TECHNISCHE UNIVERSITÄT DRESDEN

Neural Network for Model Identification from Noisy Spatio-temporal Data

MASTER THESIS

This work is submitted to
Chair of Scientific Computing for Systems Biology
under faculty of Computer Science

by

Happy Rani Das

Course: Computational Logic

1st Reviewer & Advisor: Prof. Dr. Ivo F. SBALZARINI
Chair of Scientific Computing for Systems Biology
TU Dresden, Dresden

Co-Supervisor: MSc. Suryanarayana MADDU
Center for Systems Biology Dresden

2nd Reviewer: Prof. Dr. Sebastian RUDOLPH
Chair of Computational Logic
TU Dresden, Dresden

Dresden, 25th June 2020

Declaration of Authorship

Das, Happy Rani — 4576505

Name, First Name — Matriculation number

I, Happy Rani Das, hereby declare that I have written this thesis entitled "**Neural Network for Model Identification from Noisy Spatio-temporal Data**" independently and have used no sources other than those specified. All references and verbatim extracts have been quoted, and all the sources of information have been specifically acknowledged.

Furthermore, I declare that all rights in this thesis are delegated to Chair of Scientific Computing for Systems Biology under the Faculty of Computer Science of Technische Universität Dresden.

This work has not yet been submitted to another examination institution – neither in Germany nor outside Germany – neither in the same nor in a similar way and has not yet been published.

Dresden, 25th June 2020 — Happy

Place, Date — Signature

Abstract

Partial differential equations (PDEs) are one of the most popular forms of mathematical models used to describe Spatio-temporal processes occurring in nature. Recently, there has been a huge surge of interest in the machine learning community for the inference of PDEs directly from observational data. In this work, we explore the popular DeepMOD (Deep Learning-based Model Discovery) algorithm to discover partial differential equations from noisy Spatio-temporal data-sets. In this regard, we systematically evaluate the robustness of the DeepMOD algorithm for different choice of hyper-parameters and produce achievability plots to study the consistency of the algorithm for several sample sizes. Initially, some investigations have been targeted to analyze dependencies on the number of iteration, sample-size, noise-levels, and tuning parameters to enable consistent model recovery. We present numerical experiments on 1D Burgers and 2D-Advection-Diffusion equations by adding Gaussian noise for discovering the governing PDEs.

Acknowledgements

Foremost, I wish to express my gratitude to both of my examiners Prof. Dr. Ivo F. Sbalzarini, Chair of Scientific Computing for Systems Biology and Prof. Dr. Sebastian Rudolph, Chair of Computational Logic at TU Dresden; for the opportunity of doing this research work.

Besides my examiners, I would like to thank my thesis co-supervisor Suryanarayana Maddu from Center for Systems Biology Dresden. The door to his office was always open whenever I had difficulties in understanding and questions regarding my research or writing. He allowed me to do work independently but steered me in the right direction whenever he thought I needed it. In addition, Mr. Maddu has supported me not only academically but also emotionally through the rough road to finish this thesis.

I must express profound gratefulness to friends and family, especially to my elder brother Bidhan Das, for providing me unconditional support and continuous encouragement during my entire period of study since childhood. Additionally, my thanks go to my parents and parents-in–law for their great role in my life and their numerous sacrifices for me. Finally, I want to thank my husband who stood besides me and supported me all the way to carry out this incredible journey. They have cherished with me every great moment and this accomplishment would not have been possible without them.

Contents

List of Abbreviations

ANN	**A**rtificial **N**eural **N**etwork
FFNN	**F**eed **F**orward **N**eural **N**etwork
MSE	**M**ean **S**quared **E**rror
MAE	**M**ean **A**bsolute **E**rror
PDE	**P**artial **D**ifferential **E**quation
RNN	**R**ecurrent **N**eural **N**etwork
SD	**S**tandard **D**eviation

1 Introduction

Background

Nowadays, with the emergence of modern technologies, high-quality data are available in plenitude. About 2.5 Quintillion bytes of data are generated every single day and 90% of these data are created in the last two years [1]. Millions of users create data by smart sensors on social media like Facebook, Twitter, and so on, where those data can be effectively stored using computational power in the data storage. A vital concern of the researcher from the industrial and scientific sector is how to transform the data in the real world to extract patterns which can give acumen in the mathematical physical model.

Due to the abundant growth of sufficient data and computing systems, current advances in deep learning and data sciences have yielded remarkable results in the last fifteen years over several scientific disciplines and engineering, including graphics visualization [2], natural language processing [3], computer vision [4], cognitive science [5] and so on. Despite the tremendous success in those similar fields, deep learning has not yet been widely used in complex physical or biological systems. Recently, data-driven science with physical modeling [6] via deep learning has emerged successfully in fluid mechanics [7, 8] and material science [9]. The data-driven methods have been investigated in several works and used in the mathematical physics for solving dynamical systems [10] like different types of differential equations, especially partial differential equations (PDE). PDEs play a significant role in different areas, from physics to economics, epidemiology, neuroscience, computer science, applied mathematics, chemistry, etc.

Generally, PDEs are obtained based on simple physical laws such as conservation of mass, energy, and momentum for a control volume in fluid mechanics, phenomenological behaviors, and so on [11, 12]. Examples of some popular PDEs are Burgers equation, Navier- Stokes equations in fluid dynamics, and Kuramoto-Sivashinsky Equation. It is mentioned in the PDE-STRIDE [13], the PDEs are considered for the state variable $u \in \mathbb{R}^d$ for dimension d, consists of polynomial functions (e.g. u^2, u^3), spatial derivatives (e.g. u_x, u_{xx}, uu_x) and the parameter $\mu \in \mathbb{R}$.

The parameterized and non-linear PDE of the general form is given below:

$$u_t = F(u, u^2, u^3, u_x, u_{xx}, uu_x..., x, \mu) \tag{1.1}$$

where $F(.)$ is a function consists of spatial derivatives, non-linear polynomials, combinations of both, and parameters in μ for a one-dimensional (d = 1) Spatio-temporal data. The subscripts denote partial differentiation in either time or space [14, 15]. Recently, sparse regression techniques have shown promising success in identifying the governing partial differential equations from a Spatio-temporal dataset in several problems [16–18]. The sparse regression technique is used here to determine which right-hand-side terms are most informative from the large dictionary of potential candidate functions to infer PDE. For example, the Burgers equation ($F = \mu u_{xx}$ - uu_x) has only two informative terms. As described in the PDE-FIND algorithm [18], the goal of the sparse regression method is to find the simplest model that best describes the data. Given a randomly selected clean or noisy Spatio-temporal dataset $u(x, t)$, the data-driven discovery of PDE is identified by formulating the Equation 1.1 into the regression problem.

$$u_t = \Theta\xi \tag{1.2}$$

where u_t is the derivative of state variable u with respect to time t, Θ is the dictionary matrix containing polynomial and spatial derivative functions, and ξ is an unknown coefficient vector of the corresponding terms of the dictionary matrix.

In the beginning, the dictionary matrix Θ is constructed by taking polynomial and spatial derivatives of the state variable u, which represents a potential feature that explains the intrinsic dynamics of the Spatio-temporal data [13, 18, 19]. For instance, the following equation can be written for the two-dimensional problem:

$$\Theta(u) = [1, u, u^2, ...u_x, u_y, u_{xx}, u_{xy}, u_{yy}, ...] \tag{1.3}$$

Here each column of the dictionary matrix Θ contains all of the possible terms for several orders of derivative of u with respect to the spatial variables x and y. The non-linear polynomial functions are u, u^2 etc. and spatial derivatives are u_x, u_y, u_{xx}, u_{xy}, u_{yy} etc. All of the Spatio-temporal data are collected into a single column vector $u \in \mathbb{R}^{m*n}$, where m represents time points and n represents the spatial locations.

Dictionary matrix Θ is an overcomplete list meaning that more terms are required than needed to discover the governing PDE. Only few terms contain informative dynamics, and by summing up them, it should be possible to recover the underlying governing PDEs. Thus, in order to recover some non-zero coefficients, in PDE-FIND

[18], Rudy et al. proposed the Sequential Threshold Ridge Regression (STRidge) algorithm. By using this algorithm, it was possible to recover the most desirable sparse coefficient vectors ξ that have non-zero values, and the rest of the coefficients have zero values.

Given a dictionary matrix Θ and time derivative of u, the sparse coefficient vectors are discovered by solving the non-convex optimization problem.

$$\vec{\xi} = argmin_{\xi} ||\Theta\xi - u_t||_2^2 + \lambda||\xi||_0 \tag{1.4}$$

Here, a complexity parameter λ is applied to the L_0-norm of the coefficients. When λ increases, the overfitting problem gets reduced, and most of the coefficients shrink towards zero. From the dictionary of potential candidate models, the regression method identifies the non-linear polynomial and spatial derivative functions that most accurately represent data. The coefficients which are most informative of the dynamics are recovered to constitute PDE.

Previous sparse identification algorithms (SINDY) [16] faced several problems to recover coefficients of PDEs. For instance, the authors were not able to handle the subsampled spatial data, and the algorithm did not scale nicely to 2D and 3D measurements. The PDE-FIND [18] algorithm has been tested on the different physical problems, such as the Kuramoto-Sivashinsky, KdV equation, Reaction-diffusion equation, Navier Stokes equation. It is able to effectively handle 1D, 2D, and 3D Spatio-temporal data. However, the drawback of the algorithm is that it is not capable of recovering PDE with a small number of training samples. Moreover, it is noise-sensitive too.

To overcome the limitation of PDE-FIND, Gert-Jan Both et al. [20] proposed DeepMoD: Deep Learning for Model Discovery in noisy data. Instead of using numeric differentiation, they used automatic differentiation inside the neural network to calculate the derivatives of state variable u with respect to x and t for constructing the dictionary matrix Θ. They tested their method on Burgers equation, Korteweg-de Vries (KdV) equation, 2D Advection-Diffusion, Keller-Segel equation, and noisy time-series data from the electrophoresis experiment.

The performance evaluation of the Burgers equation was done by considering different samples size and Gaussian noise levels. The algorithm outperformed on ID data. After having a successful implementation of 1D data, they applied the DeepMoD algorithm on 2D data to test the robustness of the model. The Advection-Diffusion equation has been considered, and the model recovered the PDEs correctly. Finally, they applied DeepMoD on time-series images from the gel electrophoresis experiment to evaluate the performance of it. It was also possible to infer PDE from

time-series images.

Motivation

The partial differential equation is widely used in different quantitative fields, from physics to engineering, in an attempt to explain the evolution and dynamics of phenomena. However, in the field of biology, finance, economy, or neuroscience, the Spatio-temporal dynamical system is not widely used as the datasets are often noisy and difficult to analyze through traditional means. For that, we realize the need for the automation of such processes to recover those equations. The aim of this thesis is to learn the behavior of the existing DeepMoD technique [20], which is able to discover the governing PDEs from noisy Spatio-temporal data. The original DeepMoD algorithm was implemented on the Tensorflow framework, whereas we have explored it by using the PyTorch framework. We explore the Neural Network-based technique to learn the mapping from spatiotemporal domain (x, t) to output variable u. Within the neural network, the dictionary matrix is constructed by taking the polynomial and spatial derivatives of the predicted output, and the total loss is calculated to denoise the dataset and to infer the governing PDEs. We benchmark our model on physical problems such as the Burgers equation and the 2D-Advection-Diffusion equation.

Outline of the thesis

The rest of this thesis report is structured as follows; an overview of the Neural Network is discussed in Chapter 2. In the following chapter, the methodology of the project is described. The results and the discussion of the work are summarized in Chapter 4. Finally, the report is concluded with Chapter 5 where future work is discussed.

2 Neural Network

Deep Learning is a sub-field of a broader machine learning family that has powerful learning ability from data. It contains several layers of representation and performs computations to learn automatically from training data [21]. The computations are proceeded by a model, called an artificial or deep neural network.

An artificial neural network (ANN) in short neural network is a computational model, which consists of highly interrelated processing elements called neurons or nodes, working together with their incredible capability to derive meaning from imprecise or complex data [22]. ANN is capable of solving numerous complicated real-world problems in the areas of trend detection, pattern recognition, classification, prediction, clustering, image processing, and many more. ANN is an information processing system that is inspired by the biological counterparts such as the biological nervous systems and brains. In contrast, it does not process the function of the biological system. Instead, it performs high parallelism, the non-linear relationship between input and output, learning, fault tolerance, robustness and capability to handle imprecise information [23].

2.1 Biological Neuron

The biological neuron is the fundamental building block of the neural network that helps information transfer from one nerve cell to another. Its functionality is described here first to understand the artificial neuron in a better way. The human nervous system comprises millions of cells of several types and length called neurons. The symmetric diagram of a biological neuron is shown in Figure 2.1. It consists of three principal functional components, namely – soma or cell body, axon, and dendrites. The cell body or soma comprises a nucleus and organelles for building proteins and processes sugar and oxygen. The dendrites work as an input unit that receives information from other neighboring neurons and sends it to the cell body. In contrast, axons receive information from the cell body and pass them to the other adjacent neurons.

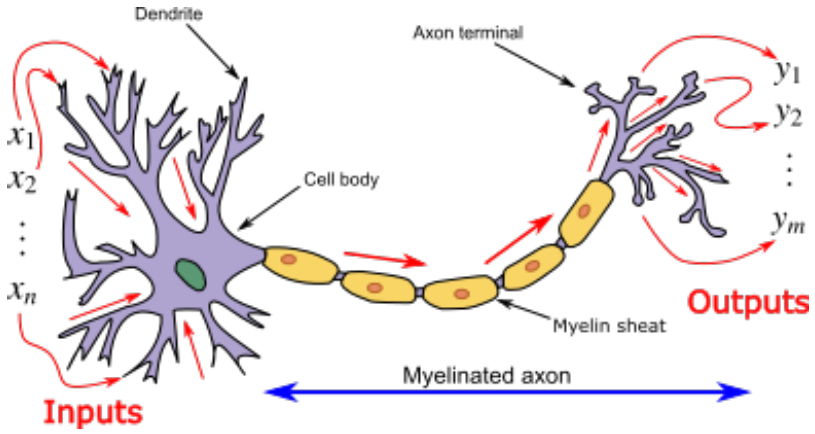

FIGURE 2.1: Biological neuron and myelinated axon, with signal flow from the input layer dendrites to the output layer axon terminals [24]

2.2 Artificial Neuron

The artificial neuron is the elementary unit of the neural network. It is a mathematical function, and its functionality and design are derived from the biological neuron. The schematic diagram of an artificial neuron is shown in Figure 2.2 . It is constructed with inputs, bias, weights, activation functions, and outputs.

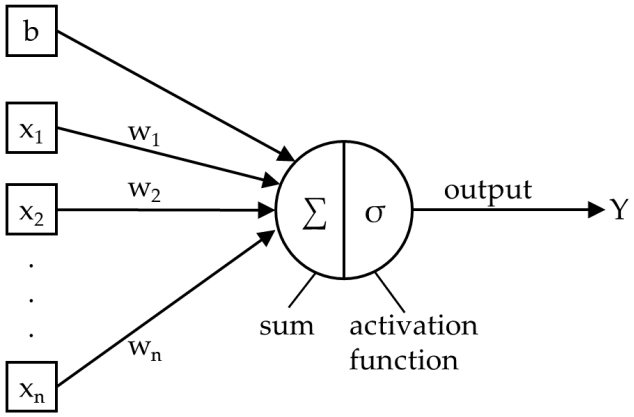

FIGURE 2.2: Artificial Neuron.

The information comes into the neuron via inputs, where inputs are multiplied by their corresponding weights. Then, the weighted inputs and bias are summed up for further processing. The activation function is applied to have the output value of a neuron to a desired limited boundary value. Activation function or transfer function is a function that restricts the output value of a neuron into a limited range like

(0,1) or (-1,1) etc. [25]. The most frequently used activation functions are Sigmoid or Logistic Activation Function, Softmax, Tanh, and ReLU (Rectified Linear Unit) Activation Function. The mathematical formula of an artificial neuron can be expressed by following Equation 2.1.

$$y = w_1x_1 + w_2x_2 + ... + w_nx_n + b \tag{2.1}$$

where x_1, x_2 .... x_n are the input values; w_1, w_2 .... w_n are the respective weights of the input values, b is the bias and y is the predicted output. The following equation is the generalized form of Equation 2.1.

$$y = \sum_{i=1}^{n} w_i x_i + b \tag{2.2}$$

$$Y = \sigma(y) \tag{2.3}$$

Here Y is the final output (Equation 2.3). The sigmoid activation function σ squashes the output value of the neuron in the range (0, 1). Based on the requirement, one can use alternative activation functions too.

2.3 Neural Network Architecture

Based on the connection between nodes, the neural network architecture can be categorized into two main categories: the feed-forward neural network (FFNN) and the recurrent neural network (RNN) [26]. When there is no connection to send feedback from the outputs of the neurons to the inputs, then it is called the FFNN [27]. On the other hand, if there is a connection from outputs of the neurons to inputs for feedback, then it is referred to as RNN. The inputs are either the same neuron's input or the other neuron's inputs. A standard feed-forward neural network with one hidden layer is shown in Figure 2.3. The network is fully connected to the previous layer of neurons but not connected to the same layer of neurons. Each connection has a numeric value called weight. The first layer of the neural network is called the input layer, which contains one or more neurons. At first, the hidden layer is computed by taking the input values with their respective weights. The output of the hidden layer is connected to the next layer, referred to as the output layer, which computes the final results. The input layer not counted as no mathematical computation is needed for this layer.

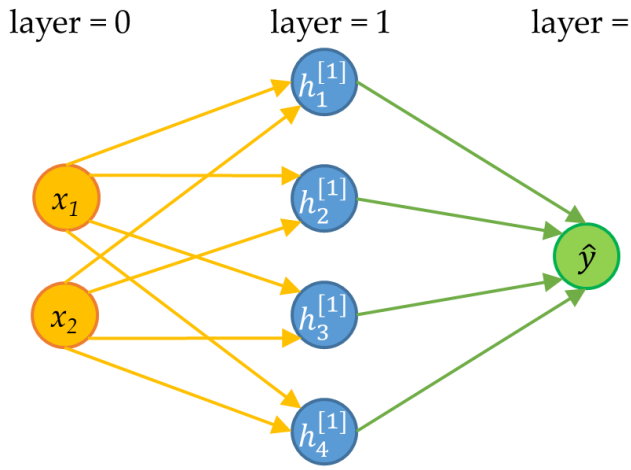

FIGURE 2.3: A feed-forward neural network with one hidden layer

2.4 Training Neural network

A neural network contains at least one hidden layer between the input and output layers. Using one or more hidden layers in a neural network helps the model to extract useful complicated features from the input data. The procedure of capturing hidden information from data is called training the neural network or learning the neural network [28]. A feed-forward neural network can learn from its surroundings and enhance the performance by the capacity of learning. Learning is an iterative process that adjusts the parameter weights and biases of a feed-forward neural network from the training data [26]. The technique of fine-tuning the parameters from the input data is called learning the neural network or training the neural network.

2.4.1 Backpropagation

Backpropagation is the most common and widely used algorithm for learning or training a neural network. It is the procedure to repeatedly adjust the weights and biases to minimize the loss function. The loss function is the difference between the actual output and the predicted output of the neural network. Each repetition of the entire training set is called an epoch. In every epoch, the network updates the weights and biases in the direction that reduces the loss function. Each repetition in backpropagation consist of two passes: forward and backward pass [29]. Forward pass computes the output, whereas backward pass adjusts the weights and biases to reduce the error in the loss function.

Forward Pass

The forward pass calculates the final output. At first, it computes the hidden layer output [30]. Then, the hidden layer output is fed into the output layer to calculate the final output of the neural network. The equation for calculating the output for l layers and n neurons is given below:

$$h_n^{[l]} = tanh(w_n^{[l]}.h_n^{[l-1]} + b_n^{[l]}) \tag{2.4}$$

The superscript inside the square brackets represents the layer and the subscript denotes the neurons on the layer. Here, $tanh$ is the activation function that squashes the output within the range (-1, 1). For m samples, the dimension of the previous hidden layer and weight matrix represent by $h_n^{[l-1]} \in \mathbb{R}^{n^{[l-1]}*m}$ and $w_n^{[l]} \in \mathbb{R}^{n^{[l]}*n^{[l-1]}}$, respectively, where $n^{[l]}$ and $n^{[l-1]}$ denotes number of neurons in layer l and layer $(l-1)$. The dimension of the bias vector denote by $b_n^{[l]} \in \mathbb{R}^{n^{[l]}*m}$ and the dimension of output is same as the bias vector.

After calculating the final output, the loss function L is calculated by using Mean Squared Error (MSE) [31]. The MSE is the average of the squared difference between the actual and predicted output, is given below:

$$L = \frac{1}{m}\sum_{i=1}^{m} |y - h_n^{[l]}| \tag{2.5}$$

where, the actual and predicted outputs are y and $h_n^{[l]}$; and m is the number of examples.

Backward Pass

The key to backward pass is to calculate the derivative of the loss function with respect to the various weights and biases [29]. The gradient descent is used to adjust the parameter weights and biases by propagating error through the backward direction. It is an optimization algorithm, used to learn the value of the parameter that minimizes the loss function [32]. It updates the parameter in the opposite direction to find the steepest decrease of the loss function. The procedure is repeated in every layer for all the neurons and calculates the derivatives of the loss function. The derivative of the loss function is taken for all the layers with respect to weights and biases and it is given below:

$$\frac{\partial(L)}{\partial w_n^{[l]}}, \frac{\partial(L)}{\partial b_n^{[l]}}$$

The parameters weights and biases are updated by using the Equation 2.6 and 2.7 respectively, where α is the learning rate, which denotes the step size of the movement towards a minimal point.

$$w_n^{[l]} := w_n^{[l]} - \alpha \frac{\partial(L)}{\partial w_n^{[l]}} \tag{2.6}$$

$$b_n^{[l]} := b_n^{[l]} - \alpha \frac{\partial(L)}{\partial b_n^{[l]}} \tag{2.7}$$

Adaptive Moment Estimation (ADAM) Optimizer

Adaptive Moment Estimation (ADAM) is another optimization algorithm that calculates adaptive learning rates for the parameters from the estimates of the first and second moments of the gradients [33]. It takes the advantages of two popular methods; AdaGrad [34] and RMSProp [35]. The update rule of exponentially decaying past gradients is given below:

$$m_t = \beta_1 . m_{t-1} + (1 - \beta_1) . g_t \tag{2.8}$$

$$v_t = \beta_2 . v_{t-1} + (1 - \beta_2) . g_t^2 \tag{2.9}$$

where, m_t and v_t are first moment (the mean) and second moment (the uncentered variance) of the gradients respectively. The hyperparameters are β_1 and β_2 and g_t is the gradient of the objective with respect to the parameter at timestep t [33]. The first and second-moment estimates, m_t and v_t are initialized as (vectors of) 0's, the authors observe that they are biased towards zero, especially during the initial time-steps and especially when the decay rates are low (i.e. β_1 and β_2 are close to 1).

The bias-corrected first and second-moment estimate is given below:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{2.10}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_1^t} \tag{2.11}$$

The ADAM rule to update weights $w_n^{[l]}$ is given below:

$$w_n^{[l]} := w_n^{[l]} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{2.12}$$

The authors [33] proposed the default values of β_1, β_2 and ϵ and they are 0.9, 0.999 and 10^{-8} respectively.

3 Methodology

3.1 Graphical Abstract

The core architecture of the thesis work is described in Figure 3.1 below, where Burgers and Advection-Diffusion datasets have been considered for the experiment. Here, a neural network based technique is explored, which takes the space and time coordinates of the Burgers and Advection-Diffusion datasets as input and produces state variable $\hat{u}$ as a predicted output. After that, the dictionary matrix is constructed by calculating the polynomial and spatial derivatives of the predicted output. And then, the loss function is calculated to denoise the noisy data and to discover the governing

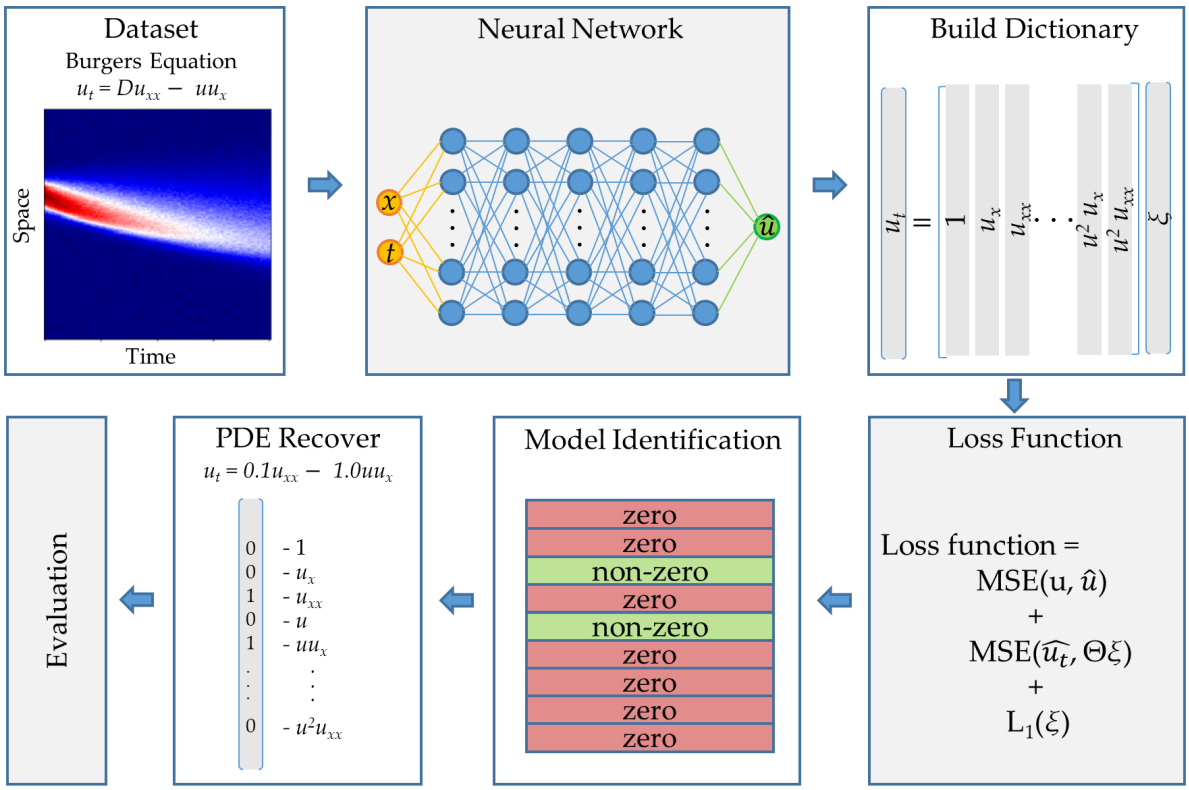

FIGURE 3.1: Schematic outline for deep learning based model identification (DeepMOD) from noisy data

PDEs. Finally, normalization and thresholding are applied to recover the true sparsest coefficient vectors for identifying the underlying PDEs.

3.2 Dataset

As our approach applies to a broad range of physical problems, we have considered the Burgers equation and the 2D-Advection-Diffusion dataset. Both are described below in detail.

3.2.1 Burgers Equation

Burgers equation is a basic nonlinear partial differential equation derived from the Navier-Stokes equation for the velocity field. It appears in several areas of physical worlds and applied mathematics, including gas dynamic, fluid mechanics, traffic flow modeling, and nonlinear acoustics [14, 36, 37]. It consists of a nonlinear polynomial item and second-order spatial derivatives. The general form of the Burgers equation is given below:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = D\frac{\partial^2 u}{\partial x^2} \tag{3.1}$$

Here $u(x,t)$ is the velocity and D is the Diffusion coefficient or viscosity. The value of diffusion coefficient D is 0.1.

The Burgers equation is solved numerically for 256 spatial steps between the interval [-8 to 8] and 101 time steps between 0 to 10 total of 25856 steps. The solution

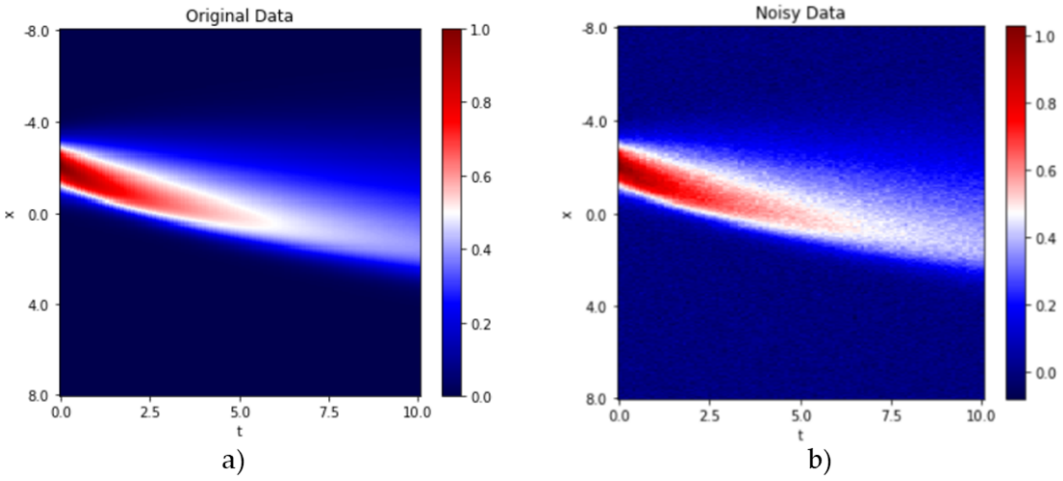

FIGURE 3.2: Burgers data visualization: a) Original and b) after adding 10% Gaussian noise

is visualized in time and space in Figure 3.2, where the time t is in the x-axis, and space x is in the y-axis. As our motivation is to identify the PDEs from the noisy data, understandably we have added noise to the dataset. Noise is added on the dataset using a standard deviation (σ) scale. The equation to add noise to the given dataset u in space and time is given below:

$$u_{noisy} = u + \eta\sigma(u)N(0,1) \tag{3.2}$$

where η denotes noise level ranging from 1% to 25%, and N is a Gaussian distribution. The graphical visualization is shown in Figure 3.2b after adding 10% Gaussian noise.

3.2.2 2D-Advection-Diffusion

The 2D-Advection-Diffusion is used to show the robustness of the neural network method on high dimensional spatiotemporal data. The general form of the 2D-Advection-Diffusion equation is given below:

$$u_t = -\nabla.(-D\nabla u + \vec{v}.u) \tag{3.3}$$

Here $\vec{v}$ is the velocity vector representing the advection, and D is the viscosity or diffusion coefficient. The value of advection coefficient $\vec{v}$ is 0.25 and 0.5 in x and y direction respectively, and the diffusion coefficient D is 0.5 [20]. The noise is also added here by using the same Equation 3.2.

3.3 Feed-Forward Neural Network

A deep feed-forward fully connected neural network is exploited to approximate the spatio-temporal problem solution. The neural network consists of an input layer, five hidden layers and the output layer. Each hidden layer contains twenty neurons or nodes. For the Burgers equation, the network takes x and t as input and results in $u(x,t)$ as output. On the contrary, for the 2D-Advection-Diffusion equation x, y, and t are considered as inputs and it produces output $u(x,y,t)$.

The neural network architecture is explored with deep learning framework PyTorch that performs efficiently computations of tensors like matrix multiplication, softmax to classify multiple classes, automatic differentiation and many more [38]. The steps to build a neural network with PyTorch is described below:

1. In PyTorch, the "nn" package describes a couple of modules that can be easily applied to build neural network layers [38]. Torch.nn module is the base class for all of the modules.

2. A constructor is created to prepare the module for invocation and to initialize a couple of parameters and submodules. The nn.Module is combined with super().__init__(), creates a class for providing a method and attributes.

3. The nn.Linear method is used to create a linear transformation. This module automatically creates weights and bias to compute the forward function. The neural network is created with PyTorch nn.module that should have a defined forward function, which takes tensor x to send it to the defined __init__ method.

4. The nn.sequential method contains other methods and applies them sequentially in operation to produce output. The selected activation function should be differentiable many times for the higher-order derivatives, and the ReLu activation function might not compute the higher-order derivatives. Given this, we choose tanh, which is infinitely differentiable.

The implementation code is given below:

```
import torch
import torch.nn as nn
class PINN(nn.Module):
def __init__(self, sizes, activation=nn.Tanh()):
    super(PINN, self).__init__()
    layer = []
    for i in range(len(sizes)-2):
        layer += [
            nn.Linear(in_features=sizes[i],
            out_features=sizes[i+1]),
            activation
        ]
    layer += [nn.Linear(in_features=sizes[-2],
    out_features=sizes[-1])]
    self.net = nn.Sequential(*layer)
def forward(self, x):
    return self.net(x)
```

3.4 Dictionary Construction

After designing the neural network, automatic differentiation [39] is used to accurately and efficiently calculate the derivatives of predicted output u with respect to time t and space x. Then the dictionary matrix is constructed by taking the polynomial and spatial derivatives of the predicted output.

This work aims to explore an automated system that can identify the partial differential equations from the Spatio-temporal measurements by using the dictionary matrix and the sparse regression method. To discover the governing PDE, we can write the following equation for the given spatiotemporal domain $u(x,t)$.

$$u_t = F(u, u_x, u_{xx}, ...) \tag{3.4}$$

The nonlinear function $F(.)$ is written as a sum of all basic functions and their derivatives. To find the desired value of F, we create a large set of possible combinations by taking all permutations of a dictionary of the candidate terms. The dictionary matrix jointly comprises of polynomial functions and their corresponding derivatives of the spatial domain. We can rewrite the model identification problem as below:

$$u_t = \Theta\xi \tag{3.5}$$

where u_t is the column vector comprising the derivative of state variable u with respect to time t for n samples size and Θ contains all of the possible terms for each sample.

Burgers equation

For the Burgers equation, we consider second-order polynomial and spatial derivatives for computing the dictionary matrix Θ. In the equation below, the derivatives of u are calculated with respect to the temporal and spatial domain using automatic differentiation.

$$\frac{\partial u}{\partial t} = u_t, \frac{\partial u}{\partial x} = u_x, \frac{\partial}{\partial x}\left(\frac{\partial u}{\partial x}\right) = u_{xx}$$

where u_t is the derivative of u with respect to time t, u_x is the derivative of u with respect to space x and u_{xx} is the second order derivative of u with respect to x. The polynomial and spatial derivative orders are given below:

Polynomial order = [1, u, u^2]
Spatial order = [1, u_x, u_{xx}]

The polynomial functions and spatial derivatives are taken together to compute the dictionary matrix Θ, as shown below:

$$\Theta = [1, u_x, u_{xx}, u, uu_x, uu_{xx}, u^2, u^2u_x, u^2u_{xx}] \tag{3.6}$$

The dictionary matrix Θ contains potential candidate terms that are achieved by performing the matrix multiplication in polynomial and spatial derivation order terms. For the Burgers equation, the size of the dictionary matrix is nine. From Equation 3.5 we can write:

$$\begin{bmatrix} u_t(x_1,t_1) \\ u_t(x_2,t_2) \\ u_t(x_3,t_3) \\ \vdots \\ u_t(x_{n-1},t_{n-1}) \\ u_t(x_n,t_n) \end{bmatrix} = \begin{bmatrix} 1 & u_x(x_1,t_1) & u_{xx}(x_1,t_1) & \dots & u^2u_{xx}(x_1,t_1) \\ 1 & u_x(x_2,t_2) & u_{xx}(x_2,t_2) & \dots & u^2u_{xx}(x_2,t_2) \\ 1 & u_x(x_3,t_3) & u_{xx}(x_3,t_3) & \dots & u^2u_{xx}(x_3,t_3) \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & u_x(x_{n-1},t_{n-1}) & u_{xx}(x_{n-1},t_{n-1}) & \dots & u^2u_{xx}(x_{n-1},t_{n-1}) \\ 1 & u_x(x_n,t_n) & u_{xx}(x_n,t_n) & \dots & u^2u_{xx}(x_n,t_n) \end{bmatrix} \xi \tag{3.7}$$

2D-Advection-Diffusion equation

For the 2D-Advection-Diffusion equation, we consider first-order polynomial and second-order spatial derivatives to calculate the dictionary matrix Θ. In the following, the derivatives of u are calculated with respect to the Spatio-temporal data.

$$\frac{\partial u}{\partial t} = u_t, \frac{\partial u}{\partial x} = u_x, \frac{\partial u}{\partial y} = u_y, \frac{\partial}{\partial x}\left(\frac{\partial u}{\partial x}\right) = u_{xx}, \frac{\partial}{\partial y}\left(\frac{\partial u}{\partial y}\right) = u_{yy}, \frac{\partial}{\partial x}\left(\frac{\partial u}{\partial y}\right) = u_{xy}$$

where, u_t is the derivative of u with respect to time t, u_x is the derivative of u with respect to space x, u_{xx} is the second order derivative of u with respect to x, u_{yy} is the second order derivative of u with respect to y and u_{xy} is the second order derivative of u with respect to x and y. The polynomial and spatial derivative orders are given below:

Polynomial order = [1, u]
Spatial order = [1, u_x, u_y, u_{xx}, u_{yy}, u_{xy}]

The polynomial functions and spatial derivatives are taken together to constitute the dictionary matrix Θ for the 2D-Advection-Diffusion equation.

$$\Theta = [1, u_x, u_y, u_{xx}, u_{yy}, u_{xy}, u, uu_x, uu_y, uu_{xx}, uu_{yy}, uu_{xy}] \tag{3.8}$$

To compute the potential candidate terms of the dictionary matrix, we perform matrix multiplication between polynomial and spatial derivatives. The size of the dictionary matrix is twelve. We can derive the following equation from Equation 3.5.

$$\begin{bmatrix} u_t(x_1,y_1,t_1) \\ u_t(x_2,y_2,t_2) \\ u_t(x_3,y_3,t_3) \\ \vdots \\ u_t(x_{n-1},y_{n-1},t_{n-1}) \\ u_t(x_n,y_n,t_n) \end{bmatrix} = \begin{bmatrix} 1 & u_x(x_1,y_1,t_1) & u_y(x_1,y_1,t_1) & \dots & uu_{xy}(x_1,y_1,t_1) \\ 1 & u_x(x_2,y_2,t_2) & u_y(x_2,y_2,t_2) & \dots & uu_{xy}(x_2,y_2,t_2) \\ 1 & u_x(x_3,y_3,t_3) & u_y(x_3,y_3,t_3) & \dots & uu_{xy}(x_3,y_3,t_3) \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & u_x(x_{n-1},y_{n-1},t_{n-1}) & u_y(x_{n-1},y_{n-1},t_{n-1}) & \dots & uu_{xy}(x_{n-1},y_{n-1},t_{n-1}) \\ 1 & u_x(x_n,y_n,t_n) & u_y(x_n,y_n,t_n) & \dots & uu_{xy}(x_n,y_n,t_n) \end{bmatrix} \xi \tag{3.9}$$

The python code to construct a dictionary matrix for 2D-Advection-Diffusion equation is given below:

```
def construct_Dictonary_2D(data, uhat, poly_order, deriv_order):
    # build polynomials
    poly = torch.ones_like(uhat)
    # concatinate different orders
    for o in np.arange(1, poly_order+1):
        poly_o = poly[:,o-1:o]*uhat
        poly = torch.cat((poly, poly_o), dim=1)
    # build derivatives
    du = grad(outputs=uhat, inputs=data,
              grad_outputs=torch.ones_like(uhat), create_graph=True)[0]

    dudt = du[:, 0:1]
    dudx = du[:, 1:2]
    dudy = du[:, 2:3]
    dudu = grad(outputs=dudx, inputs=data,
              grad_outputs=torch.ones_like(uhat), create_graph=True)[0]
    dudxx = dudu[:, 1:2]
    dudxy = dudu[:, 2:3]
    dudyy = grad(outputs=dudy, inputs=data,
              grad_outputs=torch.ones_like(dudxx), create_graph=True)[0]
```

```
dudyy = dudyy[:, 2:3]
for o in np.arange(1, deriv_order):
    dudu = torch.cat((torch.ones_like(dudx), dudx, dudy, dudxx,
    dudyy, dudxy), dim=1)
# build all possible combinations of poly and dudu vectors
theta = None
for i in range(poly.shape[1]):
    # print('i:', i)
    for j in range(dudu.shape[1]):
        # print('j:', j)
        comb = poly[:, i:i + 1] * dudu[:, j:j + 1]

        if theta is None:
            theta = comb
        else:
            theta = torch.cat((theta, comb), dim=1)
return dudt, theta
```

The dictionary matrix Θ contains more items than required for discovering PDE. By applying the sparse regression method, most of the uninformative coefficients will be turned to zero. Section 3.5 describes the LASSO regression method to select the sparse coefficient vectors.

3.5 Loss Function

After designing the feed-forward neural network and constructing the dictionary matrix, the loss function is calculated to denoise the Spatio-temporal dataset and to discover the governing PDEs. In each training, the mean squared error (MSE), regression loss, and L_1 (LASSO) regularization are calculated. Total loss is the sum of all three losses. The neural network model is identified as a good model if the total loss is close to 0.

MSE

The MSE is the average of the squared difference between the actual output and the predicted output. The equation to calculate the MSE for the Burgers equation is given

below:

$$MSE_{loss} = \frac{1}{n}\sum_{i=1}^{n} |(u\{x,t\}_i) - \hat{u}_i|^2 \tag{3.10}$$

where n denotes the number of samples. A substantial value of MSE indicates that the predicted output is far from the actual output. Since weights and biases are updated in every epoch, hence the value of predicted output gets reduced. The MSE equation to calculate the 2D-Advection-Diffusion equation is given below:

$$MSE_{loss} = \frac{1}{n}\sum_{i=1}^{n} |(u\{x,y,t\}_i) - \hat{u}_i|^2 \tag{3.11}$$

Sparse Regression Loss

Sparse regression loss is computed with Equation 3.12, which reduces the overfitting problem of the noisy Spatio-temporal dataset.

$$Reg_{loss} = \frac{1}{n}\sum_{i=1}^{n} |\Theta_i \xi_i - \hat{u}_{t_i}|^2 \tag{3.12}$$

The sparse regression loss is the squared difference between the dictionary matrix Θ and the derivative of predicted output $\hat{u}$ with respect to t.

LASSO Regularization

Least absolute shrinkage and selection operator (LASSO) regression is called the penalized regression system used in the neural network to select the subset of coefficients [40, 41]. It calculates the sum of the absolute values of the coefficients, which causes some coefficients to shrink in the direction of zero, and it is called shrinkage procedure. For subset selection, the shrinkage procedure provides better interpretation and selects essential features that are strongly connected to the target coefficients, which is called the subset selection procedure. LASSO regression is vital because it reduces overfitting problems in the neural network model. As we need few amounts of terms from the dictionary matrix, with LASSO regression, the unimportant coefficients will be set to zero, which efficiently helps to remove the unnecessary terms from the dictionary matrix. The most informative coefficients will remain non-zero.

A tuning parameter λ is applied with coefficients in LASSO regression. When λ increases, most of the coefficients shrink towards zero. The addition of tuning parameter λ in the regression method is called regularization. Regularization is an important concept in deep learning, which reduces the variance of the model without increasing bias. The regularization used in the following equation is called L_1 or LASSO

regularization.

$$L_1 = \lambda \sum_{i=2}^{m} |\xi^*| \tag{3.13}$$

where ξ^* denotes the i^{th} component of the normalized coefficient vector (Normalization is described in the section 3.6). The initial value of i begins from 2 because the tunning parameter λ is not applied to the constant term. The total loss is the sum of all the above mentioned losses.

$$Total_{Loss} = MSE_{loss} + Reg_{loss} + L_1 \tag{3.14}$$

The calculation of total loss helps the neural network model to denoise the noisy dataset in order to discover the underlying governing PDEs.

The gradient loss is computed first to determine if the neural network has converged or not. For calculating gradient loss, the derivative of total loss is taken with respect to the coefficients ξ_i. The equation to calculate the gradient loss is given below:

$$Gradient\_loss = \left(\frac{\partial Loss}{\partial \xi_i} \frac{||u_t||}{||\Theta_i||} \right) \tag{3.15}$$

If the maximum value of the gradient loss is less than the predefined tolerance, the network continues training until the maximum number of iterations; otherwise, it converges and produces the final output.

$$max(Gradient\_loss) < tolerance \tag{3.16}$$

For the experiment, we set tolerance of 10^{-6} for both of the Burgers equation and the 2D-Advection-Diffusion equation.

3.6 Model Identification

When the network has completed the training, the sparse vectors have been recovered. Despite using LASSO regularization, most of the coefficients are non-zero, and we are looking for the actual sparse representation. Thresholding is used here to recover the true sparse vectors. Each element has a different magnitude and scales, therefore we normalize the elements before thresholding. Normalization allows the model to take the elements with several scales and ranges, and keep them into a common standard scale. By applying normalization, the sparse regression method will

have only a standard length vector that provides a simple interpretation of coefficients ξ.

$$\xi^* = \left(\xi_i \frac{||\Theta_i||}{||u_t||}\right) \tag{3.17}$$

where ξ^* is the normalized coefficient. $||\Theta||$ contains row-wise elements that comprise the norm of every column of dictionary matrix Θ. Additionally, $||u_t||$ is the norm of the temporal derivative vector.

Finally, we apply thresholding on the scaled coefficients vector. As most of the coefficient values are close to zero, the thresholding operation finds the true sparsest vector by considering the median value from the scaled coefficients vector ξ^*. The equation of thresholding to obtain the true sparse representation is given below:

$$\xi_i = \begin{cases} \xi_i, & if \;\; l < \xi^* < h \\ 0 & otherwise \end{cases} \tag{3.18}$$

where, lower order l = $median(\xi^*)$ - $\sigma(\xi^*)$ and higher order h = $median(\xi^*)$ + $\sigma(\xi^*)$. The lower-order is achieved by subtracting the standard deviation of coefficients ξ^* from the median of coefficients ξ^*. The higher-order is the sum of the standard deviation of coefficients ξ^* and the median of coefficients ξ^*. The true sparse coefficient vectors ξ_i are those which are within the lower order and the upper order range.

3.7 PDE Recover

We combine the actual sparse coefficients vector first with the respective terms from the dictionary matrix and then take the sum of them to recover PDE.

For the Burgers equation, two sparse coefficients vector ξ_1 and ξ_2 are recovered out of nine and combined with the respective terms u_{xx} and uu_x from the dictionary matrix. For 2500 samples and 1% noise level, the identified PDE is -

$$u_t = 0.100u_{xx} - 1.007uu_x$$

For 2D-Advection-Diffusion equation, we recover four sparse coefficient vectors ξ_1, ξ_2, ξ_3 and ξ_4 out of twelve and combine them with the respective terms u_x, u_y, u_{xx}, u_{yy} from the dictionary matrix. With 2500 samples and 1% noise level, the discovered PDE is -

$$u_t = 0.250u_x + 0.487u_y + 0.501u_{xx} + 0.475u_{yy}$$

3.8 Evaluation

In this thesis work, we have chosen the Burgers equation and the 2D-Advection-Diffusion equation to evaluate the system's performance. We consider 1%, 5%, 10%, 15%, and 25% noise levels and add them with the actual output respectively. For the 1% noise level, we select 100 samples and iterate the same process ten times to avoid inconsistency. Every time we calculate the loss function, coefficients, coefficient errors, and count the number of correct predictions to calculate the accuracy. Then we divide the correct predictions with the total number of iterations to acquire the accuracy. Accuracy is the ratio of the number of correct predictions to the total number of iterations. Later, 500 samples are chosen randomly for the 1% noise level and the same procedure applied to compute the loss, coefficient, and coefficient errors. Then, we iterate the model ten times with 1000, 1500, 2000, 2500, 3000, 4000, 5000, and 8000 samples, respectively, to evaluate the performance of the model and to observe the number of samples that give fewer coefficient error and the highest accuracy. Finally, we repeat the whole process for other noise levels and evaluate the performance.

3.8.1 Error Calculation for Burgers Equation

For the Burgers equation, we recover two coefficients and assign them with label ξ_1 and ξ_2 respectively. After that, we calculate the error for both coefficients and give the name ξ_{1e} and ξ_{2e} accordingly.

The first average coefficient $\bar{\xi}_1$ is calculated by taking the average of all coefficients ξ_1 for different iterations. Firstly, the summation of ξ_1 is calculated. To achieve the average coefficient $\bar{\xi}_1$, the summation of ξ_1 is divided by the number of correct predictions.

$$\bar{\xi}_1 = \frac{1}{C} \sum_{i=1}^{C} \xi_{1_i} \tag{3.19}$$

where, C is the number of correct predictions. First coefficient error ξ_{1e} is the difference between actual coefficient Ξ_1 and the predicted coefficient ξ_1, where Ξ_1 is 0.1.

$$\xi_{1e} = \Xi_1 - \xi_1 \tag{3.20}$$

First average coefficient error $\bar{\xi}_{1e}$ is the average of all coefficient errors ξ_{1e}. The equation is given below:

$$\bar{\xi}_{1e} = \frac{1}{C} \sum_{i=1}^{C} \xi_{1e_i} \tag{3.21}$$

Second average coefficient $\bar{\xi}_2$ is the average of all the coefficient ξ_2 for several iterations. First, the sum of all ξ_2 is calculated. To achieve average coefficients of $\bar{\xi}_2$, the sum of ξ_2 is divided by the number of correct predictions.

$$\bar{\xi}_2 = \frac{1}{C} \sum_{i=1}^{C} \xi_{2_i} \tag{3.22}$$

The second coefficient error ξ_{2e} and the average coefficient error $\bar{\xi}_{2e}$ are calculated by using the following formulas in Equation 3.23 and 3.24 respectively. Second coefficient error ξ_{2e} is the difference between the actual coefficient Ξ_2 and the predicted coefficient ξ_2, where Ξ_2 = 1.0.

$$\xi_{2e} = \Xi_2 - \xi_2 \tag{3.23}$$

Average coefficient error $\bar{\xi}_{2e}$ is the average of all the coefficient errors ξ_{2e}.

$$\bar{\xi}_{2e} = \frac{1}{C} \sum_{i=1}^{C} \xi_{2e_i} \tag{3.24}$$

Mean Absolute Error (MAE)

The mean absolute error is calculated by averaging both of the average coefficient errors $\bar{\xi}_{1e}$ and $\bar{\xi}_{2e}$. The equation is given below.

$$MRE = (\bar{\xi}_{1e} + \bar{\xi}_{2e})/2 \tag{3.25}$$

3.8.2 Error Calculation for 2D-Advection-Diffusion Equation

For the 2D-Advection-Diffusion equation, we discover four coefficients from total twelve, which are non-zero and assign them with name ξ_1, ξ_2, ξ_3 and ξ_4 respectively. After that, we calculate the error for all of the coefficients and give the label ξ_{1e}, ξ_{2e}, ξ_{3e}, and ξ_{4e}. The coefficient error, average coefficient, and average coefficient error are calculated in the same way as we calculated for the Burgers equation. The only difference is that we have four actual coefficients for the 2D-Advection-Diffusion equation. The value of the first actual coefficient Ξ_1 is 0.25, and the value for the other three actual coefficients Ξ_2, Ξ_3, and Ξ_4 is same and it is 0.5.

4 Results and Discussion

The Burgers equation and the 2D-Advection-Diffusion equation are chosen to test the performance of the feed-forward neural network model. The network model is constructed with an input layer, five hidden layers, where each hidden layer consists of twenty neurons and an output layer to produce the final output $\hat{u}$. The Burgers equation considers spatiotemporal data x and t as input, and the 2D-Advection-Diffusion takes x, y, and t as input. We train the neural network using Adam optimizer to minimize the total loss function. In order to compute the total loss, we use three loss functions, namely: Mean Squared Error(MSE), Regression Loss and LASSO regularization. We assigned β_1 to 0.9998 and λ to 10^{-6} to minimize the loss function and constitute the governing PDEs. When we train the neural network, the PDE is constructed using the recovered sparse coefficients. When the PDE predictions are correct, only then the coefficient errors and correction predictions are calculated to compute the average coefficients and average coefficient errors.

4.1 Burgers Equation

In the beginning, the neural network is trained for 2000 epochs. The learning rate is set to 0.003 to update the weights and biases and 0.03 to adjust the coefficients ξ_i in every epoch to make the convergence faster. The coefficient ξ_i is also updated along with the weights, biases. In every epoch, the model minimizes the loss function. When the network finished training with 1000 samples, the total loss was 0.0003 for 10% Gaussian noise addition and and it was possible to recover the partial differential equation. The recovery PDE is given below (Equation 4.1):

$$u_t = 0.088u_{xx} - 0.887uu_x \tag{4.1}$$

The losses against the number of epochs is shown in Figure 4.1.

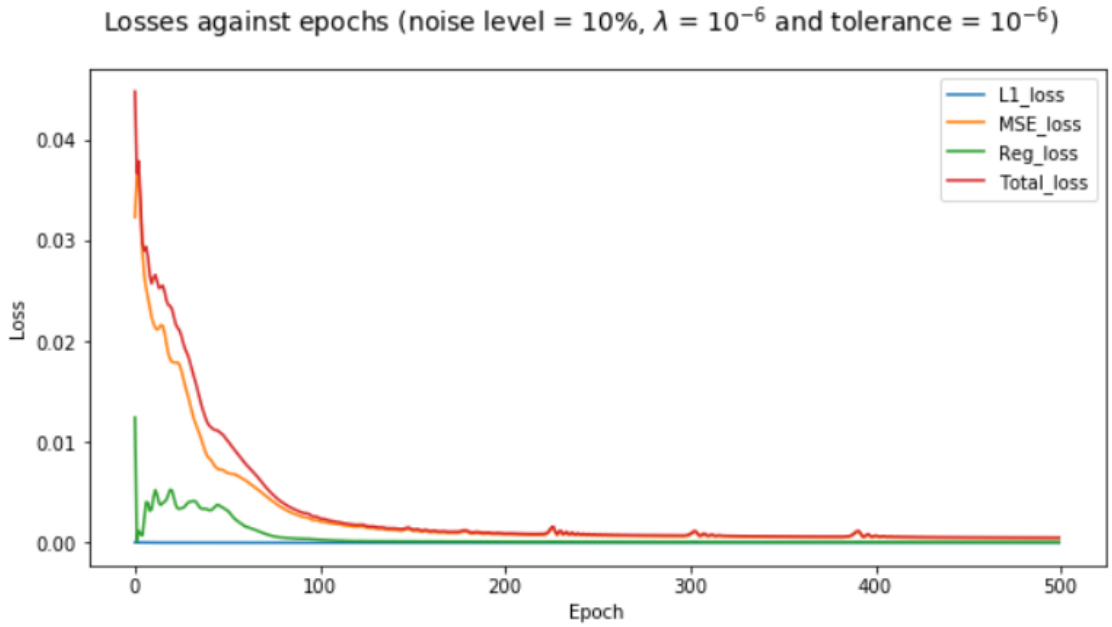

FIGURE 4.1: Losses over epochs for 1000 samples and 10% additive Gaussian noise

The original, noisy, and reconstructed solution for the Burgers equation is shown in Figure 4.2 for 10% noise level, 1000 samples, and 2000 epochs.

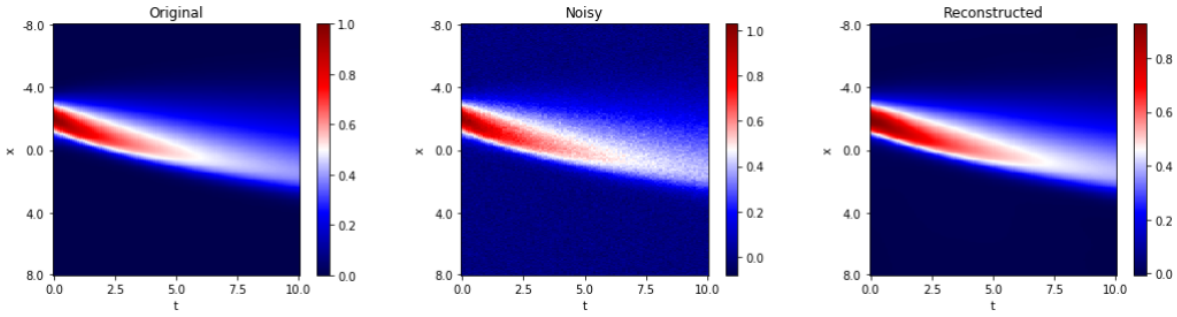

FIGURE 4.2: Original, noisy and reconstructed solution for Burgers equation

The neural network is also trained with 2000 samples and 10% noise addition, where the regression loss converges after 1000 epochs. At first, the network is trained till 1000 epochs with MSE loss and LASSO to reduce the error between the original and predicted output as the predicted output was far away from the original output. After 1000 epochs, we added regression loss along with the MSE loss and LASSO.

Once the neural network finishes training, the underlying PDE is identified. The identified PDE is given below in Equation 4.2.

$$u_t = 0.085u_{xx} - 0.895uu_x \tag{4.2}$$

Though it is possible to recover the underlying PDE with 2000 epochs, the coefficient value of ξ_1 and ξ_2 is far away from the actual value. For example, the actual value of ξ_1 and ξ_2 is 0.1 and 1.0 respectively, whereas the discovered coefficient values are 0.088 and 0.887 (see Equation 4.1). Often, the model is not able to predict the correct PDE, which means that the model is not good enough with 2000 epochs. To obtain the predicted coefficient value near to the actual coefficient value and to recover the PDEs correctly, we increase the epochs from 2000 to 10000 for further processing.

It has already been discussed in the evaluation part that we considered different noise levels (1%, 5%, 10%, 15%, and 25%) and samples to evaluate the performance of the model. For every noise level, the considered sample sizes are (100, 500, 1000, 1500, 2000, 2500. 3000, 4000, 5000, 8000). For each sample size, we iterate the model ten times by taking data randomly from the training data-set. For training the neural network model, the learning rate is assigned to 0.001 to adjust the parameter weights and biases, and 0.01 to update the coefficients ξ_i. We set the seed to identify the data taking part in every run. Seed helps to create the same set of random numbers on several executions of the program on the same or various machines. The assigned seeds are 0, 5, 26, 57, 73, 80, 95, 104, 129, 151. We run the neural network model ten times, separately for each noise level and sample size to avoid inconsistency. In every iteration, coefficients, coefficient errors, and correct predictions are calculated. In the beginning, the correct prediction is set to zero. Correct prediction is the sum of all the prediction, which predicts the correct PDE for selected data. When the first predicted PDE is correct, the correct prediction is incremented from zero to one. We repeat the process ten times and evaluate system performance. When the PDE prediction is correct only then we calculate ξ_{1e}, ξ_{2e}, $\bar{\xi}_{1e}$, $\bar{\xi}_{2e}$ and $\bar{\xi}$. For 1%, 10% and 25% noise levels, the acquired average coefficients ($\bar{\xi}_1$, $\bar{\xi}_2$) and average coefficient errors ($\bar{\xi}_{1e}$, $\bar{\xi}_{2e}$), for several samples sizes are captured in Figure 4.4.

4.1.1 Correct Predictions

The correct predictions over the number of sample sizes is shown in Figure 4.3. With the increasing number of sample sizes, the correct prediction is increasing. On the contrary, the correct prediction is decrease with the increased noise level. It is visible

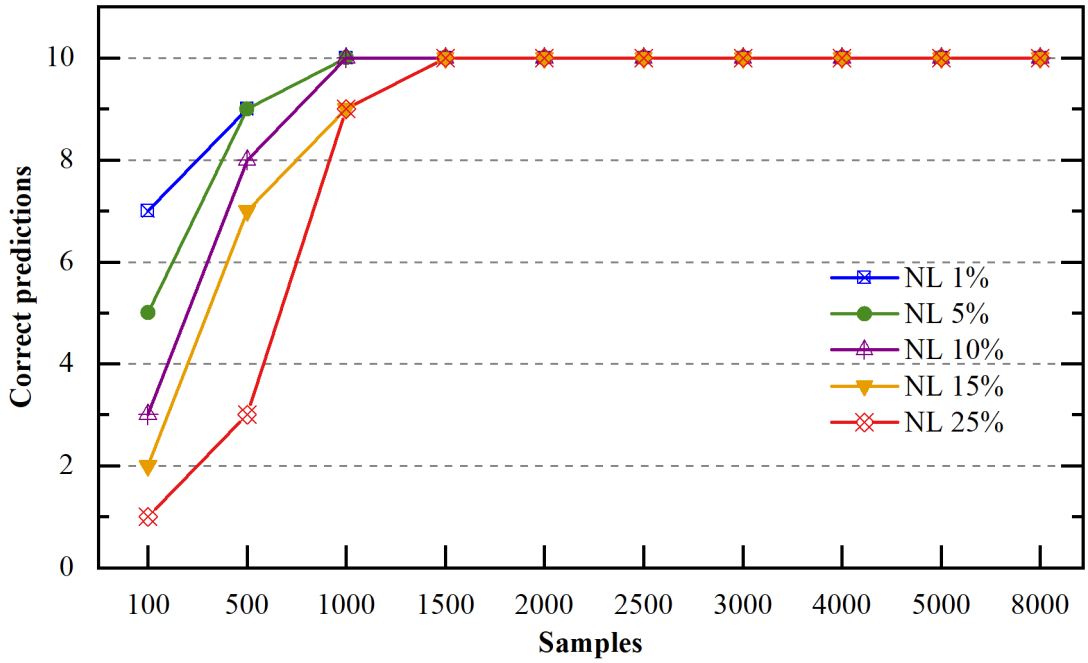

FIGURE 4.3: In the achievability plot, the correct predictions is plotted over the number of samples for the Burgers equation for different noise levels

that for the sample size (1500 - 8000), the model predicted all of the PDEs correctly. However, the model recovered the PDEs for 100 and 500 samples, but the number of PDE predictions are lower than other sample sizes. With 1000 samples, the PDE predictions are reasonably good for all of the noise levels.

4.1.2 Average Coefficients and Average Coefficient Errors

The average coefficients ($\bar{\xi}_1$, $\bar{\xi}_2$) and the average coefficient errors ($\bar{\xi}_{1e}$, $\bar{\xi}_{2e}$) against number of sample size is plotted for different noise levels (1%, 5%, 10%, 15% and 25%). Here, only the results for 1%, 10% and 25% noise level (NL) are presented in Figure 4.4. Moreover, we would like to refer Figure B.1 in Appendix A for other noise levels (5% and 15%).

In Figure 4.4, the average coefficients and the average coefficient errors for the Burgers equation are shown from which we can analyze the results in a better way. It is noticeable from Figure 4.4(a, c, e) that when the noise level is increasing, the average coefficient values deviate significantly from the actual. Besides, there is no significant difference between actual and average coefficient $\bar{\xi}_1$ for 2500 – 8000 samples size in comparison to 100-2000 samples. With an increasing number of samples, the deviation window is decreasing. There are more fluctuations for 25% noise level

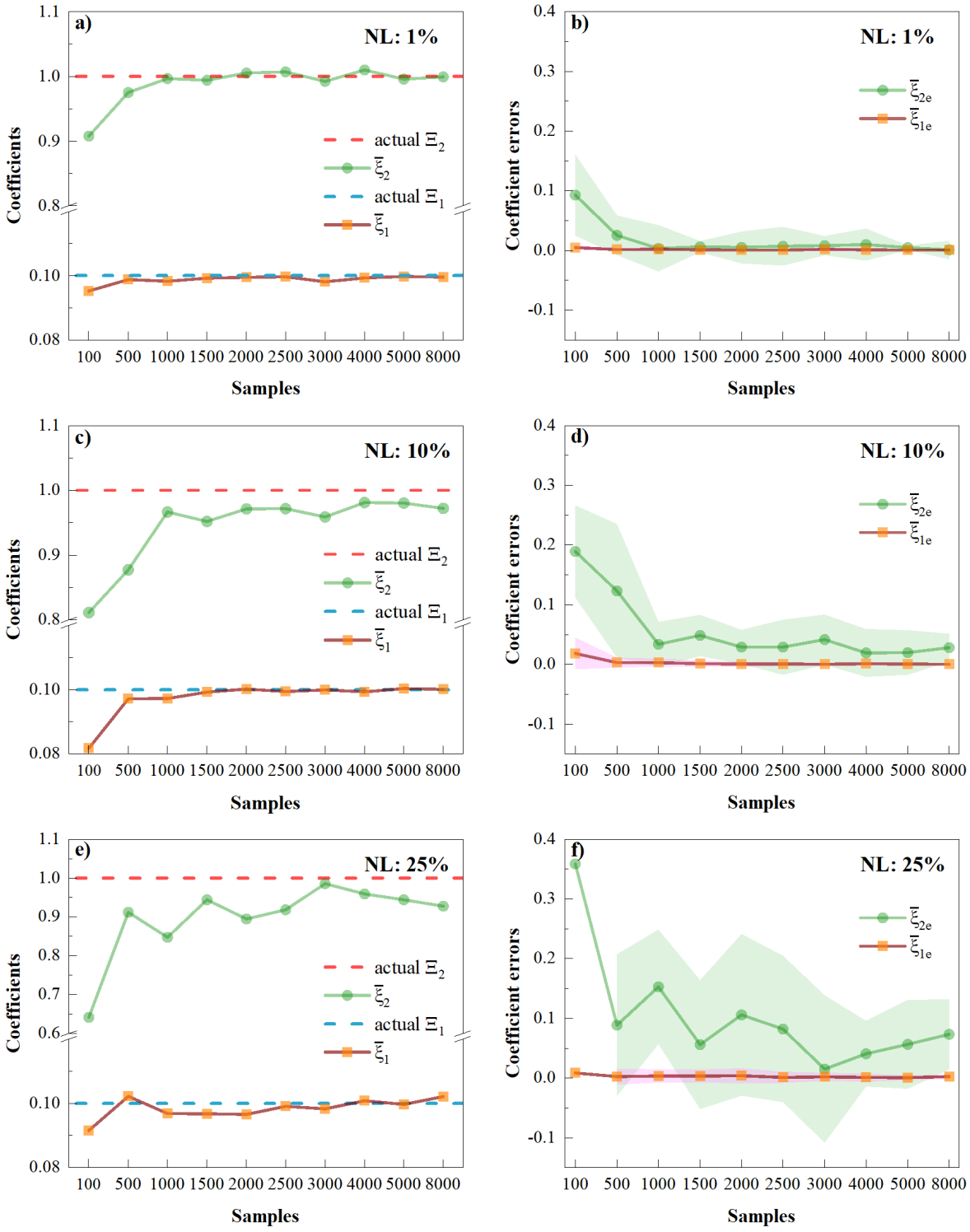

FIGURE 4.4: Average coefficients ($\bar{\xi}_1$ and $\bar{\xi}_2$) and coefficient errors ($\bar{\xi}_{1e}$ and $\bar{\xi}_{2e}$) against samples in Burgers equation. a), c), e) show coefficients for 1%, 10% and 25% noise level (NL); and b), d), f) show coefficient errors with standard deviation (SD, shaded areas) for 1%, 10% and 25% NL respectively. The SD is 0 for 25% NL and 100 samples because there is only one correct prediction and hence the coefficient error and average coefficient error values are same.

compared to others. However, $\bar{\xi}_1$ follows a similar trend for all the noise levels; for example, there is a slight increase up to 500 sample sizes and then it remains stable. Furthermore, $\bar{\xi}_2$ has also similar trend as like as $\bar{\xi}_1$ with more fluctuation.

The corresponding average coefficient errors over sample size for individual noise level is shown in Figure 4.4 (b, d, f). In general, an error is there when a small or large deviation is induced. Therefore, the value of average coefficient errors $\bar{\xi}_{2e}$ for 25% NL is substantial compared to others. As we described earlier that the values are not only fluctuating significantly, but also they show a significant standard deviation. With 100 samples, there is only one correct prediction, and hence, the SD is zero. For all NL, the SD of errors is decreasing with an increased number of samples size.

4.1.3 Dependency on Dictionary Size

We have performed all of the above experiments where the dictionary matrix is built with second-order polynomials and second-order spatial derivatives of a total of nine terms. We want to see if we expand the dictionary size, whether it will be possible to recover the correct PDE. For that, the neural network model is tested with the highest fifth-order spatial derivatives and fourth-order polynomials for 1% and 10% noise levels. The size of the dictionary matrix are 9, 12, 16, 20, 24, and 30 for the combination of polynomial and spatial derivatives. The polynomial and spatial derivative orders, dictionary size, and the discovered PDE for 1% noise addition are tabulated in Table 4.1.

TABLE 4.1: Effect of dictionary size to recover PDE for Burgers Equation with 1% noise level and 2000 samples

Total terms in dictionary matrix for identifying Burgers equation			
Spatial Order	Poly Order	Dictionary Size	Average PDE
2	2	9	$0.0995u_{xx}$ - $1.0049uu_x$
2	3	12	$0.0965u_{xx}$ - $0.9944uu_x$
3	3	16	$0.0985u_{xx}$ - $0.9854uu_x$
4	3	20	$0.0970u_{xx}$ - $1.0069uu_x$
5	3	24	$0.0924u_{xx}$ - $1.1249uu_x$
5	4	30	$0.0919u_{xx}$ - $1.0332uu_x$

We compute the correct predictions for the total terms of the dictionary matrix for 1% and 10% noise level, as shown in Figure 4.5. For both noise levels, the neural network model recovered all of the ten PDEs correctly against 9, 12, 16, and 20 dictionary size, and the correct predictions are 7 against 24 dictionary size. The correct predictions are 6 and 9, respectively, for the respective noise levels for dictionary size 30.

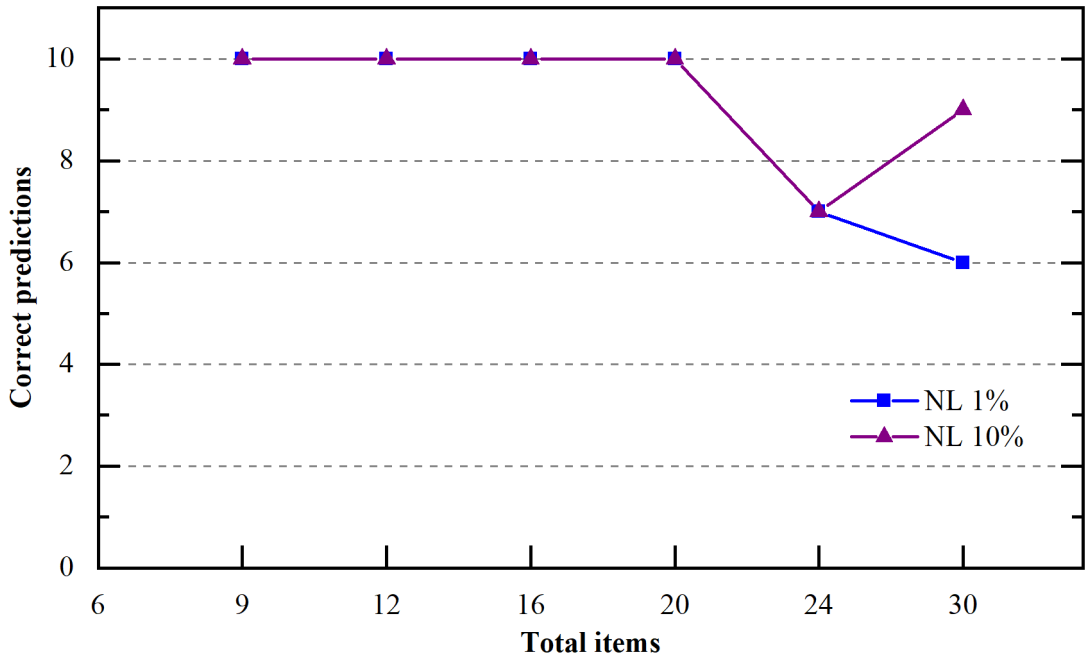

FIGURE 4.5: Correct predictions for ten iterations over total number of terms of the dictionary matrix for Burgers equation

We iterated the neural network model ten times, and in every run, it is possible to identify the two coefficients correctly for constructing PDE from the maximum 20 items. The computational cost increases with the increasing number of derivative orders. As an example, for the second-order spatial derivative, the system took around 1 hour on the CPU machine with 16 GB RAM to complete ten iterations. The process took near 2 hours and 4 hours, respectively, for third and fourth-order derivatives. For the fifth-order derivative, the total time is around 12 to 13 hours.

4.1.4 Dependency on Nodes per Layer

We test the neural network model by changing neuron numbers for each hidden layer. The neural network consists of five hidden layers. In the beginning, for each hidden layer, ten neurons are considered and iterate the algorithm ten times. After that,

twenty, thirty, and forty neurons are chosen to observe if the changing number of neurons affects the results or not. It is shown in Figure 4.6 that with 2000 samples and 1%, 10%, and 25% noise levels, the model discovered all of the ten PDEs correctly for the defined number of neurons. When the noise level is 25%, the correct predictions are 5, 9, 7, and 6, respectively, for the respective neuron numbers.

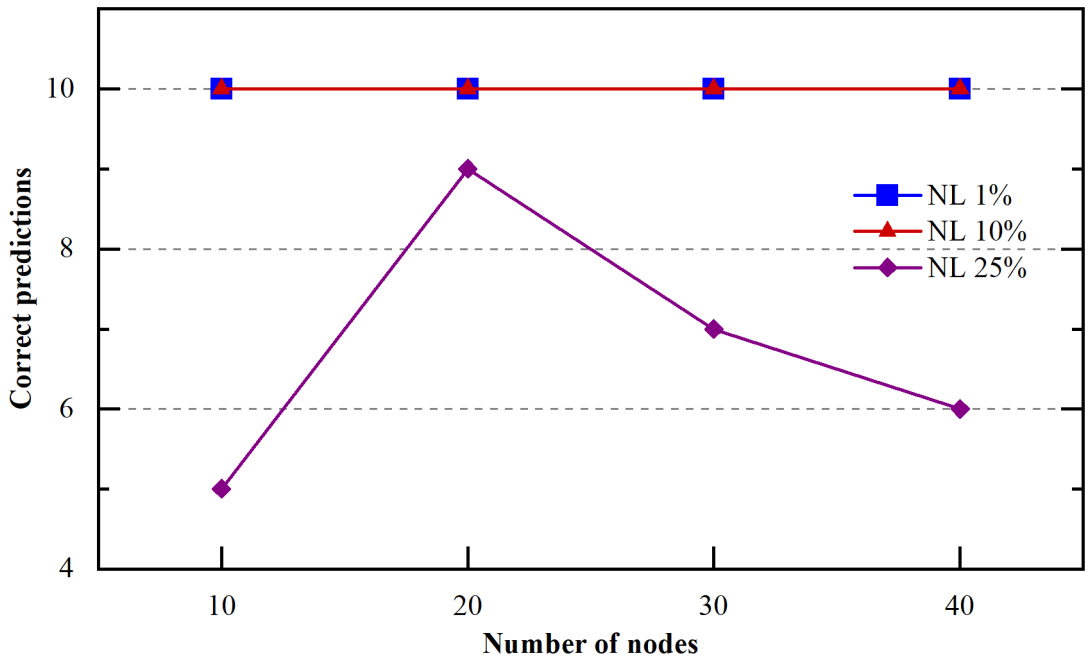

FIGURE 4.6: Correct predictions for ten iterations against the number of neurons in Burgers equation for 1%, 10%, and 25% noise level. The neural network model contains five hidden layers

The number of neurons does not significantly affect the recovery of the correct coefficients for building PDEs. The discovered coefficient values are almost close to the actual values, as tabulated in Table 4.2.

TABLE 4.2: Effect of nodes in neural network architecture to recover PDE for Burgers equation

Effect of nodes in neural network architecture (noise level = 1%, samples = 2000)					
Nodes	$\bar{\xi}_1$	$\bar{\xi}_2$	$\bar{\xi}_{1e}$	$\bar{\xi}_{2e}$	Average PDE
10	0.09931091	1.00004711	0.00068909	0.00004711	$0.0993u_{xx}$ - $1.0000uu_x$
20	0.0996334	1.00048077	0.0003666	0.00048077	$0.0996u_{xx}$ - $1.0005uu_x$
30	0.09980288	0.98828142	0.00019712	0.01171858	$0.0998u_{xx}$ - $0.9883uu_x$
40	0.09988279	0.99531149	0.00011721	0.00468851	$0.0999u_{xx}$ - $0.9953uu_x$

4.1.5 Dependency on Hidden Layers

The neural network model is also tested with different number of hidden layers, where each layer contains twenty nodes or neurons. The graphical representation of the number of correct predictions against the number of layers is shown in Figure 4.7. From layers four to six, the correct predictions are 10, i.e., the neural network model predicts the entire PDEs correctly for all of the noise levels.

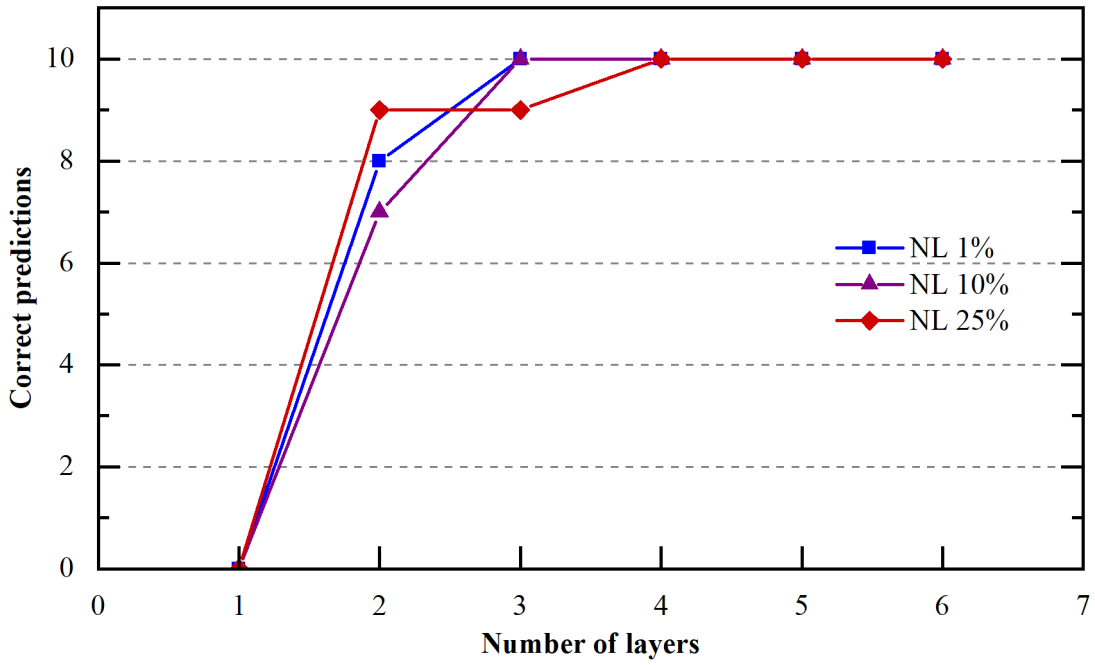

FIGURE 4.7: Correct predictions for ten iterations against the number of layers in Burgers equation for 1%, 10% and 25% noise levels, where each hidden layer contains twenty nodes

We can see, with the increasing number of layers the number of correct prediction is increasing and it becomes saturated afterwards. For example, when the neural network model is constructed with one hidden layer where there is no single correct prediction from ten iterations for 1%, 10%, and 25% noise levels. The recovered PDEs are given in Table 4.3. It is visible that the identified coefficient values are very close to the actual coefficient values from layer four to layer six for 1% noise level and 2000 samples.

TABLE 4.3: Effect of hidden layers in neural network architecture to recover PDE for Burgers equation

Effect of hidden layers in neural network architecture (noise level = 1%, samples = 2000)					
Hidden Layers	$\bar{\xi}_1$	$\bar{\xi}_2$	$\bar{\xi}_{1e}$	$\bar{\xi}_{2e}$	Average PDE
1	-	-	-	-	No Recovered PDE
2	0.09559381	1.03249209	0.00440619	0.03249209	$0.0956u_{xx}$ - $1.0325uu_x$
3	0.09871085	0.99039274	0.00128915	0.00960726	$0.0987u_{xx}$ - $0.9904uu_x$
4	0.09921096	1.00155002	0.00078904	0.00155002	$0.0992u_{xx}$ - $1.0016uu_x$
5	0.09952279	1.00111433	0.00047721	0.00111433	$0.0995u_{xx}$ - $1.0011uu_x$
6	0.09923294	0.99068273	0.00076706	0.00931727	$0.0992u_{xx}$ - $0.9907uu_x$

4.1.6 Performance Evaluation

We compared our obtained results with existing DeepMoD results [20]. The comparison is shown in Figure 4.8. The written value inside the square boxes is mean absolute error (MAE), and different density of colors indicate the percentage of correct PDEs. The mean absolute error is calculated by taking the average of both average coefficient errors $\bar{\xi}_{1e}$ and $\bar{\xi}_{2e}$ against the number of sample size for the defined noise levels. The main difference of the neural network based technique with existing DeepMoD is that they iterated it for five times [20], where we run ten times. The red color in the square box of the Figure 4.8b indicates no correct PDE is recovered for the five iterations. By visualizing the color contrast, a conclusion has made that the correct PDE prediction by the neural network model is better than existing DeepMoD. If you see Figure 4.8b, with 100 datapoints, none of the PDE prediction is correct for 10%, 15% and 25% noise levels. In contrast, the neural network model prediction rate are 30%, 20% and 10% for the respective noise levels (in Figure 4.8a). With the neural

network model, the highest MAE (= 10.4) is achieved by 100 samples for a 10% noise level compared to no correct prediction for the existing DeepMoD model. With this model, the highest MAE has acquired for 500 samples and a 15% noise level. Existing DeepMoD predicted that MAE value (28) is slightly lower than three times our predicted value (10.1).

We have received better results than existing DeepMoD results. The following directions are observed during the evaluation of the system-

1. For every sample size, the neural network model iterates ten times instead of five. In each iteration, the coefficient values are fluctuating. We calculate the sum of all coefficient errors by considering real values instead of absolute values. For instance, if the acquired coefficient values of the ξ_2 are 1.081 and 0.978 for the two iterations, coefficient errors will be -0.081 (1.0 - 1.081) and +0.022 (1.0 - 0.978) and the sum of those two coefficient error values will be 0.059 (-0.081+0.022) instead of 0.103 ($|-0.081|+|+0.022|$) as we did not consider the absolute value. In the same way, we computed the average coefficients for ten iterations.

2. The values of hyperparameter also affect the results. In DeepMoD, they assign the values of λ and learning rate 10^{-5} and 0.002 respectively, and we assign 10^{-6} and 0.001 for the neural network model. The choice of right parameter settings

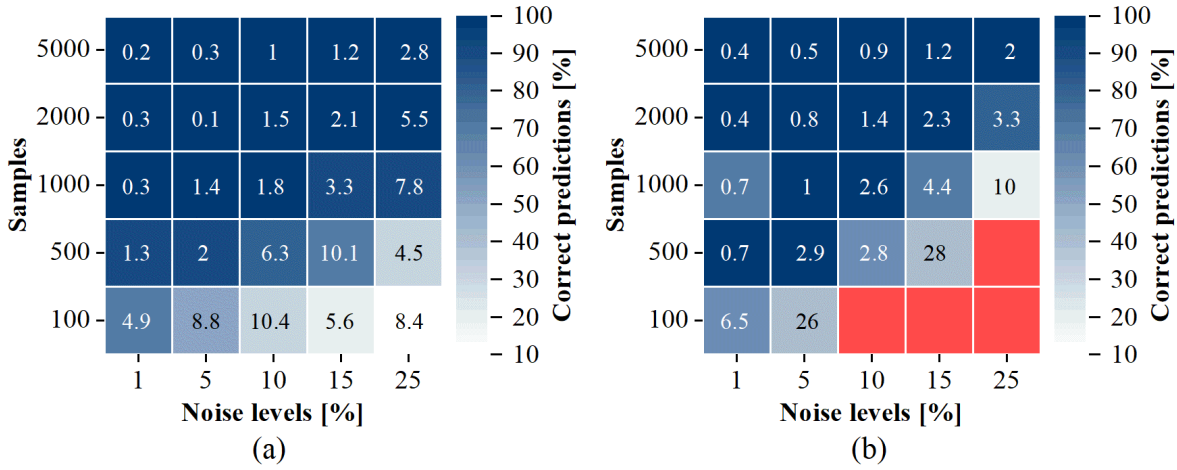

FIGURE 4.8: Performance evaluation of the existing DeepMoD with our results. The values in the square box represent the mean absolute error of the average coefficient errors $\bar{\xi}_{1e}$ and $\bar{\xi}_{2e}$ for the samples size and noise levels. Different color contrast indicates the fraction of the correct predictions. Figure (a) presents our results and Figure (b) is reconstructed from DeepMoD model [20]. Red square box indicates no correct PDE is recovered after five iterations.

is essential for a better outcome. At first, we tested the neural network model with $\lambda = 10^{-5}$ and the learning rate 0.003. For 10% noise and 100 samples, the correct PDE was only one out of ten, and the discovered PDE is given below:

$$u_t = 0.056u_{xx} - 0.696uu_x \tag{4.3}$$

4.2 2D-Advection-Diffusion

To show the robustness of our implemented neural network model on the high dimensional spatiotemporal domain, we apply it to the 2D-Advection-Diffusion equation. Similar to Burgers equation, it is also possible to identify PDE for the 2D-Advection-Diffusion equation with 2000 epochs. In the Burgers equation, we add 10% noise to recover the PDE with 2000 epochs. For the 2D-Advection-Diffusion equation, the model could not recover the PDE with a 10% noise level; instead, we add 1% noise level. The discovered PDE has given below:

$$u_t = 0.266u_x + 0.493u_y + 0.424u_{xx} + 0.359u_{yy}$$

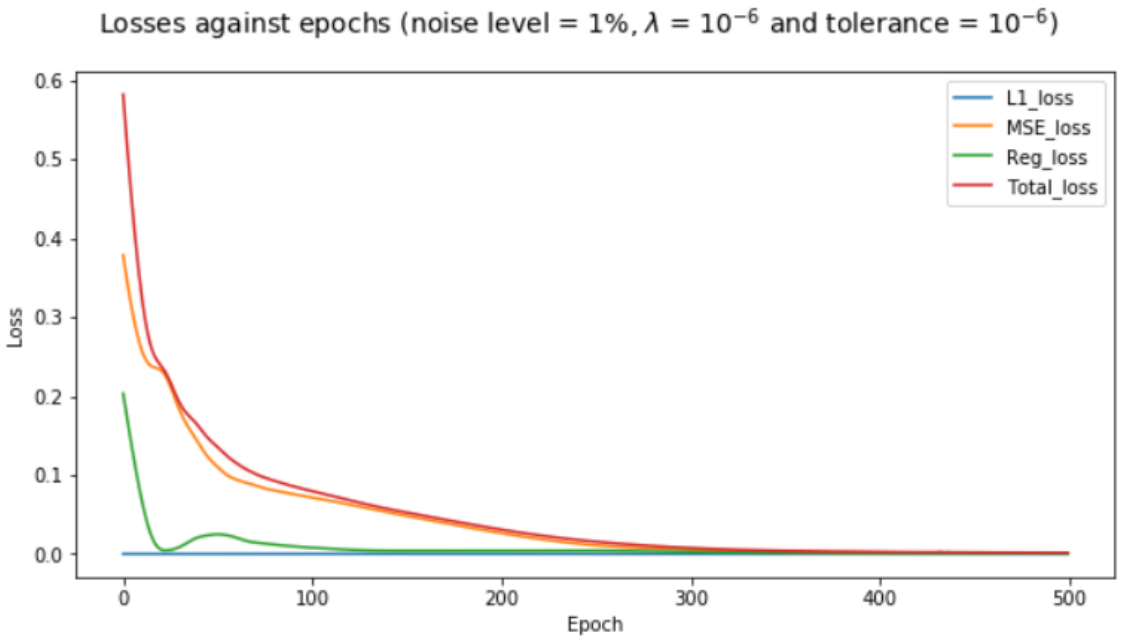

FIGURE 4.9: Losses over epochs with 2000 samples and 1% noise level

Here we have used 2000 samples, and the parameter β_1 and λ are the same as in the Burgers equation. The learning rate is assigned to 0.001 to adjust the parameter weights and biases, and 0.01 to update the coefficient ξ. The three losses are captured

with the total loss against the number of epochs in Figure 4.9. We can see from Figure 4.9 that in the beginning, the total loss is around 0.6, and after 300, it decreased significantly from 0.6 to 0.008. It is also noticeable that the three losses MSE, regression, and LASSO converge at almost 300 epochs.

We consider 10000 epochs here too for the further testing process. It is possible to reconstruct the 2D-Advection-Diffusion equation from noisy data. The original, noisy, and reconstructed solution for 2D-Advection-Diffusion is shown in Figure 4.10 with 2000 samples, 10% NL, and 10000 epochs.

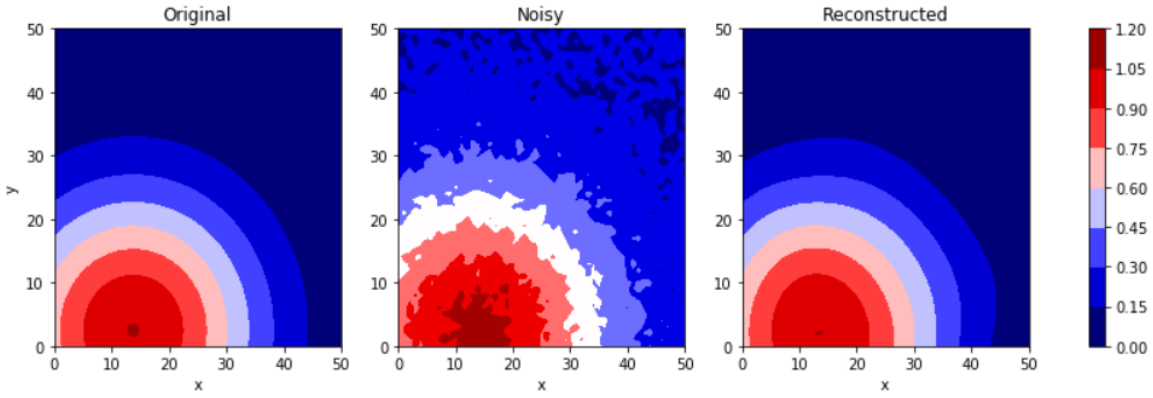

FIGURE 4.10: Original, noisy and reconstructed solution for 2D-Advection-Diffusion equation

The performance of the model is tested for the 2D-Advection-Diffusion equation. We run the model ten times and use the same network specification as we use for the Burgers equation. The noise levels, sample sizes, seeds, and parameter settings were also similar to the Burgers equation. The correct predictions, coefficients ξ_1, ξ_2, ξ_3 and ξ_4 and the coefficient errors ξ_{1e}, ξ_{2e}, ξ_{3e} and ξ_{4e} are calculated the same way as we calculated for the Burgers equation.

4.2.1 Correct Predictions

The correct predictions against the number of samples for all of the considered noise levels is shown in Figure 4.11. Here we can see that the number of correct predictions is increases with the larger sample size and the number of correct predictions is decreases with increased noise levels. With 100 samples, the correct predictions are 7 for 1% noise addition, and no PDE is recovered for 5% noise level. For other samples size, the system predicted all of the PDEs correctly for both 1% and 5% noise levels. When the noise level is 10%, no correct PDE is recovered for 100 and 500 samples. With 1000

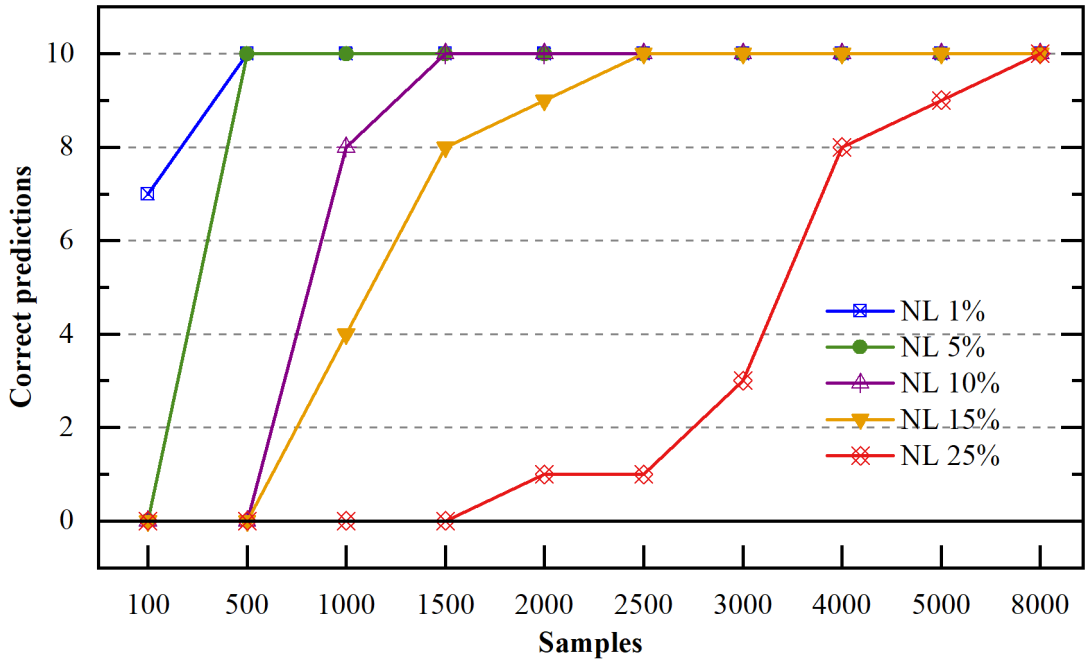

FIGURE 4.11: In the achievability plot, the correct predictions is plotted over the number of samples for the 2D-Advection Diffusion equation for different noise levels

samples, the identified PDEs are 8, and for the rest of the sample size, the identified PDEs are 10. Also, with the 15% noise level, the model could not recover the PDEs for 100 and 500 samples, and 1000, 1500, and 2000 samples, the correct predictions are 4, 8, and 9, respectively. For the rest of the sample size, the model predicts the entire PDEs correctly. The lowest PDEs are recovered with the 25% noise level compared to 1% and 5% noise levels. No correct PDE is identified from 100 to 1500 samples, and with 2000 and 2500 samples, the correct PDE is only 1. The model recovered 3, 8, and 9 PDEs respectively for 3000, 4000, and 5000 samples. Finally, all of the 10 PDEs are discovered for 8000 samples.

4.2.2 Average Coefficients and Average Coefficient Errors

The average coefficients ($\bar{\xi}_1$, $\bar{\xi}_2$, $\bar{\xi}_3$, $\bar{\xi}_4$) and the average coefficient errors ($\bar{\xi}_{1e}$, $\bar{\xi}_{2e}$, $\bar{\xi}_{3e}$, $\bar{\xi}_{4e}$) for the 2D-Advection-Diffusion equation are visualized in Figure 4.12. We have two advection and two diffusion, in total four coefficients in the 2D-Advection-Diffusion equation. The actual values of the advection coefficients Ξ_1 and Ξ_2 are 0.25 and 0.5 respectively and both of the diffusion coefficients (Ξ_3, Ξ_4) are 0.5. Here, only the results for 1%, 10% and 25% NL are presented in Figure 4.12.

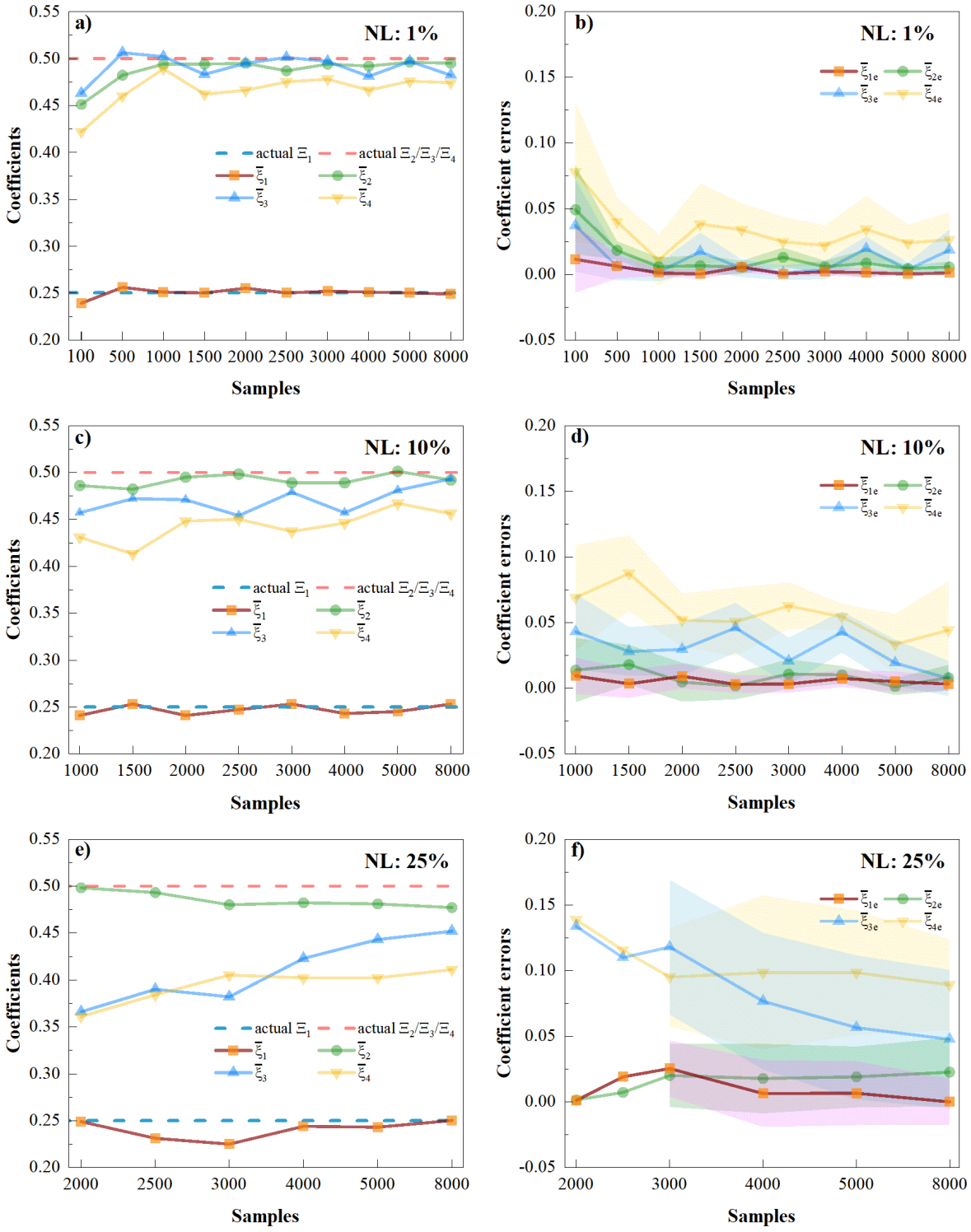

FIGURE 4.12: Average coefficients ($\bar{\xi}_1$, $\bar{\xi}_2$, $\bar{\xi}_3$, $\bar{\xi}_4$) and average coefficient errors ($\bar{\xi}_{1e}$, $\bar{\xi}_{2e}$, $\bar{\xi}_{3e}$, $\bar{\xi}_{4e}$) against samples size in 2D-Advection-Diffusion equation. Actual value of $\bar{\xi}_2$, $\bar{\xi}_3$, $\bar{\xi}_4$ are same (= 0.5). a), c), e) show coefficients for 1%, 10% and 25% noise level (NL); and b), d), f) show coefficient errors with standard deviation (shaded areas) for 1%, 10% and 25% NL respectively. The SD is 0 for 25% NL with 2000, 2500 samples because there is only one correct prediction, and the coefficient error and average coefficient error values are same.

Moreover, we would like to refer Figure B.2 in Appendix B for other noise levels (5% and 15%).

The average coefficients are shown in Figure 4.12 (a, c, e). With 1% noise level, the values of average coefficients $\bar{\xi}_2$, $\bar{\xi}_3$ and $\bar{\xi}_4$ are increased gradually from 100-1000 sample size, then reached at plateau. The coefficient values are all fluctuating at 10% NL, where there is a slight increase for $\bar{\xi}_3$, $\bar{\xi}_4$ at 25% NL. In addition, $\bar{\xi}_1$ values are also fluctuating where the deviation window is very narrow. For 25% NL, the average coefficients and the average coefficient errors are plotted for the higher sample size as we did not have any correct predictions for the lower number of sample size (100-1500).

Figure 4.12 (b, d, f) represent the corresponding average coefficient errors. It shows that the SD window is wider for 25% NL than others. Here the values of coefficients are decreasing sharply. On the contrary, those are decreasing slightly and fluctuating at 5% and 10% NL. Moreover, SD of $\bar{\xi}_{1e}$ is very narrower compared to others for all noise levels.

4.2.3 Dependency on Dictionary Size

We performed all of the above experiments where the dictionary matrix is built with first-order polynomial and second-order spatial derivatives of a total of twelve terms. We want to see if we expand the dictionary size, will it be possible to recover the correct PDE. For that, the neural network model is tested with the highest third-order spatial derivatives and third-order polynomials for 1% and 10% noise levels. The size of the dictionary matrix are 12, 18, 24, 30, and 40 for the combination of polynomial and spatial derivatives. The polynomial and spatial derivative orders, dictionary size, and the discovered PDE for 1% noise addition are tabulated in Table 4.4.

TABLE 4.4: Effect of dictionary size to recover PDE for the 2D-Advection-Diffusion equation with 1% noise level and 2000 samples

Total items in dictionary matrix for identifying 2D-Advection-Diffusion equation			
Spatial Order	Poly Order	Dictionary Size	Average PDE
2	1	12	$0.255u_x + 0.495u_y + 0.495u_{xx} + 0.466u_{yy}$
2	2	18	$0.254u_x + 0.504u_y + 0.495u_{xx} + 0.485u_{yy}$
2	3	24	$0.248u_x + 0.499u_y + 0.489u_{xx} + 0.490u_{yy}$
3	2	30	$0.252u_x + 0.504u_y + 0.484u_{xx} + 0.481u_{yy}$
3	3	40	$0.251u_x + 0.499u_y + 0.499u_{xx} + 0.489u_{yy}$

We compute the correct predictions for ten iterations against the total terms of the dictionary matrix for 1% and 10% noise level, as shown in Figure 4.13. When the noise level is 1%, the neural network model recovers all of the ten PDEs correctly for 12 total terms of the dictionary matrix, and for the rest of the terms, the correct predictions are 9. When the noise level is 10%, all of the recovered PDEs are correct only for 12 total terms of the dictionary matrix. Then the correct predictions dramatically decrease for 18 terms to 3. For 24 total terms, we have only one correct prediction, and for 30 and 40 terms, there is no correct prediction.

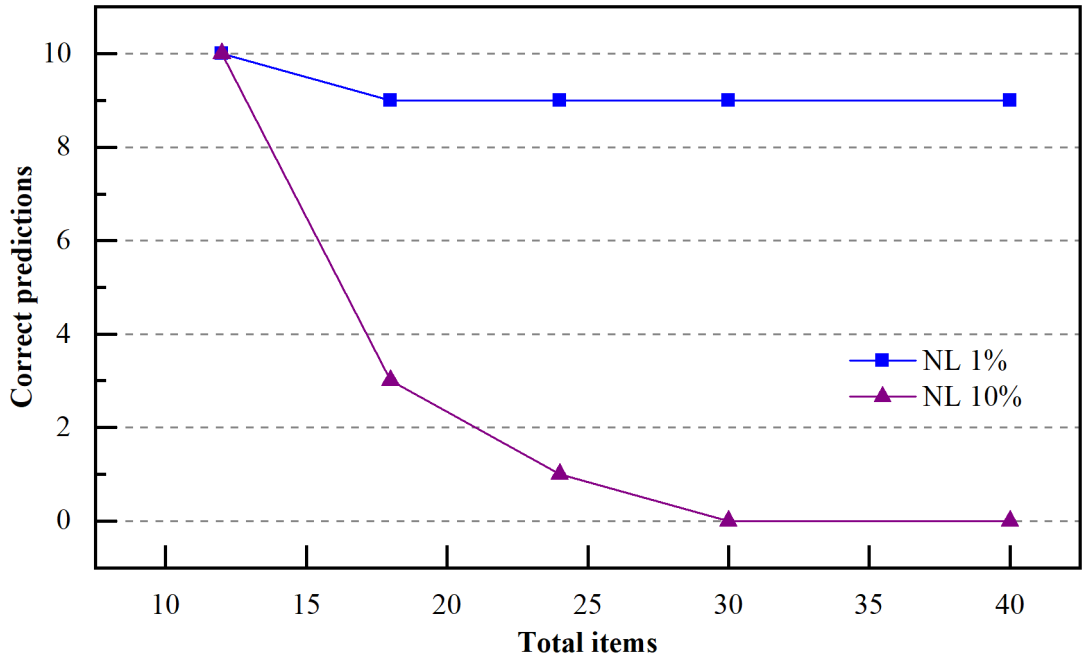

FIGURE 4.13: Correct predictions for ten iterations over total number of terms of the dictionary matrix for 2D-Advectin-Diffusion equation

In every iteration, it is possible to identify the two coefficients correctly for 12 terms. For the increasing number of terms, with a 10% noise addition, it is not possible to recover the PDEs successfully. The high-dimensional data could not identify the correct coefficients from lots of possibilities for the higher noise level, like 10%. In the existing DeepMoD [20], the authors did not show the results for more than twelve dictionary size for the respective noise levels.

4.2.4 Dependency on Nodes per layer

The neural network model is also tested for the 2D-Advection-Diffusion equation by changing neuron numbers for each hidden layer. The neural network contains five

hidden layers. First, ten neurons are considered for each hidden layer and we run the model for ten times. After that, twenty, thirty and forty neurons are chosen to observe if the changing number of neurons affects the results. The model is tested with 2000 samples and 1%, 10% and 25% noise levels. It is visible from Figure 4.14, for twenty, thirty and forty neurons, the model discovered the ten PDEs correctly for 1% and 10% NL. For 25% NL, there is only one correct prediction with ten and twenty neurons and no PDE is recovered with thirty and forty neurons.

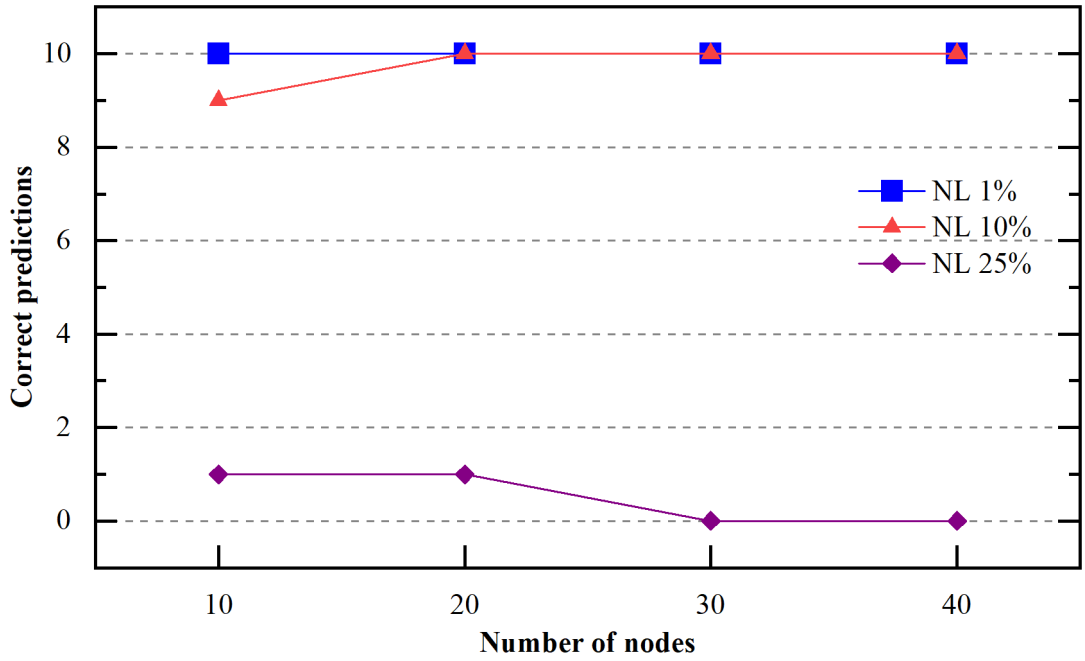

FIGURE 4.14: Correct predictions for ten iterations against the number of neurons in 2D-Advection-Diffusion equation for 1%, 10% and 25% noise level. The neural network model contains five hidden layers

The number of neurons does not affect significantly the recovery of the correct coefficients for building PDE, as tabulated in Table 4.5. The discovered coefficient values near to the actual values.

TABLE 4.5: Effect of nodes in neural network architecture to recover PDE for 2D-Advection-Diffusion

Effect of nodes in neural network architecture (noise level = 1%, samples = 2000)					
Nodes	$\bar{\xi}_{1e}$	$\bar{\xi}_{2e}$	$\bar{\xi}_{3e}$	$\bar{\xi}_{4e}$	Average PDE
10	0.006	0.005	0.049	0.073	$0.256u_x + 0.495u_y + 0.451u_{xx} + 0.427u_{yy}$
20	0.002	0.001	0.033	0.023	$0.248u_x + 0.501u_y + 0.467u_{xx} + 0.477u_{yy}$
30	0.0003	0.0037	0.0011	0.0149	$0.250u_x + 0.496u_y + 0.499u_{xx} + 0.485u_{yy}$
40	0.00034	0.0029	0.0028	0.0102	$0.250u_x + 0.497u_y + 0.497u_{xx} + 0.490u_{yy}$

4.2.5 Dependency on Hidden Layers

Our neural network model is tested with different number of layers, where each layer contains twenty neurons. The graphical representation of the number of correct predictions against the number of layers is shown in Figure 4.15. We can see, when the neural network model is constructed with one hidden layer, no single correct PDE is recovered from ten iterations for the defined noise levels. When there are two layers

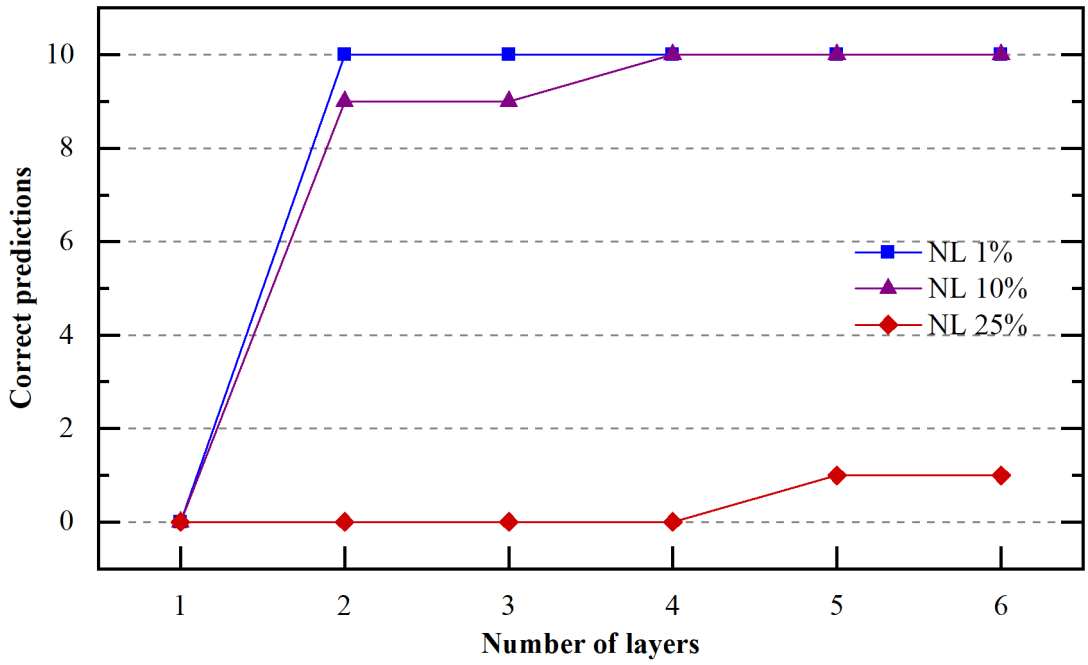

FIGURE 4.15: Correct predictions for ten iterations against the number of layers in the 2D-Advection-Diffusion equation for 1%, 10% and 25% noise levels, where each hidden layer contains twenty nodes

in the neural network, the correct predictions are 10, 9, respectively, for noise levels 1% and 10%. From layers four to six, the correct predictions are 10, i.e., the neural network model predicted the entire PDEs correctly for the noise levels 1% and 10%. With 25% NL, no PDE is recovered for the first four layers and the discovered PDE is only 1 for both fifth and sixth layers.
The recovered PDEs are given in Table 4.6. It is visible that the identified coefficient values are very close to the actual coefficient values from layer four to layer six for 1% noise level and 2000 samples.

TABLE 4.6: Effect of hidden layers in neural network architecture to recover PDE for 2D-Advection-Diffusion

Effect of hidden layers in neural network architecture (noise level = 1%, samples = 2000)					
Hidden Layers	$\bar{\xi}_{1e}$	$\bar{\xi}_{2e}$	$\bar{\xi}_{3e}$	$\bar{\xi}_{4e}$	Average PDE
1	-	-	-	-	No Recovered PDE
2	0.0042	0.0049	0.0082	0.0337	$0.254u_x + 0.495u_y + 0.492u_{xx} + 0.466u_{yy}$
3	0.0019	0.0094	0.0103	0.0329	$0.252u_x + 0.491u_y + 0.490u_{xx} + 0.467u_{yy}$
4	0.0085	0.0086	0.0163	0.05988	$0.259u_x + 0.491u_y + 0.484u_{xx} + 0.440u_{yy}$
5	0.0003	0.0009	0.0039	0.0189	$0.250u_x + 0.499u_y + 0.504u_{xx} + 0.481u_{yy}$
6	0.0019	0.0014	0.0051	0.0161	$0.252u_x + 0.499u_y + 0.495u_{xx} + 0.484u_{yy}$

In the existing DeepMoD [20] algorithm, the 2D-Advection-Diffusion equation recovered the correct PDE with up to 25% noise level by randomly selecting 5000 samples. It was also possible to recover PDE for 200 samples at the vanishing noise level.

5 Conclusion and Outlook

In this thesis, we are exploring the neural network-based technique DeepMoD for discovering the governing PDEs from noisy Spatio-temporal data. We have done systematic evaluation of the neural network model by changing the hyperparameters for different sample sizes on the Burgers equation and the 2D-Advection-Diffusion equation. Therefore, different values of hyperparameters have been chosen than the ones in DeepMoD, which shows that the hyperparameter affects the results. The number of layers and neurons did not affect the results for both equations. For the Burgers equation, the dictionary size also did not influence the results. It was possible to recover the entire PDE correctly with until total of twenty terms. However, for the 2D-Advection-Diffusion equation, it was not possible to recover the whole PDE with the increasing number of total terms of the dictionary matrix and the noise levels. Some important hyperparameter like learning rate (α), regularization parameters (λ) affects the overall outcome significantly. In this regard, we have tested the algorithm by changing the value of those hyperparameters. For that, the results and accuracy rate of recovered PDE were better than the existing DeepMoD.

In this work, we chose only two test cases: the Burgers and the 2D-Advection-Diffusion equations for the experiments. In the future, anyone can think about the Navier- Stokes equation, Kuramoto-Sivashinsky equation, Korteweg-de Vries (KdV) equation, and Keller-Segel equation.

Bibliography

[1] K. Rojko and D. Jelovac, "Challenges due to excessive amount of online data and (mis) information", in *Central European Conference on Information and Intelligent Systems*, Faculty of Organization and Informatics Varazdin, 2018, pp. 33–38.

[2] A. Stettner and D. P. Greenberg, "Computer graphics visualization for acoustic simulation", in *Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, 1989, pp. 195–206.

[3] C. D. Manning, C. D. Manning, and H. Schütze, *Foundations of statistical natural language processing*. MIT press, 1999.

[4] D. A. Forsyth and J. Ponce, *Computer vision: a modern approach*. Prentice Hall Professional Technical Reference, 2002.

[5] B. M. Lake, R. Salakhutdinov, and J. B. Tenenbaum, "Human-level concept learning through probabilistic program induction", *Science*, vol. 350, no. 6266, pp. 1332–1338, 2015.

[6] A. Karpatne, G. Atluri, J. H. Faghmous, M. Steinbach, A. Banerjee, A. Ganguly, S. Shekhar, N. Samatova, and V. Kumar, "Theory-guided data science: A new paradigm for scientific discovery from data", *IEEE Transactions on Knowledge and Data Engineering*, vol. 29, no. 10, pp. 2318–2331, 2017.

[7] T. P. Miyanawala and R. K. Jaiman, "An efficient deep learning technique for the navier-stokes equations: Application to unsteady wake flow dynamics", *arXiv preprint arXiv:1710.09099*, 2017.

[8] R. Maulik and O. San, "A neural network approach for the blind deconvolution of turbulent flows", *Journal of Fluid Mechanics*, vol. 831, pp. 151–181, 2017.

[9] N. Wagner and J. M. Rondinelli, "Theory-guided machine learning in materials science", *Frontiers in Materials*, vol. 3, p. 28, 2016.

[10] J. P. Crutchfield and B. S. McNamara, "Equations of motion from a data series", *Complex systems*, vol. 1, no. 417-452, p. 121, 1987.

[11] Z. Long, Y. Lu, X. Ma, and B. Dong, "Pde-net: Learning pdes from data", *arXiv preprint arXiv:1710.09668*, 2017.

[12] Z. Long, Y. Lu, and B. Dong, "Pde-net 2.0: Learning pdes from data with a numeric-symbolic hybrid deep network", *Journal of Computational Physics*, vol. 399, p. 108 925, 2019.

[13] S. Maddu, B. L. Cheeseman, I. F. Sbalzarini, and C. L. Müller, "Stability selection enables robust learning of partial differential equations from limited noisy data", *arXiv preprint arXiv:1907.07810*, 2019.

[14] M. Raissi, P. Perdikaris, and G. E. Karniadakis, "Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations", *arXiv preprint arXiv:1711.10561*, 2017.

[15] M. Raissi, P. Perdikaris, and G. Karniadakis, "Physics informed deep learning (part ii): Data-driven discovery of nonlinear partial differential equations, arxiv preprint (2017)", *arXiv preprint arXiv:1711.10566*,

[16] S. L. Brunton, J. L. Proctor, and J. N. Kutz, "Discovering governing equations from data by sparse identification of nonlinear dynamical systems", *Proceedings of the national academy of sciences*, vol. 113, no. 15, pp. 3932–3937, 2016.

[17] M. Quade, M. Abel, J. Nathan Kutz, and S. L. Brunton, "Sparse identification of nonlinear dynamics for rapid model recovery", *Chaos: An Interdisciplinary Journal of Nonlinear Science*, vol. 28, no. 6, p. 063 116, 2018.

[18] S. H. Rudy, S. L. Brunton, J. L. Proctor, and J. N. Kutz, "Data-driven discovery of partial differential equations", *Science Advances*, vol. 3, no. 4, e1602614, 2017.

[19] H. Schaeffer, "Learning partial differential equations via data discovery and sparse optimization", *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 473, no. 2197, p. 20 160 446, 2017.

[20] G.-J. Both, S. Choudhury, P. Sens, and R. Kusters, "Deepmod: Deep learning for model discovery in noisy data", *arXiv preprint arXiv:1904.09406*, 2019.

[21] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[22] A. K. Jain, J. Mao, and K. M. Mohiuddin, "Artificial neural networks: A tutorial", *Computer*, vol. 29, no. 3, pp. 31–44, 1996.

[23] I. A. Basheer and M. Hajmeer, "Artificial neural networks: Fundamentals, computing, design, and application", *Journal of microbiological methods*, vol. 43, no. 1, pp. 3–31, 2000.

[24] Wikipedia, *Biological neuron model*, Last accessed: 01 May, 2020. [Online]. Available: `https://en.wikipedia.org/wiki/Biological_neuron_model`.

[25] B. Karlik and A. V. Olgac, "Performance analysis of various activation functions in generalized mlp architectures of neural networks", *International Journal of Artificial Intelligence and Expert Systems*, vol. 1, no. 4, pp. 111–122, 2011.

[26] M. H. Sazlı, "A brief review of feed-forward neural networks", 2006.

[27] D. J. Montana and L. Davis, "Training feedforward neural networks using genetic algorithms.", in *IJCAI*, vol. 89, 1989, pp. 762–767.

[28] D. Svozil, V. Kvasnicka, and J. Pospichal, "Introduction to multi-layer feed-forward neural networks", *Chemometrics and intelligent laboratory systems*, vol. 39, no. 1, pp. 43–62, 1997.

[29] C. C. Aggarwal, "Neural networks and deep learning", *Springer*, vol. 10, pp. 978–3, 2018.

[30] M. A. Nielsen, *Neural networks and deep learning*. Determination press San Francisco, CA, USA: 2015, vol. 2018.

[31] K. Suzuki, *Artificial neural networks: methodological advances and biomedical applications*. BoD–Books on Demand, 2011.

[32] S. Ruder, "An overview of gradient descent optimization algorithms", *arXiv preprint arXiv:1609.04747*, 2016.

[33] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization", *arXiv preprint arXiv:1412.6980*, 2014.

[34] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization", *Journal of machine learning research*, vol. 12, no. Jul, pp. 2121–2159, 2011.

[35] T. Tieleman and G. Hinton, "Neural networks for machine learning", *Coursera (Lecture 65-RMSprop)*, 2012.

[36] N. I. Binti Zainuddin and M. D. J. Yacob, "Mathematical modelling of burger's equation applied in traffic flow", 2017.

[37] C. Basdevant, M. Deville, P. Haldenwang, J. Lacroix, J. Ouazzani, R. Peyret, P. Orlandi, and A. Patera, "Spectral and finite difference solutions of the burgers equation", *Computers & fluids*, vol. 14, no. 1, pp. 23–41, 1986.

[38] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, "Pytorch: An imperative style, high-performance deep learning library", in *Advances in Neural Information Processing Systems*, 2019, pp. 8024–8035.

[39] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, "Automatic differentiation in machine learning: A survey", *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 5595–5637, 2017.

[40] R. Tibshirani, "Regression shrinkage and selection via the lasso: A retrospective", *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, vol. 73, no. 3, pp. 273–282, 2011.

[41] H. Wang, G. Li, and C.-L. Tsai, "Regression coefficient and autoregressive order shrinkage and selection via the lasso", *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, vol. 69, no. 1, pp. 63–78, 2007.

List of Figures

List of Tables

A Values of Coefficients and Errors

A.1 Burgers Equation

TABLE A.1: Recovered Burgers equation for 1% noise level

Burgers equation experiment (noise level = 1%, $\lambda = 10^{-6}$ and tolerance = 10^{-6})					
Samples	$\bar{\xi}_1$	$\bar{\xi}_2$	$\bar{\xi}_{1e}$	$\bar{\xi}_{2e}$	Average PDE
100	0.09524016	0.90748896	0.00475984	0.09251104	$0.0952u_{xx}$ - $0.9075uu_x$
500	0.09881823	0.97481663	0.00118177	0.02518337	$0.0988u_{xx}$ - $0.9748uu_x$
1000	0.09828597	0.99662744	0.00171403	0.00337256	$0.0983u_{xx}$ - $0.9966uu_x$
1500	0.09922307	0.99383403	0.00077693	0.00616597	$0.0992u_{xx}$ - $0.9938uu_x$
2000	0.09945804	1.00494455	0.00054196	0.00494455	$0.0995u_{xx}$ - $1.0049uu_x$
2500	0.09965104	1.0069299	0.00034896	0.0069299	$0.0997u_{xx}$ - $1.0069uu_x$
3000	0.09810801	0.99196398	0.00189199	0.00803602	$0.0981u_{xx}$ - $0.9920uu_x$
4000	0.09928961	1.00971736	0.00071039	0.00971736	$0.0993u_{xx}$ - $1.0097uu_x$
5000	0.09974824	0.99552694	0.00025176	0.00447306	$0.0997u_{xx}$ - $0.9955uu_x$
8000	0.09955698	0.99919518	0.00044302	0.00080482	$0.0996u_{xx}$ - $0.9992uu_x$

TABLE A.2: Recovered Burgers equation for 5% noise level

Burgers equation experiment (noise level = 5%, $\lambda = 10^{-6}$ and tolerance = 10^{-6})					
Samples	$\bar{\xi}_1$	$\bar{\xi}_2$	$\bar{\xi}_{1e}$	$\bar{\xi}_{2e}$	Average PDE
100	0.10093689	0.8256466	0.00093689	0.1743534	$0.1009u_{xx}$ - $0.8256uu_x$
500	0.09802992	0.96215436	0.00197008	0.03784564	$0.0980u_{xx}$ - $0.9622uu_x$
1000	0.09913993	0.97312713	0.00086007	0.02687287	$0.0991u_{xx}$ - $0.9731uu_x$
1500	0.09909632	0.98906715	0.00090368	0.01093285	$0.0991u_{xx}$ - $0.9891uu_x$
2000	0.09856205	0.9989295	0.00143795	0.0010705	$0.0986u_{xx}$ - $0.9989uu_x$
2500	0.09919947	0.98956116	0.00080053	0.01043884	$0.0992u_{xx}$ - $0.9896uu_x$
3000	0.0998698	0.98369617	0.0001302	0.01630383	$0.0999u_{xx}$ - $0.9837uu_x$
4000	0.09813231	1.00062754	0.00186769	0.00062754	$0.0981u_{xx}$ - $1.0006uu_x$
5000	0.09985197	0.99461277	0.00014803	0.00538723	$0.0999u_{xx}$ - $0.9946uu_x$
8000	0.09957043	1.00003417	0.00042957	0.00003417	$0.0996u_{xx}$ - $1.0000uu_x$

TABLE A.3: Recovered Burgers equation for 10% noise level

Burgers equation experiment (noise level = 10%, $\lambda = 10^{-6}$ and tolerance = 10^{-6})					
Samples	$\bar{\xi}_1$	$\bar{\xi}_2$	$\bar{\xi}_{1e}$	$\bar{\xi}_{2e}$	Average PDE
100	0.0816921	0.8107755	0.0183079	0.1892245	$0.0817u_{xx}$ - $0.8108uu_x$
500	0.09711929	0.87700386	0.00288071	0.12299614	$0.0971u_{xx}$ - $0.8770uu_x$
1000	0.09719698	0.96646481	0.00280302	0.03353519	$0.0972u_{xx}$ - $0.9665uu_x$
1500	0.09917977	0.95165936	0.00082023	0.04834064	$0.0992u_{xx}$ - $0.9517uu_x$
2000	0.10019435	0.97099532	0.00019435	0.02900468	$0.1002u_{xx}$ - $0.9710uu_x$
2500	0.09938006	0.97136998	0.00061994	0.02863002	$0.0994u_{xx}$ - $0.9714uu_x$
3000	0.09988734	0.95875282	0.00011266	0.04124718	$0.0999u_{xx}$ - $0.9588uu_x$
4000	0.09934316	0.98109257	0.00065684	0.01890743	$0.0993u_{xx}$ - $0.9811uu_x$
5000	0.10033986	0.98034019	0.00033986	0.01965981	$0.1003u_{xx}$ - $0.9803uu_x$
8000	0.10008628	0.97213231	0.00008628	0.02786769	$0.1001u_{xx}$ - $0.9721uu_x$

TABLE A.4: Recovered Burgers equation for 15% noise level

Burgers equation experiment (noise level = 15%, $\lambda = 10^{-6}$ and tolerance = 10^{-6})					
Samples	$\bar{\xi}_1$	$\bar{\xi}_2$	$\bar{\xi}_{1e}$	$\bar{\xi}_{2e}$	Average PDE
100	0.1013846	0.88943505	0.0013846	0.11056495	$0.1014u_{xx}$ - $0.8894uu_x$
500	0.09074888	0.80685056	0.00925112	0.19314944	$0.0907u_{xx}$ - $0.8069uu_x$
1000	0.09795312	0.93657379	0.00204688	0.06342621	$0.0980u_{xx}$ - $0.9366uu_x$
1500	0.09581111	0.9016164	0.00418889	0.0983836	$0.0958u_{xx}$ - $0.9016uu_x$
2000	0.09978672	0.95912558	0.00021328	0.04087442	$0.0998u_{xx}$ - $0.9591uu_x$
2500	0.09998458	0.95348642	0.00001542	0.04651358	$0.1000u_{xx}$ - $0.9535uu_x$
3000	0.10148664	0.97018862	0.00148664	0.02981138	$0.1015u_{xx}$ - $0.9702uu_x$
4000	0.10055102	0.9567265	0.00055102	0.0432735	$0.1006u_{xx}$ - $0.9567uu_x$
5000	0.10027702	0.97557651	0.00027702	0.02442349	$0.1003u_{xx}$ - $0.9756uu_x$
8000	0.10027798	0.97080131	0.00027798	0.02919869	$0.1003u_{xx}$ - $0.9708uu_x$

TABLE A.5: Recovered Burgers equation for 25% noise level

Burgers equation experiment (noise level = 25%, $\lambda = 10^{-6}$ and tolerance = 10^{-6})					
Samples	$\bar{\xi}_1$	$\bar{\xi}_2$	$\bar{\xi}_{1e}$	$\bar{\xi}_{2e}$	Average PDE
100	0.09148441	0.64128816	0.00851559	0.35871184	$0.0915u_{xx}$ - $0.6413uu_x$
500	0.10218042	0.91142414	0.00218042	0.08857586	$0.1022u_{xx}$ - $0.9114uu_x$
1000	0.09677741	0.84739795	0.00322259	0.15260205	$0.0968u_{xx}$ - $0.8474uu_x$
1500	0.09669872	0.94416192	0.00330128	0.05583808	$0.0967u_{xx}$ - $0.9442uu_x$
2000	0.09654109	0.8944587	0.00345891	0.1055413	$0.0965u_{xx}$ - $0.8945uu_x$
2500	0.09907884	0.91799387	0.00092116	0.08200613	$0.0991u_{xx}$ - $0.9180uu_x$
3000	0.09829525	0.98511825	0.00170475	0.01488175	$0.0983u_{xx}$ - $0.9851uu_x$
4000	0.10073296	0.95930107	0.00073296	0.04069893	$0.1007u_{xx}$ - $0.9593uu_x$
5000	0.09966274	0.94382107	0.00033726	0.05617893	$0.0997u_{xx}$ - $0.9438uu_x$
8000	0.10210484	0.92713824	0.00210484	0.07286176	$0.1021u_{xx}$ - $0.9271uu_x$

A.2 2D-Advection-Diffusion Equation

TABLE A.6: Recovered 2D-Advection-Diffusion equation for 1% noise level on different number of randomly selected samples

Advection-Diffusion equation experiment (noise level = 1%, $\lambda = 10^{-6}$ and tolerance = 10^{-6})					
Samples	$\bar{\xi}_{1e}$	$\bar{\xi}_{2e}$	$\bar{\xi}_{3e}$	$\bar{\xi}_{4e}$	Average PDE
100	0.01146	0.04917	0.03690	0.07785	$0.239u_x + 0.451u_y + 0.463u_{xx} + 0.422u_{yy}$
500	0.00602	0.01818	0.00588	0.03988	$0.256u_x + 0.482u_y + 0.506u_{xx} + 0.460u_{yy}$
1000	0.00100	0.00573	0.00166	0.01075	$0.251u_x + 0.494u_y + 0.502u_{xx} + 0.489u_{yy}$
1500	0.00015	0.00655	0.01720	0.03818	$0.250u_x + 0.494u_y + 0.483u_{xx} + 0.462u_{yy}$
2000	0.00542	0.00524	0.00462	0.03385	$0.255u_x + 0.495u_y + 0.495u_{xx} + 0.466u_{yy}$
2500	0.00031	0.01284	0.00095	0.02477	$0.250u_x + 0.487u_y + 0.501u_{xx} + 0.475u_{yy}$
3000	0.00181	0.00564	0.00317	0.02194	$0.252u_x + 0.494u_y + 0.497u_{xx} + 0.478u_{yy}$
4000	0.00120	0.00840	0.01935	0.03431	$0.251u_x + 0.492u_y + 0.481u_{xx} + 0.466u_{yy}$
5000	0.00019	0.00448	0.00346	0.02391	$0.250u_x + 0.496u_y + 0.497u_{xx} + 0.476u_{yy}$
8000	0.00092	0.00545	0.01830	0.02651	$0.249u_x + 0.495u_y + 0.482u_{xx} + 0.474u_{yy}$

TABLE A.7: Recovered 2D-Advection-Diffusion equation for 5% noise level on different number of randomly selected samples

Advection-Diffusion equation experiment (noise level = 5%, $\lambda = 10^{-6}$ and tolerance = 10^{-6})					
Samples	$\bar{\xi}_{1e}$	$\bar{\xi}_{2e}$	$\bar{\xi}_{3e}$	$\bar{\xi}_{4e}$	Average PDE
100	-	-	-	-	No Recovered PDE
500	0.00463	0.03650	0.00825	0.09005	$0.255u_x + 0.464u_y + 0.508u_{xx} + 0.410u_{yy}$
1000	0.00231	0.01222	0.00549	0.05572	$0.252u_x + 0.488u_y + 0.495u_{xx} + 0.444u_{yy}$
1500	0.00181	0.01221	0.01953	0.06825	$0.252u_x + 0.488u_y + 0.481u_{xx} + 0.432u_{yy}$
2000	0.00058	0.00750	0.01375	0.03518	$0.251u_x + 0.493u_y + 0.486u_{xx} + 0.465u_{yy}$
2500	0.00629	0.00010	0.00054	0.02520	$0.256u_x + 0.500u_y + 0.500u_{xx} + 0.475u_{yy}$
3000	0.00016	0.01086	0.02175	0.04178	$0.250u_x + 0.489u_y + 0.472u_{xx} + 0.458u_{yy}$
4000	0.00682	0.00097	0.03432	0.02605	$0.243u_x + 0.501u_y + 0.466u_{xx} + 0.474u_{yy}$
5000	0.00248	0.00268	0.02649	0.03504	$0.248u_x + 0.497u_y + 0.474u_{xx} + 0.465u_{yy}$
8000	0.00038	0.00592	0.01898	0.03688	$0.250u_x + 0.494u_y + 0.481u_{xx} + 0.463u_{yy}$

TABLE A.8: Recovered 2D-Advection-Diffusion equation for 10% noise level on different number of randomly selected samples

Advection-Diffusion equation experiment (noise level = 10%, $\lambda = 10^{-6}$ and tolerance = 10^{-6})					
Samples	$\bar{\xi}_{1e}$	$\bar{\xi}_{2e}$	$\bar{\xi}_{3e}$	$\bar{\xi}_{4e}$	Average PDE
100	-	-	-	-	No Recovered PDE
500	-	-	-	-	No Recovered PDE
1000	0.00921	0.01376	0.04301	0.06895	$0.241u_x + 0.486u_y + 0.457u_{xx} + 0.431u_{yy}$
1500	0.00327	0.01787	0.02800	0.08737	$0.253u_x + 0.482u_y + 0.472u_{xx} + 0.413u_{yy}$
2000	0.00899	0.00461	0.02951	0.05182	$0.241u_x + 0.495u_y + 0.471u_{xx} + 0.448u_{yy}$
2500	0.00292	0.00163	0.04590	0.05047	$0.247u_x + 0.498u_y + 0.454u_{xx} + 0.450u_{yy}$
3000	0.00310	0.01083	0.02065	0.06269	$0.253u_x + 0.489u_y + 0.479u_{xx} + 0.437u_{yy}$
4000	0.00741	0.01010	0.04273	0.05430	$0.243u_x + 0.489u_y + 0.457u_{xx} + 0.446u_{yy}$
5000	0.00500	0.00127	0.01928	0.03348	$0.245u_x + 0.501u_y + 0.481u_{xx} + 0.467u_{yy}$
8000	0.00307	0.00809	0.00739	0.04422	$0.253u_x + 0.492u_y + 0.493u_{xx} + 0.456u_{yy}$

TABLE A.9: Recovered 2D-Advection-Diffusion equation for 15% noise level on different number of randomly selected samples

Advection-Diffusion equation experiment (noise level = 15%, $\lambda = 10^{-6}$ and tolerance = 10^{-6})					
Samples	$\bar{\xi}_{1e}$	$\bar{\xi}_{2e}$	$\bar{\xi}_{3e}$	$\bar{\xi}_{4e}$	Average PDE
100	-	-	-	-	No Recovered PDE
500	-	-	-	-	No Recovered PDE
1000	0.00154	0.03088	0.06679	0.12134	$0.248u_x + 0.469u_y + 0.433u_{xx} + 0.379u_{yy}$
1500	0.00300	0.02245	0.06598	0.11775	$0.247u_x + 0.478u_y + 0.434u_{xx} + 0.382u_{yy}$
2000	0.00822	0.01748	0.07667	0.10296	$0.242u_x + 0.483u_y + 0.423u_{xx} + 0.397u_{yy}$
2500	0.00781	0.01815	0.06415	0.08294	$0.242u_x + 0.482u_y + 0.436u_{xx} + 0.417u_{yy}$
3000	0.00651	0.00703	0.05589	0.06174	$0.243u_x + 0.493u_y + 0.444u_{xx} + 0.438u_{yy}$
4000	0.01147	0.00363	0.05641	0.05177	$0.239u_x + 0.496u_y + 0.444u_{xx} + 0.448u_{yy}$
5000	0.00290	0.01255	0.03685	0.02806	$0.247u_x + 0.513u_y + 0.463u_{xx} + 0.472u_{yy}$
8000	0.00160	0.00416	0.01347	0.03811	$0.252u_x + 0.496u_y + 0.487u_{xx} + 0.462u_{yy}$

TABLE A.10: Recovered 2D-Advection-Diffusion equation for 25% noise level on different number of randomly selected samples

Advection-Diffusion equation experiment (noise level = 25%, $\lambda = 10^{-6}$ and tolerance = 10^{-6})					
Samples	$\bar{\xi}_{1e}$	$\bar{\xi}_{2e}$	$\bar{\xi}_{3e}$	$\bar{\xi}_{4e}$	Average PDE
100	-	-	-	-	No Recovered PDE
500	-	-	-	-	No Recovered PDE
1000	-	-	-	-	No Recovered PDE
1500	-	-	-	-	No Recovered PDE
2000	0.00096	0.00151	0.13356	0.13871	$0.249u_x + 0.498u_y + 0.366u_{xx} + 0.361u_{yy}$
2500	0.01902	0.00724	0.10974	0.11557	$0.231u_x + 0.493u_y + 0.390u_{xx} + 0.384u_{yy}$
3000	0.02529	0.02003	0.11791	0.09492	$0.225u_x + 0.480u_y + 0.382u_{xx} + 0.405u_{yy}$
4000	0.00635	0.01772	0.07656	0.09844	$0.244u_x + 0.482u_y + 0.423u_{xx} + 0.402u_{yy}$
5000	0.00652	0.01893	0.05657	0.09838	$0.243u_x + 0.481u_y + 0.443u_{xx} + 0.402u_{yy}$
8000	0.00000	0.02270	0.04766	0.08912	$0.250u_x + 0.477u_y + 0.452u_{xx} + 0.411u_{yy}$

B Graphical Presentation

B.1 Burgers Equation

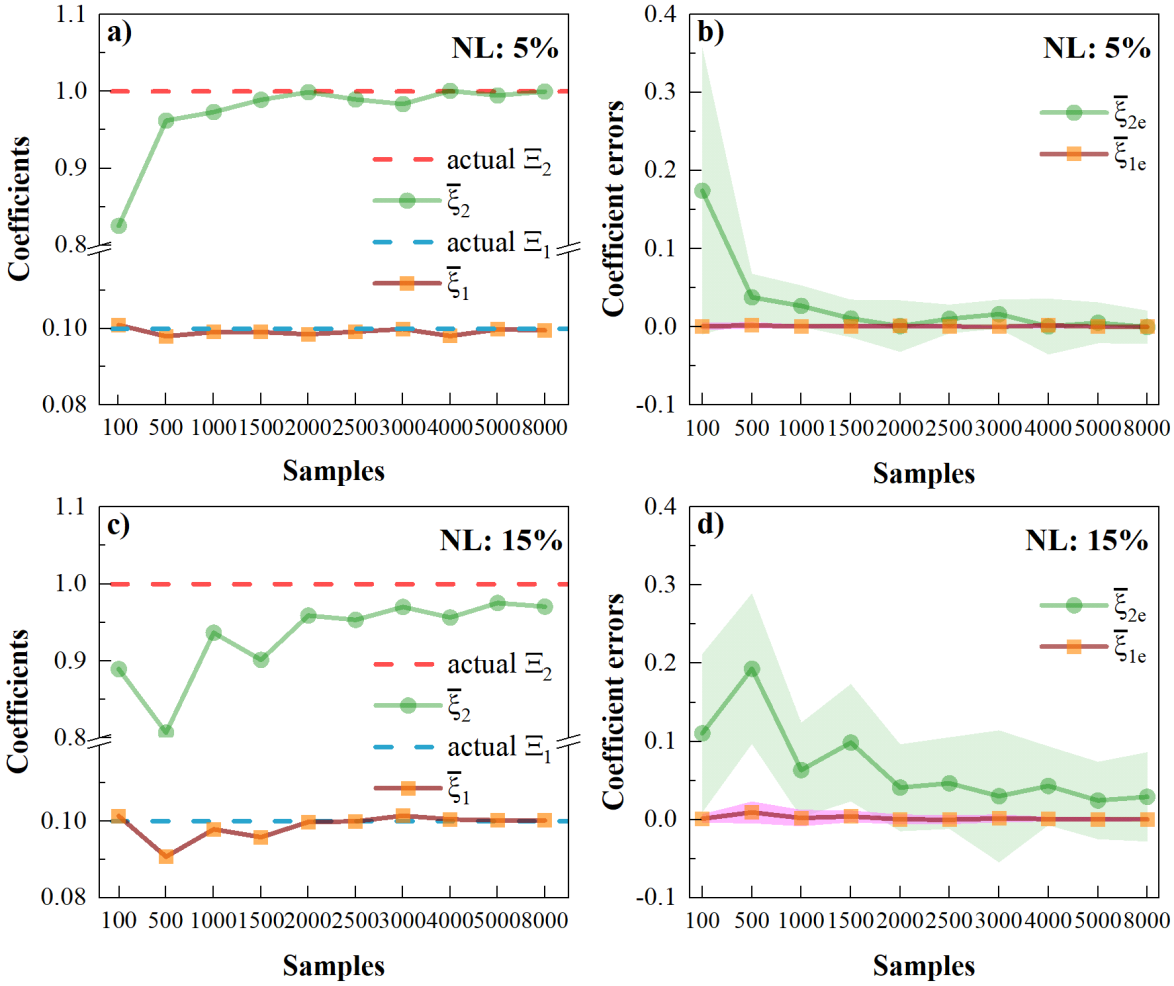

FIGURE B.1: Average coefficients ($\bar{\xi}_1$ and $\bar{\xi}_2$) and coefficient errors ($\bar{\xi}_{1e}$ and $\bar{\xi}_{2e}$) against samples in Burgers equation. a), c) show coefficients for 5% and 15% noise level (NL); and b), d) show coefficient errors with standard deviation (SD, shaded areas) for 5% and 15% NL respectively.

B.2 2D-Advection-Diffusion

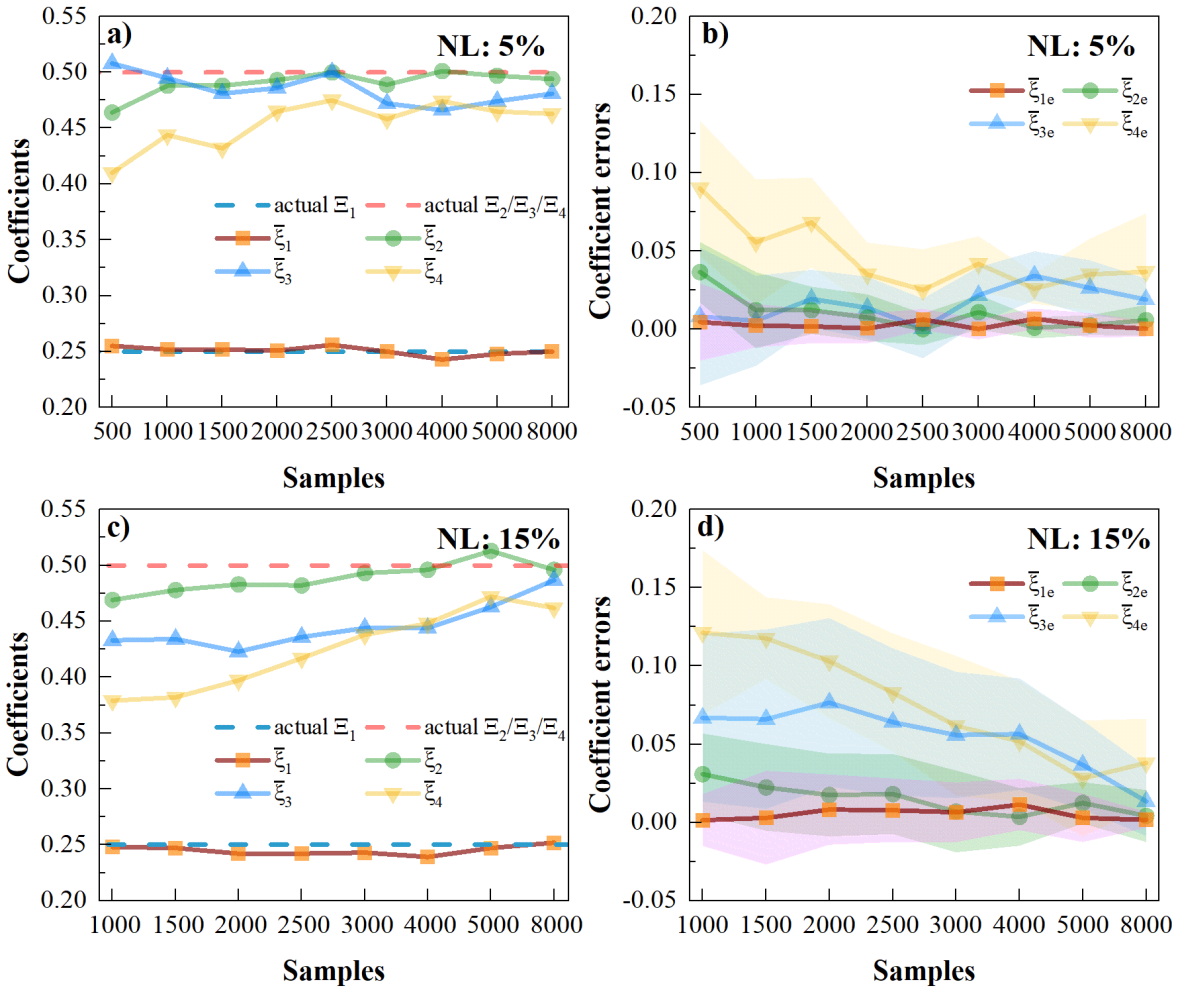

FIGURE B.2: Average coefficients ($\bar{\xi}_1$, $\bar{\xi}_2$, $\bar{\xi}_3$, $\bar{\xi}_4$) and coefficient errors ($\bar{\xi}_{1e}$, $\bar{\xi}_{2e}$, $\bar{\xi}_{3e}$, $\bar{\xi}_{4e}$) against samples size in 2D-Advection-Diffusion equation. Actual value of $\bar{\xi}_2$, $\bar{\xi}_3$, $\bar{\xi}_4$ are same and it is 0.5. a), c) show coefficients for 5% and 15% noise level (NL); and b), d) show coefficient errors with standard deviation (shaded areas) for 5% and 15% NL respectively.