UNIVERSITAT DE BARCELONA

Facultat de Matemàtiques
i Informàtica

DOBLE GRAU DE MATEMÀTIQUES I ENGINYERIA INFORMÀTICA

Treball final de grau

AN INTRODUCTION TO NEURAL ORDINARY DIFFERENTIAL EQUATIONS

Autor: Pau Baldillou Salse

Director: Dr. Alex Haro
Dr. Jordi Vitrià
Realitzat a: Departament de
Matemàtiques i Informàtica

Barcelona, 17 de gener de 2024

Contents

Abstract

This project introduces the concept of neural ordinary differential equations as well as some of its practical uses. To do so, it provides a review of machine learning and ordinary differential equations which allow the rest of the discussion to be well founded and understood by readers of different backgrounds.

Neural ODEs are an exciting and interesting field because they manage to bring together the two modelling paradigms of neural networks and differential equations. Apart from their theoretical relevance in linking these fields, they are very promising for their applications. The incorporation of a differential structure into the models simplifies crucial aspects that allow the models to be more complex and expressive.

In spite of not producing new results, this project includes a compilation of experiments and demonstrations that aim at making the jump from theory to practice smoother. Generally, this work tries to be an accessible introduction to the topic, while being extensive and maintaining a high level of mathematical formalism.

The work than conforms this project allows one to conclude that neural ODEs are a promising development in the realm of machine learning. They can be very useful to solve problems such as probability density estimation, with applications in generative models. Moreover, their theoretical properties alone make them a topic worth studying.

2020 Mathematics Subject Classification. 34F05, 34A12, 76A02, 68T07, 65L05, 62G05, 62G07

Resum

Aquest projecte introdueix el concepte d'equació diferencial ordinària neuronal així com alguns dels seus usos pràctics. Per fer-ho, proporciona una revisió de "machine learning" i equacions diferencials ordinàries que permet que la resta de la discussió estigui ben fonamentada i sigui entesa per lectors amb diferents experiències.

Les EDOs neuronals són un camp fascinant i interessant perquè aconsegueixen unir els dos paradigmes de modelatge que són les xarxes neuronals i les equacions diferencials. A part de la seva rellevància teòrica per unir aquests camps, prometen molt per les seves aplicacions. La incorporació d'una estructura diferencial al model simplifica aspectes crucials que permeten que els models siguin més complexes i expressius.

A pesar de no produir nous resultats, aquest projecte inclou una recopilació d'experiments i mostres que tenen com a intenció fer el salt de la teoria a la pràctica més fàcil. Generalment, aquest treball intenta ser una introducció accessible, a la vegada que exhaustiva i mantenint un nivell alt de formalisme matemàtic.

La feina que conforma aquest projecte permet concloure que les EDOs neuronals són un desenvolupament prometedor en el terreny de "machine learning". Poden ser molt útils per resoldre problemes com l'estimació de densitats de probabilitats, amb aplicacions a models generatius. A més a més, simplement les seves propietats teòriques les fan un tema que val la pena estudiar.

Chapter 1

Introduction

Neural Ordinary Differential Equations are a bridge between modern deep learning and classical mathematical modelling. As a conjoining of two of the most ubiquitous modelling paradigms, they are of great theoretical interest. For example, they illustrate that some neural network architectures are a discretisation of differential equations. Moreover, their practical uses make them an exciting and promising field.

Neural Differential Equations were brought to the spotlight in 2018 by Chen et al. in their paper *Neural Ordinary Differential Equations* [6]. However, the topic was not new: dynamical systems theory was being used to improve neural network performance for some time [22, 15, 29]. Since then, there have been developments in their approximation capabilities [10, 19] and usage in different tasks like generative models [6, 12] or time-series modelling [6, 24, 21], to name a few. This project is centred around NODEs and their usage for probabilistic modelling via continuous normalising flows.

The objectives of this work are focused on four main goals:

- Understand what neural ODEs are and present a detailed explanation to introduce them to the new reader.
- Formalise the concepts present in neural ODE theory and examine the topic through a mathematical lens.
- Research a probability density estimation model and how it can be improved with neural ODEs.
- Provide examples of how this family of models can be used in an approachable manner so the code and algorithms are easy to understand for inexperienced readers.

For that purpose, this dissertation is organised in three main chapters, and two appendices:

- **Chapter 2** provides an introduction to machine learning and a review of ODE theory. Some important concepts are explored here, like automatic differentiation or relevant neural network architectures. Furthermore, it lays a foundation to formalise the concepts in the next chapters.
- **Chapter 3** defines Neural ODEs and provides a theoretical basis for their usage and properties.
- **Chapter 4** introduces normalising flows, a machine learning algorithm capable of learning complex probability distributions, and shows how it can be improved by using their continuous-time analogous.
- **Appendix A** explains two numerical methods used to solved initial value problems.
- **Appendix B** explains the code written as part of this project.

The experiments that have also been conducted as part of this work are a practical proof of concept. Their main goal is to provide a better insight into how the models studied here work. The code for these experiments can be found here[1] and all of the figures in this document were generated by it or by the author using PowerPoint.

[1] https://github.com/pbaldisa/neural-odes/tree/main

Chapter 2

Machine Learning and Ordinary Differential Equations

This chapter serves as introduction to the two fields that conform the main topic of the present work: machine learning and differential equations. It starts with a detailed explanation of what is machine learning and neural networks in particular and then goes into some important results in ODE theory.

2.1 Machine Learning

According to Zhi-Hua Zhou [34], "*Machine learning is the technique that improves system performance by learning from experience via computational methods. [...] the main task of machine learning is to develop learning algorithms that build models from data.*"

Tom Mitchell gave a more formal definition [23]: "*A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.*"

In other words, the goal of Machine Learning (ML) is to use data to create an algorithm or model that performs a certain task, of which its performance can be measured. Then, the learning part is simply the process in which one can find such a model by optimising the measured performance using the available data. The main goal of ML is not simply to find the model that best fits the current data, but one that can predict the outcome to new unseen data. It is about predicting the future, not fitting the present.

2.1.1 Conceptualising the problem

2.1.1.1 Notation and terminology

The information that is known and one uses at the time of prediction is called *input data*. Normally, it is a vector and from now on it will be assumed to be in $\mathbb{R}^d$. What one aspires to predict is called *output data*. Depending on the task this can be a quantitative or a categorical value. For the sake of simplicity, it is assumed to be a numerical value in $\mathbb{R}$, even though it can also be a vector.

It is convenient to understand the input data as samples generated by a random vector denoted as X, where the components are written as X_j and are themselves random variables. Observed values are written in lowercase; i.e. the i-th observation of X is $x_i \in \mathbb{R}^d$. Let the total number of observations be N. It is customary to write N-vectors and matrices in bold in order to distinguish them. Hence, one would write $\mathbf{x}$ as the $N \times d$ matrix of all input vectors, and $\mathbf{x_j} \in \mathbb{R}^N$ all observations of variable X_j. Column vectors are used, so the i-th row of $\mathbf{x}$ is x_i^T.

Predicted outputs are written as $\hat{y}_i$. Evidently, all predicted values should live in the same space as observed outputs. $\hat{Y}$ is used to denote the random variable that produces the predictions, i.e. the distribution the model generates.

In ML, one normally divides all the available data into various sets: *training set*, *test set* and *validation set* (see remark 2.3). The training data is the set of pairs $(x_1, y_1), ..., (x_N, y_N) \in \mathbb{R}^d \times \mathbb{R}$ that is used to construct the model. The other two are used to make sure the model generalises, since they are not as relevant when it comes to defining the problem, they might be left without specific notation.

Remark 2.1. In this section, for the description of the training data it has been assumed that one is facing a *supervised learning* problem in which there are outputs one wishes to predict. This is not always the case, as in *unsupervised learning* one might want to extract information about data without observed outcomes.

2.1.1.2 Machine Learning and mathematical optimisation

In order to further formalise the problem, let $(\Omega, \mathcal{F}, \mathbb{P})$ and $(\Omega', \mathcal{F}, \mathbb{P})$ be probability spaces and let the random variables X and Y such that $X : \Omega \to \mathbb{R}^d$ and $Y : \Omega' \to \mathbb{R}$.

Then, the model one wishes to build can be seen as a mapping between the input and output spaces: $f_\theta : \mathbb{R}^d \to \mathbb{R}$. From now on, $\theta \in \mathbb{R}^p$ denotes the parameters that define the model. For instance, in linear regression with $d = 1$, the parameters would be the slope and the y-intercept of the line.

Since the goal for f_θ to predict outcomes as well as possible, a valid approach is to fit it to the observed data. To do so, a *loss function* that represents the error

of the model is defined. The loss function is a measure of the difference between reality and the model's predictions, a usual notation is $\mathcal{L}_\theta$. Notice that the loss function depends on θ because the prediction is the output of f_θ.

The problem has been reduced to mathematical optimisation: one wishes to find which parameters minimise a function. Concretely, one wants the parameters that minimise $\mathcal{L}_\theta$. To be precise, one looks for $\mu = \arg\min_\theta \mathcal{L}_\theta(\hat{\mathbf{y}})$, where $\hat{\mathbf{y}}$ denotes the predicted output and has components $\hat{y}_i = f_\theta(x_i)$ for $i \in 1, \ldots, N$.

Remark 2.2. In this section one has not defined a particular loss function. Furthermore, the notation has been left rather general. This is because there are many functions to choose from depending on the task and the kind of data one has. In addition, the model can be as complex as one can imagine. These two facts make the problem of finding a minimum not trivial.

Remark 2.3. Notice how one is minimising the error for the observed data. This might not generalise to other unobserved inputs. To solve this problems the available data is divided into training, testing and validation sets. The first one being the described above which is used to find the parameters. The other two are used to evaluate the resulting model, decide whether it is good and help choose hyperparameters.

2.1.2 The loss function

Also referred to as *cost function*, it is simply a measure of 'distance'[1] between the model's output and reality. This similarity, or lack thereof, is a real number. That is, $\mathcal{L} : V \to \mathbb{R}$, if $\mathcal{L}$ is the loss function and V its domain[2]. Selecting a loss function is a problem in itself, since there is no "one fits all" solution, and it must be chosen with the task in mind, but also the learning algorithm that will be used. In this section, different examples of loss functions will be provided.

2.1.2.1 Quadratic loss

Known as *Mean Square Error* (MSE) or L_2*-loss*, this is the most popular cost function for regression problems. The mathematical formulation is

$$MSE(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 \tag{2.1}$$

[1] Here *distance* should not be understood as a mathematically precise notion, but rather as a real number that expresses how much two objects differ from one another.

[2] The domain of the loss function might be only the predictions' space or it might include also the observations.

It is the average of the squared differences between observations and predicted values. It is only concerned with the general extent of the error, not its direction (whether positive or negative). Moreover, large deviations are punished more harshly than small ones because of the square.

This function is widely used because of its nice properties. Namely, it is differentiable and easy to find its gradient. This is vital in modern Deep Learning applications which use gradient methods to optimise the cost function.

2.1.2.2 Kullback-Leibler divergence

Sometimes called *relative entropy*, it defines a distance between two probability distributions over the same sample space. It has a wide range of uses including, but not limited to, pattern recognitions, neural networks, information theory and mathematical statistics.

Let $(\Omega, \mathcal{A})$ be a measurable space with sample space Ω. Suppose P and Q are two probability distributions defined over Ω. If P is absolutely continuous[3] with respect to Q, then the Kullback-Leibler divergence is

$$D_{KL}(P\|Q) = \mathbb{E}_{P(x)}[\log P(x) - \log Q(x)] \tag{2.2}$$

otherwise, $D_{KL}(P\|Q) = +\infty$.

To be more specific, let p and q be probability mass functions for P and Q if they are distributions of discrete random variables, and densities if they are distributions of continuous random variables. Then, in the discrete case one has that equation 2.2 is equivalent to

$$D_{KL}(P\|Q) = \sum_{x\in\Omega} p(x)\log(\frac{p(x)}{q(x)}) = -\sum_{x\in\Omega} p(x)\log(\frac{q(x)}{p(x)}) \tag{2.3}$$

and, in the continuous case

$$D_{KL}(P\|Q) = \int_{-\infty}^{\infty} p(x)\log\frac{p(x)}{q(x)}dx$$

Remark 2.4. In the discrete case (equation 2.3) all terms with $p(x) = 0$ are interpreted as 0 because $\lim_{x\to 0^+} x\log x = 0$.

Relative entropy is an example of a loss function that is used for density estimation tasks. In [6] the authors use it to train *Continuous Normalising Flows*, where the task lacks observed data for comparing predictions.

[3] If P and Q are two measures on the same measurable space $(\Omega, \mathcal{A})$, P is *absolutely continuous* with respect to Q if $\forall A \in \mathcal{A}\,(Q(A) = 0 \implies P(A) = 0)$.

2.1.3 Neural Networks

One of the most ubiquitous model types in ML is the *Neural Network*. The concept is rather general and there are numerous architectures. In this section, one presents the main idea behind this model, as well as some specific architectures that will be of use in the following chapters.

Neural Network is an umbrella term used for a plethora of non-linear models that are usually represented with a network diagram. This kind of system is modelled after the biological neurons in our brain. The central principal can be stated as: each unit or neuron receives some inputs and outputs a signal to other interconnected neurons based on those inputs and an activation function. Neurons are organised in layers, so that the information travels from layer to layer as described above.

More formally, a neuron is simply a system that does the following computation:

$$y = \sigma(\sum_i w_i x_i + b) = \sigma(w^T x + b) \tag{2.4}$$

where $x = (x_1, ..., x_n)^T$ is the input vector, $w = (w_1, ..., w_n)^T$ the weights of the neuron, b is the bias term, σ the activation function, and y denotes the neuron's output.

Let one start with an overview of some commonly used activation functions. Then, some specific models belonging to what is known as *Deep Learning* (DL) will be presented. DL only means that there are more than two layers.

2.1.3.1 Activation functions

Recalling the definition of a basic neuron in equation 2.4, the activation function is an element-wise[4] non-linear function that acts on the result of the linear part of the neuron. In other words, the inputs and the weights get multiplied and added together. Then the result is passed through the activation function. Going back to the biological model, the idea behind σ is that it decides whether the neuron activates or not, given the current inputs.

Notice that it is rather important for the activation function to be non-linear. Otherwise, the neural network would be a composition of linear functions which is, in its turn, only a linear function with a very limited approximation capacity. As is explored in section 2.1.3.2, neural networks are trained using their gradients. Therefore, it is an important condition for activation functions to have "well-behaved" gradients.

[4]This is important when looking at the whole layer as a computation with vectors. If looking at the equation of a single neuron, it is a function that acts on a real value.

The rest of the section is a compendium of popular activation functions.

Sigmoid Also known as *logistic function*, the basic idea is that it is a smooth transition from 0 to 1. It is defined as

$$f(x) = \frac{1}{1 + e^{-x}}$$

This activation function introduces the problem of vanishing gradient as the network gets deeper. Since the derivative of f is nearly zero for big inputs, the gradient of the network becomes very small and the model stops learning.

Hyperbolic tangent It is an analogue of the ordinary tangent, but defined using the hyperbola instead of the circle. The most common way to define it is using the exponential function:

$$\tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Its properties are very similar to those of the sigmoid function, and has the same problem with vanishing gradient. The most noticeable difference is the image of the function is $(-1, 1)$ instead of $(0, 1)$. This is why it normally works better than sigmoid: it is zero-centred, which means not all outputs have the same sign and thus the network is easier to train [25].

Rectified linear unit (ReLU) The Rectified Linear Unit is perhaps the most commonly used activation function for hidden layers. It is defined as

$$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{otherwise} \end{cases} = \max(0, x)$$

It is simple and overcomes the limitations of the sigmoid and hyperbolic tangent by being less susceptible to vanishing gradients. However, it is not zero-centred and it is not differentiable at 0. Moreover, some neurons might die. That is, once the output of the linear phase of the neuron becomes negative, it will stop learning because the derivative for any negative value is zero.

Softmax In contrast with the other functions in this section, the softmax function is only used as an output function. Its output is a vector of values that sum to 1 and can be interpreted as the probabilities of the input belonging to a certain class. This is why it is used in classification tasks. In this case, the input of the function is a vector corresponding to the outputs of the previous layer. It is defined as

$$f(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$$

where z is a n-vector and $f(z)_i$ represents the i-th component of output.

2.1.3.2 Training

When talking about Neural Networks, as with other Machine Learning algorithms, to train the model means to find the parameters' values that get the best performance possible, as described in section 2.1.1.2. There are different methods to do it, and it is still one of the biggest problems in the field.

To train Neural Networks, a rather simple mathematical method is used: gradient descent. It is an iterative method that, given an error function with respect to the network's weights, tries to minimise the loss by moving in the direction of greatest local descent. That is, the direction opposite to the gradient.

Gradient descent Sometimes referred to as *steepest descent*, *gradient descent* is a first-order iterative optimisation method that intends to find a local minimum of a differentiable function. Here, first-order only means that the method is based in a linear approximation, using the first derivative. On the other hand, iterative refers to the fact its based on the construction of a sequence which, under the right conditions, converges to a solution. Here, this method will be used to find the parameters that minimise the loss function.

More formally, if $\mathcal{L} : V \to \mathbb{R}$ is a multivariate, differentiable function, then one can define the sequence $\mu_{n+1} = \mu_n - \gamma \nabla \mathcal{L}(\mu_n)$, where $\gamma \in \mathbb{R}_+$ is called learning rate.

If the learning rate is small enough, the previous collection $\{\mu_n\}$ gives rise to a monotonic sequence $\mathcal{L}(\mu_0) \geq \mathcal{L}(\mu_1) \geq ...$, where μ_0 is an initial guess for a local minimum. Under the right conditions for $\mathcal{L}$ and particular choices of γ, one can guarantee that the previous sequence converges to a local minimum. For example, the following theorem sets some conditions for convergence:

Theorem 2.5. *Let $\mathcal{L} : \mathbb{R}^d \to \mathbb{R}$ be a convex and differentiable functions, and suppose its gradient is Lipschitz continuous with constant $C > 0$. Then, computing k iterations of gradient descent with a fixed learning rate $\gamma \leq 1/C$ will yield an approximation μ_k of the minimum μ_* that satisfies*

$$\mathcal{L}(\mu_k) - \mathcal{L}(\mu_*) \leq \frac{\|\mu_0 - \mu_k\|^2}{2\gamma k},$$

where the norm is the euclidean norm. This means that the sequence generated by the gradient descent algorithm converges and it does so at a rate of $\mathcal{O}(1/k)$.

Proof. See [31], Theorem 6.1. □

Notice how this method, although simple, generates many problems and uncertainties. For once, the fact in only searches for a local minimum might be

discouraging when one would like to have the best performance possible, i.e. a global minimum. Another problem that comes to mind is how to compute the gradient when the network might have thousands of parameters. This issues are analysed in the following sections.

Automatic differentiation To compute the gradient of a function using a computer, there are three possible approaches:

1. **Symbolic differentiation**: First differentiate the function by hand, then code the result to get the derivative at any point. This gives an exact result with only machine error.

2. **Numerical differentiation**: Use a numerical method to approximate the result, like finite differences. This approach has an error of approximation.

3. **Automatic differentiation** (autodiff): Exploits the fact that all computer calculations can be divided into a set of simple operations and applies the chain rule to find the value of the derivative at each point. It also gives an exact result with only representation error.

Notice that a great difference between symbolic and automatic differentiation is that in the former, one knows the complete formula of the derivative, while in the latter, the symbolic derivative is unknown and the values are computed at each point.

To train neural networks, only automatic differentiation is viable because of the great number of parameters. Specifically, *reverse-mode autodiff* is used, which is very related to the concept of backpropagation. But reverse-mode autodiff is a much more general and older method [13] that computes the Jacobian of an arbitrary function $f : \mathbb{R}^m \to \mathbb{R}^n$, and even algorithms making use of control flow such as branching, loops, recursion and procedure calls [3, 8].

Let $f_1, \ldots, f_n$ be a collection of functions whose derivatives are known. Let those functions be called *differentiable primitives*. For any composition $f(x) = f_{i_m}(\ldots(f_{i_1}(x)))$, where $i_1, \ldots, i_m \in \{1, \ldots, n\}$, the chain rule states:

$$\frac{df}{dx} = \frac{df_{i_m}}{df_{i_{m-1}}} \cdots \frac{df_{i_2}}{df_{i_1}} \frac{df_{i_1}}{dx} \tag{2.5}$$

Given this, reverse-mode autodifferentiation consists of recursively computing

$$\frac{df_{i_m}}{df_{i_{q-1}}} = \frac{df_{i_m}}{df_{i_q}} \frac{df_{i_q}}{df_{i_{q-1}}} \tag{2.6}$$

for $q = m - 1, \ldots, 1$, and writing $x = f_0$.

Consider the following example borrowed from [14], Appendix B. Suppose one wants to find the gradient of $f(x,y) = x^2y + y + 2$ at $(x,y) = (1,2)$. Analytically, it directly follows that $\nabla f(x,y) = (2xy, x^2+1)$ and $\nabla f(1,2) = (4,2)$. To use reverse-mode autodiffernetiation in this example, one only needs to consider the constant function, addition, multiplication and the variables, all whose derivatives are known.

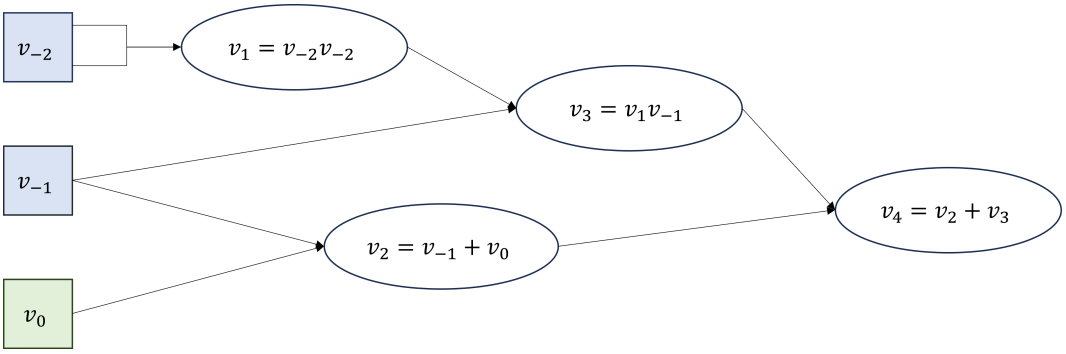


Figure 2.1: Computational graph. This shows the computational graph for $f(x,y) = x^2y + y + 2$. Blue nodes are variables, and green ones are constants.

The first step is to build a computational graph. This is simply a visual way to display the deconstruction of f into a composition of differentiable primitives. In figure 2.1 there is an example of such a graph using the example function. In the example, $v_{-2} = x, v_{-1} = y$ and $v_0 = 2$. Likewise, the notation v_i has been used to denote each composition, until $f = v_4$. Notice how each node represents one of the differentiable primitives mentioned above.

The following step is called *forward pass*, which is simply the computation of $f(x,y)|_{(x,y)=(1,2)}$, and all intermediate values. This goes, if using the graph in figure 2.1, from left to right. This calculation is displayed in figure 2.2, where one can appreciate the sequence v_1, v_2, v_3, v_4 being computed in that order. These values are stored for future reference, since they are going to be used in the next stage.

Finally, one has all the tools necessary to find the derivative. This last step is called *backward pass* because it goes from right to left, in the opposite direction as the previous stage. For simplicity, let $\overline{v_i} = \frac{df}{dv_i}$, which is only a new way of writing equation 2.6. Notice, however, that in equation 2.6, the function f only depends on x, while generally it can depend on an arbitrary number of variables.

This is the most critical part of the process, so a detailed explanation of the computations is given in order. The results can be visualised in figure 2.3.

1. $\overline{v_4} = \frac{df}{dv_4} = \frac{dv_4}{dv_4} = 1$

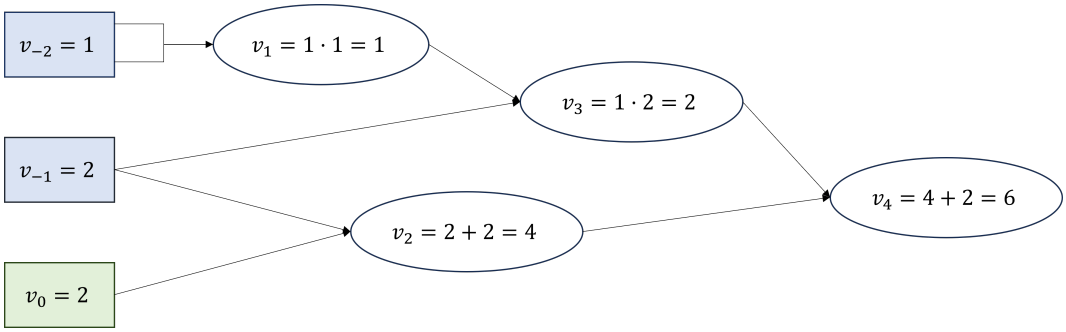


Figure 2.2: Forward pass. It is the same computational graph as in figure 2.1, but with the values of the evaluation of the function at each step.

2. $\overline{v_3} = \frac{df}{dv_3} = \frac{df}{dv_4}\frac{dv_4}{dv_3} = \overline{v_4}\frac{dv_4}{dv_3} = 1 \cdot \frac{d(v_2+v_3)}{dv_3} = \frac{dv_3}{dv_3} = 1$. Here, the derivative of a primitive function (addition) has been used. Also, the previous step has been recovered when asserting $\frac{df}{dv_4} = 1$. From now on, these details will be omitted.
3. $\overline{v_2} = \frac{df}{dv_2} = \overline{v_4}\frac{dv_4}{dv_2} = 1$.
4. $\overline{v_1} = \frac{df}{dv_1} = \overline{v_3}\frac{dv_3}{dv_1} = 1 \cdot v_{-1} = 2$.
5. $\overline{v_0} = 0$. It is the derivative with respect to a constant.
6. $\overline{v_{-1}} = \frac{df}{dv_{-1}} = \overline{v_2}\frac{dv_2}{dv_{-1}} + \overline{v_3}\frac{dv_3}{dv_{-1}} = 1 \cdot 1 + 1 \cdot v_1 = 1 + 1 = 2 = \frac{df}{dy}$.
7. $\overline{v_{-2}} = \frac{df}{dv_{-2}} = \overline{v_1}\frac{dv_1}{dv_{-2}} + \overline{v_1}\frac{dv_1}{dv_{-2}} = 2 \cdot v_{-2} + 2 \cdot v_{-2} = 2 + 2 = 4 = \frac{df}{dx}$.

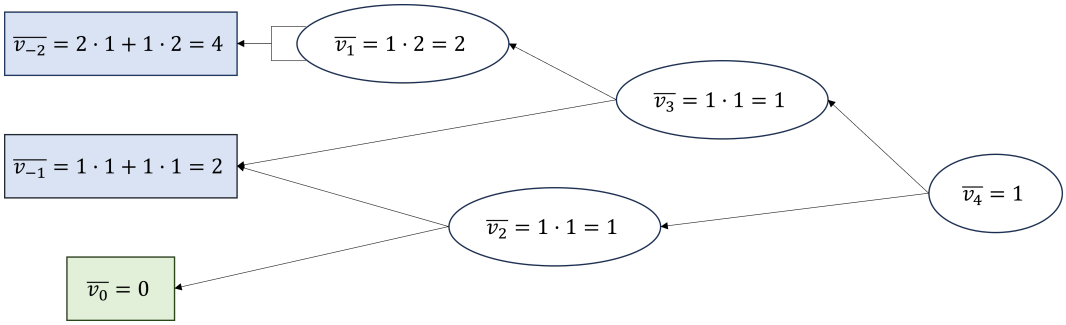


Figure 2.3: Backward pass. It is the same computational graph as in figure 2.1, but with all the values of the partial derivatives.

There are a few things worth commenting. Firstly, the graph in figure 2.3 goes from right to left. Here, each arrow goes from a node to those that will use its value when computing $\overline{v_i}$. Notice how, when calculating the values for $\overline{v_{-1}}$ and

$\overline{v_{-2}}$, two arrows end in those nodes. This is why the chain rule results in a sum of two members. More formally, the chain rule to differentiate $h(g_1(x), g_2(x))$ states $\frac{\partial h}{\partial x} = \frac{\partial h}{\partial g_1}\frac{\partial g_1}{\partial x} + \frac{\partial h}{\partial g_2}\frac{\partial g_2}{\partial x}$. Notice that, writing $f(x,y) = v_4(v_3(x,y), v_2(x,y))$, it is more proper to compute $\overline{v_3}$ as

$$\frac{\partial f}{\partial v_3} = \frac{\partial v_4}{\partial v_3}\frac{\partial v_3}{\partial v_3} + \frac{\partial v_4}{\partial v_2}\frac{\partial v_2}{\partial v_3}$$

This is obviously the same as what is written above because $\frac{\partial v_2}{\partial v_3} = 0$. The shorter version has been adopted for simplicity, and the arrows in the graph show what terms remain.

The other issue worth mentioning is that in the backward pass phase, all the data used in each computation is available because it was either calculated in the forward pass, or an earlier iteration of the backward pass.

Once one has gone over all the steps, recovering $\overline{v_{-2}}$ and $\overline{v_{-1}}$ one find that the value of the gradient is precisely $\nabla f(1,2) = (\overline{v_{-2}}, \overline{v_{-1}}) = (4,2)$, as expected. Then, the remarkable thing is that with only two iterations through the graph, the derivatives with respect to all the variables have been found.

Backpropagation and stochastic gradient descent In broad terms, backpropagation is an algorithm that can be used to train all sorts of computational graphs. At its core, it is the application of the reverse-mode autodiff method to training Neural Networks. It consists of the following steps:

1. Randomly initiate all the weights in the network.
2. Perform the forward pass of reverse-mode autodiff, while storing all the intermediate values. This computes the current loss of the model. If this loss has reached a desirable level, stop the algorithm here.
3. Perform the backward pass and, at each node of the computational graph, use the partial derivative found to update that parameter according to gradient descent.
4. Go back to step 2.

Thanks to reverse-mode autodiff, it is now feasible to compute the gradient of the loss function with respect to all the parameters. However, one needs to use all the training data to compute the loss at step 2. Since this process is generally repeated many times before gradient descent converges, it would be too costly to use all the date at every step. This is called *batch gradient descent* and it is slow and inefficient. Because of this, *stochastic gradient descent* is used.

Stochastic gradient descent (SGD) is at its core the same algorithm, but instead of using all the training data (the whole batch) at each iteration, only one example,

or a mini-batch[5] is used. There are different ways to accomplish this. The first one would be to select an instance or a mini-batch completely at random at each iteration. This might cause that some data points of the training set are used multiple times, while others are not used at all. Another way to proceed, is to shuffle all the data and use it in that order. Once all data has been used, shuffle it again and repeat. This ensures that the order in which each instance is taken changes, which is necessary for SGD to work properly. Like normal gradient descent, it can also be proved to converge under the right conditions, see [11].

A sweep over all the data is called an *epoch*. In classical gradient descent, each iteration represents an epoch because the whole training set is considered. However, in SGD, the definition becomes ambiguous. If one uses the first method, i.e. using a random subset of data at each iteration, there is no guarantee that all the data will be used before any repetitions occur. Therefore, in the case of SGD, an epoch is defined by a previously set number of iterations. Which, if using the second method, can iterate over the whole data as in the classical version.

Stochastic gradient descent introduces both issues and solutions. On the one hand, it makes the algorithm much faster by not taking all the data at each iteration when computing the loss function, and, by adding noise to the direction of the gradient, it allows the algorithm to not get stuck in local minima [4]. On the other hand, this noise makes the algorithm jump around even when it is in a minimum. Thus, making it to never stop, unless one changes the learning rate.

A solution for this last dilemma is to employ a *learning schedule*. This is just a function that determines the learning rate at each iteration. If one starts with large steps, it helps escape local minima. Then, gradually making those steps smaller allows to settle for a global minimum [14].

Finally, with all these changes and strategies, backpropagation is a very good algorithm to train Neural Networks. In fact, it is the most widely used method to train Machine Learning models.

2.1.3.3 Multi Layer Perceptron (MLP)

Sometimes called a *Vanilla Neural Network*, a MLP is one of the most basic architectures. In simple terms, it is a structure that receives a vector x as input and outputs y, which can be a vector or a single scalar value (depending on the task). This output is computed by nesting a generalisation of equation 2.4, as will be explored in the following lines.

[5] A subset of the training data with a fixed size. There are optimisations using vectorisation and GPU computing to make this very efficient.

Definition 2.6 (MLP)**.** *A* Multi Layer Perceptron *is a* ***feed-forward*** *neural network consisting of at least three* ***fully-connected layers****. Suppose there are K layers. Each layer is a function* $\ell_k : \mathbb{R}^{n_k} \to \mathbb{R}^{n_{k+1}}$, $k \in \{1, \ldots, K\}$*. If* x_k *is the input vector to layer k (*$x_k \in \mathbb{R}^{n_k}$*, where* n_k *is the number of neurons in layer k),* W_k *the weight matrix where each row represents the weights for a certain neuron*[6]*(*$W_k \in \mathbb{R}^{n_{k+1} \times n_k}$*),* $b \in \mathbb{R}^{n_{k+1}}$ *a bias vector, and* $\sigma_k : \mathbb{R}^{n_{k+1}} \to \mathbb{R}^{n_{k+1}}$ *the activation function, then:*

$$x_{k+1} = \ell_k(x_k) = \sigma_k(W_k x_k + b_k) \tag{2.7}$$

Then the network's output is a nesting of all the layers in the system: if x_1 *denotes the input, the network's output is* $(\ell_K \circ \ell_{K-1} \circ ... \circ \ell_1)(x_1)$*.*

Reading definition 2.6, one might feel that some ideas are not clear. Firstly, the definition states that at least three layers need to be present. These layers are: the *input layer*, one or more *hidden layers*, and a *output layer*. Since the first layer is not computational (it only transmits the input data to the next layer), if one were to remove the hidden layer(s), one would be left with single layer perceptron.

Regarding unintroduced concepts, *fully-connected layers* means that every neuron on layer k is connected to all the neurons in layer $k+1$; while *feed-forward* refers to the fact that the information inside the network travels in only one direction - forward - from the input, to the output, without feedback loops.

Finally, W_k and b_k, for $k = 1, \ldots, K$, are learnt parameters that vary from layer to layer. If one recalls the concepts of section 2.1.1.2, this refers to the parameters θ that one uses to optimise the loss function.

2.1.3.4 Residual Neural Networks

Residual Networks (ResNets) were introduced to solve what is known as the *degradation problem*. As described by the original researchers [16]:

"*With the network depth increasing, accuracy gets saturated (which might be unsurprising) and then degrades rapidly. Unexpectedly, such degradation is* ***not caused by over-fitting****, and adding more layers to a suitably deep model leads to* ***higher training error****.*"

This degradation can be caused by different problems, but the solution ResNets provide is very simple. Since it is evident that, by construction, the accuracy of a deeper network should be at least as good as the shallower one - by learning the identity mapping in all the extra layers - the idea is to allow the network a simple way to do it. This also solves a possible complication of exploding/vanishing gradient.

[6] In other words, each row corresponds to the row-vector that appears in equation 2.4

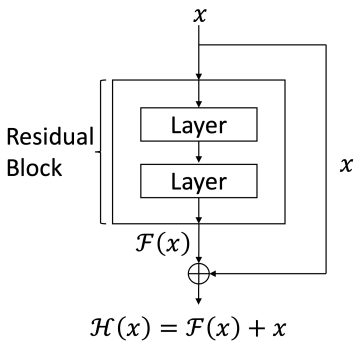


Figure 2.4: Basic representation of a residual block. Some steps like activation functions have been ignored for the sake of simplicity. Also, the shortcut connection is the identity function, as it was the original idea.

It is capital to understand the idea behind the model before formalising and generalising the concepts. Suppose one wishes to learn $\mathcal{H}(x)$. It does not have to be the mapping the whole network has to learn, it can be what a certain subset of layers should approximate. Now, if one defines $\mathcal{F}(x) := \mathcal{H}(x) - x$, learning $\mathcal{H}$ becomes equivalent to learning $\mathcal{F}$. It constitutes and advancement because the authors hypothesised that not all systems are equally easy to optimise. This leads to the introduction of *residual blocks*, which are the layers in the network that learn the new function $\mathcal{F}$, and use a shortcut connection from the previous block to add x and obtain the goal mapping $\mathcal{H}$. This is illustrated in figure 2.4. In the present context, a *Residual Function* refers to an auxiliary function $\mathcal{F}$ that is introduced in the learning process.

Definition 2.7 (ResNet)**.** *In its general form, a* Residual Neural Network *(ResNet) is a deep learning model that consists of layers grouped in residual blocks. Each block is formed by a set of layers*[7] *and a shortcut connection. More formally, if there are K residual blocks with L layers each (normally 2 or 3), the k-th block computes*

$$y_k = h(x_k) + f_{\mathcal{W}_k}(x_k) \tag{2.8}$$

$$x_{k+1} = \sigma_k(y_k) \tag{2.9}$$

where x_k and x_{k+1} are the input and output of the block, respectively. $f_{\mathcal{W}_k}$ is the output of a set of layers with parameters $\mathcal{W}_k = \{\mathcal{W}_{k,l} | 1 \leq l \leq L\}$. h is a function applied to the input of the block and gets added to the residual function. Lastly, σ_k is the activation function.

In the original paper [16], h was taken to be the identity function, and σ to be ReLu. However, they later found that it is best to use the identity function for both

[7] Essentially a sub-network that learns a residual function.

σ and h [17]. Then one can summarise equations 2.8 and 2.9 as:

$$x_{l+1} = x_l + f_{\mathcal{W}_l}(x_l) \tag{2.10}$$

Remark 2.8. Notice that definition 2.7 assumes the input and output dimensions are the same. This is not necessarily the case and there are strategies to solve the problem [16]. However, for the sake of simplicity, one will assume they are of equal dimension.

Remark 2.9. The notion of a ResNet is rather broad. There are no specific requirements about the architecture of each residual block. Many modifications have been explored and work when applied to various objectives [17].

2.1.3.5 Universal approximation

A sensible question to ask oneself is whether there are limitations to what Neural Networks can learn. Mathematically, this refers to the density of the class of functions that are Neural Networks, with respect to a given function space.

More concretely, it is often desirable to prove that a set of Neural Networks $\mathbb{R}^{d_1} \to \mathbb{R}^{d_2}$ can approximate any function in $\mathcal{C}(K; \mathbb{R}^{d_2})$, where $K \subseteq \mathbb{R}^{d_1}$ is a compact, and $\mathcal{C}(K; \mathbb{R}^{d_2})$ denotes the space of continuous functions $K \to \mathbb{R}^{d_2}$.

Before presenting formal results, let one introduce some notation, and a reminder of key concepts.

Definition 2.10 (Normed vector space)**.** *Let V be a vector space over $\mathbb{R}$. It is said that $(V, \|\cdot\|)$ is a* normed vector space *(normally referred to simply as V) if there exists a map $\|\cdot\| : V \to \mathbb{R}_+$, which is called a norm, and has the following properties:*

i) $\|v\| = 0 \Leftrightarrow v = 0,\ \forall v \in V$

ii) $\|v + u\| \leq \|v\| + \|u\|,\ \forall u, v \in V$

iii) $\|\lambda v\| = |\lambda|\|v\|,\ \forall \lambda \in \mathbb{K},\ \forall v \in V$

Remark 2.11. If one defines $d(u, v) = \|u - v\|$, then d is a distance in V. Therefore, (V, d) is a metric space.

Theorem 2.12. *Let $K \subseteq \mathbb{R}^d$ be a compact. Then $V = \mathcal{C}(K, \mathbb{R})$ is a vector space over $\mathbb{R}$, $\|\cdot\| : V \to \mathbb{R}$ defined as $\|f\| = \sup_{x \in K} |f(x)|$ is a norm, and $(V, \|\cdot\|)$ is complete*[8]*.*

[8] A metric space is complete if every Cauchy sequence in V converges in V.

Proof. For any $f, g \in V$ one can define addition as $(f+g)(x) = f(x) + g(x),\ \forall x \in K$; and, for any $\lambda \in \mathbb{R}$, the scalar product $(\lambda f)(x) = \lambda f(x),\ \forall x \in K$. Then, it is trivial that V is a vector space.

To prove that $\|\cdot\|$ is a norm, first notice that $\|f\| = \max_{x\in K} |f(x)| < +\infty$ because K is a compact, and f is continuous. Then, the properties of a norm hold. For any $f, g \in V$ and any $\lambda \in \mathbb{R}$:

i) $\|f\| \geq 0$ and $\|f\| = 0 \Leftrightarrow \max\limits_{x\in K} |f(x)| = 0 \Leftrightarrow |f(x)| = 0\ \forall x \in K \Leftrightarrow f = 0$

ii) $\|\lambda f\| = \max\limits_{x\in K} |\lambda f(x)| = |\lambda| \max\limits_{x\in K} |f(x)| = |\lambda| \|f\|$

iii) $\|f+g\| = \max\limits_{x\in K} |f(x)+g(x)| \leq \max\limits_{x\in K} |f(x)| + \max\limits_{x\in K} |g(x)| = \|f\| + \|g\|$

Finally, let $\{f_n\}_{n\geq 1} \in V$ be a Cauchy sequence. Then for all $x \in K$ $\{f_n(x)\}_{n\geq 1}$ is a Cauchy sequence in $\mathbb{R}$. Let $f(x) := \lim\limits_{n\to\infty} f_n(x)$. f is bounded (because the sequence it is point-wise Cauchy for all x and f_n is bounded for all n). Using the triangle inequality, the fact the sequence is Cauchy, and $\lim\limits_{n\to\infty} \|f_n - f_N\| = \|f - f_N\| < \epsilon/2$, one has $\|f - f_n\| \leq \|f - f_N\| + \|f_N - f_n\| < \epsilon$. Then it follows that $\{f_n\}$ converges to $f \in V$. To finish, f is continuous because of the uniform limit theorem. □

Definition 2.13 (Supremum norm)**.** *The norm defined above is normally referred to as the* uniform norm *or* supremum norm*, and often denoted as* $\|\cdot\|_\infty$.

From now on, all norms are the supremum norm unless stated otherwise.

Definition 2.14 (Universal Approximation)**.** *Given a normed vector space* $(V, \|\cdot\|)$ *and some subset* $W \subseteq V$*, it is said that* W *exhibits* universal approximation with respect to V *if for all* $\epsilon > 0$ *and for all* $v \in V$*, there exists* $w \in W$ *such that* $\|v - w\| < \epsilon$.

Remark 2.15. Using a metric space (V, d), definition 2.14 states that a subset $W \subseteq V$ exhibits universal approximation with respect to V if $\forall \epsilon > 0,\ \forall v \in V,\ \exists w \in W$ such that $d(v, w) < \epsilon$. This is, in fact, the definition of a dense set $W \subseteq V$.

Before delving into the details of neural networks, let one recall a similar result for polynomials in order to contextualise the issue and relate it to a better known problem. The following theorem states that polynomials exhibit universal approximation with respect to continuous functions $\mathcal{C}([a, b], \mathbb{R})$.

Theorem 2.16 (Weierstrass approximation theorem)**.** *For any continuous function* $f : [a, b] \to \mathbb{R}$ *and for any* $\epsilon > 0$*, there exists a polynomial* p *such that* $\|f - p\| < \epsilon$.

In the case of neural networks, the vector space V is the space of continuous functions, and the subset W the set of neural networks. Obviously, this is a problem and, to give some specific results, one needs to restrict it to special kinds of networks. The theorems presented here, are the ones about MLP with one hidden layer of arbitrary width, and MLP of arbitrary depth (i.e. number of layers).

Let $\mathcal{N}_d^\sigma$, for any continuous function $\sigma : \mathbb{R} \to \mathbb{R}$, and $d \in \mathbb{N}$, denote the set of feed-forward neural networks with activation function σ, with d neurons in the input layer, one neuron in the output layer, and an arbitrary number of neurons in a single hidden layer.

Theorem 2.17 (Universal Approximation Theorem)**.** *Let $K \subseteq \mathbb{R}^d$ be compact. Then, $\mathcal{N}_d^\sigma$ is dense in $\mathcal{C}(K, \mathbb{R})$ if, and only if, σ is non-polynomial.*

Proof. See [27], Theorem 3.1. □

Similarly, let $\mathcal{NN}_{d_{in}, d_{out}, d_w}^\sigma$, for any continuous function $\sigma : \mathbb{R} \to \mathbb{R}$, and for any $d_{in}, d_{out}, d_w \in \mathbb{N}$, denote the set of feed-forward neural networks with d_{in} neurons in the input layer, d_{out} neurons in the output layer, and an arbitrary number of hidden layers of width d_w, with activation function σ.

Theorem 2.18 (Deep and Narrow Universal Approximation Theorem)**.** *Let $\sigma : \mathbb{R} \to \mathbb{R}$ be any non-affine function of class $\mathcal{C}^1$, such that there is at least one point with nonzero derivative. Let $K \subseteq \mathbb{R}^{d_{in}}$ be compact. Then, $\mathcal{NN}_{d_{in}, d_{out}, d_{in}+d_{out}+2}^\sigma$ is dense in $\mathcal{C}(K, \mathbb{R}^{d_{out}})$.*

Proof. See [20], Theorem 3.2. □

2.2 Ordinary Differential Equations

Now, let one move on to Ordinary Differential Equations and some relevant results. The goal of this section is to contextualise and provide a foundation for the following chapters. From this section onward, $x : \mathbb{R} \to \mathbb{R}^d$ is a function, and $\dot{x}$ denotes its derivative with respect to t, unless stated otherwise. That is, $\dot{x} = \frac{dx}{dt}(t)$.

2.2.1 Initial Value Problem

Definition 2.19 (IVP)**.** *Let $D \subseteq \mathbb{R} \times \mathbb{R}^d$ and $f : D \to \mathbb{R}^d$ continuous. Consider the differential equation $\dot{x} = f(t, x)$. Then, given a point $(t_0, x_0) \in D$, the* Initial Value Problem

$$\begin{cases} \dot{x} = f(t, x) \\ x(t_0) = x_0 \end{cases} \tag{2.11}$$

consists in finding a differentiable function $\phi : I \to \mathbb{R}^d$ such that

i) $\phi(t_0) = x_0$,

ii) $\forall t \in I \ \ (t, \phi(t)) \in D$,

iii) $\forall t \in I \ \ \frac{d\phi}{dt}(t) = f(t, \phi(t))$

Remark 2.20. In the previous definition, ϕ is, a posteriori, not only differentiable, but C^1, due to the continuity of f.

Lemma 2.21 (Volterra Integral Equation)**.** *Let $\dot{x} = f(t, x)$ be a differential equation, where $f : D \subseteq \mathbb{R} \times \mathbb{R}^d \to \mathbb{R}^d$ is continuous. Let $(t_0, x_0) \in D$, and $\mathcal{I} \subseteq \mathbb{R}$ an interval containing t_0. Let $\phi : \mathcal{I} \to \mathbb{R}^d$ be a function. The following statements are equivalent:*

a) ϕ is differentiable, and a solution to the IVP 2.11.

b) ϕ is continuous, and solution to the Volterra Integral Equation*:*

$$x(t) = x_0 + \int_{t_0}^{t} f(s, x(s))ds. \tag{2.12}$$

2.2.2 Solutions

It is useful to introduce specific notation for a unique solution to the IVP 2.11. It is custom to write such a solution as $\phi(t; t_0, x_0)$. If t_0 and x_0 remain fixed, this is the same function of t introduced in definition 2.19, but referencing that it solves a particular IVP. Additionally, it is useful to see it as a function $\phi : D \subseteq \mathbb{R} \times \mathbb{R} \times \mathbb{R}^d \to \mathbb{R}^d$ that changes with respect to the initial conditions. This mapping ϕ is sometimes called an *evolutionary process* [2], and D is its *domain*.

2.2.2.1 Existence and Uniqueness

Let one start with a specific version of Picard's Theorem that will be useful in the following chapters.

Theorem 2.22 (Picard's Existence and Uniqueness Theorem)**.** *Consider the initial value problem 2.11 where $f : D = [t_0, t_f] \times \mathbb{R}^d \to \mathbb{R}^d$ is continuous and Lipschitz with respect to x in $[t_0, t_f] \times \mathbb{R}^d$. Then, for all $(t_0, x_0) \in [t_0, t_f] \times \mathbb{R}^d$ there exists a unique solution to the initial value problem, and this solution is defined in $[t_0, t_f]$.*

The following result is a direct consequence of this theorem.

Corollary 2.23. *Let ϕ_1 and ϕ_2 be solutions of the IVP 2.11 with initial conditions $\phi_1(t_0) \neq \phi_2(t_0)$. Then $\forall t \in [t_0, t_f]$, $\phi_1(t) \neq \phi_2(t)$. Informally, ODE trajectories do not intersect.*

2.2.2.2 Extensibility of solutions

Solutions to the IVP 2.11 might not exist for all or any $t \in \mathbb{R}$. This raises the question about the maximal interval of definition.

Definition 2.24 (Extension)**.** *Let ϕ_1 and ϕ_2 be two solutions to the IVP 2.11, defined at intervals $\mathcal{I}_1$ and $\mathcal{I}_2$ respectively. If $\phi_1\big|_{\mathcal{I}_2}$, then ϕ_2 is an* extension *of ϕ_1.*

Definition 2.25 (Maximal interval)**.** *$\mathcal{I}$ is a* maximal interval *for an IVP if the solution defined at $\mathcal{I}$ cannot be extended to any interval $\mathcal{J}$, $\mathcal{I} \subset \mathcal{J}$. The maximal interval for an IVP with initial condition $x(t_0) = x_0$ and unique solution is written as $\mathcal{I}(t_0, x_0)$.*

Definition 2.26 (Maximal solution)**.** *A solution that cannot be extended is also called* maximal solution*.*

Theorem 2.27 (Existence of maximal solutions)**.** *Suppose the IVP 2.11 has a unique solution. Then, for all $(t_0, x_0) \in \Omega$, the corresponding IVP has a unique maximal solution defined at $\mathcal{I}(t_0, x_0)$. Furthermore, $\mathcal{I}(t_0, x_0)$ is an open interval.*

Definition 2.28 (Global solution)**.** *A* global solution*, is a maximal solution with maximal interval $\mathcal{I} = \mathbb{R}$.*

2.2.3 Flows

Definition 2.29 (Non-autonomous flow)**.** *Let $\Omega = \mathcal{I} \times U \subseteq \mathbb{R} \times \mathbb{R}^d$, and ϕ a solution to the differential equation $\dot{x} = f(t, x)$ of an IVP like 2.11. For any $s, t \in \mathcal{I}$ let $\phi_t^s(x) = \phi(t; s, x)$ denote the solution at time t of the equation with initial condition $x(s) = x$. This is called a* non-autonomous flow*.*

More specifically, let one fix an initial time t_0 and denote $\phi_t : U_t \to U$ as a function of the initial condition corresponding to $\phi_t(x) = \phi(t; t_0, x)$ where $U_t = \{x \in U | t \in \mathcal{I}(t_0, x)\}$. Notice that $U_0 = U$ and $\phi_0 = \mathrm{Id}\big|_U$.

From now on, ϕ_t will be referred to as the *flow* of the IVP, considering a fixed initial time.

Remark 2.30. The concept of *flow* is normally associated to autonomous differential equations. In definition 2.29, this notion has been modified, by also fixing an initial time t_0.

Proposition 2.31. *Let $f : [t_0, t_f] \times U \subseteq \mathbb{R} \times \mathbb{R}^d \to U$, U an open set. If f is continuous and Lipshitz with respect to x, and given $s, t, r \in [t_0, t_f]$ then $\phi_t^s = \phi_t^r \circ \phi_r^s$. In particular, $\phi_t^s \circ \phi_s^t = \mathrm{Id}$ is invertible for all s and t.*

Proof. [32] It follows from the uniqueness of solutions. If $x \in U$, $\phi_t^s(x)$ is the value at time t of the unique solution of $\{\dot{y} = f(t, y), \quad y(s) = x\}$ which is equal to x at time s; it is equal to $\tilde{x} := \phi_r^s(x)$ at time r, and thus also equal to $\phi_t^r(\tilde{x})$. □

Theorem 2.32. *If $f : [t_0, t_f] \times U \subseteq \mathbb{R} \times \mathbb{R}^d \to U$, the associated flow ϕ_t^s is a homeomorphism of U for all times s and t.*

Proof. Proposition 2.31 implies that ϕ_t^s is invertible and $(\phi_t^s)^{-1} = \phi_s^t$. Therefore, it is sufficient to prove ϕ_t^s is continuous for any $s, t \in [t_0, t_f]$. This follows from proposition 2.33 □

2.2.4 Regularity

Proposition 2.33. *Let $f : \Omega \subseteq \mathbb{R} \times \mathbb{R}^d \times \mathbb{R}^p \to \mathbb{R}^d$ be a family of functions depending on a parameter $\theta \in \mathbb{R}^p$. Suppose f is continuous and locally Lipschitz with respect to x in the open set Ω. Consider the IVP $\{\dot{x} = f(t, x, \theta);\ x(t_0) = x_0\}$ for $(t_0, x_0, \theta) \in \Omega$ and its solution $\phi : \mathcal{D} \subseteq \mathbb{R} \times \Omega \to \mathbb{R}^d$. Then,$\mathcal{D}$ is an open set and ϕ is continuous and locally Lipschitz with respect to $(t; t_0, x_0)$. Additionally, if f is locally Lipschitz with respect to θ, then ϕ is locally Lipschitz with respect to $(t; t_0, x_0, \theta)$.*

Theorem 2.34 (Differentiability with respect to initial conditions and parameters)**.** *Let $f : \Omega \subset \mathbb{R} \times \mathbb{R}^d \times \mathbb{R}^p \to \mathbb{R}^d$ be a family of functions depending on a parameter $\theta \in \mathbb{R}^p$. Suppose Ω is an open set and f is continuous and of class $\mathcal{C}^1(\Omega)$ with respect to x and θ. Consider the IVP $\{\dot{x} = f(t, x, \theta);\quad x(t_0) = x_0\}$ for some $(t_0, x_0, \theta) \in \Omega$. Let $\phi(t; t_0, x_0, \theta)$ denote its solution. Then, $\phi(t; t_0, x_0, \theta)$ is of class $\mathcal{C}^1$ with respect to (t, t_0, x_0, θ). Furthermore, the derivatives satisfy the following equations:*

The matrix of partial derivatives of $\phi(t; t_0, x_0, \theta)$ with respect to x_0 is the solution to

$$\begin{cases} \dot{X} = D_x f(t, \phi(t; t_0, x_0, \theta), \theta) X \\ X(t_0) = \mathrm{Id}_d \end{cases} \tag{2.13}$$

Similarly, the vector of partial derivatives of $\phi(t; t_0, x_0, \theta)$ with respect to t_0 satisfies

$$\begin{cases} \dot{X} = D_x f(t, \phi(t; t_0, x_0, \theta), \theta) X \\ X(t_0) = -f(t_0, x_0, \theta) \end{cases} \tag{2.14}$$

Finally, the matrix of partial derivatives of $\phi(t; t_0, x_0, \theta)$ with respect to θ solves

$$\begin{cases} \dot{X} = D_x f(t, \phi(t; t_0, x_0, \theta), \theta) X + D_\theta f(t, \phi(t; t_0, x_0, \theta), \theta) \\ X(t_0) = 0 \end{cases} \tag{2.15}$$

Chapter 3

Neural Ordinary Differential Equations

Now that the foundations have been laid, it is possible to define neural ordinary differential equations and introduce their properties. This chapter intends to provide a solid base for this kind of model and how it can be used.

3.1 Defining Neural Ordinary Differential Equations

In order to approach *Neural Ordinary Differential Equations* (Neural ODE or ODE-Net), one can take two separate paths, depending on one's background.

The most direct path is what leads to a "mathematical" definition. In this case, one takes the concept of a classic ODE and uses modern deep learning (Neural Networks) to parameterise the vector field.

Definition 3.1 (Neural ODE)**.** *Consider the following initial value problem:*

$$\begin{cases} \frac{dx}{dt}(t) = f_\theta(t, x(t)) \\ x(0) = x_0 \end{cases} \tag{3.1}$$

Where $x : \mathbb{R} \to \mathbb{R}^d$, $d \in \mathbb{N}$, $t \in \mathbb{R}$, $f_\theta : \mathbb{R} \times \mathbb{R}^d \to \mathbb{R}^d$ *and* $\theta \in \mathbb{R}^p$, *with* $p \in \mathbb{N}$. *The IVP in equation 3.1 is said to be a* Neural Ordinary Differential Equation *if the function* f_θ *is a neural network, i.e. the differential equation's vector field is parameterised using a neural network. Here,* f_θ *can be any neural architecture.*

Remark 3.2. Despite being technically correct, definition 3.1 is not very useful without some extra conditions. As it is, there is no guarantee that the solution is unique or that it even exists. Some basic requirements are for f_θ to be continuous and Lipschitz with respect to the second variable.

Another approach to Neural ODEs is to regard them as "*continuous-depth*" neural networks [6].

Recalling equation 3.1, and discretising it using Euler's method, on is left with

$$x_{n+1} = x_n + hf_\theta(t_n, x_n) = x_n + hf_\theta(n \cdot h, x_n) \tag{3.2}$$

using the same notation as in section A.1.

For convenience, it is useful to rewrite equation 2.10 as

$$x_{n+1} = x_n + f_\theta(n, x_n) \tag{3.3}$$

where n denotes the residual block, θ includes the weights for all blocks, and f_θ has an added argument n to reference the current block.

Now one can easily observe that equations 3.2 and 3.3 are the same by taking $h = 1$. With all this, one has been able to see an ODE-Net as the continuum limit of a ResNet. Or, rather more precisely, a ResNet as the discrete version of an ODE-Net.

Remark 3.3. Notice how equation 3.3 has introduced the notion that not all residual blocks need to be of the same form. As is usually done in image recognition tasks, it might be useful to change the network's architecture from layer to layer. This change can be seen as a function of l, being l the block, that controls the weights and layers inside the residual block. In the case of neural ODEs, the vector field (which is a neural network) also changes as a function of t, this time, however, in a continuous fashion.

3.1.1 Modelling with Neural ODEs

From definition 3.1 and the subsequent interpretation, it is not clear how one can use this structure. Disregarding critical aspects like its learning methodology, which will be addressed later in this dissertation, it is essential to define how the model makes predictions based on input data.

Suppose the IVP 3.1 has a solution $\phi : \Omega \to \mathbb{R}^d$ where $\Omega \subseteq \mathbb{R} \times \mathbb{R} \times \mathbb{R}^d \times \mathbb{R}^p$ is an open set. Then, for any $x \in \mathbb{R}^d$ one can define the mapping

$$\begin{aligned} \psi \colon \mathbb{R}^d &\to \mathbb{R}^d \\ x &\mapsto \phi(T; 0, x, \theta) \end{aligned} \tag{3.4}$$

which corresponds to a flow ϕ_t where $t_0 = 0$.

Then, the ODE-Net model is actually the mapping in equation 3.4 that, given an input vector x, outputs a prediction corresponding to the value of the solution to the IVP with initial condition $x(0) = x$, at time T.

This is a very rigid structure. Notice that the source and target spaces are the same, so it would mean that it can only be used to predict values in the same space as the input variables. This can be solved by composing with auxiliary functions before and after ψ, as the following example illustrates.

For simplicity, let one focus on an image classification problem, using a neural ODE [19]. This example will allow for the introduction of some important theoretical concepts, as well as an illustration of the usage of this kind of models.

3.1.1.1 Data

Suppose one has some images, like the MNIST dataset (see B.1.1), which can be represented as a tensor $\mathbb{R}^{28\times 28}$, corresponding to height (28 pixels), and width (28 pixels), respectively. Suppose also that one wishes to classify these images using a class label in $\mathbb{R}^{10}$ corresponding to a one-hot encoding[1] of what digit the image represents.

3.1.1.2 Basic model

Let $f_\theta : \mathbb{R} \times \mathbb{R}^{28\times 28} \to \mathbb{R}^{28\times 28}$ be a convolutional neural network, and let $l_\theta : \mathbb{R}^{28\times 28} \to \mathbb{R}^{10}$ be affine.

Then, a first version of the desired image classification model, what will be referred to as the 'basic model' from now on, can be defined as follows, using the same notation introduced previously in this section:

$$\begin{aligned} \Psi \colon \mathbb{R}^{28\times 28} &\to \mathbb{R}^{28\times 28} \to \mathbb{R}^{10} \\ x \mapsto \psi(x) \quad &\mapsto \operatorname{softmax}(l_\theta(\psi(x))) \end{aligned} \tag{3.5}$$

Looking at the first part of the model, it is simply the flow of the ODE to some time T. What the model is doing is approximate some function $h(x)$ that allows for the second part of the model to classify the input accordingly.

3.1.1.3 Augmented model

The model defined by equation 3.5 can be improved using a technique called 'augmentation'[2]. This refers to the practice of inserting an affine map between input and initial value, to increase the dimension of the IVP with respect to the input. The relevant justifications will be given in the next section.

[1] One-hot encoding transforms a categorical variable with N categories into a binary vector of length N where only one element is 1, representing the category by its position.

[2] Not all models allow for the use of this technique. For example, Continuous Normalising Flows cannot use augmentation, as it is a requirement that every operation is bijective.

Let one start with a general explanation, before applying it to the current example. Given some input $x \in \mathbb{R}^d$, the initial value of the ODE is taken to be $g_\theta(x)$ instead of simply x. Here, $g_\theta : \mathbb{R}^d \to \mathbb{R}^{d_l}$ is some learnt function, with $d_l > d$.

With this formulation, $\psi(x)$ becomes $\phi(T; 0, g_\theta(x), \theta)$ instead of $\phi(T; 0, x, \theta)$. Also, note that the dimension of θ might not be the same in both cases, since it might contain parameters for the function g_θ.

The main point is that g_θ increases dimensionality, not its particular choice. The reason is that the continuous flow of an ODE cannot change the topology of its input. Therefore, if the target space has the same dimension as the source space, topological properties of the input manifold are preserved. More on this later.

Returning to the image classification example, taking g_θ as the zero augmentation function (i.e. $g_\theta(x) = (x, 0)$), and keeping the same ψ function as in equation 3.5[3], the resulting model becomes:

$$\begin{aligned} \Psi : \mathbb{R}^{28\times 28} &\to \mathbb{R}^{28\times 28\times 1} \to \mathbb{R}^{28\times 28\times 1} \to \mathbb{R}^{10} \\ x &\mapsto (x, 0) \quad \mapsto \psi((x, 0)) \mapsto \text{softmax}(l_\theta(\psi((x, 0)))) \end{aligned} \tag{3.6}$$

To summarise, in general, a NODE is a model of the form

$$\begin{aligned} \Psi : \mathbb{R}^{d_{in}} &\to \mathbb{R}^{d_l} \to \mathbb{R}^{d_l} \to \mathbb{R}^{d_{out}} \to \mathbb{R}^{d_{out}} \\ x &\mapsto g_\theta(x) \mapsto \psi(g_\theta(x)) \mapsto l_\theta(\psi(g_\theta(x))) \mapsto \text{act}(l_\theta(\psi(g_\theta(x)))) \end{aligned}$$

where $g_\theta : \mathbb{R}^{d_{in}} \to \mathbb{R}^{d_l}$ is a function that increases the dimensionality of the input ($d_{in} < d_l$), $l_\theta : \mathbb{R}^{d_l} \to \mathbb{R}^{d_{out}}$ is an affine transformation, and act is an activation function. Not all of these are always necessary. For example, g_θ is only used in *augmented* NODEs, and the activation function might be ignored in some scenarios.

3.2 Existence and uniqueness

The three attributes of an initial value problem that have to be considered are whether there exists a solution, whether it is unique, and how sensitive it is to small perturbations to the initial information.

The answer to these questions lies in the same results as for other ordinary differential equations, presented in section 2.2.2.1.

[3] If one were to use $\psi(x) = \phi(T; 0, g_\theta(x), \theta)$, the model formulation would be the same as with the basic model. The old ψ is used to make the changes more explicit.

3.3 Approximation properties

It is a natural question whether neural ODEs are capable of approximating any kind of function. Approximation theory was introduced in section 2.1.3.5 when examining the capabilities of some basic neural networks. The goal of the current section is to do the same for neural ODEs. For notation simplicity, all NODEs will be assumed to apply no activation function to the final output. In appendix B.5 one can find a comparison of an augmented and an unaugmented model.

3.3.1 Unaugmented neural ODEs

Recall that, in this context, an unaugmented model is one which does not apply any transformation to the input before solving the neural ODE. Using the same notation as in equation 3.4, the model may be written as

$$\begin{aligned} \Psi \colon \mathbb{R}^{d_{in}} &\to \mathbb{R}^{d_{in}} \to \mathbb{R}^{d_{out}} \\ x &\mapsto \psi(x) \mapsto h_\theta(\psi(x)) \end{aligned} \tag{3.7}$$

where $d_{in}, d_{out} \in \mathbb{N}$ represent the dimensions of the input space, and the output space, respectively; and h_θ is some affine transformation adequate for the task at hand. For example, it might be the composition softmax $\circ \psi$ as in equation 3.5.

Unfortunately, this model cannot approximate many functions for the reason previously hinted. The continuous evolution of the ODE ensures that any topological property of the input is preserved. Let one illustrate this with a class of functions that cannot be represented by a NODE [10].

Definition 3.4 (Representation)**.** *A family of models given by the functions $m_\theta : \mathbb{R}^{d_{in}} \to \mathbb{R}^{d_{out}}$ for parameters $\theta \in \Theta$ is said to* represent *an arbitrary function $h : \mathbb{R}^{d_{in}} \to \mathbb{R}^{d_{out}}$ if $\exists \theta \in \Theta$ such that $m_\theta(x) = h(x)\ \forall x \in \mathbb{R}^{d_{in}}$.*

Let $0 < r_1 < r_2 < r_3$ and let $h : \mathbb{R}^d \to \mathbb{R}$ be a function such that

$$h(x) = \begin{cases} -1 & \text{if} \quad \|x\| \leq r_1 \\ 1 & \text{if} \quad r_2 \leq \|x\| \leq r_3 \end{cases}$$

where $\|\cdot\|$ is the Euclidean norm. Let $\mathcal{A} = \{x | \|x\| \leq r_1\}$ denote the disk with radius r_1 and let $\mathcal{B} = \{x | r_2 \leq \|x\| \leq r_3\}$ be the d-dimensional spherical shell.

Proposition 3.5. *Neural ODEs cannot represent $h(x)$.*

Remark 3.6. Recalling the structure of an (unaugmented) NODE ($\Psi = l_\theta \circ \psi$), for it to map points in $\mathcal{A}$ to -1 and points in $\mathcal{B}$ to 1, the affine transformation l_θ must

map the points in $\psi(\mathcal{A})$ to -1, and the points in $\psi(\mathcal{B})$ to 1. This implies that $\psi(\mathcal{A})$ and $\psi(\mathcal{B})$ must be linearly separable. That is, there exists a hyperplane such that all points in $\psi(\mathcal{A})$ lie above it, and all points in $\psi(\mathcal{B})$ lie below it. The following proof consists on proving this is not possible if ψ is a homeomorphism.

Proof. Define a disk $D = \{x \in \mathbb{R}^d | \|x\| \leq r_2\}$ with boundary ∂D and interior $\text{Int}(D)$. Now, $\mathcal{A} \subset \text{Int}(D)$, $\partial D \cap \mathcal{A} = \emptyset$ and $\partial D \subset \mathcal{B}$. So the problem is reduced to seeing that $\psi(\text{Int}(D))$ and $\psi(\partial D)$ are not linearly separable.

Since ψ is a homeomorphism (2.32), then $\psi(\text{Int}(D)) = \text{Int}(\psi(D))$ and $\psi(\partial D) = \partial\psi(D)$. For convenience, let one write $D' = \psi(D)$.

Suppose $\partial D'$ and $\text{Int}(D')$ to be linearly separable. This is, $\forall x \in \text{Int}(D')\ L(x) > 0$ and $\forall x \in \partial D'\ L(x) < 0$ for some affine function $L(x) = w^T x + c$ for $w \in \mathbb{R}^d$ and $c \in \mathbb{R}$.

Sine D is a compact and ψ a homeomorphism, D' is also compact. This means that D' is bounded (Heine-Borel theorem). Let $B_\delta(p) = \{x \in \mathbb{R}^d | \|x - p\| < \delta\}$ be the open d-dimensional ball centered in p with radius δ. Then, for any $x \in \text{Int}(D')$ there exist $a < b \in \mathbb{R}$ such that $B_a(x) \subset \text{Int}(D')$ (definition of interior point) and $\delta D' \subset B_b(x)$. Consider now a diameter of $B_b(x)$. This segments must intersect $\delta D'$ at least in two points $x_1, x_2 \in \delta D'$ (and $x_1 \neq x \neq x_2$ since $\|x - x_i\| > a$, $i = 1, 2$).

One can write this segment as $\{\lambda x_1 + (1-\lambda)x_2\}$ for $0 \leq \lambda \leq 1$. Furthermore, for some $0 < \tilde{\lambda} < 1$, $x = \tilde{\lambda} x_1 + (1 - \tilde{\lambda})x_2$. Then,

$$\begin{aligned} L(x) &= w^T x + c \\ &= w^T(\tilde{\lambda} x_1 + (1-\tilde{\lambda})x_2) + c \\ &= \tilde{\lambda} w^T x_1 + (1-\tilde{\lambda}) w^T x_2 + c \\ &< \tilde{\lambda}(-c) + (1-\tilde{\lambda})(-c) + c \\ &= 0 \end{aligned}$$

But it was supposed that $L(x) > 0$ for all $x \in \text{Int}(D')$. Therefore, $\text{Int}(D')$ and $\partial D'$ are not linearly separable, and neither are $\psi(\mathcal{A})$ and $\psi(\mathcal{B})$. □

In practice, even if NODEs are not able to represent this class of functions, it is often possible to approximate these functions, albeit the resulting flows are complex and lead to problems that are computationally hard to solve [10].

3.3.2 Augmented neural ODEs

As discussed above, the main idea of augmentation is to increase the dimensionality of the ODE input as to avoid trajectories intersecting. This leads

to lower losses, better generalisation and lower computational cost that unaugmented NODEs [10]. This section will be focused on their status as universal approximators (definition 2.14).

3.3.2.1 The vector field is a universal approximator

In this situation, f has universal approximation properties in Lipschitz functions $\mathbb{R} \times \mathbb{R}^{d_l} \to \mathbb{R}^{d_l}$. For simplicity, let one state the following result assuming the vector field is drawn from the space of Lipschitz functions instead of just a dense subset of it.

Theorem 3.7. *Let $d, d_l, d_{out} \in \mathbb{N}$ with $d_l \geq d + d_{out}$. Suppose $f : \mathbb{R} \times \mathbb{R}^{d_l} \to \mathbb{R}^{d_l}$ is a continuous and Lipschitz function, and $g : \mathbb{R}^d \to \mathbb{R}^{d_l}$ and $l : \mathbb{R}^{d_l} \to \mathbb{R}^{d_{out}}$ are affine. Let $\Psi_{f,g,l}(x)$ denote the map*

$$\begin{aligned} \Psi_{f,g,l} : \mathbb{R}^d \to \mathbb{R}^{d_l} \quad &\to \mathbb{R}^{d_l} \qquad \to \mathbb{R}^{d_{out}} \\ x \mapsto g(x) &\mapsto \psi(g(x)) \mapsto l(\psi(g(x))), \end{aligned}$$

where the same notation as before has been used to the denote $\psi(x) = \phi(T; 0, x)$ and ϕ is the solution to the corresponding IVP $\{\dot{x} = f(t, x);\ x(0) = x_0\}$.

Then,

$$\{\Psi_{f,g,l} | f \in \mathrm{Lip}(\mathbb{R} \times \mathbb{R}^{d_l}; \mathbb{R}^{d_l}), g : \mathbb{R}^d \to \mathbb{R}^{d_l} \text{ affine}, l : \mathbb{R}^{d_l} \to \mathbb{R}^{d_{out}} \text{ affine}\}$$

is a universal approximator for $\mathcal{C}(\mathbb{R}^d; \mathbb{R}^{d_{out}})$.

Proof. See [33], Theorem 7. □

3.3.2.2 The vector field is not a universal approximator

Now, one can get universal approximation using only a vector field f for each dimension d.

Theorem 3.8. *Let $d, d_{out} \in \mathbb{N}$. For $d_l \in \mathbb{N}$, $f \in \mathcal{C}(\mathbb{R} \times \mathbb{R}^{d_l}; \mathbb{R}^{d_l})$, and $g : \mathbb{R}^d \to \mathbb{R}^{d_l}$ and $l : \mathbb{R}^d \to \mathbb{R}^{d_l}$ affine functions, let $\Psi_{d_l,f,g,l} : \mathbb{R}^d \to \mathbb{R}^{d_{out}}(x)$ denote the ODE-Net like previously, for those f for which there exists a unique solution to the corresponding IVP. For each $d_l \in \mathbb{N}$ there exists an $f_{d_l} \in \mathcal{C}(\mathcal{R}^{d_l}; \mathbb{R}^{d_l})$ for which the IVP has a unique solution such that*

$$\{\Psi_{d_l,f,g,l} | d_l \in \mathbb{N}, g : \mathbb{R}^d \to \mathbb{R}^{d_l} \text{ affine}, l : \mathbb{R}^{d_l} \to \mathbb{R}^{d_{out}} \text{ affine}\}$$

is a universal approximator for $\mathcal{C}(\mathbb{R}^d; \mathbb{R}^{d_{out}})$

Proof. See [19], Theorem 2.13. □

3.4 Training

To learn the parameters of the model, its derivatives need to be computed. To accomplish this, there are a few options available. The ones that are explored in this dissertation are:

- Discretise-then-optimise
- Optimise-then-discretise
- Variational equations

Each of this methods have their own advantages and disadvantages and their use has to be evaluated in a case by case fashion. The content presented in this section is largely derived from [19] and [6].

3.4.1 Variational Equations

When looking for the derivatives of an ODE solution with respect to the vector field's parameters, classic mathematics has an answer: variational equations. By using 2.15, one can solve the new equation with an ODE solver and use the chain rule to compute $\frac{dL}{d\theta}$. As with regular neural networks, this approach is not recommended because it computes the whole Jacobian matrix when only its product with $\frac{dL}{dx(t)}$ is relevant, making this method less efficient both in terms of space and time.

3.4.2 Discretise-then-optimise

This is the most straightforward option since it consists simply on backpropagating through the internal operations of the ODE solver. Since the solver is a composition of differentiable operations such as addition or multiplication, it is also differentiable.

Implementing this is also rather effortless: if the differential equation solver was written in an autodifferentiable framework, the gradients can be computed directly as any other function. Examples of such implementations are `torchdiffeq`[4] for `PyTorch` and `diffrax`[5] for `JAX`.

3.4.2.1 Advantages

- *Accuracy*: The derivatives are computed directly on the discretised version of the differential equation, as opposed to an ideal continuous representation.

[4] https://github.com/rtqichen/torchdiffeq
[5] https://github.com/patrick-kidger/diffrax

Because of this, it offers higher accuracy to represent the gradients of the model.

- *Time complexity*: It is often the fastest way to backpropagate because the computational graph is already known and the underlying libraries can leverage parallelism.

3.4.2.2 Disadvantages

- *Memory inefficient*: This approach requires that every internal operation of the ODE solver is stored. This leads to a memory consumption of $\mathcal{O}(ST)$ where S represents the number of operation in each step, and T the time horizon.

3.4.2.3 Use cases

This is generally the preferred technique. One should only consider other methods if there are memory constraints.

3.4.3 Optimise-then-discretise

In contrast with the previous option, this works by finding the derivatives of the ideal continuous model. This is done by numerically solving a backwards-in-time differential equation, which arises from the following result.

Theorem 3.9 (Continuous adjoint equations)**.** *Let $x_0 \in \mathbb{R}^d$ and $\theta \in \mathbb{R}^p$. Let $f_\theta : [t_0, t_1] \times \mathbb{R}^d \to \mathbb{R}^d$ be continuous in t, and of class $\mathcal{C}^1$ in x and θ. Let $\phi(t; t_0, x_0, \theta) : \mathbb{R} \times \mathbb{R} \times \mathbb{R}^d \times \mathbb{R}^p$ denote the solution to the IVP*

$$\dot{x} = f_\theta(t, x), \quad x(t_0) = x_0. \tag{3.8}$$

For simplicity, let $x_1 = \phi(t_1; t_0, x_0, \theta)$.

Let $L : \mathbb{R}^d \to \mathbb{R}$ be a differentiable scalar function and let one define the function $\ell(t_1, t_0, x_0, \theta) := L(\phi(t_1; t_0, x_0, \theta))$. Lastly, let one define the following:

$$a_x(t)^T := \frac{dL}{dx}(x_1) D_{x_0}\phi(t; t_1, x_1, \theta)^{-1} \tag{3.9}$$

$$a_\theta(t)^T := \int_t^{t_1} a_x(t)^T D_\theta f(s, \phi(s; t_1, x_1, \theta), \theta) ds \tag{3.10}$$

Then, a_x and a_θ satisfy the following ODEs

$$\dot{a}_x(t)^T = -a_x(t)^T D_x f(t, \phi(t; t_1, x_1, \theta), \theta); \qquad a_x(t_1) = \frac{dL}{dx}(x_1) \tag{3.11}$$

$$\dot{a}_\theta(t)^T = -a_x(t)^T D_\theta f(t, \phi(t; t_1, x_1, \theta), \theta); \qquad a_\theta(t_1) = 0, \tag{3.12}$$

Furthermore, $a_\theta(t_0)^T = \frac{\partial \ell}{\partial \theta}(t_1, t_0, x_0, \theta)$.

Proof. The IVP 3.12 follows directly from the definition 3.10. Similarly, from definition 3.9 one has $\frac{dL}{dx}(x_1) = a_x(t)^T D_{x_0}\phi(t; t_1, x_1, \theta)$ for any $t \in [t_0, t_1]$ and taking the derivative with respect to t:

$$0 = \frac{d}{dt}\left(\frac{dL}{dx}(x_1)\right) = \dot{a}_x(t)^T D_{x_0}\phi(t; t_1, x_1, \theta) + a_x(t)^T \frac{d}{dt}\left(D_{x_0}\phi(t; t_1, x_1, \theta)\right)$$

Using the variational equation with respect to the initial conditions (2.13) this can be written as

$$0 = \dot{a}_x(t)^T D_{x_0}\phi(t; t_1, x_1, \theta) + a_x(t)^T D_x f(t, \phi(t; t_1, x_1, \theta), \theta) D_{x_0}\phi(t; t_1, x_1, \theta)$$

Or, equivalently, $\dot{a}_x(t)^T = -a_x(t)^T D_x f(t, \phi(t; t_1, x_1, \theta), \theta)$ and by using the definition of $a_x(t)$ and evaluating for $t = t_1$, 3.11 follows.

Finally, to see $a_\theta(t_0)^T = \frac{\partial \ell}{\partial \theta}(t_1, t_0, x_0, \theta)$, let one start by using the chain rule,

$$\frac{\partial \ell}{\partial \theta}(t_1, t_0, x_0, \theta) = \frac{dL}{dx}(\phi(t_1; t_0, x_0, \theta)) D_\theta \phi(t_1; t_0, x_0, \theta). \tag{3.13}$$

Recalling the variational equation with respect to the parameters (2.15), the derivative with respect to the parameters $D_\theta\phi(t_1; t_0, x_0, \theta)$ satisfies

$$\dot{X} = D_x f(t, \phi(t_1; t_0, x_0, \theta), \theta) X + D_\theta f(t, \phi(t_1; t_0, x_0, \theta), \theta); \quad X(t_0) = 0 \tag{3.14}$$

and since, once again, $D_{x_0}\phi(t_1; t_0, x_0, \theta)$ is the solution to the variational equation with respect to the initial conditions (2.13), using the variation of constants method one has

$$D_\theta \phi(t_1; t_0, x_0, \theta) = D_{x_0}\phi(t_1; t_0, x_0, \theta) \int_{t_0}^{t_1} D_{x_0}\phi(s; t_0, x_0, \theta)^{-1} D_\theta(s, \phi(s; t_0, x_0, \theta), \theta) ds \tag{3.15}$$

Then, recalling that $\phi(t_1; s, \phi(s; t_0, x, \theta), \theta) = \phi(t_1; t_0, x, \theta)$ for any $s \in [t_0, t_1]$ and $x \in \mathbb{R}^d$, it follows that $D_{x_0}\phi(t_1; t_0, x_0, \theta) = D_{x_0}\phi(t_1; s, \phi(s; t_0, x_0, \theta), \theta) D_{x_0}\phi(s; t_0, x_0, \theta)$ and therefore $D_{x_0}\phi(t_1; t_0, x_0, \theta) D_{x_0}\phi(s; t_0, x_0, \theta)^{-1} = D_{x_0}\phi(t_1; s, \phi(s; t_0, x_0, \theta), \theta)$.

Consequently, 3.15 is equal to

$$\int_{t_0}^{t_1} D_{x_0}\phi(t_1; s, \phi(s; t_0, x_0, \theta), \theta) D_\theta f(s, \phi(s; t_0, x_0, \theta), \theta) ds \tag{3.16}$$

In turn, $\phi(s; s, x, \theta) = x = \phi(s; t_1, \phi(t_1; s, x, \theta), \theta)$ implies that $D_{x_0}\phi(s; s, x, \theta) = \mathrm{Id} = D_{x_0}\phi(s; t_1, \phi(t_1, s, x, \theta), \theta) D_{x_0}\phi(t_1; s, x, \theta)$ for any $s \in [t_0, t_1]$ and $x \in \mathbb{R}^d$. By taking $x = \phi(s; t_0, x_0, \theta)$, $\mathrm{Id} = D_{x_0}\phi(s; t_1, \phi(t_1, t_0, x_0, \theta), \theta) D_{x_0}\phi(t_1; s, \phi(s; t_0, x_0, \theta), \theta)$ and then $D_{x_0}\phi(t_1; s, \phi(s; t_0, x_0, \theta), \theta) = D_{x_0}\phi(s; t_1, x_1, \theta)^{-1}$.

Therefore, 3.16 can be written as

$$\int_{t_0}^{t_1} D_{x_0}\phi(s;t_1,x_1,\theta)^{-1} D_\theta f(s,\phi(s;t_1,x_1,\theta),\theta)ds$$

By using this series of equalities in 3.13 it follows that

$$\begin{aligned}\frac{\partial \ell}{\partial \theta}(t_1,t_0,x_0,\theta) &= \int_{t_0}^{t_1} \frac{dL}{dx}(x_1) D_{x_0}\phi(s;t_1,x_1,\theta)^{-1} D_\theta f(s,\phi(s;t_1,x_1,\theta),\theta)ds \\ &= \int_{t_0}^{t_1} a_x(s)^T D_\theta f(s,\phi(s;t_1,x_1,\theta),\theta)ds = a_\theta(t_0)^T \end{aligned} \tag{3.17}$$

□

Using theorem 3.9, computing $\frac{dL}{d\theta}(x_1)$ becomes a matter of solving the system of equations backwards in time from $t = t_1$ to $t = t_0$.

Remark 3.10. An ODE-Net is seldom just an ODE solve step, as explained in section 3.1.1. If the model has other phases such as affine transformations or activation functions for the output, these are backpropagated as usual to find $\frac{dL}{dx}(x_1)$ and the technique exposed here is used merely for the ODE solve step. The same applies for any preceding operations, which would be backpropagated as usual from the results of this technique for $a_x(t_0)$ and $a_\theta(t_0)$

This way to backpropagate is normally called the *continuous adjoint method*. The naming comes from the fact that the adjoint usually refers to the gradient with respect to the hidden state at a specified time or step, and this method considers this in the idealised continuous-time model.

3.4.3.1 Advantages

- *Memory efficient*: Since x is recomputed on the backwards pass, it is not necessary to store all the forward computations of x. Therefore, using the same notation as before, memory cost becomes merely $\mathcal{O}(S)$ as it is independent of the time horizon and only requires to solve an extra ODE.

- *Ease of implementation*: There are no constraints on what differential equation solvers to use, as they do not need to be autodfferentiable.

3.4.3.2 Disadvantages

- *Time complexity*: It is slightly slower because it needs to recompute x during the backward pass.

- *Truncation errors*: There are multiple sources of errors. Namely, re-computation of $x(t)$ and numerical solutions for $a_\theta(t)$ and $a_x(t)$.

  Firstly, in the backward pass $x(t)$ is computed again starting from the numerical approximation to the terminal condition $x(T)$. Therefore, there will be a difference between the value of $x(t)$ computed in the forward pass, and the same value computed in the backward pass.

  Secondly, the continuous adjoint equations will have to be solved numerically, making the gradients calculated with this method not as accurate as those calculated by backpropagating through the solver, which represent the gradients of the actual model.

  Al this might make training slower and impact model performance.[6]

3.4.3.3 Use cases

This method is adequate when working with strict memory constraints which would make the discretise-then-optimise approach unfeasible. Additionally, this method might be required if ODE solvers with autodifferentiation support are not available.

An empirical comparison of the last to methods can be found in appendix B.3.

[6]The issue with numerical errors can be addressed with some changes to the algorithm like recording $x(t)$ at some points of the forward pass, but not all the internal operations. This modification is called *interpolated adjoints*.

Chapter 4

Normalising Flows

This chapter explores a non-parametric density estimation technique called *Normalising Flows* (NF). As with other kinds of models aforementioned, they can be approached from a discrete or continuous point of view. This approaches were called by Rezende and Mohamed [28], finite flows and infinitesimal flows, respectively. NF normally refers to the former, while *Continuous Normalising Flow* (CNF) is commonly used for the latter. This terminology is used in this work.

The problem of density estimation can be expressed as follows: given a set of independent observations x_i, $i = 1, \dots, m$, estimate the underlying probability distribution that has produced them [30]. The simplest methodology is parametric estimation. This approach is problematic because assumptions need to be made about the underlying density. In non-parametric density estimation, no such assumptions are made. This added flexibility allows for better approximations, since one is not limited to a certain family of probability distributions.

4.1 Normalising Flows

The main idea behind NF is that by transforming a simple probability distribution through a sequence of invertible mappings, one can obtain a more complex distribution that better describes the underlying one. In order to obtain the final probability distribution, the following theorem is applied.

Theorem 4.1 (Smooth change of variables [5])**.** *Let $X \in \mathbb{R}^d$ be a random vector, with PDF (Probability Density Function) p_X. Let $f : U \subseteq \mathbb{R}^d \to \mathbb{R}^d$ where U is an open set and $f \in \mathcal{C}^1(U)$, and let $V = f(U)$. Additionally, let one write $D_x f = \frac{\partial f}{\partial x}$.*

Suppose $\det D_x f(x) \neq 0 \quad \forall x \in U$ *(i.e. f is invertible with inverse f^{-1} and $f^{-1} \in \mathcal{C}^1(V)$, or simply f is a diffeomorphism between U and its image). If $\mathbb{P}(X \in U) = 1$,*

then $Y = f(X)$ is a random vector with density

$$p_Y(y) = p_X(f^{-1}(y))|\det D_x f(f^{-1})(y)|^{-1}\mathbb{1}_V(y) \tag{4.1}$$

where $\mathbb{1}_V(y)$ represents the indicator function.

Proof. For any $V_0 \subset V$ one has $\mathbb{P}(Y \in V_0) = \int_{V_0} p_Y(y)dy$. At the same time,

$$\begin{aligned}\mathbb{P}(y \in V_0) = \mathbb{P}(X \in f^{-1}(V_0)) &= \int_{f^{-1}(V_0)} p_X(x)dx \\ &= \int_{V_0} p_X(f^{-1}(y))|D_x f(f^{-1}(y))|^{-1}dy.\end{aligned}$$

In the last equation, the change of variable $x = f^{-1}(y)$ has been used in the integral. Also, from the inverse function theorem, the fact that $D_x f^{-1}(y) = D_x f(f^{-1}(y))^{-1}$ has been applied to the last equality.

□

4.1.1 Definition

Let $\{f_k : \mathbb{R}^d \to \mathbb{R}^d\}_k$ be a sequence of bijective and at least $\mathcal{C}^1$ functions.

Let Ω be a sample space and $X : \Omega \to \mathbb{R}$ a random vector with density $p_X : \mathbb{R}^d \to [0, \infty)$. Let this refer to a known distribution π.

Let $\{Y_k\}_k$ be a sequence of random vectors with $Y_1 = f_1(X)$ and defined iteratively as $Y_{k+1} = f_{k+1}(Y_k)$. Let one write as p_k the density for Y_k.

Then the change of variables formula (Theorem 4.1) applied to a chain of K transformations gives the density p_K of $Y_K = f_K \circ f_{K-1} \circ \cdots \circ f_2 \circ f_1(X)$:

$$\log p_K(y) = \log p_X(f^{-1}(y)) - \sum_{k=1}^{K} \log \left|\det D_x f_k(f_k^{-1}(y_k))\right| \tag{4.2}$$

where $f = f_K \circ f_{K-1} \circ \cdots \circ f_2 \circ f_1$ and $y = f(x)$, being x a sample from the original distribution.

The resulting function f can then be used to go from the known distribution π to a new one described by density p_K. Moreover, by sampling from π and applying f, one gets a sample from the new distribution. Therefore, this technique can be useful to approximate densities, but also for generative purposes.

The path formed by the successive distributions p_k is a *Normalising Flow*.

4.1.2 Training

4.1.2.1 Maximum Likelihood

In parametric statistics, one selects the "best" model from a family of distributions with densities p_θ, $\theta \in \Theta$ where Θ is the parameter space. The parameters that define the best fitting density is often found via *maximum likelihood estimation*.

This technique assumes that the best probability density to describe the distribution that generates the data is the one that maximises the probability of the observed samples.

Definition 4.2 (Likelihood function)**.** *For samples* $\mathbf{y} = (y_1, \dots, y_N)^T$ *and a family of densities* $\{p_\theta | \theta \in \Theta\}$ *the* likelihood function *is*

$$\mathcal{L}(\theta; \mathbf{y}) = \prod_{i=1}^{N} p_\theta(y_i).$$

Sometimes, the log-likelihood *is used because of the convenience of its additive form:*

$$\ell(\theta; \mathbf{y}) = \sum_{i=1}^{N} \log p_\theta(y_i).$$

In both cases, it is to be seen as a function of θ*, with the data fixed.*

Consequently, maximum likelihood estimation consists of choosing the distribution with density $p_{\hat{\theta}}$ where $\hat{\theta}$ maximises $\mathcal{L}(\theta; \mathbf{y})$. That is,

$$\hat{\theta} = \arg\max_\theta \mathcal{L}(\theta; \mathbf{y})$$

4.1.2.2 Application to Normalising Flows

One advantage of Normalising Flows with respect to other density estimation techniques such as Variational Auto Encoders (VAE) is that they offer a precise formula for the likelihood of the model, so approximations like the ELBO (Evidence Lower Bound) are unnecessary. Therefore, Normalising Flows can be trained by minimising the negative log-likelihood or, equivalently, $-\frac{1}{N}\sum_{i=1}^{N} \log p_\theta(y_i)$.

To do so, one samples $x \sim p_X$ and does a forward pass recalling all intermediate values y_k to then apply gradient descent on $-\frac{1}{N}\mathcal{L}(\theta; \mathbf{y})$, using formula 4.2 for the density. This, however, has one flaw. Namely, computing $\det D_x f$ is very expensive. To be precise, it has a computational complexity of $\mathcal{O}(N^3)$.

As a result, the complexities of the transformations need to be restricted to functions with easy to compute determinant Jacobians, as well as being invertible. Some of these architectures are explored as examples in the following section.

4.1.3 Examples

4.1.3.1 Planar flows

As Rezende and Mohamed explore in the original paper [28], using *planar flows* with transformations of the form $f(y) = y + uh(w^T y + b)$ where $\Theta = \{w \in$

$\mathbb{R}^K$; $u \in \mathbb{R}^K$; $b \in \mathbb{R}\}$ and h is an element-wise smooth non-linear function with derivative h'. This is useful because the determinant of the Jacobian is known and easy to compute: $1 + u^T h'(w^T y + b)w$.

4.1.3.2 Real NVP

Another structure for the transformations that allows for an easy computation of the determinant of the Jacobian is known as *real-valued non-volume preserving*, as was introduced by Dinh et al. [9].

Once again, let $f(y)$ be the transformation, which the authors call a *coupling layer*. For an input $y \in \mathbb{R}^d$, let $y_{1:i}$ and $y_{i+1:d}$ denote the first i components, and the last $d - i$ for a fixed $0 < i < d$. Then, the transformation is as follows:

$$f(y) = \begin{pmatrix} y_{1:i} \\ y_{i+1:d} \odot \exp\left(s(y_{1:i})\right) + t(y_{1:i}) \end{pmatrix} \tag{4.3}$$

Where $\odot$ is the Hadamard or element-wise product.

This architecture exploits the fact that computing the determinant of a triangular matrix is cheap because it is simply the product of the diagonal. In this case, the Jacobian is

$$\frac{\partial f}{\partial y} = \begin{pmatrix} \mathrm{Id}_i & 0 \\ \frac{\partial f_{i+1:d}}{\partial y_{1:i}} & \mathrm{diag}(\exp\left(s(y_{1:i})\right)) \end{pmatrix}$$

Where $\mathrm{diag}(\exp\left(s(y_1 : i)\right))$ is the diagonal matrix with elements $\exp\left(s(y_1 : i)\right)$ in the diagonal.

Finally, this transformation is also easy to invert. Given a vector $z \in \mathbb{R}^d$ such that $z = f(y)$,

$$y = f^{-1}(z) = \begin{pmatrix} z_{1:i} \\ (z_{i+1:d} - t(z_{1:i})) \odot \exp\left(-s(z_{1:i})\right) \end{pmatrix}$$

4.2 Change of variables for a continuous in time transformation

This section is devoted to the proof of an equivalent theorem to 4.1 but for a continuous transformation through time, instead of a discrete set of layers. This will allow for the introduction of ODE dynamics to Normalising Flows.

Chen et al. called this theorem *Instantaneous change of variables* in their paper on Neural ODEs [6], in which they provided a proof based on the definition of the derivative as a limit. In this section a different approach will be taken, as the theorem can be derived from some basic notions of fluid dynamics.

4.2.1 A brief introduction to the continuity equation

In the following pages, one will try to introduce some of Euler's equations while developing a connection with properties of probability densities. Most of this section's proofs and definitions are based on the first chapter of [7].

Theorem 4.3 (Jacobi-Liouville Formula)**.** *Let $\dot{x} = A(t)x$ be a linear differential equation for some matrix $A(t) : \mathbb{R} \to \mathbb{R}^{n\times n}$ and $x \in \mathbb{R}^n$. Let $M(t)$ be a solution matrix, for which the Wronskian is defined as $w(t) = \det M(t)$. Then w satisfies the following differential equation:*

$$\dot{w} = \mathrm{tr}A(t)w \tag{4.4}$$

where $\mathrm{tr}A(t)$ is the trace of $A(t)$.

Proof. Let $M(T) = (m_1(t), \dots, m_n(t))$ in column notation. Notice that since $M(t)$ is a solution matrix, one has $m_i'(t) = A(t)m_i(t)$; $1 \le i \le n$. Furthermore, if $M(t)$ is not fundamental, then $w(t) = 0$ and equation 4.4 is satisfied trivially. From now on, let one suppose $M(t)$ is fundamental, that is, the columns are linearly independent solutions of the system.

As a n-linear function of its columns, one can differentiate the determinant of $M(t)$ as follows:

$$\begin{aligned} w'(t) &= \sum_{i=1}^{n} \det(m_1(t), \dots, m_i'(t), \dots, m_n(t)) \\ &= \sum_{i=1}^{n} \det(m_1(t), \dots, A(t)m_i(t), \dots, m_n(t)) \end{aligned} \tag{4.5}$$

Since $M(t)$ is fundamental, $\mathcal{B} = \{m_1(t), \dots, m_n(t)\}$ is a basis of $\mathbb{R}^n$.

Let $\alpha(t) : x \mapsto A(t)x$; $\forall x \in \mathbb{R}^n$ be a linear map at every fixed time t. Suppose as well that $\alpha(t)$ has matrix $(\alpha_{ij}(t))_{1\le i,j\le n}$ in basis $\mathcal{B}$. Consequently, the trace of $A(t)$ is $\mathrm{tr}A(t) = \sum_{i=1}^{n} \alpha_{ii}(t)$, which is invariant for change of basis.

Then

$$m_i'(t) = A(t)m_i(t) = \begin{pmatrix} \alpha_{11}(t) & \alpha_{12}(t) & \cdots & \alpha_{1n}(t) \\ \alpha_{21}(t) & \alpha_{22}(t) & \cdots & \alpha_{2n}(t) \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{n1}(t) & \alpha_{n2}(t) & \cdots & \alpha_{nn}(t) \end{pmatrix} \begin{pmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{pmatrix} = \begin{pmatrix} \alpha_{1i}(t) \\ \vdots \\ \alpha_{ii}(t) \\ \vdots \\ \alpha_{ni}(t) \end{pmatrix}$$

where $m_i(t)$ in vector form in basis $\mathcal{B}$ is all 0 except for row i, which is 1. Therefore the result is the i-th column of matrix $A(t)$ in basis $\mathcal{B}$. Expressing it as a linear

equation of the vectors in $\mathcal{B}$, it follows from the above equation that $m_i'(t) = \sum_{j=1}^{n} \alpha_{ji} m_j(t)$.

With this, one can rewrite 4.5 as

$$\begin{aligned} w'(t) &= \sum_{i=1}^{n} \det(m_1(t), \dots, \sum_{j=1}^{n} \alpha_{ji}(t) m_j(t), \dots, m_n(t)) \\ &= \sum_{i=1}^{n} \det(m_1(t), \dots, \alpha_{ii}(t) m_i(t), \dots, m_n(t)) \\ &= \sum_{i=1}^{n} \alpha_{ii}(t) \det(m_1(t), \dots, m_i(t), \dots, m_n(t)) \\ &= \sum_{i=1}^{n} \alpha_{ii}(t) w(t) = \mathrm{tr} A(t)\, w(t) \end{aligned} \tag{4.6}$$

Where the linearity of determinants with respect to their columns was used in the second line, as well as the fact that a determinant with repeated columns is zero. For the same reason $\alpha_{ii}(t)$ can be taken out of the determinant in line 3.

Finally, 4.6 is exactly what one wanted to prove. □

Let $V \subseteq \mathbb{R}^d$ be an open set and $f(t,x)$ where $f : \mathbb{R} \times V \to \mathbb{R}^d$ is a non autonomous vector field of class $\mathcal{C}^1$. Consider the following IVP for $t_0 \in \mathbb{R}$ and $x_0 \in V$:

$$\frac{\partial x}{\partial t}(t) = f(t, x(t)); \quad x(t_0) = x_0 \tag{4.7}$$

whose solution will be written as $\phi(t; t_0, x_0)$ in reference to the initial condition.

Theorem 4.4 (Transport theorem)**.** *Let $g : \mathbb{R} \times V \to \mathbb{R}^d$ be a function of class $\mathcal{C}^1$ that from now on will be called* observable*. One defines the measure of the observable over a bounded open set $U \subseteq \overline{U} \subseteq V$ at time t as*

$$G(t, U) = \int_U g(t,x) dx$$

Consider the bounded open set $U_0 \subseteq \overline{U_0} \subseteq V$ at time t_0 that evolves according to 4.7 and let $U_t = \phi(t; t_0, U_0)$. Then

$$\frac{d}{dt}(G(t, U_t)) = \int_{U_t} \left(\frac{\partial g}{\partial t}(t,x) + D_x g(t,x) f(t,x) + \mathrm{div} f(t,x) g(t,x) \right) dx \tag{4.8}$$

where $D_x g(t,x) = \frac{\partial g}{\partial x}(t,x)$, and div *is the divergence with respect to x. More precisely,* $\mathrm{div} f(t,x) = \mathrm{tr} D_x f(t,x) = \sum_{i=1}^{d} \frac{\partial f_i}{\partial x_i}(t,x)$*, supposing $f = (f_1, \dots, f_d)$*

Proof. Let one start by finding the derivative of $G(t, U_t)$ with respect to time.

$$\begin{aligned}
&\frac{d}{dt}\int_{U_t} g(t,x)dx = \frac{d}{dt}\int_{U_0} g(t,x_0)|\det D\phi(t;t_0,x_0)|dx_0 \\
&= \int_{U_0}\left(\left[\frac{\partial g}{\partial t}(t,x_0) + D_x g(t,x)\frac{\partial \phi}{\partial t}(t,x_0)\right]|d(t,x_0)| + g(t,x_0)\frac{\partial}{\partial t}|d(t,x_0)|\right)dx_0
\end{aligned}$$

where $d(t,x) := \det D\phi(t;t_0,x)$ and $D\phi$ is the derivative with respect to x_0. Writing $x = \phi(t;t_0,x_0)$, the change of variable $x \mapsto x_0$ using ϕ^{-1} has been used to change the domain of integration.

Since $D\phi$ is the derivative of a solution with respect to the initial condition, it satisfies the variational equation

$$\dot{X} = D_x f(t,x_0)X; \quad X(t_0) = \mathrm{Id} \tag{4.9}$$

Furthermore, by the corresponding variational theorem, the solution to equation 4.9 is a principal fundamental matrix, which means $d(t,x_0) \neq 0\ \forall t \in \mathbb{R}$. Since $D(t_0,x_0) = \mathrm{Id}$ this means $d(t,x_0) > 0$ and one can consequently ignore the absolute value in the previous equations.

Considering $A(t) = D_x f(t,x_0)$ as the matrix of the IVP in theorem 4.3, and applying said theorem it follows that

$$\frac{\partial d}{\partial t}(t,x_0) = \mathrm{tr}(D_x f(t,x_0))\det D\phi = \mathrm{div} f(t,x_0)\det D\phi$$

Now, one can put everything together to get the sought result

$$\begin{aligned}
&\frac{d}{dt}\int_{U_t} g(t,x)dx \\
&= \int_{U_0}\left[\frac{\partial g}{\partial t}(t,x_0) + D_x g(t,x_0)f(t,x_0)\right]d(t,x_0) + g(t,x_0)\mathrm{div}(f(t,x))d(t,x_0)dx_0 \\
&= \int_{U_0}\left[\frac{\partial g}{\partial t}(t,x_0) + D_x g(t,x_0)f(t,x_0) + g(t,x_0)\mathrm{div}(f(t,x))\right]d(t,x_0)dx_0 \\
&= \int_{U_t}\frac{\partial g}{\partial t}(t,x) + D_x g(t,x)f(t,x) + g(t,x)\mathrm{div}(f(t,x))dx
\end{aligned}$$

Where in the last line the change of variables has been reversed. □

Remark 4.5. The relationship between g in the previous theorem and a probability density is clear. Without loss of generality, consider an absolutely continuous random variable $X : \Omega \to \mathbb{R}$ with probability $\mathbb{P}$, cumulative distribution function $G : \mathbb{R} \to [0,1]$ and probability density function $g \geq 0$. Let $(a,b) \subset \mathbb{R}$ be an open set with $a < b$. Then, $\mathbb{P}(a < x < b) = G(b) - G(b) = \int_{(a,b)} g(x)dx$. Although in this case the measure would be the probability $\mathbb{P}$ and not G as the notation would suggest.

Now let $X : \Omega \to \mathbb{R}^n$ be a random vector from a sample space Ω. Suppose the points of such distribution are transported through IVP 4.7, just like the fluid mentioned above. Then, let $g : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}$ denote the probability density at each time t for point $x \in \mathbb{R}^n$. Probability density functions have two properties. Namely, they are non-negative and $\int_{\mathbb{R}} g(t,x)dx = 1$. A straightforward way to keep this last property is by conservation of mass. This is, the rate of change of G with respect to time is zero.

Theorem 4.6 (Continuity equation)**.** *Let $g : \mathbb{R} \times V \to \mathbb{R}$ be a function of class $\mathcal{C}^1$. Suppose that the measure $G(t,U) = \int_U g(t,x)dx$ is conserved. Then the following equation holds and is called the* continuity equation*:*

$$\frac{\partial g}{\partial t}(t,x) + D_x g(t,x) f(t,x) + g(t,x)\mathrm{div} f(t,x) = 0 \tag{4.10}$$

where $D_x g(t,x) = \frac{\partial g}{\partial z}(t, z(t))$. This can also be written as $\frac{\partial g}{\partial t} + \mathrm{div}(gf) = 0$.

Proof. If the measure is conserved,

$$\frac{d}{dt}\left(G(t,U)\right) = 0. \tag{4.11}$$

Let $U_0 \subset \overline{U_0} \subset V$ be a bounded open set and $U_t = \phi(t; t_0, U_0)$. Then applying the transport theorem (equation 4.8) to 4.11 one gets

$$\begin{aligned} 0 &= \frac{d}{dt}\left(G(t,U)\right) \\ &= \int_{U_t} \frac{\partial g}{\partial t}(t,x) + D_x g(t,x) f(t,x) + g(t,x)\mathrm{div} f(t,x)dx \end{aligned}$$

Since $\mathrm{div} f \in \mathcal{C}^1$, $g \in \mathcal{C}^1$, and the previous equality holds for any bounded open set, then the expression inside the integral sign is zero. If the function was positive at a certain point, it would be positive in an open set and then the integral over that set would be positive. □

4.2.2 Instantaneous change of variables theorem

Although this result is more general, for convenience when using it in the following pages, let one give the problem the adequate context.

Consider the following IVP where the initial condition is sampled from a base distribution, for instance a Gaussian:

$$\frac{dz}{dt}(t) = f(t, z(t)); \quad z(0) \sim \mathcal{N}(0, \mathrm{Id}_{d\times d}) \tag{4.12}$$

for $t \in [t_0, t_f]$, where $f : [t_0, t_f] \times \mathbb{R}^d \to \mathbb{R}^d$ is of class $\mathcal{C}^1$. Allowing for some abuse of notation, let $z(t)$ be the solution to 4.12.

Additionally, let $p : [t_0, t_f] \times \mathbb{R}^d \to \mathbb{R}$ where $p(t, z(t))$ denotes the probability density of $z(t)$ at time t. Notice that, as in the previous section, this density will be transported while conserving the probability measure.

Theorem 4.7. *(Instantaneous change of variables) Assume f and p are of class $\mathcal{C}^1$. Then p evolves according to the differential equation*

$$\frac{d}{dt}(\log p(t, z(t))) = -\text{div} f(t, z(t)) \tag{4.13}$$

Proof. Differentiating the time-dependant function $t \mapsto p(t, z(t))$ and using the continuity equation 4.10 and the fact that $\frac{dz}{dt}(t) = f(t, z(t))$ it follows directly that

$$\begin{aligned}
\frac{d}{dt}(p(t, z(t)) &= \frac{\partial p}{\partial t}(t, z(t)) + \frac{\partial p}{\partial z}(t, z(t))\frac{dz}{dt}(t) \\
&= -\frac{\partial p}{\partial z}(t, z(t))f(t, z(t)) - p(t, z(t))\text{div} f(t, z(t)) - \frac{\partial p}{\partial z}(t, z(t))f(t, z(t)) \\
&= -p(t, z(t))\text{div} f(t, z(t))
\end{aligned} \tag{4.14}$$

Now, taking the log-density, differentiating with respect to time, and using 4.14 one has the result

$$\frac{d}{dt}\log p(t, z(t)) = \frac{1}{p(t, z(t))}\frac{\partial p}{\partial t}(t, z(t)) = -\text{div} f(t, z(t))$$

□

4.3 Continuous Normalising Flows

Just like with Normalising Flows, the idea is to transform a simple distribution to something more expressive that hopefully better describes the observed system. However, there is one big difference between CNF and NF: the former uses continuous dynamics by transforming the distribution using a differential equation instead of a series of compositions of invertible functions.

4.3.1 Definition

Suppose one observes a probability distribution with density π over some state space $\mathbb{R}^d$ and one wishes to approximate it.

Consider the following IVP with a random initial condition sampled from a known distribution $\mathcal{D}(\mu)$ which has density $p(0, x) \in \mathcal{C}^1$ for all $x \in \mathbb{R}^d$, and μ

denotes whatever parameters this distribution has. For instance, if one were to use a normal distribution with location 0 and covariance $\mathrm{Id}_{d\times d}$, then $\mu = (0, \mathrm{Id}_{d\times d})$.

$$\frac{dz}{dt}(t) = f_\theta(t, z(t)) \text{ for } t \in [0, T]; \quad z(0) \sim \mathcal{D}(\mu) \tag{4.15}$$

where $f_\theta : [0, T] \times \mathbb{R}^d \to \mathbb{R}^d$ is a neural network with parameters θ. Notice that here, the initial and terminal times are set to 0 and T. This is for simplicity and works for any $t_0 < t_f$.

Using this model, one seeks the distribution of $z(T)$ to be approximately the observed one, and therefore $p(T, \cdot)$ to approximate $\pi(\cdot)$.

Then, by using the instantaneous change of variables theorem (4.7) one knows the dynamics of the probability density over time, which allows for maximum likelihood training.

4.3.2 Training

As with normalising flows, knowing the probability density of the modelled distribution allows for maximum likelihood training. In this case, solving a system of differential equations backward in time will suffice.

Given a terminal condition $x \in \mathbb{R}^d$ consider the following IVP:

$$\begin{cases} \frac{d}{dt}\begin{pmatrix} z(t) \\ I(t, z(t)) \end{pmatrix} = \begin{pmatrix} f_\theta(t, z(t)) \\ -\operatorname{tr}\left(\frac{\partial f_\theta}{\partial z}(t, z(t))\right) \end{pmatrix} & \forall t \in [0, T]; \\ \begin{pmatrix} z(T) \\ I(T, z(T)) \end{pmatrix} = \begin{pmatrix} x \\ 0 \end{pmatrix} \end{cases} \tag{4.16}$$

Solving 4.16 from $t = T$ to $t = 0$, $I(0, z(0)) = -\displaystyle\int_T^0 \operatorname{tr}\left(\frac{\partial f_\theta}{\partial z}(t, z(t))\right) dt$. This allows one to compute $\log p(T, z(T)) = \log p(0, z(0)) - I(0, z(0))$ where $p(0, \cdot)$ is the density of the known distribution in 4.15. Then, it is a matter of minimising the negative log-likelihood.

4.3.3 Advantages over Normalising Flows

The main advantage of CNF over NF is the use of the trace-Jacobian instead of the determinant-Jacobian. In the general case, this allows the cost of computation to go from $\mathcal{O}(d^3)$ to $\mathcal{O}(d^2)$. This decrease in complexity makes it possible to have transformations that are more complex and expressive, which leads to better results in approximating the observed distribution [6, 12]. An experiment on this can be found in appendix B.2.

Moreover, the complexity of the trace-Jacobian can be further decreased by using the *Hutchinson's trace estimator*, which brings it down to $\mathcal{O}(d)$ [12].

Definition 4.8. *Let $A \in \mathbb{R}^d \times \mathbb{R}^d$ be any matrix. Let $x \in \mathbb{R}^d$ be a random variable such that $\mathbb{E}[xx^T] = \mathrm{Id}_{d\times d}$. Then $\mathbb{E}[x^T Ax] = \operatorname{tr} A$ is known as the* Huntchinson's trace estimator.

Proposition 4.9. *Let $A \in \mathbb{R}^d \times \mathbb{R}^d$ be any matrix and let $x \in \mathbb{R}^d$ be a random variable such that $\mathbb{E}[xx^T] = \mathrm{Id}_{d\times d}$. Then $x^T Ax$ is an unbiased estimator of* $\operatorname{tr} A$.

Proof. [1] Using the linearity of the expected value and the trace, as well as the cyclic property (for any matrices C and B such that $C^T B$ and BC^T can be multiplied, $\operatorname{tr}(C^T B) = \operatorname{tr}(BC^T)$) it follows that

$$\begin{aligned}\mathbb{E}[x^T Ax] &= \mathbb{E}[\operatorname{tr}(x^T Ax)] = \mathbb{E}[\operatorname{tr}(Axx^T)] \\ &= \operatorname{tr}(\mathbb{E}[Axx^T]) = \operatorname{tr}(A\mathbb{E}[xx^T]) \\ &= \operatorname{tr}(A)\end{aligned}$$

□

This holds for any random variable x such that $\mathbb{E}[x] = 0$ and $\mathrm{Cov}(x) = \mathrm{Id}$, for example, a multivariate normal distribution or a Rademacher random variable[1]. In practice, this is computed using Monte-Carlo approximation and, in order to keep the dynamics deterministic within each call of the ODE solver, one can use a fixed noise vector ϵ for the duration of each solve. Then, using the same notation as in the previous section,

$$\begin{aligned}\log p(T, z(T)) &= \log p(0, z(0)) - \int_0^T \operatorname{tr}\left(\frac{\partial f_\theta}{\partial z}(t, z(t))\right) dt \\ &= \log p(0, z(0)) - \int_0^T \mathbb{E}_\epsilon\left[\epsilon^T \frac{\partial f_\theta}{\partial z}(t, z(t))\epsilon\right] dt \\ &= \log p(0, z(0)) - \mathbb{E}_\epsilon\left[\int_0^T \epsilon^T \frac{\partial f_\theta}{\partial z}(t, z(t))\epsilon dt\right]\end{aligned}$$

4.3.4 Example of usage for image generation

A possible usage of this kind of model is to generate images similar to provided samples. Using the MNIST dataset, CNF can be used to learn the distribution of the digits in the 28×28-dimensional space. Once an approximation to this distribution has been obtained, sampling from it generates new images corresponding to hand-written images. This experiment is explained in more detail in B.1.

[1] It is a discrete random variable with probability mass function $f(k) = \begin{cases} 1/2 & \text{if } k = \pm 1 \\ 0 & \text{otherwise} \end{cases}$.

4.4 CNF as optimal transport problems

One problem with CNF is that they do not always learn the most efficient trajectories to transform the original distribution to the observed one.

In [26] the researchers propose a way to solve this by adding transport costs to the loss function. This is useful because it reduces the number of time steps required to solve 4.16. Furthermore, by encoding the underlying regularity of optimal transport into the model, it is free from learning unwanted dynamics. This, in turn, reduces the number of parameters required.

4.4.1 Deriving the loss function

As seen in section 2.1.2.2, a common loss function to measure the similarity of two probability distributions is the Kullback-Leibler divergence. Let one start by deriving a cost function for the CNF model.

Proposition 4.10 ([26])**.** *Under the same conditions as in 4.12, minimising the Kullback-Leibler divergence*

$$D_{KL}(p(0, \mathbf{z}(0;\mathbf{x}))\|p_n(\mathbf{z}(0;\mathbf{x})))$$

where $\mathbf{z}(t;\mathbf{x})$ *represents the vector of all samples (the vector* $\mathbf{x}$ *is the terminal condition) transported backwards through the ODE flow, and* p_n *is the multivariate normal used as terminal condition of the CNF model, is equivalent to the following optimisation problem*

$$\min_{\theta} \mathbb{E}_{p_f(x)}[\mathcal{C}(T,\theta)] \tag{4.17}$$

where p_f *is the real observed probability and*

$$\mathcal{C}(T,\theta) := \frac{d}{2}\log(2\pi) + \frac{1}{2}\|\mathbf{z}(0;\mathbf{x})\|^2 - \log|\det D_x \mathbf{z}(0;\mathbf{x})|. \tag{4.18}$$

Here, the θ *variable refers to the fact that the flow* z *depends on the vector field's parameters.*

Proof. Given an empirical sample $x \in \mathbb{R}$ generated by the probability distribution associated to density p_f, one can consider the invertible $\mathcal{C}^1$ function that maps $x \mapsto z(t;x)$ for $t \in [0,T]$. Then, using the change of variables theorem (4.1) one has

$$p_f(x) = p(t, z(t;x))|\det D_x z(t;x)| \tag{4.19}$$

where, like before, $p(t,\cdot)$ is the probability density transformed by the ODE at time t.

Using 4.19 and the change of variables inside the integral,

$$\begin{aligned} D_{KL}(p(0,x)\|p_n(x)) &= \int_{\mathbb{R}^d} \log\left(\frac{p(0,x)}{p_n(x)}\right) p(0,x)dx \\ &= \int_{\mathbb{R}^d} \log\left(\frac{p(0,z(0;x))}{p_n(z(0;x))}\right) p(0,z(0;x))|\det D_x z(0;x)|dx \\ &= \int_{\mathbb{R}^d} \log\left(\frac{p_f(x)}{p_n(z;x))|\det D_x z(0;x)|}\right) p_f(x)dx \end{aligned} \tag{4.20}$$

Consider the formula for the multivariate probability density $\mathcal{N}(0, \mathrm{Id}_{\mathbb{R}^d})$ is

$$p_n(x) = (2\pi)^{-d/2}\exp\left(-\frac{1}{2}x^T x\right) = (2\pi)^{-d/2}\exp\left(-\frac{1}{2}\|x\|^2\right)$$

Then, applying it on 4.20 and allowing for $D_{KL} := D_{KL}(p(0,x)\|p_n(x))$ one has

$$\begin{aligned} D_{KL} &= \int_{\mathbb{R}^d} \left[\log p_f(x) - \log|\det D_z(0;x)| + \frac{d}{2}\log(2\pi) + \frac{1}{2}\|x\|^2\right] p_f(x)dx \\ &= \int_{\mathbb{R}^d} \left[\log p_f(x) + \mathcal{C}(T,\theta)\right] p_f(x)dx \\ &= \mathbb{E}_{p_f}\left[\log p_f(x) + \mathcal{C}(T,\theta)\right] \end{aligned}$$

Since $\log p_f(x)$ is an unknown constant, minimising $\mathcal{C}(T,\theta)$ is sufficient to minimise the original Kullback-Leibler divergence. □

Remark 4.11. At first it might seem strange that one is looking to minimise $D_{KL}(p(0,\mathbf{z}(0;x))\|p_n(\mathbf{z}(0;x)))$ instead of $D_{KL}(p(T,x)\|p_f(x))$ considering p_f is the objective density and $p(T,\cdot)$ the model's approximation. However, this is not possible because p_f is not known.

The idea behind this is simple. If $\mathbf{x}$ is a vector of samples following p_f and it is transported backward in time using the learnt flow, at time $t = 0$ the points should be distributed as the multivariate normal $\mathcal{N}(0, \mathrm{Id}_{\mathbb{R}^d})$. Since it is known that flowing backwards from the learnt distribution $p(T,\cdot)$ results in p_n, one can assess the similarity between $p(T,\cdot)$ and p_f by how close the transported $p(0,\cdot)$ and p_n are.

Then, the model can be trained by minimising 4.18 instead of maximum likelihood.

4.4.2 Adding a transport cost

Now the objective is to make the trajectories be as short as possible. To do this, one can add a regularisation factor to the optimisation problem. This factor will be the *transport cost* that will penalise solutions with long trajectories.

Consider the following transport cost:

$$L(T,\theta) = \int_0^T \frac{1}{2}\|f_\theta(t,z(t))\|^2 dt \tag{4.21}$$

In practice, L is computed using an ODE solver like one does with $I(0,z(0))$ in 4.16.

With this, the optimisation problem in 4.17 becomes

$$\min_\theta \mathbb{E}_{p_f(x)}[\mathcal{C}(T,\theta) + L(T,\theta)] \tag{4.22}$$

In practice, $L(T,\theta)$ is computed when calculating the trajectories in 4.16. Therefore, it is found using an ODE solver. In fact, the following IVP is added to the original training problem:

$$\begin{cases} \frac{d}{dt}L(t,\theta) = \frac{1}{2}\|f_\theta(t,z(t))\|^2 \ \ \forall t \in [0,T]; \\ L(T,\theta) = 0 \end{cases} \tag{4.23}$$

Chapter 5

Conclusion

This project has accomplished its purpose and is the culmination of months of work and learning. Firstly, it has produced a comprehensive review of neural ODEs and their integration to the broader contexts of modern deep learning and mathematics. It is the author's opinion that the topic has been explained in detail and regard for formalism, while also being accessible to a non-specialised reader. Furthermore, chapter 4 addresses the objective to understand and explain a discrete density estimation family of models, as well as their continuous-in-time counterparts. Not only does this provide a survey of discrete and continuous NF, but it also explains the necessary mathematical foundation they are built upon.

In addition to the theoretical investigation of the topic, the empirical experiments and demonstrations illustrate the concepts discussed here and allow for a smooth introduction to the models and their implementation. As their main goal, they provide valuable insight into the behaviour and structure of neural ODEs and CNFs. However, none of the results obtained are state-of-the art since the examples had to be simplified due to a computational restriction of the hardware available. Existing literature shows these kinds of model can achieve excellent results in areas like density estimation [6, 12].

The main contributions lie in the development of a different proof for the continuous adjoint equations built upon variational equations, and the framing of the instantaneous change of variables through fluid mechanics. This shows how most, if not all, of the developments that popularised this field were not new, but rediscoveries of techniques and results already known in other areas of knowledge. In summary, this project represents a new approach compared to most existing literature [6, 12, 19] in terms of its mathematical contextualisation.

Lastly, neural differential equations as a field is much richer than what could be explored in this project. Future examination of the topic might include the conjoining of neural networks and other kinds of differential equations, such as

stochastic or controlled differential equations. Additionally, the application of neural ODEs to time-series modelling, especially irregular time-series, is one of the most promising uses, which has not been explored here.

Appendix A

Numerical Methods for ODEs

Assume one wants to approximate the solution to the following IVP:

$$\begin{cases} \dot{x} = f(t, x) \\ x(t_0) = x_0 \end{cases} \tag{A.1}$$

Where $f : \mathcal{I} \times \mathbb{R}^d \longrightarrow \mathbb{R}^d$ is a well-behaved function, $\mathcal{I} \subseteq \mathbb{R}$ is an interval, and $x_0 \in \mathbb{R}^d$ is the initial condition.

In order to talk about *the* solution to the IVP one must insist on it being unique at least in a certain interval. Therefore, at the very least, one needs for f to be continuous, and Lipschitz with respect to x.

A.1 Euler's method

Let ϕ be the actual solution to A.1 that one wishes to approximate. Euler's method consists on obtaining a set of approximations x_n of $\phi(t_n)$ for some equidistant grid points $t_n \in [t_0, t_N] \subseteq \mathcal{I}$ with $h := \frac{t_N - t_0}{N} = t_{n+1} - t_n$, $0 \leq n < N$.

Each point of the approximation is computed as follows:

$$x_{n+1} = x_n + hf(t_n, x_n) \tag{A.2}$$

This can be interpreted geometrically as approximating the solution at t_{n+1} by moving along the tangent line at x_n. Alternatively, one can see it as a Taylor expansion of first order centred at t_n each step.

The astute reader might wonder about the study of stability so that the approximation does not fall too far from the actual solution. The critical question is whether, as $h \to 0$, the numerical solution tends to the exact solution ϕ. Luckily for the purpose of this dissertation, if f is Lipschitz, then Euler's method is convergent (See [18]; Theorem 1.1).

A.2 Explicit Runge-Kutta methods

Considering the same problem as in the previous section, these family of methods uses the slope at more points to extrapolate a better approximation. Then, an explicit Runge-Kutta computes the next value x_{n+1} following these steps:

$$x_{n+1} = x_n + h \sum_{i=1}^{s} b_i k_i \tag{A.3}$$
$$k_i = f(t_n + c_i, x_n + h \sum_{j=1}^{i-1} a_{ij} k_j) \;\; i = 1, \ldots, s$$

Each method is defined by the number of stages s and the coefficients a_{ij}, b_i, c_i. A Taylor series expansion gives certain conditions under which the method is consistent ($\sum_{i=1}^{s} b_i = 1$) and of order p, i.e. the local truncation error is $\mathcal{O}(h^{p+1})$ (this can be seen as each x_{n+1} being a Taylor expansion of order $mathcalO(h^{p+1})$).

To represent these coefficients, a device called the *Butcher Tableau* is normally used. For the general method described above, it is as follows

$$\begin{array}{c|ccccc} 0 & & & & & \\ c_2 & a_{21} & & & & \\ c_3 & a_{31} & a_{32} & & & \\ \vdots & \vdots & & \ddots & & \\ c_s & a_{s1} & a_{s2} & \ldots & a_{s,s-1} & \\ \hline & b_1 & b_2 & \ldots & b_{s-1} & b_s \end{array}$$

Embedded methods are those that produce an estimate of the local truncation error and use it to adapt the step size. This is done by using one method of order p and one of order $p-1$.

In this case, the lower-order step is given by

$$x^*_{n+1} = x_n + h \sum_{i=1}^{s} b^*_i k_i.$$

Then, the resulting Butcher tableau is written as

$$\begin{array}{c|ccccc} 0 & & & & & \\ c_2 & a_{21} & & & & \\ c_3 & a_{31} & a_{32} & & & \\ \vdots & \vdots & & \ddots & & \\ c_s & a_{s1} & a_{s2} & \dots & a_{s,s-1} & \\ \hline & b_1 & b_2 & \dots & b_{s-1} & b_s \\ & b_1^* & b_2^* & \dots & b_{s-1}^* & b_s^* \end{array}$$

And the error predicted by using both approximations is

$$e_{n+1} = x_{n+1} - x_{n+1}^* = h \sum_{i=1}^{s} (b_i - b_i^*) k_i.$$

This error is then used to change the step size h. A common way to do it is using user-defined absolute (atol) and relative (rtol) tolerance:

$$\begin{aligned} \text{tol}_{n+1} &= \text{atol} + \text{rtol} \cdot \max(|x_n|, |x_{n+1}|) \\ E_{n+1} &= \text{norm}\left(\frac{e_{n+1}}{tol_{n+1}}\right) \\ h_{n+1} &= h_n \left(\frac{1}{E_{n+1}}\right)^{\frac{1}{p+1}} \end{aligned} \tag{A.4}$$

where E_n is the normalised error.

A.2.1 Dormand-Prince method

Dormand-Prince or Dopri is an embedded method part of the Runge-Kutta family. It calculates forth and fifth order approximations and it is the default ODE solver for MATLAB and GNU Octave. Also the one used for all experiments in this dissertation.

Its Butcher tableau is:

$$
\begin{array}{c|ccccccc}
0 & & & & & & & \\
\frac{1}{5} & \frac{1}{5} & & & & & & \\
\frac{3}{10} & \frac{3}{40} & \frac{9}{40} & & & & & \\
\frac{4}{5} & \frac{44}{45} & \frac{-56}{15} & \frac{32}{9} & & & & \\
\frac{8}{9} & \frac{19372}{6561} & \frac{-25360}{2187} & \frac{64448}{6561} & \frac{-212}{729} & & & \\
1 & \frac{9017}{3168} & \frac{-355}{33} & \frac{46732}{5247} & \frac{49}{176} & \frac{-5103}{18656} & & \\
1 & \frac{35}{384} & 0 & \frac{500}{1113} & \frac{125}{192} & \frac{-2187}{6784} & \frac{11}{84} & \\
\hline
 & \frac{35}{384} & 0 & \frac{500}{1113} & \frac{125}{192} & \frac{-2187}{6784} & \frac{11}{84} & 0 \\
 & \frac{5179}{57600} & 0 & \frac{7571}{16695} & \frac{393}{640} & \frac{-92097}{339200} & \frac{187}{2100} & \frac{1}{40}
\end{array}
$$

Appendix B

Experiments

In this appendix one can find brief explanations for the experiments conducted, as well as images generated by them. The code can be found here[1].

B.1 Generating MNIST digits

B.1.1 The dataset

Perhaps one of the most well-known datasets used in Machine Learning, the MNIST database is a compilation of handwritten digits. Its name stands for *Modified National Institute of Standards and Technology*, which is the institution that created the dataset as a combination of images of digits written by high school students and employees of the United States Census Bureau.

Te database consists of 60,000 training images and 10,000 testing ones. The images are black and white and have a resolution of 28×28 pixels. Moreover, the digits are centred and labelled. An example of such images can be appreciated in figure B.1.

B.1.2 Loss function

As explained in chapter 4, normalising flows, especially continuous normalising flows, can be trained via maximum likelihood. Equivalently, another measure of loss is that of bits per dimension, which is the one used in [12] and some of the experiments done in this dissertation.

More precisely, an image z of the MNIST dataset, it can be expressed as a a 28×28 matrix with integer values in $[0, 256]$ representing shades of grey (0 being black and 256 white). Let $D := 28 \cdot 28$ be the dimension of the image (i.e. its total

[1] https://github.com/pbaldisa/neural-odes/tree/main

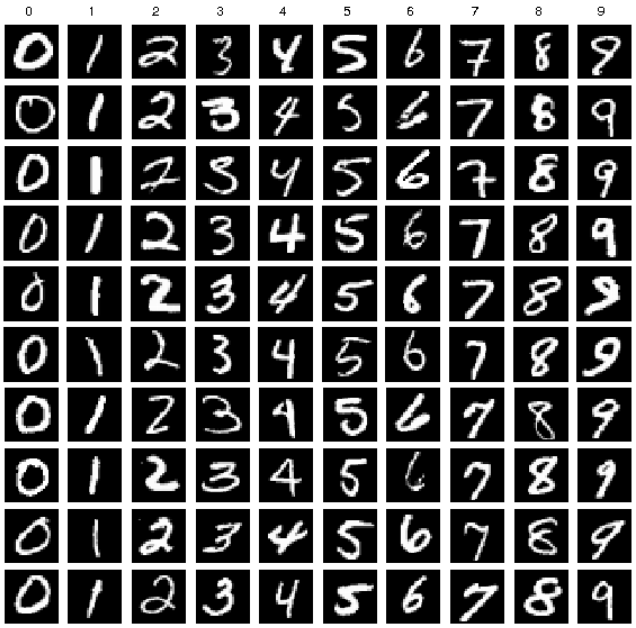


Figure B.1: MNIST images. This shows 100 images from the dataset and their labels.

number of pixels). The space of all images with D pixels is enormous. In fact, the images of the MNIST database lie in a small subspace of $([0,256] \cap \mathbb{Z})^D$ and one wants to learn how to sample from this subspace. The real distribution of images in $([0,256] \cap \mathbb{Z})^D$ is given by a probability density p_d.

Then, given such a representation of an image z, one computes bits per pixel as

$$b(z) = -\frac{\log_2 p_d(z)}{D}. \tag{B.1}$$

Another way to represent grey-scale images is to use real values in the range $[0,1]$. When working with CNF, one often uses this representation in the $[0,1]$ range. Let p denote the probability density in such space and x an image in it. To account for discretisation, one can obtain p_d as

$$p_d(z) = \frac{p\left(\frac{z}{256}\right)}{256} = \frac{p(x)}{256},$$

where the change of variables $z = 256x$ was used.

Finally, using this in B.1, one obtains

$$b(x) = -\frac{\left(\frac{\log p(x)}{D} - \log 256\right)}{\log 2}.$$

B.1.3 Methodology and results

In this experiment, a simplification of the model used by the paper FFJORD: Free-form continuous dynamics for scalable reversible generative models [12] was used. It is a RealNVP with multiple CNF layers. The images are down-sampled to different scales, where a number of CNFs are applied to learn the corresponding probability distribution.

The model was trained during 500 epochs (300 iterations each) on a single GPU for around a week. The final validation error was 1.1573 bits per dimension. Some samples generated but the trained model can be seen in figure B.2.

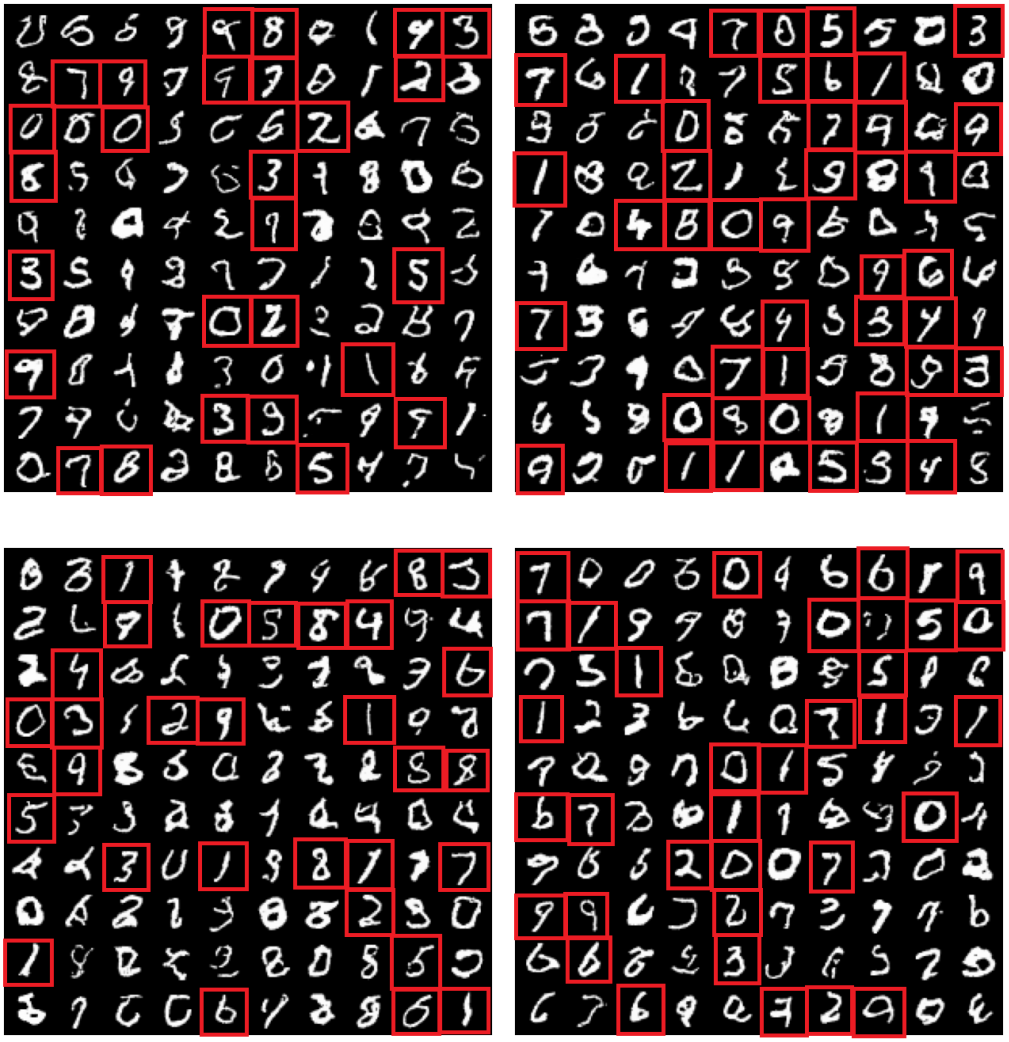


Figure B.2: Images generated by the trained model. The most "authentic" looking images are highlighted in red.

B.2 NF and CNF comparison

In this experiment, the performances of a NF and a CNF model were compared using two toy datasets: a triangle-shaped uniform distribution in $\mathbb{R}^2$, and two concentric circles, also in $\mathbb{R}^2$.

The CNF uses a hypernetwork to learn the parameters of a linear network depending on time. Therefore, the neural ODE corresponds to a non-autonomous equation. This hypernetwork is a 3-layer MLP with 32 hidden units per layer and hiperbolic tangent activation.

For the triangle distribution, the NF uses a RealNVP architecture with 6 coupling layers. Each of these layers has 2 networks: scale and translation. Both have 1 layer and 256 hidden units with LeakyReLU[2] activation. On the other hand, for the circles distribution 14 coupling layers with a depth of 4 each were used.

Table B.1 shows a summary of the performances of both models, and the number of parameters used.

	Triangle distribution		Circles distribution	
	Number of Parameters	Loss	Number of Parameters	Loss
NF	804888	-1.2070	7404600	0.8107
CNF	**15904**	**-1.2433**	**15904**	**0.8128**

Table B.1: Comparison of Triangle and Circle Models in negative log-likelihood, the lower the better.

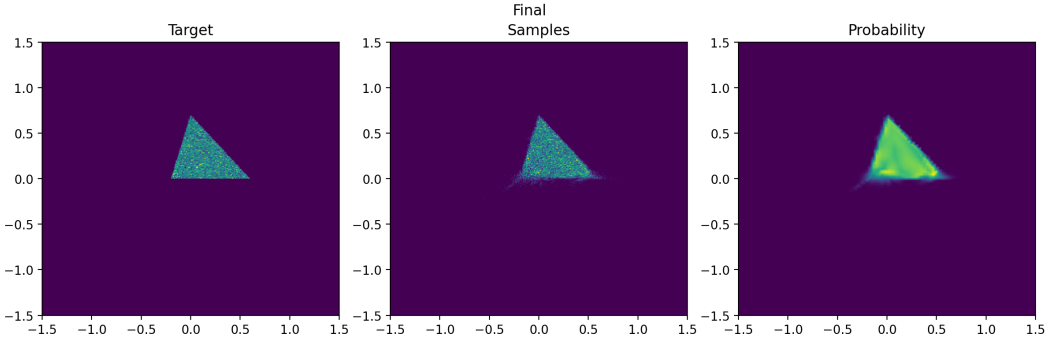
These results can be appreciated in figures B.3 and B.4.

B.3 Training with and without adjoint comparison

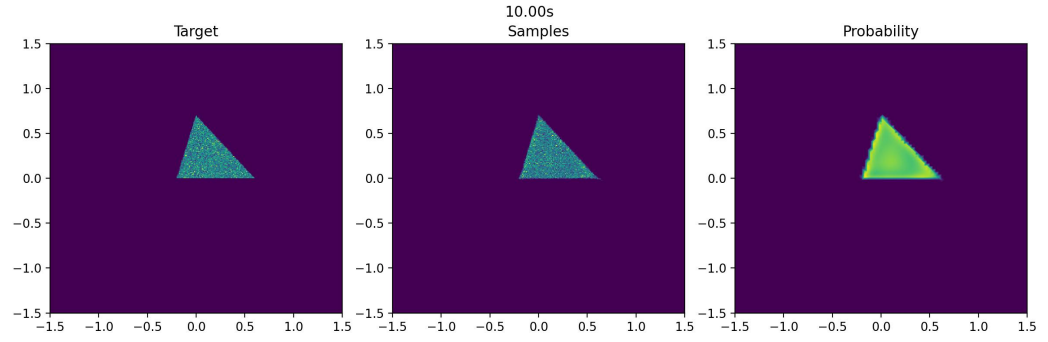
This experiment is aimed at comparing the speed of training when using the discrete-then-optimise and the optimise-then-discrete approach (informally, using the adjoint). The same model was used in both cases, the only difference being the usage of the continuous adjoint equations.

The model used was a neural ordinary differential equation with a single layer MLP vector field, trained for 5000 iterations. In order to compare the results, the training was repeated 5 times. Training with the adjoint method reported an average time of 1552.62 seconds, while not using this yields a training time

[2]A modificaiton of the ReLU function: $f(x) = \begin{cases} x & \text{if} x > 0, \\ 0.01x & \text{otherwise.} \end{cases}$



(a) Results for the NF RealNVP model.



(b) Results for the CNF model.

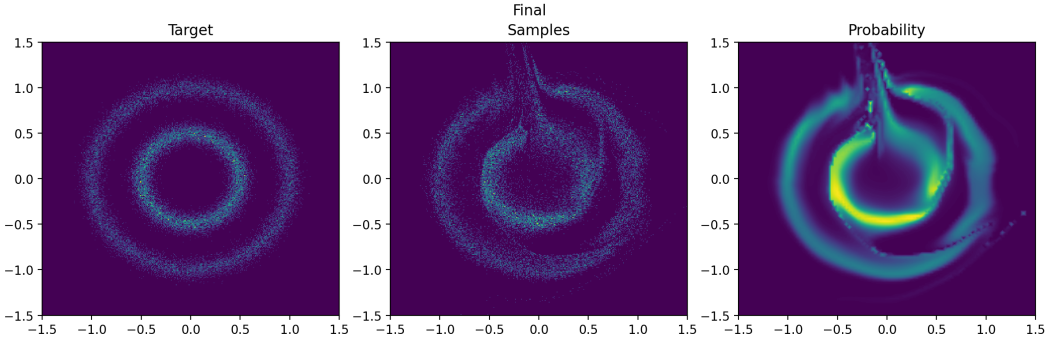
Figure B.3: Comparison of the results obtained by both models for the triangle distribution. The first image is the target distribution, the middle one shows a sample from the model and the image on the right is model's learnt density.

of merely 11.73 seconds. In both cases the average loss was around 7.8, which suggests the method used did not have an effect on it.

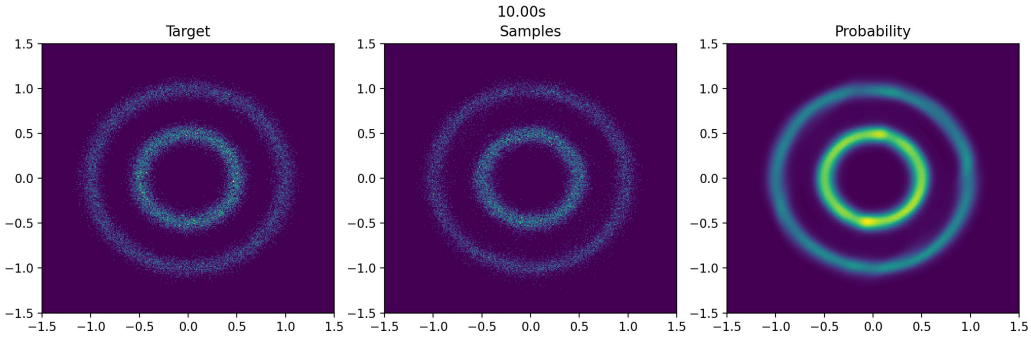
B.4 Learning a linear ODE illustration

In this demonstration a neural ODE was used to learn a linear continuous dynamical system. The goal is to illustrate how neural ODEs can be used in simple ways to learn systems that are modelled by differential equations. This shows great potential to learn physical systems which are normally described by dynamical systems by simple observation (recollecting data to train the model).

A visualisation of this can be seen in figure B.5.



(a) Results for the NF RealNVP model.



(b) Results for the CNF model.

Figure B.4: Comparison of the results obtained by both models for the circles distribution. The first image is the target distribution, the middle one shows a sample from the model and the image on the right is model's learnt density.

B.5 Augmentation illustration

Neural ODEs cannot learn all possible functions. In fact, there are very simple functions like $h(x) = -x$ that cannot be learnt using a neural ODE. The goals of this experiment are to demonstrate the following theorem empirically, and illustrate how to use an augmented ODE to accomplish the same task.

Proposition B.1. *Let* $h : \mathbb{R} \to \mathbb{R}$ *be a continuous function such that*

$$\begin{cases} h(-1) = 1 \\ h(1) = -1 \end{cases}$$

Then the flow of an ODE cannot represent $h(x)$ *and, consequently, a NODE cannot model its behaviour.*

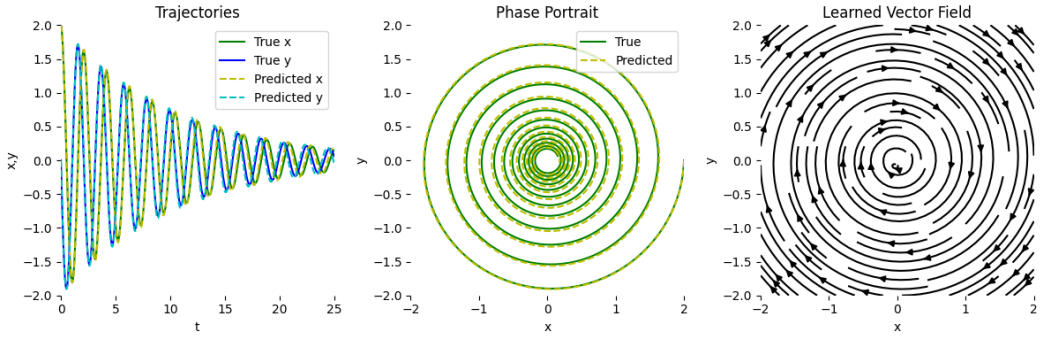


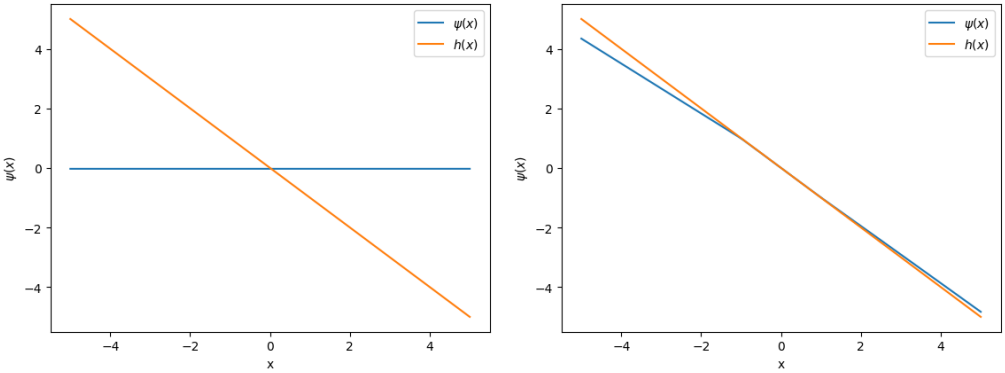
Figure B.5: Example of the dynamics learnt by the NODE. The two images on the left represent the trajectory predicted by the model compared to the real trajectory. The image on the right is the plot of the learnt vector field.

Proof. It is a consequence from the fact that continuous trajectories mapping -1 to 1 and 1 to -1 must cross each other, and the result that states that no ODE trajectories can cross (proposition 2.23).

Only remains to prove that continuous trajectories mapping -1 to 1 and 1 to -1 must intersect. Suppose there exists a vector field f such that ϕ_1 and ϕ_2 are solutions to the IVPs $\{\dot{x} = f(t,x);\ x(0) = -1\}$ and $\{\dot{x} = f(t,x);\ x(0) = 1\}$ respectively, and $\phi_1(T) = 1$, $\phi_2(T) = -1$.

As ϕ_1 and ϕ_2 are solutions to an IVP, they are continuous (proposition 2.33). Therefore, if one defines $\phi(t) = \phi_2(t) - \phi_1(t)$, it is also continuous. Since $\phi(0) = 2$ and $\phi(T) = -2$, by Bolzano's theorem, there exists some $\tilde{t} \in [0, T]$ where $\phi(\tilde{t}) = 0$. Consequently, $\phi_1(\tilde{t}) = \phi_2(\tilde{t})$ and their trajectories intersect. □

In the code, the performances of an augmented and an unagumented neural ODE are compared. Both models have the same architecture, except for the fact the augmented model first increases the dimensionality of the input by adding a zero (in other words, it performs the transformation $x \mapsto (x, 0)$) and after running it through the ODE, it reduces it back with a linear transformation $\mathbb{R}^2 \to \mathbb{R}$. Figure B.6 shows the functions learnt by both models compared to the objective. The models were trained with samples from the interval $[-1, 1]$ and evaluated between $[-5, 5]$.



(a) Unaugmented model. **(b)** Augmented model.

Figure B.6: Comparison of learnt functions by both models.

Bibliography

[1] Ryan P Adams, Jeffrey Pennington, Matthew J Johnson, Jamie Smith, Yaniv Ovadia, Brian Patton, and James Saunderson, *Estimating the spectral density of large implicit matrices*, arXiv preprint arXiv:1802.03451 (2018), 10.

[2] V.I. Arnold, *Ordinary differential equations*, Springer Textbook, Springer Berlin Heidelberg, 1992.

[3] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind, *Automatic differentiation in machine learning: a survey*, Journal of Marchine Learning Research **18** (2018), 1–43.

[4] Léon Bottou, *Stochastic gradient learning in neural networks*, Proceedings of Neuro-Nîmes 91 (Nimes, France), EC2, 1991.

[5] Pierre Brémaud, *Markov chains*, Texts in applied mathematics, Springer International Publishing, Cham, 2020.

[6] Tian Qi Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud, *Neural ordinary differential equations.*, NeurIPS (Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, eds.), 2018, pp. 6572–6583.

[7] Alexandre Joel Chorin and J E Marsden, *A mathematical introduction to fluid mechanics*, Heidelberg Science Library, Springer, New York, NY, January 1984 (en).

[8] George Corliss, Christèle Faure, Andreas Griewank, Lauren Hascoët, and Uwe Naumann (eds.), *Automatic differentiation of algorithms: From simulation to optimization*, Springer-Verlag, Berlin, Heidelberg, 2000.

[9] Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio, *Density estimation using real nvp*, arXiv preprint arXiv:1605.08803 (2016), 1–32.

[10] Emilien Dupont, Arnaud Doucet, and Yee Whye Teh, *Augmented neural odes*, Advances in neural information processing systems **32** (2019), 14–15.

[11] Guillaume Garrigos and Robert M. Gower, *Handbook of Convergence Theorems for (Stochastic) Gradient Methods*, arXiv e-prints (2023), arXiv:2301.11235.

[12] Will Grathwohl, Ricky TQ Chen, Jesse Bettencourt, Ilya Sutskever, and David Duvenaud, *Ffjord: Free-form continuous dynamics for scalable reversible generative models*, arXiv preprint arXiv:1810.01367 (2018), 1–13.

[13] Andreas Griewank, *Who invented the reverse mode of differentiation?*, Optimization Stories, EMS Press, January 2012, pp. 389–400.

[14] Aurlien Gron, *Hands-on machine learning with scikit-learn and tensorflow: Concepts, tools, and techniques to build intelligent systems*, 1st ed., O'Reilly Media, Inc., 2017.

[15] Eldad Haber and Lars Ruthotto, *Stable architectures for deep neural networks*, Inverse problems (2017), 014004.

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, *Deep residual learning for image recognition*, Proceedings of the IEEE conference on computer vision and pattern recognition, 2016, pp. 770–778.

[17] _____, *Identity mappings in deep residual networks*, Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part IV 14, Springer, 2016, pp. 630–645.

[18] Arieh Iserles, *A first course in the numerical analysis of differential equations*, 2 ed., Cambridge Texts in Applied Mathematics, Cambridge University Press, 2008.

[19] Patrick Kidger, *On Neural Differential Equations*, arXiv e-prints (2022), arXiv:2202.02435.

[20] Patrick Kidger and Terry Lyons, *Universal approximation with deep narrow networks*, Conference on learning theory, PMLR, 2020, pp. 2306–2327.

[21] Patrick Kidger, James Morrill, James Foster, and Terry Lyons, *Neural controlled differential equations for irregular time series*, Advances in Neural Information Processing Systems **33** (2020), 6696–6707.

[22] Yiping Lu, Aoxiao Zhong, Quanzheng Li, and Bin Dong, *Beyond finite layer neural networks: Bridging deep architectures and numerical differential equations*, International Conference on Machine Learning, PMLR, 2018, pp. 3276–3285.

[23] Tom M Mitchell, *Machine learning*, vol. 1, McGraw-hill New York, 1997.

[24] James Morrill, Cristopher Salvi, Patrick Kidger, and James Foster, *Neural rough differential equations for long time series*, International Conference on Machine Learning, PMLR, 2021, pp. 7829–7838.

[25] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall, *Activation functions: Comparison of trends in practice and research for deep learning*, arXiv preprint arXiv:1811.03378 (2018).

[26] Derek Onken, Samy Wu Fung, Xingjian Li, and Lars Ruthotto, *Ot-flow: Fast and accurate continuous normalizing flows via optimal transport*, Proceedings of the AAAI Conference on Artificial Intelligence, vol. 35, 2021, pp. 9223–9232.

[27] Allan Pinkus, *Approximation theory of the mlp model in neural networks*, Acta Numerica **8** (1999), 143–195.

[28] Danilo Rezende and Shakir Mohamed, *Variational inference with normalizing flows*, International conference on machine learning, PMLR, 2015, pp. 1530–1538.

[29] Lars Ruthotto and Eldad Haber, *Deep neural networks motivated by partial differential equations*, Journal of Mathematical Imaging and Vision (2018), 352–364.

[30] E. Tabak and Turner Cristina, *A family of nonparametric density estimation algorithms*, Communications on Pure and Applied Mathematics **66** (2013), 20.

[31] Ryan Tibshirani, *10-725: Optimisation. Lecture 6*, `https://www.stat.cmu.edu/~ryantibs/convexopt-F13/scribes/lec6.pdf`, 2013, [Accessed 15-01-2024].

[32] Laurent Younes, *Shapes and diffeomorphisms*, second edition. ed., Applied Mathematical Sciences, vol. 171, Springer Nature, Berlin, Heidelberg, 2019 (eng).

[33] Han Zhang, Xi Gao, Jacob Unterman, and Tom Arodz, *Approximation capabilities of neural odes and invertible residual networks*, International Conference on Machine Learning, PMLR, 2020, pp. 11086–11095.

[34] Z.H. Zhou and S. Liu, *Machine learning*, Springer Nature Singapore, 2021.