TÉCNICO LISBOA

Sparse Medical Time Series Modeling with Neural Differential Equations

Pedro Miguel Gonçalves Lopes

Thesis to obtain the Master of Science Degree in

Mathematics and Applications

Supervisors: Prof. Paulo Alexandre Carreira Mateus
Prof. Alexandra Sofia Martins de Carvalho

Examination Committee

Chairperson:
Supervisor: Prof. Paulo Alexandre Carreira Mateus
Member of the Committee:

December 2021

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

I would like to thank my parents who have always provided me with the conditions to make the most of my studies as well as for their continued encouragement. I give many thanks to my brother, for his immense patience while lending me his computer, as if I were to have only used my trash machine I would still be training these models.

I thank to Instituto Superior Técnico and particularly its department of mathematics, for the consistently high quality education they provide. Their effort to always instill the fundamentals with rigor while at the same time encouraging me to find which paths in mathematics I wished to explore and never letting down my curiosity, was definitely marking in the way I view both my professional life and my approach to knowledge in general. I also thank the University of Ghent, for the accommodation they provided me in my semester as an exchange student as part of the Erasmus program, often going above and beyond to allow me to make the most of the opportunity.

I would also like to give many thanks to all my colleagues, as without the joint sense of perseverance and light hearted moments during times of pressure, this journey would never have been as rewarding.

Abstract

In the field of time series modeling, the problem of irregularly sampled data proves hard to overcome. Neural ordinary differential equation models (neural ODEs) are a family of models that have proved to be a successful approach to learning sparse and irregularly measured time series, having obtained several state of the art results on data interpolation. Their ability to model data continuously in time give them the ability to make accurate prediction in any time point of the time series.

In this thesis, we review the concepts necessary to understand neural ODEs, ranging from numerical ODE solvers to various neural network architectures. We construct neural ODE models on top of multiple widely used neural network architectures, and test their performance on multiple tasks on sparse medical data.

This study aims at finding in what conditions does the increased performance of upgraded classical neural networks with neural ODEs out weights the additional computational cost associated.

The main contributions of this work are, the further testing of this family of models in order to further study its strengths and weaknesses, on multiple tasks and the testing of a new way to extrapolate data using recurrent neural networks with a neural ODE, without re-feeding predictions and relying solely on the dynamics of the neural ODE.

Keywords

Neural Differential Ordinary Equations, Variational Autoencoders, Recurrent Neural Networks, Sparse Time Series.

Resumo

Na área de aprendizagem de séries temporais, o problema de dados com medições irregulares é dificil de ultrapassar. Equações diferenciais ordinárias neuronais (Neural ODEs) provou ser uma abordagem fiável para aprender series temporais esparsas e com medições irregulares, tendo obtido vários resultados de estado da arte em interpolação de dados. A sua abilidade de modelar continuamente os dados no tempo permite obter previsões precisas em qualquer ponto do tempo na série temporal.

Nesta tese, revemos os conceitos necessários para compreender Neural ODEs, desde calculadores de ODEs a várias arquitecturas de redes neuronais. Construímos Neural ODEs usando várias arquitecturas de redes neuronais populares, cujas capacidades são testados em múltiplas tarefas sobre dados esparsos médicos.

O objectivo deste estudo é encontrar quais as condições em que a melhoria dos resultados obtidos pelas redes neuronais com Neural ODEs, relativamente às suas contrapartes clássicas, compensa o custo computacional que vem associado.

As contribuições desta tese são, os testes adicionais a esta família de modelos, de forma a testar os seus pontos fortes e fracos, em várias tarefas, assim como testar uma nova maneira de extrapolar dados usado redes neuronais recorrentes com neural ODEs, não devolvendo previsões ao modelo e usando apenas a dinâmica da neural ODE.

Palavras-chave

Equações diferenciais Ordinárias Neuronais, Auto-Codificadores Variacionais, Redes neuronais recurrentes, Series temporais esparsas.

Contents

1 Introduction **1**

1.1 Motivation . . . 3

1.2 Contributions . . . 3

1.3 Thesis Outline . . . 4

2 Background **5**

2.1 Ordinary Differential Equations . . . 7

2.2 Neural Networks . . . 9

2.2.1 Feedforward Neural Networks . . . 9

2.2.2 Backpropagation Algorithm . . . 11

2.2.3 Recurrent Neural Networks . . . 12

2.2.4 Variational Autoencoder . . . 14

2.2.4.A AutoEncoder models . . . 14

2.2.4.B Variational Inference . . . 14

2.2.4.C Variational Autoencoder Arquitecture . . . 15

2.2.4.D Variational Lower Bound and Training . . . 16

3 Neural Ordinary Differential Equations **19**

3.1 Residual Networks and Continuous Hidden State . . . 21

3.2 Training of Neural Ordinary Differential Equations . . . 21

4 Neural Ordinary Differential Networks for Time Series **25**

4.1 Sparse Data and Masking functionality . . . 27

4.2 Ordinary Differential Equation Recurrent Neural Network . . . 28

4.3 Latent Ordinary Differential Equations . . . 28

5 Problem Definition and Experimental Setup **31**

5.1 Data analysis . . . 33

5.2 Problem Specification . . . 34

5.3 Experimental Models . . . 38

5.4 Data Treatment . . . 40

6 Experimental Results **43**
6.1 Interpolation Task 46
6.2 Extrapolation Task 47
6.3 Classification Task 47
6.4 Discussion 48
7 Conclusion **51**
7.1 Contributions 53
7.2 Future Work 53
Bibliography **55**

List of Figures

2.1 Feedfordward neural network $\mathcal{F}((3,g),(5,g),(5,g),(1,g))$. 10
2.2 ReLU, sigmoid and hyperbolic tangent functions. 11
2.3 Diagram of a RNN. 13
2.4 Arquitecture of a VAE. 16

4.1 Latent ODE with ODE-RNN encoder, left hand side shows the encoding of the data backwards in time into z_0, while right hand side shows how to obtain the latent state in any point of interest by solving the IVP, and decode it back into the input space. Figure from Rubanova et al. [2019]. 30

5.1 Graphic representation of $U_{TP}(s,t)$ and $U_{FN}(s,t)$ on the relevant time window. Figure obtained from https://physionet.org/content/challenge-2019/1.0.0/. 36
5.2 Graphic representation of $U_{FP}(s,t)$ and $U_{TN}(s,t)$ on the relevant time window. Figure obtained from https://physionet.org/content/challenge-2019/1.0.0/. 37

List of Tables

6.1 Utility score and Likelihood estimation for different values of β. 45
6.2 Interpolation Test Mean Squared Error (MSE) $(\times 10^{-2})$. 46
6.3 Extrapolation Test Mean Squared Error (MSE) $(\times 10^{-2})$. 47
6.4 Utility Score, accuracy, precision and recall for classification task. 48

List of Algorithms

4.1 ODE-RNN 28

1

Introduction

Contents

1.1 Motivation 3

1.2 Contributions 3

1.3 Thesis Outline 4

This chapter provides the motivation behind this thesis, as well as its main contributions and the outline of the text.

1.1 Motivation

In the field of medical learning, time series carry immense importance. Some values necessary to diagnose and treat a patient appropriately are difficult to obtain, sometimes by requiring extensive laboratory work, sometimes because the process of acquiring them is invasive and dangerous, meaning one has minimal access to these values for each series. Therefore learning the dynamics of these specific features proves an exciting problem with very tangible benefits, as it allows both, better automatic diagnosis by inputting all missing values, and allows physicians to use approximated values when not deemed entirely essential to use the real ones, or when the real ones are to risky to obtain. Leading to a safer and less resource-intensive diagnosis for each patient. From this set of specific difficulties associated with medical data, it arises the need for models which can learn sparse data sequences with arbitrary lengths and irregular time stamps.

Specifically, we explore the neural ODE family of models, Chen et al. [2018], motivated by their elegant solutions to the constraints mentioned above. All neural ODE models use a neural network to approximate the dynamics of a relevant space in a neural ODE model, for example, the hidden space in a recurrent neural network, and use numerical methods to extract the hidden state in all relevant points. Adding this continuity to the modeled space allows making quality predictions for every timestamp and provides better ways of dealing with missing values.

We wish to further identify what are the strengths and weaknesses of these models when applied to medical data and continue to build empirical evidence to support, or dispute, the usage of these models in medical machine learning products.

1.2 Contributions

This work provides a comprehensive and self-contained introduction to sparse time series modeling (with neural ordinary differential equations), it highlights the most relevant constraints in this particular area of learning and some popular solutions used on the models trained for this thesis.

Novel contributions in this thesis are the following:

- We present medical data, sourced from ICU patients and explore the performance of two different neural ODE models, when compared to their non-ODE variants, at interpolation, extrapolation and classification tasks.

- We provide a way of extrapolating data using a neural ODE recurrent neural network without re-feeding extrapolated values back into the model as in a classical recurrent neural network, and instead relying solely on the dynamics. We test this approach on the previously mentioned data.
- We present a GitHub project containing the code so the experiments may be replicated, https://github.com/PedroMGLopes/Masters.

1.3 Thesis Outline

In the following chapter we provide an overview of the concepts relevant for this thesis. We review ordinary neural ordinary differential equations with a focus on numerical methods to solve them. We introduce neural networks and gradient based optimization, and lay out the architectures used thorough this work.

In chapter 3 we introduce neural differential equations, and explore the adjoint sensitivity method for parameter optimization.

In chapter 4 we lay out the architecture of the neural ODE models which will be used in this thesis, as well as explain what mechanisms must be introduced so these models can deal with sparse time series.

In chapter 5 we explore the data on which the tasks will be built upon, as well as define the objective of every task. We explicitly show every model which is to be tested, and the data modifications that must be made for every model to be able to do every single task.

In chapter 6 we investigate the results of every experiment, and try and draw conclusions from them.

Finally, a conclusion is made, highlighting the contributions made in this work and suggesting topics to be studied as a continuation of this thesis.

2

Background

Contents

2.1 Ordinary Differential Equations 7
2.2 Neural Networks 9

This chapter will provide the background knowledge for the following chapters. It will initially focus on a brief introduction to ordinary differential equations and the numeric methods one can utilize to solve them. We will continue by compiling some general knowledge on neural and how to train them.

2.1 Ordinary Differential Equations

An ordinary differential equation (ODE) of order n is an equation of the form

$$z^{(n)} = f(z^{(n-1)}, ..., z^{(2)}, z, t),$$

with z being a function of t and $z^{(i)} = \frac{d^i z}{dt^i}$. The function f is said to be the dynamics of the ODE.

An ODE is said to be linear if its dynamics is a linear combination of the derivatives of z.

If given an ODE and an initial condition $z_0 = z(t_0)$, the problem of finding a solution to the restricted equation is called an initial value problem (IVF). For $n = 1$ one can write the IVP as

$$\begin{aligned} &\frac{dz}{dt} = f(z, t) \text{ and} \\ &z(t_0) = z_0. \end{aligned}$$

We can now move to giving sufficient conditions on the dynamics of the ODE to ensure the existence and uniqueness of a solution. As in this thesis we are focused mostly on first order ODEs, the Picard–Lindelöf suffices.

Local Lipschitz condition: A function $f(z, t)$ satisfies the locally Lipschitz condition on z if each point has a neighborhood D_0 where a constant L exists such that

$$||f(z_1, t) - f(z_2, t)|| \leq L||z_1 - z_2||,$$

for any z_1, $z_2 \in D_0$.

Theorem 2.1.1. *(Picard–Lindelöf) If $f(z, t)$ is a continuous function in t satisfying the local Lipschitz condition in z then for the differential equation*

$$\frac{dz(t)}{dt} = f(z,t) \text{ and}$$
$$z(t_0) = z_0,$$

there exists a unique solution $z(t)$ *on an interval containing* t_0*.*

Numerical Methods for Ordinary Differential Equations

This section will provide a small introduction to the numeric methods which are more commonly used to solve ODEs. Even though they are not often explicitly mentioned in this thesis they stand at its very basis.

Consider the IVP

$$\frac{dz(t)}{dt} = f(z,t)$$
$$z(t_0) = z_0.$$

For a predetermined step h the Euler's method approximates the solution of the IVP at $t_k = t_0 + hk$ as

$$z_k = z_{k-1} + hf(t_{k-1}, z_{k-1}).$$

The local error of the Euler method for a single step is $\mathcal{O}(h^2)$. As for the global error, accumulated error made for multiple steps, it is $\mathcal{O}(h)$. One notable problem of the Euler's method is its numerical instability.

Runge Kutta Methods: This family of methods follows a form

$$z_{k+1} = z_k + h\sum_{i=1}^{s} b_i k_i,$$

with

$$
\begin{aligned}
k_1 &= f(z_k, t_k) \\
k_2 &= f(z_k + h(a_{2,1}k_1), t_k + c_2 h) \\
&\vdots \\
k_s &= f(z_k + h(a_{s,1}k_1 + \cdots + a_{s,s-1}k_{s-1}), t_k + c_s h).
\end{aligned}
$$

To specify a particular method, one needs to provide the integer s, and the coefficients a_{ij}, b_i and c_i.

These methods work very well for adaptive step control. To control the step size and the error, one uses two methods, one of convergence rate p and one of convergence rate $p-1$ differing only on the parameters b_i. Since the intermediate steps are the same running both methods comes at a negligible computational cost increase to running only one.

This way it is possible to estimate the single step error of the lower convergence rate method as

$$
e = ||z_k - z_k^*|| = h \left|\left| \sum_{i=1}^{s} (b_i - b_i^*) k_i \right|\right|,
$$

where z_k and z_k^* refer to the outputs of the methods of order p and $p-1$ respectively. If e is greater than a predetermined threshold the step is rejected and a new one is calculated with a smaller step, if e is lower than another separate threshold the step is increased as to save computational time. This ensures an almost optimal step size each iteration and removes the need from the user to find an appropriate step size.

The most used method in this thesis is a variant of the Dormand-Prince method Dormand and Prince [1980]. This is an adaptive Runge Kutta method of order five. This method has seven stages yet needs only six function evaluations as it makes use of ”first same as last” property, evaluating the last stage of each step at the same point as the first stage of the next step.

2.2 Neural Networks

Neural Networks are a class of functions, originally inspired by the circuits of the brain. The fundamental idea behind neural networks are layers of interconnected processing units, called neurons. In this section we will take a look at the most simple classes of neural networks and how their training algorithm works.

2.2.1 Feedforward Neural Networks

One can define the class of layers of a feedforward neural network as

$$\mathcal{L}(i, o, g) := \{f : \mathcal{R}^i \to \mathcal{R}^o \mid x \mapsto g(\mathbf{W}x + \boldsymbol{b}), \mathbf{W} \in \mathcal{R}^{o \times i}, \boldsymbol{b} \in \mathcal{R}^o\},$$

where g is a function, called activation function, **W** are called weights and b are the biases.

With the layers defined one can define the class of multi-layer feedforward neural network as

$$\mathcal{F}(\{s_k, g_k\}_{k=0}^l) := \{f : \mathcal{R}^{s_0} \to \mathcal{R}^{s_d} \mid f = f_{l-1} \circ \cdots \circ f_0, f_k \in \mathcal{L}(s_k, s_{k+1}, g_k)\},$$

where $i_k = o_{k-1}$ for $k > 0$.

Figure 2.1 shows the most common representation of the architecture of a feedforward neural network.

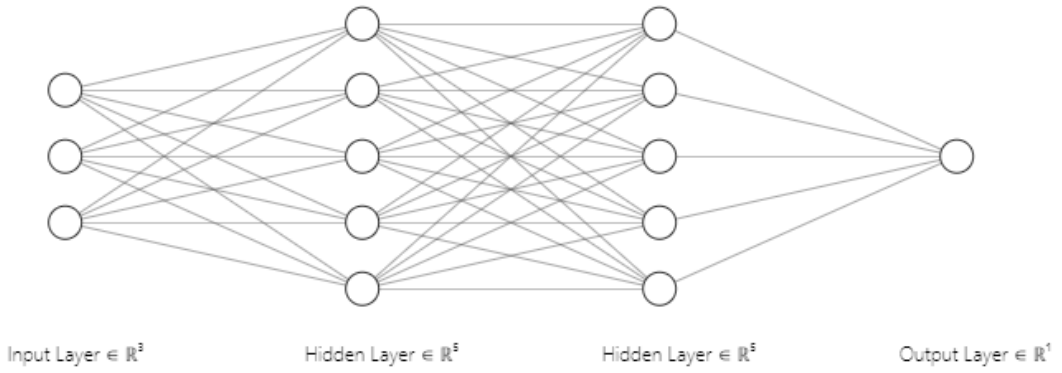

Figure 2.1: Feedfordward neural network $\mathcal{F}((3, g), (5, g), (5, g), (1, g))$.

From the definition one can see that any non-linear behaviour of the neural network must come from the activation function. The study of which activation functions are most adequate for each task is a subject on its own, therefore we will merely mention the most widely used ones. The rectified linear unit (ReLU)

$$g(\mathbf{x}) = \max(\mathbf{x}, \mathbf{0}),$$

is probably the most popular activation function used today. This function allows a much faster training than the more complicated alternatives making its use very attractive on deep neural networks.

Two other widely used activation functions are the sigmoid and the hyperbolic tangent. The latter differs a lot from the others as it can return negative values.

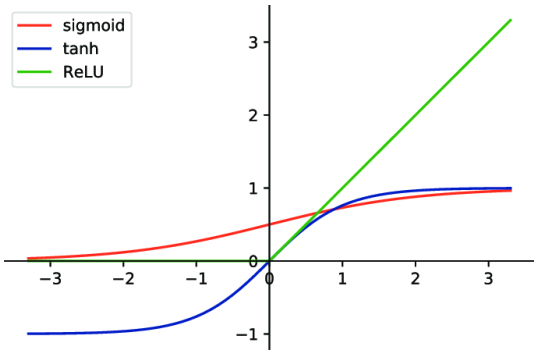

Figure 2.2: ReLU, sigmoid and hyperbolic tangent functions.

2.2.2 Backpropagation Algorithm

We can define feed forward neural networks as an arbitrary class of functions, yet their main purpose lies approximating a variety of functions. The main problem then becomes finding the which neural network better approximates our target function. From this point on we will call the set of weights and biases the model's parameters and its other components like depth, dimensions, etc, hyperparameters or architecture.

Unfortunately there exists no efficient way of finding the best architecture for a certain problem. Therefore we will focus on the task of, after deciding on an architecture, finding the parameters that work best for the task at hand.

The most widely used way of approximating these parameters consists on declaring a cost function C, and finding $\min_\theta C(y, f_\theta(x))$ this function on a set of points $\{(x, y)_k\}$ where f_θ is the neural network of predefined architecture with parameters θ.

As for approximating the minimum the backpropagation algorithm is by far the most widely used. The basics of continuous backpropagation were derived in the context of control theory by Kelley [1960] and and by Bryson [1961]. The use of this algorithm on the training of neural networks was popularized by Rumelhart et al. [1986].

The idea is to update each parameter by gradient descent with a certain learning rate η_k

$$\theta_{k+1} = \theta_k - \frac{\eta_k}{l} \sum_{i=1}^{l} \nabla_\theta C(y_i, f_\theta(x_i)).$$

The essence of backpropagation is how to efficiently compute $\nabla_\theta C(y_i, f_\theta(x_i))$.

Denote the value of layer l before activation on an input x by $\boldsymbol{a}^l = \boldsymbol{W}_l \boldsymbol{h}_{l-1} + \boldsymbol{b}_l$, and the partial derivative of C with regard $\boldsymbol{a}^l$ by $\boldsymbol{\delta}^l$, i.e,

$$\boldsymbol{\delta}^l = \frac{\partial C}{\partial \boldsymbol{a}^l}.$$

Writing the derivatives with regard to $\boldsymbol{a}^l$ allows us to compute the error in the layer l by using the

error on $l+1$ since

$$\boldsymbol{\delta}^l = \boldsymbol{W}_{l+1}^T \boldsymbol{\delta}^{l+1} \odot g^{'}(\boldsymbol{a}^l),$$

where the operator $\odot$ is the Hadamard product.

Finally we can calculate the gradient of the cost function with regard to the weights and biases on a layer l by

$$\frac{\partial C}{\partial \boldsymbol{b_l}} = \boldsymbol{\delta^l}$$

$$\frac{\partial C}{\partial \boldsymbol{W}_l} = \boldsymbol{\delta}^l (\boldsymbol{h}_{l-1})^T.$$

Updating the parameters only after reading the entire set is costly so more modern variations of this algorithm split the dataset into minibatches and update θ every time a minibatch is processed.

2.2.3 Recurrent Neural Networks

Multivariate time-series analysis has always been an important problem in statistics, and one that, much more than univariate time-series, presents a challenge for classical methods. For that reason neural networks have gained a lot of popularity in this area. Feedforward neural networks can learn sequence data by transforming a $R^n \times R^d$ sequence into a $R^{n \times d}$ input and running it normally, this process brings problems. The most obvious one is that it cannot work with sequences of varying size, as its input dimension is fixed. The second one is that even for fixed size sequences it requires a greater amount of parameters to learn the time dependency.

For that reason a special class of neural network, the recurrent neural network (RNN) were introduced. This class of neural networks reads a different time step of the input sequence on every layer and combines it with the output of the previous layer, one can write it as

$$h_{t+1} = g(W_h h_t + W_x x_{t+1} + b)$$

$$y_t^{'} = g_{out}(W_{out} h_t + b_{out}),$$

where W_h, W_x and W_{out} are the hidden, input and output weights, b and b_{out} the hidden and output biases. The last layer is a simple feedforward layer to map the hidden state into the desired output dimension. The most used activation function used is the hyperbolic tangent.

Figure 2.3 shows very simply how a RNN reads a sequence, it also becomes very apparent that by allowing outputs per step it solves many to many sequence problems very elegantly, while also allowing

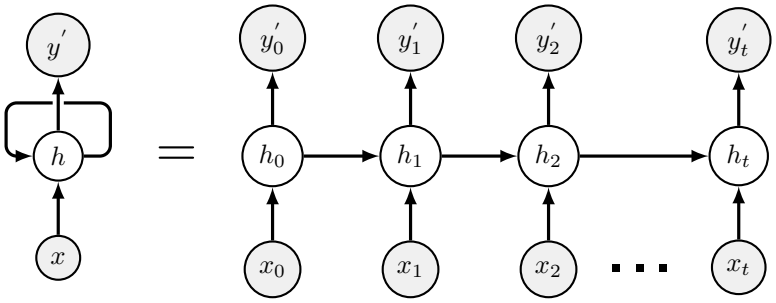

Figure 2.3: Diagram of a RNN.

it to be used for many to one problems by altering the cost function as to only consider the relevant output.

The backpropagation algorithm works very well on RNNs as for a sequence of size n it is very similar to a feedforward network of depth $n + 1$ with the constraint of the parameters being the same for every layer.

Based on the standard RNN there are quite a few variants of which the most widely used are the Long-short term memory neural network(LSTM), which uses two different channels to pass information down the chain of recursion, and the Gated recursive unit(GRU), which will be used in this thesis. When it comes to GRU there are also different variants, yet what unifies all of them is the existance of a mechanism called ”forget gate”. This mechanism can be seen as a more complex way of connecting the previous hidden state and the new input in order to separate the tasks of ”updating” and ”forgetting” previous information. For this thesis the following GRU type will be used

$$
\begin{aligned}
r_{t+1} &= \sigma(W_{ir}x_{t+1} + b_{ir} + W_{hr}h_t + b_{hr}) \\
z_{t+1} &= \sigma(W_{iz}x_{t+1} + b_{iz} + W_{hz}h_t + b_{hz}) \\
n_{t+1} &= \tanh(W_{in}x_{t+1} + b_{in} + r \odot (W_{hn}h_t + b_{hn})) \\
h_{t+1} &= (1 - z_{t+1}) \odot n + z_{t+1} \odot h_t,
\end{aligned}
$$

in this variant r, z and n are the reset, update, and new gates, respectively.

For the rest of this thesis when using the term RNN we will be referencing then GRU variant stated above, as it will be used many times for different tasks, and further modifications made on RNN models can be done on most variants.

The RNN belongs to the class of autoregressive models. Autoregressive models make a one-step-ahead prediction conditioned on the history of observations, i.e. they factor the joint density $p(x) = \prod_i p_\theta(x_i|x_{i-1}, \ldots, x_0)$.

2.2.4 Variational Autoencoder

Autoregressive models such as RNNs are easy to train and allow fast online predictions. However, autoregressive models can be hard to interpret, since their update function combines both their model of system dynamics, and conditioning on new observations. Furthermore, their hidden state does not explicitly encode uncertainty about the state of the true system. In terms of predictive accuracy, autoregressive models are often sufficient for densely sampled data, but perform worse when observations are sparse. An alternative to autoregressive models are latent-variable models. These models map the input into a n-dimensional latent space, and work from there.

We introduce the variational autoencoder Kingma and Welling [2014], Rezende et al. [2014], as one of these models.

2.2.4.A AutoEncoder models

Autoencoders (AE) Rumelhart et al. [1986], Bourlard and Kamp [1988] are neural networks which aim to reconstruct its own input. They are composed by an encoder and a decoder, both of which are usually neural networks. The encoder maps its input into a latent space, this image is usually called the code, the decoder then maps the code from the latent space back into the input space.

The autoencoders training procedure consists of finding the set of parameters (θ, ϕ), with respect to the decoder and encoder models, which minimize a loss function measuring the error of the approximation.

Autoencoders have the interesting property of being able to learn compressed representations of the data. By mapping high dimensional data into a lower dimensional latent space autoenconders learn a lower representation of the data holding the greatest amount of information as possible.

2.2.4.B Variational Inference

Consider the latent variable model of parameter θ with the following factorization:

$$p_\theta(x, z) = p_\theta(z)p_\theta(x|z),$$

where x are the observed variables and z the latent ones. We will call $p_\theta(z)$ the prior of z as it is not conditioned on any observations. The inference problem refers to the computation of the posterior distribution $p_\theta(z|x)$, which can be written as

$$p_\theta(z|x) = \frac{p_\theta(x, z)}{p_\theta(x)},$$

where $p_\theta(x)$ is the marginal distribution over the observed variables

$$p_\theta(x) = \int_z p_\theta(x, z)dz.$$

The main difficulty is that the integral is intractable by analytical methods and does not have a closed form solution. In order to solve this problem we use variational inference and turn it into an optimization one.

Variational inference aims to find a variational approximation $q_\phi(z|x)$ of the intractable true posterior distribution $p_\theta(z|x)$ by minimizing the Kullback-Leibler (KL) divergence between both. The KL-divergence, despite not being a distance, measures how different two probability distributions are, thus minimizing it allows us to approximate $p_\theta(z|x)$. If q and p are two continuous probability distribution, the KL-divergence is given by:

$$D(q||p) = \int_z q(z)\log\frac{q(z)}{p(z)}dz.$$

The optimization consists in find the parameters ϕ which minimize the KL-divergence.

2.2.4.C Variational Autoencoder Arquitecture

Autoencoders were traditionally used for dimensional reduction and representation learning. The introduction variational inference into this model resulted in the variational autoencoder, which is mainly used as a generative model.

In the context of an autoencoder, we can view the unobserved latent variables z as the code of the autoencoder. The recognition model $q_\phi(z|x)$ can then be seen as the probabilistic encoder, which given a data point x produces a distribution over the possible values of the code z from which x could have been generated.

Since the true posterior $p_\theta(z|x)$ is intractable, we use variational inference. The enconder then becomes a variational approximation $q_\phi(z|x)$ of the true posterior. The prior distribution over the latent variables, $p_\theta(z)$, is often a standard isotropic normal distribution. In that case, the encoder is designed to output the mean, μ_z, and the standard deviation, σ_z, of the approximate posterior distribution.

The decoder is designed to act as a generative model, given a code z it produces a distribution, $p_\theta(x|z)$, over the possible corresponding values of x.

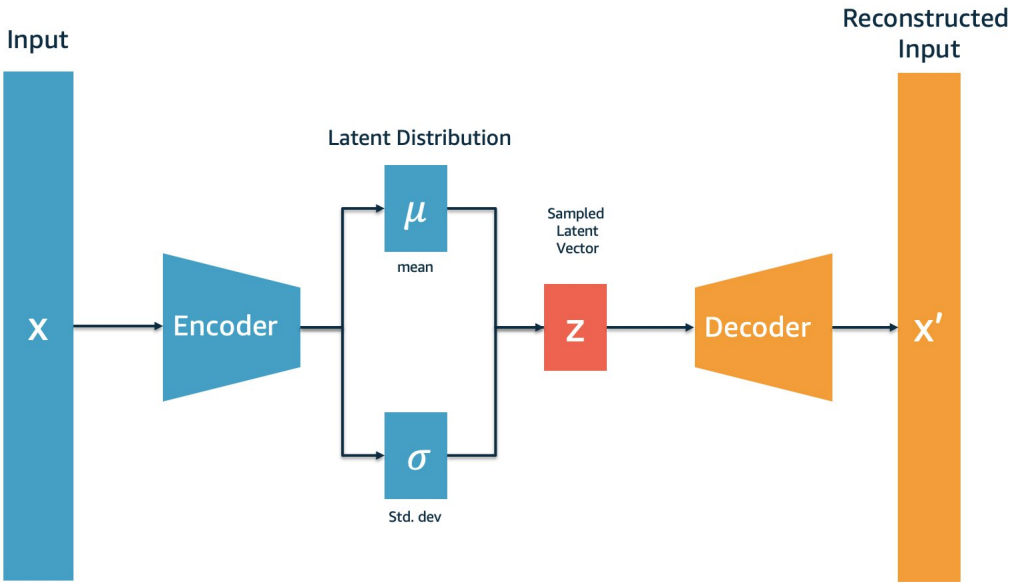

Figure 2.4: Arquitecture of a VAE.

A simplified overall architecture of these models can be seen in Figure 2.4. Both the encoder and decoder networks have their own architecture, which can be any of the many types of neural networks used for various tasks. For example, in this thesis, one VAE will utilize RNNs as both encoder and decoder.

2.2.4.D Variational Lower Bound and Training

Variational autoenconders use a very specific loss function. Given we want to maximise the log-likelihood of the data, $\log p_\theta(x)$, we can write

$$\log p_\theta(x) = \log\left(\int_z p_\theta(x,z)dz\right) = \log\left(\int_z p_\theta(x|z)\frac{p(z)}{q(z)}q(z)dz\right),$$

for any distribution q over z. Since the logarithmic function is concave and the integral can be written as an expectation, we can use Jensen's inequality to swap the logarithm and the expectation

$$\begin{aligned}\log p_\theta(x) &= \log\left(\mathbb{E}_{q(z)}\left[p_\theta(x|z)\frac{p(z)}{q(z)}\right]\right)\\ &\geq \mathbb{E}_{q(z)}\Big[\log p_\theta(x|z) + \log p(z) - \log q(z)\Big]\\ &= \mathbb{E}_{q(z)}\Big[\log p_\theta(x|z)\Big] - D(q(z)||p(z)).\end{aligned}$$

Since this holds for any q we can chose the tractable $q_\phi(z|x)$ and write

$$\log p_\theta(x) \geq \mathbb{E}_{q_\phi(z|x)}\left[\log p_\theta(x|z)\right] - D(q_\phi(z|x)||p(z)) := L(\theta, \phi, x).$$

The loss function for variational autoencoders is then $-L(\theta, \phi, x)$.

For training we need a final parametrisation trick. In order to train the network by backpropagation, we need our model to be differentiable w.r.t. its learned parameters. For this, the model needs to be deterministic, such that only the inputs are stochastic. Clearly this is not the case if we sample z based on a probability distribution with parameters that come from inside the model. In order to solve said problem we define auxiliary random variables ϵ following a standard normal distribution, which we treat as input. We can now see $z = \mu + \sigma \odot \epsilon$ and treat the autoencoder as deterministic.

3

Neural Ordinary Differential Equations

Contents

3.1 Residual Networks and Continuous Hidden State 21

3.2 Training of Neural Ordinary Differential Equations 21

In this chapter we introduce neural ordinary differential equations, we compare the backpropagation algorithm with the adjoint sensitivity method for training and examine how these models can be used together with other types of neural networks.

3.1 Residual Networks and Continuous Hidden State

As neural networks became more and more popular on image classification tasks, mostly due to the introduction of the convolutional neural network, the problem of how to use depth to achieve better results became an even more important area of investigation. Yet, He et al. [2015] showed empirically that, for classical convolutional neural networks, greater depth sometimes leads to poorer results. One of the major reasons to this phenomenon is the vanishing/exploding gradient problem. To alleviate this effect on training residual neural networks were introduced. Very simply residual networks consist of stacked residual units with "skip connection" between them, one can write them as

$$h_{t+1} = h_t + f(h_t, \theta_t),$$

where f is neural network.

These iterative updates can be seen as an Euler discretization of a continuous transformation Lu et al. [2017], Haber and Ruthotto [2017], Ruthotto and Haber [2018].

What happens as we add more layers and take smaller steps? In the limit, we parameterize the continuous dynamics of hidden units using an ordinary differential equation (ODE) specified by a neural network:

$$\frac{dh(t)}{dt} = f(h_l, \theta_l, t).$$

We can now define the output layer $h(t)$ as an IVP with $h_0 = h(0)$ and use an ODE solver to arrive at the solution. By pushing a lot of the calculations onto the ODE solver, and only evaluating the hidden state when the ODE solver needs it, one can make use of the capabilities of modern ODE solvers, such as adapting the evaluating strategy on the fly to achieve the specified level of numerical accuracy, and of course monitor the numerical error, as to avoid the vanishing/exploding gradient problem which plague very deep models.

3.2 Training of Neural Ordinary Differential Equations

Again training means approximating the set θ which minimizes a certain function,which we will call it loss. This can still be done by backpropagation through the discretized version of the ODE, which would

mean backpropagating through the ODE solver, however Chen et al. [2018] proposed an alternative method for training, the adjoint sensitivity method Pontryagin [1962].

The adjoint sensitivity method

The main technical difficulty in training continuous-depth networks is performing the backpropagation through the ODE solver. Even though backpropagating through the ODE solver steps is straightforward, the memory cost is high and additional numerical errors occur. Chen et al. [2018] presents the adjoint sensitivity method as a solution which can be used regardless the choice of ODE solver and with constant memory consumption. This section gives an overview of how the adjoint sensitivity method works.

Consider optimizing a loss function L, whose input is the result of an ODE solver:

$$L(z(t_1)) = L\left(z(t_0) + \int_{t_0}^{t_1} f(z(t), \theta, t)dt\right) = L(\text{ODESolve}(f(z(t_0), t_0, t_1, \theta))).$$

To optimize L, we require gradients with respect to θ. The first step towards this end is computing the adjoint state

$$a(t) = \frac{dL}{dz(t)},$$

whose dynamics are given by the differential equation,

$$\frac{da(t)}{dt} = -a(t)\frac{\partial f(z(t), t, \theta)}{\partial z}.$$

We can then solve the differential equation backwards in time, in a similar way to the backpropagation algorithm

$$a(t_0) = a(t_n) + \int_{t_n}^{t_0} \frac{da(t)}{dt}dt = a(t_n) - \int_{t_n}^{t_0} a(t)^T \frac{\partial f(z(t), t, \theta)}{\partial z(t)}dt.$$

Here we assumed that loss function L depends only on the last time point t_n. If function L depends also on intermediate time points $t_1, t_2, ..., t_{n-1}$, etc., we can repeat the adjoint step for each of the intervals $[t_{n-1}, t_n]$, $[t_{n-2}, t_{n-1}]$ in the backward order and sum up the obtained gradients.

We also need the gradients with respect to t and θ. To do this, we view t and θ as states and write

$$\frac{dt(t)}{dt} = 1, \qquad \frac{d\theta(t)}{dt} = 0,$$

combining these with z forms an augmented state

$$\frac{d}{dt}\begin{bmatrix} z \\ \theta \\ t \end{bmatrix} = f_{\text{aug}}([z, \theta, t]) := \begin{bmatrix} f(z(t), \theta, t) \\ 0 \\ 1 \end{bmatrix},$$

with corresponding adjoint state,

$$a_{\text{aug}} = \begin{bmatrix} a \\ a_\theta \\ a_t \end{bmatrix}, \quad a_\theta(t) = \frac{dL}{d\theta(t)}, \quad a_t(t) = \frac{dL}{dt(t)}.$$

We then obtain the ODE

$$\frac{da_{\text{aug}}(t)}{dt} = -\begin{bmatrix} a(t) & a_\theta(t) & a_t \end{bmatrix} \frac{\partial f_{\text{aug}}(t)}{\partial [y, \theta, t]},$$

where $\frac{\partial f_{\text{aug}}(t)}{\partial [y,\theta,t]}$ is the Jacobian of the augmented state

$$\frac{\partial f_{\text{aug}}(t)}{\partial [y, \theta, t]} = \begin{bmatrix} \frac{\partial f}{\partial z} & \frac{\partial f}{\partial \theta} & \frac{\partial f}{\partial t} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}(t).$$

The second element can be used to obtain the total gradient with respect to the parameters, by integrating over the full interval and setting $a_\theta(t_n) = 0$

$$\frac{dL}{d\theta} = a_\theta(t_0) = -\int_{t_n}^{t_0} a(t) \frac{\partial f(z(t), t, \theta)}{\partial \theta} dt.$$

Finally, we can also get gradients with respect to t_0 and t_n, the start and end of the integration interval.

$$\frac{dL}{dt_n} = a(t_n) f(z(t_n), t_n, \theta), \quad \frac{dL}{dt_0} = a(t_n) - \int_{t_n}^{t_0} a(t) \frac{\partial f(z(t), t, \theta)}{\partial t} dt.$$

For training the neural ODE the derivatives with respect to t_0 and t_n are not needed so we skip them. The needed vector-Jacobian products can all be efficiently evaluated by automatic differentiation, at a time cost similar to that of evaluating f. All integrals for solving z, a and $\frac{dL}{d\theta}$ can be computed in a single call to an ODE solver, which concatenates the original state, the adjoint, and the other partial derivatives into a single vector.

Most ODE solvers have the option to output the state $z(t)$ at multiple times. When the loss depends on these intermediate states, the reverse-mode derivative must be broken into a sequence of separate solves, one between each consecutive pair of output times. At each observation, the adjoint must be adjusted in the direction of the corresponding partial derivative $\frac{dL}{dz(t_i)}$.

4

Neural Ordinary Differential Networks for Time Series

Contents

4.1 Sparse Data and Masking functionality 27

4.2 Ordinary Differential Equation Recurrent Neural Network 28

4.3 Latent Ordinary Differential Equations 28

Recurrent neural networks and its variants are the most prominent model class for high-dimensional, regularly-sampled time series data, such as text or speech. However, they are an awkward fit for irregularly-sampled time series data, many times sparse, common in medical or business settings. A standard trick for applying RNNs to irregular time series is to divide the timeline into equally-sized intervals, and impute or aggregate observations using averages. To deal with the missing values it is standard to use masks.

Neural ODE's offer a way to generalize state transitions in RNNs to continuous time dynamics, naturally allowing time gaps between observations, and remove the need to group observations into equally-timed bins.

One can also apply Neural ODE's on a variational autoencoder framework. The model represents each time series by a latent trajectory. Each trajectory is determined from a local initial state, z_{t_0}, obtained by encoding the input sequence, and a global set of latent dynamics shared across all time series.

4.1 Sparse Data and Masking functionality

A very common problem with multivariate data is sparsity, meaning for every data point only a subset of the input space actually has values and the rest is missing. Obviously neural networks can not receive these missing values, therefore the problem on how to deal with them is very pertinent. One method that immediately comes to mind is to input said missing data using a good predictor, maybe a generative model trained on the data or on the worst case scenario just the average of said variable in the dataset. All the imputation approaches carry their amount of issues of course. Choose an impute method to simple and you might end up with inaccurate predictions sometimes by not considering interdependency between variables, yet by trying to learn as much of the data behaviour as possible to correctly input it, and at the end of the day one would be training two models equally as heavy, the one solving the original task and the imputator. And as sparsity increases it carries the disadvantage of the model for the task originally at hand being trained on data mostly generated, pilling errors one on top of another.

One way to limit the problem of training on imputed data when it comes to RNNs is by using masking. Given one element from the dataset $\{\boldsymbol{x}_t\}_{t=0...T}$ one creates a mask $\{\boldsymbol{m}_t\}_{t=0...T}$ such that

$$m_{t,f} = \begin{cases} 1 \; if \; x_{t,f} \; not \; missing \\ 0 \; otherwise \end{cases},$$

and imputes the missing values on the original dataset with zeros.

The neural network can now read the input, should the mask be entirely zeros on a certain time point the RNN can skip updating its hidden state. This is particularly important if the data is also measured on irregular time frames meaning that bucketing the data end up with complete missing time steps. If the

output space is also very sparse masking the output space allows skipping using the backpropagation algorithm on the imputed targets. This becomes particularly useful when it comes to autoencoders where both the input and the output masks are the same.

4.2 Ordinary Differential Equation Recurrent Neural Network

The main objective of introducing neural ODE machinery into the recurrent neural network family is to model its hidden state between observations as a continuous state evolving over time instead of a fix state until a new observation is made. For that we need to introduce a neural network f_θ which will model these dynamics. Following this one can define a pre-input hidden state at every point $h'_i = \text{ODESolve}(f_\theta, h_{t-1}, (t_{i-1}, t_i))$.

Algorithm 4.1: ODE-RNN

Data: $\{x_i, t_i\}_{i=0\ldots N}$
$h_0 = \mathbf{0}$
for *i in 0...N* **do**
  $h'_i = \text{ODESolve}(f_\theta, h_{i-1}, (t_{i-1}, t_i))$
  $h_i = \text{RNNCell}(h'_i, x_i)$
$o_i = \text{OutputNN}(h_i)$ for all i in 0...N
Result: $\{o_i\}_{i=0\ldots N}$

One can see that ODE-RNNs consist of 2 main networks, and that as the sparsity of the data increases the dynamics network would be the one with greater impact on the overall model, since there would be less updates to the state because of the time steps of only missing features. Also, as argued by Chollet [2017], given that no feature which is often missing can naturally be zero, "the network will learn from exposure to the data that the value 0 means missing data and will start ignoring the value", in this case the only information left is the hidden state provided by the dynamics.

4.3 Latent Ordinary Differential Equations

The latent variable models coupled with neural ODE machinery were proposed in Chen et al. [2018]. Using the variational autoencoder framework for both training and prediction, requires estimating the approximate posterior $q_\phi(z_0|\{x_i, t_i\}_{i=0}^n)$. Inference and prediction in this model is effectively sequence-to-sequence architecture, in which a variable-length sequence is encoded into a fixed-dimensional embedding, which is then decoded into another variable-length sequence, as in Sutskever et al. [2014].

The initial approach was to use an RNN as encoder yet the ODE-RNN was shown to obtain better results in Rubanova et al. [2019]. To get the approximate posterior at time point t_0, we run the ODE-RNN encoder backwards-in-time from t_N to t_0 in order to sample the initial state z_0 for the IVF.

The decoder is where the main neural ODE resides. After sampling the initial state z_0 and observation times $t_0, \ldots, t_n$, an ODE solver is used to produce $z_1, \ldots, z_n$ which describe the latent state at each observation time, one can then decode every point in the latent space. The entire process is:

$$\begin{aligned} z_0 &\sim q_\phi(z_0|\{x_i, t_i\}_{i=0}^n) \\ z_0, \ldots, z_n &= \text{ODESolve}(f_\theta, z_0, \{t_0, \ldots, t_n\}) \\ x_i &\overset{indep.}{\sim} p_\theta(x_i|z_i), \end{aligned}$$

where f_θ is the neural network parameterizing the dynamics of the latent space and p_θ is a neural network which maps the hidden state at any point back into the input space.

Decoder and encoder are trained simultaneously by maximizing the variational lower bound (ELBO)

$$L(\theta, \phi) = \sum_{i=0}^{n} p_\theta(x_i|z_i) + \log\, p(z_0) - \log\, q_\phi(z_0|\{x_i, t_i\}_{i=0}^n), \tag{4.1}$$

where $p(z_0)$ is a standard normal.

This latent variable framework comes with several benefits: First, it explicitly decouples the dynamics of the system (ODE), the likelihood of observations, and the recognition model, allowing each to be examined or specified on its own. Second, the posterior distribution over latent states provides an explicit measure of uncertainty, which is not available in standard RNNs and ODE-RNNs. Finally, it becomes easier to answer non-standard queries, such as making predictions backwards in time, or conditioning on a subset of observations.

When compared with a VAE with RNN encoder and decoder it becomes apparent that the main function of the neural ODE in this approach is to substitute the RNN decoder. The ODE-RNN encoder latent ODE is also very parameter heavy, as one has 2 neural networks running for encoding, and specifying the dynamics of the hidden state of the encoder, another neural network to serve as the dynamics of the latent space and finally another one to project the latent space back into the input space. Together with its heaviness also comes some advantages, as seen in previous chapters the main challenge of variational learning is to approximate the posterior, and an ODE-RNN is more well equipped at dealing with sparse data leading to a better approximation. Likewise by learning the dynamics of the latent space instead of simply decoding with an RNN, the decoder becomes much more robust, obtaining much better results when it comes to extrapolation in time for example, a task traditional autoencoders are obviously not well equipped to do.

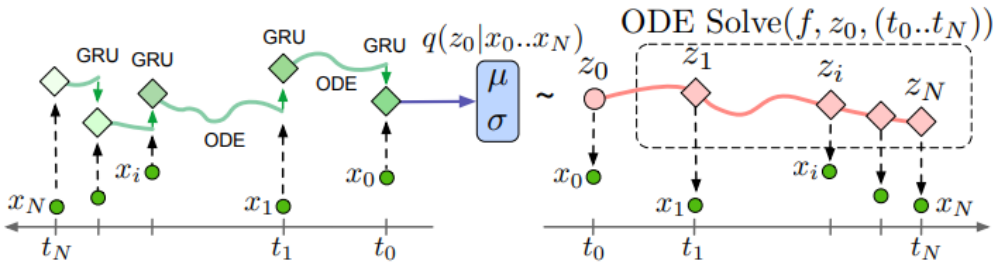

Figure 4.1: Latent ODE with ODE-RNN encoder, left hand side shows the encoding of the data backwards in time into z_0, while right hand side shows how to obtain the latent state in any point of interest by solving the IVP, and decode it back into the input space. Figure from Rubanova et al. [2019].

Figure 4.1 shows the entire process of computing the latent state in any point in time, and while it is not explicit the sequence of time points to decode can be entirely different than the ones from the input, as long as it is not in the training phase.

5

Problem Definition and Experimental Setup

Contents

5.1 Data analysis 33
5.2 Problem Specification 34
5.3 Experimental Models 38
5.4 Data Treatment 40

This chapter will be divided in four sections. In the first we will define the characteristics we are searching for in a dataset to showcase the properties of the models defined in the sections above, we will introduce a dataset with said properties and do a small exploratory analysis. On the following one, we will specify which are the problems on the dataset we will be experimenting on, how performance will be measured in each of them and why these measures showcase the strengths and weaknesses of the models. We will then define what models will be trained, how exactly they will be used and the loss function to be used to achieve the intend results as well as initial challenges for each model. Lastly we will focus on the treatment of the data.

5.1 Data analysis

Considering the main objective of this thesis is to empirically measure the benefit of introducing neural ODE machinery into widely used neural networks, the data used to measure it should cater to the theoretical strengths one expects from the modified models. When it comes to time series, these strengths would be sparsity of the data and irregularity of the measurements.

This was precisely the case on the PhysioNet challenge data used in Rubanova et al. [2019], and as this thesis is built mainly on the conclusions extracted from that dataset, the choice for data in this thesis should be made to complement these finds and broaden them.

One exciting conclusion on Rubanova et al. [2019] is that for binary classification for mortality, the modified models did not present significant benefits when compared to their classical variants. This lack of improvement in one of the most widely studied machine learning tasks in the medical field encouraged the further study of this field, and it was decided that the data should present a binary classification problem.

After compiling all the constraints, “PhysioNet Computing in Cardiology Challenge 2019” dataset seemed very close to ideal. It shared the medical setting of the data used in Rubanova et al. [2019], and despite not having irregular measurements, its data was very sparse. When it comes to binary classification, the challenge provided a unique classification problem.

Feature overview

The dataset is a compilation of demographics, vital signs, and laboratory values corresponding to ICU stays in two separate hospitals. The purpose of the data challenge was to train models capable of early prediction of sepsis Singer et al. [2016], for this purpose each time series has a set of time points associated to help position the occurrence of sepsis on the correct hour of the patient record.

The relevant time points are:

$t_{suspicion}$

- Clinical suspicion of infection identified as the earlier timestamp of IV antibiotics and blood cultures within a specified duration.
- If antibiotics were given first, then the cultures must have been obtained within 24 hours. If cultures were obtained first, then antibiotic must have been subsequently ordered within 72 hours.
- Antibiotics must have been administered for at least 72 consecutive hours to be considered.

t_{SOFA}

- The occurrence of end organ damage as identified by a two-point deterioration in Sequential Organ Failure Assessment (SOFA) score within a 24-hour period.

t_{sepsis}

- The onset time of sepsis is the earlier of $t_{suspicion}$ and t_{SOFA} as long as t_{SOFA} occurs no more than 24 hours before or 12 hours after $t_{suspicion}$; otherwise, the patient is not marked as a sepsis patient. Specifically, if $t_{suspicion} - 24 \leq t_{SOFA} \leq t_{suspicion} + 12$, then $t_{sepsis} = \min(t_{suspicion}, t_{SOFA})$.

There are a total of 40 features of which 8 are vital signs, 26 are laboratory results and 6 are demographic values. There is also a label for each time point SepsisLabel.

SepsisLabel

- For sepsis patients, SepsisLabel is 1 if $t \geq t_{sepsis} - 6$ and 0 if $t < t_{sepsis} - 6$. For non-sepsis patients, SepsisLabel is 0.

For a more in detail analysis of the data there is Reyna et al. [2020].

5.2 Problem Specification

There is a particular interest in the ability of the models to both interpolate and extrapolate in the input space. These experiments are intended to try and obtain further empirical evidence that in these tasks the ODE machinery works particularly well, reinforcing the results obtained in Rubanova et al. [2019].

The interpolation and extrapolation problems can be defined as:

Interpolation experiment of probability p: Given a record $\{\boldsymbol{x}_t\}_{t=t_0,\ldots,t_n}$, create a random subsampled record $\{\boldsymbol{x'}_t\}_{t=t_0,\ldots,t_n}$, by cloning the original and for every non-missing value substitute it with missing with probability $1-p$, while keeping a memory set m of all indexes (t, f) which were removed.

Run $\{\boldsymbol{x'}_t\}_{t=t_0,...,t_n}$ through a model M with an output space equal to input, on the set of time points of the original sequence, in order to create a predicted record $M(\{\boldsymbol{x'}_t\}_{t=t_0,...,t_n}, \{t_0, ..., t_n\})$. To measure the quality of interpolation calculate $\frac{1}{\#m}\sum_{(t,j)\in m}(x_{(t,j)} - M(\{\boldsymbol{x'}_t\}_{t=t_0,...,t_n}, \{t_0, ..., t_n\})_{(t,j)})^2$ which is the mean squared error on the subset of removed points.

Extrapolation experiment: Given a record $\{\boldsymbol{x}_t\}_{t=t_0,...,t_n}$, create the subsequence $\{\boldsymbol{x}_t\}_{t=t_0,...,t_k}$ containing every time point smaller than or equal to $\lfloor t_n/2 \rfloor$. Run the subsequence through a model M with an output space equal to input, on the set of time points of the original sequence greater than $\lfloor t_n/2 \rfloor$, in order to create a predicted record $M(\{\boldsymbol{x}_t\}_{t=t_0,...,t_k}, \{t_{k+}), ..., t_n\})$. To measure the quality of extrapolation calculate $\text{MSE}_{masked}(\{\boldsymbol{x}_t\}_{t=t_{k+1},...,t_n}, M(\{\boldsymbol{x}_t\}_{t=t_0,...,t_k}, \{t_{k+1}, ..., t_n\}))$, in which MSE_{masked} is the mean squared error between inputs on the greatest subset of points in which neither input has missing values.

To obtain the values for extrapolation and interpolation on the whole dataset, average each measurement over all records on the dataset.

The final problem to consider will is the early prediction of sepsis as specified on the physionet challenge. Essentially it is a cost sensitive classification problem per time point, meaning for every record $\{\boldsymbol{x}_t\}_{t=t_0,...,t_n}$ we wish to create a binary sequence $\{y_t\}_{t=t_0,...,t_n}$ which maximizes the following cost function:

$$U(y, s, t) = \begin{cases} U_{TP}(s,t) & \text{if positive prediction in } y \text{ at time } t \text{ for sepsis patient } s, \\ U_{FN}(s,t) & \text{if negative prediction in } y \text{ at time } t \text{ for sepsis patient } s, \\ U_{FP}(s,t) & \text{if positive prediction in } y \text{ at time } t \text{ for non-sepsis patient } s, \\ U_{TN}(s,t) & \text{if negative prediction in } y \text{ at time } t \text{ for non-sepsis patient } s. \end{cases}$$

In which each bracket function is defined as

$$U_{TP}(s,t) = \begin{cases} -0.05 & \text{if } t \leq t_{sepsis} - 12, \\ \frac{1}{6}(t - (t_{sepsis} - 12)) & \text{if } t_{sepsis} - 12 < t \leq t_{sepsis} - 6, \\ 1 - \frac{1}{9}(t - (t_{sepsis} - 6)) & \text{if } t_{sepsis} - 6 < t \leq t_{sepsis} + 3, \\ 0 & \text{otherwise} \end{cases}$$

$$U_{FN}(s,t) = \begin{cases} 0 & \text{if } t \leq t_{sepsis} - 6, \\ -\frac{2}{9}(t - (t_{sepsis} - 6)) & \text{if } t_{sepsis} - 6 < t \leq t_{sepsis} + 3, \\ 0 & \text{otherwise} \end{cases}$$

$U_{FP}(s,t) = -0.05$

$U_{TN}(s,t) = 0.$

This utility function rewards or penalizes classifiers using their predictions on each patient:

- For patients that eventually have sepsis (i.e., with at least one SepsisLabel entry of 1), we reward classifiers that predict sepsis between 12 hours before and 3 hours after $tsepsis$, where the maximum reward is a parameter (1.0). We penalize classifiers that do not predict sepsis or predict sepsis more than 12 hours before $tsepsis$, where the maximum penalty for very early detection is a parameter (0.05) and the maximum penalty for late detection is also a parameter (-2.0).

- For patients that do not eventually have sepsis (i.e., all SepsisLabel entries of 0), we penalize classifiers that predict sepsis, where the maximum penalty for false alarms is a parameter (0.05; equal to the very early detection penalty). We neither reward nor penalize those that do not predict sepsis.

The following figures illustrate the utility function for a sepsis patient (upper plot) with $t_{sepsis} = 48$ as an example, and a non-sepsis patient (lower plot).

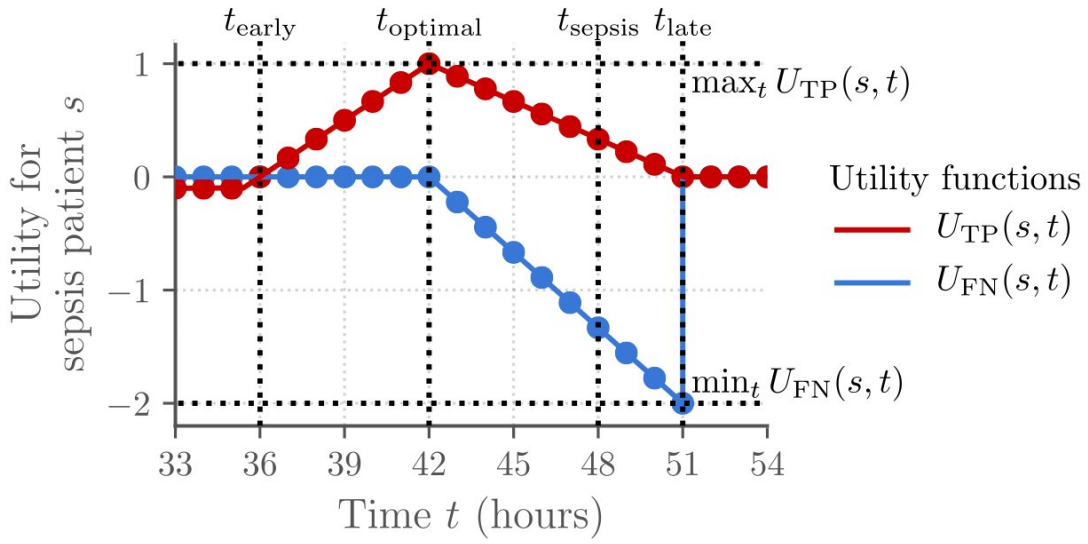

Figure 5.1: Graphic representation of $U_{TP}(s,t)$ and $U_{FN}(s,t)$ on the relevant time window. Figure obtained from https://physionet.org/content/challenge-2019/1.0.0/.

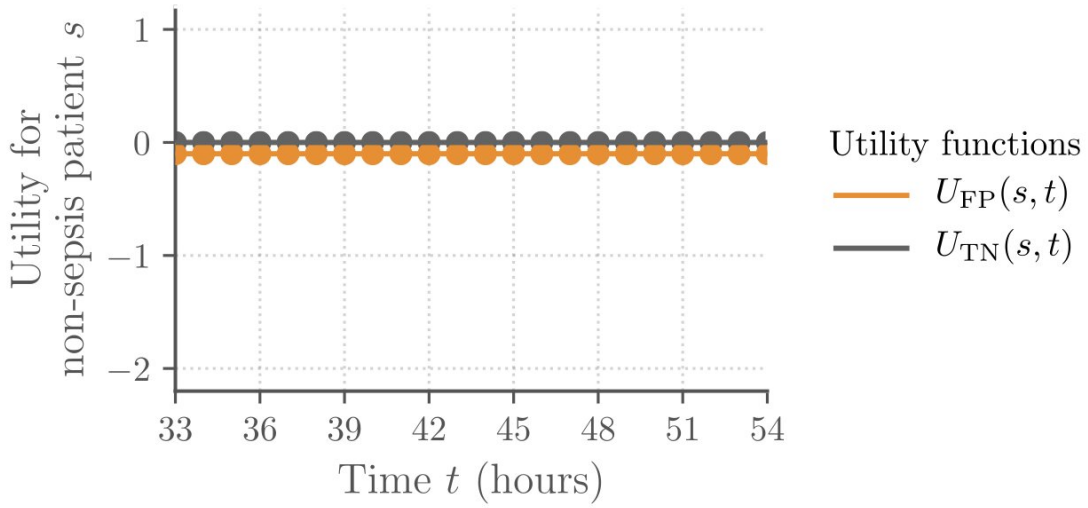

Figure 5.2: Graphic representation of $U_{FP}(s,t)$ and $U_{TN}(s,t)$ on the relevant time window. Figure obtained from https://physionet.org/content/challenge-2019/1.0.0/.

In order to compute a score for a classifier M sum $U(M(s),s,t)$ over each prediction, i.e., over each patient s and each time interval t (each line in the data file):

$$U_{total}(M) = \sum_{s \in S} \sum_{t \in T} U(M(s),s,t).$$

To improve interpretability, we normalized the above classifier score so that the optimal classifier (highest possible score) receives a normalized score of 1 and that a completely inactive classifier (no positive predictions) receives a normalized score of 0:

$$U_{normalized}(M) = \frac{U_{total}(M) - U_{no\ predictions}}{U_{optimal} - U_{no\ predictions}}.$$

This cost function has one dangerous detail within, it requires knowing if the patient has sepsis and exactly when, to calculate what the cost of a prediction is. When looking at the problem as a cost sensitive logistic regression problem, the issue becomes even more apparent. As shown in Elkan [2001], the threshold probability to make optimal decisions is dependent on the cost matrix, yet our cost matrix is dynamic and dependent on an unknown value, meaning we would have to somehow approximate said threshold to be able to make predictions.

To still be able to make predictions without having to find the threshold which would provide the best results we frame the classification problem as a regression problem on an alternative target variable.

We define the new label as

$$= \begin{cases} U_{TP}(s,t) - U_{FN}(s,t) & \text{if sepsis patient s at time t,} \\ U_{FP}(s,t) - U_{TN}(s,t) & \text{if non-sepsis patient s at time t.} \end{cases}.$$

This new variable quantifies the cost benefit of predicting a one instead of a zero, so the threshold to turn it into a binary value is obviously 0. While this transformation carries disadvantages, such as decreases on the MSE of predictions on the target variable for a patient record not implying increases on $\sum_{t \in T} U(s,t)$ for that patient. It, however, carries the major advantage that it encodes, while training, the difference in costs over time. When compared to the binary approach, in which one either uses a single threshold for simplicity and loses the cost dynamics or is forced to approximate a complicated dynamic threshold, this approach provides a much needed balance between simplicity and preserving the cost dynamics.

The problem can now be defined as:

Binary classification experiment: Given a record $\{\boldsymbol{x}_t\}_{t=t_0,...,t_n}$ and SepsisLabel, $\{y_t\}_{t=t_0,...,t_n}$, feed $\{\boldsymbol{x}_t\}_{t=t_0,...,t_n}$ to a model M with output space on the target variable, on the set of time points of the original sequence, then create $\{y'_t\}_{t=t_0,...,t_n}$, which equals one when $M(\{\boldsymbol{x}_t\}_{t=t_0,...,t_n}, \{t_0, ..., t_n\})$ is greater or equal to zero, and zero otherwise. Repeat for the entire dataset and calculate $U_{normalized}(M)$.

5.3 Experimental Models

To effectively study the impact of ODE machinery, one needs to train models with and without it; therefore, it was chosen that four models architectures would be trained, one RNN, one ODE-RNN, one RNN-VAE and one RNN-ODE encoder latent ODE.

In the problem specification section, however, we defined problems with two distinct output spaces. We will however, not make two models for each architecture. Instead, each model will output the concatenation of input and label, through two different output neural networks, both of which will read the same hidden/latent state. This is done for simplicity, as each model is heavy and takes a long time to train, and because updating the parameters to learn better one task should not particularly hurt the other, as the hidden/latent state should be able to encode both the input state and label, and the decoders are updated independently from each other.

To this end, the loss functions for each model will be a weighted sum of the loss for each task. Meaning for the RNN and ODE-RNN models, their loss function will be

$$L(\theta, \theta_{input}, \theta_{label}, \{\boldsymbol{x}_t, y_t, t\}_{t=t_0,\dots,t_n}) = -\alpha * LL_{masked}(\theta, \theta_{input}, \{\boldsymbol{x}_t, t\}_{t=t_0,\dots,t_n})$$
$$+ \beta * \text{MSE}_{masked}(\theta, \theta_{label}, \{\boldsymbol{x}_t, y_t, t\}_{t=t_0,\dots,t_n}),$$

where LL_{masked} is the Log-Likelihood for the subset of unmasked points, α, β are positive constants, θ the parameters of the model except for output layers and $\theta_{input}, \theta_{label}$ the parameters for the input space output layer and the label space output layers respectively.

When it comes to the autoencoder family of models, the way to use them changes dramatically in this dataset. The classifier score function needs a label prediction for every timestep by looking at the classifier score function. While on the RNN variant this is trivial, as each prediction only uses information from previous time points, it does not apply to the autoencoders. By encoding the entire prior into an initial state z_0 and then predicting all labels from this state, one would be using future information to classify the past, which is not the objective here. This way, while RNN's want to turn an input sequence on T time points into a label sequence on the same time points, autoencoders want to output a single label on the last time point t_T. If we require knowing the labels on previous time points, we need to encode and decode the subsequences up to said points. This way, the loss function is

$$L(\theta, \phi, \theta_{input}, \theta_{label}, \{\boldsymbol{x}_t, y_t, t\}_{t=t_0,\dots,t_n}) = -\alpha * \text{ELBO}_{masked}(\theta, \phi, \theta_{input}, \{\boldsymbol{x}_t, t\}_{t=t_0,\dots,t_n})$$
$$+ \beta * \text{MSE}_{masked}(\theta, \phi, \theta_{label}, \{\boldsymbol{x}_t, y_t, t\}_{t=t_0,\dots,t_n}),$$

where ELBO_{masked} is the evidence lower bound for the subset of unmasked points, α, β are positive constants, θ,ϕ are the decoder and encoder parameters of the model except for output layers and $\theta_{input}, \theta_{label}$ the parameters for the input space output layer and the label space output layers respectively.

By looking at 4.1 one can see that the model only learns how to encode complete sequences, since the term $-\log\, q_\phi(z_0|\{\boldsymbol{x}_i, t_i\}_{i=0}^n)$ always computes the encoding of the entire sequence, meaning the only way of learning the encoder parameters to encode subsequences is to run those subsequences through the model. Since each sequence is run backwards through time, one must encode the entire subsequence instead of simply updating the encoding of smaller subsequences. This means the training data for the autoencoder will be all subsequences of the training data for the RNN family.

In order to keep the data shape equal for every model, we allow the autoencoder models to still output a label per time point, but since the loss function only reads the last the output layer will only be calibrated to output the last label accurately.

The model hyperparameters were based on the findings of Rubanova et al. [2019]. Initial toy experi-

ments were made to find parameters that would allow comparing the different models. A small ODE-RNN network was used to find α and β for the respective loss function. From there one can extrapolate the weights for the other loss functions, such that the balance between the parts in each loss function be the same as the one found for the RNN. The network hyperparameters for an ODE model should match their non-ODE version, on the overlapping components. Even though this means we will be comparing models with quite a difference in the number of parameters, one can measure what benefit these extra parameters bring and whether it proves beneficial enough to justify their use.

5.4 Data Treatment

It was decided that modifications to the data should be kept to a minimum, as the main objective was to study how well the models can learn the original data and introducing new relevant features could skew the experiment. This way, all modifications were deemed essential for the models to learn the original data properly.

The first modification was to discard variables which were deemed unnecessary for learning, in this case, the identifier of the hospital, and separate the time feature, so as to not have it as a network node input.

The second modification was the scale normalization of the data, as having all features on the same scale is of great help to the gradient descent algorithm, making the training process much faster, significantly when feature ranges vary a lot, as in this case.

The third and most extensive modification was the downsampling of non-sepsis records in the data. Sepsis patients number only 1790 out of the total 40643 records, a very significant class imbalance. If this were to be kept as a logistic regression problem, downsampling would not be needed as changing the acceptance threshold would solve the tendency to classify everything as non-sepsis. In the SepsisScore regression variant of this problem, the imbalance does prove an issue, with toy models showing much better results if some form of downsampling was performed on the training data. The idea of why the imbalance is a problem is roughly the same as the one in the logistic regression if the threshold was not updated. The model is trying to approximate a function which will be constant of value -0.05 for almost every record, and on those which are from sepsis patients a good chunk of the function is still constant -0.05. This means that the easiest way of lowering loss is simply to predict -0.05 all the time. There is then the additional problem that even if the model does, somehow, detect a good chance of sepsis and increase its prediction it might not be enough to go over the 0 threshold. Because of the similarity to the logistic regression problem, we try and adapt its solution to the issue, and even though we do not want to change our threshold, we can take inspiration from the downsampling proportions shown in Elkan [2001]. It can be noticed that when the sum of SepsisScore over a dataset is 0, equal to the

threshold, its similarities with a logistic regression with a threshold of 0.5 on a balanced dataset become apparent. We hypothesise that by random downsampling with proportions, such that the expected sum of SepsisScore is 0 would obtain the best results.

The final modification is the removal of every record that does not start on $t = 0$. This is meant to help calculate $U_{normalized}$ for autoencoder models when batching is being used. We have already mentioned that the training data for autoencoders will be increased by introducing all subsequences of each record to the training data. When batching is being used, we must pad the sequences, so all have the same length; by having records that do not start at $t = 0$ padding would also have to be added on the start of sequences. For large batches it would become much harder to reconstruct one sequence of predicted values from every subsequence of a record. It is worth mentioning that the model could promptly deal with these missing entries on early time points; the added difficulty in calculating the score is the only reason for their removal.

It is also worth mentioning that despite removing records on two steps, the dataset is still extensive, and when subsequences are introduced, it becomes too large to work. To aggravate the problem even more, since all sequences must be the same size for batching and there is a significant variance in sequence size, the average sequence size decreasing with the subsequences does not mean the average batch length will decrease. All these reasons force a downsample on the entire dataset, in which for every model it can be chosen to use n records for training. The downsampling process is very noticeable on the autoencoder training data.

After treatment, we split the data into training, validation and testing. After considering that, on the original Physionet challenge, the testing data for the classification problem was the entirety of the records from a hospital, which is not available, it was decided to use the data from one of the two hospitals for testing and the other for training and validation. This way 80% of Unit1 hospital's records are used for training, 20% for validation and Unit2 hospital's records are used for testing. The training data received all modifications mentioned, validation and testing data received only the first two.

6

Experimental Results

Contents

6.1 Interpolation Task 46
6.2 Extrapolation Task 47
6.3 Classification Task 47
6.4 Discussion 48

To find the random downsampling probability p on the train data, we summed SepsisScore over the entire training dataset, obtaining -27402 for all non-sepsis and 15262 for all sepsis patients, therefore $p = 0.443$ to remove a non-sepsis record from the dataset.

To approximate α and β it was decided to keep α constant and change the values of β, plotting the LL_{masked} and $U_{normalized}$ of each toy ODE-RNN model in the validation dataset.

Table 6.1: Utility score and Likelihood estimation for different values of β.

Value of β	Sepsis Score	LogLikelihood
500	0.1659	-26.505
1000	0.1852	-28.032
1500	0.1647	-26.546
2000	0.2101	-26.950
2500	0.1695	-27.735

As both $U_{normalized}$ and the LL_{masked} did not vary much with the change of β, as shown in table 6.1 we decided to use to use $\alpha = 1$ and $\beta = 2000$ for RNN type models as it performed the best in the classification task.

For autoencoder models we train one toy model with $\beta = 2000$ and $\alpha = 1$ and calculate the percentage of the loss due to $\beta * \text{MSE}_{masked}$, finally we simply multiply α by a constant in order to give the classification task the same share of the loss as the RNN models, so we use $\alpha = \frac{1}{30}$. We do not replicate the $U_{normalized}$ table for the toy autoencoders since it takes a very long time to calculate the utility score for every model.

All models were trained using the following hyper parameters:

- RNN and ODE-RNN model

  We used a hidden space of size 20 for both models, and for the ODE-RNN dynamics network we used feedforward network of 3 layers with 50 nodes each.

- RNN-VAE

  For the encoder we used a GRU with hidden space of size 40, and for the decoder a GRU with hidden space of size 20, and a standard normal distribution as prior.

- Latent-ODE

  For the encoder we used an ODE-RNN model with hidden space of size 40, and for the encoder dynamics network we used a feedforward network of 3 layers with 50 nodes each. We use a latent space of size 20 and for the latent dynamics network we used a feedforward network of 3 layers with 50 nodes each; finally we use a single layer feedforward network decoder.

For the classification task we equipped each model with an alternative single layer feedforward neural network decoder.

Every autoregressive model was trained using the all available sequences in the training set, with batches of 100 for 10 epochs with a learning rate of 10^{-2}. All autoencoder models were trained using subsequences of the available data, up to a maximum of the total number of sequences used for the autoregressive models. Again we used batches of 100 for 10 epochs and learning rate of 10^{-2}.

6.1 Interpolation Task

For the interpolation task we removed all constant variables from the evaluation metric, as what we intend to evaluate is how well the model can provide information we don't already know.

For the experiment we used probability of removal 25%, 50%, 75% and 90%, obtaining the following results:

Table 6.2: Interpolation Test Mean Squared Error (MSE) $(\times 10^{-2})$.

	Probability of point removal			
Model	25%	50%	75%	90%
RNN	0.6408	1.0921	1.8073	2.2678
ODE-RNN	0.5123	0.7758	1.1087	1.2772
RNN VAE	0.4303	0.4576	0.5132	0.5581
Latent ODE	0.6882	1.0404	1.5569	2.1399

One can see a major improvement on the interpolation task with the addition of ODE machinery for the auto-regressive models, which becomes more pronounced the sparser the input sequences become, which support our initial hypothesis.

As for the autoencoder models the Latent-ODE actually under performs, even though it is more parameter intensive, which goes against what was observed in Rubanova et al. [2019], where the RNN VAEs under performed when compared to both Latent ODEs and RNN-ODEs. We hypothesise that the RNN VAE has obtained much better relative performance in this experiment due to regular measurement times, which was not the case in Rubanova et al. [2019]. Therefore, while for the RNN missing values pose a big challenge, the RNN VAE can deal with them quite well and its bigger weakness seems to be measurement irregularity.

It is also surprising how much more the data sparsity affects the latent ODE when compared to the RNN VAE.

6.2 Extrapolation Task

We have again removed all constant variables from the evaluation metric, and have used 25%, 50% and 75% of each sequence for encoding and extrapolate what remains.

We hypothesise that running the ODE-RNN by re-feeding predictions just like a classic RNN will obtain poorer results than just using its dynamics for extrapolation. To test this hypothesis we also tried both types of extrapolation.

Table 6.3: Extrapolation Test Mean Squared Error (MSE) $(\times 10^{-2})$.

Model	25% length	50% length	75% length
RNN	1.1330	0.8472	0.5819
ODE-RNN (pure dynamics)	0.3771	0.3659	0.3626
ODE-RNN (re-feed predictions)	1.6013	1.3788	1.0173
RNN VAE	2.4517	1.4545	1.1221
Latent ODE	0.4215	0.4241	0.4260

Again, the ODE-RNN outperforms its classic version, and one can explicitly see the benefit of relying only on the dynamics for extrapolation. Here the Latent ODE outperforms the RNN VAE, we hypothesise that the reason for this difference in performance is the same as the RNN and ODE-RNN, by not relying on re-feeding predictions the Latent ODE becomes much more robust and is able to consistently return better predictions.

To justify the much worse performance obtained when re-feeding predictions to the ODE-RNN we have to look at difference in sparsity of the original input and the predictions. While the model was trained on data which consisted mostly of missing values, and therefore learned that a value 0 would mean missing, the re-fed predictions have no missing values, something the model never got to learn, and therefore throwing it into disarray. This can still be overcome while using re-feeding, by, for example, re-feeding predictions instead of observations during training with a probability p. This, however, also carries disadvantages, since a missing value carries information a model can learn from, and therefore would imply a worse performance in the other tasks. Another way of dealing with this problem is to manually add sparsity to the re-fed predictions, by removing points with a certain probability per feature. Still, it is not obvious if these approaches would obtain better results than using the dynamics without GRU cell updates. Further testing would be needed to confirm what effects these approaches would bring.

6.3 Classification Task

For this task we calculated the utility scores, our main evaluation metric, followed by accuracy, precision and recall. We did not use the usual binary classification metrics such as the AUC as we were not using

a logistic regression.

Table 6.4: Utility Score, accuracy, precision and recall for classification task.

Model	Utility Score	Accuracy	Precision	Recall
RNN	0.08002	0.721	0.049	0.593
ODE-RNN	0.13794	0.683	0.055	0.7304
RNN VAE	0.22745	0.897	0.081	0.417
Latent ODE	0.24384	0.877	0.078	0.478

In this task all ODE variants outperformed their classical variants, with autoencoder models being dominant. The fact that the Latent ODE obtained the best results in this task is surprising considering that it performed quite poorly on the interpolation task. Similarly the results of the ODE-RNN are also surprising considering how well it performed in both interpolation and extrapolation. As expected the cost function leads to a very low precision, as a false positive is penalized much less than a false negative.

6.4 Discussion

Every model trained was designed to be able to perform every task proposed, however this lack of specialization affects the obtained results, so for this discussion it is important to take this multi-tasking into account, since the benefit of not having to train one model per task comes at a cost.

We start with the autoregressive models. In all tasks the ODE-RNN outperformed the RNN by a significant amount. This difference in results increases as the number of true points the models receive decreases, both in sparsity when it comes interpolation and on the ratio of observed points to predicted points in extrapolation. This helps cement the hypothesis that the ODE-RNN is consistently a better model choice than the RNN when it comes to sparse data. One can also conclude that for sparse data, relying solely on the dynamics for extrapolation obtains better results when using the ODE-RNN. The ODE-RNN's increased cost in training time and parameter size is very manageable, the ODE-RNN has a total 141928 parameters and the RNN a total of 105658, and the ODE-RNN is 16% slower in training. As for prediction time, the ODE-RNN is about 5 times slower when it comes to interpolation. This difference aggravates on the extrapolation task where the ODE-RNN is 10 times slower at 25% length encoding length. The ODE version also has the added benefit of being able to do predictions in between time points. Considering autoregressive models are most commonly used in extrapolation tasks, the robustness that the dynamics network provide is crucial if there is a need to extrapolate longer arbitrary periods of time. The autoregressive models performed the worse in the classification task, tending to be much more inert than their autoencoder variants.

As for the autoencoder, the picture is less clear than their autoregressive counterparts. The classical version vastly outperformed all other models in the interpolation task, which is where this model type is

usually used the most, with the ODE upgraded version underperforming not only its classical version but also the ODE-RNN, which is surprising since the latent-ODE is not only the most parameter intensive model, at 162688 when compared to the RNN-VAE's 123788. Further experiments are needed to identify if the latent-ODE is really a worse model for interpolation on regularly timed sparse time series, or if it is incapable of multitasking and needs one model trained for each. As for extrapolation, the RNN-VAE obtained the poorest results out of all models, while the Latent ODE obtained the second best, and reasonably close to the first. We attribute the vastly superior performance on not having to re-feed predicted values, therefore leading to a slower accumulation of errors. As for the classification task the Latent-ODE obtained the best results followed by the RNN-VAE. We consider this surprising as the Latent-ODE did quite poorly on interpolation, so we expected it to be the same in this task, which was not. We think that the Latent ODE is not good at multi-tasking and since we selected a loss function weights that prioritize the classification task, the model did its function for the task it considered the most important and somewhat ignored the others. As for training time the Latent-ODE is considerably slower than the RNN VAE, by a factor of 5, with the same happening for the interpolation prediction time where it is slower by 3 times, while extrapolation on 25% encoding is 5 times slower. When compared to the autoregressive models, by having to consume a full sequence per timepoint-label prediction, autoencoder models are significantly slower for the classification task.

7

Conclusion

Contents

7.1 Contributions 53
7.2 Future Work 53

To conclude, we highlight the contributions of this thesis and present some topics which we think should be further explored.

7.1 Contributions

This thesis was built upon the work of Rubanova et al. [2019], and designed to reinforce their empirical results. To achieve this end, this work presented three tasks, interpolation, extrapolation and classification, on a dataset with the characteristics necessary to further analyze the performance of the neural ODE family of models.

We presented a classification problem on medical data with labels per time point as opposed to a single label per sequence, to explore whether neural ODEs bring no major benefit as in the mortality experiment in Rubanova et al. [2019], or if this model family can obtain better results on a label time series, and obtained empirical results to back our hypothesis.

We presented an alternative way of using autoregressive neural ODEs for extrapolation by relying solely on its dynamics and not re-feeding previous predictions to the model. We obtained results to show that it achieves more accurate extrapolations and stays more stable as the extrapolated points increase.

7.2 Future Work

This thesis used the same neural ODE architecture as Rubanova et al. [2019], since its purpose was to reinforce the empirical evidence that its use was beneficial. This does not mean, however, that the models presented are the only ones one can build. Therefore attempting to create new neural ODE architectures and testing their results is a promising area of research, as for example, instead of upgrading the RNN-VAE into a latent ODE as in this thesis, one could try a ODE-RNN encoder and decoder VAE, which would hold different benefits and challenges than the latent ODE, and still be able to model the time series continuously.

Another possible path would be developing a Pytorch compatible package that would implement a fundamental family of neural ODEs, and provide the tools for the user to construct new architectures easily. This would prove very beneficial since, even though we already have an ODE solver package used to create this family of networks, it forces one to build every net from scratch. In order to propel the use of these network types, particularly in industry, one user-friendly package where one could create, for example, an ODE-RNN with the same ease one creates a classic RNN would be very beneficial.

Bibliography

Yulia Rubanova, Ricky T. Q. Chen, and David Duvenaud. Latent odes for irregularly-sampled time series. *CoRR*, abs/1907.03907, 2019. URL http://arxiv.org/abs/1907.03907.

Tian Qi Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. Neural ordinary differential equations. *CoRR*, abs/1806.07366, 2018. URL http://arxiv.org/abs/1806.07366.

J.R. Dormand and P.J. Prince. A family of embedded runge-kutta formulae. *Journal of Computational and Applied Mathematics*, 6(1):19–26, 1980. ISSN 0377-0427. doi: https://doi.org/10.1016/0771-050X(80)90013-3. URL https://www.sciencedirect.com/science/article/pii/0771050X80900133.

H. Kelley. Gradient theory of optimal flight paths. *ARS Journal*, 30:947–954, 1960.

Arthur Bryson. A gradient method for optimizing multi-stage allocation processes. *Proc. Harvard Univ. Symposium on digital computers and their applications*, 72, 1961.

David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986. ISSN 1476-4687. doi: 10.1038/323533a0. URL https://doi.org/10.1038/323533a0.

Diederik P Kingma and Max Welling. Auto-encoding variational bayes, 2014.

Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. Stochastic backpropagation and approximate inference in deep generative models, 2014.

H Bourlard and Y Kamp. Auto-association by multilayer perceptrons and singular value decomposition. *Biological Cybernetics*, 59:291–294, 1988. ISSN 1432-0770. doi: 10.1007/BF00332918. URL https://doi.org/10.1007/BF00332918.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

Yiping Lu, Aoxiao Zhong, Quanzheng Li, and Bin Dong. Beyond finite layer neural networks: Bridging deep architectures and numerical differential equations. *CoRR*, abs/1710.10121, 2017. URL http://arxiv.org/abs/1710.10121.

Eldad Haber and Lars Ruthotto. Stable architectures for deep neural networks. *CoRR*, abs/1705.03341, 2017. URL http://arxiv.org/abs/1705.03341.

Lars Ruthotto and Eldad Haber. Deep neural networks motivated by partial differential equations. *CoRR*, abs/1804.04272, 2018. URL http://arxiv.org/abs/1804.04272.

L. Pontryagin. Mathematical theory of optimal processes. 1962.

François Chollet. *Deep Learning with Python*. Manning, November 2017. ISBN 9781617294433.

Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. *CoRR*, abs/1409.3215, 2014. URL http://arxiv.org/abs/1409.3215.

Mervyn Singer, Clifford S. Deutschman, Christopher Warren Seymour, Manu Shankar-Hari, Djillali Annane, Michael Bauer, Rinaldo Bellomo, Gordon R. Bernard, Jean-Daniel Chiche, Craig M. Coopersmith, Richard S. Hotchkiss, Mitchell M. Levy, John C. Marshall, Greg S. Martin, Steven M. Opal, Gordon D. Rubenfeld, Tom van der Poll, Jean-Louis Vincent, and Derek C. Angus. The Third International Consensus Definitions for Sepsis and Septic Shock (Sepsis-3). *JAMA*, 315(8):801–810, 02 2016. ISSN 0098-7484. doi: 10.1001/jama.2016.0287. URL https://doi.org/10.1001/jama.2016.0287.

Matthew A Reyna, Christopher S Josef, Russell Jeter, Supreeth P Shashikumar, M Brandon Westover, Shamim Nemati, Gari D Clifford, and Ashish Sharma. Early prediction of sepsis from clinical data: The physionet/computing in cardiology challenge 2019. *Critical Care Medicine*, 48, 2020. ISSN 0090-3493. URL https://journals.lww.com/ccmjournal/Fulltext/2020/02000/Early_Prediction_of_Sepsis_From_Clinical_Data__The.10.aspx.

Charles Elkan. The foundations of cost-sensitive learning. *Proceedings of the Seventeenth International Conference on Artificial Intelligence: 4-10 August 2001; Seattle*, 1, 05 2001.