# Introduction to Neural Ordinary Differential Equation

Della Bona Sarah, Dumez Erika

June 18, 2021

# Contents

# 1 Introduction

In this document, we introduce Neural Ordinary Differential Equation Networks (ODE-Nets), which are deep neural networks models using ordinary differential equations. ODE-Nets are a fairly recent field of study. It uses ODEs which are, on the contrary, very old and have been studied for a long time, and thus are well-known. Unlike classical neural networks, the hidden layer in an ODE-Net is defined as a black box that uses an ODE solver.

This model is very promising since it has advantages over simple neural networks, such as a constant memory cost as well as better results on continuous time series data. Furthermore, this infinite-depth approach helps model dynamical systems better.

Why use ODE-Nets over regular neural networks? Sometimes it is more natural to describe a system by its dynamics rather than the observable which is sometimes hard to understand and describe. This often happens in Physics and Engineering.

Here, we will focus in particular on the mathematical aspects of these neural networks. We will give definitions and properties for different notions such as ordinary differential equations, regular and residual neural networks, ...

At the end, we'll conclude with the advantages and disadvantages of ODE-nets. The code used to make the examples can be found at `https://github.com/DumezErika/ProjetMachineLearning`.

We can begin by the basis and describe what a machine learning problem is.

# 2 Machine Learning

In a typical machine learning problem [8], we have an output variable $Y$ to $p$ predictors $X_1, \dots, X_p$, also called input variables, where $p \in \mathbb{N} \backslash \{0\}$. The inputs belongs to an input space $\mathcal{X}$ and usually $\mathcal{X} \subset \mathbb{R}^p$. The output belongs to a output space $\mathcal{Y}$. If this is a regression problem, $\mathcal{Y} \subset \mathbb{R}$. But if we have a classification problem with $K$ categories, $\mathcal{Y} = \{1, 2, \dots, K\}$.

Let us assume that there is some relationship between $Y$ and $X = (X_1, \dots, X_p)$, which can be written in the general form

$$Y = f(X) + \epsilon$$

where $f$ is some fixed but unknown function, called *target function*, of $X_1, \dots, X_p$ and $\epsilon$ is a random error term which is independent of $X$ and has mean zero and finite variance.

There is multiple types of machine learning problem. One of them is *supervised learning* and it is the one we will consider in this document. The goal of supervised learning is to estimate the function $f$ as precisely as possible thanks to a model that we will train. To do that, we need a *data set* in order for the model to learn. The data is a set of $n$ points in $\mathcal{X} \times \mathcal{Y}$

$$\mathcal{D} = \{(x_1, y_1), \dots, (x_n, y_n)\}.$$

a sample of the variables.

Let $x$ be a point in the input space $\mathcal{X}$, then we can predict its output $y$ using

$$\hat{y} = \hat{f}(x),$$

where $\hat{f}$ represents our estimate for $f$, and $\hat{y}$ represents the resulting prediction for $y$.

To measure the accuracy of a prediction given by $\hat{f}$, we use a *loss function* $\mathcal{L}$ from $\mathbb{R}^2$ to $\mathbb{R}$, which is a function of a prediction and the output given by the target function. Some example of loss functions are

- Square error loss: $\mathcal{L}(y, \hat{y}) = (y - \hat{y})^2$;
- Absolute error loss: $\mathcal{L}(y, \hat{y}) = |y - \hat{y}|$;
- Zero-one loss: $\mathcal{L}(y, \hat{y}) = \mathbb{1}_{\{(y,\hat{y})|y \neq \hat{y}\}}(y, \hat{y})$.

To train the model we use a data set called the *train set.* The model is constructed with different parameters $\theta$. During the training we search for the parameters that will minimize a chosen error measure (loss function). Once the model is trained, we can test it on an other data set, called the *test set*, using an error measure which can be a loss function. The error on the test set is called *out-of-sample error* whereas the error on the training set is called the *in-sample error.*

We will focus on one type of model in particular which are neural networks.

## 2.1 Neural Networks

A *neural network* [12] is used to solve a machine learning problem. It consists of a series of layers. There are three types of layers :

- The *input* layer
- The *output* layer
- The *hidden* layers

Each layer consist of a certain number of neurons. We give an input $x$ to the neurons of a layer and they give an output $z$. An *activation function* is then applied to this output and obtain a value $h$ before transmitting it to the next layer thanks to the connections between the neurons of each layer. The most used activation functions are :

- Sigmoid : $sigmoid(z) = \frac{1}{1+e^{-z}}$;
- Hyperbolic tangent : $\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$;
- ReLU : $ReLU(z) = \max(0, z)$.

We use an activation function to add non-linearity to the network.

The simplest example of a neural network layer is

$$h = \sigma(Wx + b)$$

where $\sigma$ is an activation function, $W$ is a weight matrix and $b$ a bias vector. We consider these values as parameters of the network.

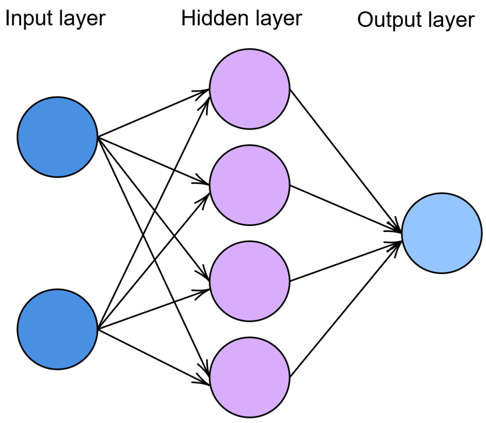

Figure 1: Example of neural network

We begin by giving an input to the input layer, which transmits information to the first hidden layer[1]. In turn, it transmit information to the next layer and so on, until the output layer gives us the final output, the *prediction*. It is called the *forward pass*, which is the process of getting a prediction from an input. An example of neural network is given in Figure 1.

The goal is to minimize the training error for every input of the training set. To do that, we need to find the optimal parameters for the network which minimize the loss function. We need to compute the derivatives of the loss with respect to the parameters $\theta$, which in the case of a neural network are the weights and biases. The process used to get these derivatives is the *backward pass*.

## 2.2 Back propagation

Let $\theta$ be the parameters of the network. We want to find $\theta^*$ which minimize the loss function in order to have the in-sample error as small as possible.

To find a value that minimizes a function, we can first try to find a value for which the derivative of the function is 0. A theorem (see Appendix, Lemma 1) states that such a value is a local optimum of the function. If we suppose that the function is convex, then a local optimum is a global minimum.

In the case of the neural network, the function we consider is the loss function with respect to the parameters $\theta$. Therefore, we need to determine the partial derivative of the loss function with respect to the parameters, $\frac{\partial \mathcal{L}}{\partial \theta}$.

Back propagation [5] is the process used to compute this derivative. It works by computing the gradient of the loss function with respect to each parameter by the chain rule, computing the gradient one layer at a time, iterating backward from the final layer to avoid redundant calculations of intermediate terms in the chain rule.

[1] There isn't always an hidden layer in a neural network

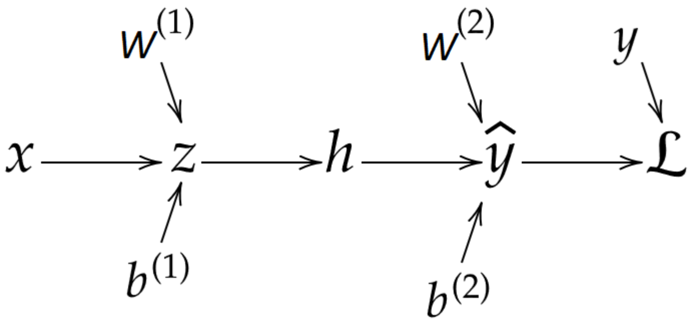

Figure 2: Computation graph

### 2.3 Example

Let's consider a neural network with one hidden layer that takes a two-dimensional input $x = (x_1, x_2)^T$ and gives a 2-dimensional output $\hat{y} = (\hat{y_1}, \hat{y_2})^T$. We can represent this network with the following equations:

$$\begin{aligned} z &= W^{(1)}x + b^{(1)} \\ h &= \sigma(z) \\ \hat{y} &= W^{(2)}h + b^{(2)} \\ \mathcal{L} &= \frac{1}{2}\|\hat{y} - y\|_2^2 \end{aligned}$$

where $W^{(1)}, W^{(2)} \in \mathbb{R}^2 \times \mathbb{R}^2$ and $b^{(1)}, b^{(2)} \in \mathbb{R}^2$ are the parameters $\theta$ of the network and $\sigma$ is an activation function.

We can now use the back propagation algorithm to easily compute $\frac{\partial \mathcal{L}}{\partial W^{(1)}}, \frac{\partial \mathcal{L}}{\partial W^{(2)}}, \frac{\partial \mathcal{L}}{\partial b^{(1)}}, \frac{\partial \mathcal{L}}{\partial b^{(2)}}$, the partial derivatives of the loss function with respect to the parameters.

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \mathcal{L}} &= 1 \\
\frac{\partial \mathcal{L}}{\partial \hat{y}} &= \frac{\partial \mathcal{L}}{\partial \mathcal{L}}\,(\hat{y}-y) \\
\frac{\partial \mathcal{L}}{\partial w^{(2)}} &= \frac{\partial \mathcal{L}}{\partial \hat{y}}\,h^T \\
\frac{\partial \mathcal{L}}{\partial b^{(2)}} &= \frac{\partial \mathcal{L}}{\partial \hat{y}} \\
\frac{\partial \mathcal{L}}{\partial h} &= (w^{(2)})^T\,\frac{\partial \mathcal{L}}{\partial \hat{y}} \\
\frac{\partial \mathcal{L}}{\partial z} &= \frac{\partial \mathcal{L}}{\partial h} \circ \sigma'(z) \\
\frac{\partial \mathcal{L}}{\partial w^{(1)}} &= \frac{\partial \mathcal{L}}{\partial z}\,x^T \\
\frac{\partial \mathcal{L}}{\partial b^{(1)}} &= \frac{\partial \mathcal{L}}{\partial z}
\end{aligned}$$

### 2.4 Gradient descent

Our goal is to find $\theta^*$ that minimizes the loss function. With back propagation, we computed the gradient of the loss function with respect to $\theta$. Now we need to use this gradient to find $\theta^*$.

*Gradient descent* [1] is a process used to approximate a local minimum of a differentiable function. It works as follow: at each step of the process, we take a step in the opposite direction of the gradient of this function at the current point, because this is the direction of the steepest descent.

More formally, if we have a function $g : \mathbb{R}^m \to \mathbb{R}$, $m > 1$, differentiable and a point $x_0 \in \mathbb{R}^m$, we have that if

$$x_{n+1} = x_n - \gamma_n \nabla g(x_n), n \geq 0$$

for $\gamma_n \in \mathbb{R}^+$ small enough, then $g(x_n) \geq g(x_{n+1})$. The variables $\gamma_n$ are called *step-size*, and are used to control the number of iteration needed to get to the optimal value. These values need to be carefully chosen because if they are too small or too large, the gradient descent algorithm might not converge.

We get a sequence $x_0, x_1, \ldots$ that converges to the desired local minimum under some conditions (see theorem 2 in Appendix), such that

$$g(x_0) \geq g(x_1) \geq \ldots.$$

If the function $g$ is convex, all local minima are also global minima, so the gradient descent can converge to the global minimum.

The problem when using the gradient descent algorithm on neural network is that each weight is updated using the partial derivative of the loss function with respect to the current weight, and if this gradient is too small, it will prevent the weight from changing its value. In this case, the neural network will not be able to learn. This is the *vanishing gradient problem* [7].

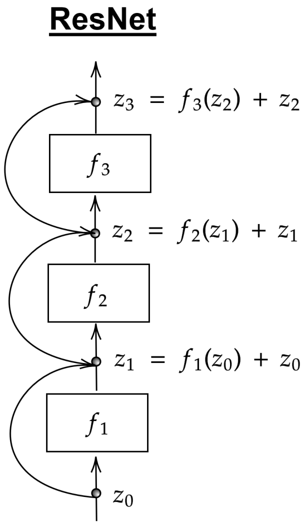

Figure 3: Example of residual neural network

One example of this problem is when we use the hyperbolic tangent as activation function. Because this function has gradients in the range $]0,1[$ and back propagation computes gradients by the chain rule, we multiply several of these small numbers which leads the gradient to decrease exponentially. The deeper is the neural network, the more likely this problem can occur.

The *exploding gradient problem* is the opposite, it happens when the derivatives take on larger values.

## 2.5 Residual neural network

A *residual neural network* [6], also called ResNet, is a neural network with more connections. Not only do we feed the output of the previous layer to the next, but also the input of that layer. An example of the representation of a ResNet is given in Figure 3.

In these networks, the output of the $k+1$th layer is given by

$$x_{k+1} = x_k + f_k(x_k) \tag{1}$$

where $f_k$ is the function of the $k$th layer and its activation.

With these additional connections, we can avoid the problems of the *vanishing gradient* and the *exploding gradient* and thus have a better accuracy.

Residual networks avoid the problem of vanishing gradient by introducing short paths which can carry a gradient over the entire depth of very deep networks. This is because adding the information from the previous layer will make these activations larger, so to some extent, they will prevent these activations from becoming exponentially small.

### 2.6 Implicit Layers

There is two different ways to define a layer : *explicitly* or *implicitly* [9]. When we define a layer explicitly, we specify the exact sequence of operations to do from the input to the output layer like in the example in Section 2.3.

However, when we add some functionality to the layers, it can become complex to define them explicitly. Instead, we can define them implicitly: we specify the condition we want the layer's output to satisfy.

An *explicit layer* is defined by a function $f : \mathcal{X} \to \mathcal{Y}$. For an implicit layer, we give a condition that a function $g : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}^n$ should satisfy. For example we can search for a $y$ such that $g(x, y) = 0$.

## 3 Ordinary Differential Equations

An *ordinary differential equation* (ODE) [4] is an equation that describes the changes of a function through time, a continuous variable. The goal is to compute that function from the ODE which describes its derivative.

**Definitions 1.** Let $f : \Omega \to \mathbb{R}^N$ where $\Omega \subseteq \mathbb{R} \times \mathbb{R}^N$ is an open set.

- A *first order ODE* takes the form

$$\frac{\partial u}{\partial t}(t) = f(t, u(t)).$$

- A *solution* for this ODE is a function $u : I \to \mathbb{R}^N$, where $I$ is an interval of $\mathbb{R}$, such that
  - $u$ is differentiable on $I$,
  - $\forall t \in I, (t, u(t)) \in \Omega$,
  - $\forall t \in I, \frac{\partial u}{\partial t}(t) = f(t, u(t))$

- An *initial condition* (IC) is a condition of the type

$$u(t_0) = u_0$$

  where $(t_0, u_0) \in \Omega$ is given.

- A *Cauchy problem* is an ODE with IC

$$\begin{cases} \frac{\partial u}{\partial t}(t) & = & f(t, u(t)) \\ u(t_0) & = & u_0 \end{cases}$$

To be sure to understand what an ODE is, we can look at a little example. Let $\frac{\partial f}{\partial t}(t) = f(t)$ an ODE. The solutions of this ODE are

$$\{f(t) = ae^t \mid a \in \mathbb{R}\}.$$

Indeed, for all $a \in \mathbb{R}$ we have

$$\frac{\partial ae^t}{\partial t} = ae^t$$

If we add an initial condition $f(0) = 1$, we have a Cauchy problem and its solution is $e^t$, since $e^0 = 1$ and $\frac{\partial e^t}{\partial t} = e^t$.

If we want to find the solution to an ODE, we need to know the conditions under which this ODE has a solution. Therefore, we define *Lipschitz continuous functions*. This notion is crucial for the following theorem which gives conditions for the existence and uniqueness of a solution to an ODE.

**Definition 1.** Let $(X, d_X)$ and $(Y, d_Y)$ be two metric spaces. A function $g : X \to Y$ is called *Lipschitz continuous* if

$$\exists K \geq 0,\ \forall x_1, x_2 \in X,\ d_Y(g(x_1), g(x_2)) \leq K d_X(x_1, x_2).$$

**Theorem 1.** ***Picard-Lindelöf theorem***
*Consider the Cauchy problem*

$$\frac{\partial u}{\partial t}(t) = f(t, u(t)), \quad u(t_0) = u_0.$$

*Suppose $f$ is uniformly Lipschitz continuous in $u$ and continuous in $t$. Then for some value $T > 0$, there exists a unique solution $u(t)$ to the Cauchy problem on the interval $[t_0, t_0 + T]$.*

## 3.1 One-step methods

Unfortunately, it is not always possible to find a closed-form solution to a Cauchy problem. However we can approximate the solution and there exists a lot of ways to do that. One type of methods are the *one-step methods* on which we will focus here [2].

Let $T > 0$ such that the solution $u$ exists on $[t_0, t_0 + T]$ and let $n \geqslant 2$ be a natural. Let $t_0 < ... < t_n \in [t_0, t_0 + T]$ where $t_n = t_0 + T$. We can compute a finite number of points $(u_1, \ldots, u_n)$ such that:

$$\forall i \in \{0, \ldots, n\}, u_i \approx u(t_i).$$

To compute those points, we use *one-step methods* which compute the points $u_{i+1}$ from the previous point $u_i$, the time $t_i$ and the *step* $h_i := t_{i+1} - t_i$.

Some of the most used one-step methods are

- Euler's method;
- Runge-Kutta.

Here we will focus on Euler's method.

[2] Other types of methods, for example multi-steps methods, are more accurate and find solutions closer the the real functions.

### 3.2 Euler's method

Euler's method is a one-step method with a constant step $h$ such that for all $i \in \{1, \dots, n\}$, $h_i = h$. It is similar to a Taylor development (See Appendix). The idea is to compute $u(t_{i+1})$ using the formula

$$u(t_{i+1}) \approx u(t_i) + h\frac{\partial u}{\partial t}(t_i) \tag{2}$$

where

$$\frac{\partial u}{\partial t}(t_i) = f(t_i, u(t_i)).$$

for a function $f$.

If we look back at the formula in the ResNet (see Equation 1), we can see that this is a special case of the formula for Euler method

$$x_{k+1} = x_k + hf_k(x_k),$$

when $h = 1$. It is with this observation that we can introduce neural ODE networks since ODE networks are inspired from this idea of using ode solver as implicit layers.

# 4 Neural ODE

### 4.1 Introduction

In a residual neural network, the output for an input $x$ is a composition of functions. We want to extract all these individual layers to only have one "shared" layer.

A *neural ODE network* (or ODE-Net) [2, 3, 9] takes a simple layer as a building block. This "base layer" is going to specify the dynamics of an ODE. ODE-Net enable us to replace layers of neural networks with a continuous-depth model. This means that we do not need to specify the number of layers beforehand.

Let us return to ResNets to give intuition behind this definition. We know that any output of the $k^{th}$ layer of a residual network can be computed with the function

$$F(z_t, t; \theta) = f(z_t, t; \theta) + z_t$$

where $t = k - 1$ and $\theta$ represents the parameters of the layers.

Thus, in the ResNet, the output for the input $z_0 = x$ is a composition of the functions $F(z_t, t; \theta)$.

We can then view the variables $z_t$ as a function $z$ of $t$. For example,

$$z_1 := z(1) = f(x, 0) + x.$$

With that, we can write $F(z_t, t; \theta) = F(z(t), t, \theta)$.

We can see that in ResNets, the outputs of each layer are the solutions of an ODE using Euler's method (see Section 3.2). The ODE from which it is a solution is $\frac{\partial z}{\partial t}(t) = f(z(t), t; \theta)$.

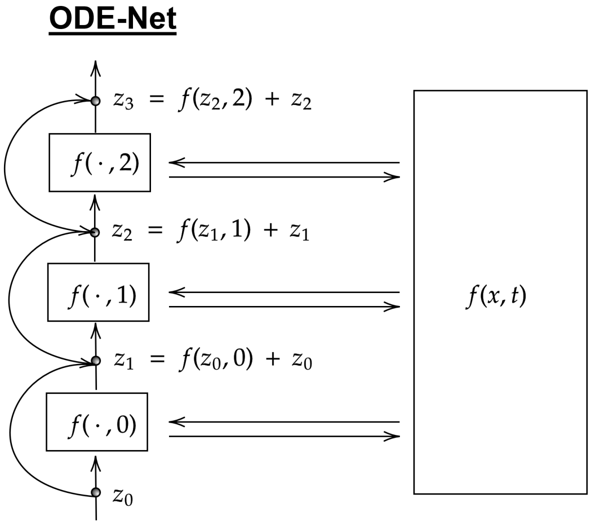

Figure 4: Representation of an ODE-Net

However, to find the solution to this Cauchy problem, we need the initial value of $z$, which is $z(t_0) := z_0 = x$ (the input). Here we want to use a more precise method and then use a more complex ODE solver such as linear multistep methods. With what we've just shown, it is possible.

We obtain the following Cauchy problem:

$$\begin{cases} \frac{\partial z}{\partial t}(t) = f(z(t), t; \theta) \\ z(t_0) = x \end{cases} \tag{3}$$

## 4.2 Forward pass

The layer in an ODE-Net is implicit. The output $z(t_N)$ of an ODE-Net with the input $z(t_0)$ is defined by the Cauchy problem (3). We see that the Cauchy problem depends on the parameters $z(t_0), t_0, \theta$.

But how do we solve this problem? We can simply use an ODE Solver with the parameters given above. In the case of the Euler method, the result is equivalent to a residual neural network, as we saw in Section 2.5.

To be able to use an ODE solver we need to make sure that the function satisfies the hypotheses in the theorem of existence and uniqueness (see Theorem 1).

## 4.3 Backward pass: the Adjoint method

Now that we know how to compute the output of an ODE-Net, we need a method to find the optimal parameters that minimize the loss function.

In regular neural networks, we usually use gradient descent. However in our case, it is more difficult because we used an ODE solver in the forward pass which is some sort of black box. This is why we use the *adjoint method* [2].

This method computes the gradient by solving a second ODE backwards and is applicable to all ODE solvers.

Let $\mathcal{L}$ be a loss function. To minimize this loss function, we need gradients with respect to the parameters $z(t_0), t_0, t_N, \theta$ as we explained in Section 2.2. To achieve that, we can determine how the gradient of the loss depends on the hidden state $z(t)$ for each $t$, which is

$$a(t) = \frac{\partial \mathcal{L}}{\partial z(t)}. \tag{4}$$

This quantity is called the *adjoint*. We would like to determine its dynamics, so we need to compute its derivative with respect to $t$.

With a continuous hidden state, we can write the transformation after an $\varepsilon$ change in time as :

$$z(t+\varepsilon) = \int_t^{t+\varepsilon} f(z(t), t, \theta)dt + z(t) \tag{5}$$

Let $G : \varepsilon \mapsto z(t+\varepsilon)$. We can apply the Chain rule to obtain

$$\frac{\partial \mathcal{L}}{\partial z(t)} = \frac{\partial \mathcal{L}}{\partial z(t+\varepsilon)} \frac{\partial z(t+\varepsilon)}{\partial z(t)}.$$

In other words

$$a(t) = a(t+\varepsilon)\frac{\partial G(\varepsilon)}{\partial z(t)} \tag{6}$$

We can now compute the derivative of $a(t)$ :

$$\begin{aligned}
\frac{\partial a}{\partial t}(t) &= \lim_{\varepsilon \to 0^+} \frac{a(t+\varepsilon) - a(t)}{\varepsilon} \\
&= \lim_{\varepsilon \to 0^+} \frac{a(t+\varepsilon) - a(t+\varepsilon)\frac{\partial G(\varepsilon)}{\partial z(t)}}{\varepsilon} \\
&= \lim_{\varepsilon \to 0^+} \frac{a(t+\varepsilon) - a(t+\varepsilon)\frac{\partial z(t) + \varepsilon f(z(t),t,\theta) + O(\varepsilon^2)}{\partial z(t)}}{\varepsilon} \\
&= \lim_{\varepsilon \to 0^+} \frac{a(t+\varepsilon) - a(t+\varepsilon)(\mathbb{1} + \varepsilon\frac{\partial f(z(t),t,\theta)}{\partial z(t)} + O(\varepsilon^2))}{\varepsilon} \\
&= \lim_{\varepsilon \to 0^+} \frac{-\varepsilon a(t+\varepsilon)\frac{\partial f(z(t),t,\theta)}{\partial z(t)} + O(\varepsilon^2)}{\varepsilon} \\
&= \lim_{\varepsilon \to 0^+} -a(t+\varepsilon)\frac{\partial f(z(t),t,\theta)}{\partial z(t)} + O(\varepsilon) \\
&= -a(t)\frac{\partial f(z(t),t,\theta)}{\partial z(t)}
\end{aligned}$$

We now have the dynamics of $a(t)$

$$\frac{\partial a(t)}{\partial t} = -a(t)\frac{\partial f(z(t),t,\theta)}{\partial z(t)} \tag{7}$$

As we are searching for $a(t_0) = \frac{\partial L}{\partial z(t_0)}$, we need to solve an ODE for the adjoint backwards in time because the value for $a(t_N)$ is already known. The constraint on the last time point, which is simply the gradient of the loss with respect to $z(t_N)$,

$$a(t_N) = \frac{\partial \mathcal{L}}{\partial z(t_N)},$$

has to be specified to the ODE solver. Then, the gradients with respect to the hidden state can be calculated at any time, including the initial value. We have

$$\begin{aligned} a(t_0) &= a(t_N) + \int_{t_N}^{t_0} \frac{\partial a(t)}{\partial t} dt \\ &= a(t_N) - \int_{t_N}^{t_0} a(t) \frac{\partial f(z(t), t, \theta)}{\partial z(t)} dt. \end{aligned}$$

If we want to compute the gradient with respect to the parameters $\theta$, we have to evaluate another integral, which depends on both $z(t)$ and $a(t)$. We compute it using the chain rule and we obtain

$$\frac{\partial L}{\partial \theta} = - \int_{t_N}^{t_0} a(t) \frac{\partial f(z(t), t, \theta)}{\partial \theta} dt. \tag{8}$$

To avoid computing each ODE on its own, we can do all of them at the same time. To do that we can generalize the ODE to

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

$$a_{aug}(t) := \begin{bmatrix} a \\ a_\theta \\ a_t \end{bmatrix} (t), \; a(t) = \frac{\partial \mathcal{L}}{\partial z(t)}, \; a_\theta(t) = \frac{\partial \mathcal{L}}{\partial \theta(t)}, \; a_t(t) := \frac{\partial \mathcal{L}}{\partial t(t)}.$$

The jacobian of $f_{aug}$ has the form

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

where each $\mathbf{0}$ is a matrix of zeros with the corresponding dimensions.

We can inject $a_{aug}$ in (7) and we get

$$\begin{aligned} \frac{\partial a_{aug}(t)}{\partial t} &= -[a(t) \; a_\theta(t) \; a_t(t)] \frac{\partial f_{aug}}{\partial [z(t), \theta, t]}([z(t), \theta, t]) \\ &= -\left[ a\frac{\partial f}{\partial z} \; a\frac{\partial f}{\partial \theta} \; a\frac{\partial f}{\partial t} \right](t). \end{aligned}$$

We can see that the first component $-a(t)\frac{\partial f(z(t), t, \theta)}{\partial z(t)}$ of the vector $\frac{\partial a_{aug}(t)}{\partial t}$ is the adjoint differential equation that we calculated previously in (7). The total gradient with respect to the parameters is given by integrating the second component, $-a(t)\frac{\partial f(z(t), t, \theta)}{\partial \theta(t)}$ over the full interval and by setting $a_\theta(t_N) = \mathbf{0}$. We obtain

$$\frac{\partial \mathcal{L}}{\partial \theta} = a_\theta(t_0) = - \int_{t_N}^{t_0} a(t) \frac{\partial f(z(t), t, \theta)}{\partial \theta} dt.$$

We can also get gradients with respect to $t_0$ and $t_N$ by integrating the last component, $-a(t)\frac{\partial f(z(t),t,\theta)}{\partial t(t)}$, and by using the Chain rule respectively. We have

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial t_0} &= a_t(t_0) = a_t(t_N) - \int_{t_N}^{t_0} a(t)\frac{\partial f(z(t),t,\theta)}{\partial t}dt;\\ \frac{\partial \mathcal{L}}{\partial t_N} &= \frac{\partial \mathcal{L}}{\partial z(t_N)}\frac{\partial z(t_N)}{\partial t_N} = a(t_N)f(z(t_N),t_N,\theta).\end{aligned}$$

With this generalized method, we have gradients of $\mathcal{L}$ for all possible inputs to a Cauchy problem solver. In the development above, we assumed that the loss function $\mathcal{L}$ depends only on the last time point $t_N$. If $\mathcal{L}$ depends also on intermediate time points $t_1, t_2, \ldots, t_{N-1}$, we can repeat the adjoint step for each of the intervals $[t_{N-1}, t_N], [t_{N-2}, t_{N-1}], \ldots, [t_0, t_1]$ in the backward order and sum up the obtained gradients.

We can represent this process with the Figure 5. As we can see, during the forward pass, the loss function is evaluated a each time to be able to determine the smallest error.

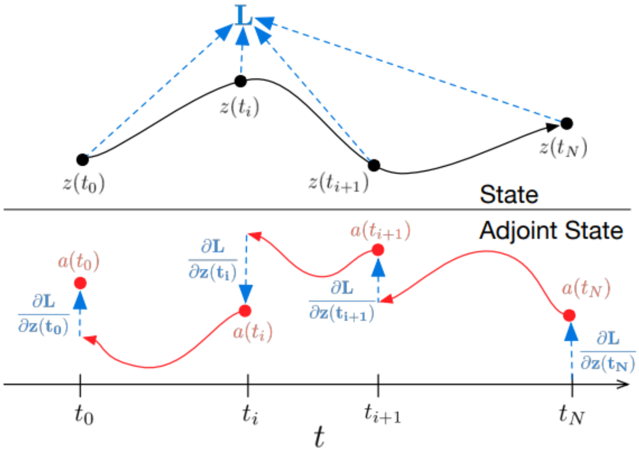
L
$z(t_i)$
$z(t_0)$
$z(t_{i+1})$
$z(t_N)$
State
Adjoint State
$a(t_{i+1})$
$a(t_0)$
$\frac{\partial L}{\partial z(t_i)}$
$\frac{\partial L}{\partial z(t_{i+1})}$
$a(t_N)$
$\frac{\partial L}{\partial z(t_0)}$
$a(t_i)$
$\frac{\partial L}{\partial z(t_N)}$
$t_0$
$t_i$
$t_{i+1}$
$t_N$
$t$

Figure 5: Graphical representation of the forward and backward pass for the ODE-Net. The figure has been taken from [2].

## 5 Simulated data

Let $h : \mathbb{R} \to \mathbb{R}$ be a function such that

$$h(x) = x^3 + 0.1x.$$

We can train a ResNet and an ODE-Net to approximate this function. To do that, we need a training set of points. We generate 10 points between $-2.5$ and 2.5. Their associated output comes from the function

$$h(x) + \varepsilon,$$

where $\varepsilon$ is a noise variable with mean 0 and standard deviation 1.

We train a ResNet with 3 layers, the hidden one having 20 neurons. Each layer has the hyperbolic tangent as activation function. The step-size used during the training was 0.01. After 1000 iterations, we get the function given in Figure 6. The training is fairly fast and lasts less than 5 minutes.

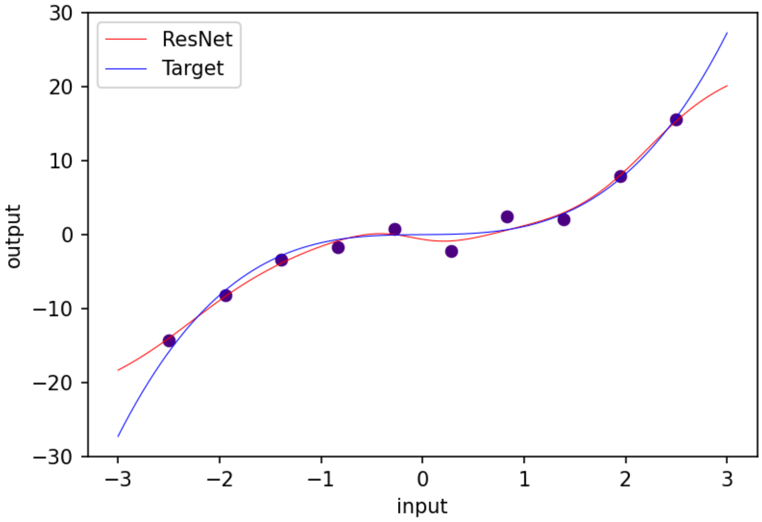

Figure 6: Result of the training for the ResNet

The purple points represent the data used for the training, the blue line is the function we want to approximate and the red line is the function represented by the ResNet. We tested the function returned by the ResNet on the points we used to trace the blue line. The out-of-sample error for these points is 4.673. The loss function was the mean squared error.

Now we will do the same for an ODE-Net. We use the same training data as for the ResNet.

The dynamics of the ODE-Net is specified by a layer of size 20. The step-size in this case was 0.005. Each layer has the hyperbolic tangent as activation function. After 1500 iterations, we get the function given in Figure 7. The training is slightly slower than the training of the ResNet, but is still relatively fast and lasts less than 5 minutes too.

The purple points represent the data used for the training, the blue line is the function we want to approximate and the red line is the function represented by the ODE-Net. With the mean squared error, we had an out-of-sample error of 7.72.

We can compare the results of both models and we can see that the ResNets is slightly better with these parameters. The comparison graph is in the Figure 8.

We can also train the models on different training data sets, generated randomly, to see how it effects the resulting approximation. To do that, we gen-

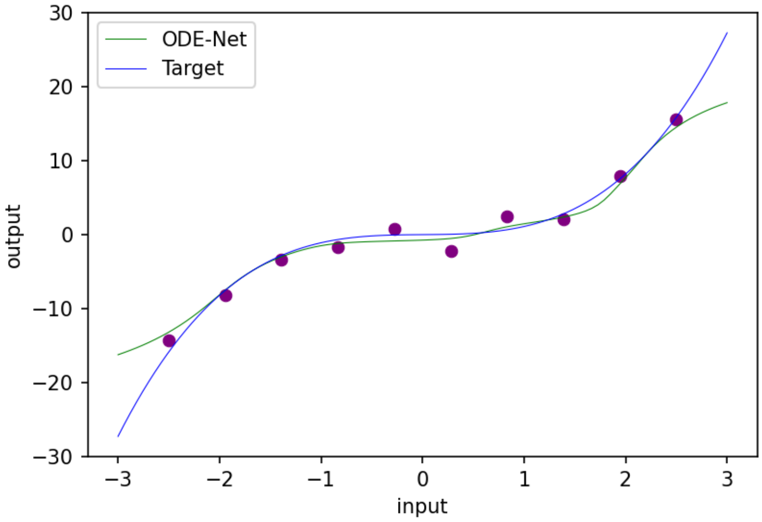

Figure 7: Result of the training for the ODE-Net

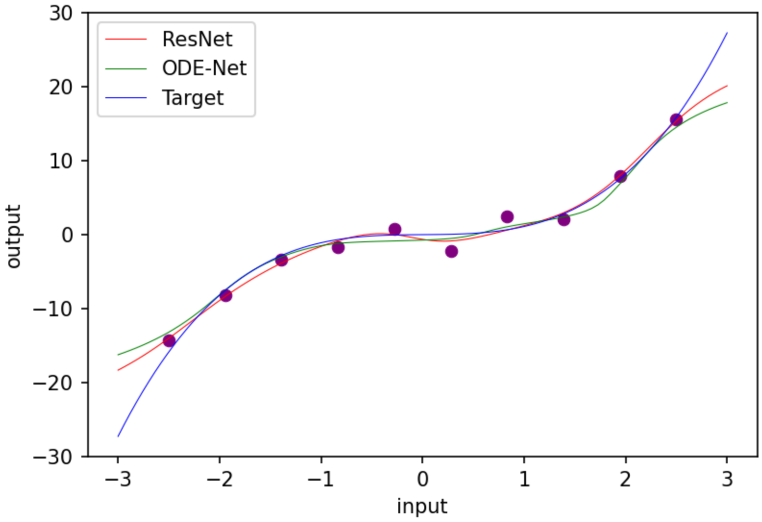

Figure 8: Comparison of ResNet and ODE-Net

erated 5 different training data sets of 20 points in the interval $[-3, 3]$ and we trained the models on those sets with the same parameters as before.

We can see the results of this in Figures 9 and 10. The resulting approximations are close to the target function and there is little variation between each trained model.

To make sure we are using the parameters that gives the "best" results, we can train models using different values of the step-size to see which one gives

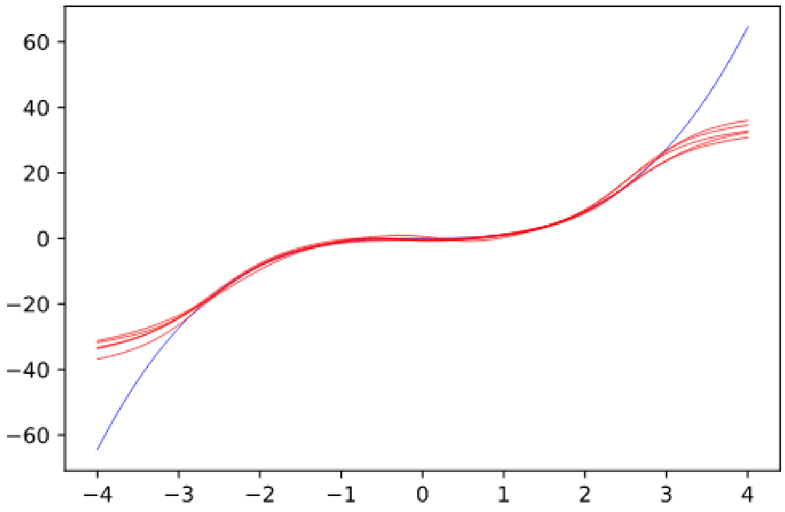

Figure 9: Approximations resulting from training a ResNet on different training datasets. The blue line is the target function and the red lines are ResNets.

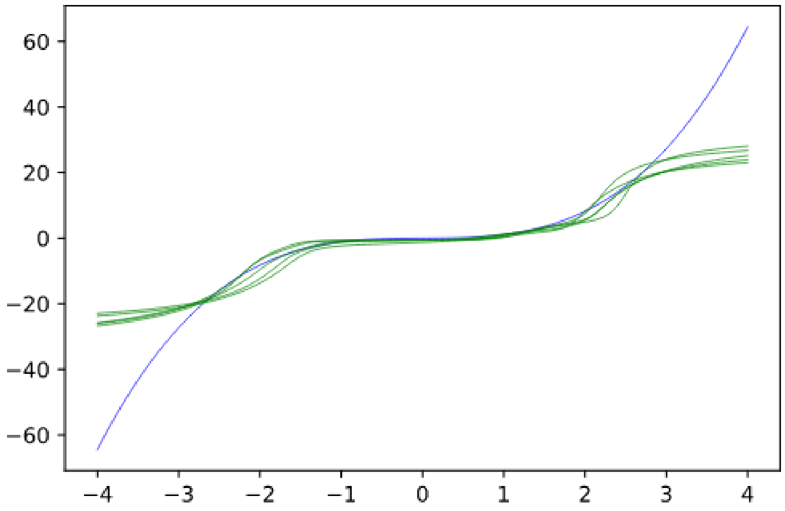

Figure 10: Approximations resulting from training an ODE-Net on different training datasets. The blue line is the target function and the green lines are ODE-Nets.

the best results.

- For the ResNet, we trained it on the following step-sizes: 0.001, 0.01 and 0.1. The results are in Figure 11;
- For the ODE-Net, we trained it on the following step-sizes: 0.0005, 0.001 and 0.005. Here we used smaller step-sizes because with a step-size equal or bigger than 0.01 an overflow occurs during training. The results are in

Figure 12.

We can see that for the ResNet, a step-size too small (0.001 and under) gives good results but needs more iterations to stabilise. But when we get to bigger step-sizes, 0.1 and over, the results start to become less accurate and the errors during training are very unstable.

The same observations can be made about the ODE-Net, with the exception that the errors become unstable with smaller step-sizes. Thus we need even more iterations for the ODE-Net to converge since we use a smaller step-size.

# 6 Example with real data

Now that we have tested the ODE-Net architecture on different small examples, we can test it on real data to see how it performs.

The data we chose is the MIT Beth Israel Hospital (BIH) electrocardiogram dataset that can be found at `https://www.kaggle.com/shayanfazeli/heartbeat`. It contains two data sets: one for training and one for testing.

The training dataset consists of around $90,000$ samples, classified as either

- 0: normal;
- 1: supraventricular premature beat;
- 2: premature ventricular contraction;
- 3: fusion of ventricular and normal beat;
- 4: unclassified beat;

and the test dataset is composed similarly, with around $20,000$ samples.

We will build a ResNet and an ODE-Net as similar as possible so that we can compare them better.

- The Resnet is composed of 3 first fully connected layers with ReLU as activation function, followed by 6 residual block also with ReLU. The output layer is a simple linear layer (with the input flattened).
- The ODE-Net is composed similarly of 3 first fully connected layers, each followed by the activation function ReLU. The dynamics are defined by a fully connected layer taking time into account. The output layer is the same as for the ResNet, i.e. a simple linear layer with the input flattened.

To optimize our model we use the cross entropy error. The parameters are then updated using stochastic gradient descent with a learning rate of 0.1 and a momentum of 0.9.[3] We trained each models on the training set for 25 epochs with batches of size 128. The training for the ODE-Net was again slower than for the ResNets. While the ResNet lasted less than 10 minutes, the ODE-Net lasted around 30 minutes.

[3] Stochastic Gradient Descent with momentum is a method that accelerate the computation of the gradient and thus leads to faster converging. [10]

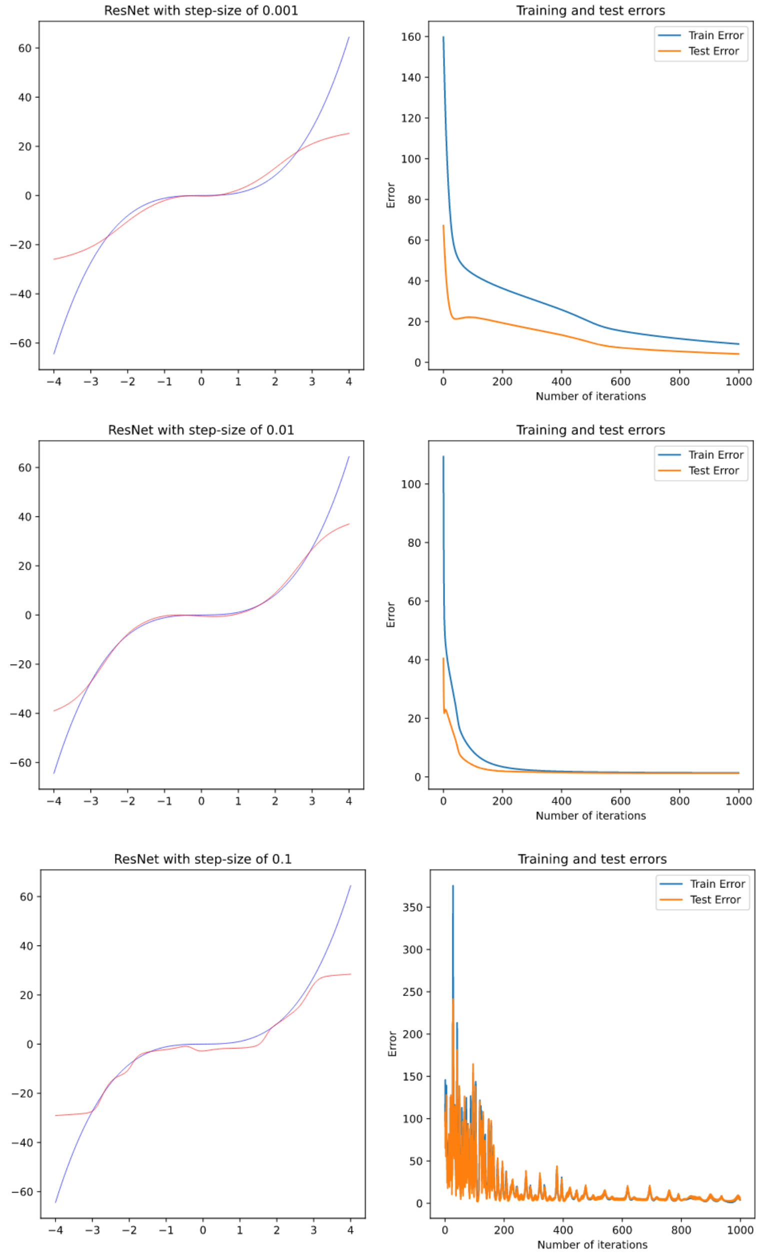

Figure 11: Comparison of ResNets trained with different step-sizes.

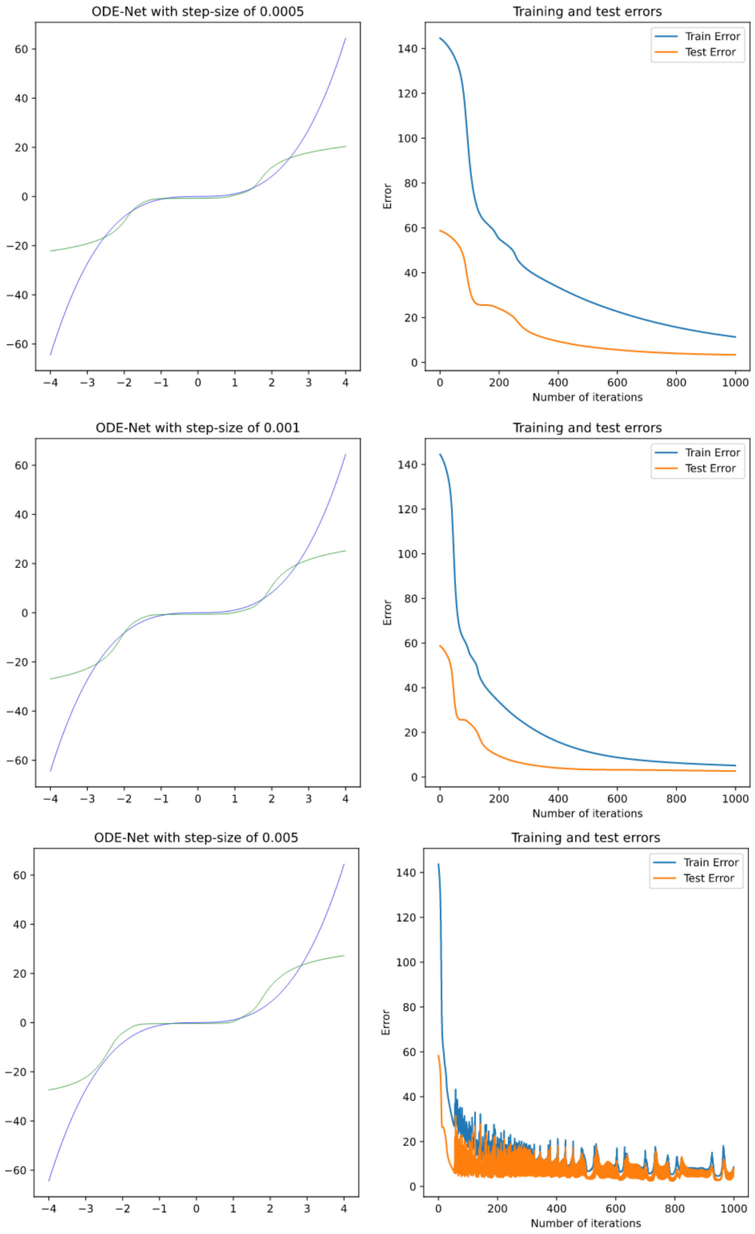

Figure 12: Comparison of ODE-Nets trained with different step-sizes.

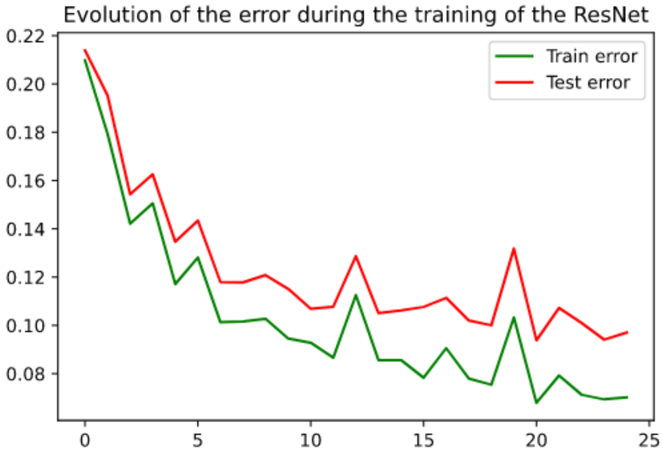

Figure 13: Evolution of the error during training of the ResNet

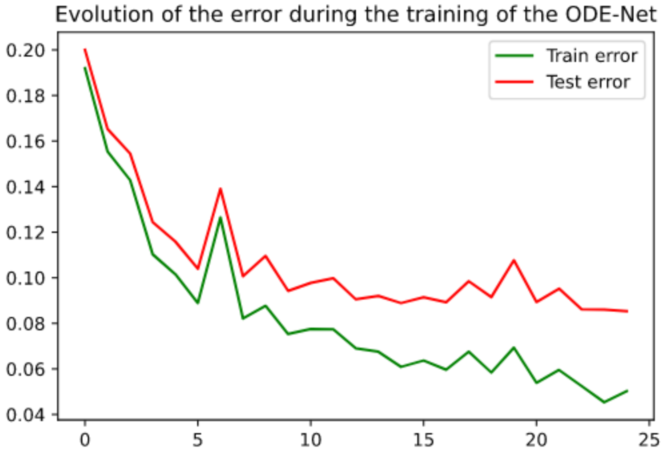

Figure 14: Evolution of the error during training of the ODE-Net

In Figure 13, we can see the evolution of the training and test loss. We tested our models on test data and used the cross entropy error to evaluate it. At the end of the training, the Resnet had a test error of 0.1.

For the ODE-Net, we can see the evolution of the loss on Figure 14. At the end of the training, the model had a test error of 0.09. Thus the ODE-Net is slightly better than the ResNet. Moreover, the figures show that the ODE-Net learns faster since the curve decreases in less epochs.

We also computed the accuracy of each models. To do so we first used softmax to compute predictions for the test data, then we used the zero-one loss on those predictions. The ResNet has an accuracy of 97.5% while the ODE-Net has a slightly better accuracy of 97.8%.

Another way of judging the quality of a model in the case of classification is to look at its *confusion matrix.* The confusion matrix is a matrix that has the same number of columns and lines as the number of classes. In position $(i, j)$ in the matrix is the number of point that are in the $i^{th}$ class and were classified in the $j^{th}$ class by the model. A good models needs to have high numbers on the main diagonal and numbers close to 0 elsewhere.

The confusion matrix for the ResNet is

$$\begin{pmatrix} 18056 & 12 & 36 & 5 & 9 \\ 209 & 339 & 5 & 1 & 2 \\ 99 & 8 & 1313 & 23 & 5 \\ 52 & 0 & 16 & 94 & 0 \\ 47 & 0 & 10 & 0 & 1551 \end{pmatrix}$$

and the confusion matrix for the ODE-Net is

$$\begin{pmatrix} 17989 & 43 & 74 & 3 & 9 \\ 151 & 389 & 15 & 0 & 1 \\ 59 & 2 & 1376 & 10 & 1 \\ 36 & 0 & 26 & 100 & 0 \\ 36 & 0 & 16 & 0 & 1556 \end{pmatrix}.$$

We can see that the ODE-Net is slightly better since the number on the main diagonal are, for the most part, higher than those in the ResNet. But both models have high numbers on the first columns, it means that they classify a lot of point int the class 0 when they aren't in this class.

Another thing we can notice is that the ODE-Net has trouble classifying points in the class 0 compared to the ResNet: when we look at the first line of each confusion matrix, the line from the ODE-Net has higher numbers in columns other than the first, which means that when points are in the class 0, it often classifies them wrong.

# 7 Advantages and disadvantages of ODE-Nets

### Advantages

- *ODE solvers*

  We can use ordinary differential equations solvers instead of gradient descent. These solvers have more than a hundred years of theory behind them which is a great advantage against gradient descent.

- *Robustness* [11]

  It was proved that ODE-Net are very robust against perturbed data compared to regular neural network. Two experiments were conducted: in the first one they trained an ODE-Net and a convolutional neural network[4] on real images without perturbations. They tested these models on the original images and the ODE-Net outperformed the CNN. In the second experiment, they trained these networks on the original and perturbed images. Again, the ODE-Net was much better.

[4] A neural network that is usually good with images.

- *Constant memory cost*

  Lastly, there's a constant memory cost, instead of increasing the cost linearly with each layer in a regular network. In ODE-Net, we know the state at every time $t$. Because of that, we can always reconstruct the entire trajectory of an ODE forwards and backwards in time only by knowing this point. This means that ODE-Nets can be trained with a memory cost constant in the number of evaluations of $f$. There is a trade-off between memory cost and computational time: ResNets are faster but use more memory and ODE-Net are slower but use less memory.

- *Continuous time series predictions*

  The biggest advantage of ODE-Nets is that they have more accurate results for time series predictions. Regular neural network have discrete layers, which means they expect the intervals for these time series data sets to be fixed whereas ODE-Net have a continuous layer which means we can evaluate the hidden states at every point $t$ in time. Therefore, regular neural networks are bad at predicting output for time series data that is irregular.

**Disadvantages**

- *Slower training time*

  ODE-Net have a slower training time. Indeed, during training, the dynamics we want to learn tend to become expensive to solve since the network becomes deeper. However, regular neural networks can be evaluated with a fixed amount of computation, and are typically faster to train. In this case, we don't have to choose an error tolerance for a solver.

  There is then a trade-off between accuracy and computational time: if we choose a small error tolerance, then the computational time be bigger.

- *More Hyperparameters*

  In ODE-Nets we need to choose a solver and its error tolerance, which induces more choices to find the parameters which works better.

- *Restriction on activation functions*

  To ensure that the ODE has a solution we have to make sure the dynamics are uniformly continuous Lipschitz (see Theorem 1). This is why we mostly use tanh as an activation function.

# 8 Appendix

**Definition 2.** Let

$$\begin{array}{cccc} f: & \mathbb{R}^n & \to & \mathbb{R}^m \\ & x=(x_1,\ldots,x_n) & \mapsto & f(x_1,\ldots,x_n) \end{array}$$

be a function.

The *partial derivative* of $f$ with respect to the variable $x_i$ is denoted by

$$\frac{\partial f}{\partial x_i}:\mathbb{R}^n \to \mathbb{R}^m.$$

For $a \in \mathbb{R}^n$, the partial derivative of $f$ with respect to $x_i$, if it exists, is defined as

$$\frac{\partial f}{\partial x_i}(a)=\lim_{h\to 0}\frac{f(a_1,\ldots,a_{i-1},a_i+h,a_{i+1},\ldots a_n)-f(a_1,\ldots,a_i,\ldots,a_n)}{h}.$$

**Notation 1.** Let $f:\mathbb{R}^n \to \mathbb{R}$ a function and $x \in \mathbb{R}^n$. We write

$$\nabla f(x)=\begin{pmatrix} \frac{\partial f}{\partial x_1}(x) \\ \vdots \\ \frac{\partial f}{\partial x_n}(x) \end{pmatrix}.$$

This is called the *gradient* of $f$ at $x$.

**Definition 3.** Let $f:\mathbb{R}\to\mathbb{R}$ be a function and $a\in\mathbb{R}$. We can write the *first-order Taylor's development* for $f$ at $x$ as :

$$f(x+a)=f(x)+a.\partial f(x)+O(\|a\|_2^2).$$

**Definition 4.** Le $f:\mathbb{R}^n\to\mathbb{R}$ be function, $n\geq 2$. Then $f$ is *convex* if and only if

$$\forall 0\leq t\leq 1, \forall x_1,x_2\in\mathbb{R}^n, \quad f(tx_1+(1-t)x_2)\leq tf(x_1)+(1-t)f(x_2).$$

**Lemma 1.** *If $f$ is a convex function, then every local optimum is a global minimum.*

**Definition 5.** If $f$ and $g$ are differentiable functions, then the *chain rule* expresses the derivative of their composite $f\circ g$ in terms of the derivatives of $f$ and $g$ and the product of functions as follows:

$$\frac{\partial f\circ g}{\partial x}=\Big(\frac{\partial f}{\partial x}\circ g\Big)\frac{\partial g}{\partial x}.$$

**Definition 6.** Let $L\in\mathbb{R}^+$. A smooth function $f:\mathbb{R}^d\to\mathbb{R}$ is $L$-Lipschitz if for all $x,y\in\mathbb{R}^d$,

$$\|\nabla f(x)-\nabla f(y)\|_2 \leqslant L\|x-y\|_2.$$

**Theorem 2.** *If $f:\mathbb{R}^d\to\mathbb{R}$ is a function $L$-Lipschitz convex and if $x^*=argmin_x f(x)$, then the gradient descent algorithm with step-size $\eta\leqslant\frac{1}{L}$ satisfies*

$$f(x_k)\leqslant f(x^*)+\frac{\|x_0-x^*\|_2}{2\eta k}.$$

# References

[1] Yaser S. Abu-Mostafa, Malik Magdon-Ismael, and Hsuan-Tien Lin. Learning from data: A short course. 2012.

[2] Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. Neural ordinary differential equations, 2019.

[3] Ayan Das. Neural ordinary differential equation (neural ode). `https://ayandas.me/blog-tut/2020/03/20/neural-ode.html`.

[4] David F. Griffiths and Desmond J. Higham. *Numerical Methods for Ordinary Differential Equations Initial Value Problems.* Springer Undergraduate Mathematics Series. Springer, London, 2010.

[5] Roger Grosse. Csc321 lecture 6: Backpropagation. `http://www.cs.toronto.edu/~rgrosse/courses/csc321_2017/slides/lec6.pdf`.

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[7] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6:107–116, 04 1998.

[8] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. An introduction to statistical learning with applications in r. Springer.

[9] Zico Kolter, David Duvenaud, and Matt Johnson. Deep implicit layers - neural odes, deep equilibirum models, and beyond. `https://implicit-layers-tutorial.org/`.

[10] Ali Ramezani-Kebrya, Ashish Khisti, and Ben Liang. On the generalization of stochastic gradient descent with momentum. *CoRR*, abs/2102.13653, 2021.

[11] Hanshu Yan, Jiawei Du, Vincent Tan, and Jiashi Feng. On robustness of neural ordinary differential equations. `https://openreview.net/forum?id=B1e9Y2NYvS`.

[12] Amer Zayegh and Nizar Bassam. *Neural Network Principles and Applications.* 11 2018.