Faculty of Mathematics and Informatics

VILNIUS UNIVERSITY

FACULTY OF MATHEMATICS AND INFORMATICS

MODELLING AND DATA ANALYSIS

MASTER'S STUDY PROGRAMME

Ordinary Differential Equations Based Model for Robot Trajectories Learning on Irregular Time Series

Master's thesis

Author: Evelina Bartosevič

VU email address: evelina.bartosevic@mif.stud.vu.lt

Supervisor: dr. Virginijus Marcinkevičius

Vilnius

2022

Abstract

Irregularly-sampled time series occur in many real-world applications, such as robots movement, astronomy, medicine. In this master's thesis, we concentrate on learning robot movement trajectories with imitation learning type of algorithms. Recently, a huge interest has been in the methods that perform end-to-end trajectory learning directly using irregularly-sampled time series as input without any additional data preparation steps because data gaps are informative themself. Such methods are usually based on Recurrent Neural Networks (RNNs). In this thesis, we propose the new deep learning model called Ordinary Differential Equations based Gated Recurrent Unit with Trainable Decays (ODE-GRU-D), which is based on state-of-the-art RNN models ODE-RNN and Gated Recurrent Unit with trainable Decays (GRU-D). Additionally, three other commonly used algorithms are selected for the experiments: one standard RNN model - Gated Recurrent Unit (GRU), and two specific algorithms for irregularly-sampled time series - GRU-D and ODE-RNN. The models are applied and compared on the irregularly-sampled MuJoCo Hopper trajectories datasets. ODE-GRU-D showed several advantages: it is more stable and converges faster than other investigated RNNs. Both ODE-based models notably outperformed the other two. Also, the proposed algorithm achieved significantly better results than the previous ODE-based model ODE-RNN on sparse time series cases.

Keywords: recurrent neural networks, irregularly-sampled time series, ordinary differential equations, trajectory learning

Santrauka

Nereguliarios laiko eilutės dažnai pasitaiko realaus pasaulio taikymuose, tokiuose kaip robotų judėjimas, astronomija ar medicina. Šiame magistro baigiamajame darbe koncentruojamės į roboto judėjimo trajektorijų išmokimą naudojant imitacinio mokymosi tipo algoritmus. Pastaruoju metu yra didelis susidomėjimas metodais, kurie atlieka mokymąsi naudojant nereguliarias laiko eilutes kaip modelio įvedimo duomenis be jokių papildomų duomenų paruošimo veiksmų, kadangi laikoma, kad duomenų spragos yra savaime informatyvios. Tokie metodai paprastai yra pagrįsti rekurentiniais neuroniniais tinklais (RNN). Šiame darbe mes pristatome naują gilaus mokymosi modelį – paprastosiomis diferencialinėmis lygtimis pagrįstas sulaikomas rekurentinis vienetas su treniruojamais nykimais (ODE-GRU-D), kuris yra paremtas naujausiais (*angl. state-of-the-art*) RNN modeliais ODE-RNN ir sulaikomu rekurentiniu vienetu su treniruojamais nykimais (GRU-D). Be to, eksperimentams parenkami kiti trys dažniausiai naudojami algoritmai: vienas standartinis RNN modelis – GRU ir du specifiniai algoritmai nereguliarioms laiko eilutėms – GRU-D ir ODE-RNN. Modeliai yra taikomi ir lyginami ant nereguliariai atrinktų MuJoCo Hopper trajektorijų duomenų. ODE-GRU-D parodė keletą privalumų: yra stabilesnis ir greičiau konverguoja nei kiti nagrinėjami RNN. Abu paprastosiomis diferencialinėmis lygtimis paremti modeliai gerokai pranoko kitus du. Be to, pasiūlytas algoritmas parodė reikšmingai geresnius rezultatus už ankstesnę modelio versiją ODE-RNN retų (*angl. sparse*) laiko eilučių atvejais.

Raktiniai žodžiai: rekurentiniai neuroniniai tinklai, nereguliarios laiko eilutės, paprastosios diferencialinės lygtys, trajektorijų mokymasis

Contents

List of Figures

List of Tables

Introduction

In the last few years, robots have increasingly become a natural part of daily life. Nowadays, robots have a wide range of applications in almost all industries (from manufacturing to space exploration and health care) because of their precision and convenience. However, most of the robotics applications are rather simple, where the tasks are pre-programmed [3]. Human beings can explore the world in various environments, so scientists and companies aim to make modern robots as capable as people. Therefore, **artificial intelligence** (AI) has become an increasingly common presence in robotic solutions, offering learning capabilities and flexibility in previously impossible applications. Although robotics, thanks to **deep learning** (DL) and **reinforcement learning** (RL), has gained vast progress in orientation in an environment [45], it is still a poor proxy for human dexterity and reasoning. A reasonable solution to this problem could be **imitation learning** (IL). IL is the process of learning from demonstrations (data) and the study of such algorithms. While RL may produce a superhuman-level performance on competitive tasks like games, IL is better suited to achieve human-level performance in real-world tasks [23, 41, 50]. Although IL methods are applicable in many applications related to robotics, we will concentrate on the **Trajectory Learning** task in this thesis. Trajectory Learning is one of the most fundamental issues for robotic applications, and automation [15]. It is a major area in robotics because it paves the path for autonomous vehicles and is widely used in different kinds of robots, such as industrial, flying, or humanoid.

A **trajectory** is a path that an object follows through space as a function of time [42]. We interpret the robot movement trajectories as **time series** in our case [19]. A large part of sequential and time-series problems fall within the class of partially observable systems, for example, such problems occur in sensor networks, movements of planets, citizen science, multi-robot systems, and many others [44, 64]. While there are many applications for irregularly-sampled time series, this thesis is dedicated to robotics examples. Most existing models assume that the observations are regularly sampled, and the objects can be fully observed at each sampling time, which is impractical for many real-world applications. As an example, when a robot wants to push a set of blocks into a target configuration, only the blocks in the top layer are visible to the camera [24, 38]. What is even more challenging is that the visibility of some objects could change over time, which means that observations can occur at non-uniform intervals for an agent, i.e., irregularly-sampled data. Various reasons can cause such data. The most common causes are - failed data transmissions, broken sensors, and damaged storage [24, 61].

For a long time, machine learning algorithms only had worked for fully observable systems, requiring the observations of some object at each data sample timestamp. Recurrent neural networks (RNNs) are the most popular choice for multidimensional, regularly-sampled time series data, for example, speech and text related problems. Though, RNNs are quite an awkward match for irregular time series data. Imputation or aggregation of averages, smoothing, interpolation, and spline methods are widely used to deal with such problems, but these methods destroy information about the timing of measurements, which can be informative, do not capture variable correlations, and may not capture complex patterns. Therefore, it is crucial to design models that can handle irregular multivariate time series. One of the most used such models is **Gated Recurrent Unit with trainable decays (GRU-D)**, proposed in 2018 by Che et al. [5]. An even more promising approach to this field was proposed in the second half of 2018 by Chen et al. [8]. The authors of the paper introduced the new family of deep neural network models, which use black-box ODE solvers as a deep neural network component (**Neural Ordinary Differential Equations**). Rubanova et al. [54] extended this idea: introduced a family of time series models, **ODE-RNNs**, which hidden state dynamics are specified by Neural ODEs. These ODE-RNNs family models outperform earlier known RNN-based algorithms on irregularly-sampled time series datasets [54].

The main aim of the master's thesis is to propose changes and improve the Neural Ordinary Differential Equations based Recurrent Neural Network algorithm (ODE-RNN) and to apply the proposed model for robot movement trajectories learning when we have irregularly-sampled data. To achieve the goal, the following tasks were performed:

- Make an overview of the needed theory and achievements in the field;
- Analyze and compare the selected common used in the field methods (GRU [9], GRU-D [5], ODE-RNN [54]);
- Describe and implement ODE-based Gated Recurrent Unit with Trainable Decays (ODE-GRU-D) model;
- Perform experiments on the MuJoCo Hopper trajectories dataset with different percentages (90%, 70%, 50%, 30%) of observed time points. At first, with 300 epochs and secondly multiple times with the found optimal number of epochs;

- Compare the proposed algorithm with selected neural netwroks using MSE and RMSE performance measures;
- Apply appropriate t-tests to check if obtained test RMSEs for ODE-RNN and ODE-GRU-D differ significantly.

1 Methods for Robots Trajectory Learning

This chapter presents the general overview of the most relevant and often used methods for robots movement trajectories learning. The first part of the chapter briefly introduces Reinforcement Learning and Imitation Learning fields, and the second part is about classic Recurrent Neural Networks architectures.

1.1 Reinforcement Learning

Reinforcement learning (RL) is a unique area of machine learning (ML) because it does not learn from any data but the environment. RL is mainly based on rewarding an agent for desired behaviors and punishing otherwise. It is also commonly called learning through trial and error [31, 60]. Most probably, the main application of RL is gaming, as with this technique is possible to achieve a superhuman performance on various computer games. The most famous example is the AlphaGo game, where the algorithm beats the best human player [21]. Another widespread application of RL methods is in robotics [31]. For example, RL is often applied in trending problems nowadays as autonomous cars' and robots' trajectory planning, dynamic paths, trajectory optimizations, and similar [11, 33, 65].

RL methods aim to learn a **policy** π that maps the system's state to the control input so that the **expected return**, denoted as $J(\pi)$, is maximized. The quality of the given action, state, or trajectory at some time t is measured with the reward $r^{(t)}$ [50]. As an example, the reward $r^{(t)}$ would be large if a robot is close to the needed trajectory and small if it is far from the trajectory. The expected return is calculated by the following formula [50]:

$$J(\pi) = \mathbb{E}\left[\sum_{t=0}^{T} r^{(t)} \mid \pi\right], \tag{1}$$

here T denotes a finite horizon. If a horizon is infinite, the formula is in such form [50]:

$$J(\pi) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r^{(t)} \mid \pi\right]. \tag{2}$$

In formula (2), γ^t is a **discount-rate** which controls the trade-off between short-term and long-term rewards. The desired policy is found by the formula [50]:

$$\pi^* = \arg\max_{\pi} J(\pi) \tag{3}$$

The **value function** attempts to find such a policy that maximizes the return by maintaining a set of estimates of expected returns for the particular policy. The value function of a state x under policy π is computed this way [50, 63]:

$$V^{\pi}(x) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r^{(t)} \mid x_0 = x, \pi\right]. \tag{4}$$

1.2 Imitation Learning

Although RL methods have shown promising results in recent years, especially in games, these methods are usually difficult to apply to some real-world tasks. The first faced issue is that the game's environment is fully observable for an agent, but the state of the real world is only partially visible at any time. Secondly, there is often no direct reward (e.g., teaching an autonomous car). Therefore, a reward function should be designed manually in such tasks, which is usually very complicated. The solution to these problems can be Imitation Learning (IL) [41]. IL is defined as learning from expert demonstrations (instead from scratch as in RL) and the study of such algorithms. Expert is often a human, but not necessary, it also can be demonstrations in the form of simulated robotics examples. IL has two main forms: **Behavioral Cloning** - demonstrated actions/trajectories are imitated (cloned) directly, and **Inverse Reinforcement Learning** - recovering the reward function from the data. Expert demonstrations are usually given as a set of trajectories, so the dataset of such demonstrations is given this way: $\mathcal{D} = \{\tau_0, \tau_1, \ldots, \tau_m\}$ [41, 50].

Assume that the behavior of the expert is observed as a trajectory in the form $\tau = \left[\phi^{(0)}, \phi^{(1)}, \ldots, \phi^{(T)}\right]$ - sequence of features ϕ. The features ϕ are measurements chosen based on the given problem. Demonstrations are often recorded under some conditions. These conditions are referred to as **context** vector $\boldsymbol{s}$. This vector can contain any related information to the investigated problem (e.g., the initial state of the robotic system). The problem context is normally fixed during execution, and the only dynamic aspects of the task are the features. A collected dataset of demonstration in IL consists of trajectories, contexts, and optionally reward, which is sometimes available in specific problems: $\mathcal{D} = \{(\boldsymbol{\tau}_i, \boldsymbol{s}_i, r_i)\}_{i=1}^N$. The selected optimization-based strategy learns a policy π^* from the dataset as follows [50]:

$$\pi^* = \arg\min D(q(\phi), p(\phi)), \tag{5}$$

here $q(\phi)$ denotes the experts' induced distribution of the features, and $q(\phi)$ - the learner's induced distribution. $D(q(\phi), p(\phi))$ is a similarity measure between both distributions [50].

Supervised learning of neural networks is often used in IL, especially in Behavioral Cloning. **Recurrent Neural Networks (RNNs)** based models deserve special attention as they have made IL on complex, irregular time series data possible [50]. The main reason for such a contribution of RNNs to IL is the ability to consider previous actions during the learning process since many applications rely on performing trajectories of dependent motion primitives. So, RNN-based models are especially useful in IL for partially observable problems (irregularly-sampled time series case) and when observations are non-markovian [25].

1.3 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are the state-of-the-art algorithms for various tasks with sequential data (speech synthesis, translating natural language, weather forecasting, learning robot trajectories, etc.). The most common example of sequential data is time series data. **Time series** can be defined as a sequence of data points collected at successive time points over some time period. As in the case of many other machine learning algorithms, the first ideas about RNNs appeared in the 1980s [13], but only during recent years, these models show their real potential. The main reasons for it are the huge amount of available data and an increment of computational resources. Other types of NNs assume variables are independent. However, the independence assumption fails in the sequential data case. RNNs are famous because of their ability to memorize previous computations (sequential memory). This capacity allows them to remember meaningful information about the input and be very precise in forecasting using sequential data like time series [40].

RNNs are such types of Neural Networks that allow previous outputs to be used as inputs while having hidden states [1]. The hidden state records historical information of some sequence up to the current time step. Nodes with recurrent edges, at timestamp t, receive an input from the state $\mathbf{x}^{(t)}$ and saved values from the previous hidden node $\mathbf{h}^{(t-1)}$. This way, the output $\mathbf{y}^{(t)}$ is obtained based on the hidden layer $\mathbf{h}^{(t)}$. Therefore, an input $\mathbf{x}^{(t-1)}$ at time $t-1$ influences the later output $\mathbf{y}^{(t)}$ what is already at time t [40]. This process can be described via the following equations:

$$\mathbf{h}^{(t)} = \sigma_1(W_{hh}\mathbf{h}^{(t-1)} + W_{hx}\mathbf{x}^{(t)} + \mathbf{b}_h), \tag{6}$$

$$\mathbf{y}^{(t)} = \sigma_2(W_{yh}\mathbf{h}^{(t)} + \mathbf{b}_y). \tag{7}$$

Where $\mathbf{b}_h$, $\mathbf{b}_y$ are appropriate bias parameters, σ_1 and σ_2 are some nonlinear activation functions (Table 1.1), W_{hh} - this matrix consists of the weights between the hidden state and itself at adjoining time steps, W_{hx} is the matrix of weights between the input and the hidden node and similarly W_{yh} is the weights matrix between the hidden node and the output layer.

The described above architecture of such an RNN model is summarized in **Figure 1.1** and the more detailed view of the RNN node structure in **Figure 1.2**.

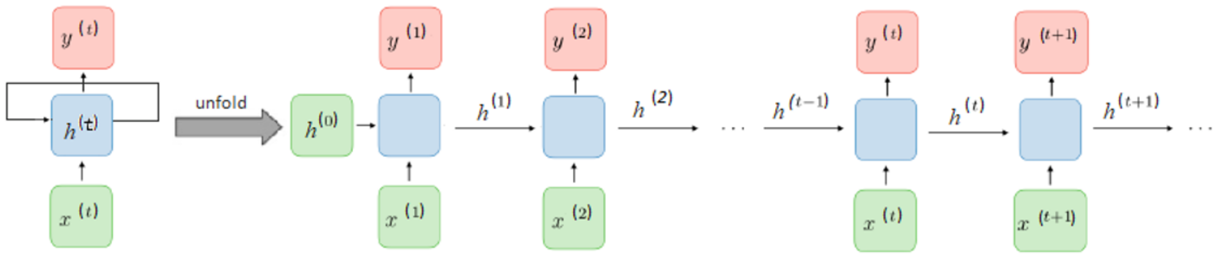


Figure 1.1: The simple RNN model (left side) and its' unfolded in time version (right side).

At each time step, the activation functions σ_1 and σ_2 are applied in every hidden state (**Figure 1.2**). The most widely used activation functions in RNNs are **sigmoid, tanh**, and **ReLU** [1, 40]. These functions are described in **Table 1.1**.

Loss function evaluates how close is the predicted value $\hat{y}^{(t)}$ to actual value $y^{(t)}$. In the case of an RNN model, the overall loss function $\mathcal{L}(\hat{y}, y)$ can be computed as the sum of the losses at every time step $(1, 2, ..., T)$ [1]:

$$\mathcal{L}(\hat{y}, y) = \sum_{t=1}^{T} \mathcal{L}(\hat{y}^{(t)}, y^{(t)}) = \sum_{t=1}^{T} \mathcal{L}^{(t)}. \tag{8}$$

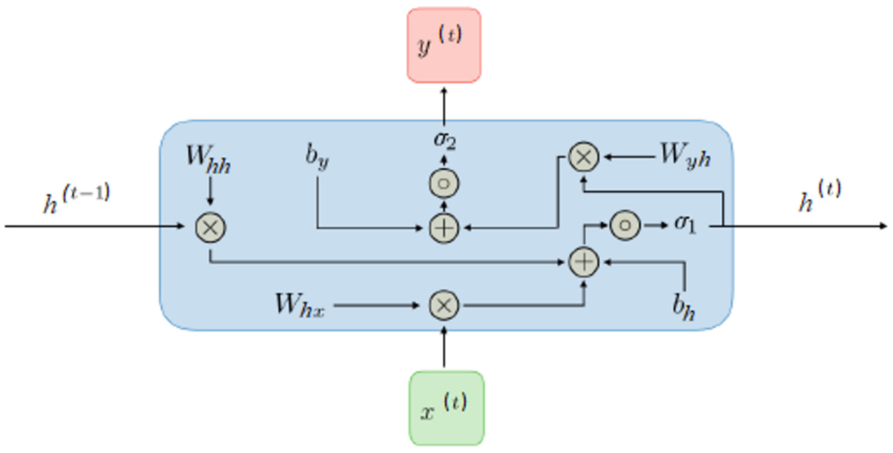


Figure 1.2: The structure of a Recurrent Neural Network Node.

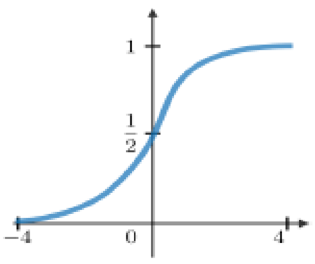
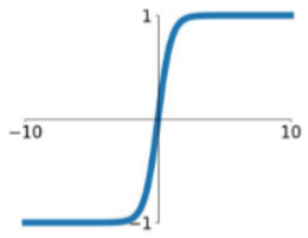
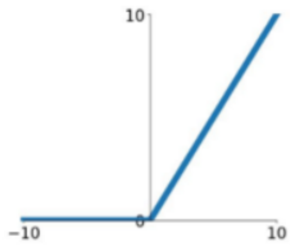
Sigmoid	**Hyperbolic tangent (tanh)**	**Rectified linear unit (ReLU)**
$\sigma(z) = \dfrac{1}{1+e^{-z}}$	$\sigma(z) = \dfrac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$	$\sigma(z) = \mathbf{max}(0, z)$
		

Table 1.1: Commonly used activation functions in RNNs.

From the *unfolded* part of the scheme in **Figure 1.1**, it is clear that such a network can be trained using the **backpropagation (BP)** technique [16]. However, in RNNs, the algorithm called **backpropagation through time (BPTT)** is used because these types of neural networks are trained across many time steps - a dependency between previous and current time steps is faced [6]. In the BPTT algorithm case, the **chain rule** is applied multiple times. At first, let's compute the gradient with respect to the weight W_{hh}. If only the output $y^{(t)}$ is considered, the gradient w.r.t. W_{hh} is following [6]:

$$\frac{\partial \mathcal{L}^{(t)}}{\partial W_{hh}} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}^{(t)}}{\partial y^{(t)}} \frac{\partial y^{(t)}}{\partial h^{(t)}} \frac{\partial h^{(t)}}{\partial W_{hh}}. \tag{9}$$

However, the hidden state $h^{(t)}$ partially dependents on the state $h^{(t-1)}$. Therefore, the equation (9) can be rewritten this way [6]:

$$\frac{\partial \mathcal{L}^{(t)}}{\partial W_{hh}} = \sum_{k=1}^{t} \frac{\partial \mathcal{L}^{(t)}}{\partial y^{(t)}} \frac{\partial y^{(t)}}{\partial h^{(t)}} \frac{\partial h^{(t)}}{\partial h^{(k)}} \frac{\partial h^{(k)}}{\partial W_{hh}}. \tag{10}$$

At this step, all gradients with respect to W_{hh} should be aggregated over the given time sequence. This way, the following gradient is obtained [6]:

$$\frac{\partial \mathcal{L}}{\partial W_{hh}} = \sum_{t=1}^{T} \sum_{k=1}^{t} \frac{\partial \mathcal{L}^{(t)}}{\partial y^{(t)}} \frac{\partial y^{(t)}}{\partial h^{(t)}} \frac{\partial h^{(t)}}{\partial h^{(k)}} \frac{\partial h^{(k)}}{\partial W_{hh}}. \tag{11}$$

Similarly, in the gradient w.r.t. W_{hx} case. The hidden node $h^{(t-1)}$ and the input $x^{(t)}$ both make contribution to state $h^{(t)}$. Hence, summing up all contributions via BP and taking derivative w.r.t. W_{hx} over the whole sequence such gradient is obtained [6]:

$$\frac{\partial \mathcal{L}}{\partial W_{hx}} = \sum_{t=1}^{T} \sum_{k=1}^{t} \frac{\partial \mathcal{L}^{(t)}}{\partial y^{(t)}} \frac{\partial y^{(t)}}{\partial h^{(t)}} \frac{\partial h^{(t)}}{\partial h^{(k)}} \frac{\partial h^{(k)}}{\partial W_{hx}}. \tag{12}$$

Despite all the advantages, classic RNN has several drawbacks. With the RNN model, it is challenging to capture long-term dependencies, resulting in vanishing (most of the time) or exploding (rarely) gradients problems. The gradient vanishing means that the value of gradient can shrink layer over layer and vanish after some time steps. It causes that the influence of previous events disappears, and the algorithm loses its memory. Another scenario is the gradient explosion, which happens because of large values in matrix multiplication. The gradient explosion can result in an unstable network that cannot learn over long input sequences [1, 6, 40].

Such Recurrent Neural Networks also are widely used for robots trajectory learning and paths planning because of their ability to consider previous actions during the learning process. Some examples can be found in the articles [2, 4, 47, 51].

1.3.1 Long Short-Term Memory

The **Long Short-Term Memory (LSTM)** algorithm first time was introduced in 1997 [20]. The main aim of this extended recurrent model is based on classic RNNs weaknesses - to overcome the vanishing gradients problem. The architecture of the LSTM is basically the same as RNN (**Figure 1.1**). The main differences are that LSTM introduces

memory cell and **gates** (forget, input, and output) concepts. This subsection is written based on [1,6,40] sources.

The memory cell $c^{(t)}$ is a composite unit because it is built from simpler nodes, and in this way, it can control information flow. As in the usual RNN case, it has $h^{(t-1)}$ and $x^{(t)}$ as inputs and outputs $h^{(t)}$ but also has gates. The gating mechanism controls the memoizing process in LSTMs. In the general case, the gate can be described as following:

$$\Gamma = \sigma(Wx^{(t)} + Uh^{(t-1)} + b), \tag{13}$$

where W, U are corresponding weights, b is an appropriate bias term, and σ means here sigmoid activation function.

The following equations describe the full LSTM algorithm [6,40]:

$$\mathbf{f}^{(t)} = \sigma(W_{hf}\mathbf{h}^{(t-1)} + W_{xf}\mathbf{x}^{(t)} + \mathbf{b}_f), \tag{14}$$

$$\mathbf{i}^{(t)} = \sigma(W_{hi}\mathbf{h}^{(t-1)} + W_{xi}\mathbf{x}^{(t)} + \mathbf{b}_i), \tag{15}$$

$$\mathbf{g}^{(t)} = \phi(W_{hc}\mathbf{h}^{(t-1)} + W_{xc}\mathbf{x}^{(t)} + \mathbf{b}_c), \tag{16}$$

$$\mathbf{c}^{(t)} = \mathbf{f}^{(t)} \odot \mathbf{c}^{(t-1)} + \mathbf{i}^{(t)} \odot g^{(t)}, \tag{17}$$

$$\mathbf{o}^{(t)} = \sigma(W_{ho}\mathbf{h}^{(t-1)} + W_{xo}\mathbf{x}^{(t)} + \mathbf{b}_o), \tag{18}$$

$$\mathbf{h}^{(t)} = \mathbf{o}^{(t)} \odot \phi(\mathbf{c}^{(t)}), \tag{19}$$

where $\mathbf{f}^{(t)}$ denotes a *forget gate*, $\mathbf{i}^{(t)}$ is an *input gate*, $\mathbf{o}^{(t)}$ - an *output gate*, in this work $\mathbf{g}^{(t)}$ denotes an *input node*, $\mathbf{c}^{(t)}$ means a *memory unit (cell)*, and $\mathbf{b}_f$, $\mathbf{b}_i$, $\mathbf{b}_c$, $\mathbf{b}_o$ are appropriate bias parameters. A symbol $\odot$ represents element-wise multiplication, ϕ denotes **tanh** activation function and σ - sigmoid function. W_{hf}, W_{hi} W_{xf}, W_{xi}, W_{hc}, W_{xc}, W_{ho}, W_{xo} are the weights matrices.

The purposes of the three LSTM gates are:

- *Forget gate* - responsible for what information to keep and what to forget from the previous cell;
- *Input gate* - controls what parts of the new cell content are written to the cell;
- *Output gate* - controls what part of the information is output to the hidden state.

The structure of the LSTM algorithm is presented in **Figure 1.3**.

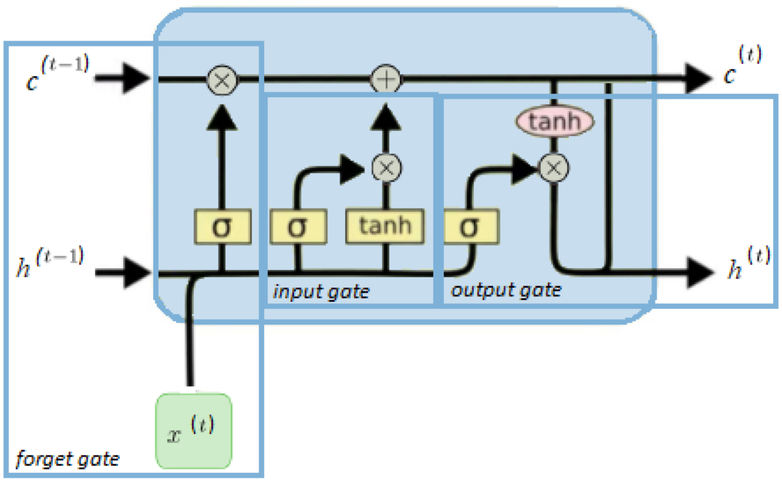


Figure 1.3: The structure of a Long Short-Term Memory cell.

LSTM outperforms classic RNN in modeling long-term sequential dependencies. This algorithm is also often used in robot trajectories-related tasks, especially when a longer time period is considered [26, 49, 51, 68]. However, LSTM has several disadvantages: it takes longer to train, requires much more memory resources, easy to overfit, and still can have exploding gradients.

1.3.2 Gated Recurrent Unit

Gated Recurrent Unit (GRU) model is a newer generation of Recurrent Neural Networks and was introduced by Cho et al. in 2014 [9]. GRU and earlier described LSTM algorithms are closely related variants. Similarly to LSTM, this algorithm also has a long-term memory and deals with vanishing gradients problem. However, Gated Recurrent Units have simpler architecture, which causes that they are much more time-efficient (usually about 30 % faster). The main differences in architecture compared to the LSTM model are that GRU has not a separate memory cell and two instead of three gates [9, 10, 27, 66].

In a similar way as with the above recurrent neural networks, the GRU algorithm can be described with the following equations [27]:

$$\mathbf{z}^{(t)} = \sigma(W_{hz}\mathbf{h}^{(t-1)} + W_{xz}\mathbf{x}^{(t)} + \mathbf{b}_z), \tag{20}$$

$$\mathbf{r}^{(t)} = \sigma(W_{hr}\mathbf{h}^{(t-1)} + W_{xr}\mathbf{x}^{(t)} + \mathbf{b}_r), \tag{21}$$

$$\tilde{\mathbf{h}}^{(t)} = \phi(W_{hh}(\mathbf{r}^{(t)} \odot \mathbf{h}^{(t-1)}) + W_{xh}\mathbf{x}^{(t)} + \mathbf{b}_h), \tag{22}$$

$$\mathbf{h}^{(t)} = (1 - \mathbf{z}^{(t)}) \odot h^{(t-1)} + \mathbf{z}^{(t)} \odot \tilde{\mathbf{h}}^{(t)}, \tag{23}$$

where $\mathbf{z}^{(t)}$ denotes an *update gate*, $\mathbf{r}^{(t)}$ - a *reset gate*, $\tilde{\mathbf{h}}^{(t)}$ is a current memory content, and $\mathbf{h}^{(t)}$ - a hidden state (as above). W_{hz}, W_{hr}, W_{hh}, W_{xz}, W_{xh} are weights matrices, $\mathbf{b}_z$, $\mathbf{b}_r$, $\mathbf{b}_h$ are bias parameters, σ denotes sigmoid activation function, and ϕ is tanh activation function [10, 27].

Short explanations of new notations:

- *Update gate* - controls what parts of the information from the previous time steps need to be passed to the future;
- *Reset gate* - controls what parts of the information from the previous hidden state to forget;
- *Current memory content* - uses the reset gate to select relevant parts from the previous hidden content.

The schematic representation of the GRU algorithm is visualized in **Figure 1.4**.

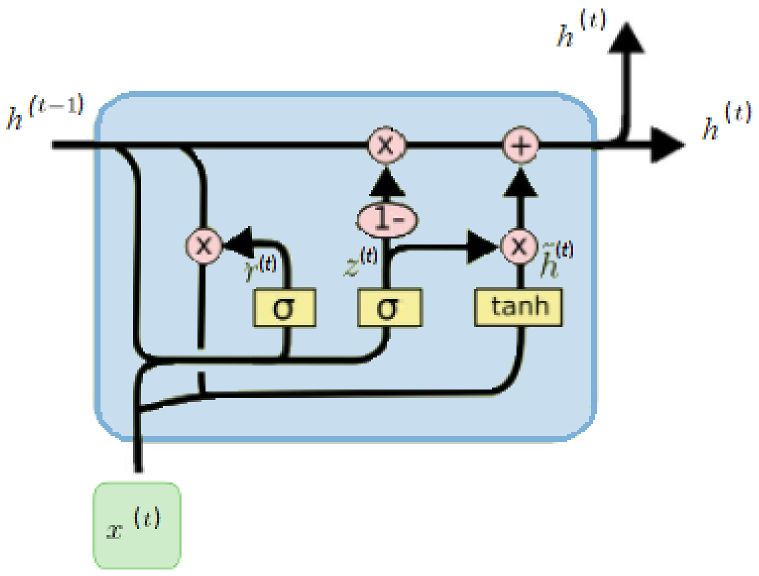


Figure 1.4: The structure of the Gated Recurrent Unit cell.

GRU-RNN based models are also often used for robots' movement trajectories learning tasks [28,32,51,67]. GRU and LSTM both have their pros and cons, but as GRU consumes less memory and is significantly faster than LSTM, that is why it is usually a better choice for robotics related tasks. The reasons are that robots have weaker hardware, electricity consumption, and in the robotics field are popular real-time tasks where the algorithm speed is essential.

2 Methods for Irregularly-Sampled Time Series

The second chapter mainly presents the overview of the various methods for end-to-end learning on irregularly-sampled time series and a short comparison of these methods. Also, the new proposed in the thesis model, which is based on current state-of-the-art methods, is described. Finally, performance measures and statistical tests used in the practical part are described at the end of this chapter.

2.1 Irregular Time Series

As described in the introduction, **irregularly-sampled time series** are common in a majority of fields where time-series data are met (astronomy, medicine, robotics, financial market, citizen science, climatology, ecology, geology) [44, 64]. In this master's thesis, we will concentrate on applications in robotics. Thus, robot movement trajectories will be treated as time series. In robotics, irregularly sampled data occur very often. The main reasons for it are broken sensors, damaged storage, failed data transmissions, etc. What is even trickier, for a robot, the visibility of some objects could change over the investigated time, which means that observations can occur at non-uniform time intervals [24, 61]. Smoothing, interpolation, and spline methods are widely used to deal with such problems, but these methods do not capture variable correlations and may not capture complex patterns. Unfortunately, there is limited work on exploiting the missing patterns for effective imputation and improving prediction performance in such cases. However, during the last years, there has been notable progress in developing specialized algorithms that can accommodate sparse and unevenly spaced time series or time series with missing values as an input [5, 24, 54, 57].

In short, irregularly-sampled time series can be described as a sequence with large and irregular time periods between observations. Irregularly spaced data are called sparse when the intervals between successive observations are often large. Also, a very close problem to this one is time series with missing values [57]. The observation times are strictly increasing ($t_1 < t_2 < ... < t_n$, $\forall n \geqslant 1$). We denote irregular multivariate time series of length T and with D variables as $X = (\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ..., \mathbf{x}^{(T)}) \in \mathbb{R}^{T \times D}$, $\mathbf{x}^{(t)}$ denotes the t-th sample and $x_d^{(t)}$ is the measurement of d-th variable of $\mathbf{x}^{(t)}$, where $\forall t \in \{1, 2, ..., T\}$ and $\forall d \in \{1, 2, ..., D\}$. Let's describe $s^{(t)} \in \mathbb{R}$ which denotes a **time-stamp** of the t-th observations (example in Figure 2.1). We set an assumption that the first observation obtained at time-stamp 0 ($s^{(1)} = 0$). [5, 12]

2.2 Gated Recurrent Unit with trainable Decays

The above described RNN-based models achieved state-of-the-art results in many applications with time series. Nevertheless, they are an inconvenient fit for irregular time series [54]. A usual procedure for applying standard RNN methods to irregularly-sampled time series data is to divide the timeline into equally-sized intervals and aggregate or impute observations using the calculated averages. However, it destroys important information about the timing of the measurements. In the last few years, a noticeable interest has been in the methods that perform end-to-end learning directly using multivariate irregularly sampled time series as input without the need for additional data preparation steps, which were mentioned in **Section 2.1**. One of the main examples of such models is **Gated Recurrent Unit with trainable Decays (GRU-D)**, proposed in 2018 by Che et al. [5]

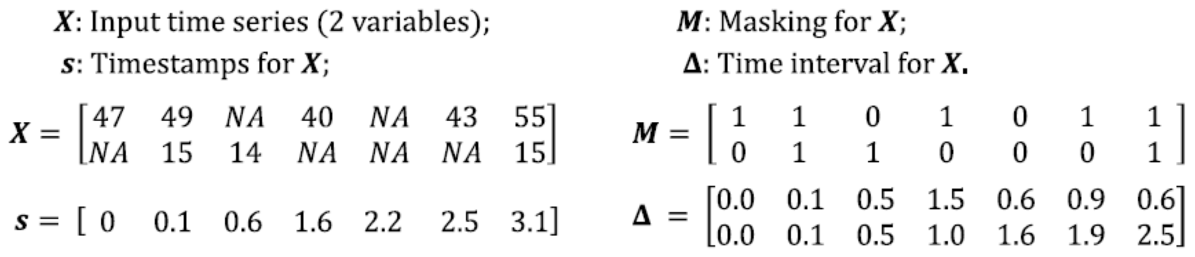
GRU-D model deals well with informative missingness patterns by applying **masking** and **time interval**. Masking aims to spot which inputs are missing and time interval covers the original input data patterns. This algorithm has a **decay mechanism** which is designed for the input time series data $\mathbf{x}^{(t)}$ and hidden states $\mathbf{h}^{(t)}$ to capture the essential properties. First of all, let's introduce **masking vector** and **time interval** concepts [5, 37]. A masking vector $\mathbf{m}^{(t)} \in \{0, 1\}^D$ shows which variables are missing at time step t, more specifically it can be described this way [5]:

$$\mathbf{m}_d^{(t)} = \begin{cases} 1, & \text{if } x_d^{(t)} \text{ is observed,} \\ 0, & \text{otherwise.} \end{cases} \tag{24}$$

The time interval $\delta_d^{(t)} \in \mathbb{R}$ for each variable d since its last observation can be described as following:

$$\delta_d^{(t)} = \begin{cases} s^{(t)} - s^{(t-1)} + \delta_d^{(t-1)}, & t > 1, m_d^{(t-1)} = 0, \\ s^{(t)} - s^{(t-1)}, & t > 1, m_d^{(t-1)} = 1, \\ 0, & t = 1. \end{cases} \tag{25}$$

A simple illustrative example of such data is given in **Figure 2.1**.



X: Input time series (2 variables);
s: Timestamps for ***X***;

$$\boldsymbol{X} = \begin{bmatrix} 47 & 49 & NA & 40 & NA & 43 & 55 \\ NA & 15 & 14 & NA & NA & NA & 15 \end{bmatrix}$$

$$\boldsymbol{s} = [\,0 \quad 0.1 \quad 0.6 \quad 1.6 \quad 2.2 \quad 2.5 \quad 3.1\,]$$

M: Masking for ***X***;
Δ: Time interval for ***X*.**

$$\boldsymbol{M} = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\boldsymbol{\Delta} = \begin{bmatrix} 0.0 & 0.1 & 0.5 & 1.5 & 0.6 & 0.9 & 0.6 \\ 0.0 & 0.1 & 0.5 & 1.0 & 1.6 & 1.9 & 2.5 \end{bmatrix}$$

Figure 2.1: Example of the notations $\mathbf{x}^{(t)}$, $s^{(t)}$, $\mathbf{m}^{(t)}$, $\delta_d^{(t)}$ [5].

What is more, **decay rates** were introduced to control mentioned decay mechanism based on two factors. First, the decay rates should differ from variable to variable. Secondly, decay rates are learned from the training data and are not prefixed constants. This way, useful information from the missing patterns is learned. In a general case, a vector of decay rates γ can be expressed by the equation [5, 37]:

$$\boldsymbol{\gamma}^{(t)} = \exp\left\{-\max\left(0, W_\gamma \delta^{(t)} + \mathbf{b}_\gamma\right)\right\}, \tag{26}$$

where parameters $W_{h\gamma}$, $W_{x\gamma}$ and $\mathbf{b}_\gamma$ are trained together with other GRU model parameters. The decay rate is always between 0 and 1. GRU-D model has two different trainable decay mechanisms. The first one is called **input decay** γ_x, which deals with missing values – decay an input over time towards the empirical mean. This trainable decay is applied to the measurement vectors in such a way [5]:

$$\hat{x}_d^{(t)} = m_d^{(t)} x_d^{(t)} + \left(1 - m_d^{(t)}\right)\left(\gamma_{dx} x_d^{(t')} + (1 - \gamma_{dx})\,\tilde{x}_d\right), \tag{27}$$

where $x_d^{(t')}$ denotes the last observation of variable d and $t' < t$. The empirical mean of the d-th variable is denoted as $\tilde{x}_d$, the weights matrix $W_{x\gamma}$ is diagonal, and because it is diagonal, the decay rate of each variable is independent from the other variables [5, 37].

The second decay mechanism in the GRU-D algorithm is a **hidden state decay** γ_h. Sometimes, the input decay is not enough to catch all missing information, that's why the second type of decay is needed. This one is responsible for decaying the extracted features from GRU hidden states. It is achieved by decaying the previous hidden state

$\mathbf{h}^{(t-1)}$ before computing the next hidden state $\mathbf{h}^{(t)}$ [5]:

$$\hat{\mathbf{h}}^{(t-1)} = \gamma_h \odot \mathbf{h}^{(t-1)}, \tag{28}$$

here the weights matrix $W_{h\gamma}$ is not diagonal. Additionally, masking vectors are added to standard GRU model equations. Summing up all described changes, the GRU-D algorithm is described with the following equations [5]:

$$\mathbf{z}^{(\mathbf{t})} = \sigma\left(W_{xz}\hat{\mathbf{x}}^{(t)} + W_{hz}\hat{\mathbf{h}}^{(t-1)} + W_{mz}\mathbf{m}^{(t)} + \mathbf{b}_z\right) \tag{29}$$

$$\mathbf{r}^{(\mathbf{t})} = \sigma\left(W_{xr}\hat{\mathbf{x}}^{(t)} + W_{hr}\hat{\mathbf{h}}^{(t-1)} + W_{mr}\mathbf{m}^{(t)} + \mathbf{b}_r\right) \tag{30}$$

$$\tilde{\mathbf{h}}^{(t)} = \phi\left(W_{xh}\hat{\mathbf{x}}^{(t)} + W_{hh}\left(\mathbf{r}^{(\mathbf{t})} \odot \hat{\mathbf{h}}^{(t-1)}\right) + W_{mh}\mathbf{m}^{(t)} + \mathbf{b}_h\right) \tag{31}$$

$$\mathbf{h}^{(t)} = \left(1 - \mathbf{z}^{(\mathbf{t})}\right) \odot \hat{\mathbf{h}}^{(t-1)} + \mathbf{z}^{(\mathbf{t})} \odot \tilde{\mathbf{h}}^{(t)} \tag{32}$$

From the equations above the main differences with GRU (described in subsection **1.3.2**) can be spotted. First, standard $\mathbf{x}^{(t)}$ and $\mathbf{h}^{(t-1)}$ are replaced with $\hat{\mathbf{x}}^{(t)}$ and $\hat{\mathbf{h}}^{(t-1)}$ described in equations (27), (28). The second change - the masking vector $\mathbf{m}^{(t)}$ (described in 24) is fed into the model, and W_{mz}, W_{mr}, W_{mh} are new parameters for it. [5]

The structure of the GRU-D model and its changes in comparison with the GRU structure are shown in **Figure 2.2**.

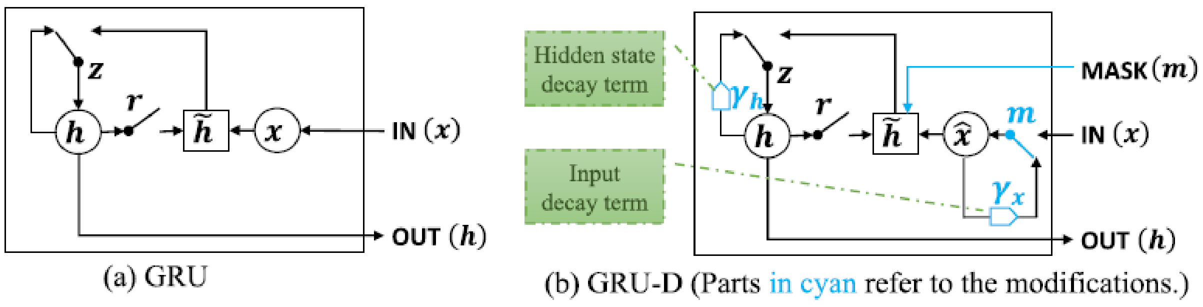


Figure 2.2: Graphical illustrations of the standard GRU model (left side) and the GRU-D model (right side) [5].

2.3 Neural Ordinary Differential Equations

Neural Ordinary Differential Equations (Neural ODEs) first time were introduced by Chen et al. in the year 2018 [8]. This paper received the Best Paper Award at the prestigious NeurIPS 2018 conference and even is treated as a possible landmark paper for a new era of the Deep Learning field [22]. Indeed, the article's authors present an interesting and novel view on neural networks. The proposed new family of deep neural network models uses black-box ODE solvers as a deep neural network component. This idea has many advantages: memory efficiency, adaptive computations, parameters efficiency, scalable and invertible normalizing flows, and is suitable for continuous time series. Although the mentioned article is mainly theoretical, the authors proposed possible applications in supervised learning, density estimation, and irregular time-series modeling (these applications are described in more detail at the end of the section). The most important application for our work, and most probably the most significant one in the mentioned paper, is time-series modeling through **Ordinary Differential Equations (ODEs)**. In comparison to neural networks modeling, such as in earlier sections described RNNs, Neural Ordinary Differential Equations enable flexibility for the tasks related to irregularly and incomplete sampled time series data. [8, 29]

In such deep learning models as Recurrent Neural Networks, Normalizing Flows [53], or Residual Networks (ResNet) [18] hidden layers allow transitioning from a state at time t to the next layer at time $t + 1$ [8]:

$$\mathbf{h}^{(t+1)} = \mathbf{h}^{(t)} + f\left(\mathbf{h}^{(t)}, \boldsymbol{\theta}^{(t)}\right), \tag{33}$$

where $t \in \{0, \dots, T\}$, $\mathbf{h}^{(t)} \in \mathbb{R}^D$ and a function $f(\cdot)$ is differentiable. The equation (33) is very similar to well known **Euler's scheme** [69]:

$$\mathbf{h}^{(t+1)} = \mathbf{h}^{(t)} + f\left(\mathbf{h}^{(t)}, \boldsymbol{\theta}^{(t)}\right) \Delta t. \tag{34}$$

So, it is clear that if in the equation above to set $\Delta t = 1$, exactly the equation (33) will be obtained.

If to add to mentioned well-known Neural Networks much more layers and take smaller steps, the process will be dynamic. Let's set $\Delta t \to 0$, then such a limit is attained [34]:

$$\lim_{\Delta t \to 0} \frac{\mathbf{h}^{(t+1)} - \mathbf{h}^{(t)}}{\Delta t} = \frac{d\mathbf{h}(t)}{dt} = f(\mathbf{h}(t), t, \boldsymbol{\theta}). \tag{35}$$

By using the limit (35), a **discretized ODE** converts into a **continuous ODE**. The main idea of such an approach is to parameterize the continuous dynamics of hidden states using ODEs together with the neural network:

$$\frac{d\mathbf{h}(t)}{dt} = f(\mathbf{h}(t), t, \boldsymbol{\theta}), \tag{36}$$

where function $f(\cdot)$ is the particular neural network parameterized by parameters $\boldsymbol{\theta}$, $\boldsymbol{\theta}$ consists of all the weights of the NN. Also, the obtained equation (36) can be treated as the **Initial Value Problem (IVP)**. With the given input layer $\mathbf{h}(0)$ (initial condition), the output layer $\mathbf{h}(T)$ can be defined as the solution to the ODE initial value problem at time T. The value can be obtained by using the so-called Black-Box Differential Equation Solver. In **Figure 2.3**, both discussed approaches (33) and (36) are presented. [8,34]

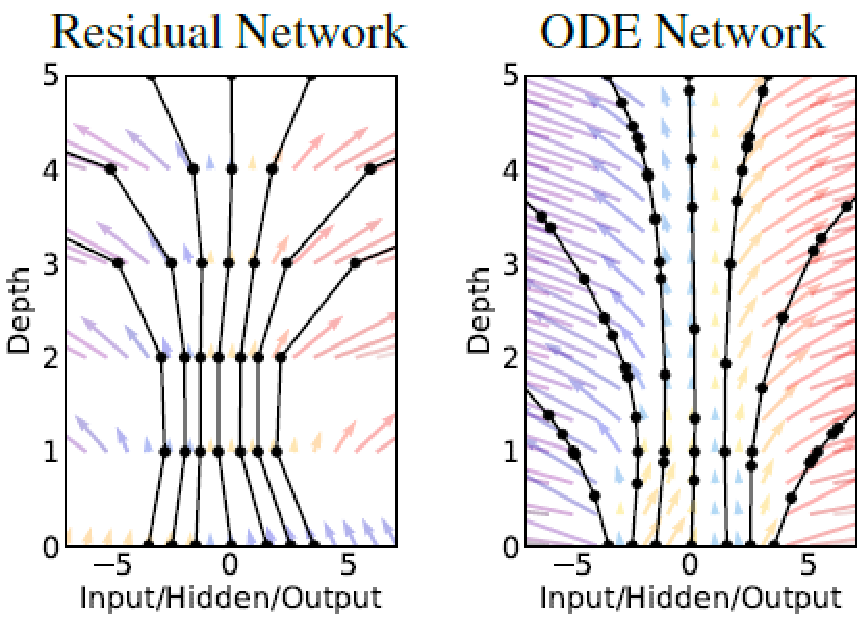


Figure 2.3: In the first image, a Residual Network defines a discrete sequence of finite transformations. The image on the right shows an ODE Network which defines a vector field, which continuously transforms the state. In both cases circles represent evaluation locations. [8]

The standard backpropagation technique, which is described in **Section 1.3**, is not suitable for training continuous-depth neural networks with the ODE solver because solving differential equations is a numerically and memory costly task. In source [8], the ODE solver is treated as a Black Box, and the **Adjoint Sensitivity Method** is proposed for the computation of the gradients [52]. This method scales linearly with the problem size, has significantly lower memory cost, and explicitly controls numerical error.

Let's consider a loss function $\mathcal{L}(\cdot)$ (defined in **Section 1.3**), which evaluates the distance between the output and desired measurement. The input to the loss function is the result of an ODE solver [8, 34]:

$$\mathcal{L}\left(\mathbf{h}\left(t_1\right)\right) = \mathcal{L}\left(\mathbf{h}\left(t_0\right) + \int_{t_0}^{t_1} f(\mathbf{h}(t), t, \boldsymbol{\theta}) dt\right) = \\ \mathcal{L}\left(\text{ODESolve}\left(\mathbf{h}\left(t_0\right), f, t_0, t_1, \boldsymbol{\theta}\right)\right), \tag{37}$$

where the NN $f(\cdot)$ is trained with respect to parameters $\boldsymbol{\theta}$ that minimize the loss function $\mathcal{L}$. The first thing which should be considered is how the gradient depends on the hidden layer $h(t)$ at each instant. For this aim, the quantity called **adjoint** is introduced:

$$\mathbf{a}(t) = \frac{\partial \mathcal{L}}{\partial \mathbf{h}(t)}. \tag{38}$$

Later on, the instantaneous analog to the chain rule (used earlier in the backpropagation) is applied, and this way, dynamics of the adjoint are given via the following ODE:

$$\frac{d\mathbf{a}(t)}{dt} = -\mathbf{a}(t)^{\top} \frac{\partial f(\mathbf{h}(t), t, \boldsymbol{\theta})}{\partial \mathbf{h}}. \tag{39}$$

Now, partial derivative $\frac{\partial \mathcal{L}}{\partial \mathbf{h(t_0)}}$ can be computed starting from the initial value of $\frac{\partial \mathcal{L}}{\partial \mathbf{h(t_1)}}$. However, there is one complication. The value of $\mathbf{h}(t)$ should be known along the entire trajectory. That's why $\mathbf{h}(t)$ is recomputed backward in time, starting from the final value $\mathbf{h}(t_1)$.

Finally, we can compute the gradients with the respect to $\boldsymbol{\theta}$. In this case, the integral depends on $\mathbf{a}(t)$ and $\mathbf{h}(t)$:

$$\frac{d\mathcal{L}}{d\boldsymbol{\theta}} = -\int_{t_1}^{t_0} \mathbf{a}(t)^{\top} \frac{\partial f(\mathbf{h}(t), t, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} dt. \tag{40}$$

All mentioned integrals can be computed at once in a single call to the ODE solver [8]. Algorithm 1 presents how to do it.

The reverse-mode derivative is broken down into the sequence of separate solves, one between each consecutive pair of output times because the investigated loss depends on the intermediate states [8]. It is presented in **Figure 2.4**.

Algorithm 1: Reverse-mode derivative of an ODE initial value problem

```
Input: dynamics parameters $\boldsymbol{\theta}$, start time $t_0$, stop time $t_1$, final state $\mathbf{h}(t_1)$, loss
       gradient $\frac{\partial \mathcal{L}}{\partial \mathbf{h}(\mathbf{t_1})}$

    $s_0 = \left[\mathbf{h}(t_1), \frac{\partial \mathcal{L}}{\partial \mathbf{h}(t_1)}, \mathbf{0}_{|\boldsymbol{\theta}|}\right]$ // Define initial augmented state
  // Define dynamics on augmented state
    Def $aug\_dynamics$ $([\mathbf{h}(t), \mathbf{a}(t), \cdot], t, \boldsymbol{\theta})$:
        return $\left[f(\mathbf{h}(t), t, \boldsymbol{\theta}), -\mathbf{a}(t)^\top \frac{\partial f}{\partial \mathbf{h}}, -\mathbf{a}(t)^\top \frac{\partial f}{\partial \boldsymbol{\theta}}\right]$ // Compute vector-Jacobian
        products
    $\left[\mathbf{h}(t_0), \frac{\partial \mathcal{L}}{\partial \mathbf{h}(t_0)}, \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}\right] =$ ODESolve $(s_0, aug\_dynamics, t_1, t_0, \boldsymbol{\theta})$ // Solve
    reverse-time ODE
return $\frac{\partial \mathcal{L}}{\partial \mathbf{z}(t_0)}, \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$ // Return gradients
```

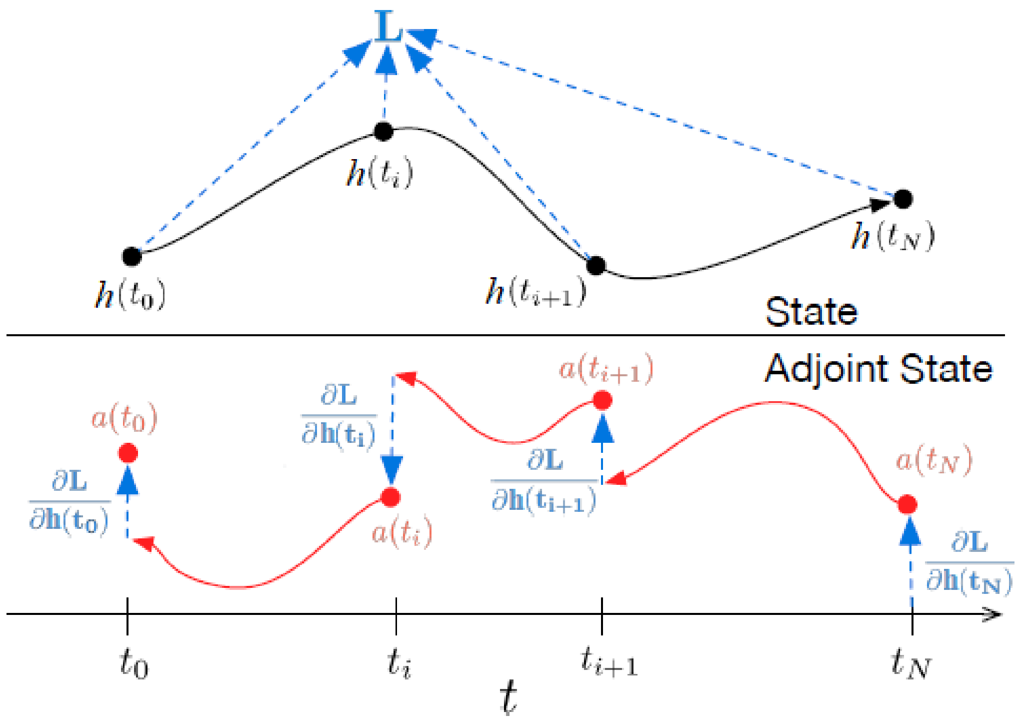

L
$h(t_i)$
$h(t_0)$
$h(t_{i+1})$
$h(t_N)$
State
Adjoint State
$a(t_0)$
$\frac{\partial L}{\partial h(t_0)}$
$\frac{\partial L}{\partial h(t_i)}$
$a(t_i)$
$a(t_{i+1})$
$\frac{\partial L}{\partial h(t_{i+1})}$
$a(t_N)$
$\frac{\partial L}{\partial h(t_N)}$
t_0
t_i
t_{i+1}
t_N
t

Figure 2.4: Reverse-mode differentiation of an ODE solution [8]. The upper image shows the forward propagation. The image below illustrates the adjoint sensitivity method, which solves an augmented ODE backwards in time.

Several applications are proposed in the discussed paper. The examples of applications are on different generated "toy" datasets but show good results and look promising. The first one is the Neural ODE usage in the field of supervised learning. A meaningful result was achieved in this case. With three times fewer parameters, ODE-Net achieved roughly the same results as the ResNets, and the memory complexity of ODE-Net is constant. The second application is to use Neural ODEs to extend normalizing flows into the continuous domain. This approach also simplified the computation of the normalizing constant. The last and the most significant application is the possibility to model irregularly-sampled time series through Neural ODEs [8, 22]. This novel approach has already been tried on time-series trajectories data in recent years [17, 39, 54].

2.4 Ordinary Differential Equations based Recurrent Neural Network

Recently, Rubanova et al. [54] extended the idea proposed in the article [8]. As mentioned earlier, standard Recurrent Neural Networks (described in **Section 1.3**) are an awful fit for the irregular time series data. The timing of the data points is usually a powerful predictor itself. What is more, RNNs are discrete-time models, but usually, real-world trajectories are with continuous-time dynamics, in this thesis too [39]. So, the authors generalized RNNs to have continuous-time hidden dynamics defined by Neural ODEs. This novel model is called **Ordinary Differential Equations based Recurrent Neural Network (ODE-RNN)**. The ODE-RNN model naturally handles time gaps between observations. This ODE-based model is used in the study to solve numerous real-world related problems with sparse data. In all the cases, the ODE-RNN model performed better than several other RNN variants and even showed the new state-of-the-art performance at both interpolation and extrapolation tasks on **MuJoCo Hopper** trajectories simulation [62] and the **PhysioNet** [58] datasets. [54]

The conventional trick is to include the time gap Δt between observations when using RNNs on irregularly sampled time series data. Based on it, let's rewrite the equation (6) for the RNN's hidden state computation in the following way [54]:

$$\mathbf{h}^{(t)} = \mathrm{RNNCell}\left(\mathbf{h}^{(t-1)}, \Delta t, \mathbf{x}^{(t)}\right). \tag{41}$$

However, such an approach is not effective because it is unclear how to define the hidden state between observations. So, the solution to this problem was proposed by earlier discussed Che et al. paper [5] and Mozer et al. [46]. The authors introduced an exponential decay of the hidden state when no observations are made in the time interval:

$$\mathbf{h}^{(t)} = \mathrm{RNNCell}\left(\mathbf{h}^{(t-1)} \odot \exp\left\{-\tau \Delta t\right\}, \mathbf{x}^{(t)}\right), \tag{42}$$

here τ denotes a decay rate parameter. Though, this strategy still did not improve predictive performance.

Rubanova et al. [54] noted that the previous approach - RNN with exponentially-decayed hidden state, implicitly obeys the following ordinary differential equation:

$$\frac{d\mathbf{h}(t)}{dt} = -\tau \mathbf{h}(t), \tag{43}$$

where τ is the parameter of the model and it is the IVP with the initial value $\mathbf{h}(t_0) = \mathbf{h}^{(0)}$. The solution to the differential equation (43) is as the pre-update term from the equation (42): $\mathbf{h}^{(0)} \odot \exp\{-\tau \Delta t\}$. The ODE (43) is **time-invariant**, and it has the special **stationary point** (zero-valued state). The Neural ODE can be used to describe the hidden state. The **Algorithm 2** summarizes this process [54].

Algorithm 2: The ODE-RNN model. The differences from the standard RNNs are highlighted in cyan color.

```
Input: Data points and their timestamps $\left\{\left(\mathbf{x}^{(i)}, t_i\right)\right\}_{i=1...T}$
$\mathbf{h}^{(0)} = \mathbf{0}$
for $i$ in $1, 2, ..., T$ do
    $\mathbf{h}'^{(i)} = \mathrm{ODESolve}\left(f, \mathbf{h}^{(i-1)}, (t_{i-1}, t_i), \boldsymbol{\theta}\right)$ // ODESolve to get state at $t^{(i)}$
    $\mathbf{h}^{(i)} = \mathrm{RNNCell}\left(\mathbf{h}'^{(i)}, \mathbf{x}^{(i)}\right)$ // Updating a hidden state given current
    // observation $\mathbf{x}^{(i)}$
end for
$\mathbf{o}^{(i)} = \mathrm{OutputNN}\left(\mathbf{h}^{(i)}\right)$ for all $i = 1...T$
return $\left\{\mathbf{o}^{(i)}\right\}_{i=1...T}$; $\mathbf{h}^{(T)}$
```

From **Algorithm 2** is seen what changes are made in the standard Recurrent Neural Network. Therefore, the pre-activation states $\mathbf{h}'^{(i)}$ evolve according to a Neural ODE between observations instead of being fixed. Then, the next hidden state is updated using the usual RNN approach: $\mathbf{h}^{(i)} = \mathrm{RNNCell}\left(\mathbf{h}'^{(i)}, \mathbf{x}^{(i)}\right)$.

2.5 Comparison of Algorithms

This section shortly sums up all the methods commonly used for irregularly-sampled time series described in this thesis. The main difference between different kinds of RNNs is the definition of their hidden state dynamics. Table 2.1 lists the hidden state dynamics of the RNN models family, GRU-D and ODE-RNN.

From table 2.1 is seen that the hidden state remains the same between updates in simple RNNs. In the ODE-RNNs models family case, the hidden state is defined by

Model	Hidden state between observations
Standard RNN	$\mathbf{h}^{(t-1)}$
GRU-D	$\mathbf{h}^{(t-1)}e^{-\tau\Delta t}$
ODE-RNN	$\text{ODESolve}\left(f, \mathbf{h}^{(i-1)}, (t_{i-1}, t_i), \boldsymbol{\theta}\right)$

Table 2.1: Change of the hidden states in different types of RNNs between observations t_{i-1} and t_i.

ODE and updated at each observation. Also, ODE-based models allow more flexible parameterization of the dynamics than RNNs with exponential decays like GRU-D [35,54].

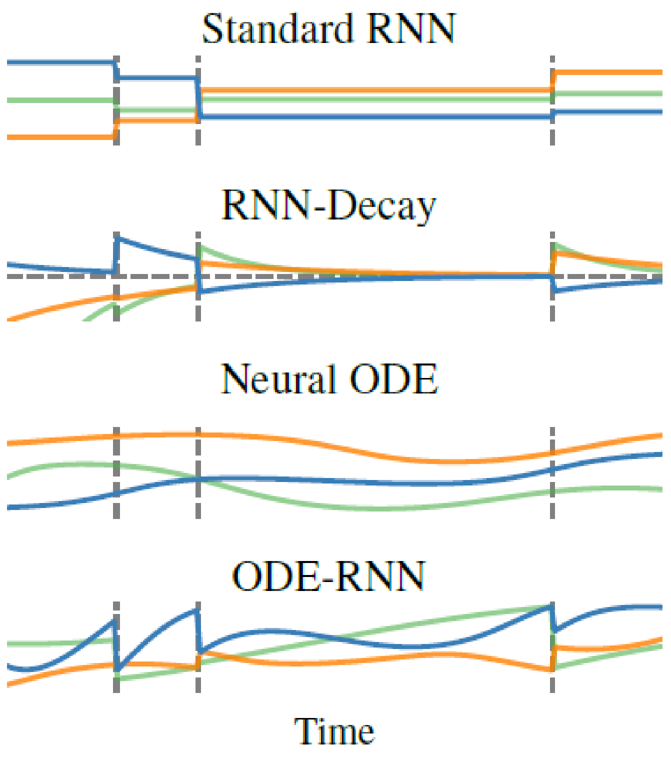


Figure 2.5: Hidden states trajectories in different kind of models. Vertical lines mark time points with observations [54].

Figure 2.5 shows the hidden state trajectories of earlier discussed models types. It is seen that simple RNNs have constant hidden states between observations when time series are irregularly-sampled. In the case of RNN-Decay models (e.g., GRU-D) - the hidden states decay exponentially to zero and are updated at time point with observations. Neural ODEs can follow more complex trajectories than both previous algorithms but depend on the initial state. The last ODE-RNN model's hidden states follow an ODE between observations and are modified at each observed point [35, 54].

2.6 ODE-based Gated Recurrent Unit with Trainable Decays

This section presents the new proposed model called **Ordinary Differential Equations based Gated Recurrent Unit with Trainable Decays (ODE-GRU-D)**. As the model's name says, it is created based on earlier described GRU-D (section 2.2) and ODE-RNN (section 2.4) models. The ODE-RNN model showed promising results on irregularly-sampled time series data by combining Neural ODEs with standard RNN. Instead of using standard RNN, we decided to combine Neural ODEs with the GRU-D model, which is itself the method that performs end-to-end learning on multivariate irregularly-sampled time series. The first reason such modification is selected is that the decay part of the model allows spotting the information directly from irregular time series. For example, the decay mechanism can spot that if the variable is missing for a while, the influence of the input variables fades away over time [5]. Secondly, Lechner et al. [35] proved the theorem which states that ODE-RNN suffers from a vanish or exploding gradient similarly to standard RNN. This problem can be solved by replacing vanilla RNN with LSTM or GRU neural networks variants. So, GRU-D is a suitable choice to avoid those issues.

Algorithm 3 summarizes the ODE-GRU-D model. The main changes in comparison with Algorithm 2 is the state's definition between observations as the ODE solution: $\mathbf{h}'^{(i)} = \text{ODESolve}\left(f, \mathbf{h}^{(i-1)}e^{-\tau\Delta t}, (t_{i-1}, t_i), \boldsymbol{\theta}\right)$ - simple $\mathbf{h}^{(i-1)}$ changed to $\mathbf{h}^{(i-1)}e^{-\tau\Delta t}$ as in the standard GRU-D case. Also, instead of updating the hidden state using a standard RNN as in the ODE-RNN algorithm, hidden states are updated with the GRU-D approach: $\mathbf{h}^{(i)} = \text{GRUDCell}\left(\mathbf{h}'^{(i)}, \mathbf{x}^{(i)}, \mathbf{m}^{(i)}\right)$.

Algorithm 3: The ODE-GRU-D model.

```
Input: Data points and their timestamps $\left\{\left(\mathbf{x}^{(i)}, t_i\right)\right\}_{i=1...T}$
$\mathbf{h}^{(0)} = \mathbf{0}$
$\left\{\mathbf{m}^{(i)}\right\}_{i=1...T}$ calculated based on formula (24)
for $i$ in $1, 2, ..., T$ do
    $\mathbf{h}'^{(i)} = \text{ODESolve}\left(f, \mathbf{h}^{(i-1)}e^{-\tau\Delta t}, (t_{i-1}, t_i), \boldsymbol{\theta}\right)$ // ODESolve to get state at time $t^{(i)}$
    $\mathbf{h}^{(i)} = \text{GRUDCell}\left(\mathbf{h}'^{(i)}, \mathbf{x}^{(i)}, \mathbf{m}^{(i)}\right)$ // Update of the hidden state
end for
$\mathbf{o}^{(i)} = \text{OutputNN}\left(\mathbf{h}^{(i)}\right)$ for all $i = 1...T$
return $\left\{\mathbf{o}^{(i)}\right\}_{i=1...T}$; $\mathbf{h}^{(T)}$
```

In Algorithm 3, $\mathbf{m}^{(i)}$ denotes masking vectors, $\boldsymbol{\theta}$ are Neural Network's parameters, and τ is a decay rate parameter.

2.7 Performance Metrics and Statistical Tests

2.7.1 Performance Metrics

Performance measures are a crucial part of every deep learning pipeline. In this subsection, the performance metrics used to evaluate the investigated models are described. Because our problem is of the **regression** type (models have continuous outputs), the performance measurements based on calculating a distance between predicted and ground truth should be chosen. Therefore, we are using the two most commonly used absolute error metrics: **Mean Squared Error (MSE)** and **Root Mean Squared Error (RMSE)**.

The MSE metric finds the average of the squared difference between the target value and the value predicted by the model. It is calculated based on the formula [56]:

$$MSE = \frac{1}{n}\sum_{t=1}^{n}\left(y^{(t)} - \widehat{y}^{(t)}\right)^2. \tag{44}$$

Similarly, RMSE corresponds to the square root of the average of the squared difference between the target value and the value predicted by some ML model [56]:

$$RMSE = \sqrt{\frac{1}{n}\sum_{t=1}^{n}(y^{(t)} - \widehat{y}^{(t)})^2} = \sqrt{MSE}. \tag{45}$$

Here n denotes the number of observations, $y^{(t)}$ is a ground-truth value at the time t, and $\widehat{y}^{(t)}$ - the predicted value by the model.

2.7.2 Statistical Tests

A **statistical hypothesis test** is a mathematical tool for analyzing quantitative data. Such tests also can be useful in comparing machine learning models and choosing the best or final model. There are many different statistical tests. For this master thesis, two widely used tests are selected: **Two-Sample t-Test** and **Welch's t-Test** based on the data distribution, structure, and variable type (more details in **Chapter 3**). [36, 48]

The **Two-Sample t-Test** determines if two data samples means are equal or differ significantly [59] . The null hypothesis states that the two samples' means (μ_1 and μ_2)

are the same, while the alternative hypothesis states that they are not [48]:

$$\mathrm{H}_0 : \mu_1 = \mu_2 \tag{46}$$

$$\mathrm{H}_1 : \mu_1 \neq \mu_2 \tag{47}$$

If sample sizes are equal the t statistic to test the above hypotheses can be calculated this way [59]:

$$t = \frac{\bar{Y_1} - \bar{Y_2}}{s_p\sqrt{\frac{2}{n}}}, \tag{48}$$

where

$$s_p = \sqrt{\frac{s^2_{Y_1} + s^2_{Y_2}}{2}}. \tag{49}$$

Here $\bar{Y_1}$ and $\bar{Y_2}$ are the observed two groups means, s_p denotes the pooled standard deviation when $n = n_1 = n_2$ (our case), $s^2_{Y_1}$ and $s^2_{Y_2}$ are the unbiased estimators of the variances.

The Two-Sample t-Test has several assumptions [48]:

1. **Independence**. The observations are independently sampled - no relation in observations between the groups and within the groups.

2. **Normality**. Both groups are normally distributed. It can be checked with **Shapiro-Wilk test** (52).

3. **Homogeneity (homoscedasticity) of variance**. Standard deviations of samples are approximately equal. It can be checked with **Levene's** or **Bartlett's Test** (54).

The **Welch's t-Test** has the same assumptions as listed above except the third. The t statistic to check if the groups' means are different is calculated by the following formula [36,48]:

$$t = \frac{\bar{Y_1} - \bar{Y_2}}{\sqrt{s^2_{\bar{Y_1}} + s^2_{\bar{Y_2}}}}, \tag{50}$$

where

$$s_{\bar{Y_i}} = \frac{s_i}{\sqrt{n_i}} \tag{51}$$

Here $\bar{Y_1}$ and $\bar{Y_2}$ are the samples means, $s_{\bar{Y_1}}$ and $s_{\bar{Y_2}}$ are standard errors, n_i is the i^{th} sample size, and s_i denotes the i^{th} sample standard deviation.

Earlier mentioned **Shapiro-Wilk test** calculates a W statistic that tests whether a random sample is normally distributed (H_0 - data are normally distributed) [55]. The test statistic is calculated as follows [48]:

$$W = \frac{\left(\sum_{i=1}^{n} a_i x_{(i)}\right)^2}{\sum_{i=1}^{n}\left(x_i - \bar{X}\right)^2}, \tag{52}$$

where $x_{(i)}$ is the i^{th} smallest value in the sample, $\bar{X}$ denotes a sample mean, and a_i are constants generated from the means, variances, and covariances of the order statistics of a sample of size n drawn from a normal distribution (more details in the [55] article).

The **Bartlett's Test** is used in this work for checking homogeneity assumption [59]. In this test case, the null hypothesis H_0 states that all k samples have equal variances against the alternative hypothesis H_1 that at least two of them are different:

$$\begin{aligned} \mathrm{H}_0 &: \ \sigma_1^2 = \sigma_2^2 = \ldots = \sigma_{\mathrm{k}}^2 \\ \mathrm{H}_1 &: \ \sigma_i^2 \neq \sigma_j^2 \quad \text{for at least one pair } (i,j). \end{aligned} \tag{53}$$

Th Bartlett's test statistic can be found by the formula [59]:

$$\chi^2 = \frac{(N-k)\ln\left(S_p^2\right) - \sum_{i=1}^{k}(n_i - 1)\ln\left(S_i^2\right)}{1 + \frac{1}{3(k-1)}\left(\sum_{i=1}^{k}\left(\frac{1}{n_i-1}\right) - \frac{1}{N-k}\right)}, \tag{54}$$

here

$$N = \sum_{i=1}^{k} n_i \ \text{ and } \ S_p^2 = \frac{1}{N-k}\sum_i (n_i - 1)\, S_i^2. \tag{55}$$

Also, n_i denotes sample sizes and S_i^2 - sample variances.

In all the tests, we reject H_0 if a p-value is less than or equal to the significance level α. The most often used α values are 0.1, 0.05 or 0.01. We used $\alpha = 0.05$ in the practical part. The p-value is obtained by using the sampling distribution of the test statistic under the null hypothesis, type of statistical test, and the sample data [36, 48].

3 Experiments

In this work, all the experiments were performed using Python 3 high-level programming language, PyTorch framework, Google Colaboratory, and NVIDIA Tesla K80 GPU. Two main libraries for our experiments are **torchdiffeq** [8] which provides ODE solvers implemented in PyTorch, and **umap-learn** [43] which provides techniques for dimension reduction.

Four different models are selected for the experiments based on the overview of the commonly used methods in the previous chapters. The first selected model is one of the classic RNNs - GRU. It is selected to analyze if such a model can be useful in some cases of irregular time series data. Also, two specific algorithms for irregularly-sampled time series are chosen GRU-D and ODE-RNN. GRU-D is probably the most often used algorithm in this field, and ODE-RNN is the current state of the art. The fourth model is the one proposed in this thesis - ODE-GRU-D.

For all the models' same hyperparameters are used. Hyperparameters were selected mainly based on the article [54]. We are using **Adamax** optimizer (extension of the standard Adam) [30] with the learning rate of 0.001. The batch size is 50, and the models' hidden states are 15-dimensional. In ODE-based models, ODE functions have 3 layers and 100 units, and the **Euler** method [69] is used in the ODE-Solver.

The experiments can be divided into two main parts. In the first part, the models are trained with 300 epochs for each data case to select the optimal number of epochs and compare the models during the training. In the second part, the experiments are done with the optimal number of epochs. Each algorithm for each data case is trained ten times, and averages with standard deviation are computed and compared between the models.

3.1 Data

The **MuJoCo Hopper** trajectories dataset [7] is used for the experiments. Hopper is a one-legged jumping robot. The dataset is taken from University of Toronto, Vector Institute resources. The data were simulated using the advanced MuJoCo physics engine for multi-body dynamics and DeepMind Control Suite [62]. 10000 trajectories with 100 regularly sampled time points for each are generated. The dataset is 14-dimensional, the state space is 11-dimensional - position and velocity of each joint are tracked, the action space is 3-dimensional [14]. In this set, the position of the body is uniformly sampled from the interval [0, 0.5]. Also, the relative positions of the limbs are sampled from [-2, 2] and initial velocities are in [-5, 5] interval. The figures below show a small part of the investigated data [54].

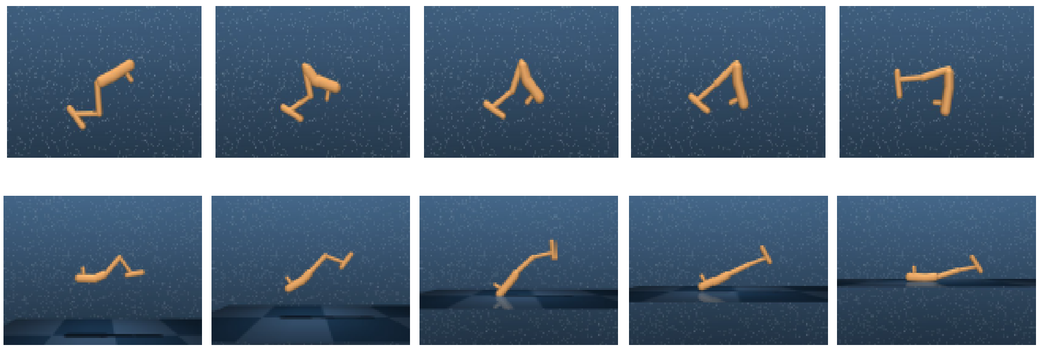


Figure 3.1: A small example from the MuJoCo Hopper trajectories dataset. [54]

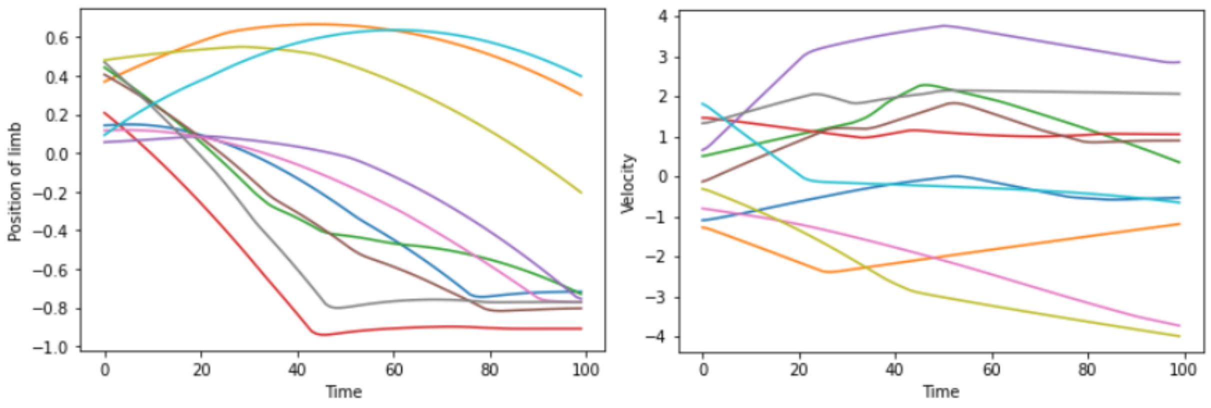


Figure 3.2: Example of 2 dimensions of the MuJoCo Hopper data (10 first trajectories).

The MuJoCo Hopper dataset is subsampled before the experiments into the train set where are 70% of the data (trajectories) and the remaining data are for validation 15% and testing 15%.

3.2 Experiment 1: Choice of the suitable number of epochs

The first experiment aims to select the optimal number of epochs for training. Before running the models, we subsampled some percentage of time points to have irregularly-sampled data. This way, four different data cases are obtained. In the first case, 90% of time points are observed, so the irregularity of the data is small. In the second case, training and test datasets have 70% of time points with the observations, and in the third case - 50% of the observations are successive. Only 30% of the time points are observed in the last case - such data are called sparse because the intervals between successive observations are large.

All selected models were trained with 300 epochs (recommended in the article [54]) for each mentioned data case. Tables 3.1-3.4 present the results of all the trials every 50 epochs. The best results for the particular epoch are marked as **bold**.

Model/Epoch	**50**	**100**	**150**	**200**	**250**	**300**
GRU	0.00235	0.00189	0.00181	0.00199	0.00167	0.00171
GRU-D	0.00225	0.00211	0.00191	0.00197	0.00169	0.00169
ODE-RNN	0.00147	0.00141	0.00137	**0.00133**	0.00143	0.00141
ODE-GRU-D	**0.00133**	**0.00132**	**0.00129**	0.00137	**0.00129**	**0.00137**

Table 3.1: Models validation MSE during the epochs - 90% case.

Model/Epoch	**50**	**100**	**150**	**200**	**250**	**300**
GRU	0.00546	0.00493	0.00491	0.00509	0.00443	0.00464
GRU-D	0.00545	0.00467	0.00461	0.00546	0.00458	0.00422
ODE-RNN	0.00411	0.00407	0.00406	0.00393	**0.00394**	**0.00403**
ODE-GRU-D	**0.00401**	**0.00368**	**0.00401**	**0.00369**	0.00396	0.00405

Table 3.2: Models validation MSE during the epochs - 70% case.

Model/Epoch	**50**	**100**	**150**	**200**	**250**	**300**
GRU	0.00938	0.00876	0.00848	0.00883	0.00786	0.00778
GRU-D	0.00808	0.00751	0.00778	0.00761	0.00734	0.00719
ODE-RNN	0.00692	0.00684	0.00687	0.00659	**0.00666**	0.00685
ODE-GRU-D	**0.00688**	**0.00655**	**0.00677**	**0.00653**	0.00671	**0.00669**

Table 3.3: Models validation MSE during the epochs - 50% case.

Model/Epoch	**50**	**100**	**150**	**200**	**250**	**300**
GRU	0.01465	0.01373	0.01290	0.01391	0.01225	0.01198
GRU-D	0.01234	0.01308	0.01225	0.01326	0.01319	0.01168
ODE-RNN	**0.01026**	0.01011	0.01005	0.01012	**0.00995**	0.01018
ODE-GRU-D	0.01041	**0.01004**	**0.00989**	**0.00971**	**0.00995**	**0.01011**

Table 3.4: Models validation MSE during the epochs - 30% case.

The proposed ODE-GRU-D model showed the best validation MSE in almost all situations, but the ODE-RNN model achieved slightly better MSE in table 3.1 with 200 epochs, 250 and 300 in table 3.2, with 250 epochs in 50% case (table 3.3). Also, from the tables above, we see that the proposed in this thesis model needs a smaller amount of epochs than ODE-RNN algorithm, but validation MSEs become closer between these two models after more epochs. A similar pattern can be spotted between GRU and GRU-D models. So, it can be concluded that such an effect is achieved by adding a decay mechanism (26) to the RNNs. Interesting that simple GRU in a few scenarios showed even better results than more advanced GRU-D model, although it is a simpler model and much faster to train, especially on more irregular data.

The learning curves are plotted to determine the optimal number of epochs for each model and dataset. From the figures 3.3 - 3.6, similar conclusions can be made as from the tables. The ODE-GRU-D notably converges faster than the original ODE-RNN. The gap between ODE-GRU-D and ODE-RNN learning curves, which is at the beginning of training, seems to decrease when data become more irregular, nevertheless, just in the case with 30% of successive observations, the ODE-GRU-D learning curve is almost whole the time below the ODE-RNN curve. Additionally, we can spot that ODE-based models are much more stable (losses are less deviated) than simple recurrent networks, especially on more irregularly-sampled time series. Obviously, losses start to converge earlier than after 300 epochs in all the investigated cases. Therefore, based on the learning curves, 75 is selected as the optimal number of epochs for the 90% case, for all the others - 50 seems to be enough.

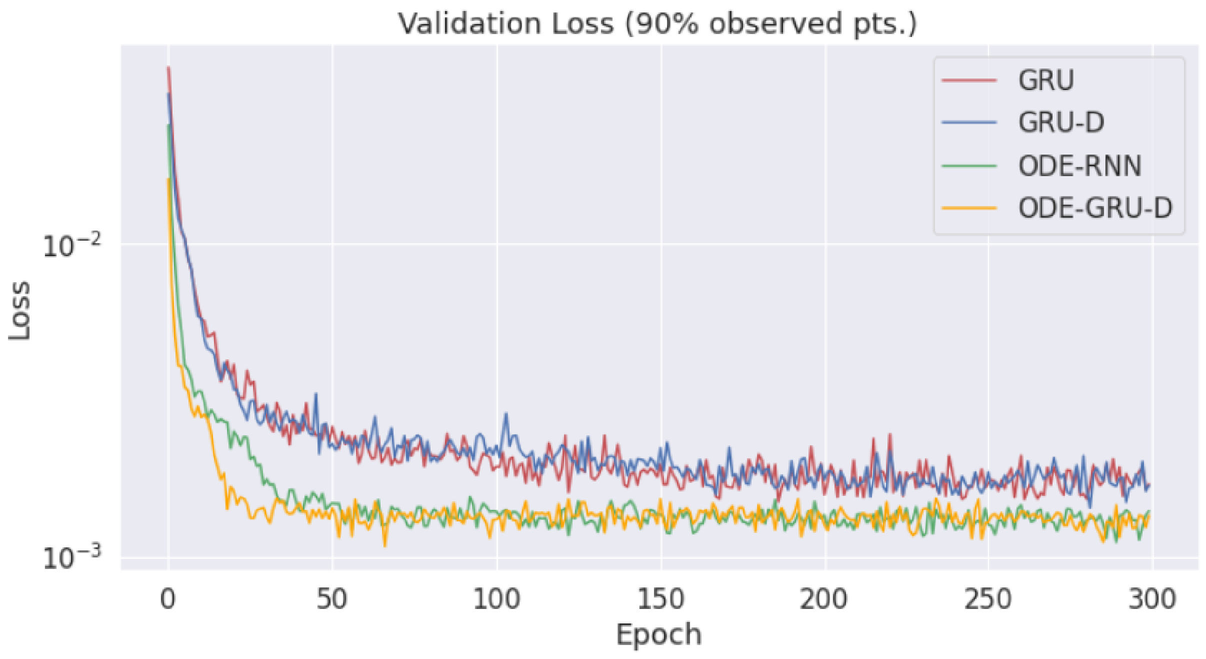


Figure 3.3: Learning curves when 90% of time points are observed.

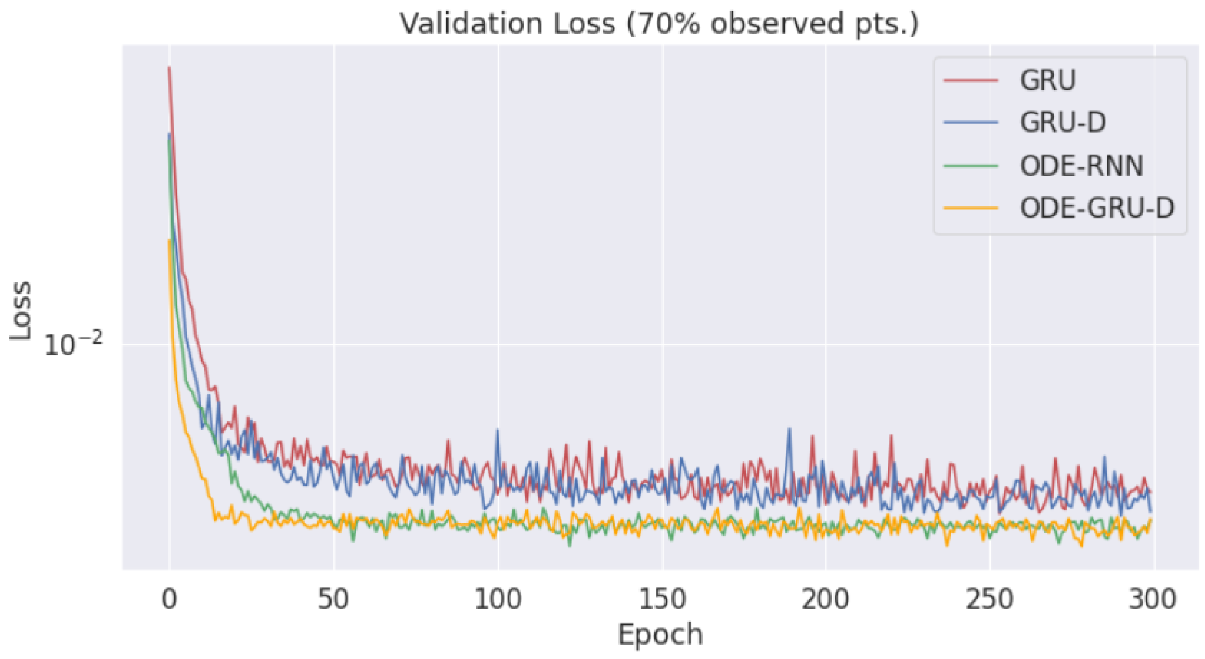


Figure 3.4: Learning curves when 70% of time points are observed.

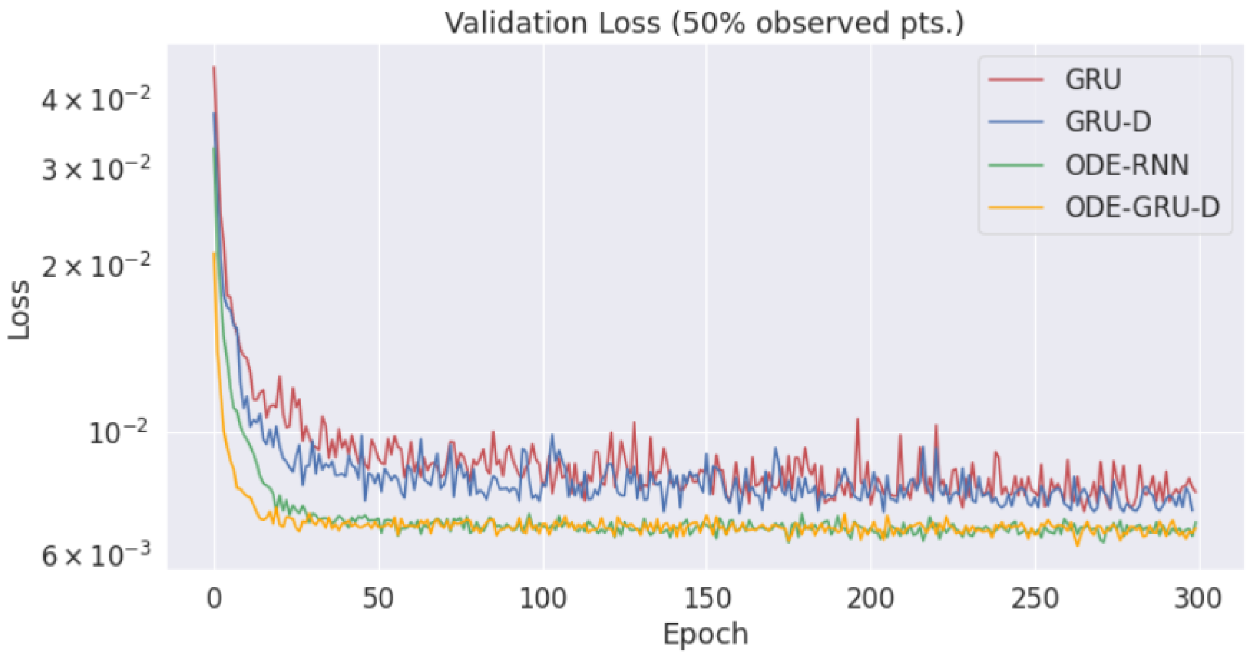


Figure 3.5: Learning curves when 50% of time points are observed.

Finally, we tested all four fully trained models. Table 3.5 summarizes the results. The best test MSE for each data case is marked as **bold**. In all the situations, our proposed ODE-GRU-D showed slightly better test MSE than the rest of the algorithms. It can be seen that in the cases of more irregular time series data (where 50% and 30% of time points are observed) the distance between two ODE-based models losses is bigger, than

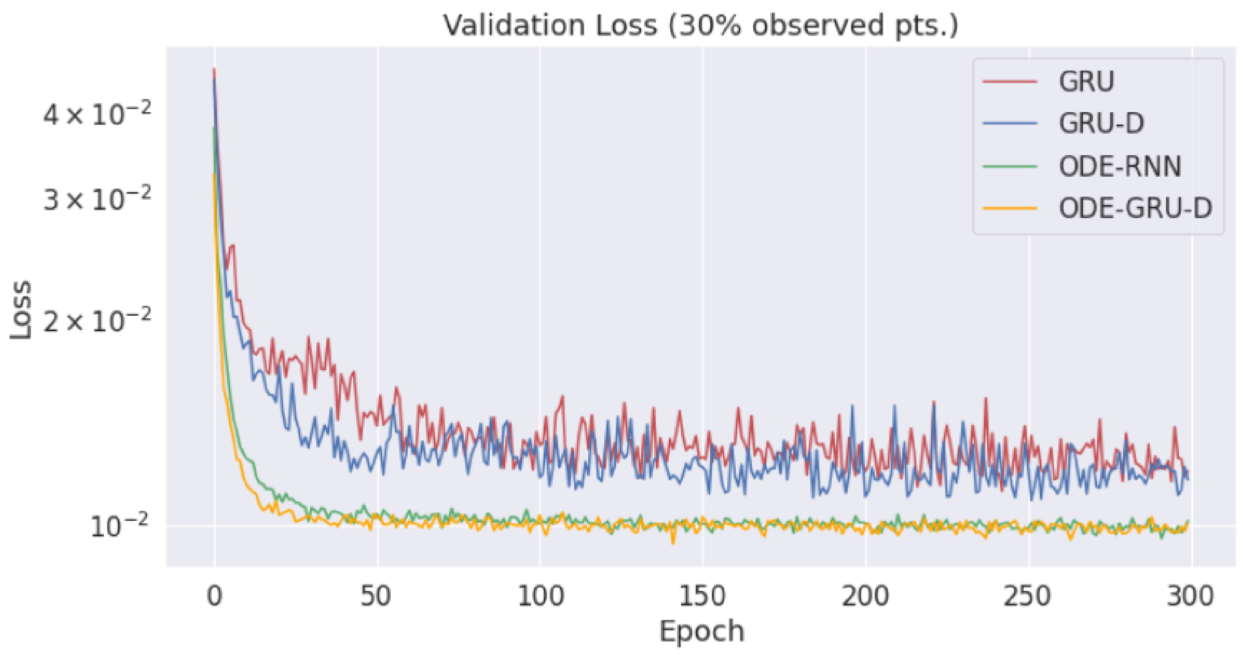


Figure 3.6: Learning curves when 30% of time points are observed.

in less irregular cases (90% and 70%). That can mean that our proposed implementation is more beneficial for sparse time series cases. When comparing GRU with GRU-D, the same trends emerge. Also, when data are more sparse the difference between the ODE-based and simple RNNs losses becomes more significant too.

Model/Data	**90%**	**70%**	**50%**	**30%**
GRU	0.00165	0.00465	0.00781	0.01198
GRU-D	0.00163	0.00421	0.00721	0.01168
ODE-RNN	0.00140	0.00404	0.00687	0.01018
ODE-GRU-D	**0.00136**	**0.00401**	**0.00671**	**0.01010**

Table 3.5: Models test MSE on the different percentage of observed time points.

Shortly, from the first experimental part can be concluded:

- A simple GRU recurrent neural network is a better choice than the GRU-D algorithm for time-series data with small irregularity. They both achieve similar MSE values on the test set, but GRU is much easier to implement and to train;
- ODE-based models are a notably better choice for all the cases of irregularly-sampled time series data than models without ODE part;
- The proposed in this work ODE-GRU-D model starts to converge after fewer epochs than earlier proposed ODE-RNN;

- ODE-GRU-D achieved slightly better results on all four irregularly-sampled time-series datasets. This implementation looks to be especially useful on more sparse data cases.

3.3 Experiment 2: Models evaluation

Even though we got quite many conclusions from the first practical part, and it showed some benefits of the proposed ODE-GRU-D algorithm, these experiments are not enough to claim that the proposed new neural network is always better. To achieve more reliable results, in this experimental part we train the models with the optimal numbers of epochs found in the previous section (75 for 90% case and 50 for all the rest). Each model is trained 10 times on each case of the data (90%, 70%, 50%, and 30% of time points with the observations). Time points are randomly sampled from different parts of the timeline before every experiment on every data case. So, every time we obtain a different dataset, but with the same percent of the observations. In sum, 160 such experiments are done.

Box plots 3.7 - 3.10 are created to interpret obtained results for all 4 investigated irregularly-sampled time series data cases. These graphs provide a short, visual summary of how the losses are distributed. The information that we get from the box plot is the five-number summary: minimum, first quartile, median, third quartile, and maximum.

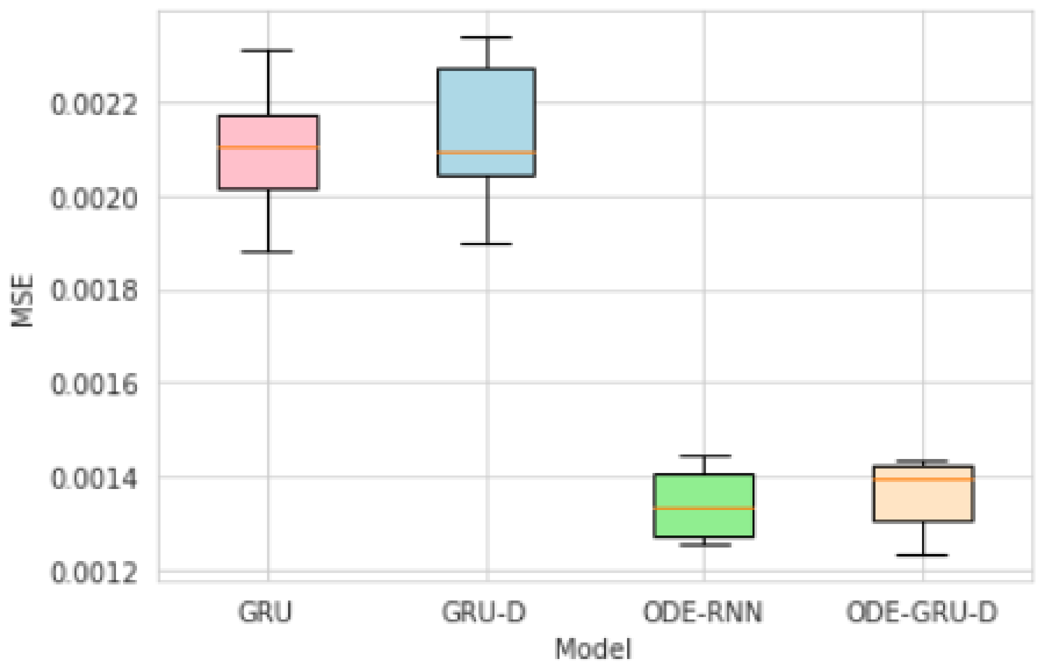


Figure 3.7: Boxplot of the models test MSEs for 90% case.

From the plot 3.7 it is seen that simple RNNs test MSEs values differ significantly in comparison with ODE-based RNNs. Generally, boxplots for GRU and GRU-D models look similar, but the GRU model even looks more stable. So again we can claim that GRU is better choice for the 90% case than its modification. Also, boxplots for ODE-RNN and ODE-GRU-D models look similar. The median value is a bit lower for the ODE-RNN model than for the ODE-GRU-D, but it is difficult to make strong conclusions only based on it.

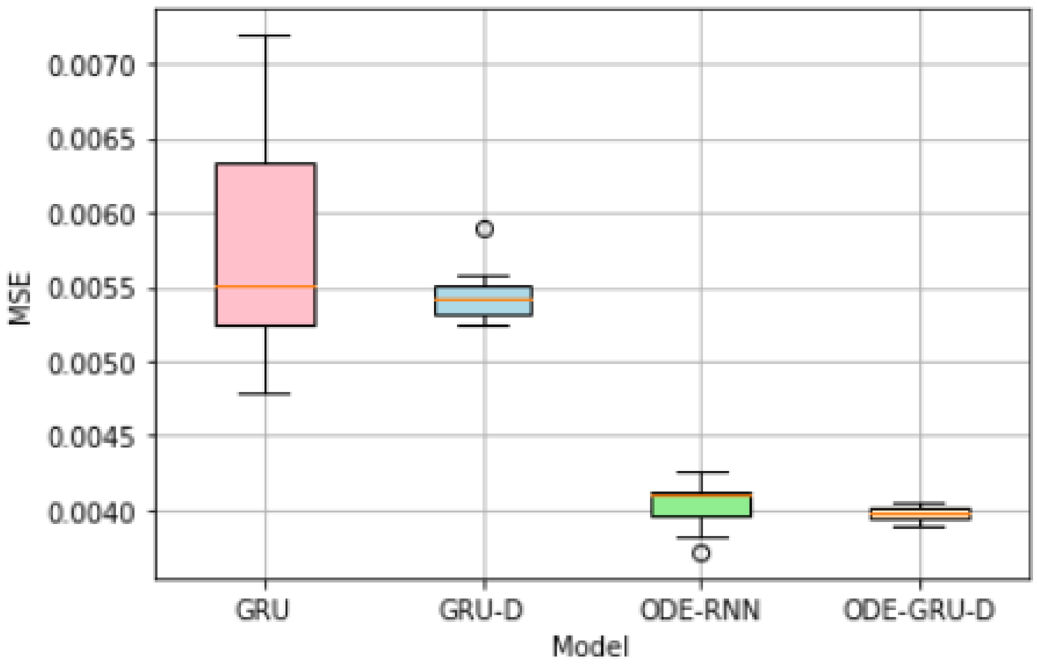


Figure 3.8: Boxplot of the models test MSEs for 70% case.

Figure 3.8 shows more interesting results. Again, ODE-based models are much better choice than simple RNNs. However, the GRU-D model is much less deviated than the simple GRU in this data case. Also, the proposed ODE-GRU-D model looks to perform better than in the previous 90% case - losses are more stable, and the median is lower than those obtained with ODE-RNN.

Figure 3.9 presents the box plots for the data case with 50% of observed time points. Generally, the insights are the same as in earlier situations. Based on the figure, some advantages are achieved by adding a decay mechanism to RNNs models, and of course Neural ODEs improve the models significantly. The proposed in the thesis model is the best choice for this case.

The last graph (Figure 3.10) of this type presents the boxplots for the sparse time series case where only 30% of time points had observations. Here, the GRU-D algorithm

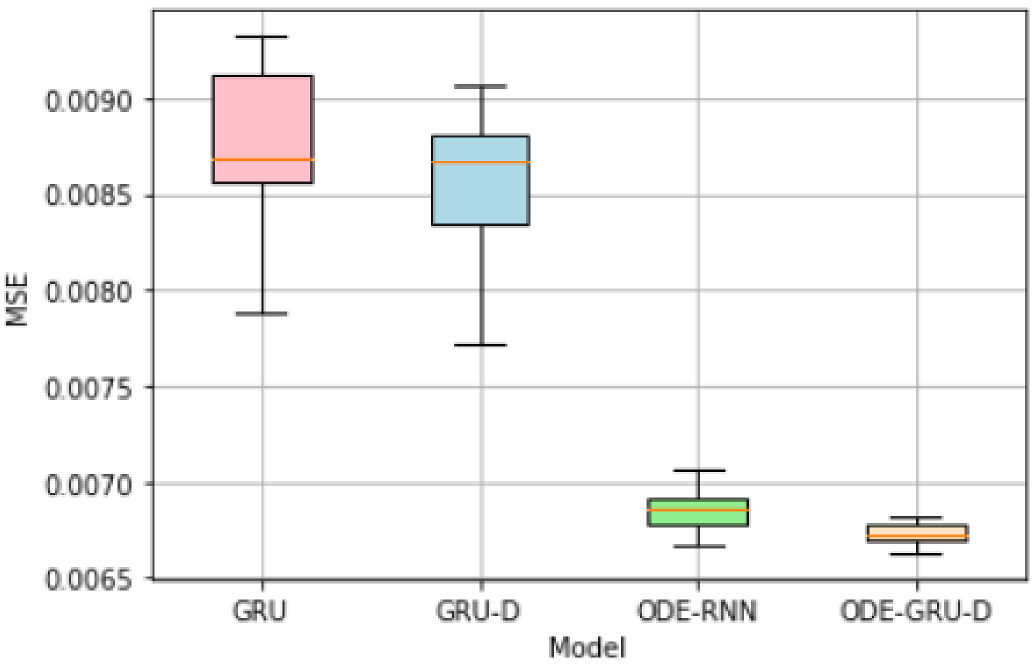


Figure 3.9: Boxplot of the models test MSEs for 50% case.

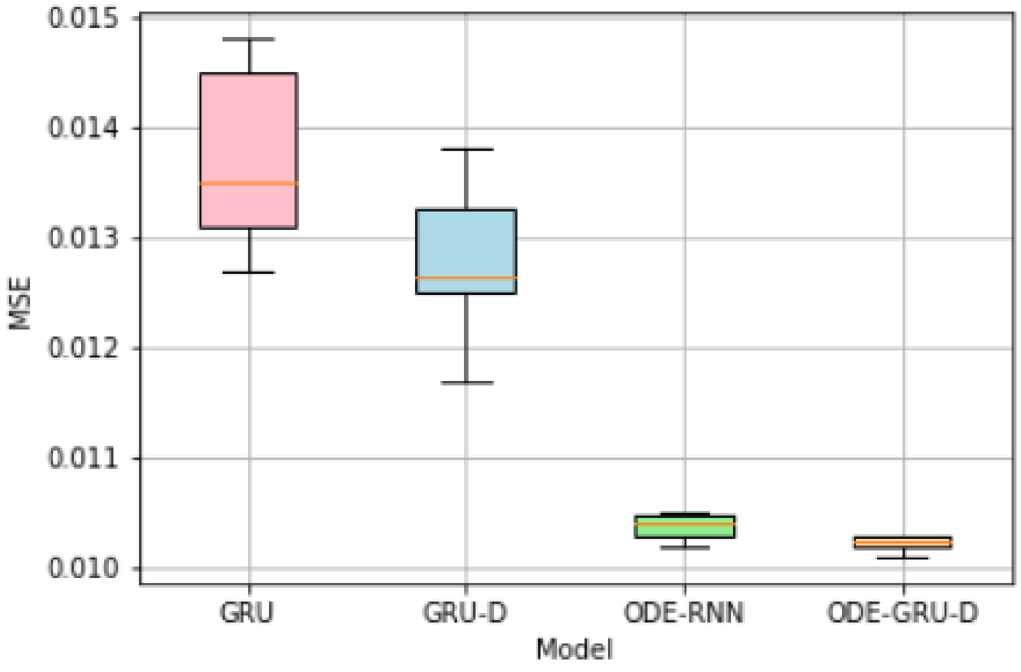


Figure 3.10: Boxplot of the models test MSEs for 30% case.

already shows much better results than GRU (less deviated and lower median, minimum, maximum values). ODE-based models are more stable than simple RNNs, as in all the cases, but in this one especially. Furthermore, the ODE-GRU-D model is better than others in all the measures provided by the boxplots.

Another type of experiments summary is depicted in table 3.6. In this table, **average**

test RMSEs together with **standard deviations** are calculated for all the investigated cases. Basically, it shows similar trends which we discussed above based on the boxplots. In all the cases except the 90% case, the best average test RMSE was obtained with the ODE-GRU-D algorithm. What is more, all the standard deviations are the lowest with the proposed in this paper algorithm, which means this model is even more stable than the ODE-RNN.

Model/Data	90%	70%	50%	30%
GRU	0.0458 ± 0.0015	0.0759 ± 0.0053	0.0936 ± 0.0024	0.117 ± 0.0034
GRU-D	0.0461 ± 0.0016	0.0738 ± 0.0013	0.0926 ± 0.0025	0.1127 ± 0.003
ODE-RNN	$\mathbf{0.0366 \pm 0.0011}$	0.0636 ± 0.0013	0.0829 ± 0.0008	0.1019 ± 0.0006
ODE-GRU-D	0.0369 ± 0.001	$\mathbf{0.0631 \pm 0.0004}$	$\mathbf{0.0820 \pm 0.0004}$	$\mathbf{0.1012 \pm 0.0004}$

Table 3.6: RMSE (mean ± std) on the different percentage of observed time points.

Based on this practical part, we can make similar conclusions as in Section 3.2, but now they are more substantiated. However, the ODE-GRU-D achieved slightly worse results on the test sets than the ODE-RNN on the least sparse time series case. The new conclusion from these repetitive experiments is that the new proposed model is less deviated in general than the earlier proposed variant.

3.4 Comparison using statistical tests

In Section 3.3, we compared the average losses between the models. In this section, we will analyze if these means differ statistically significantly using samples with test RMSE losses obtained in the second experiment (Section 3.3). The proposed ODE-GRU-D model showed lower test RMSEs in the cases with time series where 30%, 50%, and 70% of time points have observations, and slightly worse only in the case with 90% of the observed data. Obviously, both ODE-based models achieved much better results than RNNs without ODE parts (GRU and GRU-D). Therefore, it will be enough to test ODE-GRU-D against ODE-RNN to claim if our introduced model is significantly better in some irregularly-sampled time series cases. The analysis will be done using the statistical tests described in Subsection 2.7.2, and $\alpha = 0.05$ is selected as a significance level.

First of all, we should check the assumptions introduced in Subsection 2.7.2 to know which t-test to use. The first independence assumption is fulfilled - observations between the samples are not related. The second assumption claims that samples should be drawn from the normal distribution. We used the Shapiro-Wilk test (52) to check it on the data (losses) groups. Table 3.7 shows obtained p-values on all cases samples using this test. All the groups are drawn from normal distribution because p-values are non-significant (> 0.05). Therefore, the second assumption is also matched.

Model/Data	**90%**	**70%**	**50%**	**30%**
ODE-RNN	0.3081	0.2659	0.8563	0.9724
ODE-GRU-D	0.0638	0.9261	0.8953	0.1214

Table 3.7: p-values obtained with Shapiro-Wilk test on the models test RMSE losses.

The last assumption is about the homogeneity (homoscedasticity) of variances. Bartlett's test (54) for homogeneity of variances is used to check this point. If this assumption is met, the Two-Sample t-test (47) should be applied to test if means differ statistically significantly. Otherwise, Welch's t-test (50) can be used to test the hypothesis. These tests outcomes are summarised in table 2.7.2.

Test/Data	**90%**	**70%**	**50%**	**30%**
Bartlett's Test	0.9576	0.0012	0.0480	0.2910
Two-Sample t-Test	0.4557	-	-	0.0041
Welch's t-Test	-	0.2899	0.0088	-

Table 3.8: p-values obtained with Bartlett's Test, Two-Sample t-Test, and Welch's t-Test.

From table 3.8, we see that in cases where 90% and 30% of the time points with successive observations, the homogeneity assumption is fulfilled (p-value>0.05). Therefore, the Two-Sample t-Test should be used for these two cases. In the other cases (70% and 50%), variances are not approximately equal (p-value<0.05). So, it means that Welch's test should be selected to check if ODE-RNN and ODE-GRU-D test losses means differ significantly.

The ODE-RNN and ODE-GRU-D test RMSEs averages do not differ significantly (p-value=0.9576) in 90% of the irregularly-sampled time series case. So, even though the ODE-RNN model showed slightly better losses in the previous section, they are not significantly better than those obtained with the new proposed model. In the second (70%) case, we got p-value=0.2899>0.05 with Welch's t-Test. So, although the ODE-GRU-D algorithm showed a bit better results in the experiments on this case, the losses are not significantly better than obtained with the ODE-RNN model. In two more sparse time series cases (50% and 30% of observed points), p-values obtained with t-tests, 0.0088 and 0.0041, are lower than significance level alpha. Therefore, our proposed ODE-GRU-D recurrent neural network achieved significantly better results than the earlier ODE-RNN model variant on these datasets.

Based on the analysis done in this section, we can conclude that our proposed ODE-based model implementation is useful on relatively sparse time series data cases. In less irregular investigated time series cases, the difference between the ODE-GRU-D model and the current ODE-RNN model results is not significant.

Conclusions

In this thesis, the new model called ODE-GRU-D for irregularly-sampled time series data is proposed. The idea of the proposed algorithm is similar to the state-of-the-art ODE-RNN, but instead of using standard Recurrent Neural Network as in the original model, we used Gated Recurrent Unit with trainable Decays (GRU-D) in the architecture. Such combination is not considered earlier in related works. What is more, the suggested algorithm was applied to the irregularly-sampled robot trajectories data (MuJoCo Hopper dataset). Based on the literature overview of the field, four different models are selected for the experiments: one classic RNN model - GRU, two specific algorithms for irregular time series - GRU-D and ODE-RNN, and the proposed one. With these selected models, two experiments were done. In the first experiment, before training, we randomly subsampled some percentage of time points to have four different irregularly-sampled time series data cases: 90%, 70%, 50%, and 30% of time points with the observations, and then we trained and tested all the implemented models on such datasets. All the models were trained with 300 epochs to select the optimal number of epochs. In the second one, training was done with optimal numbers of epochs found in the previous part. Similarly, in this part are four irregular time series data cases, but time points are randomly sampled from different parts of the timeline before every experiment. The models are trained ten times in each case (every time datasets differ, but with the same percent of the observations). The average test RMSEs and standard deviations are calculated and compared. Finally, means obtained in the second part were compared using appropriate t-tests to find if they differ significantly.

The main conclusions obtained in the thesis are:

1. ODE-based RNNs are a notably better choice for all the cases of irregularly-sampled time series data than other investigated RNNs (GRU and GRU-D).

2. The proposed ODE-GRU-D model starts to converge faster than ODE-RNN - it needs fewer epochs. This advantage was achieved because of the added decay mechanism.

3. In the data case with 90% of successful observations, the previous model variant ODE-RNN slightly outperforms the suggested ODE-GRU-D (average test RMSE 0.0366 against 0.0369). However, in the other three data cases (70%, 50%, 30%), the new algorithm showed better test RMSEs.

4. The proposed model is generally more stable than other considered RNNs variants - it shows notably lower standard deviations on different cases of irregularly-sampled time series.

5. Our proposed ODE-GRU-D recurrent neural network achieved significantly better results than the earlier ODE-RNN model variant on the two most sparse time series cases based on appropriate t-tests (p-values 0.0088 and 0.0041).

In future researches, the proposed in this master's thesis ODE-GRU-D model can be applied to different types of robot trajectories datasets that have more joints and dimensions, for example, Walker, Cheetah, or Humanoid from MuJoCo physics engine. Also, it would be valuable to consider longer trajectories with more time points. If the results on such datasets would also be satisfactory, it is even possible to apply the model to real-world robots trajectories tasks.

References

[1] Afshine Amidi and Shervine Amidi. Recurrent Neural Networks. `https://stanford.edu/~shervine/teaching/cs-230/`, 2018.

[2] Ni Bin, Chen Xiong, Zhang Liming1, and Xiao Wendong. Recurrent Neural Network for Robot Path Planning. *Parallel and Distributed Computing: Applications and Technologies*, 2004.

[3] Bastian Bischoff, Duy Nguyen-Tuong, Herke van Hoof, Andrew McHutchon, Carl E. Rasmussen, Alois Knoll, Jan Peters, and Marc P. Deisenroth. Policy Search For Learning Robot Control Using Sparse Data. *IEEE International Conference on Robotics and Automation (ICRA)*, 2014.

[4] Hajer Brahm, Boudour Ammar, and Adel M. ALimi. Intelligent path planning algorithm for autonomous robot based on Recurrent Neural Networks. *IEEE 2013 International Conference on Advanced Logistics and Transport (ICALT)*, 2013.

[5] Zhengping Che, Sanjay Purushotham, Kyunghyun Cho, David Sontag, and Yan Liu. Recurrent Neural Networks for Multivariate Time Series with Missing Values. *Scientific Reports*, 2018.

[6] Gang Chen. A Gentle Tutorial of Recurrent Neural Network with Error Backpropagation. *arXiv preprint arXiv:1610.02583v3*, 2018.

[7] Ricky T. Q. Chen. MuJoCo Hopper Physics dataset. `http://www.cs.toronto.edu/~rtqichen/datasets/HopperPhysics/training.pt.`, 2019.

[8] Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. Neural Ordinary Differential Equations. *arXiv preprint arXiv:1806.07366v5*, 2019.

[9] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2014.

[10] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. *NIPS 2014 Workshop on Deep Learning*, 2014.

[11] Josiah Coad, Zhiqian Qiao, and John M. Dolan. Safe Trajectory Planning Using Reinforcement Learning for Self Driving. *arXiv preprint arXiv:2011.04702v1*, 2020.

[12] Andreas Eckner. A Framework for the Analysis of Unevenly Spaced Time Series Data. `http://eckner.com/papers/unevenly_spaced_time_series_analysis.pdf`, 2014.

[13] David E.Rumelhart and James L. McClelland. *Learning Internal Representations by Error Propagation*, pages 318–362. 1986.

[14] Hiroki Furuta, Tatsuya Matsushima, Tadashi Kozuno, Sergey Levine Yutaka Matsuo, Ofir Nachum, and Shixiang Shane Gu. Policy Information Capacity: Information-Theoretic Measure for Task Complexity in Deep Reinforcement Learning. *Proceedings of the 38th International Conference on Machine Learning*, 2021.

[15] Alessandro Gasparetto, Paolo Boscariol, Albano Lanzutti, and Renato Vidoni. Trajectory Planning in Robotics. *Mathematics in Computer Science*, 2012.

[16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[17] Ramin Hasani, Mathias Lechner, Alexander Amini, Daniela Rus, and Radu Grosu. Liquid Time-constant Networks. *arXiv preprint arXiv:2006.04439v4*, 2020.

[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[19] Sven Hellbach, Julian P. Eggert, Edgar Korner, and Horst-Michael Gross. Time Series Analysis for Long Term Predictionof Human Movement Trajectories. *Advances in Neuro-Information Processing*, 2009.

[20] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 1997.

[21] Sean D. Holcomb, William K. Porter, Shaun V. Ault, Guifen Mao, and Jin Wang. Overview on DeepMind and Its AlphaGo Zero AI. *Proceedings of the 2018 International Conference on Big Data and Education*, 2018.

[22] Branislav Holländer. Paper Summary: Neural Ordinary Differential Equations. `https://towardsdatascience.com/paper-summary-neural-ordinary-differential-equations-37c4e52df128`, 2018.

[23] Jiang Hua, Liangcai Zeng, Gongfa Li, and Zhaojie Ju. Learning for a Robot: Deep Reinforcement Learning, Imitation Learning, Transfer Learning. *Sensors*, 2021.

[24] Zijie Huang, Yizhou Sun, and Wei Wang. Learning Continuous System Dynamics from Irregularly-Sampled Partial Observations. *arXiv preprint arXiv:2011.03880v1*, 2020.

[25] Ahmed Hussein, Mohamed Medhat Gaber, Eyad Elyan, and Chrisina Jayne. Imitation Learning: A Survey of Learning Methods. *Association for Computing Machinery (ACM) Computing Surveys*, 2018.

[26] Masaya Inoue, Takahiro Yamashita, and Takeshi Nishida. Robot Path Planning by LSTM Network Under Changing Environment. *Advances in Computer Communication and Computational Sciences*, 2018.

[27] Ian D. Jordan, Piotr Aleksander Sokół, and Il Memming Park. Gated Recurrent Units Viewed Through the Lens of Continuous Time Dynamica Systems. *Frontiers in Computational Neuroscience*, 2021.

[28] Abdul Rehman Khan, Ameer Tamoor Khan, Masood Salik Masood, and Sunila Bakhsh. An Optimally Configured HP-GRU Model Using Hyperband for the Control of Wall Following Robot. *International Journal of Robotics and Control Systems*, 2021.

[29] Suyong Kim, Weiqi Ji, Sili Deng, Yingbo Ma, and Christopher Rackauckas. Stiff Neural Ordinary Differential Equations. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 2021.

[30] Diederik P. Kingma and Jimmy Lei Ba. ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION. *arXiv preprint arXiv:1412.6980v9*, 2017.

[31] Jens Kober, J. Andrew Bagnell, and Jan Peters. The International Journal of Robotics Research. *Proceedings of the 2018 International Conference on Big Data and Education*, 2013.

[32] Philipp Kratzer, Marc Toussaint, and Jim Mainprice. Motion Prediction with Recurrent Neural Network Dynamical Models and Trajectory Optimization, 2019.

[33] Geesara Kulathunga. A Reinforcement Learning based Path Planning Approach in 3D Environment. *arXiv preprint arXiv:2105.10342v1*, 2021.

[34] Zhilu Lai, Charilaos Mylonas, Satish Nagarajaiah, and Eleni Chatzi. Structural identification with physics-informed neural ordinary differential equations. *Journal of Sound and Vibration*, 2021.

[35] Mathias Lechner and Ramin Hasani. Learning Long-Term Dependencies in Irregularly-Sampled Time Series. *arXiv preprint arXiv:2006.04418v4*, 2020.

[36] Erich L. Lehmann and Joseph P. Romano. *Testing Statistical Hypotheses.* Sprigner, third edition, 2005.

[37] Qianting Li and Yong Xu. VS-GRU: A Variable Sensitive Gated Recurrent Neural Network for Multivariate Time Series with Massive Missing Values. *MDPI Applied Sciences*, 2019.

[38] Yunzhu Li, Jiajun Wu, Jun-Yan Zhu, Joshua B. Tenenbaum, Antonio Torralba, and Russ Tedrake. Propagation Networks for Model-Based Control Under Partial Observation. *International Conference on Robotics and Automation (ICRA)*, 2019.

[39] Yuxuan Liang, Kun Ouyang, Hanshu Yan, Yiwei Wang, Zekun Tong, and Roger Zimmermann. Modeling Trajectories with Neural Ordinary Differential Equations. *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence (IJCAI-21)*, 2021.

[40] Zachary C. Lipton, John Berkow, and Charles Elkan. A Critical Review of Recurrent Neural Networks for Sequence Learning. *arXiv preprint arXiv:1506.00019v4*, 2015.

[41] Zoltán Lőrincz. A brief overview of Imitation Learning. `https://smartlabai.medium.com/a-brief-overview-of-imitation-learning-8a8a75c44a9c`.

[42] Ellips Masehian. The Role of Motion Planning in Robotics. *Applied Mechanics and Materials*, 2015.

[43] Leland McInnes, John Healy, Nathaniel Saul, and Lukas Grossberger. Umap: Uniform manifold approximation and projection. *The Journal of Open Source Software*, 2018.

[44] Erich Merrill, Stefan Lee, Li Fuxin, Thomas G. Dietterich, and Alan Fern. Deep Convolution for Irregularly Samples Temporal Point Clouds. *arXiv preprint arXiv:2105.00137v1*, 2021.

[45] Radouan Ait Mouha. Deep Learning for Robotics. *Journal of Data Analysis and Information Processing*, 2021.

[46] Michael C. Mozer, Denis Kazakov, and Robert V. Lindsey. Discrete-Event Continuous-Time Recurrent Nets. *arXiv preprint arXiv:1710.04110v1*, 2017.

[47] Ramya S. Nair and P. Supriya. Robotic Path Planning Using Recurrent Neural Networks. *2020 11th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, 2020.

[48] Danielle Navarro. *Learning Statistics with R - A tutorial for Psychology Students and other Beginners.* LibreTexts, 2020. `https://stats.libretexts.org/Bookshelves/Applied_Statistics/Book%3A_Learning_Statistics_with_R_-_A_tutorial_for_Psychology_Students_and_other_Beginners_(Navarro)`.

[49] Fiorato Nicola, Yasutaka Fujimoto, and Roberto Oboe. A LSTM Neural Network applied to Mobile Robots Path Planning. *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)*, 2018.

[50] Takayuki Osa, Joni Pajarinen, Gerhard Neumann, J. Andrew Bagnell, Pieter Abbeel, and Jan Peters. *An Algorithmic Perspective on Imitation Learning.* Now Foundations and Trends, 2018.

[51] Raj Patel, Meysar Zeinali, and Kalpdrum Passi. Deep Learning-based Robot Control using Recurrent Neural Networks (LSTM; GRU) and Adaptive Sliding Mode Control. *Proceedings of the 8th International Conference of Control Systems, and Robotics (CDSR'21)*, 2021.

[52] Lev Semenovich Pontryagin, Evgenij Frolovich Mishchenko, Vladimir Grigorevich Boltyanski, and Revaz Valerianovic Gamkrelidze. *The mathematical theory of optimal processes.* Gordon and Breach Science Publishers, 1986.

[53] Danilo Jimenez Rezende and Shakir Mohamed. Variational inference with normalizing flows. *arXiv preprint arXiv:1505.05770*, 2015.

[54] Yulia Rubanova, Ricky T. Q. Chen, and David Duvenaud. Latent ODEs for Irregularly-Sampled Time Series. *Advances in Neural Information Processing Systems*, 2019.

[55] Samuel Sanford Shapiro and Martin Bradbury Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 1965.

[56] Maxim Vladimirovich Shcherbakov, Adriaan Brebels, Nataliya Lvovna Shcherbakova, Anton Pavlovich Tyukov, Timur Alexandrovich Janovsky, and Valeriy Anatol'evich Kamaev. A Survey of Forecast Error Measures. *World Applied Sciences Journal 24*, 2013.

[57] Satya Narayan Shukla and Benjamin M. Marlin. Interpolation-Prediction Networks for Irregularly Sampled Time Series. *The International Conference on Learning Representations (ICLR)*, 2019.

[58] Ikaro Silva, George Moody, Daniel J Scott, Leo A Celi, and Roger G Mark. Predicting In-Hospital Mortality of ICU Patients The PhysioNet Computing in Cardiology Challenge. *Computing in cardiology*, 2012.

[59] George W. Snedecor and William G. Cochran Ames. *Statistical Methods.* Iowa State University Press, eighth edition, 1989.

[60] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* The MIT Press, second edition, 2015.

[61] Xianfeng Tang, Huaxiu Yao, Yiwei Sun, Charu Aggarwal, Prasenjit Mitra, and Suhang Wang. Joint Modeling of Local and Global Temporal Dynamics for Multivariate Time Series Forecasting with Missing Values. *AAAI Conference on Artificial Intelligence (AAAI-20)*, 2020.

[62] Yuval Tassa, Yotam Doron, Alistair Muldal, Tom Erez, Yazhe Li, Diego de Las Casas, David Budden, Abbas Abdolmaleki, Josh Merel, Andrew Lefrancq, Timothy Lillicrap, and Martin Riedmiller. DeepMind Control Suite. *arXiv preprint arXiv:1801.00690v1*, 2018.

[63] Martijn van Otterlo and Marco Wiering. *Adaptation, Learning, and Optimization. Reinforcement Learning and Markov Decision Processes.* Springer Berlin Heidelberg, 2012.

[64] Arun Venkatraman. *Training Strategies for Time Series: Learning for Prediction, Filtering, and Reinforcement Learning.* PhD thesis, Carnegie Mellon University, The Robotics Institute, 2017.

[65] Kyle R. Williams, Rachel Schlossman, Daniel Whitten, Joe Ingram, Srideep Musuvathy, Anirudh Patel, James Pagan, and Kyle A. Williams. Trajectory Planning with Deep Reinforcement Learning in High-Level Action Spaces. *arXiv preprint arXiv:2110.00044v1*, 2021.

[66] Shudong Yang, Xueying Yu, and Ying Zhou. LSTM and GRU Neural Network Performance Comparison Study: Taking Yelp Review Dataset as an Example. *2020 International Workshop on Electronic Communication and Artificial Intelligence (IWECAI)*, 2020.

[67] Jianya Yuan, Hongjian Wang, Changjian Lin, Dawei Liu, and Dan Yu. A Novel GRU-RNN Network Model for Dynamic Path Planning of Mobile Robot. *IEEE Access*, 2019.

[68] Xuan Zhao, Sakmongkon Chumkamon, Shuanda Duan, Juan Rojas, and Jia Pan. Collaborative Human-Robot Motion Generation using LSTM-RNN. *2018 IEEE-RAS 18th International Conference on Humanoid Robots (Humanoids)*, 2018.

[69] Raimondas Čiegis. *Diferencialinių lygčių skaitiniai sprendimo metodai.* Technika, 2003.