Neural Ordinary Differential Equation

M.Facci, D.Zurlo

Dipartimento di Ingegneria Informatica Automatica e Gestionale
Antonio Ruberti
Control Engineering
Sapienza Università di Roma

January 15, 2022

Contents

1 Introduction

Neural Networks (NN) is a set of algorithms that allows computers to solve a common problem and recognize patterns, nowadays used in a wide range of tasks, thank the effort made by researchers since the first intuition made by D. O. Hebbo in the book *The Organization of Behaviour* published in 1949, in which are proposed connection with a complex model of the brain. After that in 1958 J. Von Neumann published *The Computer and the Brain* in which points out the poor precision that the structures of the models proposed by D. O. Hebb have to perform complex problems. In the same year, Frak Rosenblatt published the first scheme of NN called *Perceptron*, which is the basic element of all the NN used nowadays. This last publication gives inspiration to many researchers for a decade, where some of these researchers were financed by the United State of America until Marvin Minsky and Seymour A. Papert have shown the big limit of these simple structures models. After some years thanks to the works made, first by Paul Werbos in his Ph.D. thesis and then by J. J. Hopfield in his work where he studies models to recognize very general patterns, the researchers began to study NNs and continue to this day. In these years, one of the most used techniques, called Backpropagation, has been developed to train a NN. It allows adjusting all the weights present in a network systematically in two phases; the forward phase, in which is computed the output of the network and the error with the real value *label*, and the backward phase in which the error is backpropagated from the output to the input. Moreover, new architectures suitable for specific tasks such as Convolutional Neural Networks (CNN) have been developed, suitable for example for image classification tasks, because they are able to work with very large input and they can extract important features from the images. Another example is represented by Recurrent Neural Networks (RNN) suitable for tasks in which there is a self-correlation among inputs e.g. translation, image captioning, sentiment analysis, etc. Many other architectures have been developed. In particular, recently a new architecture, called Neural Ordinary Differential Equation (NeuralODE) has been developed by Ricky Tian Qi Chen et al. in [1] from the University of Toronto, this paper became prominent after being named one of the best student papers at NeurIPS 2018 in Montreal because it showed a radical new design capable to describe a dynamical system in continuous time. In this way, they give a powerful tool to identify a system and its parameters without using well-known techniques based on System Identification theory. Therefore, NeuralODEs make a bridge between Artificial Intelligence and dynamical systems, and not only. In fact, they can be used also for simpler tasks such as image classification.
In this project the concept of NeuralODE and its architecture are developed, first focusing on the capability of this Deep Neural Network (DNN) to perform a simple System Identification task, then exploiting it for the classification of electrocardiogram signals (ECG Heartbeat Classification task), also comparing it with another architecture from which it takes shape, ResNet.

2 Residual Networks

Simple NNs with just one layer are not used in practice because they are not able to learn complex relationships between input and output. For this reason are used Deep Neural Networks (DNN), which are NNs with more than one layer. This kind of NNs are more accurate than the simple one, but the price to pay is that they are computationally expensive due to the number of weights in the network. Moreover, they are prone to overfitting, in particular, the more the network is deep more are the eventuality of overfitting and also they have the so-called *degradation problem*, so when training deep networks there comes a point where an increase in depth causes accuracy to saturate, then degrade rapidly. The researchers of Mi-

crosoft introduced Residual Neural Networks to avoid this problem and discovered that many problems can be addressed using this kind of network.
ResNet is composed of layers stacked one on top of the other and in addition, it has a skipping connection, as it is possible to see from the figure 1.

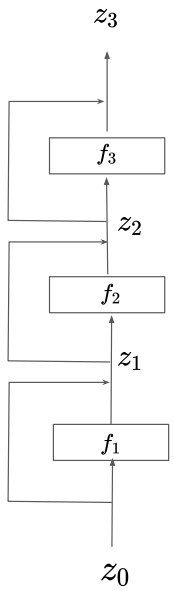


Figure 1: ResNet architecture

The output after the first layer is given by the sum of the output of the layer $f(z_t, \theta_{t+1})$ and the skip connection z_{t-1}. The generic state transformation in a ResNet can be formalized as:

$$z_{t+1} = z_t + f(z_t, \theta_{t+1}) \tag{1}$$

The importance of the skip connection is that prevent to discard of important information by f due to the backpropagation, in this way the information bypass the layer. The depth of this network is given by t that is the number of transformations. There is an evident correlation between ResNet and ODE, in fact, if we look at all the transformations between the input of the network and the output, given by

$$\begin{aligned} z_1 &= z_0 + f(z_0, \theta_1) \\ z_2 &= z_1 + f(z_1, \theta_2) \\ &\cdot \\ &\cdot \\ &\cdot \\ z_{t+1} &= z_t + f(z_t, \theta_{t+1}) \end{aligned} \tag{2}$$

it relies upon the same recursive relationship as the method of Euler to solve differential equation numerically. One of the biggest downside of ResNet is that they are computationally expensive to train due to its depth.

3 Neural Ordinary Differential Equation

3.0.1 Ordinary Differential Equations

ODEs describe a dynamical system and, in general, the rate of change of a variable $y(t)$ with respect to an independent variable, in this case, the time t.
An explicit ODE of the n-th order has the form

$$y^{(n)} = f(t, y, y', y'', ..., y^{(n-1)}, \theta) \tag{3}$$

where f is the differential, a function of the independent variable t, the dependent variable $y(t)$ and its derivatives with respect to time, that is $y^{(i)} = \frac{d^i y}{dt^i}$ with $i = 1...n$, and an optional parameter θ. A higher-order ODE, with derivatives of order greater than one, can always be re-expressed as a system of first-order ODEs by redefining the higher-order differentials as new variables.
A system of N linear first-order ODEs can be written in matricial form as

$$y' = Ay + b \tag{4}$$

with the $N \times N$ coefficient matrix A and the $N \times 1$ constant vector b.

3.0.2 Initial Value Problems (IVPs)

A solution $y(t) = G(t)$ to a first-order ODE is only uniquely defined up to an unknown integration constant. Similarly, to an n-th order ODE correspond n integration constants. This aspect implies that, for a full solution, knowledge of an initial condition (t_0, y_0) is required to define these constants. An ODE together with a given initial condition is known as an initial value problem (IVP). And the solution $y(t)$ to the IVP satisfies both the ODE and the initial condition $y(t_0) = y_0$.
More precisely an IVP for a first-order ODE is defined as:

$$\frac{dy(t)}{dt} = f(y(t), t, \theta) \tag{5}$$

$$y(t_0) = y_0 \tag{6}$$

supposing $y(t_1)$ is needed, the solution is stated as:

$$y(t_1) = y(t_0) + \int_{t_0}^{t_1} f(y(t), t, \theta) dt \tag{7}$$

If the (7) cannot be analytically integrated one has to use approximation methods, i.e. numerical integration.
One of the most simple and well-known is the aforementioned Euler's method which, starting from the initial condition $y(t_0) = y_0$, takes linear steps along the ODE $y'(t) = f(t, y(t))$ with a fixed time-step h. This means that the updating law is:

$$y(t_{i+1}) = y(t_i) + hf(y(t_i), t_i) \tag{8}$$

with $t_{i+1} = t_i + h$ and f is the derivative.

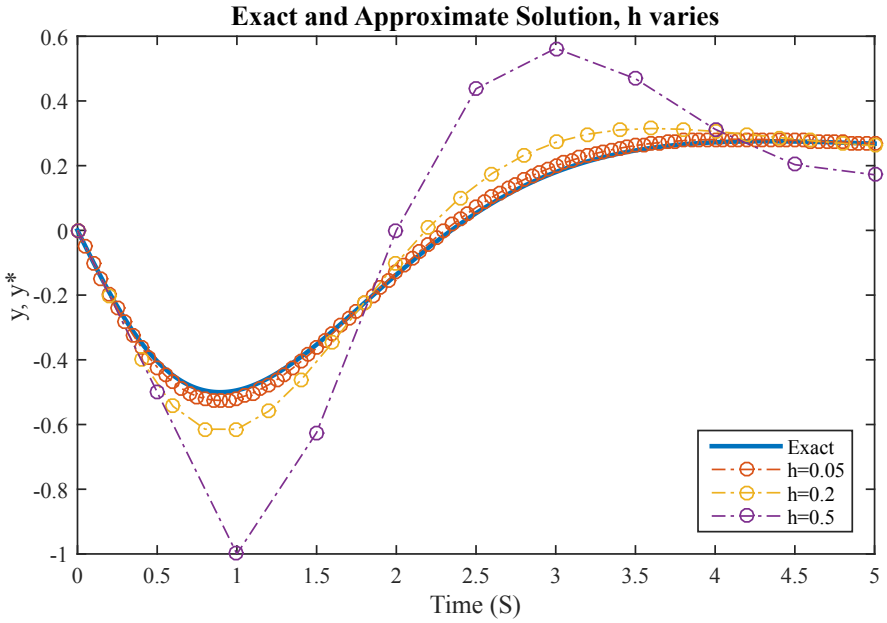


Figure 2: Euler's method with several values of h, the solution is better with smaller values of h.

Most methods for numerically solving IVPs are made for systems of first-order ODEs since all higher-order ODEs can be rewritten as such. Other first-order solvers are the ones belonging to the Runge-Kutta family, and they are based on the idea that instead of going from y_i to y_{i+1} directly, a set of intermediate steps are taken at fractions of the step size h and added in a weighted average.
Therefore, summarizing Euler's method takes a single step in the direction of the local derivative. Runge-Kutta takes several intermediate shorter steps in the time interval, and from these constructs a final long step.
There are many other ODE solvers to choose from, the main criteria to care about for the choice are stiffness, number of calculations per iteration, implicit or explicit solver, single-step size or multi-step size (adaptive).
However, all of these methods are computationally intensive, especially during the training phase, which requires differentiation of the integration steps to be able to add up all gradients of the network parameters θ, incurring a high memory cost.
In the paper, Chen et al. presented an alternative approach to calculating the gradients of the ODE by using the so-called adjoint method by Pontryagin. This method works by solving a second augmented ODE backward in time, and can be used with all ODE solvers, and has a low memory cost.

3.0.3 NeuralODE Structure

As said in the section 1 the NeuralODEs are an evolution of ResNets. In fact both build complex transformation on the input in a sequential manner where the ResNets, as seen in section 2, can be interpreted as discretization of ODEs, instead in NeuralODEs the discretization step tends to zero. The general architecture of a NeuralODE can be seen in figure 17, where it is possible to see that it is composed by one shred layer $f(x, t)$ where t is the layers number. In analogy with the ResNet the first layer of the ResNet can be represented in thee NeuralODE architecture as $f(z_0, 0)$. In particular the input z_0 goes to the shared layer and also goes through the skip connection and the result is z_1, that is the input of the second layer and so on. In this way it is possible to compute any output in the ResNet using only a shared layer. In

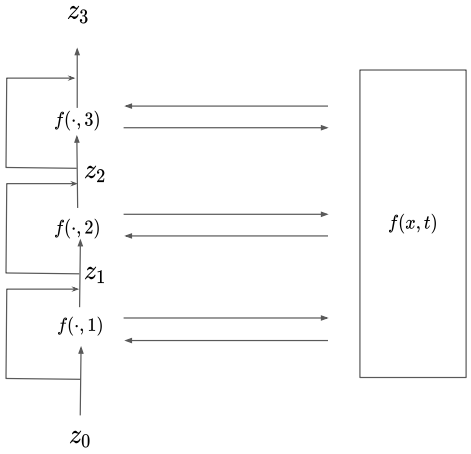


Figure 3: NeuralODE architecture

NerualODE the function f has an additional parameter that is t instead in the ResNet there is only one parameter that is z and this is the first big insight of NeuralODE. In other word each intermediate layer of ResNet can be mimic just by calling the hared layer in the NeuralODE in which the value of t is incremented. This makes the NeuralODE more memory efficient. In fact the idea proposed by the authors is that they consider the continuous limit of each discrete layer in the ResNet, so that the transformation of the hidden state become continuous, that exactly describes a set of ordinary differential equations as follow

$$\frac{\partial z}{\partial t} = f(z(t), t, \theta_t) \tag{9}$$

Another innovation of NeuralODE is the back propagation through depth. To understand how it woks it is necessary to compute the loss function given by

$$\mathcal{L}(t_0, t_1, \theta_t) = \mathcal{L}(ODESolve(z(t_0), t_0, t_1, \theta, f)) \tag{10}$$

where $ODESolve$ is any solver and f is the NN. Then it is necessary to compute the gradient of the loss with respect to the hidden state

$$\frac{\partial \mathcal{L}}{\partial z(t)} \tag{11}$$

that depend on time (depth).

3.0.4 Backpropagation Using the Adjoint Sensitivity Method

The function approximation problem now takes place over a continuous hidden state dynamics and t_i and t_{i+1} can be considered as two additional free parameters to be optimised with a numerical ODE solver called ODESolve: $\mathbf{z}(t_{i+1}) = ODESolve(\mathbf{z}(t_i), f, t_i, t_{i+1}, \theta)$, where f is a neural network.
The central innovation of the paper is an algorithm for backpropagating (reverse-mode differentiation) through the continuous hidden state dynamics.
Even though backpropagating through the ODE solver steps is straightforward, the memory cost is high and additional numerical errors occur. In order to perform backpropagation, the gradient of the loss function with respect to all parameters must be computed.

As different ODE solvers may take a varying number of steps when numerically integrating between two time points, a general method for computing the gradient of the loss at each intermediate step is required. The adjoint sensitivity method can be used regardless of the choice of the ODE solver and with constant memory consumption.
Therefore, a differentiable loss function is defined as:

$$L(\mathbf{z}(t_1)) = L\left(\mathbf{z}(t_0) + \int_{t_0}^{t_1} f(\mathbf{z}(t), t, \theta)dt\right) = L(ODESolve(\mathbf{z}(t_0), f, t_0, t_1, \theta) \tag{12}$$

Alongside the original dynamical system that describes the process, the adjoint system describes the derivative states at each point of the process backward, via the chain rule. In this way it is possible to obtain the derivative by the initial state, and, similarly, by the parameters of a function that is modeling the dynamics (one "residual block", or the discretization step in the "old" Euler's method).

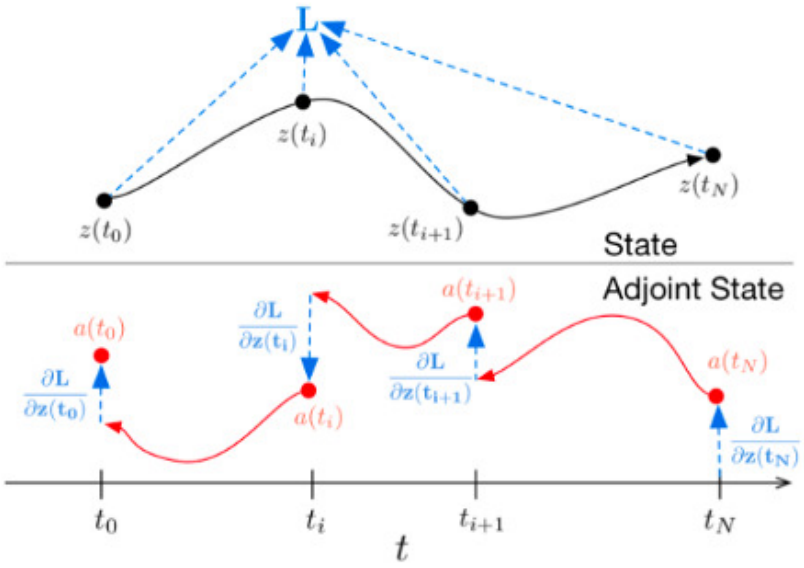


Figure 4: Backpropagation gradients for the ODESolve() method.

The adjoint method works by constructing the so-called adjoint state $a(t) = \frac{dL}{d\mathbf{z}(t)}$ where L is the loss function and $\mathbf{z}(t)$ is the output after each step taken by the ODE solver, which follows the well-known differential equation

$$\frac{d\mathbf{z}(t)}{dt} = f(\mathbf{z}(t), t, \theta) \tag{13}$$

The adjoint state follows the differential equation

$$\frac{d\mathbf{a}(t)}{dt} = -\mathbf{a}(t)\frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}(t)} \tag{14}$$

$$\frac{d\mathbf{a}(t)}{dt} = \lim_{\varepsilon \to 0^+} \frac{\mathbf{a}(t+\varepsilon) - \mathbf{a}(t)}{\varepsilon} \tag{39}$$

$$= \lim_{\varepsilon \to 0^+} \frac{\mathbf{a}(t+\varepsilon) - \mathbf{a}(t+\varepsilon)\frac{\partial}{\partial \mathbf{z}(t)} T_\varepsilon(\mathbf{z}(t))}{\varepsilon} \quad \text{(by Eq 38)} \tag{40}$$

$$= \lim_{\varepsilon \to 0^+} \frac{\mathbf{a}(t+\varepsilon) - \mathbf{a}(t+\varepsilon)\frac{\partial}{\partial \mathbf{z}(t)} \left(\mathbf{z}(t) + \varepsilon f(\mathbf{z}(t), t, \theta) + \mathcal{O}(\varepsilon^2)\right)}{\varepsilon} \quad \text{(Taylor series around } \mathbf{z}(t)\text{)} \tag{41}$$

$$= \lim_{\varepsilon \to 0^+} \frac{\mathbf{a}(t+\varepsilon) - \mathbf{a}(t+\varepsilon)\left(I + \varepsilon \frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}(t)} + \mathcal{O}(\varepsilon^2)\right)}{\varepsilon} \tag{42}$$

$$= \lim_{\varepsilon \to 0^+} \frac{-\varepsilon \mathbf{a}(t+\varepsilon)\frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}(t)} + \mathcal{O}(\varepsilon^2)}{\varepsilon} \tag{43}$$

$$= \lim_{\varepsilon \to 0^+} -\mathbf{a}(t+\varepsilon)\frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}(t)} + \mathcal{O}(\varepsilon) \tag{44}$$

$$= -\mathbf{a}(t)\frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}(t)} \tag{45}$$

Figure 5: Original proof from paper (Appendix B).

Solving the differential equation backwards in time, from time t_N to t_0, similarly to a regular backpropagation, one obtains the gradients with respect to the hidden state at any time as:

$$\frac{dL}{d\mathbf{z}(t_N)} = \mathbf{a}(t_N) \tag{15}$$

$$\frac{dL}{d\mathbf{z}(t_0)} = \mathbf{a}(t_0) = \mathbf{a}(t_N) + \int_{t_N}^{t_0} \frac{d\mathbf{a}(t)}{dt} dt = \mathbf{a}(t_N) - \int_{t_N}^{t_0} \mathbf{a}(t)\frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}(t)} dt \tag{16}$$

One can simply integrate backward in order to get the gradient at all different timesteps.
To find the initial value for the adjoint differential equation, the chain rule starting from the loss function is used, therefore also the gradients with respect to t_0, t_N and θ are needed. At the start , t and θ are considered as states with constant time derivatives such that:

$$\frac{dt(t)}{dt} = 1 \tag{17}$$

$$\frac{d\theta(t)}{dt} = 0 \tag{18}$$

and they can be combined with $z(t)$ to form the augmented state

$$f_{aug}(\begin{bmatrix}\mathbf{z} & \theta & t\end{bmatrix}) = \frac{d}{dt}\begin{bmatrix}\mathbf{z} \\ \theta \\ t\end{bmatrix} = \begin{bmatrix}f(\mathbf{z}(t), t, \theta) \\ 0 \\ 1\end{bmatrix} \tag{19}$$

and the augmented adjoint state

$$\mathbf{a}_{aug}(t) = \begin{bmatrix}\mathbf{a} \\ \mathbf{a}_\theta \\ \mathbf{a}_t\end{bmatrix}(t) \tag{20}$$

$$\mathbf{a}_\theta(t) = \frac{dL}{d\theta(t)} \tag{21}$$

$$\mathbf{a}_t(t) = \frac{dL}{dt(t)} \tag{22}$$

obtaining the ODE

$$\frac{\mathbf{a}_{aug}(t)}{dt} = -\begin{bmatrix}\mathbf{a}(t) & \mathbf{a}_\theta(t) & \mathbf{a}_t(t)\end{bmatrix}\frac{\partial f_{aug}(t)}{\partial\begin{bmatrix}\mathbf{z} & \theta & t\end{bmatrix}} \tag{23}$$

where $\frac{\partial f_{aug}(t)}{\partial\begin{bmatrix}\mathbf{z} & \theta & t\end{bmatrix}}$ is the Jacobian of the augmented state

$$\frac{\partial f_{aug}(t)}{\partial\begin{bmatrix}\mathbf{z} & \theta & t\end{bmatrix}} = \begin{bmatrix}\frac{\partial f}{\partial \mathbf{z}} & \frac{\partial f}{\partial \theta} & \frac{\partial f}{\partial t} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0}\end{bmatrix}(t) \tag{24}$$

The gradient with respect to the model parameters $\frac{dL}{d\theta}$ can be obtained setting $\mathbf{a}(t_N) = \mathbf{0}$ and integrating the second element of (23) backward in time. One obtains:

$$\frac{dL}{d\theta} = \mathbf{a}_\theta(t_0) = -\int_{t_N}^{t_0}\mathbf{a}(t)\frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial\theta}dt \tag{25}$$

At the end, the gradients at time instant t_0 and t_n are given by:

$$\frac{dL}{dt_N} = \mathbf{a_t}(t_N) = \frac{dL}{d\mathbf{z}(t_N)}\frac{d\mathbf{z}(t_N)}{dt_N} = \mathbf{a}(t_N)f(\mathbf{z}(t_N), t_N, \theta) \tag{26}$$

$$\frac{dL}{dt_0} = \mathbf{a_t}(t_N) - \int_{t_N}^{t_0}\mathbf{a}(t)\frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial t}dt \tag{27}$$

Therefore all the gradients needed to backpropagate through the ODE solver are computed by solving the ODE for the augmented state backward in time.

4 Possible applications

4.1 System Identification

In this section, the problem of system identification with NNs is formalized using NeuralODE. In the first part, it is given an overview of the problem and is presented the dynamical model used for the training. At the end some additional considerations will be formulated also based on the results obtained

4.1.1 Learning Dynamical Systems

Since ODENet approximates the derivative of a system, it can be used as a tool for modeling dynamical systems. In fact, it is possible to learn the underlying dynamics of the system by integrating the model between any two data points in the data set and then backpropagating to update the model parameters.
As the model is continuously defined, it can be integrated between any two time points, avoiding the problem of modeling data with irregular time steps.
One of the applications proposed by the authors regards the time-series modeling through ODEs having irregularly sampled data, in particular ill-defined data, problems with missing data in some time intervals, or simply inaccurate latent variables.
Some approaches concatenate time information to the input of an RNN, but these aren't able to solve the problem at all.
A solution to this, based on the ODENet module, is a continuous-time generative model, which, given an initial state $\mathbf{z}(t_0)$ and observation times $t_0...t_N$, computes the latent states $\mathbf{z}(t_1)...\mathbf{z}(t_N)$ and the outputs $\mathbf{x}(t_1)...\mathbf{x}(t_N)$:

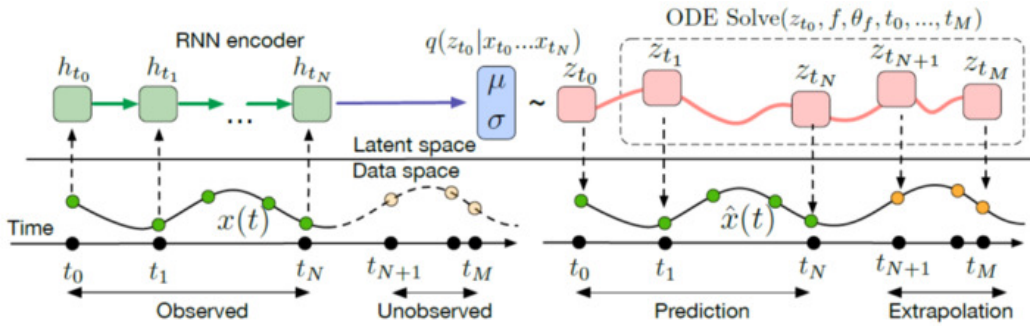
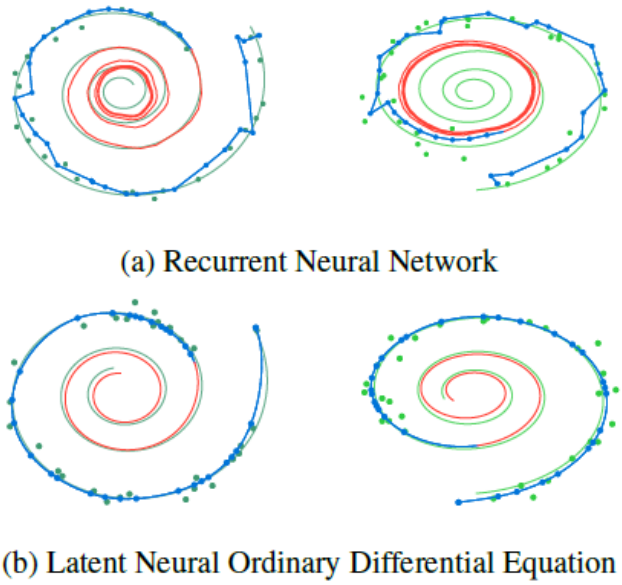


Figure 6: Computation graph of the latent ODE model.

The function f (a neural network function) is responsible for calculating the latent state $\mathbf{z}$ at any time t beginning at the current timestep. The model is a variational autoencoder that encodes the past trajectory (green in the figure above) in the initial latent state $\mathbf{z}(t_0)$ using an RNN.
The latent state distribution is hereby captured through the parameters of the distribution (a Gaussian with mean μ and standard deviation σ). From this distribution, a sample is taken and passed through ODENet.
The architecture has been tested on a dataset of bi-directional two-dimensional spirals, sampled at irregular time points with the addition of Gaussian noise. The following figure shows qualitatively the better performance of the Neural ODE:



(a) Recurrent Neural Network

(b) Latent Neural Ordinary Differential Equation

Figure 7: Dots are sampled noisy trajectories, the blue line is the true trajectory, the orange line stands for recovered and interpolated trajectory.

4.1.2 Training Neural ODEs

For sake of completeness, in order to see how NeuralODEs behave, a simple neural network with neural ordinary differential equations (ODEs) to learn the dynamics of a physical system has been trained.
The program is developed in python in order to make use of *PyTorch* Library, a powerful library for developing and evaluating deep learning models. It is an open source machine learning library based on the Torch library, used for applications such as computer vision and

natural language processing and allows to define and train neural network models in just a few lines of code.
Colab is used as a development environment. It allows to write and execute arbitrary python code through the browser, and it is especially well suited to machine learning and data analysis. More technically, Colab is a hosted Jupyter notebook service that requires no setup to use, while providing free access to computing resources including GPUs.
The first example aims to learn the dynamics z of a given physical system $z' = Az$ (a simple spiral function), where A is a 2×2 matrix.

$$A = \begin{bmatrix} -0.1 & -1 \\ 1 & -0.1 \end{bmatrix} \tag{28}$$

To test this, the evolution of the ODE is analyzed and sampled on its trajectory, and then restored.
After a training phase of 500 iterations, the final good results are the followings:

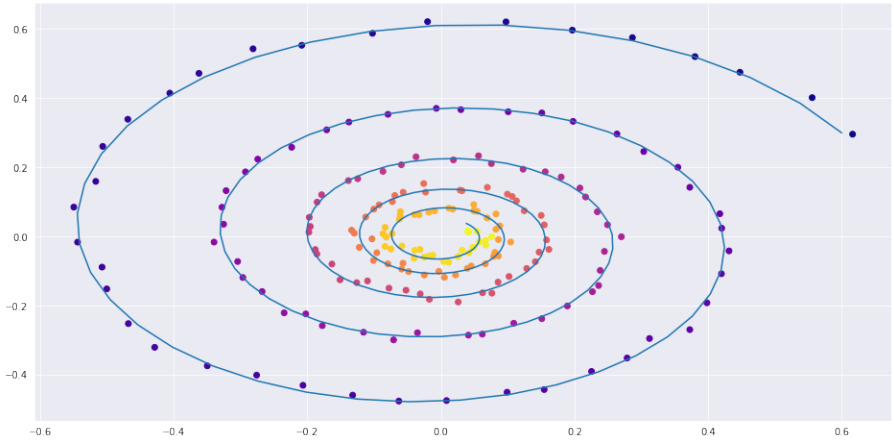


Figure 8: Network trained on a Spiral ODE.

And after at least 1500 iterations, the result obtained for a more sophisticated dynamics are:

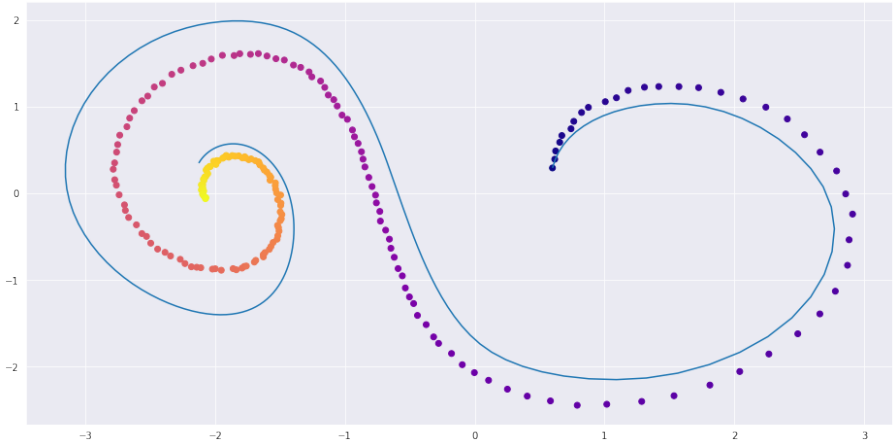


Figure 9: Network trained on a more complex dynamics.

4.1.3 Limitations of Neural ODE

If the map which is going to be modeled cannot be described by a vector field, i.e. the data does not represent a continuous transformation, and since a Neural ODE is a continuous transformation that cannot lift data into a higher dimension, it would try to smush around the input

data to a point where it is mostly separated. However, this wrong approach often leads to the network learning overly complicated transformations. Dupont et al. [2] introduced the concept of Augmented Neural ODEs, providing a few examples of intractable data for a Neural ODE, and tried to point out a solution to the problem, that is the increase of the dimensionality of the data.
The way to encode this into the Neural ODE architecture is to increase the dimensionality of the space the ODE is solved in, e.g. if the hidden state is a vector in $\mathbb{R}^n$, it is possible to add m more dimensions and solve the ODE in $\mathbb{R}^{(n+m)}$, in such a way that the augmented ODE becomes:

$$\frac{d}{dt}\begin{bmatrix} h(t) \\ a(t) \end{bmatrix} = f\left(\begin{bmatrix} h(t) \\ a(t) \end{bmatrix}, t, \theta\right) \tag{29}$$

$$\begin{bmatrix} h(0) \\ a(0) \end{bmatrix} = \begin{bmatrix} x \\ 0 \end{bmatrix} \tag{30}$$

Therefore there is a simple concatenation of a vector of zeros to the end of each datapoint x, allowing the network to learn some values for the extra dimensions. The data can be arranged in a linearly separable form with extra freedom, and the extra dimensions can be ignored when using the network. In this way, instead of learning a complicated map in a given dimension, the augmented Neural ODE learns a simpler map with a higher dimension. This permits the validation error to be almost null, against the results of the standard Neural ODE. A drawback of this solution is the introduction of more parameters, compensated by the degree of freedom that can lead to more feasible solutions.
Another criticism is that adding dimensions reduces the interpretability and elegance of the Neural ODE architecture. Extra dimensions may be unnecessary and may influence a model away from physical interpretability. Therefore, augmenting the hidden state is not always the best idea, and augmented Neural ODEs are useful only in isolated situations. Practically, Neural ODEs should be used for areas in which a smooth transformation increases interpretability and results, potentially areas like physics and irregular time series data.

4.2 ECG Heartbeat Classification

4.2.1 Problem Formulation

In this section, the problem of classification of electrocardiogram signal is formalized using both ResNet and NeuralODE. In the first part it is given an overview of the problem and are presented the data on which the networks are trained and tested. At the end are shown the structures of both networks.

4.2.2 Data Description

Electrocardiogram (ECG) records the electrical signals of the heart over a period of time. It is used to detect heart problems and also to monitor the heart's health. Moreover is an important step toward identifying arrhythmias. In this work, we want to classify the ECG measured from a certain number of patients into four classes by using NNs. For the training and test phase, we used the MIT Beath Israel Hospital (BIH) electrocardiogram dataset. This dataset contains about 110,000 labeled data points. Each sample can belong to four possible classes; normal (0), characterized by a sharp dropoff at the beginning followed by a small peak and then a narrow spike before flatline, super-ventricular premature beat (1), characterized by the fact that there is more activity before the main peak and after the main spike there is another bump, premature ventricular contraction (2), characterized by the fact that a lot of stuff going on towards the

front with short main peak, a fusion of ventricular and normal beat (3), characterized by the fact that everything compressed towards the front of the beat. and finally, unclassifiable beat (4), where each sample seems to deviate from the next significantly which makes sense since these did not fit in any other category and are present a lot of jagged peaks. These classes are shown respectively from figure 11 to figure 15. The recordings were digitalized at 360 samples per second per channel with 11-bit resolution over a 10 mV range. Each sample is taken over 0.52 seconds since there are 187 measurements per sample. Thus, the dataset has 188 columns where the last column is the class. In figure 10 are show the first 5 rows and first 14 columns of the test set. In the table 1 is reported the distribution of the classes in the training and test set where it is clear that the dataset is not balanced in fact, the normal class is prevalent with 0.828% of sample and the classifiable beat class with just 0.007% of samples.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	1.000000	0.758264	0.111570	0.000000	0.080579	0.078512	0.066116	0.049587	0.047521	0.035124	0.030992	0.028926	0.035124	0.026860	0.039256
1	0.908425	0.783883	0.531136	0.362637	0.366300	0.344322	0.333333	0.307692	0.296703	0.300366	0.304029	0.336996	0.377289	0.391941	0.439560
2	0.730088	0.212389	0.000000	0.119469	0.101770	0.101770	0.110619	0.123894	0.115044	0.132743	0.106195	0.141593	0.128319	0.150442	0.132743
3	1.000000	0.910417	0.681250	0.472917	0.229167	0.068750	0.000000	0.004167	0.014583	0.054167	0.102083	0.122917	0.150000	0.168750	0.172917
4	0.570470	0.399329	0.238255	0.147651	0.000000	0.003356	0.040268	0.080537	0.070470	0.090604	0.080537	0.104027	0.093960	0.117450	0.097315

5 rows × 188 columns

Figure 10: First 5 rows and first 14 columns of the test set

Class	Training set [#]	%	Test set [#]	%
N	72471	0.828	18118	0.828
S	6431	0.073	1608	0.073
P	5788	0.066	1448	0.066
F	2223	0.025	556	0.025
U	641	0.007	162	0.007
Total	87554		21892	

Table 1: Class distribution of the training set and the test set

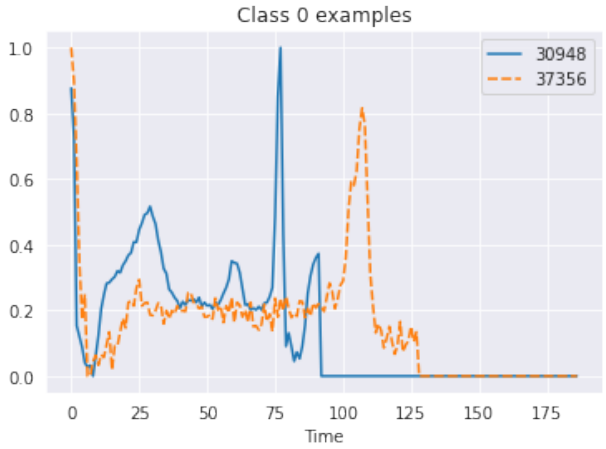


Figure 11: Signal belong the class of Normal heartbeats

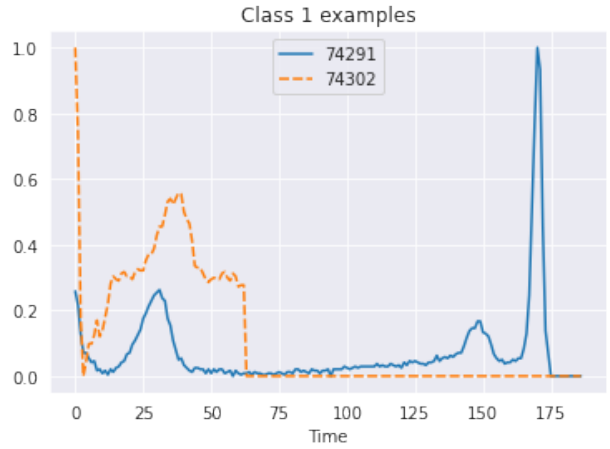


Figure 12: Signal belong the class of Supraventricular premature beat

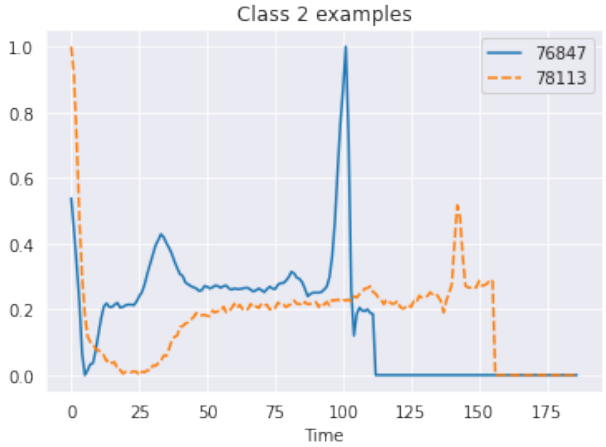


Figure 13: Signal belong the class of Premature ventricular contraction

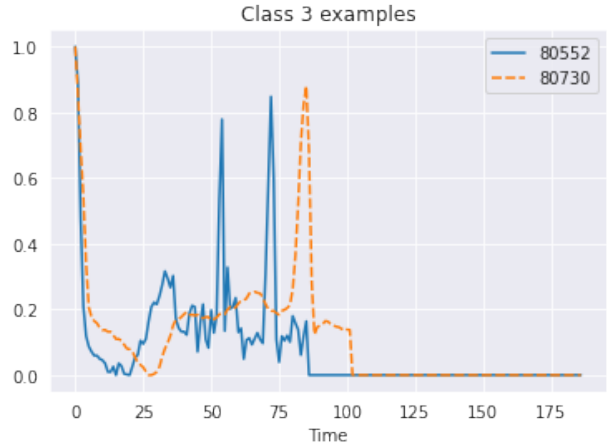


Figure 14: Signal belong the class of Fusion of ventricular and normal beat

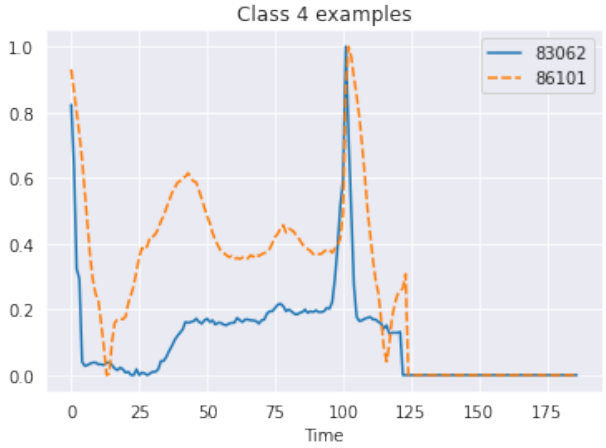


Figure 15: Unclassifiable beats

4.2.3 Networks Structure

The first network implemented is the ResNet. This network is composed of 17 main layers, in which the first 6 layers are essentially a simple CNN. After the 6th layers begin the true ResNet, which is composed of 6 residual blocks. Each residual block is composed of two convolutional layers all of the same size. In the end, the last 5 layers are necessary to generate the output.

In figure 16 it is shown the summary of the layers for the ResNet. The total number of tunable parameters is 182853. The second network that we implemented is the NeuralODE. This network is composed of 12 layers. The first 6 layers are equal to the previous network, in fact, they are necessary to extract important features from the input data. Then there is the ODE layer, that as mentioned in section 3, has only one shared layer. In the end, the remaining layers are necessary to generate the output and are equal to the last group of layers of the ResNet. The reason why we chose the beginning and the end of the network equal for both networks, is because we want to compare them and in this way, we can perform a fair comparison. In this case, the tunable parameters are remarkably reduced with respect to the ResNet, there are only 59333 tunable parameters, and this is the big advantage of this. In figure 17 it is shown the summary of the layers for the NeuralODE.

```
Sequential(
 (0): Conv1d(1, 64, kernel_size=(3,), stride=(1,))
 (1): GroupNorm(32, 64, eps=1e-05, affine=True)
 (2): ReLU(inplace=True)
 (3): Conv1d(64, 64, kernel_size=(4,), stride=(2,), padding=(1,))
 (4): GroupNorm(32, 64, eps=1e-05, affine=True)
 (5): ReLU(inplace=True)
 (6): Conv1d(64, 64, kernel_size=(4,), stride=(2,), padding=(1,))
 (7): ResBlock(
  (gn1): GroupNorm(32, 64, eps=1e-05, affine=True)
  (conv1): Conv1d(64, 64, kernel_size=(3,), stride=(1,), padding=(1,), bias=False)
  (gn2): GroupNorm(32, 64, eps=1e-05, affine=True)
  (conv2): Conv1d(64, 64, kernel_size=(3,), stride=(1,), padding=(1,), bias=False)
  (relu): ReLU(inplace=True)
 )
 (8): ResBlock(
  (gn1): GroupNorm(32, 64, eps=1e-05, affine=True)
  (conv1): Conv1d(64, 64, kernel_size=(3,), stride=(1,), padding=(1,), bias=False)
  (gn2): GroupNorm(32, 64, eps=1e-05, affine=True)
  (conv2): Conv1d(64, 64, kernel_size=(3,), stride=(1,), padding=(1,), bias=False)
  (relu): ReLU(inplace=True)
 )
 (9): ResBlock(
  (gn1): GroupNorm(32, 64, eps=1e-05, affine=True)
  (conv1): Conv1d(64, 64, kernel_size=(3,), stride=(1,), padding=(1,), bias=False)
  (gn2): GroupNorm(32, 64, eps=1e-05, affine=True)
  (conv2): Conv1d(64, 64, kernel_size=(3,), stride=(1,), padding=(1,), bias=False)
  (relu): ReLU(inplace=True)
 )
 (10): ResBlock(
  (gn1): GroupNorm(32, 64, eps=1e-05, affine=True)
  (conv1): Conv1d(64, 64, kernel_size=(3,), stride=(1,), padding=(1,), bias=False)
  (gn2): GroupNorm(32, 64, eps=1e-05, affine=True)
  (conv2): Conv1d(64, 64, kernel_size=(3,), stride=(1,), padding=(1,), bias=False)
  (relu): ReLU(inplace=True)
 )
 (11): ResBlock(
  (gn1): GroupNorm(32, 64, eps=1e-05, affine=True)
  (conv1): Conv1d(64, 64, kernel_size=(3,), stride=(1,), padding=(1,), bias=False)
  (gn2): GroupNorm(32, 64, eps=1e-05, affine=True)
  (conv2): Conv1d(64, 64, kernel_size=(3,), stride=(1,), padding=(1,), bias=False)
  (relu): ReLU(inplace=True)
 )
 (12): ResBlock(
  (gn1): GroupNorm(32, 64, eps=1e-05, affine=True)
  (conv1): Conv1d(64, 64, kernel_size=(3,), stride=(1,), padding=(1,), bias=False)
  (gn2): GroupNorm(32, 64, eps=1e-05, affine=True)
  (conv2): Conv1d(64, 64, kernel_size=(3,), stride=(1,), padding=(1,), bias=False)
  (relu): ReLU(inplace=True)
 )
 (13): GroupNorm(32, 64, eps=1e-05, affine=True)
 (14): ReLU(inplace=True)
 (15): AdaptiveAvgPool1d(output_size=1)
 (16): Flatten()
 (17): Linear(in_features=64, out_features=5, bias=True)
)
```

Figure 16: ResNet structure.

```
Sequential(
  (0): Conv1d(1, 64, kernel_size=(3,), stride=(1,))
  (1): GroupNorm(32, 64, eps=1e-05, affine=True)
  (2): ReLU(inplace=True)
  (3): Conv1d(64, 64, kernel_size=(4,), stride=(2,), padding=(1,))
  (4): GroupNorm(32, 64, eps=1e-05, affine=True)
  (5): ReLU(inplace=True)
  (6): Conv1d(64, 64, kernel_size=(4,), stride=(2,), padding=(1,))
  (7): ODENet(
    (odefunc): ODEfunc(
      (norm1): GroupNorm(32, 64, eps=1e-05, affine=True)
      (relu): ReLU(inplace=True)
      (conv1): ConcatConv1d(
        (_layer): Conv1d(65, 64, kernel_size=(3,), stride=(1,), padding=(1,))
      )
      (norm2): GroupNorm(32, 64, eps=1e-05, affine=True)
      (conv2): ConcatConv1d(
        (_layer): Conv1d(65, 64, kernel_size=(3,), stride=(1,), padding=(1,))
      )
      (norm3): GroupNorm(32, 64, eps=1e-05, affine=True)
    )
  )
  (8): GroupNorm(32, 64, eps=1e-05, affine=True)
  (9): ReLU(inplace=True)
  (10): AdaptiveAvgPool1d(output_size=1)
  (11): Flatten()
  (12): Linear(in_features=64, out_features=5, bias=True)
)
```

Figure 17: ODENet structure.

4.2.4 Simulation and Results

Both models have been trained on the ECG training dataset for five epochs, with a batch size of 128. We studied the training models on the PyTorch F.Cross_entropy loss function and used the PyTorch stochastic gradient descent algorithm as optimizer. Both the ResNet and the ODENet performed well on the test set and can confidently generalize. The former has a significantly higher accuracy than the baseline. In particular the accuracy on the test set was above 97%. The ResNet has been trained for only 10 minutes with Google Colaboratory (at least one hour in local) while the ODENet for over 70 minutes with Google Colaboratory (at least seven hours in local). However, a benefit of the ODENet can be seen in the number of tunamble parameters. It has almost exactly 1/3 of the parameters as the ResNet and yet performed slightly better. This leads to the fact that Neural ODEs use constant memory (although with a high memory overhead due to the adjoint method).

5 Conclusion

Neural Ordinary Differential Equations are able to parametrize the ODE which describes the dynamical system or inputs of other nature and produce a very accurate approximation of the resulting derivative.
For sake of completeness the accuracy, speed, and memory of the Neural ODE has been compared to the ones of the state-of-the-art neural network ResNet, from which the first one comes from.
Overall, the ResNet and Neural ODE architectures perform nearly identically on the testing dataset. The main tradeoff is between speed and memory. The ResNet is significantly faster while the ODENet uses significantly fewer tunable parameters. The tradeoff is expected given the findings in the original Neural ODE paper.
Chen et al. presented an innovative and interesting approach to thinking about neural networks but it is not already proven that it would be effective for example for each kind of dynamical

system.
Neural ODE can help to build continuous-time time series models, which can easily handle data coming at irregular intervals. However, ODEs can only model deterministic dynamics.
In the case of data sampled at regular time intervals (like video or audio), there would not be any advantage with respect to other standard approaches which can be simpler and faster in that situation.
Anyway, there are some other considerations to be done about the architecture analyzed.
The first one regards mini-batching, in fact, it might be an issue with the approach, since batching together multiple samples requires solving a system of ODEs at once. This can multiply the number of required evaluations significantly.
The other is that the uniqueness of the ODE solution is only guaranteed if the network has finite weights and uses Lipschitz nonlinearities such as *tanh* or *relu*. In addition, conventional nets can be evaluated exactly with a fixed amount of computation, and are typically faster to train, without an error tolerance for the solver to be considered.
However, an important advantage of the method presented in the paper is that one may choose the balance between a fast and a precise solution by affecting the precision of the numerical integration during training and evaluation phases, this leads to a constant memory cost at training time and an adaptive time cost. The last aspect is due to the approximation of the solution of the ODE, in fact, sometimes only a few iterations of the solver are needed to get an acceptably good answer, and this could save time.
In conclusion, despite the numerous defects of various kinds, the most important aspect is that this method, this architecture, is generally applicable (requiring only that the nonlinearities of the neural network be Lipschitz-continuous) and may be applied to time-series modeling, supervised learning, density estimation or other sequential processes, including all system identification tasks.

References

[1] Chen, Ricky T. Q., Yulia Rubanova, Jesse Bettencourt, David Duvenaud
Neural Ordinary Differential Equations
19 June 2018

[2] Emilien Dupont, Arnaud Doucet, Yee Whye Teh
Augmented Neural ODEs
2 April 2019