Understanding the Neural ODE Implementation for Times Series Analysis

Davud Topalović

August 17, 2023

1 Introduction

In recent years, the field of deep learning has undergone rapid evolution, demonstrating its potential to revolutionize various domains. Among the latest advancements, Neural Ordinary Differential Equations (Neural ODEs) stand out as a compelling approach that marries the principles of differential equations and neural networks. This fusion has led to the development of a powerful paradigm that can capture complex dynamics, model continuous-time phenomena, and handle irregularly sampled data with remarkable precision.

In this paper, we delve into the realm of Neural ODEs, aiming to provide a comprehensive understanding of their theoretical foundations, practical implementation and applications. Our primary objective is to bridge the gap between theoretical concepts and practical implementation, enabling researchers and practitioners to leverage Neural ODEs effectively in various domains.

2 Euler's Method

Ordinary Differential Equations (ODEs) play a pivotal role in describing dynamic behaviors across diverse scientific and engineering domains. They offer insights into how variables evolve over time. Euler's Method is a fundamental numerical technique used to approximate solutions to ODEs when obtaining analytical solutions proves challenging or unfeasible. In this section, we'll explore the basics of Euler's Method.

Conceptual Overview

Consider a first-order ODE given as $\frac{dy}{dx} = f(x, y)$, where $f(x, y)$ represents a function of x and y. Euler's Method discretizes the problem by breaking it into small steps, estimating the value of y at discrete points. The method begins at an initial point (x_0, y_0) and iteratively calculates the next y value using the derivative $f(x, y)$ and a step size h:

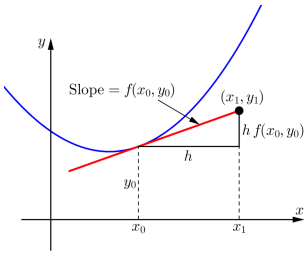
$$y_{n+1} = y_n + h \cdot f(x_n, y_n) \tag{1}$$

In this equation, x_n and y_n are the current values, and y_{n+1} is the next value of y after taking a step of size h along the x-axis.

Implementation Steps

1. **Initial Conditions:** Choose initial values x_0 and y_0.
2. **Step Size Selection:** Determine a small step size h. A smaller h improves accuracy but increases computation time.

3. **Iteration:** Start with the initial values (x_0, y_0) and iterate through the following steps:
   (a) Calculate the slope $f(x_n, y_n)$ using the current values x_n and y_n.
   (b) Compute the next value of y using the formula $y_{n+1} = y_n + h \cdot f(x_n, y_n)$.
   (c) Update x by h $(x_{n+1} = x_n + h)$.
   (d) Repeat the steps until you reach the desired point or time.




```python
def ode_solve(z0, t0, t1, f):
    """Euler ODE initial value solver"""

    h_max = 0.05
    n_steps = math.ceil((abs(t1 - t0)/h_max).max().item())

    h = (t1 - t0)/n_steps
    t = t0
    z = z0

    for i_step in range(n_steps):
        z = z + h * f(z, t)
        t = t + h
    return z
```

Figure 1: Implementation of Euler's Method The University of Queensland 2023

3 From ResNet to Neural ODE

The inspiration behind incorporating differential equations into a neural network framework comes from the remarkable achievements of residual neural networks (ResNet). We are well familiar with the issue of the vanishing gradient in deep neural networks. To improve the performance of the neural network, we might increase the number of layers. However, by doing that, gradients of the loss function L with respect to earlier layers become exceedingly small, which makes it difficult to effectively update the parameters. Residual networks overcome this issue by incorporating shortcut connections across certain layers. These connections allow the neural network to preserve the context of the input, thus preventing the gradient from vanishing.

This architectural innovation enables each layer to be conceptualized as a distinct transformation, where the state at layer $t+1$ is derived by adding a transformed residual $f(h_t, \theta_t)$ to the previous state h_t:

$$h_{t+1} = h_t + f(h_t, \theta_t) \tag{2}$$

where h_t is hidden state at layer t, f is a dimension preserving function and θ_t is vector of parameters at layer t. This formulation can also be interpreted as an Euler discretization of a continuous ordinary differential equation. In order to intuitively explain this let's rearrange the equation 2:

$$\begin{aligned} h_{t+1} &= f(h_t, \theta_t) + h_t \\ h_{t+1} - h_t &= f(h_t, \theta_t) \\ \frac{h_{t+1} - h_t}{1} &= f(h_t, \theta_t) \\ \left.\frac{h_{t+\Delta} - h_t}{\Delta}\right|_{\Delta=1} &= f(h_t, \theta_t) \end{aligned} \tag{3}$$

Now, if we multiply the above equation with Δ we get:

$$h_{t+\Delta} = h_t + \Delta f(h_t, \theta_t) \tag{4}$$

This very much reminds of the Euler's discretization method to approximate some continuous function, where Δ is the step size and f is the dynamics of the system (see 1). This inspires us to come back to equation 3 and push Δ to zero:

$$\lim_{\Delta \to 0} \frac{h_{t+\Delta} - h_t}{\Delta} = f(h_t, t, \theta)$$

Strictly mathematically speaking, on the left side, we have the definition of the derivative of the function (state) h with respect to time. However, intuitively, what we have achieved is pushing the gap between the layers to zero, thereby forming one infinitely large layer defined by the function f. By doing so, the function f now takes time as an argument and only one set of parameters θ (since we have only one layer) (see Figure 2). Since h is a discrete state, we can rewrite this and arrive at a formulation of continuous hidden state dynamics, where each layer h is a discretized evaluation of a continuous function z at time t:

$$\frac{dz(t)}{dt} = f(z(t), t, \theta) \tag{5}$$

So we have transformed the neural network into an initial value ODE problem, with its solution representing the function f that maps the input to the output across a specific time span [1]. Our network network, i.e. the infinite layer characterized with only one θ or the function f, now represents the dynamics of a continuous function z.

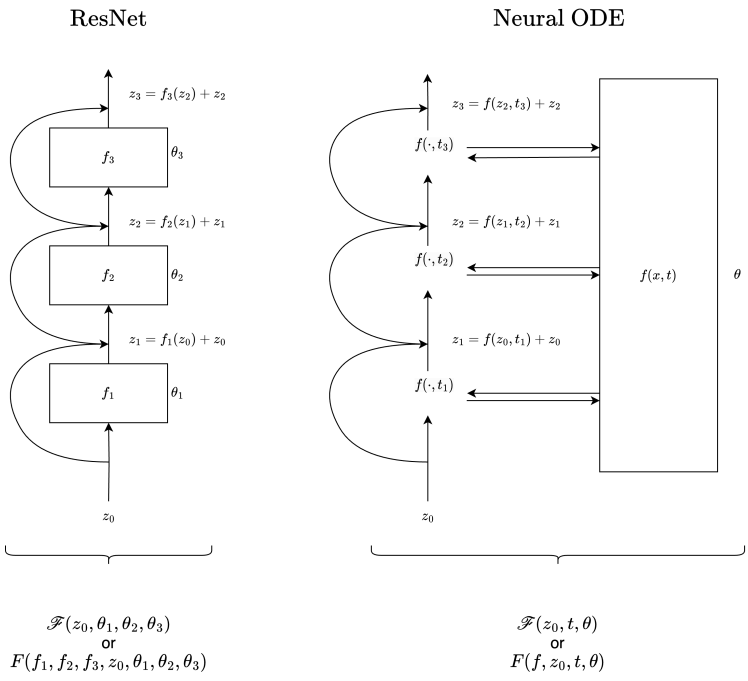


Figure 2: Transformation from ResNet to Neural ODE. Drozdyuk 2023

[1] f is solution since neural network is learning the parameters of its layers, in this case parameters of one layer defined with function $f(z, \theta, t)$.

4 Neural ODE Set-Up for Time Series Analysis

An intuitive way to understand the architecture of Neural ODEs is to look at them as a neural network with single infinitely large layer characterized with one set of parameters θ and function f that essentially represents the dynamics of the input (direction of change for z). Instead of different weights and biases that define each at layer, now our single layer takes t as an input, meaning that for different t we get different activation of the input. As such, we are interested in optimizing f, that is optimizing how the input maps to the output in the time span of the model. In other words, given the task of modeling the mapping of z_0 to z_1 in the space R^n, we are interested in optimizing the derivative such that the solution to the initial value problem:

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

will accurately predict z_1 at time t_1. The set-up of a neural ODE therefore requires three basic elements:

- derivative model f to compute the dynamics at a given time,
- set of parameters θ to calculate such a model and
- time span $[t_0, t_1]$ to evaluate the network

The continuous analog to matrix multiplication and linear algebra evaluating traditional neural networks is then to integrate the derivative model over its time span:

$$z(t_1) = z(t_0) + \int_{t_0}^{t_1} f(z, \theta, t)dt \tag{6}$$

This can be done fairly efficiently using modern ODE solvers, with the benefit that many such solvers are already available across different systems and extremely well-tested (Euler's method, Runge-Kutta 4th Order method, Dormand–Prince method, etc.). Given an input, we simply make a call to the ODE solver to evaluate the integral, plugging in the necessary initial value, derivative function, parameters, and the time span:

$$z(t_1) = \text{ODESolve}(z(t_0), f, \theta(t), t_0, t_1) \tag{7}$$

Also, similar to the standard neural network approach, the ultimate objective is for $z(t_1)$ to get as close to the desired labeled output z_1 as possible. Thus, we also need a loss function to assess the performance of neural ODEs at each iteration. We define an arbitrary loss function L that takes in the integral output at time t_1:

$$\begin{aligned} L(z(t_1)) &= L\left(z(t_0)) + \int_{t_0}^{t_1} f(z(t), \theta(t), t)dt\right) \\ &= L\left(ODESolve(z(t_0), f, \theta(t), t_0, t_1)\right) \end{aligned} \tag{8}$$

In the case of time series, we have observation data that follows certain dynamics and in order to learn it, our Neural ODE Solver takes as an input: initial observation - $z(t_0)$, randomly initialized parameters - θ and time steps for each observation - $t \in [0, .., T]$. By iterating an ODESolver through every time step we are able to obtain predictions $\hat{z} \in [\hat{z_1}, \hat{z_2}, ..., \hat{z_T}]$ and

calculate the loss (see Figure 3). In order to optimize θ we need to calculate $\frac{dL}{d\theta}$ and this is something that will require special approach that will be discussed in the upcoming section.

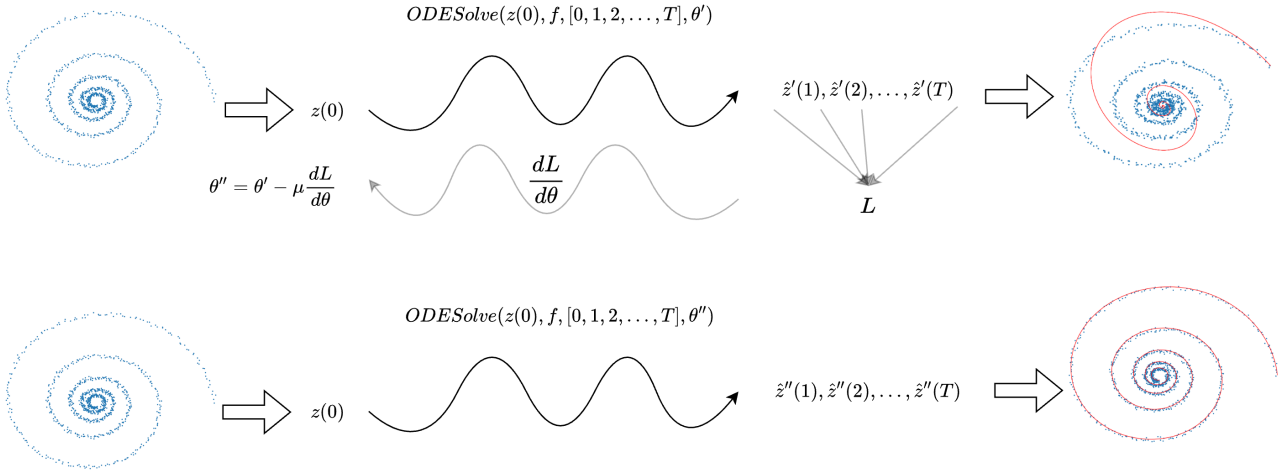


Figure 3: Neural ODE application for fitting time series data and thus optimizing its dynamics.

5 Adjoint Sensitivity Method

Similar to a traditional neural network, our aim is to optimize the parameters of the neural ODE to minimize the loss function. However, there are significant challenges that emerge when attempting to perform backpropagation through an ODE solver. To do it in traditional way this process would be complex and highly inefficient due to two key factors:

- Calculating the gradient of the loss function wrt θ is unclear without an explicit analytical form of z in terms of θ and t. In other words, we are unable to calculate $\frac{dL}{d\theta}$ because we don't know $z(\theta, t)$.
- In many ODE solvers, we are required to specify the number of function evaluations (NFE). NFE represents the count of times the mathematical function representing the differential equation is computed during the ODE solving process. Consider Euler's method, as mentioned in Section 2. Notably, NFE is inversely related to the step size (h or Δ), where a smaller step size results in a more accurate solution. Now, let's imagine a scenario where our NFE reaches 10^5 in total [2]. This implies taking a 10^5 small time steps to achieve a highly accurate solution. Analogously, this situation is similar to having a ResNet model with 10^5 layers. This would effectively mean engaging in backpropagation across an enormous number of layers Choi 2023.

Mentioned issues serve as the motivation and reason to adopt the so called **Adjoint Sensitivity Method** - method that enables gradient calculation without requiring explicit solutions. As such, the adjoint method is only a computationally cheap and fast way to backpropagate through specific classes of neural ODEs.

Now we proceed to analyzing the adjoint sensitivity method (Nguyen and Malinsky 2020). Our aim is to find the dynamics of the system z, that is the parameters θ that define the

[2] This is very realistic scenario as we can have 1000 times series observations with 100 time steps of ODESolver between each observation.

function $f(z(t), \theta, t)$. For that we need to obtain $\frac{dL}{d\theta}$. Considering that we know $z(0)$ and that loss function L only depends on the $z(t_1)$[3] we introduce a Lagrangian $\mathscr{L}$ function as follows:

$$\mathscr{L} = L(z_{t_1}) - \int_{t_0}^{t_1} \lambda \left(\frac{dz}{dt} - f(z, \theta, t) \right) dt \tag{9}$$

where t_0 and t_1 are two adjacent time stamps and λ is the Lagrange multiplier, an arbitrary row vector chosen at time t.

A clever choice of λ, however, will allow us to calculate $\frac{d\mathscr{L}}{d\theta}$ without solving for the explicit derivative of z with respect to θ. For convenience, the gradients of L are assumed to be row vectors. Let's first split the second term into two integrals:

$$\mathscr{L} = L(z_{t_1}) - \int_{t_0}^{t_1} \lambda \frac{dz}{dt} dt + \int_{t_0}^{t_1} \lambda f(z, \theta, t) dt \tag{10}$$

Now let's simplify $\mathscr{L}$ by integrating the second integral by parts:

$$\int_{t_0}^{t_1} \lambda \frac{dz}{dt} dt = \lambda z \Big|_{t_0}^{t_1} - \int_{t_0}^{t_1} \dot{\lambda} z dt \tag{11}$$

Substituting equation 11 to 10 we get:

$$\mathscr{L} = L(z_{t_1}) - \lambda(t_1) z_{t_1} + \lambda(t_0) z_{t_0} + \int_{t_0}^{t_1} \dot{\lambda} z dt + \int_{t_0}^{t_1} \lambda f(z, \theta, t) dt \tag{12}$$

Now, let's perform the total derivative of $\mathscr{L}$ with respect to θ, while keeping in mind that λ is an arbitrary vector at time t and z_{t_0} is the initial value, therefore do not depend on θ:

$$\begin{aligned} \frac{d\mathscr{L}}{d\theta} &= \frac{dL(z_{t_1})}{dz_{t_1}} \frac{dz_{t_1}}{d\theta} - \lambda(t_1) \frac{dz_{t_1}}{d\theta} + \int_{t_0}^{t_1} \dot{\lambda} \frac{dz}{d\theta} dt + \int_{t_0}^{t_1} \lambda \left(\frac{\partial f}{\partial z} \frac{dz}{d\theta} + \frac{\partial f}{\partial \theta} \right) dt \\ &= \left(\frac{dL(z_{t_1})}{dz_{t_1}} - \lambda(t_1) \right) \frac{dz_{t_1}}{d\theta} + \int_{t_0}^{t_1} \left(\dot{\lambda} + \lambda \frac{\partial f}{\partial z} \right) \frac{dz}{d\theta} dt + \int_{t_0}^{t_1} \lambda \frac{\partial f}{\partial \theta} dt \end{aligned} \tag{13}$$

Since we want to avoid calculating $\frac{dz}{d\theta}$, we want to search for such λ that will make the first and second term zero. Therefore, we come with an expression for λ:

$$\lambda(t_1) = \frac{dL(z_{t_1})}{dz_{t_1}} \tag{14}$$

$$\dot{\lambda} = -\lambda \frac{\partial f}{\partial z} \tag{15}$$

5.1 Lemma 1

Let us now define the adjoint state $a(t)$ as the solution to the initial value problem:

$$a(t_1) = \frac{dL(z_{t_1})}{dz_{t_1}}, \quad \frac{da(t)}{dt} = -a(t) \frac{\partial f}{\partial z}, \tag{16}$$

then $a(t) = \frac{dL}{dz_t}$, and the gradients of the L with respect to z_{t_0}, θ, t_0 can be computed by evaluating the initial value problem 16 at time t_0:

[3] Because we are considering only two time steps: t_0 and t_1.

$$a_\theta(t_0) = \frac{dL(z_{t_1})}{d\theta}\bigg|_{t_1}, \qquad a_\theta(t_1) = \mathbf{0}, \qquad \frac{da_\theta}{dt} = -a(t)\frac{\partial f}{\partial \theta} \tag{17}$$

$$a_t(t_0) = -\frac{dL(z_{t_1})}{dt_0}\bigg|_{t_1}, \qquad a_t(t_1) = -a(t_1)f(z_{t_1}, \theta, t_1), \qquad \frac{da_t}{dt} = -a(t)\frac{\partial f}{\partial t} \tag{18}$$

Before we jump into proving these equations let's just clarify why do we need gradients wrt to z_{t_0}, θ and t_0. These gradients are needed to compute how changes in these variables affect the final loss, and they allow us to update the model's parameters and initial conditions during the optimization process. So in order to train the model, that is to learn the dynamics of the system (essentialy learn the parameters θ) we have to compute: $a(t_0)$, $a_\theta(t_0)$ and $a_t(t_0)$. The efficient way in which we can "backpropagate"to t_0 and compute the gradients is to form an augmented form of our original state and augmented state of its corresponding adjoint state and work with those:

$$z_{aug} = \begin{bmatrix} z \\ \theta \\ t \end{bmatrix} \tag{19}$$

The dynamics of the z_{aug} is:

$$\frac{d}{dt}\begin{bmatrix} z \\ \theta \\ t \end{bmatrix}(t) = f_{aug}(\begin{bmatrix} z & \theta & t \end{bmatrix}) := \begin{bmatrix} f(z, \theta, t) \\ 0 \\ 1 \end{bmatrix} \tag{20}$$

By the definition from Lemma 5.1 the corresponding adjoint augmented state is therefore:

$$a_{aug} = \begin{bmatrix} a \\ a_\theta \\ a_t \end{bmatrix} = \begin{bmatrix} \frac{dL}{dz} \\ \frac{dL}{d\theta} \\ -\frac{dL}{dt} \end{bmatrix} \tag{21}$$

Now, since we need to compute a_{aug} at t_0 we need to know its dynamics. The dynamics of a, a_θ, a_t, and hence a_{aug} is computed as the dot product of $-a(t)$ and the Jacobian matrix of $f_{aug}(z, \theta, t)$:

$$\frac{da_{aug}}{dt} = \begin{bmatrix} -a\frac{\partial f}{\partial z} & -a\frac{\partial f}{\partial \theta} & -a\frac{\partial f}{\partial t} \end{bmatrix} \tag{22}$$

With only one call to the ODE solver on the augmented adjoints, we are then able to calculate all necessary gradients with respect to $z(t_0)$, θ, t_0, and t_1. This will be more clear after we prove the Lemma 5.1 and derive the equations.

Proof for 17

If we recall that $\mathscr{L} = L$ and by plugging the adjoint state 16 into equation 13 we obtain:

$$\left.\frac{dL}{d\theta}\right|_{t_1} = \frac{d\mathscr{L}}{d\theta} = \int_{t_0}^{t_1} \lambda \frac{\partial f}{\partial \theta}\, dt = \int_{t_0}^{t_1} a(t)\frac{\partial f}{\partial \theta}\, dt = \int_{t_1}^{t_0} -a(t)\frac{\partial f}{\partial \theta}\, dt = \int_{t_1}^{t_0} da_\theta = a_\theta(t_0) - a_\theta(t_1), \tag{23}$$

where $a_\theta(t)$ is a function of t such that $\frac{da_\theta}{dt} = -a(t)\frac{\partial f}{\partial \theta}$. To simplify the computation, we are free to assume that $a_\theta(t_1) = \mathbf{0}$. So, the solution of the ODE 17 at time t_0 is given with:

$$a_\theta(t_0) = \int_{t_1}^{t_0} \left(-a(t)\frac{\partial f}{\partial \theta}\right) dt \tag{24}$$

Proof for $a(t) = \frac{dL}{dz_t}$

Now, we would like to find the gradient of the L with respect to state z at some random point in time. The Lagrangian $\mathscr{L}$ at time t is defined as:

$$\mathscr{L} = L(z_{t_1}) - \int_t^{t_1} a(t')\left(\frac{dz}{dt'} - f(z, \theta, t')\right) dt' \tag{25}$$

Splitting the second term into two integrals and then integrating by parts the first one, we get:

$$\mathscr{L} = L(z_{t_1}) - a(t_1)z_{t_1} + a(t)z_t + \int_t^{t_1} \frac{da}{dt'} z dt' + \int_t^{t_1} a(t')f(z, \theta, t')dt' \tag{26}$$

Now let's differentiate $\mathscr{L}$ with respect to z_t:

$$\begin{aligned}
\left.\frac{dL(z_{t_1})}{dz_t}\right|_{t_1} &= \frac{d\mathscr{L}}{dz_t} \\
&= \frac{\mathrm{dL}}{dz_{t_1}}\frac{dz_{t_1}}{dz_t} - a(t_1)\frac{dz_{t_1}}{dz_t} + a(t) + \int_t^{t_1} \frac{da}{dt'}\frac{dz}{dz_t}dt' + \int_t^{t_1} a(t')\frac{\partial f}{\partial z}\frac{dz}{dz_t}dt' \\
&= \cancel{\left(\frac{\mathrm{dL}}{dz_{t_1}} - a(t_1)\right)}^{\mathbf{0}}\frac{dz_{t_1}}{dz_t} + a(t) + \int_t^{t_1} \cancel{\left(\frac{da}{dt'} + a(t')\frac{\partial f}{\partial z}\right)}^{\mathbf{0}}\frac{dz}{dz_t}dt' \\
&= a(t)
\end{aligned} \tag{27}$$

In case we assume 16, the adjoint state at time t is indeed the gradient of the L function with respect to the hidden state at time t. The gradient of the L with respect to z_0 is therefore simply $a(t_0)$. Since we eventually need gradient with respect z_0, that is $a(t_0)$, we calculate it by integrating the ODE in equation 16:

$$\frac{da(t)}{dt} = -a(t)\frac{\partial f}{\partial z} \quad \leftarrow \quad \int_{t_1}^{t_0} dt \tag{28}$$

$$a(t_0) = a(t_1) - \int_{t_1}^{t_0} a(t)\frac{\partial f}{\partial z}dt \tag{29}$$

$$a(t_0) = \left.\frac{dL(z_{t_1})}{dz_{t_1}}\right|_{t_1} + \int_{t_1}^{t_0} \left(-a(t)\frac{\partial f}{\partial z}\right) dt \tag{30}$$

Proof for 18

Let's calculate the gradient of the L with respect to time via the chain rule:

$$\left.\frac{dL(z_{t_1})}{dt}\right|_{t_1} = \left.\frac{dL(z_{t_1})}{dz_t}\right|_{t_1} \frac{dz_t}{dt} = a(t)f(z,\theta,t) \tag{31}$$

So, a_t at any time t is given with:

$$a_t(t) = -\left.\frac{dL(z_{t_1})}{dt}\right|_{t_1} = -a(t)f(z,\theta,t) \tag{32}$$

While equation 31 can be used to calculate the gradient with respect to a given time t, we will formulate it into an initial value problem. This allows us to make a single call to the ODE solver and calculate all necessary gradients at once. In that case let's see how $a_t(t)$ changes with time:

$$\frac{da_t(t)}{dt} = \frac{d}{dt}\left(-\left.\frac{dL(z_{t_1})}{dt}\right|_{t_1}\right) = \frac{d}{dt}\left(-a(t)f(z,\theta,t)\right) = -\frac{da(t)}{dt}f(z,\theta,t) - a(t)\left(\frac{\partial f}{\partial z}\frac{dz}{dt} + \frac{\partial f}{\partial t}\right) \tag{33}$$

$$= \cancel{a(t)\frac{\partial f}{\partial z}f(z,\theta,t)} - \cancel{a(t)\frac{\partial f}{\partial z}\frac{dz}{dt}} - a(t)\frac{\partial f}{\partial t} \tag{34}$$

$$= -a(t)\frac{\partial f}{\partial t} \tag{35}$$

Hence, we have proven that the 18 holds true. Since, eventually we need the gradient with respect to time t_0, we calculate it by integrating 35 from t_1 to t_0:

$$\frac{da_t(t)}{dt} = -a(t)\frac{\partial f}{\partial t} \quad \leftarrow \quad \int_{t_1}^{t_0} dt \tag{36}$$

$$a_t(t_0) = a_t(t_1) - \int_{t_1}^{t_0} a(t)\frac{\partial f}{\partial t}dt \tag{37}$$

$$a_t(t_0) = -a(t_1)f(z_{t_1},\theta,t_1) + \int_{t_1}^{t_0}\left(-a(t)\frac{\partial f}{\partial t}\right)dt \tag{38}$$

Proof for 22

Now we can summarize the necessary equations for backpropagation:

$$\frac{da(t)}{dt} = -a(t)\frac{\partial f}{\partial z} \quad \rightarrow \quad a(t_0) = \left.\frac{dL(z_{t_1})}{dz_{t_1}}\right|_{t_1} + \int_{t_1}^{t_0}\left(-a(t)\frac{\partial f}{\partial z}\right)dt \tag{39}$$

$$\frac{da_\theta}{dt} = -a(t)\frac{\partial f}{\partial \theta} \quad \rightarrow \quad a_\theta(t_0) = \int_{t_1}^{t_0}\left(-a(t)\frac{\partial f}{\partial \theta}\right)dt \tag{40}$$

$$\frac{da_t}{dt} = -a(t)\frac{\partial f}{\partial t} \quad \rightarrow \quad a_t(t_0) = -a(t_1)f(z_{t_1},\theta,t_1) + \int_{t_1}^{t_0}\left(-a(t)\frac{\partial f}{\partial t}\right)dt \tag{41}$$

By putting all the equations in this form, we can now more clearly see the utility of forming an augmented state. Since we recognize some kind of pattern in solving the above equations we simply augment all three equations and solve them in one ODESolve call. So, finally we get:

$$\frac{da_{aug}}{dt} = \frac{d}{dt}\begin{bmatrix} a \\ a_\theta \\ a_t \end{bmatrix} = -\begin{bmatrix} a(t)\frac{\partial f}{\partial z} \\ a(t)\frac{\partial f}{\partial \theta} \\ a(t)\frac{\partial f}{\partial t} \end{bmatrix} \int_{t_0}^{t_1} \rightarrow \begin{bmatrix} a(t_0) = a(t_1) + \int_{t_1}^{t_0} \left(-a(t)\frac{\partial f}{\partial z}\right) dt \\ a_\theta(t_0) = \int_{t_1}^{t_0} \left(-a(t)\frac{\partial f}{\partial \theta}\right) dt \\ a_t(t_0) = -a_t(t_1) + \int_{t_1}^{t_0} \left(-a(t)\frac{\partial f}{\partial t}\right) dt \end{bmatrix} \tag{42}$$

6 Algorithm

The core of the Neural ODEs is solving the adjoint augmented state backwards in time and thus obtaining the necessary gradients for optimization. Our implementation will follow the algorithm presented in the original Neural Differential Equations paper Chen et al. 2019.

Appendix C Full Adjoint sensitivities algorithm

This more detailed version of Algorithm 1 includes gradients with respect to the start and end times of integration.

Algorithm 2 Complete reverse-mode derivative of an ODE initial value problem

Input: dynamics parameters θ, start time t_0, stop time t_1, final state $\mathbf{z}(t_1)$, loss gradient $\partial L / \partial \mathbf{z}(t_1)$

$\frac{\partial L}{\partial t_1} = \frac{\partial L}{\partial \mathbf{z}(t_1)}^\mathsf{T} f(\mathbf{z}(t_1), t_1, \theta)$ ▷ Compute gradient w.r.t. t_1

$s_0 = [\mathbf{z}(t_1), \frac{\partial L}{\partial \mathbf{z}(t_1)}, \mathbf{0}_{|\theta|}, -\frac{\partial L}{\partial t_1}]$ ▷ Define initial augmented state

def aug_dynamics($[\mathbf{z}(t), \mathbf{a}(t), \cdot, \cdot], t, \theta$): ▷ Define dynamics on augmented state

  return $[f(\mathbf{z}(t), t, \theta), -\mathbf{a}(t)^\mathsf{T}\frac{\partial f}{\partial \mathbf{z}}, -\mathbf{a}(t)^\mathsf{T}\frac{\partial f}{\partial \theta}, -\mathbf{a}(t)^\mathsf{T}\frac{\partial f}{\partial t}]$ ▷ Compute vector-Jacobian products

$[\mathbf{z}(t_0), \frac{\partial L}{\partial \mathbf{z}(t_0)}, \frac{\partial L}{\partial \theta}, \frac{\partial L}{\partial t_0}] = \text{ODESolve}(s_0, \text{aug\_dynamics}, t_1, t_0, \theta)$ ▷ Solve reverse-time ODE

return $\frac{\partial L}{\partial \mathbf{z}(t_0)}, \frac{\partial L}{\partial \theta}, \frac{\partial L}{\partial t_0}, \frac{\partial L}{\partial t_1}$ ▷ Return all gradients

Full Adjoint sensitivities algorithm from the paper Chen et al. 2019

Let's break down the entire algorithm of Neural ODEs. We are considering a time series data where for $t \in [t_0, t_N]$ we have N observations $z \in [z_0, z_N]$. z follows a certain dynamics, whose shape (matrix) we know but not the parameters θ. We want to learn parameters of its dynamics with Neural ODE algorithm. The idea is to form the ODE IVP problem in form of neural network, where our layer will present the dynamics of the system z. The basic steps of the algorithm are:

- **Forward Propagation**: We initiate by forwarding the propagation of the system's dynamics through time ($t \in [t_0, t_N]$) to acquire an approximated trajectory denoted as $\hat{z}$. This entails solving the IVP associated with the ODE for the given time range.
- **Loss Computation**: Subsequently, we compute the loss, measuring the disparity between the obtained trajectory $\hat{z}$ and the actual observations z.
- **Backward Propagation**: At the endpoint t_N, an augmented adjoint state is formulated. This state encapsulates information required for the backward propagation phase. We then propagate this augmented state in reverse through time ($t \in [t_N, t_0]$), solving the reverse-time IVP. This step yields gradients with respect to the unknown parameters θ.

- **Parameter Update**: Armed with the gradients, we adjust the parameters θ of the neural network, enhancing its ability to model the system's dynamics. This iterative process is repeated until the desired level of loss is achieved.

Now that we have reviewed the fundamental steps, let's proceed to a more in-depth analysis of specific steps:

- In order to obtain the gradients $\frac{\partial L}{\partial \theta}$, which will be part of the augmented adjoint state at t_0, we need to solve the reverse-time ODE for the augmented adjoint state, which is defined as: $s(t) = \big[z(t), a(t), a_\theta, a_t\big]$.
- To solve it backwards in time we need the final augmented state, which will become an initial value for our reverse-time ODE IVP. This is denoted as $\rightarrow s_0 = \big[z(t_N), a(t_N), \mathbf{0}_\theta, a_{t_N}\big]$. Here keep in mind two things:
  - we are free to set the initial value of a_θ to be 0.
  - s_0 is indeed augmented adjoint state at t_N, but we denote it with 0 as it will be an initial state to our reverse-time ODESolver.
- To acquire s_0 we first need to propagate forward (with some randomly initialized θ) and thus obtain $\hat{z}$, loss L and $\frac{\partial L}{\partial z}$ for all time points. These gradients will be computed during backward pass. Propagating forward means solving ODE IVP forward in time with some ODE Solver.

$$\rightarrow \hat{z}(t_N) = z(t_0) + \int_{t_0}^{t_N} f(z, \theta, t)dt = \text{ODESolve}(z_0, f, \theta, t_0, t_N) \tag{43}$$

  IMPORTANT NOTE: we have to iterate over each time point and call the ODE Solver:

$$\rightarrow \hat{z}(t_{i+1}) = z(t_i) + \int_{t_i}^{t_{i+1}} f(z, \theta, t)dt = \text{ODESolve}(z_i, f, \theta, t_i, t_{i+1}) \text{ for } i \in [0, N-1]$$

  Once we're done with forward pass, we form an augmented state at t_N (s_0).
- Now it's time to backpropagate the augmented state, which involves solving reverse-time ODE.

$$\rightarrow \big[z(t_0), a(t_0), a_\theta, a_{t_0}\big] = \text{ODESolve}(s_0, \text{aug\_dynamics}, t_1, t_0, \theta)$$

  For this we need to know the dynamics of augmented state (aug_dynamics). If we recall it is defined as 22, however, since we added $z(t)$ to our augmented state we make sure our dynamics also includes the $f\left(z(t), t, \theta)\right)$. In a nutshell, we need to build an `aug_dynamics` function in presented algorithm. Same as in forward pass we have to iterate over each time point for $t \in [t_N, t_0]$. Important thing here is that at every time step we need to adjust the computed $a(t_i)$ and $a_t(t_i)$ with the direct current gradients, as it is described in the original paper (see Figure 4):

$$s_i = \text{ODESolve}(s_{i+1}, \text{aug\_dynamics}, t_{i+1}, t_i, \theta) = \begin{bmatrix} z_i \\ \text{ODESolve}(a(t_{i+1}), -a(t)\frac{\partial f}{\partial z}, t_{i+1}, t_i, \theta) + \underbrace{\mathbf{a(t_i)}}_{\frac{\partial L}{\partial z_i}} \\ \text{ODESolve}(\mathbf{0}_\theta, -a(t)\frac{\partial f}{\partial \theta}, t_{i+1}, t_i, \theta) \\ \text{ODESolve}(a_t(t_{i+1}), -a(t)\frac{\partial f}{\partial t}, t_{i+1}, t_i, \theta) + \underbrace{\mathbf{a_t(t_i)}}_{-a(t_i)f(z_i, t_i, \theta)} \end{bmatrix} \tag{44}$$

for $i \in [0, N-1]$.

- Once we have propagated backwards and obtained s at time t_0 we update our parameters and repeat the process until our loss reaches desired value.

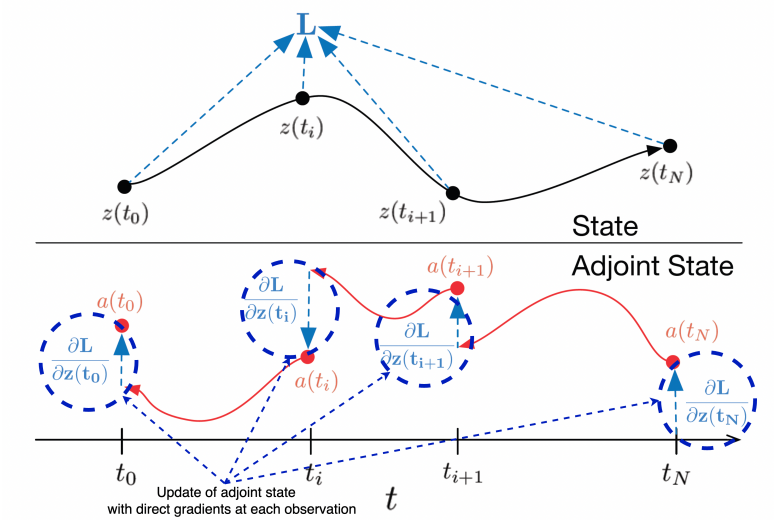


Figure 4: Reverse-mode differentiation of an ODE solution. The adjoint sensitivity method solves an augmented ODE backwards in time. The augmented system contains both the original state and the sensitivity of the loss with respect to the state. If the loss depends directly on the state at multiple observation times, the adjoint state must be updated in the direction of the partial derivative of the loss with respect to each observation. Chen et al. 2019. Note: only an update from $\frac{\partial L}{\partial z}$ is shown, however based on the paper we also need to update the $a_t(t)$ with the direct gradient at the current time step (this update is calculated as: $-a(t)f(z,t,\theta)$).

7 Implementation

We will offer a comprehensive explanation of each class/method within the implementation by provided by Surtsukov 2023. In the forthcoming sections, we will delve into the details of how each method tackles this problem and provides solutions.

7.1 Representing an ODE in form of Neural Network Layer - Neural ODE

We start our implementation with creating a class that will represent a specific ODE. The best way to explain the implementation is to look at specific example. For that purpose let's consider this system of ODEs:

$$\begin{aligned} \frac{dz_1}{dt} &= -0.1z_1 - 1.0z_2 \\ \frac{dz_2}{dt} &= 1.0z_1 - 0.1z_2 \end{aligned} \tag{45}$$

How can we represent this in matrix form?

First, let's define the vector $z = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix}$. Then, we can express the derivatives of the components z_1 and z_2 with respect to time t as a matrix-vector product:

$$\frac{d}{dt}\begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = \begin{bmatrix} \frac{dz_1}{dt} \\ \frac{dz_2}{dt} \end{bmatrix}$$

Now, we can rewrite the component equations in matrix-vector form using the coefficients from the given matrix:

$$\frac{d}{dt}\begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = \begin{bmatrix} -0.1 & -1.0 \\ 1.0 & -0.1 \end{bmatrix}\begin{bmatrix} z_1 \\ z_2 \end{bmatrix} \tag{46}$$

This matrix-vector equation compactly represents the given system of component equations. The matrix on the right side determines how the components z_1 and z_2 change over time, and the vector on the left side represents their derivatives with respect to time. This matrix-vector equation is a concise way to represent the original ODE in a matrix form.

Putting an ODE in this form we can see that we can actually model the dynamics (right side of the 46) as Neural Network layer. This is shown in Figure 5:

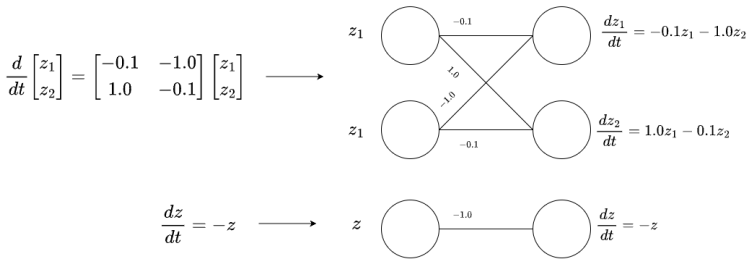


Figure 5: Representing an ODE problem (45, 47) in form of Neural Network Layer

We implement ODE or NN layer in form of class as in Figure 6.

```
class LinearODEF1(ODEF):
    def __init__(self, W):
        super(LinearODEF1, self).__init__()
        self.lin = nn.Linear(2, 2, bias=False)
        self.lin.weight = nn.Parameter(W)

    def forward(self, x, t):
        return self.lin(x)

class Example1(LinearODEF1):
    def __init__(self):
        super(Example1, self).__init__(Tensor([[-0.1, -1.], [1., -0.1]]))


class LinearODEF2(ODEF):
    def __init__(self, W):
        super(LinearODEF2, self).__init__()
        self.lin = nn.Linear(1, 1, bias=False)
        self.lin.weight = nn.Parameter(W)

    def forward(self, x, t):
        return self.lin(x)

class Example2(LinearODEF):
    def __init__(self):
        super(Example2, self).__init__(Tensor([[-1.]]))
```

Figure 6: Neural Network-based ODE Solver: A custom ODE function class 'ExampleODEF' encapsulating a linear layer to represent the ODE system (upper for 45, lower for 47). The learnable weights in the neural network layer approximate the system's behavior, allowing numerical solution of the ODE.

7.2 Super `class ODEF`

Let's recall the equation for the dynamics of augment adjoint state 22:

$$\frac{da_{aug}}{dt} = \left[-a\frac{\partial f}{\partial z} \quad -a\frac{\partial f}{\partial \theta} \quad -a\frac{\partial f}{\partial t}\right], \text{where} f = f(z(t), \theta, t)$$

We see that in order to compute it we first need to compute three gradients: $\frac{\partial f}{\partial z}, \frac{\partial f}{\partial \theta}, \frac{\partial f}{\partial t}$. This is why we need to define a method that will be called during backpropagation (i.e. during solving reverse-time ODE for s) and compute $f(z(t), \theta, t)$ as well as all three gradients. In a nutshell, this is a method that enables computing the dynamics of augmented adjoint state ($\frac{da_{aug}}{dt}$). Since this has to be method available to Neural ODE, we define a `super class ODEF` that encapsulates the defined class for Neural ODE and provides two methods:

- `forward_with_gradient` - returns $f(z(t), \theta, t), \frac{\partial f}{\partial z}(t), \frac{\partial f}{\partial \theta}(t), \frac{\partial f}{\partial t}(t)$ for $t \in [t_N, t_0]$
- `flatten_parameters` - method that flattens all the parameters θ function f depends on.

The code for this class is shown in Figure 7.

```python
class ODEF(nn.Module):
    """Superclass of parameterized dynamics function"""

    def forward_with_grad(self, z, t, grad_outputs):

        """
        Inputs:
        ------
            z: The state tensor at a particular time. This tensor represents the state of the system we're modeling.
            t: The time tensor at which the dynamics are being evaluated.
            grad_outputs(list of dL/dz for each time point)

        Returns:
        --------
            out: dynamics (the value of f(z(t), theta, t)),
            gradients:
                adfdz: a(t)*df/dz(t),
                adfdt: a(t)*df/dt(t) and
                adfdp: a(t)*(df/dp)

        """

        # Define the batch_size based on the shape of z.
        batch_size = z.shape[0]

        # Compute the dynamics of the system at the given state and time.
        out = self.forward(z, t)

        # The gradients of the loss(L) with respect to the current outputs. This is passed as a parameter 'a'.
        a = grad_outputs

        # Compute gradients using automatic differentiation. The torch.autograd.grad takes the following arguments:
                #outputs: A tuple of tensors for which gradients should be computed (in this case, just out).
                #inputs: A tuple of tensors with respect to which gradients should be computed
                #       (in this case, (z, t) + tuple(self.parameters())).
                #grad_outputs: The gradients of the outputs with respect to the current outputs (a in this case).
                #allow_unused: Allows for gradients to be unused if some inputs are not used in the computation.
                #retain_graph: Indicates whether to retain the computation graph for further backpropagation.

        adfdz, adfdt, *adfdp = torch.autograd.grad(
            (out,), (z, t) + tuple(self.parameters()), grad_outputs=(a),
            allow_unused=True, retain_graph=True
        )

        # grad method automatically sums gradients for batch items, we have to expand them back
        if adfdp is not None:
            adfdp = torch.cat([p_grad.flatten() for p_grad in adfdp]).unsqueeze(0)
            adfdp = adfdp.expand(batch_size, -1) / batch_size
        if adfdt is not None:
            adfdt = adfdt.expand(batch_size, 1) / batch_size
        return out, adfdz, adfdt, adfdp

    def flatten_parameters(self):
        """Method to flatten all the parameters a function depends on"""
        p_shapes = []
        flat_parameters = []
        for p in self.parameters():
            p_shapes.append(p.size())
            flat_parameters.append(p.flatten())
        return torch.cat(flat_parameters)
```

Figure 7: Code for `super class ODEF`. `forward_with_gradient` method first determines `batch_size` based on the shape of z. It then computes the forward pass by calling `self.forward(`z`,` t`)` to get the dynamics out. The `torch.autograd.grad` method is then used to compute the gradients with respect to z, t, and θ. The resulting gradients are expanded and normalized by the `batch_size` for consistency.

7.3 `class ODEAdjoint`

Now that we have defined essential classes and method for Neural ODE, we proceed to implementing class that will have methods for forward propagation (43) and backward propagation (44).

```
class ODEAdjoint(torch.autograd.Function):

    @staticmethod
    def forward():
        pass

    @staticmethod
    def backward():
        pass
```

Figure 8: `class ODEAdjoint` has two methods: `forward` and `backward`.

In the following sections we will explain each method and it's implementation, as well as provide an illustrative example of running forward and backward pass for 4 randomly chosen time steps of very simple ODE:

$$\frac{dz}{dt} = -z \tag{47}$$

7.3.1 `forward` method

To illustrate, we will address a straightforward initial value problem (IVP) for an ordinary differential equation (ODE): The purpose of the forward method is to calculate the trajectory of the system by solving the ordinary differential equation (ODE). For each time step we are employing the ODESolver(in this case Runge-Kutta 4th Order Method), to obtain the value of z at the next time step. In other words we're calculating the integral:

$$\hat{z}(t_{i+1}) = z(t_i) + \int_{t_i}^{t_{i+1}} f(z, \theta, t)dt = \text{ODESolve}(z_i, f, \theta, t_i, t_{i+1}) \text{ for } i \in [0, N-1]$$

As we can see the in code snippet that represents the forward pass 9 the computation is wrapped within a `torch.no_grad()` context. This is done to ensure that during the forward pass, no gradient information is being tracked. The reason why we want to calculate the trajectory without tracking gradients is that we are interested in the state evolution of the system for later use in the backward pass, specifically for computing gradients. The trajectory (z) is then stored in the context (ctx) for later use in the backward pass.

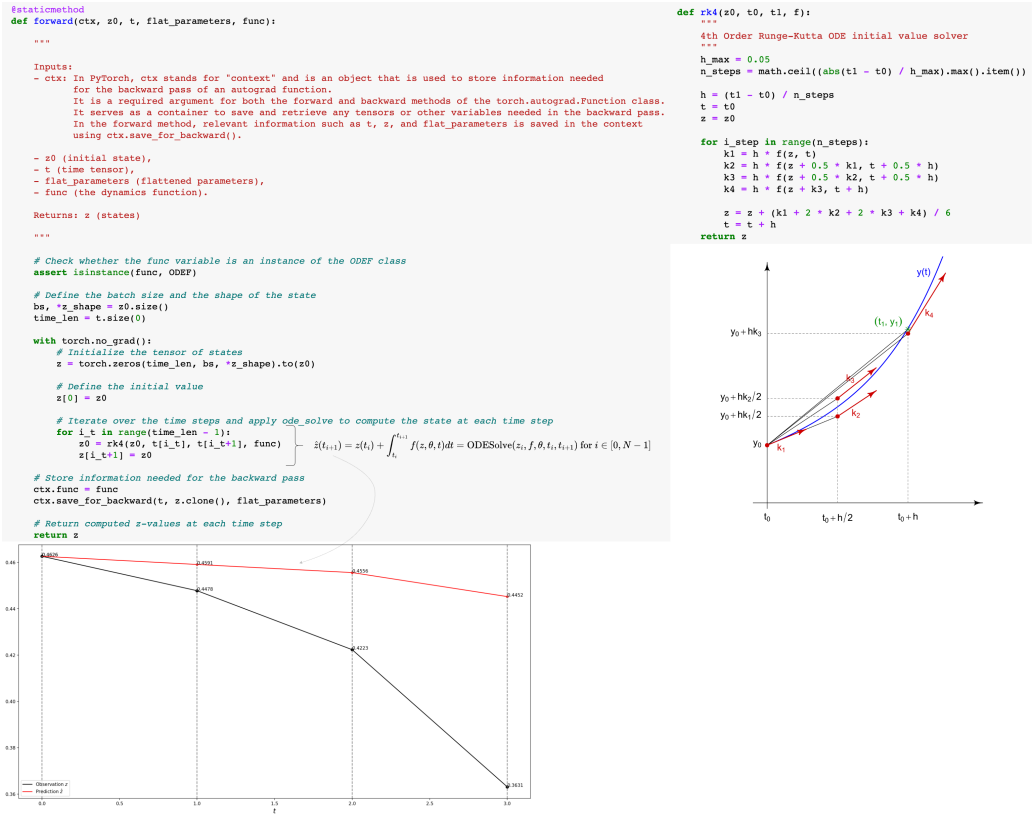


Figure 9: Forward method of `ODEAdjoint class` (left) and function `rk4` - Runge-Kutta 4th Order ODE Solver (right). Below the code we can see a graph showing prediction ($\hat{z}$) and observation z curve for 4 randomly choose time points. $\hat{z}$ is the result of integrating the ODE between each time step for $t \in [t_0, t_3]$ θ parameter was also randomly chosen as an initial guess.

7.3.2 `backward` method

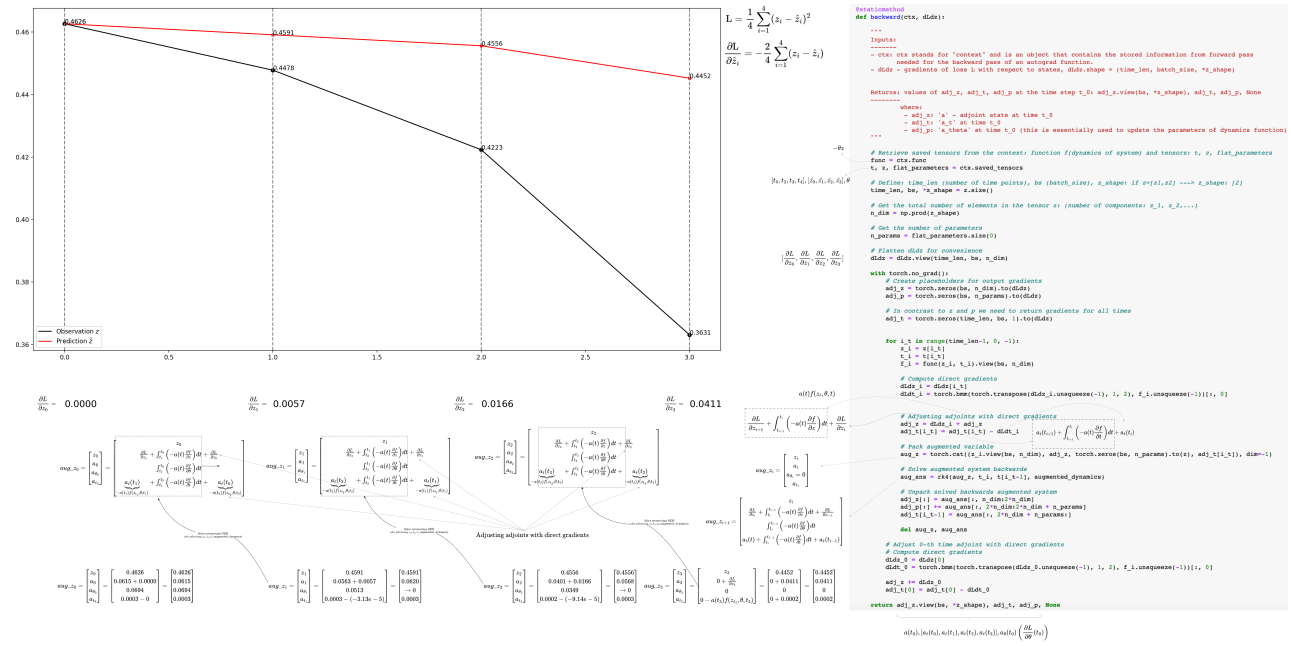


Figure 10: Step by step process of backpropagation of agumented adjoint state. As said, this example we have taken only 4 time points. The predictions $\hat{z}$ are computed with forward pass, that is by simply integrating the ODE between each time step for $t \in [t_0, t_3]$. Calling backward method we compute gradients of the loss wrt. to outputs at every time point (i.e. $\frac{\partial L}{\partial z_i} = a(t_i)$). We proceed to defining an initial adjoint augmented state $aug_{z_3} = [z_3, a(t_3), a_\theta(t_3) = 0, a_t(t_3)]$ that is an input to our reverse-time ODE solver. ODE solver then integrates the augmented state from t_3 to t_2, obtaining aug_{z_2} that is further adjusted with direct gradients at the current time point (t_2). Before passing current augmented state to ODE solver from t_2 to t_1 we set a_θ to zero. This is what we do for every time point, except for the last augmented state. After iterating from t_3 to t_0 we return the last augmented state aug_{z_0}.

```python
def augmented_dynamics(aug_z_i, t_i):
    """Function that calculates the fynamics of augmented system

    Inputs:
    -------
    aug_z_i: current augmented state [z_i, adj_z_i, 0, adj_t_i]
    t_i : current time point


    Returns:
    -------
    [f(t_i), -a*dfdz, -a*dfdp, -adfdt]
    """

    # Extract z and a from augmented variable
    z_i, a = aug_z_i[:, :n_dim], aug_z_i[:, n_dim:2*n_dim]  # ignore parameters and time

    # Unflatten z and a
    z_i = z_i.view(bs, *z_shape)
    a = a.view(bs, *z_shape)

    with torch.set_grad_enabled(True):
        t_i = t_i.detach().requires_grad_(True)
        z_i = z_i.detach().requires_grad_(True)
        func_eval, adfdz, adfdt, adfdp = func.forward_with_grad(z_i, t_i, grad_outputs=a)

        #Some gradients might be None, so let's make sure we turn those into zeros
        adfdz = adfdz.to(z_i) if adfdz is not None else torch.zeros(bs, *z_shape).to(z_i)
        adfdp = adfdp.to(z_i) if adfdp is not None else torch.zeros(bs, n_params).to(z_i)
        adfdt = adfdt.to(z_i) if adfdt is not None else torch.zeros(bs, 1).to(z_i)

    # Flatten f and adfdz
    func_eval = func_eval.view(bs, n_dim)
    adfdz = adfdz.view(bs, n_dim)

    return torch.cat((func_eval, -adfdz, -adfdp, -adfdt), dim=1)
```

Figure 11: Function for calculating augmented dynamics ($\frac{da_{aug}}{dt}$). As an input it takes current augmented adjoint state and current time point. We call the `forward_with_gradient` method to compute the $f, \frac{\partial f}{\partial z}, \frac{\partial f}{\partial \theta}, \frac{\partial f}{\partial t}$. We call it by passing z and t with `requires_grad_(True)` but detaching it from the previous computational graph (we want to prevent gradients from flowing through previous steps. For each t_i and $z(t_i)$ we want to compute independent gradients, otherwise gradients would add up for all previous layers). Since some gradients can be None we convert those to zero (for example if f doesn't depend on t, $\frac{\partial f}{\partial t}$ will be None). Finally we return $f, -a\frac{\partial f}{\partial z}, -a\frac{\partial f}{\partial \theta}, -a\frac{\partial f}{\partial t}$.

Our backward method also needs to include the function that calculates the augmented dynamics. Here we show it separately from backward method but keep in mind that it needs to be defined inside the backward method.

7.4 `class NeuralODE`

For convenience we will wrap `class ADjoint` with **nn.Module** and thus define NeuralODE class. This class takes defined ODEF function as an input and has `forward` method that applies forward and backward methods from `class ADjoint` class.

```
class NeuralODE(nn.Module):
    def __init__(self, func):
        super(NeuralODE, self).__init__()
        assert isinstance(func, ODEF)
        self.func = func

    def forward(self, z0, t=Tensor([0., 1.]), return_whole_sequence=False):
        t = t.to(z0)
        z = ODEAdjoint.apply(z0, t, self.func.flatten_parameters(), self.func)
        if return_whole_sequence:
            return z
        else:
            return z[-1]
```

Figure 12: Implementation of `class NeuralODE`

8 Training set-up

```python
def conduct_experiment(ode_true, ode_trained, z0, n_steps, t_max, n_poins,min_delta_time, max_delta_time,
                       max_points_num, wrt_t = False, plot_freq=10,
                       optimizer = torch.optim.AdamW(ode_trained.parameters(), lr=0.05),scheduler = None,
                       desired_loss):

    # Create the data
    index_np = np.arange(0, n_points, 1, dtype=np.int)
    times_np = np.linspace(0, t_max, num=n_points)
    times = torch.from_numpy(times_np).to(z0)

    # Get observations
    obs = ode_true(z0, times, return_whole_sequence=True).detach()
    # Add random noise to the observation tensor
    obs = obs + torch.randn_like(obs) * 0.01


    def create_batch():
        """Function to get a mini_batch"""

        t0 = np.random.uniform(0, t_max - max_delta_time)
        t1 = t0 + np.random.uniform(min_delta_time, max_delta_time)

        idx = sorted(np.random.permutation(index_np[(times_np > t0) & (times_np < t1)])[:max_points_num])

        obs_ = obs[idx]
        ts_ = times[idx]
        return obs_, ts_, idx

    #Train Neural ODE
    #----------------#

    # Initialize the LOSS array
    LOSS = []
    mean_losses = []
    mean_current_loss = np.float64("inf")

    for i in range(1, n_steps + 1):

        # Generate a mini-batch of observations and corresponding time values
        obs_, ts_, idx = create_batch()

        # Compute z_ by passing the first observation obs_[0] and its corresponding time values ts_
        # through the ode_trained model.
        z_ = ode_trained(obs_[0], ts_, return_whole_sequence=True)

        #compute the loss
        loss = F.mse_loss(z_, obs_.detach())
        LOSS.append(loss.item())

        if mean_current_loss <= desired_values:

            z_p = ode_trained(z0, times, return_whole_sequence=True)

            plot_trajectories(obs=obs, times=times, trajs=z_p,
                              parameters = ode_trained.state_dict(),
                              wrt_t = True, plot_loss=True, losses=mean_losses,
                              current_loss = mean_current_loss, step=i,
                              learning_rate = lr)
            break

        # Reset optimizer gradients to zero
        optimizer.zero_grad()

        # Propagate loss through the computation graph
        loss.backward(retain_graph=True)

        # Update the model's parameters based on the computed gradients
        optimizer.step()

        # Extract the learning rate and update it if specified
        lr = optimizer.param_groups[0]['lr']
        if scheduler is not None:
            scheduler.step()

        if i % plot_freq == 0:
            mean_current_loss = np.mean(LOSS)
            mean_losses.append(mean_current_loss)
            LOSS = []

            z_p = ode_trained(z0, times, return_whole_sequence=True)

            if i == n_steps:
                plot_trajectories(obs=obs, times=times, trajs=z_p,
                                  parameters = ode_trained.state_dict(),
                                  wrt_t = True, plot_loss=True, losses=mean_losses,
                                  current_loss = mean_current_loss, step=i,
                                  learning_rate = lr)
            else:

                plot_trajectories(obs=obs, times=times, trajs=z_p,
                                  parameters = ode_trained.state_dict(),
                                  wrt_t = True, current_loss=mean_current_loss, step=i, learning_rate = lr,
                                  idx=idx)

                clear_output(wait=True)
```

Figure 13: Function that wraps the training of Neural ODEs

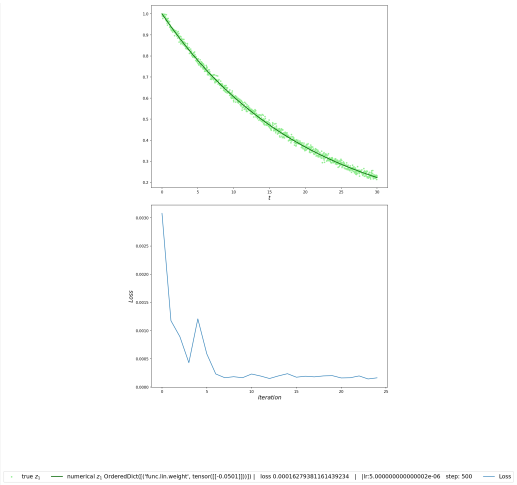
9 Examples

9.1 First example

$$\frac{dz}{dt} = -z \text{ with } z(0) = 1$$

```
class LinearODEF(ODEF):
    def __init__(self, W):
        super(LinearODEF, self).__init__()
        self.lin = nn.Linear(1, 1, bias=False)
        self.lin.weight = nn.Parameter(W)

    def forward(self, x, t):
        return self.lin(x)

class exponential_decayExample(LinearODEF):
    def __init__(self):
        super(exponential_decayExample, self).__init__(Tensor([[-0.05]]))

class RandomLinearODEF(LinearODEF):
    def __init__(self):
        super(RandomLinearODEF, self).__init__(torch.randn(1,1)/100.)

ode_true = NeuralODE(exponential_decayExample())
ode_trained = NeuralODE(RandomLinearODEF())

t_max = 30
n_points = 1000

# Get trajectory of random timespan
min_delta_time = 1.0
max_delta_time = 10
max_points_num = 256

#initial point
z0 = Variable(torch.Tensor([[1]]))

optimizer = torch.optim.AdamW(ode_trained.parameters(), lr=0.05)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=100, gamma=0.1)

conduct_experiment(ode_true=ode_true,
                   ode_trained=ode_trained,
                   z0=z0,
                   n_steps=500,
                   t_max=t_max,
                   n_poins=n_points,
                   min_delta_time=min_delta_time,
                   max_delta_time=max_delta_time,
                   max_points_num=max_points_num,
                   wrt_t = True, plot_freq=20, optimizer = optimizer, scheduler = scheduler,
                   desired_loss = 1.3e-4)
```

(a) Neural ODE setup for solving first example



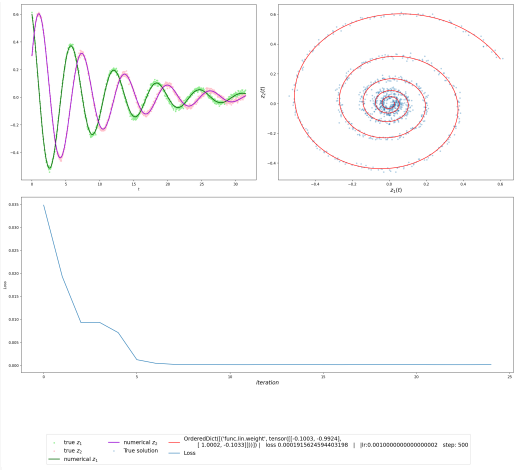
(b) Results for the first example

9.2 Second example

$$\frac{dz}{dt} = \begin{bmatrix} -0.1 & -1.0 \\ 1.0 & -0.1 \end{bmatrix} z \text{ with } \begin{bmatrix} z_1 \\ z_2 \end{bmatrix}(0) = \begin{bmatrix} 0.6 \\ 0.3 \end{bmatrix}$$

```
class LinearODEF(ODEF):
    def __init__(self, W):
        super(LinearODEF, self).__init__()
        self.lin = nn.Linear(2, 2, bias=False)
        self.lin.weight = nn.Parameter(W)

    def forward(self, x, t):
        return self.lin(x)

class SpiralFunctionExample(LinearODEF):
    def __init__(self):
        super(SpiralFunctionExample, self).__init__(Tensor([[-0.1, -1.], [1., -0.1]]))

class RandomLinearODEF(LinearODEF):
    def __init__(self):
        super(RandomLinearODEF, self).__init__(torch.randn(2, 2)/100.)

ode_true = NeuralODE(SpiralFunctionExample())
ode_trained = NeuralODE(RandomLinearODEF())


t_max = 6.29*5
n_points = 1000

# Get trajectory of random timespan
min_delta_time = 1.0
max_delta_time = 5.0
max_points_num = 32

#initial point
z0 = Variable(torch.Tensor([[0.6, 0.3]]))

optimizer = torch.optim.AdamW(ode_trained.parameters(), lr=0.1)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=200)

conduct_experiment(ode_true=ode_true,
                   ode_trained=ode_trained,
                   z0=z0,
                   n_steps=500,
                   t_max=t_max,
                   n_poins=n_points,
                   min_delta_time=min_delta_time,
                   max_delta_time=max_delta_time,
                   max_points_num=max_points_num,
                   wrt_t = True, plot_freq=20, optimizer = optimizer, scheduler = scheduler,
                   desired_loss = 1.3e-4)
```

(a) Neural ODE setup for solving second example



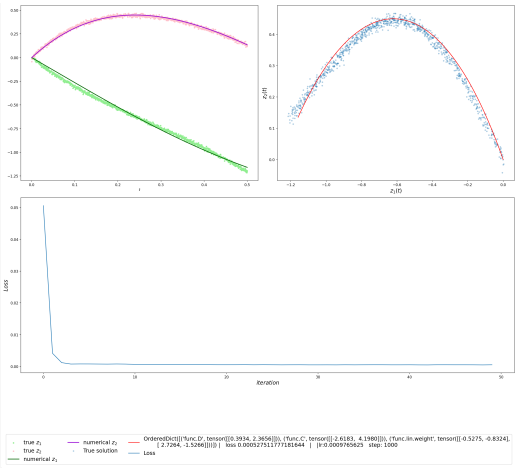
(b) Results for the second example

9.3 Third example

$$\frac{d}{dt}\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a_1 & b_1 \\ a_2 & b_2 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} c_1 e^{-d_1 t} \\ c_2 e^{-d_2 t} \end{bmatrix} \quad \text{with} \quad \begin{bmatrix} x \\ y \end{bmatrix}(0) = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

```
class MyODEF(ODEF):
    def __init__(self, AB, D, C):
        super(MyODEF, self).__init__()
        self.lin = nn.Linear(2, 2, bias=False)
        self.lin.weight = nn.Parameter(AB)
        self.D = nn.Parameter(D)
        self.C = nn.Parameter(C)

    def forward(self, x, t):
        return self.lin(x) + torch.mul(self.C,torch.exp(torch.mul(self.D,-t)))

a1 = 1.11
b1 = 2.43
c1 = -3.66
d1 = 1.37
a2 = 2.89
b2 = -1.97
c2 = 4.58
d2 = 2.86

class MyExample(MyODEF):
    def __init__(self):
        super(MyExample, self).__init__(Tensor([[a1, a2], [b1, b2]]),  Tensor([[d1,d2]]), Tensor([[c1,c2]]))

class RandomMyODEF(MyODEF):
    def __init__(self):
        super(RandomMyODEF, self).__init__(torch.randn(2, 2)/100,  torch.randn(1, 2)/100, torch.randn(1, 2)/100)

ode_true = NeuralODE(MyExample())
ode_trained = NeuralODE(RandomMyODEF())

t_max = 0.5
n_points = 1000

#get trajectory of random timespan
min_delta_time = 0
max_delta_time = 0.5
max_points_num = 128

#initial point
z0 = Variable(torch.Tensor([[0, 0]]))

optimizer = torch.optim.AdamW(ode_trained.parameters(), lr=0.5)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=100, gamma=0.5)


conduct_experiment(ode_true=ode_true,
                   ode_trained=ode_trained,
                   z0=z0,
                   n_steps=1000,
                   t_max=t_max,
                   n_points=n_points,
                   min_delta_time=min_delta_time,
                   max_delta_time=max_delta_time,
                   max_points_num=max_points_num,
                   wrt_t = True, plot_freq=20, optimizer = optimizer, scheduler = scheduler,
                   desired_loss=1e-4)
```

(a) Neural ODE setup for solving third example



(b) Results for third example

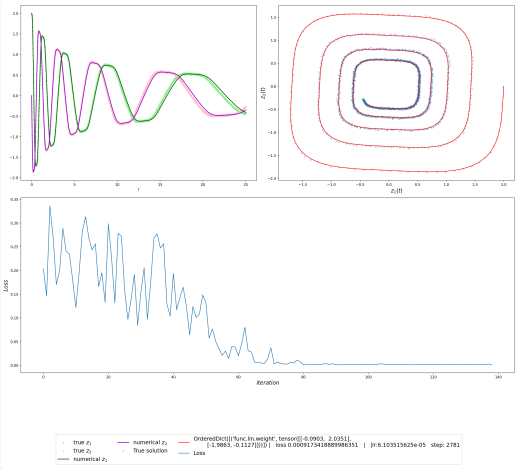
Figure 16: Comparison of Neural ODE setup and results for the 4th example

9.4 Fourth example

$$\frac{dz}{dt} = \begin{bmatrix} -0.1 & 2.0 \\ -2.0 & -0.1 \end{bmatrix} z^3 \quad \text{with} \quad \begin{bmatrix} z_1 \\ z_2 \end{bmatrix}(0) \begin{bmatrix} 0.0 \\ 2.0 \end{bmatrix}$$

```
class LinearODEF(ODEF):
    def __init__(self, W):
        super(LinearODEF, self).__init__()
        self.lin = nn.Linear(2, 2, bias=False)
        self.lin.weight = nn.Parameter(W)

    def forward(self, x, t):
        return self.lin(x**3)

class forthExample(LinearODEF):
    def __init__(self):
        super(zerothExample, self).__init__(Tensor([[-0.1, 2.], [-2., -0.1]]))

class RandomODEF(LinearODEF):
    def __init__(self):
        super(RandomODEF, self).__init__(torch.randn(2, 2)/10000)

ode_true = NeuralODE(forthExample())
ode_trained = NeuralODE(RandomODEF())

t_max = 25
n_points = 2000

# Get trajectory of random timespan
min_delta_time = 5
max_delta_time =10
max_points_num = 32
#initial point
z0 = Variable(torch.Tensor([[2., 0.]]))

optimizer = torch.optim.AdamW(ode_trained.parameters(), lr=0.5)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=200, gamma=0.5)

conduct_experiment(ode_true=ode_true,
                   ode_trained=ode_trained,
                   z0=z0,
                   n_steps=4000,
                   t_max=t_max,
                   n_points=n_points,
                   min_delta_time=min_delta_time,
                   max_delta_time=max_delta_time,
                   max_points_num=max_points_num,
                   wrt_t = True, plot_freq=20, optimizer = optimizer, scheduler = scheduler,
                   desired_loss = 0.001)
```

(a) Neural ODE setup for solving fourth example



(b) Results for fourth example

Figure 17: Comparison of Neural ODE setup and results for the 4th example

References

[1] Ricky T. Q. Chen et al. *Neural Ordinary Differential Equations*. 2019. arXiv: 1806.07366 [cs.LG].

[2] Edward Choi. *Neural ODE*. https://mp2893.com/. YouTube link: https://www.youtube.com/watch?v=sIFnARdTVvE&list=PLLENHvsRRLjDHllrXj0B8sz5-4xVbisBL&index=27&ab_channel=EdwardChoi Accessed on August 15, 2023. 2023.

[3] Andriy Drozdyuk. *Neural ODE*. https://github.com/drozzy/Neural-Ordinary-Differential-Equations-YouTube/blob/main/Nov_29_2020_NeuralODEs.pdf. YouTube link: https://www.youtube.com/watch?v=uPd0B0WhH5w&ab_channel=AndriyDrozdyuk Accessed on August 15, 2023. 2023.

[4] Long Huu Nguyen and Andy Malinsky. "Exploration and Implementation of Neural Ordinary Differential Equations". In: *Capstone Showcase* 8 (2020). URL: https://scholarworks.arcadia.edu/showcase/2020/comp_sci_math/8.

[5] Mikhail Surtsukov. *Nural ODE Implementation*. https://github.com/msurtsukov/neural-odel. Accessed on August 1, 2023. 2023.

[6] The University of Queensland. *Euler's Method*. https://teaching.smp.uq.edu.au/scims/Appl_analysis/Eulers_method.html. Accessed on August 16, 2023. 2023.