Modeling from Measurements Course Project

Leonardo Iacussi

leonardo.iacussi@polimi.it

Abstract

Machine Learning (ML) and Deep Learning algorithms are increasingly being used given the growing availability of real data and computing power. Most of the developed algorithms deal with image and audio recognition, but interest in such algorithms is also expanding for solving purely engineering problems usually concerning systems of differential and partial differential equations. In this project, several recent regression methods are discussed: Dynamic Mode Decomposition (DmD), SINDy models, Neural Network to model and predict the dynamic of some historical problems such as Predator-Prey model, the Kuramoto-Sivashinsky equations and Lorenz equations and compare the results with well defined solutions.

1. Introduction and Overview

Dynamical system provide a mathematical framework to describe the real-world phenomena, in particular for modeling the interaction between different quantities and how these quantities evolve in time. The main goals and challenges of dynamical systems modeling are to analyse, to understand and to forecast the behaviour of real world systems, for example in physics fluid dynamics, continuous mechanics, heat propagation, etc... Real world systems are generally non-linear and exhibit a multi-scale behaviour in both space and time, therefore they are very complex to model [1].
Generally dynamical systems have been described using a set of differential equations or partial differential equations and the mathematical framework will be shortly summarized in the first part of Section 2.

1.1. Data-driven Dynamical System

The Data-Driven approach for modeling dynamical system is currently developing due to the advent of new machine learning algorithms and due to the increasing availability of data coming from measurements, sensors, simulations, shared database, etc... In this project data-driven methods as Dynamic Mode Decomposition (DMD), Sparse Identification of Non Linear Dynamics (SINDy) and Neural Network will be described and used to analyse and to find the best fitting model of a famous predator prey data set (Canadian lynx and Snowshoe hare) and to fit data generated by numerically solving the Kuramoto-Sivashinsky equation and Lotkta-Volterra equations.

1.2. High dimensional data and Dimensionality Reduction techniques

One of the main problem of data-driven modeling is the high dimensionality of the data set, and often the high dimensionality of the data set is completely not correlated with the phenomena to model. Generally a dynamical systems is a low rank system which means that there are few dominant patterns that are able to explain the high-dimensional data. Therefore some dimensionality reduction techniques as Singular Value Decomposition (SVD) can be used to find the best low rank approximation of the data set. The SVD is also used in the Principal Component Analysis (PCA) algorithm where high dimensional data is decomposed into its most statistically descriptive factors. Projecting the initial data set in a lower dimensional space brings two main advantages: 1) improving of the physical understanding of the system, number of real degree of freedom of the system for example, 2) faster fitting of the machine learning algorithm avoiding overfitting of non related system variables.

2. Theoretical Background

As previously mentioned there are several numerical methods and algorithms for solving a dynamical system starting from raw data. Before going in deep in their mathematical model, it is important to build a general mathematical framework of the dynamical systems. A general dynamical system will be considered as follow:

$$\frac{d}{dt}\mathbf{x}(t) = \mathbf{f}(\mathbf{x}(t), t, \boldsymbol{\beta}), \tag{2.0.1}$$

where $\mathbf{x}$ is the state vector of the system, describing its evolution over time, $\mathbf{f}$ is the vector field which depends on the state $\mathbf{x}$ itself, time t and a set of parameters stored in the vector $\boldsymbol{\beta}$.
If possible, it is beneficial to work with linear dynamics because they admit closed-form solution and there are several numerical techniques to solve linear dynamics system in the form:

$$\frac{d}{dt}\mathbf{x}(t) = \mathbf{A}\mathbf{x}. \tag{2.0.2}$$

The solution of the above equation is given by:

$$\mathbf{x}(t_0 + t) = e^{\mathbf{A}t}\mathbf{x}(t_0). \tag{2.0.3}$$

It is important to keep in mind, the dynamics behaviour of the system is completely characterized by the eigenvalues and eigenvectors of the matrix $\mathbf{A}$.

As it is necessary to solve differential equations using numerical methods a discrete-time formulation for differential equation is required and can be written as follow:

$$\mathbf{x}_{k+1} = \mathbf{A}\mathbf{x}_k. \tag{2.0.4}$$

2.1. Dynamic Mode Decomposition (DMD)

Dynamic Mode Decomposition (DMD) algorithm is based on Singular Value Decomposition (SVD) which aims to provide an efficient reduction of high dimensional systems resulting in a hierarchy of modes based entirely on spatial correlation and energy content. The DMD provides in addition how the modes evolve in time, so this algorithm is used for fitting the dynamics of systems from real data (measurements), and to find a good model which describes the system behaviour in time.

As any data-driven algorithm the starting point is always data collection of some system state snapshots in time. All the snapshots in time are stored in two data matrices $\mathbf{X}$ and $\mathbf{X}'$:

$$\mathbf{X} = \begin{bmatrix} | & | & & | \\ \mathbf{x}(t_1) & \mathbf{x}(t_2) & \dots & \mathbf{x}(t_m) \\ | & | & & | \end{bmatrix}, \quad (2.1.1a)$$

$$\mathbf{X}' = \begin{bmatrix} | & | & & | \\ \mathbf{x}(t'_1) & \mathbf{x}(t'_2) & \dots & \mathbf{x}(t'_m) \\ | & | & & | \end{bmatrix}. \quad (2.1.1b)$$

The k snapshot pairs are denoted by $\{(\mathbf{x}(t_k), \mathbf{x}(t'_k))\}_{k=1}^{m}$ where $t_k = k\Delta t$ and $t'_k = t_{k+1} = t_k + \Delta t$.
Assuming that no variable sampling time will be considered in this project it is possible to write $\mathbf{x}_k = \mathbf{x}(k\Delta t)$. Considering a discrete time formulation of a differential equation (2.0.4) the framework of DMD problem becomes:

$$\mathbf{X}' = \mathbf{A}\mathbf{X}. \quad (2.1.2)$$

Mathematically the best-fit operator is defined as (2.1.3) where $||.||_F$ is the Frobenius norm and † denotes the pseudo-inverse of the matrix.

$$\mathbf{A} = argmin||\mathbf{X}' - \mathbf{A}\mathbf{X}||_F = \mathbf{X}'\mathbf{X}^\dagger. \quad (2.1.3)$$

In most of data-driven applications the initial data set is high-dimensional (high dimensional state vector $\mathbf{x} \in \mathbb{R}^n$), therefore the matrix $\mathbf{A}$ is massive and hard to calculate. So a dimensionality reduction algorithm as SVD has applied to the data set in order to project it into a lower dimensional space before the calculation of the $\mathbf{A}$ matrix. The exact DMD is given by the following steps:

Step 1 Compute the Singular Value Decomposition of the matrix $\mathbf{X}$ and project data into a lower dimensional space $r \leq m$.

$$\mathbf{X} \approx \tilde{\mathbf{U}}\tilde{\Sigma}\tilde{\mathbf{V}}^*, \quad (2.1.4)$$

where $\tilde{\mathbf{U}} \in \mathbb{C}^{n\times r}$, $\tilde{\Sigma} \in \mathbb{C}^{r\times r}$, $\tilde{\mathbf{V}} \in \mathbb{C}^{m\times r}$, with $r \leq m$. It is important to choose the correct rank truncation factor so that the equation (2.1.5) can well approximate the initial data set.

Step 2 Project the matrix $\mathbf{A}$ into the lower dimensional space:

$$\tilde{\mathbf{A}} = \tilde{\mathbf{U}}^*\mathbf{A}\tilde{\mathbf{U}} = \tilde{\mathbf{U}}^*\mathbf{X}'\tilde{\mathbf{V}}\tilde{\Sigma}^{-1}. \quad (2.1.5)$$

Therefore the new linear model which describes the system in the lower dimensional space is defined as:

$$\tilde{\mathbf{x}}_{k+1} = \tilde{\mathbf{A}}\tilde{\mathbf{x}}_k. \quad (2.1.6)$$

Step 3 Find the eigen-values (matrix $\mathbf{\Lambda}$) and corresponding low dimensional eigen-vectors (matrix $\mathbf{W}$) of the $\tilde{\mathbf{A}}$ matrix. At this point the interpretability of the dynamic behaviour of the system becomes clear and it is possible to build a model to forecast the future behaviour.

$$\tilde{\mathbf{A}}\mathbf{W} = \mathbf{W}\mathbf{\Lambda}. \quad (2.1.7)$$

Step 4 The high-dimensional modes $\mathbf{\Phi}$ are reconstructed using the eigen-values and eigen-vectors obtained in the previous step using the time shifted matrix $\mathbf{X}'$:

$$\mathbf{\Phi} = \mathbf{X}'\tilde{\mathbf{V}}\tilde{\Sigma}^{-1}\tilde{\mathbf{W}}. \quad (2.1.8)$$

At the end it is possible to predict the system behaviour over time expanding the system state in terms of data-driven spectral decomposition:

$$\mathbf{x}_k = \sum_{j=1}^{r} \mathbf{\Phi}_j \lambda_j^{k-1} b_j = \mathbf{\Phi}\mathbf{\Lambda}^{k-1}\mathbf{b}, \quad (2.1.9)$$

where $\mathbf{b}$ correspond to the mode amplitudes and it is generally calculated as $\mathbf{b} = \mathbf{\Phi}^\dagger\mathbf{x}_1$ using the first snapshots to determine the mixture of DMD mode amplitudes.

The main limitation of the standard DMD algorithm is the high sensitivity to noisy data, common situation when real data coming from measurements are used to build the model. When noise is present the model tend to diverge to infinite or converge to zero depending on the eigen-values real part position. The first alternative to de-noise and robustify the standard DMD model is the Optmized DMD formulation based on the minimization of the following problem:

$$\underset{b_j,\psi_j,\lambda_j}{argmin}(||\mathbf{X} - \sum_{j=1}^{r} \mathbf{\Phi}_j \lambda_j^{k-1} b_j||_F + \gamma||\mathbf{b}||_1), \quad (2.1.10)$$

where the $||.||_1$ denotes the $l_1 - norm$ penalization to promote sparsity of $\mathbf{b}$.

The second possibility to stabilize and robustify the DMD algorithm is to use the "Boosted Optimized DMD" (BOP-DMD) method which can also provide uncertainty estimates of the eigen-values and DMD eigen-modes. It consist in solving the optimized DMD minimization problem (2.3.3) several times, each time with a randomly selected subset with a dimension $p \leq m$ picked from the initial data set $\mathbf{X}$. At the end the mean eigen-values $\langle\lambda_j\rangle$, eigen-vectors $\langle\psi_j\rangle$ and mode amplitudes $\langle b_j\rangle$ can be used to build the model.

$$\langle\mathbf{x}_k\rangle = \sum_{j=1}^{r} \langle\mathbf{\Phi}_j\rangle\langle\lambda_j^{k-1}\rangle\langle b_j\rangle. \quad (2.1.11)$$

2.2. Sparse Identification of Nonlinear Dynamics (SINDy)

As explained in section 2.1 the best-fit linear model can be obtained using DMD algorithm, but fitting a non linear model is considerably more challenging because there are combinatorially many possible model structures based on many non linear functions. The sparse identification of nonlinear dynamics (SINDy) algorithm discovers parsimonious models through a sparsity-promoting optimization to select only a few model terms from a library of candidate functions. It helps to avoid overfitting and it gives physical interpretability of the model [1]. Considering that many dynamical systems can be represented as:

$$\frac{d}{dt}\mathbf{x} = \mathbf{f}(\mathbf{x}), \quad (2.2.1)$$

where $\mathbf{f}$ represents a few active terms in the space of a possible right-hand functions and can be approximated by a generalized linear model as follow:

$$\mathbf{f}(\mathbf{x}) \approx \sum_{k=1}^{p} \Theta_k \xi_k = \mathbf{\Theta}(\mathbf{x}). \quad (2.2.2)$$

As previously mentioned for the DMD algorithm in section 2.1, first of all, a time-series data must be collected in a matrix $\mathbf{X}$ and its derivative must be calculated numerically and stored in the matrix $\dot{\mathbf{X}}$. A library of possible nonlinear functions may be build from the data $\mathbf{X}$ and stored in the matrix $\mathbf{\Theta}(\mathbf{X})$. Usually a polynomial library with a maximum polynomial degree of d is used as standard fitting library. The dynamical system can now be represented in terms of data matrices as in (2.2.3a).

$$\mathbf{\Theta}(\mathbf{X}) = \left[\mathbf{1}, \mathbf{X}, \ldots, \mathbf{X^d}, \ldots, sin(\mathbf{X}), \ldots\right], \tag{2.2.3a}$$

$$\dot{\mathbf{X}} = \mathbf{\Theta}(\mathbf{X})\mathbf{\Xi}, \tag{2.2.3b}$$

where the column ξ_k in $\mathbf{\Xi}$ is a vector of coefficients determining the active terms in the k^{th} row in (2.2.3a). At the end it is possible to determine the best fitting parameters by minimizing the distance between the derivative of real data and the approximated model:

$$\boldsymbol{\xi_k} = \underset{\boldsymbol{\xi'_k}}{argmin}||\dot{\mathbf{X}}_k - \mathbf{\Theta}(\mathbf{X})\boldsymbol{\xi}'_k||_2 + \lambda||\boldsymbol{\xi}'_k||_1, \tag{2.2.4}$$

where $\dot{\mathbf{X}}_k$ is the k^{th} column of $\dot{\mathbf{X}}$ and λ is the sparsity-promoting knob. As for BOP-DMD algorithm it is also possible to apply bagging technique in SINDy algorithm by initially randomly selecting k number of samples and solving the minimization problem several times. At the end the model can be build by using the average of the $\boldsymbol{\xi}_k$ coefficients.

2.3. Neural Network (NN) for Dynamical Systems

The importance of Neural Network (NN) algorithms is increasing in the last decade because of the big advancements in computational power and data availability. The generic structure of a feed-forward neural network is represented in Figure 2.1 and it is composed by one input layer $\mathbf{x}$ and one output layer $\mathbf{y}$ with respectively the system input and output dimensions (number of perceptrons). Moreover there are several hidden layers $\mathbf{h}^{(j)}$ generally composed by an high number of perceptrons which can learn the system main features. Mathematically the basic architecture of a feed-forward neural network composed by M layers can be written as:

$$\mathbf{x}^{(j)} = f_j(\mathbf{A}_j, \mathbf{x}^{(j-1)}), \tag{2.3.1a}$$

$$\mathbf{y} = f_M(\mathbf{A}_M, f_{M-1}(\mathbf{A}_{M-1}, ..., f_2(\mathbf{A}_2)f_1(\mathbf{A}_1\mathbf{x}))), \tag{2.3.1b}$$

where the matrix $\mathbf{A}_j$ contains all the coefficients which map each variable from one layer to the other and the $f_j(.)$ is the activation function between layers which is usually nonlinear. In general the number of layers and how to map between the layer is chosen by the user. Usually these parameters must be tuned in order to achieve a good classification or regression depending on the problem.

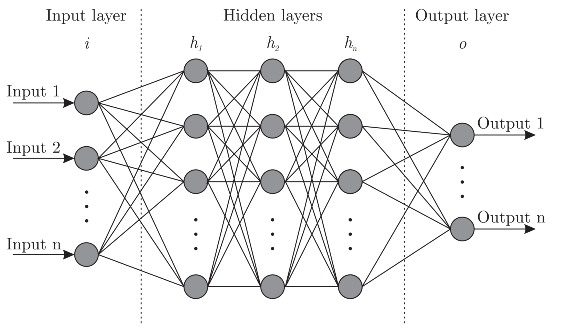


Figure 2.1: Neural Network generic architecture [2].

2.3.1. *Stochastic Gradient Descent and Backpropagation*

The Neural Network training is based on two optimization algorithms: the Stochastic Gradient Descent which can provide a more rapid evaluation of the optimal network weights and Backpropagation which allows for an efficient computation of the objective function gradient [1]. Others optimizer can be used for find an optimal solution for the network weights, for example Adam is a popular optimizer for training a Neural Network.
Considering the function describing the general for of data fit in a Neural Network:

$$f((x)) = f(\mathbf{x}, \mathbf{A}_1, \mathbf{A}_1, ..., \mathbf{A}_M). \tag{2.3.2}$$

The goal of training a Neural Network is the minimization of the error between the network prediction and the data to fit. Consequently the standard root-mean-square for neural network is defined as:

$$\underset{A_j}{argmin}E(\mathbf{A}_1, \mathbf{A}_1, ..., \mathbf{A}_M) = \underset{A_j}{argmin}\sum_{k=1}^{n}(f(\mathbf{x}, \mathbf{A}_1, \mathbf{A}_1, ..., \mathbf{A}_M) - \mathbf{y}_k)^2, \tag{2.3.3}$$

which can be minimized by setting the partial derivative with respect to each matrix component to zero $\partial E/\partial(a_{ij})_k = 0$, considering $k = 1, 2, ..., M$. This gives the gradient of the function $\nabla f(\mathbf{x})$ with respect to the neural network parameters and this leads to a Newton-Raphson iteration scheme for finding the minima:

$$\mathbf{x}_{j+1}(\delta) = \mathbf{x}_j - \delta\nabla f(\mathbf{x}_j), \tag{2.3.4}$$

where δ is the learning rate and it determines how far a step should be taken along the gradient direction. As the number of parameters is usually very large, the calculation of the gradient with respect to each parameter could be computationally prohibitive, therefore the Stochastic Gradient Descent uses a subset of randomly selected points to approximate the gradient at each step.

3. Algorithm Implementation and Development

The mathematical models explained in Section 2 have been then implemented using MATLAB and Python code to find the best fitting model of some famous data sets. In this section the algorithms will be explained step by step before the results explanation.
All the results applied to the given data sets will be released in Section 4.

3.1. Dynamic Mode Decomposition for Predator-Prey model forecasting

The implementation of the Dynamic Mode Decomposition (DMD) has been done on MATLAB using the given *opt-DMD* algorithm (see Appendix A.3). The DMD with bagging algorithm is explained in Algorithm 1 as is is the most complex. Standard Opt-DMD is already implemented by one single function on both MATLAB and Python. The user mast choose the number times the optimized DMD algorithm is calculated K, the number of snapshots $p < m$ to use for fitting the model, considering m as the number of total snapshots in the initial dataset $\mathbf{X}$.

For the Time Delay DMD the only difference is in the $\mathbf{X}$ matrix which has different dimensions as follow:

$$\mathbf{X}_{td} = \begin{bmatrix} \mathbf{x}(t_1) & \mathbf{x}(t_2) & \dots & \mathbf{x}(t_{m-td}) \\ \mathbf{x}(t_2) & \mathbf{x}(t_3) & \dots & \mathbf{x}(t_{m-td+1}) \\ | & | & & | \\ \mathbf{x}(t_{td}) & \mathbf{x}(t_{td+1}) & \dots & \mathbf{x}(t_m) \end{bmatrix}, \quad (3.1.1)$$

When calling the $Opt - DMD$ MATLAB function it is possible to activate different flags: *var2opt* optimizer options, initial eigen-values guess, Tikhonov regularization term γ and it is possible to add a proximal operator in order to constrain the eigen-values on the left ($\mathbb{R}e - \mathbb{I}m$) plane to avoid the diverging of future predictions.
Moreover during bagging, in both $BOP - DMD$ and $TimeDelayedBOP - DMD$, it is important to discard non converging solutions, in particular when any value of the vector $\mathbf{b}_k$ is higher of a certain threshold determined by looking to the $\mathbf{b}$ vector at each iteration.

Algorithm 1 Optimized BOP-DMD algorithm [4].

```
  Input: Input $(\mathbf{X}, p, K)$
  procedure BOP-DMD$(\mathbf{X}, p, K)$
      Compute $\mathbf{\Phi}_0$,$\mathbf{\Omega}_0$,$\mathbf{b}_0$                ▷ opt-DMD regression
      for $k \in \{1, 2, ..., K\}$ do
          Choose $p$ of $m$ snapshots $(p < m)$             ▷ Bagging
          opt-DMD $\mathbf{\Phi}_k$,$\mathbf{\Omega}_k$,$\mathbf{b}_k$
          if ( any($\mathbf{b}_k \leq$ threshold)) then
              Update $\mathbf{\Phi}_0$,$\mathbf{\Omega}_0$,$\mathbf{b}_0$        ▷ keep the solution valid
          else
              Discard this solution
          end if
      end for
      Compute mean $\boldsymbol{\mu} = \{\langle\mathbf{\Phi}\rangle, \langle\mathbf{\Omega}\rangle, \langle\mathbf{b}\rangle\}$
      Compute variance $\boldsymbol{sigma} = \{\langle\mathbf{\Phi}^2\rangle, \langle\mathbf{\Omega}^2\rangle, \langle\mathbf{b}^2\rangle\}$
      return $\boldsymbol{\mu}$,$\boldsymbol{\sigma}$              ▷ Return opt BOP-DMD parameters
  end procedure
```

3.2. Fit the best Lotka-Volterra parameters

Given a generic Predator-Prey data set, the four parameters of the Lotka-Volterra equations have been fit to the initial data set using Python in order to find the best fitting model. The Lotka-Volterra equations is written as follow:

$$\begin{aligned} \frac{dx}{dt} &= \alpha x - \beta xy, \\ \frac{dy}{dt} &= \delta xy - \gamma y, \end{aligned} \quad (3.2.1)$$

where x represents the *predator* population, y represents the *prey* population, $\alpha, \beta, \delta, \gamma$ are the four positive parameters to fit in order to find the best Lotka-Volterra model which describes the populations behaviour [6].

Before fitting the model the data set has been shifted from time $t\_0 = 0$ to time $t\_end = 58$ with a $dt = 2$ years. Moreover a linear interpolation has been applied to the initial data (see Appendix A.1) in order to reduce the dt between points and enable the ODE solver to converge ($dt\_new = 0.01$).

In order to fit the parameters the loss function has been defined as the Mean Square Error between the real data and the predicted one using the *odeint* integrator from *scipy.integrate* Python library (see Appendix A.4). The loss function has been then minimized using the *fmin* optimization function from *scipy.optimize* Python library.

An important step before fitting the model is the choice of the parameters initial guess. In order to do so some Lotka-Volterra model simulations have been carried out by manually changing the parameters in order to find a solution which is closer to the real data. At the end of this process the chosen initial parameters are $\alpha\_0 = 0.75$, $\beta\_0 = 0.025$, $\delta\_0 = 0.0075$, $\gamma\_0 = 0.4$.

3.3. Fit the best non linear model using SINDy

The SINDy algorithm with bagging described in Algorithm 2 has been implemented in Python using the *pySINDy* Python library [5] (see Appendix A.5). In addition to the same p,K parameters for bagging described in Section 3.1, the user must choose the library of possible functions $\mathbf{\Theta}$ used then for fitting the data. Usually the pySindy Polynomial library can be used choosing the degree of the polynomial otherwise a Fuorier library or any custom library can be build before running the algorithm [3].
Often if the dt is not enough small the Runge-Kutta (RK45) integration algorithm for simulate the predicted time series can diverge from the expected solution. Therefore it is important to discard all the coefficients which make the solution divergent. It has been done by discarding all the solutions where (see Appendix A.6):

- absolute value of any solution point is higher of a certain threshold
- any solution point is negative
- the variance of the last 30% of the points is lower of a certain threshold (to avoid static solutions)

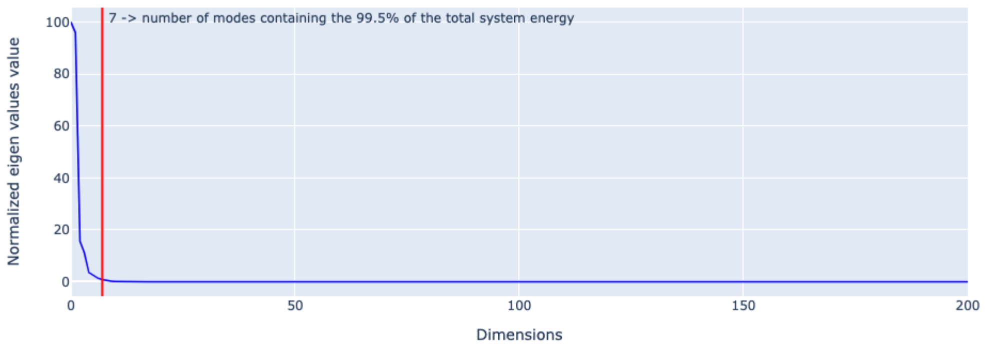
At the end the average coefficient matrix has been calculated for predict the dynamic model behaviour using only valid coefficients Ξ_k.

Algorithm 2 Ensemble-SINDy.

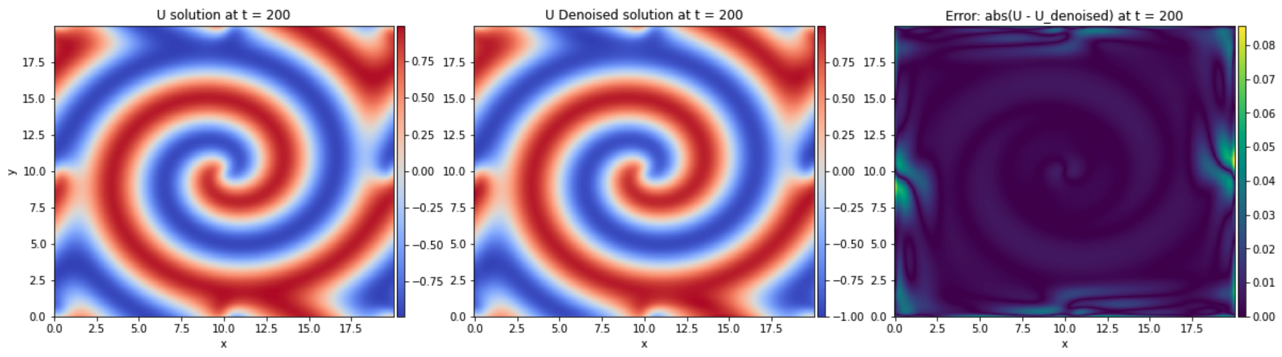
```
  Input: Input $(\mathbf{X}, t, p, K, \mathbf{\Theta})$
  procedure ENSEMBLE-SINDY$(\mathbf{X}, p, K)$
      for $k \in \{1, 2, ..., K\}$ do
          Choose $p$ of $m$ snapshots $(p < m)$             ▷ Bagging
          SINDy $\Xi_k$
          Calculate $\mathbf{x}_k(t)$        ▷ using Runge Kutta integration
  method
          if ( $\mathbf{x}_k(t)$ converge ) then
              Keep valid $\Xi_k$
          else
              Discard $\Xi_k$
          end if
      end for
      Compute mean $\boldsymbol{\mu} = \{\langle\Xi\rangle\}$
      Compute variance $\boldsymbol{\sigma} = \{\langle\Xi^2\rangle\}$
      return $\boldsymbol{\mu}$,$\boldsymbol{\sigma}$              ▷ Return opt SINDy coefficients
  end procedure
```

3.4. Train a Neural Network (NN) for solving Ordinary Differential Equations and Partial Differential Equations

In general for training a Neural Network, as a supervised learning algorithm, it is necessary to split the initial data set into the input matrix $\mathbf{X}$ and into the output matrix $\mathbf{y}$ which corresponds to the known labels. For time series forecasting the input and output layer dimension is the same and the $\mathbf{y}$ matrix contains all the $\mathbf{X}$ time snapshots shifted by one timestamp dt. Considering the initial dataset matrix as $\mathbf{DataSet}[snapshots, features]$,



(a) Sorted eigen-values of the system and rank truncation $r = 7$ containing the 99.5% of the system energy.



(b) Reaction Diffusion u solution at time $t = 200$ before and after dimensionality reduction. The error calculated as the difference between the two.

Figure 3.1: Singular Value Decomposition (SVD) applied on the entire Reaction Diffusion System.

$\mathbf{X} = \mathbf{DataSet}[0 : -1, :]$ and $\mathbf{y} = \mathbf{DataSet}[1 :, :]$ so that the network can learn how to forecast the model one snapshot in advance.
The network has been build and trained using *TensorFlow* Python package as shown in Appendix A.8. Many types of layers, layers dimensions, activation functions, optimizer, etc... can be used, therefore an hyper parameters tuning is required to improve the results and it can take long time.
To forecast the model behaviour using the trained neural network it is necessary to start from some initial conditions and then loop over time for the number of snapshots required by the problem (see Appendix A.9).

3.4.1. Train the NN on a lower dimensional space

Training a Neural Network on a lower dimensional space brings some advantages briefly explained in Section 1.2. In order to do so, the Singular Value Decomposition (SVD) has been used to project the initial data set into a lower dimensional space. As the model to find is described by the Reaction Diffusion set of equations (3.4.1) which is composed by two space dimensions evolving over time, before applying the SVD, the matrices describing the equation on the space must be reshape in a vector using the row major order convention. Moreover, as it has two solutions U and V, the two vectorized matrices has been concatenated over the space dimension (see Appendix A.10).

$$\begin{aligned} u_t &= 0.1\nabla^2 u + \lambda(A)u - \omega(A)v \\ v_t &= 0.1\nabla^2 v + \omega(A)u + \lambda(A)v \\ A^2 &= u^2 + v^2,\ \omega(A) = -\beta A^2, \lambda(A) = 1 - A^2 \end{aligned} \tag{3.4.1}$$

The SVD has been applied to the new matrix and as can be seen from Figure 3.1, with a rank truncation factor $r = 7$ it is possible preserve the 99.5% of the system energy and the reconstructed equation using the rank truncation is basically the same and the maximum error is on the order of 8%.

The matrices $\mathbf{X}$ and $\mathbf{y}$ have been now generated starting from the truncated vh vector obtained using the $numpy.linalg.svd$ algorithm (see Appendix A.11).

3.4.2. Train a Neural Network able to forecast parametric Lorenz Equation

Considering the Lorenz equations (3.4.2), the task has to train a Neural Network with a given set of parameters (ρ, σ, β) to forecast the Lorenz equations model behaviour with a different ρ values never seen by the Neural Network model during the training and cross-validation procedure. In order to improve the model prediction in this case the ρ value has been added as an additional training label together with the x, y, z values.

$$\begin{aligned} \frac{dx}{dt} &= \sigma(y - x), \\ \frac{dy}{dt} &= x(\rho - z) - y, \\ \frac{dz}{dt} &= xy - \beta z, \end{aligned} \tag{3.4.2}$$

4. Computational Results

In this section the computational results will be analysed, in particular in the first part three different fitting methods described in Section 3 will be used to find the best model which can describe

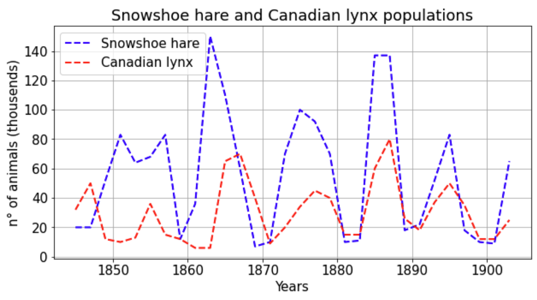


Figure 4.1: Snowshoe hare and Canadian lynx population.

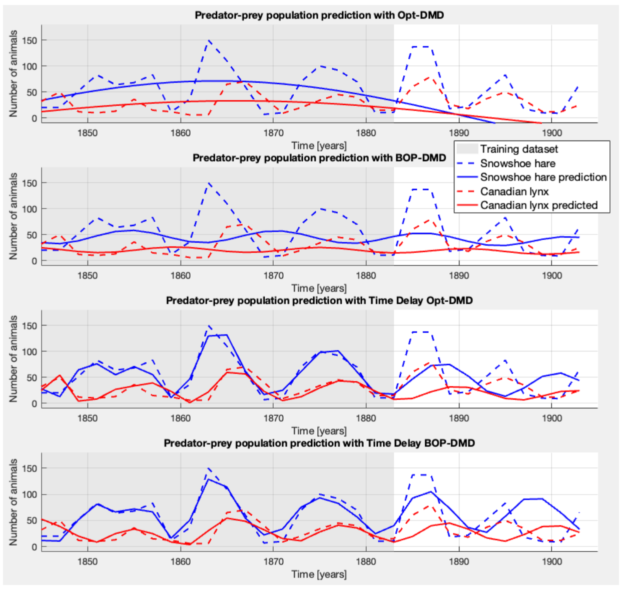


Figure 4.2: Optimized Dynamic Mode Decomposition for forecasting the population behaviour.

the behaviour of two animal population considering a predator-prey dataset. In the second part Neural Network and Singular Value Decomposition will be used to solve some famous ordinary differential equations and partial differential equations described in Section 3.4.

4.1. Find best Predator-prey model fitting

The initial given data set is composed by two time-series containing the historical data of two animal populations: Canadian Lynx and Snowshoe hare, respectively the predator and the prey, from 1845 to 1903 (Figure 4.1). The data set size is of thirty snapshots, with a time distance of two years.

4.1.1. Optimized Dynamic Model Decomposition with bagging and time delay embedding

The Optimized Dynamic Mode Decomposition (Opt-DMD) has been used to forecast the two population behaviour, therefore the data set has been split in two part: *training* (the first 20 snapshots) represented in gray in Figure 4.2 and *test* (the last 10 snapshots).

Considering that the data set is composed only by two time series, applying the standard Opt-DMD it is possible to describe the dynamical behaviour of the system with only two eigen

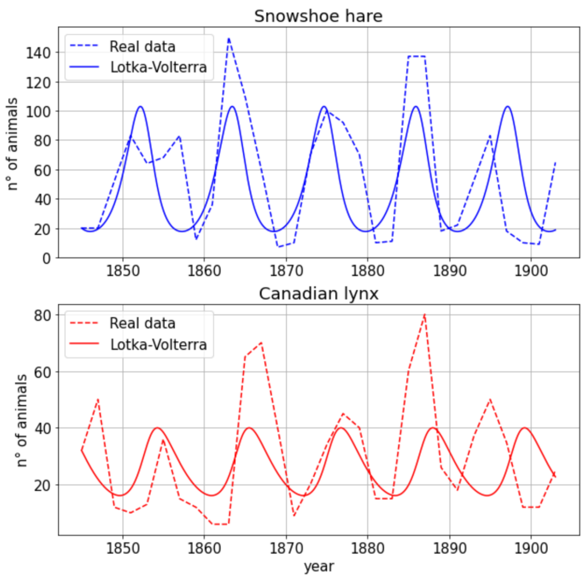


Figure 4.3: Lotka-Volterra model fitting to the initial data set.

values. As the eigen values are usually complex conjugate, considering the Equation (2.1.9), the resulting system behaviour can be described with maximum one frequency which strongly depends on the initial eigen values guess before fitting the Opt-DMD algorithm.

For examples if the initial eigen values guess has the imaginary part close to zero the fitted model follow more or less the average value (very slow dynamics) of the data as shown in the first plot in Figure 4.2.

On the contrary if the initial eigen values guess has an higher imaginary part, the Opt-DMD algorithm find the minimum close to that eigen values giving as a result a faster dynamics behaviour, more oscillations but around the zero axis. It is clear that all these components are required to better fit the model: the static component (slower dynamics) plus the higher dynamics components to describe the system oscillation around the mean. As a consequence more eigen values are required to describe the system behaviour. In fact the model obtained using BOP-DMD algorithm does not bring any big improvement on the prediction.

Time delay Opt-DMD enable to have multiple time series starting from a single time series (3.1.1), consequently it is a good solution to increase the number of eigen values achievable when the Opt-DMD is fitted to the data set. Each time series (Canadian Lynx and Snowshoe hare) has been time delayed three times obtaining a total number of 8 time series which generates 8 complex conjugate eigen values. As can been seen from the last two plots in Figure 4.2, within the training data set the model can perfectly follow the real data. On the new data the model is not so precise and applying bagging it improves a little bit the robustness of the model forecasting, in particular in the near time (first oscillation just after the training data set). Bagging has been applied by randomly picking the 60% of the training snapshots each time for 200 times.

4.1.2. Fit the best Lotka-Volterra parameters

In this case the entire dataset has been considered for fitting the model. Before the application of the algorithm explained in

Section 3.2 a linear data interpolation has been applied to the initial data set in order to have more snapshots and then to better fit the data (see Appendix A.1). As previously pointed out the fitting algorithm is very sensitive to the the initial condition, therefore to find a good fitting result which converges to the initial data set, the initial conditions listed in Section 3.2 have been used.

At the end of the optimization procedure the optimal Lotka-Volterra coefficients obtained are:

- $\alpha = 1.1102890304964905$,
- $\beta = 0.042296262728844354$,
- $\delta = 0.006303090082685032$,
- $\gamma = 0.30463300150365447$.

The graphical result is represented in Figure 4.3 and as can be seen the predicted model seems to follow quite well the first four cycles and then it starts to be not aligned with the real data set.

4.1.3. Fit the best SINDy model

As for Lotka-Volterra model fitting, the entire data set has been considered but in this case a third order *spline* interpolation has been applied to the initial data set with a new $dt = 0.1$ in order to have a more dense and smooth data. Moreover the dataset time axis has been shifted from zero to sixty.

The *SR3* is the used optimizer with a very low regularization threshold ($threshold = 1e - 20$). Bagging has been done on 500 models and each of them is composed by a sub data set composed by the 90% of the initial number of snapshots. The *pySindy Polynomial Library* has been used considering a maximum polynomial degree of six (see Appendix A.5). After the stable model selection algorithm (Section 3.3) only 81 model over 500 converge to a stable solution. Therefore the average polynomial coefficients have been calculated averaging only these 81 stable solutions.

In order to simulate the model, the *solve-ipv* Python function has been used with the Runge-Kutta ($RK45$) integrator and the simulation time has a small time discretization ($dt = 1e^{-3}$) in order to avoid integration error due to big numerical integration steps (see Appendix A.7).

The graphical result is represented in Figure 4.4.

4.2. Train Neural Network (NN) for Kuramoto-Sivashinsky (KS) equation forecasting with different random initial conditions

The Kuramoto-Sivashinsky equation (4.2.1) has been numerically solved to generate the training dataset for training a neural network. The space and time domain of the equation is small $x = 0 : 0.1 : 200$ and $t = 0 : 0.04 : 3.92$ so that the trainig could be faster. The total training dataset is composed by twenty queued up solved equations with different random initial conditions. The initial conditions are composed by a set numbers between zero and one generated using a random normal. The $\mathbf{X}$ and $\mathbf{y}$ matrices have been then prepared as explained in Section 3.4.

$$u_t + u_{xx} + u_{xxxx} + \frac{1}{2}u_x^2 = 0, \tag{4.2.1}$$

The neural network has been generated using $TensorFlow$ Python package and is composed by three Fully Connected layers (Dense layers). The input and hidden layers have a *ReLu* non linear activation function with an *L1* regularization while the output layer has a *linear* activation functions as in the most of the regression problems. The regularization terms have the role of increase the robustness of the model prediction on new data avoiding overfitting.
A new random initial condition has been then generated and then the KS equation has been solved using standard numerical method and using the trained neural network. The comparison of the two solutions is shown in Figure 4.5a. In Figure 4.5b the plot of the solution over time in the x position where the total error is lower and bigger. The total error has been calculated for each x_k position as:

$$Err_k = \sum_{t=0}^{T} |x^{(k)}_{t-true} - x^{(k)}_{t-NN}|. \tag{4.2.2}$$

Even at the point where the error is the lowest the prediction using the Neural Network is not so accurate, probably due to the initial transitory part which strongly depends on the initial conditions, but also a bigger network has probably needed to increase the prediction accuracy and robustness on new data. Moreover different network architectures seems to work well in PDE prediction as for example the *Recurrent Neural Network*.

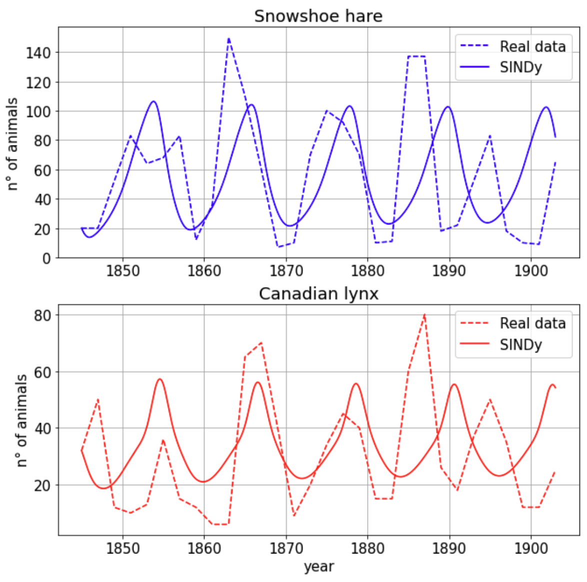
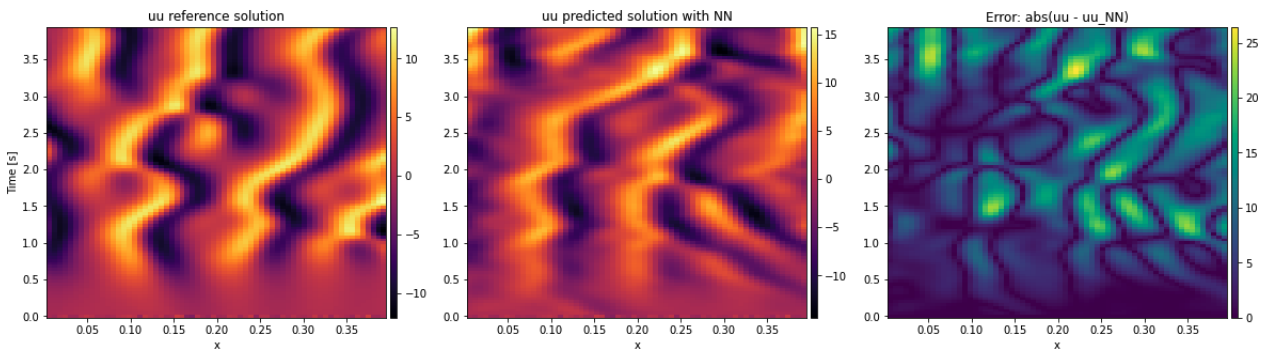


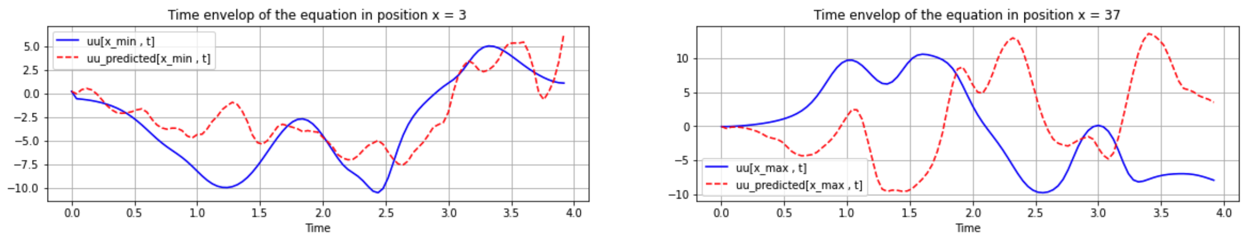
Figure 4.4: SINDy model fitting to the initial data set

4.3. Solving reaction diffusion equations projected in a lower dimensional space using Singular Value Decomposition and Neural Network

The reaction diffusion data set has been generated by using a given MATLAB script and then imported in a Python project. The dataset is composed by two tensors $\mathbf{U}_{512,512,201}$ and $\mathbf{V}_{512,512,201}$ with space dimension in x and y of 512 samples envelop along a time axis composed by 201 snapshots. The two tensors have been reshaped and concatenate in order to obtain a single data set containing both equations: $\mathbf{UV}_{512\times512\times2,201}$.
The Singular Value Decomposition is then been applied to the $\mathbf{UV}$ matrix and as pointed out in Section 3.4.1 and in Figure 3.1 the dynamical system can be described by 7 dimensions preserving the 99.5% of the system energy. As a consequence $\mathbf{vh}$ truncated matrix have been used to train a simple Neural Network (see Appendix A.11) after have generated the $\mathbf{X}$ and $\mathbf{y}$



(a) Kuramoto-Sivashinsky equation solution and relative Neural Network prediction. The error between the two solutions has been represented in the right figure.



(b) Predicted vs Actual solution over time considering x coordinates with the minimum (left) and maximum (right) total error over time.

Figure 4.5: Prediction of the Kuramoto-Sivashinsky equation using Neural Network.

matrices as explained in Section 3.4. The Neural Network used is composed by two fully connected hidden layers composed by 1000 neurons with a non linear *ReLu* activation function. The output layer has a *linear* activation function.

Adam is the chosen optimizer and after 200 epochs the loss has reached a very low value of $7.1230e^{-05}$. The prediction on the original variable space is very good as can be seen from Figure 4.6, moreover it can perfectly follow the equation also along the time axis (Figure 4.6b). The coordinates where the prediction is not able to follow the actual equation is probably because the values are very small considered noise when dimensionality reduction have been applied.

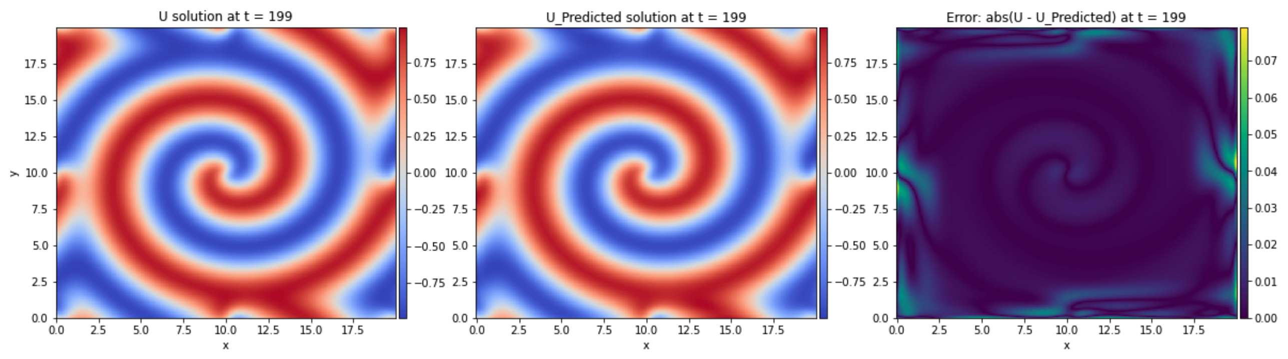
4.4. Train Neural Network (NN) for Lorenz equations forecasting

The data set has been generated by solving multiple times the Lorenz equations (3.4.2) using numerical integration method for different ρ values, specifically using $\rho = 10, 28, 35$. The dataset is composed by several concatenated solutions of the Lorenz equation with different random initial conditions. An additional column containing the ρ value has been added as explained in Section 3.4.2. As for the previous problems a simple Neural Network has been trained, it is composed by three fully connected layers, the two hidden layers have a depth of 10 neurons with a non linear *ReLu* activation function and the optimizer used is *Adam* optimizer.
The results on predict the Lorenz equations for $\rho = 17$ and $\rho = 40$ is not so precise, it is able to follow only the first time snapshots, and the prediction for $\rho = 40$ is better than the other one (see Figure 4.7).

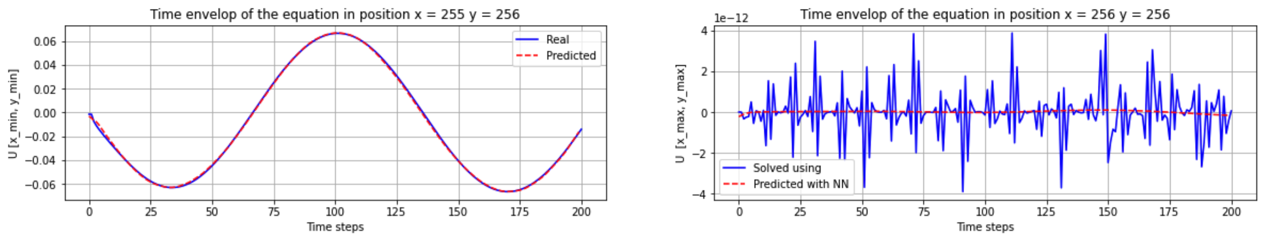
5. Summary and Conclusions

Working on this project many difficulties came out, first of all it emerged how it is difficult to forecast time series and to find a good model fitting real data. The tools used in this project are very powerful but also very sensitive to many parameters and to the data set conditions, therefore a lot of trials and errors are required to improve the model of the dynamical system.

- In the first part, consisting in finding the best fitting model on Canadian lynx and Snowshoe hare data set, the main problem was the small size of the data set. This problem has been well solved using linear and spline interpolation of the dataset before the application of the fitting algorithm (SINDy and Lotka-Volterra). But above all *bagging* technique has allowed to find a robust model improving the model prediction also for small data sets.
  Secondly a lot of parameters have been tuned in order to improve algorithms convergence which strongly depends on the chosen time step and on the model initial guess which must be as close as possible to the actual solution. In general, the models found fit quite well the training data set, but still the predictions on new data are difficult to obtain their precision decreases as much the forecasting time increases.
- In the second part of the project, the one regarding the use of Neural Network to calculate some Partial Differential Equations and Ordinary Differential Equations, the results are not so precise when the trained neural network has been tested on new initial conditions or on equations with different parameters. Probably a bigger network with more layers is required in order to recognise multiple patterns and also some different kind of network architectures can improve the results *Recurrent Neural Network*.
- Dimensionality reduction technique as Singular Value Decomposition (SVD) has turned out to be very effec-



(a) Reaction Diffusion u solution at time step $t = 199$ of the actual System Results and the Predicted one. The error calculated as the difference between the two (right).



(b) Predicted vs Actual solution over time considering x and y coordinates with the minimum (left) and maximum (right) total error over time.

Figure 4.6: Prediction of the Reaction Diffusion System using Neural Network in a lower dimensional space.

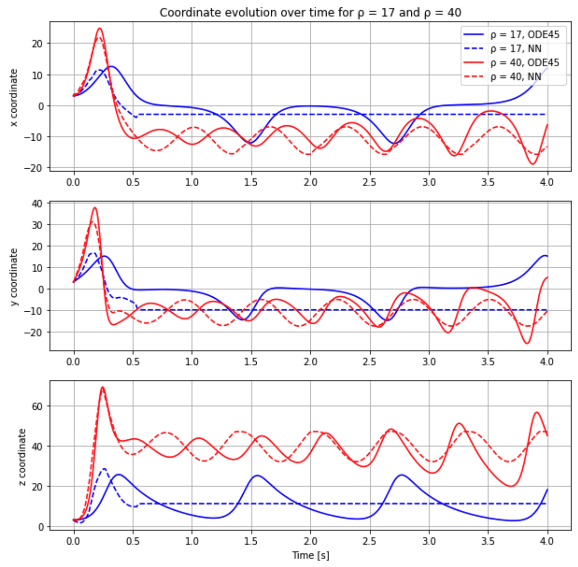


Figure 4.7: Prediction of the Lorenz equations using Neural Network for $\rho = 17$ and $\rho = 40$.

tive: the Neural Network trained on a lower dimensional space has been able to well predict the Reaction Diffusion equations with a small error even if it was trained in a way lower dimensional space. Moreover the trainig time was very fast and the network layers can be smaller because the number of features to recognise is smaller too.

6. Acknowledgements

I want to thank Professor J. Nathan Kutz for teaching the course with a lot of enthusiasm making the subject easy and pleasant to follow. I want also to thank the professor for his great helpfulness and support during the development of this project.

7. References

[1] Brunton, Steven L., and Jose Nathan Kutz. Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control. Cambridge University Press, 2022.

[2] Facundo Bre, Juan M. Gimenez, Víctor D.Fachinotti,Prediction of wind pressure coefficients on building surfaces using artificial neural networks,Energy and Buildings,Volume 158,2018,Pages 1429-1441, ISSN 0378-7788, https://doi.org/10.1016/j.enbuild.2017.11.045.

[3] FaselU,KutzJN,BruntonBW, Brunton SL. 2022 Ensemble-SINDy: Robustsparse model discovery in the low-data,high-noise limit, with active learning andcontrol.Proc.R.Soc.A478: 20210904.https://doi.org/10.1098/rspa.2021.0904

[4] Diya Sashidhar, J. Nathan Kutz (2021). Bagging, optimized dynamic mode decomposition (BOP-DMD) for robust, stable forecasting with spatial and temporal

[5] PySINDy, https://pysindy.readthedocs.io/en/latest/index.html.

[6] Lotka-Volterra equations, `https://en.wikipedia.org/w/index.php?oldid=610165245`

List of Figures

List of Algorithms

A. Source code

Partial source code. The entire source code used in this project has been store in a GitHub repository in the following link: `https://github.com/LeonardoIacussiPoli/Modeling-from-Measurements`.

A.1. Data set linear interpolation

```python
def interpolation(t ,X, t_new):
    f_0 = interpolate.interp1d(t, X[:,0])
    f_1 = interpolate.interp1d(t, X[:,1])
    X_new = np.zeros((len(t_new),2))
    X_new[:,0] = f_0(t_new)
    X_new[:,1] = f_1(t_new)

    return X_new
```

Listing 1: Data set linear interpolation.

A.2. Data set three points spline interpolation

```python
def interpolation(t ,X, t_new):
    f_0 = interpolate.splrep(t, X[:,0])
    f_1 = interpolate.splrep(t, X[:,1])
    X_new = np.zeros((len(t_new),2))
    X_new[:,0] = interpolate.splev(t_new, f_0)
    X_new[:,1] = interpolate.splev(t_new, f_1)

return X_new
```

Listing 2: Data set three points spline interpolation.

A.3. Bagging Optimized Dynamic Mode Decomposition

```matlab
for ii=1:number_of_bagging
    % randomly select
    index = randperm(n,k);
    index = sort(index);
    X_red = X_train(:,index);    % reduced matrix
    t_red = t(index);            % reduce time
    vector
    e_init = e;                  % initial guess
    update
    [w,e,b] = optdmd(X_red,t_red,r,imode, opts,
    e_init,[],[],[],proxfun);
    % sort eigen-values by imaginary part
    [sorted_imag, idx_imag] = sort(imag(e));
    time_series_matrix(:,:,ii) = real(w*diag(b)*
    exp(e*t));
    %store each iteration result
    e_total_imag(:,ii) = e(idx_imag);
    b_total_imag(:,ii) = b(idx_imag);
    w_total_imag(:,:,ii) = w(:,idx_imag);
end
```

Listing 3: BOB-DMD.

A.4. Lotka-Volterra loss function

```python
def loss_function(params, years,SH_pop, CL_pop):
    y0 = [SH_pop[0], CL_pop[0]]
    t0 = 0
    t = np.linspace(years[0], years[-1], num=len
    (years))
    output = odeint(sim, y0, t, rtol = r_tol,
    atol = a_tol , hmax = hmax , hmin = hmin ,
    args=(params,))
    loss = 0
    for i in range(len(years)):
        data_SH = SH_pop[i]
        model_SH = output[i,0]
        data_CL = CL_pop[i]
        model_CL = output[i,1]
        res = (data_SH - model_SH)**2 + (data_CL
   - model_CL)**2
        loss += res
    return(loss)
```

Listing 4: Lotka-Volterra loss function.

A.5. SINDy model fitting

```python
# SINDY model fitting
feature_names = ['x', 'y']
threshold = 1e-20
alpha = 0
n_models = 500
subset_size = X.shape[0] - int(X.shape[0] *
    0.10)
max_iter = 1000
ensemble_optimizer = ps.SR3(threshold =
    threshold ,max_iter= max_iter)
model = ps.SINDy(feature_names=feature_names,
    optimizer=ensemble_optimizer,
    feature_library=ps.PolynomialLibrary(6,
    library_ensemble=True))
model.fit(X, t=dt, ensemble=True, n_models=
    n_models, n_subset=subset_size, quiet=False
    , n_candidates_to_drop=4)
```

Listing 5: Lotka-Volterra loss function.

A.6. SINDy stable models selection

```python
# function to zero out any short-term unstable
    models
def integration_metric(model,x0_test, t_test,
    coef_list, optimizer,integrator,
    integrator_kws):
    stable_list = np.zeros((np.shape(coef_list)
    [0]),dtype='int')
    print(stable_list.shape)
    kk = 0

    for k in range(1):
        for i in tqdm(range(np.shape(coef_list)
    [0])):
            optimizer.coef_ = coef_list[i, :, :]
            x_test_sim = model.simulate(x0_test,
     t_test, integrator=integrator ,
    integrator_kws = integrator_kws)

            var_1 = np.var(x_test_sim[int(0.33*
    len(x_test_sim[:,0])):-1,0])
            var_2 = np.var(x_test_sim[int(0.33*
    len(x_test_sim[:,0])):-1,0])

            if ((var_1 < 200) or (var_2 < 200)):
                coef_list[i, :, :] = 0.0
            elif np.any(np.abs(x_test_sim) >
    200):
                coef_list[i, :, :] = 0.0
            elif(np.any((x_test_sim) < 0)):
                coef_list[i, :, :] = 0.0
            else:
                stable_list[kk] = i
                kk = kk+1

    print(kk)
    stable_list = stable_list[0:kk]
    stable_list = stable_list.tolist()
    return coef_list , stable_list
```

Listing 6: SINDy algorithm for stable model selection.

A.7. SINDy model fitting

```
# set initial conditons
x0_test = X[0,:]

# set integrator: solve_ivp , odeint
integrator = 'solve_ivp'

# set integration  parameters
integrator_kws = {}

# Only for solve_ivp
# RK45 , RK23 , DOP853 , Radau , BDF , LSODA
if integrator == 'solve_ivp':
    integrator_kws['method'] = 'RK45'
    #integrator_kws['max_step'] = 1e20
    #integrator_kws['min_step'] = 1e15
else:
    print()
    #mxstep = 1e20
    #integrator_kws['hmax'] = 10
    #integrator_kws['hmin'] = 10

integrator_kws['rtol'] = 1e-8
integrator_kws['atol'] = 1e-8

dt_test = 1e-3
t_test = np.arange(0,58,dt_test)

# function to zero out any short-term unstable
    models
stable_ensemble_coefs , stable_list =
    integration_metric(model,x0_test,t_test,np.
    asarray(ensemble_coefs), ensemble_optimizer
    ,integrator,integrator_kws)
```

Listing 7: SINDy model simulation

A.8. Neural Network trainig procedure

```
trainX = uu_train_X
trainy = uu_train_Y

regularizers = False        % regularizers
    activation flag

if regularizers == True:
  # Build model
  deep_approx = tf.keras.models.Sequential()
  deep_approx.add(tf.keras.layers.Dense(600,
    input_dim=uu_train_X.shape[1], activation='
    linear', kernel_regularizer=tf.keras.
    regularizers.L1(0.01)))
  deep_approx.add(tf.keras.layers.Dense(600,
    activation='linear', kernel_regularizer=tf.
    keras.regularizers.L1(0.01)))
  deep_approx.add(tf.keras.layers.Dense(
    uu_train_X.shape[1], activation='linear'))

else:
  # Build model
  deep_approx = keras.models.Sequential()
  deep_approx.add(tf.keras.layers.Dense(100,
    input_dim=uu_train_X.shape[1], activation='
    relu'))
  deep_approx.add(tf.keras.layers.Dense(100,
    activation='relu'))
  deep_approx.add(tf.keras.layers.Dense(
    uu_train_X.shape[1], activation='linear'))

decayRate = 1e-4
nrSamplesPostValid = 2
learningRate = 3e-3
nEpochs = 500
batchSize = 128
verbosity = 1

# optimizer ex. adam or SGD
adam = tf.keras.optimizers.Adam( learning_rate =
     learningRate, decay = decayRate )

# Compile model
deep_approx.compile(loss='mse', optimizer=adam)

# Training procedure with cross-validation
History = deep_approx.fit(trainX, trainy, epochs
    =nEpochs)
```

Listing 8: Neural Network trainig procedure.

A.9. PDE and ODE forecasting using Neural Network

```
#generate reference data for comparison
uu, x , tt = generate_data(rng_mnumber = 200)
# initialize the matrix for store theNN
    prediction
uu_NN = np.zeros_like(uu)
uu_NN[0,:] = uu[0,:]
#advance the solution snapshot by snapshot for
    the entire pre-defined time domain
for ii in tqdm(range(1,len(tt))):
  uu_NN[ii,:] = deep_approx.predict(np.reshape(
    uu_NN[ii-1,:],(1,uu_NN.shape[1])))
```

Listing 9: PDE and ODE forecasting using Neural Network.

A.10. Reshape and concatenate PDE matrices

```
U_reshaped = np.reshape(U,(len(x)*len(y),len(t))
    )
V_reshaped = np.reshape(V,(len(x)*len(y),len(t))
    )
reshaped_equation = np.concatenate((U_reshaped,
    V_reshaped),axis=0)

# Singular Value Decomposition (SVD)
u, s, vh = np.linalg.svd(reshaped_equation,
    full_matrices=False)
```

Listing 10: Reshape and concatenate PDE matrices.

A.11. PDE and ODE forecasting using Neural Network and a lower dimensional space for the data set

```
# rank truncation
vh_denoised = (vh[range(r),:]).T

# prepare dataset for training
train_X =  vh_denoised[0:-1,:]
train_Y =  vh_denoised[1:,:]

# build the model
deep_approx = tf.keras.models.Sequential()
deep_approx.add(tf.keras.layers.Dense(1000,
    input_dim=train_X.shape[1], activation='
    relu'))
deep_approx.add(tf.keras.layers.Dense(1000,
    activation='relu'))
deep_approx.add(tf.keras.layers.Dense(train_Y.
    shape[1], activation='linear'))

decayRate = 1e-4
nrSamplesPostValid = 2
learningRate = 1e-4
nEpochs = 200
batchSize = 128
verbosity = 1

# Compile model
adam = tf.keras.optimizers.Adam( learning_rate =
     learningRate, decay = decayRate )
```

```
deep_approx.compile(loss='mse', optimizer=adam)

# train the model
History = deep_approx.fit(train_X, train_Y,
    epochs=nEpochs )
```

Listing 11: PDE and ODE forecasting using Neural Network and a lower dimensional space for the data set.