National University Ireland, Maynooth

Department of Computer Science

Ordinary Differential Equation based Recurrent Neural Network Models for Learning Continuous Time Series

by Mansura Habiba

A thesis presented in fulfillment of the requirements for the Degree of Doctor of Philosophy

Supervisor: Professor Barak A. Pearlmutter

June 13, 2022

ACKNOWLEDGEMENTS

Throughout the writing of this dissertation, I have received a great deal of support and assistance. I would first like to thank my supervisor, Professor Barak A. Pearlmutter, whose expertise was invaluable in formulating the research questions and methodology. Your insightful feedback pushed me to sharpen my thinking and brought my work to a higher level. In addition, you were very kind and patient with me throughout my four years of my PhD. You are the best supervisor for me.

I want to thank Niamh Dootson, PhD Student at Maynooth University; Mehrdad Maleki, Postdoctoral Research Fellow of the Irish Research Council in the Computer Science at Maynooth University; Eoin Brophy, Insight Centre for Data Analytics at Dublin City University and Tomás E. Ward Insight Centre for Data Analytics at Dublin City University for your excellent collaboration. I would like to express my gratitude to Professor John McDonald and Dr Joseph Timoney for their fantastic support throughout four years at Maynooth University. You were always kind and helped me to go through different complex situations.

I want to thank my managers. Wayne Leone, and Eoin O'Neil, IBM in Ireland, for your continuous support.

In addition, I would like to thank my parents, my sisters and my brother for their wise counsel and sympathetic ear. Finally, I could not have completed this dissertation without the support of my friends, colleagues and teammates, who provided continuous support through stimulating discussions as well as happy distractions to rest my mind outside of my research.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION AND BACKGROUND

The most significant challenge in continuous time series modelling using deep learning is predicting a shorter time step and variable sampling rate in real-valued time series data. For example, each sensor sends continuous-time data at different sampling rates and short time steps in processing sensor data. A neural network needs numerous amounts of computation and power consumption to predict the outcome. In this case, the network needs to compute the output at every step. The computation for each step itself is an expensive operation in memory, computation and performance. Another relevant challenge in time series modelling is real-time prediction. The existing neural networks used for time series modelling are mostly discrete dynamic systems. The computation takes place at a constant time step with an optimised gap. These network models also have a different mechanism for skipping a step to optimise the computation. For example, the Phased LSTM model [1] uses a time gate in order to distinguish between phases when incoming data is available or not. Skip- RNN [2] computes several skipped time steps ahead of time and skips their computation within a budget constraint. The Gru-D [3] mdoel incorporates two different representations of missing patterns, i.e., masking and time interval, into a deep model. These models have a substantial success rate in different use cases, but they have an architecture of a discrete system dynamic, so they still have significant limitations for real-time prediction. These models consider continuous time series as a discrete sequence of observations. This existing approach is not very suitable for several real-time data-driven applications. These constraints on constant time steps also significantly impact efficiency and precision.

Recent deep learning problems are primarily data-driven and continuous. For example, industries like automobiles, energy, finance, and the internet of things (IoT) analyses real-time data continuously. The system collects data from different sources at different dynamic sampling rates. These collected data need to be sampled at irregular sampling rates with higher frequency for accuracy and efficiency. Modelling real-time data using a continuous-time sequence is a very challenging task. Different research works have tried to solve different challenges of modelling continuous time series using different neural network models and processing data and learning approaches. In this regard, Recurrent Neural Network (RNN) has become a state of art choice for solving several challenges and complexities in time sequence analysis tasks, e.g., short-term forecasting, time series analysis, continuous time series processing etc. [1]. Several ongoing types of research [1, 2, 4, 5] on RNN have demonstrated its impressive efficiency in analysing temporal sequences with a fixed data sampling rate. However, the existing deep learning models are not free from challenges and limitations. Due to diversity among different attributes, behaviour, duration of steps, energy, data sampling rate and other challenges, modelling continuous-time data using these models is challenging.

This work aims to design a neural network model that can overcome the limitation of constant time steps and process data in real-time. This model can compute within the marginal budget cost with shorter time steps and accurate timing. It can also process data at an irregular sampling rate with a higher frequency.

1.1 Problem Domain

In a traditional residual neural network, most of the time, two consecutive cell states are related. Although each cell state is computed using the previous state and input from the previous state, it is necessary to establish a relation among the consecutive time steps. Eq. (1.1) describes a residual neural network, where f is the update function of the n-th state and its activation.

$$y_{n+1} = y_n + f(y_n) \tag{1.1}$$

The time interval between two consecutive time steps can be important for modelling the system and the subsequent prediction at the k-th time step. In addition, each of the k-th time steps may aim at different goals [1]. Therefore, along with the previous cell state, it is also essential to consider the time interval between the current and previous time step while computing the current cell state. However, the computation overload and the energy consumption are enormous for a neural network. Furthermore, the traditional residual neural network also has some architectural limitations. For example, Fig. [1.1] shows the typical architecture of a neural network which is a series of layers. Each of the layers is a matrix operation that possesses some error. Moreover, each of the hidden states propagates to the next layer. Therefore, To design a neural network, it is required to determine the number of layers, as the number of layers and the number of neurons impact the output significantly.

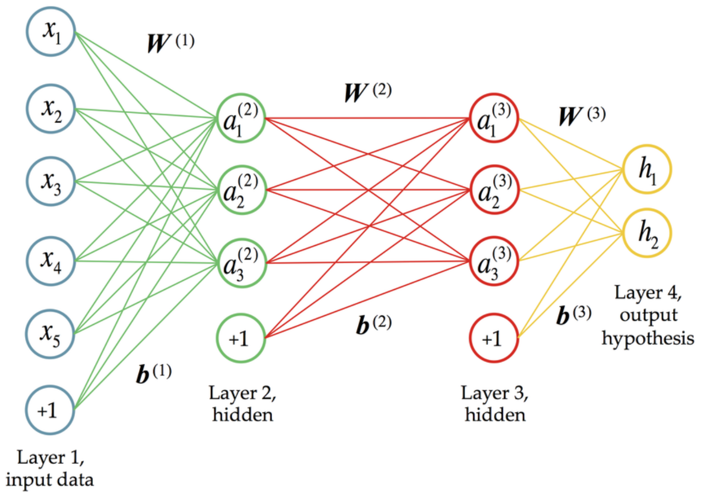


Figure 1.1: A general neural network and its layers

Fig [1.2] describes the influence of several hidden layers on the final output of a neural network. If the number of layers is too small, it may cause under-fitting; on the other hand, too many layers would cause computation overhead, longer training time, and even an over-fitted output. Therefore, in the case of designing a neural network, it is a significant factor to choose the number of layers correctly.

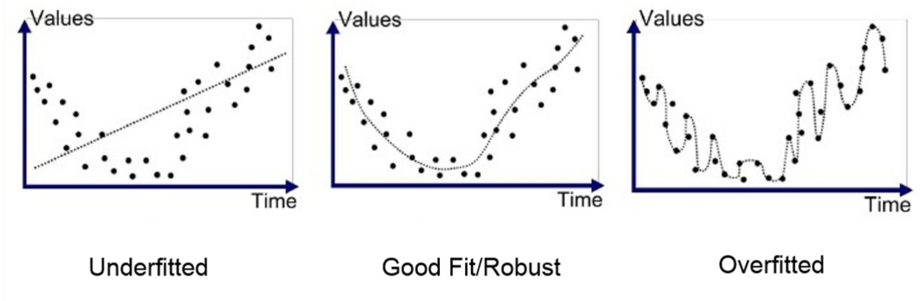


Figure 1.2: Influence of the number of layers of a neural network

On the other hand, RNN and its different variations are usually a discrete system model that accepts input at a constant time step. This limitation may lead to inaccurate prediction for irregular sampling rates, higher frequency, and dynamic input systems. In addition, modifying or re-designing any neural network is often not straightforward. Every problem needs to design for the architecture corresponding network.

Recent work [6] has proposed that the activation function of the neural network, as described in Eq. (1.1), can be depicted as Euler discretization of continuous transformations, as shown in Eq. (1.2).

$$y_{n+1} = y_n + hf\left(y_n\right) \tag{1.2}$$

Neural ODE [7] proposed that a neural network can be defined as a differential equation by parametrising the continuous dynamics of hidden states using an ordinary differential equation. Eq. (1.3) represents the dynamics of hidden state as an ordinary differential equation.

$$\frac{d(h_t)}{dt} = f\left(h\left(t\right), t, \delta\right) \tag{1.3}$$

Neural ODE introduces a whole new family of ordinary neural networks. Neural ODE can compute at an accurate time with fewer parameters and constant memory cost. Furthermore, this model does not require backpropagating through the solver's operations; instead, an ordinary differential equation (ODE) solver computes the state at any time t and helps the gradient train itself with marginal budget cost.

1.2 Research Question

The main research question for this work is finding a method that can model the continuous temporal process using deep learning as a continuous function of time. This method would overcome the limitation of missing patterns, vanishing gradient, higher sampling rate and efficiency. Moreover, this method will use a minimum number of parameters and a faster training time.

Recurrent Neural Network is one of the pioneer deep learning models for realistic high-dimensional time-series prediction tasks, and it has some pitfalls. This PhD work mainly focuses on leveraging ordinary differential equation capability to design recurrent neural network architecture (RNN). The main goal of this work is to answer the following research questions:

- Q1. What are the significant challenges in the domain of continuous-time series modelling?
- Q2. How to leverage the functionality of the ODE solver in deep learning with higher efficiency?
- Q3. Can the two widely used recurrent neural networks, e.g., GRU and LSTM,

be modelled using ordinary deep neural networks?

- Q4. Can ODE base deep neural network overcome the limitations of missing patterns in continuous time series learning?
- Q5. Can ODE based deep neural network solve the following problems of deep neural network (i) irregular sampling rate, (ii) short time steps, and (iii) high sampling frequencies?
- Q6. How to improve the influential feature selection for deep learning models?

Chapter 2 conveyed a detailed survey on the characteristics of time series and the challenges and limitations of existing models for continuous-time series modelling.

To tackle these questions, we first propose a neural ordinary-based GRU model that can leverage the capabilities of an ordinary differential equation (ODE) solver to compute the update functions and train the model. The main contribution of this work is the development of a formal model architecture for GRU with ordinary differential equations and evaluating its performance for different optimiser and loss functions. Then, we implemented the algorithm emerging from this theory in Python and used the HIPS autograd and PyTorch framework.In Section 2.1, this proposed model is formally introduced along with performance evaluation results. The performance evaluation of this proposed model would answer the questions mentioned above. We plan to design a similar ODE-based model for the LSTM unit for our future work.

1.3 Motivation

Most existing deep learning models represent continuous-time data as a sequence of discrete events at a fixed time step. However, a time series is mainly a sequence of observations, and it is not a series of time points with related events at those time points. These models are better suited for sequential data where the time step between two consecutive observations does not affect the model's efficiency or the time step is permanently fixed. However, for time data in most real-time applications such as Electronic Health Records (EHR), and Sensor data from different IoT devices, the time step is not fixed, and observation of event data timing is also essential. For example, the time gap between patients' consecutive visits is vital for doctors to analyze their health condition. For sensor data, the time steps are dynamic and very small.

The most significant challenge in time series prediction is predicting a smaller time difference in real-valued time series. It needs numerous amounts of computation and speedy result generation. Another relevant changeless in time sequence prediction is to predict continuous time series. Recurrent neural network (RNN) has become state of the art for time series modelling for its unique and dynamic characteristics. RNN considers continuous-time data as a discrete dynamic system with constant time steps and a constant sampling rate at each time step. However, these constraints on regular time steps significantly impact efficiency and precision for continuous-time series. Therefore, some research has been performed. Some others are still ongoing to re-model or re-design different RNN architectures such as LSTM and GRU. Due to the dependency on the computation of previous states, RNN models are still prone to some vulnerabilities. If the time step is too big, it may adversely affect the model's efficiency. A longer time step might make sense for weather prediction and energy consumption prediction, but application areas like event-based sensor signal processing need higher frequencies and smaller time steps. It is often expected to record states with short time steps and accurate timing for data-driven applications. For example, for a sine wave prediction in continuous time series, the most suitable choice is the RNN model. However, RNN still discretizes the continuous-time series, as shown in Fig [1.3]. However, it would be more efficient if RNN models could model the time series in continuous time with accurate timing as a function in Fig [1.3].

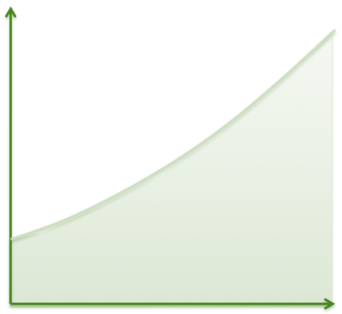


Figure 1.3: Continuous function

Traditional RNN models compute the states either cell (c_t) or hidden (h_t) by composing a sequence of previous states:

$$c_{t+1} = c_t + f(c_t, \delta_t) \tag{1.4}$$

$$h_{t+1} = h_t + f(h_t, \delta_t) \tag{1.5}$$

Recent work [6] proposes considering a neural network as a combination of n number of layers. It can be presented as a continuous function, and the hidden state can be parametrized using an ordinary differential equation (ODE) solver:

$$\frac{dh(t)}{dt} = f(h_t, t, \delta_t) \tag{1.6}$$

Here f is an initial problem-based ODE solver. Using the initial $h(0)$ condition, the ODE solver computes the output $h(t)$ at time t. This approach provides faster training time than residual network; constant memory cost instead of linearly increasing memory cost and simpler architecture design for any neural network model. Fig [1.4] was presented by [6] to distinguish the difference in learning vector transformation between residual neural network and ODE neural network. As depicted in Fig [1.4], the residual neural network usually models time series as a discrete sequence of finite transformation where the learning vector for the ODE neural network is a continuous transformation of states.

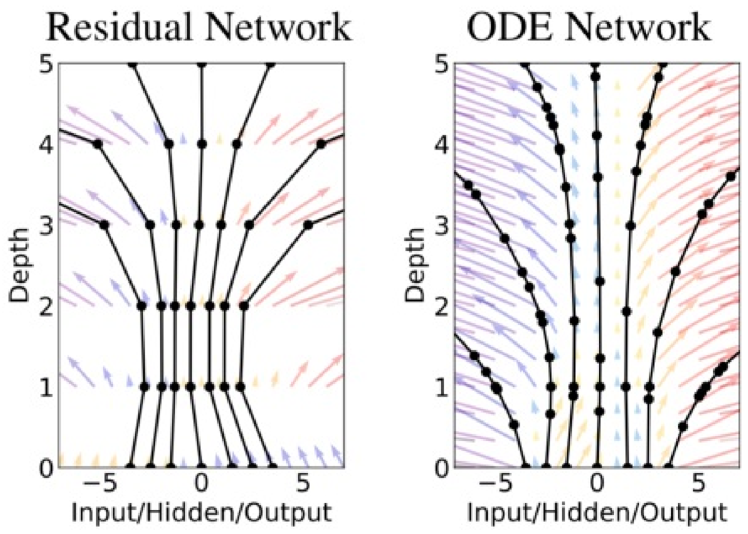


Figure 1.4: Influence of the number of layers of a neural network [6]

This new family of neural networks has some substantial improvement over the residual neural network as following

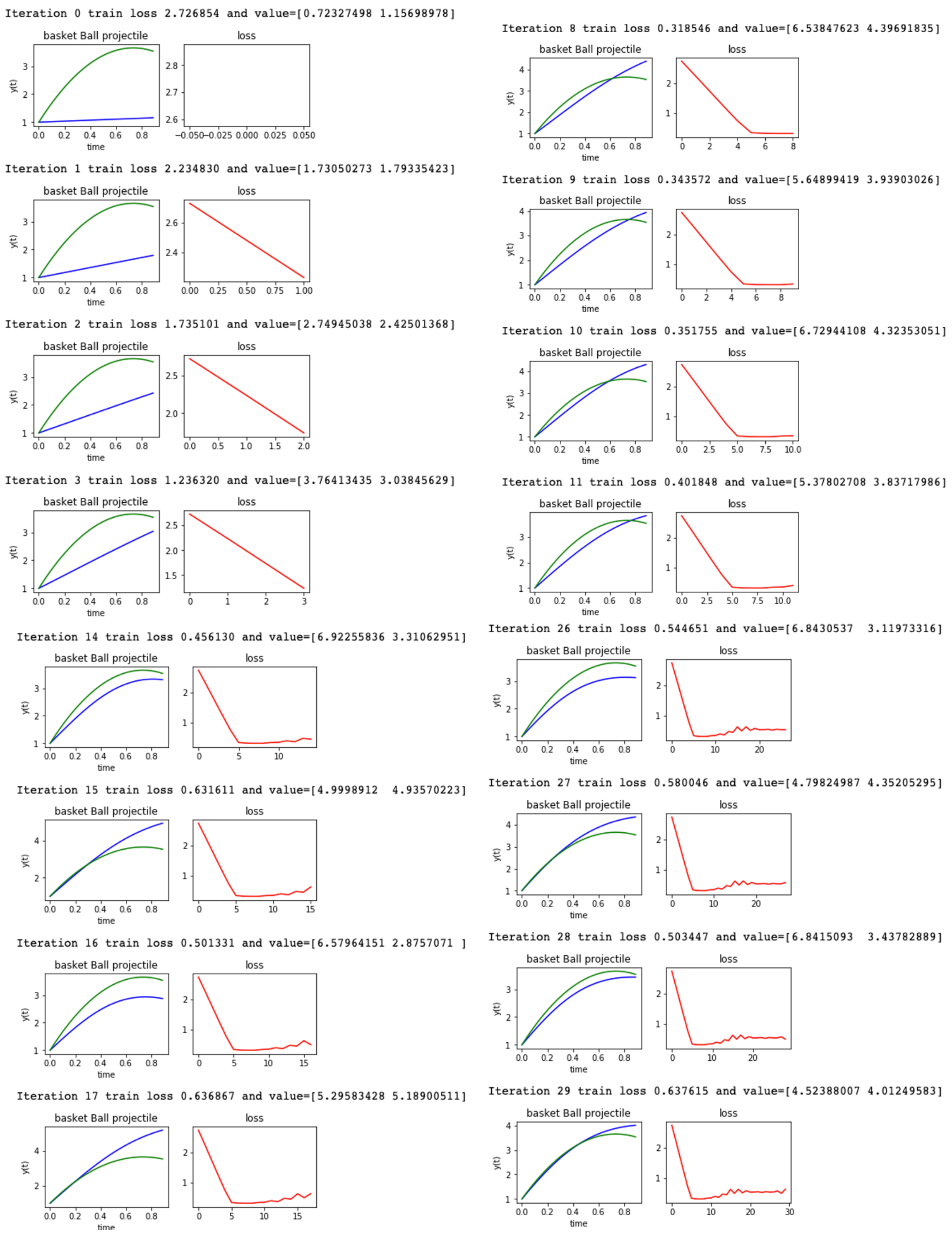
- Ordinary Differential Equations require one independent variable and one derivative of an unknown function. An unknown ODE solver can compute the value of the independent variable at any time t with the desired accuracy.
- There are two different methods for first-order differential equations, e.g., the Euler method [8] and the adjoint sensitive method.
- The construction of the neural network is much easier. Any optimizer can be used for learning without determining the number of layers.
- It does not need to define the number of discrete layers. Instead, the network is a continuous function where the gradient can train itself within marginal error.
- Finally, it would help design the neural network model as a function of time t

for continuous-time learning problems.

In this PhD, I target to leverage these characteristics of the ODE neural network for the architecture of deep models to learn continuous time series in real-time.

1.3.1 Ordinary Differential Equation (ODE) based Neural Network

A simple problem helps us understand the strength of ODE based neural network for continuous time series. For this problem, I explore the Projectile Motion: Shooting a Basketball Problem. In this problem, a basket has its initial position X_0, and overtime T, it crosses the path and appears at a new position X_T. I use a simple ODE solver to determine X_T for any input of initial condition X_0 and time T. This result, shown in Fig 1.3.1, motivates us to explore Ordinary Differential Equation (ODE) based Neural Networks as a Solution to continuous time series modelling and prediction problems.



```
Iteration 0 train loss 2.726854 and value=[0.72327498 1.15698978]
Iteration 1 train loss 2.234830 and value=[1.73050273 1.79335423]
Iteration 2 train loss 1.735101 and value=[2.74945038 2.42501368]
Iteration 3 train loss 1.236320 and value=[3.76413435 3.03845629]
Iteration 14 train loss 0.456130 and value=[6.92255836 3.31062951]
Iteration 15 train loss 0.631611 and value=[4.9998912  4.93570223]
Iteration 16 train loss 0.501331 and value=[6.57964151 2.8757071 ]
Iteration 17 train loss 0.636867 and value=[5.29583428 5.18900511]
Iteration 8 train loss 0.318546 and value=[6.53847623 4.39691835]
Iteration 9 train loss 0.343572 and value=[5.64899419 3.93903026]
Iteration 10 train loss 0.351755 and value=[6.72944108 4.32353051]
Iteration 11 train loss 0.401848 and value=[5.37802708 3.83717986]
Iteration 26 train loss 0.544651 and value=[6.8430537  3.11973316]
Iteration 27 train loss 0.580046 and value=[4.79824987 4.35205295]
Iteration 28 train loss 0.503447 and value=[6.8415093  3.43782889]
Iteration 29 train loss 0.637615 and value=[4.52388007 4.01249583]
```

Figure 1.5: Predicting the projectile for a basketball using ODE solver

1.4 Contributions

In my PhD thesis, I have focused on minimizing the gap between discrete time steps based time series modelling to continuous time series modelling with the dynamic characteristics of Ordinary Differential Equations.

1. I analyzed different deep neural network model family members for modelling different continuous time series problems. I identified the limitations and challenges for existing models in the case of continuous-time series modelling. I also surveyed several deep neural network models. This survey compares the strengths and weaknesses of available models for continuous-time series problems. I also conducted a detailed survey on the characteristics of continuous-time series as follows

   - Irregular and higher sampling rate
   - informative missingness due to missing values
   - high frequency and dimension
   -

2. I introduced two neural network models based on Recurrent Neural Network and Neural ODE. The first model, GRU-ODE, is a Neural ODE implementation of the Recurrent GRU model. Another model is the Ordinary differential neural network model architecture for LSTM. These two models can compute the hidden state and cell state using an Ordinary Differential Equation solver at any point in time. This model reduces the computation overhead of hidden state and cell state with a huge amount of parameter. The continuous-time latent variable model demonstrated the performance of learning continuous time series with irregular sampling rates. This proposed LSTM models use a differential equation solver to compute the hidden and cell state at any time. As the model

is designed as a function, it can learn continuous series without discretizing the actual time series. Proposed ODE-LSTM and ODE-GRU models predict the output (y_t) at any time t using the initial value of the observation (y_0) at time $t = t_0$. Therefore, the predicted output is generated as a function instead of computing each state based on its previous state individually. I have shown that these models can provide stability to the structure of the existing Neural ODE model.

3. I introduce two different Neural ODE models based on the GRU-D model to resolve the informative missingness problem of traditional models where time-series data is presented as discrete observations at a fixed time step.

4. Together with my co-authors, I introduce a range of Generative Adversarial Network (GAN) and Neural ODE based GAN architectures to generate synthetic medical data such as ECG or continuous time series such as sine wave. I demonstrate that Neural ODE architecture outperforms GAN in terms of data quality. The GAN triumph in terms of the variety or length of the time series.

5. I introduce the architecture of Convolutional Neural Network (CNN). I develop a couple of Partial Differential Equation (PDE) solvers and Ordinary Differential Equation (ODE) solvers to construct this model.

6. Together with my co-authors, I introduced two models that leverage the dynamic characteristics of the Heun method to design a new deep Neural Network architecture. This new model is an extension of Residual Neural Networks (ResNet) based on an idea from Heun's method for numerically solving ODEs.

7. 7. I introduce a neural network architecture that works as a temporal wormhole connection based on the explicit iterate-to-fixedpoint operator that derives the associated system for forward and reverse mode algorithmic differentiation to connect with the previous hidden state. This model can optimize the parameter

by creating a temporal wormhole connection between different layers instead of following only a feed-forward sequential connection to the next state. This model can retrain the model from a previous state to optimize the parameter and generate a better solution.

1.5 Thesis structure

Chapter 1 provides a high-level abstraction of the problem domain, my research questions, motivation for this work and an overview of the scientific contributions introduced by this thesis.

Chapter 2 presents a comparative survey on available models and application areas that motivate this work. First, I formally define the time series and the challenges in the case of modelling continuous time series using available models. Next, I overview traditional models for time series modelling, forecasting, classification, augmentation and anomaly detection. Finally, I described how an Ordinary Differential Equation (ODE) could be a building block for a deep neural network model for continuous time series. I also explained how ODE could improve model performance by moving away from the discrete-time sequence-based data modelling toward continuous time-series data.

Chapter 3 introduces the Ordinary differential neural network model architecture for LSTM [5] and GRU [9] Recurrent Neural Networks. These proposed models in this chapter use differential equation solvers to compute the hidden and cell state at any time. This model reduces the computation overhead of hidden states and cell states with a massive amount of parameters. The continuous-time latent variable model demonstrated the performance of learning continuous time series with irregular sampling rates. These proposed models use an ordinary differential equation solver to compute the cell state and hidden dynamics of the time series at any given time. As the model is designed as a function, it can learn continuous series without discretizing the actual time series.

Chapter 4 introduces two Ordinary differential neural network models for GRU-D [3] resolve the informative missingness problem in multivariate time series data in practical applications. These two models solve the informative missingness in multivariate time series by leveraging a black box ordinary differential equation solver to compute the instant change in decay, hidden dynamics and GRU-D cell state as a derivative of time. In addition, these models also leverage the decay mechanism of GRU-D as a derivative of the change in the hidden states over time.

Chapter 5 investigates the ability of generative adversarial networks (GANs) and Neural ODE based models to produce realistic medical time series data. I introduce a new generative model based on neural ODE to generate training data for the medical dataset in continuous time series. I also demonstrated the ability of the Neural ODE based model to generate continuous time series that are diverse in nature, shape and pattern.

Chapter 6 introduces a new architecture for the Convolutional Neural Network (CNN) as a function of continuous-time. A couple of Partial Differential Equation (PDE) solvers and Ordinary Differential Equation (ODE) solvers produce this new architecture for better performance within a short training time and with fewer parameters.

Chapter 7 introduces a new architecture for the Neural Network as an extension of the existing Residual Neural Network or ResNet. This Idea also leverages the characteristics of the Ordinary Differential Equation solver especially using Heun's method. This proposed model benefits from the dynamic characteristics of the Heun method, which improves the proposed model's performance over other traditional models.

Chapter 8 introduces a new architecture for the Neural Network where the fixed point iteration method controls the training for different layers in a ResNet model. This model can control the training for different layers in the ResNet model. The training for the different layers is optimized based on the characteristics of fixed-point iteration.

Chapter 9 contains final remarks and discussion on the problem domain and proposed models, the significance of contributions of this thesis and topics for future research.

1.6 Published papers

The following papers have been published as part of this thesis:

- Habiba, Mansura, and Barak A. Pearlmutter. "Neural ordinary differential equation based recurrent neural network model." 2020 31st Irish Signals and Systems Conference (ISSC). IEEE, 2020.
- Habiba, Mansura, and Barak A. Pearlmutter. "Neural ODEs for informative missingess in multivariate time series." 2020 31st Irish Signals and Systems Conference (ISSC). IEEE, 2020.
- Maleki, Mehrdad, Mansura Habiba, and Barak A. Pearlmutter. "HeunNet:

Extending ResNet using Heun's Methods." 2021 32nd Irish Signals and Systems Conference (ISSC). IEEE, 2020.

- Habiba, Mansura, and Barak A. Pearlmutter. "Continuous Convolutional Neural Networks: Coupled Neural PDE and ODE." 2021 2nd International Conference on Electrical, Computer and Energy Technologies (ICECET). IEEE, 2021.
- Habiba, Mansura, and Barak A. Pearlmutter. "Neural Network based on Automatic Differentiation Transformation of Numeric Iterate-to-Fixedpoint." 2021 2nd International Conference on Electrical, Computer and Energy Technologies (ICECET). IEEE, 2021.
- Mansura Habiba, Eoin Borphy, Barak A Pearlmutter and Tomas Ward. "ECG synthesis with Neural ODE and GAN models." 2021 2nd International Conference on Electrical, Computer and Energy Technologies (ICECET). IEEE, 2021.

1.7 References

[1] Daniel Neil, Michael Pfeiffer, and Shih-Chii Liu. Phased lstm: Accelerating recurrent network training for long or event-based sequences. In Advances in Neural Information Processing Systems, pages 3882–3890, 2016.

[2] Víctor Campos, Brendan Jou, Xavier Giró-i Nieto, Jordi Torres, and Shih-Fu Chang. Skip rnn: Learning to skip state updates in recurrent neural networks. arXiv preprint arXiv:1708.06834, 2017.

[3] Zhengping Che, Sanjay Purushotham, Kyunghyun Cho, David Sontag, and Yan Liu. Recurrent neural networks for multivariate time series with missing values. Scientific reports, 8(1):6085, 2018.

[4] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. IEEE transactions on Signal Processing, 45(11):2673–2681, 1997.

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

[6] Tian Qi Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. In Advances in Neural Information Processing Systems, pages 6572–6583, 2018.

[7] Yulia Rubanova, Tian Qi Chen, and David K Duvenaud. Latent ordinary differential equations for irregularly-sampled time series. In Advances in Neural Information Processing Systems, pages 5321–5331, 2019.

[8] Roger Alexander. Solving ordinary differential equations i: Nonstiff problems (e. hairer, sp norsett, and g. wanner). Siam Review, 32(3):485, 1990.

[9] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. arXiv preprint arXiv:1406.1078, 2014.

CHAPTER 2

RECENT TRENDS IN MODELLING THE CONTINUOUS TIME SERIES USING DEEP LEARNING: A SURVEY.

Continuous-time series is essential for different modern-day application areas, e.g. healthcare, automobile, energy, finance, Internet of things (IoT) and other related areas. Different practical applications need to process and analyze a massive amount of continuous time-series data to determine a data-driven result. For example, financial trend prediction, potential probability of a particular event occurrence identification, patient health record processing and other practical applications continuously analyze time-series data. However, modelling real-time data using a continuous-time series is challenging since the dynamical systems behind the data could be a differential equation. Several research works have tried to solve the challenges of modelling the continuous-time series using different neural network models and data processing and learning approaches. The existing deep learning models are not free from challenges and limitations due to diversity among different attributes, behaviour, duration of steps, energy, and data sampling rate. In this paper, we have described the general problem domain of time series and reviewed the challenges in modelling the continuous time series. We have presented a comparative analysis of recent developments in deep learning models and their contribution to solving different challenges of modelling the continuous time series. We have also identified the limitations of the existing neural network model and open issues. The main goal of this review is to understand the recent trend of neural network models used in a different real-world applications with continuous-time data.

2.1 Introduction

Time series modelling is very challenging due to the unique nature of the problem domain and the data structure. The challenges for continuous-time series processing using deep learning are that (i) the data can be collected at a different rate, and (ii) this variable sampling rate needs a very complex deep learning architecture. Furthermore, the data collection and sampling rate can be quite irregular. For example, the sampling of sensors in the Internet of Things (IoT) framework can be sporadic. Additionally, continuous time series often have many parameters to provide a real-time result. Some of these parameters are unknown, uncertain, and dynamic; therefore, they are sometimes even unobservable. As a result, an efficient neural network model needs to be well-equipped to deal with high-dimensional data.

One of the main limitations of time series modelling is that existing models are suitable for the static problem domain, but real-time time series are noisy and dynamic. Additionally, there is not enough research tackling the problems of dealing with sparse and asynchronous continuous time series, processing multi-variate time series with a massive number of parameters and dealing with the dynamic, heterogeneous and uncertain nature of the time series. Besides, in most cases, existing deep learning models fail to collect real-time data due to the complex and higher-order temporal dynamics. In order to avoid the challenges of dynamic input size and variable sampling rate, most deep learning research mainly focuses on fixed-size inputs and consistent data format to represent time series which lacks of continuity. However, to get a better result for a dynamic system with uncertain nature in the era of Internet-of-Things (IoT), it is essential to improve the performance of deep learning algorithms. . In this survey, we aim to identify different challenges for time series modelling using deep learning algorithms.

Continuous time data hold revolutionary potential. For example, Helbing et al. [1]

describes the design of a complex system to understand the data flow of crime, terrorism, wars, epidemics, and other crowd disasters using continuous time data. Helbing et al. [1] also demonstrates the potential of real-time data analysis to address several serious issues that are disrupting human life over the last decades. We are in the prime time where there is no alternative of real-time data analysis for continuous time data. For example, self-driving cars are already on the street; Alexa [2] is analysing our daily activity in real-time. As the applications of autonomous objects are increasing widely, scientists can no longer settle with the result of approximate accuracy; they need real-time as a well accurate results.

Different types of deep learning neural network models have been researched and analysed to solve different challenges in terms of modelling time series modelling. However, these traditional algorithms show measurable limitations, when applied to continuous time data. This survey attempts to identify the challenges in different research fields with continuous time data as well as different aspects of existing Deep learning algorithms to overcome those challenges. This paper primarily focuses on the following questions,

1. What challenges for continuous-time data processing evolve with emerging technologies and research fields, e.g., IoT, healthcare and others?

2. How is the state of the art deep learning algorithms closing the gap to mitigate existing challenges?

3. How does continuous data processing using deep learning affect research in different real-life applications?

4. What are some recent promising neural networks for processing continuous-time data?

5. How ordinary and partial differential equations are changing the behaviour of

neural networks?

To understand the answer of above mentioned questions, we reviewed the recent trend in using deep learning algorithms for modelling continuous time data. We studied the problem domain to determine the main challenges. We also taxonomise different challenges and identified the state of art solutions for existing limitations in the case of modelling continuous-time series. Another significant contribution of this paper is that it discusses a comparative analysis of some promising recent research works corresponding to addressing challenges regarding computing efficiency in processing continuous time series. This survey aims to analyses the strength and weakness of existing deep learning algorithms to process continuous time series and related problems with real time continuous time data.

To achieve these goals, the rest of the paper is organised as follows: Section §2.2 uses deep learning to describe time-series properties and different aspects of the continuous-time series learning problem domain. Section §2.3 discusses different types of a continuous series of data. Comparative analysis of some existing neural network architectures and algorithms is described in Section §2.6. Section §2.5 analyses the existing challenges for modelling time sequences and several recent research to overcome those challenges. In §2.4, a wide range of applications of the continuous-time series is discussed. Some future research areas regarding time series modelling are described in Section §2.8.

2.2 Time Series Analysis

Due to the dynamic nature of the time series, learning the continuous time data with deep learning algorithms is a very complex task. Moreover, efficient and precise computation is even more challenging. In addition, continuous time series vary in length, which imposes additional complexity. Besides, there is an implicit dependency between different time states in a series. The previous input and past computation have a significant role in calculating the current or future state in a time series. Therefore, most of the time, there is a requirement for memorizing the previous state. As a result, a traditional feed-forward neural network, where input at each state is independent of each other, cannot learn continuous time series. Therefore, it is essential to understand the dynamic nature of continuous time series to design an efficient neural network model. In this section, we discuss the unique characteristics of a time series.

2.2.1 What is the time series?

Time series is a sequence of numeric observations of a variable at a continuous time. For example, Fig. 2.1shows the observation of certain variables, i.e. the stock price of Dell over a definite period. More precisely, a time series is a function $x : T \to \mathbb{R}^N$, where $T \subseteq \mathbb{R}$, with a probability distribution $\mathcal{P}$, i.e., $(x_t)_{t \in T} \sim \mathcal{P}$. If $t \in T$, then $x(t)$ or x_t refers to the observed value (x) at time t. If T is a finite period of time, we have the definition of discrete-time series. If $T = \{t_1, \dots, t_n\}$ with $t_{i+1} - t_i = h$ then (x_t) where $t \in T$ generates a discrete time sequence. If $T = [a, b]$ then a continuous time series [3] is produced.

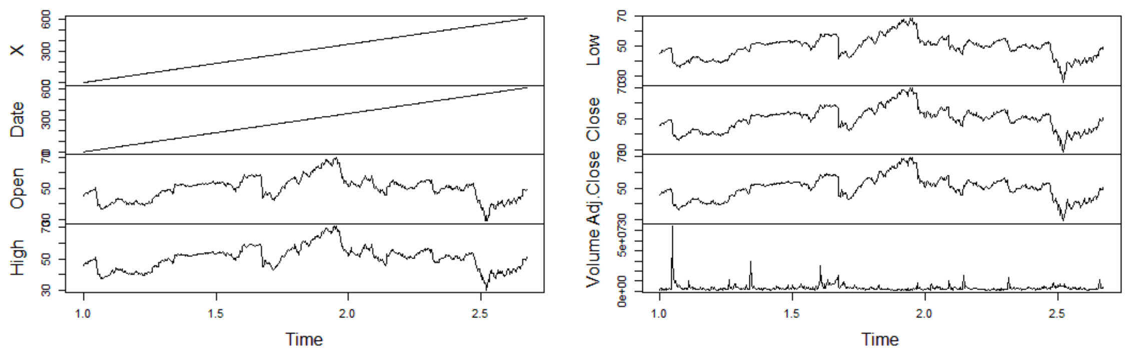


Figure 2.1: Continuous time series for the stock price of Dell company from 2 January 2018 until 5 June 2020

For example, any variable x can have values as $x_1, x_2, \ldots, x_n$ at different time steps, such as $t_1, t_2, \ldots, t_n$. Here, Eq (2.1) shows the time series as a vector [4], where each value $x_t \in \mathbb{R}$. Each entry in the X vector is the real-time value of a time-dependent variable measured at a corresponding time t of a continuous or discrete time series T, i.e., $x_1 = x(t_1), x_2 = x(t_2), \ldots,$ where $t_1, t_2, \cdots \in T$.

$$X_n = (x_1, x_2, \ldots, x_n) \tag{2.1}$$

2.2.2 What are the properties of a continuous time series?

Continuous time data has unique properties. Some of them are listed as follows:

- **Length**: Different applications such as automobiles and health need to observe the value for any variable for a very long period to get correct data. Therefore, most continuous real-time data is usually very long. Here are some real-time data examples, such as the weather data analysis for ten years, financial trend analysis for the last decade, unemployment statistics analysis, and event data from sensors for a self-driving car for one hour. They all use continuous time data for an extended period. Furthermore, for accurate analysis of any time series problem, including classification, anomaly detection, and synthetic generation, it is essential to either train a very large data set and/or train the data set for a very long time.

- **Higher Sampling Frequency**: The intermediate time step is short for a continuous time series, and the sampling frequency is much higher. Existing Recurrent Neural Network (RNN) models struggles to model such time series with higher sampling rate as they usually use a fixed intermediate time step. Instead of modelling a continuous time series, these models first convert the time data to a discrete time series data with fixed intervals.

- **Pathological dependencies**: One of the main goals for time series modelling using deep learning algorithms is to identify the underline temporal relation of consecutive data points to detect the pattern. In the case of most time series, there are dependencies between consecutive data points. This dependency can be either implicit or explicit. However, the Length of this mutually dependent sequence is unknown. A time state can be dependent on a near past time state, on the other hand, time state can also depend on a state that occurs in the time series very long ago. For example, x_t can depend on previous observations at a very distant past(x_{t-n}) or a very short distance past(x_{t-1}).

- **Higher Dimension**: Every data point x_t in Eq. (2.1) consists of thousands of attributes. Multiple parameters are a common feature for continuous real-time data. For example, x_t in Eq. (2.1) can be rich in dimension with a large number of parameters in order to describe an individual observation. Multi-variate time series show very high dimensional. The value for x at time t does not only depend on time t but also depend on other variables(multivariate time series), i.e., if $\mathbf{X}(t) = [x_1(t), \ldots, x_N(t)]$ then $\mathbf{X}(t) = A_1\mathbf{X}(t-1) + \cdots + A_p\mathbf{X}(t-p) + \mathbf{b} + \varepsilon(t)$ in its simplest form, where A_i are $N \times N$ autoregressive matrices and error term $\mathbf{e}$ and $(x_i(t))_{t\in T}$ for $i = 1, \ldots, N$ are different time series [5].

- **Additional Noise**: Along with the real-time value of any observations x_t at a time t, continuous-time also contains excessive noise components for the entire series.

- **Higher Energy Consumption**: The dimension of a continuous-time series is often very long, requiring extensive computation and energy.

- **Uncertainty**: Continuous-time series is often very dynamic and uncertain with the missing pattern and irregularities. Therefore, it is often impossible to model the complete time sequence. However, different well-known data imputation techniques are used to replace the missing values in a time series.

- **Irregular Sampling Rate** : Besides having a higher sampling rate, continuous-time series also exhibit an irregular sampling rate. The irregular sampling rate, high complexity, and massive Length of real-world continuous time series often cause missing data points that negatively affect the result of the neural network. As a result, it is impossible to process an incomplete model precisely. An irregular sampling rate often causes missing observations. The value of observation x can be missing at any time step t in the continuous-time series. Not every step has enough information to describe the observed value for x at time t.

- **Memory**: Time series needs Memory to encapsulate information from previous steps. Sometimes, the interrelation between different steps in the time sequence.

In short, continuous-time data is dynamic, enormous in amount, unique, sophisticated, and high dimensional. Moreover, the characteristics of different time series are extensively dynamic. All these properties of continuous-time series make it very challenging to process. However, several research works have set in in the recent era and significantly improved results with continuous-time data. Therefore, a recent trend is to design deep learning models to process continuous-time data without converting the continuous-time series to discrete-time series. This paper discussed how recent deep learning models are designed to adjust to continuous-time data characteristics mentioned above. The architecture of deep learning models has to consider all these unique time-series behaviours to achieve a better result with higher accuracy precision. The complexity, training data requirement, and computation power can vary a lot based on the characteristics of time series and the architecture of the neural network model.

2.3 Problem Domain Analysis

Time series related problems are not new, and the categories of these problems are wide. Primarily, these problems can be classified into four categories [6],as shown in figure [2.2]. Again Modeling can be either sequence-to-sequence mapping or augmentation. Time series classification has a wide range of applications, as shown in Table 2.4. Classification problems can be related to classifying a pattern embedded in the data or classification of continuous-time series data such as video, text, wavelet and others. Prediction is probably the most popular problem type in continuous time series. Besides these, sometimes detection of embedding pattern in the continuous-time data is also an essential problem category. Another kind of problem in time series is anomaly detection. There is no abundant number of work in this area of the time series problem domain. [7] proposed an anomaly detection mechanism using low-level tracking algorithms, and [8] highlights the following questions that need to be solved for deep learning to be ready to provide accurate results.

- How does understanding (explicitly extracting the geometrical structure of a low-dimensional system) relate to learning (adaptively building models that emulate a complex system)?
- When a neural network correctly forecasts a low-dimensional system, it has to have formed a representation of the system.
- What is this representation?
- Can the representation be separated from implementing the corresponding network?
- Can a connection be found between the entrails of the internal structure in a possibly recurrent network, the accessible structure in the state-space reconstruction, the structure in the time series, and ultimately the structure of

the underlying system?

It is fascinating to say that most of the questions are now known. The answer to the third question is that the time series can be represented using a graph or N × M vector as a neural network input. The neural network proposed in [9, 10] describes a neural network model where the structure of space, as well as the structure of the time series, are considered for neural network model architecture. These works are practical examples to answer the fifth question mentioned above. Spatiotemporal-graph [9] has become a popular representation of space and time sequence.

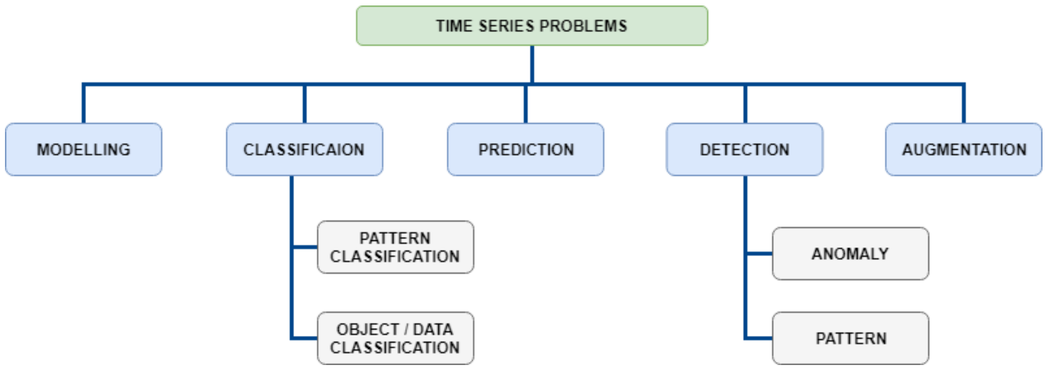


Figure 2.2: Different category of time series problems

The unique behaviour of continuous-time series attracts the attention of deep learning researchers in the early 90s. Table 2.4 shows some related works in every type of continuous-time series-based problem. I have evaluated recent works in mainly the last decade in this work.

There is no single solution for each problem as mentioned above. Different types of neural network models are suitable for different problems types. For example, an Artificial wavelet neural network (ANN) [4] is suitable for modelling and prediction of continuous time series. On the other hand, RNN is a pioneer in time series modelling, classification and prediction problem domains due to its properties. For time series classification, CNN is the most popular model. CNN can learn classes from a continuous-time series through unsupervised learning with minimum human interaction. In most cases, CNN and time series modelling are not a practical choice. CNN is often used in a hybrid model and other neural networks such as AWNN or RNN, which can significantly improve performance. A recent trend in using hybrid neural network models is becoming very popular in a different applications of deep learning, where the model has the attribute from both RNN and CNN is promising to achieve a better result. Table 2.1 describes the usage of the different neural network models in the case of solving different categories of problems.

Classification: Time series classification task is a complex task for any deep learning frameworks. Time series can be either univariate as described by Eq (2.1) or M-Dimensional as described by Eq. (2.2), where X_i itself is a univariate time series. Sometimes time series can be even more complex where X_i in Eq. (2.2) can be a multi-variate time series instead of being uni-variate.

$$X = X_1, X_1 \ldots X_n \tag{2.2}$$

For the Time series classification task, time series is described as a collection of a tuple in [11] as shown in Eq.(2.3). Here time series is a data set, D, which consists of a collection of pair (X_i, Y_i), where X_i can be either uni-variate or multi-variate time series and Y_i is a label.

$$D = \{(X_1, Y_1), (X_2, Y_2), \ldots, (X_N, Y_N)\} \tag{2.3}$$

The time series classification task becomes more challenging as the number of dimensions increases.

prediction : Prediction tasks use previous observations for any variable such as stock price, rainfall, house price, energy consumption and others; and forecast a future

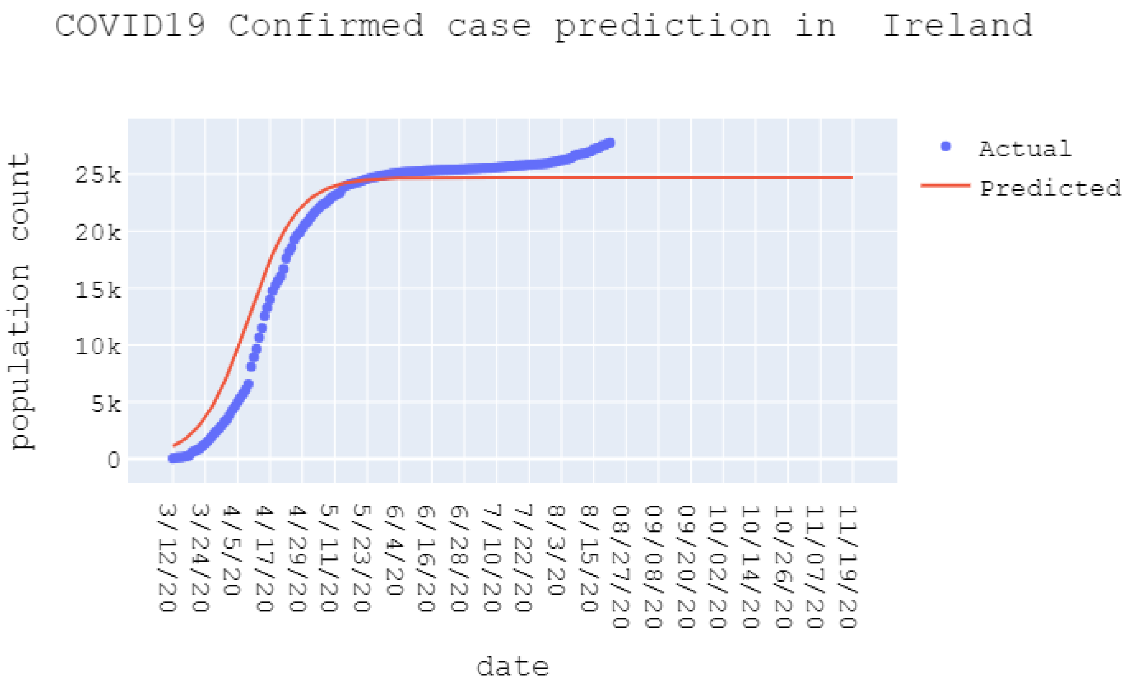


Figure 2.3: Time series prediction for confirmed covid-19 case in Ireland

observation value for the corresponding variable. [4, 12] describes the time series prediction using a neural network that identifies the relationship between the previous value and the current value of a variable. For example, in Eq. (2.4a), f is a neural network that can identify the relation between past values of x from time $t-n$ to time $t-1$ with it's current value $x(t)$ at time t. This is an example of the simplest time series prediction with one step. Similarly Eq. (2.4b) shows multi-step ahead prediction, where the next h-the value for variable x is predicted using neural network f.

$$x(t) = f\{x(t-1), x(k-2), \ldots, x(t-n-1)\} \tag{2.4a}$$

$$x(t+h) = f\{x(k), x(k-1), x(k-2), \ldots, x(k-n-1)\} \tag{2.4b}$$

In Fig 2.3 the predicted number of confirmed Covid-19 cases in Ireland are platted.

Detection: Detection task is often used for classification and prediction. For example, to understand the difference or anomaly in the Actual value and Predicted value is shown in Fig2.3, time series anomaly detection is essential.

Augmentation: A recent survey work, [13] describes different techniques for successful time series augmentation used for generating synthetic time series data. 2.4 shows the taxonomy of time series augmentation for deep learning algorithms proposed by [13]. This review finds the most available time-series augmentation technique. However, due to the complex nature of different fields, the time series data has particular characteristics that need additional treatment. For example, finance multi-variate time data often shows intrinsic probabilistic patterns as a relationship among different variables; health record data shows a direct relationship among different variables at any time point, represented as a graph. In this section, these additional time-series augmentation techniques are analysed at length.

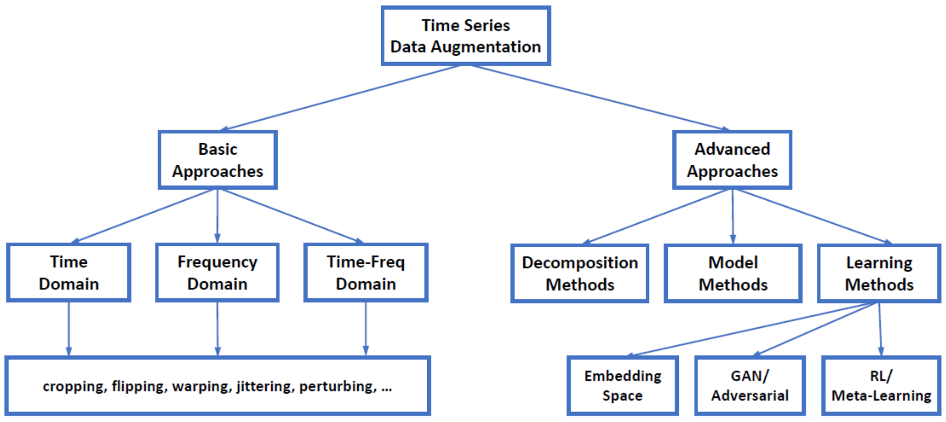
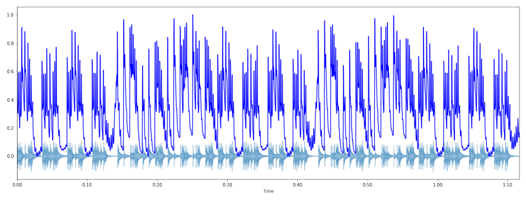
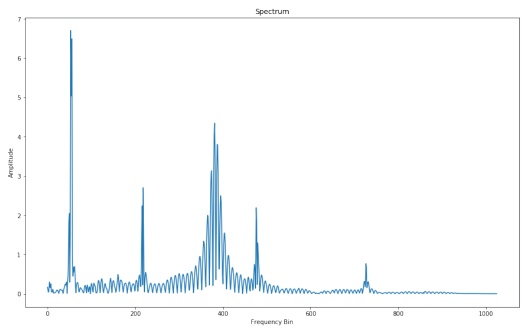


Figure 2.4: Taxonomy of time series data augmentation proposed by [13].

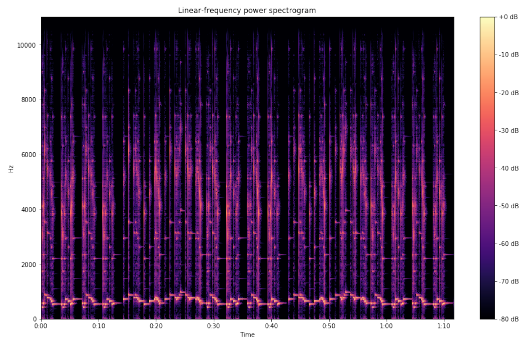
Fig. 2.3, 2.3 and 2.3 show different kind of augmentation for a continuous audio file *Pokemon - pokecentre theme.wav* from the Pokemon-midi [14] dataset.



(a) Time Domain



(b) Frequency Domain



(c) Time-Frequency Domain

Figure 2.5: Different types of augmentation for time series

Besides these time series data augmentation techniques, some other advanced augmentations, such as (i) Graph Augmentation (ii) Embedding Entropy with Ordinal patterns to describe the intrinsic patterns, are increasingly getting attention among researchers.

2.3.1 Different dataset

This section discusses some well-known datasets used for additional research. Different dataset focuses on different characteristics of continuous-time data as discussed in 2.3. Table 2.2 shows the feature for some well-known datasets.

Different libraries and packages are developed to generate time targeting different algorithms for time series problems. For example, [37] is a python package that provides the implementation along with an example dataset for different time series classification algorithms such as Dynamic Time Warping, Shapelet Transform, Markov

Table 2.1: Different Neural Network Models for Different Time Series Problems

Deep Model	Modelling /Augmentation	Classification	Prediction	Detection
ANN	[4, 15]		[4, 16, 17]	
CNN		[11]		
RNN			[18, 19]	
LSTM	[20–23]		[19]	
GRU	[24]		[6, 19]	
DNN	[25]			
Hybrid	[26–28]		[29–31]	[32]
AWNN	[33]	[34]		[35]
Graph	[9, 10]			

Table 2.2: Well-known datasets for continuous-time data research

Dataset	**Data type**	**Application**	**Nature**
MIMIC-III	Electronic Health Record (EHR)	Health Care	Missing pattern, irregular sampling rate
GMIS		Energy	High Dimension
DMSP/OLS (NSL)		Energy	Missing pattern
Global Population Density Grid Time Series Estimates		Energy	
British Petroleum (BP) Statistical Review of World Energy		Energy	irregular sampling rate
BreizhCrops	satellite image time series	Agriculture	
MNIST [36]	Image processing	object detection and classification	complex series
Multivariate Time Series Search (NASA)	sensor generated data for media and event	Aerospace	Multi-variate

Transition Field, Time Series Forest and others.

For healthcare, the availability challenges of dynamic datasets are more severe than in other research fields. Due to data privacy issues, a limited number of the dataset is publicly available, which focuses on a limited number of aspects of health care. Most research in health care with continuous-time data is biased towards these same EHR datasets, which significantly impacts overall research for health care. There are some efforts [38] to generate synthetic health data that can simulate the temporal relations of multiple variables in a patient's health record.

Like health care, some other research areas also suffer from the absence of enough labelled data. These fields generate their data. They are using automated tools to generate a synthetic dataset for unexplored input space in various time series problems, including time series classification, forecasting, and anomaly detection. A successful tool requires the following feature.

1. can generate very long time series with multiple variable
2. can generate high-quality continuous-time data
3. uses an efficient data augmentation technique.

Table 2.3 shows some well-known dataset used in different problem-solving in the problem domain of continuous time series.

Table 2.3: Some well-known dataset for time series problems

Classification	**Modelling**	**Prediction**	**Regression**
Human Activity Recognition[39]	SPSS with USDA feed grains [40]	Daily Demand Forecasting [41]	Air Quality [42]
Gesture Phase Segmentation [43]	EEG [44]	Google Web Traffic Time Series Forecasting	
MINST [36]	Activities modelling as Fuzzy Temporal Multivariate model	Population Time Series	
CIFAR10	Free Music Archive (FMA)	Climate Change Earth Surface Temperature	
PhysioNet Challenges dataset[45]		Hourly Energy Consumption	
MMIC-III [45]		Financial Distress Prediction	
Time Series [46]			

2.4 Different Applications of Continuous Time Series

There are several applications of sequential data modelling, for example, speech recognition, bioinformatics and human activity recognition. The input is a continuous or discrete audio clip P in speech recognition, which must be mapped to the corresponding text transcript Q. In this example, the input and the output are sequential data containing temporal information. Input, P, is an audio clip that plays out over time and output, Q, is a sequence of words. Therefore, deep learning models, suitable for sequential data such as recurrent neural networks and their other variations, have become promising for speech recognition, another example of continuous-time series modelling in music generation. In the case of music generation, only the output Q is a sequence of music notes, where the input can be an empty set, or a single integer, just representing the genre of music. The input can also be a set of the first few notes. The output Q is a sequence of data. This sequence also has implicit temporal information. A third example is sentiment classification, where the input P is a sequence of information, and the output is a discrete set of values. Even in bio-informatics, sequential data modelling is extensively essential for the different fundamental research areas, such as DNA sequence analysis. For example, in DNA analysis, the input P is a sequence of the alphabetic letter. The corresponding output is a corresponding label (Q) such as protein for each part of the input sequence. So here input, P is a sequence of data and output, Q is a set of labels. Human activity recognition or video activity recognition is also related to research fields where sequential data models like time series can benefit. For example, the input P is a sequence of video frames, and the output is the corresponding activity. Future trend prediction in financial data or event prediction from a continuous sequence of sensory information is some recent area where sequence modelling can bring a revolution in inaccuracy and efficiency. These are just a small subset of sequence modelling problem domains. There are a whole lot of different types of sequence modelling problems. These kinds of problems have some common structure and challenges. Following are some features from different sequence modelling problem

- Both input (P) and output (Q) can be sequenced
- Only one of the Input (P) and Output (Q) can be sequenced, either input or output

- P and Q can have different length
- P and Q can have the same length
- P and Q can be either continuous or discrete

Time sequences can be different in length, characteristics, nature and behaviour. On top of that, the corresponding output can also vary in type. Therefore, the deep learning model needs to be coherent, robust, dynamic and efficient to achieve a better result.

Different applications and problem domains are prone to different challenges of the continuous-time series. For example, in [20] the main challenges of event prediction using sensor data have been described are as follows.

Table 2.4: Different applications of time series problems

Application	**Modelling / Augmentation**	**Classification**	**Prediction**	**Detection**
Time series behaviour	[47, 48]	[11, 35, 49–51]	[34, 52, 53]	[32, 54]
Climate	[23, 55, 56]			
Financial trend forecasting	[9, 57, 58]		[41, 59, 60]	
Event prediction	[20]		[61, 62]	
Human Activity Tracking	[23, 63]			[43]
Image Processing		[64]		
Helath Data	[38]	[65, 66]	[47, 67–69]	[70]
Speech recognition	[2, 25, 71]			
Character detection	[71]	[30, 31]		[2]
Human activity recognition [63]				
Signal processing	[72–75]		[76, 77]	
Climate	[15, 78]		[79–82]	[42]
Robotics				[83]

2.5 Different challenges in modelling time sequence

Continuous-time dataset possesses several unique behaviours, making modelling data and learning the hidden dynamics or patterns extensively challenging. This section elaborates on different challenges of continuous-time data processing using neural networks and highlights the recent trend for solving those challenges.

2.5.1 Modelling hidden dynamics of dynamic temporal system

Modelling continuous time series is the fundamental task for each of the continuous-time series problem categories mentioned in section 2.3. Different deep learning models demonstrate various techniques to overcome challenges and design seamless dynamical systems with continuous-time data. ANN, RNN, CNN, DNN, DBN, and other hybrid neural network models exhibit higher precision accuracy among different deep learning models. However, their performance varies based on the system architecture and characteristics of underline data. No single model is suitable for all different continuous-time dynamics systems. For example, for a continuous-time dynamical system with regular as well as irregular data sampling rat, LSTM and different variations of LSTM show more promising results than other neural networks [20, 22]. LSTM can capture the long-term dependency in continuous-time series, which increases the success rate, model efficiency, and accuracy precision. Still, LSTM has some weaknesses as well. One of the limitations of LSTM based neural network models is that they cannot explicitly model the pattern in the frequency distribution, which is a critical component in solving the time series prediction problem. To solve this limitation of LSTM, a recent novel work [84] decomposes the memory states of an input sequence into a set of frequency sets. [84] uses the time sequence of length of T represented by Eq. (2.1) in section 2.2 as the input, where each observation (x_t) belongs to an N-dimensional space $\mathbb{R}^N$, i.e., $x_t \in \mathbb{R}^N$. Similar to LSTM, this model uses a sequence of the memory cell, with a length same as the time duration (T), to model the dynamics of the input sequence. However, each memory cell state is decomposed into a set of frequency components as shown in Eq. (2.5). Here, F represents a state-frequency decomposition of the input sequence across different states and frequencies.

$$F = \ w_1, w_2, \ldots, w_K \tag{2.5}$$

To update the cell state, this model uses a State Frequency Memory (SFM) matrix, $S_t \in \mathbb{C}^{D\times K}$, where D is the number of dimensions and K is the number of frequency states as shown in Eq. (2.6). Forget gate (f_t) and input gate (g_t) control the current as well as previous memory state and frequency to update the SFM at time t using a modulation (i_t) of the current input. Eq. (2.7) shows that (i_t) combines the current input and the output vector from previous state z_t. SFM combines memories from the previous cell to the current input, similar to LSTM.

$$\mathbf{S}_t = \mathbf{f}_t \circ \mathbf{S}_{t-1} + (\mathbf{g}_t \circ \mathbf{i}_t) \begin{bmatrix} e^{j\omega_1 t} \\ \ldots \\ e^{j\omega_K t} \end{bmatrix}^T \in \mathbb{C}^{D\times K} \tag{2.6}$$

$$\mathbf{i}_t = \tanh\left(\mathbf{W}_i \mathbf{z}_{t-1} + \mathbf{V}_i \mathbf{x}_t + \mathbf{b}_i\right) \in \mathbb{R}^D \tag{2.7}$$

This work proposed an improved forget gate which is a combination of state forget gate ($\mathrm{f}_t^{\mathrm{ste}}$), described in Eq (2.8), and frequency forget gate($\mathrm{f}_t^{\mathrm{fre}}$),described in Eq (2.9). Therefore the forget gate as shown in Eq. (2.10)) is decomposed over memory states and frequency of the states and control the input.

$$\mathrm{f}_t^{\mathrm{ste}} = \sigma\left(\mathbf{W}^{\mathrm{ste}} \mathbf{z}_{t-1} + \mathbf{V}^{\mathrm{ste}} \mathbf{x}_t + \mathbf{b}^{\mathrm{ste}}\right) \in \mathbb{R}^D \tag{2.8}$$

$$\mathrm{f}_t^{\text{fre}} = \sigma\left(\mathbf{W}^{\text{fre}}\mathbf{z}_{t-1} + \mathbf{V}^{\text{fre}}\mathbf{x}_t + \mathbf{b}^{\text{fre}}\right) \in \mathbb{R}^K \tag{2.9}$$

$$\mathbf{f}_t = \mathbf{f}_t^{\text{ste}} \otimes \mathbf{f}_t^{\text{fre}} = \mathbf{f}_t^{\text{ste}} \cdot \mathbf{f}_t^{\text{fre}'} \in \mathbb{R}^{D\times K} \tag{2.10}$$

Here, W and V in Eq (2.8) and (2.9) are the weight vectors and z_t is an output vector computed from the amplitude ($\mathbf{A}_{\mathbf{z}}^k$) of SFM at time t and the output of the output gate for each frequency component (k), of the previous cell as shown in Eq. ((2.11)). The multi-frequency aggregated output(z_t) as shown in Eq. ((2.12)) controls the input for forget and input gate.

$$\mathbf{z}_t^k = \mathbf{o}_t^k \circ f_o\left(\mathbf{W}_{\mathbf{z}}^k\mathbf{A}_{\mathbf{z}}^k + \mathbf{b}_{\mathbf{z}}^k\right),\ \text{for } k = 1, \cdots, K \tag{2.11}$$

$$\mathrm{z}_t = \sum_{k=1}^{K}\mathrm{z}_t^k = \sum_{k=1}^{K}\mathrm{o}_t^k \circ f_o\left(\mathbf{W}_{\mathrm{z}}^k\mathrm{A}_{\mathrm{z}}^k + \mathrm{b}_{\mathrm{z}}^k\right) \in \mathbb{R}^M \tag{2.12}$$

Finally, the output of the output gate as shown in Eq. (2.13) uses a weight matrix $\mathbf{U}_{\mathbf{o}}^k$

$$\mathbf{o}_t^k = \sigma\left(\mathbf{U}_{\mathbf{o}}^k\mathbf{A}_t^k + \mathbf{W}_{\mathbf{o}}^k\mathbf{z}_{t-1}^k + \mathbf{V}_{\mathbf{o}}^k\mathbf{x}_t^k + \mathbf{b}_{\mathbf{o}}^k\right) \in \mathbb{R}^M \tag{2.13}$$

The frequency state helps learn the dependencies of both low and high-frequency patterns. Learning frequency pattern ultimately improves the performance in the case of time series prediction problems. Time-frequency analysis is most famous for noisy time series prediction problems. This is a common feature for multivariate time series data, e.g. weather data, financial time series, pattern recognition from continuous-time data, etc.

Table 2.5 shows several works where time-frequency analysis is used for time series prediction. The most popular neural network for time-frequency analysis is based on ANN, LSTM, FFT and back-propagation neural network (BPNN) models. However, a few one-dimensional CNN based works also exhibit better performance in case of insufficient training data.

Table 2.5: Application of Time-frequency analysis

Application	**Base Neural network**	**Proposed work**
Rain-fall prediction	ANN	[82]
Multi-variate time series forecasting(e.g. wind, financial, energy consumption)	LSTM	[85]
Sea surface temperature forecasting	BPNN	[81]
Intelligent Fault diagnosis	CNN	[83]
Sleep stage classification	Cascade LSTM	[65]
Signal processing	Short-Time Fourier (STFNet)	[73]

Irregular data sampling rate and fixed time step

Existing deep learning models for solving continuous-time data usually use fixed data sampling rates, but the real-time rate is primarily irregular. For example, event stream data in the automobile industry, patient records in healthcare, weather data in climate applications, and real-time data demonstrate irregular sampling rates. So far, RNN based model is the pioneer among all state-of-art neural network models for modelling irregularly sampled data by considering the continuous-time data as a sequence of discrete fixed-step data that often suffers from loss inaccuracy.

The variable data sampling rate is another significant challenge for continuous-time data. Most of the time, in a continuous-time series, the state value does not update at every time step. Therefore, the value of the selected parameter cannot be found at

Table 2.6: Potential deep learning-based solutions for irregular sampling rate

Proposed works	**Neural Network Model**
[24]	GRU-D
[20]	Phased LSTM
[86]	Dilated RNN
[23]	Time- LSTM
[55]	SKIP -RNN
[87]	Latent-ODE
[50]	Temporal clustering
[48]	Multi-resolution Flexible Irregular Time series Network (Multi-FIT)
[88]	Neural Differential equation
[89]	RNN based Neural Differential equation

every state of the time sequence, which may lead to an inaccurate result. Therefore, solving irregular data sampling is very important. Moreover, every continuous-time series suffers from the problem of having an irregular sampling rate, where the interval between consecutive data collection points can vary in length. An excellent example of such a scenario in the healthcare industry is a patient health record. The interval of two consecutive hospital visits for a patient can be a few days or even a few years. Similarly, the voice command for Alexa can be sampled within a few minutes or even days. Therefore, an efficient deep learning model needs to understand the irregularity in the data sampling rate. Temporal clustering invariance[50] is a unique technique to solve irregular healthcare data by grouping regularly spaced time-stamped data points together and then clustering them, yielding irregularly-passed time-stamps. Another recent neural network called Multi-resolution Flexible Irregular Time-series Network (Multi-FIT) [48] fights against irregularly-paced observation of a multivariate time series. Instead of using the data imputation technique to replace missing observation used by [24], [48] uses a FIT network to create an informative representation at each time step using the last observed value or time interval since the last observed time stamp and overall mean for the series. Some recent works, as shown in Table [2.6], mainly focus on sampling irregular data sampling.

Informative missingness

One of the severe outcomes of the irregular sampling rate is informative missingness [24]. Informative missingness is a significant challenge in terms of processing time series. Problems in the prediction category suffer missingness challenges the most. Missing values in the time series and their missing patterns are often correlated. Understanding this correlation can lead to better prediction results. Although due to some valuable attributes in the design, RNN is well equipped to capture long-term temporal dependencies and variable-length observations. There are some relevant models based on RNN, as shown in table 2.7. Among these works, [24] impressively improves RNN structures to incorporate the patterns of missingness for time series classification problems. Besides, the sparse and asynchronous nature of the data sampling rate causes missing observations. For each missing observation, the time series modelling gets interrupted and can never recover again. The primary way to battle this irregular data sampling rate is to impute the missing data to provide value for missing observations similar to proposed in GRU-D[24] model. Another efficient mechanism, usually used by RNN based models, is to let the model know when there is data available and take action accordingly, as demonstrated in Phased-LSTM [20] model. Over the last decades, several research works have been motivated to fix the informative missingness due to missing observations in the time series. Here are some conventional approaches to solve this problem as follows:

1. To ignore the missing data point and to perform the analysis only on the observed data. The limitation of this approach is that if the missing rate is high and the sampling rate is too few, the performance decreases significantly.

2. To substitute in the missing values using data imputation. Data imputation mechanism [24, 90, 91] are widely popular for solving missing data. However,

data imputation does not always capture variable correlations, complex patterns and other essential attributes. GRU-D [24] is based on the idea that. This model combines two representations of missing patterns, such as masking and time interval, in the deep learning architecture to detect the long-term pathological time dependencies in the time series and uses the missing pattern to achieve better prediction results.

3. To Leverage ordinary and neural networks based on partial differential equation helps to learn the change in data over time and use the instant derivative of the state of any dynamical system over time to replace the missing observation state value.

Table 2.7: Different NN models as solution for informative missingness

Proposed work	**Underline Neural network model**
[24]	GRU-D
[57]	Phased -LSTM- D
[20]	Phased LSTM
[92]	partial Differential equation based
[93]	GRU-D based Neural Differential equation

Recovering Missing data from the asynchronous at a heterogeneous sampling rate is challenging but essential for the model's efficiency. The missing entries in a continuous-time series can result in large and random distribution. The phased-LSTM [20] has introduced the time gate to overcome complexity with the asynchronously sampled data, and it can sample data at any continuous-time within its open period. Several recent works emphasise hybrid LSTM to recover missing data. For example, J. Zhou et al. [74] proposed a neural network consists of the standard LSTM [22] for regularly sampled data and the Phased-LSTM [20] for irregularly sampled data as shown in Eq. (2.14). The benefit of model learning is that it utilises the information from collected observations and previous input's missing values and, thereby, deals with the data sparsity caused by missing data.

$$\begin{aligned} h_s^f &= LSTM^f\left(h_{s-1}, x_s\right) \\ h_u^b &= LSTM^b\left(h_{u+1}, x_u\right) \end{aligned} \tag{2.14}$$

Interpolation and data imputation are the most commonly used mechanism to overcome the informative missingness. Here is a list of widely used imputation mechanism

- PCA (Principle component analysis)
- MICE (Multiple Imputation by chained equation)
- MF (Matrix Factorization)
- Miss Forest

One well-known technique is to sample the data only when available. Several neural networks [20, 23, 55] are designed with this principle. A discrete-time series can be defined as Eq. (2.1), which is a sequence of observed data points at consecutive time steps. For the time series X, x_i is a set of observations at time step t_i. Usually, the time sequence is long, and it needs sampling to optimise the observation data set at different time steps. Some steps can have data, while others may have no data for the underlined subject of observation. Therefore, neural networks require learning a long time series where data is collected at an arbitrary sampling rate. Missing data patterns in time series impact the outcome significantly. As a result, neural network models must overcome the informative missingness.

Temporal and Spatial Coherence

Real-world continuous-time data dynamically evolve continuously with noise as well as statistical properties. Both temporal and spatial coherence influence the neural network's efficiency used for time series modelling. Temporal coherence refers to the correlation between $z(t_1)$ and $z(t_2)$, z is a vector representing any dynamical system, and $z(t_1)$ and $z(t_2)$ are the state value of z in two different data points t_1 and t_2. Different real-world applications [94] are tightly dependent on temporal coherence. The recent trend of data-driven neural network [89, 95] supports the importance of temporal coherence in most time series problems. Some domain-specific continuous-time data, such as prediction of wind, energy consumption, climate phenomenon [96, 97], exhibits coherent spatial pattern in corresponding multivariate time series. Wavelets analysis [33, 98, 99] is one of the most popular mechanisms to learn transient coherent patterns in time series. Dynamic spatial correlation for multivariate time series [54] combines wavelet analysis along with a non-stationary Multifractal surrogate-data generation algorithm to detect short-term spatial coherence in multivariate time series. The stochastic process generates the surrogated data, preserving the amplitude and time-frequency distributions of the original data. An LSTM based neural network [10] demonstrates the usage of spatial coherence in neural network modelling for time series with Spatial Coherence.

High Dimensionality

Real-world time series contain noise components that may add additional complexity to the modelling and processing of time series [54, 76]. Besides, multivariate time-series data samples exhibit high dimensionality. For time-series data processing, it is essential to optimise the noise and Dimension. There are several mechanisms to overcome high dimensionality problems for multivariate time series, such as restricted Boltzmann Machine [21], feature extraction, wavelet analysis, filtering feature and others. One solution is to use Wavelet analysis which removes some noise and reduces the dimensionality. Different kinds of optimisation functions can be used to optimise the weight matrix. Peng et al. [53] have transformed the weight matrix optimisation problem into a dual Lagrangian problem to overcome the high dimensionality. Another popular solution is feature extraction. In feature extraction, neural network models only focus on a set of features from the long time series for computation. However, this can negatively affect the time series process as an essential or relevant feature can be excluded, resulting in misleading results. On the other hand, if the number of selected features is large, the associated computation, time, memory, and energy would be extensive, negatively influencing the performance.

Over-fitting

Any deep learning model must provide a mechanism for avoiding over-fitting. In the case of modelling continuous-time data, it is mandatory and challenging. Some common approaches for avoiding over-fitting are listed as follows.

1. Bach normalization [100]
2. Dropout [25]
3. Optimizing weight matrix [53]
4. Using un-tuned weight matrix
5. Reducing the Dimension of the representation of the input vector by shifting a row or column
6. Using Gaussian or similar process for feature extraction
7. Using comparatively less memory size so the model cannot memorise the input sequence
8. ANN-based model uses an early stopping procedure to avoid the over-fitting problem. In the case of early stopping, a predetermined number of steps controls an automated stopping procedure.
9. Using a broad set of training and test datasets can reduce the chances of

over-fitting.

10. Regularisation is a well-known procedure for overcoming over-fitting. Sampling noise is a very excellent solution for Regularisation.

11. Fine-tuning of the neural network model prone to stop early, thereby reducing over-fitting.

12. A unique method generative pre-training [101], where representation mapping of input data is done using feature detectors before the actual training. In this process, multiple layers of feature detectors are for a discriminative fine-tuning phase and adjust the weight found in the pre-training phase. This process significantly reduces the chances of over-fitting.

Length of Sequence

Recently, several works, as shown in Table 2.8, argue that the long time series is more efficient for learning the continuous-time data. Usually, a long time series has shorter time steps that improve the result's accuracy, but it still needs to overcome the higher sampling rate complexity. At the same time, the longer the sequence is, the more complex the neural network needs to be.

A minimal number of works demonstrate efficient processing of lengthy continuous-time series. The usual approach is to normalise the long continuous-time data and train it in multiple batches with a subset of data for each batch. The Wavenet [102] is one of the most popular works on long sequence time series classification. Dilated convolutional neural networks are often used to overcome the sequence length problem. Another new addition to this solution is phased-LSTM, a number of variations [23] of Phased-LSTM [20] have been proposed since 2016 to combat the larger sequence length problem.

Memory size

Memory size is another problem that may arise due to the sequence length of the time series. RNN is more famous for memory-related problems than ANN and CNN due to its ability to recover memory from long past events. However, recently Temporal Convolution Networks (TCNs) have achieved better performance in maintaining a more extended history than RNNs.

Table 2.8: Accuracy on the copy memory task

Model	**Sequence Length**	**Accuracy**
LSTM	50	$<$ 20%
GRU	**200**	**<20%**
TCN	250	100%

Table 2.8 shows that for the same data set as described in [91] TCNs maintains 100% accuracy for all sequence lengths. On the other hand, the accuracy level falls below 20% for LSTMs and GRUs after the sequence length reaches 50 and 200, respectively. For LSTM and GRU, the model quickly degenerates to random guessing as the sequence length grows. TCN is an excellent example of a hybrid deep learning model, where the attributes of CNN and RNN are combined efficiently. Large memory size is not a feasible choice for modelling neural networks as that may cause an over-fitted training model.

2.5.2 Pathological long-term dependencies

Another sequence-related limitation of deep learning architecture for time series problems is pathological long-term dependencies [103]. Some of the data points in a long time series dataset, as shown in 2.1, can be related, while some others are entirely independent. An excellent example is described in [57], where N consecutive visits of a single patient can be related to the same physical problem where the range between two consecutive visits is comparatively short, while N consecutive visits of a single patient can be for entirely independent of each other. However, the range between two consecutive visits is relatively broad in this case. Usually, problems in long-term time series prediction problems have a long sequence of inputs. The future prediction for such an input series can only depend on a few time-stamps at the end of the input series, and there is a long irrelevant part of random input in the sequence. Healthcare shows this kind of problem more often than in other sectors. This kind of long-term pathological dependency makes the problem even harder to solve. The challenges increase relative to the length of the sequence (L) as longer sequences exhibit a broader range of dependencies.

Influential feature selection

Time series usually have a massive number of parameters. In order to model the time series, some of the parameters are more significant than others. Also, domain-specific feature strongly influences the accuracy of the result generated by different neural network model. As a result, different models demonstrate different performance for similar tasks with the same dataset and domain. The correct feature for each task is critical for time series modelling, especially in the time series classification problem. However, it is not feasible to design a domain-specific neural network for different tasks. Therefore, it is essential to distinguish the essential parameters and removes irrelevant parameters to reduce noise. If the length of the time series is L and M is the number of features to describe any problem, the scope of M features is M$\times$L. For this reason, influential feature selection is crucial. Many methodologies are available for feature extraction from different static domain-specific data, for example, static images and computer vision. However, extracting features from continuous-time data is still an open problem. One solution to improve the efficiency of a neural network model in terms of feature selection is that the design of the models needs to be data-driven.

Several data-driven neural networks, as shown in Table 2.9 demonstrate promising results for feature selection and optimising the performance of neural network models.

Table 2.9: Data driven neural network

Proposed work	**Underline Neural Network**
[104]	Neural ODE
[92]	Neural PDE
[95]	PDEs with Differentiable Physics
[95]	PDEs with Differentiable Physics
[105]	Symplectic ODE-Net

Another approach to overcome the challenges of proper feature selection is using external methodologies to select features during the data collection and cleaning phase before training the neural model and later using that feature during the training phase. Some of the efficient methodologies for feature selection are mentioned in Table 2.10.

Table 2.10: Efficient methodologies for feature selection

Proposed work	**Methodology**	**Remark**
[35]	Generic	time series classification.
[47]	multi-objective evolutionary algorithms (MOEA) along with evaluators based on the most efficient state-of-the-art regression algorithms	This method improves the performance of multivariate time series forecasting

2.5.3 Challenges faced by different Applications

There are several applications of sequential data modelling, for example, speech recognition, bioinformatics and human activity recognition. Different real-world application data suffers from different challenges mentioned in this section. Table 2.11 shows how different characteristics of continuous-time data affect different applications.

Table 2.11: Challeneges for proecessing continous-time data in different applications

Challenges	HealthCare	Finance	Event Based IoT	Frequency Anasis & Prediction	Classification
Informative Missingness	✓		✓		✓
Irregular sampling rate	✓		✓		
Higher sampling frequency			✓		
High Dimensional Data			✓		✓
Pathological dependencies	✓	✓			
High Memory Consumption			✓		
Complex Computation	✓		✓	✓	✓

Table 2.12: Different Projects in Health care with time series

Health care projects	**Deep learning models**
Heart failure prediction	Doctor AI [106]
Patient Visit analysis	Time LSTM [23], Phased–LSTM-D [57] Choi, Edward, et al. [2]
Multi-outcome Prediction	DeepPatient [107]
Sleep stage classification	[66]

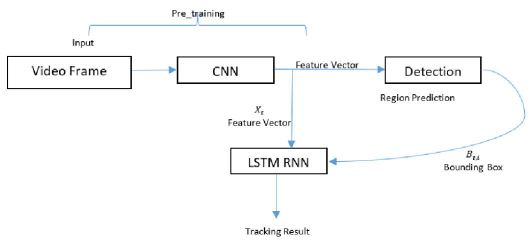
As shown in the table 2.11, healthcare data significantly suffers from informative missingness, sampling irregularity and Pathological dependencies. Table [2.12] depicts the current trends of using continuous time series as the data model for health care problems. As a result, most research in the Healthcare industry uses RNN based neural networks. However, due to privacy, the Healthcare industry suffers from a lack of actual clinical data. Therefore, some researchers use GAN based neural networks to generate sample clinical data for further research.

Continous-time data is the primary block for the Internet of Things(IoT). The work presented [20] has significantly contributed to opening new areas of investigation for processing asynchronous sensory events that carry timing information. This work has been extended and used in several applications [26, 75, 108, 109]. Some core characteristics of sensor-based data are explained by[20], such as *Irregular sampling rate, Higher sampling frequency and High Dimension.* A high Dimension is an unavoidable outcome of multivariate time series. Most sensor-generated data for weather, climate, automobile and other applications are generally multivariate. For example, sensor data from wearables consists of multiple data channels, such as step-count in intelligent mobile devices that use the accelerometer and optical heart rate sensor. This sensor collects data on heart rate and step count. There are several use cases where data is coming from multiple channels. For a more specific example, the heart rate monitor in smart mobile devices. In recent days, most research fields deal with a temporal sequence where the input channel is more than one. In addition, another significant limitation is that it requires long time series with higher frequencies and short time steps to predict an asynchronous future event from n previous events from the sequence. This processing possesses a huge computation load and energy consumption. For multivariate sensor-generated time series, RNN and CNN based hybrid models [10, 20, 24, 84] are comparatively better in performance as well as accuracy.

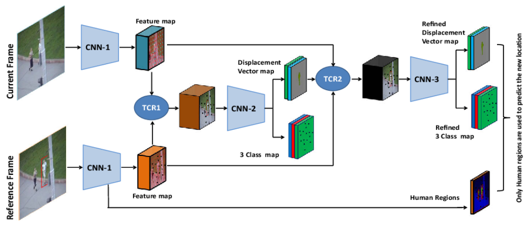
The early and most popular application of time series is Time series frequency analysis for future prediction. Deep learning state of artworks for predicting future events has achieved impressive accuracy and performance. However, the current works are mainly based on an underlying assumption that the test data and the train data share a similar scenario [62]. However, in reality, that is hardly true. Therefore, only a few relatively simple practical sequence models with a fixed data rate and some semi-supervised learning algorithms can provide satisfactory results under specific circumstances with state-of-the-art research works. In addition, the performance of these deep learning models for processing time series does not provide expected performance in real-world problem-solving.

Besides time-series frequency analysis, continuous-time data has become very popular in event prediction. Both regular and rare event prediction tasks, deep learning algorithms use time series analysis. However, this is challenging as the time series is mainly a multi-variant series consisting of heterogeneous variables. In addition, the dependency between variables is very complicated. Data are sparse and not sampled uniformly. The irregular sampling rate is prevalent in the case of event prediction. Furthermore, the series is asynchronous, making the task even more complicated. So far, the most used solution is to divide the problem up into a homogeneous variate. Therefore, several works[20, 23, 24, 57] focus on dealing with the irregular data sampling rate and asynchronous time series.

Visual recognition is a perfect example of time series classification or detection from continuous-time data. Recently, human activity recognition from the video has become a trendy field as an example of temporal sequence learning using a deep learning algorithm. The main challenge in processing the time sequence from the video is that the data is spatially imbalanced with an irregular sampling rate. The quality of the image and the object's place in the image make this even more challenging. In a single video frame, there can be many parameters. On top of that, additional noise in the data set makes it harder for a deep learning algorithm. Hybrid neural network models are more successful than traditional vanilla models in this area. A combination of CNN and LSTM networks can successfully overcome the problem of spatially imbalanced data. In most cases, CNN models outperform other neural network models for continuous data visual recognition and image classification tasks. For example, a recent deep learning model [63] has proposed a similar architecture, where a CNN model is used for the feature extraction phase. Later the feature vector an LSTM was placed in the model architecture for human tracking detection and result tuning as shown in Fig[2.5.3]. Fig[2.5.3] shows a similar deep learning model [110] where multiple CNN models are used as a cascaded CNN model.



(a) Hybrid CNN-LSTM Model



(b) Cascaded CNN Model

2.6 Different Neural Networks Models For Time Series Processing

This section presents different neural network architectures, which are generally used to learn a continuous time series. The different neural network models have their strengths and weaknesses in modelling continuous time series. For example, fig. 2.6 shows the primary types of neural networks for modelling continuous-time series over decades.

A single type of neural network is not suitable for all different solutions. Based on the nature of the time series, the different model performs better than others.

2.6.1 Artificial Neural Network (ANN)

Wind speed prediction, energy prediction, financial time series forecasting, and many other continuous series prediction problems have been solved using the Artificial neural network (ANN). In early 2000, along with the CNN and the RNN based solution, ANN became a widely popular neural network for modelling non-linear time series. ANN does not need any previous assumption or linearity. The ANN successfully provides a mechanism to determine the number of neurons fired in the hidden layer. Some other characteristic features of ANN are listed below

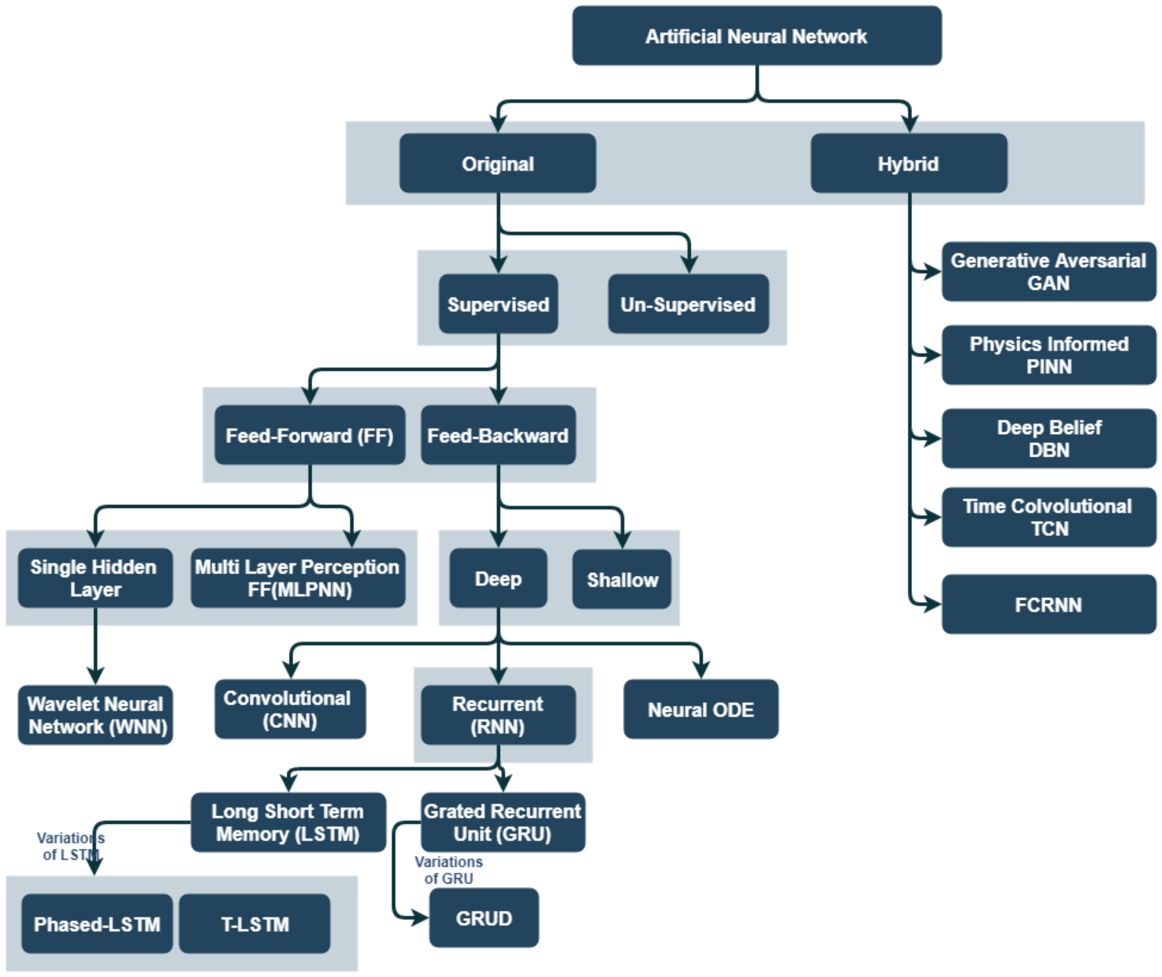


Figure 2.6: Different types of neural network used in continuous time series modelling

- ANN can derive its computing power through massively parallel distributed structure and learn the corresponding model.
- The architecture of the model is straightforward. It consists of units. These units are connected using the symmetric weighted connection. This connection can be either one-directional or bi-directional. The weight of the connection can be inhibitory or excitatory.
- Usually, a sigmoid function is used as the activation function, also known as the squashing function. Other activation functions such as Linear, Atanh, Logistic, Exponential and Sinus are used as the activation function in the hidden and

output layers.

- As the ANN is simple in terms of design and system structure, it is possible to use different artificial intelligence algorithms, such as particle swarm optimisation algorithm (PSO), for learning and adjusting the parameters. Furthermore, due to the simplicity of the underlying design of ANN, most ANN-based models are hybrid, whereas other algorithms are integrated to improve the performance of ANN. Some of the most popular methods used with ANN for time series prediction, classification and detection problems are as follows:
  - Hidden Markov model (HMM)
  - Genetic Algorithms (GA)
  - Generalised regression neural networks model (GRNN)
  - Fuzzy regression models
  - Simulated Annealing algorithm (SA)
  - Echo State Network (ESN)
  - Particles Swarm Optimisation algorithm (PSO). [111]
  - Elman Recurrent Neural Networks (ERNN) [60]
  - Hydrodynamic Neural Network
  - Back Propagation Neural Network (BPNN)
  - Recurrent Multiplicative Neuron Model (RMNM)[112]
  - Wavelet Neural network (WNN) [78]
  - Hilbert-Huang transform (HHT) [15]
  - Multilayer Perceptron Networks (MLP)

    - Support Vector Machine (SVM) [79, 80]

- Mean square error (MSE) and mean absolute percentage error (MAPE) is the loss functions used in ANN-based models.
- Artificial neural networks (ANNs) are very suitable for time series prediction, univariate forecasting, financial trend detection, wind speed and water fluctuation detection, and similar continuous-time problems. This neural network is also beneficial for pattern classification, and pattern recognition [34].
- ANN models are mostly data-driven without any initial assumption.
- ANN models usually perform better for time series prediction because of their ability to model any continuous functional relationship between input and output.
- ANN models can capture the underlying non-linearity of the system with highly non-linear dynamics by using a non-linear activation function.
- ANN model can adapt conditional training quickly. Therefore, conditional time series forecasting is one of the most used areas for ANN.

Challenges of ANN

The main challenges of ANN models are as follows

- *Less Robust*: Noise data influences the optimisation as well as the robustness of the model. if the noise is not handled properly in the design of the network model, it can reduce the robustness of unknown input data.
- *Optimum architecture*: The structure of the neural network model controls the performance of the model. therefore, it is crucial to determine the architecture or structure design of the network, including the number of layers, number of neurons and other parameters.

- *No previous memory*: ANN models can not preserve the previous state in the memory.
- *Back-Propagation Error*:
- *Performance accuracy*: ANN models often suffer from reduced accuracy for multi-variate, missing patterns and other complex problem. So, the feasibility of the proposed methodology needs to be evaluated against a different kind of dataset. The prediction accuracy of ANN models directly depends on the right choice of parameters.
- *Selection of right related variable and parameters*: Different related variable such as the weight for each connection between neuron controls the accuracy and robustness of the model. Therefore, most ANN models pay attention to determining intelligent techniques to choose the correct variable and evaluate them over time. Selection of the right variable or parameters also expands to determine data transformation, initial values of the parameters, and stop criterion. It is essential to choose the right parameter to avoid unnecessary overfitting.
- Time series learning: ANN models need to capture the main features of the continuous-time series and its generalisation.

Recent ANN models to overcome different challenges

Recurrent Multiplicative Neuron Model(RMNM) proposed by [112] is a recent work deal the lagged variable of error as input along with its recurrent structure in the case of learning time series. It overcomes the traditional challenge of ANN of deciding the number of neurons in the hidden layer. Fig 2.7 shows the RMNM model architecture.

Another significant limitation of ANNs is that they cannot preserve the previous state in memory. SRWNN [113] proposes an outstanding solution for memory state

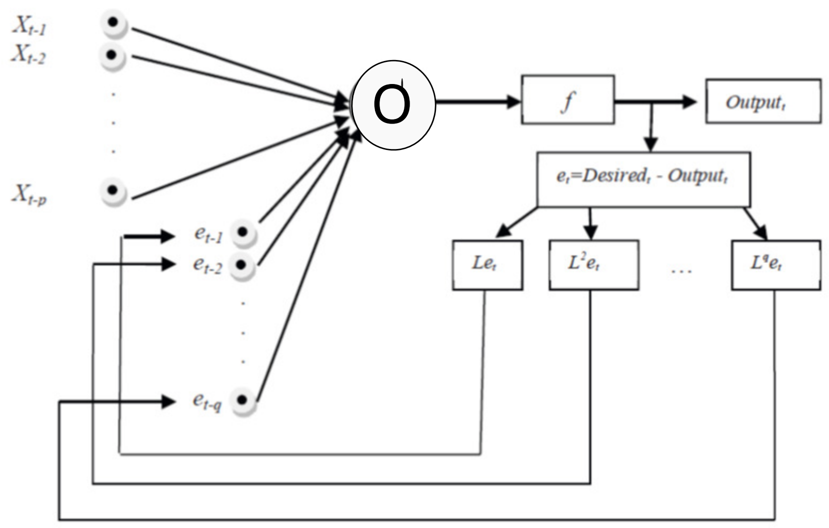


Figure 2.7: The Architecture of RMNM [112]

preservation. Fig 2.8 shows the architecture of SRWNN. SRWNN accepts N inputs but generates only one output. SRWNN usually consists of four different layers. The first layer accepts the input signal. The second layer consists of wavelet neurons (wavelon). Each of the wavelon has its self-feedback loop, introducing the recurrent nature in traditional Artificial wavelet neural networks (AWNN). The wavelon loss function described in Eq. (2.17) computes the wavelet. The input of the second layer contains the memory term, as shown in (2.18). In this layer, the current dynamics of the system is conserved for the next step. The third layer is a product layer, and the fourth layer generates the output. Compared to AWNN, SRWNN shows that better performance is inefficient in solving the sequential temporal problem. SRWNN can store information temporarily. As a result, SRWNN is suitable for chaotic time series and non-linear systems.

Most of the above mentioned ANN models use hybrid methodologies to improve their performance. Despite AWNN's performance in sequence modelling, it has some limitations which often set obstacles to achieving a better result. The recent trends of using AWNN and CNN or RNN can overcome some limitations and achieve better

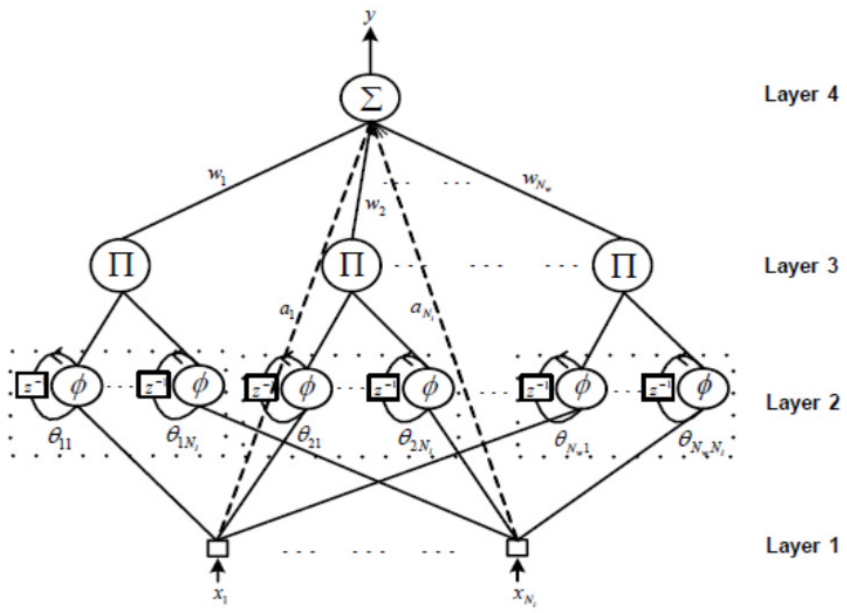


Figure 2.8: Architecture of SRWNN [113]

performance. However, AWNN is not famous for its sophisticated time series modelling.

[114] presented a comparative qualitative analysis of different ANNs proposed for forecasting time series in the time between the years 2006 and 2016. This analysis shows that the most successful ANN model is hybrid and different algorithms are used to improve the performance and overcome different challenges of multi-variate time series. Hybrid ANN models demonstrate better performance than basic ANN models, especially for dynamic datasets with a missing pattern, and hidden non-linear and non-stationary characteristics.

Choosing the correct variable, weight, and the initial parameter value is significant for artificial neural network modelling. Therefore, artificial intelligence algorithms such as Genetic Algorithms (GA), Simulated annealing algorithms and PSO algorithms are usually used in ANN to adjust the model's parameters.

Different ANN-based models adopt several approaches to overcome the limitation of back-propagation. For example, [115] combines a three-dimensional hydrodynamic model in conjunction with an ANN to improve prediction accuracy. Integration with reinforcement learning (RL) is another solution for back-propagation. Usually, the BP learning algorithm uses the traditional Euclidean distance to determine the error between the output of the model and the training data. However, in RL, the reward can be defined as an error zone. Therefore, the influence of the noise data can be reduced; at the same time, the robustness of the anonymous data may be raised [52].

There are different ANN-based neural network models, as shown in Table 2.13. These recent models adopt different hybrid mechanisms in order to overcome the challenges described in §2.6.1.

Table 2.13: Different Kind of ANN

Proposed work	**Challenges To Solve**	**Characteristics**
Recurrent Multiplicative Neuron Model (RMNM) [112] ARMA TYPE PI-SIGMA ANN [116]	Adjust the parameter as well as a related variable such as weights of neuron connection to avoid over-fitting and improve accuracy.	Use a PSO algorithm for learning.

Proposed work	**Challenges To Solve**	**Characteristics**
Wavelet net [78, 113]	Support vector machine, Wavelet decomposition	Usually Wavelet nets are hybrid models combining data-driven least square support vector machine (LSSVM), (ANN) and wavelet decomposition for disaggregation of continuous-time series. Instead of identifying the correlation among feature, [113] model proposed a new parameter named as mutual information (MI) which can be defined by Eq. (2.15) $$I(X,Y) = \iint dxdy\mu(x,y)\log\frac{\mu(x,y)}{\mu_x(x)\mu_y(y)} \qquad (2.15)$$
Hybrid [117]	Learn hidden patterns or dynamics of the system	Uses different methods such as singular spectrum analysis (SSA) to identify hidden oscillation patterns in the data
Multi-layer artificial neural networks [17]	Multi-variate and interdependent dataset	Uses Mutual Information(MI)based feature selection process In addition to the input layer and output layer of the neuron, this model has multiple layers of hidden units. These hidden units learn features and hidden dynamics of data with multiple layers of abstraction.

Proposed work	**Challenges To Solve**	**Characteristics**
Functional link artificial neural network (FLANN) [59]	Multi-variate time series, where along with continuous-time data, location is also an influential parameter for future prediction	Introduces a low-complexity artificial neural network and employing incremental and diffusion learning strategies
Hilbert-Huang transforms [15]	Analysing the spectral and temporal information of non-linear and non-stationary time series	Introduce hybrid ensemble empirical mode decomposition (EEMD)-ANN that uses HHT to identify the time scales and change in continuous data.
Hybrid ANN-(U-)MIDAS[118]	Explore hidden non-linear patterns in raw mixed frequency data without information loss	introduce mixed data sampling to use raw input directly without any latent preprocessing. Use back-propagation and chain rule to derive the gradient vector for optimisation algorithms
SciANN [119]	Solution and discovery of partial differential equations (PDE)	Uses Physics Informed NN and Variational Physics-Informed Neural Networks With Domain Decomposition

Proposed work	**Challenges To Solve**	**Characteristics**
hline Multiplicative Neuron Model [112]	Use simple activation function for hidden layer	The output of time series at step t , yt for a typical MLP can be defined by Eq. 2.16. $$y_t = G\left(\alpha_0 + \sum_{j=1}^{h} \alpha_j F\left(\sum_{i=1}^{p} \beta_{ij} y_{j-1}\right)\right) \quad (2.16)$$ Here, α, β are the weight for the neural network model. F and G is the activation function of the hidden and output layer, respectively, as described by (2.17) and (2.18). $$F(x) = \frac{1}{1+e^{-x}} \quad (2.17)$$ $$G(x) = x \quad (2.18)$$
Self recurrent wavelt neural network (SRWNN) [120]	This model is suitable for chaotic time series described in §2.2	The main self recurrent wavelet layer uses gradient descent method, which is derived from stability theorem along with adaptive learning.

2.6.2 Recurrent Neural Network(RNN)

In processing continuous sequence, the recurrent neural network is the pioneer. A repetitive cell connected with a lateral connection creates a more extensive neural network in an RNN that processes data sequentially, precisely what a temporal data sequence requires. Therefore, it is very efficient to model sequence structure. Even though RNN has demonstrated significant satisfactory result for modelling variable-length time sequence, as mentioned by [9] RNN architecture lack an intuitive spatiotemporal structure. To model time sequence is much more comfortable in RNN due to some novel features of RNN as follows

- RNN processes data sequentially, one record at a time. It can model sequences with recurrent lateral connections.
- It extracts the inherent sequential nature of time series.
- It can model different lengths of the series.
- It supports time distributed joint processing.

Although RNN is one of the pioneer deep learning models for realistic high-dimensional time-series prediction tasks, it has some its pitfalls. Some of the main drawbacks of RNN are as follows

Vanishing gradient problem: [121, 122]. Most of the current work mainly focuses on the Vanishing gradient problem. There is already a significant improvement in the case of dealing with the Vanishing gradient. The continuous-time series can be extended in length, where the data point in the later of the sequence can have a long-term dependency on the data point at the earlier state of the sequence. So, a much earlier time state can affect a much later time state in the sequence. As the neural network can be profound, even for a 100-layer neural network, it is challenging for the gradient to propagate back to affect the earlier state in the sequence. Back-propagate to an earlier state of the sequence to modify the computation in the earlier state. One of the major limitations of RNN and its different variants in the case of modelling a continuous time series is to overcome the Vanishing gradient. Training a complex deep learning architecture with multiple layers of hidden units becomes critical due to the vanishing gradient problem when the error is usually backpropagated. LSTM [22] is an impressive addition to the RNN family, which solves the vanishing gradient problem. Another way to solve the problem is to use unsupervised greedy layer-wise pre-training of each layer. This kind of regularisation helps to achieve better initialisation of a model.

Exploding gradient problems: Another significant limitation when using RNNs for time sequence modelling is the exploding gradient problem [123]. This problem makes it difficult to train a continuous time series using RNN.

Butterfly Effect: For example, due to the dependencies among different time states in a long term time sequence, a minimal change in an iterative process of an RNN can result in a significant change in future time states after n iterations, where $n \to \infty$ [124]. The main reason is that the loss function reacts significantly to small changes and can be ultimately discontinuous, which would result in a complete failure in terms of predicting real-time continuous time series [122]. Some of the conventional approaches to deal with the butterfly effect are to use different data imputation methods, e.g., PCI, MissForest, and KNN SoftImpute.

In this section, some of the critical architecture designs of RNN are discussed.

Long Short-Term Memory (LSTM)

Vanilla RNN has been used for time sequence processing for past years. However, processing time sequence using vanilla RNN requires extensive modification in the architecture. Therefore, several works have been presented in the last few years, where Vanilla RNN, mainly LSTM, has been modified and extended to improve the performance and accuracy of time series processing. In an LSTM, it is possible to pass information from one cell to the next selectively. This model can process long term pathological dependency along with short term dependency, which is one of the most significant challenges in sequence modelling [20]. Long Short-Term Memory (LSTM) [121] has already become famous for both long-term and short-term time steps for its performance and efficiency.

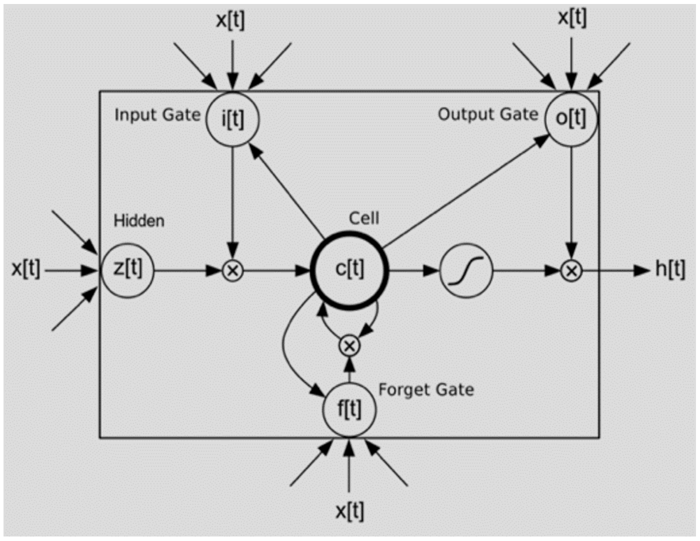


Figure 2.9: Traditional LSTM

As depicted in Fig. 2.9, traditional LSTM cell is composed of 5 different non-linear components described in Eq. (2.19) to Eq. (2.23). They interact with each other through linear interaction, therefore, information can be transferred through back propagation across time. Each cell has an input gate i[t] Eq. (2.20), output gate, o[t], Eq. (2.21) and forget gate f[t], Eq. (2.19). The cell state c[t], Eq. (2.22) represents the current state of the cell and hidden output vector $[h[$, Eq. (2.23) influence the computation of cell state of the next cell in the RNN sequence.

$$f\ \ [t] = \sigma_f(W_{xf}x\,[t] + h\,[t-1]\,W_{hf} + w_c f \odot c\,[t-1] + b_f) \tag{2.19}$$

$$i\ [t] = \sigma_i(W_{xi}x\,[t] + h\,[t-1]\,W_{hi} + w_c i \odot c\,[t-1] + b_i) \tag{2.20}$$

$$o\ [t] = \sigma_o(W_{xo}x\,[t] + h\,[t-1]\,W_{ho} + w_c o \odot c\,[t-1] + b_o) \tag{2.21}$$

$$c\ [t] = f\,[t] \odot c\,[t-1] + i\,[t] \odot\ \sigma_c(W_{xc}x\,[t] + h\,[t-1]\,W_{hc} + b_c) \tag{2.22}$$

$$h\,[t] = o\,[t] \odot \sigma_h(c\,[t]) \tag{2.23}$$

Here x_t is the input vector and h_t is the hidden output of LSTM cell at time t. Fig 2.9 represents an overview of how different components of the different cell in traditional LSTM cell interacts with each other.

Several recent research works [6, 9, 108, 125] have demonstrated significantly impressive results for modelling as well as processing time series. The main idea behind the success of LSTM in the case of modelling time-series is its universal approximation ability for the open dynamical system [120] which can learn the dynamics of the system and can generate it at the same time. For a deep LSTM neural model, it is not recommended to use a considerable memory size, as the model can be over-fitted by memorizing the input sequence data.

Several works [20, 23] have used different systems for time series modelling. For example, Shengdong Zhang et al. [10] have used a set of the temporal sequence of observation collected in n consecutive steps, as shown in Eq. (2.1). They also have considered that the number of observations in each set is not constant. Then they have used a sequence of labels $\{e_1, e_2, \ldots, e_n\ \}$ for X where $e_\mathrm{i}\{0, 1\ \}$. e_i= 1 indicates the event is observed, on the other hand, $e_{\mathrm{i}}= 0$ indicates absence of the event. In order to define the time gap between time duration between consecutive change in the value of e_{i}, this work has used a target label sequence y $= \{y_1, y_2, \ldots . , y_n \}$ $y_{\mathrm{i}} = 1$ refers that an event is observed in the next K time-steps, i.e. j+K j=i+1 ej > 0, and $y_{\mathrm{i}} = 0$ refers otherwise. In this work, K is referred to as a monitor window. This monitor window helps to build the distribution of positive as well as negative events efficiently as the value of e_{i} indicates if the event at a particular time step is either positive or negative. Feeding the entire sequence to the network at once is not recommended. This can impact the outcome negatively. An alternative best practice is to feed the time series by batch. The time sequence of length T is divided into M subsequence; each of them is of maximum length of $\frac{T}{M}$. This technique is known as time sequence chopping. The chopped size is selected very carefully; otherwise, this technique sacrifices temporal dependency longer than $\frac{T}{M}$ time steps. This LSTM based architecture has been used for fault prediction on multi-variant heterogeneous time-series data. One of the most significant contributions of this work is that it handles time sequence modelling for rare events where the distribution of positive and negative samples are highly imbalanced.

LSTM is the most used neural network model to solve time series problems. Table [2.14] shows some recent work where LSTM has been used on order model time series.

Recent trends are focusing on improving or modifying the architecture to model time series and related problems. Some of the promising models are described here as follows:

Adaptive time scales Recurrent Units (ARU) Adaptive time scales Recurrent Units is a recent variation of RNN [126], where time constraints τ_r is used to drive

Table 2.14: Different Types of RNN for Modelling Continuous Time Series

Neural network Model	**Related work**
Vanilla RNN	Skip-RNN [55], Structural-RNN[9], State-Frequency-RNN
LSTM	Phased LSTM [20], Time LSTM [23], Phased LSTM-D [57]
LSTM + CNN	Fcn-rLSTM [10]
GRU	GRU-D [24]

the rate of firing $-r_n$ neuron of a neural network unit as shown in Equation.

$$\tau_r \frac{dr_n}{dt} = -r_n + f\left(I_n\right) \tag{2.24}$$

Here f is a *sigmoid* activation fucntion.

Eq (2.6.2) shows the complete model for Adaptive time scales Recurrent Units (ARU). The input signal x is processed using weight matrices W, U and V. N-dimensional Intermediate state variable I and r are used to capture synaptic couplig between neurons. $\alpha_s = \Delta t/\tau_s$ and $\alpha_r = \Delta t/\tau_r$ are rate constants to define the relation between state variable and time constraints τ_r and τ_s.

$$\begin{aligned}
&\mathrm{I}_t = \left(1-\alpha_s\right)\mathrm{I}_{t-1} + \alpha_s\left(\mathrm{W}\mathbf{r}_{t-1} + \mathrm{U}\mathbf{x}_t\right)\right) \\
&\mathbf{r}_t = \left(1-\alpha_r\right)\mathrm{r}_{t-1} + \alpha_r\left(\mathrm{f}\left(\mathrm{I}_t\right)\right) \\
&\mathrm{y}_t = \mathrm{f}\left(\mathrm{V}\mathbf{r}_t\right)
\end{aligned}$$

The significant contribution of this paper is to identify the impact of slower rate constants that can lead to longer memory retention, which enables the maintenance of memory for more extended periods. ARU is a suitable solution for vanishing gradients problems.

Phased-LSTM Neil et al. [20] have added a new time gate to the existing LSTM, and the timestamp, which is the input for the new time gate, controls the update of the cell state, hidden state and the final output. For improved performance, this model of LSTM has been designed to sample time rates captured in the model's active state only. Therefore, associated sample time rates are captured only when the time gate is open. As a result, Phased-LSTM can attain a faster learning convergence in the training phase. The phased LSTM described in Fig [9] shows that the time gate decides the phase as either openness when the time gate value raises from 0 to 1 or closed when the corresponding time gate value drops from 1 to 0. The cell state is only updated when the time gate is open. Eq. (2.25, 2.26 and 2.27) describes the time gate in three different phases shown in Fig [2.6.2 and 2.6.2]

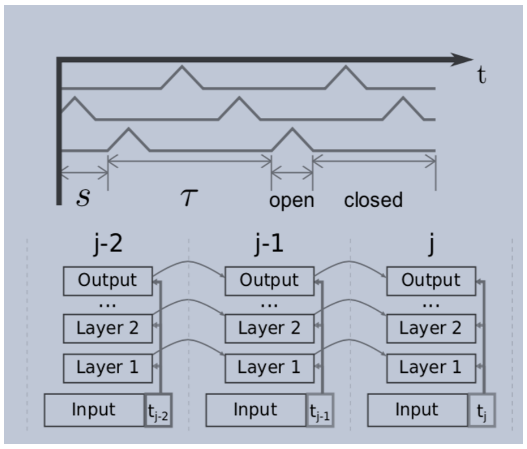
$$k_{topen1} = \frac{2\emptyset_t}{r_{on}} \;\; if\ \emptyset_t < \frac{1}{2}\, r_{on} \;\; and\ \emptyset_t = \frac{(t-s)\ \ mod\ \tau}{\tau} \tag{2.25}$$

$$k_{topen2} = 2 - \frac{2\emptyset_t}{r_{on}} \;\; if\ \frac{1}{2}\, r_{on} < \ \emptyset_t < \ r_{on} \tag{2.26}$$

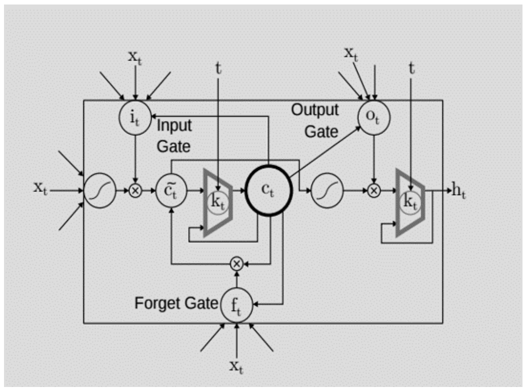
$$k_{tclose} = \ \alpha\emptyset_t \tag{2.27}$$

The leakage rate (α) is active when the closed time gate. The ratio (r_{on}) of time duration when the gate is open to the entire period helps to control the open phase and close phase of the gate to fine-tune the sequence. Another important controlling parameter is s, which controls the shit between the open and close phase of the time gate (τ) to determine the duration for each oscillation period.

The main advantages of this model are that it uses the time gate to convert the continuous-time series to a discrete sequence of sensor input with a minimum loss.



(a) The openness of Phased LSTM



(b) Phased LSTM Model Cell

Therefore existing LSTM can deal with the discretized sensory event. One of the limitations is that this work is still converting the continuous-time series to a discrete sequence; therefore, the accuracy is still not 100 percent. Another limitation of this work is that it cannot de-sparsifying sparse sensor data. Furthermore, this model cannot reverse back to the original continuous time series from the discretized sequence. Therefore, there are still missing inputs, and the input is not entirely asynchronous.

Phased LSTM-D For continuous sequences such as Electronic Health Records (EHR), where each feature is asynchronously sampled, it is challenging to apply Phased-LSTM straightforwardly. EHR data sampling can be used for disease diagnosis and prediction. In the case of EHR related problems for an individual patient, the time gap between two consecutive visits is irregular, and also, all the clinical data about the patient collected during other visits are not the same. Some features may be collected the first time and may not be collected next time. Therefore, continuous data sampling is irregular and has missing parameters. Therefore, S. J. Bang et al. [57] have proposed Phased-LSTM-D based on Phased-LSTM [20] to achieve better prediction. Phased-LSTM-D is mainly designed to solve the irregularity of the sampling interval and the asynchronous feature.

Phased-LSTM-D, as shown in Fig [2.10], is a new predictive deep learning model that can simultaneously however, individually deal with two missing patterns are as follows

- The irregularity of sampling interval
- Asynchronous sampling features

The new feature introduced in Phased-LSTM-D is the decay rater (γ_t) in the existing Phased-LSTM. This additional parameter introduces a decay mechanism. Input and hidden states have corresponding decay rater γ_x and γ_h for each event-driven time point t. Therefore, if the inputs are missing at some time t, the missing features are replaced with a weighted sum of the last and average measurements. The decay rate at time t controls these. Similarly, the previously hidden state is also updated with a fraction of the previously hidden state controlled by the decay rate.

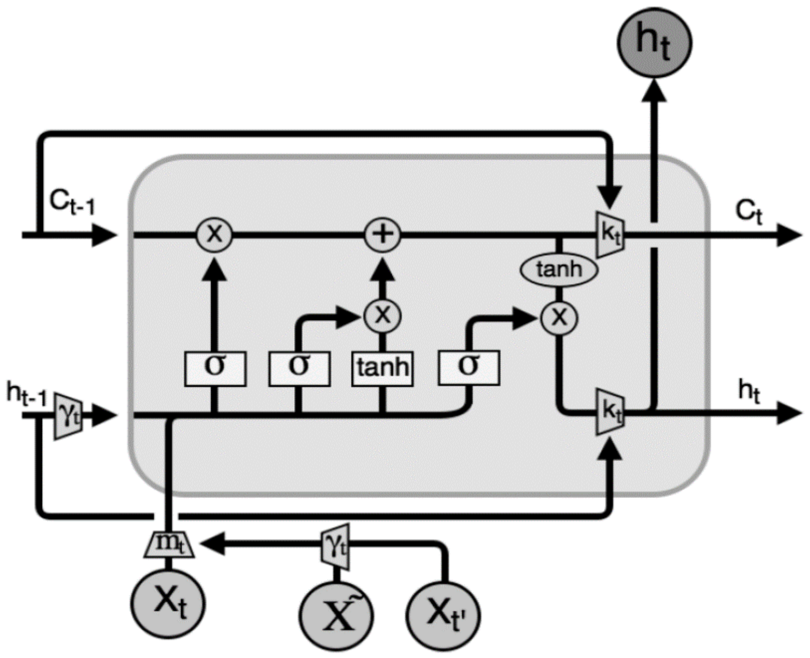


Figure 2.10: Cell of a Phased LSTM-D model

Phased-LSTM-D is designed to take advantage of both Phased-LSTM and GRU-D[24]. In this work, the new vector j, which keeps track of the timestamps when the time gate of Phased-LSTM [20] was open, and the cell state was updated. Therefore, j and j-1 represent the timestamp when the cell was updated consecutively. Phased-LSTM-D adopts Phased-LSTM to solve the irregularity, and the additional requirement for Electronic Health Records (EHR) is to solve the missing feature at every update time point. In this work, in terms of EHR, the state of an individual patient (n) is updated at each time sequence for j =1, 2,3 .... Tn and only then the time gate is opened. At the same time, at each timestamp, the missing feature x_j is replaced by a weighted sum of the previous measurement (x_{j-1}) and an average measurement (x_j) of the same feature, controlled by decay rate (γ_j). According to Phased-LSTM-D, along with cell state, which is the state of the patient, the previous hidden state (h_{j-1}) is also replaced by a fraction of the previous hidden state controlled by decay rate (γ_j). Therefore, when the time gate is open, the missing feature can be described as Eq. (2.28)

$$x_j^{(d)} = m_j^{(d)} x_j^{(d)} + \left(1 - m_j^{(d)}\right) \gamma_{x_j}^{(d)} x_{j'}^{(d)} + \quad \left(1 - m_j^{(d)}\right) (1 - \gamma_{x_j}^{(d)}) \hat{x}^{(d)} \tag{2.28}$$

$x_j^{(d)}$ determines the value of the d-th variable at time t_j, The empirical mean of the d-th variable ($\hat{x}^{(d)}$) and the previous value of the d-the variable $x_{j'}^{(d)}$ are used as weight is an order to determine $x_j^{(d)}$. The value for masking variable $m_j^{(d)}$ is either 1 in the case of the feature is not missing and 0 otherwise. Therefore, this masking variable controls if the measured value for the missing feature would be replaced or not. If $m_j^{(d)}$ =1, the observed value is used, otherwise, it is replaced by a weighted sum of $x_{j'}^{(d)}$ and $\hat{x}^{(d)}$.The decay rate is presented as a D-dimensional vector as in Eq. (2.29)

$$\gamma_{x_j} = \exp - \max(0, W_{\gamma_x} \delta_j + b_{\gamma_x}) \tag{2.29}$$

W_γ and b_γ are modelled parameters which are trained jointly. δ_j the time difference between two consecutive time sequences when the missing feature was observed and available. In this work, the decay rate also controls the change in the hidden state as shown in Eq. (2.30),

$$h_{j-1} = \gamma_{h_j} \odot h_{j-1} \tag{2.30}$$

In addition to existing Phased-LSTM [20], Phased-LSTM-D has used masking variables and time interval vectors for training additional hidden layers to estimate the decay coefficient. Therefore, the proposed work enables simultaneous decay in input and hidden state and model training. This work introduces Phased-LSTM in the field of predictive modelling for continuous-time series with sampling irregularity. The benefits of this proposed work over another approach is that it introduces additional masking and time interval to impute missing features instead of the direct imputation of the missing features. This significantly improves the performance and accuracy.

Time LSTM The recommendation system can demonstrate another essential use of continuous-time series prediction. Recommendation systems need to analyze the insight of some features in the continuous or discrete sequences of the user's action. The different recurrent neural network architectures are recently being used for the recommendation system. In the case problem domain such as recommendation system, it is not enough to only consider the order of different time steps as, like a traditional recurrent neural network, it is also necessary to establish a relationship between the sequence of time series. In a recommendation system, it is necessary to store the user's short-term actions. Such as, if a user has purchased a flight ticket, he/she might need to have a recommendation for the hotel. This is an example of short-term dependencies. At the same time, some recommendations needed to consider a user's past actions for a long time ago. Time-LSTM [23] designed the time gate of Phased-LSTM to capture both long-term as well as a short-term dependency to predict the next time sequence or recommendation for the user. However, Time-LSTM has adopted the time gate of Phased-LSTM to model the timestamp time gate may or may not implicitly capture the time intervals and the consecutive data points, where Time-LSTM explicitly models time intervals, thereby capturing the relation between two consecutive observations or data points. The main component of Time-LSTM is the time gate which has been designed in three different ways to capture the relation between time intervals.

Based on the nature of the time gate, Time-LSTM has three different models as follows

1. Time-LSTM 1
2. Time-LSTM 2
3. Time-LSTM 3

Time-LSTM 1 This Time-LSTM architecture has only a one-time gate used to exploit time intervals simultaneously, both long term and short term. The newly introduced time gate for Time-LSTM 1 for m-th sequence in the time series can be described as Eq. (2.31).

$$T\left[m\right] = \sigma(x\left[m\right]W_{x}t + \sigma(\Delta t \Delta t m W t t + b t)) \tag{2.31}$$

In this proposed architecture of LSTM, the cell state and the output gate has been modified as follows Eq. (2.32) and Eq. (2.33)

$$c\left[m\right] = f\left[m\right] \odot c\left[m-1\right] + i\left[m\right] \odot T\left[m\right] \odot \sigma_{c}\left(x\left[m\right]W_{xc} + h\left[m-1\right]W_{hc} + b_{c}\right) \tag{2.32}$$

$$o\left[m\right] = \sigma_{o}(x\left[m\right]W_{xo} + t\left[m\right]W_{t}o + hm - 1W_{h}o + w_{c}o \odot c\left[m\right] + b_{o}) \tag{2.33}$$

Here $\triangle t$ is the time interval, and σ is the sigmoid function. Therefore, the cell state c[m] is filtered by the input and time gates. The time interval $\triangle t$ is stored in the time gate (T[m]) and later transferred to the cell to compute the cell state. Like the Phased-LSTM, time gate controls the input vector for any m-th sequence. Fig[2.11] shows the architecture of Time-LSTM 1.

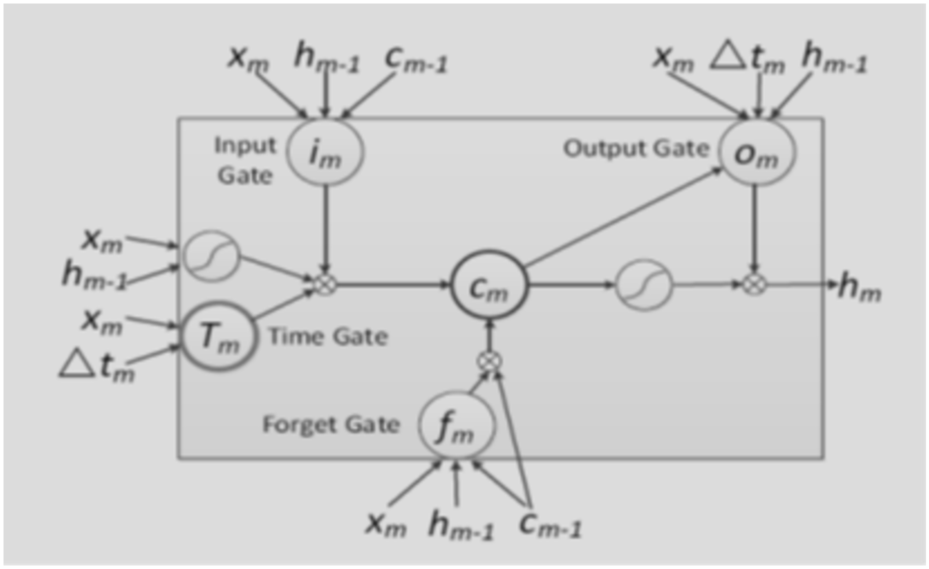


Figure 2.11: Architecture of Time-LSTM 1

Time-LSTM 2 The second type of Time-LSTM architecture has two different time gates. One time gate exploits time intervals to capture the short-term time value, and another is used to save the time intervals to model long-term time values. The first-time gate for Time-LSTM 2 for m-th sequence in the time series can be described as Eq. (2.34). T1 [m] deals with only short-term time values.

$$T1[m] = \sigma_1\left(x[m]Wx1 + \sigma_{\Delta t}\left(\Delta t[m]W_{t1}\right) + b_1\right), \text{ s.t } W_{t1} \leq 0 \tag{2.34}$$

Where the second gate can be described as follows Eq. (2.35).

$$T2[m] = \sigma_2\left(x[m]Wx2 + \sigma_{\Delta t}\left(\Delta t[m]W_{t2}\right) + b_2\right) \tag{2.35}$$

Time-LSTM 2 also modifies the actual cell state, output gate and the hidden state equations (2.21, 2.22 and 2.23) of the cell as follows in Eq. (2.36, 2.37, 2.38 and 2.39).

$$\begin{aligned}\widetilde{c_m} =& f_m \odot c_{m-1} \\ &+ i_m \odot T1_m \odot \sigma_c \left(x_m W_{xc} + h_{m-1} W_{hc} + b_c\right)\end{aligned} \tag{2.36}$$

$$\begin{aligned}c_m =& f_m \odot c_{m-1} \\ &+ i_m \odot T2_m \odot \sigma_c \left(x_m W_{xc} + h_{m-1} W_{hc} + b_c\right)\end{aligned} \tag{2.37}$$

$$\begin{aligned}o_m =& \sigma_o \left(x_m W_{xo}\right. \\ &\left. + \triangle t_m W_{to} + h_{m-1} W_{ho} + w_{co} \odot \widetilde{c_m} + b_o\right)\end{aligned} \tag{2.38}$$

$$\begin{aligned}o_m =& \sigma_o \left(x_m W_{xo}\right. \\ &\left. + \triangle t_m W_{to} + h_{m-1} W_{ho} + w_{co} \odot \widetilde{c_m} + b_o\right)\end{aligned} \tag{2.39}$$

$$h_m = o_m \odot \sigma_h \left(\widetilde{c_m}\right) \tag{2.40}$$

The new cell state $\widetilde{c_m}$ is later aggregated with output state to compute hidden state. Time-LSTM 2 can distinguish the impact of current recommendation and future recommendation, therefore, it can give a better prediction for rare event prediction and where the interval between consecutive event can be longer. Along with the input gate i[m], the time gate $T1\,[m[$ is another filter for $\sigma_c(x\,[m]\,W_{xc}{+}h\,[m{-}1]\,W_{hc}{+}b_c)$.

The intermediate cell state $c\,[m[$ store the result which is later passed to output gate $o\,[t[$ to generate the hidden state $h\,[m[$ which is used for short term time sequence prediction.

On the other hand, $T2\,[m[$ stores the time interval $\delta t\,[m[$ and later pass it to cell state $c\,[m[$ as the current cell state for computing the next cell state in a recurrent neural network, $\Delta t\,[m[$ is transferred to $c\,[m+1[$, $c\,[m+[$ and so on. Therefore, $T2\,[m[$ influences the long-term time series prediction.

Therefore, if $\Delta t\,[m[$ is smaller, according to Eq. (2.36), $T1\,[m[$ would be larger and $T1\,[m[$, the cell state has a larger influence on short-term time intervals. On the other hand, if $\Delta t\,[m[$ is a larger input vector $x\,[m[$, the influence and correspondingly would be smaller. Fig [2.12] shows the architecture of Time-LSTM 2.

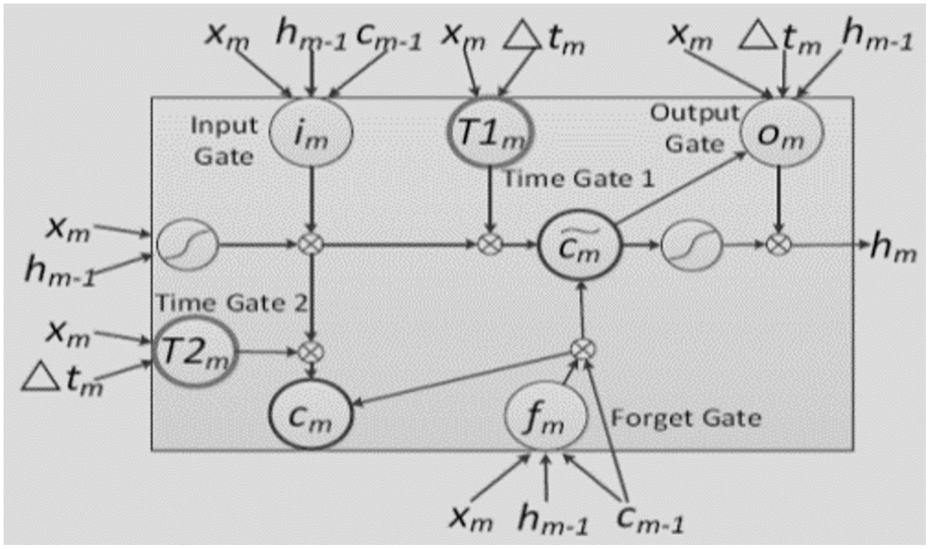


Figure 2.12: Architecture of Time-LSTM 2

Time-LSTM 3 The third type of Time-LSTM architecture has two different time gates. One time gate exploits time intervals to capture the short-term time value, and another is used to save the time intervals to model long-term time values. The first-time gate for Time-LSTM 2 for m-th sequence in the time series can be described as Eq. (2.41 and 2.42). T1 [m] deals with only short-term time values.

$$\begin{aligned}\widetilde{c_m} &= (1 - i_m \odot T1_m) \odot c_{m-1} \\ &\quad + i_m \odot T1_m \odot \sigma_c\left(x_m W_{xc} + h_{m-1} W_{hc} + b_c\right)\end{aligned} \tag{2.41}$$

$$
\begin{aligned}
c_m &= (1 - i_m) \odot c_{m-1} \\
&\quad + i_m \odot T2_m \odot \sigma_c \left(x_m W_{xc} + h_{m-1} W_{hc} + b_c \right)
\end{aligned}
\tag{2.42}
$$

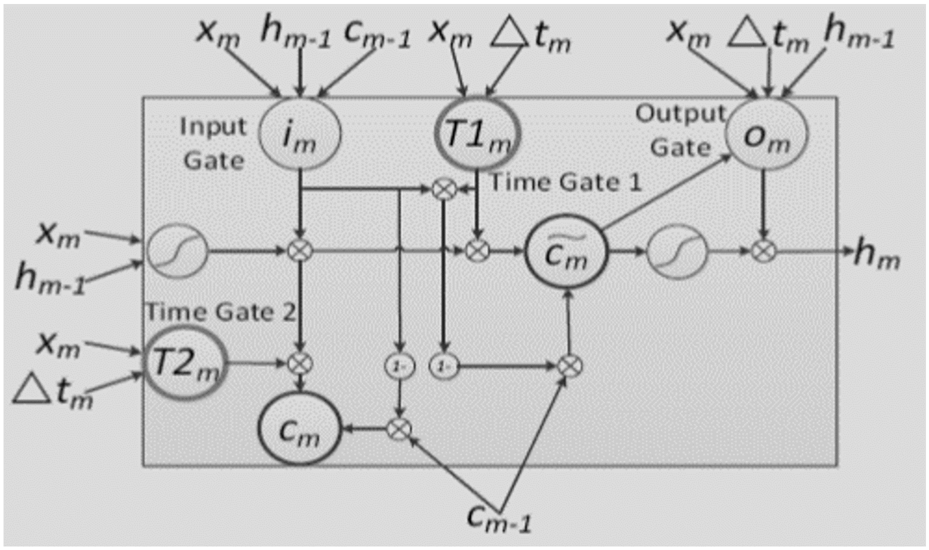


Figure 2.13: Architecture of Time-LSTM 3

FCN-rLSTM FCN-rLSTM. [10] proposed a spatial-temporal neural network by combining FCN [127] and LSTM [22] to overcome the limitation of continuous sequence due to co-relation between the sequence. This model has used a residual learning framework.

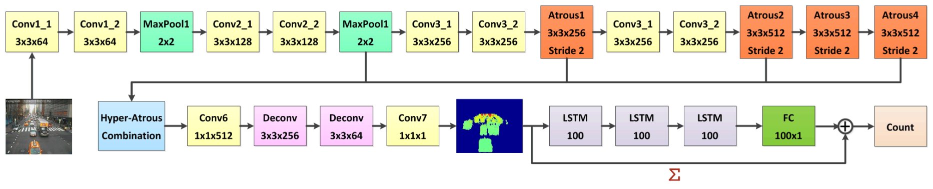


Figure 2.14: Network architecture and parameters of FCN-rLSTM

Grated Recurrent Unit (GRU)

Grated Recurrent Unit (GRU) is the other recurrent neural network [128]. Fig 2.15 explains the architecture of the GRU cell.

The GRU cell usually has two gates reset gate (r) and update gate (z). Fig [2.15] explains the gate performance in a GRU cell.

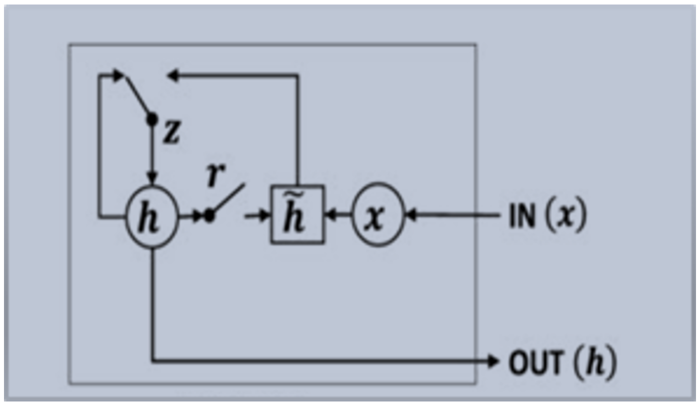


Figure 2.15: Reset and update gate of GRU

There are several variations of GRU, as shown in Table2.15. The total number of the parameter (N) for different GRU cells differs from one to another, and it is dependent on the dimension of the hidden state, n, and m, the dimension of the input vector. For example, the reset gate (r_t) decides how much data needs to be forgotten, whereas the update gate (z_t) determines the amount of data to be passed to the future cell.

Table 2.15: Different type of GRU cell

	Gates		
Cell	Reset gate	Update gate	Number of Parameters
GRU	$r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r)$	$z_t = \sigma(W_2 x_t + U_2 h_{t-1} + b_2)$	$N = 3\left(n^2 + nm + n\right)$
GRU-1	$r_t = \sigma(U_r h_{t-1} + b_r)$	Gru-1	$N = 2 \ \times nm$
GRU-2	$r_t = \sigma(U_r h_{t-1})$	$z_t = \sigma(U_2 h_{t-1})$	$N = 2(n \times (m+n))$
GRU-3	$r_t = o(b_r)$	$z_t = 0(b_l)$	
GRU-D	$r_t = \sigma\left(W_r \hat{x}_t + U_r \hat{h}_{t-1} + V_r m_t + b_r\right)$	$z_t = \sigma\left(W_z \hat{x}_t + U_z \hat{h}_{t-1} + V_z m_t + b_z\right)$	

GRU1 considers only the previous hidden state and bias to determine the output, while GRU2 uses the previous hidden state to compute the output, and GRU3 uses only bias to compute the cell state. GRU1, GRU2 and GRU3 use fewer parameters than traditional GRU.

GRU-D [24] is a variant of GRU which is mainly focused on solving the informative missingness problem of recurrent neural networks in general. This model uses a decay mechanism to decay the input over time towards the empirical mean. So, it does not use the value of the observation (x_t) as it is. Therefore, instead of x_t, it uses the vector defined by Eq. (2.41). Here, $\hat{x}_t^d$ is the last observation of the d-th variable, and x^d is the empirical mean of the d-th variable.

$$\hat{x}_t^d = m_t^d x_t^d + (1 - m_t^d)(\gamma_{x_t}^d x_t^d + (1 - \gamma_{x_t}^d) x^d) \tag{2.43}$$

Another decay mechanism used in this mode to understand the missingness. Eq. (2.44) defines the hidden state decay.

$$\hat{h}_{t-1} = \gamma_{t-1} \odot h_{t-1} \tag{2.44}$$

The equivalent equation for Eq. (2.41) and (2.42) for GRU-D can be defined using Eq. (2.45a) and (2.45b).

$$\tilde{h}_t = \tanh\left(W_h \hat{x}_t + U_h \left(r_t \odot \hat{h}_{t-1}\right) + V m_t + b_r\right) \tag{2.45a}$$

$$h_t = (1 - z_t) \odot \hat{h}_{t-1} + z_t \odot \hat{h}_t \tag{2.45b}$$

In Table 2.15 the equation for GRU-D has some new parameters, e.g., V_r, V_z and V. The masking vector m_t is fed into the model. This model succeeds to have a better result in terms of informative missingness with the help of different interpolation and imputation methods, e.g., Mean, Forward, Simple, K- nearest neighbor [129], CubicSpline [130], matrix factorization (MF)[131], principal component analysis (PCA) model [132], missforest [133], softimpute [134] and multiple imputations by chained equations (MICE)[135].

The main limitation is that it may fail or provide inefficient results in the absence of informative missingness or if the relationship between the missing patterns and the prediction tasks is not clear enough for the model. Therefore, this model is not suitable for unsupervised learning. The model's significant contribution is to use trainable decay to correlate the missing pattern and the task to determine a better prediction result. In terms of time and space complexity, this model is the same as RNN.

2.6.3 Convolutional Neural Network (CNN)

Convolutional neural networks are widely used for image processing. However, several recently proposed using CNN for modelling continuous time series and related problems such as anomaly detection and structured time series classification. Usually, for time series problems, 1D CNN architecture is widely used. Another popular approach is combining LSTM and other RNN models and designing a hybrid-CNN model for time series processing. Most of the related works focus on modifying the existing traditional design of CNN to make it adjustable to work with the unique behaviour of continuous time series. Some other works focus on combining the different valuable features from both CNN and RNN and developing the architecture of the hybrid neural network. Table 2.16 describes an overall trend in CNN models for time series problems

A convolutional neural network (CNN) is mainly used to classify and augment problem categories. In most cases, CNN is used in a hybrid model. CNN mainly suffers over-fitting less than other fully connected networks. Still, some specific time series augmentation techniques are used to improve the performance. One of the most used data augmentation techniques for the CNN model is Window Slicing (WS)[49] to slice up the time series into several slices and train those slices in batches, and each slice from a test time series is classified using the learned classifier. another data augmentation often used for CNN based time series classification algorithms is window wrapping (WW)[49]

Table 2.16: CNN models and associated time series applications

CNN Model	Design Type	Application	Performance
TCN [136]	Fully convolutional neural network (FCN)	Sequence Modelling	Demonstrate better performance than vanilla RNN
CNN-LSTM [77]	Hybrid	Prediction	
CNN-RNN [32]	Hybrid	Anomaly Detection /Pattern Recognition	Works for multi-variate time series
CNN [64]	CNN	Classification	
CNN-FCM [137]	Hybrid (CNN+ Fuzzy Cognitive Block)	Prediction	Shows promising performance for anomaly detection during training for handling data deviation

As the non-stationary and dynamic data of any time series evolves, most deep learning models suffer from uncertainty during training. One of the most challenging problems is maintaining the stability of the neural networks. Different CNN based models mentioned in table 2.16 shows that often hybrid approaches, where CNN can contribute to sequence modelling, data processing or classification phase with the help of other artificial intelligence techniques. Therefore, hybrid models are getting attention from different research areas for the time series problem domain. For example, combining LSTM and CNN is also a popular hybrid model for time series classification, anomaly detection, pattern recognition, and prediction problems.

This section discusses some popular CNN models for time series problem domains.

Temporal Convolutional Neural Network (TCN)

Most recent deep learning models for the time series are mainly based on RNN architecture (LSTM or GRU). However, convolutional neural network architecture also contributes to the learning time sequence for deep learning. TCN[136] outperform standard recurrent architectures across a broad range of sequence modelling tasks.

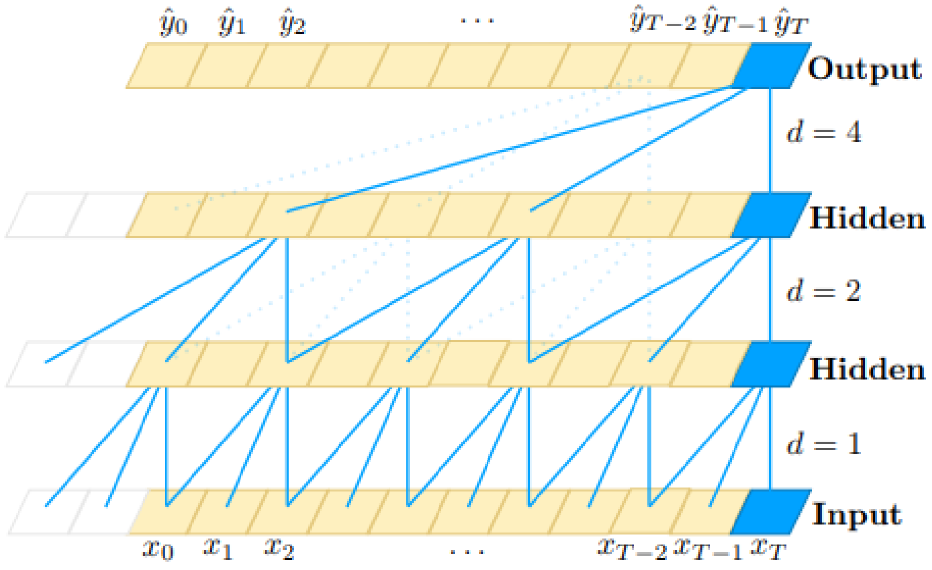

$\hat{y}_0$ $\hat{y}_1$ $\hat{y}_2$ ... $\hat{y}_{T-2}$ $\hat{y}_{T-1}$ $\hat{y}_T$
Output
$d = 4$
Hidden
$d = 2$
Hidden
$d = 1$
Input
x_0 x_1 x_2 ... x_{T-2} x_{T-1} x_T

Figure 2.16: Architectural element in TCN [136]

TCN is based on one fully convolutional neural network (FCN) where each hidden layer and the corresponding input layer are of the same length. TCN uses causal convolutions, where the elements from time t and earlier in the previous layer convolved the output at time t. Fig.[2.16] shows the architectural element of the TCN architecture. The input sequence $x_0 \ldots\ldots x_T$, and corresponding outputs $y_0 \ldots\ldots yT$ at each time. d is the dilation factor, k is the filter size. The main advantages of TCN over standard RNN are as follows

- Any long-time sequence can be processed simultaneously, unlike RNN which needs sequential processing.
- Authors claimed that TCN is better than RNN in memory size management.

Filter size and dilation factor provide better control over memory usage. This work has also shown that RNNs likely to use up to a multiplicative factor more memory than TCNs.

- TCN battles against the vanishing gradient problem by imposing a back-propagation path different from the temporal direction of the sequence.
- TCN also supports arbitrary input length.

Instead of all the advantages mentioned above of TCN over standard RNN, there is some limitation of TCN: it requires comparatively more memory during the evaluation period. Besides, TCN performs very poorly in the absence of many parameters regarding the time series. The primary concern of TCN is to minimise the memory size, reduce time by imposing parallel processing and overcome the vanishing gradient problem.

Spiking Neuron Neural Network(SNNN)

Michael et al. [138] proposed a relatively new variation of convolutional neural network SNNN, which has a straightforward mechanism for ignoring data points. Instead of complex computation, additional gates like RNN or phase parameters like Phased-LSTM [20], SNNN only computes whenever sudden bursts of activity are recorded and create more spikes, but when the recorded information is too little, it does not compute.

SNNNs are a suitable candidate for modelling Spatio-temporal event-based time sequences. This model only computes data when there is a high spike of neurons, and it is very power efficient. Furthermore, this model ignores irrelevant data when the spike is too low. Therefore for real-time data analysis, the model can learn data in a more data-efficient training than other traditional neural networks model. Unlike DNN and DCN, it does not wait for the entire input sequence to be finished to start

Table 2.17: Different type of hybrid model proposed for time series modelling

Proposed Work	**Used Neural Network Model**
Shi et al. [139]	Convolutional LSTM
Shaojie et a. l [136]	TCN
Bradbury el al. [140]	Quasi-RN

the learning phase. However, as a relatively new model, there is no benchmark data to measure efficiency and accuracy vastly. Moreover, designing training algorithms for this model is also very difficult. Therefore, this model is not a suitable candidate for all sorts of problems. However, this model can achieve results faster in the case of classification problems.

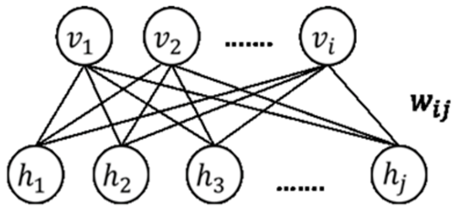
2.6.4 Hybrid Deep learning model

The recent trend in deep learning research is to find an optimised way to design a model architecture that can combine all useful attributes of RNN and CNN. Table [2.17] shows some works in this field.

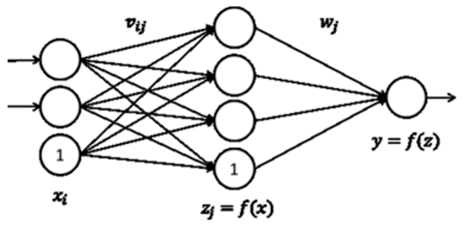
Deep Belief Neural Network(DBN)

T. Hirata et al. [141] have proposed a novel time series prediction model using ARIMA and DBN. There are some other works [75, 142] based on DBN. DBN only takes care of the non-linear feature of time series. Therefore, it is not a suitable model for linear time series features. The fundamental component of DBN is Restricted Boltzmann machine (RBM) [56] and Multi Later perceptron (MLP) [75]. DBM may also consist of only multiple RBM as well as [26]. RBM is one of the well-known mechanisms to reduce the number of dimensions. Fig 2.6.4 shows the architecture of RBM, which consists of one visible layer (v) and one hidden layer (h). Each visible unit is connected with the asymmetric weight matrix (w$_{ij}$). One important feature of RBM is that the connection between units is bidirectional. Each unit outputs either 1 or 0.

Another core component of DBN is an MLP. MLP usually has three layers like other



(a) Architecture of RBM



(b) Architecture of MLP system

neural networks, such as input, hidden, and output layers, as shown in Fig [2.6.4]. The activation function for MLP is a logistic sigmoid as shown in Eq. (2.46) . If $w_i x_i > b_i$. the neuron of the hidden layer gets fired, where x_i is the input, w_i is the weight, and the threshold is the biases (b_i).

$$f(x) = \frac{1}{1 + \exp\left(-\frac{x}{\epsilon}\right)} \tag{2.46}$$

Here ϵ is the gradient. Fig [2.17] describes the architecture of DBN, where the inputs are feed into an RBM and output is learned by an MLP.

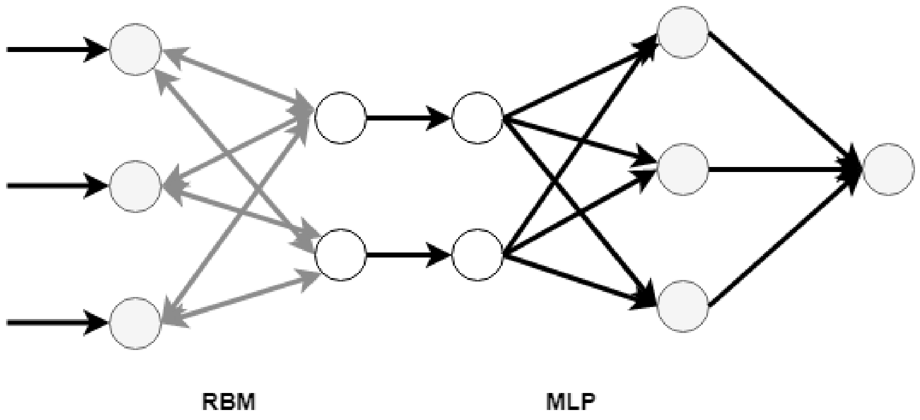


Figure 2.17: Architecture of DBN

The time series prediction work proposed in [108] re-write the time series as the following equation Eq. (2.47).

$$y\left(t\right) = L\left(t\right) + N(t) \tag{2.47}$$

Here L is the linear part of the time series, where N is the non-linear feature of y(t). In [108], an ARIMA has been used to determine the prediction value from the L(t), and a DBN predicted the prediction value from N(t). Therefore, the output time series from [108] can be described using Eq. (2.48). Here $\hat{L}\left(t\right)$ is the predicted result from ARIMA and the $\hat{N}\left(t\right)$ is the predicted result of DBN.

$$\hat{y}\left(t\right) = \hat{L}\left(t\right) + \hat{N}(t) \tag{2.48}$$

This kind of hybrid neural network shows the exact result for chaotic time series.

Cen et al. [26] have proposed another type of DBN which is a combination of DBN and a non-linear kernel-based parallel evolutionary support vector machine (ESVM). This model extracted different features from different relevant parameters of the time series using DBN and combined them with the original attribute of the series; this combined attribute vector is imported to a non-linear SVM for training and solving classification or prediction problems. A parallel PSO algorithm is used to optimise SVMs. Different optimisation function has been used to overcome the curse of dimensionality caused by the high dimensionality of ω, the weight vector.

Deep-Sense Model

This neural network model is a perfect example of a hybrid model where the nature of recurrent and convolutional models have been used together for better performance in the case of solving the sequential temporal problem. This model has three different layers, the CNN layer, the RNN layer and the output layer. Fig 2.18 shows the architecture of this model.

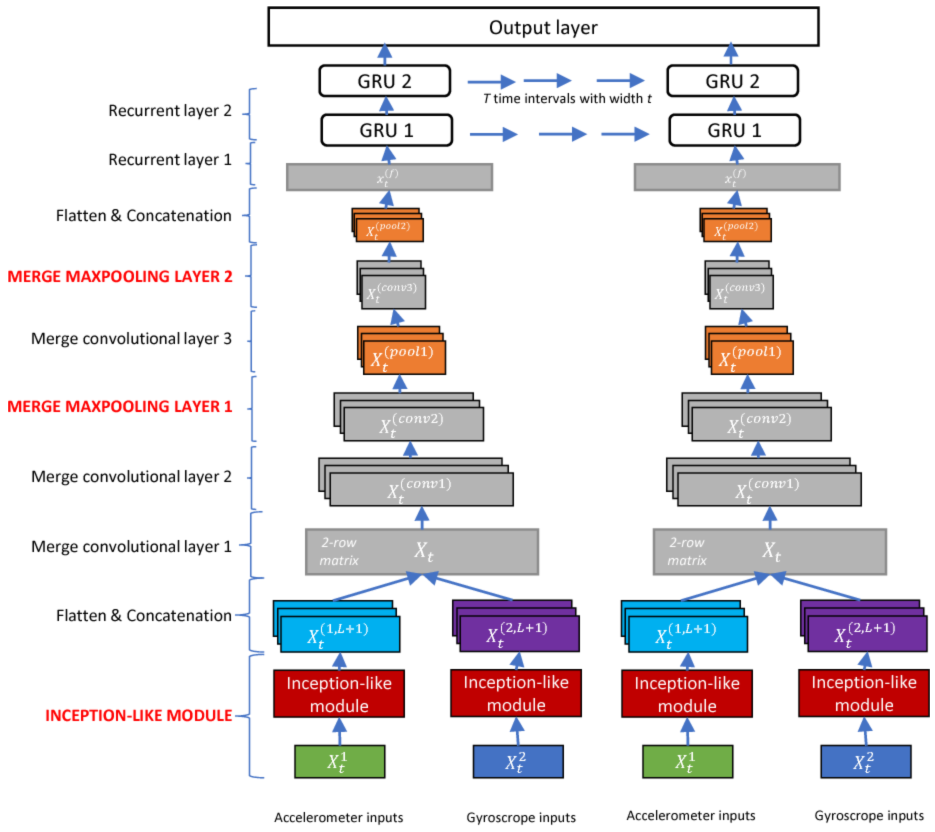


Figure 2.18: Architecture of Deepsense - a hybrid neural network model [26]

This model successfully uses the strong feature of both of the general model CNN and RNN and provides an architecture where different design components can be added or removed as required.

Semi-Supervised Sequence Learning

B. Ballinger et al. [68] have recently proposed a neural network model architecture to process multi-channel input data through semi-supervised learning. The proposed architecture is a combination of the convolutional 1D neural network [143], and a bidirectional LSTM [144]. Fig 2.19 shows that the proposed model takes multi-channel as well as multi-timescale input from the various data source and pass them through three layers of temporal convolutional 1 D layer, where all the required features are efficiently extracted and model temporal translation invariance [136].

Once the features are extracted, the model is first trained with a purely-supervised four bi-directional layer of LSTM without pre-training. Then this supervised LSTM is initialised with weight from the heuristic pre-training. The author has demonstrated that the LSTM without pre-training performs significantly better than the LSTM with pre-training for some data. On the other hand, the insights of some data discovered by the heuristic pre-training influence positively the corresponding output. This work contributes to processing multi-channel input data using semi-supervised learning.

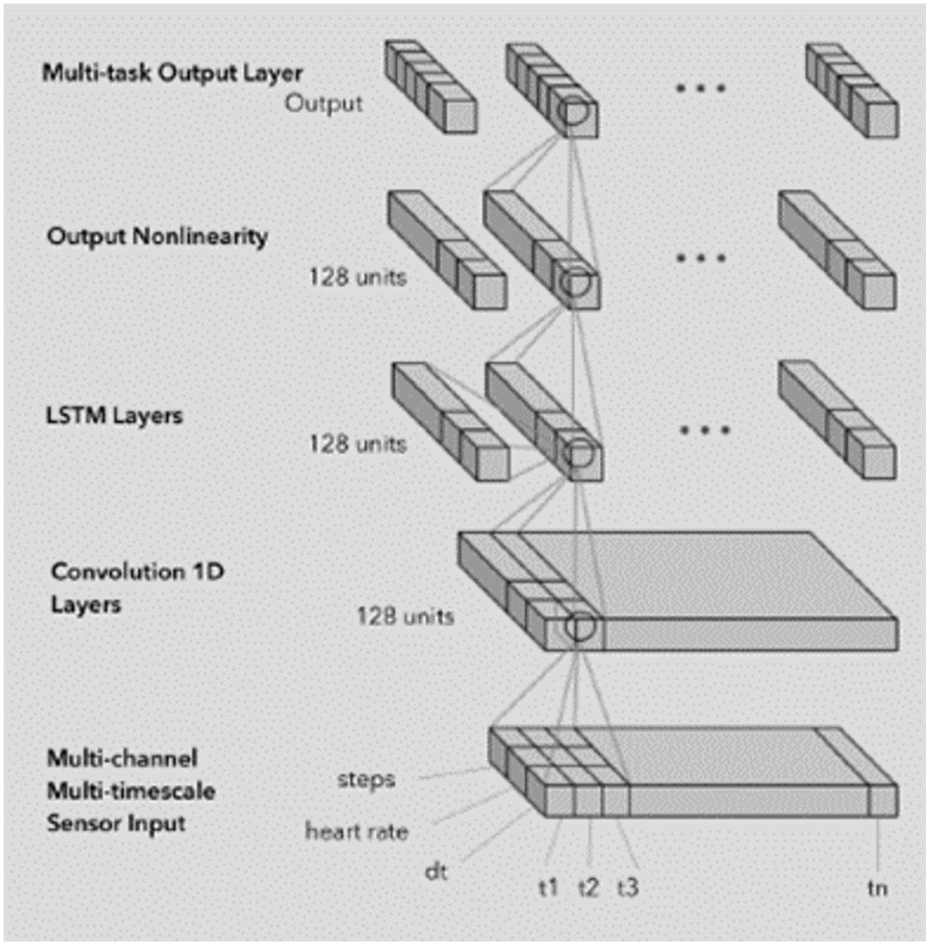


Figure 2.19: Semi-supervised deep learning model for multi-channel input

Graph based Neural Network model

[67] proposed a graph based augmentation for continuous-time data with relation between multiple variables of the data. In this graphical augmentation method, a patent record D consists of U clinician and v patients. Therefore each record in a continuous sequence of individual patient's visit (S), each visit record (D) is represented as a graph, $G = \{U, V, \mathcal{E}\}$, where $\mathcal{E} = \left\{ \{w_{ij}\}_{i=1}^{M} \right\}_{j=1}^{N}$, represents the set of edges connecting U and V; w_{ij} is the count of patient j visiting clinician i.

Runge-Kutta Neural Network

The core component of Runge-Kutta Neural Network [145] is higher-order Runge-Kutta methods. Higher-order RK methods are utilized to build network models with higher accuracy. Generally, RK methods are widely-used procedures to solve ODEs in numerical analysis. RK methods can also be used for image data processing. If a system x can be described as ODE as Eq. 2.49 with initial condition $x(0) = x_0$ and the boundary value for t is [0,T]. This ODE represents the rate of change of the system states. The rate of change is a function of time and the current system state

$$x(t) = f(x(t)) \tag{2.49}$$

2.6.5 Physics Informed Neural Network (PINN)

As defined in [27, 146, 147] Physics Informed Neural Networks are models for supervised learning that follow any given law of physics described by general non-linear partial differential equations (PDE). PINN has successfully established the foundation for a new paradigm to model time series using physics informed mathematical model which is fully differentiable using partial differential equations. This model is suitable for both discrete as well as a continuous-time series. PINN uses a one-dimensional non-linear Schrodinger equation [148] to represent a continuous-time model and uses a large number of data points to enforce physics informed constraints. This large amount of data points can introduce a severe bottleneck in processing higher dimensional continuous time series. This work mainly focuses on data-driven solutions and data-driven discovery using partial differential equations. The general form of PINN model is f(t,x) as defined by Eq. (2.50)

$$f := u_t + N\left[t\right] \tag{2.50}$$

Here u_t is the hidden latent and $N[.]$ is a non-linear differential operator. The chain rule of differentiating compositions of functions using automatic differentiation [149] is used to derive this neural network. This novel neural network model has successfully demonstrated the success of using automatic differentiation to differentiate the network concerning corresponding input and model parameters and finally obtain a novel physics informed neural network f. The authors have improved their work to the next level by defining a deeply hidden physics model as described in Eq. (2.51)

$$f{:=}u_t{-}N(t, x, u_x,\ u_{xx},\ \ldots\ldots\ldots) \tag{2.51}$$

In Eq. (2.51) u is the derivative of neural network work to time t and space x, which has been used in the processes described in their earlier work which is simply applying the chain rule of differentiating compositions of functions using automatic differentiation.

This work is a novel addition to the research field of continuous-time modelling using the neural network and automatic differentiation. Although, there are several questions open, such as

- Is it possible to design the architectural components of LSTM neural network in partial differential equations and obtain an efficient model that is fully differentiable for a continuous-time sequence as input?

- What are the potential relation between neural network and classical physics?
- How to resolve the over-fitting and vanishing gradients problems?

PINNs are valid for an inverse problem where data is unstructured, noisy and multivariate. Furthermore, PINN models are suitable for continuous data where the state can be at any time with an irregular time interval. However, this neural network does not deal with the variable parametrised model. Therefore, it requires further collaborative work to improve the performance of data-driven dynamics and infer the parametrised partial differential equations. Still, the model is subject to the high complexity associated with neural networks for input with a higher dimension. Furthermore, this neural network model only supports fixed model parameters; therefore, this network is not efficient enough in the case of the dynamic model parameters.

A generative adversarial networks (GAN) implementation [150] of PINN encoded the governing physical laws of the dataset in the form of stochastic differential equations (SDEs) using automatic differentiation into the architecture of GANs. This model uses three generators; two are the feed-forward deep neural network, and SDE induces the third one to capture the inherent stochasticity and uncertain extrinsic dynamics of the system. This model, PI-GAN, is the best fit for the dynamic system where the underlying physics is partially known, but with the help of some measurement, e.g. initial as well as boundary conditions, it is possible to determine missing pattern and parameters in the PDE as well as the system. To avoid the instability of GAN, this model uses weight-clipped WGANs. Similar to DNN, PI-GAN trains multiple generators and descriptors iteratively with the loss function for the optimiser. PI-GAN is suitable for forwarding, inverse and mixed problems, but the computational cost of training is much higher than training a single feed-forward neural network or PINN [27]. Both generator and discriminator exhibit over-fitting. The structure of the model is relatively complex, with multiple generators and descriptors with multiple levels of depth and width.

The main focus of the PINN-based model is to find a data-driven solution for PDEs. One of the significant limitations of PINN is the vanishing gradient issue, and it fails to represent the high-frequency sampling rate of the data. PINN also struggles to represent complex functions; therefore, it uses additional continuity constraints as a global regulariser. Distributed PINN (DPINN) [151] model provides a solution to these challenges by using a local regulariser instead of a global regulariser, which helps DPINN to be comparatively more data-efficient. This model divides the computational domain into uniformly distributed non-overlapping cells and installs a PINN in each cell with a local regulariser for initial and boundary conditions. each PINN in the corresponding cell compute loss, which is later aggerated to compute the total loss for the gradient descent algorithm of the optimiser.

One of the hybrid PINN models [152] where Convolutional Neural Networks(CNN) Furthermore, conditional Generative Adversarial Neural Networks(cGANs) are used to extract the hidden dynamics of the data simulated by PDE e.g. Shallow Water Equation (SWE). Due to the hybrid nature, this model provides real-time flood predictions with comparatively higher precision. The main challenges of this model are that it can ensure the stability of SWE dynamics. Another winning point of this model is that it can overcome the limitation of cGAN where a trivial missing point in the continuous-time series can result in a noticeable divergence of the prediction result.

Table 2.18 shows several application of PINN with real-time data.

Table 2.18: Different type of PINN

PINN	**Applications**	**Characteristics**
[153]	Wind Turbine Main Bearing Fatigue Life Estimation	
PI-GAN [150]	Solving stochastic differential equation	GAN with multiple generators and discriminators
[69]	Non-invasive 4D flow modelling for MRI data	Solve conservation laws in graph topologies
(DPINN) [151]	Solve Advection algorithm	Divide the computational domain and deploy PINN for each each division
fPINN [154]	solve space-time fractional advection-diffusion equations (fractional ADEs)	
[152]	Flood Prediction	CNN and cGANs
nPINNs [155]	High-dimensional parameter spaces.	Flexible and minimal implementation

2.6.6 Differential Equation Neural Network (DiffNN)

The main goal of deep learning is to find an unknown function (f) that can correctly predict the value in future time for any given observations based on their past values. Generally, this function f concatenates the hidden layer and non-linear functions. This section reviews the recent trend in neural networks architecture based on both Ordinary Differential equations (ODE) and Partial Differential Equations (PDE). Different challenges and strengths are identified for existing limitations in this recent family of Differential Equations based neural networks. One of the significant challenges of the neural network model is that they process a continuous time series as a discrete-time sequence with a fixed sampling rate and fixed sampling frequency [20]. This characteristic makes real-time continuous data processing difficult. It also imposes a high computation load and memory usage. Besides, due to the dependency on the computation of previous states for some neural networks, such as RNN models, they are prone to vulnerabilities such as if the time gap between two consecutive observations is too big, it can affect the efficiency of the model adversely. Therefore, the neural network performs better for discrete-time series of moderate length with a fixed sampling rate, few missing values, and short time intervals between observations. However, the neural network model needs different additional techniques when it comes to continuous real-time series with multiple variables and irregular sampling rates.

A differential equation can learn the underlying hidden dynamics of continuous-time data [156] in real-time. [26, 87] demonstrate the strength of Neural Ordinary Differential Equations (Neural-ODEs). Neural-ODEs mainly re-factored the design of the Residual Neural Network (ResNet).

Residual neural network

ResNets can be re-designed as a dynamic system []. The architecture of a ResNet can be described using Euler Integration, as shown in Algorithm 1. First, take the initial step of the vector, and then on every step, compute the update and add that with the hidden vector. This is similar to Euler integration.

Algorithm 1 ResNet as Euler Integration

```
1: procedure RESNET($z, \theta$)
2:     while $t < T$ do
3:         $z = z + f(z, t, \theta)$
4:     $returnz$
```

Here f is a ResNet as defined in Algorithm 2. Simply feed the current depth to the ResNet block and use the fixed set of parameters for the entire block. That refers the hidden dynamics can change between layer continuously by ODEs.

Algorithm 2 ResNet Function

```
1: procedure F($z, t, \theta$)
2:     $returnresnet(z, \theta[t])$
```

The while loop in Algorithm 1 can be replaced by an ODE solver. Algorithm 1 is same as Algorithm 3. As a result, [157, 158] considers pre-active ResNets as forward Euler method.

Ordinary Differential Equation as Neural Network

Residual Neural Network is discritization of Euler method. In the residual network, the state of the layer $i+1$ is related to the state of the layer i as shown in Eq. (2.52);

$$y_{i+1} = y_i + f(y_i, \theta_i) \tag{2.52}$$

However, we obtain a residual network if we consider the Euler method for solving ODE with step size $h = 1$. So ResNet is a special case of the Euler method. Briefly, the Euler method is a first-order numerical method for solving an ordinary differential equation with the initial condition introduced by Leonhard Euler. Differential equations can model the dynamics of continuous systems that are changed by time since time is not countable in physics, and it is a real number. Therefore, it is not feasible to use a differential equation for the systems' dynamics. Nevertheless, discretise a continuous dynamical system for modelling time series for the computer. So if $x(t)$ is the state of a system at time t, $\dot{x}(t)$ refers to the derivative of x for time, i.e., rate of change of x to time. For example, if $x(t)$ is a population of an entity at time t then $\dot{x}(t)$ is the rate of change in population at time t. Differential equations are the relations of the rate of changes with the original state, i.e.,

$$\dot{x}(t) = f(x(t))$$

Solving a differential equation means finding the function $x(t)$ that satisfies the differential equation with the initial condition(the starting point of the computation). So if the state of the system at the starting time is x_0, i.e., $x(t_0) = x_0$, where t_0 is the starting time of the computation, then the initial condition for solving an ordinary differential equation(ODE) are as follows,

$$\begin{aligned}\dot{\mathbf{x}}(t) &= \mathbf{f}(\mathbf{x}(t)) \\ \mathbf{x}(t_0) &= \mathbf{x_0}\end{aligned}$$

The solution of 2.6.6 is given by the following formula,

$$\begin{aligned}&\mathbf{x}(t) = \mathbf{x}(t_0) + \int_{t_0}^{t} \mathbf{f}(x(s))ds \\ &\forall t \in (a, b)\end{aligned} \tag{2.53}$$

where (a, b) is the domain of the definition of solution and $t_0 \in (a, b)$. Euler used the first-order approximation of the function f by Taylor's formula, so:

$$x(t + h) = x(t) + h\dot{x}(t) + o(h^2)$$

where $o(h^2)$ is the term containing h^k with $k \geq 2$ such that $\lim_{h\to 0} \frac{o(h^2)}{h^2} = 0$. If we replace $\dot{x}(t)$ with $f(x(t))$ in this formula and ignore $o(h^2)$ term we have Euler formula for solving ODEs, i.e.,

$$\begin{aligned}&x(t + h) = x(t) + hf(x(t)) \\ &x(t_0) = x_0\end{aligned}$$

If the parameter of the system θ change with time, i.e., $\theta : \mathbb{R} \to \mathbb{R}$, we have Neural Ordinary Differential Equation with parameter θ as in Eq. (2.54):

$$\begin{aligned}x(t + h) &= x(t) + hf(x(t), \theta(t)) \\ x(t_0) &= x_0\end{aligned} \tag{2.54}$$

If $h = 1$ it is a ResNet. Suppose the loss function is a real-valued function of the state, i.e., $L : \mathcal{C} \to \mathbb{R}$, where $\mathcal{C}$ is the class of functions $x : \mathbb{R} \to \mathbb{R}^N$ which are the solutions of ODE of the system. The goal is to minimize $L()$ with respect to the controls parameter $\theta(t)$. Let us consider a system of differential equations consisting of n differential equations with initial conditions is given as follows:

$$\begin{aligned} &\dot{x}_i(t) = f_i(x_1(t), \ldots, x_n(t), t, \theta(t)), i = 1, \ldots, n \\ &x_1(t_0) = \tilde{x}_0, \ldots, x_n(t_0) = \tilde{x}_n \end{aligned} \tag{2.55}$$

A specific choice of control function $\theta(t)$ yield a solution of (2.55), $x_i(t)$ for $t_0 \leq t \leq t_1$, where we assume $\theta : \mathbb{R} \to \mathbb{R}$ is a piecewise continuous function. We call this solution a trajectory. The control function θ is chosen in such a way that at time t_1, a set of equations holds as like Eq. (2.56):

$$\Psi_i(x_1(t_1), \ldots, x_n(t_1), t_1) = 0, \;\; i = 1, \ldots, p \leq n \tag{2.56}$$

This set of equations is called terminal conditions. The problem of Mayer is to find the optimal control function $\theta(t)$ such that minimize the function:

$$\Phi(x_1(t), \ldots, x_n(t), t) \tag{2.57}$$

At the terminal time t_1 such that (2.56) holds. This function is called criterion or objective function. Note that 2.57 is first calculated at t_1 and then minimization applied with respect to θ such that (2.56) holds. The set of all point $(x_1, \ldots, x_n, t)$ that satisfies in (2.56) is called terminal manifold. The control function θ that yield trajectories reach to the terminal manifold is called admissible. The goal is to find an admissible control function θ such that minimize the objective function which known as optimal control function (Φ). The corresponding trajectories is called optimal trajectories. Let $(x_1, \ldots, x_n, t)$ be a point in terminal manifold such that minimum value of Φ attained if the admissible trajectory satisfy the initial condition $(x_1, \ldots, x_n)$

at time t. Let L be a function of $(x_1, \ldots, x_n, t)$ with this property, i.e.,

$$L(x_1, \ldots, x_n, t) = \text{minumum value of } \Phi \text{ with the above conditions} \tag{2.58}$$

so the optimal control also depend on $(x_1, \ldots, x_n, t)$, i.e., if θ^* is the optimal control value then,

$$\theta^*(t) = u(x_1, \ldots, x_n, t)$$

For some objective value function u, the optimal policy function.

If we expand $x_i(t)$ to the first order Taylor expansion in the neighbourhood of t and put this value in L and use optimality of L we have,

$$\begin{aligned} x_i(t + \Delta t) &= x_i(t) + \dot{x}_i(t)\Delta t + o(\Delta t^2) \\ &= x_i(t) + f_i(x_1(t), \ldots, x_n(t), t, \theta^*(t))\Delta t + o(\Delta t^2) \end{aligned} \tag{2.59}$$

Now by calculating L on $x_i(t + \Delta t)$ and use (2.59) we have,

$$\begin{aligned} & L(x_1(t + \Delta t), \ldots, x_n(t + \Delta t), t + \Delta t) \\ & = L(x_1(t) + f_1(\mathbf{x}(t), t, \theta^*(t))\Delta t + o(\Delta t^2), \ldots, x_n(t) + f_n(\mathbf{x}(t), t, \theta^*(t))\Delta t + o(\Delta t^2), t + \Delta t) \\ & \geq L(x_1(t), \ldots, x_n(t), t) \end{aligned} \tag{2.60}$$

where $\mathbf{x}(t) = (x_1(t), \ldots, x_n(t))$. The last inequality holds because $(x_1(t), \ldots, x_n(t), t)$ is the optimum trajectory in the interval $[t, t + \Delta t]$. Thus,

$$L(x_1(t), \ldots, x_n(t), t) = \min_{\theta}[L(x_1(t)+f_1\Delta t+o(\Delta t^2), \ldots, x_n(t)+f_n\Delta t+o(\Delta t^2), t+\Delta t)] \tag{2.61}$$

Now if we apply first-order Taylor formula to L we obtain,

$$\begin{aligned} &L(x_1(t) + f_1\Delta t + o(\Delta t^2), \ldots, x_n(t) + f_n\Delta t + o(\Delta t^2), t + \Delta t) \\ &= L(x_1(t), \ldots, x_n(t), t) + \sum_{i=1}^{n} \frac{\partial L}{\partial x_i} f_i \Delta t + \frac{\partial L}{\partial t}\Delta t + o(\Delta t^2) \end{aligned} \tag{2.62}$$

if we remove $L(x_1(t), \ldots, x_n(t), t)$ from both hand side of (2.61) we have,

$$0 = \min_{\theta}[\sum_{i=1}^{n} \frac{\partial L}{\partial x_i} f_i \Delta t + \frac{\partial L}{\partial t}\Delta t + o(\Delta t^2)] \tag{2.63}$$

by deviding both hand side of 2.63 and letting $\Delta t \to 0$ we have,[159],

$$0 = \min_{u} \left[\sum_{i=1}^{n} L_{x_i} f_i + L_t \right] \tag{2.64}$$

where $L_{x_i} = \frac{\partial L}{\partial x_i}$ and $L_t = \frac{\partial L}{\partial t}$. If $u^* = \operatorname{argmin}_u \left[\sum_{i=1}^{n} L_{x_i} f_i + L_t\right]$ then,

$$\sum_{i=1}^{n} L_{x_i} f_i(x_1, \ldots, x_n, t, u^*) + L_t = 0 \tag{2.65}$$

since L is not explicitly depend on u, then $\frac{\partial L}{\partial u} = 0$. Our goal is to obtain an ODE to describe the dynamic of L_{x_j} with respect to time. The first step is to compute $(\frac{dL_{x_j}}{dt})_u$ which subscribe u means that L is computed with respect to the control u. By the chain rule and the fact that L is a function of $(x_1, \ldots, x_n, t)$ and each x_i is a function of t we have,

$$\begin{aligned} (\frac{dL_{x_j}}{dt})_u &= \sum_{i=1}^{n} \frac{\partial L_{x_j}}{\partial x_i}\frac{dx_i}{dt} + \frac{\partial L_{x_j}}{\partial t} \\ &= \sum_{i=1}^{n} \frac{\partial L_{x_j}}{\partial x_i} f_i + \frac{\partial L_{x_j}}{\partial t} \end{aligned} \tag{2.66}$$

The partial derivative in Eq. (2.65) with respect to x_j can be described as Eq. (2.67),

$$\begin{aligned}\sum_{i=1}^{n}(\frac{\partial L_{x_i}}{\partial x_j}f_i + L_{x_i}\frac{\partial f_i}{\partial x_j}) + \frac{\partial L_t}{\partial x_j} &= \sum_{i=1}^{n}\frac{\partial L_{x_i}}{\partial x_j}f_i + \sum_{i=1}^{n}L_{x_i}\frac{\partial f_i}{\partial x_j} + \frac{\partial L_t}{\partial x_j}\\ &= \sum_{i=1}^{n}\frac{\partial L_{x_i}}{\partial x_j}f_i + \frac{\partial L_t}{\partial x_j} + \sum_{i=1}^{n}L_{x_i}(\frac{\partial f_i}{\partial x_j} + \frac{\partial f_i}{\partial u}\frac{\partial u}{\partial x_j})\\ &= \sum_{i=1}^{n}\frac{\partial L_{x_i}}{\partial x_j}f_i + \frac{\partial L_t}{\partial x_j} + \sum_{i=1}^{n}L_{x_i}\frac{\partial f_i}{\partial x_j}\\ &= 0\end{aligned} \tag{2.67}$$

the last equality holds because of 2.72. If we combine Eq. (2.66) and Eq. (2.67) we derive the following differential equation,[1]

$$(\frac{dL_{x_j}}{dt})_u = -\sum_{i=1}^{n}L_{x_i}\frac{\partial f_i}{\partial x_j} \tag{2.68}$$

Let $a_i(t) = \frac{\partial L}{\partial x_i}$ then Eq. 2.68 is the adjoin equation for Eq. (2.69),

$$(\frac{da_j}{dt})_u = -\sum_{i=1}^{n}a_i(t)\frac{\partial f_i}{\partial x_j} \tag{2.69}$$

In the matrix notation, if $\mathbf{a}(t) = [a_1(t), \dots, a_n(t)]^\top$ and $\mathbf{f}(\mathbf{x}(t), t, \theta) = [f_1, \dots, f_n]^\top$ then Eq. (2.69) can also be described as Eq. (2.70),

$$(\frac{da_j(t)}{dt})_u = -\mathbf{a}(t)^\top\frac{\partial \mathbf{f}(\mathbf{x}(t), t, \theta)}{\partial x_j} \tag{2.70}$$

If we know the value of $a_j(t)$ at time t_1 then we could compute the value of $a_j(t)$ at time t_0, where $t_0 \leq t_1$, by,

$$a_j(t_0) = a_j(t_1) - \int_{t_1}^{t_0}\mathbf{a}(t)^\top\frac{\partial \mathbf{f}(\mathbf{x}(t), t, \theta)}{\partial x_j}dt \tag{2.71}$$

[1]If L and its partial derivative are continuous then $\frac{\partial^2 L}{\partial x_i \partial t} = \frac{\partial^2 L}{\partial t \partial x_i}$ so $\frac{\partial L_{x_j}}{\partial t} = \frac{\partial L_t}{\partial x_j}$

but to solve this integral, we need to know the value of $\mathbf{x}(t)$ at the entier time interval $[t_0, t_1]$ but since we know the initial condition of $\mathbf{x}(t_0)$ we could obtain the trajectory of $\mathbf{x}(t)$ for all $t \in [t_0, t_1]$.

Now, we can obtain the sensitivity of L with respect to the control parameter u. If we compute partial derivative of (2.65) with respect to u, the control parameter, then we have,

$$\sum_{i=1}^{n} L_{x_i}(\frac{\partial f_i}{\partial u})_{u^*} + \frac{\partial L_t}{\partial u} = 0 \tag{2.72}$$

or equivalently,

$$\sum_{i=1}^{n} a_i(t)(\frac{\partial f_i}{\partial u})_{u^*} + \frac{\partial L_t}{\partial u} = 0 \tag{2.73}$$

since $a(t) = L_{x_i}(x_1(t), \ldots, x_n(t), t, u)$, but we know $\frac{\partial L_t}{\partial u} = \frac{dL_u}{dt}$ so,

$$\frac{\partial L_u}{dt} = -\sum_{i=1}^{n} a_i(t)(\frac{\partial f_i}{\partial u})_{u^*} \tag{2.74}$$

the solution of this equation at time t_0 is obtaine by taking integral from t_1 to t_0 backward in time, because we know the value of L_u at time t_1, i.e.,

$$\frac{\partial L}{\partial u} = (\frac{\partial L}{\partial u})|_{t=t_1} - \int_{t_1}^{t_0} \sum_{i=1}^{n} a_i(t)(\frac{\partial f_i}{\partial u})_{u^*} \tag{2.75}$$

but $(\frac{\partial L}{\partial u})|_{t=t_1} = 0$ and in matrix notation we have,

$$\frac{\partial L}{\partial u} = -\int_{t_1}^{t_0} \mathbf{a(t)}^\top \frac{\partial \mathbf{f}(\mathbf{x}(t), t, u(t))}{\partial u} dt \tag{2.76}$$

The value of $\frac{\partial \mathbf{f}(\mathbf{x}(t),t,u(t))}{\partial u}$ and the Jacobian $\frac{\partial \mathbf{f}(\mathbf{x}(t),t,u(t))}{\partial \mathbf{x}}$ can be computed by automatic differentiation [160]. If we consider the ODE solver as the black box then the initial value problem,

$$\begin{aligned}
&\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), t, u(t)) \\
&\mathbf{x}(t_0) = x_0 \\
&t \in [t_0, t_1]
\end{aligned} \tag{2.77}$$

then this equation can be solve by calling an ODE solver $\texttt{ODESOLVER}(\mathbf{x}(t_0), \mathbf{f}, t_0, t_1, u)$ which get initial condition $\mathbf{x}(t_0)$, dynamic of the problem $\mathbf{f}$, start time t_0, end time of t_1 and control parameter u as input and produce $\mathbf{x}(t_1)$ as the output. So the reverse mode derivative of an ODE can be computed by Algorithm 3.

Algorithm 3 Reverse-mode derivative for ODE

```
1: procedure REVERSE-ODE($u, t_0, t_1, \mathbf{x}(t_1), \mathbf{a}(t_1)$)
2:     $\mathbf{a}(t_0) = \texttt{ODESOLVER}(\mathbf{a}(t_1), -\mathbf{a}(t)\frac{\partial \mathbf{f}}{\partial \mathbf{x}}, t_1, t_0, u)$
3:     $\frac{\partial L}{\partial u}|_{t_0} = \texttt{ODESOLVER}(0, -\mathbf{a}(t)\frac{\partial f}{\partial u}, t_1, t_0, u)$
4:     Return $(\mathbf{a}(t_0), \frac{\partial L}{\partial u}|_{t_0})$
```

As an example, consider the following ODE as the dynamic of the problem,

$$\begin{aligned}
&\frac{dx(t)}{dt} = -kx \\
&x(0) = 1
\end{aligned} \tag{2.78}$$

where the control parameter k is constant. By calling the `odeint` from `scipy` we could obtain $x(1)$[2]. Then the dynamic of the lose function is as follow,

$$\begin{aligned}
&\frac{da(t)}{dt} = ka(t) \\
&a(1) = x(1)
\end{aligned} \tag{2.79}$$

i.e., we start from $x(1)$ backward in time until $t = 0$. Note that by definition $a(t_i) = \frac{\partial L}{\partial x(t_i)}$. We use `grad` from `autograde`(Python) to obtain $\frac{\partial f(x,t,k)}{\partial x}$ where $f(x, t, k) = -kx(t)$. By another call to theODEsolver we obtain $a(t)$ in $[1, 0]$[3].The optimal control

[2]In fact by calling theODEsolver we obtain $x(t)$ in all $t \in [0, 1]$

[3]since $a(t)$ is calculated backward we write the time interval in the opposit order

policy is a value of k such that $L_k(t)$ is minimum at $t = 0$. But the dynamic of L_k is as follow,

$$\frac{dL_k(t)}{dt} = a(t)x(t) \\ L_k(1) = 0 \tag{2.80}$$

where $a(t)$ and $x(t)$ are the solutions of 2.78 and 2.79, respectively. In the Figure 2.20 we plot the graph of $x(t), a(t)$ and $L_k(t)$ for $k = 0.9$ and $k = 2$ in the intgerval $[0, 1]$.[4]

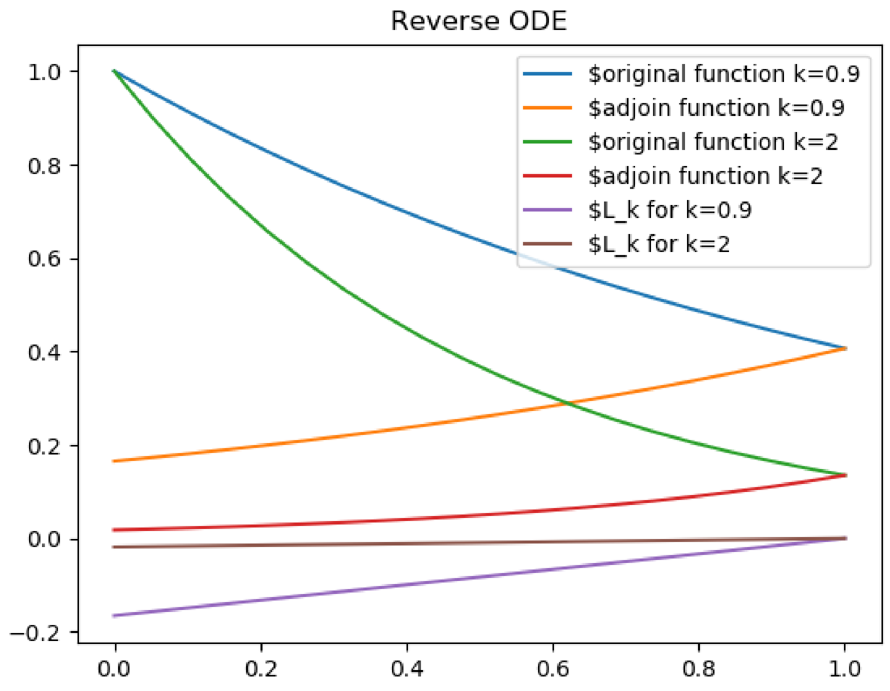


Figure 2.20: Reverse ODE withODEsolver from scipy of Python

Continuous Normalising Flows and Neural ODE

Now suppose we have a evolution of a random variable. Let $\mathbf{x}_0 \in \mathbb{R}^N$ be a N-dimentional random variable and suppose it continously transform to another random

[4]Note that the whole process is done automatically by obtaining dynamic of adjoin and dynamic of control function by automatic differentiation and solving with the ODE solver

variable $\mathbf{x}_T \in \mathbb{R}^N$ at time T via the following dynamic,

$$\begin{aligned}\frac{d\mathbf{x}}{dt} &= \mathbf{f}(\mathbf{x}(t), t)\\ \mathbf{x}(t_0) &= \mathbf{x}_0 \\ \mathbf{x}_0 &\sim \mathbf{p}_{\mathbf{x}_0}\end{aligned} \tag{2.81}$$

where in the last line we mean the probability density function of the random variable $\mathbf{x}_0$ is $\mathbf{p}_{\mathbf{x}_0}$. The probability density function of a continous random variable $\mathbf{x} \in \mathbb{R}^N$ is a real valued function $p : \mathbb{R}^n \to \mathbb{R}$ such that,[161] ,

1. $p(\mathbf{z}) \geq 0$ for all $\mathbf{z} \in \mathbf{R}^N$
2. $\int_{\mathbb{R}^N} p(\mathbf{z})d\mathbf{z} = 1$

To obtain the dynamic for $\mathbf{p}_{\mathbf{x}(t)}$ we need to undrestand the nature of transformation in one step. Suppose $\mathbf{x} \in \mathbb{R}^N$ is a random variable with probability density function $\mathbf{p}_{\mathbf{x}}$. Suppose $\mathbf{x}$ change continously to random variable $\mathbf{y} \in \mathbb{R}^N$ via the smooth function $\mathbf{f} : \mathbb{R}^N \to \mathbb{R}^N$, i.e., $\mathbf{y} = \mathbf{f}(\mathbf{x})$. For simplicity suppose $\mathbf{J}(\mathbf{f}(\mathbf{x}))$ is invertible for every value of $\mathbf{x} \in \mathbb{R}^N$. To obtain probability density function of $\mathbf{y}$ we use change of variable formula [162] we have,

$$\begin{aligned}\int_{\mathbb{R}^N} \mathbf{p}_{\mathbf{x}}(x)dx &= \int_{\mathbb{R}^N} \mathbf{p}_{\mathbf{x}}(x)|\det(\frac{\partial x}{\partial y})|dy\\ &= \int_{\mathbb{R}^N} \mathbf{p}_{\mathbf{x}}(x)|\det(\mathbf{J}(\mathbf{f}(x)))^{-1}|dy\\ &= \int_{\mathbb{R}^N} \mathbf{p}_{\mathbf{x}}(x)|\det(\mathbf{J}(\mathbf{f}(x)))|^{-1}dy\\ &= 1\end{aligned} \tag{2.82}$$

where $\mathbf{J}$ is the Jacobian[5] operator[6]. Since $\mathbf{p}_{\mathbf{x}}(x)|\det(\mathbf{J}(\mathbf{f}(x)))|^{-1} \geq 0$ and its integral

[5]If $x = g(y)$ then $\frac{\partial x}{\partial y}$ is another notation for the Jacobian $\mathbf{J}g(y)$

[6]By the Inverse Function Theorem if $\mathbf{Jf}(\mathbf{x}_0)$ is invertible, then the function f has an inverse in the neighborhood of $\mathbf{x}_0$ and $\mathbf{J}(\mathbf{f}^{-1}(\mathbf{y}_0)) = (\mathbf{J}(\mathbf{f}(\mathbf{x}_0)))^{-1}$ where $\mathbf{y}_0 = \mathbf{f}(\mathbf{x}_0)$,[162]

over $\mathbb{R}^N$ is equal to 1 it is probability density function of $\mathbf{y}$, i.e., $\mathbf{p}_\mathbf{y}(y) = \mathbf{p}_\mathbf{x}(x)|\det(\mathbf{J}(\mathbf{f}(x)))|^{-1}$, where $y = \mathbf{f}(x)$.

Now if we divide both side of this equation by $\mathbf{p}_\mathbf{x}$ and apply log function in both side we have,

$$\begin{aligned}\log(\frac{\mathbf{p}_\mathbf{y}(y)}{\mathbf{p}_\mathbf{x}(x)}) &= \log|\det(\mathbf{J}(\mathbf{f}(x)))|^{-1} \\ &= -\log|\det(\mathbf{J}(\mathbf{f}(x)))|\end{aligned} \tag{2.83}$$

hence,

$$\log(\mathbf{p}_\mathbf{y}(y)) = \log(\mathbf{p}_\mathbf{x}(x)) - \log|\det(\mathbf{J}(\mathbf{f}(x)))| \tag{2.84}$$

Now suppose $\mathbf{x}_0 \sim \mathbf{p}_{\mathbf{x}_0}$ and,

$$\mathbf{x}_i = \mathbf{f}_i(\mathbf{x}_{i-1}), i = 1, \dots, M \tag{2.85}$$

,i.e., $\mathbf{x}_M = \mathbf{f}_M \circ \cdots \circ \mathbf{f}_1(\mathbf{x}_0)$. We can compute the probability destribution of $\mathbf{p}_{\mathbf{x}_M}$ by applying equation 2.84 M times. So if we suppose $\mathbf{x}_i \sim \mathbf{p}_{\mathbf{x}_i}$ we have,

$$\begin{aligned}\log(\mathbf{p}_{\mathbf{x}_M}(x_M)) &= \log(\mathbf{p}_{\mathbf{x}_{M-1}}(x_{M-1}) - \log|\det(\mathbf{J}(\mathbf{f}_{M-1})(x_{M-1}))| \\ &= \log(\mathbf{p}_{\mathbf{x}_{M-2}}(x_{M-2}) - \log|\det(\mathbf{J}(\mathbf{f}_{M-1}(x_{M-2})))| - \log|\det(\mathbf{J}(\mathbf{f}_M(x_{M-1})))| \\ &\vdots \\ &= \log(\mathbf{p}_{\mathbf{x}_0}(x_0)) - \log|\det(\mathbf{J}(\mathbf{f}_1(x_0)))| - \cdots - \log|\det(\mathbf{J}(\mathbf{f}_M(x_{M-1})))| \\ &= \log(\mathbf{p}_{\mathbf{x}_0}(x_0)) - \sum_{i=1}^{M} \log|\det(\mathbf{J}(\mathbf{f}_i(x_{i-1})))|\end{aligned} \tag{2.86}$$

Now suppose $\mathbf{x}(t)$ change continuously by the following ODE,

$$\begin{aligned}\dot{\mathbf{x}}(t) &= \mathbf{f}(\mathbf{x}(t), t) \\ \mathbf{x}(t_0) &= \mathbf{x}_0 \sim \mathbf{p}_{\mathbf{x}_0}\end{aligned} \tag{2.87}$$

Now let $\mathbf{x}(t)$ transform continiously to $\mathbf{x}(t+\varepsilon)$ for small positive value of ε via smooth function $\mathbf{F}$, i.e., $\mathbf{x}(t+\varepsilon) = \mathbf{F}(\mathbf{x}(t))$. If we use first order Taylor expansion of $\mathbf{x}(t+\varepsilon)$ we have,

$$\begin{aligned} \mathbf{x}(t+\varepsilon) &= \mathbf{x}(t) + \varepsilon\dot{\mathbf{x}}(t) + o(\varepsilon^2) \\ &= \mathbf{x}(t) + \varepsilon\mathbf{f}(\mathbf{x}(t)) + o(\varepsilon^2) \end{aligned} \tag{2.88}$$

So $\mathbf{F}(\mathbf{x}(t)) = \mathbf{x}(t) + \varepsilon\mathbf{f}(\mathbf{x}(t)) + o(\varepsilon^2)$. On the other hand by equation 2.84 we have,

$$\log(\mathbf{p}_{\mathbf{x}(t+\varepsilon)}(x(t+\varepsilon)) - \log(\mathbf{p}_{\mathbf{x}(t)}(x(t)) = -\log|\det(\mathbf{J}(\mathbf{F}(x(t))))| \tag{2.89}$$

If we divide both hand side of this equation by ε and take limit while $\varepsilon \to 0$ we have,

$$\lim_{\varepsilon\to 0} \frac{\log(\mathbf{p}_{\mathbf{x}(t+\varepsilon)}(x(t+\varepsilon)) - \log(\mathbf{p}_{\mathbf{x}(t)}(x(t)))}{\varepsilon} = -\lim_{\varepsilon\to 0} \frac{\log|\det(\mathbf{J}(\mathbf{F}(x(t))))|}{\varepsilon} \tag{2.90}$$

The left hand side is the definition of $\frac{d\mathbf{p}_{\mathbf{x}(t)}(x(t))}{dt}$. We need to simplify the right hand side of the equation by unfolding $\mathbf{J}(\mathbf{F}(\mathbf{x}(t)))$,

$$\begin{aligned} \mathbf{J}(\mathbf{F}(x(t))) &= \frac{\partial \mathbf{F}(x(t))}{\partial x(t)} \\ &= \frac{\partial x(t)}{\partial x(t)} + \varepsilon\frac{\partial \mathbf{f}(x(t),t)}{\partial x(t)} + \frac{\partial}{\partial x(t)} o(\varepsilon^2) \\ &= I + \varepsilon\frac{\partial \mathbf{f}(x(t),t)}{\partial x(t)} + \frac{\partial}{\partial x(t)} o(\varepsilon^2) \end{aligned} \tag{2.91}$$

so in limit we have,

$$\lim_{\varepsilon\to 0} \frac{\log|\det(\mathbf{J}(\mathbf{F}(x(t))))|}{\varepsilon} = \lim_{\varepsilon\to 0} \frac{\log|\det(I + \varepsilon\frac{\partial \mathbf{f}(x(t),t)}{\partial x(t)}|)}{\varepsilon} \tag{2.92}$$

Before proceed, we need some results from linear algebra. If A is a $N \times N$ positive definite matrix then all of its eigenvalues are positive real numbers [163] and its characteristic polynomial is

$$\begin{aligned}\chi_A(\varepsilon) &= \det(\varepsilon I - A) \\ &= (\varepsilon_1 - \varepsilon)\dots(\varepsilon_N - \varepsilon)\end{aligned} \tag{2.93}$$

where $\varepsilon_1, \dots, \varepsilon_N$ are eigenvalues of A. By applying the log function[7] on both hand side of the charactristic polynomial and taking derivative with respect to ε we have,

$$\frac{\chi'_A(\varepsilon)}{\chi_A(\varepsilon)} = \frac{-1}{\varepsilon_1 - \varepsilon} + \cdots + \frac{-1}{\varepsilon_N - \varepsilon} \tag{2.94}$$

now if $\varepsilon \to 0$ we have,

$$\begin{aligned}\lim_{\varepsilon\to 0} \frac{\chi'_A(\varepsilon)}{\chi_A(\varepsilon)} &= \frac{-1}{\varepsilon_1} + \cdots + \frac{-1}{\varepsilon_N} \\ &= -\,\mathbf{tr}(A^{-1})\end{aligned} \tag{2.95}$$

we have $\det(I + \varepsilon A) = \det(A) \cdot \chi_{-A^{-1}}(\varepsilon)$ since,

$$\begin{aligned}\det(I + \varepsilon A) &= \det(A(A^{-1} + \varepsilon I)) \\ &= \det(A) \cdot \det(\varepsilon I - (-A^{-1})) \\ &= \det(A) \cdot \chi_{-A^{-1}}(\varepsilon)\end{aligned} \tag{2.96}$$

so if A is a positive definite matrix we have

$$\begin{aligned}\lim_{\varepsilon\to 0} \frac{\log|\det(I + \varepsilon A)|}{\varepsilon} &= \lim_{\varepsilon\to 0} \frac{\log|\det(A) \cdot \chi_{-A^{-1}}(\varepsilon)|}{\varepsilon} \\ &= \lim_{\varepsilon\to 0} \frac{\chi'_{-A^{-1}}(\varepsilon)}{\chi_{-A^{-1}}(\varepsilon)} \\ &= \mathbf{tr}(A)\end{aligned} \tag{2.97}$$

Here we used L'Hopital's rule since the first limit is in the indeterminate forms $\frac{0}{0}$.

[7] here we mean $\log_e$ which is ln function

Hence if $\frac{\partial \mathbf{f}(x(t),t)}{\partial x(t)}$ is positive definit at $x(t)$ then,

$$\lim_{\varepsilon \to 0} \frac{\log |\det(I + \varepsilon \frac{\partial \mathbf{f}(x(t),t)}{\partial x(t)})|}{\varepsilon} = \mathbf{tr}(\frac{\partial \mathbf{f}(x(t),t)}{\partial x(t)}) \tag{2.98}$$

and finaly,

$$\frac{d \log(\mathbf{p}_{\mathbf{x}(t)}(x(t))}{dt} = -\,\mathbf{tr}(\frac{\partial \mathbf{f}(x(t),t)}{\partial x(t)}) \tag{2.99}$$

by solving this ODE with respect to $\mathbf{p}_{\mathbf{x}(t)}(x(t))$ we obtain probability density function at each instant of time and we could compute density at the final time ,i.e., $\mathbf{p}_{\mathbf{x}(T)}$,[8].

In Figure 2.21 we show the evolution of the exponential distribution $p_{x(0)}(x) = e^{-x}$ for $x \geq 0$ under the dynamic $\frac{dx}{dt} = \frac{x(t)}{t-2}$. In Figure 2.22 we have shown the evolution of the exponential distribution under the dynamic $\frac{dx}{dt} = x(t)(1-x(t))$ and finaly in Figure 2.23 we show the evolution of the probability density function of a normal distributed random variable $x(0) \sim \mathcal{N}(2,1)$, i.e., $p_{x(0)}(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{(x-2)^2}{2}}$, under the dynamic $\frac{dx}{dt} = x(t)(1-x(t))$.

Time complexity comparsion For computing $p_{\mathbf{x}_M}(x_M)$ using equation (2.86) we need to compute M determinant of $N \times N$ Jacobian matrices $\mathbf{Jf}_i(x_{i-1})$, $i = 1, \dots M$. We could compute the partial derivative of the function $g : \mathbb{R}^N \to \mathbb{R}$ by automatic differentiation with cost $O(\mathbf{size}(\frac{\partial g}{\partial x_i})) = O(\mathbf{size}(g))$ where $\mathbf{size}(g)$ is the size of computational graph of g. But by Strassen algorithm we have an $O(N^3 \cdot \mathbf{size}(\mathbf{f}_i))$ algorithm to compute $\mathbf{Jf}_i(x_{i-1})$. So the time complexity for computing $p_{\mathbf{x}_M}(x_M)$ is,

$$O(N^3 \cdot (\mathbf{size}(\mathbf{f}_1) + \cdots + \mathbf{size}(\mathbf{f}_M)) \tag{2.100}$$

On the other hand, for computing continious normalizing flow we need to compute

[8] If $\mathbf{f}$ is Lipschits continious with respect to $\mathbf{x}$ then by Rademacher's theorem $\mathbf{f}$ is differentiable almost everywhere(the points where $\mathbf{f}$ is not differentiable is of measure zero). So we need a stronger condition for $\mathbf{f}$ to be sure that $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$ exists.

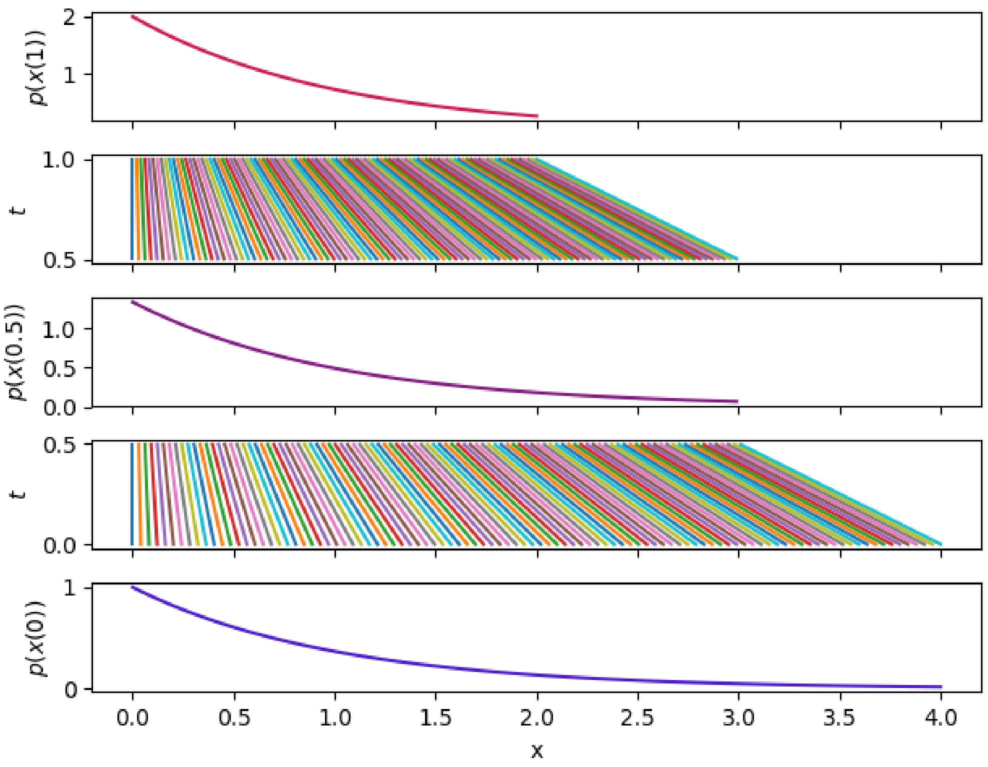


Figure 2.21: Evolution of $p_{x(0)}(x) = e^{-x}$ by $\frac{dx}{dt} = \frac{x}{t-2}$

$\mathbf{tr}(\frac{\partial \mathbf{f}(x(t),t)}{\partial x(t)})$ and a call to an ODE solver. We know that,

$$\mathbf{tr}(\frac{\partial \mathbf{f}(x(t),t)}{\partial x(t)}) = \frac{\partial \mathbf{f}_1(x(t),t)}{\partial x_1(t)} + \cdots + \frac{\partial \mathbf{f}_N(x(t),t)}{\partial x_N(t)} \tag{2.101}$$

then the time complexity for computing $\mathbf{tr}(\frac{\partial \mathbf{f}(x(t),t)}{\partial x(t)})$ is $O(N \cdot \mathbf{size}(f))$ which is less than standard normalizing flow with time complexity $O(N^3)$.

Table 2.19 shows several Neural-ODE and their characteristics.

Different applications are explored to understand the efficiency of Neura-ODE. Table 2.20

This new family of neural networks has some substantial improvement over the residual neural network as follows

- Ordinary Differential Equations require one independent variable and one

Table 2.19: Different type of Neural-ODE

Neural-ODE	**Characteristics**
ODE-RNN [26]	The ODE-RNN model is a combination of RNN and Neural ODE model. It demonstrates the capabilities of Neural ODE between consecutive observations, and similar to RNN, it also also updates observations at every step.
Latent-ODE [87]	Similar to ODE-RNN, Latent ODEs can also take care of irregular sampling rate and the intermediate gap between observations.
GRU-ODE-Bayes [164]	A continuous-time version of the GRU, built on Neural-ODE and a Bayesian update network that processes the sporadic observations of continuous-time series.
Ordinary Differential Equation Variational Auto-Encoder (ODE^{2}VAE) [28]	A latent second-order ODE model for high-dimensional sequential data. This model demonstrates probabilistic latent ODE dynamics, which are parameterized by deep Bayesian neural networks.

Table 2.20: Different applications of Neural-ODE

Neural-ODEs	**Application**	**Characteristics**
Continuous ODE-GRU-D and Extended ODE-GRU-D [93]	Multi-Variate time series with irregular sampling rate	Leverage *ODESolve* similar to Neural-ODEs and compute hidden as well as input decay rate based on the dynamics of the data
[165]	Detection of Heart Rate from Facial Videos	Train Neural-ODE using heart rate from original videos, and predict heart rates of deepfake videos using trained Neural-ODE.
[166]	Data Augmentation	Used the optimized parameters from a Neural ODE network to compute local gradient of the time series and later used for gereting perturbed adversarial samples.

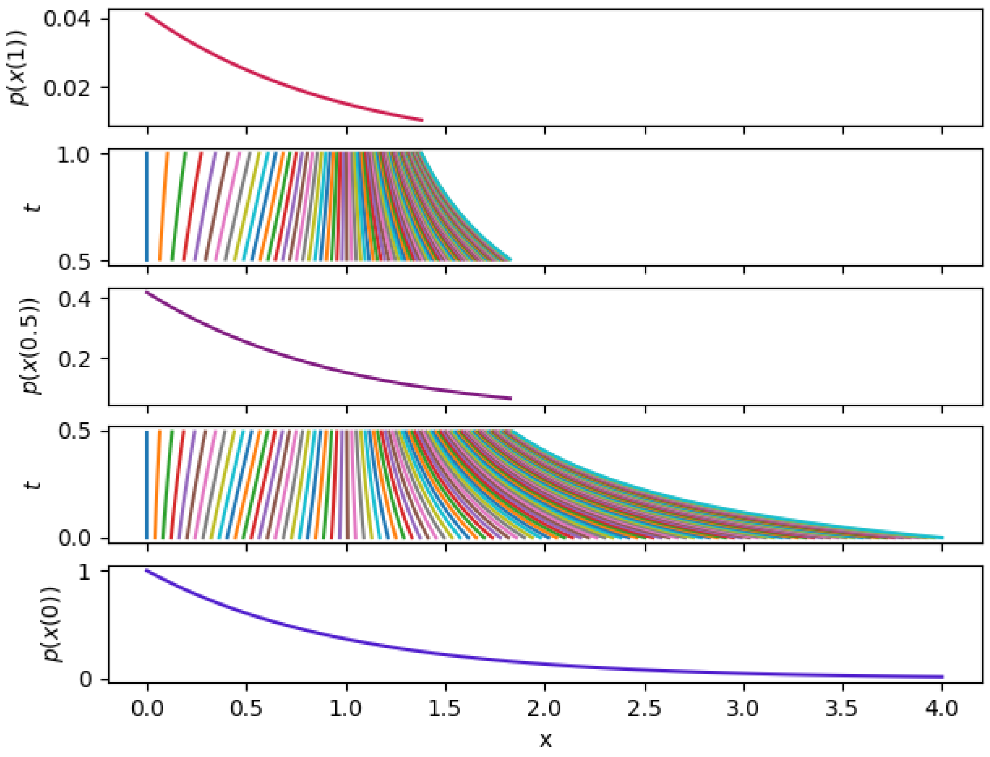


Figure 2.22: Evolution of $p_{x(0)} = e^{-x}$ by $\frac{dx}{dt} = x(1-x)$

derivative of an unknown function. An unknown ODE solver can compute the value of the independent variable at any time t with the desired accuracy.

- two different methods can be used for first-order differential equations, e.g., the Eulers method [167]and adjoint sensitive method.
- The construction of neural networks is much easier. it is no longer required to determine the number of layers, and any optimizer can be used for learning.
- It does not need to define the number of discrete layers. Instead, the network is a continuous function where the gradient can train itself within marginal error.
- Most common neural networks used for time series modelling are vulnerable to adversarial attacks. Therefore, high robustness is essential for time series data augmentation. [166] shows that Neural ODE can generate a local guided gradient which improves the robustness of time series data augmentation significantly.

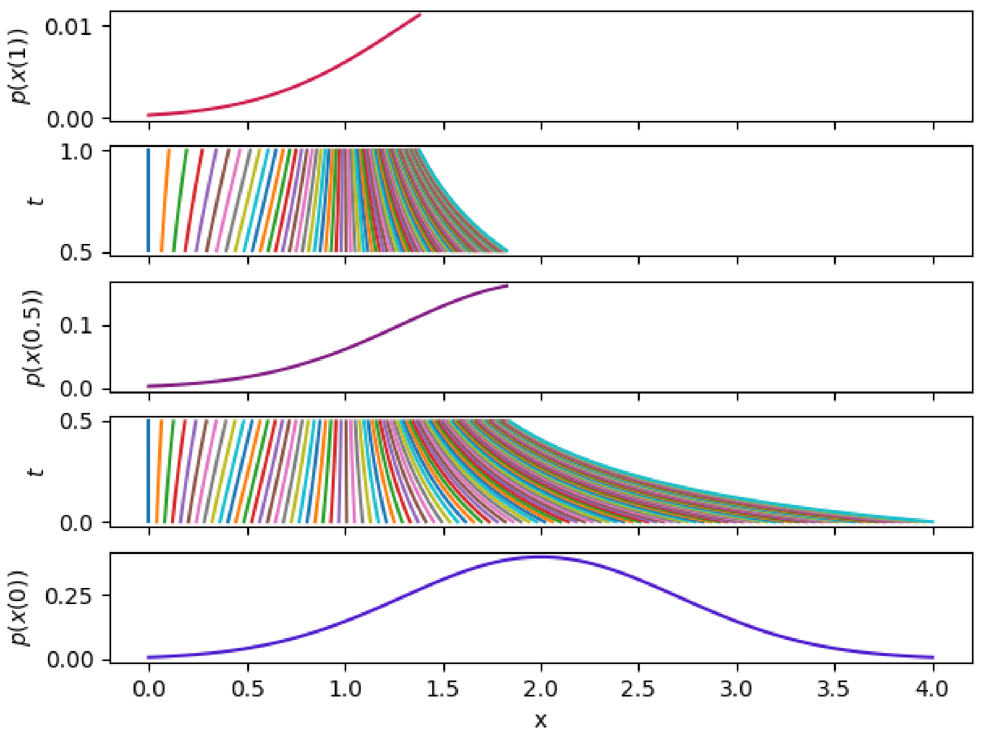


Figure 2.23: Evolution of $x(0) \sim \mathcal{N}(2, 1)$ by $\frac{dx}{dt} = x(1-x)$

- Finally, it would help design the neural network model as a function of time t for continuous-time learning problems.

However, Neural ODE based neural network still has some limitations to overcome as follows

Less Accuracy: The accuracy of Neural-ODE is comparatively low for long term prediction than short term prediction. Neuro-ODE is mainly a generative neural network. For parallel mode prediction, the past values in the series are usually not actual system output; instead, they are the predicted value of the network. Therefore, the accuracy can be reduced over time for long term prediction. The main reason for this limitation is that a Nero-ODE network learns the system's state over time instead of the changing rate of system states over the training period. This causes potential obstacles for the network to learn the long term behaviour of the system.

Training time: Neural-ODEs are usually slower than other neural networks; at the same time, it requires longer training time.

High error rate: The error rate is higher for Neural-ODE in comparison to the stranded neural network. One missing component of Neural-ODEs is various regularisation mechanisms, which are crucial for reducing generalisation error and improving the robustness of neural networks against adversarial attacks. The initial input for Neural SDE [72] is the output of a convolutional layer, and the value at time t is passed to a linear classifier after average pooling. The unique characteristics of neural SDE are that it adds randomness to the continuous neural networks using a Stochastic Differential Equation (SDE) framework.

Difficult architecture: The proper order of the neural identifier for identifying an ODE system is not easy to know.

Stabilization of the structure: ODE-GRU and ODE-LSTM [89], GRU-ODE-Bayes [164] models also contribute to the stabilization of the structure of the Neural-ODEs. Neural SDE [72] also uses stochastic noise injection in the SDE network to regularise the continuous ODEs,

Fixed time interval : The existing neural identifier can only predict the system behaviour well at a fixed time interval (using a fixed, regular sampling rate). This is not the nature of an ODE system. Although a high-order discretisation is more accurate than the first-order discretisation, the resulting ordinary differential equations of the former are usually complex and intractable.

Informative Missingness : Continuous ODE-GRU-D and Extended ODE-GRU-D [93] leverage ordinary differential equation solver (*ODESolve*) to compute the hidden dynamics of the model. These models use *ODESolve* to replace the missing value in continuous time series (T) by the derivative of the value of available observations of T. These models are evaluated against the multi-variate time series. Over time, the decay in hidden dynamics and input has a significant impact on the final output on multi-variate time series. Both models compute the decay rate as the derivatives of time (t). Therefore, the decay rate can control the gradient optimisation of the model over time. The continuous ODE-GRU-D model is used to generate the hidden dynamics of the GRU-D model [24] as continuous-time dynamics using *ODESolve*. The Extended ODE-GRU-D model computes the decay rate and continuous hidden dynamics of the GRU-D model and exhibits an efficient way to generate both hidden and input decay rates based on the data dynamics. Experiment on Physio net demonstrates that these models can successfully solve the informative missingness from continuous data.

High Dimension datasets : Neural-ODEs suffer from higher computation cost for time series with higher dimension. Unlike black-box ODEs, the ODE^{2}VAE model explicitly decomposes the latent space ($\dot{\mathbf{z}}_t$) into momentum ($\dot{\mathbf{s}}_t$) and position ($\dot{\mathbf{v}}_t$) components similar to variational auto-encoders (VAEs) and solves dynamical system governed by a second order Bayesian neural ODE model $\mathbf{f}_{\mathcal{W}}(\mathbf{s}_t, \mathbf{v}_t)$ as shown in Eq. (2.102). As first-order ODEs are incapable of modelling high-order dynamics, ODE^{2}VAE uses three different components; (i) a distribution for the initial position $p(s_0)$ and velocity $p(v_0)$ in the latent space , (ii) hidden dynamics defined by an acceleration field, and (iii) a decoding likelihood $p(xi|si)$, which is only computed by the velocity positions. .

$$\dot{\mathbf{z}}_t = (\dot{\mathbf{s}}_t, \dot{\mathbf{v}}_t) = \{ \begin{array}{ccc} \dot{\mathbf{s}}_t & = & \mathbf{v}_t \\ \dot{\mathbf{v}}_t & = & \mathbf{f}_{\mathcal{W}}(\mathbf{s}_t, \mathbf{v}_t) \end{array} , \quad \begin{bmatrix} \mathbf{s}_T \\ \mathbf{v}_T \end{bmatrix} = \begin{bmatrix} \mathbf{s}_0 \\ \mathbf{v}_0 \end{bmatrix} + \int_0^T \underbrace{\begin{bmatrix} \mathbf{v}_t \\ \mathbf{f}_{\mathcal{W}}(\mathbf{s}_t, \mathbf{v}_t) \end{bmatrix}}_{\vec{\mathbf{f}}_{w(\mathbf{z}_1)}} dt \tag{2.102}$$

Representation of continuous function : [168] demonstrate that even some straightforward function can not be represented using Neural-ODE as this model preserve the state of the input and follow continuous limit or boundary. This limitation often leads to expensive computation for Neural-ODE. To overcome such limitations, Augmented Neural ODEs (ANODEs)[168] add dimension to learn complex functions using simple flow. Therefore, instead of $\mathbb{R}^d$, ANODEs uses $\mathbb{R}^{d+p}$ to move points to an additional dimension and avoid intersection among trajectories. Eq (2.52) is formulated for ANODEs as Eq. (2.103).

$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{bmatrix}\mathrm{h}(t)\\ \mathrm{a}(t)\end{bmatrix}=\mathrm{f}\left(\begin{bmatrix}\mathrm{h}(t)\\ \mathrm{a}(t)\end{bmatrix},t\right),\quad\begin{bmatrix}\mathrm{h}(0)\\ \mathrm{a}(0)\end{bmatrix}=\begin{bmatrix}\mathrm{x}\\ 0\end{bmatrix}\tag{2.103}$$

Fig. 2.24 describes the differences between Neural-ODE and ANODEs for the function $g(x)$. ANODEs achieve better precision for learning the functions and are consistently faster than Neural-ODEs. As ANODEs always learn simpler flow, the computation cost is less than Neural-ODEs.

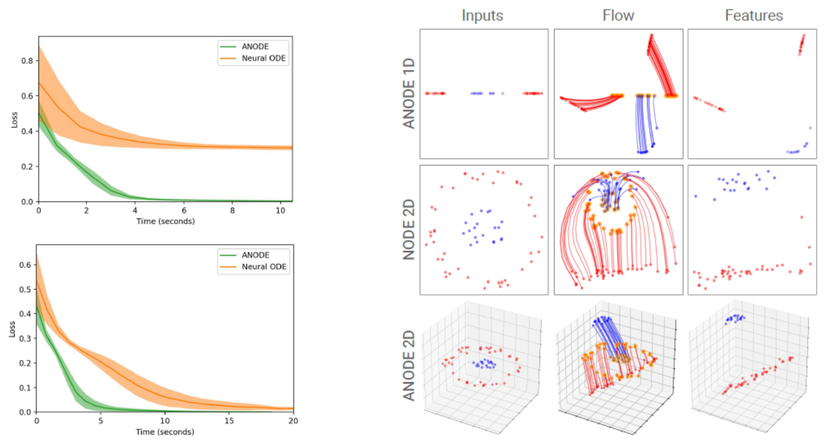


Figure 2.24: (Left) Loss plots for Neural-ODEs and ANODEs (Right) Flows learned by NODEs and ANODEs.

Partial Differential Equation as Neural Network

Partial Differential Equation (PDE) contributes significantly to the advancement of the neural network. Convolution neural networks (CNN) are highly appropriated in image processing and computer vision. The basic architecture of a CNN model can be represented through mathematical equations. Recent works have been proposed to pre-process the image dataset as a dynamical system of continuous-time series. Therefore, a series of ODEs can be used to interpret these systems. [169] shows the relation between PDE and CNN. Convolution neural networks (CNN) are highly appropriated in image processing and computer vision. For example, RKNN [145] as described in section 2.6.4 has some strong characteristics along with some limitations. In order to use RKNN for a system like image classification, it requires some adjustment. For example, each period is modelled by a time-dependent first order dynamical system. Besides, all coefficients and time step size (T) can be computed implicitly. The time step size (h) of RK methods, as shown in (1.5) needs to be predefined to control the truncation error. With these modifications, RKNet [170] is proposed.

$$X = x_1, x_2, ..., x_n \in \mathbb{R}^n \tag{2.104}$$

The main characteristics of RKNet are as follows

- In an RKNet, a period is composed of iterations of time steps.
- A particular RK method is adopted throughout the time steps in a period to approximate the increment in each step
- The increment in each step is broken down into increments in several stages according to the adopted RK method.
- A convolutional subnetwork approximates each stage due to the versatility of

neural networks on approximation.

For PDE based CNN, the neural network function f filters the input feature Y as shown in Eq. 2.106. Here f consists of layers (F) of liner transformations and pointwise nonlinearities.

The core layer of deep neural network architecture can be simplified as described as Eq. (2.105). Any deep neural network can be formalized as concatenation of many of the layers (F) given in Eq. (2.105). The linear operations K_2 and K_1 in Eq. (2.105). Each of these two operators is parametrized θ^3 and θ^1 respectively. N is the normalization layer parametrized with θ^2. A common choice for N in (1) is the batch normalization layer. θ^2 represents the scaling factors and biases. of a neural network For activation function any activation method can be used, e.g. σ.

$$F(\theta, Y) = K_2(\theta^3)\sigma(N(K_1(\theta^1)Y, \theta^2) \tag{2.105}$$

Most of the cases, the input Y is a time despondent data and can be represented as can be seen as a discretization of a continuous function $Y(x)$. For example, time series forecasting, weather prediction, image data speech and other applications [169] has modified Eq. (2.105). The linear operators $K_1 = K_1(\theta)\epsilon R^(wxw_{in})$ and $K_2 = K_2(\theta)\epsilon R^(w_{out}xw)$ are considered as convolution operators as shown in Eq. (2.106).

$$\mathbf{F}_{\text{sym}}(\boldsymbol{\theta}, \mathbf{Y}) = -\mathbf{K}(\boldsymbol{\theta})^\top \sigma(\mathcal{N}(\mathbf{K}(\boldsymbol{\theta})\mathbf{Y}, \boldsymbol{\theta})) \tag{2.106}$$

In this section, we discuss the Physics Informed Neural Network, which uses a combination of neural networks and partial differential equation(PDE) to obtain the hidden dynamic of the model. In this model, we should consider a penalty for the loss function of a neural network that deviates from the model's dynamic. In [27], authors introduced Physics Informed Neural Network to obtain not only the solution of the PDE but also obtain the parameters of the PDE for an optimization task. Assume we want to obtain the solution of a PDE by a neural network. Here there is no training set. By converting the PDE to an optimization problem, we can solve the PDE by a neural network using initial conditions and boundary value conditions and use the new solutions as the training set of the neural network. As the new value of approximation of the solution is obtained, a neural network uses them to learn and produce the new solution.

We consider the general form of a class of PDE in the form:

$$\begin{cases} u_t(x,t) + \mathcal{N}[u(x,t),\lambda] = 0, x \in \Omega \subset \mathbb{R}^N, t \in [0,T] \\ u(x,0) = g(x), x \in \Omega \text{ Initial Condition} \\ u(x,t) = h(x,t), x \in \partial\Omega \text{ Boundary Value Condition} \\ \dfrac{\partial u}{\partial n}(x,t) = k(x,t), x \in \partial\Omega \text{ Boundary Value Condition} \end{cases} \tag{2.107}$$

where $\mathcal{N}$ is a nonlinear differential operator, λ is the parameter, $\partial\Omega$ is the boundary of Ω, and boundary condition is the Cauchy boundary condition. In this setting, $g : \Omega \subset \mathbb{R}^n \to \mathbb{R}$, $h : \partial\Omega \times \mathbb{R} \to \mathbb{R}$ and $k : \partial\Omega \times \mathbb{R} \to \mathbb{R}$ are known function and we want to find unknown function $u : \Omega \times \mathbb{R} \to \mathbb{R}$.

Assume there is a neural network $neural\_net$ that approximate the function $u(x,t)$, i.e., on input (x,t) it compute approximation of $u(x,t)$. So if we assume $\hat{u} = neural\_net(x,t)$ then by reverse mode Automatic Differentiation we could compute partial derivatives of $\hat{u}$ with respect to t and x, [160]. But, what is the loss function? Our goal is to find an approximation of u such that it is the solution of the Eq (2.107). So,

$$\hat{u}_t + \mathcal{N}[\hat{u},\lambda] \tag{2.108}$$

is not exactly zero. Thus we need to minimize the following function,

$$Minimize[\hat{u}_t + \mathcal{N}[\hat{u}, \lambda]]^2 \tag{2.109}$$

since $\hat{u}$ has been obtain by weights and biases, we need to find optimum value of weights and biases to find the solution. But we need also our approximation respect the initial condition and boundary value conditions, so,

$$\begin{aligned} L(\theta, x, t, \lambda) = &[\hat{u}_t + \mathcal{N}[\hat{u}, \lambda]]^2 + [\hat{u}(x, 0) - g(x)]^2 + \\ &\chi_{\partial\Omega}(x).([\hat{u}(x, t) - h(x, t)]^2 + [\frac{\partial \hat{u}}{\partial n}(x, t) - k(x, t)]^2) \end{aligned} \tag{2.110}$$

where θ is the parameter of the neural network and $\chi_{\partial\Omega}$ is the characteristic function of $\partial\Omega$, i.e.,

$$\chi_{\partial\Omega}(x) = \begin{cases} 1 & \text{if } x \in \partial\Omega \\ 0 & \text{otherwise} \end{cases} \tag{2.111}$$

The loss function depends not just on the neural network parameter but also on space and time. So, we have a loss function for every instant (x, t). Suppose we fixed the parameter λ. Let N_b be the number of boundaries and initial training data and N_c be the number of collocation points. Then the cost function is as follows:

$$MSE = MSE_b + MSE_c \tag{2.112}$$

where,

$$MSE_b = \frac{1}{N_b} \sum_{i=1}^{N_b} [\hat{u}(x_i^b, 0) - g(x_i^b)]^2 + \chi_{\partial\Omega}(x_i^b).([\hat{u}(x_i^b, t_i^b) - h(x_i^b, t_i^b)]^2 + [\frac{\partial \hat{u}}{\partial n}(x_i^b, t_i^b) - k(x_i^b, t_i^b)]^2) \tag{2.113}$$

and

$$MSE_c = \frac{1}{N_c}\sum_{i=1}^{N_c}[\hat{u}_t(x_i^c, t_i^c + \mathcal{N}[\hat{u}(x_i^c, t_i^c, \lambda)]^2 \tag{2.114}$$

As an example consider one-dimentional heat equation with homogeneous Dirichlet boundary conditions on $(x, t) \in [0, 5] \times [0, 1]$

$$\begin{cases} u_t = u_{xx} \\ u(x, 0) = x(5 - x) \\ u(0, t) = u(5, t) = 0 \end{cases} \tag{2.115}$$

we know the analytic solution of this equation is as follow,

$$u(x, t) = \frac{1}{\sqrt{4\pi t}}\int_0^5 e^{-\frac{(x-y)^2}{4t}} y(5 - y)dy \tag{2.116}$$

the solution and the evolution of initial condition is depicted in Fig. 2.25.

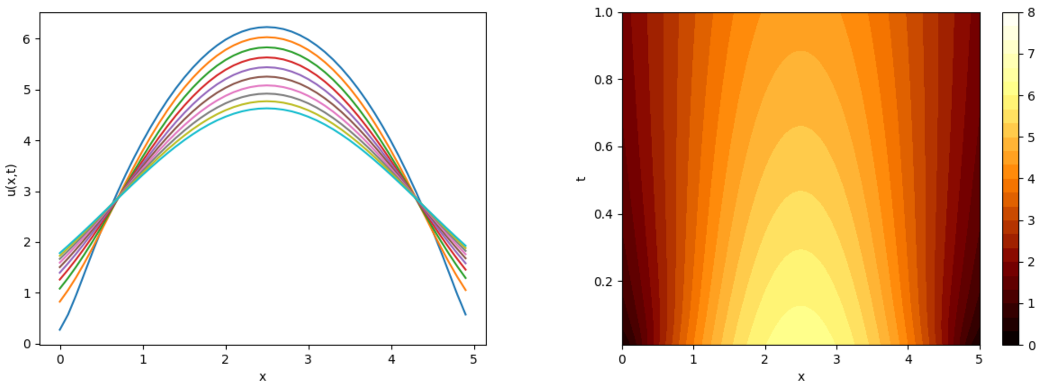


Figure 2.25: Heat equation on $[0, 5] \times [0, 1]$

If we represent MSE_0 as mean squar error of the initial condition, MSE_b the mean squar error of the boundary points and MSE_c as the mean squar error of the collocation points then we have,

$$MSE_0 = \frac{1}{N_0}\sum_{i=1}^{N_0}(\hat{u}(x_0^i, 0) - x_0^i(5 - x_0^i))^2 \tag{2.117}$$

where $\{x_0^i\}_{1\leq i\leq N_0}$ are training data of initial condition in $[0, 5]$.

$$MSE_b = \frac{1}{N_b}\sum_{i=1}^{N_b}(|\hat{u}(0, t_b^i)|^2 + |\hat{u}(5, t_b^i)|^2) \tag{2.118}$$

where $\{t_b^i\}_{1\leq i\leq N_b}$ are training data of boundary conditions in $[0, 1]$.

$$MSE_c = \frac{1}{N_c}\sum_{i=1}^{N_c}(\hat{u}_t(x_c^i, t_c^i) - \hat{u}_{xx}(x_c^i, t_c^i))^2 \tag{2.119}$$

where $\{(x_c^i, t_c^i)\}_{1\leq i\leq N_c}$ are training data of the collocation points in $[0, 5] \times [0, 1]$.

We know we can compute partial derivative of a neural network by reverse mode automatic differentiation, [160], i.e.,

$$\begin{aligned}
\hat{u}(x^i, t^i) &= neural\_net(x^i, t^i) \\
\hat{u}_t(x^i, t^i) &= \frac{\partial}{\partial t}(neural\_net(x^i, t^i)) \\
\hat{u}_{xx}(x^i, t^i) &= \frac{\partial^2}{\partial x^2}(neural\_net(x^i, t^i))
\end{aligned} \tag{2.120}$$

so the cost function of the heat equation with the given initial and boundary condition is as follow;

$$MSE(\theta) = MSE_0 + MSE_b + MSE_c \tag{2.121}$$

by minimizing $MSE(\theta)$ with respect to θ we could find the learning parameter of the heat equation.

2.7 Short Overview

From these comprehensive survey we have found some characteristics of continuous time series

- Irregular Sampling rate reduce the efficiency of the fixed-length and fixed-time-step based time series model such as RNN, CNN, ANN and other deep neural networks
- High dimension of time series and Discrete time sequence with fixed time steps do not allow traditional neural network to model continuous time series with higher accuracy.
- Pathological dependency introduces additional memory consumption and computation for traditional neural network models.
- Pathological dependency can be resolved with efficient technique to recognize Influential state of neural network models training

In this thesis, I have focused on resolving the above issue by introducing different neural network models.

2.8 Future Direction for Potential Research Area

Therefore, the primary concern of this survey work is "How can we ensure efficient event-based sensor's signal processing that generates an asynchronous data-driven output on a continuous-time sequence?" To have a better solution for this raised question, we also need to answer some other questions:

- How the existing neural network model can be improved to provide a solution.
- How does open data helps in risk assessment, fault tolerance, failure detection, real-time accuracy?

- How can output be updated asynchronously to be mapped with data from one or multiple events on a continuous-time series?
- How energy consumption and computation time can be optimised?

The main aim of the proposed work is to define a recurrent neural network-based (RNN) deep learning model for processing and modelling event-based sensor signal with a short and dynamic time gap in continuous time series and generate an asynchronous output at higher frequencies to deal with the following challenges:

- To reduce the high amount of computation for the short time gap in data sampling.
- To design the LSTM model using an ordinary differential equation solver.
- To develop asynchronous output in continuous time series. Asynchronous output can be beneficial for learning continuous series data generated by event-based sensors of IoT devices. For example, sensors in an autonomous vehicle are synthesised and processed in real-time with more accuracy.
- To immolate human brain workflow to generating asynchronous and event-triggered a spike in continuous time.

Recent emerging technologies, such as the Internet of things (IoT), cloud computing, edge computing, and 5G, use continuous-time data analysis and time-frequency analysis as a core fundamental technology for the interconnected ecosystem. For example, different cloud providers are extensively using deep learning algorithm for processing continuous-time data for multiple purposes, such as, to monitor performance and activity within a software-defined network (SDN), to allocate dynamic resources, to scale up or down the memory and power of compute engine and other resources, and to track activity for security and privacy purpose. Table 2.21 shows the use of a deep learning algorithm for continuous-time data analysis in different emerging technology.

Table 2.21: Different Projects in Health care with time series

Applications	**Popular neural Network**
Resource Scheduling	LSTM[171]
Edge Intelligence	RF [172]
Software defined Network (SDN)	DNN[173]
Self-protective framework	DNN[174]
Network Management	CNN[175]

2.9 Conclusion

In this paper, we have reviewed several recent research works focusing on time series modelling and prediction using a neural network and providing the mathematical background of these models. We have investigated all different challenges in the case of continuous-time series modelling with neural networks. In this paper, we have reviewed recent works mainly focused on dealing with the different challenges of modelling the continuous-time series using deep learning. We investigate several promising types of research that have established the foundation to introduce a new paradigm for modelling continuous-time series overcoming most of the associated challenges. There has been enormous attention to sensory event containing temporal information in recent days. Events happening continuously for the different real-time autonomous scenarios are significant for different research practitioners. For example, an automobile vehicle with different sensor types can continuously collect information from the context in real-time. In most cases, the information is a sequence of events in either a long or short time duration. Events occurred over continuous time are interrelated by either longer-term or short-term dependency. This dependency needs to be recognised, processed, stored, and analysed to predict the next event with similar behaviour correctly. For example, a sequence of events can detect the lane change behaviour of the driver or detect potential system failure or accident. There are also many similar research areas like mobile sensor information, where the temporal sequence plays a significantly important role. Research with time series is not a new thing. Different applications use continuous time series classification, prediction and generation. However, in most cases, real-time data are presented as a discrete-time model with a fixed sampling rate to avoid the complexity due to the massive length of series or higher dimensions. This cause a loss in the precision and accuracy of the result. In that context, most existing NN models are ill-suited to model a real-time continuous time series with a higher dimension and a long length. Continuous-time series modelling is one of the most promising research areas in deep learning for future research. Based on the comparative analysis, some of the significant research areas are as follows

- Sampling irregular data
- Reduce energy consumption
- Reduce the amount of computation
- Optimize performance
- Improve memory capacity
- Modelling long time sequence
- Efficient feature extracting
- Represent time sequence as a differential function
- Implementation of the deep learning algorithm in mobile sensor data
- Reduce the imbalance between positive and negative event
- Optimise overlapping events

2.10 References

[1] Dirk Helbing, Dirk Brockmann, Thomas Chadefaux, Karsten Donnay, Ulf Blanke, Olivia Woolley-Meza, Mehdi Moussaid, Anders Johansson, Jens Krause,

Sebastian Schutte, et al. Saving human lives: What complexity science and information systems can contribute. Journal of statistical physics, 158(3):735–781, 2015.

[2] Mark Alexa. Amazon Alexa: 2017 User Guide + 200 Ester Eggs. Independently published, 2017. ISBN 1520321562.

[3] Jan Beran. Mathematical foundations of time series analysis: a concise introduction. Springer, 2018.

[4] Boubacar Doucoure, Kodjo Agbossou, and Alben Cardenas. Time series prediction using artificial wavelet neural network and multi-resolution analysis: Application to wind speed data. Renewable Energy, 92:202–211, 2016.

[5] George EP Box, Gwilym M Jenkins, and Gregory C Reinsel. Time series analysis: forecasting and control, volume 734. John Wiley & Sons, 2011.

[6] John Cristian Borges Gamboa. Deep learning for time-series analysis. arXiv preprint arXiv:1701.01887, 2017.

[7] Wenhui Feng and Chongzhao Han. A novel approach for trajectory feature representation and anomalous trajectory detection. In 2015 18th International Conference on Information Fusion (Fusion), pages 1093–1099. IEEE, 2015.

[8] Neil A Gershenfeld and Andreas S Weigend. The future of time series. Technical report, Xerox Corporation, Palo Alto Research Center, 1993.

[9] Ashesh Jain, Amir R Zamir, Silvio Savarese, and Ashutosh Saxena. Structural-rnn: Deep learning on spatio-temporal graphs. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 5308–5317, 2016.

[10] Shanghang Zhang, Guanhang Wu, Joao P Costeira, and José MF Moura. Fcn-

rlstm: Deep spatio-temporal neural networks for vehicle counting in city cameras. In Computer Vision (ICCV), 2017 IEEE International Conference on, pages 3687–3696. IEEE, 2017.

[11] Hassan Ismail Fawaz, Germain Forestier, Jonathan Weber, Lhassane Idoumghar, and Pierre-Alain Muller. Deep learning for time series classification: a review. Data Mining and Knowledge Discovery, 33(4):917–963, 2019.

[12] Bryan Lim and Stefan Zohren. Time series forecasting with deep learning: A survey. arXiv preprint arXiv:2004.13408, 2020.

[13] Qingsong Wen, Liang Sun, Xiaomin Song, Jingkun Gao, Xue Wang, and Huan Xu. Time series data augmentation for deep learning: A survey. arXiv preprint arXiv:2002.12478, 2020.

[14] Cory Nguyens. Pokemon MIDIs. URL https://github.com/corynguyen19/midi-lstm-gan/tree/master/Pokemon%20MIDIs.

[15] QJ Liu, HY Zhang, KT Gao, B Xu, JZ Wu, and NF Fang. Time-frequency analysis and simulation of the watershed suspended sediment concentration based on the hilbert-huang transform (hht) and artificial neural network (ann) methods: A case study in the loess plateau of china. Catena, 179:107–118, 2019.

[16] MM Ali, PSV Jagadeesh, I-I Lin, and Je-Yuan Hsu. A neural network approach to estimate tropical cyclone heat potential in the indian ocean. IEEE Geoscience and Remote Sensing Letters, 9(6):1114–1117, 2012.

[17] Yifan Zhang, Peter Fitch, Peter Thorburn, and Maria de la Paz Vilas. Applying multi-layer artificial neural network and mutual information to the prediction of trends in dissolved oxygen. Frontiers in Environmental Science, 7:46, 2019.

[18] Ratneel Deo and Rohitash Chandra. Identification of minimal timespan problem for recurrent neural networks with application to cyclone wind-intensity

prediction. In 2016 International Joint Conference on Neural Networks (IJCNN), pages 489–496. IEEE, 2016.

[19] Filippo Maria Bianchi, Enrico Maiorino, Michael C Kampffmeyer, Antonello Rizzi, and Robert Jenssen. An overview and comparative analysis of recurrent neural networks for short term load forecasting. arXiv preprint arXiv:1705.04378, 2017.

[20] Daniel Neil, Michael Pfeiffer, and Shih-Chii Liu. Phased lstm: Accelerating recurrent network training for long or event-based sequences. In Advances in Neural Information Processing Systems, pages 3882–3890, 2016.

[21] Ilya Sutskever, Geoffrey E Hinton, and Graham W Taylor. The recurrent temporal restricted boltzmann machine. In Advances in neural information processing systems, pages 1601–1608, 2009.

[22] Sepp Hochreiter and Jurgen Schmidhuber. Long short-term memory. Neural computation, 9(8):1735–1780, 1997.

[23] Yu Zhu, Hao Li, Yikang Liao, Beidou Wang, Ziyu Guan, Haifeng Liu, and Deng Cai. What to do next: Modeling user behaviors by time-lstm. In Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17, pages 3602–3608, 2017.

[24] Zhengping Che, Sanjay Purushotham, Kyunghyun Cho, David Sontag, and Yan Liu. Recurrent neural networks for multivariate time series with missing values. Scientific reports, 8(1):6085, 2018.

[25] Geoffrey Hinton, Li Deng, Dong Yu, George Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Brian Kingsbury, et al. Deep neural networks for acoustic modeling in speech recognition. IEEE Signal processing magazine, 29, 2012.

[26] Tian Qi Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. In Advances in Neural Information Processing Systems, pages 6572–6583, 2018.

[27] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations. arXiv preprint arXiv:1711.10561, 2017.

[28] Cagatay Yildiz, Markus Heinonen, and Harri Lahdesmaki. ODE2VAE: Deep generative second order ODEs with bayesian neural networks. In Advances in Neural Information Processing Systems, pages 13412–13421, 2019.

[29] Xiping Wang and Ming Meng. A hybrid neural network and arima model for energy consumption forcasting. JCP, 7(5):1184–1190, 2012.

[30] Su Yang and Qing Zhu. Continuous chinese sign language recognition with cnn-lstm. In Ninth International Conference on Digital Image Processing (ICDIP 2017), volume 10420, page 104200F. International Society for Optics and Photonics, 2017.

[31] Oscar Koller, O Zargaran, Hermann Ney, and Richard Bowden. Deep sign: hybrid cnn-hmm for continuous sign language recognition. In Proceedings of the British Machine Vision Conference 2016, 2016.

[32] Mikel Canizo, Isaac Triguero, Angel Conde, and Enrique Onieva. Multi-head CNN-RNN for multi-time series anomaly detection: An industrial case study. Neurocomputing, 363:246–260, 2019.

[33] Ed AK Cohen and Andrew T Walden. A statistical study of temporally smoothed wavelet coherence. IEEE Transactions on Signal Processing, 58(6):2964–2973, 2010.

[34] Shivam Bhardwaj, E Chandrasekhar, Priyanka Padiyar, and Vikram M Gadre. A

comparative study of wavelet-based ann and classical techniques for geophysical time-series forecasting. Computers & Geosciences, page 104461, 2020.

[35] Gilseung Ahn and Sun Hur. Efficient genetic algorithm for feature selection for early time series classification. Computers & Industrial Engineering, 142: 106345, 2020.

[36] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010. URL http://yann.lecun.com/exdb/mnist/.

[37] Johann Faouzi and Hicham Janati. pyts: A python package for time series classification. Journal of Machine Learning Research, 21(46):1–6, 2020.

[38] Saloni Dash, Ritik Dutta, Isabelle Guyon, Adrien Pavao, Andrew Yale, and Kristin P Bennett. Synthetic event time series health data generation. arXiv preprint arXiv:1911.06411, 2019.

[39] Davide Anguita, Alessandro Ghio, Luca Oneto, Xavier Parra, and Jorge Luis Reyes-Ortiz. A public domain dataset for human activity recognition using smartphones. In Esann, 2013.

[40] Learn about time series cross-correlations in SPSS with data from the USDA feed grains database (1876–2015). URL https://methods.sagepub.com/dataset/time-series-cross-correlations-in-us-feedgrains-1876-2015.

[41] R. P. Ferreira, A. Martiniano, A. Ferreira, A. Ferreira, and R. J. Sassi. Study on daily demand forecasting orders using artificial neural network. IEEE Latin America Transactions, 14(3):1519–1525, 2016.

[42] . De Vito, E. Massera, M. Piga, L. Martinotto, and G. Di Francia. On field calibration of an electronic nose for benzene estimation in an urban pollution monitoring scenario. Sensors and Actuators B: Chemical, 129(22), February 2008.

[43] Yande Li, Taiqian Wang, Lian Li, Caihong Li, Yi Yang, Li Liu, et al. Hand gesture recognition and real-time game control based on a wearable band with 6-axis sensors. In 2018 International Joint Conference on Neural Networks (IJCNN), pages 1–6. IEEE, 2018.

[44] Stephen D. Bay and Dennis F. Kibler and michael j. Pazzani and Padhraic Smyth. The UCI KDD Archive of Large Data Sets for Data Mining Research and Experimentation. SIGKDD Explorations, 2., 2000.

[45] A. Johnson, T. Pollard, R. Mark, S. Berkowitz, and S. Horng. Mimic-cxr database (version 2. 0, 2019.

[46] Hoang Anh Dau, Eamonn Keogh, Kaveh Kamgar, Chin-Chia Michael Yeh, Yan Zhu, Shaghayegh Gharghabi, Chotirat Ann Ratanamahatana, Yanping, Bing Hu, Nurjahan Begum, Anthony Bagnall, Abdullah Mueen, Gustavo Batista, and Hexagon-ML. The ucr time series classification archive, October 2018. https://www.cs.ucr.edu/~eamonn/time_series_data_2018/.

[47] Fernando Jiménez, José Palma, Gracia Sánchez, David Marín, Francisco Palacios, and Lucía López. Feature selection based multivariate time series forecasting: An application to antibiotic resistance outbreaks prediction. Artificial Intelligence in Medicine, page 101818, 2020.

[48] Bhanu Pratap Singh, Iman Deznabi, Bharath Narasimhan, Bryon Kucharski, Rheeya Uppaal, Akhila Josyula, and Madalina Fiterau. Multi-resolution networks for flexible irregular time series modeling (multi-fit). arXiv preprint arXiv:1905.00125, 2019.

[49] Arthur Le Guennec, Simon Malinowski, and Romain Tavenard. Data augmentation for time series classification using convolutional neural networks. 2016.

[50] Mohammad Taha Bahadori and Zachary Chase Lipton. Temporal-clustering invariance in irregular healthcare time series. arXiv preprint arXiv:1904.12206, 2019.

[51] Lin Wang, Zhigang Wang, and Shan Liu. An effective multivariate time series classification approach using echo state network and adaptive differential evolution algorithm. Expert Systems with Applications, 43:237–249, 2016.

[52] Takashi Kuremoto, Takaomi Hirata, Masanao Obayashi, Shingo Mabu, and Kunikazu Kobayashi. Training deep neural networks with reinforcement learning for time series forecasting. In Time Series Analysis. IntechOpen, 2019.

[53] Peng Jiang, Cheng Chen, and Xiao Liu. Time series prediction for evolutions of complex systems: A deep learning approach. In 2016 IEEE International Conference on Control and Robotics Engineering (ICCRE), pages 1–6. IEEE, 2016.

[54] Mario Chavez and Bernard Cazelles. Detecting dynamic spatial correlation patterns with generalized wavelet coherence and non-stationary surrogate data. Scientific reports, 9(1):1–9, 2019.

[55] Víctor Campos, Brendan Jou, Xavier Giró-i Nieto, Jordi Torres, and Shih-Fu Chang. Skip rnn: Learning to skip state updates in recurrent neural networks. arXiv preprint arXiv:1708.06834, 2017.

[56] Takashi Kuremoto, Shinsuke Kimura, Kunikazu Kobayashi, and Masanao Obayashi. Time series forecasting using a deep belief network with restricted boltzmann machines. Neurocomputing, 137:47–56, 2014.

[57] Seo-Jin Bang, Yuchuan Wang, and Yang Yang. Phased-lstm based predictive model for longitudinal ehr data with missing values, 2018.

[58] Miguel Henry and George Judge. Permutation entropy and information recovery in nonlinear dynamic economic time series. Econometrics, 7(1):10, 2019.

[59] Usha Manasi Mohapatra, Babita Majhi, and Suresh Chandra Satapathy. Financial time series prediction using distributed machine learning techniques. Neural Computing and Applications, 31(8):3369–3384, 2019.

[60] Jie Wang, Jun Wang, Wen Fang, and Hongli Niu. Financial time series prediction using elman recurrent random neural networks. Computational intelligence and neuroscience, 2016, 2016.

[61] Jerome T Connor, R Douglas Martin, and Les E Atlas. Recurrent neural networks and robust time series prediction. IEEE transactions on neural networks, 5(2): 240–254, 1994.

[62] Shengdong Zhang, Soheil Bahrampour, Naveen Ramakrishnan, Lukas Schott, and Mohak Shah. Deep learning on symbolic representations for large-scale heterogeneous time-series event prediction. In 2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pages 5970–5974. IEEE, 2017.

[63] Masoumeh Poormehdi Ghaemmaghami. Tracking of humans in video stream using lstm recurrent neural network, 2017.

[64] Anthony Brunel, Johanna Pasquet, Jérôome PASQUET, Nancy Rodriguez, Frédéric Comby, Dominique Fouchez, and Marc Chaumont. A CNN adapted to time series for the classification of supernovae. Electronic Imaging, 2019(14): 90–1, 2019.

[65] Nicola Michielli, U Rajendra Acharya, and Filippo Molinari. Cascaded lstm recurrent neural network for automated sleep stage classification using single-channel eeg signals. Computers in biology and medicine, 106:71–81, 2019.

[66] Stanislas Chambon, Mathieu N Galtier, Pierrick J Arnal, Gilles Wainrib, and Alexandre Gramfort. A deep learning architecture for temporal sleep stage classification using multivariate and multimodal time series. IEEE Transactions on Neural Systems and Rehabilitation Engineering, 26(4):758–769, 2018.

[67] Fan Zhang, Tong Wu, Yunlong Wang, Yong Cai, Cao Xiao, Emily Zhao, Lucas Glass, and Jimeng Sun. Predicting treatment initiation from clinical time series data via graph-augmented time-sensitive model. arXiv preprint arXiv:1907.01099, 2019.

[68] Brandon Ballinger, Johnson Hsieh, Avesh Singh, Nimit Sohoni, Jack Wang, Geoffrey H Tison, Gregory M Marcus, Jose M Sanchez, Carol Maguire, Jeffrey E Olgin, et al. Deepheart: semi-supervised sequence learning for cardiovascular risk prediction. In Thirty-Second AAAI Conference on Artificial Intelligence, 2018.

[69] Georgios Kissas, Yibo Yang, Eileen Hwuang, Walter R Witschey, John A Detre, and Paris Perdikaris. Machine learning in cardiovascular flows modeling: Predicting arterial blood pressure from non-invasive 4d flow mri data using physics-informed neural networks. Computer Methods in Applied Mechanics and Engineering, 358:112623, 2020.

[70] Candy Obdulia Sosa-Jiménez, Homero Vladimir Ríos-Figueroa, Ericka Janet Rechy-Ramírez, Antonio Marin-Hernandez, and Ana Luisa Solís González-Cosío. Real-time mexican sign language recognition. In 2017 IEEE International Autumn Meeting on Power, Electronics and Computing (ROPEC), pages 1–6. IEEE, 2017.

[71] Dario Amodei, Rishita Anubhai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, et al. End to end speech recognition in english and mandarin. 2016.

[72] Xuanqing Liu, Tesi Xiao, Si Si, Qin Cao, Sanjiv Kumar, and Cho-Jui Hsieh. Neural SDE: Stabilizing neural ODE networks with stochastic noise. arXiv preprint arXiv:1906.02355, 2019.

[73] Shuochao Yao, Ailing Piao, Wenjun Jiang, Yiran Zhao, Huajie Shao, Shengzhong Liu, Dongxin Liu, Jinyang Li, Tianshi Wang, Shaohan Hu, et al. Stfnets: Learning sensing signals from the time-frequency perspective with short-time fourier neural networks. In The World Wide Web Conference, pages 2192–2202, 2019.

[74] Jingguang Zhou and Zili Huang. Recover missing sensor data with iterative imputing network. In Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence, 2018.

[75] Shuochao Yao, Shaohan Hu, Yiran Zhao, Aston Zhang, and Tarek Abdelzaher. Deepsense: A unified deep learning framework for time-series mobile sensing data processing. In Proceedings of the 26th International Conference on World Wide Web, pages 351–360. International World Wide Web Conferences Steering Committee, 2017.

[76] Dymitr Ruta and Bogdan Gabrys. Neural network ensembles for time series prediction. In 2007 International Joint Conference on Neural Networks, pages 1204–1209. IEEE, 2007.

[77] Ioannis E Livieris, Emmanuel Pintelas, and Panagiotis Pintelas. A cnn–lstm model for gold price time-series forecasting. Neural Computing and Applications, pages 1–10, 2020.

[78] Vahid Nourani and Nima Farboudfam. Rainfall time series disaggregation in mountainous regions using hybrid wavelet-artificial intelligence methods. Environmental research, 168:306–318, 2019.

[79] MR Khaledian, M Isazadeh, SM Biazar, and QB Pham. Simulating caspian sea surface water level by artificial neural network and support vector machine models. Acta Geophysica, pages 1–11, 2020.

[80] Zolal Ayazpour, Amin E Bakhshipour, and Ulrich Dittmer. Combined sewer flow prediction using hybrid wavelet artificial neural network model. In International Conference on Urban Drainage Modelling, pages 693–698. Springer, 2018.

[81] Zhiyuan Wu, Changbo Jiang, Mack Conde, Bin Deng, and Jie Chen. Hybrid improved empirical mode decomposition and bp neural network model for the prediction of sea surface temperature. Ocean Science, 15(2):349–360, 2019.

[82] Jimmy Byakatonda, BP Parida, Piet K Kenabatho, and DB Moalafhi. Prediction of onset and cessation of austral summer rainfall and dry spell frequency analysis in semiarid botswana. Theoretical and applied climatology, 135(1-2):101–117, 2019.

[83] Zhuyun Chen, Konstantinos Gryllias, and Weihua Li. Intelligent fault diagnosis for rotary machinery using transferable convolutional neural network. IEEE Transactions on Industrial Informatics, 16(1):339–349, 2019.

[84] Hao Hu and Guo-Jun Qi. State-frequency memory recurrent neural networks. In Proceedings of the 34th International Conference on Machine Learning-Volume 70, pages 1568–1577. JMLR. org, 2017.

[85] Shun-Yao Shih, Fan-Keng Sun, and Hung-yi Lee. Temporal pattern attention for multivariate time series forecasting. Machine Learning, 108(8-9):1421–1441, 2019.

[86] Shiyu Chang, Yang Zhang, Wei Han, Mo Yu, Xiaoxiao Guo, Wei Tan, Xiaodong Cui, Michael Witbrock, Mark A Hasegawa-Johnson, and Thomas S Huang.

Dilated recurrent neural networks. In Advances in Neural Information Processing Systems, pages 77–87, 2017.

[87] Yulia Rubanova, Tian Qi Chen, and David K Duvenaud. Latent ordinary differential equations for irregularly-sampled time series. In Advances in Neural Information Processing Systems, pages 5321–5331, 2019.

[88] Patrick Kidger, James Morrill, James Foster, and Terry Lyons. Neural controlled differential equations for irregular time series. arXiv preprint arXiv:2005.08926, 2020.

[89] Mansura Habiba and Barak A. Pearlmutter. Neural Ordinary Differential Equation based Recurrent Neural Network Model. arXiv e-prints, art. arXiv:2005.09807, May 2020.

[90] David M Kreindler and Charles J Lumsden. The effects of the irregular sample and missing data in time series analysis. Nonlinear dynamics, psychology, and life sciences, 2006.

[91] Shaojie Bai, J Zico Kolter, and Vladlen Koltun. Convolutional sequence modeling revisited. 2018.

[92] Yifan Sun, Linan Zhang, and Hayden Schaeffer. Neupde: Neural network based ordinary and partial differential equations for modeling time-dependent data. arXiv preprint arXiv:1908.03190, 2019.

[93] Mansura Habiba and Barak A. Pearlmutter. Neural ODEs for Informative Missingness in Multivariate Time Series. arXiv e-prints, art. arXiv:2005.10693, May 2020.

[94] Lova Sun, Eili Y Klein, and Ramanan Laxminarayan. Seasonality and temporal correlation between community antibiotic use and resistance in the united states. Clinical infectious diseases, 55(5):687–694, 2012.

[95] Philipp Holl, Vladlen Koltun, and Nils Thuerey. Learning to control pdes with differentiable physics. arXiv preprint arXiv:2001.07457, 2020.

[96] Philip Clemson, Gemma Lancaster, and Aneta Stefanovska. Reconstructing time-dependent dynamics. Proceedings of the IEEE, 104(2):223–241, 2016.

[97] Lawrence W Sheppard, James R Bell, Richard Harrington, and Daniel C Reuman. Changes in large-scale climate alter spatial synchrony of aphid pests. Nature Climate Change, 6(6):610–613, 2016.

[98] Bernard Cazelles, Kévin Cazelles, and Mario Chavez. Wavelet analysis in ecology and epidemiology: impact of statistical tests. Journal of the Royal Society Interface, 11(91):20130585, 2014.

[99] Eric KW Ng and Johnny CL Chan. Geophysical applications of partial wavelet coherence and multiple wavelet coherence. Journal of Atmospheric and Oceanic Technology, 29(12):1845–1853, 2012.

[100] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. The Journal of Machine Learning Research, 15(1):1929–1958, 2014.

[101] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. science, 313(5786):504–507, 2006.

[102] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. arXiv preprint arXiv:1609.03499, 2016.

[103] Kanad Chakraborty, Kishan Mehrotra, Chilukuri K Mohan, and Sanjay Ranka. Forecasting the behavior of multivariate time series using neural networks. Neural networks, 5(6):961–970, 1992.

[104] Will Grathwohl, Ricky TQ Chen, Jesse Bettencourt, Ilya Sutskever, and David Duvenaud. Ffjord: Free-form continuous dynamics for scalable reversible generative models. arXiv preprint arXiv:1810.01367, 2018.

[105] Yaofeng Desmond Zhong, Biswadip Dey, and Amit Chakraborty. Symplectic ode-net: Learning hamiltonian dynamics with control. arXiv preprint arXiv:1909.12077, 2019.

[106] Edward Choi, Mohammad Taha Bahadori, Andy Schuetz, Walter F Stewart, and Jimeng Sun. Doctor ai: Predicting clinical events via recurrent neural networks. In Machine Learning for Healthcare Conference, pages 301–318, 2016.

[107] Riccardo Miotto, Li Li, Brian A Kidd, and Joel T Dudley. Deep patient: an unsupervised representation to predict the future of patients from the electronic health records. Scientific reports, 6:26094, 2016.

[108] Takashi Kuremoto, Shinsuke Kimura, Kunikazu Kobayashi, and Masanao Obayashi. Time series forecasting using a deep belief network with restricted boltzmann machines. Neurocomputing, 137:47–56, 2014.

[109] Takashi Kuremoto, Masanao Obayashi, Kunikazu Kobayashi, Takaomi Hirata, and Shingo Mabu. Forecast chaotic time series data by dbns. In 2014 7th International Congress on Image and Signal Processing, pages 1130–1135. IEEE, 2014.

[110] N. Kumar and N. Sukavanam. A cascaded cnn model for multiple human tracking and re-localization in complex video sequences with large displacement. Multimedia Tools and Applications, 79:6109–6134, 2019.

[111] E Egrioglu, CH Aladag, U Yolcu, and AZ Dalar. A hybrid high order fuzzy time series forecasting approach based on pso and anns methods. Am. J. Intell. Syst, 6(1):8, 2016.

[112] Erol Egrioglu, Ufuk Yolcu, Cagdas Hakan Aladag, and Eren Bas. Recurrent multiplicative neuron model artificial neural network for non-linear time series forecasting. Neural Processing Letters, 41(2):249–258, 2015.

[113] Sung Jin Yoo, Jin Bae Park, and Yoon Ho Choi. Stable predictive control of chaotic systems using self-recurrent wavelet neural network. international journal of control, automation, and systems, 3(1):43–55, 2005.

[114] Ahmed Tealab. Time series forecasting using artificial neural networks methodologies: A systematic review. Future Computing and Informatics Journal, 3(2):334–340, 2018.

[115] Xuejun Chen, Shiqiang Jin, Shanshan Qin, and Laping Li. Short-term wind speed forecasting study and its application using a hybrid model optimized by cuckoo search. Mathematical Problems in Engineering, 2015, 2015.

[116] Cem Kocak, Ali Zafer Dalar, Ozge Cagcag Yolcu, Eren Bas, and Erol Egrioglu. A new fuzzy time series method based on an arma-type recurrent pi-sigma artificial neural network. Soft Computing, pages 1–10, 2019.

[117] Mingdong Sun, Xuyong Li, and Gwangseob Kim. Precipitation analysis and forecasting using singular spectrum analysis with artificial neural networks. Cluster Computing, 22(5):12633–12640, 2019.

[118] Qifa Xu, Xingxuan Zhuo, Cuixia Jiang, and Yezheng Liu. An artificial neural network for mixed frequency data. Expert Systems with Applications, 118: 127–139, 2019.

[119] Ehsan Haghighat and Ruben Juanes. Sciann: A keras wrapper for scientific computations and physics-informed deep learning using artificial neural networks. arXiv preprint arXiv:2005.08803, 2020.

[120] Sung-Jin Yoo, Yoon-Ho Choi, and Jin-Bae Park. Identification of dynamic

systems using a self recurrent wavelet neural network: Convergence analysis via adaptive learning rates. Journal of Institute of Control, Robotics and Systems, 11(9):781–788, 2005.

[121] Martin Längkvist, Lars Karlsson, and Amy Loutfi. A review of unsupervised feature learning and deep learning for time-series modeling. Pattern Recognition Letters, 42:11–24, 2014.

[122] Ilya Sutskever. Training recurrent neural networks. University of Toronto Toronto, Ontario, Canada, 2013.

[123] Agoston E Eiben, Emile HL Aarts, and Kees M Van Hee. Global convergence of genetic algorithms: A markov chain analysis. In International Conference on Parallel Problem Solving from Nature, pages 3–12. Springer, 1990.

[124] Edward N Lorenz. Deterministic nonperiodic flow. Journal of the atmospheric sciences, 20(2):130–141, 1963.

[125] James NK Liu, Yanxing Hu, Jane Jia You, and Pak Wai Chan. Deep neural network based feature representation for weather forecasting. In Proceedings on the International Conference on Artificial Intelligence (ICAI), page 1. The Steering Committee of The World Congress in Computer Science, Computer . . . , 2014.

[126] Silvan C Quax, Michele D'asaro, and Marcel AJ Van Gerven. Adaptive time scales in recurrent neural networks. Scientific Reports, 10(1):1–14, 2020.

[127] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 3431–3440, 2015.

[128] Rahul Dey and Fathi M Salemt. Gate-variants of gated recurrent unit (gru) neural

networks. In 2017 IEEE 60th international midwest symposium on circuits and systems (MWSCAS), pages 1597–1600. IEEE, 2017.

[129] Gustavo EAPA Batista, Maria Carolina Monard, et al. A study of k-nearest neighbour as an imputation method. His, 87(251-260):48, 2002.

[130] Carl De Boor, Carl De Boor, Etats-Unis Mathématicien, Carl De Boor, and Carl De Boor. A practical guide to splines, volume 27. springer-verlag New York, 1978.

[131] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. Computer, 42(8):30–37, 2009.

[132] Julie Josse, Marie Chavent, Benot Liquet, and François Husson. Handling missing values with regularized iterative multiple correspondence analysis. Journal of classification, 29(1):91–116, 2012.

[133] Daniel J Stekhoven and Peter Bühlmann. Missforest—non-parametric missing value imputation for mixed-type data. Bioinformatics, 28(1):112–118, 2011.

[134] Rahul Mazumder, Trevor Hastie, and Robert Tibshirani. Spectral regularization algorithms for learning large incomplete matrices. Journal of machine learning research, 11(Aug):2287–2322, 2010.

[135] Melissa J Azur, Elizabeth A Stuart, Constantine Frangakis, and Philip J Leaf. Multiple imputation by chained equations: what is it and how does it work? International journal of methods in psychiatric research, 20(1):40–49, 2011.

[136] Shaojie Bai, J Zico Kolter, and Vladlen Koltun. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. arXiv preprint arXiv:1803.01271, 2018.

[137] Penghui Liu, Jing Liu, and Kai Wu. CNN-FCM: System modeling promotes

stability of deep learning in time series prediction. Knowledge-Based Systems, page 106081, 2020.

[138] Michael Pfeiffer and Thomas Pfeil. Deep learning with spiking neurons: Opportunities and challenges. Frontiers in neuroscience, 12, 2018.

[139] SHI Xingjian, Zhourong Chen, Hao Wang, Dit-Yan Yeung, Wai-Kin Wong, and Wang-chun Woo. Convolutional lstm network: A machine learning approach for precipitation nowcasting. In Advances in neural information processing systems, pages 802–810, 2015.

[140] James Bradbury, Stephen Merity, Caiming Xiong, and Richard Socher. Quasi-recurrent neural networks. arXiv preprint arXiv:1611.01576, 2016.

[141] Takaomi Hirata, Takashi Kuremoto, Masanao Obayashi, Shingo Mabu, and Kunikazu Kobayashi. Time series prediction using dbn and arima. In 2015 International Conference on Computer Application Technologies, pages 24–29. IEEE, 2015.

[142] Rohitash Chandra and Mengjie Zhang. Cooperative coevolution of elman recurrent neural networks for chaotic time series prediction. Neurocomputing, 86:116–123, 2012.

[143] Serkan Kiranyaz, Turker Ince, and Moncef Gabbouj. Real-time patient-specific ecg classification by 1-d convolutional neural networks. IEEE Transactions on Biomedical Engineering, 63(3):664–675, 2016.

[144] Zhiheng Huang, Wei Xu, and Kai Yu. Bidirectional lstm-crf models for sequence tagging. arXiv preprint arXiv:1508.01991, 2015.

[145] Yi-Jen Wang and Chin-Teng Lin. Runge kutta neural network for identification of continuous systems. In SMC'98 Conference Proceedings. 1998 IEEE

International Conference on Systems, Man, and Cybernetics (Cat. No. 98CH36218), volume 4, pages 3277–3282. IEEE, 1998.

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

[147] 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.

[148] Edward Nelson. Derivation of the schrödinger equation from newtonian mechanics. Physical review, 150(4):1079, 1966.

[149] Atilim Gunes Baydin and Barak A Pearlmutter. Automatic differentiation of algorithms for machine learning. arXiv preprint arXiv:1404.7456, 2014.

[150] Liu Yang, Dongkun Zhang, and George Em Karniadakis. Physics-informed generative adversarial networks for stochastic differential equations. arXiv preprint arXiv:1811.02033, 2018.

[151] Xiaowei Jia, Jared Willard, Anuj Karpatne, Jordan S Read, Jacob A Zwart, Michael Steinbach, and Vipin Kumar. Physics-guided machine learning for scientific discovery: An application in simulating lake temperature profiles. arXiv preprint arXiv:2001.11086, 2020.

[152] Kun Qian, Abduallah Mohamed, and Christian Claudel. Physics informed data driven model for flood prediction: Application of deep learning in prediction of urban flood development. arXiv preprint arXiv:1908.10312, 2019.

[153] Yigit Anil Yucesan and Felipe AC Viana. Wind turbine main bearing fatigue

life estimation with physics-informed neural networks. In Proceedings of the Annual Conference of the PHM Society, volume 11, 2019.

[154] Guofei Pang, Lu Lu, and George Em Karniadakis. fpinns: Fractional physics-informed neural networks. SIAM Journal on Scientific Computing, 41(4):A2603–A2626, 2019.

[155] Guofei Pang, Marta D'Elia, Michael Parks, and George E Karniadakis. npinns: nonlocal physics-informed neural networks for a parametrized nonlocal universal laplacian operator. algorithms and applications. arXiv preprint arXiv:2004.04276, 2020.

[156] Barak A. Pearlmutter. Learning state space trajectories in recurrent neural networks. Neural Computation, 1(2):263–9, 1989.

[157] Bo Chang, Lili Meng, Eldad Haber, Lars Ruthotto, David Begert, and Elliot Holtham. Reversible architectures for arbitrarily deep residual neural networks. arXiv preprint arXiv:1709.03698, 2017.

[158] Yiping Lu, Aoxiao Zhong, Quanzheng Li, and Bin Dong. Beyond finite layer neural networks: Bridging deep architectures and numerical differential equations. In International Conference on Machine Learning, pages 3276–3285, 2018.

[159] Stuart E Dreyfus. Dynamic programming and the calculus of variations. Technical report, RAND CORP SANTA MONICA CA, 1965.

[160] Barak A. Pearlmutter and Jeffrey Mark Siskind. Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. ACM Trans. Program. Lang. Syst., 30:7:1–7:36, 2008.

[161] Athanasios Papoulis and S Unnikrishna Pillai. Probability, random variables, and stochastic processes. Tata McGraw-Hill Education, 2002.

[162] Michael Spivak. Calculus on manifolds: a modern approach to classical theorems of advanced calculus. CRC press, 2018.

[163] Paul R Halmos. Finite-dimensional vector spaces. Courier Dover Publications, 2017.

[164] Edward De Brouwer, Jaak Simm, Adam Arany, and Yves Moreau. Gru-ode-bayes: Continuous modeling of sporadically-observed time series. In Advances in Neural Information Processing Systems, pages 7377–7388, 2019.

[165] Steven Fernandes, Sunny Raj, Eddy Ortiz, Iustina Vintila, Margaret Salter, Gordana Urosevic, and Sumit Jha. Predicting heart rate variations of deepfake videos using neural ode. In Proceedings of the IEEE International Conference on Computer Vision Workshops, pages 0–0, 2019.

[166] Anindya Sarkar, Anirudh Sunder Raj, and Raghu Sesha Iyengar. Improving robustness of time series classifier with neural ode guided gradient based data augmentation. arXiv preprint arXiv:1910.06813, 2019.

[167] Roger Alexander. Solving ordinary differential equations i: Nonstiff problems (e. hairer, sp norsett, and g. wanner). Siam Review, 32(3):485, 1990.

[168] Emilien Dupont, Arnaud Doucet, and Yee Whye Teh. Augmented neural odes. In Advances in Neural Information Processing Systems, pages 3134–3144, 2019.

[169] Lars Ruthotto and Eldad Haber. Deep neural networks motivated by partial differential equations. Journal of Mathematical Imaging and Vision, pages 1–13, 2019.

[170] Mai Zhu, Bo Chang, and Chong Fu. Convolutional neural networks combined with runge-kutta methods. arXiv preprint arXiv:1802.08831, 2018.

[171] Mu Yan, Gang Feng, Jianhong Zhou, Yao Sun, and Ying-Chang Liang. Intelligent

resource scheduling for 5g radio access network slicing. IEEE Transactions on Vehicular Technology, 68(8):7691–7703, 2019.

[172] Michele Polese, Rittwik Jana, Velin Kounev, Ke Zhang, Supratim Deb, and Michele Zorzi. Machine learning at the edge: A data-driven architecture with applications to 5g cellular networks. IEEE Transactions on Mobile Computing, 2020.

[173] Songlin Sun, Liang Gong, Bo Rong, and Kejie Lu. An intelligent sdn framework for 5g heterogeneous networks. IEEE Communications Magazine, 53(11):142–147, 2015.

[174] Jorge Maestre Vidal and Marco Antonio Sotelo Monge. Framework for anticipatory self-protective 5g environments. In Proceedings of the 14th International Conference on Availability, Reliability and Security, pages 1–9, 2019.

[175] Dario Bega, Marco Gramaglia, Marco Fiore, Albert Banchs, and Xavier Costa-Perez. Deepcog: Cognitive network management in sliced 5g networks with deep learning. In IEEE INFOCOM 2019-IEEE Conference on Computer Communications, pages 280–288. IEEE, 2019.

CHAPTER 3

NEURAL ORDINARY DIFFERENTIAL EQUATION BASED RECURRENT NEURAL NETWORK MODEL

This chapter is an expanded version of a paper published in 2020 31st Irish Signals and Systems Conference (ISSC)

Neural differential equations are a promising new family member of machine learning models. They show the potential of differential equations for time series data analysis. The main goal of this work is to answer the following questions:

1. *Can ODE redefine the existing neural network model?*
2. *Can Neural ODEs solve the irregular sampling rate challenge of existing neural network models for a continuous time series, i.e., length and dynamic nature?*
3. *How do we reduce existing Neural ODE systems' training and evaluation time?*

This work leverages the mathematical foundation of ODEs to redesign traditional RNNs such as Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU). The main contribution of this section is to illustrate the design of two new ODE-based RNN models (GRU-ODE model and LSTM-ODE), which can compute the hidden state and cell state at any point in time using an ODE solver. These models reduce the computation overhead of hidden states and cell states by a vast amount. The performance evaluation of these two new models for learning continuous time series with irregular sampling rates shows that these new ODE based RNN models require less training time than Latent ODEs and conventional Neural ODEs. Moreover, they can *quickly achieve higher accuracy, and the neural network design is more straightforward than previous neural ODE systems.*

3.1 Introduction

Continuous-time series is a fundamental data structure for different research areas, i.e., sensor-based IoT, energy consumption, weather, music generation and other applications. It also has dynamic characteristics such as high sampling frequency, inconsistent sampling rate, multiple variables and parameters, higher dimension and higher length. In many applications, such as automobiles, time series can be very dynamic. Neural network models need to support these characteristics to process continuous time series. RNN has become the pioneer in time series modelling and processing among all the available neural network models due to its gating unit and memory storing capacity. However, the majority of existing RNNs process continuous time series as a discrete-time sequence with a fixed sampling rate and fixed sampling frequency [1]. This makes real-time processing difficult. It also imposes a high computation load and memory usage. In addition, due to the dependency on the computation of previous states, such RNN models are prone to some vulnerabilities. For example, if the time gap between two consecutive observations is too big, it can adversely affect the model's efficiency. Therefore, such RNNs are only suitable for time series of moderate length with fixed sampling rates, few missing values, and short time intervals between observations.

Chen et al. [2] and Rubanova et al. [3] demonstrated the strength of Neural ODEs [4, 5] for processing time series. Chen et al. [2] proposes to compute iterative updates of RNN as Euler discretization of continuous transformation [6]. Based on this idea, the continuous hidden dynamics of a neural network can be parametrized by ordinary differential equation (3.1). Equation (3.1) explains that a small change in the value of hidden dynamics $h(t)$ over time t can parameterize the continuous dynamics of hidden states of a neural network. Therefore, an ordinary differential equation (ODE) specified by (3.1) can compute the continuous change in the dynamics of hidden states of a neural network. This is the core concept of Neural ODEs or NODE models. Using the initial value problem to compute $h(T)$ at any time T from the initial hidden units $h(0)$ at time t_0, this continuous-depth model leverages a black-box ODE solver (f) to define and evaluate models. Another version of neural ODE, known as Latent-ODE [2], uses an ODE as an encoder whereas ODE-RNN [3] uses an RNN as an encoder.

$$\frac{d}{dt}h(t) = f(h_t, t, \delta_t) \tag{3.1}$$

ODE-RNN [3] uses an encoder and decoder to compute the hidden dynamics of the neural network. First, the input is fed to an encoder, and a differential equation solver is used to calculate the hidden state. Later, the decoder generates the output. This is an auto-regressive model, similar to [2], ODE-RNN can also compute the hidden dynamics at any time t. The dynamics between observations are learned rather than pre-defined, making ODE-RNN a suitable model for irregular and sparse data. As the computation is not dependent on the availability of data points, both ODE-RNN and Latent-ODE take more time for training and evaluation. The time depends on the length of the time series and the complexity of the data. The focus of this section is to consider the neural network as a function of continuous time; therefore, the new model proposed in this work does not use any additional encoder or decoder.

In the case of time series processing, RNN based neural network model is already ahead of others [1]. However, different RNN models consider continuous-time data as a discrete sequence of observations with a fixed sampling rate. Therefore, often it is challenging to compute a time series in real-time. The main goal of this work is to leverage the continuousness of neural ODE to design Neural Ordinary Differential Equation based RNN, which can reduce the overhead of additional encoding and decoding for simple time-series data processing. These proposed models can overcome the challenges imposed due to fixed time steps with marginal budget cost in shorter time duration and accurate real-time computation.

This work proposed to design the architecture of different RNN, e.g., GRU [7], or LSTM [8] using an ordinary differential equation (ODE), which is referred to as ODE-GRU and ODE-LSTM throughout this section. These proposed models can train the gradients themselves within marginal error using the ordinary differential equation solver. All parameters are learned during the training. The cell state and hidden state dynamics are computed using a black-box ODE solver. The *update* and *reset* functions of existing RNN models are redefined to leverage the ODE solver. These proposed models can process continuous time series in real-time with comparatively less computation and memory usage. These models also provide a generative process over time series similar to [3]. We compare the proposed models to several RNN variants and find that these new models can perform better when the data is sparse.

3.2 Motivation

Our main goal is to overcome the limitation of instability in the Neural Ordinary Differential Equation (NODE) model. This chapter explores the structure of Recurrent Neural networks and their potential to improve the performance of NODE models. Chapter 2 discusses different characteristics of the Neural ODE model. Some of these characteristics of the Neural ODE model are described as follows

3.2.1 Memory Efficiency

Neural ODE can compute gradients of loss concerning an ODE as shown in (3.1). For gradients computation, Neural ODE with a black box ODE solver, a Neural ODE model does not need backpropagating through the operations of the ODE solver. Neural ODE does not store the intermediate computation of the forward pass for a later back pass during this computation. Therefore, it is possible to train Neural ODE models with constant memory cost as a function of several states. I proposed two separate Neural ODE models using the RNN model structure in this work. The intermediate cell state for RNN is also computed in the proposed models for each forward pass. However, proposed models do not need to store the states for the later back-pass.

3.2.2 Efficient Computation

Although [3] describes the implementation of Neural ODE with the Euler Method, which is one of the most straightforward ODE solvers, Neural ODE can also be implemented with a modern ODE solver that can guarantee the flexibility of the solver along with the growth of the error along with capabilities for monitoring the change in the hidden dynamics. Therefore, any modern ODE solver can enrich Neural ODE computation and real-time monitoring.

3.2.3 Efficient Error and Loss function for optimization

ODE solvers are the core component of any Neural ODE model. Neural ODE model leverage the capability of ODE solvers to ensure a tolerance boundary for the output and the loss for each iteration. Therefore, to ensure a balance between accuracy and computational cost by tuning the tolerance. One of the limitations of neural networks is that if the training is done with higher accuracy with very relevant input data, and the test data are less relevant, the model often shows significantly lower accuracy for test data, which may result in wrong prediction less accurate result. However, as Neural ODE leverage the capabilities of the ODE solver to keep a balance between computation cost and accuracy, it is possible to train the model with high accuracy but switch that to a lower accuracy during test time.

3.2.4 Handling Irregular Sampling rate

Neural ODE is a suitable candidate for irregular sampling rates for continuous-time series and the characteristics above. Unlike RNN, the pioneer model for continuous-time series modelling, Neural ODE computes or learns the hidden dynamics of the system over time in the fashion of an Ordinary differential equation. On the other hand, RNN discretizes the entire time series of small and fixed-step. However, Latent ODE [3] leverages encoder and decoder based structure to simulate the continuous generation of observations. In this work, I proposed a continuous-time version of the two different RNN models, such as (i) Gated Recurrent Unit (GRU) and (ii) Long-Short Term Memory (LSTM), as these proposed models described in this chapter of my thesis focuses on integrating the input processing mechanism along with the hidden dynamics of the model. Therefore, instead of the encoder-decoder architecture of Latent ODE [3], my proposed model interwoven the input processing along with learning the hidden dynamics of the ODE directly in the sporadic incoming inputs.

3.2.5 Handle Missing Values

As discussed in chapter 2, Standard RNNs performs well when the time series has shorter gaps between consecutive observation in the continuous-time data and the data set. For RNNs to perform well, it is also essential that the missing observation values are very few. However, Neural ODEs, such as ODE-RNN and RNN-decay, do not depend on the time gap between two consecutive observations in the continuous-time data. Therefore, neural ODEs can predict the value and compute the hidden state at any desired time (t) and are suitable for interpolation tasks.

3.2.6 Dynamic depth of Neural ODE

Convolutional Neural Network (CNN) is from a Deep Neural Network (DNN) with a certain depth level in the families of neural networks. On the other hand, memory-based

RNN models have a long sequence of memory states representing their corresponding depth level. However, an ODE does not have any certain depth; therefore, defining the depth of Neural ODE is not straightforward. Often, the number of evaluations of the hidden state dynamics required during the ODE solution by the ODE solver is considered the depth of the model. However, as the number of evaluations of the hidden state is a variable dependent on the initial state or input, the depth of the Neural ODE mode can change depending on the input and the increasing complexity of the model throughout training.

All the characteristics mentioned above provide an efficient mechanism for learning the hidden dynamics of a system regardless of irregular or missing hidden dynamics. However, Neural ODE models need more freedom and stability to simulate the continuous generation of observations while learning the hidden dynamics of the system. In this paper, we focused on the structure of the Neural model and tried to increase stability in the structure by leveraging the structure of a much more stable neural network model.

The core motivation is to make the RNN models more continuous. Therefore, we can leverage the strength of RNN models for learning and the consciousness of the Neural ODE model. The strength of these two network families improves the performance of continuous-time series modelling using neural network models. Neural ODEs require constant memory, and the RNN model can ensure stability along with accuracy for time series generation, reduction and classification

3.3 Proposed Models

In a standard RNN cell either GRU or LSTM, the input (x_t), output (o_t) and hidden state (h_t) are controlled by gating units, i.e reset gate (r_t) and update gate (z_t) as shown by (3.3a) and (3.3b) respectively. Fig. 3.1 shows the architecture of the standard GRU [7] model. A single cell of the GRU model is connected with other cells sequentially. Each cell of GRU model contains a reset gate (r_t) and an update gate (z_t). The reset gate controls the amount of previous memory that needs to be forgotten, and the update gate controls the amount of previous memory that needs to be forwarded to the next state. Both gates reset gate (r_t) and update gate (z_t) contribute to the current state of the cell. These gates contribute to minimizing the vanishing gradient problem for the neural network. The GRU model does not wash all input at every stage. Rather only the irrelevant part of previous memory is forgotten, and the relevant part is forwarded to the next state. Therefore, each cell in standard RNN, uses the hidden state of previous cell (h_{t-1}) as input along with raw data input (x_t) at time t.

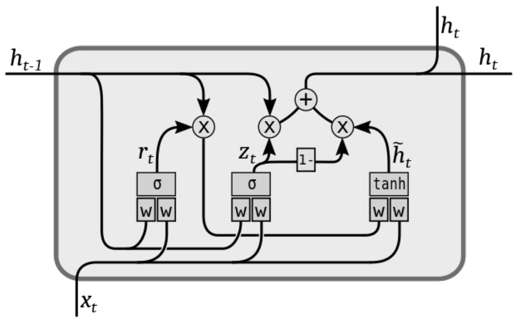


Figure 3.1: Architecture of GRU model

Different types of Neural ODE, such as Latent ODE and ODE-RNN, leverage the RNN model as encoder and decoder alongside the ODE solver. Fig. 3.2 shows that the Latent ODE, uses a ODE-RNN encoder. Besides encoding the input over time t, the encoder also runs backwards to produce an approximate posterior over the initial state. For example, based on the initial encoded value (z_0) for the ODE-Solver, it is possible to find the latent state (z_t) at any point of time (t) by solving the ODE initial-value problem. As the ODE solver solves an augmented ODE backwards in time as shown in Fig. 3.2, therefore, the model contains both the original state over time as x_t and the sensitivity of the loss with respect to the state along with the

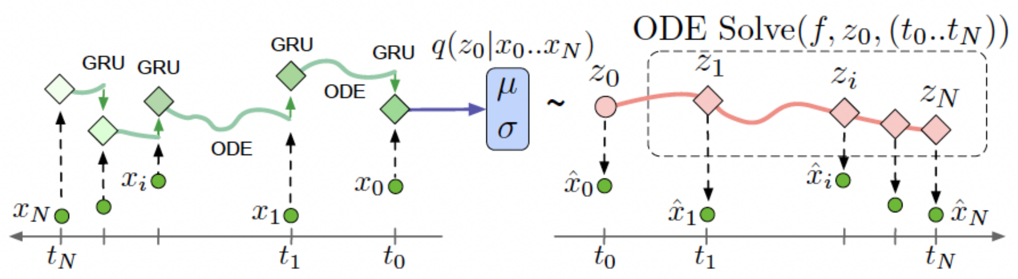


Figure 3.2: Architecture of Latent ODE [2]

derivative z_t with respect to time.

In this chapter, the existing RNN cell is redesigned as a function of ordinary differential equations of variable x over time t. Two different RNN models are proposed using the similar cell structure of GRU and LSTM. In these new RNN models, the hidden dynamics are computed using the derivative of the hidden state between consecutive observations. Fig. 3.3 shows that the proposed models progress over continuous time, where h_0 is the initial hidden state passed through an ordinary differential equation (ODE) that uses an update function for solving ODE similar to standard GRU or LSTM cell ($odeRNNCell$). Unlike Latent ODE shown in Fig 3.2, the proposed models do not have any encoder and decoder for encoding as well as decoding the hidden states (z_t) at any time t. Therefore, the structure of the proposed model is more straightforward than other variants of Neural ODEs. Instead of using separate RNN for input processing for the Neural ODEs, these proposed models use the RNN update and reset function as part of the ODE solver to update the hidden state (h_t) at any time t with only relevant information about the hidden dynamics of the system or the target ODE that can be interpreted as the ODE form of the system. In this section, both RNN structure GRU and LSTM are redesigned using the concept of Neural ODE, and new two different ODE-based RNN models are as follows:

1. ODE-GRU

2. ODE-LSTM

The proposed ORD-RNN model as shown in Fig 3.3, contains the original value of time series at any time such as y_t and the hidden state (h_t) similar to any RNN model. The ODE solver for the proposed ORD-RNN model levelrage RNN forward function , either LSTM or GRU to solve the ODE with respect to time t.

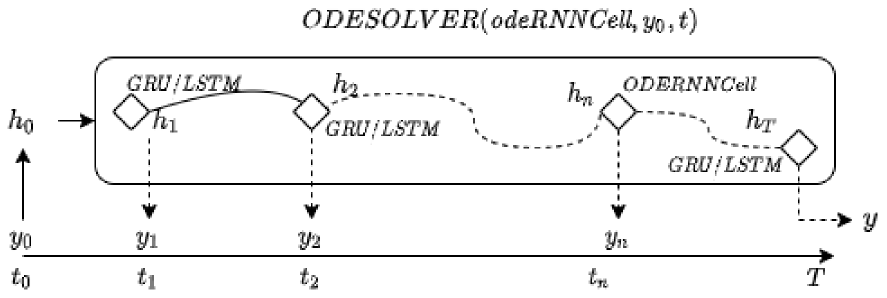


Figure 3.3: Architecture of ODE-RNN model

These two proposed models leverage the capabilities of ODE solvers. The final cell state y_t for the model can be computed using Eq. (3.2). In Eq. (3.2), *odeRNNCell* refers to an ODE based RNN cell, either GRU-ODE or LSTM-ODE.

$$y_t = \text{ODESOLVER}(\textit{odeRNNCell}, y_0, t) \tag{3.2a}$$

$$y_t = \text{ODESOLVER}(\textit{odeRNNCell}, \textit{tuple}(y_0, h_0), t) \tag{3.2b}$$

ODESOLVER can be any kind of ODE solver, e.g., an Euler, Heun, adjoint or implicit method. Models proposed in this section use a neural ordinary function *odeRNNCell* to compute change or derivative of the hidden dynamics at any time t. *odeRNNCell* is usually an initial value problem function as shown in (3.2a) which requires the initial observation y_0 and hidden state h_0 at time $t = t_0$ as initial input. In the case of (3.2a), the cell state at time t is computed based on initial time statet_0, and the hidden dynamics is considered as gradient dynamics and computed by the gradients update. *odeRNNCell*, can be either *odeGRU*, as shown in Algorithm 4 or *odeLSTM*, as shown in Algorithm 6. In other case as in Eq. (3.2b), the cell state or output at any time is computed based on the initial cell state (y_0) and hidden dynamics (h_0) at time t_0. In this implementation, the proposed neural network is initialized with initial hidden state (h_0), and only the initial observation value is passed as the input for the model.

3.3.1 Proposed ODE-GRU model

GRU model inspires the first model proposed in this thesis. The proposed ODE-GRU model is designed to propagate in time through the continuous-time state of the hidden dynamics of the system between consecutive observations. At the same time, the proposed ODE-GRUcan also updates the current hidden state of the model to incorporate the incoming observations at each time step t.

Algorithm 4 ODE-GRU ode solver function

```
Input:: initial hidden and cell state,time series and weight as well as bias parameters
1: procedure ODEGRU($states, t, parameters$)
2:     $x \leftarrow states[0]$
3:     $h \leftarrow states[1]$
4:     $r_t, z_t, \tilde{h}_t \leftarrow updateGRUFunction(x, h, parameters)$
5:     $h_t \leftarrow \tilde{h}_t(1 - z_t)$
6:     $o_t \leftarrow \sigma(W_o h_t + b_o)$
7:     return $\frac{do_t}{dt}, \frac{dh_t}{dt}$
```

Algorithm 4 considers states as the initial hidden state along with the observation value at initial time, $t = t_0$, t is the time series, parameters is a tuple of all weight parameters $(W_r, U_r, W_r, U_z, W_h, U_h)$ and bias parameters (b_r, b_z, b_h). *updateGRUFunction* takes the input and compute the traditional GRU update functions as shown in Eq. (3.3a) and (3.3b). This procedure described in algorithm 4 also allow end-to-end training of the dynamic system to minimize the loss with respect to the irregularly sampled observations.

$$r_t = \sigma\left(W_r x_t + U_r h_{t-1} + b_r\right) \tag{3.3a}$$

$$z_t = \sigma\left(W_z x_t + U_z h_{t-1} + b_r\right) \tag{3.3b}$$

$$\tilde{h}_t = \tanh\left(W_r x_t + U_r r_\tau + b_r\right) \tag{3.3c}$$

$$o_t = \sigma\left(W_o h_t + b_o\right) \tag{3.3d}$$

ODE-GRU pass the changes to the ODE solver and uses only the change in hidden state as (3.4a). The proposed RNN models do not carry hidden state of previous cell (h_{t-1}) as like standard RNN models. Only the derivative of hidden state over time($fracd\,(h_t)dt$) is progressed as a function of time .

$$\frac{dh_t}{dt} = \frac{d\tilde{h}_t}{dt}(1 - z_t) \tag{3.4a}$$

Algorithm 5 describes the complete flow for the training the ODE-GRU model. Here, in algorithm 5, *optimizer* and *lossFunction* can be used based on the complexity of the dataset. *GRADF* is the gradient function uses by the optimizer.

Algorithm 5 Train the ODE-GRU model

Input:: Initial condition (y_0 and h_0), continuous time series(t) and Initial parameter

```
1: procedure TRAIN($y_0$, $h_0$, target, t, initalParams)
2:     $input \leftarrow tuple(y_0, h_0)$
3:     $trainedParams \leftarrow$ optimizer (gradient(GRADF),
               $initalParams$, $iterations$,
               callback = GRADFUNCTION.backward())
4:     return $trainedParams$
   procedure GRADF($input, t, initalParams, target$)
2:     $prediction \leftarrow$ ODESOLVE( $odeGRU$, $input$, $t$,
                                    $initalParams$)
       $loss \leftarrow lossFunction(prediction, target)$
4:     return $loss$
```

Fig 3.4 shows that the ODESOLVER in the proposed ODE-GRU model leverage the reset gate (r_t) and update gate (z_t) of GRU model as described in Eq. (3.3a) and Eq. (3.3b) to determine the hidden state ($\tilde{h_t}$) and the output (o_t) of any time t. Finally, the hidden state and output state is considered as the value for input at the next step $(t+1)$. The proposed model can archive high performance with a simple architecture without using any secondary encoder and decoder in Latent ODE. The ODE solver uses the $ODEGRU$ function to solve the ODE that represents the system. Therefore, the solution to an ODE (y_t) is the hidden state between consecutive observations.

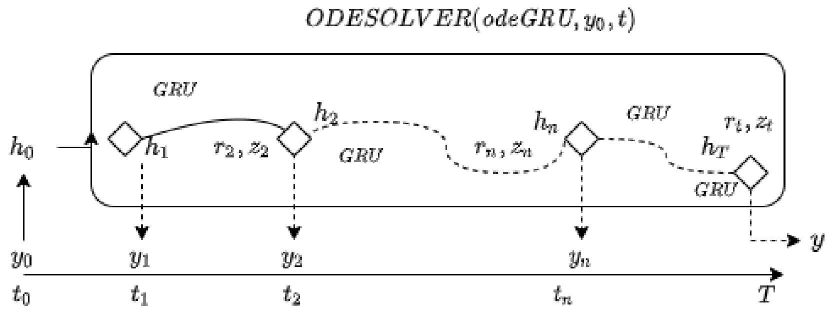


Figure 3.4: Archietcture of porposed ODE-GRU model

3.3.2 Proposed ODE-LSTM Model

Proposed ODE-LSTM model predicts the output (y_t) at any time t using *odeLSTM* ODE solver with the initial value of the observation (y_0) at time $t = t_0$. Therefore, instead of computing each state based on its previous state individually, the predicted output is generated as a function of time. (3.5) describes ODE Solvers, those that have been used for the ODE-LSTM model.

$$y_t = \text{ODESOLVER}(odeLSTM, tuple(y_0, h_0, c_0), t) \tag{3.5}$$

Here, *odeLSTM* is the function that constructs the dynamics of hidden as well as cell states and calls the ODE solver to compute all the gradients at once. The algorithm of *odeLSTM* is discussed in 6. This model uses the same input parameter as ODE-GRU, described in section 3.3.1, the initial observation value, (y_0) and the initial hidden state of the network h_0. Besides, this model also uses the initial cell state (c_0) as an input parameter. This proposed ODE solver function, as shown in Eq. 3.5 constructs the network as an initial value problem function. Therefore providing the initial condition is the only input required for the proposed model.

Algorithm 6 ODE-LSTM update function

```
Input:: initial hidden and cell state,time series and weight as well as bias parameters
1: procedure ODELSTM($states, t, param$)
2:     $x \leftarrow states[0]$
3:     $h \leftarrow states[1]$
4:     $c \leftarrow states[2]$
5:     $r_t, z_t, c_t, h_t, o_t \leftarrow updateLSTMGate(x, h, c, param)$
6:     $cell \leftarrow \frac{df_t}{dt} * c + \frac{di_t}{dt} * c_t$
7:     $hidden \leftarrow \frac{do_t}{dt} * \frac{dcell}{dt}$
8:     return $\frac{do_t}{dt}$, $hidden, cell$
```

Algorithm 6 uses *states* to contain the initial hidden state, initial cell state and the observation value at starting time, $t = t_0$, t is the time series, *param* is a tuple of weight parameters $(W_r, U_r, W_r, U_z, W_h, U_h)$ and bias parameters (b_r, b_z, b_h). *updateLSTMGate* takes the input and compute the traditional LSTM update functions as shown in Eq. 3.6b and 3.6c.

$$i_t = \sigma_i\left(x_t W_{xi} + h_{t-1} W_{hi} + w_{ci} \odot c_{t-1} + b_i\right) \tag{3.6a}$$

$$f_t = \sigma_f\left(x_t W_{xf} + h_{t-1} W_{hf} + w_{cf} \odot c_{t-1} + b_f\right) \tag{3.6b}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \sigma_c\left(x_t W_{xc} + h_{t-1} W_{hc} + b_c\right) \tag{3.6c}$$

$$o_t = \sigma_o\left(x_t W_{xo} + h_{t-1} W_{ho} + w_{co} \odot c_t + b_o\right) \tag{3.6d}$$

$$h_t = o_t \odot \sigma_h\left(c_t\right) \tag{3.6e}$$

Different update gate function such as input (i_t), forget(f_t) and output(o_t) use the typical sigmoidal non-linearities $(\sigma_i, \sigma_f, \sigma_o)$, on the other hand, cell (c_t) and hidden (h_t) activation functions uses tanh non-linearities. σ_c and σ_h along with weight parameters(W_{hi}, W_{hf}, W_{ho}, W_{xi}, W_{xf}, W_{xo}), connect the different inputs and gates with the memory cells and outputs, as well as biases (b_i, b_f, and b_o). Both biases and connection weights (w_{ci}, w_{cf}, and w_{co}) are optional for the update of input, forget, and output gates.

ODE-LSTM pass the changes in input, forget, hidden gate to the ODE solver and uses only the change in smaller fraction to compute the final output, hidden and cell state using (3.7b), (3.7c) and (3.7a):

$$c_t = \frac{df_t}{dt} * c_0 + \frac{di_t}{dt} * c_t \tag{3.7a}$$

$$\tilde{o}_t = \frac{do_t}{dt} \tag{3.7b}$$

$$h_t = \tilde{o}_t * \frac{dc_t}{dt} \tag{3.7c}$$

Algorithm 7 describes the complete flow for the training of the model using the proposed ODE-LSTM model. *odeLSTM* is computing all gradients at once for the model. The parameters are learned during training using the loss function *lossFunction*. The change in hidden data dynamics is computed and learned during the training session of the proposed model. Similar to algorithm 5, optimizer uses gradient function $(GRADF)$

3.4 Experiments

Several experiments are done in order to understand the dynamic characteristics of datasets. These experiments also show how the complexity of the data influences the proposed models. We have used different experiments to exhibit the performance of

Algorithm 7 Train the ODE-LSTM model

```
Input:: Initial conditions ($y_0$, $h_0$), input($t$), $initialParams$
1: procedure TRAIN($y_0$, $h_0$, $c_0$, $target$, $t$, $initalParams$)
2:     $input \leftarrow (y_0, h_0)$
3:     $trainedParam \leftarrow$ optimizer(gradient($GRADF$),
                 $initalParams$, $iterations$,
                 callback = GRADFUNCTION.backward())
4:     return $trainedParameters$
   procedure GRADF($input, t, initalParams, target$)
2:     $prediction \leftarrow ODESOLVE(odeLSTM, input, t,$
                                   $initalParams)$
       $loss \leftarrow lossFunction(prediction, target)$
4:     return $loss$
```

proposed models for different tasks as follows

- Continuous time series generation
- Continuous time series classification with irregular sampling rate

3.4.1 Continuous time series generation with Curve datasets

In order to understand the learning mechanism of the proposed models, two different curve time series, e.g., (i) eight-curve with low complexity and (ii) tri-knot with medium complexity, are chosen. Each of them is of different complexity. This experiment helps to understand the impact of time series flow and its complexity and sparseness on the learning of the proposed model. The learning vector for the tri-knot curve also demonstrates that the proposed model can learn itself comparatively faster than traditional LSTM and other ODE based RNN [2, 3]. For example, fig. 3.5 shows that the proposed ODE-GRU model needs only 20 iterations to identify the learning vector and the curve dynamics.

Fig. 3.6 shows the learning vector of the first and 20th iterations for eight curves. This demonstrates that the learning rate is swift, even for only 50 hidden dimensions. ODE-GRU and ODE-LSTM models can play a significant role in identifying the types

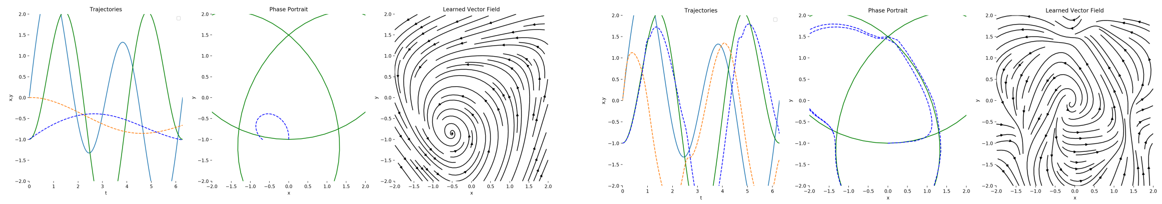


Figure 3.5: Learning Tri-Knot curve function at iteration 1(Left) and 20(Right)

of the dynamic lie in the corresponding time-series data. This experiment shows that these proposed models can learn time series data faster than traditional RNN cells with fewer parameters.

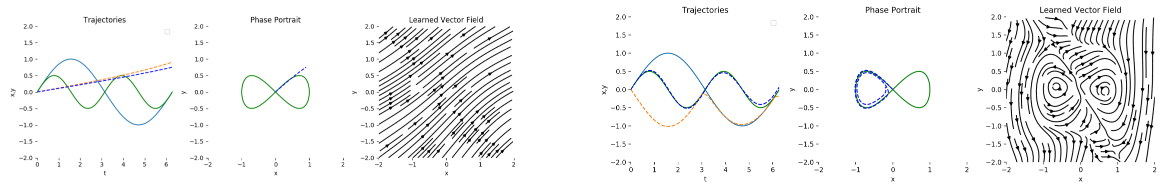


Figure 3.6: Learning eight curve function at iteration 1 (Left) and 20(Right)

Learning the curve path or trajectory for a different types of curves with proposed ODE-RNN models had some challenges. The proposed model is based on an ODE solver, which uses GRU or LSTM function to update the hidden state based on the initial state of the curve path at any time t. At the same time, the value of time and the time gap between two consecutive observations do not affect the model's training. As a result, the proposed model started its learning for the Tri-Knot curve considering the path as a sine wave. Therefore, we have leveraged the mini-batch training mechanism. The output of the ODE solver for each time point for one batch needs to be generated as a time series as well. The time for training then depends on the length of the time duration, but it depends on the number of observations during the entire time duration.

3.4.2 Continuous time series generation with Toy dataset

A simple time series with a variable sampling rate or frequency is used to learn the dynamics of the proposed model by mimicking a spiral ODE. The spiral ODE used in this test consists of 1000 spirals, and each spiral has 100 irregularly sampled time points. In [9], a simple Feed Forward Neural network (FFN) is used for ODE solver as shown in (3.2a).

```
def forward(self, t, y):
    return self.net(y**3)
```

$$pred_y = odeint(FFN, batch_{y0}, batch_t) \tag{3.8}$$

Fig. 3.7 shows the comparative result after the 25 iterations of [2] (left) and our proposed ODE-GRU (right). This demonstrates that the proposed ODE-GRU can reduce the training time significantly, which is one of the limitations of existing Neural ODE [2]. Also, for this experiment, we have used a dataset of only 500-time values ad 500-time points as a training dataset. Proposed Neural ODE-RNN models need a comparatively smaller dataset for training.

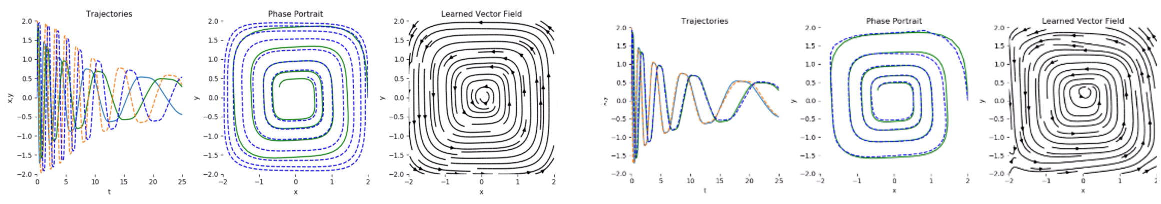
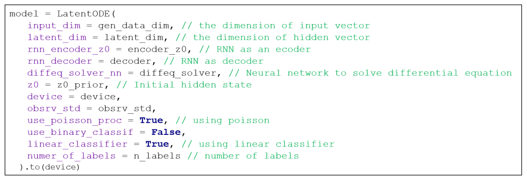


Figure 3.7: Result of ODE neural network (left) and the proposed ODE-GRU (right) after 25 iterations of training.

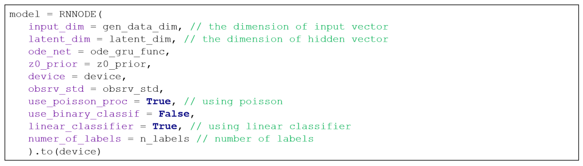
3.4.3 Continuous time series classification with Human Activity dataset

For this experiment, a comparative performance evaluation of proposed models and [3] is executed using the human activity recognition dataset [10] is used. This dataset consists of 6 different activities along with 12 features. After normalization, this dataset can be transferred to a time series of 6554 sequences at 211-time points. The configuration of Latent-ODE [3] for this comparative analysis is shown at Fig. 3.8a. Three different neural networks are used for Latent-ODE, i.e., encoder, differential equation solver, and decoder. On the other hand, the proposed neural networks (ODE-GRU and ODE-LSTM) only require a single recurrent neural network in order to compute the derivative of hidden dynamics, as shown in Fig. 3.8b. Proposed models need a single neural network which is used by a simple differential equation solver function as shown in Algorithm 4. The proposed models are elementary in structure in comparison to other neural ODE [2, 3]. Therefore, the training time is less than [3] and [2]. I have also compared the proposed model against standard GRU and LSTM. An important characteristic of the proposed models is that the difference in loss between consecutive iterations during the training process is comparatively lower than in other neural network models. This model only computes the derivative of hidden dynamic $\frac{d}{dt}h_t$ at any time t; therefore, the proposed models progress through iterations with small changes in loss value, as shown in Fig. 3.9.



```
model = LatentODE(
   input_dim = gen_data_dim, // the dimension of input vector
   latent_dim = latent_dim, // the dimension of hidden vector
   rnn_encoder_z0 = encoder_z0, // RNN as an ecoder
   rnn_decoder = decoder, // RNN as decoder
   diffeq_solver_nn = diffeq_solver, // Neural network to solve differential equation
   z0 = z0_prior, // Initial hidden state
   device = device,
   obsrv_std = obsrv_std,
   use_poisson_proc = True, // using poisson
   use_binary_classif = False,
   linear_classifier = True, // using linear classifier
   numer_of_labels = n_labels // number of labels
  ).to(device)
```

(a) Latent-ODE



```
model = RNNODE(
   input_dim = gen_data_dim, // the dimension of input vector
   latent_dim = latent_dim, // the dimension of hidden vector
   ode_net = ode_gru_func,
   z0_prior = z0_prior,
   device = device,
   obsrv_std = obsrv_std,
   use_poisson_proc = True, // using poisson
   use_binary_classif = False,
   linear_classifier = True, // using linear classifier
   numer_of_labels = n_labels // number of labels
   ).to(device)
```

(b) Proposed ODE-GRU/ODE-LSTM

Figure 3.8: Configuration for neural network used in Human activity experiment

3.4.4 Evaluation of proposed model

The significant contributions of these proposed models are as follows

1. *Simple architecture:* The architecture design of the proposed models is straightforward, it uses the update functions of standard RNN models and ODE differential equation solver ODESOLVER to design RNN models as a function of the

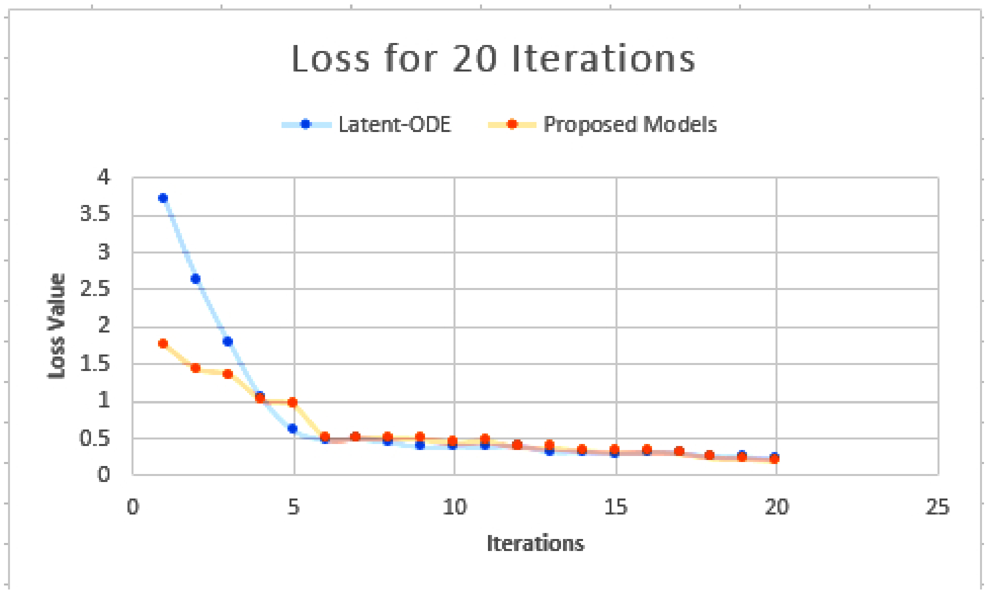


Figure 3.9: Loss value for 20 iteration for Latent-ODE and proposed models

continuous-time. As RNN models are already pioneering in terms of time series modelling, these proposed models leverage the strength of RNN and neural ODE in a single neural network. As shown in Eq (3.9a) and Eq (3.9b), Latent ODE has two building blocks, such as a ODESOLVER to solve the ODE and an RNNCell to update the hidden state of the neural network based on the solution of the ODESOLVER. On the other hand, the proposed model has only a ODESOLVER. The ODESOLVER of proposed models internally uses RNNCell as the function for ODESOLVER to solve the ODE. The ODE solution is then used as the updated hidden state of the system. Eq. (3.2a) shows that ODESOLVER itself can compute the updated hidden state of the system using the update functions of standard RNNcell, either GRU or LSTM.

$$h'_i = \text{ODESolve}\left(f_\theta, h_{i-1}, (t_{i-1}, t_i)\right) \tag{3.9a}$$

$$h_i = \text{RNNCell}\left(h'_i, x_i\right) \tag{3.9b}$$

2. *Data dynamics:* Due to continuous and straightforward characteristics, these models are very suitable for understanding the hidden dynamics of corresponding datasets. These models can compare the dynamics of different dynamics to understand the data. This feature can significantly contribute to the design of a data-driven system. These models can also participate in explainable machine learning.

3. *Fewer parameters:* In comparison to standard RNN cells and other neural ODEs, even fewer weight, as well as bias parameters, can achieve the result even faster for proposed models.

4. *Memory Usage:* Proposed models use the automatic gradient [11] to compute graphs for storing the previous state. However, not all previous states are relevant as it only focuses on the parameters of the model and the hidden dynamics. Based on the complexity of the dataset, the graph length can vary.

3.5 Conclusion

This work introduced a family of continuous RNN models enriched by ordinary differential equations (Neural ODEs). Unlike traditional RNN models, these proposed models do not need to discretize continuous time series into discrete-time sequences. In contrast to the generic RNN model, this proposed ODE-GRU and ODE-LSTM models can update the dynamics at irregularly sampled time points. Besides, the proposed ODE based two RNN models leverage the distinctive architecture of GRU and LSTM gating units to model data-driven, event-driven, asynchronously sampled input of continuous series. These ODE-GRU and ODE-LSTM models have significant advantages. Any large time series can be learned within a short time and with fewer parameters. ODE-GRU and ODE-LSTM models compute continuously, therefore it is updated at every time slot where there is a change $\frac{d}{dt}h_t > 0$. If there is no change in consecutive time slot, ODE solver would compute $\frac{d}{dt}h_t = 0$. As a result, no additional masking vector is required to distinguish missing values. The significant contribution of these proposed models is that they are effortless to compute with faster training and evaluation time. In this chapter's result evaluation for these proposed models, we only focus on time-series data with Gaussian observations as the dataset. Gaussian observations ensure an irregular sampling rate in the dataset. For the toy dataset experiment, we focus on the generative nature of the proposed ODE-RNN models. ODE-RNN model can act as a generative model and generate continuous time series. Human activity dataset, However, GRU-ODE-Bayes can also be extended to binomial and multinomial observations since the respective NegLL and KL-divergence are analytically tractable. The proposed technique allows the modelling of sporadic observations of both discrete and continuous variables.

3.6 References

[1] Daniel Neil, Michael Pfeiffer, and Shih-Chii Liu. Phased lstm: Accelerating recurrent network training for long or event-based sequences. In Advances in Neural Information Processing Systems, pages 3882–3890, 2016.

[2] Tian Qi Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. In Advances in Neural Information Processing Systems, pages 6572–6583, 2018.

[3] Yulia Rubanova, Tian Qi Chen, and David K Duvenaud. Latent ordinary differential equations for irregularly-sampled time series. In Advances in Neural Information Processing Systems, pages 5321–5331, 2019.

[4] Arthur E. Bryson, Jr. A steepest ascent method for solving optimum programming problems. Journal of Applied Mechanics, 29(2):247, 1962.

[5] Barak A. Pearlmutter. Learning state space trajectories in recurrent neural networks. Neural Computation, 1(2):263–9, 1989.

[6] Bo Chang, Lili Meng, Eldad Haber, Lars Ruthotto, David Begert, and Elliot Holtham. Reversible architectures for arbitrarily deep residual neural networks. In Thirty-Second AAAI Conference on Artificial Intelligence, 2018.

[7] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. arXiv preprint arXiv:1406.1078, 2014.

[8] Sepp Hochreiter and Jurgen Schmidhuber. Long short-term memory. Neural computation, 9(8):1735–1780, 1997.

[9] Ricky Chen. Differentiable ODE solvers with full GPU support and $O(1)$-memory backpropagation, December 2018. URL https://github.com/rtqichen/torchdiffeq.

[10] Davide Anguita, Alessandro Ghio, Luca Oneto, Xavier Parra, and Jorge Luis Reyes-Ortiz. A public domain dataset for human activity recognition using smartphones. In Esann, 2013.

[11] Barak A Pearlmutter. Gradient calculations for dynamic recurrent neural networks: A survey. IEEE Transactions on Neural networks, 6(5):1212–1228, 1995.

CHAPTER 4

NEURAL ODES FOR INFORMATIVE MISSINGESS IN MULTIVARIATE TIME SERIES

This chapter is an expanded version of of a paper published in: 2020 31st Irish Signals and Systems Conference (ISSC)

Informative missingness is unavoidable in the case of processing of continuous-time series, where the value for one or more observations at different time points is missing. Such missing observations are significant limitations of time series processing using deep learning. Real-time applications, e.g., sensor data, healthcare, weather, generate continuous data, and informative missingness is common in these datasets. These datasets often consist of multiple variables, and often there are missing values for one or many of these variables. This characteristic makes time series prediction more challenging, and the impact of missing observations on the accuracy of the final output can be significant. A recent novel deep learning model called GRU-D is one early attempt to address informative missingness in time series data. On the other hand, a new neural network called Neural ODEs (Ordinary Differential Equations) is natural and efficient for processing time-series, continuous data. This section proposes a deep learning model that leverages the effective imputation of GRU-D and the temporal continuity of Neural ODEs. A time series classification task performed on the PhysioNet dataset demonstrates the performance of this architecture.

4.1 Introduction

Continuous-time series usually consists of data collected at different time points at different sampling frequencies. Based on the sampling frequency and data availability, some time points in the series can have missing observations for many of the time points. A recent work [1] demonstrates some findings using the MIMIC-III datasets [2] which reflects the significance of missing patterns in time series prediction tasks. Suppose the value for certain variables in a multivariate time series is missing for a significant time. In that case, the impact of corresponding input variables fades away over time and can result in inaccurate predictions for output. Over time, various experiments for data processing with missing values tried to overcome the imposed challenges. These techniques are suitable for small as well as simple time series. As the complexity, dynamics, length, sampling frequencies, and the number of variable increases, these techniques become irrelevant. Some of the methods are as follows

- Omit the missing data and perform the task on only available time-series observations. This approach helps to ignore the unavailable data. However, if the missing rate is high and a significant amount of data is ignored, this solution often results in an inaccurate outcome and misguiding prediction.

- Use data imputation to fill out the missing observation with substitutes value. There are several imputation techniques to determine substitute value. The limitation of this approach is that its only suitable for simple time series. It is often too hard to find the correct substitute values for complex time series. Therefore, for complex time series, rather than using a single data imputation method, multiple imputation methods are often used combined to deal with the complexity of data series and reduce the uncertainty.

- Another practice is to apply data imputation multiple times iteratively with a target to reach an average value for the missing observations. However, data imputation is only effective for simple data series with fixed missing rates and time series lengths. In reality, time series are often variable in length, and missing observations occur in entirely random order.

Among all different Neural network families, Recurrent Neural Networks (RNN) have shown significant efficiency in solving missing patterns in time series with different gating units and their capacity to store memories [3]. Various time series tasks such as classification, prediction, and generation are usually solved using RNN models. Different gating units for RNNs (e.g., Long Short-Term Memory (LSTM) [4], Gated Recurrent Units (GRU) [5], and GRU-D [1]) typically consider time series as a dynamical system of the discrete and fixed time step. Theoretically, a fixed step can be enough for time series modelling if it is too small. However, real-world time series data are usually sampled at an irregular rate. Therefore, a short time step is necessary for some applications such as sensors to cope with the higher data sampling frequencies. On the other hand, patients' health records need to be sampled at a higher time step, as the time gap between two consecutive patient visits can be very long due to the irregular data sampling rate and variable length. Therefore, it requires a structured and dynamic model to defeat the uncertainty of informative missingness.

[1] try to handle missing pattern in continuous time series with data imputation. GRU-D [1] successfully exploits the strength of RNN models for time series to capture the long term dependencies in multivariate time series. Another recent family of neural networks, Neural Ordinary Differential Equations (ODE-NN) [6–8] helps to solve time series by using black-box differential equation solver. This kind of neural network leverages the initial value problem to compute the hidden dynamics (f) as a function of continuous time at any time t. As shown in (4.1), Neural ODE can compute the hidden state of time t (h_t) from the initial hidden state(h_0) and the hyperparameters

(θ_{t-1}) update over time. Here $t \in \{0, \ldots, T\}$ and $h_t \in \mathbb{R}$.

$$h_t = h_{t-1} + f(h_0, \theta_{t-1}) \tag{4.1}$$

In this section, two different neural network models are introduced. I call them Continuous GRU-D and Extended ODE-GRU-D. Both models leverage the differential equation solver to compute the model's hidden dynamics. They use differential equation solver *ODESolve* to impute data, where the missing observations of variables are replaced by the derivative of the value of available observations of corresponding variables. Over time, the decay in hidden dynamics and input significantly impact the final output of multivariate time series. This work computes the decay rate as the derivatives of time (t). Therefore, the decay rate can control the gradient optimization of the model over time. The proposed Continuous GRU-D model uses the GRU-D model's hidden dynamics as continuous-time dynamics using a differential equation solver. However, the Extended ODE-GRU-D model computes the decay rate in addition to the continuous hidden dynamics of the GRU-D model. For both models, the time series is a function of time t rather than a discrete sequence in this work. The Extended ODE-GRU-D model also shows an efficient way to generate hidden and input decay rates based on the data dynamics. Experiment on the PhysioNet dataset demonstrates that the proposed models outperformed the existing GRU-D model in the time series classification task. These experiments show that ODE based neural network model can successfully solve the informative missingness. The main goal of this work is to use an ODE solver as a black box ODE solver and compute the gradient for the optimizer, which is in charge of optimizing the parameters of the neural network model.

4.2 State of the Art

A multivariate time series (X_t) with D variables and of T length can be described as in (4.2). Time series (X_t) can have missing observation. Any observation for time series (X_t) is $x_t \in \mathbb{R}^D$ which represents t-th observation of all variables and x_t^d represents the value of the d-th variable at time t.

$$X_t = x_1, x_2, \ldots, x_T{}^D \in \mathbb{R}^{T \times D} \tag{4.2}$$

Fig. 4.1 depicts the problem domain. Here X is a continuous multivariate time series with some missing values. The masking vector M identifies missing observations. The timestamp vector S records the time of the t-th observation of each variable—the time interval vector Δ measures the duration of unavailability. The strength of the existing GRU-D implementation is that it uses the masking and time interval vectors to characterize the missing pattern rather than adopting the traditional missing-completely-at-random model. This approach helps to quantify the influence of missing observations and adapt accordingly.

X: Input time series (2 variables);
s: Timestamps for ***X***;

M: Masking for ***X***;
Δ: Time interval for ***X***.

$$\boldsymbol{X} = \begin{bmatrix} 47 & 49 & NA & 40 & NA & 43 & 55 \\ NA & 15 & 14 & NA & NA & NA & 15 \end{bmatrix} \qquad \boldsymbol{M} = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\boldsymbol{s} = \begin{bmatrix} 0 & 0.1 & 0.6 & 1.6 & 2.2 & 2.5 & 3.1 \end{bmatrix} \qquad \boldsymbol{\Delta} = \begin{bmatrix} 0.0 & 0.1 & 0.5 & 1.5 & 0.6 & 0.9 & 0.6 \\ 0.0 & 0.1 & 0.5 & 1.0 & 1.6 & 1.9 & 2.5 \end{bmatrix}$$

Figure 4.1: Continuous time series with missing values, from [1]

4.3 Existing GRU-D implementation

The main contribution of the GRU-D model is that it can identify the long term dependencies and missing functional patterns in data. It can utilize long term temporal missing patterns in time series. Generally, GRU-D outperforms the other two RNNs (GRU and LSTM) in continuous time series prediction. The performance evaluation and comparison described in [1] shows that GRU-D scored 2.5% higher accuracy than other RNN implementations. Using the GRU-D model can predict time series without relying on previous missing observations in the time series. As a result, this model achieves higher accuracy within less time. The core components of GRU-D are as follows:

Masking Vector

A masking vector (M) $m_t \in [0,1]^D$ indicates either the observation is available with value 1 or missing with value 0 at any time step t as shown in Eq. (4.3).

$$m_t^d = \begin{cases} 0 & \text{if } x_t^d \text{ is missing} \\ 1 & \text{if } x_t^d \text{ is available} \end{cases} \tag{4.3}$$

Time Interval Vector

This vector computes the time interval ${\delta_t}^d$ for each d-th variable since its last observation. If the first observation for d-th variable is measured at time $s_0 = 0$, then the time interval can be measured by Eq. (4.4).

$$\delta_t^d = \begin{cases} s_t - s_{t-1} + \delta_{t-1}^d & t > 1, m_{t-1}^d = 0 \\ s_t - s_{t-1} & t > 1, m_{t-1}^d = 1 \\ 0 & \text{otherwise} \end{cases} \tag{4.4}$$

Data Imputation Value

For replacing the missing data, three different approaches, GRU-Mean, GRU-Froward and GRU-Simple, are used in the GRU-D model [1].

GRU-Mean

Each missing observation is replaced with the mean (x_t^d) of the individual variable across the training examples. Both training and testing datasets is processed to compute the dynamics ($\tilde{x}^d$) of GRU-Mean as shown in (4.6). Here, N is the length of time series.

$$x_t^d \leftarrow m_t^d x_t^d + \left(1 - m_t^d\right) \tilde{x}^d \tag{4.5}$$

$$\text{Where, } \tilde{x}^d = \sum_{n=1}^{N} \sum_{t=1}^{T_n} m_{t,n}^d x_{t,n}^d / \sum_{n=1}^{N} \sum_{t=1}^{T_n} m_{t,n}^d \tag{4.6}$$

GRU-Forward

Missing value is replaced with last available observations as shown in (4.7) . Here $t' < t$ is the last time when the d-th variable was observed.

$$x_t^d \leftarrow m_t^d x_t^d + \left(1 - m_t^d\right) x_t^d \tag{4.7}$$

GRU-Simple

This approach identifies the missing variable along with the duration of informative missingness by concatenating the measurement, masking and time interval vectors as shown in (4.8).

$$x_t^{(n)} \leftarrow \left[x_t^{(n)}; m_t^{(n)}; \delta_t^{(n)}\right] \tag{4.8}$$

Hidden State Decay

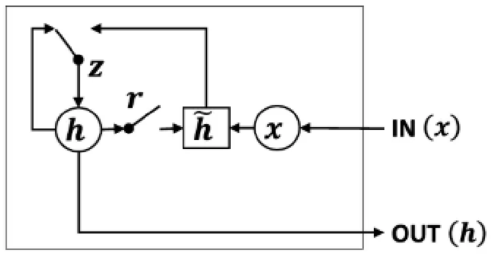
The importance of decay rate is that it can control the decay mechanism of the model considering underlying properties associated with each variable. As a result, the individual variable of a multivariate time series can influence the prediction based on its actual weight of decay. The decay rate (γ_t) is modelled as Eq. (4.9) with its associated weight (W_γ) and bias parameter(b_γ). The negative rectifier force the decay rate to decrease monotonically between ranges [0,1].

$$\gamma_t = \exp\left\{-\max\left(0, W_\gamma \delta_t + b_\gamma\right)\right\} \tag{4.9}$$

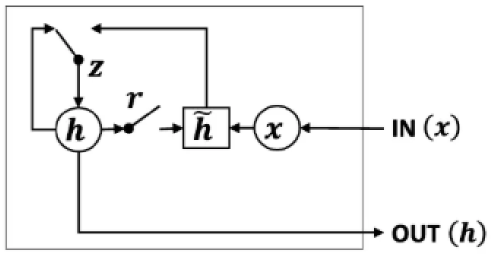
There are two different decay rate, hidden decay rate (γ_{h_t}) and input decay rate (γ_{x_t}). Hidden decay rate controls the decrease in decays of the previous hidden state h_{t-1} before computing current hidden state h_t as like Eq. (4.10). Here, $\hat{h}_{t-1}$ represents a state between observations, but it has less control over the raw input variable. Therefore, a second decay rate, the input decay rate, directly controls the decay of raw input variables.

$$\hat{h}_{t-1} = \gamma_{h_t} \odot h_{t-1} \tag{4.10}$$

As shown in Figs. 4.3 and 4.3, the main difference between traditional GRU and GRU-D is that it uses two different decay mechanisms hidden state and raw input. Using additional hidden state decay helps to capture the complete missing patterns.



(a) GRU Model Cell



(b) GRU-D Model Cell [1]

Figure 4.2: GRU vs GRU-D Model Cells

The update functions for different gate units and intermediate states of GRU-D are explained by Eq. (4.11a) (Reset Gate), (4.11b) (Update Gate), (4.11c) (Intermediate state), and Eq. (4.11d) (Hidden state). The typical GRU cell update functions [5] are modified in GRU-D. The input (x_t) and hidden (h_t) vectors are replaced by $\hat{y}_t$ and $\hat{h}_t$ respectively. Besides, a new set of parameter vectors, V_r, V_z and V are introduced for mask vector m_t.

$$r_t = \sigma(W_r \hat{x}_t + U_r \hat{h}_{t-1} + V_r m_t + b_r) \tag{4.11a}$$

$$z_t = \sigma(W_z \hat{x}_t + U_z \hat{h}_{t-1} + V_z m_t + b_z) \tag{4.11b}$$

$$\tilde{h}_t = \tanh(W \hat{x}_t + U(r_t \odot \hat{h}_{t-1}) + V m_t + b) \tag{4.11c}$$

$$h_t = (1 - z_t) \odot \hat{h}_{t-1} + z_t \odot \tilde{h}_t \tag{4.11d}$$

The neural network architecture of GRU-D shown in Fig. 4.3 shows that the input variable (x_t), masking vector (m_t) and time interval vector (δ_t) are used as input for each cell. The GRU-D cell Fig. 4.3 implement the GRU-Mean data imputation method to overcome the informative missingness problem. Different decay rates (γ_{h_t} and γ_{x_t}) shape the input and hidden state of the GRU-D cell. Finally, the output (y_t) for a GRU-D is the state of the GRU-D cell at time t.

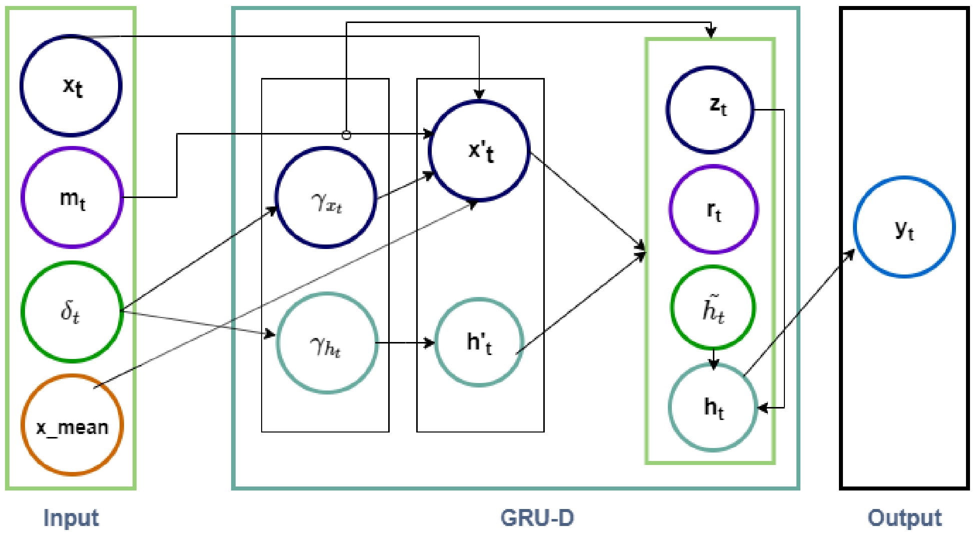


Figure 4.3: Architecture of GRU-D

4.3.1 ODE-RNN

This model leverages Neural Ordinary Differential Equation [9]. This neural network defines the hidden state between observations as the solution of an Ordinary Differential Equation (ODE) as shown in Eq. (4.12a)

$$h'_t = ODESolve(f_\theta, h_0, (t-1, t)) \tag{4.12a}$$

$$h_t = RNNCell(h'_t, x_t) \tag{4.12b}$$

Here, the function f_θ describes the dynamics of the hidden state using a neural network with parameters θ. The previous hidden state at time $t-1$ can be computed using a differential equation solver. ODE-RNN uses a standard RNN update function as shown in (4.12b) to compute current hidden state with input the intermediate state h'_t. The hidden state can be computed without depending on the time interval implicitly. The hidden state is defined by an ODE solver and later updated by another RNN network at each observation. ODE-RNN [9] is an auto-regressive model which makes one step ahead of predictions based on the previous observation. This model consists of three different neural networks: (i) an encoder, (ii) a decoder and ODE solver, as shown in Fig. 4.4.

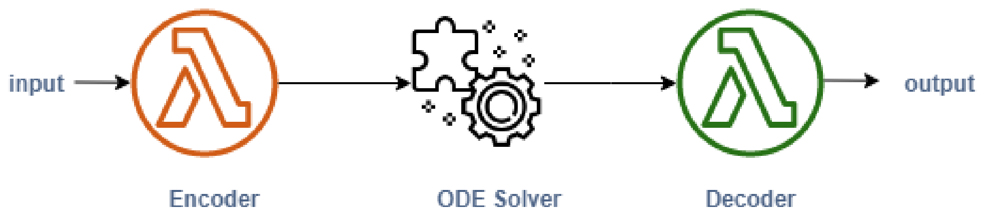


Figure 4.4: Architecture of ODE-RNN

4.3.2 Latent-ODE

latent-ODE [9] is a variational auto-encoder based continuous-time model. Both training and prediction use auto-encoder. ODE-RNN is the encoder for Latent-ODE.

This model can be characterized as encoder-decoder or fully ODE based sequence-to-sequence architecture for neural network model. Initial value problem is used to compute the latent state (z_t) at any time t. A neural network (g) generates the final hidden state of the ODE-RNN encoder as the mean and stranded deviation of the initial latent state z_0 as shown in (4.13a). To get the approximate posterior, both the encoder and decoder are trained jointly. The posterior distribution over latent states is a function of the final hidden state as shown in (4.13b). This indicates the explicit uncertainty not available in traditional RNN or ODE-RNNs.

$$\mu_{z_0}, \sigma_{z_0} = g(\text{ODE-RNN}_\phi(\{y_i, t_i\}_{i=0}^{N})) \tag{4.13a}$$

$$q(z_0|) = \eta(\mu_{z_0}, \sigma_{z_0}) \tag{4.13b}$$

The significant benefit of latent-ODE is that the hidden dynamics of the system and distribution of observations get decoupled from each other. As a result, each hidden state at any time t can be computed independently. In addition, the posterior distribution over latent states can measure uncertainty which is a unique feature for latent-ODE.

4.4 Proposed Model Design

Two different models are proposed in this section to solve the informative missingness of multivariate continuous time series, as shown in Fig. 4.1.

4.4.1 Continuous GRU-D

This model leverages an ODE solver in order to compute the hidden dynamics of continuous time series at any time t. Similar to [1], the hidden decay rate (γ_{h_t}) and input decay rate (γ_{x_t}) are computed using Eq. (4.9). These decay rates are used as parameters for the network. However, instead of computing the hidden state as a dynamical system of discrete sequential value, this proposed model uses the initial value problem to calculate the continuous dynamics using derivatives generated by ODE solver. Therefore, the value of d-th variable at time t (y_t) is computed using Eq. (4.14). Eq. (4.8) shows that y_t is a vector concatenating the value of last available observation (x_t), masking vector (m_t) value and hidden state value (h_t) at time t. The hidden dynamics y_0 is the initial value of the observation vector, $y_0^n \leftarrow [x_0^n; m_0^n; h_0^n]$. Here n is the number of variable and y_0 is an $n \times 3$ vector. The resulting vector y_t provides the value of hidden state at time t and observation value (y_t) of n variables at time t is $y_t \leftarrow [x_t^n; m_t^n; h_t^n]$.

$$(y_1, y_2, \ldots, y_t) = ODESolve(\mathscr{N}, y_0, T) \tag{4.14}$$

Here $\mathscr{N}$ is a neural network similar to the update functions of existing GRU-D model cell, described in Eq. (4.11a), Eq. (4.11b), Eq. (4.11c), and Eq. (4.11d). An ODE neural network, as shown in Fig. 4.5 is used to compute the update in hidden state and generate the target output. The input for GRU-D and ODE-GRU-D are same as shown in Fig. 4.5. The proposed ODE-GRU-D cell in Fig. 4.5 shows that instead of computing the intermediate hidden state value using the decay rate and imputation method, the proposed ODE-GRU-D leverage a neural network ($\mathscr{N}$) to compute the intermediate hidden and cell states. The neural network ($\mathscr{N}$) is a Neural ODE model that takes the intermediate input variable (x'_t), masking vector(m'_t) and hidden vector (h'_t) after the decay mechanism of GRU-D is applied on the original value of input; and generate the final value for intermediate hidden and cell state by solves the ODE derivative of them using a ODE solver (f). The ODE solver (f) then generates the hidden state (h_t). Finally, the output cell state (y_t) is derived from the hidden state (h_t) is similar fashion as GRU-D.

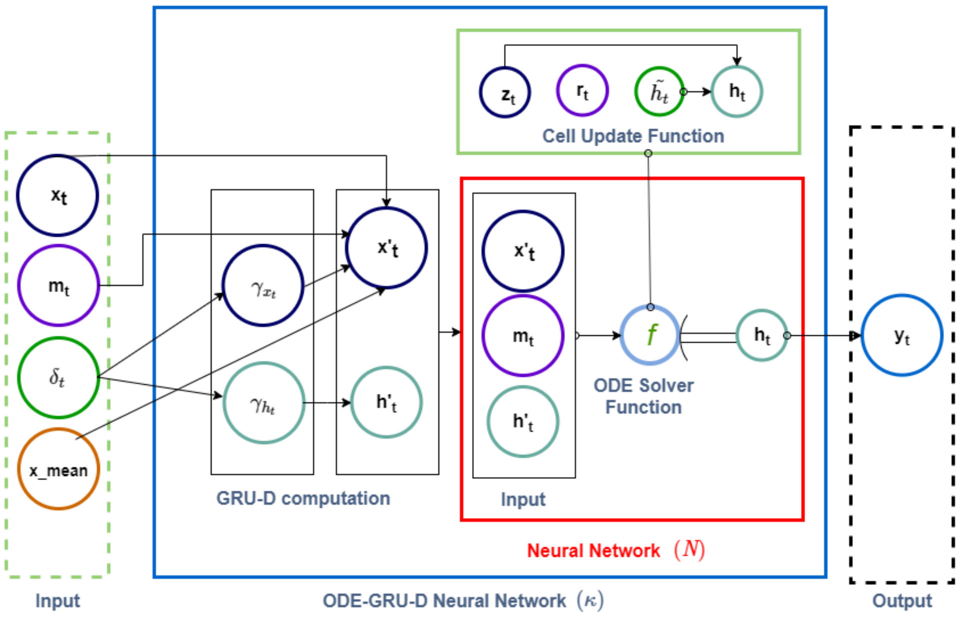


Figure 4.5: Architecture of ODE-GRU-D

This model leverage the ODE solver and uses the update functions as described in Eq. (4.11a), Eq. (4.11b), Eq. (4.11c) and Eq. (4.11d). No additional training of encoder or decoder, unlike Neural ODE in [6] or ODE-RNN in [9], is required in this proposed model. ODE-RNN [9] separately generates the hidden dynamics using a Neural ODE [6] and then computed hidden state is updated using standard RNN cell as shown in (4.1). However, proposed Continuous GRU-D model continuously update the hidden dynamics using a Neural ODE network ($\mathscr{N}$) which use the update functions of GRU-D. Eq. (4.15) shows that the proposed neural network (κ) takes the same inputs (X, M, Δ, T) similar to GRU-D as shown in Fig. 4.1. In addition to these inputs, Continuous GRU-D model has additional components, i.e. (i) an ODE based neural network ($\mathscr{N}$) and (ii) a loss function ($\mathscr{L}$) which helps the optimizer to leverage the dynamics of ODE in order to compute the update of hidden state. As shown in Eq. (4.15), the time interval between two consecutive observations is not considered in hidden dynamics computation. Therefore, the length of time interval is irrelevant and the proposed model works for both high as well as low sampling frequencies.

$$y_t = \kappa(X, M, T, \mathscr{L}, \mathscr{N}) \tag{4.15}$$

Algorithm 8 shows that an ODE based neural network *(f)* updates the hidden state. Here, *(f)* is the update function that constructs the dynamics of hidden as well as derivative of the cell states and call the ODE solver to update all the parameters for optimizer at once. Algorithm 9 describes the steps for computing the derivative of hidden dynamics (dy). The parameters of this neural network are optimized based on the output of loss function ($\mathscr{L}$). The input for $\gamma_{x_t}, \gamma_{h_t}$ are computed using the existing GRU-D computation described in Eq. (4.9). Similarly, x_t^d and $\tilde{h}_t$ are computed using Eq. (4.7) and Eq. (4.15). Here $\mathscr{N}$ is a neural network which uses x_t^d and $\hat{h}_t$ as input for a ODE based implementation of GRU-D cell update function. T is the time series.

Algorithm 8 Compute the state of an ODE-GRU-D cell

```
procedure $\mathscr{N}({x_t}^d, \tilde{h}_t, T, hiddenDynamics)$
    $y_0 \leftarrow tuple(x_0^d, m_0, \hat{h}_0)$
    $y_t, \frac{\delta L}{\delta \theta} \leftarrow ODESolve(f, y_0, T, h, \theta)$
    return $y_t$
```

Algorithm 9 uses the same update functions as [1] to compute the continuous hidden dynamics of the neural network.

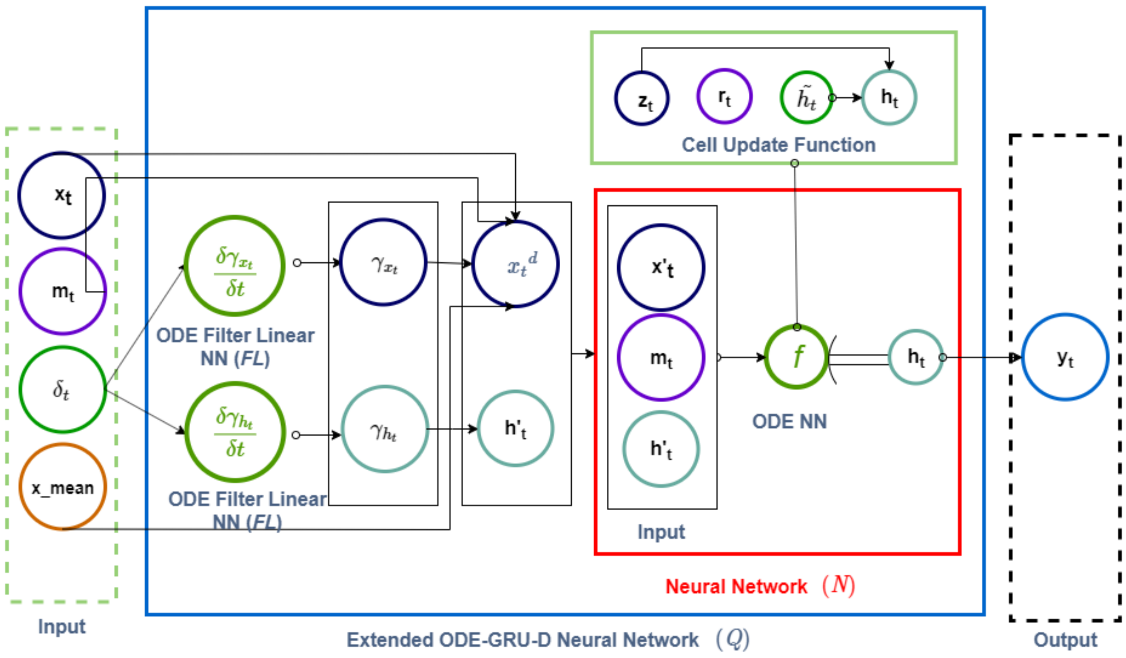
Algorithm 9 Update the derivative, the hidden state, and parameters for the optimizer

```
procedure $\mathscr{N}(y, T, \theta)$
    $x, h, m \leftarrow y$
    Compute $r_t$ using (4.11a)
    Compute $z_t$ using (4.11b)
    Compute $\tilde{h}_t$ using (4.11c)
    $h \leftarrow (h - \tilde{h}_t) * z_t$
    $y_t \leftarrow tuple(x, h, m)$
    return $y_t$
```

4.4.2 Extended ODE-GRU-D

The second proposed model ($\mathscr{Q}$) characterized the missing pattern of the time series using an Adjoint ODE solver. Fig. 4.6 shows the architecture of the proposed Extended ODE-GRU-D neural network model. The input for Extended ODE-GRU-D and GRU-D are same as shown Fig. 4.6. Similar to ODE-GRU-D (Fig. 4.5), Extended ODE-GRU-D also leverage the neural network ($\mathscr{N}$) to compute the intermediate states after the decay mechanism is applied to the original input to the Extended ODE-GRU-D cell (the blue box in Fig. 4.6). For Extended ODE-GRU-D, the decay mechanism is different than GRU-D or ODE-GRU-D models. Instead of using a static decay parameter in the GRU-D cell, we used a ODE Filter Linear neural network (FL) as the controller for the decay rates. Fig. 4.6 shows that two FL models are controlling the decay rate for the two different decay rate controller parameter, as a result, we do not need any explicit imputation method or manual control over the decay rate. FL model is learning the value for decay parameterise and updating it during training. in the Finally, the output cell state (y_t) is derived from the hidden state (h_t) is similar fashion as GRU-D.

in Existing GRU-D model computes decay rate as shown in (4.9), where W_γ and b_γ are parameters of the same GRU model. They are trained jointly with all the other parameters. However, if the dynamics of the decay rate can be computed as the derivative original input Δ, for time, it can provide the accurate value of the decay rate over time. Therefore, in this proposed model, a Filter Linear neural network FL is used to compute the derivative of the decay rate over time. W_γ and b_γ are parameters of FL, and they are trained separately to compute the derivative of decay rate ($\frac{\partial \gamma}{\partial t}$) as shown in (4.16a). ODE solver solves the neural network (FL) concerning Δ over time to compute the derivative. This derivative is used for computing the decay rate as shown in (4.16b).



x_t
m_t
δ_t
x_mean
Input
$\frac{\delta\gamma_{x_t}}{\delta t}$
ODE Filter Linear NN (FL)
$\frac{\delta\gamma_{h_t}}{\delta t}$
ODE Filter Linear NN (FL)
γ_{x_t}
γ_{h_t}
${x_t}^d$
h'$_t$
z_t
r_t
$\tilde{h}_t$
h_t
Cell Update Function
x'$_t$
m_t
h'$_t$
Input
f
ODE NN
h_t
Neural Network $(\mathcal{N})$
Extended ODE-GRU-D Neural Network $(\mathcal{Q})$
y_t
Output

Figure 4.6: Architecture of Extended ODE-GRU-D Model

Fig. 4.6 shows that two ODE Solver using Filter Linear neural network as function is used in proposed model to compute hidden state decay rate and input decay rate. The third ODE Solver compute the update in hidden state of the model.

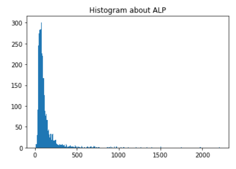
$$\frac{\delta\gamma}{\delta t} = ODESolve(FL, M, h_0, T, \mathscr{L}, \mathscr{N}) \tag{4.16a}$$

$$\gamma_t = \exp\left\{-\max\left(0, \frac{\delta\gamma}{\delta t}\right)\right\} \tag{4.16b}$$

This γ_t from (4.16b) compute $x_t^d \in X^D$ using (4.17).

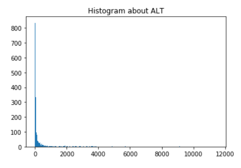
$$x_t^d \leftarrow m_t^d x_t^d + \left(\gamma_{x_t^d} x_t^d + \left(1 - \gamma_{x_t^d}\right)\right) x_{mean_t}^d \tag{4.17}$$

$X^D = \{x_0^d, x_1^d, ..., x_T^D\}$ is used as the input for the proposed Extended ODE-GRU-D model. Similar to proposed ODE-GRU-D described in §4.4.1, Extended ODE-GRU-D also uses ODE based neural network $(\mathscr{N})$ to compute the update in hidden dynamics



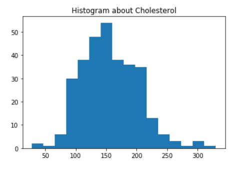
```
count: 3092, min: 12.0, max: 2205.0
mean: 116.77328589909443, median: 82.0,
```

(a) ALP



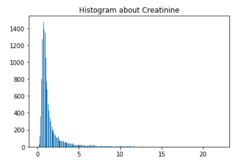
```
count: 3177, min: 1.0, max: 11470.0
mean: 394.4749763928234, median: 43.0
```

(b) ALT



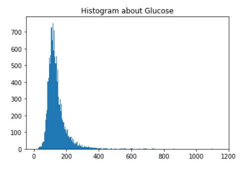
```
count: 315, min: 28.0, max: 330.0
mean: 156.51746031746032, median: 152.0
```

(c) Cholesterol



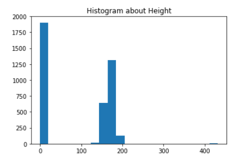
```
count: 13983, min: 0.1, max: 22.1
mean: 1.5049774726453549, median: 1.0
```

(d) Creatinine



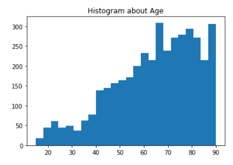
```
count: 13021, min: 10.0, max: 1143.0
mean: 141.5095614776131, median: 127.0
```

(e) Glucose



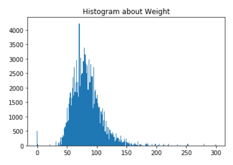
```
count: 4000, min: -1.0, max: 431.8
mean: 88.919475, median: 152.4
```

(f) Height



```
count: 4000, min: 15.0, max: 90.0
mean: 64.2475, median: 67.0,
```

(g) Age



```
count: 129165, min: -1.0, max: 300.0
mean: 83.38661525955173, median: 80.6
```

(h) Weight

Figure 4.7: Different parameters of the PhysioNet Challenges 2012 dataset

of the continuous time series as shown in (4.18). Here, h_0 is the initial hidden state.

$$y_t = \mathscr{Q}(X^D, M, h_0, T, \mathscr{L}, \mathscr{N}) \tag{4.18}$$

Proposed Extended ODE-GRU-D exploits ODE solver to understand the decay dynamics of the time series; therefore, controlling change in the decay rate is more accurate than the existing GRU-D model. this model learns the decay mechanism for raw input as well as the hidden dynamics of the continuous-time series. The main advantages of these proposed models are that they naturally handle the decay mechanism and the gap between consecutive observations.

4.5 Results

As neural ODE is used to compute the hidden state at any time (t), no additional data processing is required for these two proposed neural network models. Furthermore, unlike ODE-RNN [9], the proposed models use a single neural network. Therefore, it does not require a separate encoder and decoder.

4.5.1 Dataset and Task Description

For demonstrating the performance evaluation of proposed models, we use the PhysioNet Challenges dataset [10], a collection of multivariate time series with missing observations. It contains 8000 intensive care unit records (ICU), each a time series of 48 hours with 33 parameters. Fig. 4.7 shows the histogram for some of the parameters, i.e., ALP, glucose, age, height, weight, etc. This experiment uses Training Set A for training as the output of this set is available only. As in the original section, two prediction tasks below are attempted using this dataset.

Mortality task (Binary classification problem)

Predict a patient's death in the hospital. There are 554 patients with positive mortality labels. For this task, reference models are trained using 10% data (5898, 30) for test and 90% data for training (53078, 30)—however, Continuous GRU-D training itself uses the initial condition. Therefore, a smaller subset of data (1000,30) can provide better training results along with another small subset (100,30) for the test. Therefore, the model is trained for 500 iterations, and Fig. 4.8 shows the loss of the proposed Continuous GRU-D model for the Mortality task (Binary classification problem) Task.

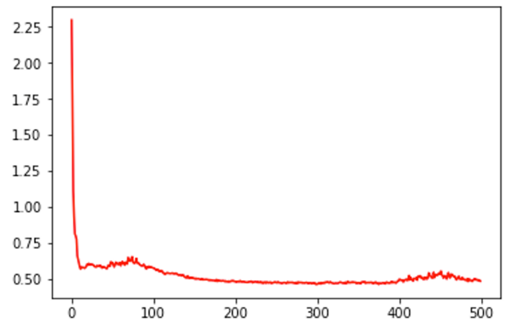


Figure 4.8: Loss in Continuous GRU-D model training for Mortality Classification task

Table 4.1: AUC on PhysioNet

Method	**AUC**
GRU-D	0.824+0.0.12
ODE-RNN	0.833+0.009
Latent-ODE (RNN Encoder)	0.781+0.018
Latent-ODE (ODE Encoder)	0.829+0.004
Latent-ODE + Poisson	0.826+0.007
ODE-GRU-D	0.8947+0.001
Extended ODE-GRU-D	0.9147+0.005

Multi-task classification problem

Predict in-hospital mortality, length-of-stay less than three days, patient's chance of having a cardiac condition, and patient' recovery state from surgery.

Table 4.1 shows that the proposed ODE-GRU-D and Extended ODE-GRU-D significantly outperform the existing GRU-D and Neural ODEs.For the comparative analysis area under curve (AUC) comparison.

4.6 Performance Evaluation

4.7 Conclusions

The most available solutions for handling missing values in continuous time series use data imputation. However, data imputation requires extensive data pre-processing; it can also impair performance. This proposed model learns the hidden dynamics of time series as progress. It can compute the hidden state at any time t without directly depending on the previous value. As the proposed model learns the derivative of changes, it does not depends on additional data imputation. The derivatives of hidden states help differentiate between longer and shorter dependencies; therefore, the proposed model works evenly fine for data with a higher sampling rate and a shorter sampling rate. As for using ODE based neural network, this proposed model requires more training time than testing time. Training time is relative to the dataset. Also, memory costs can be either fixed or incremental based on implementation. The future extension of this work mainly focuses on using a similar model when the informative missingness in any dataset is not random. Instead, it follows a particular distribution. If the time pattern of occurrence of the missing observation can be identified, it is possible to switch on or off the derivative computation. That can help predict the missingness of the information and design the model accordingly.

4.8 References

[1] Zhengping Che, Sanjay Purushotham, Kyunghyun Cho, David Sontag, and Yan Liu. Recurrent neural networks for multivariate time series with missing values. Scientific reports, 8(1):6085, 2018.

[2] A. E. Johnson, T. J. Pollard, L. Shen, H. L. Li-wei, M. Feng, M. Ghassemi, B. Moody, P. Szolovits, L. A. Celi, and R. G. Mark. MIMIC-III, a freely accessible critical care database. Scientific Data, 3:160035, 2016.

[3] Daniel Neil, Michael Pfeiffer, and Shih-Chii Liu. Phased lstm: Accelerating recurrent network training for long or event-based sequences. In Advances in Neural Information Processing Systems, pages 3882–3890, 2016.

[4] Sepp Hochreiter and Jurgen Schmidhuber. Long short-term memory. Neural computation, 9(8):1735–1780, 1997.

[5] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. Technical Report arXiv:1406.1078, arXiv, 2014.

[6] Tian Qi Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural

ordinary differential equations. In Advances in Neural Information Processing Systems, pages 6572–6583, 2018.

[7] Arthur E. Bryson, Jr. A steepest ascent method for solving optimum programming problems. Journal of Applied Mechanics, 29(2):247, 1962.

[8] Barak A. Pearlmutter. Learning state space trajectories in recurrent neural networks. Neural Computation, 1(2):263–9, 1989.

[9] Yulia Rubanova, Tian Qi Chen, and David K Duvenaud. Latent ordinary differential equations for irregularly-sampled time series. In Advances in Neural Information Processing Systems, pages 5321–5331, 2019.

[10] I. Silva, G. Moody, D. J. Scott, L. A. Celi, and R. G. Mark. Predicting in-hospital mortality of ICU patients: The physionet/computing in cardiology challenge. In CinC, 2012.

CHAPTER 5

NEURAL ODE BASED MODELS FOR ECG SYNTHESISE

This chapter is an expanded version of a paper published in 1st International Conference on Electrical, Computer and Energy Technologies (ICECET), 2022

Continuous medical time series data such as ECG is one of the most complex time series due to its dynamic and high dimensional characteristics. In addition, due to its sensitive nature, privacy concerns and legal restrictions, it is often even complex to use actual data for additional medical research. As a result, generating realistic synthetic continuous medical time series has significant practical application. Furthermore, several research works have already shown that the ability of generative adversarial networks (GANs) in the case of continuous medical time series generation is promising. Therefore, most medical data generation works, such as ECG synthesis, are mainly driven by the GAN model and its variation. On the other hand, some recent work on Neural Ordinary Differential Equations (Neural ODEs) demonstrates their strength in the face of informative missingness, high dimension, and continuous dynamic time series. Instead of considering continuous-time series as a discrete-time sequence, Neural ODEs can continuously train continuous time series in real-time. This thesis used a Neural ODE-based model to generate synthetic sine waves and synthetic ECG. We introduced a new technique to design the generative adversarial network with Neural ODE based Generator and Discriminator. We developed three new models to synthesise continuous medical data. Different evaluation metrics are then used to quantitatively assess the quality of generated synthetic data for real-world applications and data *analysis. Another goal of this thesis is to combine the strength of GAN and Neural ODE to generate synthetic continuous medical time series data such as ECG. We also evaluated both the GAN and Neural ODE models to understand the comparative efficiency of models from the GAN and Neural ODE family in medical data synthesis.*

5.1 Introduction

Neural Ordinary Differential Equations (NODE) [1] introduces a new family of neural networks, which has shown impressive performance for a continuous-time series generation. On the other hand, Generative Adversarial Network (GAN) models demonstrate impressive performance in synthetic image generation. However, the performance of GAN in terms of synthetic time series generation [2–4] is comparatively weak. ECG signal is a complex data structure. In addition, ECG signal often exhibits dynamic characteristics, such as irregular sampling rate, informative missingness [5], high dimension, multiple variables or parameters, long sequence, high frequency or data sampling rate, pathological dependency on long term memory and dynamic pattern in the dataset. Neural Ordinary Differential Equations based models [1, 6–9] demonstrate improved performance to overcome different challenges in continuous time series, such as high frequency, pathological dependency on long term memory and others.

Continuous medical time series are susceptible and require additional security. In addition, due to privacy issues, using actual data in medical research is often challenging. Therefore, we have introduced a new NODE based Generative Adversarial Network (GAN) model to generate continuous medical time series data, e.g., electrocardiogram (ECG). These generated data can be used instead of actual ECG without compromising the quality of analysis.

The main objectives of this work are as follows:

- Develop NODE based models for continuous medical time series generation
- Improve the performance of Generative Adversarial Neural Network to generate continuous medical time series
- Evaluate the performance of proposed models.

5.2 Background

GAN models are top-rated for signal synthesis. Several recent works show excellent outcomes for signal synthesis problems. However, most GAN model for signal synthesis problems is often Recurrent neural networks. This thesis focuses on developing a NODE model for signal synthesis problems. We also introduced a GAN model based on NODE to provide a better result for the signal synthesis problem. NODE [10, 11], can be presented as a continuous function instead of several layers, and the hidden state can be parametrised using an Ordinary Differential Equation (ODE) solver. ODE solvers help to compute the changes in hidden dynamics as shown in 5.1.

$$\frac{dh(t)}{dt} = \text{ODESOLVER}(f, h_t, t, \delta_t) \tag{5.1}$$

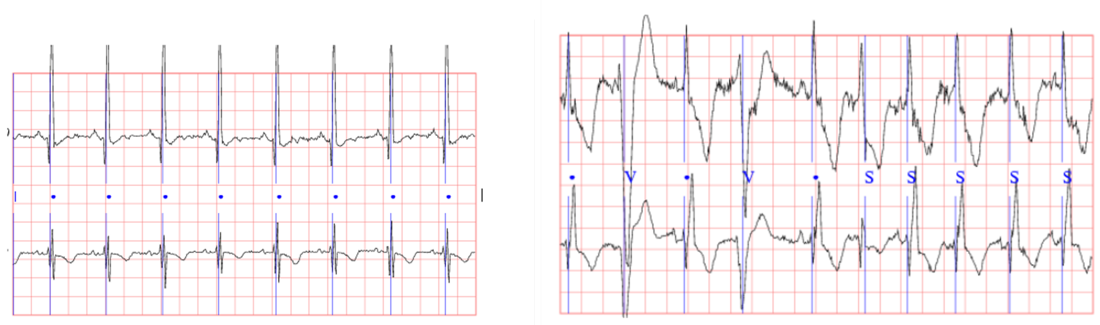
Here f is an initial problem ODE solver. Using the initial condition of $h(0)$, the ODE solver can compute the output $h(t)$ at time t. This approach provides faster training time than a residual network, constant memory cost instead of linearly increasing memory cost, and designing any neural network model is much simpler. NODE is a promising model in the field of continuous time series generation.

In another aspect, research shows that continuous time series or signal modelling can achieve a better result with recurrent neural network [12]. If the ODE solver in NODE can leverage the architecture of a Recurrent neural network (RNN), it can learn continuous time series in real-time with higher precision[6, 11]. Eq 5.2 shows that generic NODE based RNN cell [11] is used by ODE solver to compute the output y_t at time t from the initial input y_0 at time t_0.

$$y_t = \text{ODESOLVER}(ODERNNCell, y_0, t) \tag{5.2}$$

Several recent works [6, 9–11, 13] show that NODE models show significantly better performance for continuous-time series (We.e., sine wave, ECG, etc.) generation. This work introduced three models for continuous medical time series synthesis.

Our research aims to use deep learning models to generate synthetic electrocardiogram (ECG) data representing real ECG. The goal is to generate high-quality data that can reduce medical research limitations for lack of data. Fig 5.1 shows typical ECG signal. Fig. 5.1 (a) represents Normal Sinus data and Fig. 5.1 (b) represents irregular ECG for Arrhythmia. Both ECG signals are continuous in time with a higher data sampling rate. Learning ECG signals require higher efficiency and better quality. As these ECG signals are continuous, the corresponding deep learning model needs to learn them as a function of time (t) at different layers of incoming signal data.



(a) MIT-BIH Normal Sinus

(b) MIT-BIH Arrhythmia

Figure 5.1: Loss analysis for training GAN model

GANs are leading the research for ECG data generation among different deep learning models. Different models [14–17] show how GANs generate ECG data with high quality. Among these works, [16, 17] shows domain-specific knowledge such as personalisation or external stimulation can improve the performance of GANs model. Nevertheless, training personalised data with ECG data is time-consuming and needs multiple environments for training. On the other hand, [14, 15] adopt the hybrid model concept to design the architecture for GAN. GANs described in [14, 15] use Recurrent Neural Networks (RNN) as Generator and a combination of sequential RNN as well as convolutional neural networks as the Discriminator. Nevertheless, there are still limitations, as GANs are unstable in training and do not have suitable evaluation measures. In addition, learning ECG using GANs require time and memory. Therefore, the training is time-consuming. Moreover, a training model with such data as an ECG signal often requires a large training dataset. The dynamical equations of motion for ECG signal can be described as ordinary differential equations [18]. Therefore, ODE can model ECG signals as a function of continuous time. Ordinary Differential Equations also help model the heart rate's dynamic characteristics, for example, the mean and standard deviation of the ECG signal and spectral properties such as the LF/HF ratio. Therefore, NODE can provide better performance than GAN. In this thesis, we focus on using NODE to reduce the limitation of GAN and other deep learning models for ECG data generation.

5.3 Motivation

Adversarial generative approaches, like GANs, have remarkably successfully created realistic synthetic data. These approaches are usually combined with convolutional methods for images or other regularly sampled data, like speech or music. However, some signals obey physical constraints more readily modelled by differential equations and may be sparse or irregularly sampled. Here we combine the Neural ODE approach for signals of this nature with GANs to generate realistic ECG signals that can exhibit apparent non-gaussian noise processes like sensors losing contact.

5.4 Model Design

In this work, we focused on exploring the strength of NODE for ECG synthesis. Firstly, we designed a NODE based model to produce synthetic continuous-time data, which is described in section 5.4.1. Secondly, we tried to enrich the architecture of the traditional GAN model using NODE. Finally, we designed two separate GAN models, where we tried to design the Generator and Discriminator using NODE.

5.4.1 Neural ODERNN Model (ODEECGGenerator) Design for ECG Synthesis

The first model has an ODE-RNN [11] model, called *ODEECGGenerator* which is simply an Neural ODE-RNN [11] NODE model. Fig. 5.4.1 shows the architecture for the *ODEECGGenerator* model to generate ECG. The building block for this model is a NODE block that leverages Ordinary Differential Equation Solver ODESOLVER to train an ODEGRU, or ODELSTM model [11].

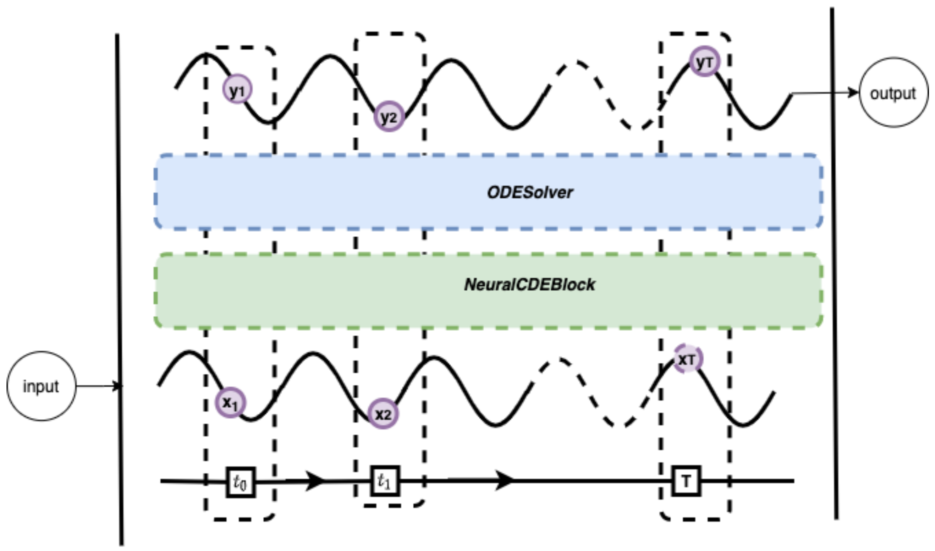


Figure 5.2: The architecture for proposed *ODEECGGenerator* model to generate ECG data

ODEECGGenerator model learns the dynamics of ECG signal in the form of Ordinary Differential Equation (ODE). The proposed Generator in this GAN model, called

ODEECGGenerator , receives ECG signal as a continuous function of time (t). Therefore, input for *ODEECGGenerator* is a value at a specific time step x_t and produces the derivative of the system at that time step. *ODEECGGenerator* takes the initial value y_0 for ECG signal at time t_0 as shown in Eq. 5.3a. *ODEECGGenerator* can be described as an ODEn as shown in Eq. 5.3b. The *NeuralODEBlock* in Fig. 5.4.1 converts the incoming signal and the hidden state of the system as ODE which is passed to the ODESOLVER. ODESOLVER uses an *ODERNNCell* as shown Eq. (5.2), which is an ODE-RNN (either *ODEGRU* or *ODELSTM*) model. To solve the ODE within time boundary $[t_0, t]$, *ODERNNCell* is optimized using the parameter γ. The incoming signal contains additional noise (z) to preserve data privacy and keep the model efficient against Adversarial attack. which is also passed as parameter for Eq. (5.2). Therefore, $\theta = tuple(z, \gamma)$.

$$y_t = ODEECGGenerator(x_t, t) \tag{5.3a}$$

$$\frac{dy}{dt} = NeuralODEBlock(\text{ODESOLVER}, y_t, t, \theta) \tag{5.3b}$$

This model is not a GAN model, and rather it is simply an Ordinary Differential Equation based RNN model. An ODESOLVER solves the Ordinary Differential Equation shown in Eq. (5.3b) within time $[t_0, t]$

5.4.2 ODEGAN Model with NODE based Generator and Discriminator

The second model proposed in this work is called ODEGAN Model which has a NODE generator, called *ODEGenerator*. The discriminator model is similar to the discriminator model used in [15]. Eq. (5.4b) shows that the *NeuralODEBlock* of *ODEGenerator* uses a *GeneratorFunc* as the neural network (f) for ODESOLVER

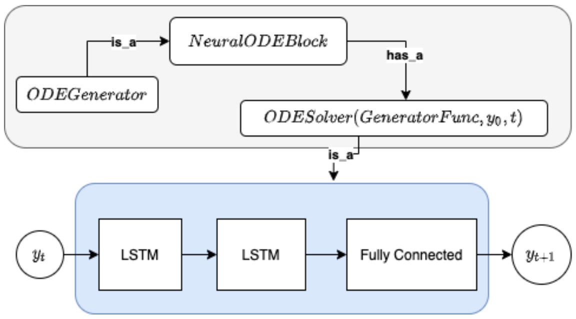


Figure 5.3: Block Diagram of Generator Architecture

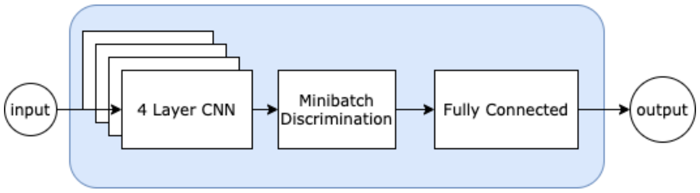


Figure 5.4: Block Diagram of Discriminator Architecture

in the generator (*ODEGenerator*). *GeneratorFunc* is an recurrent neural network, which can be optimized using parameter θ. Fig. 5.3 shows the architecture for the *ODEGenerator* generator. NeuralODEBlock computes the derivative of the input w.r.t. time. Later ODESOLVER produces a solution for the Ordinary differential equation as shown in Eq. (5.4b) for time T $[t0, ..., t]$.

$$y_t = ODEGenerator(y_0) \tag{5.4a}$$

$$\frac{dy}{dt} = NeuralODEBlock(GeneratorFunc, y_0, t, \theta, noise) \tag{5.4b}$$

The Discriminator for this GAN model is a four-layer 2-dimensional convolutional neural network with a Minibatch discrimination layer. The discriminator model is similar to the discriminator model used in [15]. The discriminator model also has noise added to the gradient of the optimiser to preserve privacy. Fig 5.4 shows the architecture for the discriminator model for this model.

Table 5.1 describes the parameter used in the Discriminator model.

Table 5.1: Parameter used in proposed ODE based Discriminator

seq_length	Length of the Input Sequence
batch_size	Size of each batch
minibatch_normal_init	Cell body
num_cv	Number of convolution Layer. Here it is 2.
cv1_out	Output shape of the first convolution Layer
cv1_s	Stride shape of the first convolution Layer
p1_k	Padding length of the first convolution Layer
cv1_k	Kernel size of the first convolution Layer
cv2_out	Output shape of the second convolution Layer
cv2_s	Stride shape of second convolution Layer
p2_k	Padding length of the second convolution Layer
cv2_k	Kernel size of the second convolution Layer
ODEDBlock	The ODE block for the ODE based Discriminator
ode_discriminator	the proposed Discriminator

Fig. 5.5 shows the pipeline for ODEGAN model. The discriminator in Fig. 5.5 is similar to the Discriminator shown in Fig 5.4. On the other hand, the Generator in Fig. 5.5 is a ODE-RNN neural network described in chapter 2. The Generator leverage an ODE solver to find the hidden state of the ECG signal at any point of interest by solving an ODE initial-value problem. The core component of the $ODEGenrator$ is an an $ODESolver$, a $NeuralODEBlock$. The $ODEGenrator$ uses $NeuralODEBlock$ to generate the hidden state at any point of time. $ODEGenrator$ passes the input as an ODE to $NeuralODEBlock$, which later uses $ODESolver$ to solve the initial value problem for the input ODE. The hidden state then generates the fake values or sample ECG signal which is then passed through the Discriminator as shown in Fig. 5.5 to be compared against the real value.

Sample ECG signal y_t is passed to the *ODEGenerator* model. The initial value for the signal is y_0. In order to preserve privacy, additional noise η is added along with parameter (γ) of the *GeneratorFunc*, therefore, $\theta = tuple(z, \gamma)$. NeuralODEBlock converts the signal to ODE, and then ODESOLVER solves the ODE, yielding a generated noisy ECG signal for time T $[t0, ..., t]$. The Discriminator then learn to

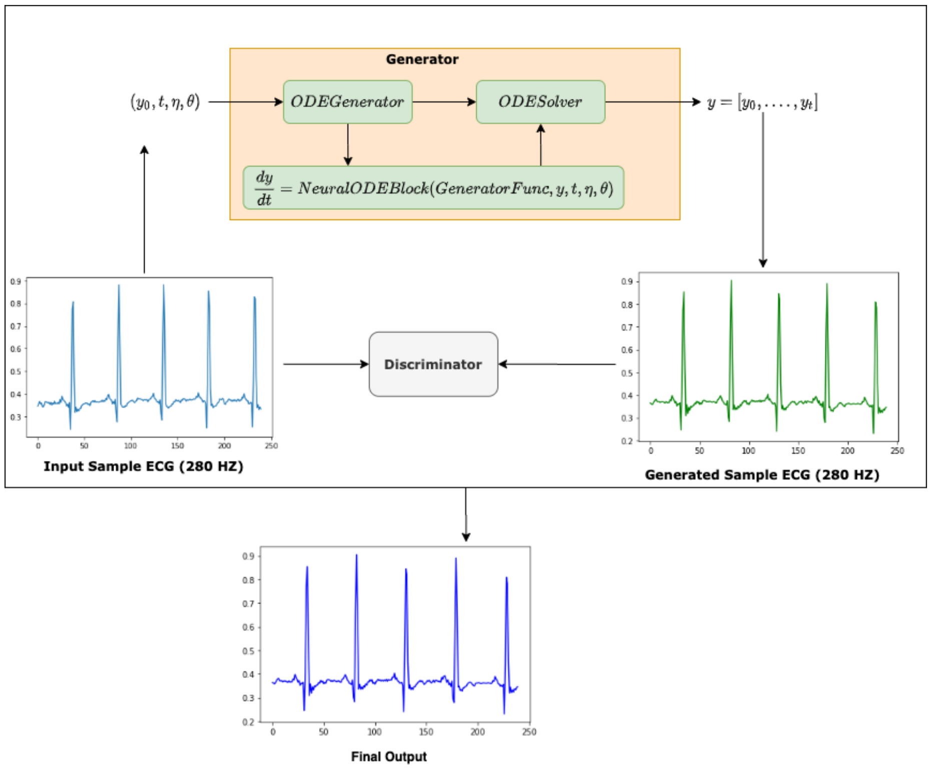

Generator
(y_0, t, η, θ)
ODEGenerator
ODESolver
$y = [y_0, \ldots, y_t]$
$\frac{dy}{dt} = NeuralODEBlock(GeneratorFunc, y, t, \eta, \theta)$
Discriminator
Input Sample ECG (280 HZ)
Generated Sample ECG (280 HZ)
Final Output

Figure 5.5: pipeline for GAN model with ODE generator

distinguish between the real and the generated signal.

During the training phase, *ODEGenerator* generates continuous time series similar to an actual ECG signal. A log probability ($\mathcal{L}$ matrix evaluates the *ODEGenerator* model by computing $\mathcal{L}$ of negative identification of the generated signal by the Discriminator. The optimiser optimises the parameter of the *ODEGenerator* model with a target to minimise $\mathcal{L}$. On the other hand, a cross-entropy loss function evaluates the Discriminator by computing $\mathcal{L}$ of the correct classification of the natural and generated signal. During the training, the optimiser designated for the Discriminator reduces $\mathcal{L}$ to its minimum value.

5.4.3 GAN Model with NODE based Generator and Discriminator

For the third model proposed in this work, the ODE-GAN-2 model, we designed both Generator and Discriminator using NODE models. The generator model described in

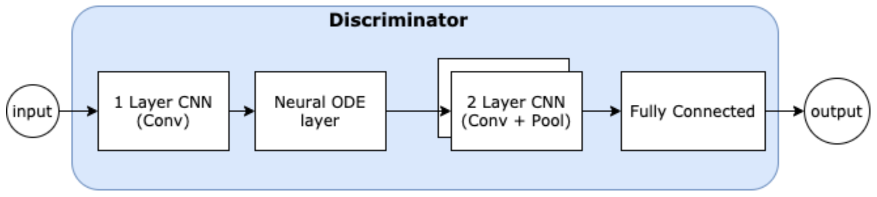


Figure 5.6: Block Diagram for the Discriminator

section 5.4.1 is the Generator for this generative adversarial neural network. Eq. (5.3) shows that the NeuralODEBlock uses an ODERNNCell, which can be either LSTM or GRU, to generate a continuous time series y_t by using an ODESOLVER on the ODE defined in Eq. (5.3b).

We have used two different kinds of discriminators for this model as follows.

- Convolution layer with NODE layer
- NeuralCDE network [7]

Convolution layer with NODE layer-based Discriminator

This Discriminator has one convolutional layer followed by a NODE layer and then two more convolutional and max-pooling pair layers. Fig 5.6 shows the block diagram of the Discriminator.

Table 5.1 describes the parameter used in proposed ODE based Discriminator. $DiscriminatorFunc$ is the neural network to learn the difference between real and generated ECG signals. The $ODESOLVER$ also uses $DiscriminatorFunc$ in the Discriminator. This Discriminator distinguishes between the derivative of ECG signal w.r.t. time of the real and generated signal.

NeuralCDE network as Discriminator

The Discriminator for this model is a NeuralCDE network [7]. NeuralCDE network, as shown in Fig 5.7, converts the data to a conditional continuous path X using interpolation and this path X is passed through ODESOLVER to solve the ODE derived from path X in order to lean the hidden dynamics of the ODE system. For our third model, we leverage the concept of the NeuralCDE network to learn the difference between a real ECG signal and the generated ECG signal. The Discriminator model described in Algorithm 5.4.3 uses a Neural controlled differential equations [7] as the ODESOLVER function. Fig. 5.7 shows that the input time series for this Discriminator has two-channel, e.g., for real ECG signal and generated ECG signal.

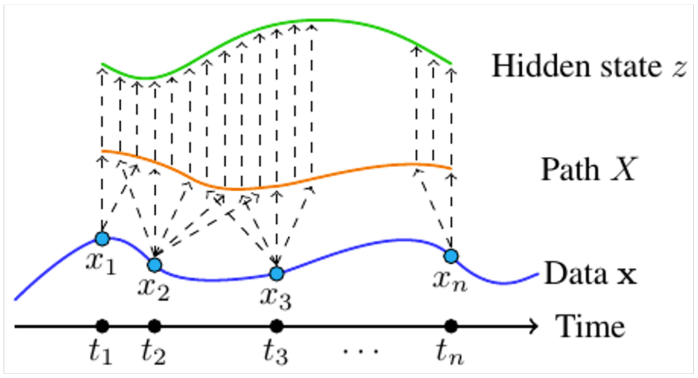


Figure 5.7: NeuralCDE model [7]

Algorithm 10 The Neural CDE based Discriminator

```
discriminatorFunc = NeuralCDE(input_channels=seq_length, hidden_channels=hidden_dim, output_channels=out_dim) ode_discriminator = nn.Sequential(discriminatorFunc)
```

Table 5.2 describes the parameter used in proposed NeuralCDE based Discriminator.

Table 5.2: The Parameters of NeuralCDE based Discriminator

input_channels	Length of the Input Sequence
hidden_dim	Hidden dimension of the NeuralCDE network
output_channels	Output dimension of the NeuralCDE network

Fig. 5.8 shows that the discriminator takes real ECG signal x_t and generated ECG signal y_t as input. Both signals x_t and y_t are controlled ordinary differential equations. interpolation between x_t and y_t generates an intermidieate continuous path U. NeuralCDE

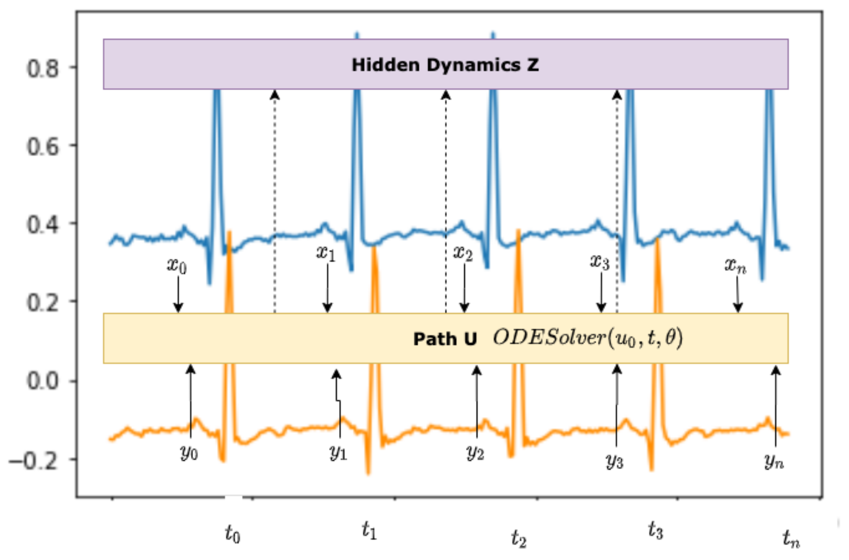


Figure 5.8: NeuralCDE based Discriminator

based Discriminator learns the hidden dynamics (Z) of U to distinguish real signal and generated signal correctly.

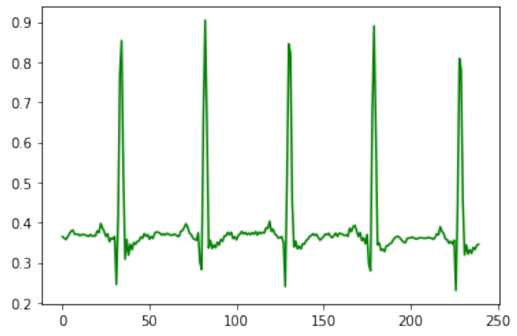
5.5 Performance Evaluation

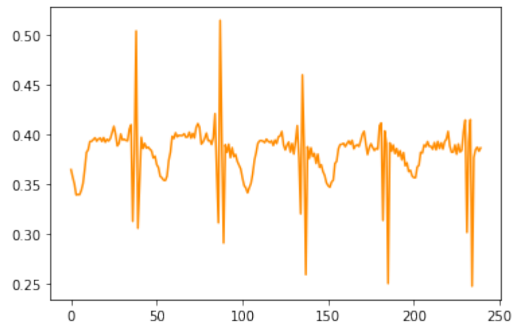
The quality and performance of the proposed model are assessed using an experiment with two different tasks, i.e. (i) Generate Normal Sinus ECG and (ii) Generate Arrhythmia ECG. Furthermore, the proposed model is evaluated against state art Neural Network used for ECG Synthesis as described in [15] and NeuralCDE.

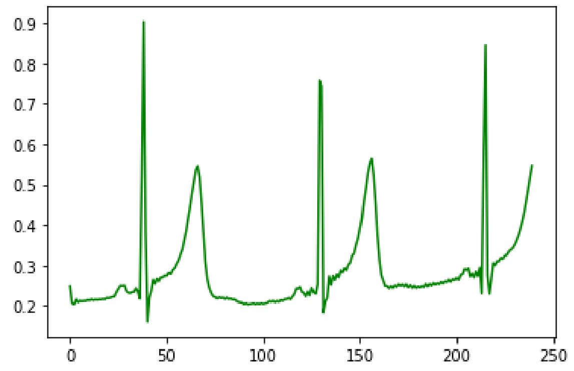
5.5.1 Dataset

This experiment uses two different kinds of multi-channel publicly available ECG datasets, e.g., (i) MIT-BIH Arrhythmia dataset on PhysioNet [19] and (ii) MIT-BIH Normal Sinus Rhythm Database [19]. Both of the Databases show unique characteristics. For example, MIT-BIH Normal Sinus Rhythm Database consists of clean ECG recordings. As shown in Fig. 5.2, this Database shows minimal noise in the ECG continuous series. On the other hand, the ECG recordings for MIT-BIH

Arrhythmia on PhysioNet were created by adding calibrated noise to clean the MIT-BIH Normal Sinus Rhythm Database. Therefore, the MIT-BIH Arrhythmia dataset contains a significant amount of noise, as shown in Fig. 5.2.


(a) MIT-BIH Normal Sinus Rhythm Database First Channel


(b) MIT-BIH Normal Sinus Rhythm Database Second Channel


(c) MIT-BIH Arrhythmia dataset on PhysioNet First Channel

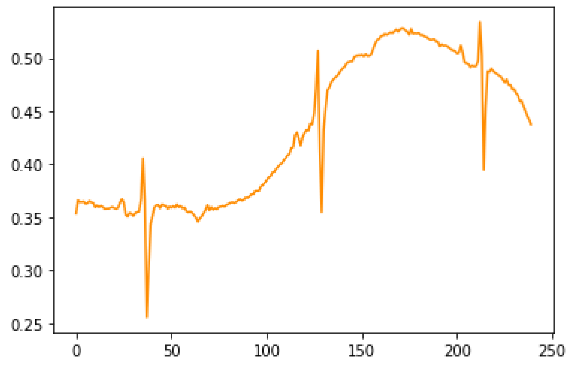
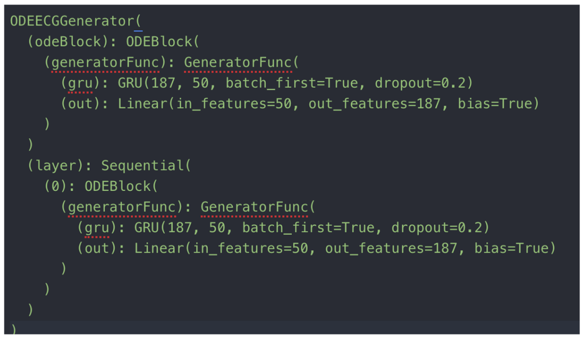

(d) MIT-BIH Arrhythmia dataset on PhysioNet Second Channel

Figure 5.9: Configuration for neural network used in Human activity experiment

On the other hand, the MIT-BIH Arrhythmia dataset on the PhysioNet database has significant noise. Therefore, the ECG recordings for MIT-BIH Arrhythmia on PhysioNet were Database was created by adding calibrated amounts of noise to clean ECG recordings from the MIT-BIH Normal Sinus Rhythm Database.

5.5.2 *ODEECGGenerator*model Training

Fig. 5.10 shows the architecture for proposed *ODEECGGenerator* model used in this experiment.



```
ODEECGGenerator(
  (odeBlock): ODEBlock(
    (generatorFunc): GeneratorFunc(
      (gru): GRU(187, 50, batch_first=True, dropout=0.2)
      (out): Linear(in_features=50, out_features=187, bias=True)
    )
  )
  (layer): Sequential(
    (0): ODEBlock(
      (generatorFunc): GeneratorFunc(
        (gru): GRU(187, 50, batch_first=True, dropout=0.2)
        (out): Linear(in_features=50, out_features=187, bias=True)
      )
    )
  )
)
```

Figure 5.10: Proposed *ODEECGGenerator* model for ECG Synthesis

Table 5.3 describes the settings used for performance evaluation. The dynamic characteristics of proposed models enable us to train them quickly and with fewer parameters. For robust training, the dataset was split at random into 80% for training and 20% for the test.

Table 5.3: The Parameters of Experiment for *ODEECGGenerator* and GAN models training

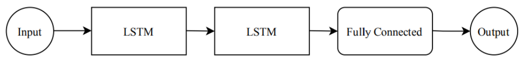
Parameters	*ODEECGGenerator*	GAN
Batch Size	50	64
dataseize	100	1000
Sequence Length	240	240
Hidden Dimension	50	50
Learning rate	0.0001	0.00005
Number of epoch	100	30×1000

5.5.3 GAN model for ECG Synthesis

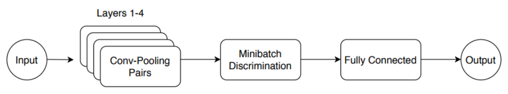
For comparative analysis, the proposed model is evaluated against the GAN model for ECG synthesis described in [15]. Fig 5.12 shows the block diagram for Generator and Discriminator used in [15].

The Generator for GAN model [15] as shown in Fig. 5.12 has two consecutive RNN layers (LSTM), followed by a fully connected layer.

The Discriminator proposed in [15] has a comparatively complex architecture as shown in Fig. 5.13 and Fig. 5.11.

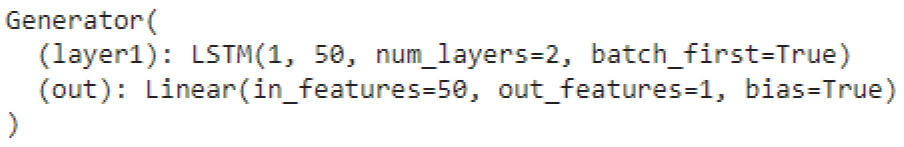


(a) Block Diagram of Generator Architecture



(b) Block Diagram of Discriminator Architecture

Figure 5.11: Block Daigram for Generator and Discriminator for GAN model for ECG synthesis described in [15]



```
Generator(
  (layer1): LSTM(1, 50, num_layers=2, batch_first=True)
  (out): Linear(in_features=50, out_features=1, bias=True)
)
```

Figure 5.12: Generator model architecture

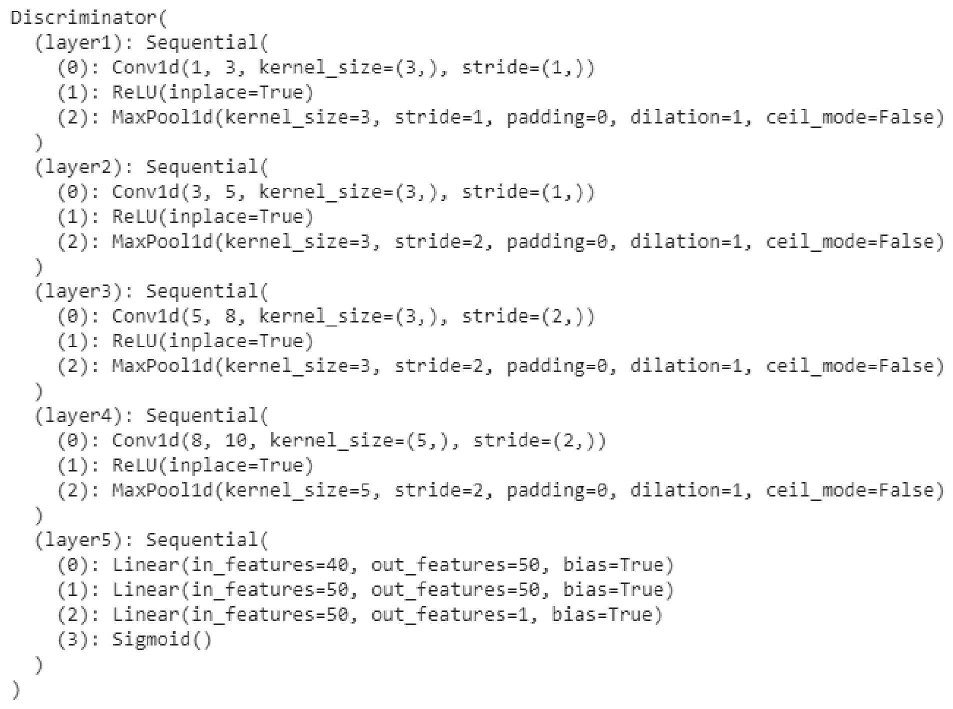
Table 5.1 describes the parameter used in GAN implementation of [15]. GAN model training requires a comparatively higher number of parameters than the proposed *ODEECGGenerator* model. It also needs to be trained for a longer time for each epoch. Out of 30 epochs runs through 1000 iterations. A comparatively longer series of 1000 data sizes are required for the GAN model. The training requires three to five hours for 30 epochs.

Fig. 5.14(b) and Fig. 5.14(c) shows the generated ECG for Normal and Arrhythmia signal by the proposed *ODEECGGenerator* model after training for 100 iterations. Fig 5.15 shows training loss for the generated ECG by by Model proposed in [15] after training for 100 iterations.

The training loss analysis shown in 5.15 shows that ODERNN based model reach a stable result within 20 iteration, which is significantly quicker than GAN based model [15].

Similarly, Fig 5.16 shows the loss for Generator and Discriminator training for the GAN model over 100 epochs.

The result evaluation between the proposed *ODEECGGenerator* model and the existing



```
Discriminator(
  (layer1): Sequential(
    (0): Conv1d(1, 3, kernel_size=(3,), stride=(1,))
    (1): ReLU(inplace=True)
    (2): MaxPool1d(kernel_size=3, stride=1, padding=0, dilation=1, ceil_mode=False)
  )
  (layer2): Sequential(
    (0): Conv1d(3, 5, kernel_size=(3,), stride=(1,))
    (1): ReLU(inplace=True)
    (2): MaxPool1d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (layer3): Sequential(
    (0): Conv1d(5, 8, kernel_size=(3,), stride=(2,))
    (1): ReLU(inplace=True)
    (2): MaxPool1d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (layer4): Sequential(
    (0): Conv1d(8, 10, kernel_size=(5,), stride=(2,))
    (1): ReLU(inplace=True)
    (2): MaxPool1d(kernel_size=5, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (layer5): Sequential(
    (0): Linear(in_features=40, out_features=50, bias=True)
    (1): Linear(in_features=50, out_features=50, bias=True)
    (2): Linear(in_features=50, out_features=1, bias=True)
    (3): Sigmoid()
  )
)
```

Figure 5.13: Discriminator model architecture

GAN model shows that NODE enables *ODEECGGenerator* to achieve comparatively higher accuracy with fewer parameters and a concise data series. As a result, the *ODEECGGenerator* model does not need to be trained for a longer time and is essentially a hidden dimension.

Fig. 5.17 shows the generated ECG by Neural CDE and proposed ODE GAN model with *ODEECGGenerator* as well as the ODE GAN model with NODE based Generator and Discriminator. These generated ECG signals demonstrate that NODE based models can perform significantly better in continuous medical data generation.

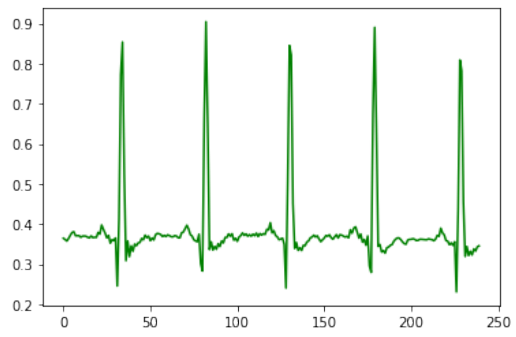
5.5.4 Comparative Analysis of GAN and proposed models for medical time series generation

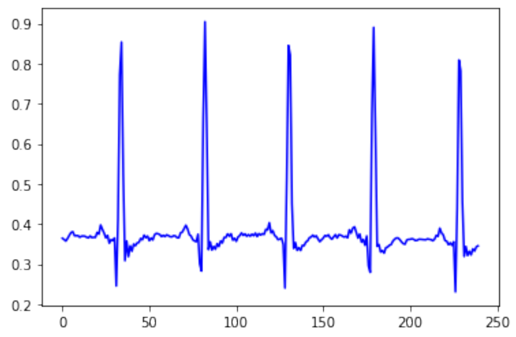
The normal sinus rhythm in the Normal Sinus Rhythm Database has a rate between 50 and 100 beats/minute at rest. This is the standard heart rate with a periodic pattern. There are two separate channels in the dataset. The first channel (Fig. 5.5.1) contains

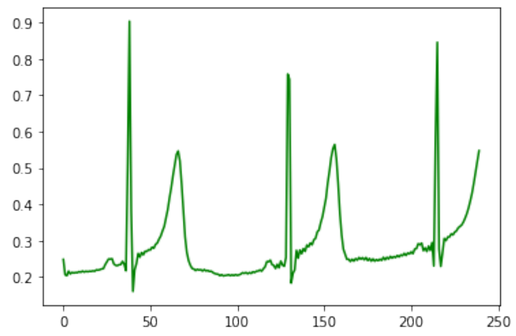
ECG of continuous rhythm, whereas the Second Channel of the Normal Sinus Rhythm Database ((Fig. 5.5.1)) presents ECG with variant rhythm. The loss is shown in Fig. 5.5.3 and 5.5.3 shows that ECG with the minor variant in the series can quickly achieve high accuracy in the case of the *ODEECGGenerator* model. However, when there is more variant in the time series and the pattern changes frequently, it takes a comparatively higher time to achieve acceptable accuracy. However, the training time for the *ODEECGGenerator* model is still significantly lower than the GAN.

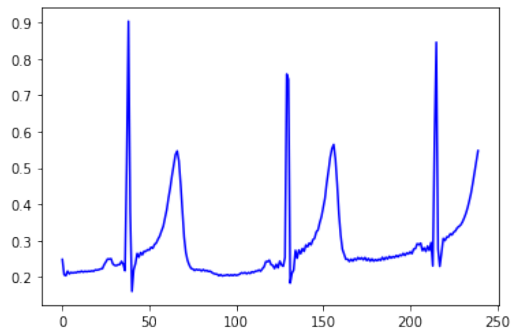
ODEECGGenerator model achieve acceptable accuracy for Arrhythmia dataset comparatively later than Normal Sinus dataset as shown in Fig. 5.5.3 and Fig. 5.5.3.

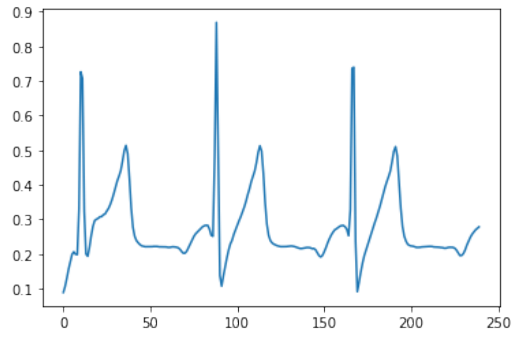
A similar model, called ECG-ODE-GAN [20] as the proposed ODEGAN model described in 5.4.2 tries to learn purely data-driven dynamics from ECG signal. This ECG-ODE-GAN model also shows the impact of physical parameters in morphological descriptors of the ECG signal instances.


(a) Original MIT-BIH Normal Sinus Rhythm


(b) *ODEECGGenerator* generated MIT-BIH Normal Sinus Rhythm


(c) Original MIT-BIH Arrhythmia First Channel


(d) *ODEECGGenerator* generated MIT-BIH Arrhythmia First Channel


(e) GAN generated MIT-BIH Arrhythmia First Channel

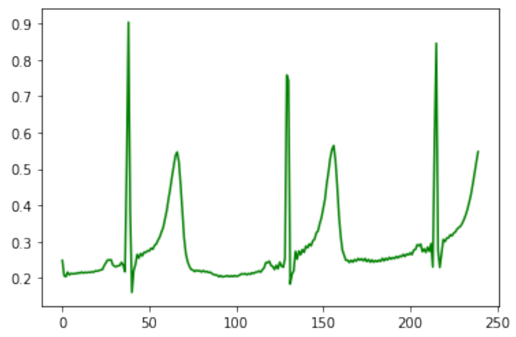
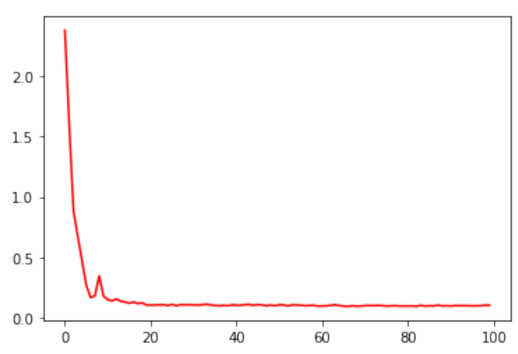
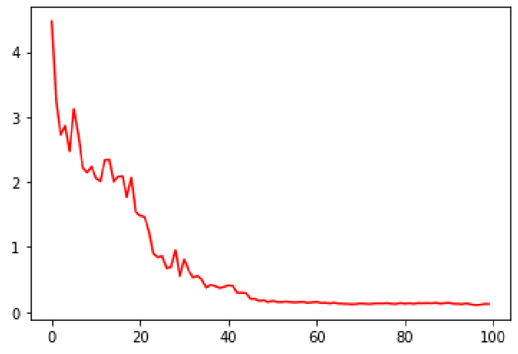
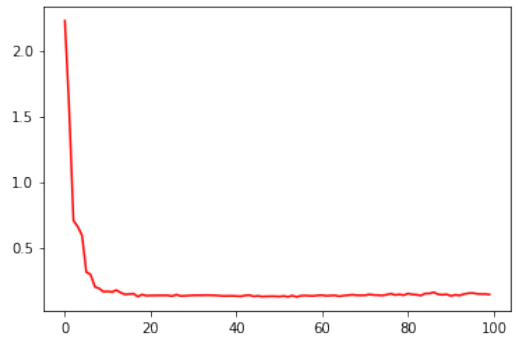

(f) *ODEECGGenerator* generated MIT-BIH Arrhythmia First Channel

Figure 5.14: Comparative result analysis for proposed model


(a) *ODEECGGenerator* model training loss for MIT-BIH Normal Sinus Rhythm Database First Channel


(b) *ODEECGGenerator* model training loss for MIT-BIH Normal Sinus Rhythm Database Second Channel


(c) *ODEECGGenerator* model training loss for MIT-BIH Arrhythmia dataset on PhysioNet First Channel

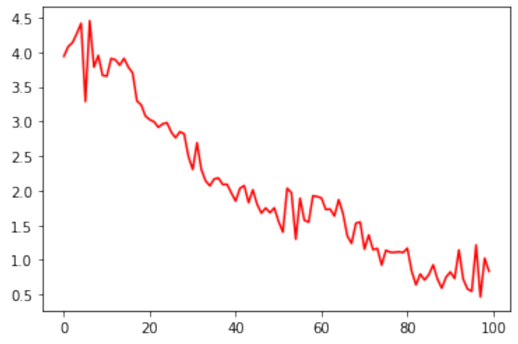
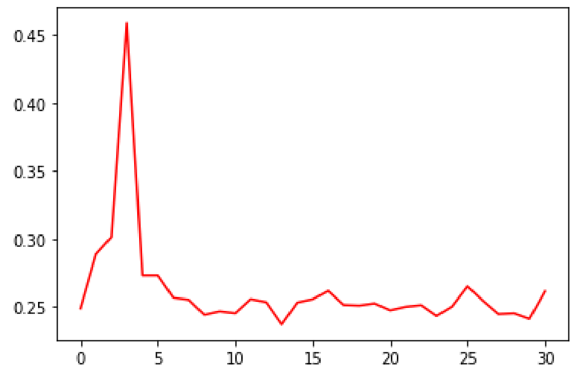

(d) *ODEECGGenerator* model training loss for MIT-BIH Arrhythmia dataset on PhysioNet Second Channel

Figure 5.15: Loss analysis for training *ODEECGGenerator* model


(a) Generator training loss for MIT-BIH Normal Sinus Rhythm Database

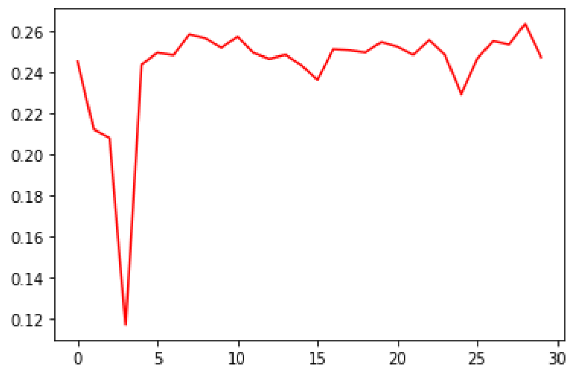
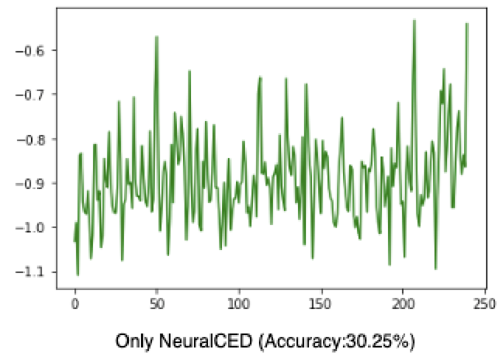
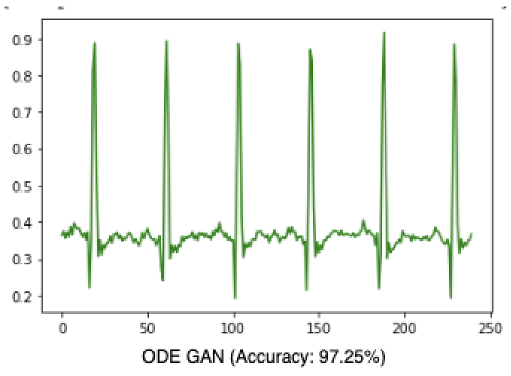

(b) Discriminator training loss for MIT-BIH Normal Sinus Rhythm Database

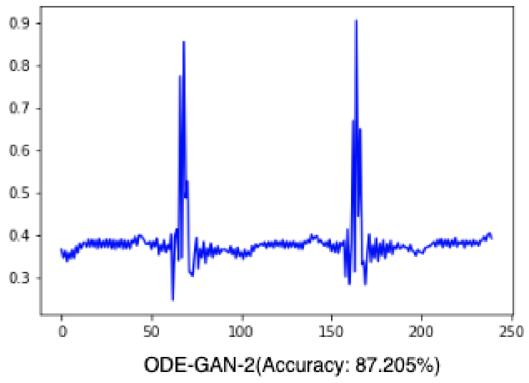
Figure 5.16: Loss analysis for training GAN model



(a) NeuralCDE generated ECG Signal



(b) ODE GAN generated ECG Signal



(c) ODE GAN 2 generated ECG Signal

Figure 5.17: Generated ECG signals by different models

5.6 Conclusion

We have presented three different models for continuous medical data synthesis. Firstly, we introduce a new technique to learn the hidden dynamics of ECG signals by ODE-RNN only. Secondly, we introduced a generative adversarial network with a NODE Generator and a standard CNN based Discriminator. We experimented with a NODE as Generator for the third model and NeuralCDE as the Discriminator. Finally, we demonstrated that NODE models could improve the accuracy of standard generative adversarial networks. NODE models are data-driven generative models, they perform better as the Generator in generative adversarial networks. These hybrid Neural generative adversarial networks can be used to generate any system that can de be described as ODE. The performance evaluation also shows that these models perform better when the batch size for training data is small (20-50) and with fewer parameters. Furthermore, if the pattern in the signal is regular, the train time is lower than in the case of a periodic signal. This model can be a suitable candidate to generate systems described by PDEs, as they are data-driven and continuous in time.

5.7 References

[1] Tian Qi Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. In Advances in Neural Information Processing Systems, pages 6572–6583, 2018.

[2] Christian Ledig, Lucas Theis, Ferenc Huszár, Jose Caballero, Andrew Cunningham, Alejandro Acosta, Andrew Aitken, Alykhan Tejani, Johannes Totz, Zehan Wang, et al. Photo-realistic single image super-resolution using a generative adversarial network. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 4681–4690, 2017.

[3] Kay Gregor Hartmann, Robin Tibor Schirrmeister, and Tonio Ball. Eeg-gan:

Generative adversarial networks for electroencephalograhic (eeg) brain signals. arXiv preprint arXiv:1806.01875, 2018.

[4] Dan Li, Dacheng Chen, Baihong Jin, Lei Shi, Jonathan Goh, and See-Kiong Ng. Mad-gan: Multivariate anomaly detection for time series data with generative adversarial networks. In International Conference on Artificial Neural Networks, pages 703–716. Springer, 2019.

[5] Zhengping Che, Sanjay Purushotham, Kyunghyun Cho, David Sontag, and Yan Liu. Recurrent neural networks for multivariate time series with missing values. Scientific reports, 8(1):6085, 2018.

[6] Mansura Habiba and Barak A Pearlmutter. Neural odes for informative missingness in multivariate time series. arXiv preprint arXiv:2005.10693, 2020.

[7] Patrick Kidger, James Morrill, James Foster, and Terry Lyons. Neural controlled differential equations for irregular time series. arXiv preprint arXiv:2005.08926, 2020.

[8] Will Grathwohl, Ricky TQ Chen, Jesse Bettencourt, Ilya Sutskever, and David Duvenaud. Ffjord: Free-form continuous dynamics for scalable reversible generative models. arXiv preprint arXiv:1810.01367, 2018.

[9] Yulia Rubanova, Tian Qi Chen, and David K Duvenaud. Latent ordinary differential equations for irregularly-sampled time series. In Advances in Neural Information Processing Systems, pages 5321–5331, 2019.

[10] Zhengping Che, Sanjay Purushotham, Kyunghyun Cho, David Sontag, and Yan Liu. Recurrent neural networks for multivariate time series with missing values. Scientific reports, 8(1):1–12, 2018.

[11] Mansura Habiba and Barak A Pearlmutter. Neural ordinary differential equation based recurrent neural network model. arXiv preprint arXiv:2005.09807, 2020.

[12] Daniel Neil, Michael Pfeiffer, and Shih-Chii Liu. Phased lstm: Accelerating recurrent network training for long or event-based sequences. In Advances in Neural Information Processing Systems, pages 3882–3890, 2016.

[13] Ori Linial, Danny Eytan, and Uri Shalit. Generative ode modeling with known unknowns. arXiv preprint arXiv:2003.10775, 2020.

[14] Eoin Brophy. Synthesis of dependent multichannel ecg using generative adversarial networks. In Proceedings of the 29th ACM International Conference on Information & Knowledge Management, pages 3229–3232, 2020.

[15] Anne Marie Delaney, Eoin Brophy, and Tomas E Ward. Synthesis of realistic ecg using generative adversarial networks. arXiv preprint arXiv:1909.09150, 2019.

[16] Tomer Golany, Kira Radinsky, and Daniel Freedman. Simgans: Simulator-based generative adversarial networks for ecg synthesis to improve deep ecg classification. In International Conference on Machine Learning, pages 3597–3606. PMLR, 2020.

[17] Tomer Golany and Kira Radinsky. Pgans: Personalized generative adversarial networks for ecg synthesis to improve patient-specific deep ecg classification. In Proceedings of the AAAI Conference on Artificial Intelligence, volume 33, pages 557–564, 2019.

[18] Patrick E McSharry, Gari D Clifford, Lionel Tarassenko, and Leonard A Smith. A dynamical model for generating synthetic electrocardiogram signals. IEEE transactions on biomedical engineering, 50(3):289–294, 2003.

[19] Ary L Goldberger, Luis AN Amaral, Leon Glass, Jeffrey M Hausdorff, Plamen Ch Ivanov, Roger G Mark, Joseph E Mietus, George B Moody, Chung-Kang Peng, and H Eugene Stanley. Physiobank, physiotoolkit, and physionet: components of a new research resource for complex physiologic signals. circulation, 101(23): e215–e220, 2000.

[20] Tomer Golany, Daniel Freedman, and Kira Radinsky. Ecg ode-gan: Learning ordinary differential equations of ecg dynamics via generative adversarial learning. In Proceedings of the AAAI Conference on Artificial Intelligence, volume 35, pages 134–141, 2021.

CHAPTER 6

CONTINUOUS CONVOLUTIONAL NEURAL NETWORKS: COUPLED NEURAL PDES

This chapter is an expanded version of a paper published in the 1st International Conference on Electrical, Computer and Energy Technologies (ICECET), 2021

Recent work in deep learning focuses on solving physical systems in Ordinary Differential Equations or Partial Differential Equations [1]. This work proposes a variant of Convolutional Neural Networks (CNNs) that can learn the hidden dynamics of a physical system using ordinary differential equation (ODE) systems and Partial Differential Equation (PDE) systems. Instead of considering the physical system to consist of multiple layers, such as image or time-series processing systems, these new technique models a system in Differential Equation (DEs). Therefore, it is possible to model the system with a certain continuity. The proposed method has been assessed by solving several steady-state PDEs on irregular domains, including heat and Navier-Stokes equations.

6.1 Introduction

It is challenging to have an optimal balance between accuracy and efficiency while finding a solution for a Partial Differential Equation (PDE) system. Some recent research in deep learning has focused on Physics informed neural networks [2–4]. At the same time, some other research on data-driven neural networks [5–7] explains that if the system can be represented as a function of continuous-time series, corresponding data-driven neural networks demonstrate impressive results. Recently, Neural ODE or NODE [8] is becoming promising for solving systems of ODE. However, we are still looking for a robust model for solving the PDE system. This paper presents a continuous CNN model that can learn a PDE system continuously. Traditional CNN models consider a continuous PDE system as a system with multiple layers, as shown in Fig.6.1. However, learning continuous time series in real-time is a very challenging task. Moreover, the sampling rate for PDE or ODE systems can be irregular and bursty [9]. As a result, despite the excellent performance of CNN in several research areas, they fail to exhibit similar performance with continuous data. On the other hand, a one-dimensional CNN or a Time Delay Neural Network (TDNN) [10], can achieve a better result for continuous data in comparison to other CNN architectures. For example, TDNN models are used for Natural Language Processing (NLP). However, a continuous system such as an integral PDE system is very complex for TDNN to achieve high accuracy.

In this work, we leverage the strength of a PDE solver and an ODE solver to define such a complex system. First, we propose a new CNN model that acts like a one-dimensional TDNN to generate the integral of a PDE system and later convert that to an ODE system. Finally, an ODE solver is used to generate the output of the continuous system. Here, a continuous system is a system that takes inputs continuously with higher frequency. For example, an IoT device continuously collected sensor data over time duration $\mathcal{T}$. We have adopted the architecture of a dimensional Convolutional Neural Network similar to TDNN and introduced continuous learning by coupling a PDE and an ODE system in the model. We call it Continuous Convolutional Neural Network (CCNN).

The main contribution of this paper is as follows:

- A novel Continuous Convolutional Neural Network (CCNN).

- A new technique to learn PDE systems using deep learning.
- Finally, a comprehensive performance evaluation of the proposed model using a simulated system for Chirps [11].

6.2 Motivation

CNN models are primarily used for spatial data, such as object recognition and image processing. Only a few specialised CNN-based models have been developed to process and learn non-uniform data. These CNN models are mainly standard feed-forward models. However, CNNs are not suitable models for continuous-time series modelling. The fundamental characteristic of a CNN model is that the sampling rate is regular and fixed for the same pattern. For example, CNN shows better performance for an object recognition task. Therefore, CNN can recognise data with a regular and fixed sampling rate, but traditional CNN can not perform well if the sampling rate varies. On the other hand, RNN models use memory-based architecture to store the pattern and learn the dynamic pattern changes for time series modelling. As a result, CNN is not used for continuous-time series modelling. In order to use CNN for time series modelling problems, time series are re-sampled uniformly; however, if we can design CNN architecture so that the CNN model can learn the change in the dynamic pattern, it can ultimately improve the performance of learning.

To leverage the strength of the CNN model for continuous-time series modelling, CNN needs the following capabilities to be able:

- to convert non-uniform time series to uniform time series with minimum variation in the pattern.
- to run continuous convolution operation on the continuous signal.
- to estimate the implicit continuous signal. Therefore, the missing value does not

affect the efficiency of the model

- to capture the relevant patterns in the input signal regardless of the irregular sampling rate.
- to optimise performance based on external parameters.

Therefore, this work proposes a model that can act as a continuous CNN model. In this proposed model, the interpolation and convolution kernels are not performed on data separately but rather co-related. We leverage the solution of the PDE solver and ODE solver to execute the interpolation and convolution operations. Instead of running interpolation and convolution separately on the input signal, they are executed in an interconnected manner. The proposed Continuous CNN (CCNN) model is inspired by Neural ODE and NeuPDE models. The CCNN convolves the input with continuous kernel functions defined by a PDE solver. At the same time, to recover the continuousness in the input data at an irregular sampling rate, the proposed model leverages ODE solvers to run interpolation on the input signal. The proposed model is data-driven, as it learns from the changes in the input and hidden dynamics of the system. Therefore, the time intervals and missing values in the time series data can be tolerated.

6.3 Model Design

The problem domain for the proposed model is a continuous time series with regular or irregular sampling, as shown in Eq. (6.1), where $t_i < t_{i+1}$ but $t_{i+1} - ti$ varies.

$$y_i = y(t_i) \tag{6.1}$$

Eq. (6.1) can be also written as a function of time $f(t)$, so that, a continuous convolutional layer can produce an output $y(t) = f(t)$ at any time t. The proposed

CCNN model needs to overcome the irregular sampling rate and non-uniformity in the input signal by conducting some implicit interpolation and continuing to execute continuous convolution. The goal of this proposed model is to generalise Convolutional Neural Networks (CNNs) [12] to continuous space and time. The proposed model needs to have the capabilities mentioned in 6.2 section. We need a mechanism which can ignore the irregular pattern of the data. In order to achieve that, data need to be interpolated implicitly during the training regardless of the time. Secondly, the convolution operation should be executed continuously.

TDNN models [10, 13] have shown some progress in modelling time series using one dimensional CNN architecture. The proposed model leverage the strength of TDNN architecture. For CCNN model, the foundation is a TDNN with a single unit which can generalise the input signal to continuous time series. Fig. 6.1 shows the architecture of a TDNN model.

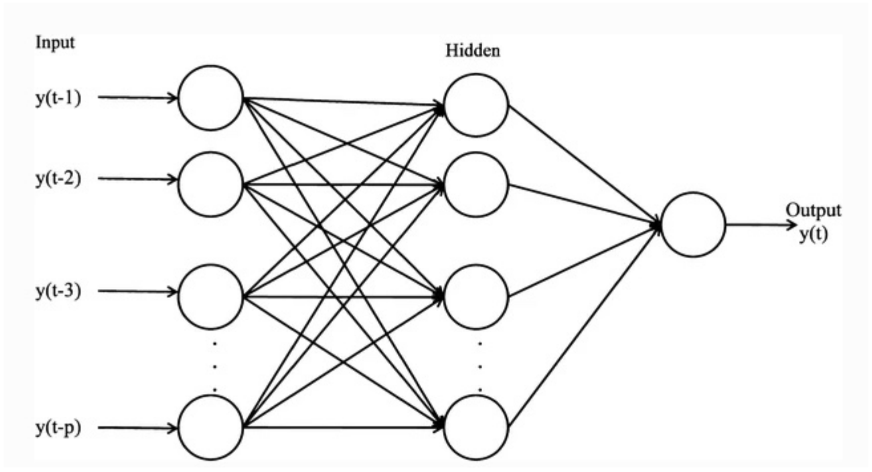


Figure 6.1: Time-delay neural network (TDNN) with hidden layers

In discrete time a TDNN model can be represented by Eq. (6.2)

$$y(t+1) = f(\sum_{\tau=0}^{T} K(\tau, W_K, t) y(t-\tau)) \tag{6.2}$$

where $y(t)$ is the activity of the unit at time t, the function f is a convenient nonlinearity, and $K(\tau, W_k)$ is a kernel function parameterized by W_k, typically using an explicit representation in the form of an array so $K(\tau, W_k) = W_k[\tau]$. This is defined for $T_0 \leq t \leq T_1$, with boundary conditions (i.e., initialization) handled specially.

To move to continuous time, a TDNN model can be described as Eq. (6.3).

$$\frac{\mathrm{d}}{\mathrm{d}t} y(t) = f(\int_{\tau=0}^{T} K(\tau, W_k, t) y(t-\tau) \mathrm{d}\tau) \tag{6.3}$$

Here, the generalised kernel function $K(\tau, W_k, t)$ is continuous and parametric rather than discrete and explicit. This nature is causal, but involves an integral for implementation. As a result, the generalised kernel function can be described as a delay-differential equation. When coding, this would mean that the differential form passed to the ODE solver would contain an invocation of an integration operator, to which the solution of $y(t')$ for $t' \leq t$ would be passed. However, typically, this is not available: only the current value of $y(t)$ is available. So instead, we encode the integral in a PDE system as shown in Eq. (6.4)(a).

$$u(t, 0) = 0 \tag{6.4a}$$
$$\frac{\partial}{\partial \tau} u(t, \tau) = K(\tau, W_k, t) y(t-\tau) \tag{6.4b}$$

with $0 \leq \tau \leq T$. Note that

$$u(t, \tau') = \int_{\tau=0}^{\tau'} K(\tau, W_k, t) y(t-\tau) \mathrm{d}\tau \tag{6.5}$$

So in particular for a time duration T=[0,T], Eq. (6.4) provides the integral form of the PDE system.

$$u(t, T) = \int_{\tau=0}^{T} K(\tau, W_k, t) y(t-\tau) \mathrm{d}\tau \tag{6.6}$$

At this point, the integral from the driving term of y can be removed, instead expressing

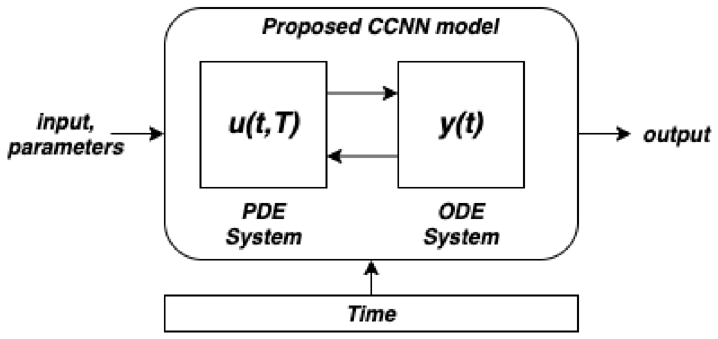


Figure 6.2: The block diagram of proposed Continuous CNN model

it in terms of u as shown in Eq. (6.6).

$$\frac{\mathrm{d}}{\mathrm{d}t} y(t) = f(u(t, T)) \tag{6.7}$$

Now we have a standard PDE for u as shown in Eq. (6.6) and an ODE for y as shown in Eq. (6.7), with no delays or integrals, and therefore Eq. (6.7) is suitable for standard solvers. Fig. [6.2] shows the block diagram of the CCNN model. Inputs are processed through a PDE system that leverages the kernel function as shown in Eq. (6.6) and encodes the integral in the PDE system as $u(t, T)$. PDE system then forwards the output $u(t)$ to a coupled ODE system. The PDE system uses the output of the ODE system from the previous state, and similarly, the ODE system leverage the output of the current state from the PDE system. So these two systems are coupled together. In summary, CCNN is a coupled system of PDE and ODE systems.

Fig. 6.3 shows the architecture diagram of proposed CCNN model. The core component of the CCNN model is an one dimensional TDNN model. CCNN model consists of a fully connected TDNN model which takes input $Y = \{y_1, y_2, ...\ y_{t-1}\}$ continuously at each time step $T = \{t1, t2, ...\ t-1\}$. In addition, TDNN also takes the parameter W_k for the kernel function K as shown in Eq. (6.2). The second component in the model is a PDE solver, which solves the input of the PDE system using Eq. (6.4)(a). In the next step, the Integral encoder converts the output of the PDE system into an

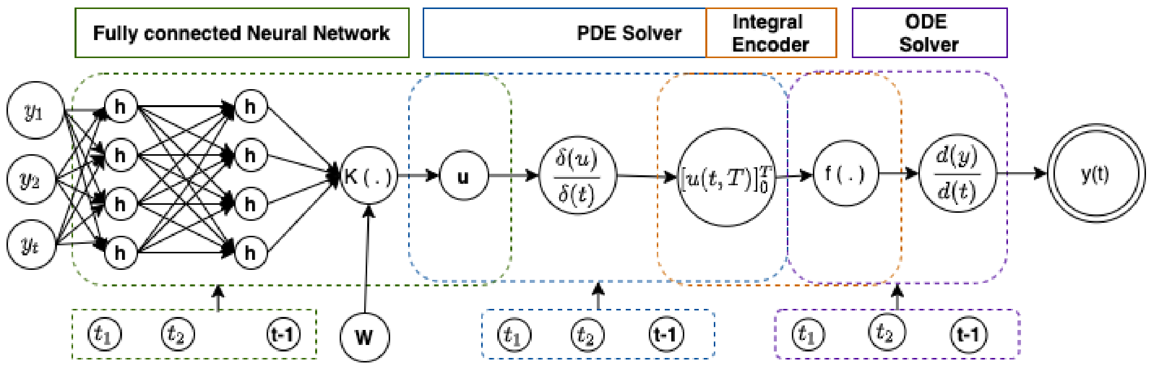


Figure 6.3: The architecture diagram of proposed Continuous CNN model

integral form as $u(t,T)$, which is a suitable continuous equation and can be passed to an ODE solver similar to NODE models. Finally, the ODE solver in Fig. 6.3 solves the continuous signal and generates the output y_t. The proposed CCNN model also handles high dimensional inputs by leveraging a similar technique as *NODE* models. For example, if $y(t) : \mathbb{R}^n$ for $n \geq 2$, i.e., a vector, then the system immediately generalises by adding some different indices for the missing values at different time step and thereby convert the discrete and irregular time series to a continuous one.

6.4 Implementation

We have leveraged the neurodiffeq [14] library to implement the proposed model. For the implementation, we have a PDE system described as Eq. (6.6), and an ODE system described as Eq. (6.7). Using an exceptional architecture TDNN neural network with the capabilities of using a PDE solver and ODE solver to reduce the delay in the time series, we can have solutions for both systems.

6.4.1 Challenges

We need some additional sophisticated components to design the model for the following challenges.

- The $\frac{\partial}{\partial \tau} u(t,\tau) = K(\tau, W_K, t) y(t-\tau)$ equation in Eq.(6.4)(a) has a shift, $t-\tau$, which can change dynamically over time and increase the delay. Standard CNN architecture finds it difficult to solve discrete time series. We have considered re-parameterising u, with a change of variables (y), to eliminate the delay. It is okay to shift the t argument to K, since K is fixed for the solver of the PDE system.

- A proper mechanism for driving term for external input, instead of burying it in the initial conditions. Therefore, we have used the optimiser and model parameter as additional inputs to the model, as shown in Fig. 6.3.
- CCNN model uses standard error function which works fine for the simple heat equation and signals with one-dimensional chirps. However, we need to design error function and adjoint system for complex time series. In future work, we will implement an adjoint ODE solver. For now, we have leveraged the ODE solver from Neurodiffeq [14] library.

6.4.2 Simulation with Chirps Signal

At this stage, we construct a one-dimensional input containing a "chirps" [13] which needs to be detected against a background white Gaussian noise. The shape of chirps varies systematically with time due to the nature of the white Gaussian noise. Fig.6.4 shows the original signal in the blue curve and the noisy signal in the orange signal. The chirps can be easily detected with a time-varying matched filter. The kernel function K of the proposed CCNN model converges to the correct time-varying matched filter over time. The performance evaluation of the CCNN model shows that with learning, K can eventually detect the chirps out of the input signal.

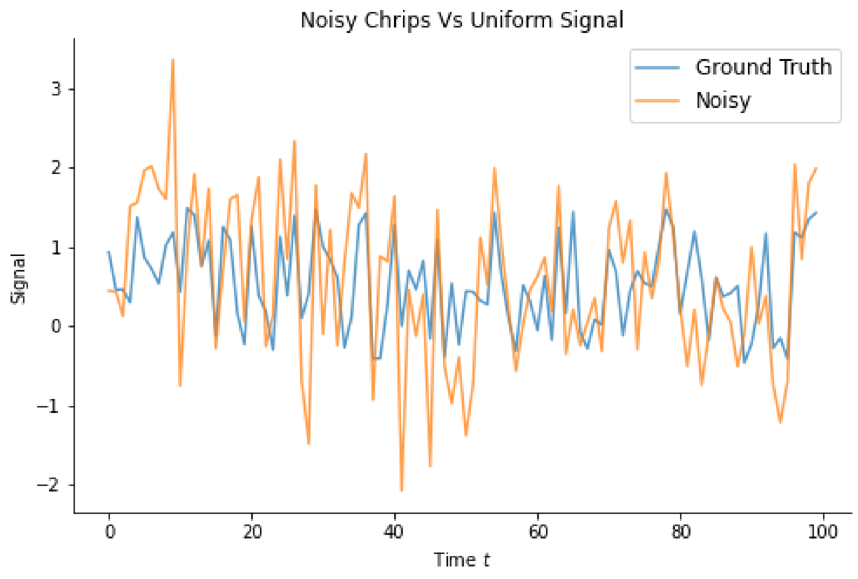


Figure 6.4: One-dimensional input containing "chirps" for proposed model

The ODE system solver function can be described as algorithm 11. The ODE solver can solve the input signal as an ODE equation over time steps t. During the process, the ODE solver can be parametric by using the internal delay for the TDNN model, weight and bias for the optimiser of the neural network.

Algorithm 11 ODE System Solver Function

Input:: input signal (y),time steps (t), internal (τ) and weight (W_k) as well as bias parameters (b)

```
1: procedure ODE_SOLVER($y, t, parameters = (\tau, W_k, b)$)
2:     return $ODESOLVER(y, t)$
```

Similarly, the PDE system solver function can be described as Algorithm [12]. Here $kernel_function$ is the function shown in Eq. (6.6) to generate the integral form of the signal. The ODE solver function described in Algorithm [11] is used in Algorithm[12] to define the coupled system architecture for the proposed CCNN model.

The proposed model uses both an ODE solver and PDE Solver. Appendix 5 shows the implementation of the proposed CCNN model. Fig. 6.5 shows the training loss for the noisy signal in Fig. 6.4.

Algorithm 12 PDE System Solver Function

Input:: input signal (y),time series (t), internal (τ) and weight (W_k) as well as bias parameters (b)

```
1: procedure PDE_SYSTEM_SOLVER_FUNCTION($y, t, parameters$)
2:     $y \leftarrow kernel\_function(y, W_k, b)$
3:     $u, v \leftarrow ode\_solver(y, t, parameters)$
4:     return $[diff(u, t, order = 1) + diff(u, tau, order = 1)]$
```

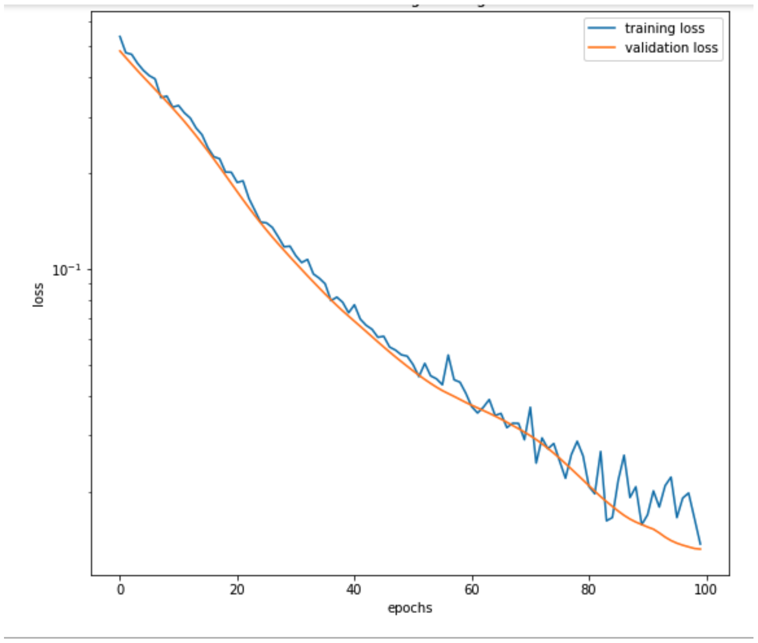


Figure 6.5: Training loss for proposed CCNN model One-dimensional input containing “chirps”

6.5 Performance Evaluation

TDNN is the foundation block of the proposed CCNN model, but this framework can be generalised to train multidimensional non-uniform data. We also have observed that the computational complexity is similar for CCNN and TDNN, where the runtime of standard CNN is much longer than the proposed CCNN as the training error for CCNN reduces drastically over iterations. This is because the nature of the input signal for CCNN is more continuous as it is processed using a coupled PDE an ODE system. Therefore, learning the change in the input time series is more manageable, so the corresponding error rate falls very quickly. Therefore, in most cases, CCNN requires less runtime than standard CNN.

Standard CNN models are less efficient for time series classification tasks as they require extensive training data sets for training. Therefore, the training runtime is also much longer for CNN than for other neural networks. Similar to other Neural ODE described in Chapter 3 and, Chapter 4. Therefore, the ODE system in the proposed CCNN does not require large training data sets for training. However, if the training data sets contain too much noise, CCNN may become unstable, and the optimisation may be disrupted. The training error becomes very much unstable in the presence of noise of more than 45%. For example, in the Chirps Signal task, the dataset was prepared with a noise of 40%. We also observed that the CCNN model could not work with a noisy signal with a noise level of more than 45%. The training error became unstable for such a noisy signal, and the CCNN model takes longer runtime to become stable again.

Similar to RNN and other traditional, CNN also uses fixed-length signals for training, where time series for different batches is fixed length discrete time series with a fixed step between consecutive observations. CNN prefers raw time series, as it requires encoding the fixed-length signals, but, in most cases, these training fixed-length signals cannot be further encoded into new representations as opposed to raw time series. Therefore, the data processing and encoding also occur during training runtime and increases the runtime significantly compared to RNN models. In addition, CNN also uses several pre-processing techniques, such as the sliding window, to make discrete signals continuous. However, the CCNN model encodes a PDE system using the integral of the kernel function $\mathcal{K}$ and in parallel, it also contributes to the optimisation of the parameter and reduces the training error. Therefore, the proposed model does not require fixed-length signals for training and additional pre-processing technique for encoding.

The error function is the controller for the training time reduction. Therefore, designing flexible and efficient adjoint error functions can improve the performance of CCNN models.

6.6 Discussion

Convolutional neural networks (CNNs) have shown enormous utility in various tasks where the data is evenly sampled, contains local features and exhibits translation invariance. Attempts to generalise them to situations where the data is irregularly sampled or varying scales have mainly been ad-hoc. Neural ODEs are a natural fit to irregularly sampled data but do not use filters or convolutions. We have attempted to combine the two ideas so that convolutions are continuous rather than discrete, length scales and other details of the convolution filters can vary, and data can be sampled irregularly. Because the convolutions introduce extra dimensions, the ODEs become PDEs, and the resulting system can be implemented using PDE solvers and optimised using standard adjoint constructions for PDEs. It might be speculated that other hardware substrates, such as chemical reaction-diffusion equations, might be used to implement these computations directly.

6.7 Conclusion

This chapter proposes a new variant for continuous Convolutional Neural Networks (CCNN) and evaluates its performance for processing noisy signals (chips). This model can also process high dimensional time series with more than one dimension. CCNN model leverages the strength of the Convolutional Neural Network (CNN) to process continuous data. In addition, CCNN helps to train PDE systems.

6.8 References

[1] Patrick Kidger, James Morrill, James Foster, and Terry Lyons. Neural controlled differential equations for irregular time series. arXiv preprint arXiv:2005.08926, 2020.

[2] Liu Yang, Dongkun Zhang, and George Em Karniadakis. Physics-informed generative adversarial networks for stochastic differential equations. arXiv preprint arXiv:1811.02033, 2018.

[3] Ehsan Haghighat and Ruben Juanes. Sciann: A keras wrapper for scientific computations and physics-informed deep learning using artificial neural networks. arXiv preprint arXiv:2005.08803, 2020.

[4] Georgios Kissas, Yibo Yang, Eileen Hwuang, Walter R Witschey, John A Detre, and Paris Perdikaris. Machine learning in cardiovascular flows modeling: Predicting arterial blood pressure from non-invasive 4d flow mri data using physics-informed neural networks. Computer Methods in Applied Mechanics and Engineering, 358:112623, 2020.

[5] Mansura Habiba and Barak A Pearlmutter. Neural ordinary differential equation based recurrent neural network model. arXiv preprint arXiv:2005.09807, 2020.

[6] Mansura Habiba and Barak A Pearlmutter. Neural odes for informative missingness in multivariate time series. arXiv preprint arXiv:2005.10693, 2020.

[7] David M Kreindler and Charles J Lumsden. The effects of the irregular sample and missing data in time series analysis. Nonlinear dynamics, psychology, and life sciences, 2006.

[8] Tian Qi Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural

ordinary differential equations. In Advances in Neural Information Processing Systems, pages 6572–6583, 2018.

[9] Daniel Neil, Michael Pfeiffer, and Shih-Chii Liu. Phased lstm: Accelerating recurrent network training for long or event-based sequences. In Advances in Neural Information Processing Systems, pages 3882–3890, 2016.

[10] Johann Lang. On illuminations of c^2-surfaces in vector graphic description. Comput. Graph., 12(1):33–38, 1988. doi: 10.1016/0097-8493(88)90005-2. URL https://doi.org/10.1016/0097-8493(88)90005-2.

[11] Juliet Perdigón-Morales, Rosario Romero-Centeno, Paulina Ordóñez Pérez, and Bradford S Barrett. The midsummer drought in mexico: perspectives on duration and intensity from the chirps precipitation database. International Journal of Climatology, 38(5):2174–2186, 2018.

[12] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. Neural computation, 1(4):541–551, 1989.

[13] Yuehjen E. Shao and Shih-Chieh Lin. Using a time delay neural network approach to diagnose the out-of-control signals for a multivariate normal process with variance shifts. Mathematics, 7(10), 2019. ISSN 2227-7390. doi: 10.3390/math7100959. URL https://www.mdpi.com/2227-7390/7/10/959.

[14] Feiyu Chen, David Sondak, Pavlos Protopapas, Marios Mattheakis, Shuheng Liu, Devansh Agarwal, and Marco Di Giovanni. Neurodiffeq: A python package for solving differential equations with neural networks. Journal of Open Source Software, 5(46):1931, 2020.

CHAPTER 7

HEUNNET: EXTENDING RESNET USING HEUN'S METHOD

This chapter is an expanded version of a paper published in: 2021 32nd Irish Signals and Systems Conference (ISSC)

There is an analogy between the ResNet (Residual Network) architecture for deep neural networks and an Euler solver for an ODE. The transformation performed by each layer resembles an Euler step in solving $dx/dt = f(t,x)$, namely $x_{t+1} = x_t + hf(t,x_t)$. We consider the Heun Method, which involves a single predictor-corrector cycle, $\tilde{x}_{t+1} = x_t + hf(t,x_t); x_{t+1} = x_t + h(f(t,x_t) + f(t+1,\tilde{x}_{t+1}))/2$, and complete the analogy, building a predictor-caorrector variant of ResNet, which we call a HeunNet. Just as Heun's method is more accurate than Euler's, experiments show that HeunNet achieves high accuracy with low computational (both training and test) time, in comparison to both vanilla recurrent neural networks and other ResNet variants.

7.1 Introduction

A significant limitation of recurrent neural networks is that they can access past states but not future states [1]. Bi-directional recurrent neural networks (BRNN) [2] overcome this limitation by computing two distinct RNN hidden layers with similar output in opposite directions. Bi-directional recurrent neural networks have the capabilities to access both the past and future. Each pass of a Bi-directional RNN consists of a forward pass and a corresponding backward pass. The training time for BRNN is similar to other recurrent neural networks such as LSTM. Instead of computing the output of a BRNN in two different directions, Heun Methods [3] compute the previous and next step at the same time, and optimise the current state based on those two past and future states.

A ResNet or Residual Network [4] can be viewed as using Euler's Method to integrate a time-dependent ordinary differential equation. Since Euler's Method is wildly inaccurate, with error $= \mathcal{O}(\Delta t)$, one might conjecture that some of the power of a ResNet with many layers is being devoted to compensating for integration error introduced by this crude approximation. This suggests using a more accurate integration method, reducing the approximation error, allowing the same number of layers to perform more efficiently.

Here we use the Heun Method [3], which is much more accurate: error $= \mathcal{O}(\Delta t^2)$. In our HeunNet model, each block computes x_t, the cell state at time t, along with an initial estimate of the next cell state, x_{t+1}. The Heun Method finds x_t using both x_{t+1} and an initial estimate of x_t. This approach helps the proposed HeunNet model to provide better result with a lower memory footprint.

Using multiple neural networks in a hybrid model is not a new concept. Several hybrid models, such as BRNN and Generative Adversarial Networks (GANs), have succeeded significantly. Recently, researchers have been finding RNN-based hybrid models to be very efficient for various applications. The main reason for the success of RNNs is their ability to store memory from previous states. However, such memory requires additional mechanisms and storage. Therefore, it is essential to optimise the memorising mechanism. This paper proposes a new technique to optimise the mechanism by which this neural network can depend on the previous state.

The fundamental block of each recurrent neural network is its memory block, which holds the previous state. This capability helps a Recurrent Neural Network (RNN) fight against the vanishing gradients problem. Similar to RNN, ResNet architectures do not have continuous memory blocks, and cannot hold values over a long duration.

This Heun Method approach can be adapted for a wide variety of layered neural network architectures. The HeunNet model only takes input for each state. It then computes both the current and the following cell states using the Heun Method, as shown in Eq. (7.5).

Each block of the HeunNet neural network uses the same weight and computation method. Therefore, it does not cause additional memory or computation energy. Although each iteration can take longer than simple neural networks, the result shows that the proposed neural network achieves higher accuracy than other neural networks at an early stage of training.

The main objectives of this work are:

- Design a new neural network architecture based on the Heun Method (HeunNet).
- Investigate different evaluation metrics to assess the performance of the proposed model.

7.2 Heun's Method for solving ODEs

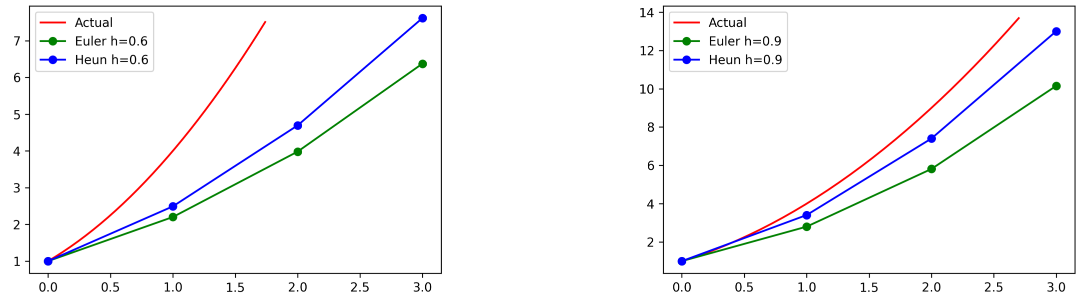


Figure 7.1: Euler vs Heun Method for $\dot{x} = 2\sqrt{x}$ with $x(0) = 1$, and $h = 0.6$ (left) or $h = 0.9$ (right). The analytic solution of this ODE is $x(t) = (t + 1)^2$ (red curve).

Heun's Method is a numerical method for solving ordinary differential equations (ODEs). Heun's Method is sometimes referred to as an "improved Euler" method, but it is better viewed as similar to a two-stage Runge-Kutta method [3]. For solving an

ODE $\dot{x} = f(t, x)$ with initial condition $x(0) = x_0$ using Euler's method, we use the slope f at x_0 and move along the tangent line to x at x_0 to reach to estimate x_{t+1},

$$x_{t+1} = x_t + hf(t, x_t) \tag{7.1}$$

where h is the step size. If the actual solution is convex (or concave), the Euler method will underestimate (or overestimate) the next state of the system. So in the long term, the numerical solution diverges from the correct solution. A better estimate can be obtained using an estimate of the slope at the next step and using the average of this slope with the current step's slope. This is the essence of Heun's Method,

$$\tilde{x}_{t+1} = x_t + hf(t, x_t) \tag{7.2a}$$

$$x_{t+1} = x_t + h\frac{f(t, x_t) + f(t+1, \tilde{x}_{t+1})}{2} \tag{7.2b}$$

In predictor-corrector methods like this there are two steps, "prediction" and "correction." A "predictor" (7.2a) estimates the value at the next step, and this estimate is improved by a "corrector" (7.2b). The Euler and Heun Method solutions for the ODE $\dot{x} = 2\sqrt{x}$ with initial condition $x(0) = 1$ are shown in Fig. 7.1.

7.3 Motivation

Consider a neural network with L layers such that the number of neurons in all the layers is equal. Let X_0 be the network's input and let W_0 be the square matrix (since all layers have the same number of neurons) of weight between the input and the first layers. Let $Z_1 = W_0X_0$ and $X_1 = \sigma(Z_1)$. So we define $\mathcal{F}(X_k) = \sigma(W_kX_k) = X_{k+1}$. The regular architecture of a neural network is straightforward, as shown in Eq. (7.3):

$$X_{k+1} = \mathcal{F}(X_k) \tag{7.3}$$

A residual neural network (ResNet) uses skip connections to deal with the problem of vanishing gradients during the training.

$$X_{k+1} = X_k + \mathcal{F}(X_k) \tag{7.4}$$

This architecture is quite efficient when dealing with an intense neural network, even with convolutional layers. However, if we look closely, we realise that this is similar to Euler's Method for solving an ODE with step size $h = 1$ [5]. So the motivation is that ResNet is a modification of the Euler method, so HeunNet is the modification of Heun's Method.

7.4 Proposed Model: HeunNet Model

Our motivation is inspired by the Heun Method [3]. Since ResNet [4] is a discretisation of the Euler method, we could see the Heun Method as the extension of ResNet. In a nutshell,

$$\boxed{\text{ResNet : Euler's Method :: HeunNet : Heun's Method}}$$

We propose another architecture for neural networks motivated by the Heun Method for solving ODE. In our model, the output of the next layer is a combination of the output from ResNet and the output of the previous layer as shown in Eq. (7.5a).

$$\tilde{X}_{k+1} = X_k + \mathcal{F}(X_k) \tag{7.5a}$$

$$X_{k+1} = X_k + \tfrac{1}{2}(\mathcal{F}(X_k) + \mathcal{F}(\tilde{X}_{k+1})) \tag{7.5b}$$

Fig. 7.2 shows the essence of this architecture.

Residual Neural Network is a discretization of the Euler method. In the residual network, the state of the layer $k + 1$ is related to the state of the layer k as shown in (7.4), where $\mathcal{F}$ can be an arbitrary layer transition function. But if we use Heun's

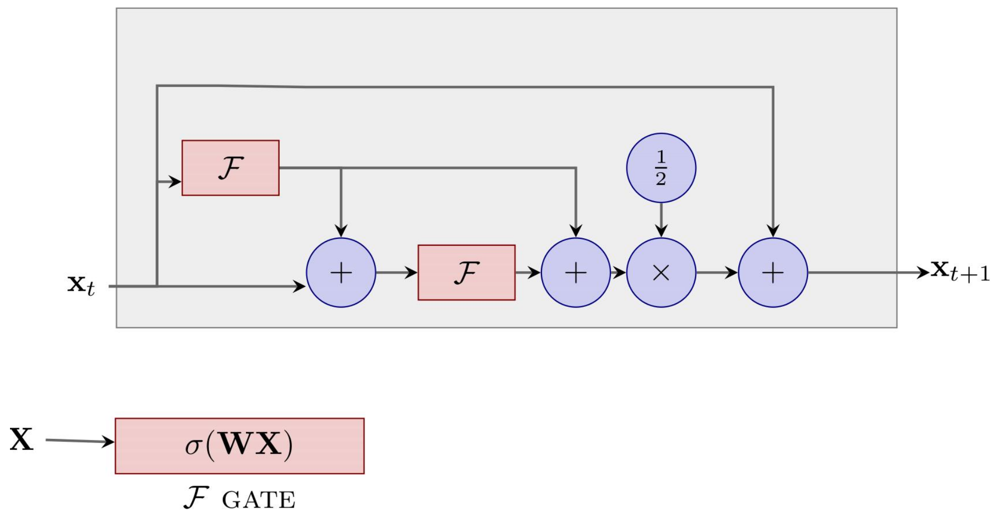


Figure 7.2: The architecture of proposed Heun based Recurrent Neural Network (HeunNet)

Method for solving $\dot{x}(t) = \mathcal{F}(x(t))$ with initial condition $x(0) = x_0$ and let step size be $h = 1$, we obtain the architecture of the Heun Neural Network (HeunNet), i.e., Eq. (7.5).

The Heun Method has quadratic accuracy, while the accuracy of Euler's Method is linear. For this reason, the Heun Method is generally preferred to Euler's Method when numerically solving ODEs. A similar situation occurs for the Heun Neural Network (HeunNet), which appears to converge to a more accurate result in fewer layers.

7.4.1 Gradient Propagation in a HeunNet

The original motivation for ResNet was to avoid the vanishing gradient problem by causing the Jacobian of the transition between layer k, and layer $k+1$ to be a near-identity matrix. Both the ResNet equation (7.4) and the analogous HeunNet equation (7.5) are of the form $X_{k+1} = X_k + \textit{something}$, which accomplishes this. Like ResNet, HeunNet is an entirely feedforward computational process that can be expressed using standard numeric linear algebra routines. It is straightforward to implement in any deep learning framework, such as PyTorch. Furthermore, the gradient can be automatically calculated via backpropagation, i.e., reverse mode Automatic Differentiation [6], and that process is just as automatic as for ResNet.

Despite its conceptual similarity, and as we see below, numerical simulations show that HeunNet obtains better performance than ResNet across various tasks.

7.5 Extension of HeunNet Model

In the Heun's Method for solving ODEs, if we put more weights on the corrector's second term we will have a better approximation of the actual function. In the Heun's Method, we first compute the $f(x_k)$ as the slope of the current state and $f(\tilde{x}_{k+1})$ as the approximation coming from Euler method and get the average of these two as the corrector. But if we put more weights on the $f(\tilde{x}_{k+1})$ and use convex hull of $f(x_k)$ and $f(\tilde{x}_{k+1})$, i.e., $(1-\alpha)f(x_k) + \alpha f(\tilde{x}_{k+1})$ with α near to 1, then we have better approximation than Heun's Method:

$$\tilde{x}_{k+1} = x_k + h\, f(x_k) \tag{7.6a}$$

$$x_{k+1} = x_k + h\left((1-\alpha)f(x_k) + \alpha f(\tilde{x}_{k+1})\right) \tag{7.6b}$$

where $0 \leq \alpha \leq 1$ and h is the step size. Fig. 7.1 shows the difference between Heun's Method and extended Heun's Method for $\dot{x} = 2\sqrt{x}$ with $x(0) = 1$.

For $h = 1$ we obtain the architecture of a Neural Network which is an extension of HeunNet (ExtendedHeunNet), i.e.,

$$\tilde{x}_{k+1} = x_k + f(x_k) \tag{7.7a}$$

$$x_{k+1} = x_k + (1-\alpha)f(x_k) + \alpha f(\tilde{x}_{k+1}) \tag{7.7b}$$

- If $\alpha = 0$ then $x_{k+1} = \tilde{x}_{k+1}$ and we have **ResNet**.

- If $\alpha = \frac{1}{2}$ then we have **HeunNet**.

- If $\alpha \neq \frac{1}{2}$ then we have **ExtendedHeunNet**.

The question is, which values of α give us a better approximation? Does this modification improve the performance of HeunNet? Since we need more weights on the next state to better approximate the actual function, we choose α to be close to 1. The architecture of this extended Neural Network is shown in Fig. 7.5.

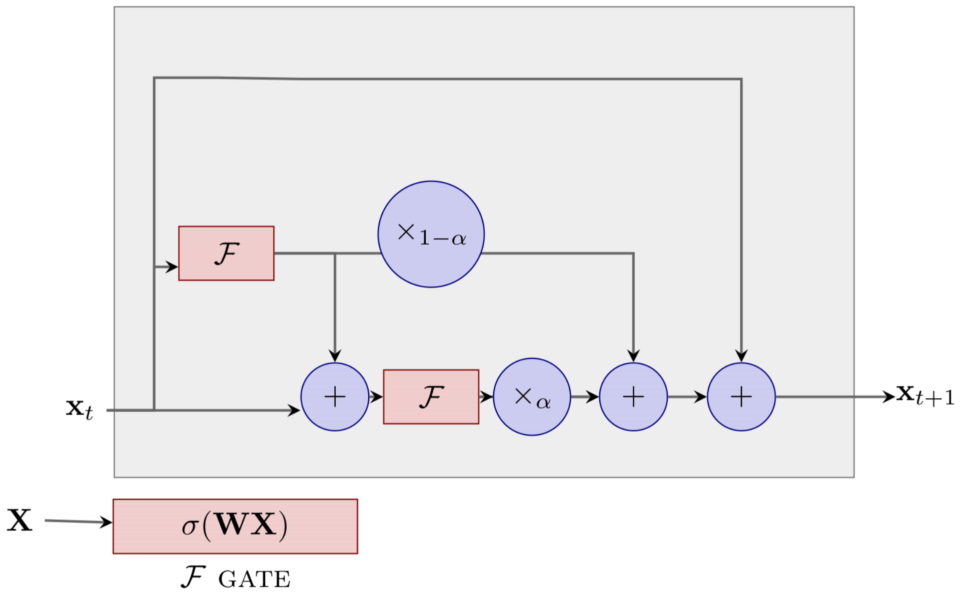


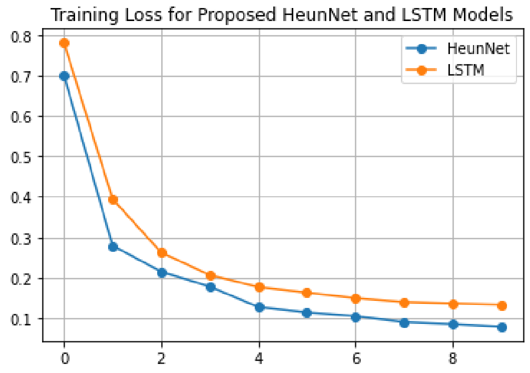
Figure 7.3: The architecture of proposed Extended Heun based Recurrent Neural Network (ExtendedHeunNet)

7.6 Results

For performance evaluation of proposed new model, we have chosen two tasks: (i) MNIST dataset classification [7], and (ii) ECG classification [8]. We compared the HeunNet model with other Recurrent Neural Network, e.g., LSTM [9] and GRU [10].

7.6.1 MNIST classification Task

Fig. 7.4 shows the comparative analysis of loss and accuracy for LSTM and HeunNet models over ten iterations. LSTM achieves an accuracy of 95.44% after the 10th iteration. On the other hand, the HeunNet model achieves an accuracy of 98.26% at the 10th iteration.



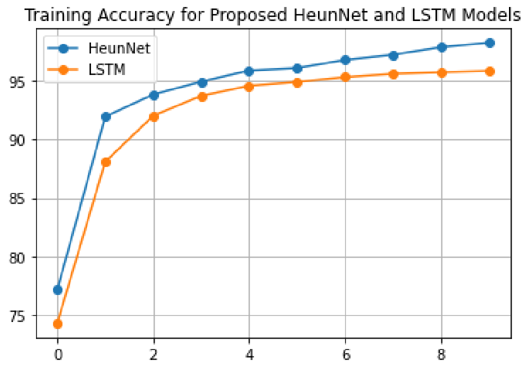


Figure 7.4: HeunNet and LSTM model loss (left) and accuracy (right) for 10 iterations on MNIST classification task

7.6.2 ECG heartbeat classification Task

For this Task, we used the dataset for MIT-BIH Arrhythmia Database v1.0.0 PhysioNet [11]. Table 7.1 shows some characteristics of the used datasets. This dataset consists of signals corresponding to electrocardiogram (ECG) shapes of heartbeats for the normal case and shapes caused by arrhythmias and other myocardial infarctions. There are five categories of beats present in the dataset as Nonectopic beat (N), Supraventricular ectopic beat (S), Ventricular ectopic beat (V), Fusion beat (F) and Unknown beat (Q).

Fig. 7.5 shows the loss and accuracy for LSTM model over 100 iterations. The best iteration for the LSTM model in the ECG heartbeat classification task is 79, with an accuracy of 90.40%. With the HeunNet model, we get the best accuracy of 98.80%. Fig. 7.5 shows the loss and accuracy for LSTM model for only 50 iterations. The

Table 7.1: Parameters of Dataset Used

#	**Parameter**	**Value**
1	Number of Samples	109446
2	Number of Categories	5
3	Sampling Frequency	125 Hz
4	Classes	[N: 0, S: 1, V: 2, F: 3, Q: 4]

proposed model can achieve higher accuracy within only half of the iterations required for other recurrent neural networks.

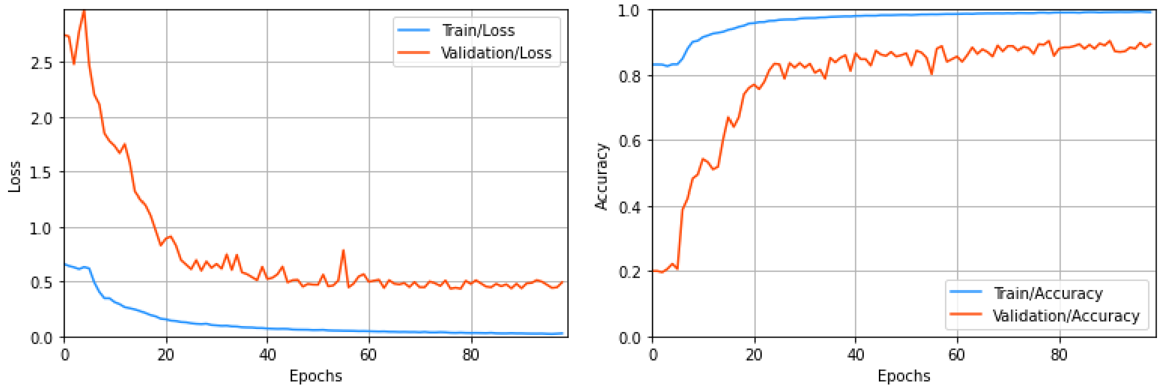


Figure 7.5: LSTM model Loss (left) and accuracy (right) for 100 iterations on ECG heartbeat classification task

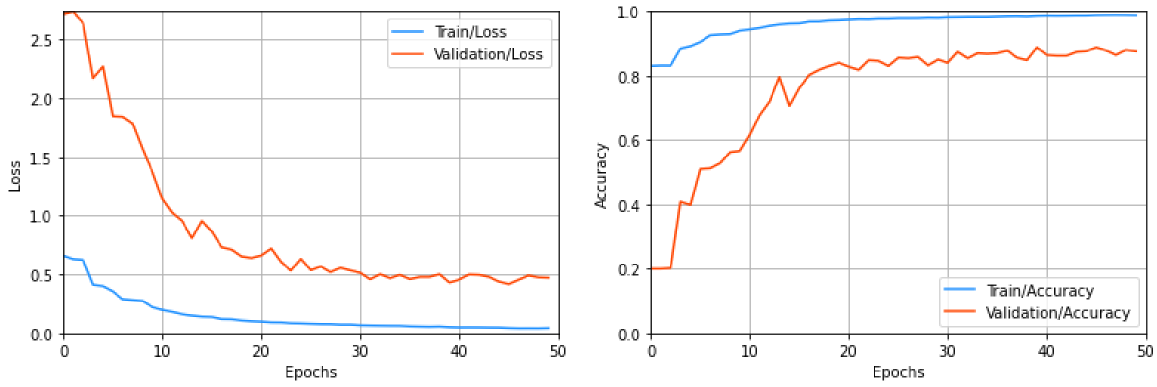


Figure 7.6: HeunNet model loss (left) and accuracy (right) for 50 iterations on ECG heartbeat classification task

7.6.3 Time Series prediction

As shown in (7.5), the neural network $\mathcal{F}$ can be any neural network. For time series generation task, we used a simple LSTM neural network for $\mathcal{F}$ in (7.5). This Heun neural network-based LSTM model is compared against Phased LSTM [12] and vanilla LSTM model. A sine wave of length 16π is used for the dataset for this Task. Each model is trained for 100 iterations. Fig. 7.7 shows the comparative performance analysis for three models. As shown in Fig. 7.7, the HeunNet model outperformed the other two model.

The proposed ExtendedHeunNet model's prediction is more accurate than other HeunNet and other neural network models. ExtendedHeunNet provides an accuracy of 97.30% with $\alpha = 0.8$ in just two iterations for Task C. For ExtendedHeunNet, $0.75 < \alpha \leq 0.8$ provides best accuracy. If α moves towards 1, the accuracy starts to drop.

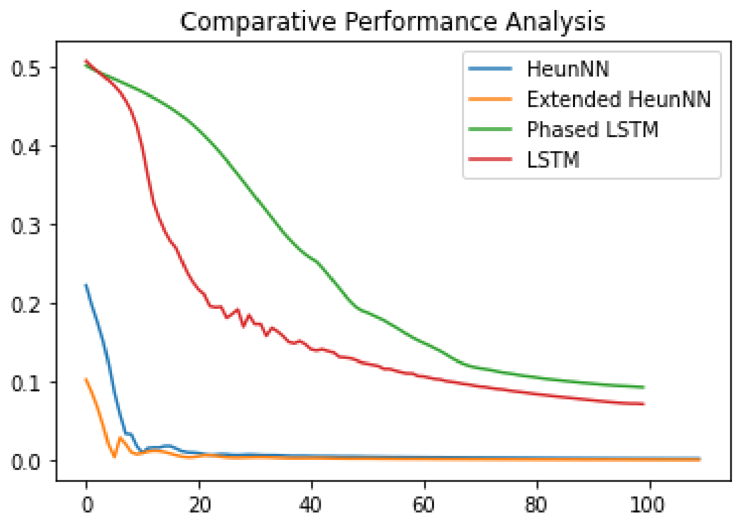


Figure 7.7: Loss for LSTM, Phased-LSTM, and HeunNet models over 100 iterations for Sine Wave generation

Fig. 7.7 shows generated sine wave for LSTM, Phased-LSTM and HeunNet model. The HeunNet and ExtendedHeunNet model's predictions are more accurate than the predictions of the other two models.

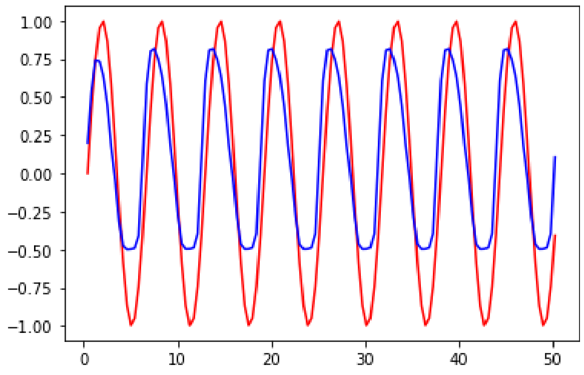


Figure 7.8: LSTM model trained for 100 iterations

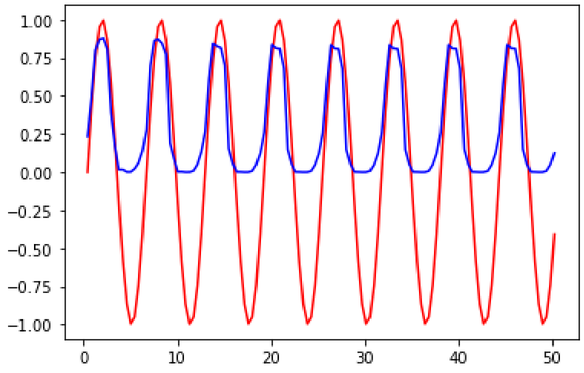


Figure 7.9: Phased-LSTM model trained for 100 iterations

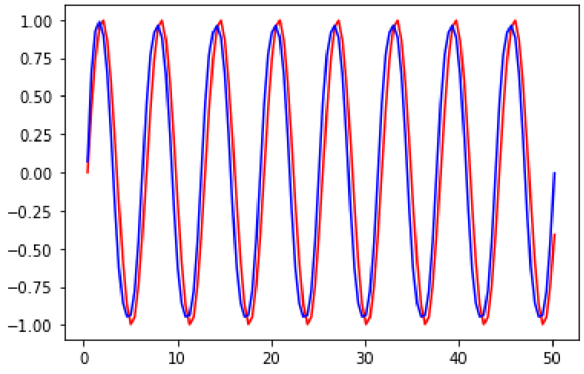


Figure 7.10: HeunNet model trained for 25 iterations

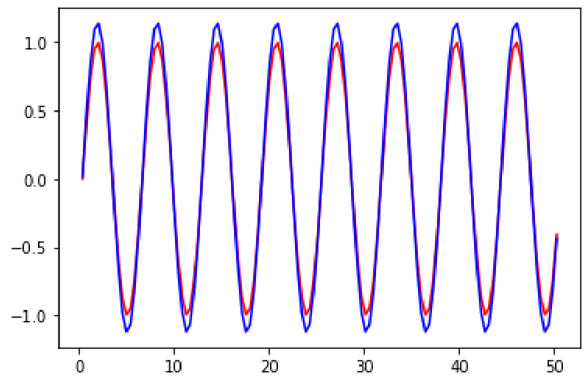


Figure 7.11: ExtendedHeunNet model trained for 2 iterations with $\alpha = 0.8$

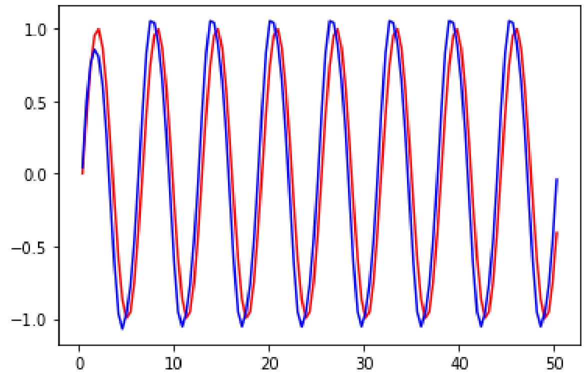


Figure 7.12: ExtendedHeunNet model trained for 10 iterations with $\alpha = 0.8$

7.6.4 Performance Evaluation

Typically, the accuracy of RNN models (LSTM or GRU) increase with the increase in the number of iteration. Therefore, the training time is comparatively long. I experimented with different ranges of iteration for different problems. For Task A, LSTM achieves 95.44% accuracy within ten iterations, and GRU achieves 92.01% accuracy. However, the HeunNet model achieves 97.98% accuracy for the same number of iterations. The result shows that the proposed model needs less training time to achieve high accuracy.

In a traditional bi-directional vanilla recurrent neural network, corresponding parameters are combined into a single LSTM. On the other hand, in this proposed model, the results of two individual neural networks are combined using the Heun Method (7.5).

In this section, both classification and time series prediction tasks are evaluated for different recurrent models against the HeunNet model. We can conclude that the proposed model can achieve higher accuracy with almost half the number of iterations than other models. The HeunNet model does not need to train for a long time to get the optimised result for time series synthesis, prediction and generation tasks. The proposed model takes significantly lower time as Phased LSTM and slightly higher than the LSTM model for the classification task. From several tests, we can conclude that proposed HeunNet takes less time to train than ResNet-50 for classification tasks. In addition, the proposed model is better suited for time-series synthesis, prediction, and generation, like Task for time series prediction. However, the proposed model performed better than other models for time series classification but not as good as it performed for the prediction task. We can conclude that proposed HeunNet model works better for time series synthesis, prediction and generation tasks.

7.7 Discussion

ResNet has the form of an Euler method for discretising and solving ODEs, and HeunNet and its extensions can be similarly viewed as having the form of Heun's Method. Some other groups have independently explored similar ideas for extending ResNet based on other numeric ODE solvers. In particular, MomentumNets [13] are constructed using an analogy with a numeric solver for second-order ODEs, while our HeunNet and its extension are derived from solution methods for first-order ODEs. Since second-order ODEs can be reduced to first-order ODEs using standard techniques, it should be possible to put these all into a common framework. Furthermore, momentumNets have ResNet as a particular case, but our extended HeunNet has both ResNet and HeunNet as exceptional cases. Finally, in MomentumNets, the velocity term for the first stage is defined as a function of the current state. For the second stage, the same term represents a combination of the input of the previous layer and the current layer. On the other hand, our HeunNet or its extension uses ResNet to capture the approximation for the next state and leverage correction to reduce the error introduced in the first step. As a result, the HeunNet makes an approximation for the next state and a weighted average of this approximation. Later, Huennet corrects the space of candidates for the next stage of the system.

Heun's Method falls in the general class of predictor-corrector methods: it makes an initial guess of the correct state of the next time step, then corrects that estimate using more accurate information based on this first estimate. This can be viewed as a local causality violation, as it makes the update between layers t and $t+1$ depend not just on the state of layer t, but also on that of layer $t+1$. However, the Method remains causal, unlike time-series processing techniques like a Bi-directional RNN, which consists of a forward-in-time pass along with a corresponding backwards-in-time pass. The technique proposed here might be naturally extended to that setting by making each of these passes update using a Heun's Method step rather than an Euler step. So we should not view these as in competition but rather as potentially synergistic.

7.8 Conclusion

We propose a new neural network model that leverages Heun's Method for hidden state optimisation. We also developed a general architecture that includes the HeunNet as a particular case. Performance evaluation demonstrates that both proposed models outperform recurrent neural networks on a few problems. The proposed models also take less time for training and achieve higher accuracy. These architectures leverage the strength of Heun's Method, which provides higher accuracy. The error generated by the first step of the Heun Method is compensated by the corrector, i.e., the second term, thus reducing the error [3].

7.9 References

[1] 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.

[2] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. IEEE transactions on Signal Processing, 45(11):2673–2681, 1997.

[3] Davide Batic, Harald Schmid, and Monika Winklmeier. The generalized heun equation in qft in curved spacetimes. Journal of Physics A: Mathematical and General, 39(40):12559, 2006.

[4] James D Keeler, Eric J Hartman, Kadir Liano, and Ralph B Ferguson. Residual activation neural network, October 4 1994. US Patent 5,353,207.

[5] Zhengping Che, Sanjay Purushotham, Kyunghyun Cho, David Sontag, and Yan Liu. Recurrent neural networks for multivariate time series with missing values. Scientific reports, 8(1):1–12, 2018.

[6] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. Journal of machine learning research, 18, 2018.

[7] Yann LeCun. The MNIST database of handwritten digits, 1998. http://yann.lecun.com/exdb/mnist/.

[8] Loren Kersey, Kat Lilly, and Noel Park. Ecg heartbeat classification: An exploratory study. In Proceedings of the Australasian Joint Conference on Artificial Intelligence-Workshops, pages 20–27, 2018.

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

[10] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. arXiv preprint arXiv:1406.1078, 2014.

[11] Ary L Goldberger, Luis AN Amaral, Leon Glass, Jeffrey M Hausdorff, Plamen Ch Ivanov, Roger G Mark, Joseph E Mietus, George B Moody, Chung-Kang Peng, and H Eugene Stanley. Physiobank, physiotoolkit, and physionet: components of a new research resource for complex physiologic signals. circulation, 101(23): e215–e220, 2000.

[12] Daniel Neil, Michael Pfeiffer, and Shih-Chii Liu. Phased lstm: Accelerating recurrent network training for long or event-based sequences. In Advances in Neural Information Processing Systems, pages 3882–3890, 2016.

[13] Michael E Sander, Pierre Ablin, Mathieu Blondel, and Gabriel Peyré. Momentum residual neural networks. arXiv preprint arXiv:2102.07870, 2021.

CHAPTER 8

AN ADAPTIVE DEPTH NEURAL NETWORK BASED ON ALGORITHMIC DIFFERENTIATION OF NUMERIC ITERATE-TO-FIXEDPOINT

This chapter is an expanded version of a paper published in the 1st International Conference on Electrical, Computer and Energy Technologies (ICECET), 2021

This chapter proposes a Deep Neural Network model called Temporal Wormhole Neural Network (TWNN) that can control its depth using an iterate-to-fixed-point operator. The architecture of TWNN starts with a standard layered network where often the connections are added to the higher layer, similar to a stranded Res-Net. On the other hand, the TWNN model adds connections to previous or lower layers. These backwards connections would dramatically slow the convergence of learning. As a result, TWNN adds gates to control information flow backwards. These gates control the flow of information among layers and can make backwards connections inactive under certain circumstances. Therefore, the adequate depth of the deep neural network is reduced, as is the training time. However, when the backward connections are active again, the depth of the TWNN model increases, and it takes a long time to settle down. In addition, the gradient calculation also becomes more laborious for a deeper neural network. Furthermore, the flow of information follows in a fashion that seems vaguely analogous to the afferent and efferent flow of information through layers of processing in the brain. We evaluate the proposed model on different long-term dependency tasks. The results are competitive with other studies, showing that the

proposed model contributes significantly to overcoming the vanishing gradient problem. At the same time, the training time is significantly shortened, as the “easy” input cases are processed more quickly than “difficult” ones.

8.1 Introduction

Neural networks use a different technique to memorize the previous state in a time series to learn the changes in the series over time. For example, traditional Memory Augmented Neural Networks (MANN) [1] as well as Recurrent Neural networks (RNN) preserve previous states in the model by explicitly storing previous hidden states in the "memory" buffer. However, if storing states in memory is too frequent, that can cause the "memory" to become very unstable. Moreover, in the beginning, the training gradients are unstable, and a significant fraction of "memory’ gets overwritten at each step during the training. Therefore, unstable "memory" causes the fast vanishing of memory and gradients.

Another well-known technique to preserve memory is wormhole connection [2]. Neural Turing machines [3]leverage these connections to provide a shortcut connection to the previous hidden state through time by explicitly storing the previous hidden state in the memory. Temporal Automatic Relation Discovery in Sequences (TARDIS) [2] shows that these connections created by the controller of the MANN can significantly reduce the effects of the vanishing gradients. Wormhole connections for MANN shorten the signal’s paths to travel between the dependencies. At every step of training, the controller ϕ in TARDIS, as shown in Eq. (8.1), controls the hidden state $\mathbf{h}_{t-1}$ based on the content of weights $\mathbf{w}_t$ and memory $\mathbf{M}_t$ read from external memory. In this work, we focus on storing only the influential memory of any previous state rather than storing all of them.

$$\mathbf{h}_t = \phi(\mathbf{x}_t, \mathbf{h}_{t-1}, (\mathbf{M}_t^\top \mathbf{w}_t^r)) \tag{8.1}$$

Some inputs can be more complicated than others because more computation is required to process them in processing data. For example, we can rapidly interpret an image of a snarling tiger, while a scene of Dalmatians in the snow may require considerable time to recognize. The recent construction of deeper and deeper models for image processing can make the system able to devote considerable computation to processing an image. However, due to their static nature, this level of processing is devoted to <u>all</u> inputs, even easy ones. The similarity of the computations in many layers also suggests that these intense networks may perform an iterative computation rather than a sequence of heterogeneous calculations. We attempt to address this by building a new class of networks which is iterative. It can be viewed as a recurrent network which massages the data repeatedly until it concludes. Alternatively, unrolling the computation can be viewed as an intense network that can terminate at any level and whose layers are identical. This seems vaguely similar to how biological nervous systems can take more or less time to interpret an input and are recurrent but not very deep.

On the other hand, a deep neural network, such as a Residual neural network, creates multiple layers (from 50 to 152 or more) of repetitive blocks. A ResNet with 100 layers provides 94% accuracy for the MNIST dataset. A recent work suggested 1001-layer deep ResNet architecture [4]. However, depth is not the only strength of this 1001-layer deep ResNet; identity mapping and better techniques to resolve vanishing gradient problems helped achieve a better result. Making a deep neural network is not always effective. Therefore, recent research works are focusing on Wide Residual Neural network [4], U-Net [5] and Neural ODE [6]. This proposed work is also based on a similar concept. Instead of making an even deeper model, we focus on the model's structure so that the same deep neural network model can act as a shallow and deep model based on the circumstances.

This work introduces a novel neural network model that uses Algorithmic Differentiation (AD) Transformation of the Numeric Iterate-to-Fixedpoint methodology to train each batch in the training datasets. In addition to dynamic training time for each layer, the proposed model can push backwards during training using a novel block in the architecture of a deep neural network. Some of the significant contributions of this work are

- A novel deep neural network that leverage Fixed point iteration which is also controlled by backward mode of Algorithmic differentiation (AD).
- New technique for training a deep neural network with the dynamic amount of time for different layers by creating a push-backwards connection with the previous layer of the model.

Models like RNN using Backpropagation through time require saving multiple copies of previous states. The fixed point iteration takes a busy time to settle down.

8.2 Background

The core concept of the proposed new model is to design a deep network that can have lower depth but higher accuracy. Instead of a fixed training time on each layer, this new model can dynamically change the training time for each layer. The target model also can have a low depth level, i.e. 50, but each layer can have a dynamic training time. Mainly, this new model trains each layer until it achieves an error less than or equal to a threshold value for local error, a.k.a tolerance error (ε). For example, instead of using a 256-layer ResNet, we can achieve similar accuracy by learning the data using a 50-layer ResNet for a longer time. It is well-known that a machine learning algorithm can reach the outcome sometimes right away, or sometimes it needs to think for a longer time. Based on the data, the proposed model can behave like a shallow or deep model. For each individual input x used in a deep learning model $y = f(x)$, the computation time of primal solution y varies. The computation time is short for some input x_a, where the corresponding solution can be achieved in time T_1. Similarly, for some input x_b, the computation time, T_2, where $T_2 \gg T_1$. As a result, a constant depth for the deep learning neural network model does not help to optimize the training. In addition, the depth of the model should also vary for different batch inputs during the training to achieve higher accuracy. Moreover, the memory requirement for different models is usually either constant such as Neural ODE [7] or constantly increasing overtime throughout the training period. In addition, models like RNN also overwrite the same memory too many times, making them expensive. Similarly, most deep neural network models adopt the forward technique where the training is uni-directional and skip connections are created to a future layer instead of a previous one. However, we find that learning a particular layer for a long time can achieve better accuracy. Therefore, it is essential to connect with the previous layer by adopting a push-backwards technique. Also its not always required to learn all layer of a model. As a result, having dynamic memory and training time for different layers is also essential.

Each layer is executed in a sequence for a traditional ResNet, or the IdentityBlock creates a skip connection between the current layer and the last ReLU layer. ResNet model always moves forward. Identity Block help to optimize the training for ResNet, but at the same time, it also has some limitation. Creating skip connection using Identity Block enable the gradient to avoid the main-stream flow of residual block weights, and it can avoid learning anything during training. Therefore, the training should be more stable, and the flow of gradients should be well defined. In addition, only a few blocks can represent the hidden dynamics of the dataset, while many other blocks contain very little information to learn useful representations. Therefore, learning only a few blocks for a long time can optimize the performance of deep neural network models. In this work, we introduce Fixed-point iteration [8] to control the learning or training duration for each block. Instead of learning each block, this new model focuses on blocks that provide good representations of the hidden dynamics of the data for a more extended period.

8.3 Model Design

A ResNet usually have three different blocks

1. basic - two consecutive 3×3 convolutions followed by a batch normalization and ReLU unit
2. bottleneck - one 3×3 convolution surrounded by dimensionality reducing and expanding 1×1convolution layers
3. identityBlock - one skip connection with input and the ReLU unit

The $Fixed - Point - Iterator$ block in Fig. 8.1 takes the input (x) and runs a fixed-point iteration loop until the error is less than or equal to the threshold for tolerance (ε). The output for $Fixed - Point - Iterator$ block is the output (x) and the following selected layer to be trained $\mathcal{L}$. If $(\mathcal{L})$ refers to a previous layer, the training restarted from that previous layer by creating a push-backwards connection between the selected layer $(\mathcal{L})$ and the main-stream for training. As shown in Fig 8.1, $Fixed - Point - Iterator$ block can send the training to the previous layer or can forward it to the next layer of a Residual neural network.

At every layer, a local objective function finds the fixed point. Eq. (8.2) shows the condition for the fixed point. This loop iterate until difference between consecutive values x_{t-1} and x_t of variable x is less than the tolerance (ε) or the number of iteration is more than max-iter (the maximum number of iteration for each Fixed-Point-Iterator block). $\mathcal{F}$ the actual numeric fixed-point, $z = x_\infty$, assuming that the objective function $g(\hat{A} \cdot b)$ has appropriate convergence properties. $\mathcal{F}$ continues looking for z until $\|\mathrm{x}_t - \mathrm{x}_{t-1}\|$ is less than some predefined threshold ε. This loop iterate until difference between consecutive values $x_{t-1} and xt$ of variable x is less than the tolerance (ε). Fig. 8.1 shows the block diagram of proposed model. Similar to traditional Res-Net, the proposed FWNN model has the following layers

- Residual Block
- Identity Block
- Normalization Layer (e.g. ReLU)

In addition to the above layers, this model also has Fixed-Iteration-Loop Block. The fixed-Iteration-Loop block is responsible for selecting the next Residual Block or Identity Block to be trained for the model. As the proposed model can move in both directions, therefore the next layer for training can be in either direction, as shown in Fig. 8.1. For example, if the first Fixed-Iteration-Loop Block in Fig. 8.1 produces the $x\_next\_layer = 0$, the first Residual Block will be trained again, and the training will continue. Similarly, for $x\_next\_layer = 2$, the next (third) layer will be trained. The main function of Fixed-Iteration-Loop Block in the proposed model is to identify the influential part of the dataset by configuring blocks of the proposed model.

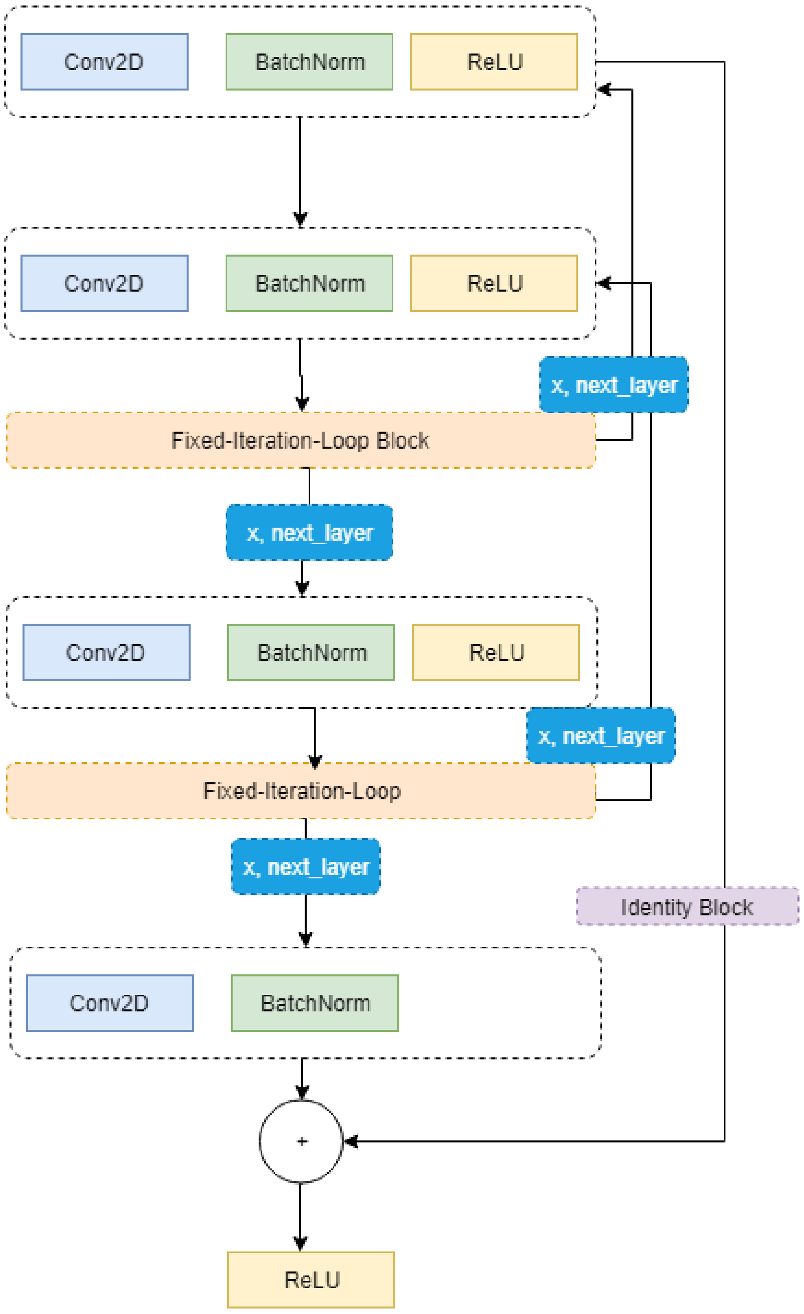


Figure 8.1: Block Diagram of architecture of push-backward technique for deep model

$$\begin{aligned}
&\text{for } t = 1, \ldots, \infty \; \{ \\
&\quad \mathrm{x}_t \leftarrow g\left(\mathrm{x}_t, \alpha, H_{t-1}, O_{t-1}\right); \\
&\quad \text{if } \|\mathrm{x}_t - \mathrm{x}_{t-1}\| \leq \varepsilon \text{ break; } \} \\
&\mathrm{z} = \mathrm{x}_t
\end{aligned} \tag{8.2}$$

Here g computes the cost for each step locally. Fig. 8.2(c) shows that g takes the input of any current state(x_t), weight(α), hidden matrix(H_{t-1}) and output (O_{t-1}) of previous step($t-1$) as input. For first iteration,t=1, $H_0 = null$ and $O_0 = null$, therefore, $H_1, O_1 = g(x_1, \alpha)$. Here, the hidden matrix $H1 = hidden1, hidden2, hidden3$ as shown in Fig. 8.2(a). The hidden matrix of previous step(t-1) is used as input for g in current step (t). Fig. 8.2(b) shows g computes the output for previous step (t-1), which is used as input for current step t as shown as shown in Fig. 8.2(c). The loop $t = 1, \ldots, \infty$ iterates until z settle down. However, with the "wormhole" reverse connections as shown in "green" color in Fig. 8.2, the execution of g may iterate a bunch of times for z to settle down.

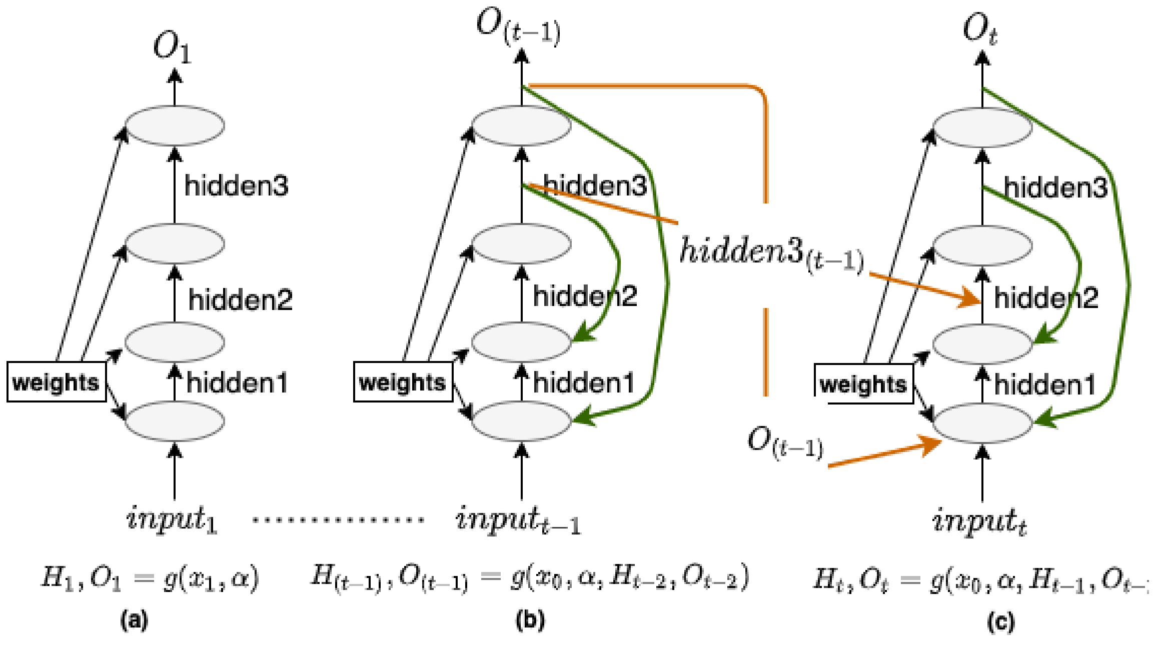


Figure 8.2: The workflow for objective Neural Network g in Eq. (8.2)

Fig. 8.2 shows that the sequence of three blocks repeats for a ResNet model based on the number of layers. These redundant networks can be described as Eq. (8.3).

$$x_D = \sum_{i=d}^{D-1} \mathcal{F}(x_i, W_i) \tag{8.3}$$

Here, $\mathcal{F}$ is a residual function. Between the shallow layer (d) and deep layer (D), some layer (Dw) contribute to the final result more than other layers (Dp). If a ResNet model is of depth D with the number of layers= D, Eq. (8.4) shows the distribution of layers of the ResNet model.

$$D = \sum_{j=0}^{P} Dw + \sum_{j=0}^{K} Dp \tag{8.4}$$

Here, let us consider layers denoted by Dw, and learn valuable representations of the hidden dynamics of the system, where layers in ResNet model architecture, denoted by Dp, provide very little information with a small contribution to the final goal for learning the system. To optimize the model's performance, it is essential to identify layers (Dw) with good representations of the hidden dynamics. In this work, we introduce a new model that can identify the Dw layers and learn the system's hidden dynamics by learning these layers only. This new model has a $fixed-point-iterative$ operator to control the training for the (Dw) layers. The $fixed-point-iterative$ operator continues training for each (Dw) layer until one of the following two conditions are met:

- The number of iteration for each block training exceeds the max_iter
- the corresponding training loss is less than tolerance error (ε).

This process a new block called $Fixed-Point-Iterator$ block in the architecture of the ResNet Model as shown in Fig. 8.1. $Fixed-Point-Iterator$ block trains the current Dw layer. Once the training for the current Dw layer is complete in the $Fixed - Point - Iterator$ block, identify the next layer $\mathcal{L}$ and create a connection between $\mathcal{L}$ layer and the mainstream of training. $Fixed - Point - Iterator$ block uses $fixed - point - iterative$ operator to control the training for the current Dw layer and enforce the conditions as mentioned earlier to stop the loop. The $fixed - point - iterative$ operator leverage Banach Fixedpoint finder ***fix*** :$(\beta \to \beta) \to \beta$ that takes a contraction of a closed region which contains the initial point. The reverse AD transform has a signature as shown in Eq. (8.5).

$$\begin{gathered} \overleftarrow{\mathcal{J}} : (\alpha_1 \to \cdots \to \alpha_n \to \beta) \to \\ \alpha_1 \to \cdots \to \alpha_n \to \\ (\beta \times (T^*\beta \to (T^*\alpha_1 \times \cdots \times T^*\alpha_n))) \end{gathered} \tag{8.5}$$

The forward and reverse transform of ***fix*** are shown as (8.6).

$$z = \text{fix}(fx) \quad f : \alpha \to \beta \to \beta \tag{8.6}$$

$$\begin{aligned} Tz &= \text{fix}(\overrightarrow{\mathcal{J}} f(Tx)) & \bar{f} : T^*\beta \to (T^*\alpha \times T^*\beta) \\ T^*x &= \rho_1 \left(\text{fix}(\underbrace{\rho_2(\overleftarrow{\mathcal{J}} fxz)}_{\bar{f}} \circ (+\,(T^*z)) \circ \rho_2) \right) & \end{aligned} \tag{8.7}$$

Here in Eq. (8.6), f, is a Neural Network. The weight to network (f) is defined by α. β represents the activity of all input. Therefore, $(\beta \to \beta) \to \beta$ can be used to compute the next moment activity. Table 8.1 describes different parameters in Eq. 8.5, Eq. 8.6 and Eq. 8.7.

Fig 8.3 explains the computational graph for a single computational block of proposed

TWNN based on Eq. (8.5). x is the input passed to the neural network (f) or residual block in the proposed TWNN model. The gradients of all activity of the model (u, T^*u) are generated as the output of neural network (f) during the forward pass of the model. The output for neural network (f) is y. The updated parameter for neural network (f) is z. During the backward pass, z, T^*u, y are used as input for the backward layer $(\bar{f})$, which generates the gradients T^*x and T^*y w.r.t input x and output of forward pass y.

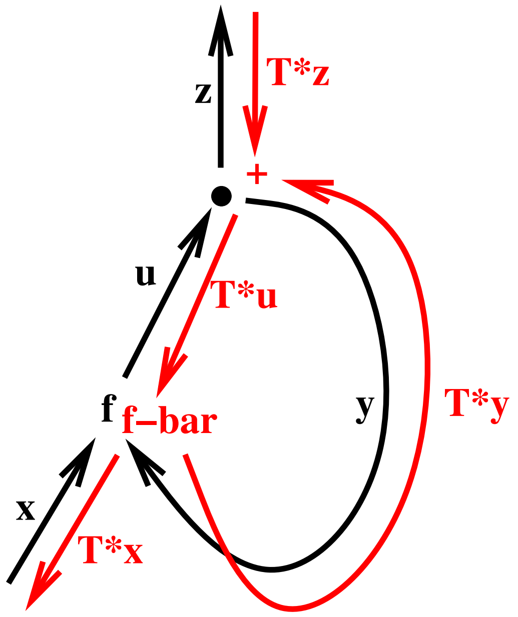


Figure 8.3: Forward and backward computation of proposed model

The gradient for each fixed-point iteration loop for each layer is also calculated over the same number of iterations in a loop.

As shown in Eq.(8.6), the proposed Temporal Wormhole Neural Network (TWNN) uses Fixed-point iterations [8]. For each input x, the forward pass of TWNN continue iterate until the output z in Eq.(8.6) settles. Fig 8.4 shows the loop for each fixed-point iteration for each batch input x. At the end of each loop, the controller *cntrl* checks either z settles by checking if the conditions in Eq (8.2) are met.

Table 8.1: The Notation Used in Temporal Wormhole neural Network

$\overleftarrow{\mathcal{J}}$	Forward mode Algorithmic differentiation(AD) of Network f
$\overrightarrow{\mathcal{J}}$	backward mode Algorithmic differentiation(AD) of Network f
α	Weight to the proposed network
β	Activity to the proposed network
f	Neural Network
$\bar{f}$	Neural Network for efferent connection
$T^{*}z$	Gradient of all the activity with respect to z
$T^{*}\alpha$	Gradient of all the activity with respect to α
$T^{*}\beta$	Gradient of all the activity with respect to β
$T^{*}x$	Gradient of all the activity with respect to x

Fig. 8.4 shows the flow diagram of the Fixed-Point iteration block of the proposed TWNN. Here x is the input for the model. The res-net model f generates the gradient of all activity w.r.t and inputs and the parameter (α), such as $T^{*}X$ and $T^{*}\alpha$ using Eq.(8.6). There is fixed-point loop in the model which starts f and ends in $\bar{f}$. We consider the flow of computation from f to $\bar{f}$ as the forward pass and the flow of computation in the opposite direction from $\bar{f}$ to f as the backward pass. The parameter (α) is optimized to β by the gradients $T^{*}z$ and $T^{*}\alpha$. On the other hand, during the backward pass, the gradients $T^{*}X$ and $T^{*}\beta$ optimized the neural network (f). The fixed point loop of computation between f and $\bar{f}$, is controlled by the local tolerance error (ε), which is optimized by the global error and loss function.

Algorithm 13 explains the training method for proposed TWNN, the *model* in algorithm 13 can be any Neural Network block such as GRU [9], ResNet [10] and others. The main training function for the proposed model is exercise training data in batches. Therefore inputs are divided into different batches. There are local errors and global errors in the training model. The tolerance error (ε) variable in the algorithm 13 indicates the local error for training each batch. $initializeNNBlock$ function initialize the proposed model with initial hyperparameter α. The actual training of the model is executed in several batches. For each batch, the model's correct Residual Block is selected using the *next* function. Then the model will be trained for each layer.

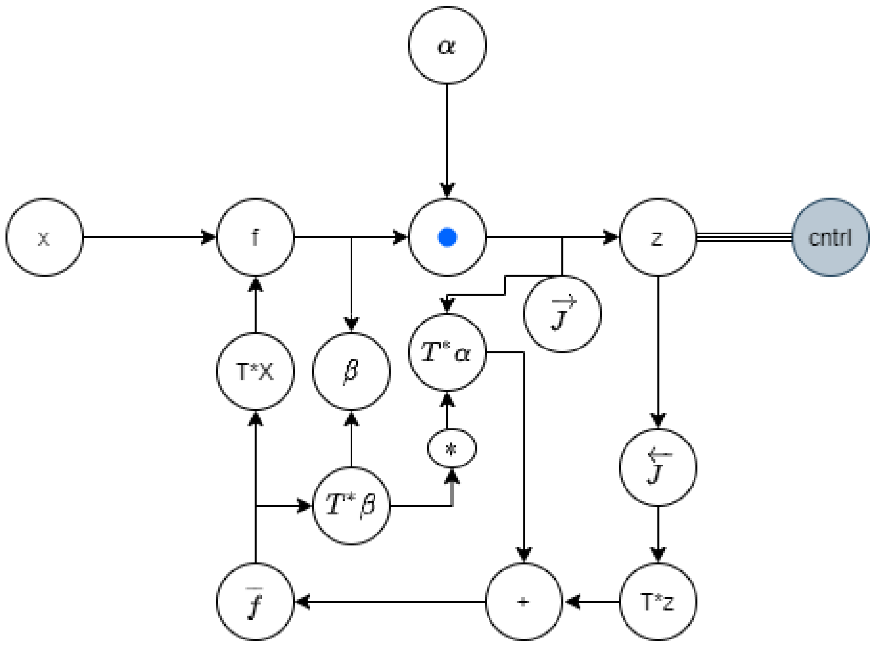


Figure 8.4: Loop for Fixed-Point iteration of the model training method

After training each layer, the Fixed-Point-Iterator block selects the next layer to be trained with an optimized parameter. This layer can be a previous layer which is already trained or a new layer in the sequence of the model structure. The training is controlled by the local error that updates the loss function($LossFun$).

The proposed TWNN is initialized with the initial parameter $params$. Then, Fixed-Point-Iterator can be computed using Forward mode AD and backward mode AD as shown in Eq. (8.8).

$$\textit{fixed-point-iterative} = (\overrightarrow{\mathcal{J}}\{f\}, \overleftarrow{\mathcal{J}}\{f\}) \tag{8.8}$$

Algorithm 14 describes the forward pass and backward pass for a single block of TWNN. Figure 8.5 explains the forward and backward computation of the fixed-point-iterative operator of proposed TWNN.Forward computation takes the input and parameter to generate the optimized parameter β and output z. At the same time, the gradients T^z and T^α are also generated. Similarly, for the backward pass, the proposed model

Algorithm 13 Fixed point temporal wormhole based Neural Network

```
    Input: Inputs are divided among batches, for each loop, a batch of inputs $x_b^0$, corresponding target $y_b$, learning rate $l_r$, tolerance error ($\varepsilon$), maximum iteration $max_i tr$;
    Output: predicted output z
 1: procedure $trainTWNNModel(x\_init)$
 2:     $model, \alpha \leftarrow initializeNNBlock()$
 3:     for each $b$ in range $(total\_batch)$ do
 4:         $model\_block \leftarrow next(model\_layers)$
 5:         $loss = 1.0$
 6:         while $model_b lock! = null$ do
 7:             $z_b \leftarrow Fixed - Point - Iterator(model_b lock, \alpha, x_b)$
 8:             $loss = LossFun(z, y_b)$
 9:             $model_b lock = next(model\_layers)||identify\_prev\_layer(model)$
        return $trained_m odel$
10:
```

takes output (z) and the optimized parameter β to generate the gradients T^x and T^β. Finally, these gradients optimized the trained parameters of the model.

Algorithm 14 Forward and Backward Pass for fix function for a single block of TWNN

```
    Input: The value of x at time t $x_t$, parameters $\alpha$;
    Output: Gradient of x
1: procedure $fix(f, \alpha, x_t)$
2:     Forward Pass:
3:     $z, \beta \leftarrow f(x_t, \alpha)$
4:     $T^* z, T^* \alpha = grad(f, z, \beta)$
5:     Backward Pass:
6:     $T^* x, T^* \beta \leftarrow \overline{f}(T^* z + T^* \alpha)$
7: return $T^* x$
```

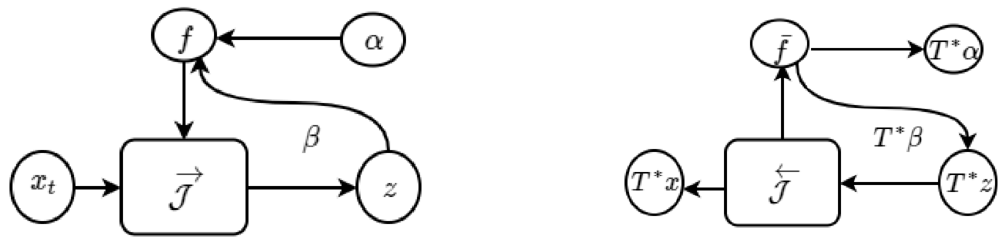
8.4 Experimental Result

The performance of proposed model is evaluated against two common tasks:

1. MNIST [11] classification and

2. Sine wave generation.

The training parameters for the two tasks are described in Table 8.2.



(a) Forward Mode for fixedPoint (b) backward Mode for fixedPoint

Figure 8.5: Forward and backward computation of the controller of TWNN

Table 8.2: Parameters for proposed model evaluation for MNIST classification

Learning Rate (l_r)	0.001
Tolerance Error (ε)	0.001
Batch Length Size	64
Maximum Number of Iteration per batch (max_iter)	300
ε	0.01

8.4.1 Task-A : CIFR classification

For the training for this task, we have chosen a TWNN model described in Fig. 8.6. After the second block and fourth block, there is Fixed-Point-iterator Block. After finishing 2nd block of the ResNet model, Fixed-Point-iterator Block identify the next suitable block as B1 and the model continue training B1 and B2 blocks until the loss $> \varepsilon$ or iteration $<$ max-iter. After the condition is met, Fixed-Point-iterator forward the training to the B3 layer. Then, Fixed-Point-iterator Block after B3 identify the next layer as the final layer of the network and forward the training to layer B. Instead of training a deep ResNet-50 layer for this training, we trained a less deep RestNet-36 model. However, we trained specific layers of the model for a longer time. Therefore, the memory consumption is significantly less than in DNN models. In addition, this model skipped block B4 during training. Therefore, the training time is not higher than ResNet.

Moreover, they show that the TWNN offers better results than the different DNN

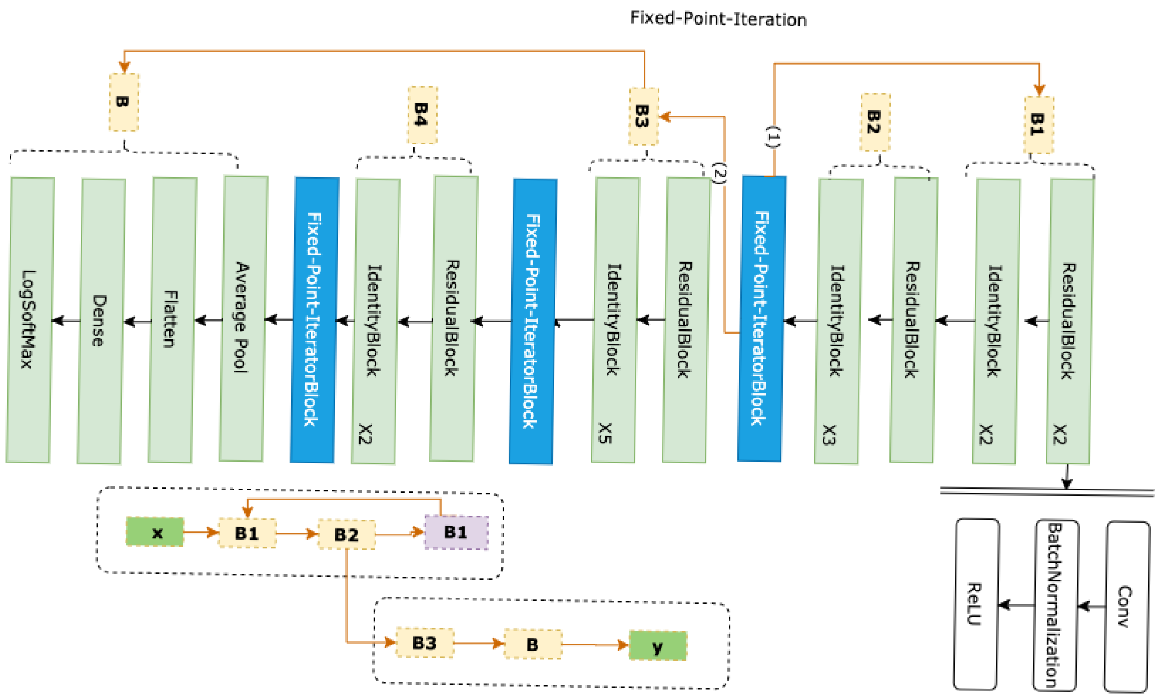


Figure 8.6: TWNN model for CIFR training

configurations. For example, table 8.3 shows the difference between the test accuracy of different similar configurations of ResNet and TWNN for CIFR.

Table 8.3: Comparative accuracy for proposed model against ResNet

Neural Network	Accuracy
RestNet- 36	88.25%
RestNet- 50	90.63%
TWRNN	96.54%

8.4.2 Task-B: Sine Wave Generation

In this task, a corresponding sine wave is generated using the proposed TWNN model for a time series T. Table 8.4 shows the parameters used in this task.

Table 8.4: Parameters for proposed model evaluation for CIFR classification

Learning Rate (l_r)	0.001
Tolerance Error (ε)	0.001
Sequence Length Size	10000
Maximum Number of Iteration per batch (max_iter)	300

For this task, we take a simple Neural Network as the objective function shown in 15. *loss* function as shown in 16 used this *net* procedure to compute gradient for the proposed TWNN.

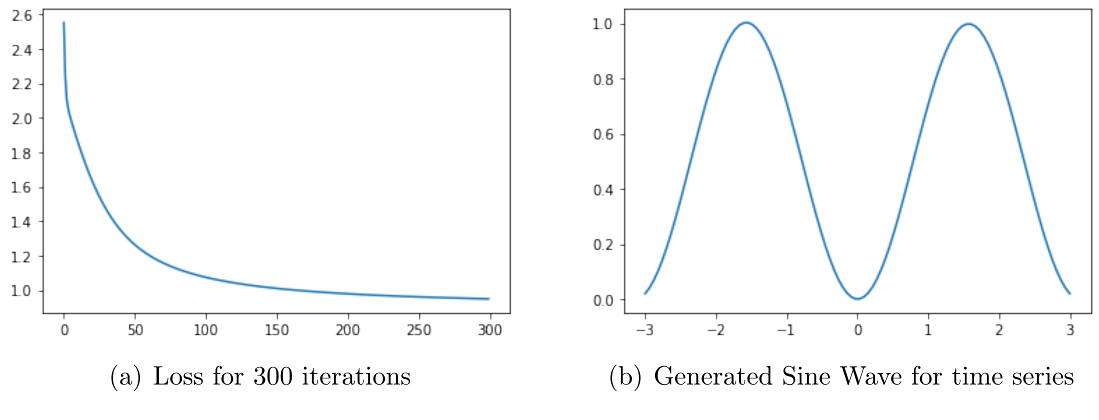
Algorithm 15 Neural Network as the objective function for the proposed model

```
    Input: The value of x at time t $x_t$, parameters $\alpha$;
    Output: the predicted value at time t $x_t$
1: procedure $net(f, \alpha, x_t)$
2:     $w1, b1, w2, b2 \leftarrow \alpha$
3:     $hidden \leftarrow tanh(dot(w1, x_t) + b1)$
4:     $y_t \leftarrow sigmoid(dot(w2, hidden) + b2)$
5: return $y_t$
```

Algorithm 16 Loss function for the proposed model

```
    Input: The value of x at time t $x_t$, parameters $\alpha$, target ;
    Output: the predicted value at time t $x_t$
1: procedure $loss(net, \alpha, inputs, target)$
2:     $solver \leftarrow fix(net, \alpha, inputs)$
3:     $predictions \leftarrow solver(inputs, params)$
4: return $mean((targets - predictions) * *2)$
```

Fig. 8.7(a) shows the gradients are optimized within 300th iteration and loss is stable. Also Fig. 8.7(b) shows the generated sine wave by training TWNN model for 300 iterations.



(a) Loss for 300 iterations (b) Generated Sine Wave for time series

Figure 8.7: Result of continuous Sine wave generation by TWNN

8.4.3 Discussion

The proposed TWNN model can be a suitable alternative against deep ResNet models with larger batch sizes; the TWNN model can achieve better accuracy than a shallow ResNet model. In the TWNN model, the same parameters are trained for a more extended period. Therefore, the corresponding memory requirement is lower than traditional DNN models. The most appropriate boundary for the threshold value for tolerance error is $0.015 < \varepsilon < 0.15$. If the value for tolerance error is too small, it significantly increases the training time for each layer. To overcome this issue, we have used another control, the maximum number of iterations. The following function will select the next layer once the loop runs for the defined maximum number of iterations.

Observation 1 — Only the best output are chosen for training

Lets say the data-set can have m number of batches. For a ResNet-50, the number of layers is $l = 50$. The maximum number of iteration is m. For a generic Res-Net, the output of the K-th residual block can be explained as the summation of the output of lower layers as shown in Eq. (8.9).

$$O_{K+1}(x) = \sum_{l=1}^{T} f(O_l(x))) \tag{8.9}$$

On the other hand, the output of the K-th residual block is the best output of lower layers, where some of the layers were not processed, and some other layers were processed multiple times, as shown in Eq. (8.10).

$$O_{K+1}(x) = \gamma(F(O_l(x))), \varepsilon, d) \tag{8.10}$$

Here F is the fixed-point iterator block for each layer l and γ determines the influential layer based on the batch dataset d and local error ε.

Observation 2 — The computation time and memory are low

For a traditional ResNet, the computation time and memory are linear in the neural network's depth. For example, for a TWNN with depth h, the computation time is $O(m \cdot h)$, where m is the average number of iterations of the Fixed-Iterator Block of the TWNN model.

A TWNN is computationally more efficient than the end-to-end Backpropagation of a deep ResNet. There is local Backpropagation for individual layers, which is data-driven only if the corresponding batch data is influential in identifying the target pattern. The backward function ($\bar{f}$) in Eq. (8.2) conducts a backward computation in order to optimize the training parameters.

Observation 3 — The training error decreases over layers exponentially

For example, if the local error is ε, the training error for the first layer (err_1) will already be close to ε, and for the next layer, it will be exponentially closer to ε as only the influential layer will be selected based on the local error and controlled by the Fixed-Iterator Block. The training error for any layer can be changed if that layer is selected for training again. For example, the error for layer 1 is err_1. After training layer 1, Fixed-Iterator Block selects layer two as the next layer to be trained. However, the Fixed-Iterator Block in the second layer selects layer one as the next layer for training. At this time, the training error for layer 1 will be reduced, $\overline{err}_1 \ll err_1$. Therefore, the training error decreases exponentially as only the influential layers are selected.

Observation 4 - Training time is data-driven depending on the batch technique

For Task B, the training for the proposed TWNN shown in Fig 8.6 progresses between layers: $B_1 \rightarrow B_2 \rightarrow B_1 \rightarrow B_2 \rightarrow B_3 \rightarrow B$. The local error for B_1 does not reach below the threshold (ε) in the first round. Once the number of iterations of a Fixed-Iterator Block reaches the maximum number of iterations, the forward pass selects B_2 as the next layer. At the same time the next function also check if the local error (err_2) for B_2 is more than the local error for B_1; $err_2 > err_1$, the, the next function of Fixed-Iterator Block selects B_1 as the next layer and runs the training until the local error reaches an acceptable value for err_1. The first layer is training twice, and then the model progresses toward the third layer B_3. As a result, the higher accuracy is not received right away, taking time. However, the proposed TWNN model chooses the influential layer to be trained. Therefore, instead of training each layer for the same number of iterations and the same time duration, it picks the more influential batches to run for a longer time, and that helps the proposed model reach a higher accuracy quicker than traditional ResNet with higher depth. For Task B, the same TWNN trains as $B_1 \to B$. On the other hand, for Task B, the data used in the minibatch was distributed. Therefore, the model reaches an accuracy of 98% just after running the first layer until the local error $err_1 < \varepsilon$. Therefore, training the model for each layer is not required, and quickly achieving higher accuracy.

8.5 Conclusion

TWNN neural network leverages fixed-point iteration with automatic differentiation. In addition, it is capable of explicitly providing more efficient performance by computing gradients for input, hidden states and parameters. Although only the influential layers are trained for a longer time, this new model can optimize the hidden dynamics learning. The proposed model reaches accuracy faster than the related Neural Network as each batch is trained under fixed point constraint through a fixed point iterated block. In the TWNN model, the accuracy is achieved faster than in other batches for batches. Therefore, the optimization of the TWNN model is more straightforward compared to other Deep Neural Networks.

8.6 References

[1] Adam Santoro, Sergey Bartunov, Matthew Botvinick, Daan Wierstra, and Timothy Lillicrap. One-shot learning with memory-augmented neural networks. arXiv preprint arXiv:1605.06065, 2016.

[2] Caglar Gulcehre, Sarath Chandar, and Yoshua Bengio. Memory augmented neural networks with wormhole connections. arXiv preprint arXiv:1701.08718, 2017.

[3] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, et al. Hybrid computing using a neural network with dynamic external memory. Nature, 538(7626):471–476, 2016.

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In European conference on computer vision, pages 630–645. Springer, 2016.

[5] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In International Conference on Medical image computing and computer-assisted intervention, pages 234–241. Springer, 2015.

[6] Zhengping Che, Sanjay Purushotham, Kyunghyun Cho, David Sontag, and Yan Liu. Recurrent neural networks for multivariate time series with missing values. Scientific reports, 8(1):1–12, 2018.

[7] Tian Qi Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. In Advances in Neural Information Processing Systems, pages 6572–6583, 2018.

[8] Barak A Pearlmutter. Algorithmic differentiation, functional programming, and iterate-to-fixedpoint.

[9] Rahul Dey and Fathi M Salemt. Gate-variants of gated recurrent unit (gru) neural networks. In 2017 IEEE 60th international midwest symposium on circuits and systems (MWSCAS), pages 1597–1600. IEEE, 2017.

[10] James D Keeler, Eric J Hartman, Kadir Liano, and Ralph B Ferguson. Residual activation neural network, October 4 1994. US Patent 5,353,207.

[11] Yann LeCun. The MNIST database of handwritten digits, 1998. http://yann.lecun.com/exdb/mnist/.

CHAPTER 9

CONCLUSION

This thesis mainly focused on the applications involving continuous time series. We primarily identified different challenges and limitations of existing research in the case of modelling continuous time series. In addition, we find out that traditionally continuous time series are converted into discrete ones to match the input of the existing neural network models. However, these types of methods can not provide real-time modelling as in order to mirror the continuous-time series, the sampling rate needs to be very small, which causes higher frequency and computation to become too chaotic to handle. In this thesis, we presented models to overcome those challenges such as irregular sampling rate, higher frequency, gradient vanishing problem, informative messiness, the large amount of computation power, memory and time requirement for modelling and the size of parameters for the neural network. Furthermore, we explored different technology to leverage Ordinary differential equations to model continuous time series.

9.1 Neural ODE based Recurrent Neural Network

The main achievement of this model is that it provides stability to the internal structure of Neural ODE. In addition, this model leverage the strength of RNN to provide better performance compared to Neural ODE.

9.2 Neural ODE and Generative Adversarial Neural Network

Time series modelling suffers from plenty of challenges. Missing informativeness is one of the core problems of using model-driven learning for continuous time series. This proposed model in section 4.1 provides a data-driven deep learning model in order to provide the solution for missing informativeness and vanishing gradient problems.

9.3 Neural ODE based GAN models for ECG synthesis

GAN model provides a state-of-art solution for ECG synthesis. However, the GAN model suffers from different challenges. Three different models leverage the dynamic characteristics of Neural ODE to design generator and Discriminator models for a GAN model. Neural ODE strengthens generator and Discriminator characteristics by learning data as continuous-time data. This new technique enables the GAN model to learn the hidden dynamics from the derivative of incoming data, which is an ODE w.r.t time.

9.4 Heun method based Neural Network

This model contributes significantly to recent work where ODE can be used as a neural network. Finally, this section introduces a data-driven deep, deep learning model for continuous-time series modelling. HeunNet adopts the architecture of the Heun method for Ordinary Differential equation solvers. This new model help to model continuous time series as a function of time instead of the sequence of discrete-time steps.

9.5 Continuous Convolutional Neural Network

Continuous Convolutional Neural networks leverage the strength of the Partial Differential Equation solver and the simplicity of ordinary Differential Equations. This model can train the PDE system continuously. This novel technique enables Convolutional Neural networks to be continuous. We have used the architecture of a 1-dimensional Convolutional Neural Network, a.k.a Time Delay Deep Neural Network

(TDNN) and ODE solver and PDE solver, to construct a continuous 1-dimensional Convolutional Neural Network.

9.6 Fixed point iteration based Neural Network

Deeper neural networks vs more comprehensive neural networks is an ongoing discussion in the field. It is now a concern: “How deep the model can be?” Or, “How comprehensive can it be?” Just developing a deeper neural network is not always a the best decision.

Finally, the main goal of my thesis was to develop different neural network architectures which can be used for continuous-time series modelling. We have presented five different models in my thesis. Instead of following the traditional discrete fixed time step based model, these models consider the time series as a continuous function of time. We have introduced twelve different machine learning algorithms to lean data as continuous time series instead of discrete sequences.

This thesis’s methodologies and models were motivated by practical applications where data is generated in real-time with a dynamic sampling rate. For example, include sensor data from IoT devices, Electronic health records containing patients’ health history, weather data, etc. Therefore, the machine learning model needs to be data-driven instead of model-driven. In retrospect, the primary motivation for this thesis is to develop machine learning algorithms that provide a better match to the properties of certain actual datasets than previous architectures.

Appendices

.1 Pytorch implementation of ODE based GRU model

```
from .ODEGRU import ODEGRU
import torch.nn as nn


class ODEGRUFunc(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(ODEGRUFunc, self).__init__()
        self.net = \ODEGRU(input_size, hidden_size, output_size)
        self.hidden = self.net.initial_hidden()


    def forward(self, t, y):
        h = self.hidden
        res,h = self.net(y, h)
        self.hidden = h
        return res

import torch
import torch.nn as nn
from torch.autograd import Variable
import autograd.numpy as np
class ODEGRU(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(ODEGRU, self).__init__()
        self.input_units = input_size
        self.output_units = output_size
        self.hidden_units = hidden_size
        # input embedding
        #self.encoder = nn.Embedding(input_size, embed_size)
        # lstm weights
        self.weight_wz = nn.Linear(input_size, hidden_size)
        self.weight_uz = nn.Linear(hidden_size, hidden_size)

        self.weight_wr = nn.Linear(input_size, hidden_size)
        self.weight_ur = nn.Linear(hidden_size, hidden_size)

        self.weight_wy = nn.Linear(hidden_size, output_size)

        self.weight_wh = nn.Linear(input_size, hidden_size)
        self.weight_uh = nn.Linear(hidden_size, hidden_size)
        self.decoder = nn.Linear(hidden_size, output_size)
        self.hidden = self.initial_hidden()

    def initial_hidden(self):
        # h_0 = Variable(torch.zeros(1, self.hidden_units))
        h_0 = Variable(torch.zeros(1, self.hidden_units))
        return h_0
    def forward(self, inp, h=None):
        if h is None:
            h= self.hidden
        # update gate
        z= self.weight_wz(inp) + self.weight_uz(h)
        z_g = torch.sigmoid(z)
        dz= self.sigmoidPrime(z)
        # reset gate
        r= self.weight_wr(inp) + self.weight_ur(h)
        r_g = torch.sigmoid(r)
        dr = self.sigmoidPrime(r)
        # hidden gate
        h1= (self.weight_wh(inp) + self.weight_uh(dr * h))
        dh = torch.tanh(h1)
        # hidden state
        hx = dh *(1-dz)#+ z_g#+ (dh * self.sigmoidPrime(z_g))
        o = self.decoder(hx)
        out =torch.sigmoid(o)
        self.hidden = hx
        return out, hx
```

.2 Pytorch implementation of ODE based LSTM model

```
from .mLSTM import mLSTM
import torch.nn as nn

class ODELSTMFunc(nn.Module):

    def __init__(self, input_size, hidden_size, embed_size,
                output_size):
        super(ODELSTMFunc, self).__init__()
        self.net = mLSTM(input_size, hidden_size, embed_size,
                        output_size)
    def forward(self, t, y):
        hidden, cell = self.net.current_state()
        res, h, cell = self.net(y, hidden, cell)
        return res

    def compute(self, y, h, cell):
        res, h, cell = self.net(y, h, cell)
        return res
import torch
import torch.nn as nn
from torch.autograd import Variable
import torch.nn.functional as F
class mLSTM(nn.Module):
    def __init__(self, input_size, hidden_size, embed_size,
                output_size):
        super(mLSTM, self).__init__()

        self.hidden_size = hidden_size
        # input embedding
        self.encoder = nn.Linear(input_size, embed_size)
        # lstm weights
        self.weight_fm = nn.Linear(hidden_size, hidden_size)
        self.weight_im = nn.Linear(hidden_size, hidden_size)
        self.weight_cm = nn.Linear(hidden_size, hidden_size)
        self.weight_om = nn.Linear(hidden_size, hidden_size)
        self.weight_fx = nn.Linear(embed_size, hidden_size)
        self.weight_ix = nn.Linear(embed_size, hidden_size)
        self.weight_cx = nn.Linear(embed_size, hidden_size)
        self.weight_ox = nn.Linear(embed_size, hidden_size)
        # multiplicative weights
        self.weight_mh = nn.Linear(hidden_size, hidden_size)
        self.weight_mx = nn.Linear(embed_size, hidden_size)
        # decoder
        self.decoder = nn.Linear(hidden_size, output_size)
        self.hidden, self.cell = self.init_hidden()

    def forward(self, inp, h_0, c_0):
        # encode the input characters
        inp = self.encoder(inp)
        # calculate the multiplicative matrix
        m_t = self.weight_mx(inp) * self.weight_mh(h_0)
        # forget gate
        f = self.weight_fx(inp) + self.weight_fm(m_t)
        f_g = torch.sigmoid(f)
        df = self.sigmoidPrime(f)
        # input gate
        i= self.weight_ix(inp) + self.weight_im(m_t)
        i_g = torch.sigmoid(i)
        di = self.sigmoidPrime(i)
        # output gate
        o= self.weight_ox(inp) + self.weight_om(m_t)
        o_g = torch.sigmoid(o)
        do =  self.sigmoidPrime(o)
        g_g = self.weight_cx(inp) + self.weight_cm(m_t)
        # intermediate cell state
        c_tilda = torch.tanh(g_g)
        dc_tilda = self.tanhp(g_g)
        # current cell state
        #cx = f_g * c_0 + i_g * c_tilda
        cx = df * c_0 + di * dc_tilda
        # hidden state
        hx = o_g * torch.tanh(cx)
        out = self.decoder(hx)
        out = torch.sigmoid(out)
        self.cell =cx
        self.hiden = hx
        return out, hx, cx

    def init_hidden(self):
        h_0 = Variable(torch.zeros(1, self.hidden_size))
        c_0 = Variable(torch.zeros(1, self.hidden_size))
        return h_0, c_0
    def current_state(self):
        return self.hidden, self.cell
```

.3 Pytorch Implementation of ODE-GRU-D-Extension

```
import torch
from torch.autograd import Variable
from torch.nn import Parameter, Linear
import numbers, warnings, math
from models.FilterLinear import FilterLinear
from torchdiffeq import odeint_adjoint as od
from models.dGru import dGru
```

```
class dGRUExt(torch.nn.Module):
    def __init__(self, input_size, hidden_size, output_size,
                 num_layers=1, x_mean=0,  bias=True,
                 batch_first=False, bidirectional=False,
                 dropout_type='mloss', dropout=0):
        super(dGRUExt, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.num_layers = num_layers
        self.zeros = torch.autograd.Variable(torch.zeros(input_size))
        self.x_mean = torch.autograd.Variable(torch.tensor(x_mean))
        self.bias = bias
        self.batch_first = batch_first
        self.dropout_type = dropout_type
        self.dropout = dropout
        self.bidirectional = bidirectional
        num_directions = 2 if bidirectional else 1
        self.identity = torch.eye(input_size)
        if not isinstance(dropout, numbers.Number)
                        or not 0 <= dropout <= 1
                        or isinstance(dropout, bool):
            raise ValueError("dropout should be a number in range [0, 1]"
        if dropout > 0 and num_layers == 1:
            warnings.warn(" non-zero dropout expects ")

        #####################################
```

```
 gate_size = 1  # not used
#####################################

 self._all_weights = []
# decay rates gamma
w_dg_x = Parameter(torch.Tensor(input_size))
w_dg_h = Parameter(torch.Tensor(hidden_size))
# y (output)
w_hy = Parameter(torch.Tensor(output_size, hidden_size))

# bias
b_dg_x = Parameter(torch.Tensor(hidden_size))
b_dg_h = Parameter(torch.Tensor(hidden_size))

b_y = Parameter(torch.Tensor(output_size))

layer_params = (w_dg_x, w_dg_h,w_hy, \
                b_dg_x, b_dg_h,  b_y)

param_names = ['weight_dg_x', 'weight_dg_h', \

               'weight_hy']
 if bias:
    param_names += ['bias_dg_x', 'bias_dg_h', \

                    'bias_y']
```

```
        for name, param in zip(param_names, layer_params):
            setattr(self, name, param)
        self._all_weights.append(param_names)
        self.gamma_x_l = FilterLinear(self.input_size,
                    self.input_size, self.identity)
        self.gamma_h_l = Linear(self.input_size,
            self.input_size)
        self.flatten_parameters()
        self.reset_parameters()

    def flatten_parameters(self):
        any_param = next(self.parameters()).data
        if not any_param.is_cuda or not torch.backends.cudnn.is_acceptab
            return
        all_weights = self._flat_weights
        unique_data_ptrs = set(p.data_ptr() for p in all_weights)
        if len(unique_data_ptrs) != len(all_weights):
            return

        with torch.cuda.device_of(any_param):
            import torch.backends.cudnn.rnn as rnn
            with torch.no_grad():
                torch._cudnn_rnn_flatten_weight(
                    all_weights, (4 if self.bias else 2),
                    self.input_size, rnn.get_cudnn_mode(self.mode), self
                    self.batch_first, bool(self.bidirectional))
```

```
def _apply(self, fn):
    ret = super(dGRUExt, self)._apply(fn)
    self.flatten_parameters()
    return ret

def reset_parameters(self):
    stdv = 1.0 / math.sqrt(self.hidden_size)
    for weight in self.parameters():
        torch.nn.init.uniform_(weight, -stdv, stdv)

def check_forward_args(self, input, hidden, batch_sizes):
    is_input_packed = batch_sizes is not None
    expected_input_dim = 2 if is_input_packed else 3
    if input.dim() != expected_input_dim:
        raise RuntimeError(
            'input must have {} dimensions, got {}'.format(
                expected_input_dim, input.dim()))
    if self.input_size != input.size(-1):
        raise RuntimeError(
            'input.size(-1) must be equal to input_size.
                Expected {}, got {}'.format( self.input_size,
                input.size(-1)))

    if is_input_packed:
        mini_batch = int(batch_sizes[0])
    else:
        mini_batch = input.size(0) if self.batch_first else input.si
    num_directions = 2 if self.bidirectional else 1
    expected_hidden_size = (self.num_layers * num_directions,
                            mini_batch, self.hidden_size)

    def check_hidden_size(hx, expected_hidden_size,
                          msg='Expected hidden size {}, got {}'):
        if tuple(hx.size()) != expected_hidden_size:
            raise RuntimeError(msg.format(expected_hidden_size,
                                          tuple(hx.size())))

    if self.mode == 'LSTM':
        check_hidden_size(hidden[0], expected_hidden_size,
                          'Expected hidden[0] size {}, got {}')
        check_hidden_size(hidden[1], expected_hidden_size,
                          'Expected hidden[1] size {}, got {}')
    else:
        check_hidden_size(hidden, expected_hidden_size)

def extra_repr(self):
    s = '{input_size}, {hidden_size}'
    if self.num_layers != 1:
        s += ', num_layers={num_layers}'
    if self.bias is not True:
        s += ', bias={bias}'
    if self.batch_first is not False:
        s += ', batch_first={batch_first}'
    if self.dropout != 0:
        s += ', dropout={dropout}'
    if self.bidirectional is not False:
        s += ', bidirectional={bidirectional}'
    return s.format(**self.__dict__)

def __setstate__(self, d):
    super(dGRUExt, self).__setstate__(d)
    if 'all_weights' in d:
        self._all_weights = d['all_weights']
    if isinstance(self._all_weights[0][0], str):
        return
    num_layers = self.num_layers
    num_directions = 2 if self.bidirectional else 1
    self._all_weights = []

    weights = ['weight_dg_x', 'weight_dg_h', \
               'weight_xz', 'weight_hz', 'weight_mz', \
               'weight_xr', 'weight_hr', 'weight_mr', \
               'weight_xh', 'weight_hh', 'weight_mh', \
               'weight_hy', \
               'bias_dg_x', 'bias_dg_h', \
               'bias_z', 'bias_r', 'bias_h', 'bias_y']

    if self.bias:
        self._all_weights += [weights]
    else:
```

```
        self._all_weights += [weights[:2]]

    @property
    def _flat_weights(self):
        return list(self._parameters.values())

    @property
    def all_weights(self):
        return [[getattr(self, weight) for weight in weights] for weight

    def forward(self, input):
        # input.size = (3, 33,49) : num_input or num_hidden, num_layer o
        X = torch.squeeze(input[0])  # .size = (33,49)
        Mask = torch.squeeze(input[1])  # .size = (33,49)
        Delta = torch.squeeze(input[2])  # .size = (33,49)
        Hidden_State = Variable(torch.zeros(self.input_size))
        step_size = X.size(1)  # 49
        output = None
        h = Hidden_State
        # decay rates gamma
        w_dg_x = getattr(self, 'weight_dg_x')
        w_dg_h = getattr(self, 'weight_dg_h')
        # bias
        b_dg_x = getattr(self, 'bias_dg_x')
        b_dg_h = getattr(self, 'bias_dg_h')

        func = dGru(1, self.hidden_size, 1)
```

```
t_l= torch.linspace(0., 1.,33)
for layer in range( self.num_layers):
    x = torch.squeeze(X[:, layer:layer + 1])
    m = torch.squeeze(Mask[:, layer:layer + 1])
    d = torch.squeeze(Delta[:, layer:layer + 1])
    # (4)
    gamma_h = torch.exp(-torch.max(self.zeros,
                        (w_dg_h * d + b_dg_h)))
    t_d = torch.linspace(0., 1., 5)
    delta_x = od(self.gamma_x_l, d, t_d)
    #delta_h =  od(self.gamma_h_l, d, t_d)
    gamma_x = torch.exp(-torch.max(self.zeros, delta_x))
    gamma_x =gamma_x[gamma_x.shape[0]-1]
    #gamma_h = torch.exp(-torch.max(self.zeros, delta_h))
    h= gamma_h
    x = x * (gamma_x * x + (1 - gamma_x) * self.x_mean)
    # (5)
    with torch.no_grad():
        func.init_hidden()
        i = torch.tensor([x[0]])
        j = torch.tensor([m[0]])
        g = od(func,tuple([i,h, m]), t_l)
    [x,h,  m] = g
   # (6)
w_hy = getattr(self, 'weight_hy')
b_y = getattr(self, 'bias_y')
```

```
        output = torch.matmul(w_hy, x) + b_y
        output = torch.sigmoid(output)

        return output
```

.4 Pytorch implementation of Continuous Colvolutional Neural Network

```
from neurodiffeq import diff
from neurodiffeq.solvers import Solver1D, Solver2D
from neurodiffeq.conditions import IVP, DirichletBVP2D
from neurodiffeq.networks import FCNN,  SinActv
from neurodiffeq.monitors import Monitor1D
import numpy as np



conditions = [
    DirichletBVP2D(
        x_min=0, x_min_val=lambda y: torch.sin(np.pi*y),
        x_max=1, x_max_val=lambda y: 0,
        y_min=0, y_min_val=lambda x: 0,
        y_max=1, y_max_val=lambda x: 0,
    )
]
nets = [TDNN(n_input_units=1, n_output_units=100, hidden_units=(512,))]

pde_solver = Solver2D(pde_system, conditions, xy_min=(0, 0), xy_max=(10,
pde_solver.fit(max_epochs=500)
```

```
class PDE_solver_Module(nn.Module):
  def __init__(self, pde_solver):
    super(PDE_solver_Module, self).__init__()
    self.pde_solver = pde_solver

  def forward(self,  y,t,parameter):
      solution =self.pde_solver.get_solution()
      return solution(y,t,parameter)

class ODE_solver_Module(nn.Module):
  def __init__(self, ode_solver):
     super(ODE_solver_Module, self).__init__()
     self.ode_solver = ode_solver
  def forward(self,  y, t,parameter):
      solution =ode_solver.get_solution()
      return solution(y,t,parameter)

class CCNN(nn.Module):

   def __init__(self,pde_solver,ode_solver,input_dim,output_dim):
      super(CCNN, self).__init__()
      self.pde_solver = pde_solver
      self.ode_solver = ode_solver
```

```
        self.net = nn.Sequential(PDE_solver_Module(pde_solver),
                                 ODE_solver_Module(ode_solver))

        self.linear = nn.Linear(input_dim, output_dim)
        for m in self.net.modules():
            if isinstance(m, nn.Linear):
                nn.init.normal_(m.weight, mean=0, std=0.1)
                nn.init.constant_(m.bias, val=0)

    def forward(self, y, t, parameters):
      y =  self.net(y,t, parameters)
      y = self.linear(y)
      return y
```

.5 Pytorch implementation of HeunNet

```
import torch



class HuneLSTM(torch.nn.Module):



    def __init__( self, input_size, hidden_size, bidirectional=True):
        super().__init__()
        self.hidden_size = hidden_size
        self.lstm = torch.nn.LSTM(
            input_size=input_size,
            hidden_size=hidden_size,
            bidirectional=bidirectional,
            batch_first=True
        )
        self.bi = 2 if bidirectional else 1

    def forward(self, u_sequence):
        c0 = u_sequence.new_zeros((self.bi, u_sequence.size(0),
            self.hidden_size))
        h0 = u_sequence.new_zeros((self.bi, u_sequence.size(0),
            self.hidden_size))
        outputs = []
        for i in range(u_sequence.size(1)):
            u_t = u_sequence[:, i, :].unsqueeze(1)
            t_t = u_sequence[:, i, -1]
            lstm_out, (c_t, h_t) = self.lstm(u_t, (c0, h0))
            x_hat = lstm_out[:,:,-1].unsqueeze(1)
            x_hat_prime = u_t + x_hat
            lstm_next, (c_s, h_s) = self.lstm(x_hat_prime, (c_t, h_t))
            x_hat_next = lstm_next[:,:,-1].unsqueeze(1)
            out = u_t +(x_hat_prime +x_hat_next) *0.5
            c0, h0 = c_s, h_s
            outputs.append(out)
        outputs = torch.cat(outputs, dim=1)
        return outputs
```

.6 Pytorch implementation of Extended HeunNet

```
import torch
```

```
class ExHuneLSTM(torch.nn.Module):

    def __init__( self, input_size, hidden_size, alpha,
                  bidirectional=True):
        super().__init__()
        self.hidden_size = hidden_size
        self.alpha = alpha
        self.lstm = torch.nn.LSTM(
            input_size=input_size,
            hidden_size=hidden_size,
            bidirectional=bidirectional,
            batch_first=True
        )
        self.bi = 2 if bidirectional else 1

    def forward(self, u_sequence):
        c0 = u_sequence.new_zeros((self.bi, u_sequence.size(0),
            self.hidden_size))
        h0 = u_sequence.new_zeros((self.bi, u_sequence.size(0),
            self.hidden_size))

        outputs = []
        for i in range(u_sequence.size(1)):
            u_t = u_sequence[:, i, :].unsqueeze(1)
            t_t = u_sequence[:, i, -1]
        lstm_out, (c_t, h_t) = self.lstm(u_t, (c0, h0))
        x_hat = lstm_out[:,:,-1].unsqueeze(1)
        x_hat_prime = u_t + x_hat
        lstm_next, (c_s, h_s) = self.lstm(x_hat_prime, (c_t, h_t))
        x_hat_next = lstm_next[:,:,-1].unsqueeze(1)
        out = u_t +(1-self.alpha)*x_hat_prime
                + x_hat_next * self.alpha
        c0, h0 = c_s, h_s
        outputs.append(out)
    outputs = torch.cat(outputs, dim=1)
    return outputs
```