Modelling from Measurements

Ludovica Illiano

Politecnico di Milano
Dipartimento di elettronica, informazione e bioingegneria
Piazza Leonardo da Vinci, 20133, Milano, Italy

ludovica.illiano@polimi.it
https://github.com/luv-lly/Modelling-from-Measurements

Abstract - **Data-driven algorithms provide a new approach to modeling complex systems without resorting to analytical derivations or empirical models. Some of these techniques are explored in depth in this report: after a brief theoretical introduction, the algorithms presented are implemented in the study of the dynamics of nonlinear systems. The algorithms covered are the Dynamic Mode Decomposition (DMD), the Sparse Identification of Nonlinear Dynamics (SINDy) and Neural Networks.**

Index Terms – Dynamic Mode Decomposition, Neural Network, Non-linear dynamical systems, Singular Value Decomposition.

I. INTRODUCTION AND OVERVIEW

A mathematical model can be defined as a mathematical representation of a real process or system: it uses a number of variables to represent inputs, outputs and internal states, and sets of equations and inequalities to describe their interaction. It is a fundamental commodity in the study of physical phenomena, since it contains information on the dynamic evolution of the system analysed. A model is generally represented by the governing equations, parsimonious equations that through few parameters allow the evaluation of the state of the system over time. Once the equations governing a process are identified, it becomes extremely straightforward to extrapolate new working conditions considering different initial conditions.

Today data-driven discovery is changing the approach implemented in the definition of model for complex systems. These complex systems are foremost non-linear, dynamical systems, and show space-time dependency. The aim of engineers is to define the dominant underlying pattens that characterize the system; this can be done by means of data-driven algorithms [1]. The growing interest in this type of algorithm is mainly due to the increasing availability of data at ephemeral costs and the improvement of computational performance. Whereas these resources seem endless, it is an engineer's responsibility to optimize their use.

In this discussion we will divide the data-driven techniques into 4 macro-sections in order to better schematize the concepts; each of these algorithms has its advantages and disadvantages, it is up to the operator to define which one to use depending on the goal to accomplish. The main algorithms that will be addressed are the Dynamic Mode Decomposition (DMD), introduced by a brief digression on the Singular Value Decomposition (SVD), the Sparse Identification of Nonlinear Dynamics (SINDy) and Neural Networks (NN).

In Section II a theoretical background of the aforementioned algorithms is presented, in Section III it is explained how the theoretical concept are implemented in solving some practical exercises. In Section IV the results of the computations, organized by exercise, are reported. And lastly, general conclusions and considerations are given in Section V.

The structure of the report follows lectures from "Modelling from Measurements" course, taught by Professor Nathan J. Kutz.

II. THEORETICAL BACKGROUND

In this Section a background for the algorithms that will be implemented in Sec. III is provided.

II.1 Singular Value Decomposition

The Singular Value Decomposition (SVD) is a robust algorithm used in this context to extract dominant patterns from high dimensional data set. It is a matrix factorization technique, that can be used for a variety of purposes and exists for every complex matrix, unlike eigendecomposition. The SVD is implemented to perform the pseudoinverse of a non-square matrix and to find the solution to the linear system $Ax = b$. In the specific application, given a large data set, its implementation permits to determine a low dimensional approximation in terms of dominant patterns. The algorithm is completely data-driven, as the pattern are evaluated considering only the set of data without possessing prior knowledge of the problem. The data are represented in terms of new coordinates system defined by dominant correlation with the data.

Generally, the data set X ∈ $\mathbb{C}^{nxm}$ is defined as

$$\mathbf{X} = \begin{bmatrix} | & | & | & | \\ \mathrm{x}(t_1) & \mathrm{x}(t_2) & \cdots & \mathrm{x}(t_m) \\ | & | & | & | \end{bmatrix} \quad \text{(II.1)}$$

Where the rows of (II.1) consists of time-series data and therefore the column vectors may represent the state of a physical system that is evolving in time.

The SVD matrix decomposition of the data set X is defined as

$$\mathbf{X} = \mathbf{U\Sigma V}^* \quad \text{(II.2)}$$

U and V are unitary matrix with orthonormal columns, and Sigma is a matrix with all positive entries on diagonal and zeros off diagonal.

One of the most helpful implementations of the SVD algorithm is the low rank approximation of the data matrix.

$$\mathbf{X_r} = \mathbf{U_r \Sigma_r V_r^*} \quad \text{(II.3)}$$

Truncation of the X-matrix at rank r is a powerful tool for processing data sets that contain high-dimensional measurements: it allows to express them through the dominant patterns via projection in a reduced dimensional space. The few dominant patterns which describes the system behavior are given by the columns of V_r and U_r.

II.2 Dynamic Mode Decomposition

The Dynamic Mode Decomposition (DMD) combines the low dimensional approximation defined through the SVD with the analysis in time domain of the modal variation. In this sense DMD provides a modal decomposition of the system and also delivers information on how these modes evolve in time. It implements SVD for spatial dimension reduction and the FFT for temporal frequency identification. In particular the DMD may be defined as an algorithm which allows the identification of the best fit linear dynamical system that follows the high dimensional measurements in time.

The data set represented in (II.1) is paired with its evolution in time, defined as:

$$\dot{\mathbf{X}} = \begin{bmatrix} | & | & | & | \\ \mathrm{x}(t_1') & \mathrm{x}(t_2') & \cdots & \mathrm{x}(t_m') \\ | & | & | & | \end{bmatrix} \quad \text{(II.4)}$$

The purpose of this decomposition is to return the eigenvalues and eigenvectors of the matrix A that best approximates the solution of the linear system in (II.5).

$$\dot{\mathbf{X}} = \mathbf{AX} \quad \text{(II.5)}$$

By solving a straightforward optimization problem, the computation of the matrix A is reduced to the equation (II.6).

$$\mathbf{A} = \dot{\mathbf{X}}\, \mathbf{V_r}\, \mathbf{\Sigma_r^{-1}}\, \mathbf{U_r^*} \quad \text{(II.6)}$$

As already described DMD combines with factorization by SVD of the A matrix to disclose the dominant modes, and thus dominant eigenvalues and eigenvectors, of the dynamical system. What is therefore of interest is not the matrix A, but the matrix reduced to a tilde, obtained by projecting A onto the POD modes in U.

$$\widetilde{\mathbf{A}} = \mathbf{U_r^*}\, \mathbf{A}\, \mathbf{U_r} = \mathbf{U}^*\, \dot{\mathbf{X}}\, \mathbf{V_r}\, \mathbf{\Sigma_r^{-1}} \quad \text{(II.7)}$$

The spectral decomposition of $\widetilde{\mathbf{A}}$ yields its eigenvalues and eigenvectors.

$$\widetilde{\mathbf{A}}\mathrm{W} = \mathrm{W}\Lambda \quad \text{(II.8)}$$

Employing the eigenvectors W thus calculated, the high dimensional DMD modes are reconstructed.

$$\mathbf{\Phi} = \dot{\mathbf{X}}\, \mathbf{V_r}\, \mathbf{\Sigma_r^{-1}}\, \mathbf{W} \quad \text{(II.9)}$$

At this point the states of the system can be derived from the spectral decomposition:

$$x_j = \sum_{j=1}^{r} b_j \Phi_j e^{\lambda_j t} \quad \text{(II.10)}$$

Therefore each DMD mode is associated with an eigenvalue consisting of frequency of oscillation and growth decay. DMD in particular are useful when dealing with dynamical systems, due to the connection of the mode to a frequency of oscillation.

Several DMD algorithms have been proposed, in particular three of them will be presented hereafter.

Exact DMD

The Exact DMD algorithm solves an optimization problem described as:

$$\min_{A} \|\dot{\mathbf{X}} - \mathbf{AX}\| \tag{II.11}$$

This optimization is really sensitive to noises and returns poor eigenvalues. It is usually preferable to implement a different algorithm, the opt-DMD.

Optimized DMD

Another way to solve the optimization problem introduced by (II.11) is to fit directly the solution obtained from the data. This is a non-linear optimization and is implemented by means of the variable projections.

$$\min_{\boldsymbol{b_j \Phi_j \lambda_j}} \left\| \dot{\mathbf{X}} - \sum_{\boldsymbol{j=1}}^{\boldsymbol{r}} \boldsymbol{b_j \Phi_j e^{\lambda_j t}} \right\| \tag{II.12}$$

Even if the optimized DMD performs better than the Exact DMD and is insensitive to noises, the non-linear optimization may fail to converge if a good initial guess is not provided. Although this algorithm can be used for system diagnostics, it is not the best tool for obtaining a forecast of system behavior.

BOP - DMD

A variance of the previous algorithm is the Bagging Optimized DMD (BOP-DMD), that accepts as input noisy data and returns as output a probabilistic forecast combined with the uncertainty quantification. The statistic approach gives robustness to the result.

In this case p snapshots in time are picked randomly from the initial data set. The opt-DMD algorithm is applied at each new sub set of data and the iteration is repeated. The result is the probability distribution of the coefficients, the eigenvectors and eigenvalues.

To evidence, the DMD mode may be applied to linear and non-linear dynamical systems, and it transform every kind of system into a linear one. Its result is a linear representation of the system through mode propagating in time.

II.3 Sparse Identification of Nonlinear Dynamics

Linear systems bring advantages: they are solvable in closed form, and there is extensive theory in literature regarding their control and manipulation. But they have also limitations; in particular the only solution of a linear system could neglect significant dynamics of the system.

The Sparse Identification of Nonlinear Dynamics (SINDy) allows to generate non-linear dynamics based on the concept of dominant balanced physics. Indeed the dynamic of many dynamical systems may be approximated considering only few active terms in the space of possible right hand side functions. The generalized linear model that allows to approximate the generic behavior described by f is defined as

$$\mathbf{f}(\mathbf{X}) = \sum_{\boldsymbol{k=1}}^{\boldsymbol{p}} \theta_k(\mathrm{x})\xi_k = \boldsymbol{\Theta}(\mathbf{X})\boldsymbol{\xi} \tag{II.13}$$

Defined the data set and its derivative as in (II.1) and (II.4) it is necessary to construct a library of candidate non-linear functions.

$$\boldsymbol{\Theta}(\mathbf{X}) = \begin{bmatrix} \mathbf{1} & \mathbf{X} & \mathbf{X^2} & \cdots & \sin(\mathbf{X}) & \cdots \end{bmatrix} \tag{II.14}$$

Dynamic system may be represented through the data matrices using the following relation

$$\dot{\mathbf{X}} = \boldsymbol{\Theta}(\mathbf{X})\boldsymbol{\Xi} \tag{II.15}$$

Each column of Xi is a vector of coefficients determining the active terms in the kth row of the functions. Through an optimization function it is possible to promote sparsity of the solution and obtain a parsimonious model. The model is identified by means of a sparse regression regularized by the L1 norm. λ is the sparsity-promoting knob.

$$\boldsymbol{\xi_k} = \underset{\boldsymbol{\xi_{k'}}}{\mathrm{argmin}} \left\| \dot{\mathbf{X}}_{\mathbf{k}} - \boldsymbol{\Theta}(\mathbf{X})\boldsymbol{\xi'_k} \right\|_2 + \boldsymbol{\lambda}\|\boldsymbol{\xi'_k}\|_{\mathbf{1}} \tag{II.16}$$

II.4 Machine learning

Machine learning consists of a set of techniques for solving optimization problems. The function to be optimized is defined objective function.

$$\underset{\boldsymbol{A_j}}{\mathrm{argmin}}\ \boldsymbol{f_M}(\boldsymbol{A_M}, \cdots, \boldsymbol{f_2}(\boldsymbol{A_2}, \boldsymbol{f_1}(\boldsymbol{A_1}, \boldsymbol{x}))\cdots) + \boldsymbol{\lambda g}(\boldsymbol{A_j}) \tag{II.17}$$

A neural network is so defined because it recalls the brain structure: it can consist of multiple layers, and each layer comprises multiple nodes closely interconnected with all the nodes in the next layer. A-matrices represent a transformation between two consecutive layers: in them the weights of each node affecting the single node of the consecutive layer are displayed. The mapping outlined by the f(A,x) functions is usually a non-linear mapping that enriches the model defined by the neural network, making it flexible; the f(.) function is called activation function. This system as defined is massively underdetermined: the term g(Aj) is added for regularization purposes. The definition of the cost function, and therefore of the regularization terms, depends on the application.

The optimization problem described in (II.17) is solved using algorithms such as stochastic gradient descent and back propagation. It must be considered that the objective function to be optimized is not the desired objective function, but a proxy for it. The aim of the backpropagation algorithm is to minimize the error produced by the NN with respect to the ground truth. Via the derivative chain rule, it is possible to backpropagate the error through the entire NN. The weights to be applied in each individual transformation are then iterated according to the formula (II.18).

$$a_{k+1} = a_k + \delta \frac{\partial E}{\partial a_k} \tag{II.18}$$

Where a_k is the weighting constant and δ is defined learning rate. One algorithm that provides a faster way to estimate weights is the stochastic gradient descent. Error minimization between network solution and data in this case is obtained by setting the partial derivative with respect to each matrix component to zero, resulting in the gradient of the function with respect to the NN parameters.

$$x_{j+1}(\delta) = x_j - \delta \nabla f(x_j) \tag{II.19}$$

A final feature of neural networks that is highlighted in this section is that neural networks are well suited to solving interpolation problems, whereas they do not perform as well in solving extrapolation problems.

III. Algorithm Implementation and Development

The following describes the implementation of the techniques discussed in Sec. II to solve three problems, which are the subject of the article. All results will then be reported in Sec. IV.

III.1 Problem #1

Referring to the historical data set composed by the variation of two populations, composed by hares and lynxes, over 60 years, it is required to predict the future state of these populations implementing some of the techniques explained in Sec. II.

The first point requires the implementation of the DMD algorithm, with and without bootstrap aggregation, to derive a dynamical model of the system. The procedures deployed for the first request are described in Algorithm 1 [1] and Algorithm 2 [2]. Algorithm 1 in particular is a compact schematic representation of the theory developed in Sec. II.

The BOP-DMD is expected to complete the model prediction by providing a statistical estimation of the forecast. In order to limit the diverging behavior of the solution, eigenvalues with real part greater than zero have been neglected in the computation of the final solution.

Algorithm 1 opt -DMD

```
Input : $(\mathbf{X}, \dot{\mathbf{X}}, \mathbf{t}, \mathbf{r})$

$[\mathbf{U}, \mathbf{S}, \mathbf{V}]$ = svd( X , 'econ')
Truncate U, S, V to rank r → Ur, Sr, Vr
$\tilde{\mathbf{A}}$ = U(:,1:r)* $\dot{\mathbf{X}}$ V(:,1:r) / S(1:r,1:r)

Solve eigenvalues problem for $\tilde{\mathbf{A}}$
$[\mathbf{W}, \mathbf{\Lambda}]$ =eig ($\tilde{\mathbf{A}}$)
$\mathbf{\Phi} = \dot{\mathbf{X}} \mathbf{V}_r \mathbf{\Sigma}_r^{-1} \mathbf{W}$

Projection onto the features space
$\boldsymbol{\alpha}_1 = \mathbf{\Sigma}_r \mathbf{V}_r^*(\mathbf{1},:)$

Amplitude of each mode
$\mathbf{b} = \mathbf{W} \mathbf{\Lambda} \boldsymbol{\alpha}_1$

Solution in continuous time introducing the continuous eigenvalues : $\boldsymbol{\omega} = \log \boldsymbol{\lambda} / \boldsymbol{\Delta t}$
$\mathbf{X} = \mathbf{\Phi} \exp(\mathbf{\Omega t}) \mathbf{b}$
```

Algorithm 2 BOP-DMD

```
Input : (X, p, K)

For K iterations
    Selecting p snapshots < length (X)
    $[\mathbf{\Phi}, \mathbf{\Omega}, \mathbf{b}]$ = opt-DMD (X)        compute K DMD models
    Filter Re{ $\mathbf{\Omega}$ }> threshold

$\mathbf{\Phi}_m$ = mean($\mathbf{\Phi}$)
$\mathbf{X}(t_{forecast}) = \mathbf{\Phi}_m \exp(\mathbf{\Omega}\, \mathbf{t}_{\mathbf{forecast}}) \mathbf{b}$

Compute
X mean
X variance
```

The second part of the first problem requires pre-processing the data set by means of time delay embedding. Time delay embedding is a technique that allows to increase the dimensional space of data: it consists into stacking delayed version of the original data set one on top of the other to form an Hankel matrix. As discussed in the previous section, DMD pursues a linear transformation that allows the dynamic behavior of the system to be related to the linear combination of the states vector. Since a complete measure of the states of the system is not available, writing a Hankel matrices is an artificial way of increasing the dimensional space of the data set and investigate the presence of latent variables.

```
Algorithm 3 TDE-DMD
Input : (X, t, r)

For K iterations
$\mathbf{X} = [\mathbf{X}(1:k-1)$                    Build the Hankel Matrix
      $\mathbf{X}(2:k)]$

Select the rank r
$[\boldsymbol{\Phi}, \boldsymbol{\Omega}, \mathbf{b}] = \text{opt-DMD}(\mathbf{X}, \mathbf{r})$

$\mathbf{X} = \boldsymbol{\Phi} \exp(\boldsymbol{\Omega} \mathbf{t}) \mathbf{b}$
```

The third part requires to optimize the parameters of the Lotka-Volterra model using the available data. The Lotka-Volterra model is a mathematical model that allows to evaluate the behavior of a biological isolated system in which two species interact: one represents the prey and the other the predator.

$$\begin{aligned} \dot{x} &= (b - py)x \\ \dot{y} &= (rx - d)y \end{aligned} \tag{III.1}$$

To find the values of b, p, r and d of the Lotka-Volterra equations that best fit the data set behavior, an optimization problem has been developed as follows:

$$\underset{\substack{b,p,d,r,\\ H(t0),L(t0)}}{\operatorname{argmin}} \sqrt{\sum_{i=1}^{N} \left(H_{data}(i) - H(i)\right)^2 + \left(L_{data}(i) - L(i)\right)^2} \tag{III.2}$$

Boundary conditions for the parameters have been imposed.

$$\begin{aligned} &b, p, r, d > 0; \\ &H(i) \in [lb; ub] \\ &L(i) \in [lb; ub] \end{aligned} \tag{III.3}$$

The fourth request was solved by means of implementation of the Ensemble SINDy algorithm; the system dynamic is approximated through sparse identification of the best fit nonlinear model. For better results the data set was firstly pre-processed by applying cubic interpolation and normalized to the maximum value. Afterwards the library of candidate functions, expanding up to the second order polynomials, has been generated. The bagging was firstly applied to the library just described, then directly to the coefficients to improve the performance of the model reconstruction, as suggested in [3].

To post-process the results the inclusion probability (i.p.) computed through the algorithm, which gives a measure of how many times a coefficient is not null in the space of the iterations, was used to remove the coefficients with an i.p. lower than a given threshold. Coefficients whose variance exceeded a defined limit were also excluded from the final set of coefficients. This set of coefficients was then averaged and employed to compute the final model, as described in (III.4).

$$\dot{\mathbf{X}} = \boldsymbol{\Theta}(\mathbf{X})\langle \boldsymbol{\Xi} \rangle \tag{III.4}$$

III.2 Problem #2

In the second problem, it is required to train two neural networks to advance the solution in time of two systems of equations, the Kuramoto-Sivanshinsky (KS) equation and the reaction-diffusion equations.

The Kuramoto-Sivanshinsky equation describes spatiotemporal chaotic behaviors. It is a fourth order non-linear partial differential equation.

$$u_t + u_{xx} + u_{xxxx} + \frac{1}{2} u_x^2 = 0 \tag{III.5}$$

To answer the first request a NN which predicts the solution of the equation at a future time instant has been trained. The NN was trained implementing the data set generated from the KS equation solution, considering 100 randomly varying initial conditions. The solution of the KS equation returns a data set composed of 61 time steps and 4 spatial states. The u(x,t) values obtained at each iteration were embedded into a single vector for each state in order to expand the training set. The final dimension of input and output matrices was equal to $[N_{states}\ ,\ N_{time-1} * N_{iterations}]$.

Table 1: KS-NN Features

Input features	Output features
$u_s(k)$ $for\ s \in [1: N_{states}]$	$u_s(k+1)$ $for\ s \in [1: N_{states}]$

The resulting neural network was then tested using a new set of initial conditions, again randomly selected. Prediction was obtained by implementing recursion of the neural network output. For each time instant the output of the network was used as the input of the network for prediction at the successive time frame.

The third point requires checking how the NN prediction of the future state of reaction-diffusion equations behaves in a lower dimensional space considering low rank variables.
The data set was generated starting from the solution of the ODE system: it is composed of 200 snapshot in time of a 512 by 512 matrix, which represents the spatial discretization of the reaction-diffusion system.

To evaluate the performance of the NN a subset of 90% of the data frames was selected to train the NN, and the NN was tested on the other 10%. Since, considering the resources available, it was not possible to train a NN by implementing even a smaller fraction of the data set, a dimensional space reduction was performed through the implementation of the SVD factorization to the flattered state matrix. The rank for the reduction was selected by looking at the energy captured by each mode. In this way the original spatial dimension has been drastically reduced thus speeding up the training phase.

As done for the previous NN the state were embedded in one single vector. Then the NN has been trained with the reduced space vector obtained solving for the reduced states vector, indicated as NN_{input}, the following linear system:

$$\mathrm{U}_{\mathrm{trunc}} \cdot \mathrm{NN}_{\mathrm{input}} = u_{flat} \qquad \text{(III.6)}$$

The system is solved for each time frame.

III.3 Problem #3

The third and final exercise requires to implement a neural network to advance the solution of the Lorenz equations in time, training it considering three different values for the Rayleigh constant, then testing how the network behaves with different Rayleigh constants as input.

The Lorenz equations are described by the system:

$$\begin{aligned} \dot{x} &= \sigma(y - x) \\ \dot{y} &= x(\rho - z) - y \\ \dot{z} &= xy - \beta z \end{aligned} \qquad \text{(III.7)}$$

The NN has been trained using a data set derived by the solutions of Lorenz equations considering different random initial conditions; to improve the performance of the neural network, the product between the coordinate x and the Rayleigh constant was added to the features space.

The representation of the NN features in discrete domain is reported in Table 2.

Table 2: Lorenz NN Features

Input features	Output features
$x(k)$	$x(k+1)$
$y(k)$	$y(k+1)$
$z(k)$	$z(k+1)$
$\rho \cdot x(k)$	$\frac{1}{x(k)}\left(\frac{y(k+1) - y(k)}{\Delta t} + y(k)\right) + z(k)$

As in the second problem, the final prediction was obtained by recursive forecast over the time span.

IV. Computational Results

This section presents the results obtained by applying the algorithms described in Sec. III. The results are briefly commented to highlight problems and improvements of the algorithms.

IV.1 Problem #1

Following the procedure described in Sec. II, the DMD algorithms have been used to derive a dynamic model from the data set. In particular, the DMD algorithms were applied to only 80% of the data set, so as to test the performance of the model on the remaining 20%. The data set is poor and noisy, so applying the opt-DMD algorithm to the data set does not give a fair representation of the system dynamics.

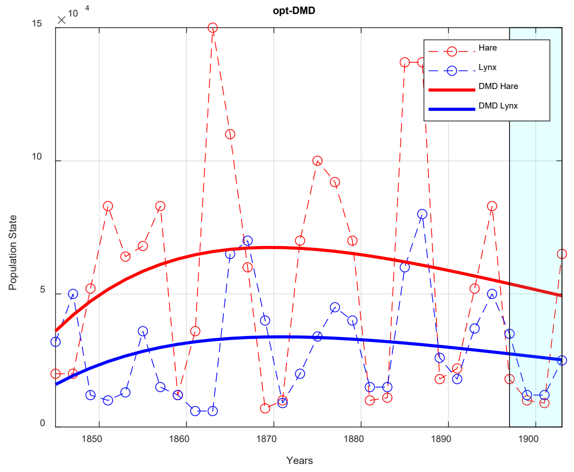

Figure 1: opt-DMD approximation of the model. Such representation is not adequate to represent the intrinsic dynamics of the system.

As shown in Figure 3, a dramatic improvement in model reproduction occurs by bootstrapping the data set. Indeed the BOP-DMD algorithm enhances robustness of the DMD method by removing noise bias. It also provides uncertainty quantification of the linear model.

Further improvement of the model is accomplished by pre-processing the data set by means of the time-delay embedding method. By increasing the dimension of the data set, it is possible in this case to select a different rank truncation. Considering the percentage of variance of each mode, shown in Figure 2, it is decided to truncate the rank at 5.

Table 3: Comparison of forecast results (t= 1899)

	DMD	TDE-DMD	BOP-DMD
Hare	52.3e3	122.4e3	/
Lynx	26.7e3	34.4e3	/
Mean(Hare)	/	/	102.6e3
Var(Hare)	/	/	4.9e9
Mean(Lynx)	/	/	12.9e3
Var(Lynx)	/	/	9.1e7

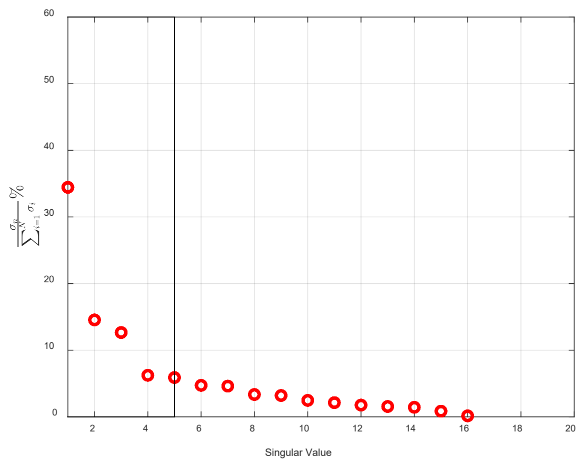

Figure 2: Percentage of variance of each mode implementing TDE-DMD Algorithm.

Observing Figure 4, it is possible to state that the more the rank selected for truncation is increased, therefore the more latent variables are introduced into the model, the better the DMD expansion approximates the dynamic evolution of the system.

So far only data driven methods have been implemented. Next, the results obtained using the Lotka-Volterra equations are considered. The optimization problem has been developed on the entire data set. Parameters that allow the equations to better approximate the data available are reported in Table 4.

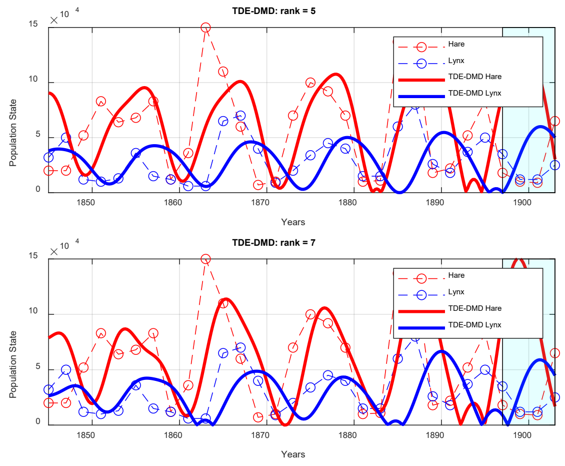

Figure 4: Comparison between TDE-DMD performances, considering different values for the rank.

Table 4: LV equations optimal parameters

‘b’	‘p’	‘r’	‘d’	‘H(t$_0$)opt’	‘L(t$_0$)opt’
1.5652	0.0722	0.0198	0.9987	23’310	36’462

Except for the delayed picks of the predator curve with respect to the prey one, the model is completely unable to represent the system dynamics. Indeed, the model as structured does not display the higher-order harmonics existing in the original data set.

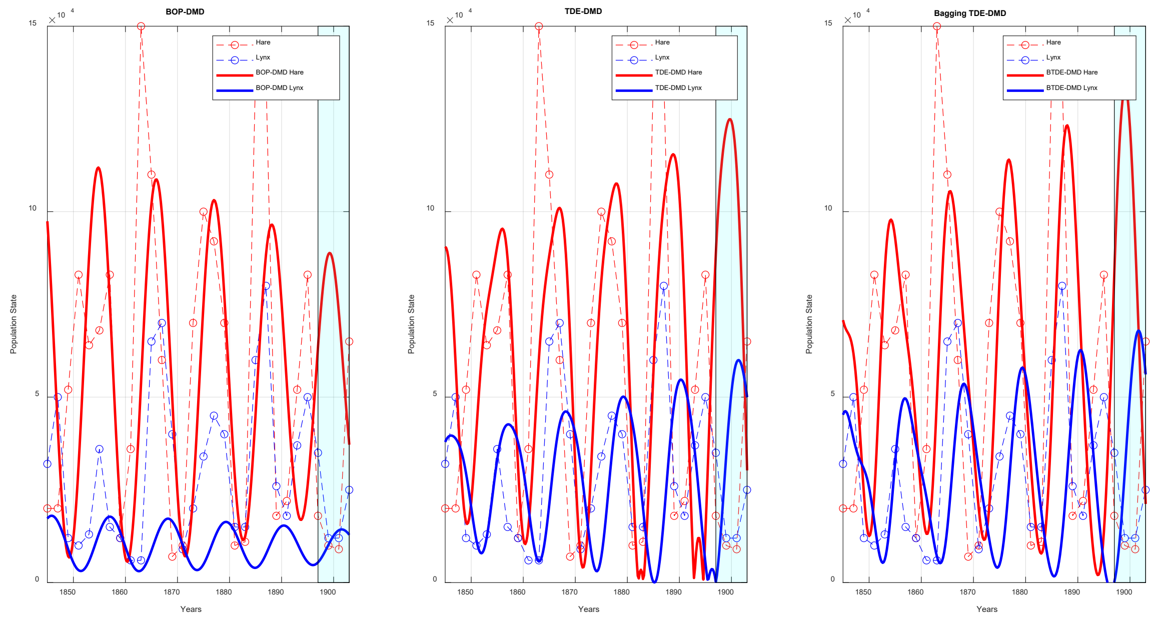

Figure 3: Comparison between BOP-DMD (left), Time Delay Embedding DMD (center) and Bagging TDE-DMD (right). TDE-DMD and BTDE-DMD results were obtained by considering the rank =5.

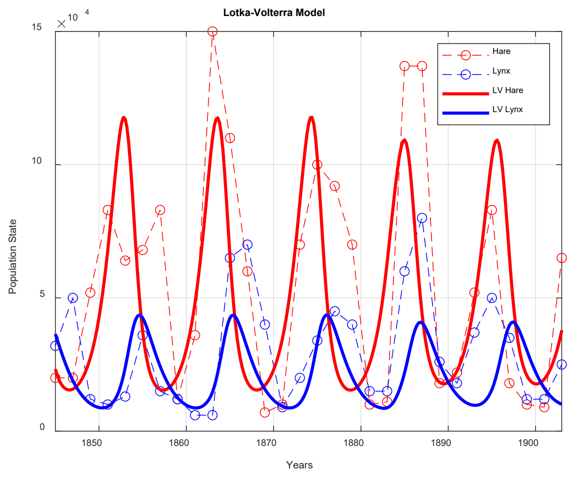

Figure 5: Lotka-Volterra model solution considering parameters reported in Table 4.

Last point to address is the development of a SINDy model. This model was developed by acting mainly on four parameters: the maximum order of the polynomials that compose the library, the number of iterations of bagging, the weight applied to sequentially thresholded least-squares (STLS), which is the sparsification knob, and the threshold for the inclusion probability.

Table 5 shows the values of the coefficients for the functions in the library that govern the dynamics of the system.

Table 5: SINDy coefficients

	1	x	y	xx	xy	yy
Hare x(t)	0.147	-0.869	0.115	0	-0.158	0
Lynx y(t)	-0.0028	-0.544	0.112	0	0.788	0

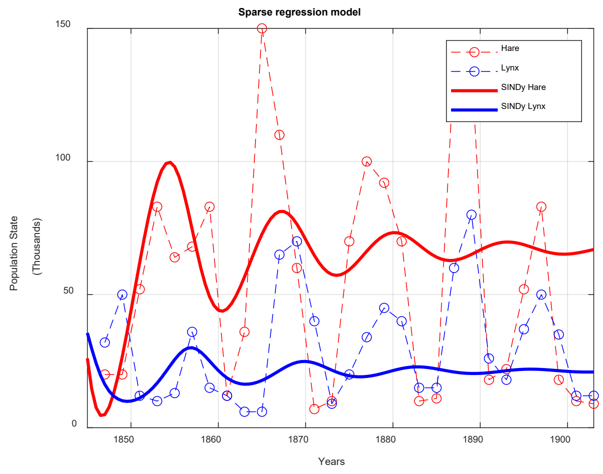

Figure 6: Reconstruction of the dynamics with SINDy algorithm, considering the coefficients reported in Table 5.

Again, the generated model is not suitable for describing the complete dynamics of the system; it follows the first few oscillations fairly well and then dies out to a mean value relative to the first few oscillations. Also in this case, the model is able to predict the delay between the two states of the populations.

IV.2 Problem #2

The behavior of the KS equation has been analyzed and reproduced using a feed forward neural network composed of 3 hidden layers of 10 neurons each.

The neural network was trained considering 100 random initial conditions; the spatiotemporal solutions obtained by solving the KS equation for each initial condition are then concatenated with respect to the 4 states.

The neural network thus defined shows good performance on the data set used for training, as can be observed in Figure 7.

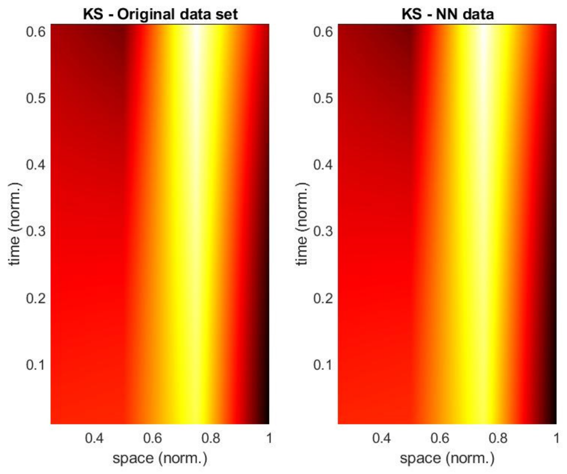

Figure 7: NN prediction performance on the training set.

Even changing the initial conditions, the neural network still returns a good approximation of the solution of the KS equation.

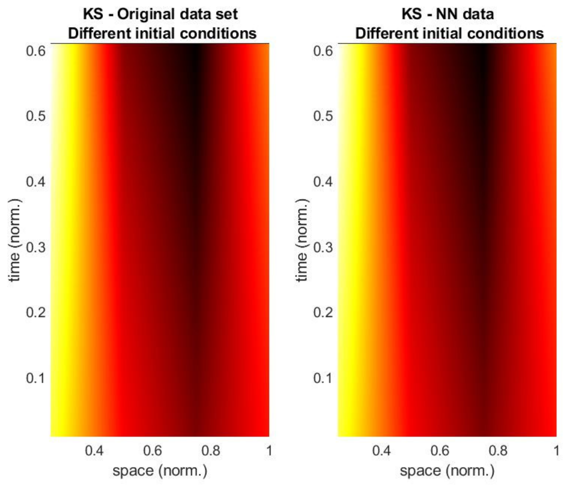

Figure 8: NN prediction performance in test phase, considering different initial conditions.

Next, the outcomes generated using the neural network that reproduces in a lower-dimensional space the behavior of the reaction-diffusion equations are analyzed.

As discussed, the states matrix is reshaped into a states vector to which the SVD factorization is applied. Looking at Figure 9 that shows the percent variance associated with each mode, and thus gives an indication of the energy contained in each mode, a fair approximation is accomplished by truncating the matrix to rank 10.

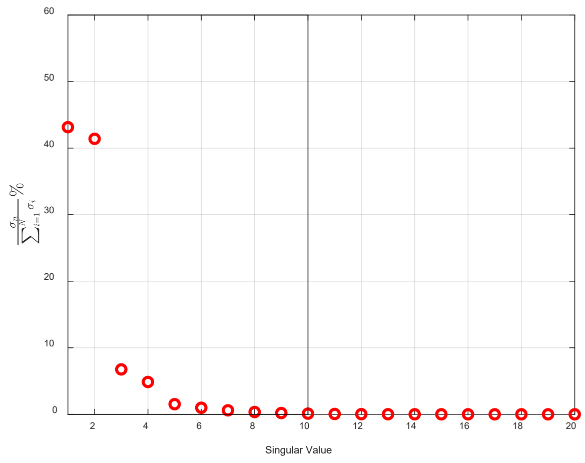

Figure 9: Percentage of variance of each mode, enforcing SVD truncation on the data set.

The neural network in this case is a fit net consisting of 3 layers of 10, 7 and 5 units, which uses Levenberg-Marquardt (LM) as training algorithm. The network is then trained on a subset of the available set of time frames so that the performance of the network can be tested on the residual part of the data set. The training set consists of 90% of the data set while the test set consists of the remaining 10%.

It is possible to state that the network, as described, is capable to correctly capture the behavior of the system.

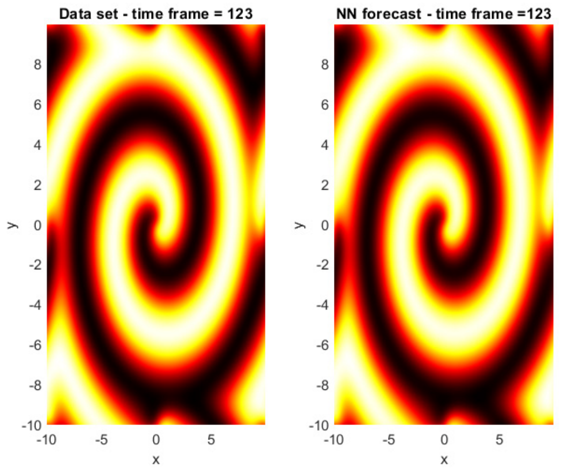

Figure 10: NN prediction performance considering a time frame belonging to the training set.

Even considering a time frame outside the training set, the neural network performs reasonably well. In the latter case, however, the error begins to be clearly noticeable; the situation worsens as one moves away from the training set.

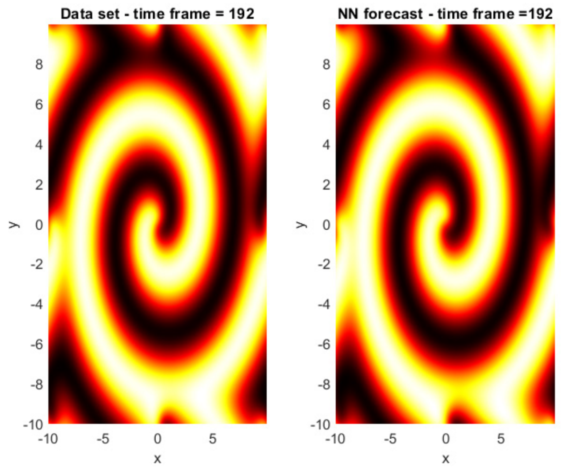

Figure 11: NN prediction performance considering a time frame outside the training set.

IV.3 Problem #3

The neural network in this example consists of only two hidden layers of 10 neurons each; the training algorithm is the Levenberg-Marquardt (LM). The network was trained using the solutions of Lorenz equations for 3 values of the Rayleigh constant as the input. The outputs are the solutions at the next time step. The NN was tested using as input Lorenz equations solutions considering different Rayleigh constant values. The development follows the procedure described in Sec. III.

Developing the neural network as described shows that it returns an optimal system model when ρ value can be derived by interpolating the values of ρ in the set used for training. It is also clear from Figure 12 that the neural network is not equally efficient in solving extrapolation problems. Better results for extrapolation could be achieved by increasing the number of iterations of the initial condition, thus increasing the number of solutions with which to train the neural network.

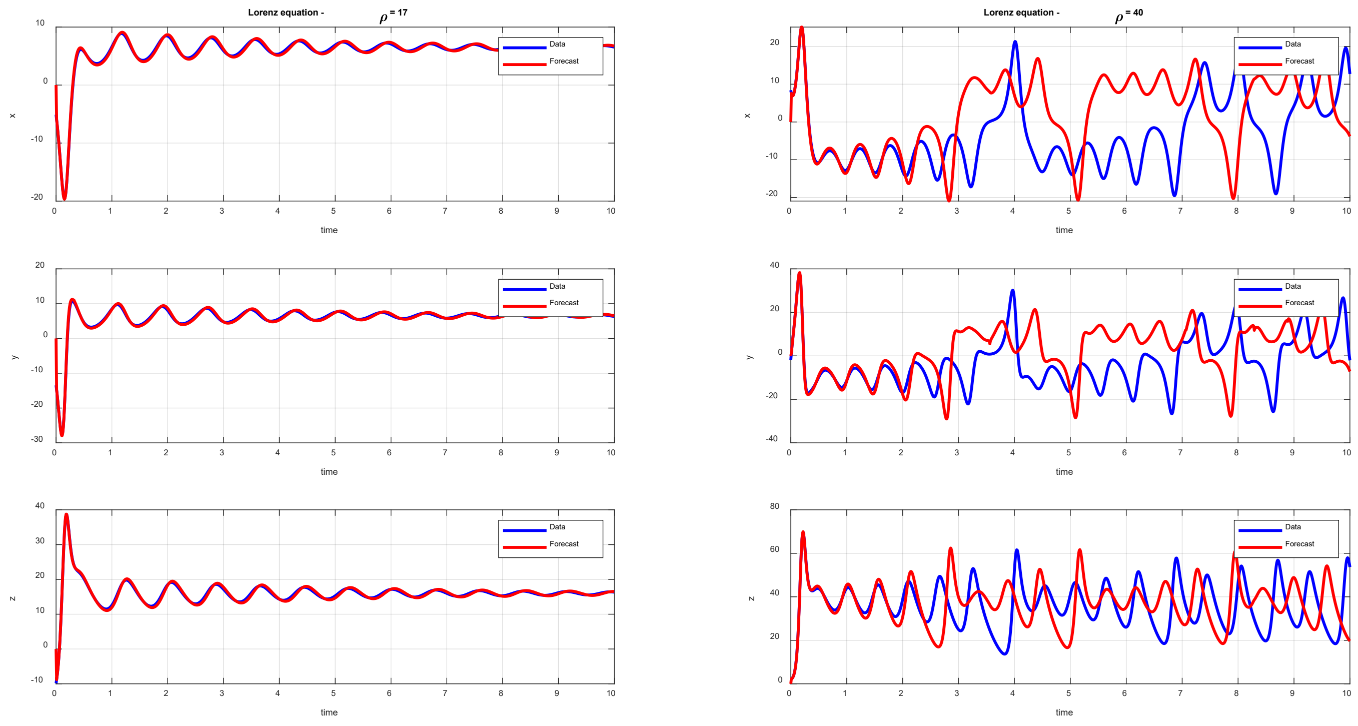

Figure 12: NN performances considering values of ρ outside the training set: on the left, the interpolation solution, on the right, the extrapolation solution.

V. Summary and Conclusions

In this report, algorithms for solving non-linear dynamical systems were outlined and applied to several case studies.

The performance of these techniques is closely related to the size and complexity of the data set. Techniques such as DMD and SINDy make it possible to change algorithm hyperparameters on the fly and quickly evaluate model performance as those parameters change. As the size of the data set increases, however, such techniques fail to achieve the level of accuracy reached by neural networks. In contrast, one must meticulously keep track of the hyperparameters tuning to determine the performance of the neural network itself, since training a neural network could take a long time. Therefore it is up to the engineer to assess what is the best technique to apply given the size of the data set and the accuracy with which a dynamic model is to be determined.

References

[1] Brunton, S. L., & Kutz, J. N. (2022). Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control. Cambridge University Press.

[2] Sashidhar, D., & Kutz, J. N. (2022). Bagging, optimized dynamic mode decomposition for robust, stable forecasting with spatial and temporal uncertainty quantification. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences, 380(2229). https://doi.org/10.1098/rsta.2021.0199

[3] Fasel, U., Kutz, J. N., Brunton, B. W., & Brunton, S. L. (2022). Ensemble-sindy: Robust sparse model discovery in the low-data, high-noise limit, with active learning and Control. Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences, 478(2260). https://doi.org/10.1098/rspa.2021.0904

[4] Dylewsky, D., Kaiser, E., Brunton, S. L., Kutz, J. N. (2022). Principal component trajectories for modeling spectrally continuous dynamics as forced Linear Systems. Physical Review E, 105(1). https://doi.org/10.1103/physreve.105.015312

Appendix A

Ex.1.1 opt-DMD

```
%% opt-DMD
clear all; close all; clc

%Datasets

load HL_population.mat

train = 0.8*length(HL);
HL_train = HL(:,1:train);
t_train = time(1:train);

[w,e,b] = optdmd(HL_train,t_train,2,1);

X = w*diag(b)*exp(e*time);

figure(1)

h = plot (years,H,'r--o',years,L,'b--o',years, abs(X(1,:)),'r',years, abs(X(2,:)),'b');
hold on
fill([years(end)-6 years(end) years(end) years(end)-6],[0 0 max(H)
max(H)],'c','FaceAlpha',0.1,'LineStyle','-')
grid on
xlim([1845 1903])
ylabel ('Population State');
xlabel ('Years');
title ('opt-DMD');
set(h,{'LineWidth'},{0.5;0.5;2;2})
legend ('Hare','Lynx','DMD Hare','DMD Lynx')

% BOP-DMD Iterations

for i=1:Niter

    ind = randperm(Imax, Nsubset);
    ind = sort(ind,2,"ascend");
    temp(:,:) = HL(:,ind);
    time_b = time(ind);
    [w1,e1,b1] = optdmd(temp,time_b,2,1,[],e1);  %Iterations of the BOP-DMD
    wx(:,:,i)=w1;
    ex(:,:,i)=e1;
    bx(:,:,i)=b1;

%     figure(2)
%     plot (real(ex(:,:,i)),imag(ex(:,:,i)),'o')
%     hold on
end

clear temp

% Constraints
[~,c1] = find( abs(real(ex(1,:,:)))>0.03 );
```

```
[~,c2] = find( abs(real(ex(2,:,:)))>0.03 );
c = union(c1, c2);

ex(:,:,c)=[];
bx(:,:,c)=[];
wx(:,:,c)=[];

% Mean Value and variance of W

Wmean = [ mean(wx(1,1,:),3), mean(wx(1,2,:),3);mean(wx(2,1,:),3), mean(wx(2,2,:),3)];

clear i

time_prevision = [0:0.1:30];

for i=1:length(ex)
    l_prev(:,:,i) = exp(ex(:,:,i).*time_prevision);
    x_prevision(:,:,i) = Wmean(:,:)*l_prev(:,:,i).*bx(:,:,i);
end

X_prev_mean = [mean(x_prevision(1,:,:),3); mean(x_prevision(2,:,:),3)];
X_prev_varn = [var(x_prevision(1,:));var(x_prevision(2,:))];

H_prev(:,:) = X_prev_mean(1,:,:);
L_prev(:,:) = X_prev_mean(2,:,:);
```

Ex.1.2 TDE-DMD

```
%% Time Delay Embedding

% Hankel Matrix
p = floor(0.7*train);   % New time interval
% m = length(HL)-p+1;     % Double data sets
m = train-p+1;
HL_tde = zeros(m,p);    % Data matrix
DT = 1;                 % Time step

for j=1:2:2*m
    for i=1:p
        temp(:,i) = HL_train(:,i+DT-1);
    end
    HL_tde(j:(j+1),:) = temp(:,:);
    DT = DT+1;
end

[u_tde,s_tde,v_tde] = svd(HL_tde,'econ');

rank = 5;
time_tde = 1:p;

[w_tde,e_tde,b_tde,atilde_tde] = optdmd(HL_tde,time_tde,rank,2);
time_int=[0:0.1:30];

HLx_prev = w_tde*diag(b_tde)*exp(e_tde.*time_int);

%% Bagging- DMD Time Delay Embedding
```

```
Niter   = 500;
Imax    = p;
Nsubset = floor(0.9*p);
temp    = [];
w_n     = [];
e_n     = [];
b_n     = [];
e_bt    = e_tde;
temp    = [];

for i=1:Niter

    ind = randperm(Imax, Nsubset);
    ind = sort(ind,2,"ascend");
    temp(:,:) = HL_tde(:,ind);
    Time = time(ind);

    [w_bt,e_bt,b_bt] = optdmd(temp,Time,rank,1,[],e_tde);
    w_n(:,:,i) = w_bt;
    e_n(:,:,i) = e_bt;
    b_n(:,:,i) = b_bt;

%     figure(2)
%     plot (real(ex(:,:,i)),imag(ex(:,:,i)),'o')
%     hold on
end

clear temp

[~,c1] = find( abs(real(e_n(1,:,:)))>0.03 );
[~,c2] = find( abs(real(e_n(2,:,:)))>0.03 );
c = union(c1, c2);
e_n(:,:,c)=[];
b_n(:,:,c)=[];
w_n(:,:,c)=[];

for i = 1:size(w_n,1)
    for j = 1:size(w_n,2)

        Wmean_bt(i,j,:) = mean(w_n(i,j,:),3);
    end
end
clear i

time_prevision_bt = [0:0.1:30];
x_prevision_bt = [];

for i=1:length(e_n)
    l_prev_bt(:,:,i) = exp(e_n(:,i).*time_prevision_bt);
    x_prevision_bt(:,:,i) = Wmean_bt*diag(b_n(:,:,i))*l_prev_bt(:,:,i);
end

X_prev_mean_bt = [mean(x_prevision_bt(1,:,:),3);mean(x_prevision_bt(2,:,:),3)];
X_prev_varn_bt = [var(x_prevision_bt(1,:));var(x_prevision_bt(2,:))];
```

```
H_prev_bt(:,:) = X_prev_mean_bt(1,:,:);
L_prev_bt(:,:) = X_prev_mean_bt(2,:,:);
```

Ex.1.3 Lotka-Volterra Model

```
clear all; close all; clc;

%% Lotka Volterra
% Optimization problem

load HL_population.mat
global HL
global H
global L
global time

H = H/1e3; L = L/1e3;

p0 = [1.1; 0.4; 0.3; 0.4; 20; 32];

fn = @lv_loss;
lb = [0 0 0 0 7 9];
ub = [3 1 4 1 150 150];
[p_opt,res] = fmincon(fn,p0,[],[],[],[],lb,ub)


disp(res)
disp('OPTIMAL PARAMETERS:')
disp({'b', 'p', 'r', 'd'})
disp(p_opt')


% Final solution optimized
options = odeset('AbsTol', 10^-12, 'RelTol', 10^-12, 'MaxStep', 0.1);
lv_opt = @(t,u) lvfun(p_opt(1),p_opt(2),p_opt(3),p_opt(4),t,u);

[t_fin,x_fin] = ode45(lv_opt , [0:0.1:30] , [p_opt(5); p_opt(6)]);
year_f = years(1) + t_fin*2;



% figure(1)

plot(years,H.*1e3,'r--o','LineWidth',.5)

hold on
plot(years,L.*1e3,'b--o','LineWidth',.5)
plot(year_f,x_fin(:,1).*1e3,'r','Linewidth',2)
plot(year_f,x_fin(:,2).*1e3,'b','Linewidth',2)

ylabel ('Population State')
xlabel ('Years')
title ('Lotka-Volterra Model')
legend ('Hare','Lynx','LV Hare','LV Lynx');
xlim([1845 1903])
```

```
grid on

function loss = lv_loss(par)

global HL
global H
global L
global time

b = par(1);
p = par(2);
r = par(3);
d = par(4);

H0 = par(5);
L0 = par(6);

timespan = [1:0.1:30];

f = @(t,u) lvfun(b, p, r, d, t, u);
[t_mod,x_mod] = ode45(f,timespan,[H0;L0]);

H_mod = x_mod(:,1);
L_mod = x_mod(:,2);

H_data = interp1(time,H,t_mod);
L_data = interp1(time,L,t_mod);

loss = sqrt(sum((H_data-H_mod).^2 + (L_data-L_mod).^2));

end
```

Ex.1.4 SINDy Model

```
import numpy as np
!pip install pysindy
from scipy.integrate import solve_ivp
from sklearn.metrics import mean_squared_error
import pysindy as ps
import statistics
from scipy.interpolate import interp1d

dt = 2
t_train = np.linspace(0, 58, 30)

# Dataset
t_train_span = (t_train[0], t_train[-1])
X = np.array([[32, 50, 12, 10, 13, 36, 15, 12, 6, 6, 65, 70, 40, 9, 20, 34, 45,
 40, 15, 15, 60, 80, 26, 18, 37, 50, 35, 12, 12, 25],[20,20, 52, 83, 64, 68, 83
, 12, 36, 150, 110, 60, 7, 10, 70, 100, 92, 70, 10, 11, 137, 137, 18, 22, 52, 8
3, 18, 10, 9, 65]])
```

```
# Cubic Interpolation
f1 = interp1d(t_train, [32, 50, 12, 10, 13, 36, 15, 12, 6, 6, 65, 70, 40, 9, 20
, 34, 45, 40, 15, 15, 60, 80, 26, 18, 37, 50, 35, 12, 12, 25], kind='cubic')
f2 = interp1d(t_train, [20,20, 52, 83, 64, 68, 83, 12, 36, 150, 110, 60, 7, 10,
 70, 100, 92, 70, 10, 11, 137, 137, 18, 22, 52, 83, 18, 10, 9, 65], kind='cubic
')


tnew = np.linspace(0, 58, num=150, endpoint=True)
dtNew=58/150
X1=f1(tnew)
X2=f2(tnew)
X=[X1,X2]
X=X/np.max(X)

feature_names = ['x', 'y']

n_candidates_to_drop=2
N=1000  #Bagging iterations

# SINDy Library Ensemble
E_library_opt = ps.STLSQ()
model = ps.SINDy(feature_names=feature_names, optimizer=E_library_opt, discrete
_time=True)
model.fit([np.transpose(X)], t=dtNew, library_ensemble=True, n_models=N, quiet=
True,n_candidates_to_drop=n_candidates_to_drop, multiple_trajectories=True)
E_library_Xi = np.asarray(model.coef_list)
n_targets = len(feature_names)
n_features = len(model.get_feature_names())
inclusion_probabilities = np.count_nonzero(model.coef_list, axis=0)

output = np.var(E_library_Xi, axis=0, dtype=np.float64)
a=output>=np.max(output)
B=inclusion_probabilities/N <= 0.3 + a
inclusion_probabilities[B] = 0.0

Theta= E_library_Xi[1,:,:]
Theta[inclusion_probabilities <= 100] = 0.0

Xix1=Theta[0,0]
Xix2=Theta[0,1]
Xix3=Theta[0,2]
Xix4=Theta[0,3]
Xix5=Theta[0,4]
Xix6=Theta[0,5]
```

```
Xiy1=Theta[1,0]
Xiy2=Theta[1,1]
Xiy3=Theta[1,2]
Xiy4=Theta[1,3]
Xiy5=Theta[1,4]
Xiy6=Theta[1,5]

for i in range(1,N):
  Theta= E_library_Xi[i,:,:]
  Theta[inclusion_probabilities <= 50] = 0.0
  Xix1=Theta[0,0]+Xix1
  Xix2=Theta[0,1]+Xix2
  Xix3=Theta[0,2]+Xix3
  Xix4=Theta[0,3]+Xix4
  Xix5=Theta[0,4]+Xix5
  Xix6=Theta[0,5]+Xix6

  Xiy1=Theta[1,0]+Xiy1
  Xiy2=Theta[1,1]+Xiy2
  Xiy3=Theta[1,2]+Xiy3
  Xiy4=Theta[1,3]+Xiy4
  Xiy5=Theta[1,4]+Xiy5
  Xiy6=Theta[1,5]+Xiy6

XiF=np.array([[Xix1/1000, Xix2/1000, Xix3/1000, Xix4/1000, Xix5/1000, Xix6/1000
],[Xiy1/1000, Xiy2/1000, Xiy3/1000, Xiy4/1000, Xiy5/1000, Xiy6/1000]])

chopped_inds = np.any(inclusion_probabilities != 0.0, axis=0)
chopped_inds = np.ravel(np.where(~chopped_inds))

X=X[:,50:90]

# Bagging of coefficients on the truncated library
library = ps.PolynomialLibrary(degree=2, library_ensemble=True,ensemble_indices
=chopped_inds)
E_opt = ps.STLSQ()
model = ps.SINDy(feature_names=feature_names, optimizer=E_opt, feature_library=
library, discrete_time=False)
model.fit([np.transpose(X)], t=dtNew, ensemble=True, n_models=N, n_subset=20, q
uiet=True,replace=True, multiple_trajectories=True)

d_st_ensemble_coefs = np.asarray(model.coef_list)
d_st_mean = np.mean(d_st_ensemble_coefs, axis=0)
d_st_std = np.std(d_st_ensemble_coefs, axis=0)
d_st_median = np.median(d_st_ensemble_coefs, axis=0)
```

```
# Restoring the original size of the matrix
for i in range(len(chopped_inds)):
    d_st_mean = np.insert(d_st_mean, chopped_inds[i], 0.0, axis=-1)
    d_st_std = np.insert(d_st_std, chopped_inds[i], 0.0, axis=-1)
    d_st_median = np.insert(d_st_median, chopped_inds[i], 0.0, axis=-1)

print(d_st_mean)
print(d_st_median)
```

Ex.2.1-2 NN Kuramoto-Sivanshinsky equation

```
%% NN - KS Equations
clear all;close all;clc;

input=[];
output=[];

N = 4;            % States
N_iteration= 100; % #Iteration random conditions


for j=1:N_iteration
    u0 = randn(N,1);
    [t,x,u] = ks_m(u0,N);

    input  = [input; u(1:end-1,:)];
    output = [output; u(2:end,:)];

end

%% Train NN
net = feedforwardnet([10 10 10]);
net.layers{1}.transferFcn = 'logsig';
net.layers{2}.transferFcn = 'logsig';
net.layers{3}.transferFcn = 'purelin';

net = train(net,input.',output.');

u1 = u(1,:).';
u_nn(1,:)=u1;
for i=2:length(t)
    % Recursive forecast
    utemp = net(u1);
    u_nn(i,:)=utemp.';
    u1=utemp;

end

%% Compare the results using different initial conditions
u_nc = randn(N,1);

[t_real,x_real,u_real] = ks_m(u_nc,N);
```

```
u_test1 = u_real(1,:).';
u_nn(1,:)=u_test1;

for jj=2:length(t_real)
    % Recursive forecast
    u_next = net(u_test1);
    u_nn(jj,:)=u_next.';
    u_test1=u_next;
end
```

Ex.2.3 NN Reaction diffusion equations

```
clear all; close all; clc;
%% Reaction - diffusion equations

load ('reaction_diffusion_big.mat')

% Two-component reaction–diffusion equations (u,v)
% t = time sequence
% x = state sequence
% y = state sequence
% u = time-state evaluation
% v = time-state evaluation


%% Dimension reduction
nu = size(u,2);         % Number of columns of u
f_ux = zeros(0);        % Initialization of the flat matrix

for i = 1:length(t)

    f_ux_i = reshape(u(:,:,i),[1,nu^2]);
    f_ux = [f_ux;f_ux_i];

end
f_ux = f_ux';
[u_rd,s_rd,v_rd] = svd(f_ux,'econ');


% Select rank
r = 10;

% Truncation up to the rank
u_tr = u_rd(:,1:r);
s_tr = s_rd(1:r,1:r);
v_tr = v_rd(:,1:r);



t_ind = round(0.9*length(t));
t_n =t(1:t_ind);
reduced_states = zeros(length(t_n), r);

for i = 1:length(t_n)
```

```
    f_state = f_ux(:,i);
    reduced_state_i = u_tr\f_state;
    reduced_states(i,:) = reduced_state_i;

end


%% Train NN

input   = reduced_states(1:end-1,:)';
output  = reduced_states(2:end,:)';

hiddenLayerSize = [10, 7, 5];
trainFcn = 'trainlm';
net = fitnet(hiddenLayerSize,trainFcn);

 net.layers{1}.transferFcn = 'logsig';
 net.layers{2}.transferFcn = 'purelin';
 net.layers{3}.transferFcn = 'purelin';

net.input.processFcns = {'removeconstantrows','mapminmax'};
net.output.processFcns = {'removeconstantrows','mapminmax'};

net.divideFcn = 'dividerand';  % Divide data randomly
net.divideMode = 'sample';  % Divide up every sample
net.divideParam.trainRatio = 70/100;
net.divideParam.valRatio = 15/100;
net.divideParam.testRatio = 15/100;

[net,tr] = train(net,input,output);

%% Test NN
y = net(input);

f_nn = [];

for i = 1:length(t_n)-1
    forecast_i = u_tr*y(:,i);
    forecast_im = reshape(forecast_i, [nu,nu]);
    f_nn(:,:,i) = forecast_im;

end

% compare frames
nframe = 123;

figure(5)
subplot(1,2,1)
pcolor(x,y,u(:,:,nframe)); shading interp; colormap(hot)
title(['Data set - time frame = ', num2str(nframe)])
xlabel('x')
ylabel('y')
subplot(1,2,2)
pcolor(x,y,f_nn(:,:,nframe)); shading interp; colormap(hot)
title(['NN forecast - time frame =', num2str(nframe)])
xlabel('x')
```

```
ylabel('y')


T = t(11:end);
reduced_states = zeros(length(T), r);

for i = 1:length(T)
    f_state = f_ux(:,i);
    reduced_state_i = u_tr\f_state;
    reduced_states(i,:) = reduced_state_i;

end
input   = reduced_states(1:end-1,:)';
y = net(input);

f_nn = [];

for i = 1:length(T)-1
    forecast_i = u_tr*y(:,i);
    forecast_im = reshape(forecast_i, [nu,nu]);
    f_nn(:,:,i) = forecast_im;

end

nframe = 192;

figure(6)
subplot(1,2,1)
pcolor(x,y,u(:,:,nframe)); shading interp; colormap(hot)
title(['Data set - time frame = ', num2str(nframe)])
xlabel('x')
ylabel('y')
subplot(1,2,2)
pcolor(x,y,f_nn(:,:,182)); shading interp; colormap(hot)
title(['NN forecast - time frame =', num2str(nframe)])
xlabel('x')
ylabel('y')
```

Ex.3 NN Lorenz equations

```
%% NN - Lorenz equations
clear all; close all; clc;

% Standard coefficients
b   = 8/3;
sig = 10;

% Time series
T = 10;
dt = 0.01;
t = 0:dt:T;

%Lorenz equations in the different conditions

rho_train =[10 28 35];
rho_test = [17 40];
```

```
Lrnz_1 = @(t,x) ([sig*(x(2)-x(1)); rho_train(1)*x(1)-x(1)*x(3)-x(2); x(1)*x(2)-b*x(3)]);
Lrnz_2 = @(t,x) ([sig*(x(2)-x(1)); rho_train(2)*x(1)-x(1)*x(3)-x(2); x(1)*x(2)-b*x(3)]);
Lrnz_3 = @(t,x) ([sig*(x(2)-x(1)); rho_train(3)*x(1)-x(1)*x(3)-x(2); x(1)*x(2)-b*x(3)]);

Lrnz_4 = @(t,x) ([sig*(x(2)-x(1)); rho_test(1)*x(1)-x(1)*x(3)-x(2); x(1)*x(2)-b*x(3)]);
Lrnz_5 = @(t,x) ([sig*(x(2)-x(1)); rho_test(2)*x(1)-x(1)*x(3)-x(2); x(1)*x(2)-b*x(3)]);


ode_options = odeset('RelTol',1e-10, 'AbsTol',1e-11);
input  = [];
output = [];

% NN Training phase

% Data generation
for j=1:50

x0=30*(rand(3,1)-0.5);
    [t,y] = ode45(Lrnz_1,t,x0);
    rho = ((y(2:end,2)-y(1:end-1,2))/dt+y(1:end-1,1).*y(1:end-1,3)+y(1:end-1,2))./y(1:end-
1,1);
    input=[input; y(1:end-1,:) rho.*y(1:end-1,1)];
    output=[output; y(2:end,:)];
    plot3(y(:,1),y(:,2),y(:,3)), hold on
    plot3(x0(1),x0(2),x0(3),'ro')

    [t,y] = ode45(Lrnz_2,t,x0);
    rho = ((y(2:end,2)-y(1:end-1,2))/dt+y(1:end-1,1).*y(1:end-1,3)+y(1:end-1,2))./y(1:end-
1,1);
    input=[input; y(1:end-1,:) rho.*y(1:end-1,1)];
    output=[output; y(2:end,:)];
    plot3(y(:,1),y(:,2),y(:,3)), hold on
    plot3(x0(1),x0(2),x0(3),'ro')

    [t,y] = ode45(Lrnz_3,t,x0);
    rho = ((y(2:end,2)-y(1:end-1,2))/dt+y(1:end-1,1).*y(1:end-1,3)+y(1:end-1,2))./y(1:end-
1,1);
    input=[input; y(1:end-1,:) rho.*y(1:end-1,1)];
    output=[output; y(2:end,:)];
    plot3(y(:,1),y(:,2),y(:,3)), hold on
    plot3(x0(1),x0(2),x0(3),'ro')

end

%% NN Train
trainFcn = 'trainlm';
net = fitnet([10 10],trainFcn);

net.input.processFcns = {'removeconstantrows','mapminmax'};
net.output.processFcns = {'removeconstantrows','mapminmax'};

net.divideFcn = 'dividerand';  % Divide data randomly
net.divideMode = 'sample';  % Divide up every sample

net.divideParam.trainRatio = 80/100;
net.divideParam.valRatio = 25/100;
net.divideParam.testRatio = 15/100;

% Train the NN
[net,tr] = train(net, input.',output.');

%% NN Test phase

%% Test for rho = 17
x0=30*(rand(3,1)-0.5);
[t,y] = ode45(Lrnz_4,t,x0);

rho_nn(1)=((y(2,2)-y(1,2))/dt+y(1,1).*y(1,3)+y(1,2))./y(1,1); %rho initial condition
for i = 2:length(t)
    % Recursive forecast
    y0= net([x0(1:3); rho_nn(i-1)*x0(1)]);
    y_nn(i,1:3) = y0(1:3).';
    rho_nn(i) = ((y(i,2)-y(i-1,2))/dt+y(i-1,1).*y(i-1,3)+y(i-1,2))./y(i-1,1);
    x0=y0;

end

figure(1)

subplot(3,2,1), plot(t,y(:,1),'b',t,y_nn(:,1),'r','Linewidth',2)
title ('Lorenz equation - \rho = 17');
xlabel ('time');
ylabel ('x');
legend ('Data','Forecast');
grid on
subplot(3,2,3), plot(t,y(:,2),'b',t,y_nn(:,2),'r','Linewidth',2)
xlabel ('time');
ylabel ('y');
legend ('Data','Forecast');
grid on
subplot(3,2,5), plot(t,y(:,3),'b',t,y_nn(:,3),'r','Linewidth',2)
xlabel ('time');
ylabel ('z');
legend ('Data','Forecast');
grid on



%% Test for rho = 40

x0=30*(rand(3,1)-0.5);
[t,y] = ode45(Lrnz_5,t,x0);

rho_nn(1)=((y(2,2)-y(1,2))/dt+y(1,1).*y(1,3)+y(1,2))./y(1,1); %rho initial condition
for i=2:length(t)
    % Recursive forecast
    y0=net([x0(1:3); rho_nn(i-1)*x0(1)]);
    y_nn(i,1:3)=y0(1:3).';
    rho_nn(i)=((y(i,2)-y(i-1,2))/dt+y(i-1,1).*y(i-1,3)+y(i-1,2))./y(i-1,1);
    x0=y0;
end
```

```
figure(1)
subplot(3,2,2), plot(t,y(:,1),'b',t,y_nn(:,1),'r','Linewidth',2)
title ('Lorenz equation - \rho = 40');
xlabel ('time');
ylabel ('x');
legend ('Data','Forecast');
grid on
subplot(3,2,4), plot(t,y(:,2),'b',t,y_nn(:,2),'r','Linewidth',2)
xlabel ('time');
ylabel ('y');
legend ('Data','Forecast');
grid on
subplot(3,2,6), plot(t,y(:,3),'b',t,y_nn(:,3),'r','Linewidth',2)
xlabel ('time');
ylabel ('z');
legend ('Data','Forecast');
grid on
```