DEEP LEARNING ALGORITHMS FOR TIME-DEPENDENT PARTIAL DIFFERENTIAL EQUATIONS

By

Jie Long

Dissertation Defense
Submitted in Partial Fulfillment
of the Requirement for the Degree of
Ph.D. in Computational and Data Science

Middle Tennessee State University
July, 2024

Dissertation Committee:

Dr. Abdul Khaliq, Chair

Dr. Jing Kong

Dr. Yixiang Wu

Dr. Wandi Ding

Abstract

Deep learning algorithms have demonstrated encouraging outcomes in resolving Partial Differential Equations. The advent of physics-informed Neural Networks has greatly enhanced the precision and effectiveness of Deep Learning-based approaches for solving partial differential equations. The basic idea of such Deep Learning algorithms is constraining the output of neural networks to satisfy the physics laws and certain conditions by incorporating the physical laws and boundary conditions directly into the loss function for training the neural networks. Using this technology, we propose a variant of the Physics-Informed Neural Network to identify time-varying parameters of the Susceptible-Infectious-Recovered-Deceased model for COVID-19 by fitting daily reported cases. The learned parameters are verified with an ordinary differential equation solver, and the effective reproduction number is calculated. Additionally, a Long Short-Term Memory network predicts future weekly time-varying parameters, demonstrating the accuracy and effectiveness of combining these two models.

Then, we explore the method that can solve the partial differential equations using the sparse data. We combine a neural network with a numerical approach to address time-dependent partial differential equations using initial conditions and limited observed data. The Gated Recurrent Units network estimates time iteration schemes, integrating prior knowledge of governing equations. A numerical implicit approach is applied to calculate new time iteration schemes, with the loss function incorporating the difference between these schemes. After that, we propose a novel physics-informed encoder-decoder gated recurrent neural network to solve time-dependent partial differential equations without using observed data. The encoder approximates the underlying patterns and structures of solutions over the entire spatio-temporal domain. The approximated solution is processed by the decoder, a Gated Recurrent Units layer, utilizing the initial condition as the initial state to retain critical information in the hidden states. Boundary conditions are enforced in the final prediction to enhance model performance. The effectiveness of these two methods has been validated through their application to several problems.

Additionally, we observe the traditional physics-informed neural network often fails to con-

verge due to imbalances in the multi-component loss function within the back-propagated gradients during training. The standard approach to mitigate this issue involves adding appropriate weights to each component of the loss function, but determining the correct weights is challenging. Therefore, we introduce the Self-Learning Physics-Informed Neural Network to solve some non-linear partial differential equations. In this method, weights are learned by separate neural networks, eliminating the need for hyper-parameter fine-tuning. The effectiveness of our method is demonstrated by the Burgers' and Burgers-Fisher equations.

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to all those who have contributed to the completion of this dissertation.

First and foremost, I am profoundly grateful to my advisor, Dr. Abdul Khaliq, for his invaluable guidance, encouragement, and unwavering support throughout my PhD journey. His expertise and insights have been instrumental in shaping my research and helping me overcome numerous challenges.

I extend my heartfelt thanks to my dissertation committee members, Dr. Jing Kong, Dr. Yixiang Wu, and Dr. Wandi Ding, for their thoughtful feedback, constructive criticism, and invaluable suggestions, which have greatly improved the quality of this work. I would like to express my sincere gratitude to Dr. John Wallin, the program director of the Computational Science program, for his funding of my studies at MTSU and his continuous support and dedication to all the students in the program. I am grateful to Dr. Khaled Furati for his numerous meetings and insightful questions, which have greatly improved my research work.

Last but not least, I would like to express my deepest gratitude to my family. To my parents, thank you for your unconditional love, support, and encouragement. I dedicate this dissertation to you.

TABLE OF CONTENTS

Page

ABSTRACT . . . . **ii**

LIST OF TABLES . . . . **ix**

LIST OF FIGURES . . . . **x**

1 INTRODUCTION **1**

1.0.1 Deep Neural Network 1

1.0.2 Basic Concepts and Terminology 1

1.0.3 Neural Network Architectures 4

1.0.3.1 Feedforward Neural Networks 4

1.0.3.2 Recurrent Neural Network 5

1.0.3.3 Long Short Term Memory 6

1.0.3.4 Gated Recurrent Units 8

1.0.4 Training Neural Networks 9

1.0.4.1 Loss Function 10

1.0.4.2 Backpropagation 10

1.0.4.3 Optimizers 12

1.0.5 Special Types of Neural Network 13

1.0.5.1 Physics-Informed Neural Network 13

1.0.5.2 Encoder-Decoder 13

2 Identification and prediction of time-varying parameters of COVID-19 model: a data-driven deep learning approach **15**

2.1 Introduction 15

2.2 Epidemiology Model . . . 18
2.2.1 SIRD Model . . . 18
2.2.2 Reproduction Number . . . 19
2.3 Main Architecture . . . 20
2.4 Data and Algorithms . . . 21
2.4.1 Data . . . 21
2.4.2 Parameter Identification Algorithm . . . 22
2.4.2.1 PINN Architecture for SIRD Model . . . 22
2.4.2.2 Parameters Identification Algorithm . . . 23
2.4.2.3 LSTM for Prediction of Infectious Cases . . . 25
2.5 Simulation . . . 26
2.5.1 Time-Varying Parameters . . . 26
2.5.2 Computational Complexity . . . 30
2.5.3 Simulated Reproduction Numbers . . . 31
2.5.4 Prediction of Infectious Cases . . . 32
2.6 Conclusions . . . 33

3 Sparse Data Regulation on GRU Neural Network . . . 34

3.1 Introduction . . . 34
3.2 Methodology . . . 36
3.2.1 Problem setup . . . 36
3.2.2 Algorithm . . . 36
3.3 Numerical Experiments . . . 39
3.3.1 Burgers' equation . . . 39
3.3.2 Allen-Cahn equation . . . 42
3.3.3 Non-Linear Schrodinger Equation . . . 44
3.3.4 Coupled Burgers' Equation . . . 46
3.3.5 Two-Dimentional Burgers' Equation . . . 49
3.3.6 Coupled Two-Dimensional Burgers' Equation . . . 52

3.3.7 Solving Inverse Problem 57
3.4 Conclusion 58

4 Physics-informed Encoder-Decoder Gated Recurrent Neural Network 60

4.1 Introduction 60
4.2 Problem Setup 62
4.3 Methodology 62
4.3.1 Encoder-Decoder GRUs 62
4.3.2 Initial/Boundary Conditions Treatment 63
4.3.3 Architecture 64
4.3.4 Physics-informed Loss function 64
4.4 Numerical Results 65
4.4.1 Burgers-Fisher Equation 66
4.4.2 Coupled Two-Dimensional Burgers' Equation 68
4.4.3 Comparision of PhyEDGNN with PINN 71
4.5 Conclusion 72

5 Self-Learning approach for solving PDEs 74

5.1 Methodology 74
5.1.1 Overview of the PINNs 74
5.1.2 PINNs with fixed weight 75
5.1.3 Self-Adaptive PINNs 76
5.1.4 Self-Learning PINNs 77
5.2 Numerical Experiments 78
5.2.1 Burgers' Equation 78
5.2.2 Burgers-Fisher Equation 80
5.3 Conclusion 82

6 Conclusion and Future Research 83

BIBLIOGRAPHY **85**

LIST OF TABLES

Table Page

2.1 Computational complexity in terms of time for the infectious cases. 30

2.2 The effect of different numbers of layers and neurons on the relative error between the learned infectious cases and collected data in New Mexico. . . . 30

2.3 The average CPU time is needed for each week in New Mexico 31

3.1 Burgers' Equation: Relative L_2 error between predicted and exact solutions at different times with different numbers of layers 41

3.2 Allen-Cahn Equation: Relative L_2 error between predicted and exact solutions at different times with different numbers of layers 43

3.3 Non-Linear Schrodinger Equation: Relative L_2 error between predicted and exact solutions at different times with different numbers of layers 46

3.4 Coupled Burgers' Equations: Relative L_2 error between predicted and exact solutions at different times with different numbers of layers 49

3.5 Relative L_2 error between predicted and exact solution at different times with different numbers of layers 52

3.6 Average relative L_2 error of u and v between predicted and exact solution at different times with different numbers of layers 54

3.7 Comparison between PINN and GRU 58

4.1 Relative L_2 errors between predicted and exact solutions using different step sizes 68

4.2 Average Relative L_2 errors between predicted and exact solutions using different step sizes 69

4.3 The comparison of PINN and PhyENGNN with respect to relative L_2 errors . 72

5.1 Relative L_2 errors of four methods for Burgers Equation 80

5.2 Relative L_2 errors of four methods for Burgers-Fisher equation 82

LIST OF FIGURES

Figure		Page
1.1	Common Activation Functions	3
1.2	Feedforward Neural Network and Backpropagation [41]	5
1.3	Basic Recurrent Neural Network Unit [85]	5
1.4	Architecture of LSTM Cell [19]	7
2.1	Flow Chart of Data-Driven Algorithm	17
2.2	COVID-19 data for New York, New Mexico, and Texas	21
2.3	Physics Informed Neural Network	22
2.4	Flow-Chart of LSTM	26
2.5	Simulation Results of New York of Daily Time-Varying Cases	27
2.6	Simulation Results of Texas of Daily Time-Varying Cases	27
2.7	Simulation Results of New Mexico of Daily Time-Varying Cases	28
2.8	Simulation Results of New York of Weekly Time-Varying Cases	28
2.9	Simulation Results of Texas of Weekly Time-Varying Cases	29
2.10	Simulation Results of New Mexico of Weekly Time-Varying Cases	29
2.11	Effective Reproduction Number	32
2.12	Predictions	33
3.1	Flow Chart	37
3.2	The Sample Code of Using Optimizers for Burgers' Equation	39
3.3	Burgers' Equation: Snapshots of the predicted solutions and exact solutions at $t = \{0.0, 0.15, 0.45, 0.60, 0.75, 0.99\}$.	41
3.4	The variation of loss with iterations for Allen-Cahn equation	41
3.5	Allen-Cahn Equation: Snapshots of the predicted solutions and exact solutions at $t = (0.0, 0.2, 0.4, 0.6, 0.8, 1.0)$.	43
3.6	The variation of loss with iterations for Allen-Cahn equation	43

3.7 Schrodinger Equation: Snapshots of the predicted solutions and exact solutions at $t = \{0, \frac{\pi}{10}, \frac{2\pi}{10}, \frac{3\pi}{10}, \frac{4\pi}{10}, \frac{5\pi}{10}\}$. 45
3.8 The variation of loss with iterations for Non-linear Schrodinger Equation . . 46
3.9 Coupled Burgers' Equations: Snapshots of the predicted solutions and exact solutions for u and v at $t = \{0.33, 0.66, 1\}$. 48
3.10 The variation of loss with iterations for Coupled Burgers' Equation 48
3.11 Two-Dimensional Burgers' Equation: Snapshots of predicted and exact solutions at $t = \{0.5, 1.0, 1.5, 2.0\}$ 51
3.12 The variation of loss with iterations for 2D Burgers' Equation 52
3.13 The variation of loss with iterations for Coupled 2D Burgers' Equation . . . 54
3.14 Coupled Two-Dimensional Burgers' Equation: Snapshots of predicted and exact Solutions for u at $t = \{0.5, 1.0, 1.5, 2.0\}$ 55
3.15 Coupled Two-Dimensional Burgers' Equation: Snapshots of predicted and exact Solutions for v at $t = \{0.5, 1.0, 1.5, 2.0\}$ 56
3.16 Inverse problem of Allen-Cahn Equation: Snapshots of predicted and exact solution 58

4.1 Architecture of Encoder-Decoder GRUs 64
4.2 PhyEDGNN Solving Burgers-Fisher equation 67
4.3 Coupled Two-Dimensional Burgers' Equation: Snapshots of predicted and exact Solutions for u at $t = \{0.6, 1.2, 1.8, 2.0\}$ 70
4.4 Coupled Two-Dimensional Burgers' Equation: Snapshots of predicted and exact Solutions for v at $t = \{0.6, 1.2, 1.8, 2.0\}$ 71

5.1 Self-Learning PINN for solving Burgers' Equation 80
5.2 Self-Learning PINN Solving Burgers-Fisher equation 82

Acronyms

Adam Adaptive Moment Estimation). 12

CNN Convolutional Neural Networks. 13

FNN Feedforward Neural Network. 4

GRUs Gated Recurrent Units. 6

L-BFGS Limited-memory Broyden–Fletcher–Goldfarb–Shanno. 12

LSTM Long Short-Term Memory. 6

MLP Multilayer Perceptron. 62

MSE Mean Square Error. 10

ODE Ordinary Differential Equation. 17

PINN Physics-Informed Neural Network. 13

ReLU Rectified Linear Unit. 2

RNN Recurrent Neural Network. 4

SEIR Susceptible Exposed Infectious Remove. 16

SGD Stochastic Gradient Descent. 12

SIR Susceptible-Infectious-Recovery. 15

SIRD Susceptible Infectious Recovered Deceased. 16

Tanh Hyperbolic Tangent Function. 2

CHAPTER 1

INTRODUCTION

Deep learning has emerged as a crucial instrument for addressing complicated problems with increasingly available data and computing resources. It has yielded numerous spectacular achievements in diverse fields, such as computer vision [79], image recognition [64], and natural language processing [52]. Implementing the deep learning technique in solving or learning differential equations is of great interest. The universal approximation theorem shows that the neural networks with a sufficient number of neurons in hidden layers can approximate any continuous function to any level of accuracy [15]. Based on this theorem, the vast majority of deep learning algorithms are designed and implemented to solve differential equations.

1.0.1 Deep Neural Network

Deep learning is a subset of machine learning that focuses on algorithms inspired by the structure and function of the brain's neural networks. It leverages multiple layers of artificial neurons to progressively extract higher-level features from raw input data, enabling the creation of models that can perform tasks such as image and speech recognition, natural language processing, and game playing with remarkable accuracy.

According to the universal approximation theorem [15], neural networks can approximate any continuous function given sufficient data and appropriate architecture. This theoretical foundation underpins the practical success of deep learning, which has become a cornerstone technology driving advancements in artificial intelligence and transforming industries by providing robust solutions to complex problems.

1.0.2 Basic Concepts and Terminology

Deep neural networks consist of interconnected units called neurons or nodes. These neurons are organized into layers, with each layer transforming the input data into progressively higher-level representations. There are three main types of layers in a neural network: the input layer receiving the raw data, hidden layers processing and extracting features from the input data,

and the output layer producing the final predictions or classifications.

Each neuron in these layers is connected to neurons in the subsequent layer through weighted connections, and these weights are crucial for the network's ability to learn from data. They determine the strength of the connection between neurons in adjacent layers. Biases are additional parameters that allow the model to fit the data more accurately by shifting the activation function.

The functioning of a neural network hinges on activation functions, which introduce non-linearity into the network, enabling it to model complex patterns and relationships in the data. Common activation functions include the Rectified Linear Unit (ReLU), Sigmoid function, Hyperbolic Tangent Function (Tanh), and Softplus function. Fig. 1.1 presents the shapes of these activation functions. We can see that the Tanh function is a smooth, continuous, and non-linear activation function. It maps input values to outputs in the range of -1 to 1. The ReLU is a widely utilized activation function in deep learning due to its simplicity and effectiveness. It outputs the positive input directly, otherwise, it outputs zero. The classical Sigmoid function maps input values to a range between 0 and 1, which can be interpreted as probabilities, making it suitable for binary classification tasks. Softplus is continuous and differentiable, providing a non-linear activation similar to ReLU but without the sharp transition at zero. Selecting the appropriate activation function is crucial and depends on the specific task at hand, as different activation functions have unique features that make them suitable for various purposes.

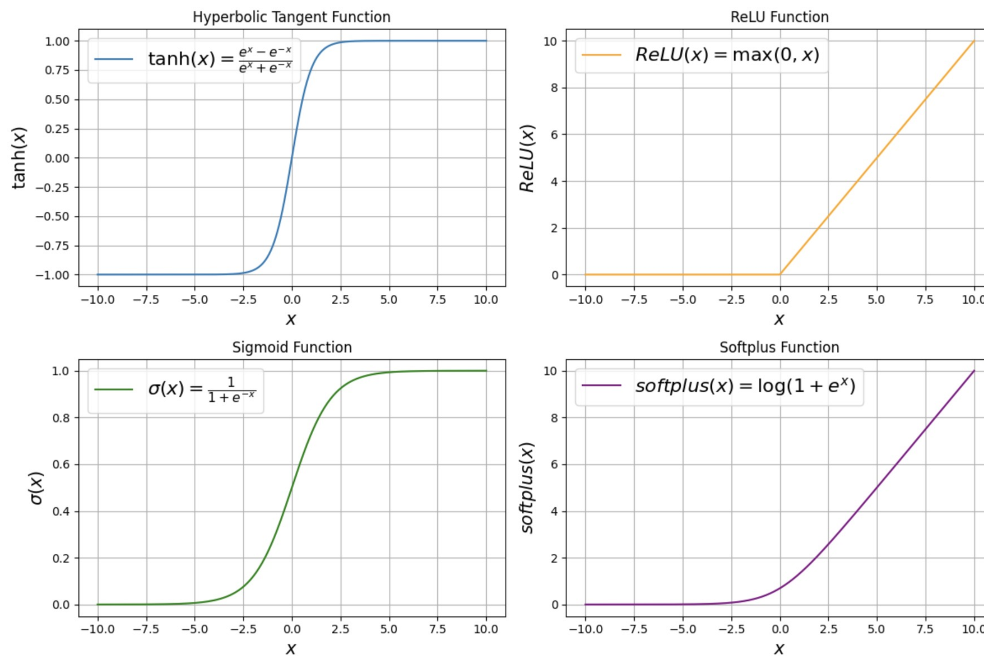


Figure 1.1: Common Activation Functions

During the training process, the input data is transmitted through the network using forward propagation, which involves calculating the output of each neuron and transmitting it to the next layer. The output of the network is subsequently compared to the target values using a loss function to calculate the error.

In order to minimize this error, the network undergoes a procedure known as backward propagation, in which the error is retroactively transmitted through the network to modify the weights and biases. The adjustment of weights is guided by optimization techniques, such as gradient descent, which updates the weights in a direction that minimizes the error. The learning rate, a crucial hyperparameter, affects the size of the updates made throughout the training process. Choosing the ideal learning rate ensures that the training process converges effectively and consistently. Using a learning rate that is too high can cause the training process to become unstable while using a learning rate that is too low can cause the training process to become overly slow. Understanding these underlying ideas and languages is essential for effectively designing and teaching deep neural networks to solve complex problems in various domains.

1.0.3 Neural Network Architectures

Selecting an appropriate neural network architecture is crucial for improving the model's ability to learn from data and perform well on specific tasks. The architecture determines the depth and complexity of the network, influencing how well it can capture intricate patterns and relationships in the data. This section will introduce two main neural network architectures: Feedforward Neural Network (FNN) and Recurrent Neural Network (RNN).

1.0.3.1 Feedforward Neural Networks

A FNN is a fundamental type of artificial neural network where the flow of information moves in one direction. Each neuron processes the input data and passes its output to the next layer until the final prediction is made. Fig. 1.2 describes the forward and backward propagation of FNN [41]. Fig. 1.2 shows that a multilayer neural network can distort input space to make data classes linearly separable, as illustrated by the transformation of a regular grid through hidden units. Fig. 1.2b the chain rule of derivatives describes how a small change in x affects z through intermediate variable y. The below two figures present the forward and backward propagation. We can use the composited function Eq. 1.1 to describe the forward propagation.

$$\hat{y} = \sigma\left(W^{(L)}\left(\sigma\left(W^{(L-1)}\cdots\left(\sigma\left(W^{(1)}x+b^{(1)}\right)\right)\cdots+b^{(L-1)}\right)\right)+b^{(L)}\right) \tag{1.1}$$

where x is the input vector, $\hat{y}$ is the output vector after the output layer, $W^{(l)}$ and $b^{(l)}$ represent the weight matrix and bias vector for layer l, respectively, and σ is the activation function. The nested composition function captures the entire forward propagation process through the neural network, from the input x to the final output $\hat{y}$. The backward propagation will be introduced in section 1.0.4.

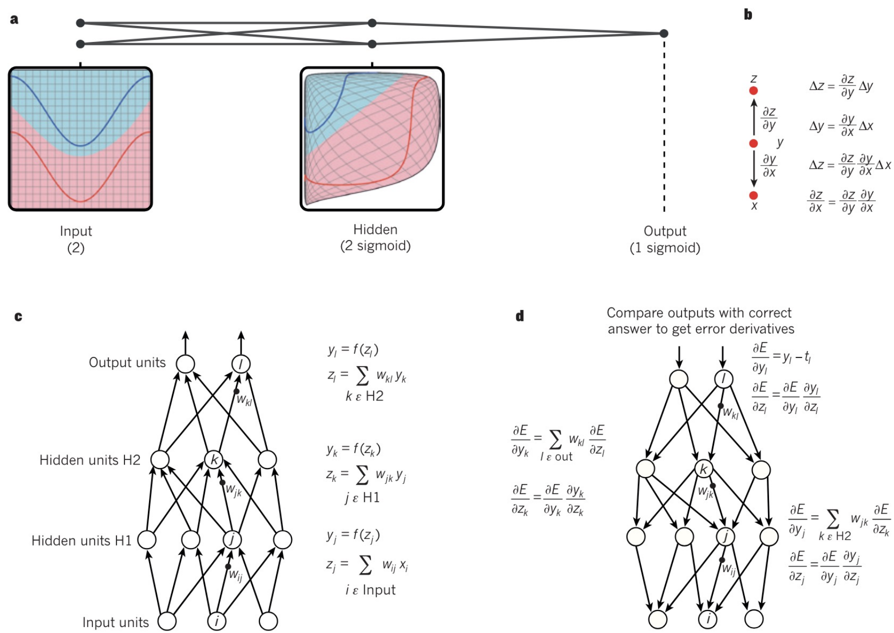


Figure 1.2: Feedforward Neural Network and Backpropagation [41]

1.0.3.2 Recurrent Neural Network

Recurrent Neural Networks (RNNs) are a type of neural networks that excel in handling sequential data problems [68]. RNNs, unlike conventional neural networks, possess connections that create directed cycles, enabling them to preserve a state and represent temporal relationships in sequences. This looping mechanism makes RNNs particularly well-suited for tasks involving time-series data, natural language processing, and other scenarios where context from previous inputs is important. Fig. 1.3 presents the compressed (left) and the unfolded (right) basic recurrent neural network unit.

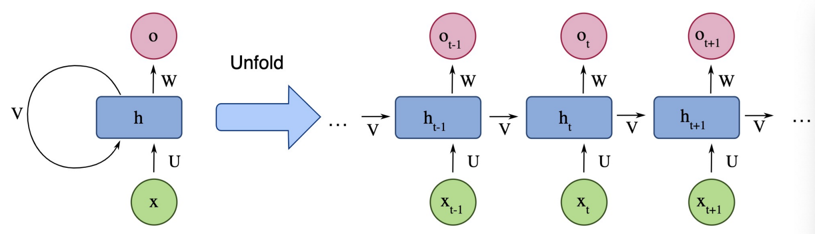


Figure 1.3: Basic Recurrent Neural Network Unit [85]

- x_t represents the input vector at time t.
- h_t is the hidden state at time step t. It's the "memory" of the network.

- o_t is the output vector at time t.
- U is the weight matrix for the input x.
- V is the weight matrix for the hidden state h_t.
- W is the weight matrix for the output vector o_t.

Eq.(1.2) shows the way to calculate the new hidden state. f is the activation function and is applied to the weighted sum of the current input x_t and the previous hidden state h_{t-1} to produce the new hidden state h_t. This process allows the RNNs to maintain a memory of previous inputs and capture temporal dependencies in the data. Eq.(1.3) is the output calculation. The weighted sum of hidden state and bias is passed through a nonlinear activation function g to obtain the final output.

$$h_t = f(Ux_t + Wh_{t-1} + b) \tag{1.2}$$

$$o_t = g(Wh_t + b) \tag{1.3}$$

RNNs are powerful for sequence modeling because they leverage their internal state to process sequences of inputs. However, traditional RNNs can suffer from issues like vanishing gradients, which limit their ability to capture long-term dependencies in sequences. Thus, advanced variants like Long Short-Term Memory (LSTM) networks and Gated Recurrent Units (GRUs) have been developed to address these limitations.

1.0.3.3 Long Short Term Memory

LSTM networks are a specialized form of RNNs designed to model temporal sequences and their long-range dependencies more accurately than conventional RNNs [30]. LSTM as shown in Fig. 1.4 are chains of memory cells containing three gates: the input gate, forget gate, and output gate as explained in [19]. The key part of these memory cells is the cell state that could maintain the global information of sequence in each time step. The content of the cell state would be modified at different time steps through the forget gates and input gates.

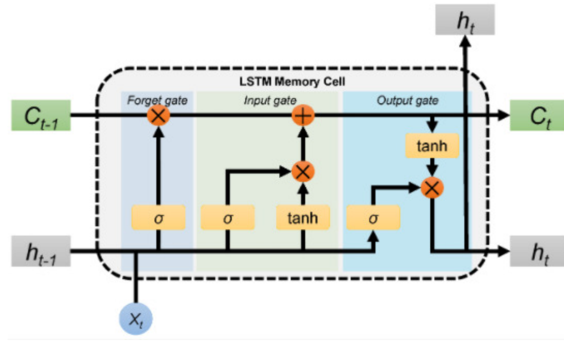


Figure 1.4: Architecture of LSTM Cell [19]

More specifically, forget gates decide what information would be thrown away by utilizing the sigmoid function. Instead, input gates determine what new information would be stored by combining sigmoid and tanh functions. Then, the cell state is updated and moves forward to the next state by implementing some operations. After that, the output is calculated in the output gate. By analyzing the structure of a memory cell, we could see that LSTM architecture is capable of storing long-term memory by putting new information into cell states.

The first gate, the input gate, determines how much of the previous cell state, C_{t-1} should be carried forward. It can be expressed as:

$$f_t = \sigma(W_f \cdot [x_t, h_{t-1}] + b_f) \tag{1.4}$$

- W_f is the weight matrix for the forget gate.
- b_f is the bias term for the forget gate.
- f_t is the forget gate determining what portion of the previous cell state to forget.

The input gate i_t and Candidate Memory Cell $\tilde{C}_t$ control the update of the cell state with new information. They can be updated as follows:

$$i_t = \sigma(W_i \cdot [x_t, h_{t-1}] + b_i) \tag{1.5}$$

$$\tilde{C}_t = tanh(W_C \cdot [x_t, h_{t-1}] + b_C) \tag{1.6}$$

- W_C is the weight matrix for the candidate cell state.

- b_C is the bias term for the candidate cell state.
- i_t is the input gate vector determining how much new information will update the cell state.

The output gate regulates the final output o_t.

$$o_t = \sigma(W_o \cdot [x_t, h_{t-1}] + b_o) \tag{1.7}$$

- W_o is the weight matrix for the output gate.
- b_o is the bias term for the output gate.
- o_t is the output vector determining how much of the cell state to output.

The cell state C_t can be updated by combining the forget and input gates.

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t. \tag{1.8}$$

The hidden state h_t is computed based on the updated cell state and the output gate.

$$h_t = o_t * tanh(C_t). \tag{1.9}$$

1.0.3.4 Gated Recurrent Units

The GRU is a variation of the traditional RNN unit that includes gating mechanisms to control the flow of information in and out of the cell state [14]. The GRU unit typically comprises two gates: the update gate and the reset gate. The update gate regulates the extent to which the previous hidden state is combined with the current input, while the reset gate controls the degree to which the previous hidden state is discarded. The equations below are given to describe the

GRU unit.

$$
\begin{aligned}
z_t &= \sigma(W^{(z)}x_t + U^{(z)}x_t + b_z) \\
r_t &= \sigma(W^{(r)}x_t + U^{(r)}x_t + b_r) \\
\hat{h}_t &= tanh(Wx_t + U^h(r_t \odot h_{t-1}) + b_h) \\
h_t &= (1 - z_t) \odot h_{t-1} + z_t \odot \hat{h}_t
\end{aligned}
$$

- x_t is the x values at time t.
- h_{t-1} is the previous hidden state.
- z_t is the update gate output.
- r_t is the reset gate output.
- h'_t is the candidate activation.
- $W^{(z)}, U^{(z)}, W^{(r)}, U^{(r)}, W, U, b_z, b_r, b_h$ are the weight matrices and bias vectors to be learned during training.

GRUs can capture long-term dependencies in sequential data and alleviate the vanishing gradient problem often encountered in traditional RNNs. Additionally, GRU is generally faster to train and requires fewer parameters than LSTM, which is another type of RNN architecture that can model sequential data. These are the main reasons why we chose the GRUs as our decoder. The next subsection will introduce how we construct our model by utilizing GRUs.

1.0.4 Training Neural Networks

Once the model has been built, it is initialized with random weights. Training the model to obtain suitable parameters is essential. The core of the training process involves feeding the network input data in batches, computing the predictions, and then measuring the error using a loss function, which quantifies the difference between the predicted outputs and the actual target values. This error is then backpropagated through the network, computing gradients of the loss with respect to each weight by applying the chain rule of calculus.

1.0.4.1 Loss Function

Loss functions measure the difference between the predicted outputs and the actual target values. They are crucial for guiding the optimization process during training. In this subsection, we mainly introduce two types of loss functions: Mean Square Error (MSE) and Cross-Entropy Loss.

Generally, the MSE is applied to regression problems, which involve predicting a continuous output variable based on one or more input variables, aiming to model the relationship between the inputs and the output. The MSE calculates the average squared difference between predicted and actual values.

$$MSE = \frac{1}{n}\Sigma_{i=1}^{n}(y_i - \bar{y}_i)^2 \tag{1.10}$$

- n is the number of data points.
- y is the collected data.
- $\bar{y}$ is the prediction of the neural network.

As for the Cross-Entropy loss function, it is commonly proposed to solve classification problems, which entail determining a discrete category label for a given input based on its characteristics, to assign the input to one of several predefined groups.

$$\text{Cross-Entropy Loss } = -\Sigma_{i=1}^{n} y_i log(\bar{y}_i) \tag{1.11}$$

To enhance model performance, it is crucial to select an appropriate loss function tailored to the specific task at hand. Simultaneously, employing efficient techniques to minimize errors is essential. In the subsequent sections, the backward propagation and several well-known optimizers are introduced.

1.0.4.2 Backpropagation

Backpropagation is the algorithm used to compute the gradients of the loss function with respect to each weight in the neural network [68]. This algorithm enables the network to adjust its weights to minimize the error between the predicted output and the actual target values. Then,

we will briefly describe the backpropagation in FNN with two hidden layers. Assume we have the loss function

$$L = 0.5 \times (y_l - \bar{y}_l)^2, \tag{1.12}$$

where $\bar{y}_l$ is observed data, and y_l represents the neural network prediction. The activation function is Sigmoid function $\frac{1}{1+e^{z_l}}$. Starting at the output layer, the gradient of the loss function with respect to the output is calculated. Then, using the chain rule, the gradient of the loss function with respect to each layer's parameters and inputs is computed iteratively. We first calculate the derivative of the loss function with respect to y_l.

$$\frac{\partial L}{\partial y_l} = y_l - \bar{y}_l \tag{1.13}$$

Then, the gradients of y_l with respect to the input of output layer z_l are computed as follows:

$$\begin{aligned}\frac{\partial y_l}{\partial z_l} &= e^{-z_l}(1+e^{-z_l})^2 \\ &= -y_l(1-y_l)\end{aligned} \tag{1.14}$$

By combining the Eq.(1.13) and Eq.(1.14), we can obtain the gradients on the output layer.

$$\frac{\partial L}{\partial z_l} = y_l(1-y_l)(y_l - \bar{y}_l) \tag{1.15}$$

Subsequently, the gradients for the hidden layers, using H_1 and H_2 to represent the first and second hidden layers respectively, are obtained using the chain rule.

$$\frac{\partial L}{\partial z_k} = y_k(1-y_k) \sum_{l \in input} w_{kl} \frac{\partial E}{z_l} \tag{1.16}$$

$$\frac{\partial L}{\partial z_j} = y_j(1-y_j) \sum_{k \in H2} w_{jk} \frac{\partial E}{\partial z_k} \tag{1.17}$$

where $k \in H2$ and $j \in H1$. Once the gradients are calculated, we can update the weights by applying an optimization algorithm, such as gradient descent, which adjusts the weights in a direction that minimizes the loss function. This process involves subtracting a fraction of the gradient from the current weights, and iteratively refining them to improve the model's

performance. The below equations represent this process, and η is the learning rate.

$$w_{kl} = w_{kl} - \eta y_k \frac{\partial E}{\partial z_l} \tag{1.18}$$

$$w_{jk} = w_{jk} - \eta y_j \frac{\partial E}{\partial z_k} \tag{1.19}$$

$$w_{ij} = w_{ij} - \eta x_i \frac{\partial E}{\partial z_j} \tag{1.20}$$

1.0.4.3 Optimizers

Optimizers are algorithms or methods used to adjust the weights of the neural network to minimize the loss function. In the previous section, we have discussed the gradient descent method to update the parameters. Different optimizers use different strategies to update the weights, each with its strengths and weaknesses. Here, we mainly introduce three other common optimizers.

Stochastic Gradient Descent (SGD) is a foundational optimization algorithm commonly used in training machine learning models, particularly neural networks [21]. Unlike traditional gradient descent, which computes the gradient of the loss function with respect to all training examples, SGD updates model parameters using the gradient computed from a single randomly selected training example. This makes SGD much faster and more suitable for large datasets. Despite its simplicity, SGD can sometimes struggle with noisy gradients and slow convergence, which can be mitigated by using techniques such as learning rate schedules and momentum [7].

Adaptive Moment Estimation) (Adam) computes adaptive learning rates for each parameter by maintaining running averages of both the gradients and their squares [37]. These running averages are then used to adjust the learning rate for each parameter individually. This results in faster convergence and more efficient training, particularly for problems involving large and sparse datasets. Adam has become one of the most widely used optimization algorithms due to its robustness and effectiveness.

Limited-memory Broyden–Fletcher–Goldfarb–Shanno (L-BFGS) is a quasi-Newton optimization algorithm particularly well-suited for large-scale optimization problems [44]. Unlike SGD and Adam, which are first-order methods using only gradient information, L-BFGS approximates the inverse Hessian matrix to guide the search for the minimum of the loss function

more effectively. This approximation reduces the computational and memory requirements compared to full Newton's methods, making L-BFGS feasible for large datasets [51]. It is particularly useful for optimization problems where the cost of evaluating the Hessian is prohibitive, and it often converges faster than first-order methods.

1.0.5 Special Types of Neural Network

1.0.5.1 Physics-Informed Neural Network

The Physics-Informed Neural Network (PINN) is a data-driven algorithm to approximate the solution of differential equations and identify parameters [60]. It could utilize any type of neural network architecture, like the FNN and Convolutional Neural Networks (CNN), as the main framework. The applied activation functions and optimization methods in PINN are the same as the usual deep learning techniques. The fascinating part of this algorithm is the loss function, which is generally comprised of boundary conditions, initial values, and physical constraints.

The outputs of the neural network are constrained to satisfy the system of differential equations by penalizing the residuals of differential equations into the loss function. In these residual equations, the derivatives of outputs with respect to time are computed by a black box, which is automatic differentiation [47]. Automatic differentiation is employed to train these models and allows PINN to take derivatives with respect to input coordinates so that it can discover the latent physical laws. PINNs are capable of obtaining good approximation accuracy given sufficient data points and an expressive neural network architecture when the given differential equation is well-posed and has a unique solution [60]. Shin et al. [72] have explained why the output of PINNs converges to the solution of differential equations.

1.0.5.2 Encoder-Decoder

The encoder-decoder architecture is a neural network design pattern primarily used for sequence-to-sequence tasks, where the input and output are sequences of different lengths [76]. This architecture is particularly prevalent in natural language processing tasks such as machine translation.

The primary function of the encoder is to analyze the input sequence and condense the

information into a context vector of a predetermined size, commonly known as the ”thought vector.” This is accomplished by a sequence of concealed layers that convert the input sequence into a latent representation. The decoder utilizes the context vector produced by the encoder to build the output sequence. The decoder, similar to the encoder, is commonly built using RNNs, LSTMs, or GRUs. The decoder sequentially produces the output, generating one element at a time. It utilizes its previous output as a component of the input to generate the subsequent element [11].

CHAPTER 2

Identification and prediction of time-varying parameters of COVID-19 model: a data-driven deep learning approach

Data-driven deep learning provides efficient algorithms for parameter identification of epidemiology models. Unlike the constant parameters, the complexity of identifying time-varying parameters is largely increased. In this paper, a variant of physics-informed neural network is adopted to identify the time-varying parameters of the Susceptible-Infectious-Recovered-Deceased model for the spread of COVID-19 by fitting daily reported cases. The learned parameters are verified by utilizing an ordinary differential equation solver to compute the corresponding solutions of this compartmental model. The effective reproduction number based on these parameters is calculated. Long Short-Term Memory neural network is employed to predict the future weekly time-varying parameters. The numerical simulations demonstrate that PINN combined with LSTM yields accurate and effective results.

2.1 Introduction

The novel SARS-CoV-2 (severe acute respiratory syndrome coronavirus 2) has spread all over the world since its discovery at the end of 2019, with millions of confirmed cases and more than a million deaths [87]. On March 11, 2020, COVID-19 was characterized as a pandemic by the World Health Organization (WHO) [86]. Due to the huge negative effects of COVID-19, precautionary measures have been aggressively carried out worldwide, such as facial masking, contact tracing, social distancing, and some governmental actions such as lockdowns. Hence, it is significant to analyze the dynamics of COVID-19 so that the effectiveness of those implemented measures can be verified.

Epidemiological models provide an efficient tool for determining and explaining the dynamics of disease transmission. In the early stage, one of the classical models is the Susceptible-Infectious-Recovery (SIR) model presented by Kermack and McKendrick in 1927 [36]. This compartmental model computes the theoretical number of susceptible people who became infected by the disease and then recovered. Based on the SIR model, many models have been

proposed for different diseases, like SARS and COVID-19 [71, 26, 81]. The Susceptible Exposed Infectious Remove (SEIR) model is proposed to analyze the effect of the precautionary measures on the dynamics of the epidemic [8]. Taking quarantine into account, a susceptible-exposed-infected-quarantined-recovered (SEIQR) model is investigated analytically and numerically [58]. The SEIR model has been extended by two extra classes of populations namely, 'C', which is the number of cumulative cases, and 'D' that is the number of severe, critical, and deceased cases, respectively, to understand the trends of COVID-19 outbreak in Wuhan, China [43].

The Susceptible Infectious Recovered Deceased (SIRD) model is governed by three parameters β, γ, and μ. Commonly, these parameters are assumed as independent of time during the epidemic. However, the interventions and measures implemented by governments to control the spread of COVID-19 result in significantly varying parameters with time. Thus, if parameters are still considered as constants, the epidemiology model cannot discover the varied dynamic of this disease transmission that is affected by those efficient precautions. Regarding parameters as time-series enables the model to react to the changes in real situations.

Recently, data-driven algorithms were developed for the determination of such time-varying parameters. Raissi et al. [63] proposed PINN to approximate parameters, which are constants, of the SIR model using a small data set. Furthermore, Wang et al. [80] divided the period time into four phases so that their parameters become piece-wise constants. Also, there are some studies relating to the time-dependent parameter, like a time-dependent SIR model, is proposed to track their parameter time-series [10]. Similarly, the time-dependent parameters of the SIR model were identified by utilizing a neural network to analyze the COVID-19 spread in South Korea [32].

Predicting the trend of COVID-19 transmission is a significant and arduous topic. A model of COVID-19 in China was developed to predict the cumulative number of reported cases [45]. By averaging the slope of approximated parameters of the SIRD model over the last several days, the extrapolated trends of infected, recovered, deceased, and susceptible cases were obtained via solving the model with estimated parameters [48]. An artificial intelligence method was used to predict the pandemic by training on the 2003 SARS data [88]. Recently, several

forecasting models including autoregressive integrated moving average, support vector regression, LSTM, and bidirectional LSTM (BiLSTM) were proposed to predict the confirmed cases, deaths, and recoveries [70]. Similarly, Recurrent Neural Network (RNN), LSTM, BiLSTM, Gated recurrent units (GRUs), and Variational AutoEncoder (VAE) algorithms were employed to forecast the number of infected and recovered cases [89].

Based on data collected, the SIRD model is used in this study to simulate the dynamics of the disease. Parameters of this model are regarded as daily and weekly time-varying. As for daily time-varying parameters, it represents that parameters would be varied from each day so that they are more similar to real situations. However, the difficulty of parameter identification increases with the smaller time interval by utilizing some general regression methods such as least square methods. PINN is still able to approximate these time-series parameters with some changes in architecture. Then, they are verified by being substituted into the SIRD model, and then a numerical algorithm is used to solve this Ordinary Differential Equation (ODE) system.

When it comes to weekly time-series, we divide the collected data week by week and employ the interpolation method for each week to gain more data so that the neural network has enough information to learn. The neural network implemented here is the PINN, which can estimate parameters to fit the collected data [60]. When the identification of these weekly time-series parameters is validated, LSTM is implemented to predict future parameters. In this way, by examining weekly values instead of daily ones, LSTM can still accurately predict future parameters. Another reason to predict the parameters rather than forecast the infectious cases directly is that we could obtain corresponding reproduction number R_0. It is an essential threshold to reflect the disease transmission speed and verify the effectiveness of these measures in curbing the outbreak of COVID-19.

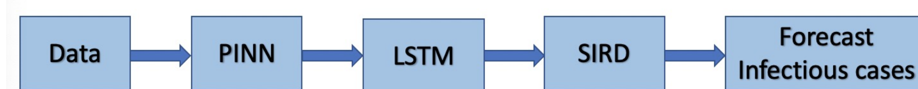


Figure 2.1: Flow Chart of Data-Driven Algorithm

In Fig. 2.1, we describe that PINN approximates the parameters of the SIRD model by training collected data. Then, LSTM is proposed to predict the future value of parameters.

After that, the obtained parameters are substituted into the SIRD model so that we can predict the infectious cases.

In Section 2.2 we introduce the SIRD model and explain model parameters (β, γ, and μ). Following that, we introduce the effective reproduction number and how to compute it. Deep learning neural networks, including FNN, PINN, and LSTM, are described briefly in Section 3. In Section 4, the collected data is described, and the loss function of PINN is formulated. After that, we present LSTM neural networks for the future prediction of parameters. The simulation results are discussed in Section 5.

2.2 Epidemiology Model

In this section, we introduce the SIRD model employed in our study. One promise of the SIRD model is that natural birth and death rates are neglected or equivalent so that the total population is considered as constant. Then, the population could be divided into four mutually exclusive groups, which are susceptible, infected, recovered, and deceased, respectively. Also, the reproduction number is introduced in this section.

2.2.1 SIRD Model

We consider the SIRD model described by the following system of ordinary differential equations [20]:

$$\begin{aligned} \dot{S}(t) &= -\frac{\beta S(t)I(t)}{N} \\ \dot{I}(t) &= \frac{\beta S(t)I(t)}{N} - (\gamma+\mu)I(t) \\ \dot{R}(t) &= \gamma I(t) \\ \dot{D}(t) &= \mu I(t), \end{aligned} \tag{2.1}$$

along with non-negative initial conditions:

$$S(0) = S_0, I(0) = I_0, R(0) = R_0, D(0) = D_0.$$

where $S(t)$, $I(t)$, $R(t)$, and $D(t)$ are the numbers of susceptible, infected, recovered, and deceased individuals, respectively, and N is the total population, $N = S(t) + I(t) + R(t) + D(t)$. For the existence and uniqueness of the solution of Eq. 2.1, see [75] and reference therein. Without loss of generality, N is a large number and difficult to compute in some optimization problems. Thus, we use the fraction of the population in each compartment, which means that $S(t)$, $I(t)$, $R(t)$, and $D(t)$ are divided by N. In this way, our calculation is simpler, and we can still adhere to the same dynamic system of the epidemic.

The parameter β represents the number of contacts each day for infected individuals. Besides, there are two essential assumptions. First, contacts between the infected and uninfected people are sufficient to spread the disease. Second, the population is mixed homogeneously. Thereby, $\beta S(t)$ susceptible people are infected by a patient each day on average, and $\beta S(t)I(t)$ is the number of newly infected people, γ is the rate at which infected cases become recovered. Then, there are $\gamma I(t)$ infected individuals turned into the third compartment. Similarly, μ is the rate of mortality by disease, and $\mu I(t)$ infected individuals are deceased.

2.2.2 Reproduction Number

There is an essential threshold used to get a better understanding of the transmission rate. This threshold is characterized by the reproduction number, which is an estimate of newly infected cases caused by an infected individual [78]. Assuming there is just one person infected in the beginning, then in the SIRD model, $I_0 = 1$, $S_0 = N - 1$, and no recovery at this moment. When we utilized the fraction for each compartment, $I_0 \approx 0$ and $S_0 \approx 1$. By the second equation of Eq. (2.1), we have:

$$\dot{I}(t) = I(t)[\beta S(t) - (\gamma + \mu)], \tag{2.2}$$

where $\beta(t)$, γ, and μ are greater than 0. In Eq.(2.2), it is simple to obtain $\dot{I} > 0$ when $\beta(t)S(t) > (\gamma + \mu)$, which means the number of infected cases keeps increasing.

A key idea about the reproduction number R_0 is that it is the gauge of the secondary infections at the beginning of this disease invasion [28]. After the disease invasion, the contact rate β decreases because of the reduction of this susceptible group. Those who are infected or have already recovered cannot be infected once again. Thereby, another similar threshold,

which is the effective reproduction number R_e [65], is used to indicate whether the disease keeps spreading, and R_0 is the upper bound of R_e [35].

$$R_e(t) = \frac{\beta S(t)}{\gamma+\mu} < \frac{\beta}{\gamma+\mu} = R_0. \tag{2.3}$$

2.3 Main Architecture

In this section, we provide a brief description of the deep learning neural network architectures utilized in our algorithm for parameter identification and disease dynamics prediction. The feedforward neural network (FNN) is an artificial neural network with a simple architecture in which the connections between the nodes do not form a cycle. FNN is utilized to approximate the target function by applying several activation functions to input recursively and then output a value or vector that is close to the target values.

The forward propagation from one layer to the next layer's nodes is given as follows [24].

$$z_j^{l+1} = \sum_i^{n_l} w_{ij}^l f_{l-1}(z_i^l) + b^l, \tag{2.4}$$

where n_l represents the number of neurons in the lth layer, z_i^l, $i = 1, \cdots, n_l$, denotes the output of the ith node in $(l-1)^{th}$ layer, f represents the activation function, w_{ij}^l are the weights between node i^{th} and node j^{th}, b^l are the bias in the lth layer. The activation functions employed in this study are the hyperbolic tangent function

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}, \tag{2.5}$$

and sigmoid function

$$\sigma(x) = \frac{1}{1+e^{-x}}. \tag{2.6}$$

When output values are obtained from the output layer, they are used to build a loss function, which is used to estimate the error of the model. An optimizer, like Adam [37] or gradient descent method [3], is employed to minimize the loss function so that the output is close to the observations. According to different problems, the selections of optimizer and loss function are different. Then, any continuous function can be well approximated by a neural network with a

sigmoid activation function and one hidden layer under some regularity conditions [15].

2.4 Data and Algorithms

In this section, we describe the collected data and introduce our algorithms for parameter identification and making predictions. In our work, the parameters of the SIRD model are viewed as daily-varying and weekly-varying parameters to simulate the complex real situation of COVID-19. Based on this perspective, two algorithms are employed to identify daily and weekly time-series parameters. Having the identified parameters by PINN as an input, we introduce the LSTM algorithm for making predictions.

2.4.1 Data

The data considered in our study is downloaded from https://covidtracking.com/data/download for New York, New Mexico, and Texas states. It is comprised of commutative infected, recovered, and deceased cases for the period from March 30, 2020, to September 30, 2020. A plot of the data is shown in Fig. 2.2.

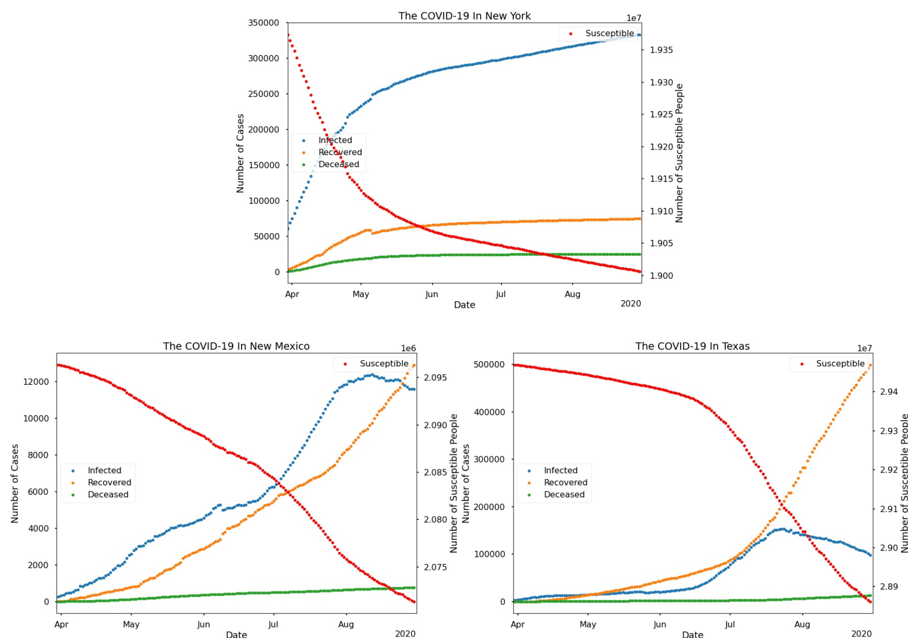


Figure 2.2: COVID-19 data for New York, New Mexico, and Texas

As can be observed, the three states have different dynamics between the compartments. In New York, while the number of cases in I, R, and D compartments is increasing, the number of recovered and deceased cases is relatively small compared to infections. New Mexico data

shows that the infected and recovered cases kept almost the same increasing rate before August but then the infections started decreasing slowly, while the number of recovered individuals kept increasing at a high rate. As for Texas, the infected population started growing fast from the second half of June, and then went down after about 30 days.

2.4.2 Parameter Identification Algorithm

In this subsection, we describe how to identify the parameters of the SIRD model by utilizing PINN.

2.4.2.1 PINN Architecture for SIRD Model

The utilized PINN for learning the parameters is composed of an FNN architecture. This architecture consists of 4 hidden layers and 60 neurons in each layer. The activation functions of hidden layers and the output layer are the hyperbolic tangent function, and sigmoid function, respectively.

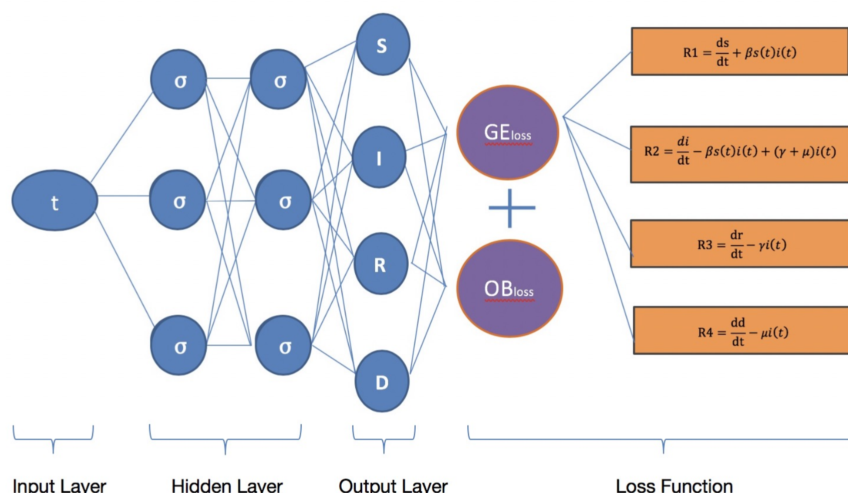


Figure 2.3: Physics Informed Neural Network

Fig. 2.3 shows the basic framework, which is FNN, and the loss function of PINN in our study. The loss function of PINN has two parts, namely GE_{loss} and OB_{loss} in this figure. The GE_{loss} represents the residuals that are obtained from the SIRD model Eq. (2.1) by subtracting the right side from the left side:

$$
\begin{aligned}
R_1 &= \frac{dS}{dt} + \beta(t)S(t)I(t) \\
R_2 &= \frac{dI}{dt} - \beta(t)S(t)I(t) + (\gamma(t) + \mu(t))I(t) \\
R_3 &= \frac{dR}{dt} - \gamma(t)I(t) \\
R_4 &= \frac{dD}{dt} - \mu(t)I(t).
\end{aligned} \tag{2.7}
$$

The OB_{loss} is the mean squared error between the outputs of the neural network and data. We employ Adam algorithm, which is the first-order gradient-based optimization of stochastic objective functions [37], to update the parameters of the neural network by minimizing the loss function.

2.4.2.2 Parameters Identification Algorithm

Algorithm 1 describes how to use PINN to identify the daily time-varying parameters. The input is time t, and outputs include the three parameters and the four compartments of the SIRD model. The weights w and bias b are initialized randomly.

Algorithm 1 PINN identify daily time-varying parameters

Data: S,I,R,D,t.

Initialize weights w**, bias** b

for number of epochs **do**

Each compartment of SIRD is obtained from forwarding propagation of the neural network

$$\hat{S},\hat{I},\hat{R},\hat{D},\beta,\gamma,\mu = NN(t) \tag{2.8}$$

Build the loss function components, where N_{ob} is the number of collected data of each compartment, and N_f is the number of collocation points

$$OB_{loss} = \frac{1}{N_{ob}}\sum_{i=1}^{N_{ob}}\left(\left|\hat{S}^i - S^i\right| + \left|\hat{I}^i - I^i\right| + \left|\hat{R}^i - R^i\right| + \left|\hat{D}^i - D^i\right|\right)$$

OB_{loss} denotes the difference between the output of the neural network and observation data

$$GE_{loss} = \frac{1}{N_t}\sum_{i=1}^{N_t}\left(\left|\frac{d\hat{S}^i}{dt^i} + \beta\hat{S}^i\hat{I}^i\right| + \left|\frac{d\hat{I}^i}{dt^i} - \beta\hat{S}^i\hat{I}^i + (\gamma + \mu\hat{I}^i)\right| + \left|\frac{d\hat{R}^i}{dt^i} - \gamma\hat{I}^i\right| + \left|\frac{d\hat{D}^i}{dt^i} - \mu\hat{D}^i\right|\right)$$

GE_{loss} represents the SIRD model residual loss, then we have the total loss

$$loss = OB_{loss} + GE_{loss}$$

Using the Adam optimizer to update the weights and bias by minimizing the loss function.

end for

Return $\hat{S}$, $\hat{I}$, $\hat{R}$, $\hat{D}$, β, γ, μ.

The approach of identifying the weekly-varying parameters is described in algorithm 2. We divide the utilized data of algorithm 1 into weekly intervals, and then they are trained to identify corresponding parameters. Cubic spline interpolation is employed to obtain an adequate

training data set for the neural network training.

Algorithm 2 PINN identify weekly time-varying parameters

Data: S, I, R, D, t.
Form the weekly data
for each week **do**
  $S, I, R, D, t = CubicSpline(S, I, R, D, t)$;
  $N = length(t)$ *Each dataset has the same length*;
  Initialize weights w, bias b, parameters β, γ, μ;
  for number of epochs **do**
    Each compartment of SIRD is obtained from forwarding propagation of the neural network

$$\hat{S}, \hat{I}, \hat{R}, \hat{D} = NN(t)$$

    Build the loss function components.

$$OB_{loss} = \frac{1}{N} \sum_{i=1}^{N} \left(\left|\hat{S}^i - S^i\right|^2 + \left|\hat{I}^i - I^i\right|^2 + \left|\hat{R}^i - R^i\right|^2 + \left|\hat{D}^i - D^i\right|^2 \right)$$

    OB_{loss} denotes the difference between the output of the neural network and observation data

$$GE_{loss} = \frac{1}{N} \sum_{i=1}^{N} \left(\left|\frac{d\hat{S}^i}{dt^i} + \beta \hat{S}^i \hat{I}^i\right|^2 + \left|\frac{d\hat{I}^i}{dt^i} - \beta \hat{S}^i \hat{I}^i + (\gamma + \mu \hat{I}^i)\right|^2 + \left|\frac{d\hat{R}^i}{dt^i} - \gamma \hat{I}^i\right|^2 + \left|\frac{d\hat{D}^i}{dt^i} - \mu \hat{D}^i\right|^2 \right)$$

    GE_{loss} represents the SIRD model residual loss

$$loss = OB_{loss} + GE_{loss}$$

    Using the Adam optimizer to update the weights and bias by minimizing the loss function;
  end for
  save the value of β γ and μ;
end for
Return $\hat{S}$, $\hat{I}$, $\hat{R}$, $\hat{D}$, β, γ, μ.

2.4.2.3 LSTM for Prediction of Infectious Cases

Having the learned parameters by PINN, the LSTM is employed to predict the future parameters by applying Keras as depicted in Fig.2.4. At first, we normalize these parameters and create new datasets with multiple inputs and one output. Inputs are the previous three-time step parameters, and output, which is the prediction of these inputs, is the next one-time step parameter. Then, the data is used to train the LSTM neural network. We utilize three hidden layers of 80 nodes each to predict the parameters of the next four weeks.

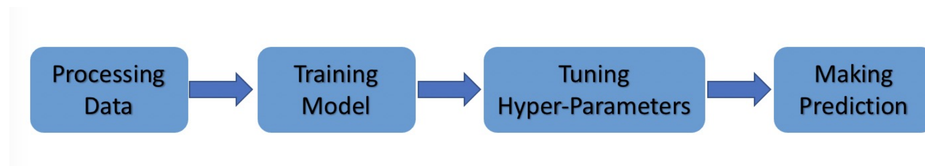


Figure 2.4: Flow-Chart of LSTM

2.5 Simulation

In this section, we present and discuss our simulation results. The accuracy of the learned parameters by PINN is validated using the relative error in the numerical solution of System Eq.(2.1), obtained using the Fourth Order Runge-Kutta (RK4) method, with respect to the exact solution. Furthermore, the relative error in the learned solution by PINN with respect to the data is examined. We start by presenting the results for the learned daily and weekly time-varying parameters followed by the associated reproduction numbers. Then, predictions of infectious cases are provided.

2.5.1 Time-Varying Parameters

In general, the daily-varying values of the parameters are more reflective of the daily data than the single constant average values of these parameters. However, identifying the daily-varying parameters is traditionally costly and adds to the complexity of the problem. Fortunately, PINN provides an efficient approach to overcome this difficulty. On the other hand, our simulations show that the learned weekly-varying data produce precisely identified parameters and more stable approximation results.

Algorithms 1 and 2 give the identified parameters and the learned values of the SIRD model by PINN. We use the RK4 method to solve the SIRD model with learned parameters. Then, the solutions of the ODE system Eq.(2.1) are compared with the learned values.

In Fig. 2.5, 2.6, and 2.7, we present the learned values by PINN, the solution of the ODE system, and the reported data. Moreover, the relative errors with respect to the reported data are also computed.

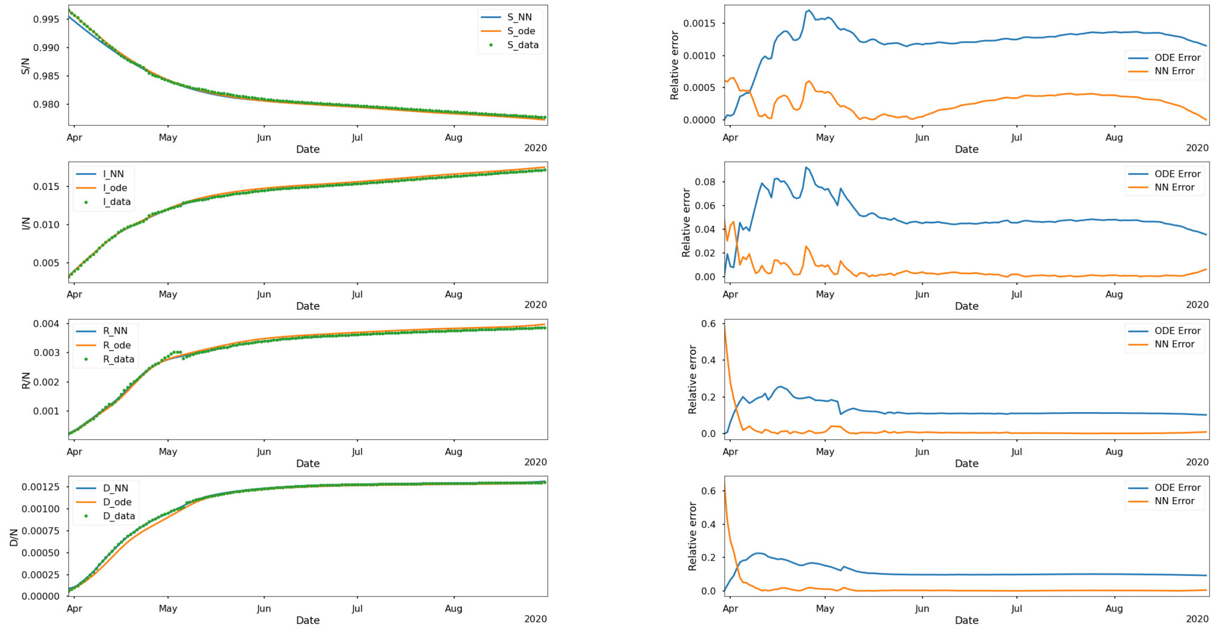


Figure 2.5: Simulation Results of New York of Daily Time-Varying Cases

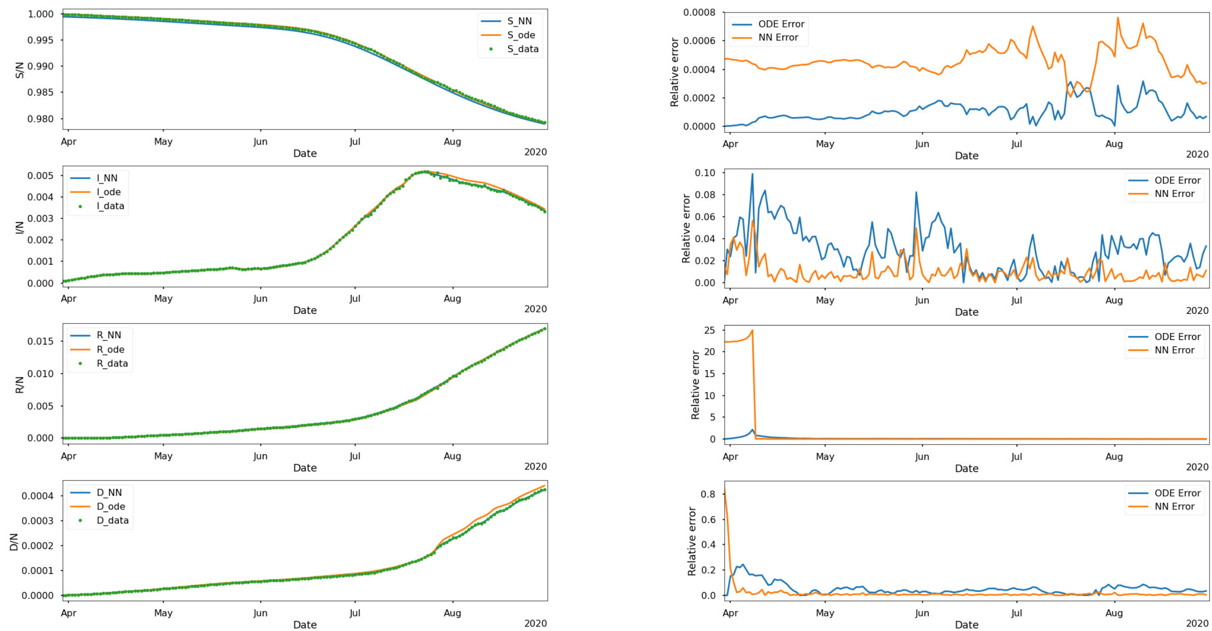


Figure 2.6: Simulation Results of Texas of Daily Time-Varying Cases

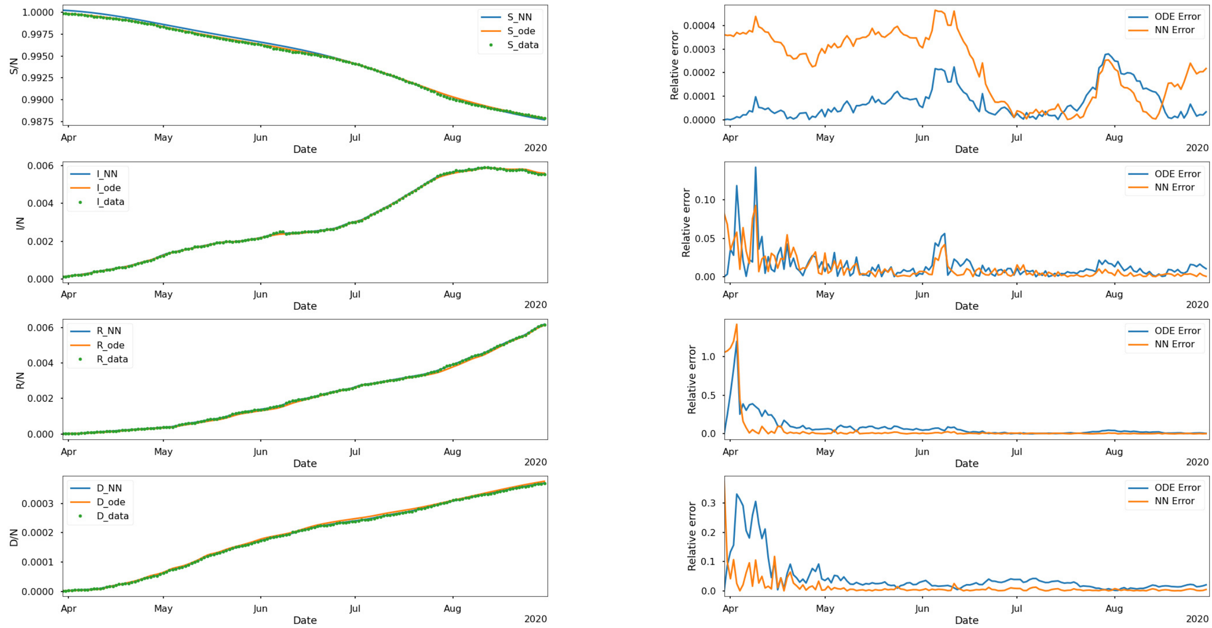


Figure 2.7: Simulation Results of New Mexico of Daily Time-Varying Cases

There are some large relative errors at the early stage of the outbreak of the disease. This could be due to the small size of the recovered and deceased populations. However the relative errors of the solution of the SIRD model are in a reasonable range, which means that we obtained ideal identified parameters. For the weekly-varying parameters, the period is divided into weeks and the learned results are presented in Figures 2.8, 2.9, and 2.10.

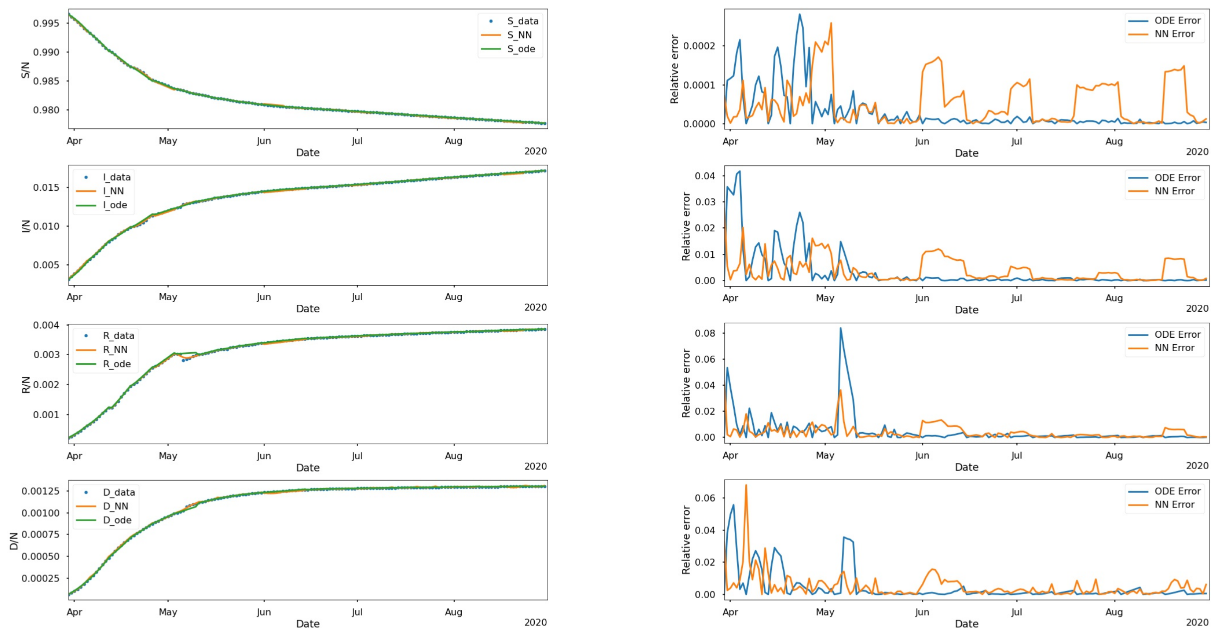


Figure 2.8: Simulation Results of New York of Weekly Time-Varying Cases

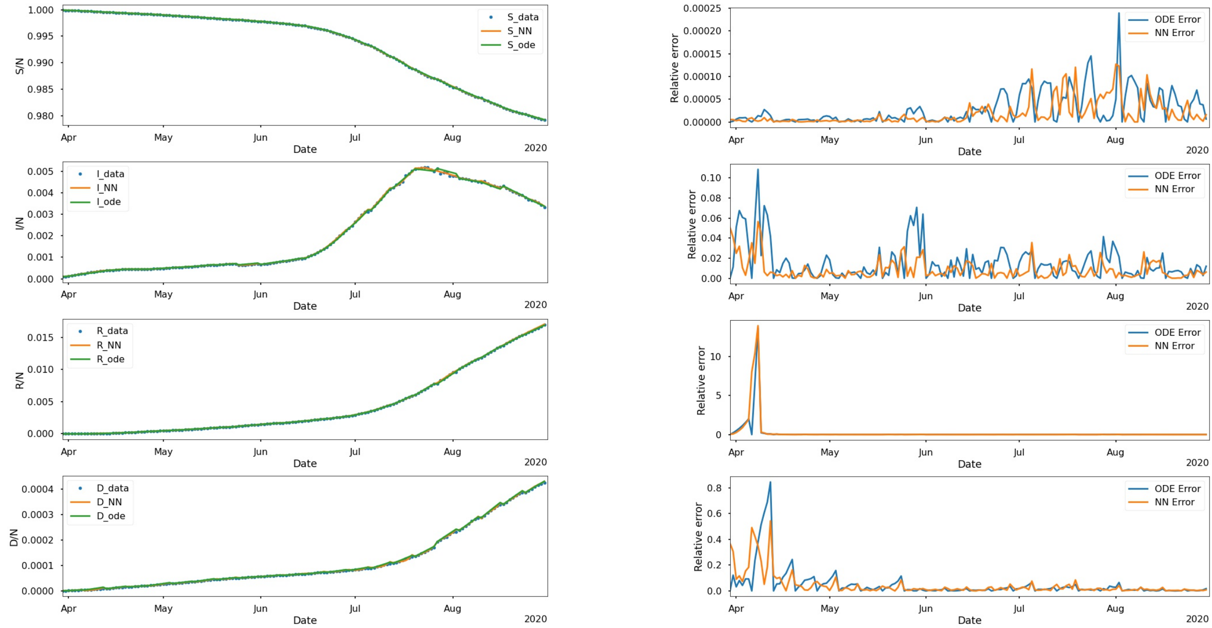


Figure 2.9: Simulation Results of Texas of Weekly Time-Varying Cases

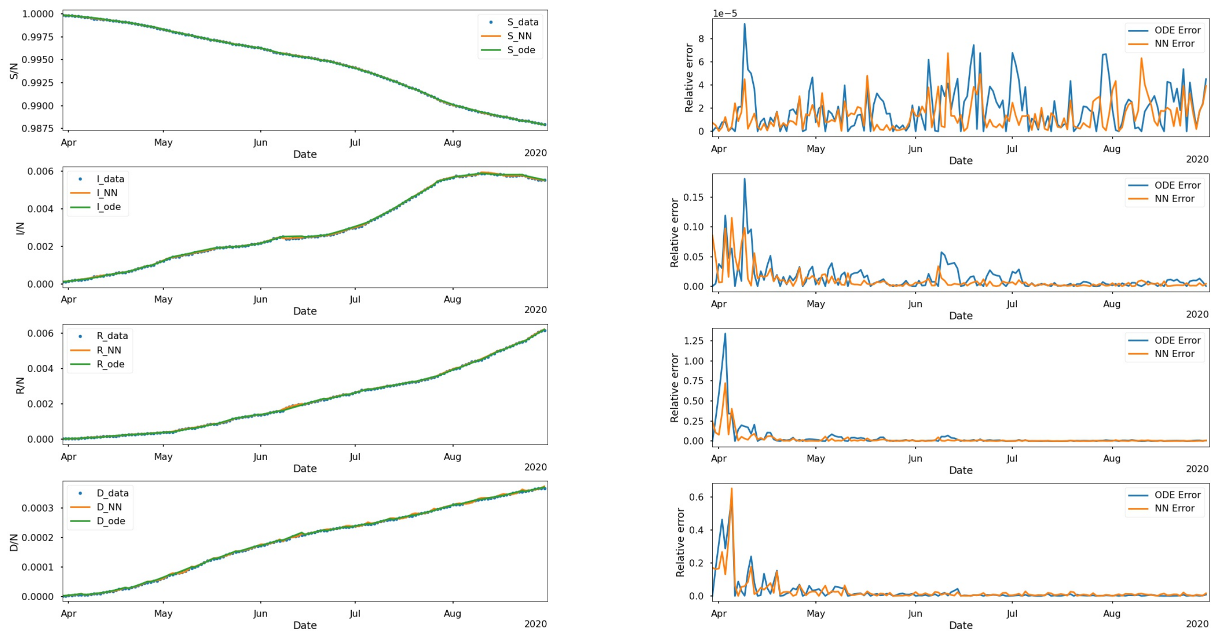


Figure 2.10: Simulation Results of New Mexico of Weekly Time-Varying Cases

The relative errors in Figures 2.8, 2.9, and 2.10 are smaller compared to the results in Figures 2.5, 2.6, and 2.7. Thus, we could infer that the approximation of constant parameters is more accurate and stable.

The smaller errors in these figures show that the parameters are identified accurately by algorithms 1 and 2. Since we utilize the cubic spline to obtain sufficient data, then PINNs can

well identify the parameters and learn the solutions. Computational resources, such as many epochs, layers, and neurons, are utilized to obtain small errors.

2.5.2 Computational Complexity

Table 2.1 shows the change in relative error of the learned number of infectious cases and average time (seconds) needed for each week in these three states as the number of epochs increases. Here, we utilize 3 hidden layers and 64 neurons. In this table, errors are basically decreasing as the number of epochs becomes bigger.

Epochs	Error (NY)	CPU	Error (NM)	CPU	Error (TX)	CPU
10000	$6.36e-01$	62	$7.02e-01$	44	$6.84e-01$	55
20000	$2.15e-03$	117	$9.27e-01$	94	$1.22e-03$	110
30000	$2.64e-03$	177	$5.35e-03$	149	$8.39e-03$	165
40000	$2.70e-03$	246	$5.07e-03$	227	$6.83e-03$	204
50000	$1.97e-03$	314	$4.27e-03$	296	$6.33e-04$	258

Table 2.1: Computational complexity in terms of time for the infectious cases.

Table 2.2 shows relative errors between the daily number of infectious cases learned by PINN and real data [25] using different numbers of layers and neurons. In this case, the smaller errors are obtained with increasing depth and width of the neural network. Table 2.3 presents the CPU time required to train the neural network with different layers and neurons. From this table, we can see that the neural network with more layers and neurons requires more time.

Neurons\Layers	2	3	4	5
16	4.98e - 03	4.65e - 03	4.13e - 03	4.26e - 03
32	4.82e - 03	4.40e - 03	4.09e - 03	3.95e - 03
64	4.56e - 03	4.46e - 03	3.85e - 03	3.84e - 03

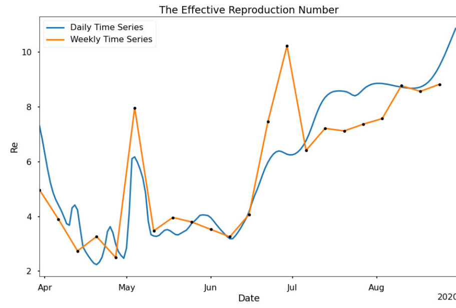
Table 2.2: The effect of different numbers of layers and neurons on the relative error between the learned infectious cases and collected data in New Mexico.

Neurons\Layers	2	3	4	5
16	138	151	164	192
32	179	209	245	279
64	265	227	323	354

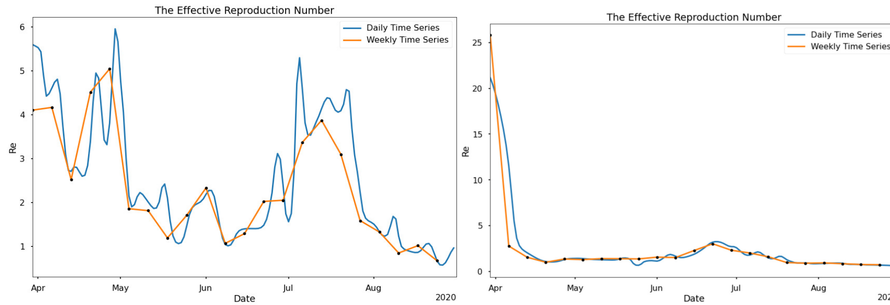
Table 2.3: The average CPU time is needed for each week in New Mexico

2.5.3 Simulated Reproduction Numbers

By Eq. (2.3), the effective reproduction number can be obtained using the learned daily and weekly varying parameters. Fig. 2.11 shows the effective reproduction number in the three states. We could see the daily and weekly time-varying reproduction numbers have similar trends, which proves the feasibility and effectiveness of our methods to some extent. In the result of New York, we could see the R_e is greater than 1 in the whole period, which represents that the number of infectious cases keeps increasing. As for the situation in New Mexico, R_e decreases to 1 from August, then the population of infected individuals decreased during that period. By observing the change of R_e in Texas, the value is too large in the beginning because the parameter identification in that period is not ideal. But after that period, the R_e goes back to normal and oscillates between 0 and 4. Besides, we can see that the growth rate of infected individuals in New York is faster than the other two states by Fig. 2.2.



(a) New York



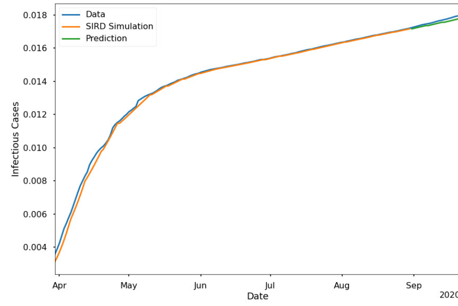
(b) New Mexico

(c) Texas

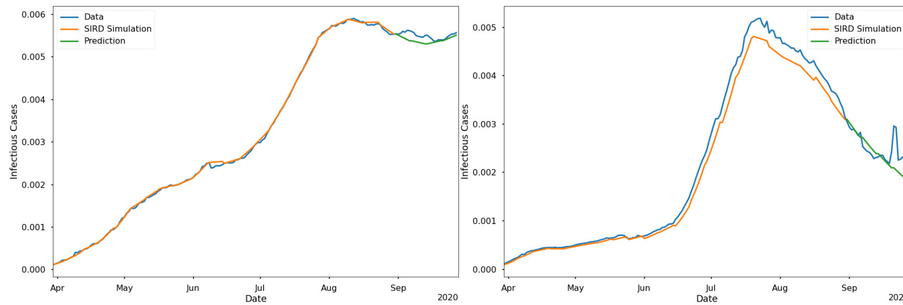
Figure 2.11: Effective Reproduction Number

2.5.4 Prediction of Infectious Cases

The LSTM is employed to predict the future values of the parameters. Fig. 2.12 presents the predictions of infected cases for the next four weeks. The prediction of New York is close to the original trend of infectious cases. Predictions in New Mexico show general accord with real situations, but it cannot capture the small fluctuations. As for Texas, there is a sharp increase and decrease in infectious cases at the end of September, and our prediction could not detect this situation.



(a) New York



(b) New Mexico (c) Texas

Figure 2.12: Predictions

2.6 Conclusions

We introduced a data-driven deep learning approach based on physics physics-informed neural network to solve the epidemiological models and identify weekly and daily time-varying parameters. The physics-informed neural network was used for parameter identification and solving the ordinary differential equation system. Long short term memory neural network was implemented to predict the weekly time-varying parameters and then substituted predicted parameters into the Susceptible-Infectious-Recovered-Deceased model so that we could obtain the future trend of COVID-19. The results and errors have shown that the weekly time-varying parameters provided a good fit to the real data. The algorithms could be further developed to achieve a more accurate approximation for complex problems.

In the future, we intend to explore other architectures of physics-informed neural networks. If the fully connected neural network is replaced by other frameworks such as residual network and convolution neural network, can the accuracy, trainability, and robustness of our algorithms be improved? Also, we plan to develop an adaptive algorithm by combining these two models so that the parameters are identified and predicted simultaneously.

CHAPTER 3

Sparse Data Regulation on GRU Neural Network

Deep learning algorithms have emerged as an alternative approach for solving partial differential equations. This research integrates a neural network with a numerical approach to address time-dependent partial differential equations with initial conditions and limited observed data from the domain. We use the Gated Recurrent Units network to estimate the time iteration schemes. We integrate the prior knowledge of governing equations into the iteration process. The numerical implicit approach is then applied to these iteration schemes to calculate the new time iteration schemes. The loss function incorporates the difference between these two iteration schemes. To attain the appropriate precision, the sparsely observed data, randomly selected from analytical or numerical solutions, is incorporated as the regulator. The algorithm's efficacy is proven through its application in many numerical experiments, such as Burgers' equation, Allen-Cahn equation, non-linear Schrodinger equation, and linked two-dimensional Burgers' equations.

3.1 Introduction

In a variety of scientific and technical domains, time-dependent partial differential equations (PDEs) depict the evolution of a dynamic event across time. Finding the analytical solutions for PDEs is challenging. Then, numerous different traditional numerical methods, like finite volume method [18], and finite element [16], are introduced to approximate the solutions. However, because of the infamous curse of dimensionality, these approaches are computationally expensive and difficult to solve for some complicated PDEs. To address these problems or find some surrogate models, plenty of recent research is developing deep learning approaches to estimate the solutions of PDEs. The universal approximation theorem states that a sufficiently large neural network can approximate any continuous function to any desired degree of accuracy [15].

Physics-informed neural networks (PINNs) have attracted researchers' attention to solving PDEs within the quickly expanding field of scientific machine learning [60]. By integrating

the mathematical model into the loss function, the PINNs confine the network's output via a set of PDEs. The concept of using previously arranged prior information about the solution is introduced before PINNs. The useful strategy of using such previous information was disclosed by Owhadi [53]. Raissi et al. [62, 61] utilize the Gaussian process regression to generate linear operator functional representations, efficiently infer solutions, and provide uncertainty estimates for various physical problems. Then, the availability of automatic differentiation [55, 5] and the significant advancement in computational power [1] allow for the implementation of this concept on PINNs.

The success of a neural network can hinge on its architecture. Various applications often necessitate distinct architectures. For instance, the Convolutional Neural Networks [69] are effective in handling image recognition while the Recurrent Neural Networks (RNNs) [67] is crucial for modeling sequential data. According to the universal approximation theorem [15], any continuous function can be arbitrarily closely approximated by a sufficiently large perception [31]. However, determining the proper parameters of networks to solve some complicated PDEs is difficult [6]. The selection of suitable architectures is essential for improving the performance. Ying et al. [42] approximate the iteration scheme by using the fully connected neural network. In this work, we mainly implement the RNNs in estimating time-dependent PDEs. The RNNs have the gradient vanishing problem hindering the model's ability to capture long-range dependencies [29]. To address this problem, two advanced variants, Long Short-Term Memory (LSTM) [30] and Gated Recurrent Unit (GRU) [14], are designed. Their gating mechanisms and improved memory cell structures make them better suited for a wide range of sequential data tasks.

In this study, we employ the Gated Recurrent Units (GRU) network to solve time-dependent partial differential equations characterized by sparse data. The neural network serves as an approximation for time iteration schemes, and we introduce the Adams-Moulton implicit method to guide convergence to solutions during network training. Integration of sparse data as a regulatory component enhances the model's accuracy significantly. The efficacy of this algorithm is demonstrated through numerous numerical experiments, encompassing scenarios such as Burgers, Allen-Cahn, non-linear Schrödinger equations, and coupled two-dimensional Burg-

ers' equations, validating its feasibility and performance across diverse applications.

3.2 Methodology

3.2.1 Problem setup

$$u_t = f\left(x, u, \frac{\partial u}{x}, \frac{\partial^2 u}{x^2}, \ldots, \frac{\partial^n u}{x^n}\right), x \in \Omega, t \in [0, T], \tag{3.1}$$

where $u = u(x,t)$ and f is a function that depends on x, t, u, and its partial derivatives with respect to x. We aim to propose a neural network to approximate the $u(x,t)$ with a few interior observations only [22].

3.2.2 Algorithm

In this section, we mainly describe our algorithm. The architecture is comprised of one layer of GRUs and several fully-connected layers. We utilize the finite difference to approximate the partial derivatives on Eq. 3.1 so we can discretize the domain into a $M \times N$ mesh grid, defined by $x_i, i = 0, 1, 2, \ldots, M-1$ and $t_j, j = 0, 1, 2, \ldots, N-1$, with step size Δx and Δt.

The flow chart of our approach is presented in Fig. 3.1. From the left side of this figure, we use the neural network, indicated as $\bar{u}_(x,t;\theta)$, to approximate $u(x,t)$. The number of neurons for each layer is N. The different neurons of the output layer will generate the approximations $\bar{u}_j(x;\theta)$ at $t_j, j = 0, 1, 2, \ldots, N-1$. The $\bar{u}_j(x;\theta)$ is the vector where $x = \{x_0, x_1, \ldots, x_{M-1}\}$. Then, we implement the central difference method to approximate the spatial derivatives [77],

$$\bar{u}_x(x_i, t_j; \theta) = \frac{1}{2\Delta x}(\bar{u}(x_{i+1}, t_j; \theta) - \bar{u}(x_{i-1}, t_j; \theta)), \tag{3.2}$$

$$\bar{u}_{xx}(x, t_j; \theta) = \frac{1}{\Delta x^2}(\bar{u}(x_{i+1}, t_j; \theta) - 2\bar{u}_{i,j} + \bar{u}(x_{i-1}, t_j; \theta)). \tag{3.3}$$

The right side of Eq. 3.1 $f(x, \bar{u}, \bar{u}_x, \bar{u}_{xx})$ is employed to approximate the new iteration scheme $\tilde{u}(x,t)$ by applying the Adams-Moulton Implicit methods [27]. The one-step implicit method or trapezoidal rule:

$$\tilde{u}^{j+1} = \bar{u}^j + \frac{\Delta t}{2}(f(\bar{u}^{j+1}) + f(\bar{u}^j)). \tag{3.4}$$

The loss function can be constructed as mean square errors between the $\bar{u}$ and $\tilde{u}$.

$$Loss_1 = \Sigma_{j=1}^{N-1}(\bar{u}_j - \tilde{u}_j)^2 \tag{3.5}$$

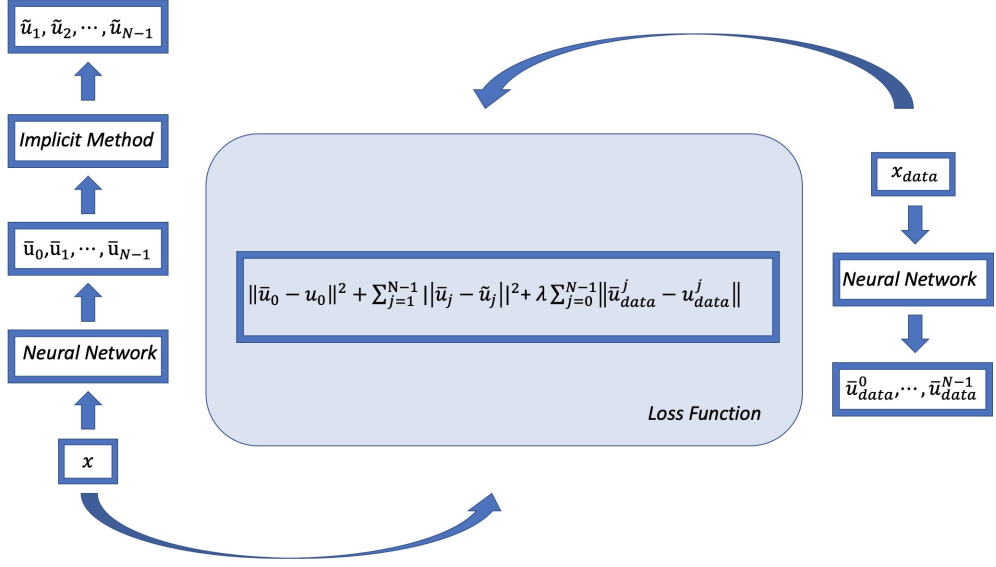


Figure 3.1: Flow Chart

As for the right side of Fig. 3.1, we randomly select some sparse observations represented by u_{data} at first. Correspondingly, the x_{data} is denoted as values at these selected spatial points. The $\bar{u}_0^{data}, \ldots, \bar{u}_{N-1}^{data}$ are the output of the neural network. Then, inspired by [84], we put the λ on the data term, which is a hyper-parameter, to force the neural network to satisfy the data points closely.

$$Loss_{data} = \lambda\Sigma_{i=0}^{N-1}(\bar{u}_j^{data} - u_j^{data})^2 \tag{3.6}$$

We also incorporate the initial condition into the loss function.

$$Loss_0 = (\bar{u}_0 - h(x))^2 \tag{3.7}$$

The total loss function is comprised of these three terms.

$$Loss = (\bar{u}_0 - h(x))^2 + \Sigma_{j=1}^{N-1}(\bar{u}_j - \tilde{u}_j)^2 + \lambda\Sigma_{j=0}^{N-1}(\bar{u}_j^{data} - u_j^{data})^2 \tag{3.8}$$

In Fig. 3.1, the neural networks on both sides share the same parameters so that the output at

the selected positions is fixed. Algorithm 3 describes the procedures mentioned above.

Algorithm 3 Sparse Data Regulation on GRU Neural Network

Input: x;x_{data}

Output: Predicted solutions of PDEs: $\bar{u}$.

1: Initialize the parameters of Neural Networks.

2: Obtain the outputs $\bar{u}_{data}$, $\bar{u}$.

3: Compute the loss at data points: $loss_{data} = (u_{data} - \bar{u}_{data})^2$.

4: Implement the central difference to calculate partial derivatives.

5: Compute $f_u = f(u, u_x, u_{xx}, \dots)$.

6: Calculate the loss: Set $loss = 0$

for i in range(0,T) **do**

  if $i = 0$ **then**

    $loss_0 = (u_0 - \bar{u}_0)^2$

  else if $i = 1$ **then**

    $\tilde{u}^i = \bar{u}^{i-1} + \frac{\Delta t}{2}(f_u^{i-1} + f_u^i)$

    $loss = loss + (\tilde{u}^i - \bar{u}^i)^2$

  else if $i = 2$ **then**

    $\tilde{u}^i = \bar{u}^{i-1} + \frac{h}{12}(5f_u^{i+1} + 8f_u^i - f_u^{i-1})$

    $loss = loss + (\tilde{u}^i - \bar{u}^i)^2)$

  else

    $\tilde{u}^i = \bar{u}^{i-1} + \frac{h}{24}(9f_u^{i+1} + 19f_u^i - 5f_u^{i-1} + f_u^{i-2})$

    $loss = loss + (\tilde{u}^i - \bar{u}^i)^2$

  end if

end for

7: Calculate the sum of loss: $L = loss_0 + loss + \lambda loss_{data}$

8: Apply the optimizer to minimize the L.

9: Get the prediction U.

3.3 Numerical Experiments

In this section, we present the results of Burgers' equation, Allen-Cahn equation, non-linear *Schrödinger*, Coupled Burgers', and Coupled Two-Dimensional Burgers' equations to validate the effectiveness of our method. We utilize the numerical method or analytical solution to generate the synthetic data and randomly pick sparse data from them. The entire spatio-temporal solutions of these equations are inferred by applying our method. The architecture comprises one layer of GRUs and some dense layers, and each cell or neuron of the output layer will give the prediction at $n\Delta t$, where n is a non-negative integer. Thus, each layer's number of cells or neurons depends on the Δt. We train the network by applying the Adam optimizer [37], and then followed by L-BFGS [44] to minimize the loss function. The idea behind this is that the Adam optimizer avoids the local minimum, and then the solutions can be refined by the L-BFGS optimizer [27]. We adopt the *tanh* function as the activation function for neural networks. We utilize the Learning Rate decay [13], which started with a relatively large learning rate and reduced it after a certain number of iterations. The algorithm 3 is implemented by utilizing the computing package PyTorch [56]. Fig. 3.2 presents the code for training the model by implementing these two optimizers for Burger's Equation.

```
model = GRU_Burgers(1,Nt).to(dev)
adam_optimizer = torch.optim.Adam(model.parameters(),lr = 0.005)
for i in range(4000):
    adam_optimizer.zero_grad()
    loss,data_loss,loss1=model.loss(x,u0,Nt,dt,dx,x_data,u0_data,data)
    loss.backward()
    adam_optimizer.step()
optimizer = torch.optim.LBFGS(model.parameters(), lr=0.005,
                              max_iter = 4000,
                              max_eval = None,
                              tolerance_grad = 1e-15,
                              tolerance_change = 1e-9,
                              history_size = 100,
                              line_search_fn = 'strong_wolfe')
optimizer.step(model.closure)
```

Figure 3.2: The Sample Code of Using Optimizers for Burgers' Equation

3.3.1 Burgers' equation

The viscous Burgers' equation is the fundamental nonlinear PDEs in fluid dynamics and shock waves. The Burgers' equation with periodic boundary condition is given by

$$\begin{aligned} &u_t + uu_x - (0.01/\pi)u_{xx} = 0, x \in [-1,1], t \in [0,1], \\ &u(0,x) = -sin(\pi x), \\ &u(t,-1) = u(t,1). \end{aligned} \tag{3.9}$$

We set the time-step $\Delta t = 0.01$ and obtain 10,000 uniformly spaced x values from -1 to 1. We utilize the neural network to approximate the solution and then apply the finite difference to approximate the $\bar{u}_x$ and $\bar{u}_{xx}$. By Eq. 3.9, we can obtain

$$f(u, u_x, u_{xx}) = -uu_x + (0.01/\pi)u_{xx}. \tag{3.10}$$

Then, we can apply the Adams-Moulton Two-Step implicit method to obtain:

$$\tilde{u}^n \approx \bar{u}^{n-1} + \frac{\Delta t}{12}(5f^n(\bar{u}^n, \bar{u}^n_x) + 8f^{n-1}(\bar{u}^{n-1}, \bar{u}^{n-1}_x) - f^{n-2}(\bar{u}^{n-2}, \bar{u}^{n-2}_x)) \tag{3.11}$$

The observations are obtained by solving the equation 3.3 numerically. Then, we randomly choose 10 data points for each time step, and the corresponding λ is set to 25 after several experiments. The network is comprised of one layer of GRUs and two dense layers. Then, we implement the algorithm 3 to construct the loss function to learn the network parameters, which is minimized by the 4,000 epochs of Adam and 4,000 epochs L-BFGS optimizers. The learning rate is set to 0.005.

$$loss = ||\bar{u}_0 - u_0||^2 + \sum_{i=1}^{N} ||\bar{u}_i - \tilde{u}_i||^2 + \lambda ||\bar{u}_{data} - u_{data}||^2. \tag{3.12}$$

Fig.3.3 presents the results at some t values and compares the prediction and exact solution. The blue data points in Fig. 3.3 are randomly selected as the regularization in the loss function. We can see that the prediction of our neural network fits the exact solution accurately. Fig. 3.4 shows the change of the errors generated from the loss function throughout iterations. Table 3.1 presents the relative L_2 errors, defined as Eq. (3.13), with different numbers of dense layers at different times. The last column illustrates that the relative errors become smaller when we put more hidden layers. From each row, we can see the errors are relatively smaller in small time regions.

$$L_2 = \sqrt{\frac{\sum_{i=1}^{N}(\bar{u}_i - u_i)^2}{\sum_{i=1}^{N}(u_i)^2}} \tag{3.13}$$

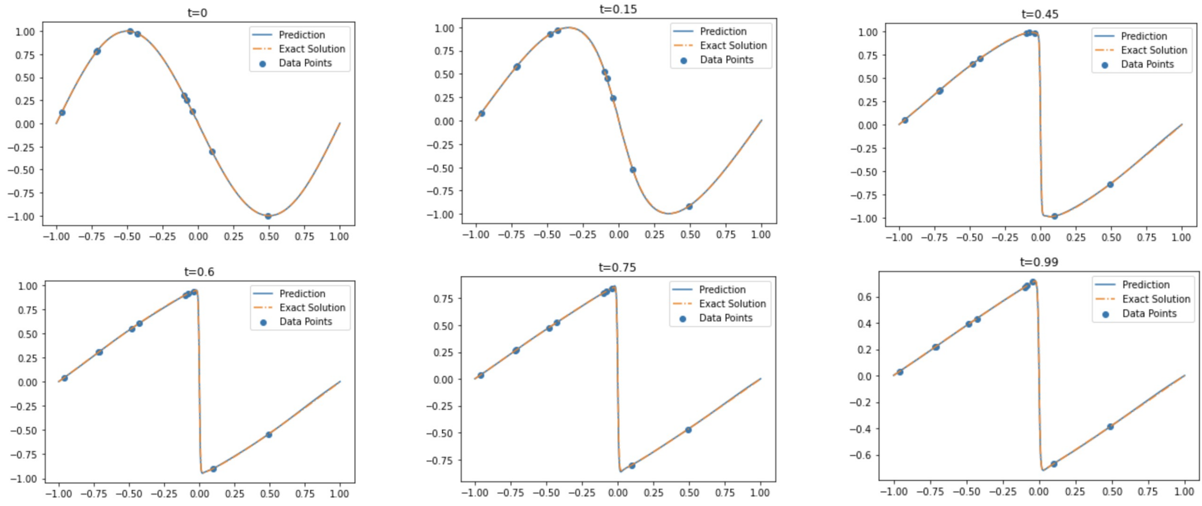


Figure 3.3: Burgers' Equation: Snapshots of the predicted solutions and exact solutions at $t = \{0.0, 0.15, 0.45, 0.60, 0.75, 0.99\}$.

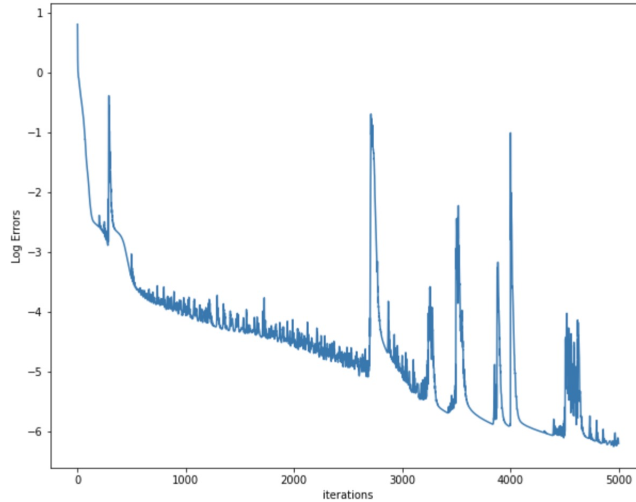


Figure 3.4: The variation of loss with iterations for Allen-Cahn equation

Layers \ *t*	0.25	0.5	0.75	1.0	All Time
2	1.35e-3	2.04e-3	2.92e-3	4.98e-3	3.68e-3
3	9.21e-4	2.15e-3	2.05e-3	4.51e-3	2.63e-3
4	7.25e-4	1.92e-3	1.04e-3	2.34e-3	1.02e-4

Table 3.1: Burgers' Equation: Relative L_2 error between predicted and exact solutions at different times with different numbers of layers

3.3.2 Allen-Cahn equation

To further validate the effectiveness of our method, we implement the algorithm 3 to solve the Allen-Cahn PDE, which is a reaction-diffusion equation. It describes the phase separation process in multi-component alloy systems, including order-disorder transitions. The Allen-Cahn PDE is given as follows:

$$
\begin{aligned}
&u_t - 0.0001u_{xx} + 5u^3 - 5u = 0, x \in [-1,1], t \in [0,1], \\
&u(x,0) = x^2 cos(\pi x), \\
&u(t,-1) = u(t,1).
\end{aligned}
\tag{3.14}
$$

We set the time-step $\Delta t = 0.005$ in this experiment and use the same x values. The numerical method is applied to solve the Eq. 3.14 to get observations. We also randomly pick 10 data points for each time step to force the network to fit in. Considering the smaller Δt in this case so that there are more collocation data points, we increase the λ to 40. The network is constructed by one GRUs layer and two dense layers, and it is trained for 5,000 Adam and 5,000 L-BFGS iterations. The learning rate is set to 0.001. Similarly, we can obtain:

$$
f(\bar{u}, \bar{u}_x, \bar{u}_{xx}) = 0.00001\bar{u}_{xx} - 5\bar{u}^3 + 5\bar{u}. \tag{3.15}
$$

The Adams-Moulton Two-Step implicit method is adopted to calculate $\tilde{u}$. Then, we have the loss function as follows:

$$
loss = ||\bar{u_0} - u_0||^2 + \sum_{i=1}^{N} ||\bar{u}_i - \tilde{u}_i||^2 + \lambda ||\bar{u}_{data} - u_{data}||^2. \tag{3.16}
$$

Fig. 3.5 illustrates that the network prediction is indistinguishable from the exact solution. The variation of the loss function is shown in Fig. 3.6, and the loss is rapidly decreasing in the beginning. Table 3.2 presents the relative L_2 errors at some time with different number of hidden layers. The errors take a minimum when three hidden layers are applied. It illustrates that accuracy is not necessary to increase with more hidden layers because the larger architectures are difficult to optimize.

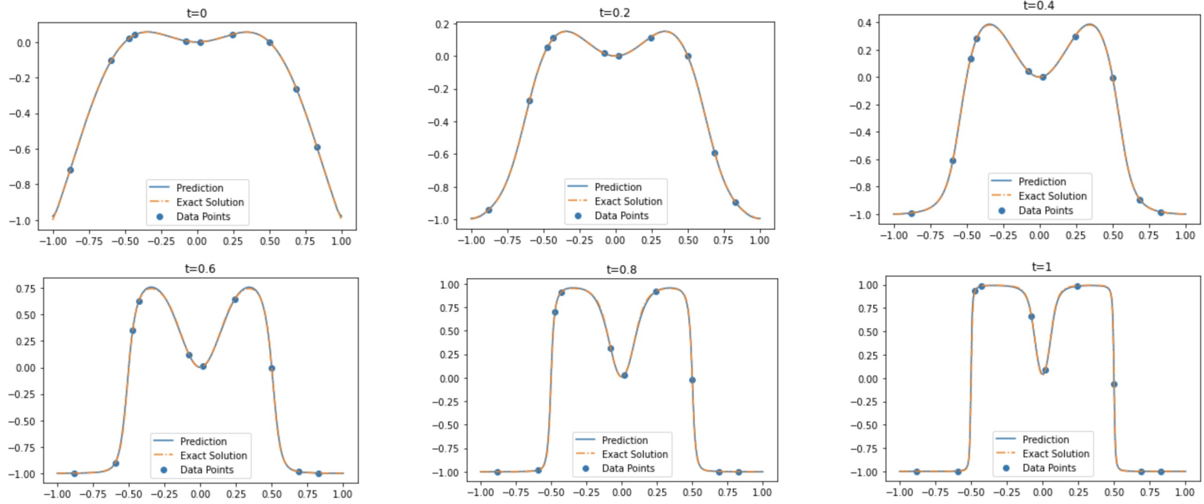


Figure 3.5: Allen-Cahn Equation: Snapshots of the predicted solutions and exact solutions at $t = (0.0, 0.2, 0.4, 0.6, 0.8, 1.0)$.

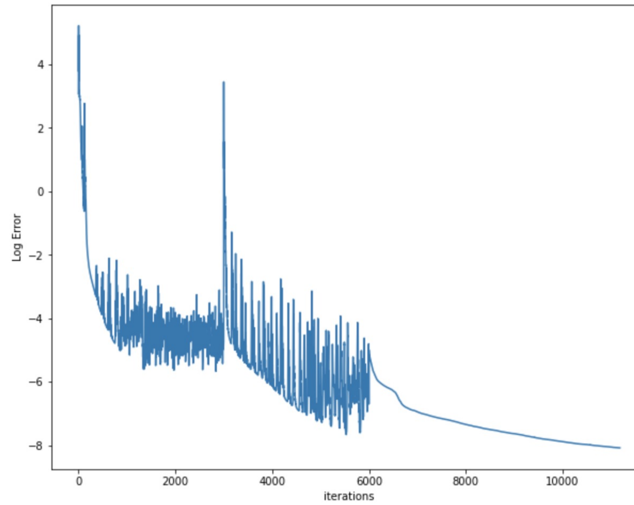


Figure 3.6: The variation of loss with iterations for Allen-Cahn equation

Layers \ t	0.25	0.5	0.75	1.0	All Time
2	2.32e-3	1.51e-3	5.13e-3	9.28e-3	6.78e-3
3	7.31e-4	1.09e-3	1.32e-3	2.21e-3	1.03e-3
4	6.45e-4	1.23e-3	2.21e-4	2.42e-3	1.14e-3

Table 3.2: Allen-Cahn Equation: Relative L_2 error between predicted and exact solutions at different times with different numbers of layers

3.3.3 Non-Linear Schrodinger Equation

The non-linear *Schrödinger* equation, describing the behavior of complex-valued wavefunctions, is chosen as another example to validate the effectiveness of our methodology further. The equation with the periodic boundary conditions is given as follows:

$$\begin{aligned} ih_t + 0.5h_{xx} + |h|^2 h = 0, x \in [-5,5], t \in [0,\pi/2], \\ h(0,x) = 2sech(x), \\ h(t,-5) = h(t,5), \\ h_x(t,-5) = h_x(t,5), \end{aligned} \tag{3.17}$$

where h is complex. Then, we can redefine $h(x,t) = u(x,t) + iv(x,t)$, where $u(x,t)$ represents the real part and $v(x,t)$ represents the complex part. Then, Eq. 3.17 could be split into

$$\begin{aligned} u_t = -0.5v_{xx} - (u^2+v^2)v, \\ v_t = 0.5u_{xx} + (u^2+v^2)u, \\ u(0,x) = 2sech(x), v(0,x) = 0, \\ u(t,-5) = u(t,5), v(t,-5) = v(t,5), \\ u_x(t,5) = u_x(t,5), v_x(t,-5) = v_x(t,5). \end{aligned} \tag{3.18}$$

We utilize two GRUs networks to approximate the u and v respectively. Each network is constructed by one layer of GRUs and two dense layers. We can obtain:

$$\begin{aligned} f(\bar{u},\bar{u}_x,\bar{u}_{xx}) = -0.5\bar{v}_{xx} - (\bar{u}^2+\bar{v}^2)\bar{v}, \\ f(\bar{v},\bar{v}_x,\bar{v}_{xx}) = 0.5\bar{u}_{xx} + (\bar{u}^2+\bar{v}^2)\bar{u}. \end{aligned} \tag{3.19}$$

Then, combining with the Adams-Moulton implicit methods, $f(\bar{u},\bar{u}_x,\bar{u}_{xx})$ and $f(\bar{v},\bar{v}_x,\bar{v}_{xx})$ are utilized to calculate the $\tilde{u}$ and $\tilde{v}$. We still randomly select 10 data collocation points as the regulator in the loss function, and the λ is set to 30. The parameters of these two networks are learned by adopting 6000 iterations of Adam with a learning rate 0.005 and 6000 iterations of *LBFGs* with a learning rate 0.001 to minimize the loss functions $loss = loss_u + loss_v$.

$$
\begin{aligned}
loss_u &= ||\bar{u}_0 - u_0||^2 + \sum_{i=1}^{N} ||\bar{u}_i - \tilde{u}_i||^2 + \lambda ||\bar{u}_{data} - u_{data}||^2, \\
loss_v &= ||\bar{v}_0 - v_0||^2 + \sum_{i=1}^{N} ||\bar{v}_i - \tilde{v}_i||^2 + \lambda ||\bar{v}_{data} - v_{data}||^2
\end{aligned}
\tag{3.20}
$$

Fig. 3.7 presents the comparison between the prediction $|h| = \sqrt{\bar{u}^2 + \bar{v}^2}$ and exact solution with time step $\Delta t = \frac{\pi}{400}$. It demonstrates that the network's predictions closely align with the exact solution. Fig. 3.8 depicts the variation in the loss function as iterations. Table 3.3 provides insight into the increasing relative L_2 errors over time. The relative L_2 errors slightly decrease when adding one more hidden layer.

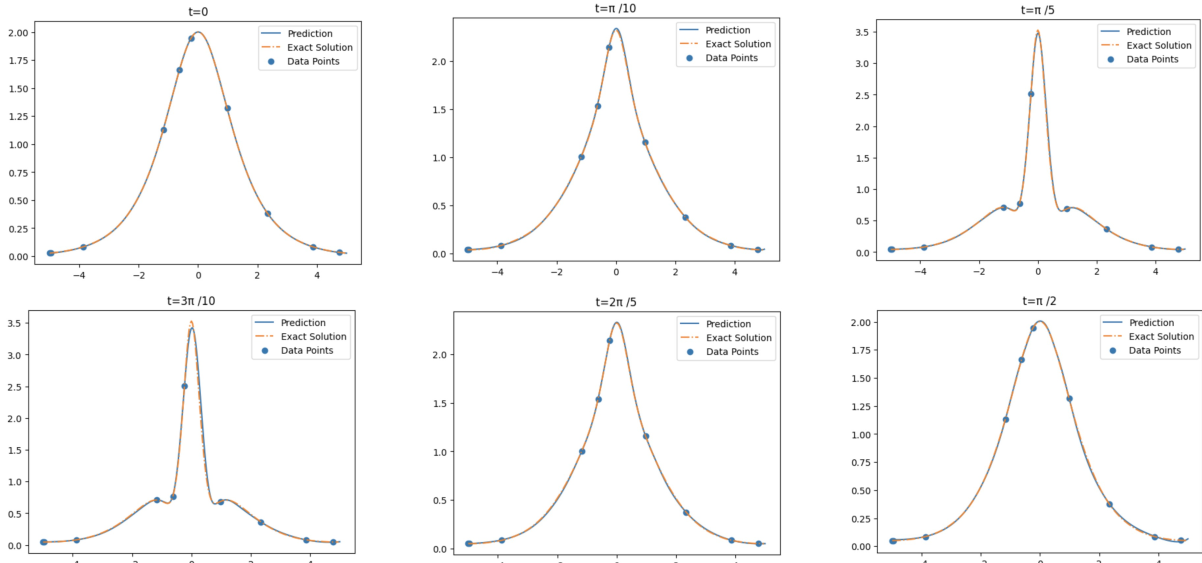


Figure 3.7: Schrodinger Equation: Snapshots of the predicted solutions and exact solutions at $t = \{0, \frac{\pi}{10}, \frac{2\pi}{10}, \frac{3\pi}{10}, \frac{4\pi}{10}, \frac{5\pi}{10}\}$.

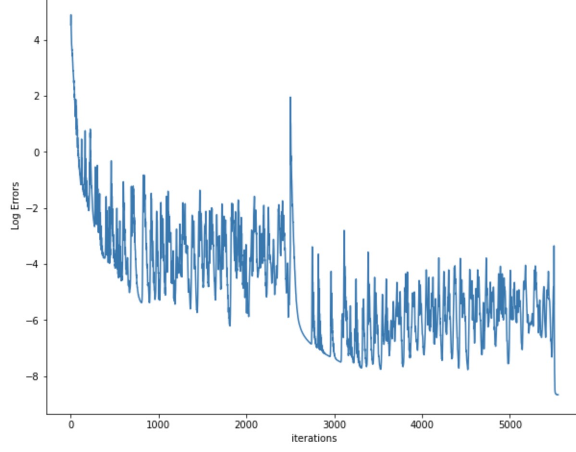


Figure 3.8: The variation of loss with iterations for Non-linear Schrodinger Equation

Layers \ t	$\pi/8$	$\pi/4$	$3\pi/8$	$\pi/2$	All Time
2	3.72e-3	4.22e-3	4.09e-3	6.28e-3	4.88e-3
3	2.41e-3	2.03e-3	4.04e-3	6.02e-3	4.62e-3
4	1.25e-3	2.59e-3	3.21e-3	4.02e-3	3.32e-3

Table 3.3: Non-Linear Schrodinger Equation: Relative L_2 error between predicted and exact solutions at different times with different numbers of layers

3.3.4 Coupled Burgers' Equation

The coupled Burgers' equation describes the change in relative concentrations of two types of particles in fluid suspensions or colloids due to the influence of gravity [17]. The model [17] is given below:

$$\begin{aligned}\frac{\partial u}{\partial t}-\delta\frac{\partial^2 u}{\partial x^2}+\eta u\frac{\partial u}{\partial x}+\alpha\frac{\partial(uv)}{\partial x}&=0, x\in[-10,10], t\in[0,1],\\ \frac{\partial v}{\partial t}-\mu\frac{\partial^2 v}{\partial x^2}+\xi v\frac{\partial v}{\partial x}+\beta\frac{\partial(uv)}{\partial x}&=0, x\in[-10,10], t\in[0,1],\end{aligned} \tag{3.21}$$

subject to the initial conditions:

$$\begin{aligned}u(x,0)&=f_1(x), x\in\Omega=[-10,10],\\ v(x,0)&=f_2(x), x\in\Omega=[-10,10],\end{aligned} \tag{3.22}$$

subject to boundary conditions:

$$
\begin{aligned}
u(a,t) = g_1(x), u(b,t) = g_2(x), t \in [0,1], \\
v(a,t) = g_3(t), v(b,t) = g_4(t), t \in [0,1].
\end{aligned}
\tag{3.23}
$$

The δ and μ, positive viscosity, are inverse of Reynolds number Re, if the $\delta = \mu$, then $\delta = \mu = 1/Re$. η, ξ, α, and β are constants. For $\delta = \mu = 1.0$, $\eta = \xi = 2.0$, and $a_0 = 0.05$, the exact solution is given [74]:

$$
\begin{aligned}
u(x,t) &= a_0 - 2A(\frac{2\alpha - 1}{4\alpha\beta - 1})tanh(A(x - 2At)), \\
v(x,t) &= a_0(\frac{2\beta - 1}{2\alpha - 1}) - 2A(\frac{2\alpha - 1}{4\alpha\beta - 1})tanh(A(x - 2At)),
\end{aligned}
\tag{3.24}
$$

where $A = a_0(\frac{4\alpha\beta - 1}{4\alpha - 2})$, and the initial conditions, u_0 and v_0, and boundary conditions, lower bound u_{lb}, v_{lb}, and upper bound u_{ub} and u_{ub}, are generated from the analytical solution. In this experiment, we set $\Delta t = 0.01$ and $\Delta x = 0.05$ to get collocation points, then observations can be obtained by plugging these points into the analytical solution. Similarly, we propose two networks, constructed by one GRUs layer and three dense layers, to estimate the u and v respectively to obtain the $\bar{u}$ and $\bar{v}$. The estimations can be differentiated by the finite difference to get $\bar{u}_t, \bar{u}_x, \bar{u}_{xx}, \bar{v}_t, \bar{v}_x$, and $\bar{v}_{xx}$. Then, we compute:

$$
\begin{aligned}
f(\bar{u}, \bar{u}_x, \bar{u}_{xx}) = \bar{u}_t - \delta\bar{u}_{xx} + \eta\bar{u}\bar{u}_x + \alpha(\bar{u}\bar{v})_x, \\
f(\bar{v}, \bar{v}_x, \bar{v}_{xx}) = \bar{v}_t - \mu\bar{v}_{xx} + \xi\bar{v}\bar{v}_x + \beta(\bar{u}\bar{v})_x,
\end{aligned}
\tag{3.25}
$$

We use the Eq. 3.25 to compute the $\tilde{u}$ and $\tilde{v}$. Ten observation data for each time step are randomly selected and incorporated into the loss function with the $\lambda = 15$. The learning rate is set to 0.005 for Adam and 0.001 for $LBFGs$. Then, the loss functions are constructed as follows:

$$
\begin{aligned}
loss_u = ||\bar{u}_0 - u_0||^2 + \sum_{i=1}^{N} ||\bar{u}_i - \tilde{u}_i||^2 + \lambda||\bar{u}_{data} - u_{data}||^2, \\
loss_v = ||\bar{v}_0 - v_0||^2 + \sum_{i=1}^{N} ||\bar{v}_i - \tilde{v}_i||^2 + \lambda||\bar{v}_{data} - v_{data}||^2.
\end{aligned}
\tag{3.26}
$$

The predictions and exact solutions for u and v are presented in Fig. 3.9, which demonstrates the remarkable accuracy of the proposed methodology. The figures of the first row of Fig. 4.2 display the prediction $\bar{u}$ and exact solutions u, and the below figures show the prediction $\bar{v}$ and analytical solutions v. The table 3.4 shows the average of relative L_2 errors of $\bar{u}$ and $\bar{v}$ with different architecture. The errors show a gradual decrease with the addition of more hidden layers.

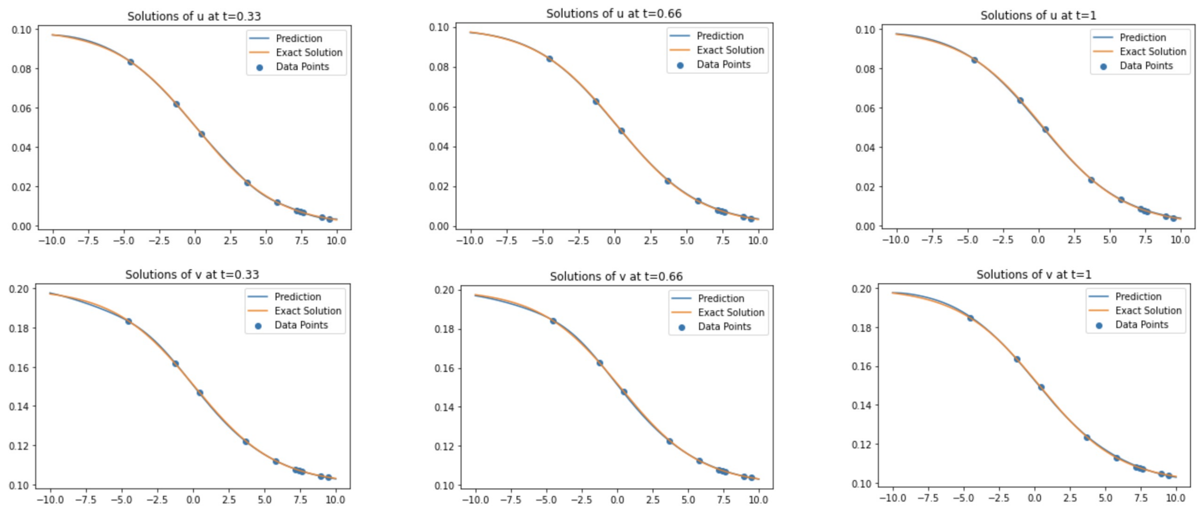


Figure 3.9: Coupled Burgers' Equations: Snapshots of the predicted solutions and exact solutions for u and v at $t = \{0.33, 0.66, 1\}$.

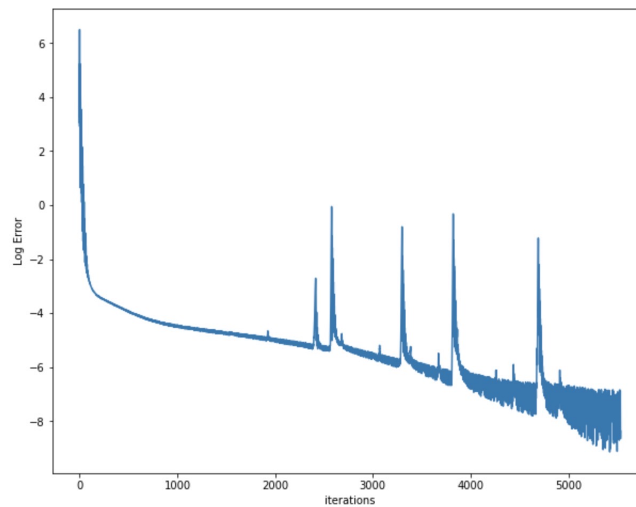


Figure 3.10: The variation of loss with iterations for Coupled Burgers' Equation

t / *Layers*	0.25	0.5	0.75	1.0	All Time
2	$6.92e-3$	$7.23e-3$	$8.46e-3$	$9.56e-3$	$6.21e-3$
3	$3.21e-3$	$3.01e-3$	$4.35e-3$	$4.88e-3$	$3.68e-3$
4	$8.81e-4$	$1.01e-3$	$2.19e-3$	$3.21e-3$	$1.98e-3$

Table 3.4: Coupled Burgers' Equations: Relative L_2 error between predicted and exact solutions at different times with different numbers of layers

3.3.5 Two-Dimentional Burgers' Equation

To explore the feasibility of our method on multi-dimensional PDEs, we apply the Network for two-dimensional uncoupled Burgers' equation:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + u\frac{\partial u}{\partial y} = \frac{1}{Re}(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}), t \in [0,2], (x,y) \in [0,1] \times [0,1], \tag{3.27}$$

subject to the initial and boundary conditions:

$$\begin{aligned} u(x,y,t_0) &= u_0(x,y), (x,y) \in [0,1] \times [0,1], \\ u(x_b,y_b,t) &= u_b, \end{aligned} \tag{3.28}$$

where the initial condition u_0 and boundary conditions u_b can be derived by analytical solution:

$$u(x,y,t) = \frac{1}{1+e^{\frac{Re(x+y-t)}{2}}}. \tag{3.29}$$

We choose $Re = 10$ for this experiment. The inputs of the network are augmented to two, spatial coordinates x and y, and the hidden state increases from one dimension to two dimensions. The output of our network is a three-dimensional tensor. Considering the higher computation complexity, we utilize four layers where the first layer is GRUs and the other three layers are dense layers. The *tanh* activation function is applied, and we still randomly select 10 data points from the solution for each time step as the regulator, and the corresponding $\lambda = 40$. We set $\Delta t = 0.01$ and $\Delta x = \Delta y = 0.005$. The learning rate is chosen to be 0.01 and reduced to 0.005 after 3,000 iterations. The network's output $\tilde{u}$ is the approximation of the solution of PDEs.

The central difference method is applied to differentiate $\bar{u}$ to obtain $\bar{u}_x$, $\bar{u}_y$, $\bar{u}_{xx}$, and $\bar{u}_{yy}$. Then, we can obtain

$$f(\bar{u}_x, \bar{u}_y, \bar{u}_{xx}, \bar{u}_{yy}) = \frac{1}{Re}(\bar{u}_{xx} + \bar{u}_{yy})^2 - \bar{u}(\bar{u}_x + \bar{u}_y)^2 \tag{3.30}$$

The Eq 3.30 is utilized to approximate the $\tilde{u}$ by adopting the Adams-Moulton Four-Step implicit method, which is presented in Algorithm 3. The loss function can be constructed as follows:

$$loss = ||\bar{u}_0 - u_0||^2 + \sum_{i=1}^{N} ||\bar{u} - \tilde{u}||^2 + \lambda ||\bar{u}_{data} - u_{data}||^2. \tag{3.31}$$

Fig. 3.11 presents three-dimensional plots of the predicted results (right column) obtained from networks and the exact solution (left column) obtained from analytical solutions for the Two-Dimensional Burgers' equation at $t = \{0.5, 1.0, 1.5, 2.0\}$. These results exhibit the effectiveness of our methodology to work on Two-Dimensional problems. Table 3.5 shows the L_2 errors are decreased as more hidden layers are employed. The architecture featuring two hidden layers lacks the necessary capacity to achieve the targeted accuracy in this case. Fig. 3.12 illustrates the changes in loss function throughout iterations.

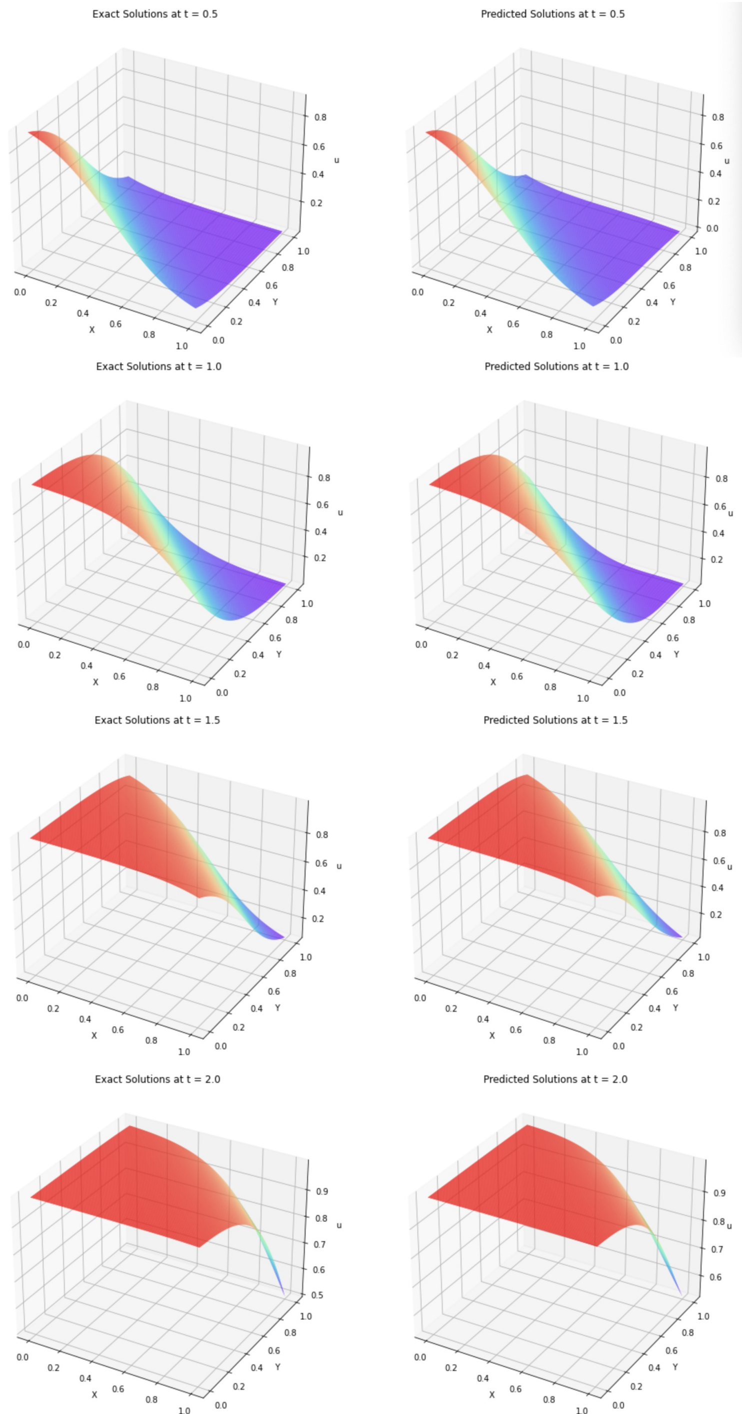


Figure 3.11: Two-Dimensional Burgers' Equation: Snapshots of predicted and exact solutions at $t = \{0.5, 1.0, 1.5, 2.0\}$

t / Layers	0.25	0.5	0.75	1	All Time
2	$8.32e-2$	$1.39e-1$	$1.89e-1$	$2.22e-1$	$1.84e-1$
3	$6.79e-3$	$3.25e-3$	$3.09e-3$	$2.87e-3$	$6.37e-3$
4	$4.79e-3$	$2.71e-3$	$1.55e-3$	$1.99e-3$	$1.87e-3$

Table 3.5: Relative L_2 error between predicted and exact solution at different times with different numbers of layers

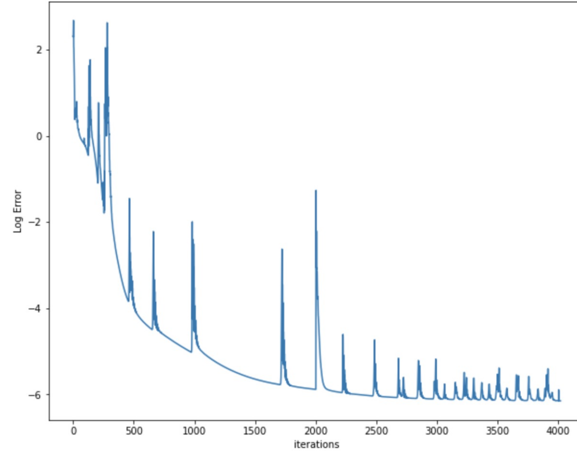


Figure 3.12: The variation of loss with iterations for 2D Burgers' Equation

3.3.6 Coupled Two-Dimensional Burgers' Equation

To explore the performance of the network on the complicated equations, we apply this methodology to the Coupled Two-Dimensional Burgers' equation, which is given as:

$$\begin{aligned} \frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} &= \frac{1}{Re}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) \\ \frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} &= \frac{1}{Re}\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right), \end{aligned} \tag{3.32}$$

where $t \in [0,2], (x,y) \in [0,1] \times [0,1]$, subject to the initial and boundary conditions:

$$\begin{aligned} u(x,y,0) &= u_0(x,y), \\ v(x,y,0) &= v_0(x,y), \\ u(x_b,y_b,t) &= u_b(x_b,y_b,t), \\ v(x_b,y_b,t) &= v_b(x_b,y_b,t). \end{aligned} \tag{3.33}$$

The analytical solutions can derive the initial and boundary conditions:

$$
\begin{aligned}
u(x,y,t) &= \frac{3}{4} - \frac{1}{4[1+exp[\frac{Re}{32}(-4x+4y-t)]]},\\
v(x,y,t) &= \frac{3}{4} + \frac{1}{4[1+exp[\frac{Re}{32}(-4x+4y-t)]]}.
\end{aligned}
\tag{3.34}
$$

In this experiment, the $Re = 100$. The sparse data points are randomly selected from the observation data generated from the analytical solutions Eq. 3.34. For this two-dimensional problem, we utilize two networks to generate the $\bar{u}$ and $\bar{v}$ respectively. Each network is constructed by one layer of GRUs and three dense layers. Let $\Delta t = 0.01$, $\Delta x = \Delta y = 0.005$. We set the learning rate equal to 0.003 for the initial $5,000$ iterations on Adam followed by $5,000$ iterations on $LBFGs$ where the learning rate equals 0.001. Then, we use the outputs, $\bar{u}$ and $\bar{v}$, to approximate $\bar{u}_x, \bar{u}_{xx}$,$\bar{u}_y, \bar{u}_{yy}$, $\bar{v}_x$, $\bar{v}_{xx}$, $\bar{v}_y$, and $\bar{v}_{yy}$ via employing finite difference method. Similarly, we have

$$
\begin{aligned}
f(\bar{u},\bar{u}_x,\bar{u}_{xx}) &= \frac{1}{Re}(\bar{u}_{xx}+\bar{u}_{yy}) - \bar{u}\bar{u}_x - \bar{v}\bar{u}_y,\\
f(\bar{v},\bar{v}_x,\bar{v}_{xx}) &= \frac{1}{Re}(\bar{v}_{xx}+\bar{v}_{yy}) - \bar{u}\bar{v}_x - \bar{v}\bar{u}_y.
\end{aligned}
\tag{3.35}
$$

Then, we utilize the implicit method to estimate the new iteration scheme $\tilde{u}$ and $\tilde{v}$. The loss function is constructed as follows:

$$
\begin{aligned}
loss_u &= ||\bar{u}_0 - u_0||^2 + \sum_{i=1}^{N} ||\bar{u}_i - \tilde{u}_i||^2 + \lambda||\bar{u}_{data} - u_{data}||^2,\\
loss_v &= ||\bar{v}_0 - v_0||^2 + \sum_{i=0}^{N} ||\bar{v}_i - \tilde{v}_i||^2 + \lambda||\bar{v}_{data} - v_{data}||^2
\end{aligned}
\tag{3.36}
$$

Fig. 3.14 and Fig. 3.15 show the analytical solutions (left columns) and predictions (right columns) for u and v respectively at $t = \{0.5, 1, 1.5, 2\}$. The strong consistency depicted in these two figures demonstrates that our model can attain high accuracy. A sufficiently large architecture can generate better results by comparing the errors from Table 3.6. Fig. 3.13 shows the errors are rapidly decreased in a few iterations.

Layers \ t	0.5	1.0	1.5	2.0	All Time
2	$1.87e-2$	$1.95e-1$	$1.48e-1$	$1.69e-1$	$1.73e-1$
3	$4.70e-3$	$6.06e-3$	$3.62e-4$	$3.55e-3$	$4.42e-3$
4	$2.34e-3$	$3.09e-3$	$1.25e-3$	$1.86e-3$	$2.60e-3$

Table 3.6: Average relative L_2 error of u and v between predicted and exact solution at different times with different numbers of layers

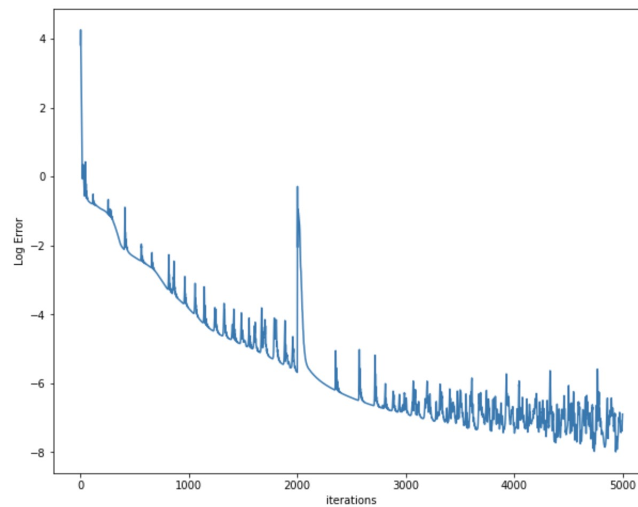


Figure 3.13: The variation of loss with iterations for Coupled 2D Burgers' Equation

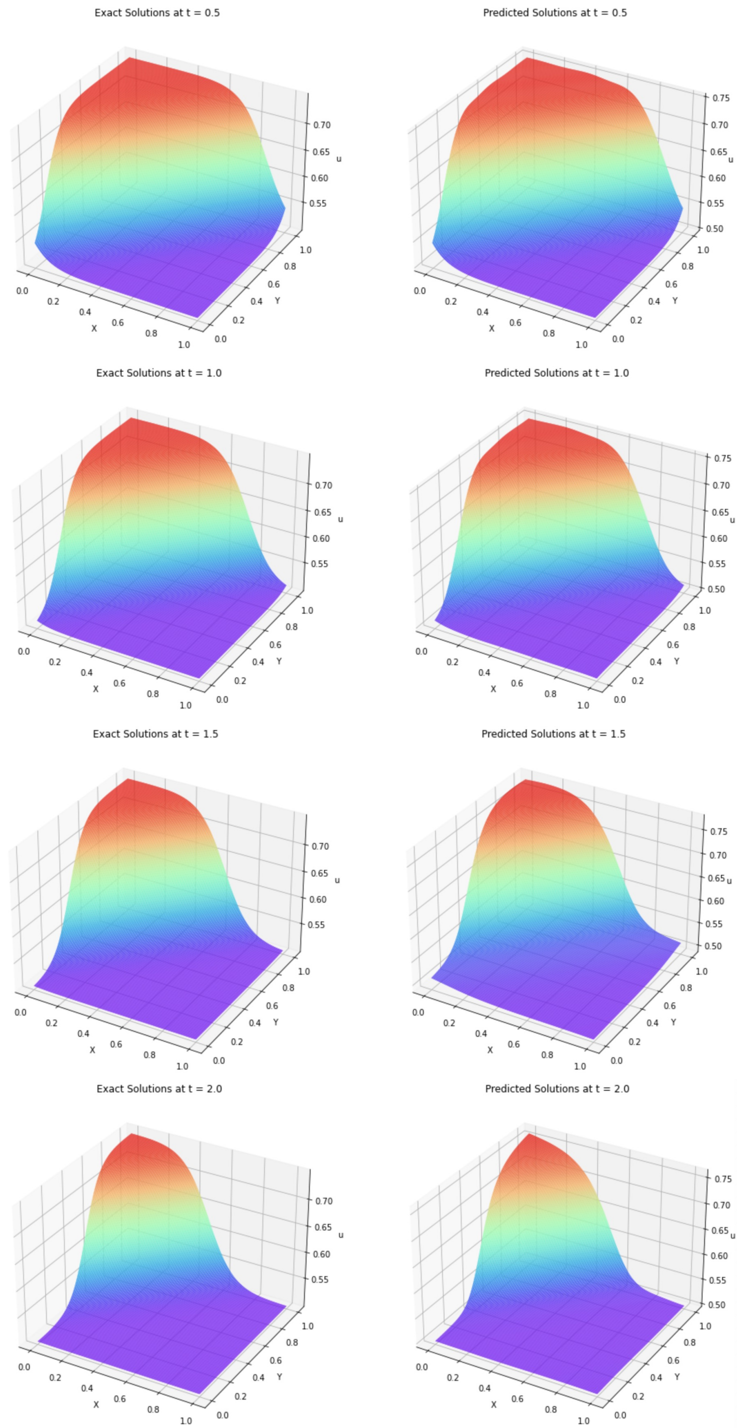


Figure 3.14: Coupled Two-Dimensional Burgers' Equation: Snapshots of predicted and exact Solutions for u at $t = \{0.5, 1.0, 1.5, 2.0\}$

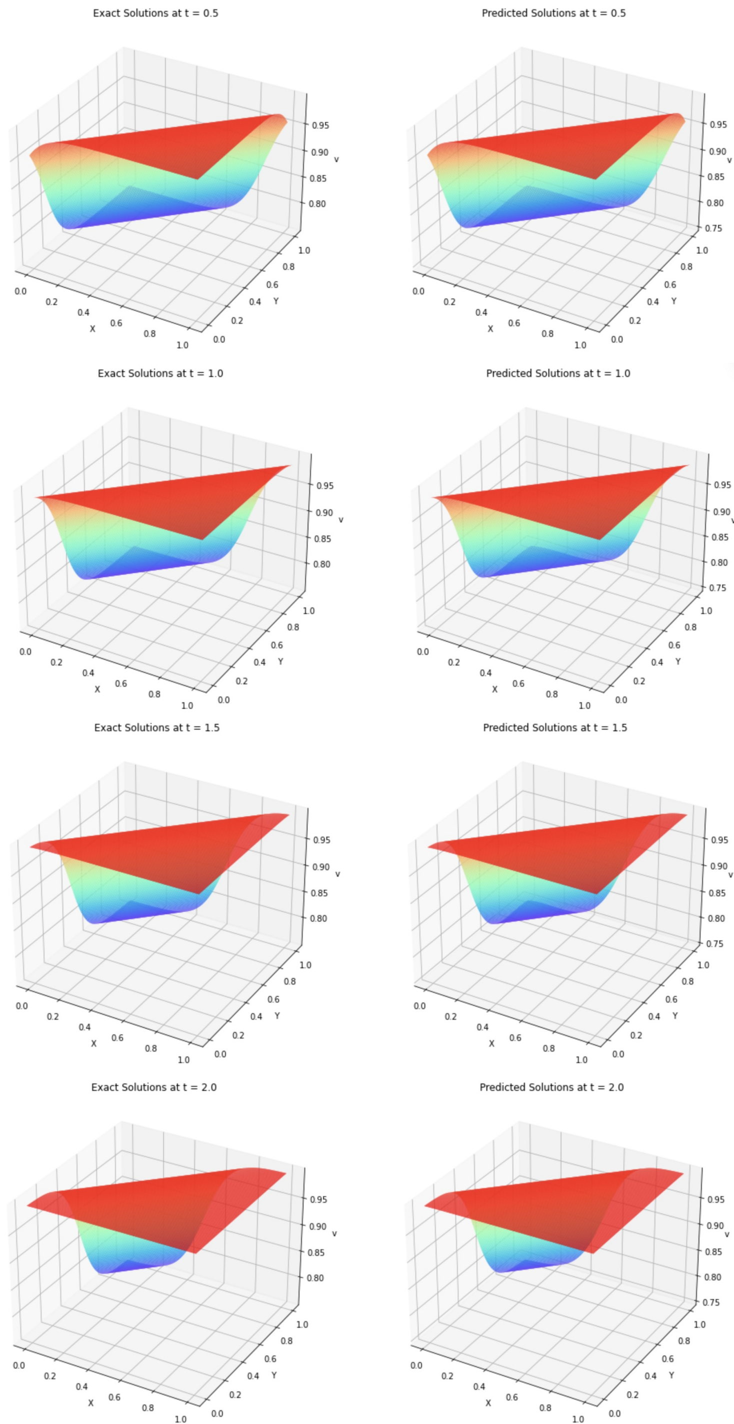


Figure 3.15: Coupled Two-Dimensional Burgers' Equation: Snapshots of predicted and exact Solutions for v at $t = \{0.5, 1.0, 1.5, 2.0\}$

3.3.7 Solving Inverse Problem

In this example, we propose our model to identify the parameters of the Allen-Cahn equation by using sparse data. The corresponding Allen-Cahn equation is

$$\begin{aligned} u_t - \lambda_1 u_{xx} + 5u^3 - 5u = 0, x \in [-1,1], t \in [0,1], \\ u(x,0) = x^2 cos(\pi x), \\ u(t,-1) = u(t,1), \end{aligned} \tag{3.37}$$

where λ_1, λ_2, and λ_3 are unknown parameters. In this case, we set $\Delta t = 0.005$, $\lambda_1 = 0.0001$, $\lambda_2 = \lambda_3 = 5$ to generate the synthetic data by utilizing the PDE solver. Then, we randomly select 10 data points for each time step. The setup of the neural network is the same as the example of solving the Allen-Cahn equation. The parameters are firstly randomly initialized and then utilized to obtain

$$f(\bar{u}, \bar{u}_x, \bar{u}_{xx}) = \lambda_1 \bar{u}_{xx} - 5\bar{u}^3 + 5\bar{u}. \tag{3.38}$$

The loss function is constructed as Eq. (3.16). Then, we also compared it to the regular PINN by using the same set-up neural network. The loss function Eq. (3.39) of the regular PINN is slightly different, and it has mean square errors on the boundary condition.

$$loss = \sum_{i=1}^{N} ||\bar{u}_i - \tilde{u}_i||^2 + 40||\bar{u}_{data} - u_{data}||^2 + ||\bar{u}_0 - u_0||^2 + ||\bar{u}(x,1) - \bar{u}(x,-1)||^2 \tag{3.39}$$

Fig. 3.16 presents the snapshots of the predicted solution and exact solution by our model. Table 3.7 shows the estimated parameters and relative L_2 errors by regular PINN and GRUs network. This table shows that the GRUs network gives the more accurate approximated parameters and smaller relative L_2 error.

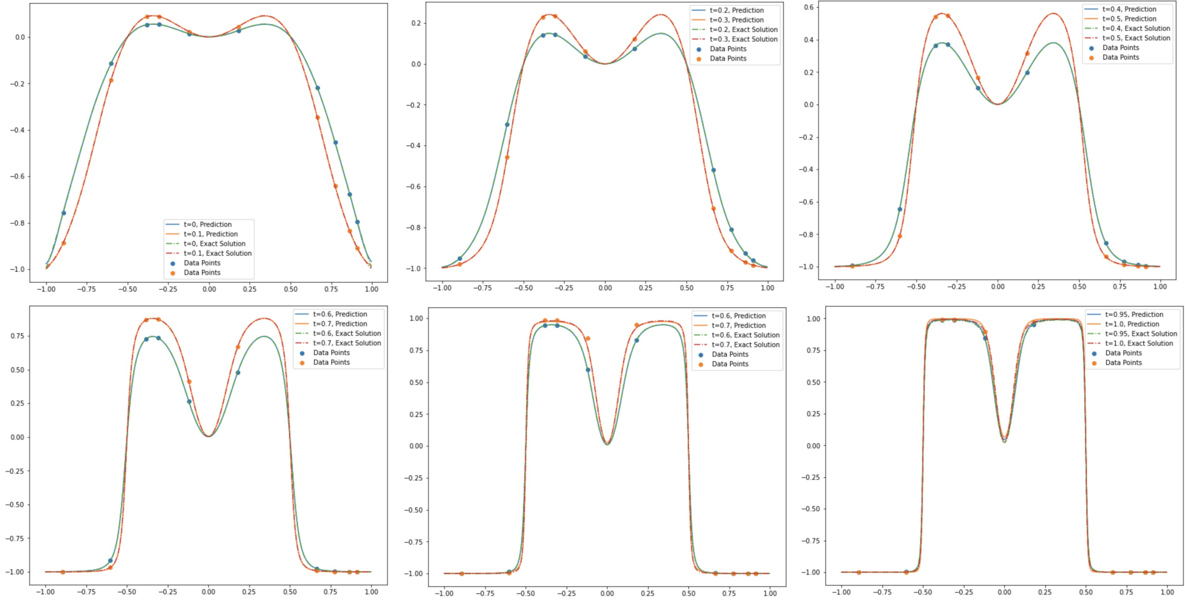


Figure 3.16: Inverse problem of Allen-Cahn Equation: Snapshots of predicted and exact solution

	PINN	GRU
λ_1	7.35e-5	$1.4e-5$
λ_2	5.2443	5.0075
λ_3	5.2389	4.9937
L_2 Error	1.2e-1	3.97e-3

Table 3.7: Comparison between PINN and GRU

3.4 Conclusion

The GRU neural network is designed to analyze time-series data. The implementation method involves utilizing a polynomial approximation to forecast the value of the answer at the subsequent time step. Merging these two strategies is instinctive for resolving the time-dependent partial differential equations. We use neural networks to approximate the iteration techniques for partial differential equations. The finite difference method is used to compute the derivatives of various schemes. New iteration schemes can be derived by utilizing the implicit technique. The network can reach the solutions of PDEs by minimizing the discrepancy between the original and new iteration schemes. Utilizing sparse interior observation data as a regulator aids in the network's convergence to the solution. Boundary conditions are unnecessary in our

methodology. Adequate architectural size is crucial for achieving the necessary accuracy, as determined by comparing the relative L_2 errors of various architectures. We also proposed this model to identify the parameters of the Allen-Cahn equation by only incorporating sparse data. To train the massive GRU network efficiently, we are contemplating using transfer learning to decrease memory utilization and enhance performance without starting from scratch.

CHAPTER 4

Physics-informed Encoder-Decoder Gated Recurrent Neural Network

The ability of deep learning approaches to approximate complex functions offers a promising alternative for solving partial differential equations. The methodology of incorporating the physics prior to the deep neural network can significantly reduce the requirement for labeled data. In this study, a novel physics-informed encoder-decoder gated recurrent units neural network is proposed to solve the time-dependent partial differential equations without using any observed data. The encoder is utilized to approximate the underlying patterns and structures of solutions over the entire spatial-temporal domain. The approximated solution is processed by the decoder, which is the Gated Recurrent Units layer. We utilize the initial condition as the initial state of the gated recurrent units to retain critical information in the hidden states. The boundary conditions are enforced in the final prediction to enhance the model's performance. Then we incorporate physical laws into the neural network during the training process. The effectiveness of this algorithm is demonstrated by solving Burgers-Fisher and Coupled Two-Dimensional Burgers equations.

4.1 Introduction

Partial differential equations (PDEs) are fundamental tools to describe systems in physics, biology, chemistry, and economics. For some complex PDEs that analytical methods can not solve, numerical methods are developed to approximate the solutions, such as Finite Difference Method (FDM) [54], Finite Element Method (FEM) [16], and Finite Volume Method (FVM) [18], etc. While they are effective in a wide range of applications, traditional numeric methods may struggle, especially for non-linear or high-dimensional PDEs, due to limitations in computational complexity, stability, and scalability. To address those challenges, alternative deep-learning approaches are rapidly growing to handle complicated PDEs, which tracks back to early contributions in the last century [39] [57].

Physics-Informed Neural Networks (PINNs) [60] are a famous type of deep-learning approach designed to solve differential equations by seamlessly integrating data-driven learn-

ing with physics-based modeling [34]. Compared with traditional numerical methods, PINNs could be a mesh-free approach due to the availability of automatic differentiation [55] [5], which makes them flexible in complex geometries. When the size of observed data is small in applications, structured prior information is encoded into a learning algorithm that satisfies given physical constraints, by penalizing the loss function, to maximize the use of available data [53] [62] [61] [73]. PINNs are also capable of discovering underlying hidden PDE models and finding solutions, thanks to the smoothness of the PINN formulation [46] [59]. Moreover, PINNs can endow a neural network (NN) with uncertainty quantification (UQ) techniques [91] [90] to improve the accuracy of predictions.

The PINN-based approach has shown significant advancements, with the integration of distinct architectures. The family of Recurrent Neural Network (RNN)[67] architectures, which we are more interested in, are effective in language modeling and machine translation [23] [76] [4] due to their ability to handle sequential data by processing input sequences step by step while maintaining a hidden state that captures information from previous time steps. However, RNNs are prone to the vanishing and exploding gradient problem during backpropagation, leading to difficulties in learning long-range dependencies [29]. Various enhancements are proposed to mitigate this issue when training RNNs [14] [33] [40], such as Long Short-Term Memory (LSTM) [30] and Gated Recurrent Unit (GRU) [12], which use gates to control the flow of information through network. Meanwhile, RNN-based techniques including RNN Encoder-Decoder [12] [76] attracts our attention, where two RNNs are used and perform well on some challenging tasks.

In this study, we propose a novel deep learning algorithm, the Physics-Informed Encoder-Decoder Gated Recurrent Neural Network (PhyEDGNN), for solving time-dependent partial differential equations (PDEs). This method leverages an encoder-decoder architecture where the encoder, a multilayer perceptron, learns the underlying patterns and structures of PDE solutions. The initial approximation of the PDE solutions is fed into the decoder after a matrix transformation. The decoder, composed of Gated Recurrent Units (GRUs), is designed to handle time series data effectively. By using the initial condition as the initial state of the GRUs, we retain crucial information in the hidden states. Furthermore, boundary conditions are enforced

in the final prediction to enhance model performance. We also integrate physical laws into the neural network during the training process. Our results demonstrate the effectiveness of this approach, showing superior performance compared to the vanilla Physics-Informed Neural Networks (PINNs).

4.2 Problem Setup

In this chapter, we consider the general form of the time-dependent PDEs in a regular domain:

$$\begin{aligned} &u_t(\mathbf{x},\mathbf{t}) + \mathscr{L}(u(\mathbf{x},\mathbf{t});\lambda) = 0, \mathbf{x} \in \Omega, \mathbf{t} \in [0,T], \\ &u(\mathbf{x},0) = u_0(\mathbf{x}), \mathbf{x} \in \Omega, \\ &u(\mathbf{x},\mathbf{t}) = g(\mathbf{x},\mathbf{t}), \mathbf{x} \in \partial\Omega, \mathbf{t} \in [0,T], \end{aligned} \tag{4.1}$$

where $t \in [0,T]$ and $\Omega \subset \mathbb{R}^d$ $(d \geq 1)$ represent the computational domain, $\mathscr{L}$ denotes the differential operator, $u_0(\mathbf{x})$ is the initial condition, $g(\mathbf{x},\mathbf{t})$ is the boundary condition defined on the domain $\partial\Omega$. In this work, the solution of Eq.(4.1) is approximated by a neural network framework without using any labeled data.

4.3 Methodology

In this section, the predictor-corrector GRUs network is proposed for solving the time-dependent PDE systems. We utilize the Multilayer Perceptron (MLP) as the encoder to generate an initial approximation of the solution based on the input. Then, the initial approximation is refined by the decoder, which is the GRUs layer, by taking into account more information, such as initial and boundary conditions.

4.3.1 Encoder-Decoder GRUs

Utilizing the architecture of the encoder-decoder can efficiently approximate solutions to partial differential equations (PDEs). In this hybrid framework, an MLP serves as the encoder, initially generating approximate solutions. The MLP, with its ability to capture complex nonlinear relationships, provides a fast and effective means of producing initial estimations. This process can be described as:

$$\tilde{u}(\mathbf{x},\mathbf{t}) = f_1(\mathbf{x},\mathbf{t};\theta_1), \tag{4.2}$$

where f_1 represents the MLP, $\tilde{u}(\mathbf{x},\mathbf{t})$ is the initial approximation of solution, θ_1 is the parameters of the MLP. To enhance the accuracy of these predictions, Gated Recurrent Unit (GRU) layers are employed as the decoder. PDEs often involve dynamic processes where the solution evolves over time. GRUs, being recurrent neural networks, are specifically designed to model sequences and capture temporal dependencies. The gating mechanisms in GRUs allow them to selectively update and retain information over time, enabling the model to capture long-range dependencies in sequential data while mitigating issues such as the vanishing gradient problem [14]. The initial approximation $\tilde{u}$ is fed into the GRUs, and the initial condition is served as the initial state. Then, we have

$$\bar{u}(\mathbf{x},\mathbf{t}) = f_2(\tilde{u}(\mathbf{x},\mathbf{t}), u_0; \theta_2), \tag{4.3}$$

where f_2 is the GRUs layer, θ_2 represents the parameters of the GRUs, $\bar{u}(\mathbf{x},\mathbf{t})$ is the final approximation of the solution. Then, we need to consider how to handle the given conditions such as initial and boundary conditions.

4.3.2 Initial/Boundary Conditions Treatment

Generally, there are two main ways to handle the initial and boundary conditions. The first one is to add them as the penalty term in the loss function [60]. However, applying the gradient descent method to the multi-objective loss function may result in poor performance [83]. This happens because gradient descent, being a greedy optimization procedure, may prioritize certain components over others. This selective focus can lead to an imbalance in the rate of improvement among different loss components, hindering convergence to the accurate solution [50].

The second method involves enforcing the initial and boundary conditions as hard constraints in the model. This approach offers several benefits. First, it imposes a stringent constraint, ensuring zero error on the initial and boundary conditions. Second, the neural network does not need to learn these conditions independently, allowing it to focus exclusively on learning the parameters that minimize the residuals of the partial differential equations. Third, there are no weights to adjust to control the impact of the initial and boundary conditions on the final solution [66]. We utilized this idea in our algorithm, and the detailed architecture is described

in the next subsection.

4.3.3 Architecture

Fig 4.1 provides more details of our architecture. The whole architecture is comprised of an encoder, a multilayer perception, and a decoder, which is a layer of GRUs. The inputs, only the spatial coordinates **x** and **y**, are propagated into the encoder to learn the underlying patterns and structures of PDE solutions $\tilde{u}$. Then, we make a matrix transformation and dimension increasing for $\tilde{u}$ to fit the GRUs layer.

We choose the initial condition as the initial state of the GRUs layer. Then, the first GRU will output the prediction $\bar{u}_0$ at t_0, and add the mean square errors between the $\bar{u}_0$ and u_0 into the loss function. We designed it this way because the instantaneous transition in dynamics is subtle, allowing the hidden state to efficiently discern which information to retain initially and carry forward to the next unit. Also, the associated loss term won't impose a significant burden on the network. Then, we enforce the boundary conditions as the hard constraints so that there are no errors in them. The loss function is well constructed by leveraging this ingenious design in subsection 4.3.4.

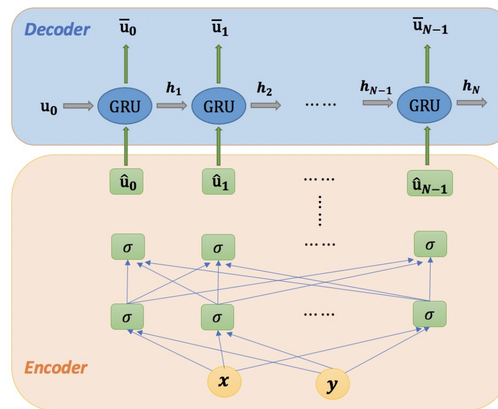


Figure 4.1: Architecture of Encoder-Decoder GRUs

4.3.4 Physics-informed Loss function

Since the boundary conditions are incorporated into the neural network, the loss function is constructed by the governing PDEs and initial conditions. Firstly, we implement the central

difference method to approximate the partial derivatives [77],

$$
\begin{aligned}
\bar{u}_j(x_i,t_j;\theta) &= \frac{1}{2\Delta t}(\bar{u}(x_i,t_{j+1};\theta)-\bar{u}(x_i,t_{j-1};\theta)),\\
\bar{u}_x(x_i,t_j;\theta) &= \frac{1}{2\Delta x}(\bar{u}(x_{i+1},t_j;\theta)-\bar{u}(x_{i-1},t_j;\theta)),\\
\bar{u}_{xx}(x,t_j;\theta) &= \frac{1}{\Delta x^2}(\bar{u}(x_{i+1},t_j;\theta)-2\bar{u}(x_i,t_j;\theta)+\bar{u}(x_{i-1},t_j;\theta)).
\end{aligned}
\tag{4.4}
$$

Then, the PDE residual is defined as:

$$
\mathscr{R}(\mathbf{x},\mathbf{t});\theta) = \bar{u}_j(x_i,t_j;\theta)+\mathscr{L}(\bar{u}(x_i,t_j);\theta;\lambda), \tag{4.5}
$$

which satisfies the Eq.(4.1). Then, the network parameters θ can be learned by minimizing the loss function

$$
loss = ||\bar{u}_0-u_0||^2+\sum||\mathscr{R}(\mathbf{x},\mathbf{t});\theta_2)||^2. \tag{4.6}
$$

The above procedures are summarized in the algorithm 4

Algorithm 4 Encoder-Decoder GRUs for Solving Time-dependent PDEs

Input: x, y;

Output: Predicted solutions of PDEs: $\bar{u}$.

1: Randomly initialize the parameters of Networks.

2: Utilize the Encoder to generate the $\tilde{u}$.

3: Transform the $\tilde{u}$ and set the transformed $\tilde{u}$ as the Decoder's input.

4: Set the initial condition as the Decoder's hidden state.

5: Utilize the Decoder to approximate the solution $\bar{u}$.

6: Enforce the boundary condition.

7: Calculate the loss function Eq.(4.6).

8: Minimize the loss function by applying the Adam and L-BFGS optimizers.

9: Get the final prediction $\bar{u}$.

4.4 Numerical Results

In this section, our proposed method is validated on the Burgers-Fisher equation and Coupled Two-Dimensional equations, and compared to the vanilla PINN algorithm. We utilize the rel-

ative L_2 errors in Eq.(3.13) to compare the performance between them. We train the network using the Adam optimizer [38] followed by L-BFGS [44] to minimize the loss function. The Adam optimizer helps avoid local minima, and L-BFGS refines the solutions. We use the tanh function as the activation function for the neural networks. Additionally, we implement learning rate decay, starting with a relatively large learning rate and reducing it after a set number of iterations. Each GRU generates a prediction at each time step, so the number of neurons and GRUs per layer depends on the time step size. To test the effectiveness and accuracy of our algorithm, we apply our model to the Burgers-Fisher and Coupled Two-Dimensional Equations and compare the results to their exact solutions. Then, we evaluate the model using different step sizes to examine how the step size affects the model's performance.

4.4.1 Burgers-Fisher Equation

To show the efficiency of PhyEDGNN, we applied it to solve the generalized Burgers-Fisher's equation [2]. This equation models various phenomena in fluid dynamics, reaction-diffusion processes, and population genetics by incorporating nonlinear advection, diffusion, and reaction terms.

$$\begin{aligned} &\frac{\partial u}{\partial t} + \alpha u^{\delta}\frac{\partial u}{\partial x} = \frac{\partial^2 u}{\partial x^2} + \beta u(1-u^{\delta}), (x,t) \in [0,1]\times[0,1], \\ &u(x,0) = (\frac{1}{2} - \frac{1}{2}tanh(\frac{-\alpha\delta}{2(\delta+1)}x))^{1/\delta}, x \in [0,1]. \end{aligned} \tag{4.7}$$

The Burgers term $\alpha u^{\delta}\frac{\partial u}{\partial x}$ represents the nonlinear convection, the Fisher term $\beta u(1-u^{\delta})$ describes logistic population growth or reaction kinetics, and the diffusion term $\frac{\partial^2 u}{\partial x^2}$ accounts for spatial spreading effects. The exact solution to Eq.(4.7) is given as below:

$$u(x,t) = (\frac{1}{2} + \frac{1}{2}tanh[\frac{-\alpha\delta}{2(\delta+1)}(x - (\frac{\alpha}{\delta+1} + \frac{\beta(\delta+1)}{\alpha})t])^{1/\delta} \tag{4.8}$$

In this case, we set the $\delta = 1$, $\alpha = 1$, and $\beta = 1$. We choose $\Delta x = 0.001$ and $\Delta t = 0.005$. N represents the number of time steps, and $N = 1 + \frac{1}{\Delta t}$. So the encoder consists of four hidden layers, each with N neurons, while the decoder is a single GRUs layer with N units. The output $\bar{u}$ is generated by the neural network, and the finite difference approach is utilized to calculate

the partial derivatives. Then, the corresponding loss function is

$$loss = \sum_{i=1}^{N-1} ||\bar{u}_t^i + \bar{u}\bar{u}_x^i - \bar{u}_{xx}^i - \bar{u}^i(1-\bar{u}^i)||^2 + ||\bar{u}_0 - u_0||^2. \tag{4.9}$$

The parameters θ are learned by minimizing the loss function. The initial learning rate for Adam is set to 0.005. After 5000 epochs, it is reduced to 0.001 for another 5000 epochs. Then, we keep using the same learning rate for L-BFGS. Fig. 4.2 demonstrates the comparison between the ground truth and predicted solutions of Eq.(5.27) at $t = (0.15, 0.3, 0.45, 0.6, 0.75, 1)$. The blue lines represent the exact solution, and the orange lines represent the predicted solution of the proposed algorithm. Our model exhibits remarkable accuracy in approximating the solutions to Burgers-Fisher equations, with the results closely aligning with the exact solutions. This high degree of precision demonstrates the model's effectiveness in capturing the underlying physics of the problem. To show the effect of the step sizes, Table 4.1 presents the relative L_2 errors at different step sizes. We set the learning rate to 0.003 for the initial 8000 epochs and then decay to 0.001 for the following 8000 epochs for all of these cases. We observe the case, $\Delta t = 0.001$ and $\Delta x = 0.001$, achieves the smallest L_2 error. Choosing the smaller Δt and Δx can bring the smaller relative L_2 errors compared to the larger step sizes.

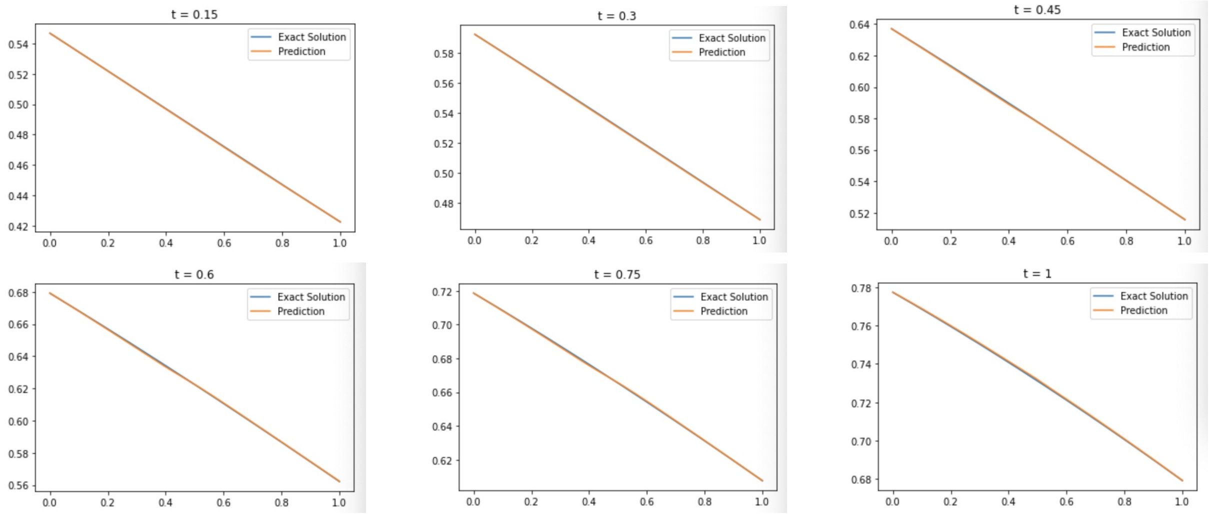


Figure 4.2: PhyEDGNN Solving Burgers-Fisher equation

Δx \ Δt	0.05	0.01	0.005	0.001
0.01	1.12e-2	4.54e-3	9.53e-3	2.90e-3
0.005	1.84e-2	8.82e-3	8.58e-3	4.25e-3
0.001	4.39e-2	2.53e-2	4.58e-3	3.77e-4

Table 4.1: Relative L_2 errors between predicted and exact solutions using different step sizes

4.4.2 Coupled Two-Dimensional Burgers' Equation

To evaluate the network's performance on complex equations, we apply this methodology to the Coupled Two-Dimensional Burgers' equation, defined as:

$$\begin{aligned} \frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} &= \frac{1}{Re}(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}) \\ \frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} &= \frac{1}{Re}(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}), \end{aligned} \tag{4.10}$$

where $t \in [0,2], (x,y) \in [0,1] \times [0,1]$, subject to the initial and boundary conditions:

$$\begin{aligned} u(x,y,0) &= u_0(x,y), \\ v(x,y,0) &= v_0(x,y), \\ u(x_b,y_b,t) &= u_b(x_b,y_b,t), \\ v(x_b,y_b,t) &= v_b(x_b,y_b,t). \end{aligned} \tag{4.11}$$

The analytical solutions can be used to derive the initial and boundary conditions:

$$\begin{aligned} u(x,y,t) &= \frac{3}{4} - \frac{1}{4[1+exp[\frac{Re}{32}(-4x+4y-t)]]}, \\ v(x,y,t) &= \frac{3}{4} + \frac{1}{4[1+exp[\frac{Re}{32}(-4x+4y-t)]]}. \end{aligned} \tag{4.12}$$

In this case, we set $Re = 100$, $\Delta x = \Delta y = 0.0025$, and $\Delta t = 0.01$, then correspondingly $N = 201$. We utilize two networks to estimate $\bar{u}$ and $\bar{v}$ respectively. Each network consists of an encoder with three hidden layers, each containing N neurons, and a decoder with N GRUs. We minimize the loss function using the same approach as in the previous numerical example. The loss

function is given as follows:

$$\begin{aligned} loss_u &= \sum_{i=1}^{N-1} ||\bar{u}_t^i + \bar{u}^i\bar{u}_x^i + \bar{v}^i\bar{u}_y^i - \frac{1}{Re}(\bar{u}_{xx}^i + \bar{u}_{yy}^i)||^2 + ||\bar{u}_0 - u_0||^2, \\ loss_v &= \sum_{i=1}^{N-1} ||\bar{v}_t^i + \bar{u}^i\bar{v}_x^i + \bar{v}^i\bar{v}_y^i - \frac{1}{Re}(\bar{v}_{xx}^i + \bar{v}_{yy}^i)||^2 + ||\bar{v}_0 - v_0||^2, \end{aligned} \tag{4.13}$$

where $loss = loss_u + loss_v$. Fig.4.3 and Fig.4.4 present the strong consistency of the predicted solution and the exact solution for u and v respectively at $t = \{0.6, 1.2, 1.8, 2.0\}$. The left columns of the figures are the exact solutions, and the right columns are the predicted values. The deep learning approach has shown exceptional performance in solving partial differential equations, yielding nearly indistinguishable results from the exact solutions. The minimal discrepancy between the model's predictions and the exact solutions indicates that the deep learning model can learn the complex patterns and dynamics inherent in this equation. Furthermore, we can observe that the shocks are well predicted in Fig. 4.4. Table 4.2 presents the average relative L_2 errors of u and v at different step sizes. It shows that the different choices of step sizes in spatial and temporal dimensions impact the accuracy of the model.

Δx \ Δt	0.05	0.01	0.005
0.01	2.37e-3	2.44e-4	4.43e-2
0.005	1.31e-3	8.33e-3	3.77e-2
0.0025	1.82e-3	4.32e-3	1.73e-3

Table 4.2: Average Relative L_2 errors between predicted and exact solutions using different step sizes

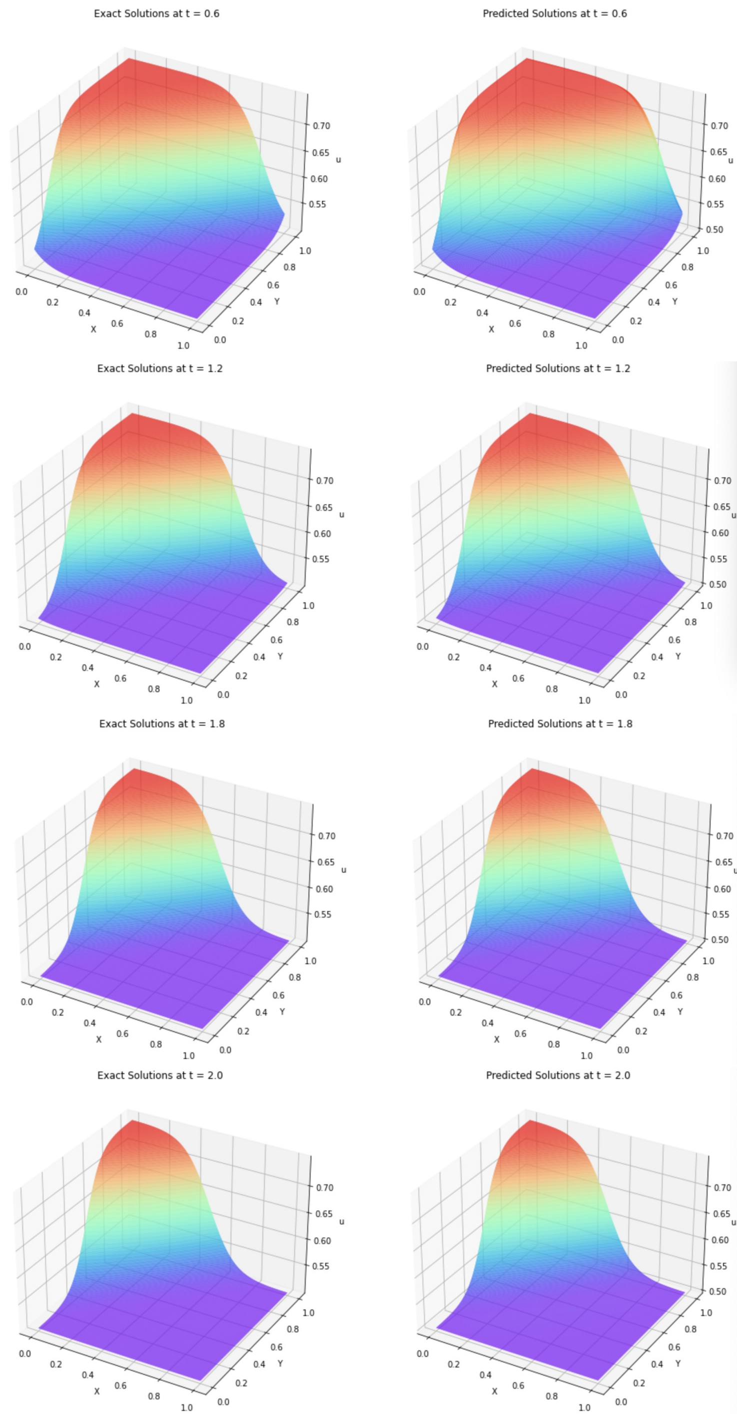


Figure 4.3: Coupled Two-Dimensional Burgers' Equation: Snapshots of predicted and exact Solutions for u at $t = \{0.6, 1.2, 1.8, 2.0\}$

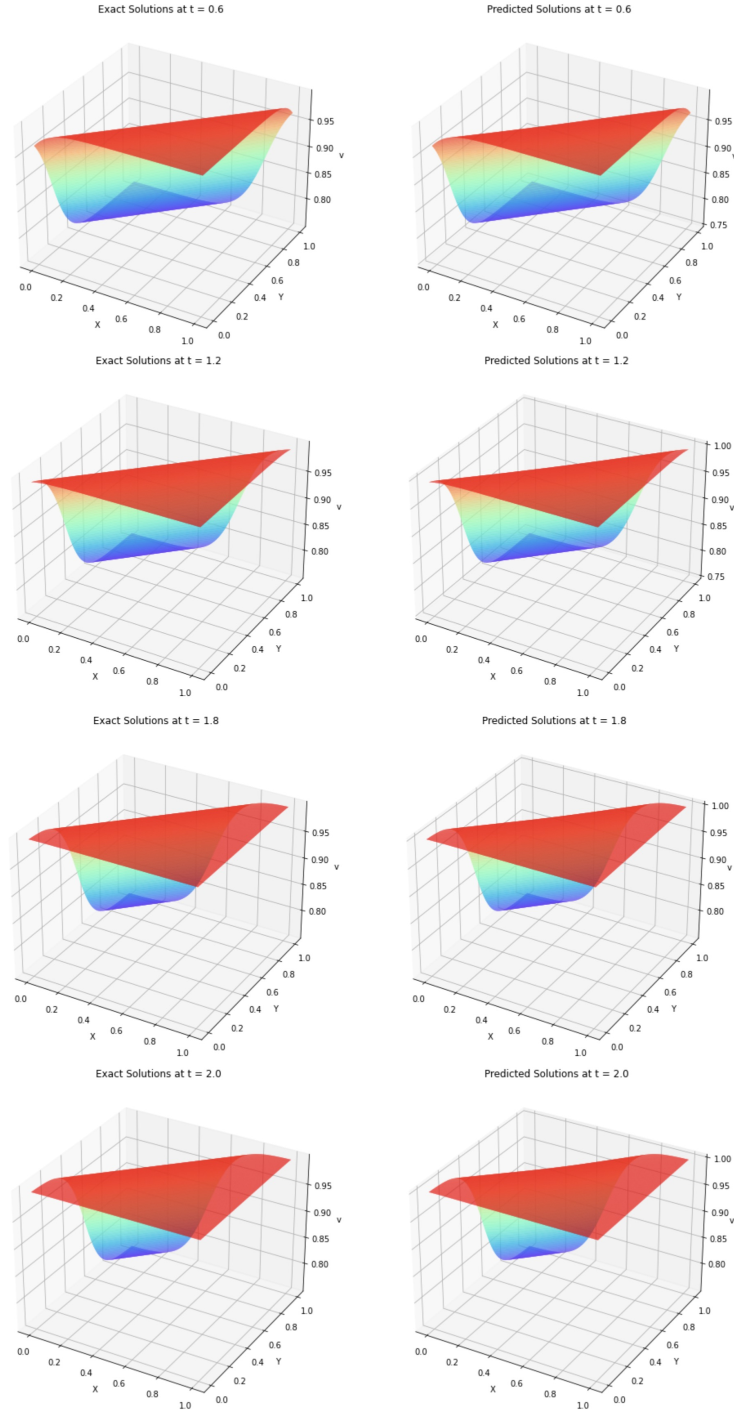


Figure 4.4: Coupled Two-Dimensional Burgers' Equation: Snapshots of predicted and exact Solutions for v at $t = \{0.6, 1.2, 1.8, 2.0\}$

4.4.3 Comparision of PhyEDGNN with PINN

To demonstrate the advantage of our approach, the relative L_2 errors of vanilla PINN and PhyEDGNN are listed in Table 4.3. We employ Latin hypercube sampling (LHS) to generate 20,000 collocation points within the spatio-temporal domain and 1,000 collocation points

on boundary and initial conditions. We use five hidden layers with 128 neurons per layer, and the Hyperbolic Tangent function is applied. The training process involves 8000 iterations using the Adam optimizer and subsequent iterations using L-BFGS until convergence. The initial learning rate is 0.001 for the first 5,000 epochs, then decreased to 0.0005 until the convergence.

The results indicate that PhyEDGNN performs better. While PINNs offer a powerful framework for solving PDEs by integrating physical laws into the neural network training process, their application to the Burgers-Fisher equation can be problematic due to the complexity and interplay of nonlinear advection, reaction-diffusion processes, and optimization challenges. As for the Coupled Two-Dimensional Burgers equation, we choose a relatively large Reynolds number, $Re = 100$, resulting in the discontinuity in the solution making it difficult for PINNs to accurately approximate the solution.

PDE	PINN	PhyEDGNN
Burgers-Fisher	$1.15e-1$	$6.81e-4$
Coupled 2D-Burgers	$2.15e-1$	$1.73e-3$

Table 4.3: The comparison of PINN and PhyENGNN with respect to relative L_2 errors

4.5 Conclusion

In this study, we proposed a novel deep learning algorithm, Physics-informed Encoder-Decoder Gated Recurrent Neural Network, for solving time-dependent PDEs. We utilized the encoder-decoder architecture in this method. The encoder is a multilayer perceptron learning the underlying patterns and structures of PDE solutions. The initial approximation of the PDE solutions is fed into the decoder after the matrix transformation. The decoder is the layer of Gated Recurrent Units to handle the time series. We utilized the initial condition as the initial state of the GRUs to easily retain the important information in hidden states to improve the ability to capture and maintain long-term dependencies in the solution. This way enables generalization to various initial condition scenarios. The boundary conditions are enforced into the final prediction to improve the performance of the model. Then, we incorporated the physical laws into the neural network for the training process. The results demonstrate the effectiveness of our model, and our model performs better than vanilla PINN. In the future, we consider applying

our model to irregular domain problems. Therefore, the Graph Neural Network is considered to handle the irregular domain by leveraging the flexibility and locality of graph-based representations.

CHAPTER 5

Self-Learning approach for solving PDEs

The regular PINN fails to converge to the solution due to an imbalance in the multi-component loss function within the back-propagated gradients during training [82]. The standard approach to mitigate this issue is to add appropriate weights to each component of the loss function. However, determining the proper weights is challenging. In this chapter, we introduce the Self-Learning Physics-Informed Neural Network (SL-PINN) to solve PDEs. In this method, the weights are learned by separate neural networks, eliminating the need for fine-tuning hyper-parameters. The effectiveness of this approach is demonstrated by the Burgers' and Burgers-Fisher equations.

5.1 Methodology

In this section, although we have introduced the PINNs in previous sections, we firstly give the overview of the PINNs to better present other methods.

5.1.1 Overview of the PINNs

Consider the general PDEs:

$$
\begin{aligned}
&u_t(\mathbf{x},\mathbf{t}) + \mathscr{L}(u(\mathbf{x},\mathbf{t});\lambda) = 0, \mathbf{x} \in \Omega, \mathbf{t} \in [0,T], \\
&u(\mathbf{x},0) = u_0(\mathbf{x}), \mathbf{x} \in \Omega, \\
&u(\mathbf{x},\mathbf{t}) = g(\mathbf{x},\mathbf{t}), \mathbf{x} \in \partial\Omega, \mathbf{t} \in [0,T],
\end{aligned}
\tag{5.1}
$$

where time $t \in [0,T]$ and space $x \in \Omega \subset \mathbb{R}^d$ $(d \geq 1)$ represent the computational domain, $\mathscr{L}$ denotes the differential operator, $u_0(\mathbf{x})$ and $g(\mathbf{x},\mathbf{t})$ represent the initial condition and the boundary condition defined on the domain $\partial\Omega$ respectively.

Utilizing the feedforward neural network to approximate the solution $u_t(\mathbf{x},\mathbf{t})$, we can construct the corresponding loss function. Firstly, we can define the PDE residual $f(\mathbf{x},\mathbf{t})$:

$$
f(\mathbf{x},\mathbf{t}) = u_t(\mathbf{x},\mathbf{t}) + \mathscr{L}(u(\mathbf{x},\mathbf{t});\lambda). \tag{5.2}
$$

Then, the loss function can be constructed as follows:

$$\mathscr{L} = \mathscr{L}_0(\theta) + \mathscr{L}_b(\theta) + \mathscr{L}_f(\theta), \tag{5.3}$$

where θ represents the parameters of the neural network, which can learned by minimizing the loss function. Each term of Eq.(5.3) can be expressed as:

$$\mathscr{L}_0(\theta) = \frac{1}{N_0}\sum_{i=1}^{N_0}(u(\mathbf{x}_0^i, \mathbf{t}_0^i, \theta) - u_0(\mathbf{x}_0^i))^2, \tag{5.4}$$

$$\mathscr{L}_b(\theta) = \frac{1}{N_b}\sum_{i=1}^{N_b}(u(\mathbf{x}_b^i, \mathbf{t}_b^i, \theta) - g(\mathbf{x}_b^i))^2, \tag{5.5}$$

$$\mathscr{L}_f(\theta) = \frac{1}{N_f}\sum_{i=1}^{N_f}(f(\mathbf{x}_f^i, \mathbf{t}_f^i, \theta))^2. \tag{5.6}$$

In the above equations, $\{\mathbf{x}_0^i, \mathbf{t}_0^i\}_{i=1}^{N_0}$ denote the initial training data, $\{\mathbf{x}_b^i, \mathbf{t}_b^i\}_{i=1}^{N_b}$ are the boundary training data, and $\{\mathbf{x}_f^i, \mathbf{t}_f^i\}_{i=1}^{N_f}$ are the collocation points in the domain Ω. The baseline PINNs are described as above.

5.1.2 PINNs with fixed weight

A fixed weight loss function in PINNs is essential for balancing the multiple objectives of data fidelity and adherence to physical laws. While it provides simplicity and stability, the choice of weights needs careful tuning to ensure optimal performance. By integrating these components effectively, PINNs can solve complex scientific and engineering problems with high accuracy.

$$\mathscr{L}_\theta = \lambda_0\mathscr{L}_0(\theta) + \lambda_b\mathscr{L}_b(\theta) + \lambda_f\mathscr{L}_f(\theta), \tag{5.7}$$

where the fixed weights $\{\lambda_0, \lambda_b, \lambda_f\}$determine the contribution of each component, balancing the data fidelity and adherence to physical laws. The largest drawback of this method is difficult to find the appropriate constant weights. This procedure may require a lot of experiments. To overcome this problem, we applied the softmax function to the loss function.

5.1.3 Self-Adaptive PINNs

The PINN algorithm is remarkably successful in approximating solutions of many well-posed PDEs. Consider the loss function given below:

$$\mathscr{L}(\theta) = \mathscr{L}_u(\theta) + \mathscr{L}_r(\theta) + \mathscr{L}_b(\theta) + \mathscr{L}_0(\theta), \tag{5.8}$$

where $\mathscr{L}_u(\theta)$, $\mathscr{L}_r(\theta)$, $\mathscr{L}_u(\theta)$ and $\mathscr{L}_0(\theta)$ force the output to satisfy the sample data points, PDEs, the boundary condition, and the initial condition respectively. As mentioned before, the PINN fails to converge to the solutions because the gradient descent is easy to create an imbalance in the rate of descent on the different parts of the loss function resulting in inaccurate convergence. To address this issue, SA-PINN puts trainable weights in the loss function as follows:

$$\mathscr{L}(\theta, \lambda_r, \lambda_b, \lambda_0) = \mathscr{L}_u(\theta) + \mathscr{L}_r(\theta, \lambda_r) + \mathscr{L}_0(\theta, \lambda_0) + \mathscr{L}_b(\theta, \lambda_b), \tag{5.9}$$

where $\lambda_r = (\lambda_r^1, \ldots, \lambda^{N_r})$, $\lambda_0 = (\lambda_0^1, \ldots, \lambda_0^{N_0})$, and $\lambda_r = (\lambda_b^1, \ldots, \lambda_b^{N_b})$. N_0, N_b and N_r are the numbers of initial data points, boundary data points, and the residual data points.

$$\mathscr{L}_0(\theta, \lambda_0) = \frac{1}{N_0} \sum_{i=1}^{N_0} m(\lambda_0) |\mathscr{N}(x_0^i, 0; \theta) - u_0(x^i)|^2, \tag{5.10}$$

$$\mathscr{L}_b(\theta, \lambda_b) = \frac{1}{N_b} \sum_{i=1}^{N_b} m(\lambda_b) |\mathscr{N}(x_b^i, t; \theta) - g(x_b^i)|^2, \tag{5.11}$$

$$\mathscr{L}_r(\theta, \lambda_r) = \frac{1}{N_r} \sum_{i=1}^{N_r} m(\lambda_r) |\mathscr{N}(x_r^i, t; \theta) - f(x_r^i)|^2, \tag{5.12}$$

where the $m(\lambda)$ is defined as a self-adaptation mask function on $[0, \infty)$, which is a differentiable, non-negative, strictly increasing function of λ [49]. The choice of the mask function is supposed to be selected carefully to avoid the numerical overflow by keep these weights values below suitable values. The loss function $\mathscr{L}(\theta, \lambda_r, \lambda_b, \lambda_0)$ will minimize the network weights θ

but maximize the self-adaptation weights $\lambda_0, \lambda_r, \lambda_b$. Then, we have

$$\theta^{j+1} = \theta^j - \rho_\theta \nabla_\theta \mathscr{L}(\theta^j, \lambda_r^j, \lambda_b^j, \lambda_0^j), \tag{5.13}$$

$$\lambda_0^{j+1} = \lambda_0^j + \rho_0 \nabla_0 \mathscr{L}(\theta^j, \lambda_r^j, \lambda_b^j, \lambda_0^j), \tag{5.14}$$

$$\lambda_b^{j+1} = \lambda_b^j + \rho_b \nabla_b \mathscr{L}(\theta^j, \lambda_r^j, \lambda_b^j, \lambda_0^j), \tag{5.15}$$

$$\lambda_r^{j+1} = \lambda_r^j + \rho_r \nabla_r \mathscr{L}(\theta^j, \lambda_r^j, \lambda_b^j, \lambda_0^j), \tag{5.16}$$

where $\rho_\theta, \rho_0, \rho_b, \rho_r$ are learning rates. More details about SA-PINN can be obtained in [49].

5.1.4 Self-Learning PINNs

In this section, we introduce another adaptive weights method. Unlike the previous self-adaptive PINNs using the differentiable, non-negative strictly increasing function of weights, we utilize three separate network to learn the weights. Then, we have the loss function

$$\mathscr{L}(\theta, \theta_0, \theta_b, \theta_r) = \mathscr{L}_u(\theta) + \mathscr{L}_r(\theta, \lambda_r(\theta_r)) + \mathscr{L}_0(\theta, \lambda_0(\theta_0)) + \mathscr{L}_b(\theta, \lambda_b(\theta_b)), \tag{5.17}$$

where $\theta_0, \theta_b, \theta_r$ are the parameters of networks for learning the weights on initial, boundary, and PDEs residual loss respectively, and $\lambda_0(\theta_0), \lambda_b(\theta_b), \lambda_r(\theta_r)$ are the output of these networks. To avoid the loss decay to zero, we need to add some proper constraints. In this study, we apply the exponential function on them. Then, we have

$$\mathscr{L}_0(\theta, \theta_0, \lambda_0(\theta_0)) = \frac{1}{N_0} \sum_{i=1}^{N_0} e^{\lambda_0^i(\theta_0)} |\mathscr{N}(x_0^i, t_0; \theta) - u_0(x_0^i)|^2, \tag{5.18}$$

$$\mathscr{L}_b(\theta, \theta_b, \lambda_b(\theta_b)) = \frac{1}{N_b} \sum_{i=1}^{N_b} e^{\lambda_b^i(\theta_b)} |\mathscr{N}(x_b^i, t_b; \theta) - g(x_b^i)|^2, \tag{5.19}$$

$$\mathscr{L}_b(\theta, \theta_r, \lambda_r(\theta_r)) = \frac{1}{N_r} \sum_{i=1}^{N_r} e^{\lambda_r^i(\theta_r)} |\mathscr{N}(x_r^i, t_r; \theta) - f(x_r^i)|^2, \tag{5.20}$$

The parameters $\theta, \theta_0, \theta_b, \theta_r$ can be learned by minimizing the loss function Eq.(5.17). Also, to avoid the $e^{\lambda(\theta)}$ decay to zero, we propose the LeakyRelu function on output layers of weights networks and set minimum values for the weights. This setup prevents the negative weight from becoming too small, ensuring that $e^{\lambda(\theta)}$ does not decay to extremely low values. The inputs

of these weights networks are uniformly generated numbers between 0 and 1. The activation function of the hidden layers are Hyperbolic Tangent function.

Algorithm 5 Self-Learning PINN for Solving PDEs

Input: x, t;

Output: Approximated solutions of PDEs: $\bar{u}$.

1: Construct the main FNN, and randomly initialize the parameters θ.

2: Construct three weights networks, and randomly initialize the parameters θ_0, θ_b, θ_r.

3: Uniformly generate the inputs for weights networks.

4: Forward propagation: the main FNN output $\bar{u}$, the weight networks output the $\lambda_0, \lambda_b, \lambda_r$.

5: Construct the loss function Eq.(5.17).

6: Minimize the loss function by Optimizers.

7: Generate the output $\bar{u}$

5.2 Numerical Experiments

In this section, we apply the above methods to the Burgers and generalized Burgers-Fisher equation. The performance of the above methods is compared. Similar to previous sections, we propose the Adam optimizer followed by L-BFGS to minimize the loss function.

5.2.1 Burgers' Equation

Consider the Burgers' equation

$$\begin{aligned} &u_t + uu_x - \nu u_{xx} = 0, x \in [0,1], t \in [1,3.5], \\ &u(1,x) = \frac{x}{1 + e^{\frac{x^2 - 0.25}{4\nu}}}, \\ &u(t,0) = u(t,1) = 0. \end{aligned} \tag{5.21}$$

Correspondingly, the exact solution of Eq.(5.21) is

$$u(x,t) = \frac{x/t}{1 + e^{\frac{x^2}{4\nu t}} \sqrt{\frac{t}{e^{1/8\nu}}}}, \tag{5.22}$$

where $\nu = \frac{1}{Re}$, and we set $\nu = 0.0005$ in our case [9]. We utilize the above four methods to solve this equation. The main architecture is a feed-forward neural network with 6 hidden layers and 120 neurons for each layer. As for the self-learning PINNs, three weight neural networks are fully connected neural networks with 3 hidden layers and 60 neurons for each layer. The activation functions are the Hyperbolic Tangent function, and the LeakyReLU function, the negative slope is 0.001, is applied to the output layers of weight networks. The loss function is constructed as previously described.

$$\mathscr{L}(\theta) = \mathscr{L}_0(\theta, \lambda_0) + \mathscr{L}_b(\theta, \lambda_b) + \mathscr{L}_r(\theta, \lambda_r), \tag{5.23}$$

where

$$\mathscr{L}_0(\theta, \lambda_0) = \frac{1}{N_0} \sum_{i=1}^{N_0} e^{\lambda_0^i} (\bar{u}(1, x_0^i) - u(1, x_0^i))^2, \tag{5.24}$$

$$\mathscr{L}_b(\theta, \lambda_b) = \frac{1}{N_b} \sum_{i=1}^{N_b} e^{\lambda_b^i} ((\bar{u}(t_b^i, 0) - 0))^2 + \bar{u}(t_b^i, 1) - 0))^2), \tag{5.25}$$

and

$$\mathscr{L}_r(\theta, \lambda_r) = \frac{1}{N_r} \sum_{i=1}^{N_r} e^{\lambda_r^i} (\bar{u}_t + \bar{u}\bar{u}_x - 0.0005\bar{u}_{xx})^2. \tag{5.26}$$

We set $N_0 = 1000$, $N_b = 1000$, and $N_r = 10000$. When $\lambda_0 = \lambda_b = \lambda_r = 1$, then Eq.(5.23) is the loss function of regular PINN. As for the fixed weights method, we set $\lambda_0 = 1$, $\lambda_r = 20$, and $\lambda_b = 1$. In self-adaptive PINN, the mask function is $m(\lambda) = \lambda^2$. The model is trained for 1500 epochs using the Adam optimizer with a learning rate of 0.001, then for another 1500 epochs using the Adam optimizer with a learning rate of 0.005, and finally for 2000 epochs using the L-BFGS optimizer. Fig.5.1 presents the snapshots of the predicted solution of self-learning PINN and the exact solution. The Table 5.1 shows the comparison of L_2 errors of these four methods. We can see that all methods perform well and obtain the relatively small L_2 errors. The self-learning PINN performs slightly better compared to other three methods.

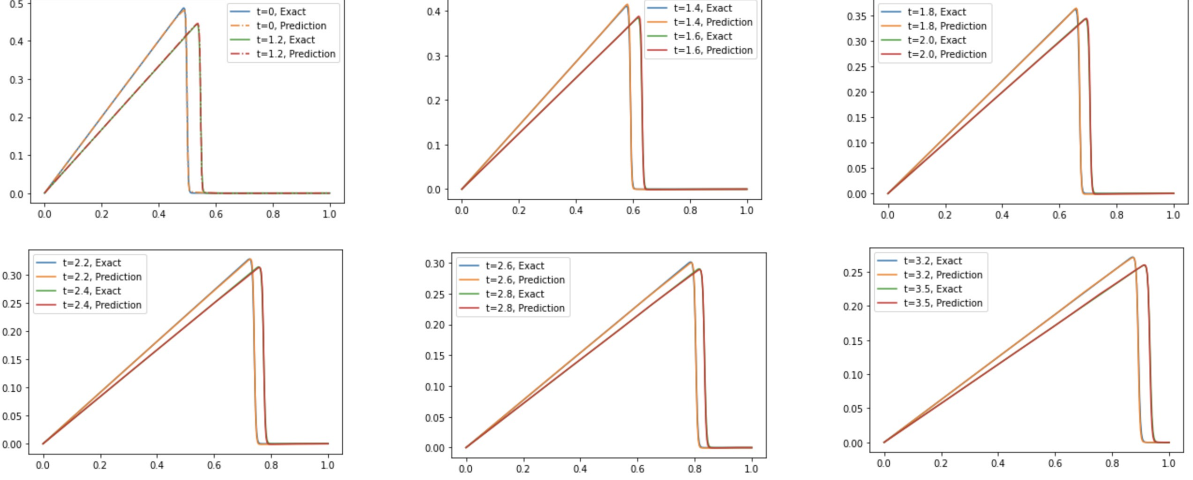


Figure 5.1: Self-Learning PINN for solving Burgers' Equation

Method	L_2 Errors
PINNs	3.8e-3
Fixed Weights	3.9e-3
Self-Adaptive PINN	4.2e-3
Self-Learning PINN	3.21e-3

Table 5.1: Relative L_2 errors of four methods for Burgers Equation

5.2.2 Burgers-Fisher Equation

To show the difference between these four methods, we applied them to solve the generalized Burgers-Fisher's equation [2].

$$\frac{\partial u}{\partial t} + \alpha u^{\delta}\frac{\partial u}{\partial x} = \frac{\partial^2 u}{\partial x^2} + \beta u(1-u^{\delta}), (x,t) \in [0,1]\times[0,T], \tag{5.27}$$

$$u(x,0) = (\frac{1}{2} - \frac{1}{2}tanh(\frac{-\alpha\delta}{2(\delta+1)}x))^{1/\delta}, x \in [0,1]. \tag{5.28}$$

The exact solution to Eq.(5.27) and Eq.(5.28) is given as below:

$$u(x,t) = (\frac{1}{2} + \frac{1}{2}tanh[\frac{-\alpha\delta}{2(\delta+1)}(x - (\frac{\alpha}{\delta+1} + \frac{\beta(\delta+1)}{\alpha})t])^{1/\delta} \tag{5.29}$$

For simplicity, we set the $\delta = 1$, $\alpha = 5$, and $\beta = 1$. We continue to use the Feedforward Neural Network (FNN) as the primary architecture, maintaining the same number of layers

and neurons as in the previous example. For the self-learning PINN, the negative slope of LeakyReLU is set to 0.01. The boundary condition can be extracted from the exact solution. Similarly, we have the loss function.

$$\mathscr{L}_0(\theta,\lambda_0) = \frac{1}{N_0}\sum_{i=1}^{N_0} e^{\lambda_0^i}(\bar{u}(1,x_0^i) - u(1,x_0^i))^2, \tag{5.30}$$

$$\mathscr{L}_b(\theta,\lambda_b) = \frac{1}{N_b}\sum_{i=1}^{N_b} e^{\lambda_0^i}((\bar{u}(t_b^i,0) - u(t_b^i,0))^2 + (\bar{u}(t_b^i,1) - u(t_b^i,0))^2), \tag{5.31}$$

$$\mathscr{L}_r(\theta,\lambda_r) = \frac{1}{N_r}\sum_{i=1}^{N_r} e^{\lambda_r^i}(\bar{u}_t + 5\bar{u}\bar{u}_x - \bar{u}_{xx} - \bar{u}(1-\bar{u}))^2. \tag{5.32}$$

Similarly, $N_0 = 1000$, $N_b = 1000$, and $N_r = 10000$. The model undergoes training for 5000 epochs with the Adam optimizer at a learning rate of 0.001, followed by an additional 5000 epochs using the Adam optimizer at a learning rate of 0.005, and completes with 2000 epochs using the L-BFGS optimizer. Fig5.2 shows snapshots of the predicted solution from the self-learning PINN alongside the exact solution, demonstrating that the results are highly accurate and closely match the exact solution. Table 5.2 presents a comparison between the four methods. It shows that the self-learning method achieves a relative L_2 error that is 15 times lower than that of the self-adaptive PINNs and performs significantly better than the other two methods.

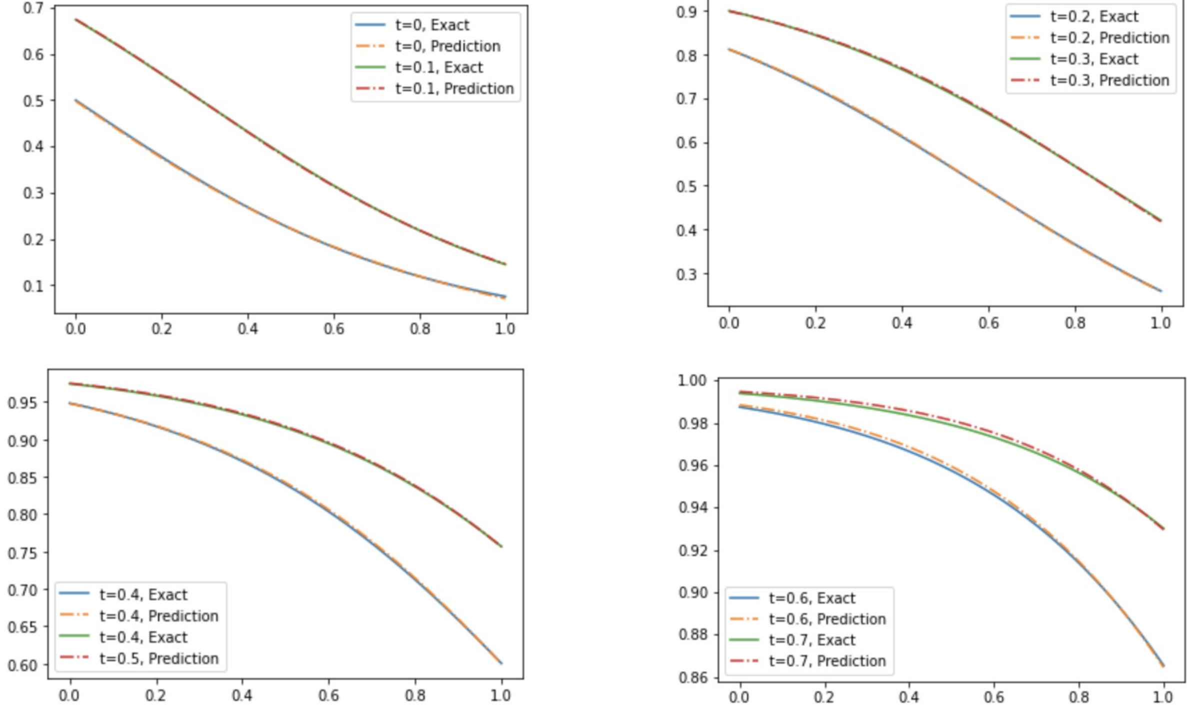


Figure 5.2: Self-Learning PINN Solving Burgers-Fisher equation

Method	L_2 Errors
PINNs	1.1e-1
Fixed Weights	8.2e-2
Self-Adaptive PINN	2.1e-2
Self-Learning PINN	1.4e-3

Table 5.2: Relative L_2 errors of four methods for Burgers-Fisher equation

5.3 Conclusion

In this chapter, we introduced the self-learning PINN to solve the Burgers' and Burgers-Fisher equations. The weights were entirely learned by three smaller fully-connected neural networks. To prevent the entire loss function from decaying to zero, we incorporated exponential functions and LeakyReLU in this method. We compared this approach to regular PINN, fixed weights PINN, and self-adaptive PINN. The self-learning PINN outperformed all three methods. Additionally, while tuning hyperparameters is crucial for enhancing performance in these methods, the self-learning approach can mitigate this issue.

CHAPTER 6

Conclusion and Future Research

Deep learning algorithms have demonstrated significant promise in solving partial differential equations. Firstly, we proposed a variant of the physics-informed neural network to identify time-varying parameters of the Susceptible-Infectious-Recovered-Deceased model for COVID-19 by fitting daily reported cases. The learned parameters were validated using an ordinary differential equation solver, and the effective reproduction number was calculated. Furthermore, we explored solving differential equations using sparse data by combining a neural network with a numerical approach to address time-dependent partial differential equations. The Gated Recurrent Units network estimated time iteration schemes while integrating prior knowledge of governing equations. A numerical implicit approach calculated new time iteration schemes, and the loss function incorporated the differences between these schemes. We also proposed a novel physics-informed encoder-decoder gated recurrent neural network to solve time-dependent partial differential equations without using any observed data. This method effectively approximated solutions over the entire spatio-temporal domain. The effectiveness of these methods was validated through applications to various problems.

The regular physics-informed neural networks fail to converge to solutions due to the imbalance in the rate of descent on the different parts of the loss function resulting in inaccurate convergence. To address this problem, we introduced the Self-Learning Physics-Informed Neural Network. This method learns weights through separate neural networks to mitigate the imbalance problem in multi-component loss function and eliminate the need for hyper-parameter fine-tuning. The effectiveness of our approach was demonstrated on the Burgers' and Burgers-Fisher equations, highlighting its potential for improving the convergence and performance of physics-informed models.

The above methods are created for our future projects. Firstly, we will focus on the irregular boundary problems by incorporating the graph neural network into the physics-informed encoder-decoder gated recurrent neural network. We partition the domain into an irregular mesh and treat this mesh as an undirected graph for training the network. Specifically, we rep-

resent the grid points as graph nodes and assign edges between nodes that are nearest neighbors. The encoder transforms these nodes and edges to proper features, enabling the Graph Neural Network to predict latent feature variations of the nodes. Finally, the decoder approximates the final solution. Secondly, we will propose and develop a self-learning method to tackle stiff problems. We expect this method to learn the characteristics of stiffness and handle it in an appropriate way, improving the accuracy and stability of the solutions in such challenging scenarios.

BIBLIOGRAPHY

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

[2] Timilehin Kingsley Akinfe and Adedapo Chris Loyinmi. A solitary wave solution to the generalized burgers-fisher's equation using an improved differential transform method: A hybrid scheme approach. *Heliyon*, 7(5):e07001, 2021.

[3] Shun-Ichi Amari. Backpropagation and stochastic gradient descent method. *Neurocomputing*, 5(4-5):185–196, 1993.

[4] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[5] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of Marchine Learning Research*, 18:1–43, 2018.

[6] Umair bin Waheed, Ehsan Haghighat, Tariq Alkhalifah, Chao Song, and Qi Hao. Pinneik: Eikonal solution using physics-informed neural networks. *Computers & Geosciences*, 155:104833, 2021.

[7] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010: 19th International Conference on Computational StatisticsParis France, August 22-27, 2010 Keynote, Invited and Contributed Papers*, pages 177–186. Springer, 2010.

[8] José M Carcione, Juan E Santos, Claudio Bagaini, and Jing Ba. A simulation of a COVID-19 epidemic based on a deterministic SEIR model. *arXiv:2004.03575*, 2020.

[9] Süleyman Cengizci and Ömür Uğur. A stabilized fem formulation with discontinuity-capturing for solving burgers'-type equations at high reynolds numbers. *Applied Mathematics and Computation*, 442:127705, 2023.

[10] Yi-Cheng Chen, Ping-En Lu, Cheng-Shang Chang, and Tzu-Hsuan Liu. A Time-dependent SIR model for COVID-19 with undetectable infected persons. *IEEE Transactions on Network Science and Engineering*, 7(4):3279–3294, 2020.

[11] Kyunghyun Cho, Bart van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder–decoder approaches. In Dekai Wu, Marine Carpuat, Xavier Carreras, and Eva Maria Vecchi, editors, *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*, pages 103–111, Doha, Qatar, October 2014. Association for Computational Linguistics.

[12] Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder–decoder approaches. In *SSST@EMNLP*, 2014.

[13] Francois Chollet. *Deep learning with Python*. Simon and Schuster, 2021.

[14] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[15] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[16] Gouri Dhatt, Emmanuel Lefrançois, and Gilbert Touzot. *Finite element method*. John Wiley & Sons, 2012.

[17] Sergei E Esipov. Coupled burgers equations: a model of polydispersive sedimentation. *Physical Review E*, 52(4):3711, 1995.

[18] Robert Eymard, Thierry Gallouët, and Raphaèle Herbin. Finite volume methods. *Handbook of numerical analysis*, 7:713–1018, 2000.

[19] Hongxiang Fan, Mingliang Jiang, Ligang Xu, Hua Zhu, Junxiang Cheng, and Jiahu Jiang. Comparison of Long Short Term Memory Networks and the Hydrological Model in Runoff Simulation. *Water*, 12(1):175, 2020.

[20] Luisa Ferrari, Giuseppe Gerardi, Giancarlo Manzi, Alessandra Micheletti, Federica Nicolussi, and Silvia Salini. Modelling provincial COVID-19 epidemic data in Italy using an adjusted time-dependent SIRD model. *arXiv:2005.12170*, 2020.

[21] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[22] Vignesh Gopakumar, Stanislas Pamela, and Debasmita Samaddar. Loss landscape engineering via data regulation on pinns. *Machine Learning with Applications*, page 100464, 2023.

[23] Alex Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

[24] Donald L Gray and Anthony N Michel. A training algorithm for binary feedforward neural networks. *IEEE Transactions on Neural Networks*, 3(2):176–194, 1992.

[25] The Atlantic Monthly Group. The COVID Tracking Project, 2020.

[26] Ding Guanghong, Liu Chang, Gong Jianqiu, Wang Ling, Cheng Ke, and Zhang Di. SARS epidemical forecast research in mathematical model. *Chinese Science Bulletin*, 49(21):2332–2338, 2004.

[27] Ernst Hairer, Syvert Paul Nørsett, and Gerhard Wanner. Solving ordinary differential equations. i: Nonstiff problems. 3rd corrected printing. *Springer Series in Computational Mathematics*, 8:528, 2010.

[28] Richard J Hatchett, Carter E Mecher, and Marc Lipsitch. Public health interventions and epidemic intensity during the 1918 influenza pandemic. *Proceedings of the National Academy of Sciences*, 104(18):7582–7587, 2007.

[29] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107–116, 1998.

[30] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[31] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[32] Hyeontae Jo, Hwijae Son, Se Young Jung, and Hyung Ju Hwang. Analysis of COVID-19 spread in South Korea using the SIR model with time-dependent parameters and deep learning. *medRxiv*, 2020.

[33] Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever. An empirical exploration of recurrent network architectures. In *International conference on machine learning*, pages 2342–2350. PMLR, 2015.

[34] George Em Karniadakis, Ioannis G Kevrekidis, Lu Lu, Paris Perdikaris, Sifan Wang, and Liu Yang. Physics-informed machine learning. *Nature Reviews Physics*, 3(6):422–440, 2021.

[35] Andreas Kergassner, Christian Burkhardt, Dorothee Lippold, Sarah Nistler, Matthias Kergassner, Paul Steinmann, Dominik Budday, and Silvia Budday. Meso-scale modeling of COVID-19 spatio-temporal outbreak dynamics in Germany. *medRxiv*, 2020.

[36] William Ogilvy Kermack and Anderson G McKendrick. A contribution to the mathematical theory of epidemics. *Proceedings of the Royal Society of London. Series A, Containing papers of a mathematical and physical character*, 115(772):700–721, 1927.

[37] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[38] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[39] Isaac E Lagaris, Aristidis Likas, and Dimitrios I Fotiadis. Artificial neural networks for solving ordinary and partial differential equations. *IEEE transactions on neural networks*, 9(5):987–1000, 1998.

[40] Quoc V Le, Navdeep Jaitly, and Geoffrey E Hinton. A simple way to initialize recurrent networks of rectified linear units. *arXiv preprint arXiv:1504.00941*, 2015.

[41] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[42] Ying Li, Zuojia Zhou, and Shihui Ying. Delisa: Deep learning based iteration scheme approximation for solving pdes. *Journal of Computational Physics*, 451:110884, 2022.

[43] Qianying Lin, Shi Zhao, Daozhou Gao, Yijun Lou, Shu Yang, Salihu S Musa, Maggie H Wang, Yongli Cai, Weiming Wang, Lin Yang, et al. A conceptual model for the outbreak of Coronavirus disease 2019 (COVID-19) in Wuhan, China with individual reaction and governmental action. *International Journal of Infectious Diseases*, 93:211–216, 2020.

[44] Dong C Liu and Jorge Nocedal. On the limited memory bfgs method for large scale optimization. *Mathematical programming*, 45(1-3):503–528, 1989.

[45] Zhihua Liu, Pierre Magal, Ousmane Seydi, and Glenn Webb. Predicting the cumulative number of cases for the COVID-19 epidemic in China from early data. *Mathematical Biosciences and Engineering*, 17:3040–3051, 2020.

[46] Zichao Long, Yiping Lu, Xianzhong Ma, and Bin Dong. Pde-net: Learning pdes from data. In *International conference on machine learning*, pages 3208–3216. PMLR, 2018.

[47] Lu Lu, Xuhui Meng, Zhiping Mao, and George E Karniadakis. DeepXDE: A deep learning library for solving differential equations. *arXiv:1907.04502v2*, 2020.

[48] Luca Magri and Nguyen Anh Khoa Doan. First-principles machine learning modelling of COVID-19. *arXiv:2004.09478*, 2020.

[49] Levi McClenny and Ulisses Braga-Neto. Self-adaptive physics-informed neural networks using a soft attention mechanism. *arXiv preprint arXiv:2009.04544*, 2020.

[50] Levi D McClenny and Ulisses M Braga-Neto. Self-adaptive physics-informed neural networks. *Journal of Computational Physics*, 474:111722, 2023.

[51] Jorge Nocedal and Stephen J Wright. *Numerical optimization*. Springer, 1999.

[52] Daniel W Otter, Julian R Medina, and Jugal K Kalita. A survey of the usages of deep learning for natural language processing. *IEEE transactions on neural networks and learning systems*, 32(2):604–624, 2020.

[53] Houman Owhadi. Bayesian numerical homogenization. *Multiscale Modeling & Simulation*, 13(3):812–828, 2015.

[54] M Necati Özişik, Helcio RB Orlande, Marcelo J Colaço, and Renato M Cotta. *Finite difference methods in heat transfer*. CRC press, 2017.

[55] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[56] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[57] Dimitris C Psichogios and Lyle H Ungar. A hybrid neural network-first principles approach to process modeling. *AIChE Journal*, 38(10):1499–1511, 1992.

[58] Muhammad Rafiq, JE Macías-Díaz, Ali Raza, and Nauman Ahmed. Design of a nonlinear model for the propagation of COVID-19 and its efficient nonstandard computational implementation. *Applied Mathematical Modelling*, 89:1835–1846, 2021.

[59] Maziar Raissi and George Em Karniadakis. Hidden physics models: Machine learning of nonlinear partial differential equations. *Journal of Computational Physics*, 357:125–141, 2018.

[60] Maziar Raissi, Paris Perdikaris, and George E Karniadakis. 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, 2019.

[61] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Inferring solutions of differential equations using noisy multi-fidelity data. *Journal of Computational Physics*, 335:736–746, 2017.

[62] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Machine learning of linear differential equations using gaussian processes. *Journal of Computational Physics*, 348:683–693, 2017.

[63] Maziar Raissi, Niloofar Ramezani, and Padmanabhan Seshaiyer. On parameter estimation approaches for predicting disease transmission through optimization, deep learning and statistical inference methods. *Letters in Biomathematics*, 6(2):1–26, 2019.

[64] Waseem Rawat and Zenghui Wang. Deep convolutional neural networks for image classification: A comprehensive review. *Neural computation*, 29(9):2352–2449, 2017.

[65] Benjamin Ridenhour, Jessica M Kowalik, and David K Shay. Unraveling R_0: Considerations for Public Health Applications. *American Journal of Public Health*, 108(S6):S445–S454, 2018.

[66] Ruben Rodriguez-Torrado, Pablo Ruiz, Luis Cueto-Felgueroso, Michael Cerny Green, Tyler Friesen, Sebastien Matringe, and Julian Togelius. Physics-informed attention-based neural network for hyperbolic partial differential equations: application to the buckley–leverett problem. *Scientific reports*, 12(1):7557, 2022.

[67] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[68] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[69] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115:211–252, 2015.

[70] Farah Shahid, Aneela Zameer, and Muhammad Muneeb. Predictions for COVID-19 with deep learning models of LSTM, GRU and Bi-LSTM. *Chaos, Solitons & Fractals*, 140:110212, 2020.

[71] Yaolin Shi. Stochastic dynamic model of SARS spreading. *Chinese Science Bulletin*, 48(13):1287–1292, 2003.

[72] Yeonjong Shin, Jerome Darbon, and George Em Karniadakis. On the convergence of physics informed neural networks for linear second-order elliptic and parabolic type pdes. *arXiv:2004.01806*, 2020.

[73] Justin Sirignano and Konstantinos Spiliopoulos. Dgm: A deep learning algorithm for solving partial differential equations. *Journal of Computational Physics*, 375:1339–1364, 2018.

[74] AA2206122 Soliman. The modified extended tanh-function method for solving burgers-type equations. *Physica A: Statistical Mechanics and its Applications*, 361(2):394–404, 2006.

[75] Samuel O Sowole, Daouda Sangare, Abdullahi A Ibrahim, and Isaac A Paul. On the existence, uniqueness, stability of solution and numerical simulations of a mathematical model for measles disease. *International journal of advances in mathematics*, 4:84–111, 2019.

[76] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27, 2014.

[77] James William Thomas. *Numerical partial differential equations: finite difference methods*, volume 22. Springer Science & Business Media, 2013.

[78] Pauline van den Driessche. Reproduction numbers of infectious disease models. *Infectious Disease Modelling*, 2(3):288–303, 2017.

[79] Athanasios Voulodimos, Nikolaos Doulamis, Anastasios Doulamis, Eftychios Protopapadakis, et al. Deep learning for computer vision: A brief review. *Computational intelligence and neuroscience*, 2018, 2018.

[80] Huwen Wang, Zezhou Wang, Yinqiao Dong, Ruijie Chang, Chen Xu, Xiaoyue Yu, Shuxian Zhang, Lhakpa Tsamlag, Meili Shang, Jinyan Huang, et al. Phase-adjusted estimation of the number of coronavirus disease 2019 cases in Wuhan, China. *Cell Discovery*, 6(1):1–8, 2020.

[81] Jinfeng Wang, Anthony J McMichael, Bin Meng, Niels G Becker, Weiguo Han, Kathryn Glass, Jilei Wu, Xuhua Liu, Jiyuan Liu, Xiaowen Li, et al. Spatial dynamics of an epidemic of severe acute respiratory syndrome in an urban area. *Bulletin of the World Health Organization*, 84:965–968, 2006.

[82] Sifan Wang, Yujun Teng, and Paris Perdikaris. Understanding and mitigating gradient flow pathologies in physics-informed neural networks. *SIAM Journal on Scientific Computing*, 43(5):A3055–A3081, 2021.

[83] Sifan Wang, Xinling Yu, and Paris Perdikaris. When and why pinns fail to train: A neural tangent kernel perspective. *Journal of Computational Physics*, 449:110768, 2022.

[84] Colby L Wight and Jia Zhao. Solving allen-cahn and cahn-hilliard equations using the adaptive physics informed neural networks. *arXiv preprint arXiv:2007.04542*, 2020.

[85] Wikipedia contributors. Recurrent neural network, 2024. Accessed: 2024-06-12.

[86] World Health Organizaiton. Rolling updates on coronavirus disease (COVID-19), 2020. Updated 31 July 2020.

[87] Worldometers. COVID-19 CORONAVIRUS PANDEMIC, 2020.

[88] Zifeng Yang, Zhiqi Zeng, Ke Wang, Sook-San Wong, Wenhua Liang, Mark Zanin, Peng Liu, Xudong Cao, Zhongqiang Gao, Zhitong Mai, et al. Modified SEIR and AI prediction of the epidemics trend of COVID-19 in China under public health interventions. *Journal of Thoracic Disease*, 12(3):165, 2020.

[89] Abdelhafid Zeroual, Fouzi Harrou, Abdelkader Dairi, and Ying Sun. Deep learning methods for forecasting covid-19 time-series data: A comparative study. *Chaos, Solitons & Fractals*, 140:110121, 2020.

[90] Dongkun Zhang, Ling Guo, and George Em Karniadakis. Learning in modal space: Solving time-dependent stochastic pdes using physics-informed neural networks. *SIAM Journal on Scientific Computing*, 42(2):A639–A665, 2020.

[91] Dongkun Zhang, Lu Lu, Ling Guo, and George Em Karniadakis. Quantifying total uncertainty in physics-informed neural networks for solving forward and inverse stochastic problems. *Journal of Computational Physics*, 397:108850, 2019.