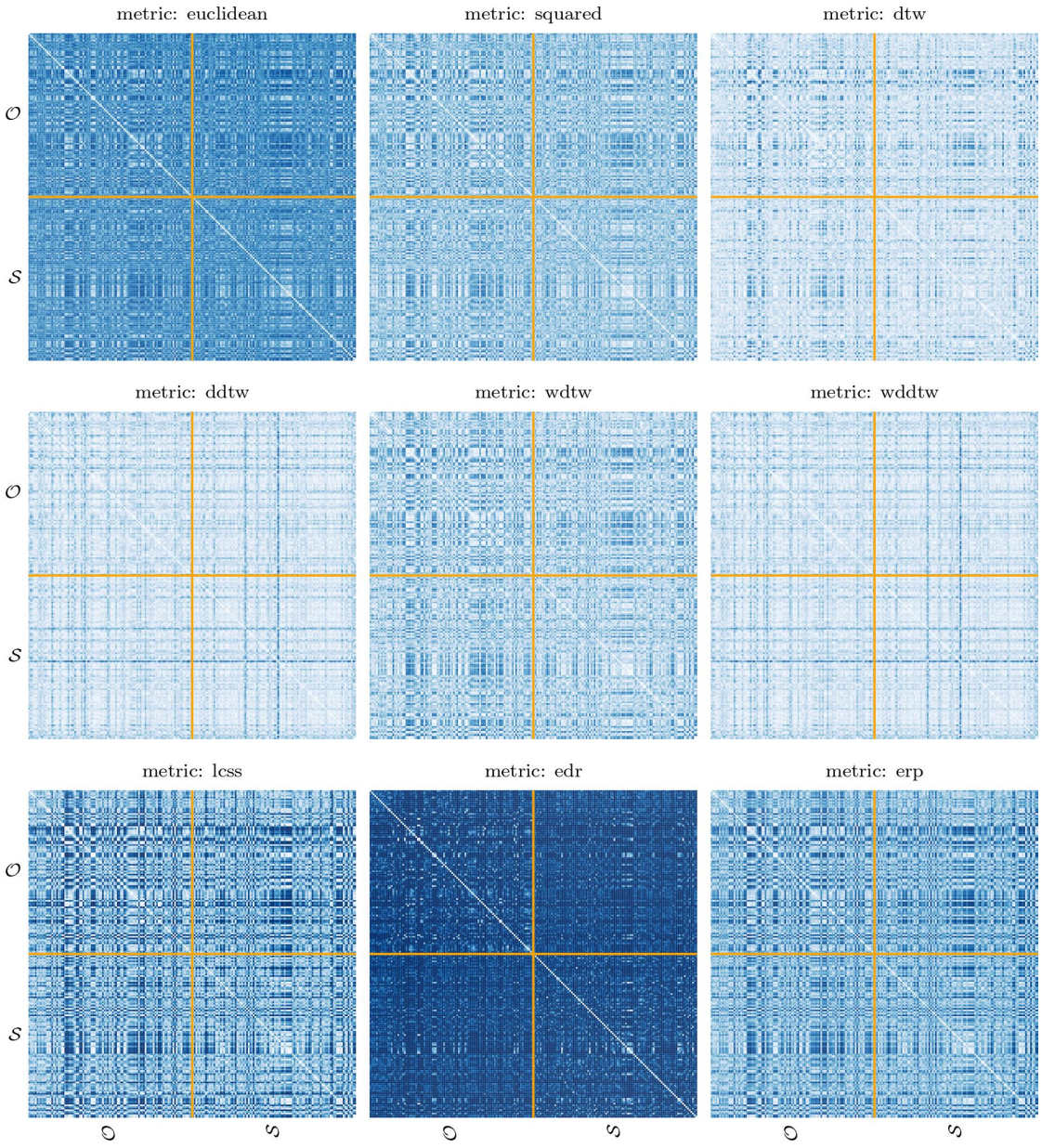


Figure 8.13: [Lorenz **M1v6**] Pairwise distance matrices between the original $\mathcal{O}$ and synthetic $\mathcal{S}$ trajectories.

8.2.2 Model **M2**

Unfortunately, due to the high-cardinality nature of the problem and the limited number of training points available, the training process for the **M2** model on the Lorenz dataset does not lead to the expected results. Specifically, this model fails to reach convergence during training, thus resulting in an output containing infinity values most likely due to exploding gradients. The lack of additional training points likely hinders the model's ability to learn the underlying patterns and variations in the Lorenz dataset, leading to suboptimal results. Therefore, it is currently not possible to properly generate synthetic data with the **M2** model for the Lorenz dataset.

8.2.3 Discussion

The training histories show that the trend of the various losses is rather stable except for **v1**, **v3** and **v5**. As for **v1** and **v3**, the trend of the reconstruction loss is particularly unstable. This is most likely due to the fact that these two models are trained using a higher learning rate than the one used for the other models. As for **v5**, the trends of all the losses except the regularization one show several spikes. This could be due to the greater weight given to the SINDy loss in $\dot{z}$, which therefore competes excessively with the other losses.

Regarding the SINDy coefficients, **v3** and **v5** discover rather sparse latent dynamics, in particular determined by fewer than 15 terms. This could be due not only to the greater weight given to the regularization loss compared to that of **v1**, but also to the following:

- higher learning rate of **v3** compared to those of **v2** and **v4**;
- shallower architecture of **v5** than that of **v6**.

Regarding the training of the `InceptionTimeClassifier` models, the trend of the training loss for **v5** is particularly stable, and the actual learning occurs within 50 epochs. Instead, the trends of the training loss for all the other models are more unstable, and the actual learning occurs within 200 epochs.

Regarding the accuracy achieved by the `InceptionTimeClassifier` models, **v1** is probably overfitted as it achieves perfect accuracy on the training set but very low accuracy, worse than that of random guessing, on the test set. Furthermore, **v2** and **v4** probably generate synthetic data that are too different from the real ones (therefore of low utility), since both the training and test accuracy are particularly high. **v3**, **v5** and **v6** are clearly the models that generate the highest utility synthetic data, since the test accuracy is approximately equal to 75% and does not deviate too much from the corresponding training accuracy.

With respect to the pairwise distance matrices between the original and synthetic trajectories, **v5** appears to introduce the highest novelty in the synthetic data. In fact, focusing on the EDR metric, a rather significant difference can be noticed between, on the one hand, the distances between the synthetic records and themselves and, on the other hand, the distances between the synthetic records and the original records.

Ultimately, it is reasonable to believe that **v5** leads to the best results. In fact, it learns a particularly sparse latent dynamical model and is able to generate synthetic data with the highest utility and novelty.

8.3 F-16 AIRCRAFT

8.3.1 Model **M1**

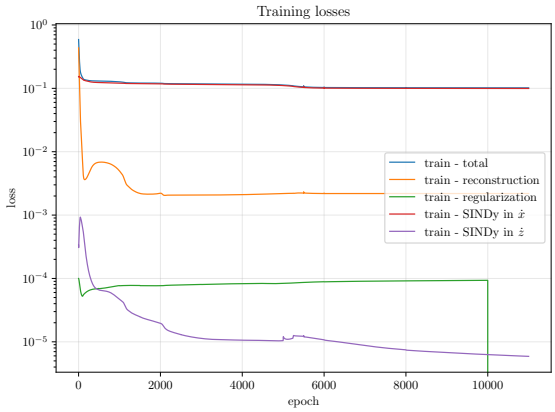
The training details of the experiments conducted with the **M1** model on the F-16 dataset are reported in Table 8.3. The model training histories and SINDy coefficients are reported in Figs. 8.14 to 8.19. Regarding the training of the `InceptionTimeClassifier` models, the trends over the epochs of the training loss are reported in Fig. 8.20. The accuracy achieved by the `InceptionTimeClassifier` models in discriminating between original and synthetic trajectories is reported in Table 8.4. The heatmaps of the pairwise distance matrices between the original and synthetic trajectories are reported in Figs. 8.21 to 8.26.

experiment version	**v1**	**v2**	**v3**	**v4**	**v5**	**v6**
seed	1	5	7	8	9	15
input dimension	5	5	5	5	5	5
latent dimension	3	3	3	3	3	3
encoder layer widths	4	5,4,3	5,4,3	5,5,4,4,3,3	5,5,5,4,4,4,3,3,3	5,5,4,4,3,3
decoder layer widths	4	3,4,5	3,4,5	3,3,4,4,5,5	3,3,3,4,4,4,5,5,5	3,3,4,4,5,5
batch size	8000	8000	8000	8000	8000	27920
activation function	ReLU	ELU	ReLU	Sigmoid	Sigmoid	ELU
SINDy loss in $\dot{x}$ – weight	10^{-4}	10^{-6}	10^{-5}	10^{-5}	10^{-5}	10^{-6}
SINDy loss in $\dot{z}$ – weight	10^{-6}	10^{-7}	10^{-6}	0	10^{-6}	10^{-7}
regularization loss – weight	10^{-4}	10^{-3}	10^{-4}	10^{-3}	10^{-2}	10^{-3}
main training – epochs	10^{4}	10^{4}	10^{4}	10^{4}	10^{4}	10^{4}
refinement training – epochs	10^{3}	10^{3}	10^{3}	10^{3}	10^{3}	10^{3}
optimizer	Adam	Adam	Adam	Adam	Adam	Adam
learning rate	10^{-4}	10^{-4}	10^{-4}	10^{-3}	10^{-4}	10^{-4}
learning rate patience	1000	1000	500	1000	1000	1000
early stopping patience	3000	3000	1500	3000	3000	3000
thresholding – threshold	0.1	0.1	0.1	0.1	0.1	0.1
thresholding – frequency	500	500	500	500	500	500
polynomial features – degree	4	4	5	4	4	4
include bias feature	yes	yes	yes	yes	yes	yes
include sine features	yes	yes	yes	yes	yes	yes
include cosine features	yes	yes	no	yes	yes	yes
# trigonometric frequencies	1	1	1	1	1	1

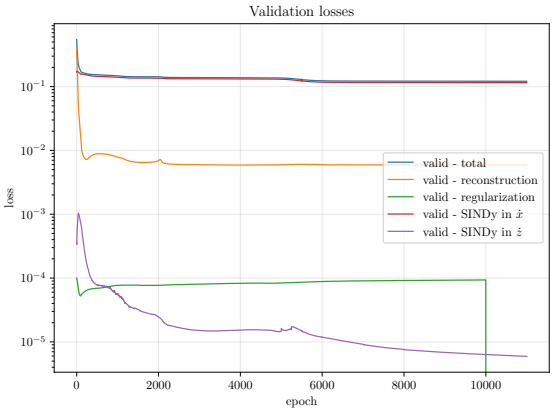
Table 8.3: [F-16 **M1**] Training details.

experiment version	**v1**	**v2**	**v3**	**v4**	**v5**	**v6**
training accuracy (%)	100	96.43	100	99.29	91.43	100
test accuracy (%)	98.33	98.33	100	100	88.33	98.33

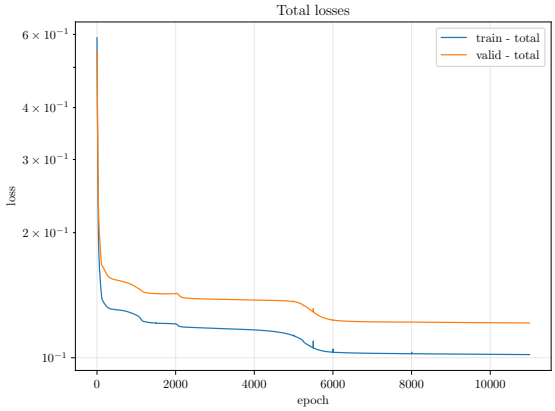
Table 8.4: [F-16 **M1**] Accuracy of the `InceptionTimeClassifier` models.



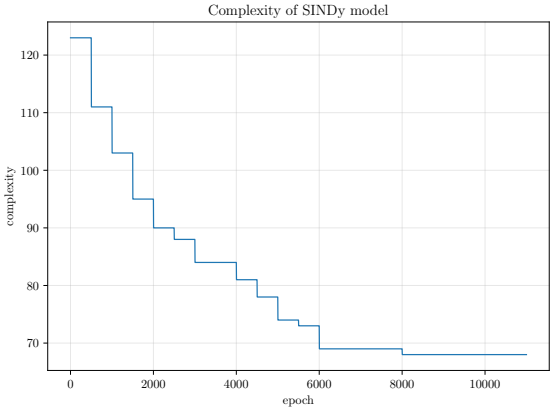
(a) Training losses.



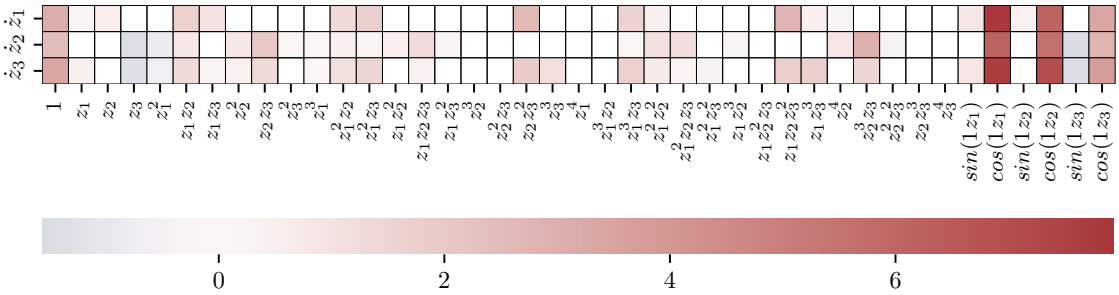
(b) Validation losses.



(c) Total losses.

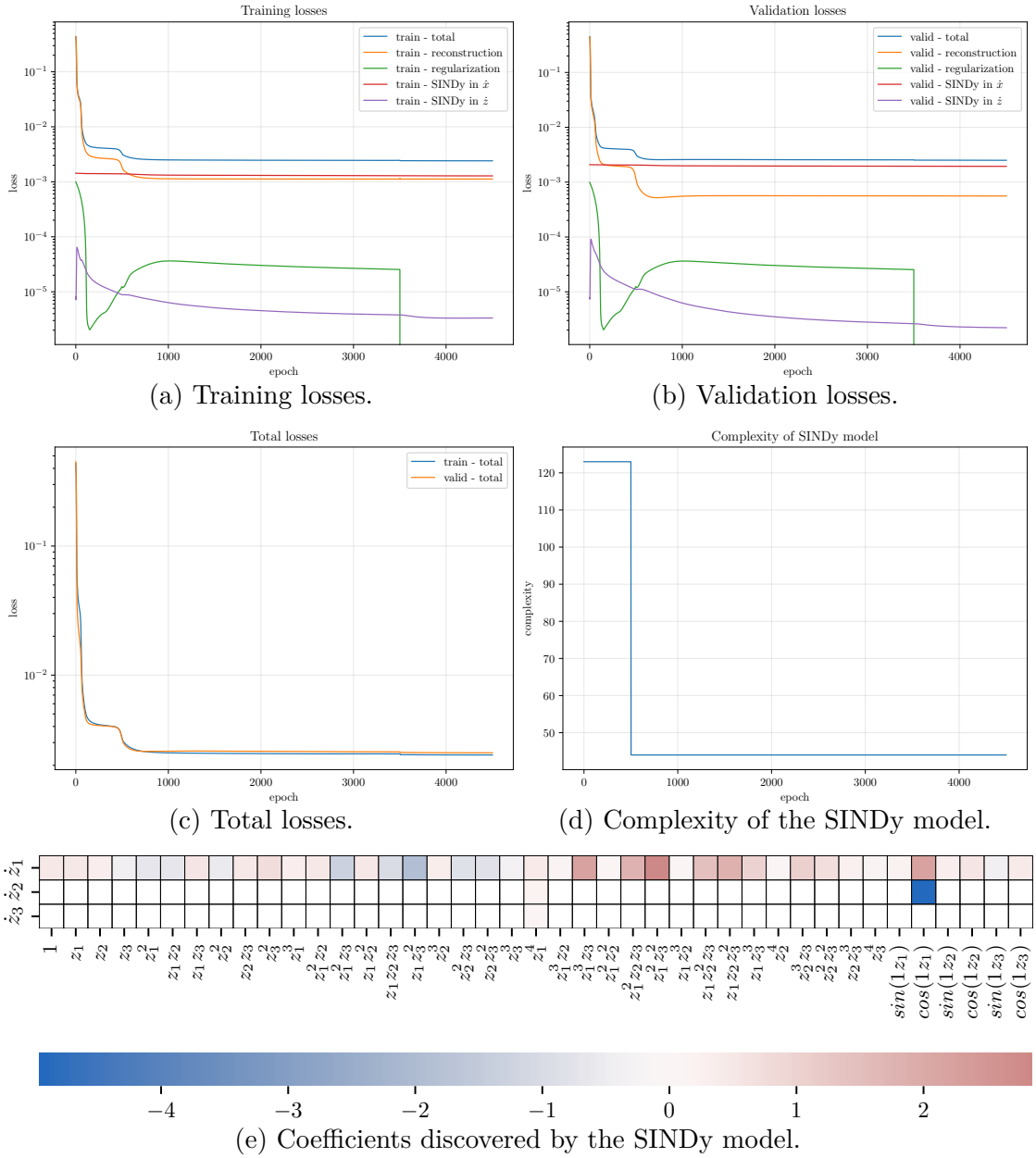


(d) Complexity of the SINDy model.



(e) Coefficients discovered by the SINDy model.

Figure 8.14: [F-16 **M1v1**] Model training history and SINDy coefficients.



(a) Training losses.

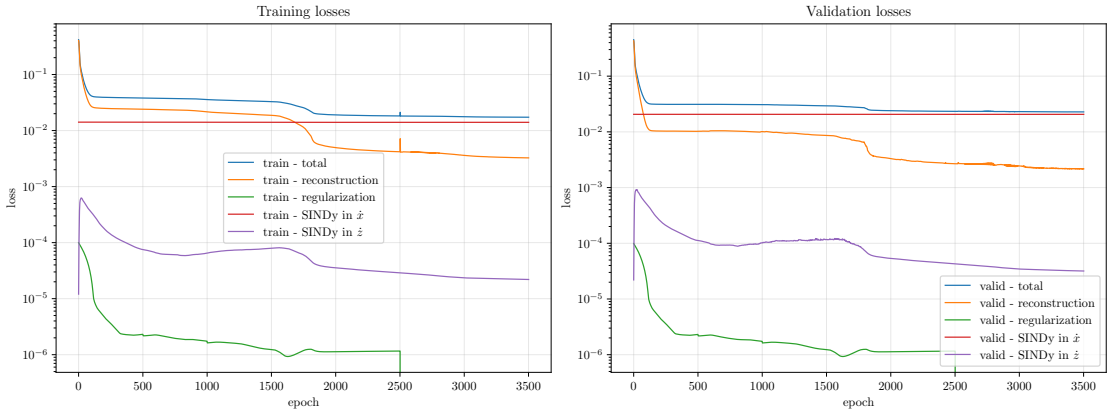
(b) Validation losses.

(c) Total losses.

(d) Complexity of the SINDy model.

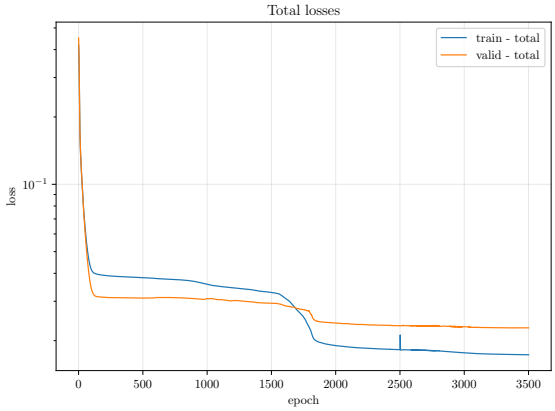
(e) Coefficients discovered by the SINDy model.

Figure 8.15: [F-16 **M1v2**] Model training history and SINDy coefficients.

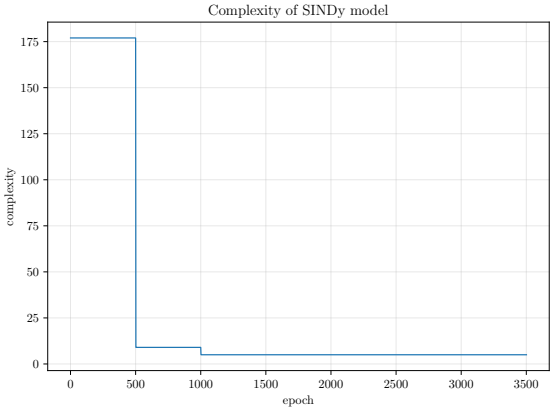


(a) Training losses.

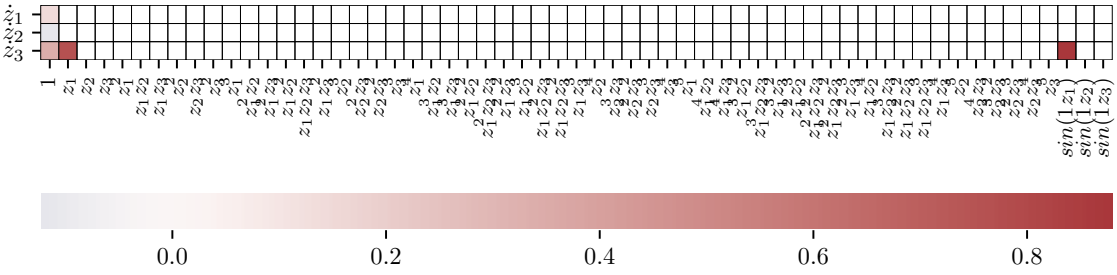
(b) Validation losses.



(c) Total losses.

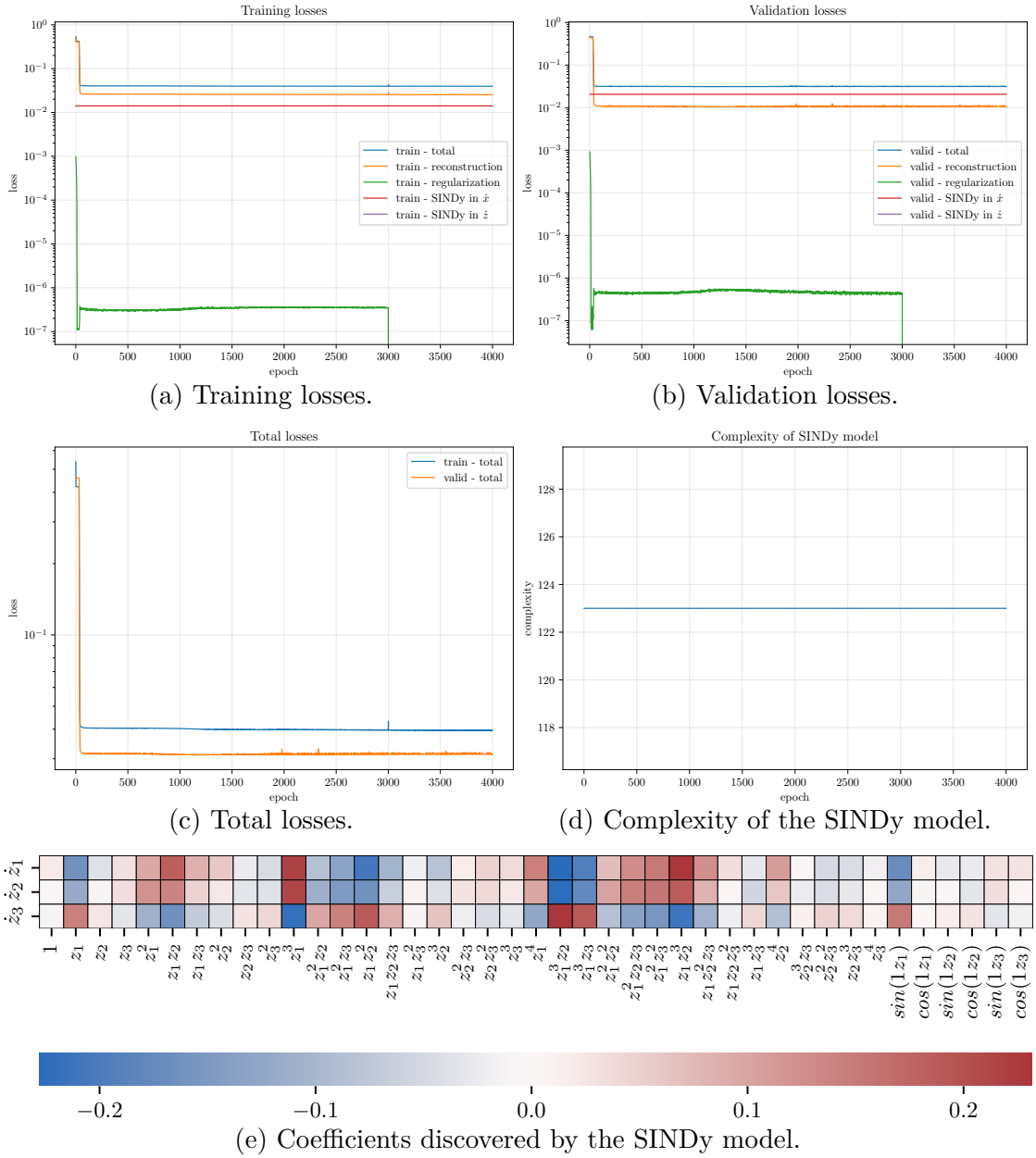


(d) Complexity of the SINDy model.



(e) Coefficients discovered by the SINDy model.

Figure 8.16: [F-16 **M1v3**] Model training history and SINDy coefficients.



(a) Training losses.

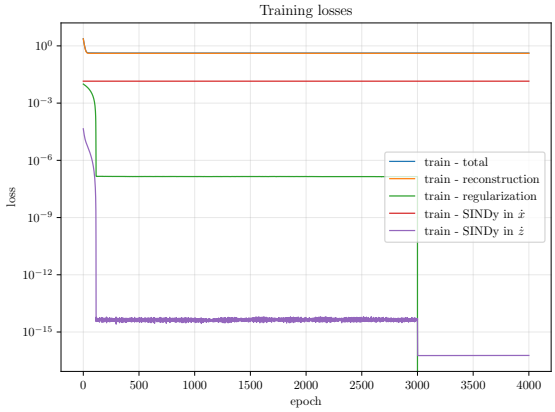
(b) Validation losses.

(c) Total losses.

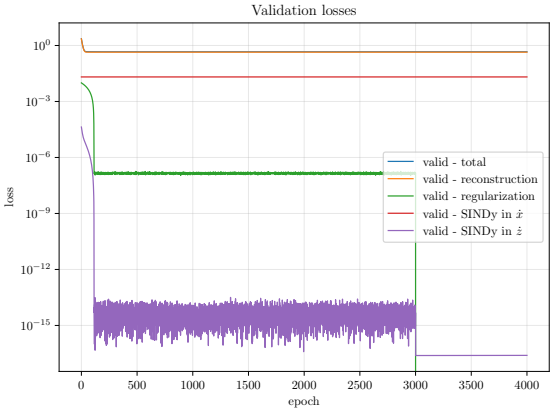
(d) Complexity of the SINDy model.

(e) Coefficients discovered by the SINDy model.

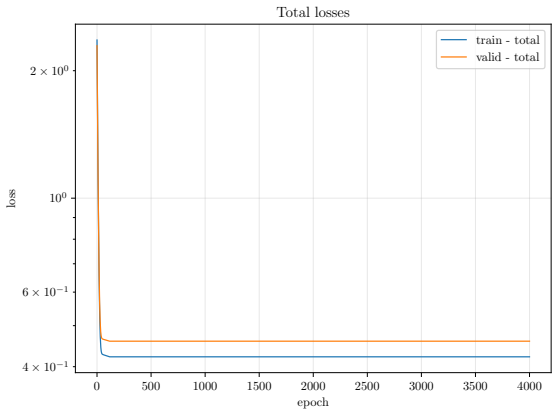
Figure 8.17: [F-16 **M1v4**] Model training history and SINDy coefficients.



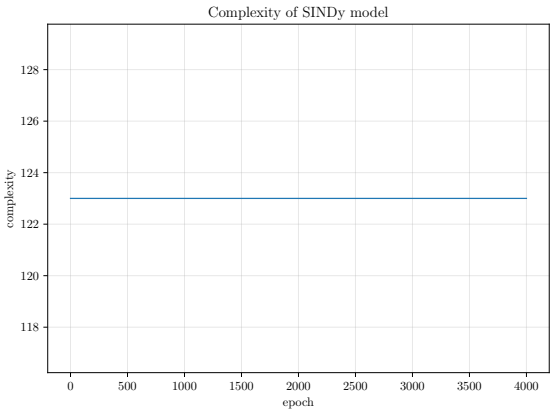
(a) Training losses.



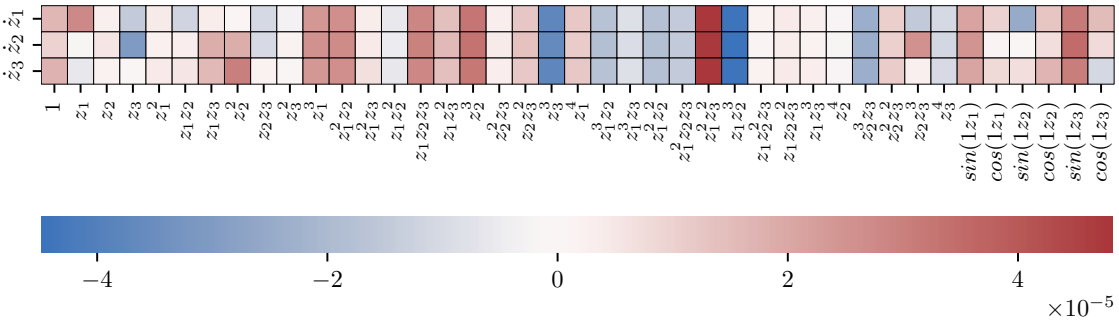
(b) Validation losses.



(c) Total losses.

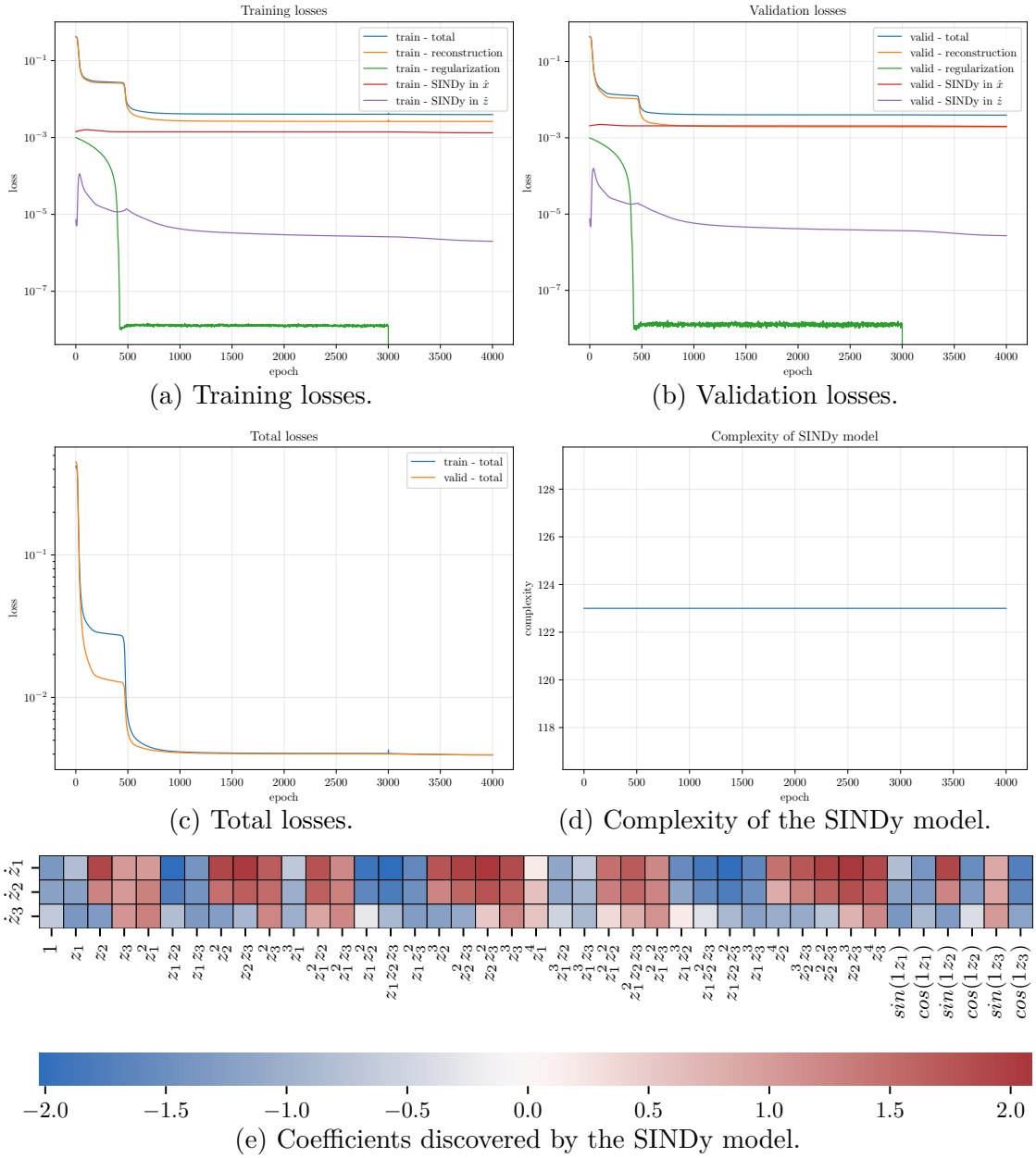


(d) Complexity of the SINDy model.



(e) Coefficients discovered by the SINDy model.

Figure 8.18: [F-16 **M1v5**] Model training history and SINDy coefficients.



(a) Training losses.

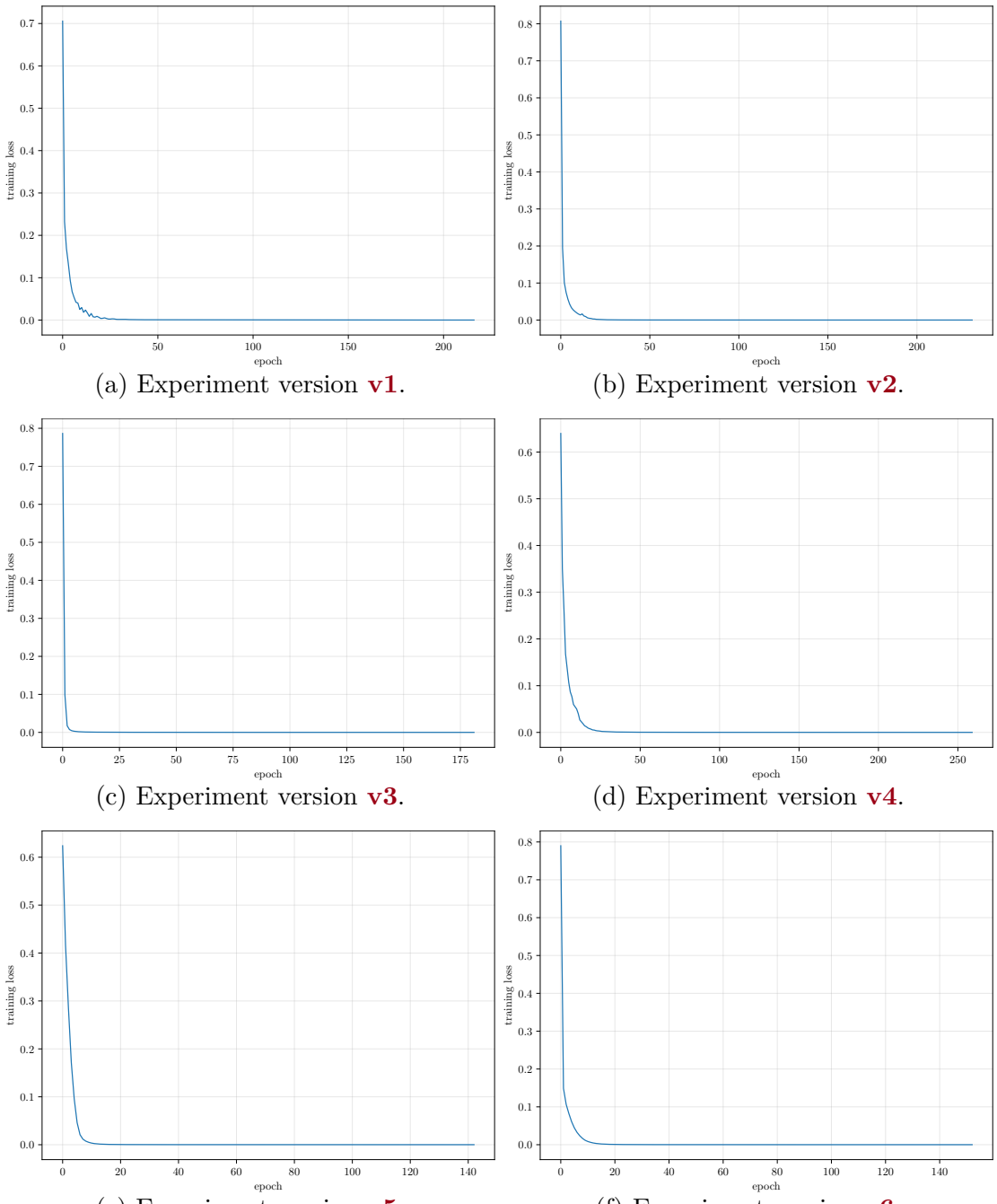
(b) Validation losses.

(c) Total losses.

(d) Complexity of the SINDy model.

(e) Coefficients discovered by the SINDy model.

Figure 8.19: [F-16 **M1v6**] Model training history and SINDy coefficients.



(a) Experiment version **v1**.

(b) Experiment version **v2**.

(c) Experiment version **v3**.

(d) Experiment version **v4**.

(e) Experiment version **v5**.

(f) Experiment version **v6**.

Figure 8.20: [F-16 **M1**] Training loss of the `InceptionTimeClassifier` models.

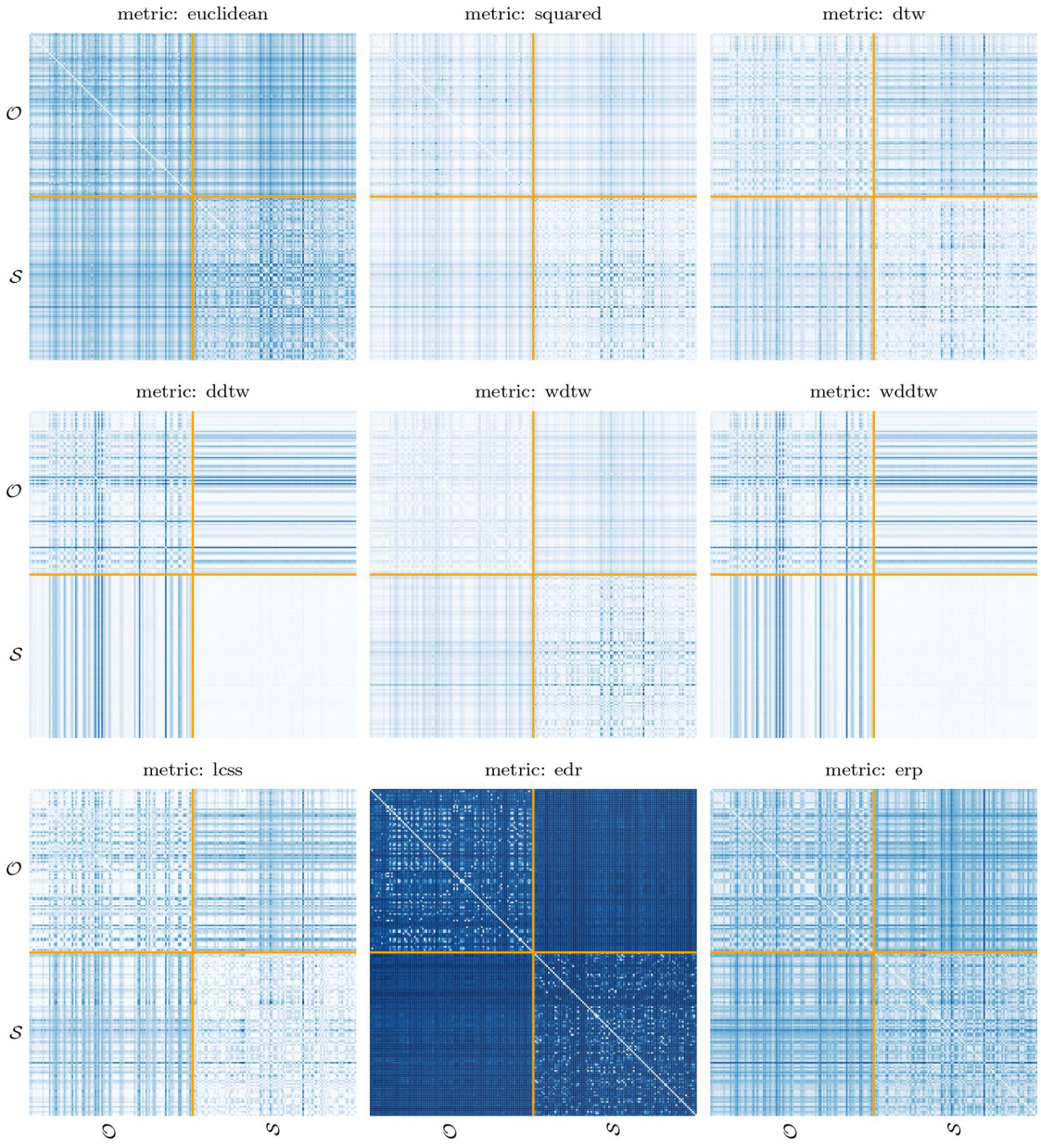


Figure 8.21: [F-16 **M1v1**] Pairwise distance matrices between the original trajectories $\mathcal{O}$ and the synthetic trajectories $\mathcal{S}$.

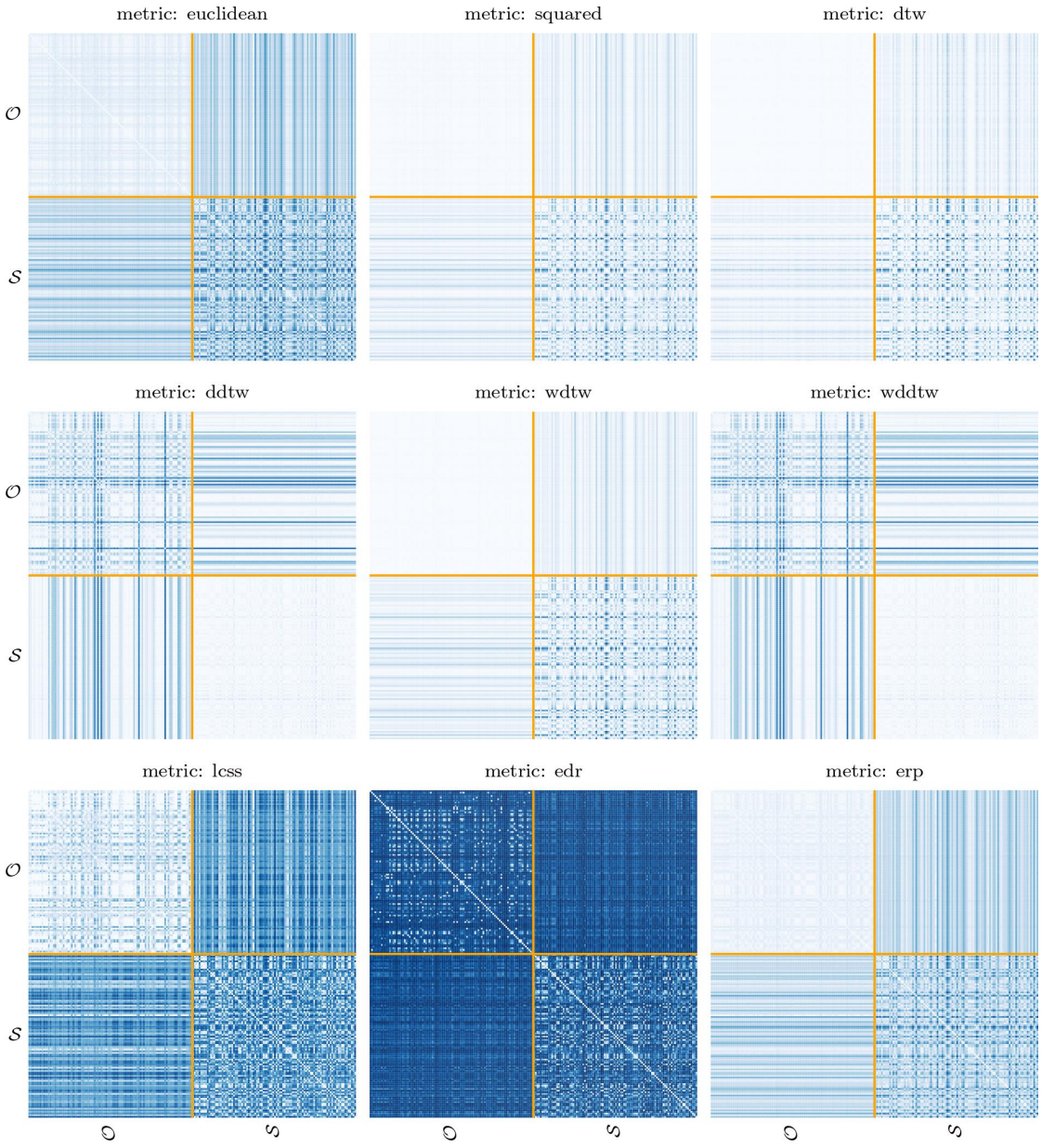


Figure 8.22: [F-16 **M1v2**] Pairwise distance matrices between the original trajectories $\mathcal{O}$ and the synthetic trajectories $\mathcal{S}$.

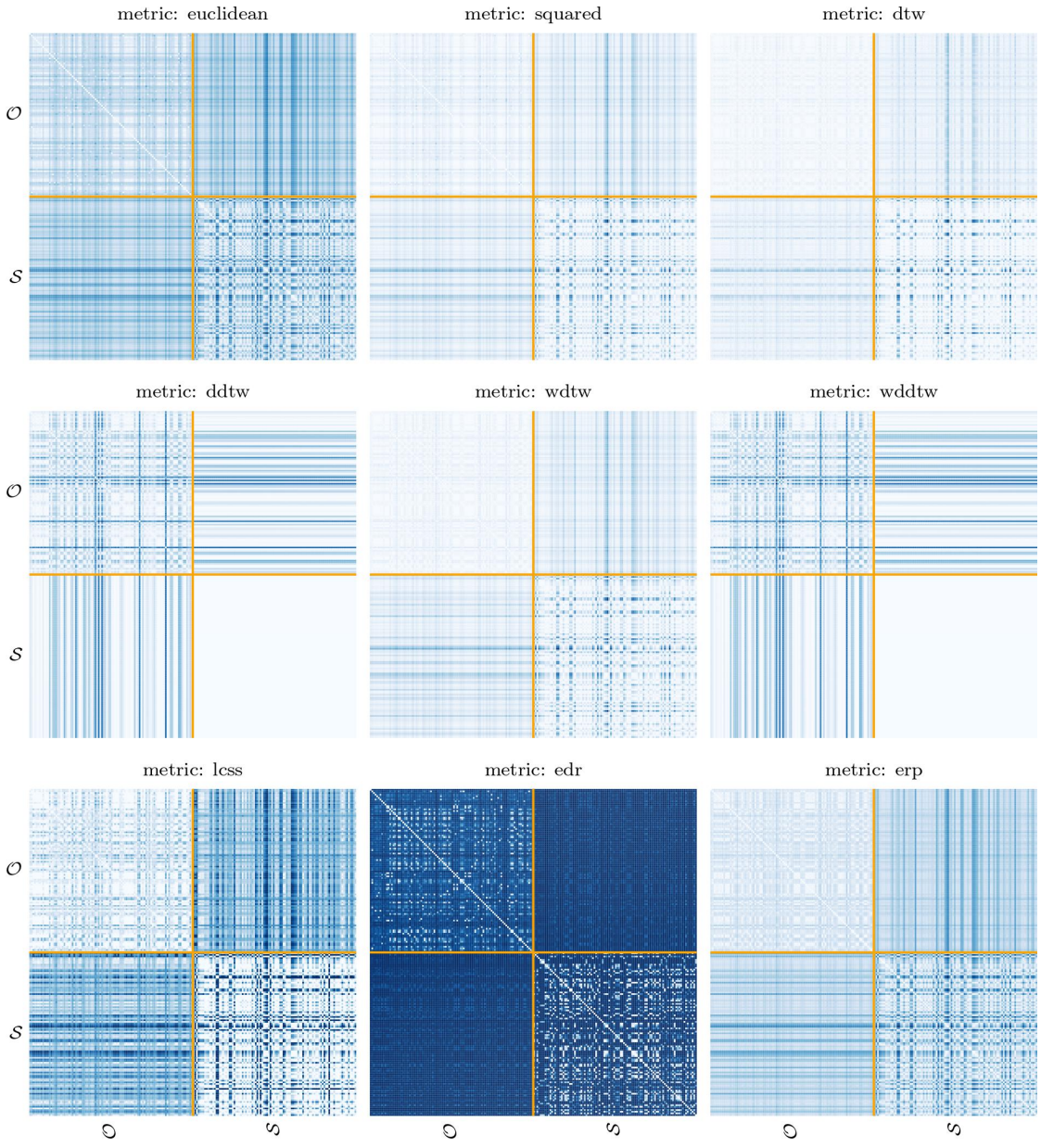


Figure 8.23: [F-16 **M1v3**] Pairwise distance matrices between the original trajectories $\mathcal{O}$ and the synthetic trajectories $\mathcal{S}$.

Figure 8.24: [F-16 **M1v4**] Pairwise distance matrices between the original trajectories $\mathcal{O}$ and the synthetic trajectories $\mathcal{S}$.

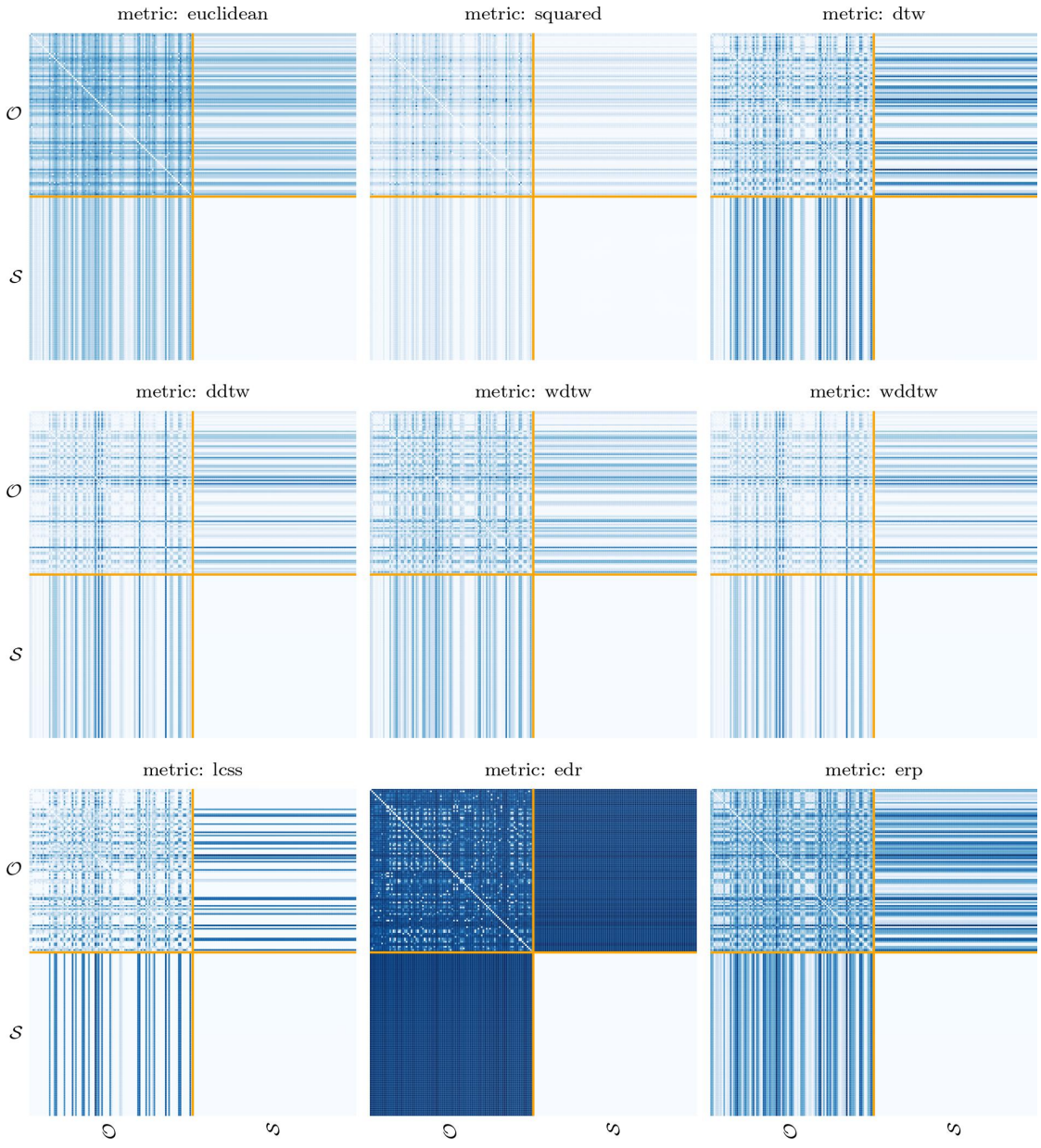


Figure 8.25: [F-16 **M1v5**] Pairwise distance matrices between the original trajectories $\mathcal{O}$ and the synthetic trajectories $\mathcal{S}$.

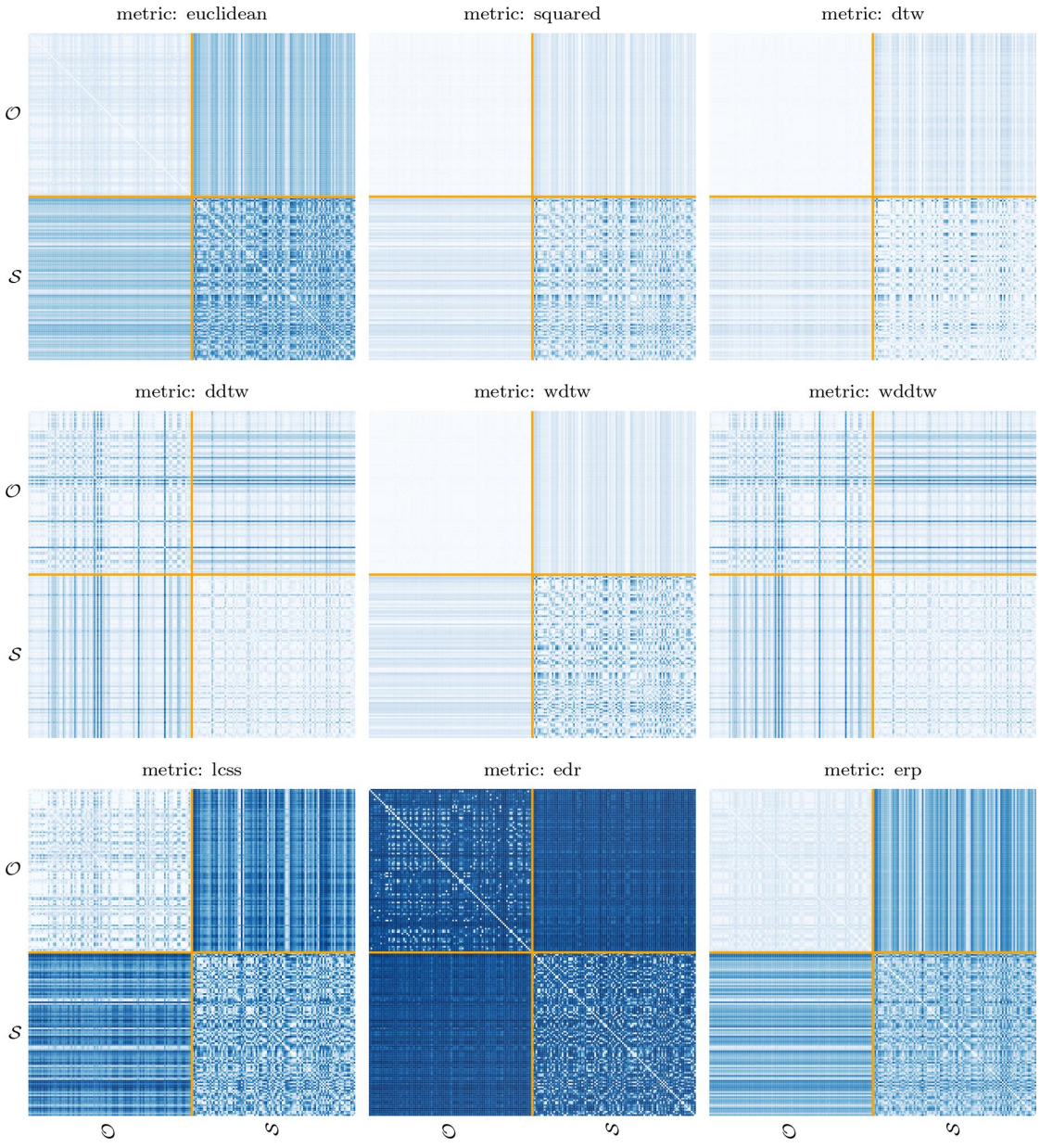


Figure 8.26: [F-16 **M1v6**] Pairwise distance matrices between the original trajectories $\mathcal{O}$ and the synthetic trajectories $\mathcal{S}$.

8.3.2 Model **M2**

Regarding the training of the `InceptionTimeClassifier` model, the trend over the epochs of the training loss is reported in Fig. 8.27. The accuracy achieved by the `InceptionTimeClassifier` model in discriminating between original and synthetic trajectories is the following:

- **training accuracy**: 96.43%;
- **test accuracy**: 71.67%.

The heatmaps of the pairwise distance matrices between the original and synthetic trajectories are reported in Fig. 8.28.

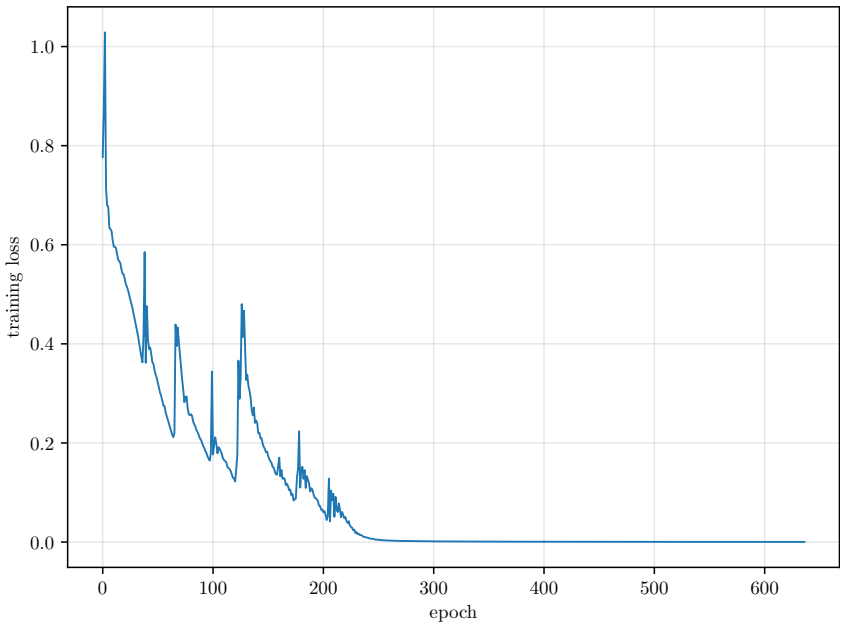


Figure 8.27: [F-16 **M2**] Training loss of the `InceptionTimeClassifier` model.

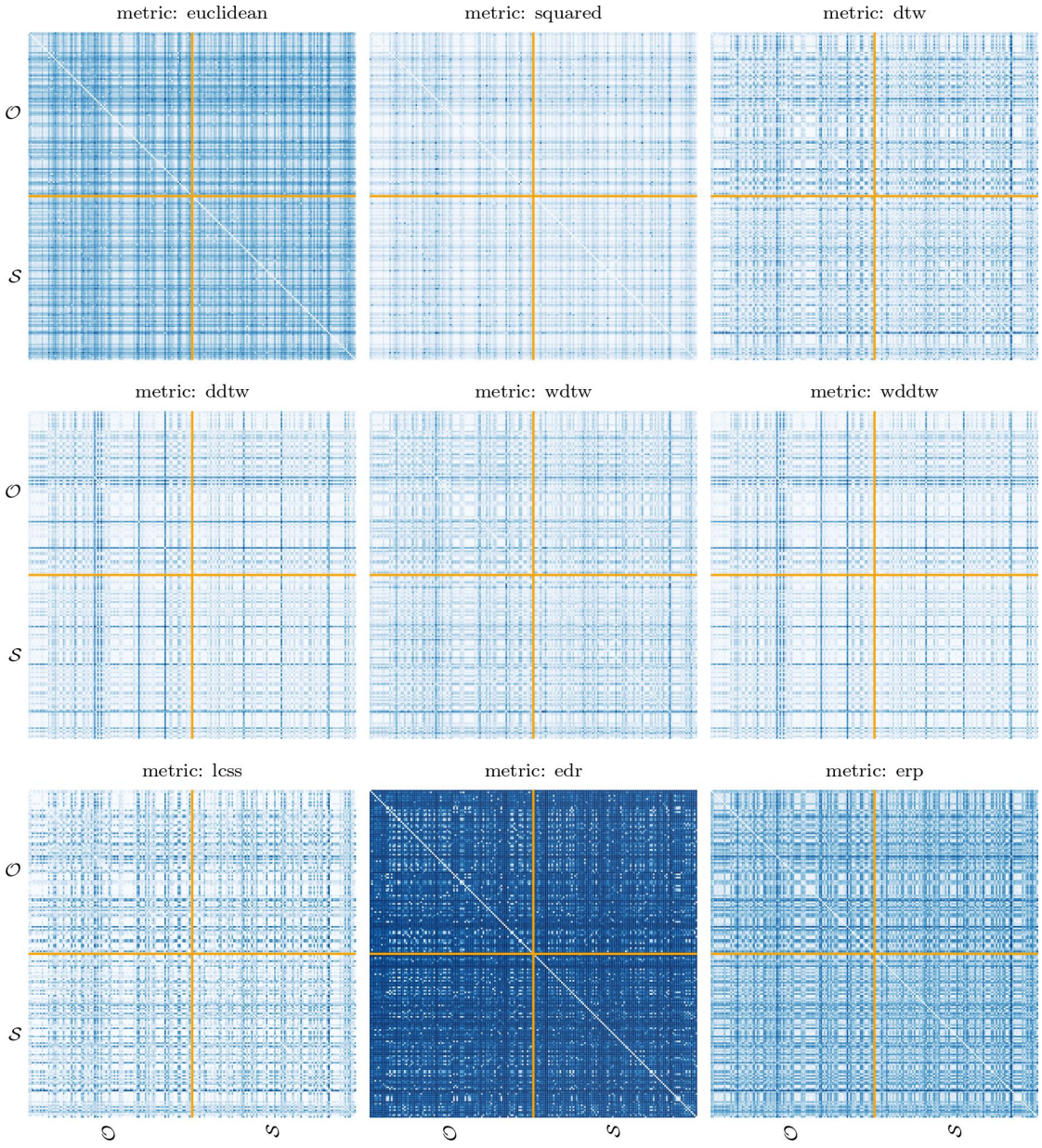


Figure 8.28: [F-16 **M2**] Pairwise distance matrices between the original $\mathcal{O}$ and synthetic $\mathcal{S}$ trajectories.

8.3.3 Discussion

The training histories show that the trend of the various losses is rather stable except for the validation SINDy loss in $\dot{z}$ of **v5**. This could be due to the fact that this model has both the deepest architecture and the most weight given to the regularization loss. Furthermore, excluding **v1**, the various SINDy losses in $\dot{x}$ are practically constant throughout the training. This could be due to the greater weight given to the SINDy loss in $\dot{x}$ of **v1** compared to that of all the other models. Moreover, the various losses stabilize within 1000 epochs in the case of **v2** and **v6**, while 6000 epochs in the case of **v1**. Instead, these losses stabilize within very few epochs in the case of **v4** and **v5**, while they continue to vary until the very end of the training in the case of **v3**.

Regarding the SINDy coefficients, **v3** discovers rather sparse latent dynamics, in particular determined by fewer than 10 terms. Instead, **v1** and **v2** discover rather complex dynamics, determined by a number of terms between 40 and 70. This difference could be due to the fact that cosine features are excluded in the case of **v3**. Furthermore, **v4**, **v5** and **v6** discover very complex dynamics, for which all the terms of the feature library are active. This may be due to their deeper architecture than all the other models.

Regarding the training of the `InceptionTimeClassifier` models, the trend of the training loss for **M2** is particularly unstable, and the actual learning occurs within 400 epochs. Instead, the trends of the training loss for all the other models are more stable, and the actual learning occurs within 50 epochs.

Regarding the accuracy achieved by the `InceptionTimeClassifier` models, all experiments except **v5** and **M2** generate synthetic data that are too different from the real ones (therefore of low utility), since both the training and test accuracy are particularly high. Instead, **v5** generates synthetic data of somewhat better utility, since the test accuracy is approximately equal to 88% and does not deviate too much from the corresponding training accuracy. **M2** is clearly the model that generates the highest utility synthetic data, since the test accuracy is approximately equal to 72% and does not deviate too much from the corresponding training accuracy.

With respect to the pairwise distance matrices between the original and synthetic trajectories, all models except **v5** and **M2** generate synthetic trajectories too far from the original ones. In fact, focusing on the DDTW and WDDTW metrics, the synthetic trajectories are all almost the same distance from the spe-

cific original trajectory and the synthetic trajectories are very close to each other. Moreover, focusing on the EDR metric, a rather significant difference can be noticed between, on the one hand, the distances between the synthetic records and themselves and, on the other hand, the distances between the synthetic records and the original records. Although similar conclusions could also be drawn for **v5**, the higher utility of the synthetic data generated by it should be taken into account. In other words, these conclusions could be interpreted as the result of some novelty introduced by **v5** in the synthetic data. In fact, focusing on the EDR metric, the most evident difference can be noticed between, on the one hand, the distances between the synthetic records and themselves and, on the other hand, the distances between the synthetic records and the original records. Finally, in the case of **M2** no significant pattern can be noticed in the various distance matrices. Therefore, although **M2** appears to generate the synthetic data with the highest utility, it seems to introduce little novelty into it.

Ultimately, it is reasonable to believe that **v5** and **M2** lead to the best results. On the one hand, **v5** learns a very complex and non-sparse dynamical latent model, but is able to generate synthetic data with quite decent utility and novelty. On the other hand, **M2** is able to generate the synthetic data with the highest utility but without introducing any significant novelty into it.

9

Conclusion

The approach explored in this thesis work proves to be a rather promising solution for generating high-fidelity synthetic data. The latent dynamical model learned by the SINDyAE turns out to be particularly convenient for generating low-dimensional synthetic data to be mapped back into the original measurement space. Thanks to its flexibility, the approach is applicable to multivariate time-series of potentially arbitrary dimension.

However, the SINDyAE training procedure is significantly complicated by the presence of multiple competing loss terms. This complication is characteristic of multi-task learning approaches and requires a considerable effort in terms of hyperparameter tuning to achieve valuable results.

Moreover, the results of this work show that the effectiveness of the approach is strongly dependent on the dataset in use and on the complexity of the corresponding dynamical system. On the one hand, the Lorenz dataset is produced by a simulation weakly affected by artificial noise and represents a relatively simple and analytically definable nonlinear dynamical system. On the other hand, the F-16 dataset consists of real-world measurements affected by non-negligible noise and represents a concrete and extremely complex nonlinear dynamical system, therefore infeasible to be modeled analytically.

Furthermore, the continuous improvement in the performance of time-series classifiers such as *InceptionTime* makes it increasingly easy to discriminate between real and synthetic data. To better define an optimal trade-off between utility and novelty, it could be useful to perform the same kind of utility analysis on a set of time-series classifiers characterized by increasing complexity. In general, it will

probably be necessary to resort to increasingly complex generative models and with a much deeper architecture than those explored in this work.

In any future work with this approach, it will certainly be necessary to focus further on hyperparameter tuning to understand which combinations lead to the best results. In addition, more specific and elaborate strategies will need to be employed to limit the effects of noise in the case of real-world data.

A possible future development could be directed towards the introduction of a further loss term in the model training process. Specifically, this loss would be justified by the data synthesis aspect of the approach. In practice, it would measure the discrepancy between the latent representation of each input data point and the corresponding data point simulated by the SINDy model in the latent space. This simulation would use the previous latent input data point as the initial condition. Such a development could improve the ability to capture the underlying dynamics in the latent space and consequently lead to higher-fidelity synthetic data.

Another possible interesting development could consist in implementing specific support for parameterized systems and exploiting the corresponding parameters to control the synthetic data generation process, thus further investigating the approach discussed in [17].

Bibliography

[1] Abraham. Savitzky and M. J. E. Golay. "Smoothing and Differentiation of Data by Simplified Least Squares Procedures." In: *Analytical Chemistry* 36.8 (July 1964), pp. 1627–1639. DOI: 10.1021/ac60214a047.

[2] H. Sakoe and S. Chiba. "Dynamic programming algorithm optimization for spoken word recognition". In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 26.1 (Feb. 1978), pp. 43–49. DOI: 10.1109/tassp.1978.1163055.

[3] Pierre Baldi and Kurt Hornik. "Neural networks and principal component analysis: Learning from examples without local minima". In: *Neural Networks* 2.1 (Jan. 1989), pp. 53–58. DOI: 10.1016/0893-6080(89)90014-2.

[4] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.5 (Jan. 1989), pp. 359–366. DOI: 10.1016/0893-6080(89)90020-8.

[5] Avrim L. Blum and Ronald L. Rivest. "Training a 3-node neural network is NP-complete". In: *Neural Networks* 5.1 (Jan. 1992), pp. 117–127. DOI: 10.1016/s0893-6080(05)80010-3.

[6] Robert Tibshirani. "Regression Shrinkage and Selection Via the Lasso". In: *Journal of the Royal Statistical Society: Series B (Methodological)* 58.1 (Jan. 1996), pp. 267–288. DOI: 10.1111/j.2517-6161.1996.tb02080.x.

[7] Eamonn J. Keogh and Michael J. Pazzani. "Derivative Dynamic Time Warping". In: *Proceedings of the 2001 SIAM International Conference on Data Mining*. Society for Industrial and Applied Mathematics, Apr. 2001. DOI: 10.1137/1.9781611972719.1.

[8] Lei Chen and Raymond Ng. "On The Marriage of Lp-norms and Edit Distance". In: *Proceedings 2004 VLDB Conference*. Elsevier, 2004, pp. 792–803. DOI: 10.1016/b978-012088469-8.50070-x.

[9] Lei Chen, M. Tamer Özsu, and Vincent Oria. "Robust and fast similarity search for moving object trajectories". In: *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. ACM, June 2005. DOI: 10.1145/1066157.1066213.

[10] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *International Conference on Artificial Intelligence and Statistics*. 2010.

[11] Rick Chartrand. "Numerical Differentiation of Noisy, Nonsmooth Data". In: *ISRN Applied Mathematics* 2011 (May 2011), pp. 1–11. DOI: 10.5402/2011/164564.

[12] Young-Seon Jeong, Myong K. Jeong, and Olufemi A. Omitaomu. "Weighted dynamic time warping for time series classification". In: *Pattern Recognition* 44.9 (Sept. 2011), pp. 2231–2240. DOI: 10.1016/j.patcog.2010.09.022.

[13] Hien Van Nguyen et al. "Kernel dictionary learning". In: *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, Mar. 2012. DOI: 10.1109/icassp.2012.6288305.

[14] Khaled El Emam et al. "A Review of Evidence on Consent Bias in Research". In: *The American Journal of Bioethics* 13.4 (Apr. 2013), pp. 42–44. DOI: 10.1080/15265161.2013.767958.

[15] Trevor Hastie, Robert Tibshirani, and Martin Wainwright. *Statistical learning with sparsity*. en. Chapman & Hall/CRC Monographs on Statistics and Applied Probability. New York, NY: Productivity Press, May 2015.

[16] Steven L. Brunton, Joshua L. Proctor, and J. Nathan Kutz. "Discovering governing equations from data by sparse identification of nonlinear dynamical systems". In: *Proceedings of the National Academy of Sciences* 113.15 (Mar. 2016), pp. 3932–3937. DOI: 10.1073/pnas.1517384113.

[17] Steven L. Brunton, Joshua L. Proctor, and J. Nathan Kutz. "Sparse Identification of Nonlinear Dynamics with Control (SINDYc)". In: *IFAC-PapersOnLine* 49.18 (2016), pp. 710–715. DOI: 10.1016/j.ifacol.2016.10.249.

[18] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL: https://www.deeplearningbook.org.

[19] Jason D. Lee et al. *Gradient Descent Converges to Minimizers*. 2016. DOI: 10.48550/ARXIV.1602.04915.

[20] Christian Szegedy et al. *Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning*. 2016. DOI: 10.48550/ARXIV.1602.07261.

[21] Hayden Schaeffer and Scott G. McCalla. "Sparse model selection via integral terms". In: *Physical Review E* 96.2 (Aug. 2017). DOI: 10.1103/physreve.96.023302.

[22] Linan Zhang and Hayden Schaeffer. *On the Convergence of the SINDy Algorithm*. 2018. DOI: 10.48550/ARXIV.1805.06445.

[23] Kathleen Champion et al. "Data-driven discovery of coordinates and governing equations". In: *Proceedings of the National Academy of Sciences* 116.45 (Oct. 2019), pp. 22445–22451. DOI: 10.1073/pnas.1906995116.

[24] Samuel H. Rudy, J. Nathan Kutz, and Steven L. Brunton. "Deep learning of dynamics and signal-noise decomposition with time-stepping constraints". In: *Journal of Computational Physics* 396 (Nov. 2019), pp. 483–506. DOI: 10.1016/j.jcp.2019.06.056.

[25] Khaled El Emam, Lucy Mosquera, and Richard Hoptroff. *Practical synthetic data generation*. Sebastopol, CA: O'Reilly Media, June 2020.

[26] Hassan Ismail Fawaz et al. "InceptionTime: Finding AlexNet for time series classification". In: *Data Mining and Knowledge Discovery* 34.6 (Sept. 2020), pp. 1936–1962. DOI: 10.1007/s10618-020-00710-y.

[27] Jean-Philippe Noël and Maarten Schoukens. *F-16 Aircraft Benchmark Based on Ground Vibration Test Data*. 2020. DOI: 10.4121/12954911.

[28] George Em Karniadakis et al. "Physics-informed machine learning". In: *Nature Reviews Physics* 3.6 (May 2021), pp. 422–440. DOI: 10.1038/s42254-021-00314-5.

[29] Joseph Bakarji et al. *Discovering Governing Equations from Partial Measurements with Deep Delay Autoencoders*. 2022. DOI: 10.48550/ARXIV.2201.05136.

[30] Salvatore Cuomo et al. "Scientific Machine Learning Through Physics-Informed Neural Networks: Where we are and What's Next". In: *Journal of Scientific Computing* 92.3 (July 2022). DOI: 10.1007/s10915-022-01939-z.

[31] Johann Faouzi. "Time Series Classification: A review of Algorithms and Implementations". In: *Machine Learning (Emerging Trends and Applications)*. Ed. by Ketan Kotecha. Proud Pen, 2022. URL: https://inria.hal.science/hal-03558165.

[32] Zhongkai Hao et al. *Physics-Informed Machine Learning: A Survey on Problems, Methods and Applications*. 2022. DOI: 10.48550/ARXIV.2211.08064.

[33] Weiheng Zhong and Hadi Meidani. "PI-VAE: Physics-Informed Variational Auto-Encoder for stochastic differential equations". In: *Computer Methods in Applied Mechanics and Engineering* 403 (Jan. 2023), p. 115664. DOI: 10.1016/j.cma.2022.115664.

[34] M. Vlachos, G. Kollios, and D. Gunopulos. "Discovering similar multidimensional trajectories". In: *Proceedings 18th International Conference on Data Engineering*. IEEE Comput. Soc. DOI: 10.1109/icde.2002.994784.