TRIBHUVAN UNIVERSITY
INSTITUTE OF ENGINEERING
PULCHOWK CAMPUS

THESIS NO:

Data-Driven Discovery of Governing Equations

by

Rojesh Man Shikhrakar

A THESIS
SUBMITTED TO THE DEPARTMENT OF MECHANICAL ENGINEERING
IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF MASTER IN
MECHANICAL SYSTEMS DESIGN AND ENGINEERING

DEPARTMENT OF MECHANICAL AND AEROSPACE ENGINEERING
LALITPUR, NEPAL

JULY, 2020

COPYRIGHT

TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

PULCHOWK CAMPUS

DEPARTMENT OF MECHANICAL AND AEROSPACE ENGINEERING

The undersigned certify that they have read, and recommended to the Institute of Engineering for acceptance, a thesis entitled Data-Driven Discovery of Governing Equations submitted by Rojesh Man Shikhrakar in partial fulfillment of the requirements for the degree of Master in Department of Mechanical and Aerospace Engineering.

Supervisor, Dr. Laxman Poudel,
Professor
Department of Mechanical and Aerospace Engineering

Supervisor, Kamal Darlami,
Assistant Professor
Department of Mechanical and Aerospace Engineering,

External Examiner, Dr. Gajendra Sharma,
Associate Professor
Department of Computer Science and Engineering,
Kathmandu University,

Committee Chair, Dr. Nawraj Bhattarai,
Head of Department
Department of Mechanical and Aerospace Engineering,

Date: _____

ABSTRACT

Theoretical equations are the basis of scientific progress. Many scientific domains still lack the appropriate theoretical model to reason about the phenomena. With the rise of data, there is an increasing need for methodology in data-driven science and engineering for understanding the physical phenomena. This thesis on Data-Driven Discovery of Governing Equations aims to provide a method for model discovery and find governing partial differential equations from data by training physics-informed neural networks. Given data, our method generalizes a neural network to compute a matrix of candidate terms for Partial Differential Equation(PDE). Minimizing the residuals from the candidate matrix allows us to find the coefficients for the governing equation. We present a framework to discover PDE not restricted to first-order time derivative equations.

Key Words: Data-Driven Discovery, Partial Differential Equation, Parameter Estimation, Sparse Optimization, Machine Learning, Discrete Inverse Problem

ACKNOWLEDGMENT

This thesis would not have been possible without the inspiration and support of many wonderful individuals — my thanks and appreciation to all of them for being part of this journey and making this thesis possible. I owe my sincere gratitude to my supervisors to Prof. Dr. Laxman Poudel, and Asst. Prof. Kamal Darlami, Pulchowk Campus. Without their continuous support, encouragement, and guidance, this research would not have completed in time. I would like to thank all of the professors, lecturers, and colleagues from the Department of Mechanical and Aerospace Engineering, Pulchowk Campus, for their encouragement and support in the completion of this thesis work.

In conducting the research and writing the thesis on Data-Driven Discovery of Governing Equations, I have incurred some intellectual debts. I would like to express my gratitude to Mr. Xiaolong He, Ph. D. student at UC SanDiego, and Dr. Rakesh Katuwal, Sr. ML Engineer, Fusemachines Inc., to provide in-depth suggestions that have enabled many improvements in this work.

I would like to take this moment to praise all of my friends in the MSMDE program, especially Milan Adhikari and Hemanta Dulal, who have been immensely helpful throughout these years at Pulchowk Campus.

Finally, my deep and sincere gratitude to my mother, Mrs. Roshani Shikhrakar, for her continuous and unparalleled love, and to my father, Mr. Rajman Shikhrakar, who inspired me to be curious and choose a scientific career. I am grateful to my sister for always being mischievous and supportive. I am forever indebted to my parents for giving me the opportunities and experiences that have made me who I am. This journey would not have been possible if not for their encouragement, and I dedicate this milestone to them.

Sincerely,
Rojesh Man Shikhrakar

Contents

List of Figures

List of Tables

List of Abbreviations

AD Automatic Differentiation.

DNN Deep Neural Network.

ELU Exponential Linear Unit.

MAE Mean Absolute Error.

MLP Multi-Layer Perceptron.

MSE Mean Square Error.

NFL No Free Lunch.

OOD Out-of-Distribution.

PDE Partial Differential Equation.

PINN Physics-Informed Neural Network.

ReLU Rectifier Linear Unit.

RMSE Root Mean Square Error.

SINDy Sparse Identification of Non-linear Dynamics.

SVD Singular Value Decomposition.

TGDS Theory Guided Data Science.

UAT Universal Approximation Theorem.

List of Symbols

C_i i^{th} candidate function.

D^k k^{th} order partial derivative.

H Matrix of candidate terms evaluated at collocation points.

N Number of Samples.

W Weights of neural network.

Θ Coefficient Vector.

$\hat{u}$ Output of Neural Network.

λ_f Coefficient for $\mathcal{L}_f$.

λ_u Coefficient for $\mathcal{L}_u$.

λ_{norm} Coefficient for $\mathcal{L}_{norm}$.

$\mathcal{L}$ Combined loss used to train the neural network.

$\mathcal{L}_f$ Residual in PDE approximation.

$\mathcal{L}_u$ Residual in fitting the model function $(u_i - \hat{u}_i)$.

$\mathcal{L}_{norm}$ L_1 regularization on parameter Θ.

u Response variable or function.

x Independent variable (indicate both spatial and temporal unless otherwise specified).

CHAPTER ONE: INTRODUCTION

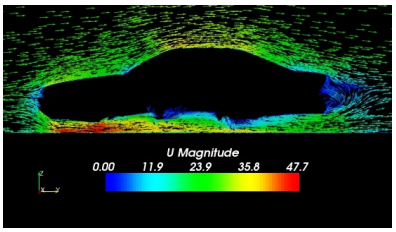
1.1 Background

For thousands of years, scientific inquiry has been empirical; we observe or replicate the natural world in a controlled environment and record the observations. In the last few hundred years, scientists have accepted theoretical models as a valid method of inquiry because they suggest new experiments and explain the observed data. Theories and experiments have been the two foundational pillars of science for our understanding of the universe. In the last fifty years, the advent of high-speed computation has allowed us to simulated phenomena that we cannot observe directly or reproduce. Computers allowed scientists to simulate complex systems at the massive scale of the universe to the small quantum realm.

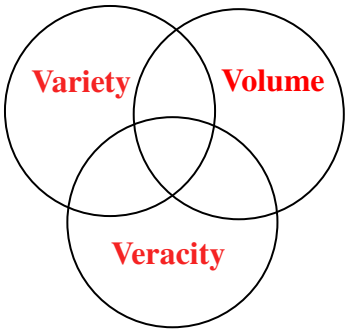
(a) Empirical Science,
Source: science.uu.nl

$$F = G\frac{M * m}{r^2}$$

(b) Theoretical Science



(c) CFD over a car,
Source: afs.enea.it



(d) Data Science

Figure 1.1: History of Scientific Discovery

Beginning in the 21st century, so-called "the golden age of data science" (or eScience), the rise of massive data from sensors with advances in machine learning and data processing have made impacts in different domains, providing simple analysis tools to knowledge discovery frameworks.

However, scientists have relied on their ability to predict complex phenomena by recording the observations and modeling them into parsimonious mathematical models.

Nevertheless, the extraction of these mathematical equations from experimental data, often in the form of differential and partial differential equations (PDEs), is a challenging endeavor that takes ingenious imagination and years to perfect. These PDEs are ubiquitous over diverse quantitative disciplines, from engineering to basic sciences and physics to economics; however, in many cases, these equations are unknown. These equations are commonly derived using first principle approaches satisfying the data observations. Using traditional methods, one can model these observations into equations, but discovering the underlying hidden physical law is much more complicated. For example, using Tycho Brahe's planetary data, Kepler discovered elliptical orbits, but this attractor-based law did not reflect the system's hidden dynamics. Newton later derived the actual governing equation describing the orbital motion.

Using machine learning methods to analyze massive sensor data and recognize patterns Bishop (2006), we can uncover the hidden governing equations for simple to complex natural phenomena, which will be helpful in domains that lack well-defined quantitative equations. However, machine learning models fundamentally assume that data for training and testing are from the same distribution. This assumption is more visible in deep learning methods Goodfellow et al. (2016) than in traditional machine learning methods such as decision trees and others. Recent theory-guided data science (TGDS) Karpatne et al. (2017) methods attempt to integrate existing scientific theory into the data science model for inference and prediction.

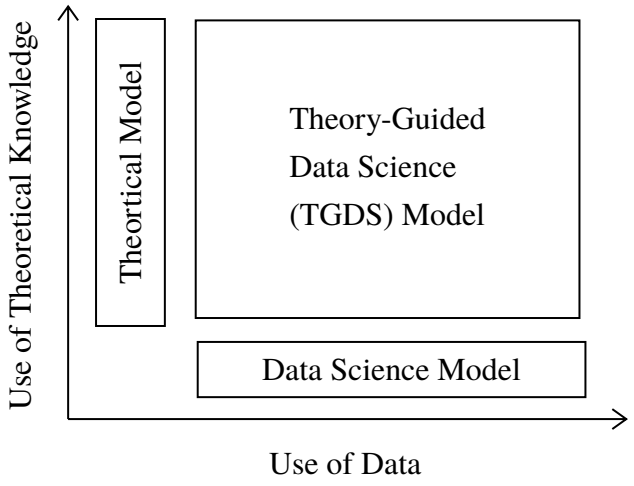


Figure 1.2: presents knowledge discovery methods along the "use of data" axis vs the "use of scientific knowledge" axis (Karpatne et al. (2017)

).

Theory-guided data science explores the space in search of knowledge using available data while being guided by the underlying scientific laws.

1.1.1 Research Gap

Many systems of interest are generally modeled as a dynamical system, i.e., the system's state evolves over time. Many unexplored domains lack governing equations such as fluid-structure interaction, biological systems, and economic systems. The combination of ample dynamic data and well-developed theory for dynamical systems motivates methods for identifying governing theoretical models from data. However, we have to find the theoretical rules from the first principles or empirically.

There is still a gap in research that allows researchers to use data to discover governing equations.This thesis attempts to apply a theory-guided technique to discover theoretical models from the machine learning model trained on data.

1.2 Objective

1.2.1 Main Objectives

1. The goal of this study is to develop a method to discover the governing equations, in the form of PDE, from state measurements of a system.

1.2.2 Specific Objectives

1. To develop a machine learning model to capture the underlying model from training data.

2. To study the effect of different hyper-parameters of the model on prediction performance.

3. To validate the prediction of law (equation) for various types of PDE equations.

1.3 Scopes

The scope of this research is limited to discrete dynamical systems, and only three equations are considered viz. wave equation, inviscid Burgers equation, and Helmholtz equation.

In real life situation, the physics are generally multi-body, multi-domain, and multi-physics problem. Many problems have intermittent dynamics, while many are transient. Those types of systems are not considered in the current research. Furthermore, systems with couple PDE are also not covered.

This study does not study the effect of boundary conditions in the discovery of PDE.

This study is an empirical study on the data-driven discovery of PDE, formal theoretical underpinning of the methods are not discussed in this research.

1.4 Structure

This thesis is organized as follows. Chapter 2 presents related research and a few background material in the discovery of the governing equation. Chapter 3 discuss the research methodology. It starts with a formal definition of the PDE estimation problem, talks about the overall research design, and finally discusses the discovery method used. Chapter 4 present the numerical results and discusses some intuition about them. Chapter 5 concludes the report with limitations of this research, and possible improvements on this method.

CHAPTER TWO: LITERATURE REVIEW

2.1 Deep Learning and Neural Networks

Deep learning is a subset of machine learning, a field dedicated to the research and development of systems that can learn from data and experience without being explicitly programmed. Mitchell (1997) defines machine learning as a computer program that is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

Many industries are applying machine learning and mostly deep learning to solve wide ranges of practical tasks in a variety of fields such as natural language processing (text), automatic speech recognition (audio), computer vision (image), forecasting, and recommendation engines. In a nutshell, deep learning builds multiple deep layers of artificial neural networks, which are a class of algorithm loosely inspired by the human brain (Trask 2019). The machine learning algorithms (including deep learning), are generally classified as supervised, unsupervised, and reinforcement learning. If training data consist of features and labels, the algorithms learn to transform these features into labels; we call these algorithms supervised learning methods. If the training data set is not labeled or labels are unknown or not clearly understood, we generally cluster them into similar groups, these types of algorithms are called unsupervised methods. While reinforcement learning differs from both of these methods, it relies on rewards as feedback, for acting on the environment.

2.1.1 Neural Network

A neuron unit in an artificial neural network receives inputs and combines them with some internal weights and applies a nonlinear activation function to produce an output, as shown in fig. 2.1. The activation function such as sigmoid, Rectifier Linear Unit (ReLU), Exponential Linear Unit (ELU), Softplus, introduces non-linearity into the output of a neuron.

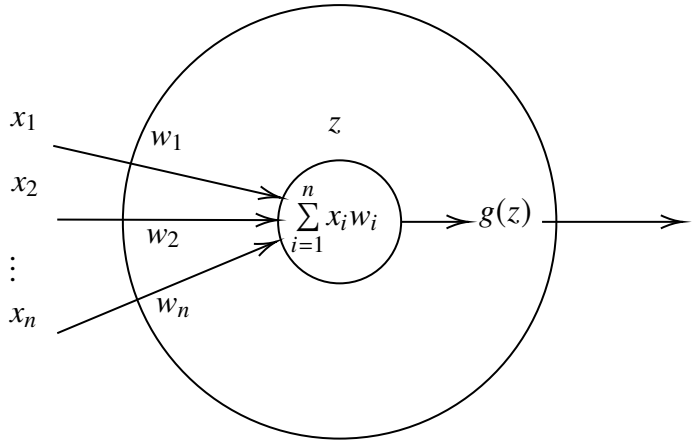


Figure 2.1: Artificial neuron unit

A large number of neurons can be stacked one after another and parallel to each other, as shown in fig. 2.2 to form a Multi-Layer Perceptron (MLP). MLPs with larger depth are generally known as feed-forward Deep Neural Network (DNN). In addition to the input or the output from previous neurons, an additional bias term can also be added to the next neuron's input to adjust the output value. The addition of bias provides better flexibility and generalization to the model.

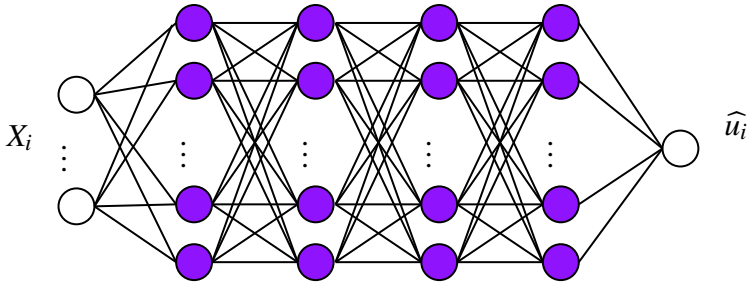


Figure 2.2: Multi Layer Perceptron (MLP)

The forward propagation computes the output $\hat{y}_i$ from input X_i through a series of computations in different neural layers. Researchers proposed different methods to estimate the weights w for each neuron connection. Hebbian learning and perceptron learning algorithm are among the oldest methods. Other methods such as Boltzmann machine learning, back-propagation (Rumelhart et al. 1986), target-propagation (Lee et al. 2015), Direct feedback alignment (Nøkland 2016), and genetic algorithms can also train neural networks. Among these, back-propagation is extensively in various industries dues to its success in several domains.

Back-propagation calculates the partial derivative $\frac{\partial L}{\partial w}$ of the cost function L with respect to any weight w (or bias b) in the network. We will use reverse-mode automatic differentiation, another name for back-propagation, as stated in section 2.5.2.

2.2 Mathematical Models

Scientists model complex systems in many applications generally into three types of mathematical models.

1. Analytical models derived from basic principles of physics that describe the behavior of the phenomena. The theoretical model of mechanics are examples of these models.

2. Black-Box model which uses a standard model and adjust its parameters to the data. Most of the machine learning models are black-box models.

3. Grey-Box model where a general form is derived and parameters adjusted to the data. There are hybrid for of model

In a broad manner, scientists and engineers write the mathematical model to relate physical parameters characterizing the model, $\mathbf{m} \in \mathbb{R}^n$, to observations or data, $\mathbf{d} \in \mathbb{R}^m$ with a function or operator $\mathcal{H}$.

$$H(\mathbf{m}) = \mathbf{d} \tag{2.1}$$

In reality, the data $\mathbf{d}$ is noisy due to some random additive noise η from our sensors or from numerical truncation during calculations. This noise is independent of our model or data. Hence,

$$H(\mathbf{m}) + \eta = \mathbf{d} \tag{2.2}$$

Here, the model is a n dimensional vector representing different parameters or coefficients of an equation. Similarly, $\mathbf{d}$ is m discrete points of observations. And H is a function that maps the data parameters to the data points. When $\mathbf{d}$ and $\mathbf{m}$ are both functions,$\mathcal{H}$ act as an operator on them. In this report, only discrete problems are considered.

There are three broad types of problem that comes out of eq. (2.1), which are as follows (Aster et al. 2019):

1. **Forward Problem:**
   This is the direct case to find output $\mathbf{d}$ given model parameters $\mathbf{m}$ and $\mathcal{H}$. This is generally straight forward solutions of algebraic, differential, integral equations.

2. **Inverse or Parameter Estimation Problem:**
   Here we find the model parameters $\mathbf{m}$ given some data points $\mathbf{d}$ and the model $\mathcal{H}$ is also known. Since there can be no or many models that adequately fit the data, computing such inverse solutions are often unstable.

3. **Model or System Identification Problem:**
   Model identification is the most complicated problem among the three. We search for the function or model $\mathcal{H}$ given examples of $\mathbf{m}$ and $\mathbf{d}$. We often must estimate both $\mathcal{H}$ and $\mathbf{m}$ from the data as well. Given a data set, one can fit thousands of models; hence over-fitting is a real danger.

In Discrete Linear Problems H is generally a matrix.

$$\mathcal{H}(\mathbf{m}) = \mathbf{Hm} = \mathbf{d} \tag{2.3}$$

We often solve nonlinear problems by converting them into a sequence of Linear Problems.

Linear equations are more commonly solved in two forms:

1. homogeneous linear equation, $\mathbf{Ax} = \mathbf{0}$
2. non-homogeneous linear equation, $\mathbf{Ax} = \mathbf{b}$

A non-homogeneous equation general can have either no solution, unique solution or infinitely many solution, while a homogeneous solution can have trivial solution (i.e $\mathbf{0}$) or infinitely many solution.

In reality, the matrix $\mathbf{A}$ is rarely square; we cannot solve them to get a unique equation. Hence, two cases arise with these systems.

1. over-determined system, where $m > n$, i.e., the number of discrete observations m is greater than the number of parameters in the equation n. In other words, we collect a large number of data points over time. Since these systems of equations have more equations than the number of unknowns, they lead to an inconsistent solution.
2. under-determined system, where $n > m$, i.e., the number of parameter n is greater than the number of measurements m. In other words, we collect only a few samples with large numbers of features. Since the system has fewer equations but more number of unknowns, they have infinite numbers of possible solutions.

Since we are focusing on dynamical systems with time-series data collected over time, we are dealing with over-determined systems, where $m >> n$, i.e., the number of measurements is much larger than the number of parameters. The over-determined system generally has an inconsistent solution. Hence, we resort to optimization methods to find the "best-fit" solution that minimizes specific criteria.

2.3 Model Selection and System Identification

System identification, defined as a field of modeling dynamic systems from experimental data, dates back to Gauss (1809). The modern identification theory used in the control system (Åström & P. 1971) was the first method used to identify fixed form models.

There are limitless volumes of data and many parametric or non-parametric models, but there is no model that is universally suitable for any data and objectives. An invalid model choice might lead to misleading conclusions and false predictions. Hence, model selection from a set of candidate models is an essential task in the statistical analysis, which is also central to the pursuit of science. Hence, different fields such as machine learning, statistics, epidemiology, chemometrics apply different model selection methods to fit their needs.

In probability and information theory, many methods can compare two distributions. Kullback-Leibler(KL) Divergence (1950) measures the non-symmetric difference between the statistical distribution of the data (p(x)), and a candidate model distribution (q(x)). Akaike Information Criteria (AIC) formalized the selection idea for a set of models, and use the relative distance of one of the candidate models and select the one with the lowest score (Akaike 1974). While Bayes Information Criteria (BIC) provided guarantee of convergence to the right model given the right model is in the set of candidate models (Schwarz 1978). Only a handful of models can be checked by these methods and compared to the data. These techniques are still used for judging the quality of a model.

On the other hand, modeling of dynamical systems (both linear and nonlinear) from data, focused on building linear models. Koopman analysis converted finite-dimensional nonlinear dynamics to linear dynamics in an infinite-dimensional Hilbert space (Koopman 1931). In practical applications, finite-dimensional approximations to the infinite-dimensional linear operator known as the Koopman operator are applied. Dynamic Mode Decomposition (DMD) (Schmid 2010) is a popular method for approximating Koopman operator to extract dynamic modes from the fluid system. It was primarily used to decompose a massive problem to few degrees of freedom. However, even though DMD captured linear dynamical systems well, it does not perform well for strongly nonlinear systems. This motivated approaches for discovering nonlinear dynamical systems models.

The following section discusses the three mainstream methods developed to discover governing PDEs of a physical system, viz. symbolic regression, sparse optimization methods, and hybrid frameworks.

2.4 Symbolic Regression

Symbolic regression (Koza 1992) is an evolutionary computation based method for searching a space of mathematical expressions. It is constructed by predefined analytical functions, constant coefficients, and algebraic operators, in such a way to minimize specific metrics of error. The symbolic regression methods (Bongard & Lipson 2007, Schmidt & Lipson 2009) were shown to extract "free-form" physical laws, of any kind in theory, such as the Hamiltonian, Lagrangian, momentum conservation and equations of motion for some fundamental and straightforward physical systems.

Symbolic regression is more general than the linear regression method. It attempts to fit a wider range of dataset as it does not rely on a predefined set of features. The search space of the mathematical expression to find the best model is very large due to the combinatorial nature of the method. Hence, symbolic regression methods are likely to face scaling and over-fitting problems. Schmidt & Lipson (2009) also proposed a principle for the identification of non-triviality based on the fact that governing equations have fewer terms and used Pareto front to find parsimonious models.

2.5 Sparse Regression

Given the fact that most governing equations exist with simpler relevant terms, sparse methods were employed. The sparse optimization methods such as Sparse Identification of Non-linear Dynamics (SINDy) framework (Brunton et al. 2016) can identify first-order differential equations of non-linear dynamical systems by expressing the first-order time derivative as linear combinations of candidate functions and determine the unknown coefficients to minimize specific metrics. These method considers dynamical systems of the form

$$\frac{d}{dt}\mathbf{x}(t) = \mathbf{f}(\mathbf{x}(t)),\ where\ \mathbf{x}(t) \in \mathbb{R}^n \tag{2.4}$$

This method work in the following manner:

Algorithm 1 Sparse Identification of Non-linear Dynamics (SINDy)

1: Collect m time series data points in n dimension $X \in \mathbb{R}^{m*n}$ with noise and either measure or calculate derivatives $\dot{X}(t) \in \mathbb{R}^{m*n}$ with numerical methods.

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

2: Build Library $\Theta(\mathbf{X}) \in \mathbb{R}^{m*p}$ of candidate nonlinear functions of columns of X, where, p is the number of candidate terms

$$\Theta(\mathbf{X}) = \begin{bmatrix} 1 & \mathbf{X} & \mathbf{X^{P_2}} & \mathbf{X^{P_3}} & \cdots & \sin(\mathbf{X}) & \cos(\mathbf{X}) \end{bmatrix}$$

where, $\mathbf{X^{P_2}}$ represent Quadratic non-linearities

$$\mathbf{X^{P_2}} = \begin{bmatrix} x_1^2(t_1) & x_1(t_1) * x_2(t_1) & \cdots & x_2^2(t_1) & \cdots & x_n^2(t_1) \\ x_1^2(t_2) & x_1(t_2) * x_2(t_2) & \cdots & x_2^2(t_2) & \cdots & x_n^2(t_2) \\ \vdots & & & & & \\ x_1^2(t_m) & x_1(t_m) * x_2(t_m) & \cdots & x_2^2(t_m) & \cdots & x_n^2(t_m) \end{bmatrix}$$

column of $\Theta(\mathbf{x})$ can be normalized if $\mathbf{x}$ are small.

3: Setup and run Sparse Regression (Linear Absolute Thresholding is used) to find sparse vector coefficient

$$\mathbf{X} = \Theta(\mathbf{X})\Xi$$

where, $\Xi = [\xi_\mathbf{1}, \xi_\mathbf{2}, \cdots, \xi_\mathbf{n}]$

Each column is a sparse vector that determines which components of Θ will be activated. This needs optimization in each column to find ξ_k of k^{th} row.

4: Extract Non-linear Models: a model of each row of the governing equations may be constructed as follows:

$$\mathbf{x_k} = \mathbf{f_k}(\mathbf{x_k}) = \Theta(\mathbf{x^T})\xi_\mathbf{k}$$

because each row $\mathbf{X}$ represent $[\mathbf{x_1(t)}, \mathbf{x_2(t)}, \cdots, \mathbf{x_n(t)}]$

$$\therefore \dot{\mathbf{X}} = f((\mathbf{X})) = \mathbf{\Xi^T}(\Theta(\mathbf{x^T}))^T$$

The sparse-promoting methods are more efficient than symbolic methods and have been generalized to discover partial differential equations. Rudy et al. (2017) applied the SINDy framework to discover partial differential equations by considering partial

derivatives in the PDE-FIND algorithm. This method also identifies a PDE directly from sub-sampled measurement data. SINDy relies on having measurements in a sensible coordinate system where the dynamics are sparse in the suitably chosen function basis, hence autoencoders were applied to extract parsimonious nonlinear governing equations from high-dimensional data by performing a simultaneous discovery of reduced coordinates and associated dynamical models (Champion et al. 2019).

However, SINDy applies a finite difference method to approximate the derivatives for estimating the governing equation. Finite difference methods perform poorly in the presence of noise. Furthermore, the method of numerical gradient computation introduces additional truncation errors to the system.

Another key limitation of SINDy is that it only operates with first-order time derivative models.

$$u_t = f(x, u, u_x, u_{xx}, \ldots; \Theta) \tag{2.5}$$

where, the subscripts refer to time or spatial variables, $f(\cdot)$ is unknown linear function of u, its partial derivatives $u_x, u_{xx}, \ldots$, and parameters Θ.

Furthermore, the above method only considers parabolic PDEs, i.e., PDE, that depend on first-order derivatives of the function in time and does not consider more general PDEs.

2.5.1 Neural Networks

As a consequence of Universal Approximation Theorem (UAT), a single (hidden) layer feed-forward network containing a finite number of neurons with some nonlinear activation function can approximate any continuous function u and its derivatives to an arbitrary degree of accuracy (Hornik 1989, 1991).

Theorem 1 *For any Lebesgue-integrable function $f : \mathbb{R}^n \to \mathbb{R}$ and any $\epsilon > 0$, there exists a fully-connected ReLU network $\mathcal{A}$ with width $d_m \leq n + 4$, such that the function $F_{\mathcal{A}}$ represented by this network satisfies*

$$\int_{\mathbb{R}^n} |f(x) - F_{\mathcal{A}}(x)| \, \mathrm{d}x < \epsilon$$

If we can approximate a function well, we will be able to find a good approximate of the derivative through automatic differentiation. Hence, many attempts were made to model the physical laws with a neural network. Moreover, to constrain the neural network to any given law of physics, the Physics-Informed Neural Network (PINN)s can be trained with general non-linear partial differential equations as regularizing priors (Raissi et al. 2019). For e.g., 1-D Burgers equation with Dirichlet boundary condition

$$u_t + uu_x - (0.01/\pi)u_{xx} = 0, x \in [-1, 1], t \in [0, 1],$$

$$u(0, x) = -sin(\pi x),$$

$$u(t, -1) = u(t, 1) = 0$$

can be modeled into neural network $u(x, t)$ and using the PDE as a regularizing condition

$$f := u_t + uu_x - (0.01/\pi)u_{xx}$$

The shared parameters between the neural networks u(t, x) and f(t, x) can be learned by minimizing the mean squared error loss $MSE = MSE_u + MSE_f$, where

$$MSE_u = \frac{1}{N_u} \sum_{i=1}^{n_u} |u(t_u^i, x_u^i) - u^i|^2$$

$$MSE_f = \frac{1}{N_f} \sum_{i=1}^{N_f} |f(t_f^i, x_f^i)|^2$$

In a way, they developed a method to approximate PDE with a neural network. Raissi et al. (2019) also used the technique to discover PDE from data.

Other methods, such as PDE-Net (Long et al. 2018) and PDE-Net 2.0 (Long et al. 2019), apply Convolutional Neural Networks (CNN) with filters constrained to finite-difference approximations, to learn the form of a PDE without sparsity constraints. This method was inspired by Dong et al. (2017), that we can relate filters and finite difference approximation of differential operators by examining the orders of sum rules of the filters (a fundamental concept in wavelet theory).

Berg & Nyström (2019) and Xu et al. (2019) approached a deep learning method for function but did not analyze in different noise conditions and consider mostly parabolic PDEs.

Qin et al. (2019) used residual network (ResNet), recurrent ResNet (RT-ResNet) method, and recursive ReNet (RS-ResNet) method, that allowed equation recovery without the need of time derivative data. Auto-encoders (Champion et al. 2019) can simultaneously discover coordinates and governing equations linking necessary transformation to the input data to coordinates transformation (Berg & Nyström 2019).

Hasan et al. (2019) recovered the underlying PDE from the null space of the dictionary function and include the extracted function as regularization term that forces the neural network to follow the PDE that best fits the data, which prevents the model from over-fitting.

Our method focuses on sparse optimization using Neural networks to discover governing equations not restricted to first-order temporal PDEs.

2.5.2 Differentiation Techniques

Different techniques can be employed to evaluate the gradients for sparse methods. The most common techniques for evaluating the gradients are listed as follows:

1. **Manual Computation** of PDE are commonly used for theoretical calculations by working out derivatives and coding them by hand. Evaluating a large number of gradients is both time-consuming and error-prone.

2. **Finite Difference** methods evaluate difference quotients as derivatives. They are easy to implement and generalizes to higher-order derivatives but are known to produce inaccurate numerical results due to truncation and round-off error. The simplest method uses the difference quotient formula.

$$f'(x) = \lim_{h \leftarrow 0} \frac{f(x+h) - f(x)}{h} \tag{2.6}$$

3. **Symbolic Differentiation** can be calculated precisely with symbolic processing tools, if the closed-form expressions are known. Computer Algebra Systems (CAS) like Mathematica, MATLAB, Maxima, Maple, and SymPy use symbolic methods, but they can produce inefficient code when applied to computer programs due to code bloat with common sub-expressions. We have also used a symbolic method to generate data points.

4. **Automatic Differentiation (autodiff or AD)**, also known as algorithmic differentiation, is a method for calculating derivatives of functions using chain rule on an appropriate computation graph. Unlike symbolic differentiation, which operates on math expression trees, autodiff operates on program code or the algorithm itself. There are two modes of autodiff (Baydin et al. 2018) used in practice:

   (a) **Forward-Mode AD (tangent linear mode)** starts with partial derivatives at inputs and ends by computing partial derivatives at outputs. It corresponds to a fully-right association of the chain rule of differentiation. Forward-mode AD can be implemented by overloading math operations to compute both original values and derivatives simultaneously.

   (b) **Reverse-Mode AD (Back Propagation)** starts with partial derivatives at outputs and propagate it backward and ends by computing partial derivatives at inputs. It corresponds to a fully-left association of the chain rule of differentiation. Back-Propagation computes derivatives in a two-phase process. Initially, we create a computation graph, for evaluation of the function. In the first phase, aka forward-propagation, the original evaluation function code evaluates intermediate variables/nodes and records the dependencies

in the computational graph through a bookkeeping procedure. In the second phase, derivatives are calculated by propagating adjoints of the node in reverse, from the outputs to the inputs (back-propagation).

In machine learning and computational practices, automatic differentiation is the most common differentiation algorithm because it is precise and efficient. There exist different approaches to automatic differentiation.

- Libraries or Framework in general or domain-specific languages such as Swift, Julia, Java, JS, and hundreds of other programming languages. The functionality is often restricted to specific types and APIs but is often useful for various purposes. We are using autograd, the automatic differentiation function in PyTorch for Python programming language (Paszke et al. 2017).
- Source code transformation tools are one of the oldest approaches for automatic differentiation. Tools like Tapenade (Hascoet & Pascual 2013) and ADIC/ADIFOR analyze the input code, generally written in Fortran or C, and generates output code that computes derivatives.
- Differential Programming Integrated into the programming language. Integrating a generalized gradient-calculation operator with semantics and code transformation can be incorporated as a first-class function (in augmented lambda calculus) into a functional programming language. For example, Stalin compiler for Scheme (Pearlmutter & Siskind 2008).

2.6 Hybrid Methods

PDE-Net 2.0 (Long et al. 2019) builds upon PDE-Net combining numerical approximation of differential operators by convolutions and a symbolic multi-layer neural network for model recovery. It combines both the symbolic regression method and the sparse optimization method.

Similarly, another method, known as "Equation Learner," combines a shallow neural network with symbolic regression (Sahoo et al. 2018). Many researchers are still exploring hybrid methods.

This research mainly focuses on the PINN-based Sparse method.

CHAPTER THREE: RESEARCH METHODOLOGY

3.1 Formal Problem Definition

A PDE relates a function $u : \Omega \rightarrow \mathbb{R}$, where $\Omega \in \mathbb{R}^n$ with variables $\mathbf{x} = (x_1, ..., x_n)$ and partial derivatives $D^k u(\mathbf{x}) := \frac{\partial u}{\partial x_1}, ..., \frac{\partial^2 u}{\partial x_1 \partial x_n}, ..., \frac{\partial^k u}{\partial x_n^k}$ where k is the order of PDE. Here, x represent any variable: spatial variable such as 'x','y','z' or temporal variable 't'. Hence, the PDE has a general form (Hasan et al. (2019))

$$f(\mathbf{x}; u(\mathbf{x}); D^k u(\mathbf{x})) = 0 \tag{3.1}$$

where, $f(\cdot)$ is a function of $\mathbf{x}$, u, and the partial derivatives $D^k u(\mathbf{x})$.

An example of PDE is wave equation which is a second order hyperbolic PDE represented as

$$\frac{\partial^2 u}{\partial t^2} - c^2 \frac{\partial^2 u}{\partial x^2} = 0$$

where, c is the velocity of wave propagation. The equation can be written as

$$u_{tt} - u_{xx} = 0, \text{ where } c = 1$$

To search PDE for a system, the space of all possible terms are infinitely large, hence we select a list of L candidate terms $C = (c_1, ..., c_L)$, where candidate terms are functions of variables x_i, dependent variable $u(x_i)$, and partial differential terms $D^k u(\mathbf{x})$. For our example of wave equation, we can choose $[u_{tt}, u_{xx}, u_t, u_x, u, u * u_x]$ as list of candidate terms.

Then, we construct PDE as the weighted linear combination of these candidate terms, which is of the following form.

$$\sum_{i=1}^{L} \theta_i^* c_i(\mathbf{x}; u(\mathbf{x}); D^k u(\mathbf{x})) = 0 \tag{3.2}$$

where, $\Theta^* = (\theta_1^*, .., \theta_L^*)$ is unknown vector of coefficients for the linear combination of candidate terms. We want to determined this coefficient vector Θ. For wave equation with list of candidates $[u_{tt}, u_{xx}, u_t, u_x, u, u * u_x]$, Θ^* should be close to $(1, -1, 0, 0, 0, 0)$.

3.2 Research Design

We conducted a numerical investigation with a positivist stance.

3.2.1 Design Factors

The response variable is the equation prediction capability of our model, i.e., the average error on the coefficient of the parameter Θ. The metric we used to measure this "distance" of the predicted coefficient vector and the actual coefficient vector is cosine similarity rather than commonly used L_2 norm. The error in the prediction of law is given by Eq. 3.11.

There are many controllable and uncontrollable factors for this experiment. The design factors, that we varied in different experiments are listed as follows:

Table 3.1: Table showing factors of experiment, their level and range

Factor	**Levels**	**Degree of Freedom**
Equation Types	[Wave, Inviscid Burgers, Helmholtz]	2
Number of Data points	[1000, 5000, 10000]	2
Number of Collocation Points	[100,1000]	2
Number of layers in NN	[2,5,7]	2
Number of neurons in each layer	[10,20,50,70]	3
Activation Function used in NN	[SoftPlus, ELU, LeakyReLU]	2
Learning rate of model	0.01	-
Learning rate of coefficient	0.005	-
Number of Training Epochs	[10000,50000]	1
Minibatch Size	50	-
Theta initialization	[500,1000,2000]	-

Since there are many factors involved, performing several experiments would be time-consuming, and the analysis more complicated. Hence, separate tests were performed, with certain factors held-constant. Learning rate for both model and coefficients were kept constant. Furthermore, the coefficient Θ was initialized and re-initialized at all three levels 500,1000 and 2000.

Furthermore, there are some other factors not listed in the above tables such as initialization of model parameters (weights of the neural network and initial value of Θ), selection of data points, and other random numbers are allowed-to-vary factors, we rely on the randomization to balance out their effects.

3.2.2 Experiment Design

The experimental design is based on Montgomery (2017). The basic principle of experimental designs considered for the experiments is as follows.

Randomization

The configuration selection and the order of individual runs are random so that observations (or errors) are independently distributed random variables. Random data generation and random initialization of the model parameters introduces further randomization. Seeding random number generation allows us to generate deterministic results for the demonstrations.

Replication

Replication is required for any experiment to show statistical significance. Multiple repetitions allow estimation of the experimental error and average to the actual response for one of the factor levels. The sample size (number of replicates) of 3 was selected. Seeding random number generation will enable us to maintain repeatability and to generate deterministic results for the demonstrations.

Blocking

When performing experiments, certain factors known to increase variability are "blocked" into groups. Blocking improves the precision of comparisons of the factors under investigation. During analysis, certain factors were blocked, such as the type of equation being used.

Factorial

Since a large number of factors are involved in this experiment, a factorial design was implemented. For each trial in individual tests, all possible combinations of the selected levels of the factors were selected. The ranges of levels are chosen to minimize the curse of dimensionality and yet cover a wide range. The full-factorial design was used rather than a fractional factorial design for each experiment.

Ablation Study

Additionally, an ablation study (Meyes et al. 2019) was performed to study the interaction of loss components, understand the effect of loss components, and their interactions. The combination of loss function were performed separately and analyzed as show in section 4.3.

3.2.3 Training Data

Symbolic Data Generation

Known solutions of the PDE, shown in Table 3.2, are converted to symbolic functions using SymPy (Meurer et al. 2017). These symbolic functions generate data points throughout a grid specified by the domain listed in Table 3.2. We also extract a few additional points to evaluate gradients, termed as collocation points.

Data Preprocessing

Both training data, as well as collocation data points, are shuffled and sampled randomly. Furthermore, for validation of the neural network, 20% of the training data is split into a validation set. We use this as a validation set to test if the model is over-fitting.

3.2.4 Research Instruments

The instruments for research, which include the software, libraries, and hardware used to conduct the study, are listed in Appendix A. Most of the research was performed on Google Colab.

For analysis and design of experiments, SAS JMP (SAS Institute Inc., Cary, NC 1989-2019) and Google spreadsheets were used.

3.3 Data Generation

3.3.1 Partial Differential Equation Selection

Partial Differential Equation (PDE) is widely used in science, engineering domains. Common engineering examples include wave propagation, fluid flow, vibration, mechanics of solid, heat flow, diffusion, electric field and potential, electromagnetism, and quantum mechanics.

In this research, we are only covering second order PDE, which is generally described by

$$A\frac{\partial^2 u}{\partial t^2} + 2B\frac{\partial^2 u}{\partial x \partial t} + C\frac{\partial^2 u}{\partial x^2} = D(x, t, u, \frac{\partial u}{\partial t} + \frac{\partial u}{\partial x}) \tag{3.3}$$

We usually come across three types of second-order PDEs in engineering, viz. elliptic, hyperbolic, and parabolic PDEs. Each type of PDE has distinct characteristics that provide insights into the smoothness of the solution, and the effect of initial and boundary conditions. It is valuable to determine if a particular finite element approach is appropriate to the problem.

1. **Elliptic PDE**:

$$B^2 - AC < 0$$

for example, Laplace equation:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

Other equations like steady heat transfer, flow and diffusion, equation of elasticity without inertial terms.

Elliptic PDE is typically characterized by steady-state systems (i.e., no time derivatives), within a closed boundary condition.

The solutions of elliptic PDEs are always smooth, even if the boundary conditions are rough.

For our experiment, we have selected the Helmholtz equation, which generally arises in physics problems such as electromagnetic radiation, seismology, and acoustics.

2. **Parabolic PDE**:

$$B^2 - AC = 0$$

for example, Heat-conduction or Diffusion equation:

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

Other equations are time dependent like Transient heat transfer (conduction), flow and diffusion.

Parabolic PDE varies in both space and time, but include only the first derivative in time. We solve a parabolic PDE with both initial and boundary conditions.

The solutions are smooth in space but may possess singularities; hence estimation of gradients can become an issue. However, the flow of information in a parabolic system is infinite.

Initial test with parabolic PDE, resulted in very poor results; hence, parabolic PDE were omitted from the scope of this study and left for future study.

3. **Hyperbolic PDE**:

$$B^2 - AC > 0$$

for example, wave propagation equation:

$$\frac{\partial^2 u}{\partial x^2} = \frac{1}{c^2}\frac{\partial^2 u}{\partial t^2}$$

Many of the equations of mechanics are hyperbolic in nature.

Hyperbolic PDE varies across both space and time too, but with a second derivative in time. We solve it similarly with both initial and boundary conditions.

The smoothness of the solution depends on the smoothness of the initial and boundary conditions. A sudden and significant change in the initial or boundary condition often results in the development of a shock in the solution. Information travels at finite wave speed along with the wave.

Two types of Hyperbolic PDE are selected, as shown in Table 3.2.

We selected the following PDE and boundary conditions with the given known solution to test our model.

Table 3.2: Table Showing solution of PDE used to generate training data

PDE	**PDE Type**	**Domain**	**Known Solution**
Wave Equation: $u_{tt} - c^2 u_{xx} = 0$ where c=1	Hyperbolic	$x, t \in [-3, 3]$	$u = \frac{1}{2} * (e^{-(-t+x)^2} + e^{-(t+x)^2})$
Invisicd Burgers: $u_t + u * u_x = 0$	Quasi-linear Hyperbolic	$x, t \in [0, 1]$	$u = \frac{1+2x(t+1)+(1+4x(t+1))^{\frac{1}{2}}}{2(t+1)^2}$
Helmholtz: $u_{xx}+u_{yy}+k^2 u = 0$ where k=1	Elliptic	$x, y \in [-\pi, \pi]$	$u = \sin x * y - \sin y * x$

The above equations are used to generate and sample data points which are depicted in the following figures.

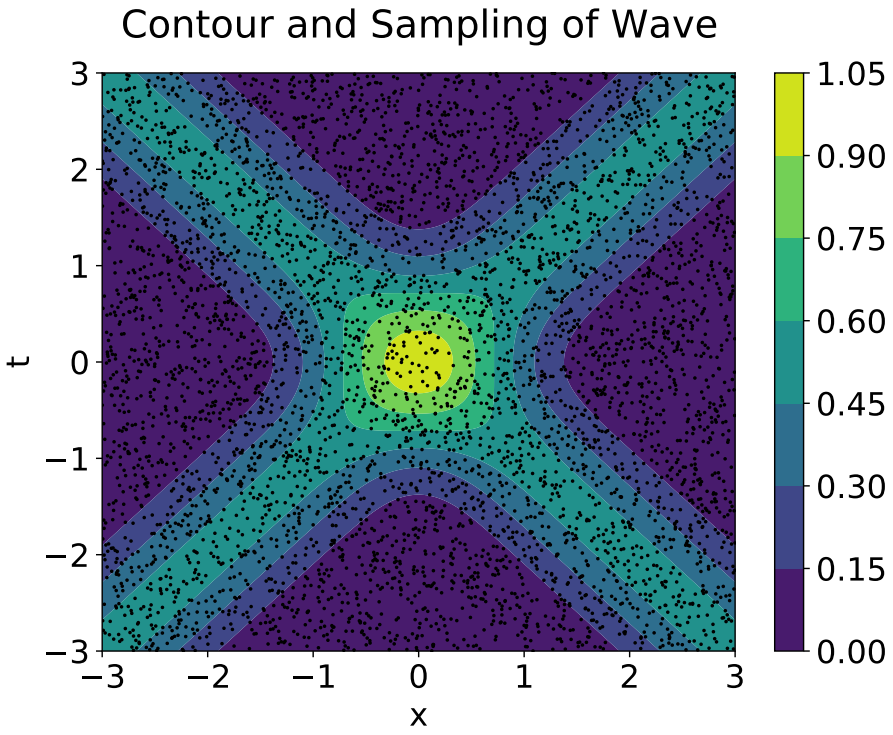


Figure 3.1: Sampled points (black dots) from contour of Wave Equation

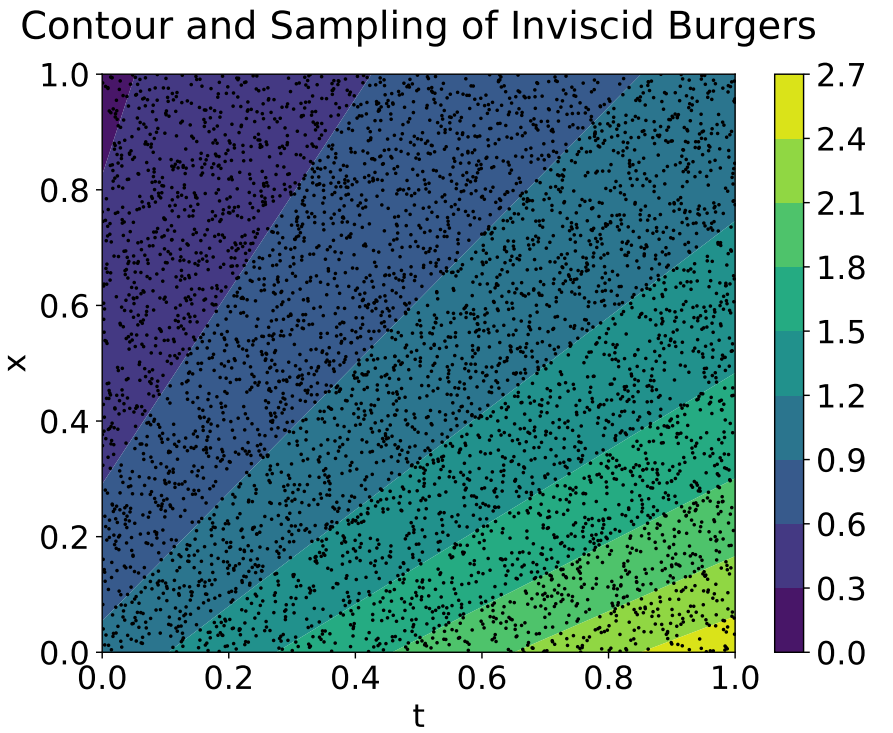


Figure 3.2: Sampled points (black dots) from contour of Inviscid Burgers Equation

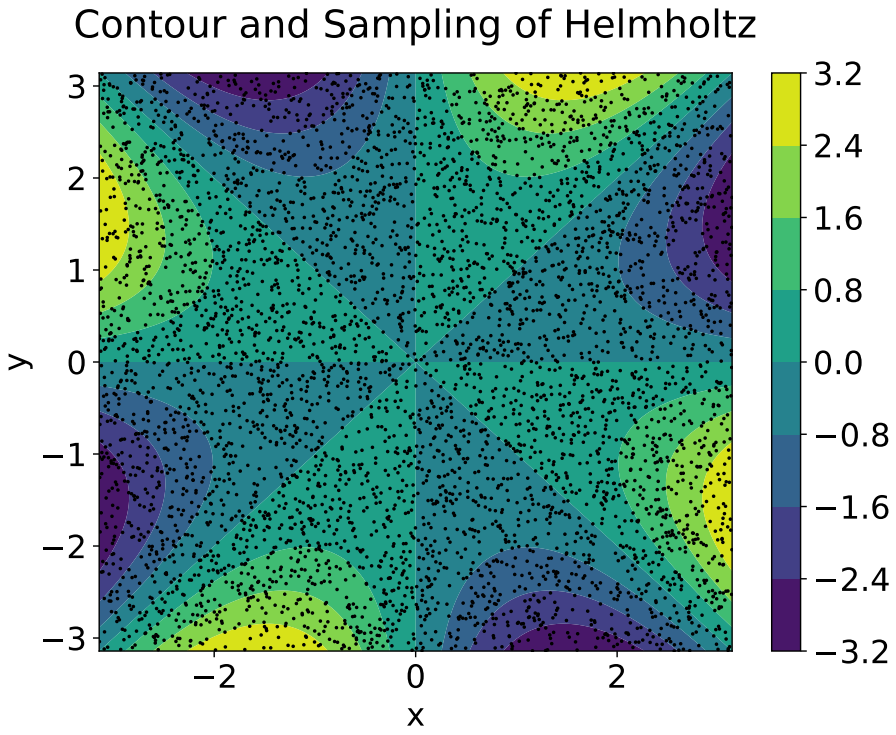


Figure 3.3: Sampled points (black dots) from contour of Helmholtz Equation

3.4 Model Architecture

A simple feed-forward full-connected multi-layer perceptron (MLP) is used. The number of layers, the number of hidden units, and the type of activation function were factors of design.

A baseline MLP with four hidden-layers, each consisting of 50 hidden units, similar to Berg & Nyström (2019), is used. It used the softplus activation function similar in all layers except the output layer, which is just a linear unit.

Softplus (Dugas et al. 2001) is a smoother version of ReLU which has smoothing and non-zero gradients properties. It improves performance with faster convergence than ReLU.

$$\text{softplus} = log(1 + exp(x))$$

ELU (Clevert et al. 2015) is another smoother version of ReLU which has smoother output till $-\alpha$.

$$elu(x) = \begin{cases} \alpha exp(x) & if X \leq 0 \\ x & if x > 0 \end{cases}$$

ELU tends to achieve zero mean activation and may lead to faster convergence.

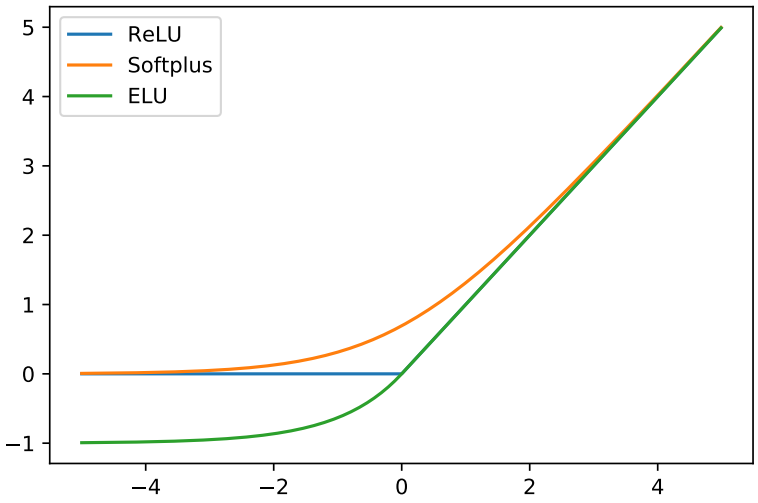


Figure 3.4: Activation Functions: ReLU, ELU and Softplus

3.5 Model Fitting

The choice of error metric is very crucial for training any model. The most common choices of error metrics used for training neural networks are as follows:

1. Mean Absolute Error (MAE) uses absolute difference as the error metric, treating all errors the same.

$$MAE = \frac{1}{N}\sum_{i=1}^{N}\|u_i - \hat{u}_i\|$$

2. Mean Square Error (MSE) uses squared error to penalize larger prediction errors.

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(u_i(x) - \hat{u}_i)^2$$

3. Root Mean Square Error (RMSE) uses squared error to penalize larger prediction errors but returns results in the same unit as the target u. Since, the error magnitude is quite small, RMSE

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(u_i(x) - \hat{u}_i)^2}$$

The first error metric, $\mathcal{L}_u$, measures how precise is the prediction of neural network $\hat{u}_i$ compared to the actual target value provided in the training data (X_i, u_i).

For the baseline, we used root mean square error (RMSE) for training the neural network. The target variable is smaller than zero, which leads to a very small residual.

Using RMSE over MSE prevents the error from being very small. Furthermore, it has the same units as the output variable and can give a quick indication of how the model performs.

$$\mathcal{L}_u(\mathbf{x}; W) = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(u_i(x) - \hat{u}_i(\mathbf{x}; W))^2} \tag{3.4}$$

, where W are neural network parameters. Training the neural network with L_u only can lead to a certain degree of performance in prediction. This network can make prediction u for new data and can generate a rough estimation of the derivatives.

3.5.1 Physics-based Regularization and Model Discovery

A general method applied to verify is the given solution is a true solution of the PDE is to substitute the function into the PDE. In our case, the correct PDE solution $\hat{u}^*$, given correct $\Theta*$, should satisfy PDE, as shown in Eq. (3.2) for every point $x \in \Omega$. This technique was used as regularization prior, to guide neural network with existing theory in PINNs (Raissi et al. 2019).

We know, neural networks can approximate gradients to arbitrary accuracy under mild assumptions on the activation function (Hornik 1991). With a better-generalized model, we can estimate differentials through auto-differentiation (auto-diff) with lower truncation errors than numerical differentiation for higher-order differentials. We can use this method to estimate gradient terms in the candidate list at any point.

Collocation points are generated symbolically and randomly sampled. In the baseline experiments, 1000 collocation points were sampled from within the prescribed domain. L terms in the candidate list is evaluated over M collocation point through autodiff, to obtain matrix $H \in \mathbb{R}^{M\times L}$.

$$H(\mathbf{x}, \hat{u}) = \begin{bmatrix} | & | & | \\ C_1(\mathbf{x}, \hat{u}_i, D^k\hat{u}_i) & \dots & C_L(\mathbf{x}, \hat{u}_i, D^k\hat{u}_i) \\ | & | & | \end{bmatrix} \tag{3.5}$$

For brevity $\hat{u}_i(\mathbf{x}; W)$ has been written as $\hat{u}_i$, where i = 1..M collocation point. PDE in Eq. (3.2) transforms into a linear system of equations, that tests if u satisfy the PDE defined by Θ and candidate list.

$$\begin{bmatrix} | & | & | \\ C_1(\mathbf{x}, \hat{u}_i, D^k\hat{u}_i) & \dots & C_L(\mathbf{x}, \hat{u}_i, D^k\hat{u}_i) \\ | & | & | \end{bmatrix} \begin{bmatrix} \theta_1^* \\ \vdots \\ \theta_L^* \end{bmatrix} = \vec{0} \tag{3.6}$$

This homogeneous form of linear equation can also be written as:

$$H(x, \hat{u}) \cdot \Theta^* = 0 \tag{3.7}$$

To determine Θ^*, we solve the above linear system of equation. We know, a homogeneous equation of form $Ax = 0$, has a trivial minimum-norm solution, $\vec{0}$. The solution vectors are in the null space of A (Strang 2016). The solutions are singular vectors associated with singular value 0. The least-squares solution is modified by imposing the constraint $\|x\|_2 = 1$. This becomes a constrained optimization problem

$$\min_{\|x\|_2=1} \|Ax\|$$

.

The minimum of $\|Ax\|$ subjected to $\|x\| = 1$, is also the smallest singular value of H. This is also cited as min-max theorem for singular values (Loan & Golub 1996).

Equivalently, the null space of M consists of all the possible solutions Θ.

$$Null(H) = \{\Theta^* \in \mathbb{R}^L \mid H\Theta^* = \vec{0}\}$$

Θ^* is singular vector of H associated with the singular value 0. Using Singular Value Decomposition (SVD), we attempt to find the Θ vectors associated with singular value 0 with above constraint. This constraint is achieved through normalization of Θ vector.

Since, lowest singular values are not 0, there are some residual in the function evaluation of $H\Theta$. For the baseline, we used MSE to calculate the residual

$$\mathcal{L}_f(\mathbf{x}; \hat{u}; \Theta) = \frac{1}{M}\|H\Theta\|_2^2, \text{ with } \|\Theta\|_2 = 1 \tag{3.8}$$

L_f acts as a regularizer to the neural network. In the words of Theory Guided Data Science (TGDS), it constraints the search space to regions that are defined by the PDE. In our case, the constraints improves as the network gets trained over time. In doing so we also estimate the coefficient vector $Theta$ and hence the PDE.

3.5.2 Parsimony

Useful scientific theories are often interpretable, and parsimonious, i.e., the governing equation consists of only a few terms. Equations with multiple terms may be more accurate, but they are likely to overfit the data, whereas others may be more parsimonious but oversimplify the system; the right balance of parsimony is difficult to specify in advance.

Hence, we constrain the Θ vector such that it tends to force some coefficients to zero, similar to shrinkage and selection operation in regression (Tibshirani 1996). We impose L_1 sparsity constraint on the Θ to get a sparse equation from a relatively large space of

potential candidate terms.

$$\mathcal{L}_{norm} = \|\Theta\|_1 \tag{3.9}$$

3.5.3 Combined Regularization and loss

The interaction of L_u, L_f and L_{norm} helps in training the neural network effectively, and support PDE identification.

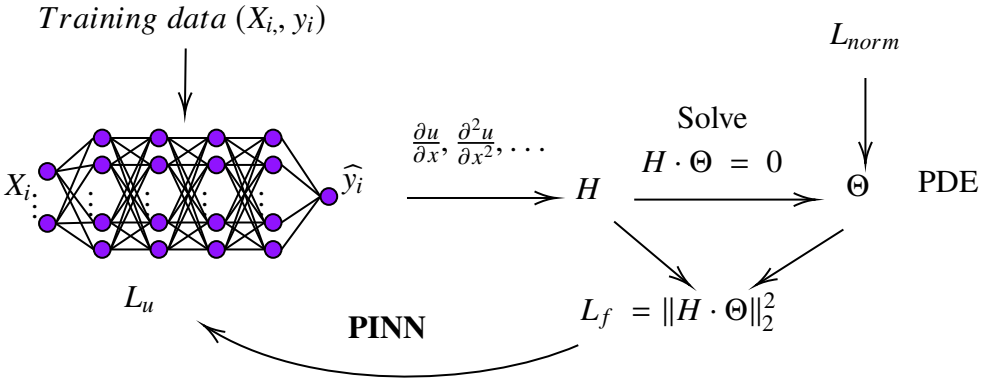


Figure 3.5: Training the model architecture

As shown in fig.3.5, the training data trains the neural network to be able to estimate gradients accurately. The gradient estimation on collocation points allows us to determine Θ by solving a homogeneous equation. The residual of this equation, L_f, can be used as an additional regularizer for training the neural network. L_{norm} forces sparsity on Θ, which results in selection of more significant coefficients. The accuracy of H and Θ is visible in L_f, since L_f represents the residual of PDE identification. This residual further acts as a physical constraint as discussed in Raissi et al. (2019), and helps in further training the neural network.

Ablation study of the total loss function is performed with various combination of $\mathcal{L}_u$, $\mathcal{L}_f$ and $\mathcal{L}_{norm}$

$\mathcal{L}_u$ can be further used as an additional regularizer when multiplied with the other regularization term Hasan et al. (2019). Here, λ_u, λ_f and λ_{norm} are coefficients for each loss function, used as additional hyper-parameters to the network. We estimated W and Θ with the minimization of $\mathcal{L}$.

$$\mathcal{L} = \lambda_u \mathcal{L}_u (1 + \lambda_f \mathcal{L}_f + \lambda_{norm} \mathcal{L}_{norm}) \tag{3.10}$$

3.5.4 Algorithm

The algorithm for training the neural network for PDE identification is depicted as follows:

Algorithm 2 Algorithm for Data Driven Discovery of Governing Equation

Initialize the Neural Network (NN) and parameter Θ
for each epoch **do**
    Predict $\hat{u}_i = NN(X_i)$
    Evaluate $\mathcal{L}_u$
    Evaluate (H) with autodiff on $\hat{u}$ on collocation points
    Solve $H \cdot \Theta = 0$ using SVD for Θ constrained by $\|\Theta\| = 1$
    Calculate $\mathcal{L}_f$ and $\mathcal{L}_{norm}$
    Apply appropriate loss combination method, such as

$$\mathcal{L} = L_u * (1 + L_f + L_{norm})$$

    Back-propagate $\mathcal{L}$ and update parameters
end for
Solve $H \cdot \Theta = 0$ using SVD and return Θ

3.6 Verification and Validation

Since our experiment is numerical computation, verification and validation of the solution are vital for credibility. Verification is "the process of assessing software correctness and numerical accuracy of the solution to a given mathematical model." At the same time, validation is "the process of assessing the physical accuracy of a mathematical model based on comparisons between computational results and experimental data" Oberkampf & Roy (2010). In other words, verification checks the simulation result with the conceptual model, while validation checks it with the real world.

Verification

The first step to ensure that the method is working is to confirm that the gradients evaluated by the neural networks are correct. Neural network models trained using back-propagation method often encounter vanishing and exploding gradient problems, which are very common in Recurrent Neural Networks (RNN).

Since we already have the equation generating the data, we can manually or symbolically differentiate it to determine the exact derivative functions. We can compare the gradients predicted by the neural network for each collocation points with the exact derivative and verify that they are correct.

We checked the loss values $\mathcal{L}_u$ and $\mathcal{L}_f$ to verify that the network is well trained. Both of these values should decrease to a small value but not reach zero. We can also verify that a significant decrease in $\mathcal{L}_f$ will constrain the neural network further, and it is likely to reduce $\mathcal{L}_u$ subsequently.

Validation

Cross-validation methods are quite popular in machine learning to validate the model. It allows us to test the model for a limited set of unseen data as well. Hence, a part of the dataset is held out as a validation set (and sometimes test set) to test the model. We used a train-test split of 20%.

Since we also have the real equation, we can check that our model is learning by comparing the predicted equation with the real equation. Cosine similarity can compare the similarity between two vectors. We calculate the error in the identification of the governing equation by using the square root of 1 - cosine similarity, we can call it Θ error or error in law.

$$\Theta \text{ error} = \sqrt{1 - \frac{x_1 \cdot x_2}{||x_1|| * ||x_2||}} \tag{3.11}$$

"No Free Lunch Theorem" (Wolpert & Macready 1997) states that any two search and optimization algorithms "are equivalent when their performance is averaged across all possible problems"; i.e., the computational cost of finding a solution, averaged over all possible problems, is the same for any solution method. In another way, a method that shows excellent performance for one particular problem might not perform well in another problem. Hence, we tested the method with three different forms of partial differential equations: Wave equation as a form of hyperbolic PDE, Helmholtz equation as a form of elliptic PDE, and Inviscid Burgers equation which is a form of first order quasi-linear hyperbolic equation was used.

Furthermore, we can compare the results of prediction accuracy with other existing research to evaluate the model's performance.

CHAPTER FOUR: RESULTS AND DISCUSSIONS

4.1 Baseline Training Results

A baseline model was developed and trained for the three types of PDE selected in section 3.3.1 with the following parameters:

- Training data points: 10000,
- Collocation points: 1000,
- Optimizer: Adam,
- Learning rate of model: 0.02,
- Learning rate of Θ : 0.002,
- Total training epochs : 50001,
- Mini-batch is not used,
- Θ initialization at 500 epoch
- MLP with 4 layer, 50 units/layer with Softplus activation is used unless specified.
- Loss weight: λ_u = 1, λ_f = 10, λ_{norm} = 1e-3,
- Loss Metric: $\mathcal{L}_u$: RMSE, $\mathcal{L}_f$: MSE
- Loss Model :

$$\mathcal{L} = \mathcal{L}_u * (1 + \mathcal{L}_f + \mathcal{L}_{norm})$$

The coefficient vector, Θ, is also a parameter being tuned as the model steps through the gradients to minimize the loss.

4.1.1 Wave Equation

The training data generated from wave equation, $u_{tt} - u_{xx} = 0$, was fed to baseline neural architecture. The error metrics during the training are plotted on the fig. 4.1.

One key and expected finding are visible here. The loss $\mathcal{L}_u$ decreases rapidly after the initialization of Θ with the gradients obtained from initially trained neural networks. $\mathcal{L}_f$ and $\mathcal{L}_u$ keeps on decreasing slowly even after 20,000 epochs. $\mathcal{L}_{norm}$ converges quickly and maintain its value.

The validation curve (dotted lines) also follows the training curve $\mathcal{L}_u$ closely, indicating good fit of the model.

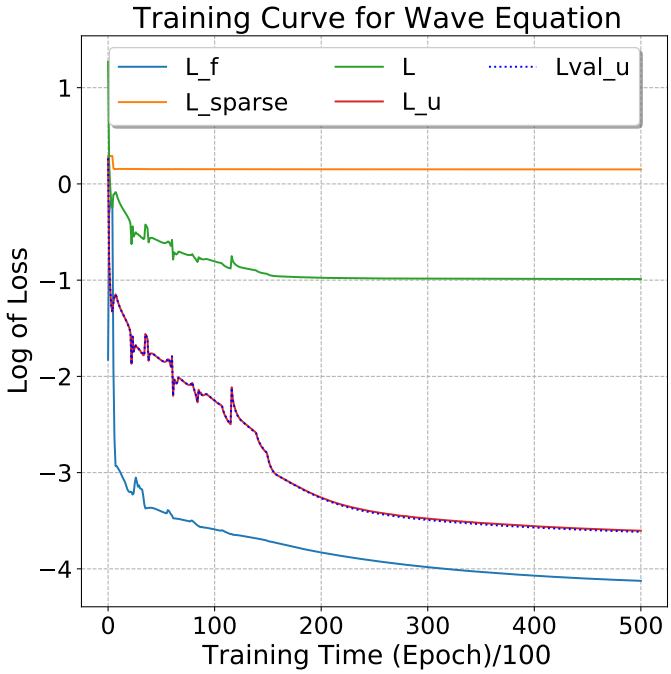


Figure 4.1: Training Curve for Wave Equation

4.1.2 Inviscid Burgers' Equation

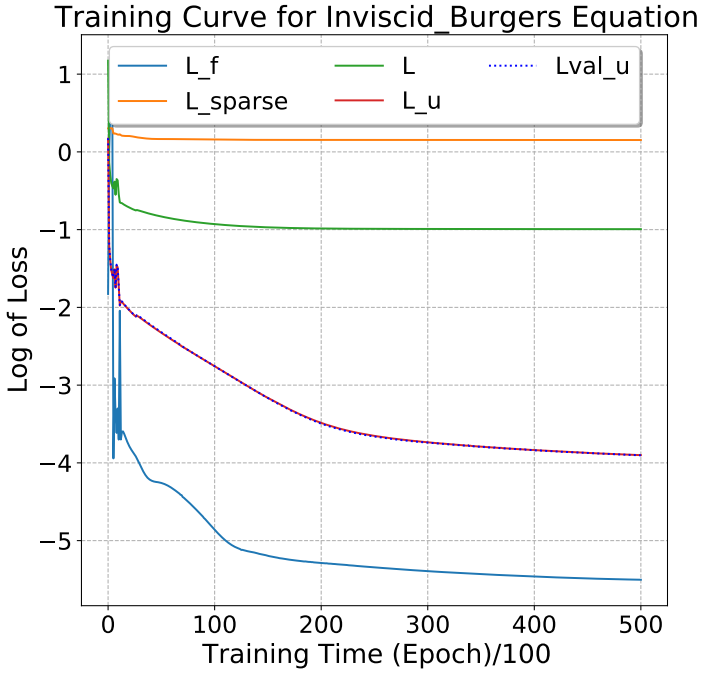


Figure 4.2: Training Curve for Inviscid Burgers Equation

The training data generated from Invisicd Burgers $u_t + u * u_x = 0$ was fed to a similar neural architecture. The error metrics during the training were calculated and plotted on the figure 4.2.

This dataset was more straightforward to train, and the same parameters resulted in a smoother learning curve. After the initialization of Θ, we can observe a drop in the loss metric $\mathcal{L}_u$. $\mathcal{L}_f$ and $\mathcal{L}_u$ keeps on decreasing slowly even after 20,000 epochs, similar to the training wave equation.

4.1.3 Helmholtz Equation

The training data generated from Helmholtz $u_{xx} + u_{yy} + k^2 u = 0$, where k=1, was fed to the neural architecture defined in the methodology chapter. Fig. 4.3 shows, error metrics throughout the training of the model.

The loss $\mathcal{L}_u$ drop after initialization of Θ was observed again. $\mathcal{L}_f$ and $\mathcal{L}_u$ converges after 20,000 epochs.

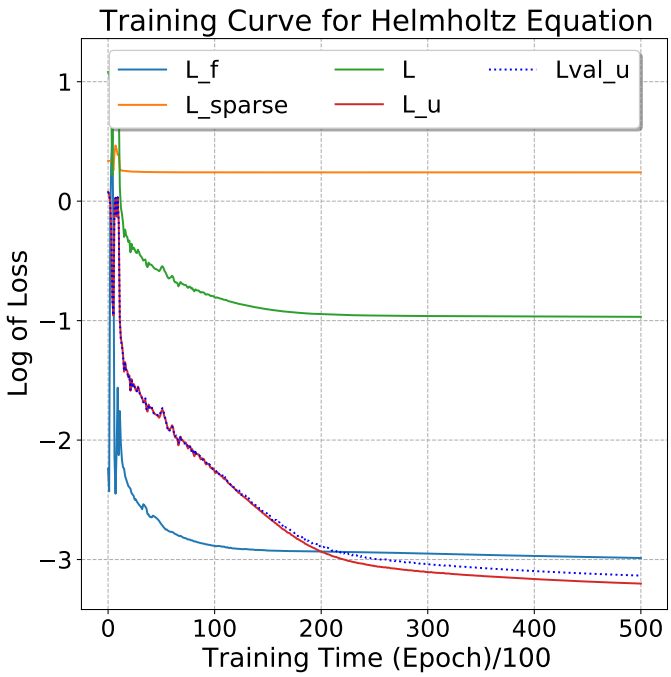


Figure 4.3: Training Curve for Helmholtz Equation

Fig. 4.1, 4.2 and 4.3, all shows similar trend in training. The $\mathcal{L}_u$ decreases below 10^{-3} and converges. The random zigzag pattern indicates that the learning rate and other hyper-parameters might not be optimal. Hence, different equations are likely to have different hyper-parameters, but for comparison, we have used the same hyper-parameters for all experiments.

The function residual $\mathcal{L}_f$, for both wave and Inviscid Burgers equation, the residual is far less compared to Helmholtz. Low residuals indicate correct identification of the coefficient vector or a local minima with low residuals from a different set of coefficient vectors.

4.1.4 Prediction of Θ and PDE Recovery

Sqrt of cosine between the two vectors, as defined in section 3.6, estimates the deviation of predicted PDE with the actual coefficient, called as Θ error or error in law. The fig. 4.4 shows the estimation of Θ error throughout the training period.

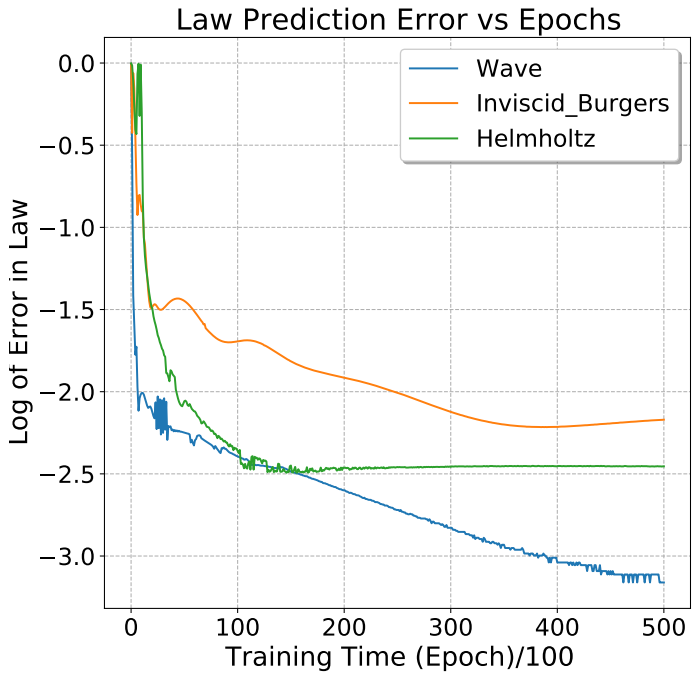


Figure 4.4: Plot of Log of Error in law throughout one particular training session

It can be observed from the figure that wave equations are recovered more accurately than inviscid Burgers and Helmholtz equations. Furthermore, the Helmholtz equation (a form of elliptic PDE) tends to converge earlier than other equations that have temporal components.

The above experiments were repeated five times too, and the statistics are listed in the table 4.1

All the values are in order of 10^{-3} or less, which indicates that the coefficients of the vector are very close to the correct vector. These values are similar to Hasan et al. (2019) as it uses similar metrics and loss functions. This model performs also performs well in comparison to Rudy et al. (2017) for the discovery of PDE, which uses Euclidean distance for measuring error in the recovery of the governing equation.

Table 4.1: Table shows average and standard deviation of error in estimating the governing equation

PDE	Candidate List	Θ Error
Wave Equation $u_{tt} - u_{xx} = 0$ $x, t \in [-3, 3]$	$u_{tt}, u_{xx}, u_t, u_x\ u, u^2, uu_x$	$(1.42 \pm 0.024) \times 10^{-3}$
Invisicd Burgers $u_t + u * u_x = 0$ $x, t \in [0, 1]$	$u_{tt}, u_{xx}, u_t, u_x,$ u, u^2, uu_x, u_x^2	$(6.43 \pm 0.13) \times 10^{-3}$
Helmholtz $u_{xx} + u_{yy} + u = 0$ $x, y \in [-\pi, \pi]$	$u_{xx}, u_{yy}, u_{yx},$ $u_y, u_x, u, u^2, uu_x, uu_y$	$(3.51 \pm 0.12) \times 10^{-3}$

4.2 Effect of Neural Architecture

4.2.1 Effect of Layers and Neurons

The number of neurons for each layer and number of layers in fully connected networks is often selected based on the feature space of the problem. Similarly, the number of hidden neuron depends mostly on the number of inputs, outputs, activation function, overall architecture, and dataset properties. Many empirical rules have been devised, yet certain hit and trial method are still used.

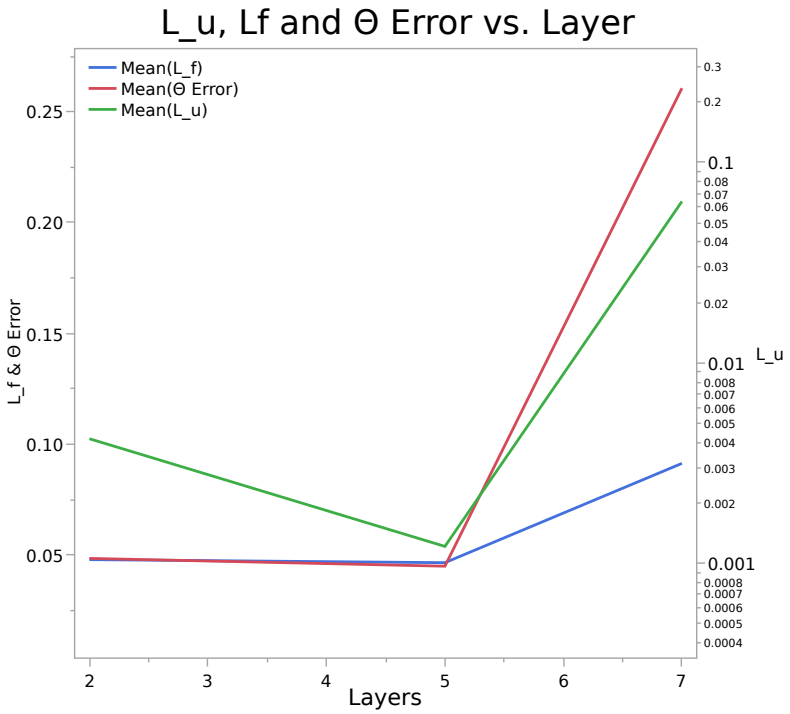


Figure 4.5: Effect of Number of Layers

The fig. 4.5 shows that 5 layers as optimal number of layers. More number of layers might overfit the model and lead to incorrect gradient evaluation.

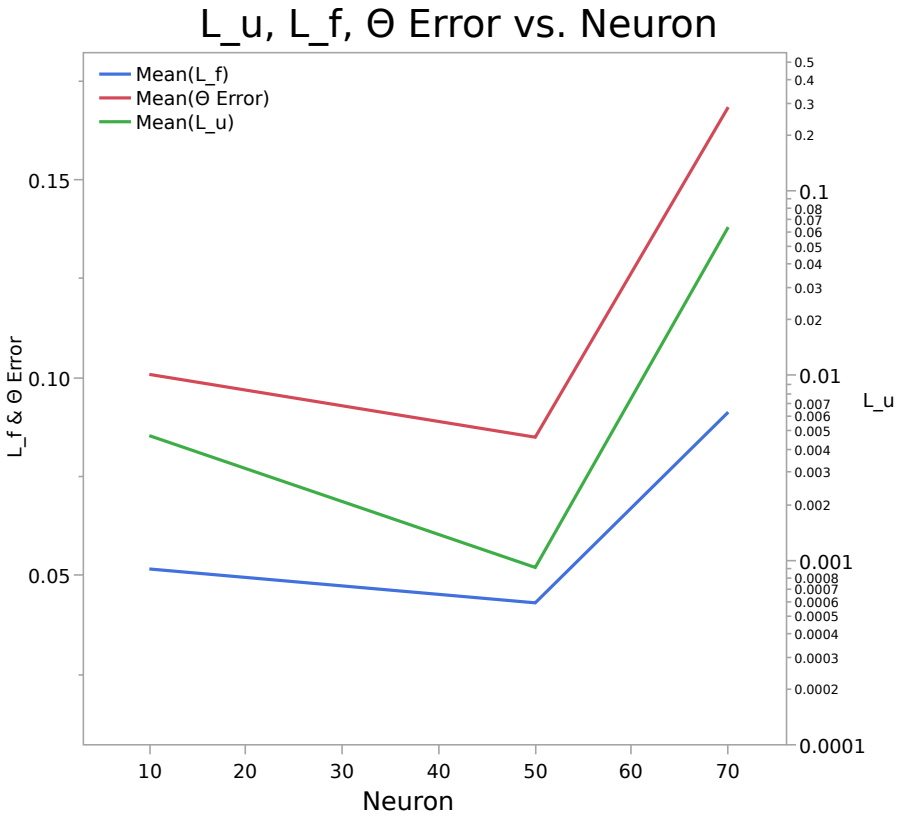


Figure 4.6: Effect of Number of Neurons

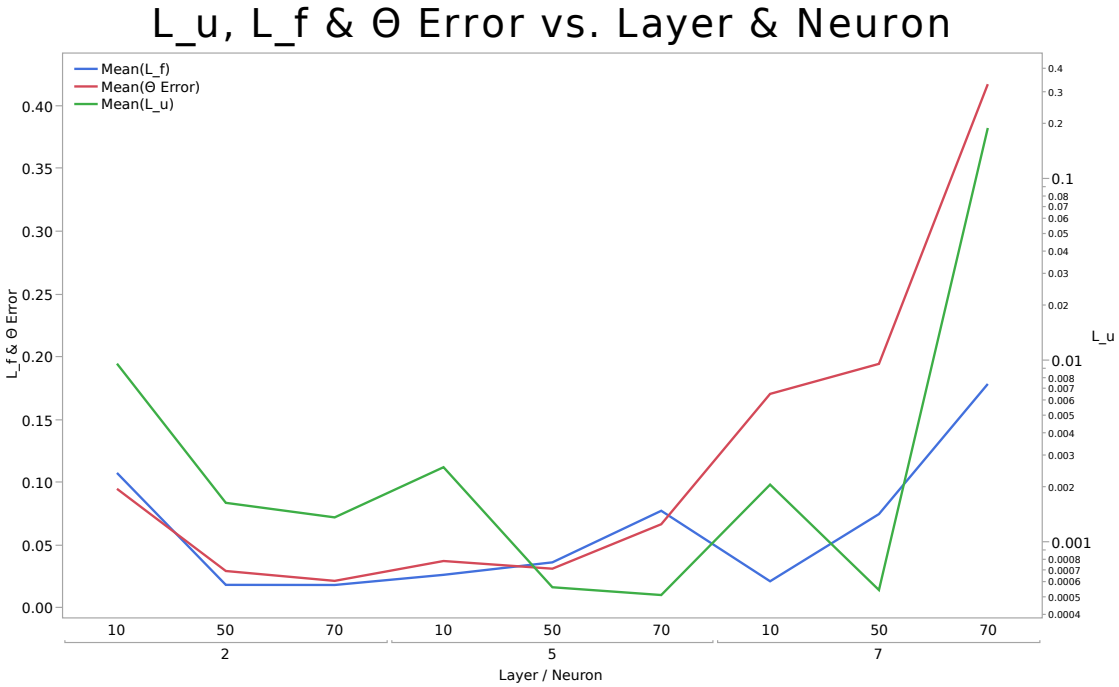


Figure 4.7: Effect of Layer and Neurons together

The number of neurons required as shown in fig. 4.6 is directly related to the number

of layers as shown in fig. 4.7. The optimum combination is 5 layers with 50 neurons as all the loss are minimum. This also shows that even with 2 hidden layers and higher number of neurons, the model can perform decently to identify the model.

4.2.2 Effect of Activation

Activation function has a huge impact on the prediction of gradients and hence identification of the governing equation. The performance of softplus and ELU was nearly equivalently in aggregation. Different activation function have different performance differently based on the type of PDE as shown by the fig. 4.8. ELU performed better in Inviscid Burgers equation, while softplus performed better in Wave equation. It was indecisive in the case of Helmholtz equation.

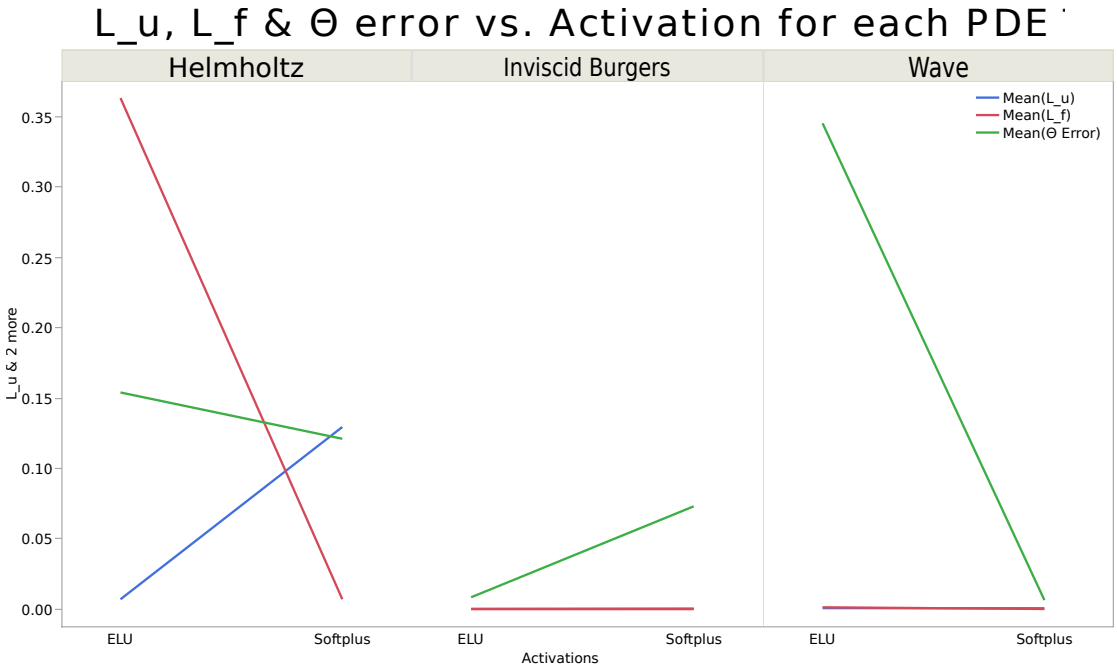


Figure 4.8: Effect of Activation for each Equation

The need for different activation function for different type of PDE might suggest, that a particular type of PDE has affinity towards a particular type of PDE. Further experiments with many PDE of similar types can be conducted to test this. This also shows that there is a need of a different type of activation function.

4.3 Effect of Loss Metric and Models

4.3.1 Effect of Loss Metric

The three common type of loss metric used in machine learning as discussed in 3.5, viz MAE, MSE, and RMSE were used for $\mathcal{L}_u$ and $\mathcal{L}_f$ in different combination as shown in the fig. 4.9.

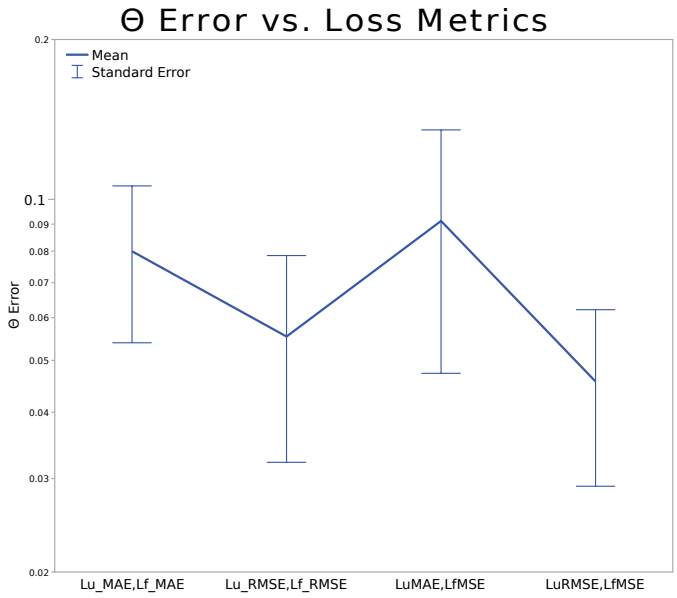


Figure 4.9: Effect of Loss Metric choice on Θ Error, The error bar is constructed using 1 standard error from the mean.

It turns out that when $\mathcal{L}_u$ is using RMSE as metric, the prediction performance improves. Since, the target value in our data sets are small, the error are smaller. The effect of range of error between the different losses changes impacting their performance.

4.3.2 Effect of Loss Model

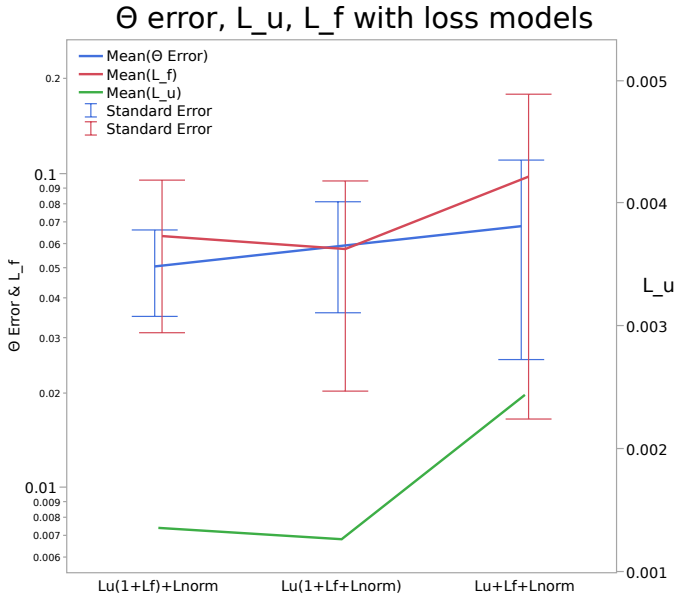


Figure 4.10: Effect of loss model on the Loss metric and Theta Error

Not only the choice of metric but also the combination of the loss functions have impact on the model. Hence, three version of the loss combination was studied to test the

interaction effect of the loss function as shown in fig. 4.10. All of the combinations were able to predict the models with certain variability.

The effect of multiplying $\mathcal{L}_f$ with $\mathcal{L}_u$ is visible in $\mathcal{L} = \mathcal{L}_u(1 + \mathcal{L}_f + \mathcal{L}_{norm})$ as well as in $\mathcal{L} = \mathcal{L}_u(1 + \mathcal{L}_f) + \mathcal{L}_{norm}$, with negligible effect on $\mathcal{L}_{norm}$. $\mathcal{L}_u$ forces the model to fit the data, while $\mathcal{L}_f$ constraints the model to follow the PDE that is being discovered. The product enforces an additional interaction between the two mode of learning. A higher $\mathcal{L}_u$ indicates that the model is not trained enough to give accurate gradients, hence forces the effect of $\mathcal{L}_f$ to be higher.

4.4 Verification and Validation

4.4.1 Gradient Errors

The error in estimating the gradient using automatic differentiation was calculate using the actual gradient estimated from symbolic gradient estimation. 3.6 Both Root Mean Square Error (RMSE) and Mean Absolute Percentage Error (MAPE) were calculated.

Table 4.2: Table showing error in Gradient calculation in autodiff

PDE	Gradient	RMSE	MAPE
Wave Equation	u_x	0.0009	1.40
	u_t	0.0009	1.30
	u_{xx}	0.0101	3.20
	u_{tt}	0.0086	3.24
Inviscid Burgers Equation	u_x	0.0067	0.15
	u_t	0.0021	0.11
	u_{xx}	0.2953	8.35
	u_{tt}	0.0583	1.17
Helmholtz Equation	u_x	0.0033	0.92
	u_y	0.0049	0.76
	u_{xx}	0.0394	14.48
	u_{yy}	0.0423	17.56

The error is within 3% for first order gradients while error varied from 1%-20% for second order gradients.

4.4.2 Validation

A validation dataset was split after data sampling with train test split ratio of 0.2. Root mean squared loss similar to Eq. (3.4), is calculated and plotted in training curve. The model performed very well on the data within the domain. Model validation was done

only within the domain from which the data was sampled but not outside the domain. In fig. 4.1, 4.2 and 4.3, the line for Lval_u which is RMSE value calculated using same equation in Eq. 3.4. The Lval_u curve follows closely with L_u, curve.

Furthermore, to validate our method with No Free Lunch (NFL) Theorem, we tested the method with three different forms of partial differential equations. Wave equation as a form of hyperbolic PDE, Helmholtz equation as a form of elliptic PDE, and Inviscid Burgers equation which is a form of first order quasi-linear hyperbolic equation was used. Models performed well in all three forms of equations. Parabolic was not omitted from the research as it resulted in erroneous results in the initial tests. Further constraints might be required to solve such problems.

We can compare our result with other research work carried with similar process.

Table 4.3: Table comparing prediction error in our method with Hasan et al. (2019)

Equation	**Our Method**	**Hasan et al. (2019)**
Wave	1.42e-3	6.6e-4
Inviscid Burgers	6.43e-3	1.85e-4
Helmholtz	3.51e-3	4.75e-3

From the table 4.3, our results are close to the results publish by Hasan et al. (2019), to within 1 order of magnitude.

4.5 Discussion

In a nutshell, the neural network was capable of discovering the governing equation for Wave, Inviscid Burgers and Helmholtz equations. Further investigation is need on other types of PDEs. The effect of Physics-based regularization was noticed significantly which helped in learning the PDE. It is also important to be aware that multiple minimum optima for $\mathcal{L}_f$ might exist leading to different formulation than expected. The neural network architecture with 5 layers and neurons was found to be optimum. Both ELU and Softplus worked well on the model.

Furthermore, with the research on error metric it was found that use of RMSE as the metric for $\mathcal{L}_u$ improves the performance of the model. Ablation study of different combination of loss functions showed interaction between model training loss $\mathcal{L}_u$ and functional residual loss $\mathcal{L}_f$.

CHAPTER FIVE: CONCLUSION

5.1 Conclusion

We present a framework to recover governing partial differential equations in general form for the case where the discovery of PDE from first principles is intractable. We present a framework to discover PDE not restricted to first-order time derivative equations.

1. We developed a machine learning model that captures the underlying model from training data for a given PDE.
2. We analyzed the effect of layers, number of neurons, activation function, loss metric as well as the combination of loss functions on prediction performance.
3. We tested our model against three types of PDE viz. wave equation, Inviscid Burgers equation, and Helmholtz equation, to validate with the "No Free Lunch" theorem. The method showed excellent performance in all three equations.

There are, however, several open issues and missing gaps that could be further studied and improved.

5.2 Limitation

Although, this method allows us to predict possible governing partial differential equations from data. There are still many limitations to this method which are listed as follows:

1. Minimum-norm solution for homogeneous systems $H\Theta = 0$ is not always unique, leading to irrelevant solutions. As with any theoretical research, experiments on real data should be verified with physical experiments.
2. SVD method used in this method to solve for homogeneous linear equation only works with the normalized coefficients, hence determining actual coefficients may need further investigation.
3. Selection of certain candidate functions can lead to a different result; hence, one should rely on the domain expertise in selecting the candidate function.
4. Coupled partial differential equations have not been considered in the research. Data generated through coupled PDE may not work with this system.

5.3 Suggestion for Future Research

There are still many unanswered questions that need further investigation, some of which are listed below.

- Sobolev training (Czarnecki et al. (2017)) can be used to fit the target value and some higher-order derivative of target value if available to improve the quality of the regressor as well as the quality of further derivatives.
- Higher-order, higher dimensions of partial differential equations, as well as other forms of partial differential equations, can be tested.
- Additional constraints can be imposed while training to ensure a unique solution. Just as a PDE solution is generally not unique and additional conditions must generally be specified on the boundary of the region where the solution is defined, finding PDE from data might lead to many multiple local minima, i.e., non-unique identification of PDE. Hence, additional constraints might be necessary.
- Effect of the choice of candidate functions can be further investigated; certain candidate functions might lead to local minima.
- Experiments not performed on noisy data, but real data are noisy as well as sparse. Further investigation of sparse and noisy data can be conducted. Furthermore, this method can be applied to real-world experiment data to verify its correctness further.
- The neural network architectures such as RNNS like LSTM and ConvLSTM can be used to capture Spatio-temporal data more effectively.

REFERENCES

Akaike, H. (1974), 'A new look at the statistical model identification', *IEEE Transactions on Automatic Control* **19**(6), 716 – 723.
URL: *https://ieeexplore.ieee.org/document/1100705*

Aster, R. C., Borchers, B. & Thurber, C. H. (2019), *Parameter Estimation and Inverse Problems*, third edition edn, Elsevier Publications.

Åström, K. J. & P., E. (1971), 'System identification—a survey', *Automatica* **7**(2), 123 – 162.
URL: *http://www.sciencedirect.com/science/article/pii/0005109871900598*

Baydin, A. G., Pearlmutter, B. A., Alexey, reyevich Radul & Siskind, J. M. (2018), 'Automatic differentiation in machine learning: a survey', *Journal of Machine Learning Research* **18**(153), 1–43.
URL: *http://jmlr.org/papers/v18/17-468.html*

Berg, J. & Nyström, K. (2019), 'Data-driven discovery of PDEs in complex datasets', *Journal of Computational Physics* **384**, 239–252.

Bishop, C. M. (2006), *Pattern Recognition and Machine Learning*, Vol. 1, Springer.

Bongard, J. & Lipson, H. (2007), 'Automated reverse engineering of nonlinear dynamical systems', *Proceedings of the National Academy of Sciences of the United States of America* **104**(24), 9943–9948.

Brunton, S. L., Proctor, J. L., Kutz, J. N. & Bialek, W. (2016), 'Discovering governing equations from data by sparse identification of nonlinear dynamical systems', *Proceedings of the National Academy of Sciences of the United States of America* **113**(15), 3932–3937.

Champion, K., Lusch, B., Nathan Kutz, J. & Brunton, S. L. (2019), 'Data-driven discovery of coordinates and governing equations', *Proceedings of the National Academy of Sciences of the United States of America* **116**(45), 22445–22451.

Clevert, D.-A., Unterthiner, T. & Hochreiter, S. (2015), 'Fast and accurate deep network learning by exponential linear units (elus)', *arXiv preprint* .

Czarnecki, W. M., Osindero, S., Jaderberg, M., Swirszcz, G. & Pascanu, R. (2017), Sobolev training for neural networks, *in* 'Proceedings of the 31st International Conference on Neural Information Processing Systems', NIPS'17, Curran Associates Inc., Red Hook, NY, USA, p. 4281–4290.

Dong, B., Jiang, Q. & Shen, Z. (2017), 'IMAGE RESTORATION: Wavelet Frame Shrinkage, Nonlinear Evolution PDEs, and Beyond *', *Society for Industrial and Applied Mathematics* **15**(1), 606–660.
URL: *http://www.siam.org/journals/mms/15-1/M103745.html*

Dugas, C., Bengio, Y., Bélisle, F., Nadeau, C. & Garcia, R. (2001), Incorporating second-order functional knowledge for better option pricing, *in* T. K. Leen, T. G. Dietterich & V. Tresp, eds, 'Advances in Neural Information Processing Systems 13', MIT Press, pp. 472–478.
URL: *http://papers.nips.cc/paper/1920-incorporating-second-order-functional-knowledge-for-better-option-pricing.pdf*

Goodfellow, I., Bengio, Y. & Courville, A. (2016), *Deep Learning*, MIT Press. `http://www.deeplearningbook.org`.

Hasan, A., Pereira, J. M., Ravier, R., Farsiu, S. & Tarokh, V. (2019), 'Learning Partial Differential Equations from Data Using Neural Networks'.

Hascoet, L. & Pascual, V. (2013), 'The tapenade automatic differentiation tool: Principles, model, and specification', *ACM Transactions on Mathematical Software* **39**(3), 1–43.

Hornik, K. (1989), 'Multilayer Feedforward Networks are Universal Approximators', *Neural Networks* **2**, 359–366.

Hornik, K. (1991), 'Approximation capabilities of multilayer feedforward networks', *Neural Networks* **4**(2), 251–257.

Karpatne, A., Atluri, G., Faghmous, J. H., Steinbach, M., Banerjee, A., Ganguly, A., Shekhar, S., Samatova, N. & Kumar, V. (2017), 'Theory-guided data science: A new paradigm for scientific discovery from data', *IEEE Transactions on Knowledge and Data Engineering* **29**(10), 2318–2331.
URL: *https://ieeexplore.ieee.org/document/7959606*

Koopman, B. O. (1931), 'Hamiltonian Systems and Transformation in Hilbert Space', *Proceedings of the National Academy of Sciences* **17**(5), 315–318.
URL: *https://www.pnas.org/content/17/5/315*

Koza, J. R. (1992), *Genetic programming: On the programming of computers by means of natural selection,*, Vol. 1, MIT Press, Cambridge, MA.

Lee, D.-H., Zhang, S., Fischer, A. & Bengio, Y. (2015), Difference target propagation, *in* 'Machine Learning and Knowledge Discovery in Databases', Springer International Publishing, Cham, pp. 498–515.

Loan, C. F. V. & Golub, G. H. (1996), *Matrix Computations*, 3 edn, Johns Hopkins University Press.

Long, Z., Lu, Y. & Dong, B. (2019), 'PDE-Net 2.0: Learning PDEs from data with a numeric-symbolic hybrid deep network', *Journal of Computational Physics* **399**.

Long, Z., Lu, Y., Ma, X. & Dong, B. (2018), PDE-Net: Learning PDEs from Data, *in* 'Proceedings of the 35th International Conference on Machine Learning, PMLR', pp. 3208–3216.
URL: *http://proceedings.mlr.press/v80/long18a.html*

Meurer, A., Smith, C. P., Paprocki, M., Čertík, O., Kirpichev, S. B., Rocklin, M., Kumar, A., Ivanov, S., Moore, J. K., Singh, S., Rathnayake, T., Vig, S., Granger, B. E., Muller, R. P., Bonazzi, F., Gupta, H., Vats, S., Johansson, F., Pedregosa, F., Curry, M. J., Terrel, A. R., Roučka, v., Saboo, A., Fernando, I., Kulal, S., Cimrman, R. & Scopatz, A. (2017), 'Sympy: symbolic computing in python', *PeerJ Computer Science* **3**, e103.
URL: *https://doi.org/10.7717/peerj-cs.103*

Meyes, R., Lu, M., de Puiseau, C. W. & Meisen, T. (2019), 'Ablation studies in artificial neural networks'.

Mitchell, T. M. (1997), *Machine Learning*, McGraw Hill.
URL: *http://www.cs.cmu.edu/ tom/mlbook.html*

Montgomery, D. C. (2017), *Design and Analysis of Experiments*, 9 edn, John Wiley & Sons, Inc, Hoboken, NJ.

Nøkland, A. (2016), Direct feedback alignment provides learning in deep neural networks, *in* 'Advances in neural information processing systems', Curran Associates, Inc., pp. 1037–1045.
URL: *http://papers.nips.cc/paper/6441-direct-feedback-alignment-provides-learning-in-deep-neural-networks.pdf*

Oberkampf, W. L. & Roy, C. J. (2010), *Verification and Validation in Scientific Computing*, Cambridge University Press.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L. & Lerer, A. (2017), Automatic differentiation in PyTorch, *in* 'Advances in Neural Information Processing Systems 32', pp. 8024–8035.
URL: *http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf*

Pearlmutter, B. A. & Siskind, J. M. (2008), 'Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator', *ACM Transactions on Programming Languages and Systems* **30**(2).

Qin, T., Wu, K. & Xiu, D. (2019), 'Data driven governing equations approximation using deep neural networks', *Journal of Computational Physics* .

Raissi, M., Perdikaris, P. & Karniadakis, G. E. (2019), 'Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations', *Journal of Computational Physics* **378**, 686–707.

Rudy, S. H., Brunton, S. L., Proctor, J. L. & Kutz, J. N. (2017), 'Data-driven discovery of partial differential equations', *Science Advances* **3**(4).

Rumelhart, D., Hinton, G. & Williams, R. (1986), 'Learning representations by back-propagating errors', **323**, 533–536.

Sahoo, S., Lampert, C. & Martius, G. (2018), Learning equations for extrapolation and control, *in* J. Dy & A. Krause, eds, 'Proceedings of the 35th International Conference on Machine Learning', Vol. 80 of *Proceedings of Machine Learning Research*, PMLR, Stockholmsmässan, Stockholm Sweden, pp. 4442–4450.
URL: *http://proceedings.mlr.press/v80/sahoo18a.html*

SAS Institute Inc., Cary, NC (1989-2019), 'Jmp®'.
URL: *https://hadoop.apache.org*

Schmid, P. J. (2010), 'Dynamic mode decomposition of numerical and experimental data', *Journal of Fluid Mechanics* **656**, 5–28.

Schmidt, M. & Lipson, H. (2009), 'Distilling free-form natural laws from experimental data', *Science* **324**(5923), 81–85.

Schwarz, G. (1978), '"Estimating the Dimension of a Model."', *The Annals of Statistics* **6**(2), 461 – 464.

Strang, G. (2016), *Introduction to linear algebra*, 5 edn, Wellesley-Cambridge Press Wellesley, MA.

Tibshirani, R. (1996), 'Regression shrinkage and selection via the lasso', *Journal of the Royal Statistical Society. Series B (Methodological)* **58**(1), 267–288.
URL: *http://www.jstor.org/stable/2346178*

Trask, A. (2019), *Grokking deep learning*, Manning Publications Co.

Wolpert, D. H. & Macready, W. G. (1997), 'No free lunch theorems for optimization', *IEEE Transactions on Evolutionary Computation* **1**(1), 67–82.

Xu, H., Chang, H. & Zhang, D. (2019), 'DL-PDE: Deep-learning based data-driven discovery of partial differential equations from discrete and noisy data'.

Appendix A: Software and Devices

A.1 Software Configuration

The software used in the research work are listed in the following table

Table A.1: Software and Libraries used in the research

Tool	Packages	Version
Python		3.7.4
	ipykernel	5.2.1
	ipython	7.14.0
	jupyter	1.0.0
	matplotlib	3.2.1
	mlflow	1.8.0
	numpy	1.18.4
	notebook	6.0.3
	pyDOE	0.3.8
	scikit-learn	0.22.2.post1
	scipy	1.4.1
	sympy	1.5.1
	torch	1.5.0+cpu

A.2 Hardware Configuration

For the computation, Google Colaboratory, or "Colab" was used.

- CPU : 2 × Intel(R) Xeon(R) CPU @ 2.30GHz
- RAM : 13 GB
- GPU : Nvidia Tesla K80 (12 GB)

Appendix B: Python Source Code

B.1 Data Generation

../../src/dataset/sample_data.py

```python
import numpy as np

def sample_data(fcn, domain, n_samples=1000, dtype=np.float32, only_X=False):
    """ Generate samples of data from the function provided
        Parameter
        ----------
        fcn: str
            String to create sympy function for generating data
        domain: Dict[str,List[int]]
            Domain min,max for each independent variable
        n_samples: int
            Number of samples to be generated
        Return
        ------
        X : ndarray
            sampled n-D array of independent variable
        y : ndarray
            sampled 1-D array of dependent variable
    """
    from pyDOE import lhs
    from sympy import sympify, lambdify

    print(f"Generating {n_samples} data samples")
    y_lambda = lambdify(domain.keys(), sympify(fcn), "numpy")
    lb = np.array([domain[d][0] for d in domain],dtype=dtype)
    ub = np.array([domain[d][1] for d in domain],dtype=dtype)
    X = lb + (ub - lb) * lhs(len(domain.keys()), samples=n_samples)
    if only_X: return X
    y = y_lambda(*X.T)
    return X, y
```

B.2 Model and Training

B.2.1 Neural Network

../../src/models/LinearModel.py

```
from torch import nn

class LinearModel(nn.Module):
    def __init__(self, in_sz, hidden_sz, n_layers, out_sz, actn=nn.Softplus(beta=1)):
        super(LinearModel,self).__init__()
        layers = [in_sz,*[hidden_sz]*n_layers, out_sz]
        self.actn = actn
        self.net = nn.Sequential()
        self.net.add_module('linear_%d' %(1), nn.Linear(in_sz, hidden_sz,bias=True))
        self.net.add_module('activation_%d' %(1), self.actn)
        for i in range(1,n_layers):
            self.net.add_module('linear_%d' %(i+1), nn.Linear(hidden_sz, hidden_sz,bias=False))
            self.net.add_module('activation_%d' %(i+1), self.actn)
        self.net.add_module('linear_%d' %(i+2), nn.Linear(hidden_sz, out_sz, bias=True))

        print(f"Linear model created with layers: {[in_sz, *[hidden_sz]*n_layers, out_sz]} ")


    def forward(self,x):
        return self.net(x)
```

B.2.2 Evaluation of matrix H

../../src/utils/dictionary.py

```
import torch
from torch.autograd import grad

def getdictionary(func,X_collocation, domain, dictionary):

    vardict = dict(zip(domain.keys(),X_collocation))

    u = func(torch.cat(X_collocation,dim=1))

    fundict = vardict.copy()
    fundict['u'] = u

    fundict_= {}

    flag = True
    while flag:
        try:
            M =  torch.cat(eval(dictionary,fundict),dim=1)
            flag = False
        except NameError as error:
            key = str(error).split("'")[1]
            for i in range(2,len(key)):
                keyi = key[:i+1]
                if not(keyi in fundict.keys()):
                    var = vardict[keyi[i]]
                    if i==2:
                        fun = u
                        fundict_[keyi] = grad(fun,var,create_graph=
    True,\
                                              grad_outputs=torch.ones_like(fun))
                        fundict[keyi] = fundict_[keyi][0]
                    else:
                        fun = fundict_[keyi[:-1]]
                        fundict_[keyi] = grad(fun,var,create_graph=
    True,\
                                              grad_outputs=torch.ones_like(fun
    [0]))
                        fundict[keyi] = fundict_[keyi][0]

    return M
```

B.2.3 Training

../../src/run.py

```
from tqdm.notebook import tqdm
import numpy as np
import pandas as pd
import torch
from torch import nn
import torch.nn.functional as F
from sklearn.model_selection import train_test_split

def run(params):
    log_freq=100
    results = dict()
    seed =random.randint(0, 999); np.random.seed(seed); torch.manual_seed(seed);

    fcn, domain, dictionary, true_coeff = equations[params["eq_type"]].values()
    c_truth = torch.from_numpy(np.array(true_coeff)).type(dtype).to(device)

    # Data Generation
    X_train, u_train = sample_data(fcn, domain, n_samples=params["n_pts"])
    X_train, X_valid, u_train, u_valid = train_test_split(X_train, u_train, test_size=0.2, random_state=seed)
    X_collocation = sample_data(fcn, domain, params["n_collocation_pts"], only_X=True)
    X_collocation = [torch.tensor(arr, dtype=dtype, device=device, requires_grad=True) for arr in np.hsplit(X_collocation,2)]

    X_train = torch.tensor(X_train, dtype=dtype).to(device)
    u_train = torch.tensor(u_train, dtype=dtype).unsqueeze(1).to(device)
    X_valid = torch.tensor(X_valid, dtype=dtype).to(device)
    u_valid = torch.tensor(u_valid, dtype=dtype).unsqueeze(1).to(device)

    # Model Creation
    in_size, layers,  width, actn = len(domain), params["model"][0], params["model"][1], params["model"][2]
    model = LinearModel(in_size, width, layers, 1, actn).to(device)
    model.train()

    # Optimiser Setup
```

```
    Theta = nn.Parameter(torch.randn(len(true_coeff), requires_grad=
True, device=device, dtype=dtype))
    optim_params = [{'params': model.parameters(), 'lr': params["
lr_model"]},
                    {'params': Theta, 'lr': params["lr_theta"]}]
    optimizer = torch.optim.Adam(optim_params)
    scheduler = torch.optim.lr_scheduler.ExponentialLR(optimizer
,.9998)
    results["params"] = params

    # Train Model
    lambda_f, lambda_norm = 0, 0
    svd_init = params["svd_init"]
    lambda_u = params["lambda_u"]
    # sv_test = torch.Tensor()
    history = []
    for epoch in tqdm(range(params["n_epochs"])):
        if epoch == svd_init:
            lambda_f, lambda_norm = params["lambda_f"], params["
lambda_sparse"]

        for i in range(0, X_train.shape[0], params["minibatch_size"])
:
            X_train_mini = X_train[i:i + params["minibatch_size"]]
            u_train_mini = u_train[i:i + params["minibatch_size"]]

            def closure():
                optimizer.zero_grad()

                u_pred = model(X_train_mini)
                M = getdictionary(model, X_collocation, domain,
dictionary)

                # Residual MAE in prediction, Validation
                loss_u = F.l1_loss(u_pred, u_train_mini)

                if epoch in svd_init:
                    su,s,sv = torch.svd(M)
                    Theta.data = sv[:,-1].data
                Theta_norm = Theta/torch.norm(Theta)
                Theta.data = Theta_norm.data

                lf = torch.sum(M*Theta_norm,dim=1)

                loss_f = torch.norm(lf)**2 / X_collocation[0].shape
[0] # LF_MSE
                loss_norm = torch.norm(Theta_norm,p=1)
```

```
                loss = lambda_u*loss_u + lambda_f*loss_f +
    lambda_norm*loss_norm
74              loss.backward(retain_graph=False)

76              # Validation
                if epoch%log_freq == 0 and i==0:
78                  u_pred = model(X_valid)
                    valid_loss_u = F.l1_loss(u_pred,u_valid)
80                  svd_err1, svd_err2, vec = model_pred_error(model,
    X_collocation,domain,dictionary,c_truth)
                    results["final"] = [loss_u.item(), loss_f.item(),
     loss_norm.item(), loss.item(), svd_err1, svd_err2]
82              return loss
            optimizer.step(closure)
84          scheduler.step()
     return results
```