CS4240 — Reproducibility Project — Blog Post

Group 41

Ian Maes (5216931), Matei Dinescu (5260248), Kyle Scherpenzeel (5310210), Jim van Oosten (4734998)

Riccardo Weis & Alejandro Castaneda

Abstract—*This blog post presents a reproduction study of the paper "Data-driven discovery of coordinates and governing equations" by Kathleen Champion, Bethany Lusch, J. Nathan Kutz, and Steven L. Brunton. In this project, conducted as part of Deep Learning (CS4240), we replicated the code from the original study. The post discusses our understanding of the original paper, the methodology, and approach we employed for reproduction, and the results of our reproduction.*

Contents

1. Introduction

Welcome to our blog post on the reproduction of the Sparse Identification of Nonlinear Dynamics (SINDy) with autoencoders presented in the paper "Data-driven discovery of coordinates and governing equations" [1]. This method tries to identify the underlying dynamical systems from complex datasets, a challenge more common now than ever in numerous scientific disciplines including fluid dynamics, biological systems, and climate science.

Understanding complex systems from high-dimensional data can be difficult due to not only needing to identify the governing equations, but also to optimize the coordinate systems for the most simple equations. Traditional approaches either focussed on only one of these aspects, or required significant simplifications, rendering them practically useless.

The integration of SINDy with autoencoders represents a significant advancement because it addresses both difficulties simultaneously. SINDy aims to extract the most simple models possible, while autoencoders excel in distilling high-dimensional data into its most important features and finding the best coordinate system to describe the data's structure. By combining both methods, the approach enhances the ability to decode complex data and also ensures that the models are simple and interpretable.

In this blog post, we describe in more detail the methodologies used, and our results in reproducing the paper from the original study.

2. Theory

SINDy needs specific data. This is typically time-series measurements for a systems' state variables, represented as X. The algorithm also requires the time derivatives of the state variables represented as $\dot{X}$

SINDy starts off by constructing a library of candidate functions $\Theta(X)$. These are potential terms that might appear in the differential equation. This library can contain simple polynomials, trigonometric functions, interactions between variables, and other nonlinear transformations. For example, if the state variables are x and y, the library might include terms like $[1, x, y, x^2, xy, y^2, \sin(x), ...]$.

The core of SINDy is to find the sparsest combination of these candidate functions that best models the derivative data $\dot{X}$. This involves solving the optimization problem:

$$\dot{X} \approx \Theta(X)\Xi \quad (1)$$

where Ξ is a matrix of coefficients. Each column of Ξ corresponds to a state variable, and each row corresponds to a function in the library. The goal is to determine the fewest non-zero coefficients in Ξ that will still allow $\Theta(X)\Xi$ to approximate $\dot{X}$ closely.

To enforce sparsity, sequential thresholding is used. This adds a regularization term to the loss function that penalizes the number of non zero entries in Ξ. This is the SINDy regularization term that can be seen in Figure 1.

The autoencoder exists of an encoder and decoder. The encorder compresses the input data into a smaller, encoded representation. This transforms the high-dimensional input x into a lower dimensional latent space z. This transformation is represented as:

$$z = \phi(x) \quad (2)$$

The decoder attempts to reconstruct the input data from the transformed form. This is represented as:

$$\tilde{x} = \psi(z) \quad (3)$$

The correct encoding and decoding is forced by including a construction loss in the loss function as seen in Figure 1. Essentially, the autoencoder reduces the dimensionality of complex data, extracting a set of reduced coordinates z that still capture the critical features of the data. This is integrated with the SINDy as one single optimization problem. The architecture and loss function can be seen in Figure 1. Both a SINDy loss in the non-transformed and transformed coordinate frame are included, this is to make sure the SINDy correctly models the dynamics in the latent space, and to make sure the SINDy model can be used to model the time-derivates in the original coordinate frame.

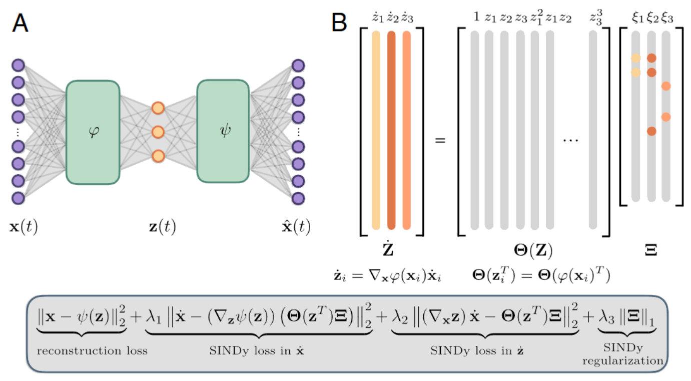
A
B
φ
ψ
$\mathbf{x}(t)$
$\mathbf{z}(t)$
$\hat{\mathbf{x}}(t)$
$\dot{z}_1\ \dot{z}_2\ \dot{z}_3$
$1\ z_1\ z_2\ z_3\ z_1^2\ z_1 z_2\ \cdots\ z_3^3$
$\xi_1\ \xi_2\ \xi_3$
$\dot{\mathbf{Z}}$
$\boldsymbol{\Theta}(\mathbf{Z})$
$\boldsymbol{\Xi}$
$\dot{\mathbf{z}}_i = \nabla_{\mathbf{x}}\varphi(\mathbf{x}_i)\dot{\mathbf{x}}_i$
$\boldsymbol{\Theta}(\mathbf{z}_i^T) = \boldsymbol{\Theta}(\varphi(\mathbf{x}_i)^T)$
$\underbrace{\|\mathbf{x}-\psi(\mathbf{z})\|_2^2}_{\text{reconstruction loss}} + \lambda_1 \underbrace{\left\|\dot{\mathbf{x}} - (\nabla_{\mathbf{z}}\psi(\mathbf{z}))\left(\boldsymbol{\Theta}(\mathbf{z}^T)\boldsymbol{\Xi}\right)\right\|_2^2}_{\text{SINDy loss in } \dot{\mathbf{x}}} + \lambda_2 \underbrace{\left\|(\nabla_{\mathbf{x}}\mathbf{z})\,\dot{\mathbf{x}} - \boldsymbol{\Theta}(\mathbf{z}^T)\boldsymbol{\Xi}\right\|_2^2}_{\text{SINDy loss in } \dot{\mathbf{z}}} + \lambda_3 \underbrace{\|\boldsymbol{\Xi}\|_1}_{\text{SINDy regularization}}$

Figure 1. Representation of the SINDy autoencoder and the used loss function [1].

3. Reproduction

The code for the original paper was written using TensorFlow, however, we transitioned the original implementation from TensorFlow to PyTorch. TensorFlow and PyTorch are both frameworks designed for deep learning tasks, but they use different computational approaches. TensorFlow uses a static graph model, where the computation graph is defined once and executed repeatedly. This model is highly efficient and well-suited for using deep learning models in production environments where performance and scalability are important. On the other hand, PyTorch operates on a dynamic graph model which is constructed during runtime. This dynamic nature provides advantages for research and development, where models require frequent adjustments. PyTorch's approach simplifies the debugging process and enhances the manageability of gradients, making it useful for exploratory tasks and iterative model adjustments.

For this reproduction, we needed to rewrite two critical components: the training script and the autoencoder script, transitioning them from TensorFlow to PyTorch. This chapter will address these two modifications separately. The third core module, of which we used the original version as it did not contain TensorFlow, implements the functions for creating and using SINDy. The functions `sindy_library` and `sindy_library_order2` build polynomial and optional sine function bases for modeling the dynamics of input data. `sindy_fit` performs sparse regression to prune the model by removing insignificant terms. `sindy_simulate` and `sindy_simulate_order2` then use these models to simulate and predict system behavior over time.

3.1. Autoencoder and Training Loop

The main altercation that had to be made when converting the code to PyTorch was the use of classes. In the original script, essentially the autoencoder is composed of individual functions and libraries. In the new script, the full network has been transformed into a class and the forward and loss functions are now grouped in this class. The layers are now created by subclasses like `LinearAutoencoder` and `NonLinearAutoencoder`. This improved the conciseness and readability of the code overall. For example, previously the weights and biases were defined separately. However, with the use of `nn.Linear` they are defined together and succinctly.

To convert the training loop from TensorFlow to PyTorch, several changes were made to align with the characteristics and typical practices of PyTorch.

Device Management
In the TensorFlow script, device management is implicitly handled, where the framework recognizes and utilizes GPU availability without explicit instructions. In contrast, our PyTorch implementation requires explicit definition of the computing device, such as specifying `cuda:0` for GPU or CPU. Our approach in PyTorch thus offers control over where computations are executed.

Network Initialization
In the original script, the model and its operations are defined statically, with components such as the Adam optimizer linked to managed placeholders and tensors. In contrast, our script uses an object-oriented approach where the model, instantiated from the `FullNetwork` class, is directly moved to a specified device. The optimizer in PyTorch directly accesses these parameters through `autoencoder_network.parameters()`, showing PyTorch's integration of parameter handling and gradient updates. This simplifies the modification and optimization process during training.

Loss and Gradient Management
In the TensorFlow script, loss operations and optimizers are configured once and executed within a session-controlled computational graph. This way, explicit session calls are necessary for model updates. In contrast, our PyTorch implementation integrates loss computation and backpropagation directly within the training loop, employing methods like `zero_grad()`, `backward()`, and `step()` to manage gradients and update parameters iteratively. This process shows PyTorch's dynamic graph capabilities, which allow for real-time adjustments to the computation during training.

Session vs. Dynamic Execution
TensorFlow uses a session to execute its computational graph, where data must be passed for each operation via feed dictionaries, making the process static and batch-oriented. PyTorch eliminates the need for a session, executing commands immediately with data loaded directly into tensors, which are then processed through the model.

Debugging and Validation
In the original script, debugging and logging are managed manually within the session environment. This way, one would need separate execution runs or additional configurations to access intermediate data for analysis. Our PyTorch script streamlines this process using the `torch.no_grad()` function during validation phases, which prevents the computation of gradients to conserve resources and ensure the model's training state is unaffected.

Model Saving and Thresholding
The TensorFlow script saves model parameters and session states using saver objects that encapsulate the entire session's state, while our PyTorch script uses a more streamlined approach, directly saving model parameters to disk with

`torch.save()`.

4. Results

The results for the training of the standard model, for which the hyperparameters are the same as in the original paper and can be seen in Table 1. Unfortunately, this model did not get accurate results. In the appendix of the original paper it is mentioned that ten models were trained with these hyperparameters [2]. Eventually, 2 were looked at which showed promising results. Likely, better results would be achieved if more models were trained, but due to time constraints, this was not done. The visual result can be seen in Figure 2. This discovered model did only have 7 active coefficients, indicating that regularization and sequential thresholding is working correctly.

Table 1. General hyperparameters for standard model.

Training samples	512000
Learning rate	1e-3
Thresholding frequency	500
# Epochs	10001
# Refinement Epochs	1001
Batch size	8000
Loss decoder weight	1
Loss SINDy (z) weight	0
Loss SINDy (x) weight	1e-4
Loss regularization weight	1e-5
Widths	[64, 32]

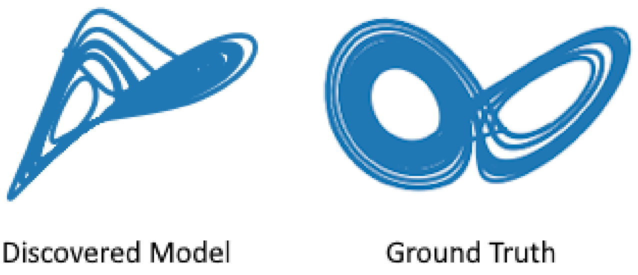

Figure 2. Results of SINDy autoencoder with standard hyperparameters.

Instead, the hyperparameters that were mainly used in this reproduction and are focused on in the hyperparameter study are mentioned in Table 2.

The results of model 1 with its additional hyperparameters shown in Figure 4 can be seen in Figure 3.

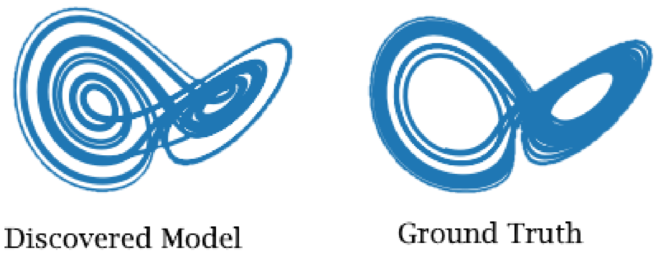

Figure 3. Results of SINDy autoencoder with standard hyperparameters.

The respective equations that were found were with this model were:

$$\dot{z}_1 = -6.90075z_1 + 8.877033z_2 \tag{4}$$

$$\dot{z}_2 = -8.19z_1z_3 - 0.212z_2z_3 - 0.21z_1z_3^2 \tag{5}$$

$$\dot{z}_3 = -6.187 - 1.56z_3 + 2.41z_1z_2 + 0.115z_2^2 \tag{6}$$

While the original equations were:

$$\dot{z}_1 = -10z_1 + 10z_2 \tag{7}$$

$$\dot{z}_2 = 28z_1 - z_2 - z_1z_3 \tag{8}$$

$$\dot{z}_3 = -2.7z_3 + z_1z_2 \tag{9}$$

The main difference in the values of the similar coefficients is an arbitrary scaling which can be solved by doing a transformation of the following sort:

$$z_1 \rightarrow a_1z_1 \tag{10}$$
$$z_2 \rightarrow a_2z_2 \tag{11}$$
$$z_3 \rightarrow a_3z_3 + b_3 \tag{12}$$

However there are some small higher order terms which affect the recreation of the original equations. These are the terms with $z_1z_3^3$ and z_2^2 which were not effectively eliminated by the sequential thresholding. This was also seen in one of the models that was not explicitly discussed in the original paper.

The reproduction included a hyperparameter study. This was done to see whether other hyperpameters might produce better results. The main hyperparameter that was tuned was the learning rate. Additionally, once, the threshold frequency was lowered to facilitate for the lower learning rate to not eliminate useful features too early. The general hyperparameters used for the hyperparameter study can be seen in Table 2. These were the same for all compared models. These hyperpameters were taken not from the additional paper [2] but were taken from the parameters in the GitHub. However, the general conclusions for the hyperparameter study are still useful to any future research, as it discovers this slightly different space of possible hyperparameters. The models that were compared and their respective hyperparameters can be seen in Table 3.

Table 2. General hyperparameters for the hyperparameter study.

Training samples	256000
# Epochs	5001
# Refinement Epochs	1001
Batch size	1024
Loss decoder weight	1
Loss SINDy (z) weight	0
Loss SINDy (x) weight	1e-4
Loss regularization weight	1e-5
Widths	[64, 32]

The validation losses were not significantly different for most models as can be seen in Figure 4. They seemed to follow the same trajectory. However, the model with the highest learning rate could not "settle" as low as the other models. Overall the models with the lowest learning rates seemed to have the least amount of "spikes" and irregularities. This might be due to the gradient descent steps that were in the "wrong" direction having more effect. However, the final validation loss of every model except for the highest one, were very close as seen in Table 3. In general when looking only at the losses, no significant distinction can be made between the models with the lower learning rates.

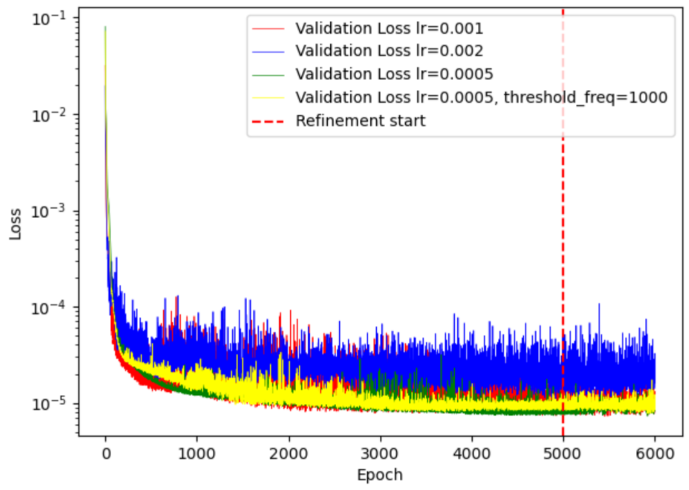

Figure 4. Validation losses for the different compared models

Table 3. Final validation losses of the different models

Model #	Learning rate	Threshold freq.	Final val. loss
1	1e-3	500	8.6273e-6
2	2e-3	500	1.6039e-5
3	5e-4	1000	8.5096e-6
4	5e-4	500	8.6705e-6

All models were able to discern the 2 lobe structure of the attractor. However some models included a lot of high order coefficients. This was particularly visible in model 2, where several high order coefficients were present. Model 1 was the most visually similar, but had several high order coefficients too much. Model 4 had several wrong coefficients as well. Model 3's SINDy coefficient matrix was the most similar to the ground truth's SINDy coefficient matrix.

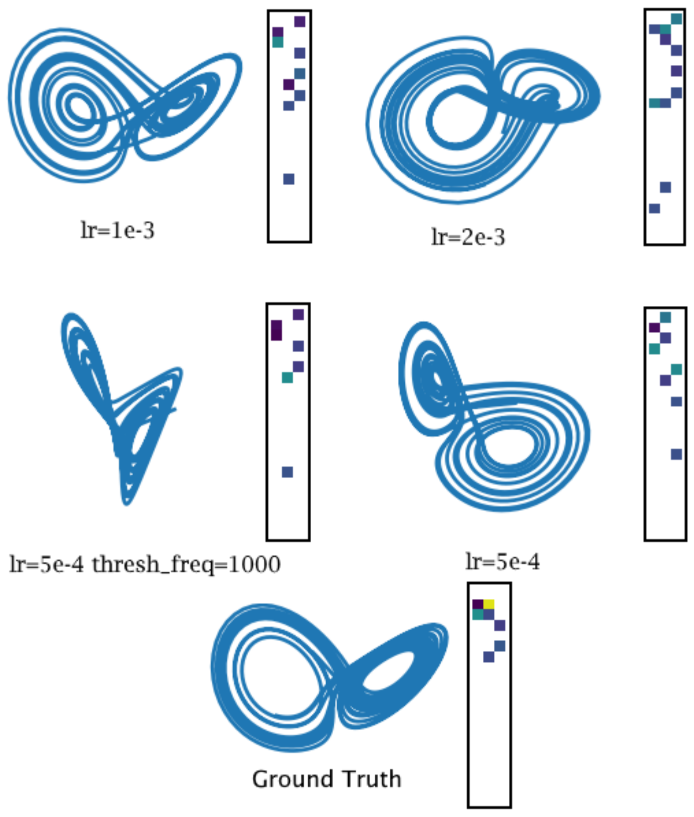

Figure 5. Attractor visualisation of the discovered models in the hyperparameter study.

Additionally, model 3 showed the following results. Remarkably, we can see that the discovered model looks very similar to that of the original paper [1], further validating our reproduction. After applying a transformation, we see that the discovered model looks similar to the ground truth again. However, similarly to previous results, one high order term causes a significant error between the discovered transformed model and the ground truth.

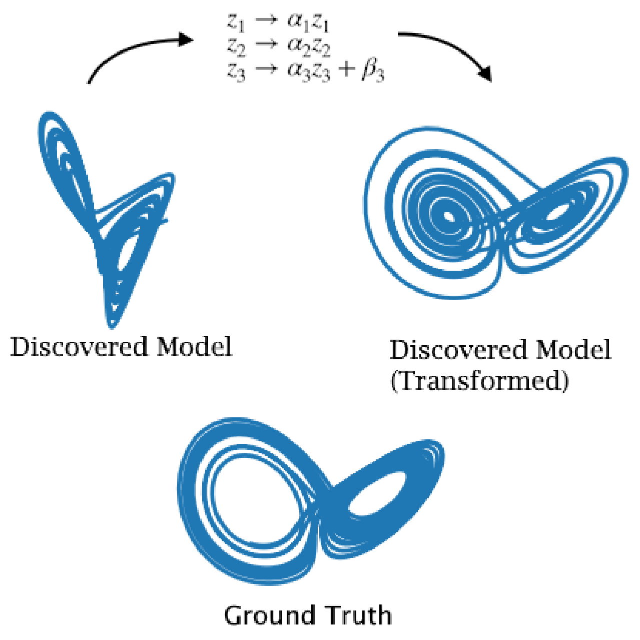
$z_1 \rightarrow \alpha_1 z_1$
$z_2 \rightarrow \alpha_2 z_2$
$z_3 \rightarrow \alpha_3 z_3 + \beta_3$
Discovered Model
Discovered Model (Transformed)
Ground Truth

Figure 6. Results for the compared model with a learning rate of 5e-4 and a threshold frequency of 1000.

5. Conclusion

Overall, the reproduction was quite successful. The reproduction showed that SINDy can reproduce state equations for nonlinear dynamic models as discussed in the paper. Reproduction like these are important to validate the results of papers discussing new methods like [1]. While the exact same results have not been reproduced, this is quite understandable. Deep learning has inherent randomness due to weight initialization. Furthermore, exact seeds could not be used to reproduce the results gathered in the original paper, as different frameworks were used. The hyperparameter study was quite limited in both the amount of models produced per setting and the amount of settings tested. More models should be generated from all different settings to get an accurate representation of how the settings exactly influence the results.

6. Contributions

Table 4. Contributions of group members

Member	Contributions
Jim van Oosten	Programming training loop, writing reproduction section in blog post, making poster.
Ian Maes	Setting up environment, programming autoencoder, bug fixes, training models, analysis of models, writing theory and results sections in blog post.
Kyle Scherpenzeel	Setting up environment, training models, bug fixes, analysis of models.
Matei Dinescu	writing reproduction section in blog post, making poster.

References

[1] K. Champion, B. Lusch, J. N. Kutz, and S. L. Brunton, "Data-driven discovery of coordinates and governing equations", *Proceedings of the National Academy of Sciences*, vol. 116, pp. 22 445–22 451, 45 2019. DOI: 10.1073/pnas.1906995116.

[2] K. Champion, B. Lusch, J. N. Kutz, and S. L. Brunton, "Data-driven discovery of coordinates and governing equations appendix", *Proceedings of the National Academy of Sciences*, vol. 116, no. 45, pp. 22 445–22 451, 2019, Supplementary Information.